AUTOMATING TOPOLOGY AWARE MAPPING FOR SUPERCOMPUTERS

BY

ABHINAV BHATELE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kale, Chair
Professor William D. Gropp
Professor David A. Padua
Matthew H. Reilly, Ph.D.

Abstract

Petascale machines with hundreds of thousands of cores are being built. These machines have varying interconnect topologies and large network diameters. Computation is cheap and communication on the network is becoming the bottleneck for scaling of parallel applications. Network contention, specifically, is becoming an increasingly important factor affecting overall performance. The broad goal of this dissertation is performance optimization of parallel applications through reduction of network contention.

Most parallel applications have a certain communication topology. Mapping of tasks in a parallel application based on their communication graph, to the physical processors on a machine can potentially lead to performance improvements. Mapping of the communication graph for an application on to the interconnect topology of a machine while trying to localize communication is the research problem under consideration.

The farther different messages travel on the network, greater is the chance of resource sharing between messages. This can create contention on the network for networks commonly used today. Evaluative studies in this dissertation show that on IBM Blue Gene and Cray XT machines, message latencies can be severely affected under contention. Realizing this fact, application developers have started paying attention to the mapping of tasks to physical processors to minimize contention. Placement of communicating tasks on nearby physical processors can minimize the distance traveled by messages and reduce the chances of contention.

Performance improvements through topology aware placement for applications

such as NAMD and OpenAtom are used to motivate this work. Building on these ideas, the dissertation proposes algorithms and techniques for automatic mapping of parallel applications to relieve the application developers of this burden. The effect of contention on message latencies is studied in depth to guide the design of mapping algorithms. The *hop-bytes* metric is proposed for the evaluation of mapping algorithms as a better metric than the previously used *maximum dilation* metric. The main focus of this dissertation is on developing topology aware mapping algorithms for parallel applications with regular and irregular communication patterns. The automatic mapping framework is a suite of such algorithms with capabilities to choose the best mapping for a problem with a given communication graph. The dissertation also briefly discusses completely distributed mapping techniques which will be imperative for machines of the future.

जो सुमिरत सिधि होइ गन नायक करिबर बदन।
करउ अनुग्रह सोइ बुद्धि रासि सुभ गुन सदन॥१॥

If one remembers Him, all efforts are successful. He, who is the master of the Ganas and has the face of a handsome elephant (Lord Ganesh), the source of wisdom and culmination of auspicious qualities, may He bless me || 1 ||

अज्ञानतिमिरान्धस्य ज्ञानाञ्जनशलाकया।
चक्षुरुन्मीलितम् येन तस्मै श्रीगुरुवे नमः॥२॥

I bow to the noble Guru who has opened my eyes, blinded by the darkness of ignorance, using a collyrium stick of knowledge || 2 ||

मेरी माँ, गिरिजा भटेले को समर्पित।

To my Maa, Girija Bhatele.

Acknowledgments

The role played by my advisor, Prof. Kale in shaping this dissertation and my career cannot be put in a few words. I will always be in his debt and admire him for the great person he is. My parents and family have made me the person I am today and I cannot thank them enough for being supportive of whatever I do. I hope that my *Maa* will be proud when she reads this dissertation and hopefully this will be a tiny token of appreciation for all that she has done for me. My gurus: teachers at school, professors at IIT Kanpur and Illinois, Brni. Sucheta Chaitanya and Neeb Karori Baba have been a source of guidance and encouragement and I cannot thank them enough. I would also like to thank Eric Bohm, Sameer Kumar, Gagan Gupta and Filippo Gioachin, who worked with me on research related to my thesis. I am also thankful to my dissertation committee for valuable suggestions and comments. Finally, I would like to thank my colleagues at the Parallel Programming Laboratory for their constant criticism and I apologize to them for occupying Prof. Kale's precious time so often. However, if you want extra meeting time, walk with him to his car when he leaves for home in the evenings.

Grants

This dissertation used allocations on several supercomputers at various NSF and DOE centers. Without running time on these machines, this work would not be as relevant and convincing. This research was supported in part by NSF through TeraGrid [1] resources provided by NCSA and PSC through grants ASC050039N and MCA93S028. I wish to thank Fred Mintzer and Glenn Martyna from IBM for access and assistance in running on the Watson Blue Gene/L. This work also used running time on the Blue Gene/P at ANL, which is supported by DOE under contract DE-AC02-06CH11357. Time allocation on Jaguar at ORNL was also used, which is supported by the DOE under contract DE-AC05-00OR22725. Accounts on Jaguar were made available via the Performance Evaluation and Analysis Consortium End Station, a DOE INCITE project.

Software Credits

Latex and gnuplot have been used in preparing this dissertation and are central to the writing of this document. Several other softwares and libraries were used in the process of working on my dissertation and writing it. The script bargraph.pl developed by Derek Bruening (Co-Founder and Chief Architect, VMware, Inc.) has been used for creating nice bar graphs. Paraview, developed as a collaboration by Kitware, Inc., Sandia, LANL and others was used for visualizing maps for OpenAtom. Omnigraffle and Adobe Illustrator (from the Creative Suite) were used for creating the mapping diagrams in this dissertation which made the text much more understandable. Triangle [2], developed by Prof. Jonathan Shewchuk was used for triangulation of meshes generated as inputs for this research. Finally, Graphviz [3], developed by AT&T Research Labs was used for visualizing regular and irregular graphs and their mappings and as a library in Chapter 10.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
4D	four-dimensional
`AFFN`	Affine Mapping
ANL	Argonne National Laboratory
API	Application Programming Interface
ApoA1	Apolipoprotein-A1
BG/L	Blue Gene/L
BG/P	Blue Gene/P
Brni.	Brahmacharini
CO	co-processor mode
`COCE`	Corners to Center Heuristic
CPAIMD	Car-Parrinello *ab-initio* Molecular Dynamics
Cray	Cray Incorporated, The Supercomputer Company
DMA	Direct Memory Access
DOE	Department of Energy
`EXCO`	Expand from Corner Heuristic
HPCT	High Performance Computing Toolkit
IBM	International Business Machines Corporation
LANL	Los Alamos National Laboratory

MD	Molecular Dynamics
MILC	MIMD Lattice Computation
MPI	Message Passing Interface
`MXOV+AL`	Maximum Overlap with Alignment Heuristic
`MXOVLP`	Maximum Overlap Heuristic
NAMD	Nanoscale Molecular Dynamics
NCSA	National Center for Supercomputing Applications
NN	near-neighbor mode
NP	nondeterministic polynomial time
NSF	National Science Foundation
ORB	Orthogonal Recursive Bisection
ORNL	Oak Ridge National Laboratory
PME	Particle Mesh Ewald
POP	Parallel Ocean Program
PSC	Pittsburgh Supercomputing Center
Prof.	Professor
RND	random processor mode
SN	single node mode
`STEP`	Step Embedding
TACC	Texas Advanced Computing Center
VLSI	Very Large Scale Integration
VN	virtual node mode
`WICON`	No contention benchmark
`WOCON`	No contention benchmark
WRF	Weather Research and Forecasting Model
et al.	et alii (and others)
avg	average
max	maximum
vs.	versus

List of Symbols

$CM_{n,n}$	communication matrix
G(,)	GSpace
$G_\rho()$	RhoG
GB	Gigabyte
GHz	Gigahertz
HB	hop-bytes
L_f	length of each flit
MB	Megabyte
MHz	Megahertz
N_g	number of planes in g-space
n_s	number of states
P(,,,)	PairCalculator
R(,)	RealSpace
$R_\rho(,)$	RhoR
Ry	Rydberg

1 Introduction

$\mathcal{P}$etascale machines with hundreds of thousands of cores are being built. These machines have varying interconnect topologies and large network diameters. Computation is cheap and communication on the network is becoming the bottleneck for scaling of parallel applications. Network contention, specifically, is becoming an increasingly important factor affecting overall performance. Communication optimizations to avoid network contention can lead to performance improvements for some applications. Most parallel applications have a specific communication topology. Mapping of tasks in a parallel application to the physical processors on a machine, based on the communication graph can potentially lead to performance improvements. Although this idea was considered in early days of parallel computing (1980s), the following section will show how it has become relevant again.

1.1 Motivation

Mapping of the communication graph for an application on to the interconnect topology of a machine while trying to localize communication is the research problem under consideration in this dissertation. Let us try to understand why mapping is important for performance. The network topology of the largest and most scalable supercomputers today, is a three-dimensional (3D) torus. Some examples are Cray's XT family (XT4 [4], XT5 [5]) and IBM's Blue Gene family (Blue Gene/L [6], Blue Gene/P [7]). For big installations of such machines, the diameter of the network can be large (somewhere between 20 to 60 hops for Blue Gene/P and XT5.)

This can have a significant effect on message latencies when multiple messages start sharing network resources. For common networks with asymptotically inadequate link bandwidth, chances of contention increase as messages travel farther and farther. Network congestion on a link slows down all messages passing through that link. Delays in message delivery can affect overall application performance. Thus, it becomes necessary to consider the topology of the machine while mapping parallel applications to job partitions.

This dissertation will demonstrate that it is not wise to assume that message latencies are independent of the distance a message travels. This assumption has been supported all these years by the advantages of virtual cut-through and wormhole routing suggesting that the message latency is independent of the distance in absence of blocking [8–14]. When virtual cut-through or wormhole routing is deployed, message latency is modeled by the equation,

$$\frac{L_f}{B} * D + \frac{L}{B} \tag{1.1}$$

where L_f is the length of each flit, B is the link bandwidth, D is the number of links (hops) traversed and L is the length of the message. In absence of blocking, for sufficiently large messages (where $L >> L_f$), the first term is insignificant compared to the second. But with large diameters of big supercomputers, this is no longer true for small to medium-sized messages. Let us say that the length of the flit is 32 bytes and the total length of the message is 1024 bytes. Now, if the message has to traverse 8 links, the first term is not negligible compared to the second one (it is one-fourth of the second term). In a network with diameter 16, messages will travel 8 links on the average for a random communication pattern. Also, message sizes in the range of 1 KB are found in several applications which deal with strong scaling to tens of thousands of processors [15, 16]. Hence, for such fine-grained applications

running on large supercomputers, we should not neglect the dependence of message latencies on hops.

Even more important is the observation that Equation 1.1 models message latencies in the absence of contention. In the situation where multiple messages share the same network links, the situation becomes more complex. The bandwidth available per link per message is reduced since it is now being shared. The phenomenon of network resource sharing leading to contention can be explained with a simple example. Let us consider a 3D torus network of size $8 \times 8 \times 8$. The total number of uni-directional links on the system is $8 \times 8 \times 8 \times 6 = 3072$. The diameter of this network is $4+4+4 = 12$ and hence, if messages travel from one random node to another, they will traverse 6 hops on average. Now, if we have four processors per node and every processor sends a message at the same time, all these messages require $512 \times 4 \times 6 = 12288$ links in total and hence, every link will be used for four messages on average. This leads to contention for each link and therefore increases message latencies. Describing this scenario in terms of bandwidth requirements, in order to operate at minimum latency, we need four times the total raw bandwidth available. However, this is not the case and thus the delivered bandwidth is one-fourth of the peak.

Another assumption made by application developers today is that supercomputers like Cray XT4/XT5 do not suffer from bandwidth congestion because of their fast interconnect and high bandwidth and thus would not benefit from topology mapping. It is often assumed that contention is inconsequential on such machines and hence, application developers should not have to worry about network latencies and topology aware optimizations. This is evident from the fact that job scheduling on Cray XT machines is not topology aware (unlike Blue Gene/L or Blue Gene/P, where users are allocated complete tori for their jobs). Also, there is no easy mechanism to obtain topology information on XT machines and for the same reason, the

`MPI_Cart` functions are not implemented efficiently. This dissertation will show that contention also affects machines with high link bandwidths and that there is a need for topology aware job schedulers on such machines.

1.2 Research Goals

The first part of this dissertation uses simple MPI benchmarks for an extensive study of message latencies and their dependence on distance (hops) on several machines in the Cray XT family and IBM Blue Gene family. As we shall see in Chapter 4, in absence of contention, message latencies depend on hops for small and medium-sized messages. In presence of contention, this dependence becomes more striking, especially for large-sized messages. Different experiments which create random as well as controlled contention on the network are conducted on these machines. Message latencies can increase by a factor of 16 depending on the communication patterns and the amount of congestion created. Hence, it is important to consider the topology of the machine, especially for 3D torus/mesh interconnects, to obtain the best performance. As we shall see, this holds true for both IBM and Cray machines. This study will enhance our understanding of the reasons for contention for network resources and will benefit the development of topology aware mapping algorithms.

Topology discovery of the machine at runtime is necessary for mapping on to the processor topology. We have developed a Topology Manager API which provides information about the job partitions to the application at runtime. Using this API, we demonstrate that topology aware mapping can significantly improve performance and scaling of communication bound applications (Chapter 7). The metric we employ to assess the success of topology aware schemes is *hop-bytes*. Hop-bytes are the weighted sum of the number of hops between the source and destination for all messages, with the weights being the message size. This metric gives an indication

of the total communication traffic on the network due to an application. Reducing the total hop-bytes reduces link sharing and contention, thereby keeping message latencies close to the ideal.

An important part of the dissertation is a framework for automatic mapping of a wide class of parallel applications with regular and irregular communication graphs. From pattern recognition of communication graphs to using different heuristics in different situations for mapping solutions, everything is handled by the framework. Parallel applications can get performance improvements using mapping solutions from this framework without any changes to the code base. This framework will save much effort on the part of application developers to generate mappings for their individual applications.

The mapping problem can be reduced to the graph-embedding problem, which is NP-hard. Hence, the focus is on developing a suite of heuristics, each dealing with a different scenario, which together can handle most parallel applications and their communication requirements. The automatic mapping framework is capable of selecting the best mapping for an algorithm based on the hop-bytes metric. We present performance results for both Charm++ [17, 18] and MPI [19–21], applications using the automatic mapping framework.

The Charm++ framework and applications are indispensable to this dissertation as research tools and testing scenarios. Hence, we also discuss the role played by virtualization in mitigating contention and facilitating topology aware mapping. In the future, machines with millions of cores will require million and possibly billion-way parallelism and our mapping algorithms should be able to handle such scales. In an effort towards making mapping techniques scalable to very large machines, the dissertation also presents some initial work on strategies for completely distributed load balancing.

1.3 Thesis Organization

The thesis is organized as follows: Chapter 2 discusses previous work in this field. There was much work in the 1980s on topology aware mapping and then research died down. There has been recent work in the past five years ignited by the development of Blue Gene/L, a 3D torus supercomputer. Contributions of this dissertation and differences with past and recent work are also discussed. We then survey the existing interconnect topologies ranging from fat-trees to Kautz graphs in Chapter 3. This chapter also presents architectural details of the machines used for experiments in this thesis.

Chapter 4 presents an evaluative study on understanding network congestion on IBM and Cray machines. Interesting results showing messaging delays caused by link sharing are presented. The rest of the dissertation presents mapping algorithms and techniques and performance results from case studies on scientific applications as well as synthetic benchmarks. Before we start on this long journey, Chapter 5 introduces hop-bytes as the evaluation metric for mapping algorithms. We argue that *hop-bytes* is a more useful metric for offline evaluation of mapping algorithms than the previously used *maximum dilation* metric.

Topology information about the machines is an important pre-requisite for mapping algorithms and the process of topology discovery on IBM and Cray machines is discussed in Chapter 6. Chapter 7 sets the foundation for the remaining chapters by demonstrating performance improvements through mapping for "production" scientific applications – OpenAtom [16] and NAMD [15, 22]. OpenAtom obtains performance improvements of up to 40% in some cases. NAMD is more latency tolerant and hence the improvements are only seen for very large runs.

Chapters 8 through 10 discuss the automatic mapping framework, an important contribution of this thesis. The pattern matching techniques to identify regular

graphs and various mapping algorithms, which are a part of the suite are explained and evaluated. Performance results are demonstrated on a two-dimensional stencil benchmark, simple2D (a ParFUM [23] benchmark) and WRF [24, 25] among other applications. The advantages of virtualization through overdecomposition, for mitigating contention are presented in Chapter 11. In the spirit of adapting the algorithms developed in the dissertation towards exascale machines, Chapter 12 discusses scalable techniques for mapping and load balancing by parallelizing the decision process. This dissertation has revealed new directions for research which are discussed in the concluding chapter.

2 Related Work

The problem of topology aware mapping has been studied extensively and proved to be NP-hard [26–28]. Pioneering work in this area was done by Bokhari in 1981, where he used pairwise exchanges between nodes to arrive at good solutions [26]. Subsequent research in the 80s can be divided into two categories – physical optimization techniques and heuristic approaches. Physical optimization techniques involve simulated annealing [29, 30], graph contraction [31, 32] and genetic algorithms [33].

The technique of pairwise exchanges used in [26, 27, 29] can take a long time to arrive at good mapping solutions. Bokhari's algorithm of pairwise exchanges coupled with probabilistic jumps takes $\mathcal{O}(N^3)$ time where N is the number of nodes in the graph. Physical optimization techniques also take a long time to derive the mapping 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 [27] and recursive mincut biparitioning [34] are theoretical studies with no results on real machines. Also, most of these techniques (heuristic techniques especially) were developed specifically for hypercubes, shuffle-exchange networks or array processors. The networks which we encounter today are primarily fat-trees and tori.

With the advent of wormhole routing and faster networks, research in this area died. In the recent years, emergence of very large parallel machines has led to the necessity of topology mapping again. The next section discusses the recent work related to mapping.

2.1 Recent Work

Most work from the 80s cannot be used in the present context because of unscalable techniques and different topologies from the ones being used today. As mentioned earlier, increasing effect of the number of hops on message latencies has fueled such studies again. The Cray T3D and T3E supercomputers were the first to raise such issues in the late 90s and the problem of congestion and benefit from mapping were re-evaluated [35–37]. Newer line of supercomputers from Cray with faster interconnects, such as the XT3 and XT4, appeared to have relieved the application writers of such problems. But as this thesis will demonstrate, this is not true. Although researchers at Pittsburgh Supercomputing Center (PSC) have demonstrated the benefit of topology aware job scheduling schemes on their Cray XT3 [38], we believe there has been no published research reporting network contention or quantifying it for the Cray XT family.

Contrary to Cray, IBM systems like Blue Gene/L and Blue Gene/P acknowledge the dependence of message latencies on distance and encourage application developers to use topology of these machines to their advantage [39–41]. On Blue Gene/L, there is a 89 nanoseconds per hop latency attributed to the torus logic and wire delays. This fact has been used both by system developers [42, 43] and application developers to improve performance on Blue Gene/L [44–49]. Bhanot *et al.* [46] have developed a framework for optimizing task layout on BG/L. Since it uses simulated annealing, it is quite slow and the solution is developed offline. Embedding of tasks onto nodes using simple graph embedding schemes (for rings, meshes *et cetera*) has been discussed in [45, 47]. These can be used in MPI topology functions. This work is not as useful when multiple objects have to be mapped to each physical processor (as in CHARM++). Several application writers have also realized the importance of topology mapping on Blue Gene machines and have used it to their benefit for

speeding up their codes [43, 49, 50]. This dissertation shows similar successes for some CHARM++ applications [51, 52].

Research on topology aware mapping also exists in the fields of mathematics and circuit design. Techniques that embed rectangular 2D grids into square 2D grids were proposed to optimize VLSI circuits and significant results were obtained [53–55]. Techniques from mathematics and circuit design are not always applicable to parallel computing. For example, mapping research motivated by reducing the total area of circuit layouts tried to minimize the length of longest wire [53]. As we shall show in a later chapter, longest edge dilation might not be the best metric for parallel machines. Also in case of VLSI circuit design, number of nodes in the host graph can be larger than that in the guest graph. This is not true in case of parallel programs where the number of processes in the application is equal to the number of physical processors in the allocated partition.

2.2 Contributions of This Thesis

This dissertation is among the first to discuss the effects of contention on Cray and IBM machines and to compare across multiple architectures. Opposing popular belief that Cray machines do not stand to benefit from topology mapping, this dissertation proves otherwise and provides detailed methods and results for performance improvements from topology mapping on them [51]. Another contribution of this dissertation is an API for topology discovery which works on both Cray and IBM machines.

We believe that the set of MPI benchmarks we have developed for quantifying message latencies would be useful for the HPC community to assess latencies on a supercomputer and to determine the message sizes for which number of hops makes a significant difference. The effective bandwidth benchmark in the HPC Challenge

benchmark suite measures the total bandwidth available on a system but does not analyze the effects of distance or contention on message latencies [56]. Results from MPI benchmarks re-establish the importance of mapping for the current supercomputers.

Our experience in developing mapping algorithms for production codes and insights discussed in this dissertation will be useful to individual application writers trying to scale their codes to large supercomputers. We believe that the automatic mapping framework is applicable to a wide variety of communication scenarios and will relieve the application writers from the burden of finding good mapping solutions for their codes. Application developers can use this framework for mapping of their applications without any changes to their code base. Unlike most of the previous work, this dissertation handles both cardinality and topological variations in the graphs. The framework provides scalable and fast, runtime solutions. Therefore, it will be useful to a large body of applications running on large parallel machines.

There has not been much research on mapping of unstructured mesh applications for performance optimizations [57, 58] and this dissertation takes up at that task. The dissertation also discusses scalable techniques for distributed load balancing in an effort to move away from centralized mapping decision algorithms.

3 Existing Topologies

Several different topologies are in use today in the largest supercomputers on the Top500 list. Most common among them are fat trees (Infiniband and Federation) and three-dimensional tori and meshes (Cray XT and IBM Blue Gene family). The following sections describe these commonly used topologies.

3.1 Fat Tree and Clos Networks

A fat tree network has a tree structure with processing nodes at the leaves and switches at the intermediate nodes [59]. As we go up the tree from the leaves, the available bandwidth on the links increases, making the links "fatter". Figure 3.1 depicts a simple three level binary fat tree. Infiniband networks are examples of fat trees and so are Federation interconnects from IBM. The Roadrunner machine built by IBM for Los Alamos National Laboratory (LANL) has a fat tree interconnect (Voltaire Infiniband). The Ranger SUN-constellation cluster at Texas Advanced Computing Center (TACC) also has an Infiniband network. As per the Top500 [60] list of November 2009, 36% of the 500 fastest supercomputers use an Infiniband network.

In practice, when building fat tree networks, it is common to have more than one switch at the top-level. As an example, Ranger has a 3-level Infiniband fat tree with two switches at the top level. Each switch can accommodate 3456 nodes. But there are only 3936 nodes in the machine which are evenly distributed between the two switches. Also, to save costs, typically links are oversubscribed and hence, in

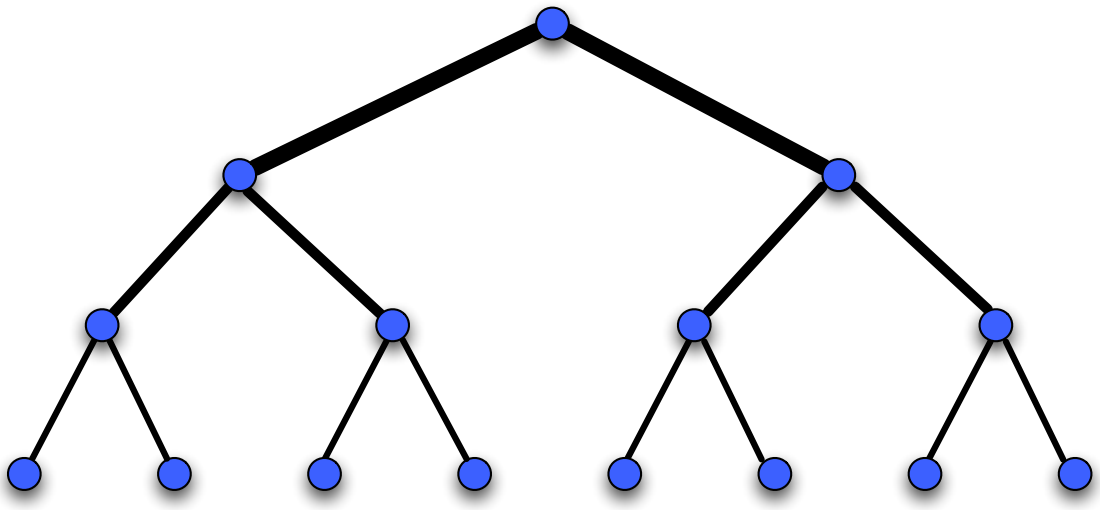


Figure 3.1: A three-level fat tree network

practice, we do not see "true" fat tree networks. The Abe cluster at NCSA has a two level Infiniband network and the links are 2:1 oversubscribed. Another interesting artifact of Abe is that it has eight top level "core" switches. Oversubscribing links and having multiple fast cores on each node often leads to network congestion on these machines. Static routing, decided at boot time for the whole machine, adds to these effects.

3.2 Mesh and Torus Networks

This section discusses topologies with asymptotically inadequate link bandwidths. They also fall under the category of direct networks, since nodes are directly connected to one another through links.

n-dimensional mesh In an n-dimensional mesh network, each processor is connected to two other processors in each physical dimension. This gives a total of $2n$ connections for each processor. The most common case is $n = 3$ which is called a 3D mesh. For a 3D mesh, if the size of the mesh in each dimension is N, the diameter

of the mesh is $3 \times (N-1)$. Figure 3.2 (left) shows a three-dimensional (3D) mesh network.

n-dimensional torus An n-dimensional torus is a mesh with the processors on the end of each dimension connected together. This reduces the diameter of the network by half. The diameter of a $X \times Y \times Z$ torus is $(X+Y+Z)/2$. Figure 3.2 (right) shows a 2D torus network.

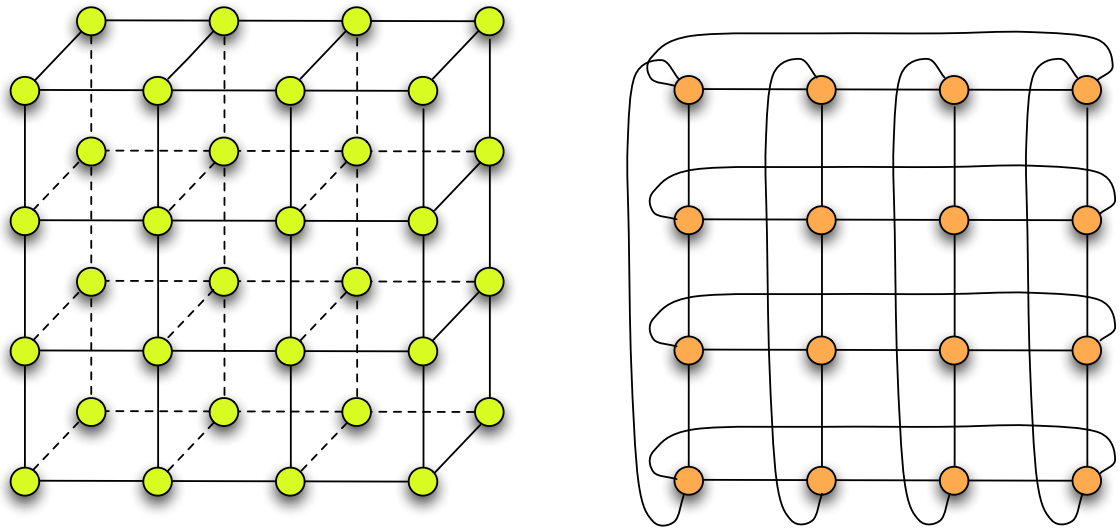


Figure 3.2: A two-dimensional torus and three-dimensional mesh

Torus topologies are not asymptotically scalable because the raw available bandwidth increases as a function of the number of nodes (P), whereas the required bandwidth (assuming communicating processors are randomly chosen) increases as a function of $P^{4/3}$. In contrast, on fully-provisioned fat-trees, the available bandwidth keeps pace with required bandwidth - the diameter is $logP$ and the number of links is proportional to $P\ logP$.

Even though torus topologies are not asymptotically scalable, because of simplicity of design and other practical considerations, torus networks are a popular choice for modern day supercomputers. Five of the ten fastest machines on the Top500 list use a 3D torus interconnect. This dissertation focuses on such networks and

we now describe the supercomputers which were used in this dissertation: two from the IBM Blue Gene family and two from the Cray XT family. The interconnect for both families is a three-dimensional torus but they have different processor speeds and network characteristics.

IBM Blue Gene/L: This is the first class of machine in the Blue Gene series. Each node of a Blue Gene/L (BG/L) machine has two 700 MHz PowerPC 440 cores. 512 nodes form a midplane which is a torus of dimensions $8 \times 8 \times 8$. Each torus link has a uni-directional bandwidth of 175 MB/s [39,61]. We used the installation at T J Watson research center which we refer to as the "Watson BG/L". This machine has a total of $20,480$ nodes.

IBM Blue Gene/P: Blue Gene/P (BG/P) is similar to its predecessor, Blue Gene/L but each node has four 850 MHz PowerPC 450 cores. The nodes are connected by 3D torus links with a uni-directional bandwidth of 425 MB/s [7]. The nodes use a DMA engine to offload communication on the torus network, leaving the cores free for computation. Like BG/L, a midplane composed of 512 nodes forms a torus of size $8 \times 8 \times 8$ in all directions. Smaller allocations than a midplane are a torus in some dimensions and mesh in others. Larger allocations than a midplane are complete tori. The installations of Blue Gene/P at Argonne National Lab (ANL), Surveyor and Intrepid were used for runs in this dissertation. They contain $1,024$ and $40,960$ nodes respectively.

Cray XT3: Each node on a XT3 has two 2.6 GHz AMD Opteron processors and the nodes are connected by a custom SeaStar interconnect. The processors are connected to the SeaStar chip through a Hyper Transport (HT) link. The unidirectional bandwidth of the HT link is $\sim$ 1.6 GB/s whereas that of the network links is 3.8 GB/s [4]. We used the XT3 installation (BigBen) at Pittsburgh Supercom-

puting Center (PSC). This installation has 2068 compute nodes arranged in a 3D torus of dimensions $11 \times 12 \times 16$. Since the job scheduler on XT3 does not allocate cuboidal partitions, nodes allocated for a particular job may not be contiguous. For results reported in this dissertation, the whole machine was reserved and then nodes were allocated (with help from PSC staff) to get completely cuboidal shapes. The largest partition used was $8 \times 8 \times 16$ which is 1024 nodes or 2048 cores and smaller sub-partitions were made from this one. The 1024 node partition has torus links in one dimension (which is of size 16) and mesh links in the other two. For any allocation smaller than 1024 nodes, we have a mesh in all dimensions. BigBen was decommissioned on March 31st, 2010.

Cray XT4/5: For XT4 and XT5 runs, Jaguar at Oak Ridge National Laboratory (ORNL) was used which has $7,832$ XT4 and $18,688$ XT5 compute cores. The XT4 nodes contain a quad-core AMD Opteron (Budapest) processor running at 2.1 GHz whereas the XT5 nodes have dual hex-core AMD Opteron (Istanbul) processors running at 2.6 GHz. Similar to XT3, the XT4 nodes are connected by a 3D SeaStar2 torus network with a unidirectional link bandwidth of 3.8 GB/s. The bandwidth of the HT transport link in this case is twice that of the XT3 – around 3.2 GB/s. The XT5 nodes are connected by a SeaStar2+ network with a link bandwidth of 4.8 GB/s. Again, for smaller allocations than the whole machine, we do not get a complete torus. Contiguous 3D partitions were allocated by a special feature implemented by the administrators (`feature=cube`) for the qsub job scheduler.

3.3 Other Topologies

Finally, we discuss some dense topologies which are asymptotically scalable *i.e.* the total system bandwidth scales proportionally as we increase the number of nodes in the network.

n-dimensional hypercube A hypercube is an n-dimensional analog of a square. An n-dimensional hypercube is also called an n-cube. In a n-dimensional hypercube each of 2^n processors are connected to n other processors. A four-dimensional hypercube can be seen in Figure 3.3 (left). A $(n+1)$-dimensional hypercube can be constructed by connecting the corresponding processors in a n-dimensional hypercube.

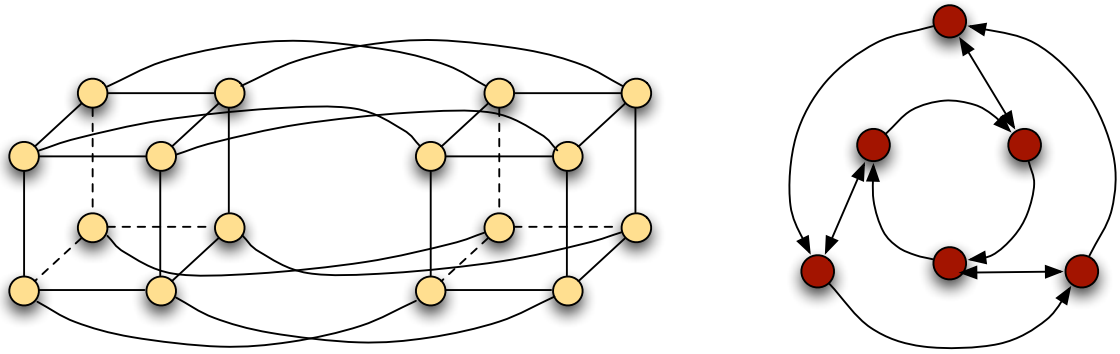


Figure 3.3: A 4-dimensional hypercube and a Kautz graph

Kautz network A Kautz network (primarily used in the SiCortex machines so far [62, 63]) uses a Kautz graph for connecting the processors. A Kautz graph is a directed graph with number of vertices $V = (M+1)M^N$ for a degree M and dimension $N+1$. Such a Kautz graph has the smallest diameter of any possible directed graph with V vertices and degree M. In the machines built by SiCortex, each node has three outgoing links and three incoming links for three other nodes. Figure 3.3 (right) shows a Kautz graph with $M = 2$ and $N = 1$.

4 Understanding Network Congestion

Interconnect topologies and their effect on message latencies in message-passing distributed supercomputers was an important factor determining communication performance in the 1980s. In the 90s, wormhole routing and small diameters of parallel machines reduced the dependence of message latencies on the distance traveled. This chapter will demonstrate that for certain topologies, contention for links by multiple messages can significantly increase message latencies.

Several MPI benchmarks were developed to evaluate the effect of hops (links) traversed by messages, on their latencies. The benchmarks demonstrate that when multiple messages compete for network resources, link occupancy or contention can increase message latencies by up to a factor of 16 times on some architectures. Findings in this chapter suggest that application developers should now consider interconnect topologies when mapping tasks to processors in order to obtain the best performance on large parallel machines.

We first describe a benchmark which is run in absence of contention in order to obtain the best case performance. Then we discuss results from benchmarks which create random and controlled contention on the network.

4.1 WOCON: No Contention Benchmark

This benchmark records message latencies for varying number of hops in absence of contention. One particular node is chosen from the allocated partition to control the execution. We will call this node the master node or master rank. It sends B-byte

messages to every other node in the partition, and expects same-sized messages in return (Figure 4.1). The messages to each node are sent sequentially, one message at a time (pseudo-code in Algorithm 4.1). For machines with multiple cores per node, this benchmark places only one MPI task per node to avoid intra-node messaging effects. The size of the message, B is varied and for each value of B, the average time for sending a message to every other node is recorded. Since the distance from the master node to other nodes varies, we should see different message latencies depending on the distance.

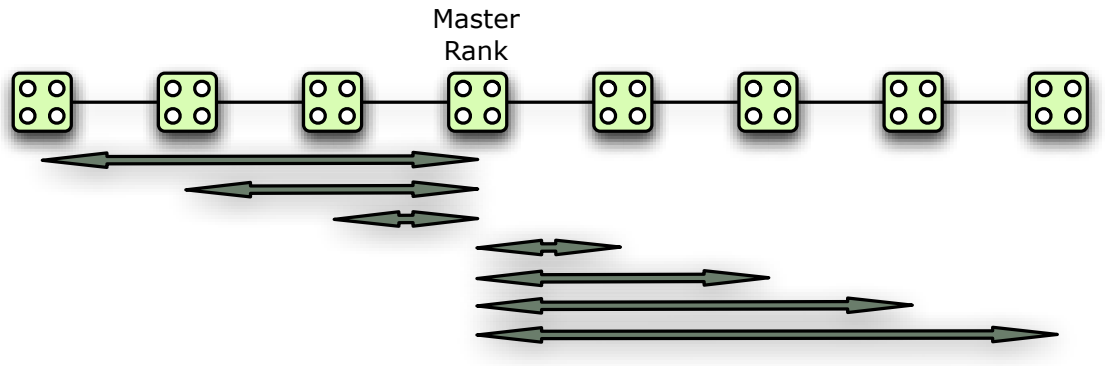


Figure 4.1: Communication patterns in the `WOCON` benchmark. This diagram is a simplified one-dimensional version of the pattern in three-dimension (3D). A master ranks sends messages to all ranks in the 3D partition.

Wormhole routing suggests that message latencies are independent of distance in the absence of contention, for sufficiently large message sizes. The benchmark `WOCON` was used to quantify the effect of the number of hops on small-sized messages. Figures 4.2 and 4.3 present the results obtained from running `WOCON` on four allocations of BG/P, ranging in size from 128 to 1024 nodes (torus sizes $8 \times 4 \times 4$ to $8 \times 8 \times 16$). There are two patterns on the plot: 1. For each message size on the x-axis, the circles represent the time for a message send from the master rank to different nodes on the allocated partition. Note that the vertical bars are actually a cluster of circles, one each for a message send to a different node; 2. Each point on the line represents the percentage difference between the minimum and maximum

Algorithm 4.1 Code fragments showing the core of `WOCON` Benchmark

```
if myrank == MASTER_RANK then
    for i := 1 to numprocs do
        if i ≤ MASTER_RANK then
            sendTime = MPI_Wtime()
            for j := 1 to num_msgs do
                MPI_Send(send_buf, msg_size, MPI_CHAR, i, 1, ...)
                MPI_Recv(recv_buf, msg_size, MPI_CHAR, i, 1, ...)
            end for
            recvTime = MPI_Wtime()
            time[i] = (recvTime − sendTime)/(num_msgs ∗ 2);
        end if
    end for
else
    for i := 1 to num_msgs do
        MPI_Recv(recv_buf, msg_size, MPI_CHAR, MASTER_RANK, 1, ...)
        MPI_Send(send_buf, msg_size, MPI_CHAR, MASTER_RANK, 1, ...)
    end for
end if
```

time for message send for a particular message size.

Message latencies should vary depending on the distance of the target rank from the master rank for very short messages. As expected, we see a regular pattern for the distribution of circles for a specific message size in the four plots (Figures 4.2, 4.3). For small and medium-sized messages, message latencies are spread over a range, the range decreasing with increasing message sizes. This is what one would expect from the wormhole routing model. To have a clearer perception of the range in which message latencies lie, the percentage difference between the minimum and maximum latencies was calculated with respect to the minimum latency for each message size. These values have been plotted as a function of the message size. The difference between the maximum and minimum values (shown by the line) decreases with increasing message size for all the plots. We see a kink in the lines and a corresponding jump in the message latencies at 2 KB messages. This can be explained by the use of different routing protocols on BG/P for different message sizes [41]. For message sizes greater than 1200 bytes, the MPI *rendezvous* protocol

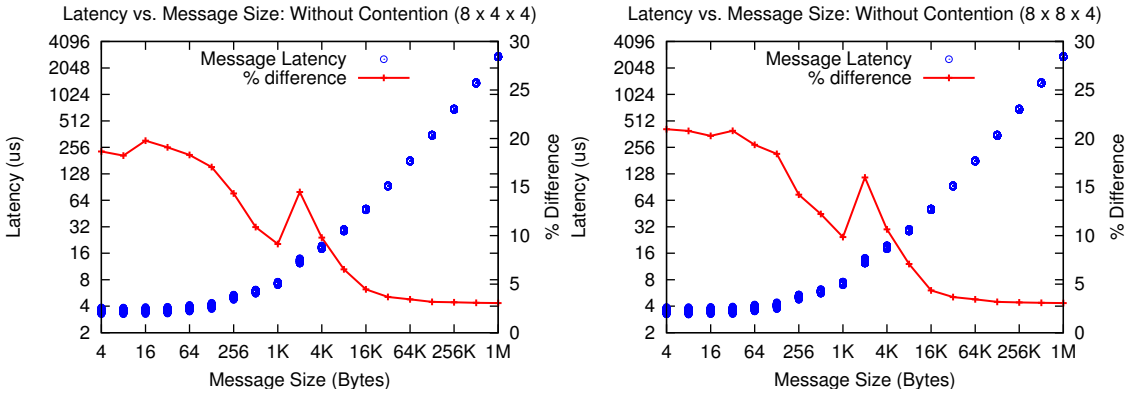


Figure 4.2: Plots showing the effect of hops on message latencies in absence of contention (for $8 \times 4 \times 4$ and $8 \times 8 \times 4$ sized tori on Blue Gene/P, Benchmark: `WOCON`)

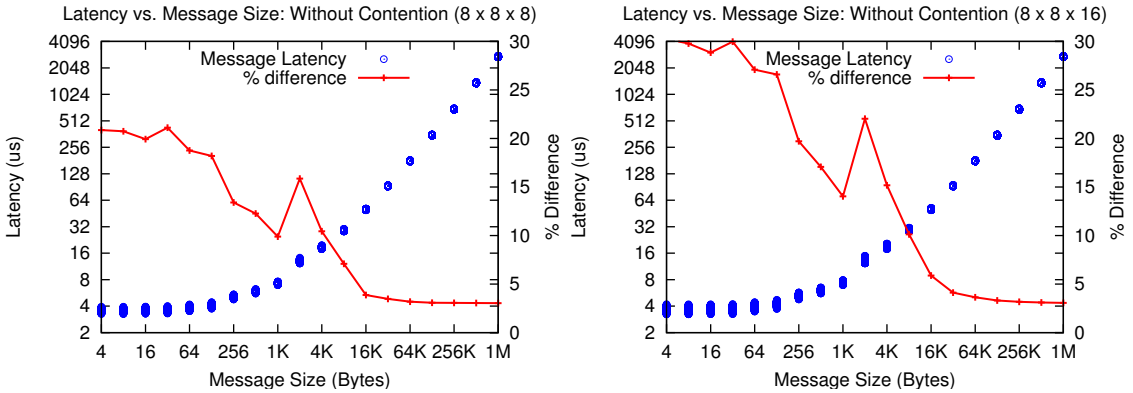


Figure 4.3: Plots showing the effect of hops on message latencies in absence of contention (for $8 \times 8 \times 8$ and $8 \times 8 \times 16$ sized tori on Blue Gene/P, Benchmark: `WOCON`)

is used where an initial handshake is done before the actual message is sent.

The important observation is that the difference is in the range of 10 to 30% for message sizes up to 8 KB (in the 1024 nodes plot, Figure 4.3). Most fine-grained applications use messages which fall in this range and hence it is not wise to always assume that message latencies do not depend on hops for most practical message sizes. Strong scaling of problems to very large number of processors also puts message sizes in this range. Another observation is that, for a fixed message size, the difference between minimum and maximum latencies increases with the increase in diameter of the partition. 128 and 256 node partitions are not complete

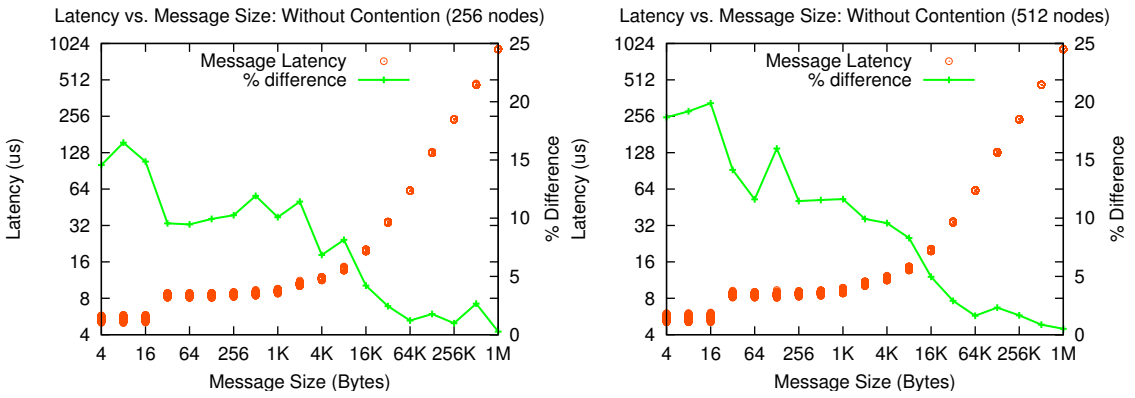


Figure 4.4: Plots showing the effect of number of hops on message latencies in absence of contention (for 256 and 512 nodes of XT3, Benchmark: `WOCON`)

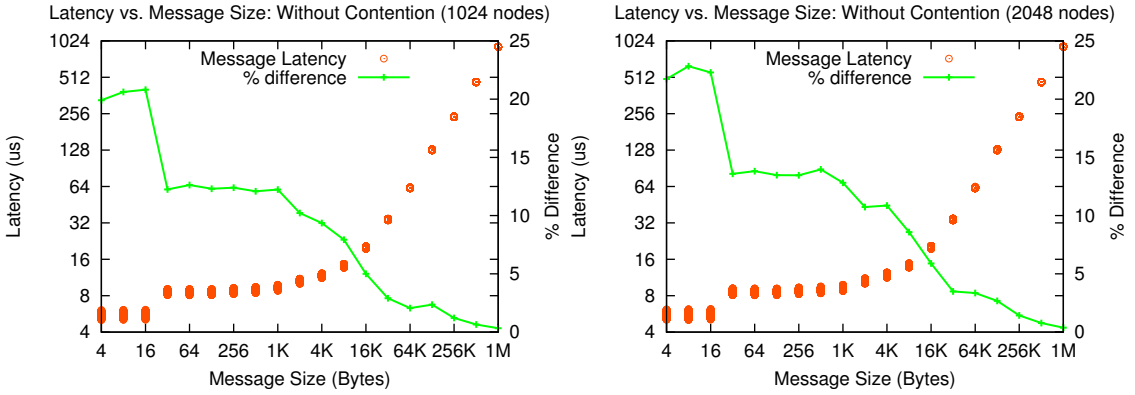


Figure 4.5: Plots showing the effect of number of hops on message latencies in absence of contention (for 1024 and 2048 nodes of XT3, Benchmark: `WOCON`)

tori in all dimensions and hence their diameter is the same as that of the 512 node partition – 12. The diameter of the 1024 node partition is 16 and hence a steep increase in the percentage difference for the small and medium messages (as an example the % difference for a 64 byte message increases from 19 to 27 as we go from the third plot to fourth). As we increase the size of the partition from 1K to 64K nodes, the diameter would increase from 16 to 64 and we can imagine the impact that will have on message latencies.

Figures 4.4 and 4.5 shows similar plots for BigBen, the Cray XT3 machine. The XT3 plots were obtained from runs on contiguous allocations of 256 to $1,024$ nodes of BigBen. Since runs were performed under similar conditions on XT3 as on BG/P,

we would expect similar results. As expected, dependence on hops is significant for message sizes up to 8 KB as seen by the lines on the plots. The only difference from the BG/P numbers is that message latencies on XT3 are significantly greater than the observed latencies on BG/P for very small messages.

4.2 WICON: Random Contention Benchmark

The second benchmark is used to quantify message latencies in presence of contention which is a regime not handled by the basic model of wormhole routing discussed earlier. It should be noted that unlike `WOCON`, this benchmark places one MPI task on each core to create as much contention as possible. All MPI tasks are grouped into pairs and the smaller rank in the pair sends messages of size B bytes to its partner and awaits a reply. All pairs do this communication simultaneously (Figure 4.6). The average time for the message sends is recorded for different message sizes (see Algorithm 4.2).

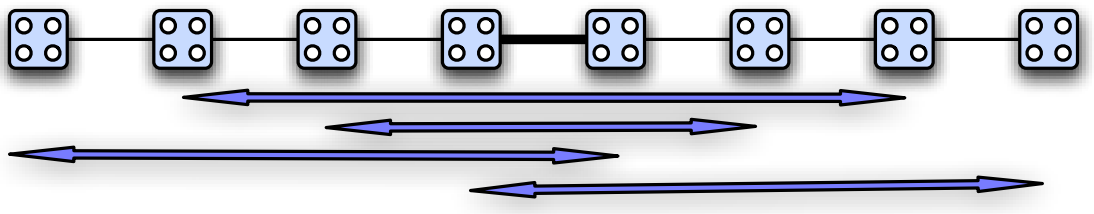


Figure 4.6: Communication patterns in the `WICON` benchmark. This diagram is a simplified one-dimensional version of the pattern in three-dimension (3D). The random pairs are chosen from all over the 3D partition.

To quantify the effect of hops on message latencies this benchmark is run in two modes:

- Near Neighbor Mode (NN): The ranks which form a pair only differ by one. This ensures that everyone is sending messages only 1 hop away (in a torus).

- Random Processor Mode (RND): The pairs are chosen randomly and thus they are separated by a random number of links.

Algorithm 4.2 Code fragments showing the core of `WICON` Benchmark

```
pe = partner[myrank];
if myrank < pe then
    sendTime = MPI_Wtime()
    for i := 1 to num_msgs do
        MPI_Send(send_buf, msg_size, MPI_CHAR, pe, 1, ...)
    end for
    for i := 1 to num_msgs do
        MPI_Recv(recv_buf, msg_size, MPI_CHAR, pe, 1, ...)
    end for
    recvTime = (MPI_Wtime() − sendTime)/num_msgs
else
    sendTime = MPI_Wtime()
    for i := 1 to num_msgs do
        MPI_Recv(recv_buf, msg_size, MPI_CHAR, pe, 1, ...)
    end for
    for i := 1 to num_msgs do
        MPI_Send(send_buf, msg_size, MPI_CHAR, pe, 1, ...)
    end for
    recvTime = (MPI_Wtime() − sendTime)/num_msgs
end if
```

Figure 4.7 shows the results of running `WICON` in the NN and RND modes on Blue Gene/P, XT3 and XT4. The first plot shows the results of `WICON` on $4,096$ cores of BG/P. It is clear that the random-processor (RND) latencies are more than the near-neighbor (NN) latencies (by a factor of 1.75 for large messages.) This is expected based on the assertion that the number of hops have a significant impact on the message latencies in the presence of contention, which increases with larger messages because of a proportional increase in packets on the network.

Similar experiments were repeated on XT3 and XT4 to understand the effects of contention on Cray XT machines. The second plot in Figure 4.7 presents the results for `WICON` benchmark on $2,048$ cores of XT3 and the third plot for $4,096$ cores of XT4. We see a significant difference between the NN and RND lines (a factor of 2.25 at 1 MB messages for XT3 which is greater than that on BG/P.) This is not

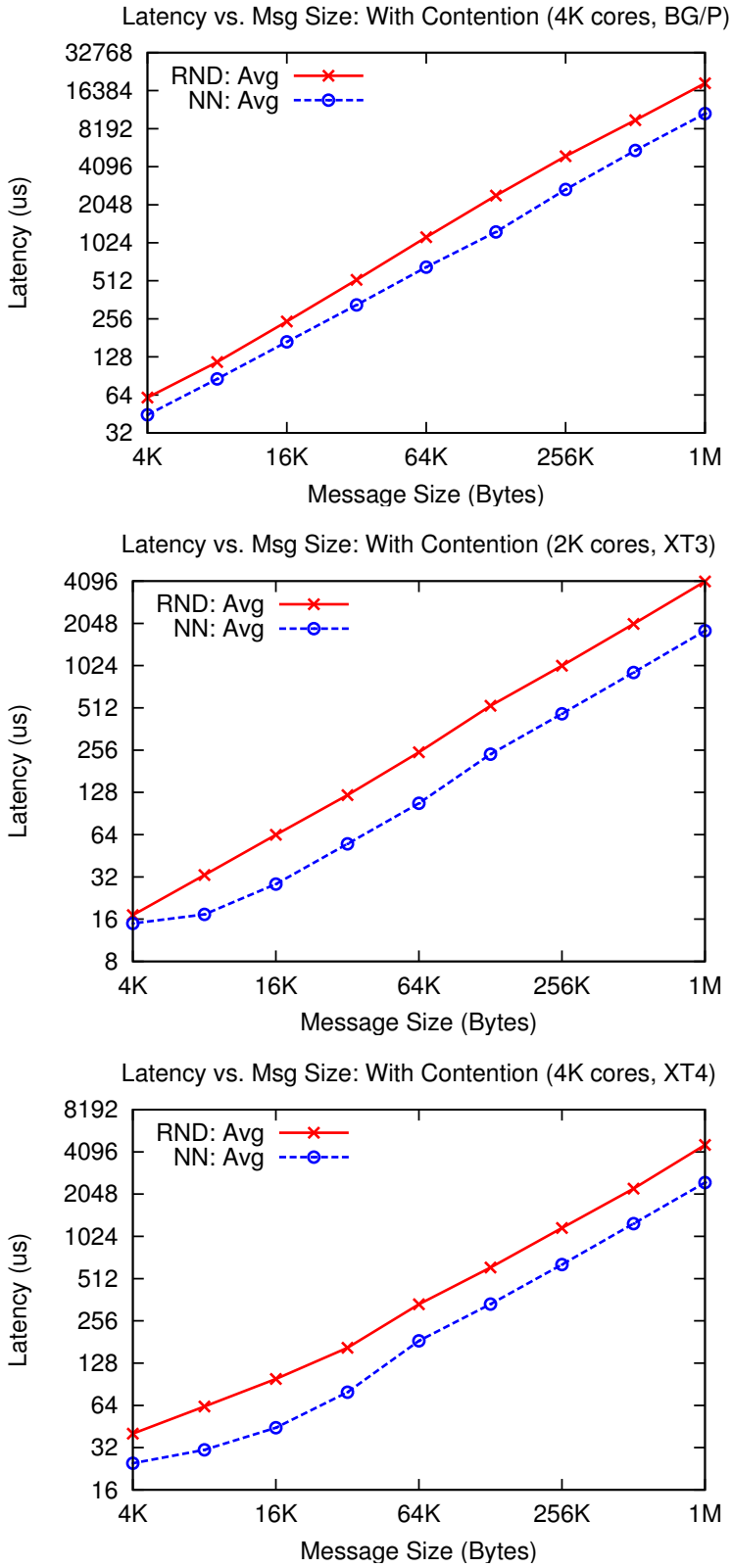


Figure 4.7: Plots showing the results of `WICON` on Blue Gene/P, XT3 and XT4

unexpected and a quantum chemistry code has shown huge benefits (up to 40%) from topology aware mapping on XT3 [64].

4.3 Controlled Contention Experiments

The benchmark in the previous section injects random contention on the network. To quantify the effects of contention under controlled conditions, `WICON` was modified to conduct controlled experiments. The next two subsections list the results of running these experiments where we inject congestion along one dimension of the torus.

4.3.1 Benchmark Stressing a Given Link

In the first experiment, we try to see the effect on message latencies when a particular link is progressively used to send more and more messages. From all ranks in the Z dimension with a specific X and Y coordinate, we choose the pair of ranks in the middle and measure the message latency between the nodes in the pair. Then we keep adding pairs around this pair in the Z dimension and measure the impact of added congestion, on the link in the center. (Figure 4.8).

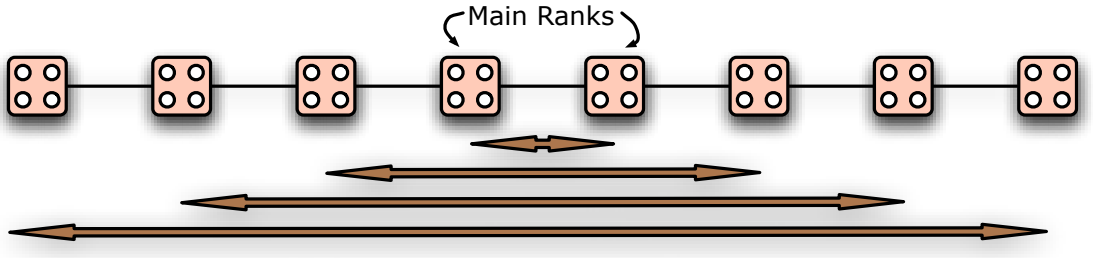


Figure 4.8: For increasing stress on a given link, pairs are chosen along the Z dimension. A baseline run is done with the middle pair and then other pairs are added around the middle one.

Figure 4.9 shows the results from running this benchmark on BG/P, XT3 and

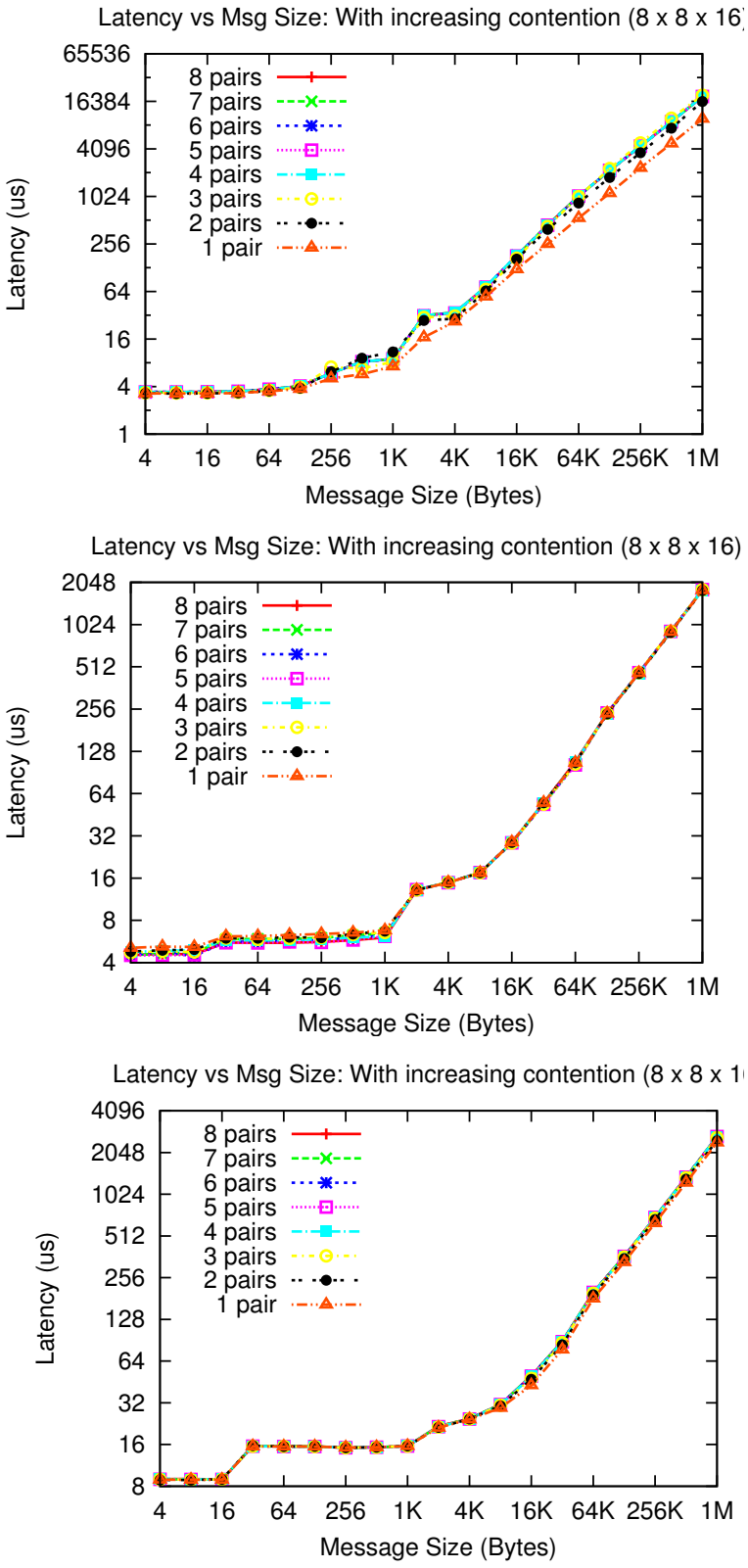


Figure 4.9: Plots showing the results of *stressing-a-link* benchmark on IBM Blue Gene/P, Cray XT3 and XT4

XT4 for different message sizes. On BG/P (top plot), we see that as message size increases, we see increased message latencies with more and more pairs. Additional pairs around the main ranks create contention on the middle links, hence, slowing down their messages. The difference between the message latencies for the 1 pair of nodes versus all 8 pairs of nodes sending messages is about 2 times (similar to what we observed for the `WICON` benchmark.)

The other two plots show the results from running the same benchmark on a 1024 node contiguous partition of XT3 and XT4 respectively. Surprisingly, leaving aside the small perturbation for small messages, we see no impact of stressing a particular link with messages on XT3/XT4. This might be due to better algorithms for congestion control in the SeaStar routers or the interconnect or in the MPI implementation.

4.3.2 Benchmark Using Equidistant Pairs

In this experiment, we try to introduce the same amount of contention on all links of the network (that is each link has the same number of messages passing through it) and try to see its effects. Similar to the `WICON` benchmark, all ranks are divided into pairs but now the pairs are chosen such that they are a fixed number of hops, say n, away from each other. All pairs send messages simultaneously and the average time for message sends of different sizes for varying hops is recorded. Pairs are chosen only along one dimension of the torus, in this case, the Z dimension (see communication pattern in Figure 4.10).

Figure 4.11 shows the results of running this benchmark on BG/P, XT3 and XT4. On each plot there are several lines, one each for a specific pairing where the communicating ranks are n hops away. The tests were done on a torus of dimensions $8 \times 8 \times 16$. Since messages are sent along Z, maximum number of hops possible is 8 and hence there are 8 lines on the plot. The Blue Gene/P plot (top) shows

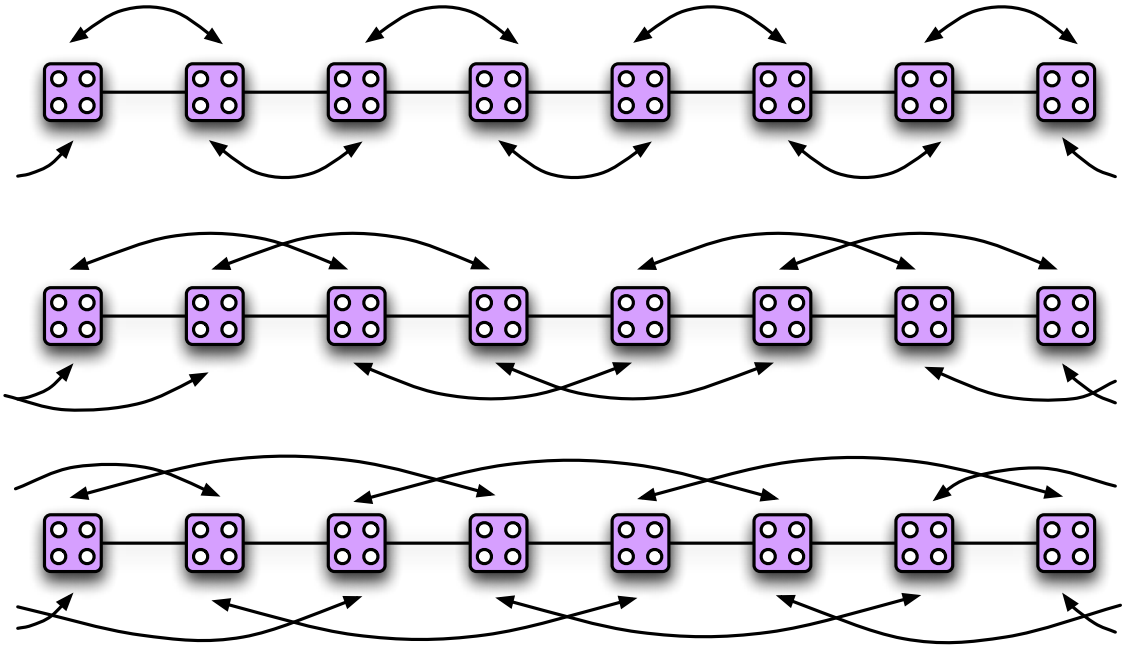


Figure 4.10: For creating pairs of processors, with the same distance between the partners in each pair, strategies as shown in this diagram were used. Pairs are created along the Z dimension and here we show distance=1, 2 and 3.

that the message latencies for large messages for the 1 hop and 8 hops case can differ by a factor of 16! As all messages travel more hops, links are shared by more and more messages increasing the contention on the network and decreasing the available effective bandwidth. This is what applications have to deal with during communication. This huge difference between message latencies indicates that it is very important to keep communicating tasks close by and minimize contention on the network. This is especially true for communication bound applications.

The second plot shows the results from the same benchmark on XT3. In this case, the difference between latencies for large messages is around 2 times. This deviation from the results on BG/P needs further analysis. One possible reason for this might be contention for the Hyper Transport (HT) link which connects the nodes to the SeaStar router instead of the network links. Another reason might be higher bandwidth and better capability of XT3 to handle random contention.

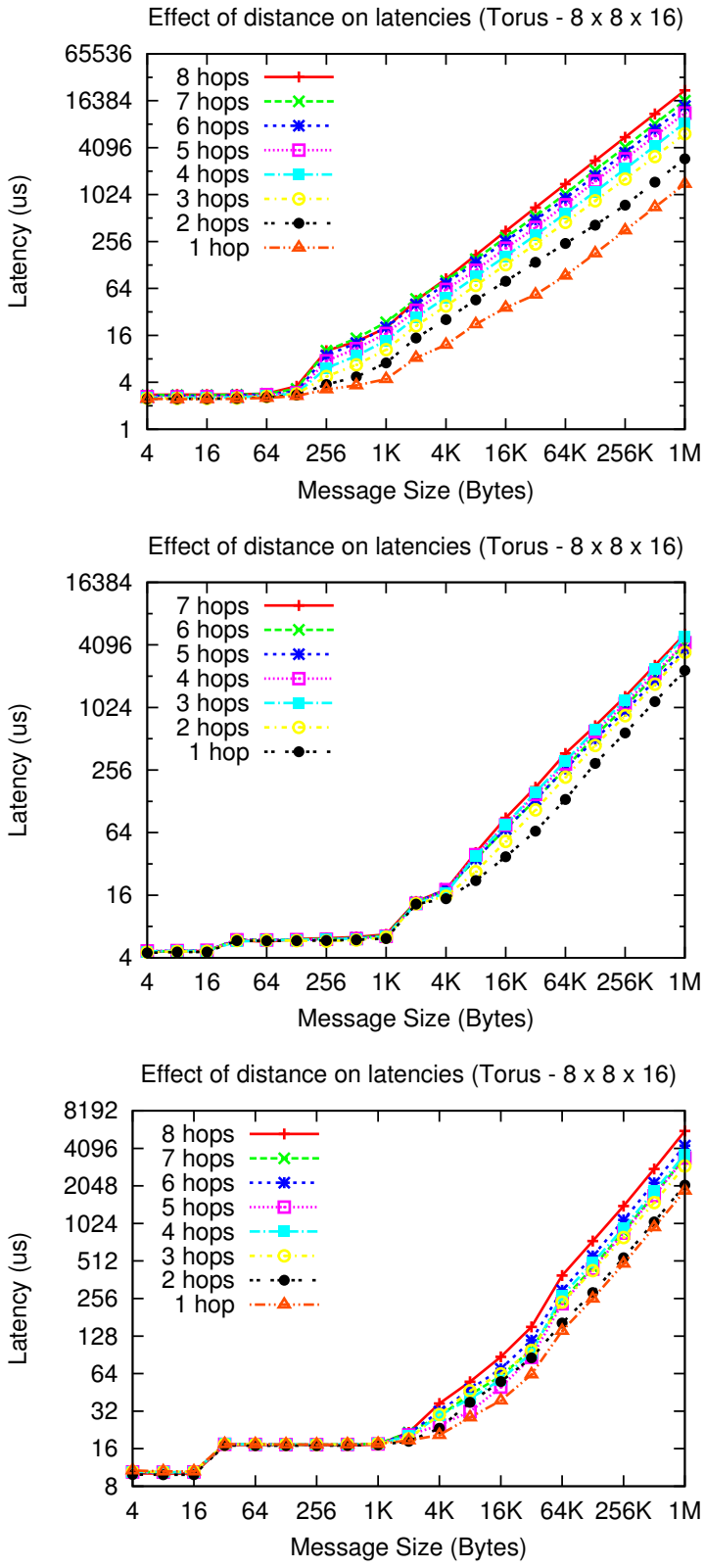


Figure 4.11: Plots showing the results of the *equidistant-pairs* benchmark on Blue Gene/P, XT3 and XT4

However, on XT4 (bottom plot), the difference between 1-hop and 8-hops latencies is 3 times which can be attributed to having more cores on each node.

In summary, results from the benchmarks above support our claim that interconnect topology can impact application performance. Hence it is important to consider the topology, map communicating neighbors closer and minimize contention on the network.

5 Hop-bytes as an Evaluation Metric

The volume of inter-processor communication can be characterized by the *hop-bytes* metric which is the weighted sum of message sizes where the weights are the number of hops (links) traveled by the respective messages. Hop-bytes can be calculated by the equation,

$$HB = \sum_{i=1}^{n} d_i \times b_i \tag{5.1}$$

where d_i is the number of links traversed by message i and b_i is the message size in bytes for message i and the summation is over all messages sent.

Hop-bytes is an indication of the average communication load on each link on the network. This assumes that the application generates nearly uniform traffic over all links in the partition. The metric does not give an indication of hot-spots generated on specific links on the network but is an easily derivable metric and correlates well with actual application performance.

In VLSI circuit design and early parallel computing work, emphasis was placed on another metric called maximum dilation which is defined as,

$$d(e) = max\{d_i | e_i \in E\} \tag{5.2}$$

where d_i is the dilation of the edge e_i. Dilation for an edge e_i is the number of hops between the end-points of the edge in the host graph. This metric aims at minimizing the longest length of the wire in a circuit. We claim that reducing the largest number of links traveled by any message is not as critical as reducing the average hops across all messages.

5.1 Experiment 1

An MPI benchmark was created to justify the claim that hop-bytes is a more suitable metric than maximum dilation. In the basic configuration of this benchmark, every pair talks with its six neighbors (two in each dimension) to create some background communication. We then add one of these patterns: 1. Each rank sends two messages, one to a neighbor three hops away and another to a neighbor six hops away (*avg: 1.88 max: 6* line on the plots), or 2. Each rank sends a message to a neighbor nine hops away (*avg: 1.88 max: 9* line on the plots.) The *avg: 1 max: 1* line due to six near-neighbor messages is plotted as a baseline.

While the difference in the maximum hops between the two cases is three, the total hop bytes is the same and hence, we expect to see similar performance. Figure 5.1 shows the results obtained from running this benchmark. We ran the benchmark on ANL's Blue Gene/P (BG/P) on $1,024$ nodes which form a $8 \times 8 \times 16$ torus. Message size is varied from 4 bytes to 1 MB and multiple trials are done for each point on the plot. Time for the entire communication pattern to finish is recorded between barriers. For small messages, the *max: 9* case does better than *max: 6 case* because only one message is sent in the first one whereas two are sent in the second case (alluding to per-message overheads). For large messages, we see that the lines coincide confirming our predictions that hop-bytes is a more important metric than dilation. Similar results are seen when running this benchmark on Cray XT4 (Figure 5.2).

However, it is important to remember that even hop-bytes is an approximate indication of the contention created by an application since it does not capture the specific loads on each link. It is a measure of the bytes the network has to deliver for an application to run to completion. So, information about specific hot-spots on the network is not expressible through this metric.

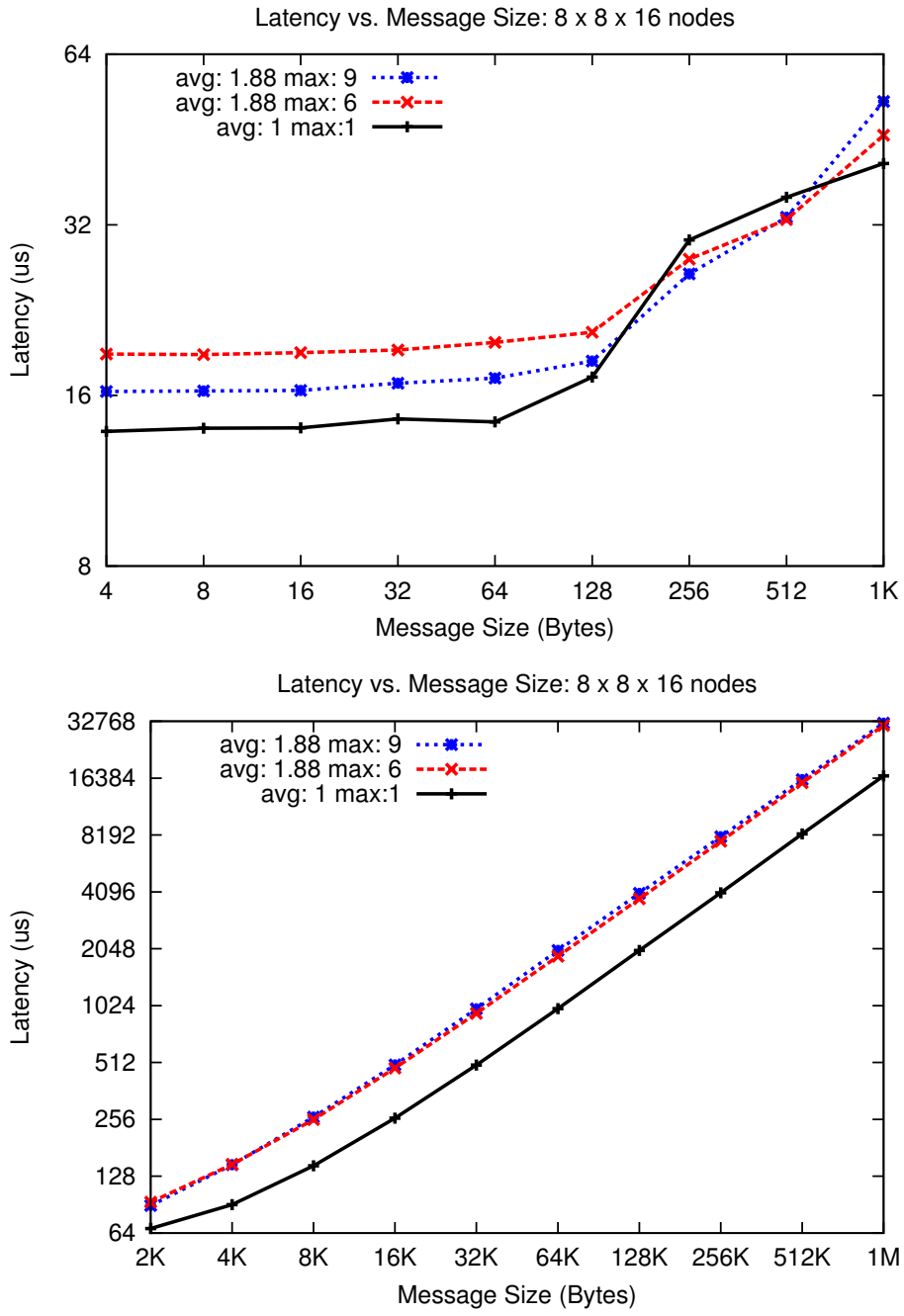


Figure 5.1: Plots showing that average hops is an important factor guiding performance. Runs were done on a $8 \times 8 \times 16$ partition of BG/P

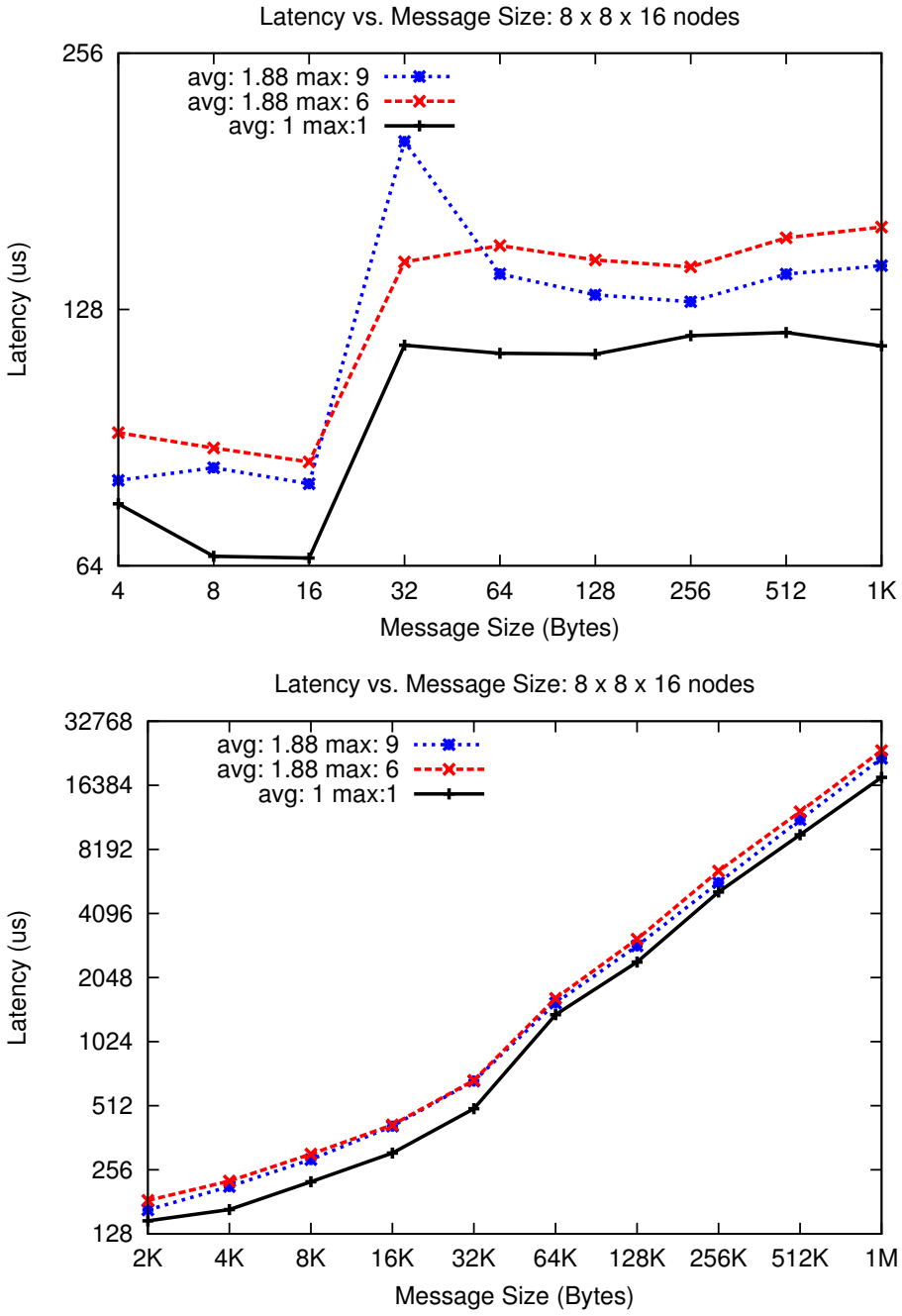


Figure 5.2: Plots showing that average hops is an important factor guiding performance. Runs were done on a $8 \times 8 \times 16$ partition of XT4

5.2 Experiment 2

Yet another MPI benchmark was written to demonstrate that hop-bytes is a good metric overall but we should still consider other factors such as routing protocols and hot-spots. In this benchmark, each MPI rank is paired with a partner and all pairs send messages simultaneously. Both partners in a pair call `MPI_Irecv`, `MPI_Send` and then `MPI_Wait`. The pairs always have the same Z coordinate on the torus. The benchmark is run in different modes depending on how the ranks are grouped into pairs:

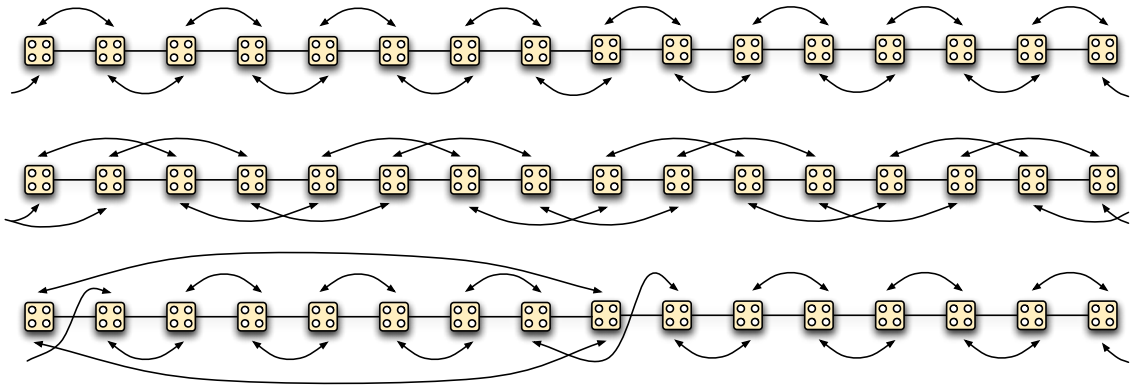


Figure 5.3: Communication patterns along the Z dimension in the synthetic benchmark

- *avg: 1 max: 1*: Every rank is paired with one which is exactly one link away from it along the Z direction.
- *avg: 2 max: 2*: Every rank is paired with one which is exactly two links away from it along the Z direction.
- *avg: 2 max: 8*: Most ranks are paired with a partner which is one link away but one pair is such that the distance between the partners is 8 links or hops. The average hops is 2.

Figure 5.3 shows the pairing of 16 nodes along the Z dimension. Figure 5.4 shows the results from running the benchmark on $1,024$ nodes of BG/P. There are several

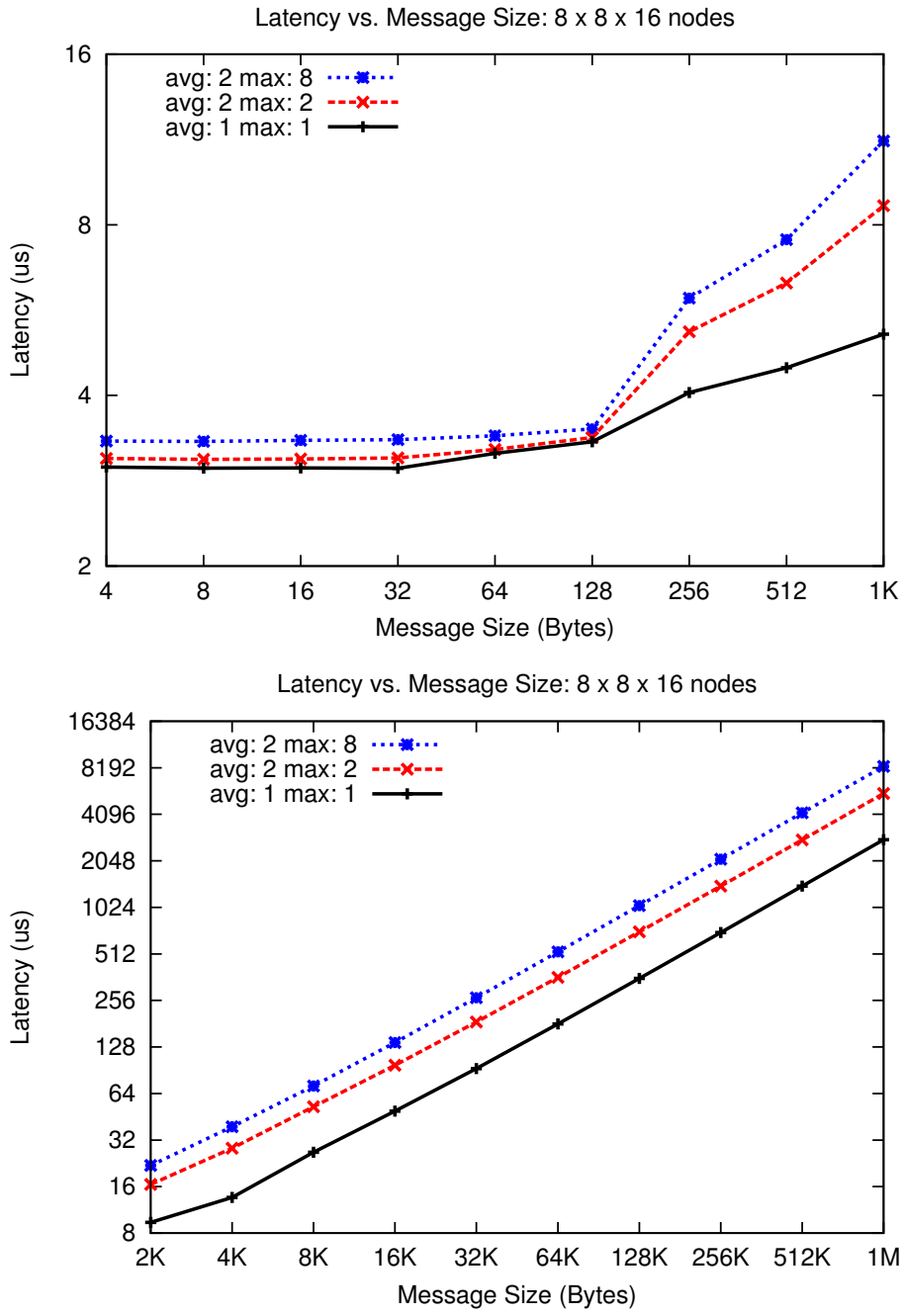


Figure 5.4: Plots showing that maximum dilation can also impact performance. Runs were done on a $8 \times 8 \times 16$ partition of BG/P

observations to be made from these plots. Small messages (less than 128 bytes) are not affected severely because there is negligible contention. For messages greater than 256 bytes, a clear trend is that as the average number of hops increases, the time increases significantly (note, the difference between avg: 1 and avg: 2 lines and that the y-axis has a logarithmic scale). So average hop-bytes is a good indicator of the contention created on the network.

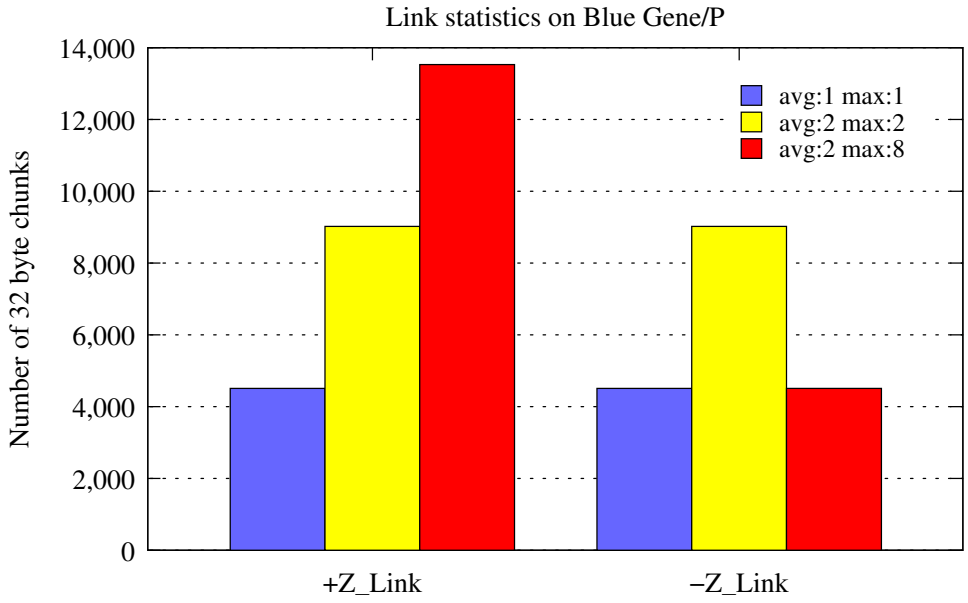


Figure 5.5: HPM Counters Data for Z links in a partition of dimensions $8 \times 8 \times 16$ on Blue Gene/P

Furthermore, when the maximum hops is different between two cases (avg: 2), the performance worsens further. For a further diagnosis of this situation, we used the IBM Performance Monitor library [65] to obtain information about the number of bytes passing through each node. Figure 5.5 shows the data obtained from `BGP_TORUS_ZP_32BCHUNKS` and `BGP_TORUS_ZM_32BCHUNKS` which gives the number of 32 byte chunks passing through the $+Z$ and $-Z$ links of each node. Since there is no communication in the X and Y direction, counters for those are zero. For these profiling runs, only $1,024$-byte messages were used.

The figure shows that when the maximum hops is 8, more packets are sent along the $+Z$ links than the $-Z$ links which leads to a degradation in performance. This exercise shows that subtle routing choices can affect contention and lead to performance degradation. Since the maximum hops are not captured by the hop-bytes metric, it is sometimes relevant to consider that as a factor affecting performance. That said, since hop-bytes is still a good indicator of the communication traffic on the network, we will use it as a basis for evaluation of mapping algorithms in the subsequent chapters.

6 Processor Graph: Topology Manager API

Mapping of communication graphs onto the processor graph requires information about the machine topology at runtime. The application should be able to query the runtime to get information like the dimensions of the allocated processor partition, mapping of ranks to physical nodes *et cetera*. However, the mapping interface should be simple and should hide machine-specific details from the application. We have developed a Topology Manager API which provides a uniform interface to the application developer for determining topology information. With the API, application specific task mapping decisions require no architecture or machine specific knowledge (BG/L or XT3 for example).

6.1 The Topology Manager

We now describe this API, which we call the *Topology Manager* that can be used by any application for mapping of objects to processors. The library is generic and can be used in parallel programs written in different programming models. The Topology Manager API provides various functions which can be grouped into the following categories:

1. **Size and properties of the allocated partition:** At runtime, the application needs to know the dimensions of the allocated partition (`getDimNX, getDimNY, getDimNZ`), number of cores per node (`getDimNT`) and whether we have a torus or mesh in each dimension (`isTorusX, isTorusY, istorusZ`).

2. **Properties of an individual node:** Each task in the application is placed on some processor which has X, Y and Z coordinates. The interface also provides calls to convert from ranks (IDs of the tasks) to physical coordinates and vice-versa (`rankToCoordinates, coordinatesToRank`).

3. **Additional Functionality:** Mapping algorithms often need to calculate number of hops between two ranks or pick the closest rank to a given rank from a list. Hence, the API provides functions like `getHopsBetweenRanks, pickClosestRank` and `sortRanksByHops` to facilitate mapping algorithms.

The functions described above are implemented on different IBM and Cray machines and the application sees a uniform interface. The next section describes the process of deriving this information at runtime.

6.2 Topology Discovery on Torus Machines

We now discuss the process of extracting this information from the system at runtime and why it is useful to use the Topology Manager API on different machines.

6.2.1 IBM Blue Gene Machines

On Blue Gene/L and Blue Gene/P [7], topology information is available through system calls to the "BGLPersonality" and "BGPPersonality" data structures, respectively. It is useful to use the Topology Manager API instead of the system calls for two reasons. First, these system calls can be expensive (especially on Blue Gene/L) and so it is advisable to avoid doing too many of them. The API does a few system calls to obtain enough information so that it can construct the topology information itself. Topology information is then available throughout the execution of the program with low overhead.

The second reason is that on Blue Gene/L and Blue Gene/P, there is a limit to the smallest size of a partition which can be allocated (32 nodes on the Watson BG/L and 64 nodes on the ANL BG/P). If fewer nodes than this smallest unit are requested, the smallest partition will be allocated though only a subset of the nodes in it are used by the application. In these cases, the lower level system calls give information about the entire booted partition and not the actual nodes being used. Our API calculates which portion of the allocated partition is being used when you use fewer nodes than the allocated partition and gives the correct information.

6.2.2 Cray XT Machines

Cray machines have been designed with a significant overall bandwidth, and possibly for this reason, documentation for topology information was not readily available at the installations we used. We thank Shawn Brown from PSC, Larry Kaplan from Cray and William Renaud from ORNL for helping us obtain topology information through personal communication. We hope that the information provided here will be useful to other application programmers.

Obtaining topology information on XT machines is a two step process: 1. First we obtain the node ID (`nid`) corresponding to a given MPI rank (`pid`) which tells us which physical node a given MPI rank is placed on. This requires using different system calls on XT3 and XT4: `cnos_get_nidpid_map` available through "catamount/cnos_mpi_os.h" and `PMI_Portals_get_nidpid_map` available from "pmi.h". These calls provide a map for all ranks in the current job and their corresponding node IDs. 2. The second step is obtaining the physical coordinates for a given node ID. This can be done by using the system call `rca_get_meshcoord` from "rca_lib.h". Once we have information about the physical coordinates for all ranks in the job, the API derives information such as the extent of the allocated partition by itself (this assumes that the machine has been reserved and we have a contiguous parti-

tion). Using the size of the partition and the size of the total machine ($11 \times 12 \times 16$ for BigBen and $21 \times 16 \times 24$ for Jaguar), the API can determine if there is a mesh or torus in each direction. Again, once the TopoManager object is instantiated, it stores this information and does not make system calls again.

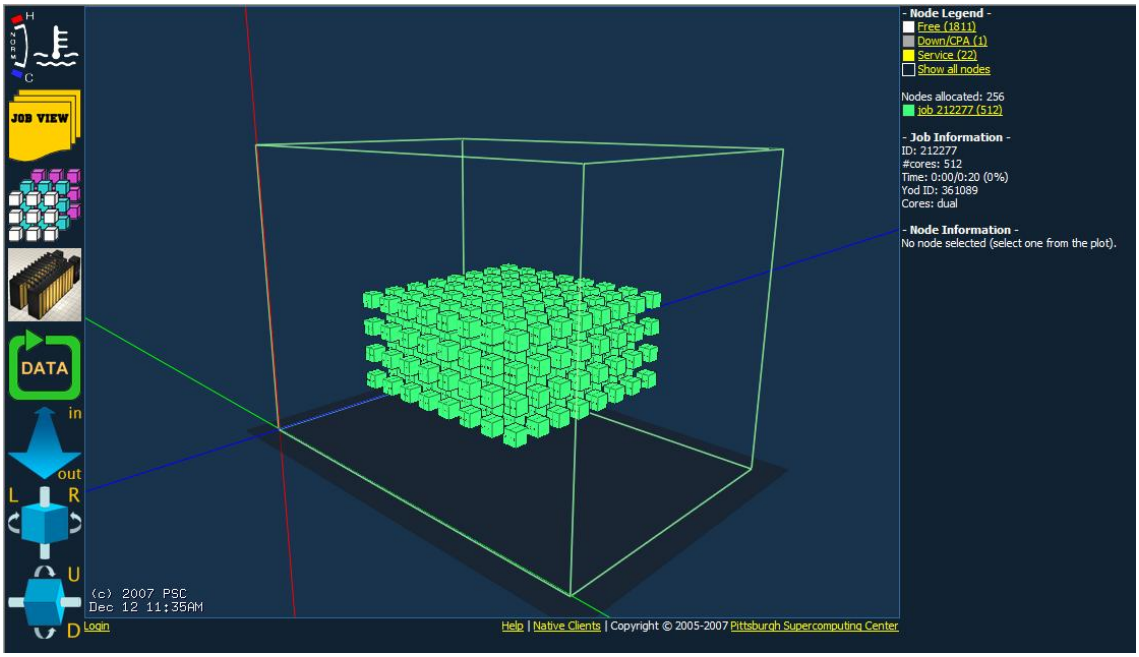


Figure 6.1: Allocation of 256 continuous nodes on the XT3 machine at PSC

Figure 6.1 shows a specific allocation of 256 contiguous nodes in a 3D shape of dimensions $8 \times 8 \times 4$. This snapshot was generated by the BigBen monitor available on the PSC website.

The API provides a uniform interface which works on all the above mentioned machines and hides architecture specific details from the application programmer. This API can be used as a library for Charm++, MPI or any other parallel program. The next chapter describes the use of object-based decomposition and the Topology Manager API in some production codes.

7 Application-specific Mapping

$\mathcal{N}$etwork topology information can be used to optimize communication by mapping parallel applications onto the allocated job partition. This chapter presents mapping techniques and performance improvements obtained for two parallel applications written in Charm++: OpenAtom and NAMD. Communication information about the applications is assumed and not obtained automatically.

7.1 OpenAtom

An accurate understanding of phenomena occurring at the quantum scale can be achieved by considering a model representing the electronic structure of the atoms involved. The CPAIMD method [66] is one such algorithm which has been widely used to study systems containing $10 - 10^3$ atoms [67–77]. To achieve a fine-grained parallelization of CPAIMD, computation in OpenAtom [16] is divided into a large number of objects, enabling scaling to tens of thousands of processors. We will look at the parallel implementation of OpenAtom, explain the communication involved and then discuss the topology aware mapping of its objects.

In an *ab initio* approach, the system is driven by electrostatic interactions between the nuclei and electrons. Calculating the electrostatic energy involves computing several terms. Hence, CPAIMD computations involve a large number of phases with high inter-processor communication: (1) quantum mechanical kinetic energy of non-interacting electrons, (2) Coulomb interaction between electrons or the Hartree energy, (3) correction of the Hartree energy to account for the quantum nature of

the electrons or the exchange-correlation energy, and (4) interaction of electrons with atoms in the system or the external energy. These phases are discretized into a large number of objects which generates a high volume of communication, but this ensures efficient interleaving of work. The entire computation is divided into ten phases which are parallelized by decomposing the physical system into fifteen *chare arrays*. For a detailed description of this algorithm please refer to [16].

7.1.1 Communication Dependencies

The ten phases referred to in the previous section are parallelized by decomposing the physical system into fifteen *chare arrays* of different dimensions (ranging between one and four). A simplified description of these arrays (those most relevant to the mapping) follows:

GSpace and RealSpace: These arrays contain the g-space and real-space representations of each electronic state [66]. Each electronic state is represented by a 3D array of complex numbers. OpenAtom decomposes this data into a 2D *chare array* of objects. Each object holds a plane of one states (see Figure 7.1). The *chare arrays* are represented by $G(s,p)$ $[n_s \times N_g]$ and $R(s,p)$ $[n_s \times N]$ respectively. GSpace and RealSpace interact through transpose operations (as part of a Fast Fourier Transform) in Phase I and hence all planes of one state of GSpace interact with all planes of the same state of RealSpace.

RhoG and RhoR: These arrays hold the g-space and real-space representations of electron density and are decomposed into 1D and 2D *chare arrays*, respectively. They are represented as $G_\rho(p)$ and $R_\rho(p,p')$. RealSpace interacts with RhoR through reductions in Phase II. RhoG is obtained from RhoR in Phase III through two transposes.

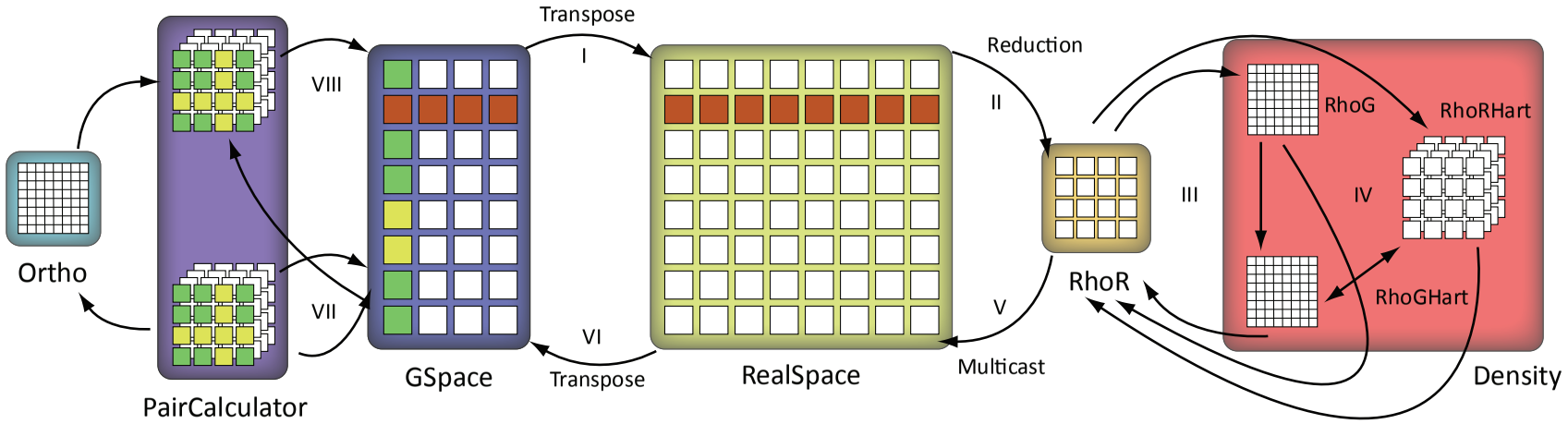


Figure 7.1: Decomposition of the physical system into chare arrays (only important ones shown for simplicity) in OpenAtom

PairCalculators: These 3D *chare arrays* are used in phase IV. They communicate with GSpace through multicasts and reductions. They are represented as $P_c(s, s', p)$ $[n_s \times n_s \times N_g]$. All elements of the GSpace array with a given state index interact with all elements of the PairCalculator array with the same state in one of their first two dimensions.

7.1.2 Mapping Techniques

OpenAtom provides us with a scenario where the load on each object is static (under the CPAIMD method) and the communication is regular and clearly understood. Hence, it should be possible to intelligently map the arrays in this application to minimize inter-processor communication and maintain load balance. OpenAtom has a default mapping scheme, but it should be noted that the default mapping is far from random. It is the mapping scheme used on standard fat-tree networks, wherein objects which communicate frequently are co-located on processors within the constraints of even distribution. This reduces the total communication volume. It only lacks a model for considering the relative distance between processors in its mapping considerations. We can do better than the default mapping by using the communication and topology information at runtime. We now describe how a complex interplay (of communication dependencies) between five of the *chare arrays* is handled by our mapping scheme.

GSpace and RealSpace are 2D *chare arrays* with states in one dimension and planes in the other. These arrays interact with each other through transpose operations where all planes of one state in GSpace, $G(s, *)$ talk to all planes of the same state, $R(s, *)$ in RealSpace (state-wise communication). The number of planes in GSpace is different from that in RealSpace. GSpace also interacts with the PairCalculator arrays. Each plane of GSpace, $G(*, p)$ interacts with the corresponding plane, $P(*, *, p)$ of the PairCalculators (plane-wise communication) through multi-

casts and reductions. So, GSpace interacts state-wise with RealSpace and plane-wise with PairCalculators. If all planes of GSpace are placed together, then the transpose operation is favored, but if all states of GSpace are placed together, the multicasts/reductions are favored. To strike a balance between the two extremes, a hybrid map is built, where a subset of planes and states of these three arrays are placed on one processor.

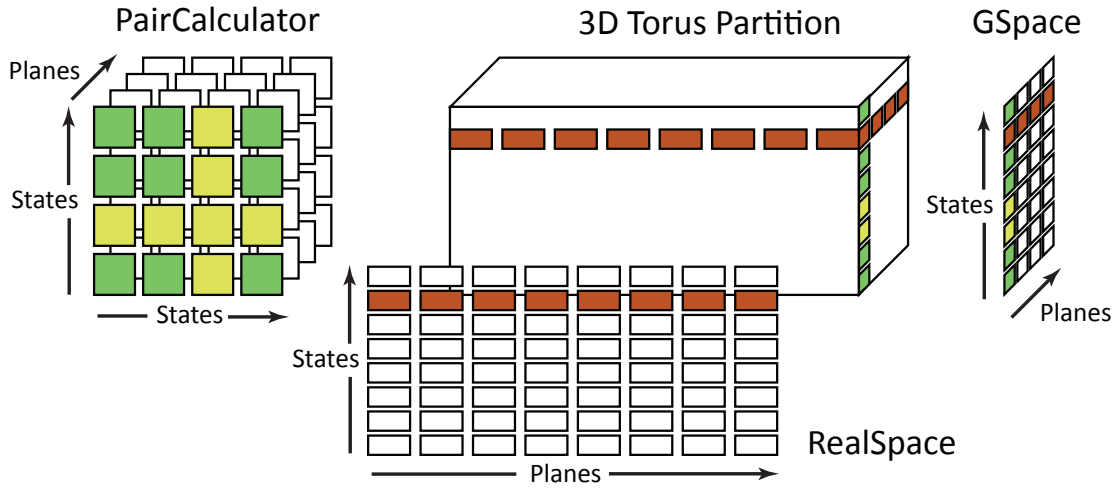


Figure 7.2: Mapping of a few OpenAtom arrays to the 3D torus of the machine

Mapping GSpace and RealSpace Arrays: Initially, the GSpace array is placed on the torus and other objects are mapped relative to its mapping. The 3D torus is divided into rectangular boxes (which will be referred to as "prisms") such that the number of prisms is equal to the number of the planes in GSpace. The longest dimension of the prism is chosen to be same as one dimension of the torus. Each prism is used for all states of one plane of GSpace. Within each prism for a specific plane, the states in G(*, p) are laid out in increasing order along the long axis of the prism. Once GSpace is mapped, the RealSpace objects are placed. Prisms perpendicular to the GSpace prisms are created, which are formed by including processors holding all planes for a particular state of GSpace, $G(s, *)$. These prisms are perpendicular to the GSpace prisms and the corresponding states of RealSpace, $R(s, *)$ are mapped

on to these prisms. Figure 7.2 shows the GSpace objects (on the right) and the RealSpace objects (in the foreground) being mapped along the long dimension of the torus (box in the center).

Mapping of Density Arrays: RhoR objects communicate with RealSpace plane-wise and hence $R_\rho(p, *)$ have to be placed close to $R(*, p)$. To achieve this, we start with the centroid of the prism used by $R(*, p)$ and place RhoR objects in proximity to it. RhoG objects, $G_\rho(p)$ are mapped near RhoR objects, $R_\rho(p, *)$ but not on the same processors as RhoR to maximize overlap. The density computation is inherently smaller and hence occupies the center of the torus.

Mapping PairCalculator Arrays: Since PairCalculator and GSpace objects interact plane-wise, the aim is to place $G(*, p)$ and $P(*, *, p)$ nearby. *Chares* with indices $P(s1, s2, p)$ are placed around the centroid of $G(s1, p), ..., G(s1 + block\_size, p)$ and $G(s2, p), ..., G(s2 + block\_size, p)$. This minimizes the hop-count for the multicast and reduction operations. The result of this mapping co-locates each plane of PairCalculators (on the left in Figure 7.2) with its corresponding plane of GSpace objects within the GSpace prisms.

The mapping schemes discussed above substantially reduce the hop-count for different phases. They also restrict different communication patterns to specific prisms within the torus, thereby reducing contention and ensuring balanced communication throughout the torus. State-wise and plane-wise communication is confined to different (orthogonal) prisms. This helps avoid scaling bottlenecks as we will see in Section 7.1.4. These maps perform no better (and generally slightly worse) than the default maps on architectures which have more uniform network performance, such as Ethernet or Infiniband.

7.1.3 Time Complexity

Although maps are created only once during application start-up, they must still be efficient in terms of their space and time requirements. The memory cost of these maps grows linearly (4 integers per object) with the number of objects, which is a few megabytes in the largest system studied. The runtime cost of creating the most complex of these maps is $\mathcal{O}(n^{3/2}log(n))$ where n is the number of objects. Despite this complexity, this time is sufficiently small that generating the maps for even the largest systems requires only a few minutes.

Table 7.1 shows the time it takes to construct two of the complex maps (RealSpace and RealParticlePlane) when running WATER_256_70Ry. Even on 8192 processors, it takes less than one-third of a minute to create a RealParticlePlane map. Algorithm 7.1 shows the pseudo-code for creating the RealSpace map from the GSpace map. The creation of RealParticlePlane is similar but there are more sorting calls inside the first `for` loop, which increases it running time. RealParticlePlane objects are not on the critical path of execution and hence the mapping of this array can be turned off if it takes a long time for very large runs. This illustrates a common tradeoff: it is always important to evaluate if the time spent in computing a mapping is worth the performance benefit achieved from it.

Cores	*RealSpace*	*RealParticlePlane*
1024	0.33	1.48
2048	0.52	3.19
4096	1.00	4.90
8192	3.07	17.89

Table 7.1: Execution time (in seconds) to obtain mapping solutions for RealSpace and RealParticlePlane objects on Blue Gene/P (System: WATER_256M_70Ry)

RealSpace objects are extremely important because of the large communication with GSpace and density objects, and their mapping cannot be ignored. The algo-

Algorithm 7.1 Mapping of RealSpace objects based on the the map for GSpace objects

Input: *nstates* (Number of states in the RealSpace chare array)
nplanes (Number of planes in the RealSpace chare array)
gsmap (Mapping of the GSpace chare array)
Output: *rsmap* Mapping of the RealSpace chare array)
for *state* $\leftarrow$ 1 to *nstates* **do**
  Create a processor list *plist* consisting of processors in $gsmap[state, *]$
  $RSobjs\_per\_pe$ = maximum number of RSMap objects per processor
  for *plane* $\leftarrow$ 1 to *nplanes* **do**
    Exclude processors which have $RSobjs\_per\_pe$ from *plist*
    Sort *plist* by increasing hops from the first processor in the list
    Assign object $rsmap_{state,plane}$ on the first element in *plist*
  end for
end for

rithm above shows that the running time of the algorithm is $nstates \times nplanes \times nstates \times log(nstates)$. Approximating *nstates* and *nplanes* by $n^{1/2}$, the time complexity is $\mathcal{O}(n^{3/2}log(n))$. There may be room for improvement if we can move the sorting out of the inner `for` loop. As an optimization, maps can be stored and reloaded in subsequent runs to minimize restart time. Offline creation of maps using more sophisticated techniques and adapting these ideas to other topologies is an area of future work.

7.1.4 Performance Improvements

To analyze the effects of topology aware mapping in a production science code, 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. The benchmarks simulate the electronic structure of 32 molecules and 256 molecules of water, respectively, with a standard g-space spherical cutoff radius of $|\mathbf{g}|^2_{cut} = 70$ Rydberg (Ry) on the states, at the Γ point (1 k-point), three dimensional periodic boundary conditions, the BLYP generalized gradient corrected

density functional [78, 79] and Martins-Troullier type pseudopotentials [80]. To illustrate that the performance improvements extend beyond benchmarks to production science systems, we also present results for GST_BIG, which is a system being studied by our collaborator, Dr. Glenn J. Martyna. GST_BIG consists of 64 molecules of Germanium, 128 molecules of Antimony and 256 molecules of Tellurium at a cutoff radius of $|\mathbf{g}|^2_{cut} = 20$ Ry on the states, at the Γ point (1 k-point) and otherwise as in the benchmarks above.

Blue Gene/L (IBM T. J. Watson) runs are done in co-processor (CO) mode to use a single core per node. Blue Gene/P (Intrepid at ANL) runs were done in VN mode which uses all four cores per node. Cray XT3 (BigBen at PSC) runs are done in two modes: single core per node (SN) and two cores per node (VN). As shown in Table 7.2, performance improvements from topology aware mapping for BG/L can be quite significant. As the number of cores and likewise, the diameter of the torus grows, the performance impact increases until it is a factor of two faster for WATER_32M_70Ry at 2048 and for WATER_256M_70Ry at 16384 cores. There is a maximum improvement of 40% for GST_BIG. The effect is not as strong in GST_BIG due to the fact that the time step in this system is dominated by a subset of the orthonormalization process which has not been optimized extensively, but a 40% improvement still represents a substantial improvement in time to solution.

	WATER_32M_70Ry		**WATER_256M_70Ry**		**GST_BIG**	
Cores	**Default**	**Topology**	**Default**	**Topology**	**Default**	**Topology**
512	0.274	0.259	-	-	-	-
1024	0.189	0.150	19.10	16.40	10.12	8.83
2048	0.219	0.112	13.88	8.14	7.14	6.18
4096	0.167	0.082	9.13	4.83	5.38	3.35
8192	0.129	0.063	4.83	2.75	3.13	1.89
16384	-	-	3.40	1.71	1.82	1.20

Table 7.2: Execution time per step (in seconds) of OpenAtom on Blue Gene/L (CO mode)

Performance improvements on BG/P are similar to those observed on BG/L (Table 7.3). The improvement for WATER_32M_70Ry is not as remarkable as on BG/L but for WATER_256M_70Ry, we see a factor of 2 improvement starting at 2048 cores. The absolute numbers on BG/P are much better than on BG/L partially because of the increase in processor speeds but more due to the better interconnect (higher bandwidth and an effective DMA engine). The execution time for WATER_256M_70Ry at 1024 cores is 2.5 times faster on BG/P than on BG/L. This is when comparing the VN mode on BG/P to the CO mode on BG/L. If we use only one core per node on BG/P, the performance difference is even greater, but the higher core per node count, combined with the DMA engine and faster network make single core per node use less interesting on BG/P.

	WATER_32M_70Ry		**WATER_256M_70Ry**			**GST_BIG**
Cores	**Default**	**Topology**	**Default**	**Topology**	**Default**	**Topology**
256	0.395	0.324	-	-	-	-
512	0.248	0.205	-	-	-	-
1024	0.188	0.127	10.78	6.70	6.24	5.16
2048	0.129	0.095	6.85	3.77	3.29	2.64
4096	0.114	0.067	4.21	2.17	3.63	2.53
8192	-	-	3.52	1.77	-	-

Table 7.3: Execution time per step (in seconds) of OpenAtom on Blue Gene/P (VN mode)

The improvements from topology awareness on Cray XT3, presented in Table 7.4 are comparable to those on BG/L and BG/P. The improvement of 27% and 21% on XT3 for WATER_256_70Ry and GST_BIG at $1,024$ cores is greater than the improvement of 14% and 13% respectively on BG/L at $1,024$ cores in spite of a much faster interconnect. However, on $2,048$ cores, performance improvements on the three machines are similar.

The improvement trends plotted in Figure 7.3 lead us to project that topology aware mapping should yield improvements proportional to torus size on larger Cray

	WATER_32M_70Ry		WATER_256M_70Ry		GST_BIG	
Cores	**Default**	**Topology**	**Default**	**Topology**	**Default**	**Topology**
			Single core per node			
512	0.124	0.123	5.90	5.37	4.82	3.86
1024	0.095	0.078	4.08	3.24	2.49	2.02
			Two cores per node			
256	0.226	0.196	-	-	-	-
512	0.179	0.161	7.50	6.58	6.28	5.06
1024	0.144	0.114	5.70	4.14	3.51	2.76
2048	0.135	0.095	3.94	2.43	2.90	2.31

Table 7.4: Execution time per step (in seconds) of OpenAtom on XT3 (SN and VN mode)

XT installations. The difference in processor speeds is approximately a factor of 4 (XT3 2.6 GHz, BG/L 700 MHz), which is reflected in the performance for the larger grained OpenAtom results on XT3 when comparing single core per node performance. The difference in network performance is approximately a factor of 7 (XT3 1.1 GB/s, BG/L 150 MB/s), when considering delivered bandwidth as measured by HPC Challenge [56] ping pong. This significant difference in absolute speed and computation/bandwidth ratios does not shield the XT3 from performance penalties from topology ignorant placement schemes. BG/P and BG/L show similar performance improvements which is expected since the BG/P architecture is similar to that of BG/L with slightly faster processors and increased network bandwidth.

OpenAtom is highly communication bound (as briefly discussed in the Introduction). Although Charm++ facilitates the exploitation of the available overlap and latency tolerance across phases, the amount of latency tolerance inevitably drops as the computation grain size is decreased by the finer decomposition required for larger parallel runs. It is important to consider the reasons for these performance improvements in more detail. Figure 7.4 compares idle time as captured by the Projections profiling system in Charm++ for OpenAtom on BG/L for the de-

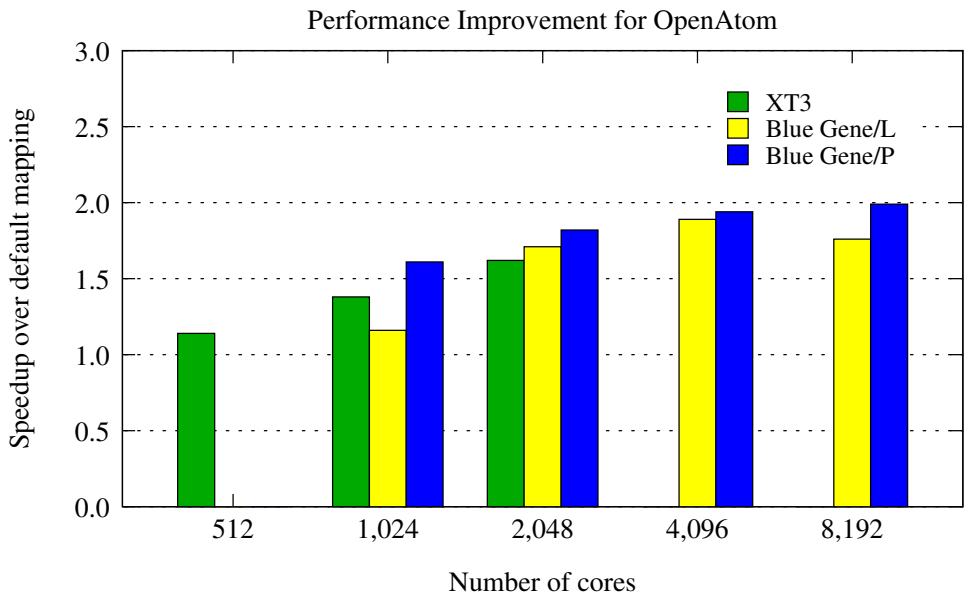


Figure 7.3: Comparison of performance improvements on BG/L, BG/P and XT3 using topology aware mapping (for WATER_256M_70Ry)

fault mapping, versus the topology aware mapping. A processor is idle whenever it is waiting for messages to arrive. It is clear from Figure 7.4 that the factor of two speed increase from topology awareness is reflected directly in relative idle time and that the maximum speed increase which can be obtained from topology aware mapping is a reduction in the existing idle time.

It is illuminating to study the exact cause for this reduction in idle time. To that end, we ported IBM's High Performance Monitor library [65] for Blue Gene/P's Universal Performance Counters to Charm++, and enabled performance counters for a single time step in WATER_256M_70Ry in both topology aware and non-topology aware runs. We added the per node torus counters (BGP_TORUS_*_32BCHUNKS), to produce the aggregate link bandwidth consumed in one step across all nodes to obtain the results in Figure 7.5. This gives us an idea of the fraction of the total bandwidth across all links on the network used in one step. If messages travel fewer hops due to topology aware placement, it will lead to smaller bandwidth consumption, thereby indicating less contention on the network. It is clear from the figure,

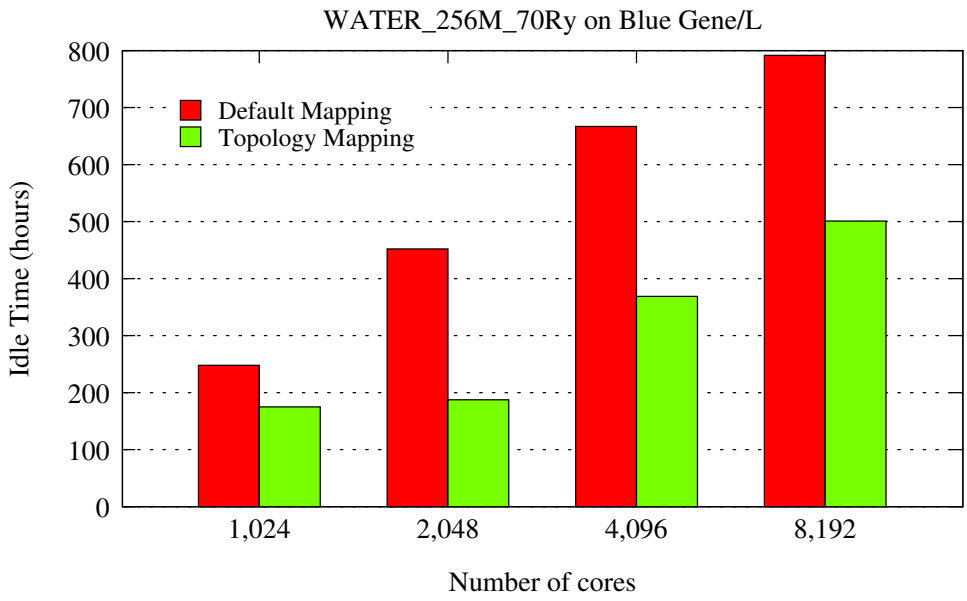


Figure 7.4: Effect of topology aware mapping on idle time (time spent waiting for messages)

that topology aware mapping results in a significant reduction, by up to a factor of two, in the total bandwidth consumed by the application. This more efficient use of the network is directly responsible for the reduction in latency due to contention and decreased idle time.

7.1.5 Multiple Application Instances

The discussion and results so far pertain to using OpenAtom for a single simulation of the evolution of the electronic states of a particular system. More information and/or improved accuracy can be obtained through the application of replica methods, which combine the results of multiple evolutions of a system. For example, applications of the path integral formulation combine multiple replicas to improve the accuracy of atomic positions. Similarly, when studying metals it is necessary to sample multiple K-points of the first Brillouin zone (the default scheme is $k = 0$) for convergence. Each of these methods and many similar applications not considered here, share the trait that most of the computation for each trajectory remains

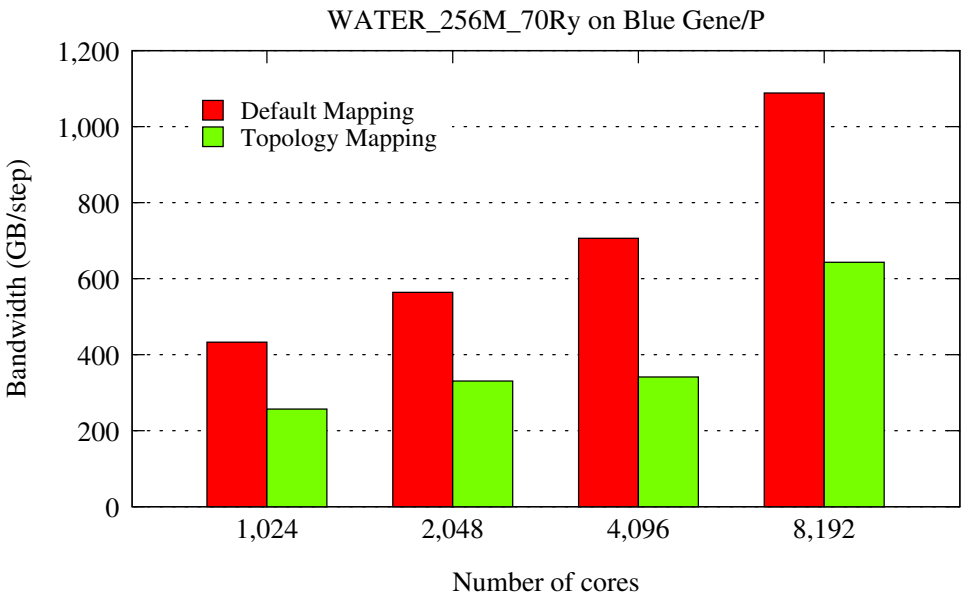


Figure 7.5: Effect of topology aware mapping on aggregate bandwidth consumption per step - smaller link bandwidth utilization suggests reduction in hops traversed by messages and hence reduction in contention

independent and their results are combined only within one phase of the computation. We can therefore treat these as largely independent instances of the simulation for most optimization considerations. In OpenAtom, we express each of these as separate instances of the Charm++ arrays.

Multiple instances of OpenAtom are mapped to different parts of the partition to prevent different instances from contending for the same resources. We divide the allocated partition along the longest dimension, into sub-partitions equal in number to the number of instances. We obtain the mappings for the first instance on the first sub-partition and then translate the maps along the longest dimension to obtain the maps for the other instances. Figure 7.6 shows the mapping of four OpenAtom instances on to a torus of dimensions $8 \times 4 \times 8$. The torus is split into four parts along the X dimension (of size 8). Two alternate instances are shown while the other two have been made invisible. We can see the color pattern repeated at $X = 0$ and $X = 4$ along this dimension for the identical mapping of the two instances.

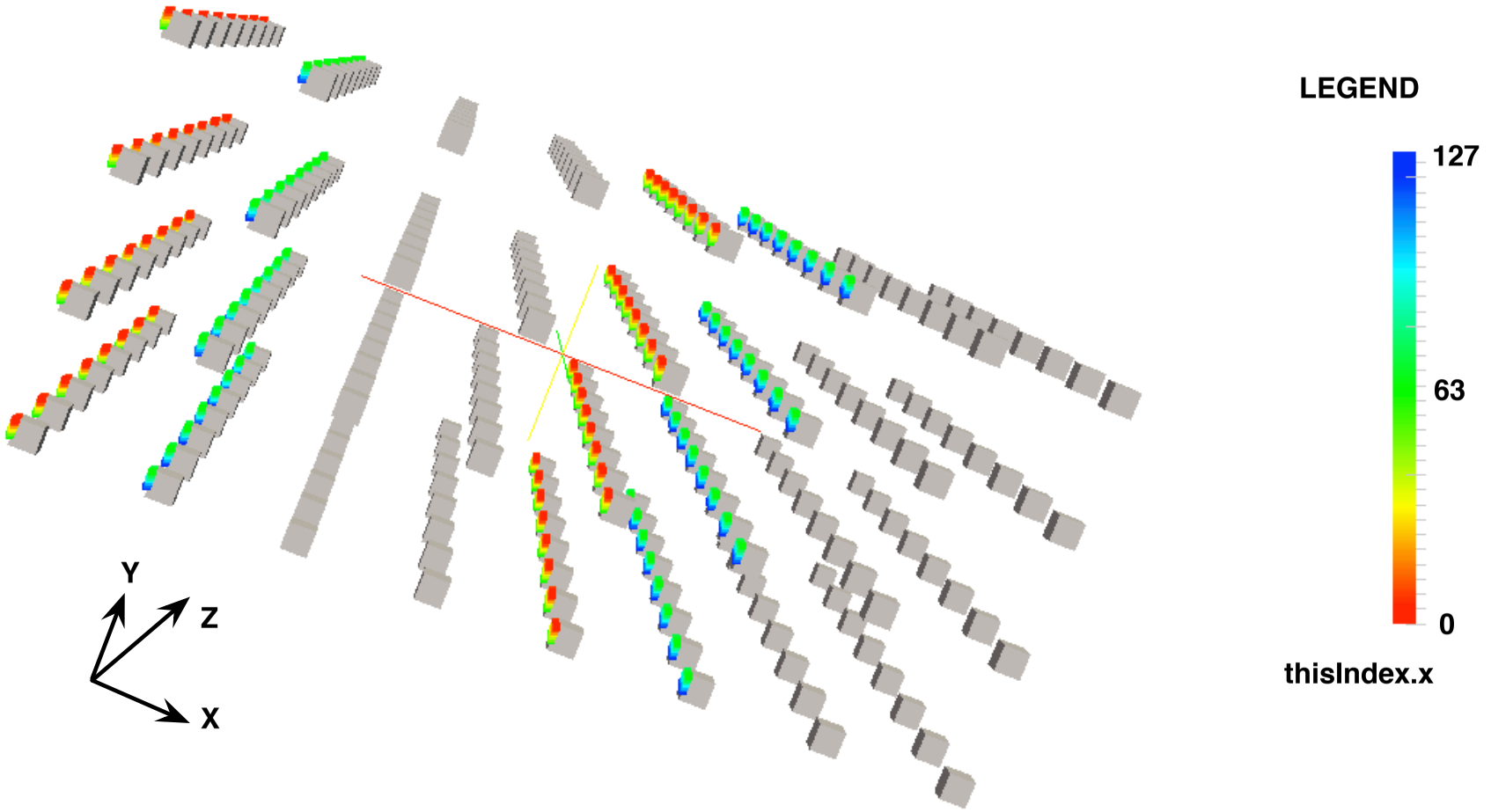


Figure 7.6: Mapping of four OpenAtom instances on a $8 \times 4 \times 8$ torus (System: WATER_32M_70Ry). Two alternate instances are visible while the other two have been made invisible.

There are other possible options which can be experimented with when mapping multiple instances. We can either split along the longest dimension or the shortest dimension. Let us take a concrete example where we have to map two instances of OpenAtom to be mapped on to a torus of dimensions $16 \times 8 \times 32$. If we split along the longest dimension, the diameter of the sub-partition is 28. If we split along the smallest dimension, the diameter of the sub-partition is still 28. Hence we do not hope to see major performance difference between the two schemes. Splitting along more than one dimension may degrade performance due to the creation of smaller meshes instead of tori.

In the next section, we discuss an application which is very different from OpenAtom. The communication pattern is dynamic and the computational loads of each object can be different. Also, this application is latency tolerant.

7.2 NAMD

Molecular Dynamics applications enhance our understanding of biological phenomena through bio-molecular simulations. Large-scale parallelization of MD simulations is challenging due to the small number of atoms and small time scales involved. Load balancing in parallel MD programs is crucial for good performance on large parallel machines. This section discusses load balancing algorithms deployed in a MD code called NAMD [15,22,81]. It focuses on new schemes deployed in the load balancers and provides an analysis of the performance benefits achieved. Specifically, we present the technique of topology aware mapping on 3D mesh and torus architectures, used to improve scalability and performance.

7.2.1 Parallelization of NAMD

Parallelization of NAMD involves a hybrid of spatial and force decomposition. The 3D simulation space is divided into cells called "patches" and the force calculation between every pair of patches is assigned to a different "compute" object. *Patches* are assigned statically to processors during program start-up. On the other hand, *computes*, can be moved around to balance load across processors. If a patch communicates with more than one compute on a processor, a proxy is placed on this processor for the patch. The proxy receives the message from the patch and then forwards it to the computes internally (Figure 7.7). This avoids adding new communication paths when new computes for the same patch are added on a processor.

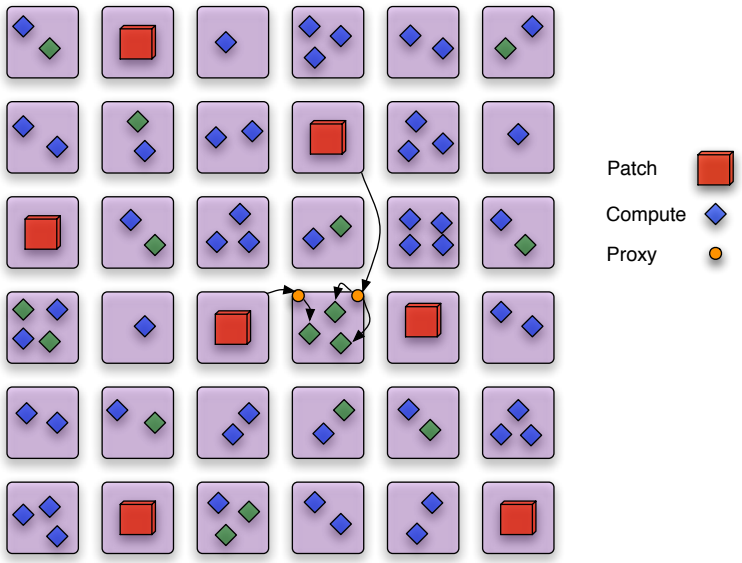


Figure 7.7: Placement of patches, computes and proxies on a 2D mesh of processors

The total computational load on a processor is the sum of individual loads of the computes it houses. The number of proxies on a processor give an indication of its communication load. Load balancing in NAMD is measurement-based. This assumes that load patterns tend to persist over time and even if they change, the change is gradual (referred to as the *principle of persistence*). The load balancing

framework records information about object (compute) loads for some time steps. It also records the communication graph between the patches and proxies. This information is collected on one processor and based on the instrumentation data, a load balancing phase is executed. Decisions are then sent to all processors. The current strategy is centralized and we shall later discuss future work to make it fully distributed and scalable.

It should be noted that communication in NAMD is a special case of a general scenario. In NAMD, every patch multicasts its atom information to many computes. However, each compute (or target of the multicast) receives data from only two patches (the sources). The general case is where each target can receive from more than two sources and the strategies deployed in NAMD can be extended to other cases.

Patches in NAMD are statically mapped in the beginning and computes are moved around by the load balancers to achieve load balance. Two load balancers are used in NAMD. An initial *comprehensive* load balancer invoked in the beginning places the computes evenly on all processors. A *refinement* load balancer is invoked multiple times during a run and it moves a small number of computes to rebalance the load. Both load balancers follow a greedy approach to distribute load evenly among the processors.

7.2.2 Load Balancing Algorithms

The decision to place patches statically and load balance the computes is based on the typical number of patches and computes for a system. For a standard MD benchmark, 92,227-atom ApoLipoprotein-A1 (ApoA1), the number of patches and computes in a typical run on 512 cores is 312 and 22212 respectively. Also, atoms in a patch move slowly and the relative density of atoms per patch does not change much as there is no vacuum inside patches – unlike large density variations we see

in cosmology applications, for example. Hence, we do not need to load balance the patches. Atoms are migrated from one patch to another every 20 time steps.

Static Placement of Patches: The 3D simulation space is divided into patches using a geometric decomposition to have roughly equal number of atoms in each patch. These patches are then assigned to a subset of the processors in a simple round-robin or strided fashion. In a typical highly parallel run, the number of patches is significantly smaller than the number of processors.

Comprehensive Strategy: This algorithm iterates over the list of all computes in decreasing order of their computational loads and finds a "suitable" processor for each one, while minimizing load imbalance. A compute is placed on a processor only if the new load of the processor remains below a threshold value (set to be some factor of the average load on all processors). The algorithm also tries to minimize communication by avoiding the creation of new proxies (additional proxies require new communication paths from a particular patch to the processor on which a new proxy is being placed). Keeping in mind that each compute communicates with two patches, the steps in the search of a "suitable" processor for a compute are:

Step I: Place the compute on an underloaded processor which hosts both the patches or proxies for both of them - this does not add any new communication paths to the graph.

Step II: If Step I fails, place the compute on an underloaded processor which hosts at least one patch or proxy for one of the patches - this requires adding one path for the other patch.

Step III: If both Step I and Step II fail, find the first underloaded available processor from the list of underloaded processors which can accept this compute.

To summarize the strategy, only underloaded processors are considered for placing a compute and among them, processors with available patches or proxies are given preference to minimize communication. This is implemented using a preference table which stores the least underloaded processors for different categories (Figure 7.8). The first three cells in the table correspond to Step I and the last two correspond to Step II. The highest preference is given to a processor with proxies for both patches (cell 1), then to one with one of the patches and a proxy for the other (cell 2) and then to a processor with both patches on it (cell 3). If Step I fails, preference is first given to a processor with a proxy (cell 4) and then to one with the patch (cell 5). We give preference to placing computes on a processor with proxies compared to the patches themselves because it was observed that performance is better if the processors with patches are not heavily loaded.

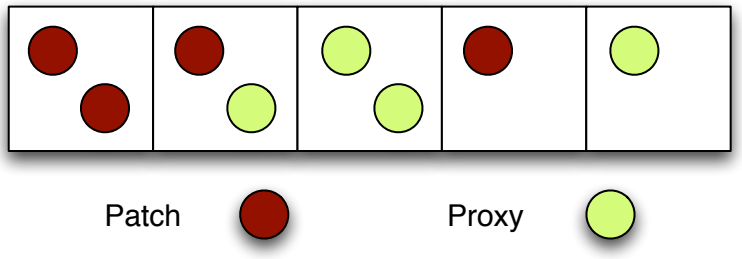


Figure 7.8: Preference table for the placement of a compute

Refinement Strategy: This is algorithmically similar to the comprehensive strategy. The difference is that it does not place all the computes all over again. This algorithm builds a max heap of over-loaded processors and *moves* computes from them to under-loaded processors. Once it has reduced the load on all overloaded processors to below a certain value, it stops. The process of choosing an underloaded processor on which to *move* a compute, is similar to that in the comprehensive strategy. The three steps outlined above for the search of a suitable processor are followed in order in this case also. For a detailed and historical perspective to the

NAMD load balancers, read [82].

7.2.3 Metrics for Evaluation

Optimal load balancing of objects to processors is NP-hard, so in practice, the best one can do is to try different heuristic strategies to minimize load imbalance. A combination of several metrics decides the success of a load balancer and we will discuss them now before we compare different load balancing strategies:

Computational Load: The most important metric which decides the success of a load balancer is the distribution of computational load across all processors. A quantity which can be used to quantify this is the ratio of the maximum to average load across the set of processors. A high *max-to-average* ratio points towards load imbalance.

Communication Volume: As we balance computational load, we should also aim at minimizing inter-processor communication. This can be achieved by using proxies, as described earlier, to avoid duplicate communication paths from a patch to a processor. Additionally, we need to minimize the number of proxies by avoiding the addition of new proxies.

Communication Traffic: Another optimization possible on non-flat topologies is to reduce the total amount of traffic on the network at any given time. This can be done by reducing the number of hops each message has to travel and thus reducing the sharing of links between messages. Number of hops can be reduced by placing communicating objects on nearby processors. This reduces communication contention and hence, the latency of messages. This is the main focus of our work. 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.

7.2.4 Topology Aware Techniques

Recent years have seen the emergence of large parallel machines with a 3D mesh or torus interconnect topology. Performance improvements can be achieved by taking the topology of the machine into account to optimize communication. Co-locating communicating objects on nearby processors reduces contention on the network and message latencies, which improves performance [83, 84]. Let us now see the deployment of topology aware techniques in the static placement of patches and the load balancers for NAMD.

Topology placement of patches: Since patches form a geometric decomposition of the simulation space, they constitute a 3D group of objects which can be mapped nicely onto the 3D torus of machines. An Orthogonal Recursive Bisection (ORB) of the torus is used to obtain partitions equal in number to the patches and then a one-to-one mapping of the patches to the processor partitions is done. This is described in detail in [85]. This idea can be used in other applications with a geometric decomposition such as cosmological and meshing applications.

Topology Aware Load Balancers: Once patches have been statically assigned onto the processor torus, computes which interact with these patches should be placed around them. We will now discuss modifications to the load balancing algorithm that try to achieve this heuristically. The three steps for choosing a suitable processor to place a compute on (for both the comprehensive as well as refinement load balancer) are modified as follows:

Step I: If the compute gets placed on a processor with both the patches, then no heuristic can do better than that because both messages are local to the processor

(and no new communication paths are added). However, if we are searching for a processor with proxies for both patches, we can give topological preference to some processors. Consider Figure 7.9 which shows the entire 3D torus on which the job is running. When placing a compute, it should be placed topologically close to the two processors that house the patches it interacts with. The two patches define a smaller brick within the 3D torus (shown in dark grey in the figure). The sum of distances for any processor within this brick to the two patches is less than that for any processor outside the brick. Hence, if we find two processors with proxies for both patches, we give preference to the processor which is within this inner brick defined by the patches.

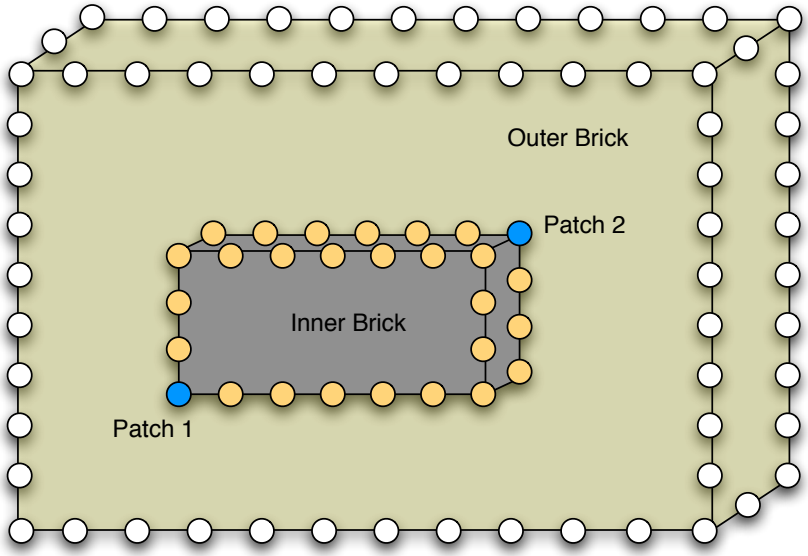


Figure 7.9: Topological placement of a compute on a 3D torus/mesh of processors

Step II: Likewise, in this case too, we give preference to a processor with one proxy or patch which is within the brick defined by the two patches that interact with the compute.

Step III: If Step I and II fail, we are supposed to look for any underloaded processor to place the compute on. Under the modified scheme, we first try to find

an underloaded processor within the brick and if there is no suitable processor, we spiral around the brick to find the first underloaded one.

To implement these new topology aware schemes in the existing load balancers, we build two preference tables (similar to Figure 7.8) instead of one. The first preference table contains processors which are topologically close to the patches in consideration (within the brick) and the second one contains the remaining processors (outside the brick). We look for underloaded processors in the two tables with preference in order to the following: number of proxies, hops from the compute and then the load on the processor.

7.2.5 Performance Improvements

Performance runs were done to validate the impact of the topology aware schemes on the execution time of NAMD. Two supercomputers were used for this purpose: IBM Blue Gene/P (Intrepid) at ANL and Cray XT3 (BigBen) at PSC. The default job scheduler for XT3 does not guarantee a contiguous partition allocation and hence those runs were done with a special reservation on the whole machine.

Figure 7.10 shows the *hop-bytes* for all messages per iteration when running NAMD on Blue Gene/P on different sized partitions. A standard benchmark used in the MD community was used for the runs: $92,227$-atom ApoLipoprotein-A1 (ApoA1). All runs in this dissertation were done with the PME computation turned off to isolate the load balancing issues of interest. As we would expect, hop-bytes consistently increase as we go from a smaller partition to a larger one. The three strategies compared are: topology oblivious mapping of patches and computes (Topology Oblivious), topology aware static placement of patches (TopoPlace Patches) and topology aware placement for both patches and load balancing for computes (TopoAware LDBs).

Topology aware schemes for the placement of patches and the load balancer help

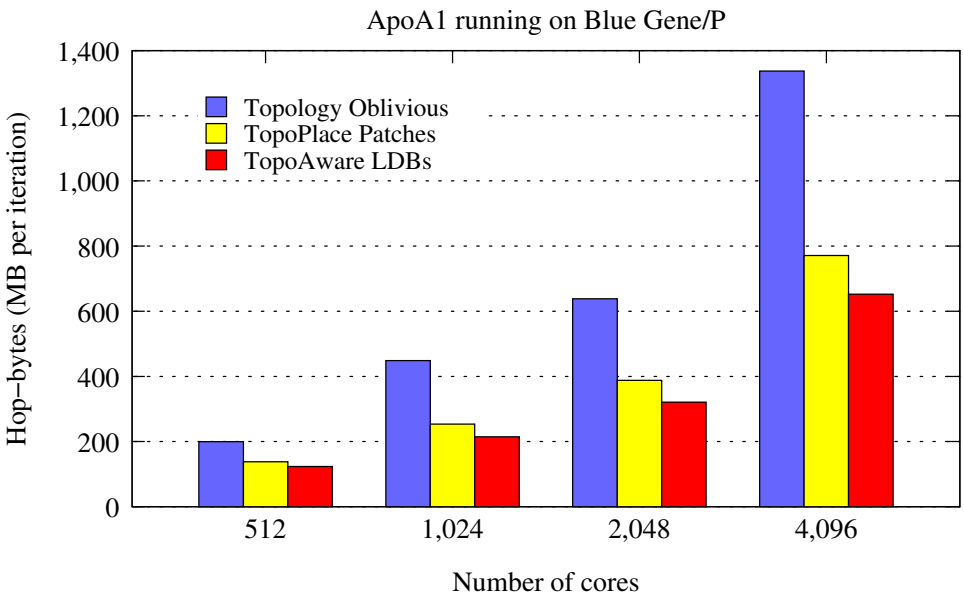


Figure 7.10: Hop-bytes for different schemes on IBM Blue Gene/P

in reducing the hop-bytes for all processor counts. Also, the decrease in hop-bytes becomes more significant as we go to larger-sized partitions. This is due to the fact that the average distance traveled by each message increases as we increase the partition size in the case of default mapping, but it gets controlled when we do a topology aware mapping. Since the actual performance of the load balancers depends on several metrics, the question remains – does the reduction in hop-bytes lead to an actual improvement in performance? As it turns out, we also see a reduction in the number of proxies and in the *max-to-average* ratio for topology aware load balancers, which is reflected in the overall performance of NAMD on Blue Gene/P (Table 7.5). The topology oblivious scheme stops scaling around $4,096$ cores and hence we did not obtain numbers for it beyond that. We see an improvement of 28% at $16,384$ cores with the use of topology aware load balancers.

Similar tests were done on Cray XT3 to assess if a faster interconnect can hide all message latencies and make topology-mapping unnecessary. Figure 7.11 shows the hop-bytes for all messages per iteration when running NAMD on Cray XT3 on different sized partitions. We could only run on up to 1024 nodes (1 core per

Cores	*Topology Oblivious*	*TopoPlace Patches*	*TopoAware LDBs*
512	13.93	13.85	13.57
1024	7.96	7.87	7.79
2048	5.40	4.57	4.47
4096	5.31	3.07	2.88
8192	-	2.33	2.03
16384	-	1.74	1.25

Table 7.5: Performance of NAMD (ms/step) on IBM Blue Gene/P

node) on XT3 and as a result we do not see a huge benefit on the lower processor counts. However, if we compare the 512 processor runs on XT3 with 2048 processor (512 node) runs on Blue Gene/P, we see a similar reduction in hop-bytes. It is also reflected in a slight improvement in performance at this point (Table 7.6).

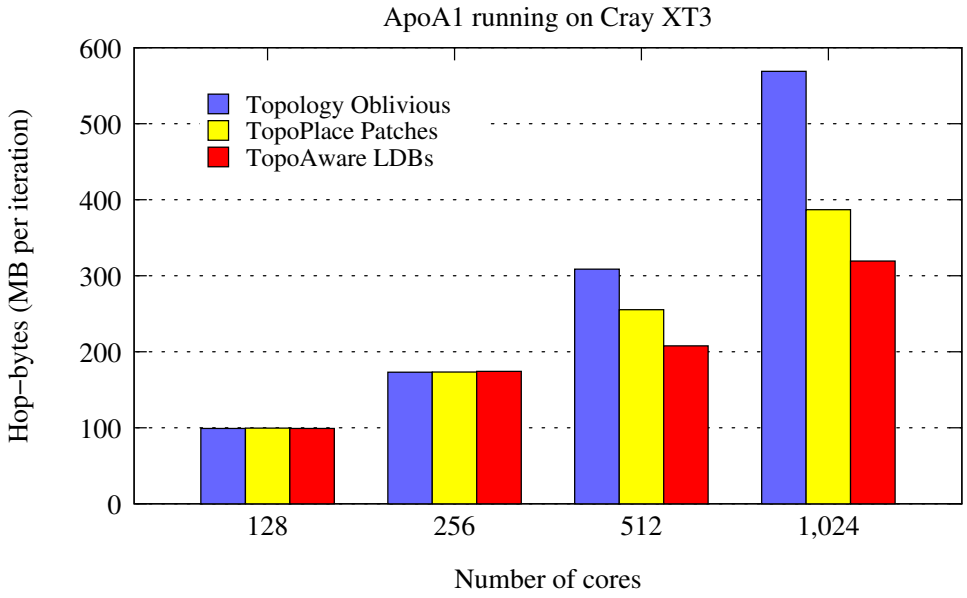


Figure 7.11: Hop-bytes for different schemes on Cray XT3

Improvement in performance indicates that computational load is balanced. Reduction in hop-bytes indicates a reduction in the communication traffic on the network. A reduction in communication volume can be inferred from the number of proxies during a simulation. Table 7.7 presents the number of proxies being used in a particular run with different topology schemes. It is clear from the table that

Cores	*Topology Oblivious*	*TopoPlace Patches*	*TopoAware LDBs*
128	17.43	17.50	17.47
256	8.83	8.88	8.78
512	5.14	5.34	5.10
1024	3.08	3.15	3.01

Table 7.6: Performance of NAMD (ms/step) on Cray XT3

Cores	*Topology Oblivious*	*TopoPlace Patches*	*TopoAware LDBs*
512	4907	4922	4630
1024	15241	15100	14092
2048	22362	22280	20740
4096	38421	28981	29572

Table 7.7: Reduction in total number of proxies on Blue Gene/P

topology aware schemes reduce the total number of proxies also apart from reducing hop-bytes.

8 Automatic Mapping Framework

The non-trivial task of mapping parallel applications (as presented in the previous chapter), motivated us to think about automating the mapping process. Several application groups have realized the importance of topology aware mapping and used it to improve the performance of individual codes. A general mapping framework which takes the communication graph of an application as input and outputs efficient mapping solutions would relieve the application developers of the mapping burden. A library with several mapping heuristics for a variety of communication graphs would be a great asset to the parallel computing community.

This chapter introduces the automatic mapping framework we have developed for mapping of a variety of communication graphs associated with different categories of parallel applications to parallel machines. All parallel applications can be divided into two categories depending on their communication graph – regular and irregular. Regular graphs refer to those where the number of edges from all the nodes is the same and there is a certain pattern to the communication. Examples of regular communication are 2D and 3D stencil-like communication and structured mesh computations. Graphs with varying number of edges from different nodes and all others which do not fall under the regular category can be labeled as irregular. Examples of irregular communication are unstructured mesh computations.

The process of automated the mapping of applications can be divided in to two steps:

1. Obtain the communication graph for an application and identify specific communication patterns in the communication graph.

2. Apply heuristics in different cases depending on the communication patterns to obtain mapping solutions.

The communication graph for an application gives information about the number of bytes exchanged between tasks or processes in the program. For example, in case of MPI, the nodes of this graph are the MPI ranks or processes in the program and edges exist between two nodes if the corresponding MPI ranks communicate through messages. A test run of the application is performed to obtain a $n \times n$ matrix of communication bytes exchanged between different pairs where n is the total number of MPI ranks. In case of Charm++, the nodes of the graph are virtual processors or VPs.

The communication graph is obtained by profiling libraries such as those in the IBM High Performance Computing Toolkit (HPCT) [86] and information from one run can be used to develop mapping solutions for subsequent similar runs. This approach assumes that the communication graph for an application running for a certain set of input parameters does not change from run to run. As of now, the situation where the graph changes with time, within one run, can not be handled either since migrating MPI tasks at runtime is not possible. However, some programming models such as Charm++ provide the capability of profiling applications as they are running. In such cases, the information can be used for dynamic mapping (and even load balancing if the need arises).

Figure 8.1 represents a schematic of the automatic mapping framework. There are two inputs to the framework:

1. The application communication graph which is used by the pattern matching algorithms (referred to as the object communication graph in the rest of the dissertation).

2. The processor topology graph which is used by the mapping algorithms.

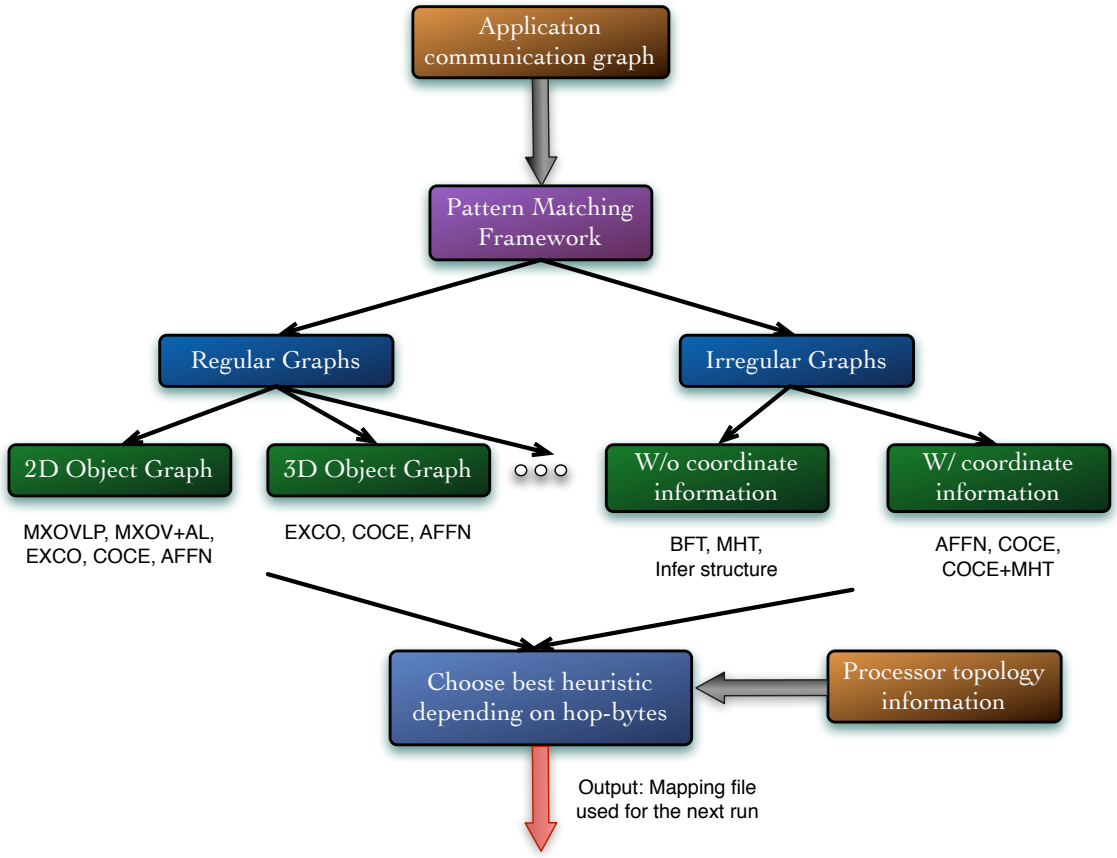


Figure 8.1: Schematic of the automatic mapping framework

The framework first searches for regular communication patterns. Depending on the communication patterns identified by pattern matching, the framework chooses the best heuristic from the suite for a given object and processor graph pair, based on the *hop-bytes* metric. If there is no regular pattern, we assume the graph to be general and use heuristics for irregular graphs. The framework outputs a mapping solution in the form a file to be used for a subsequent run (by providing it to the job scheduler). In case of CHARM++ applications, the framework can be used as a library for mapping at runtime.

The next section discusses the first part of the automatic mapping framework: identifying regular patterns in communication graphs.

8.1 Communication Graph: Identifying Patterns

Automatic topology aware mapping, as we shall see in the next few sections, uses heuristics for fast scalable runtime solutions. Heuristics can yield more efficient solutions if we can derive concrete information about the communication graph of the application and exploit it. For this, we need to look for identifiable communication patterns, if any, in the object graph. Many parallel applications have relatively simple and easily identifiable 2D, 3D or 4D communication patterns. If we can identify such patterns, then we can apply better suited heuristic techniques for such scenarios. We use relatively simple techniques for pattern identification. We believe that these can be extended to detect more complex patterns based on similar work in literature [87,88]. Some performance analysis tools also provide communication pattern identification and visualization for identifying performance problems in parallel applications [89].

Here, we explain the algorithm for identifying if the communication in an application has a near-neighbor stencil-like pattern with four neighbors in 2D. Algorithms for doing the same in 3D and 4D are similar. We first begin by ensuring that the number of communicating neighbors for all nodes in the graph is the same and the number is 5 or less. For a 2D communication pattern, a given node would typically have four communicating neighbors and may have some communication (through global operations) with the node with ID 0. Broadcasts from and reductions to node 0 cannot be optimized by remapping of nodes and hence we ignore that. We also ignore any communication edges whose weight is less than 20% of the average across all edges in the graph. This is a heuristically decided value and can be changed.

For a 2D communication pattern, if there is no wraparound, nodes on the boundaries may have fewer neighbors. Filtering these aberrations, we choose a random node and find its "distance" (difference between the node IDs) from its four neigh-

bors. The distance from two of its neighbors (left and right) would be 1 and from its top and bottom neighbors would be one of the dimensions of the 2D grid. This assumes that node IDs are ordered in a row or column major order. Then, for all other nodes, we ensure that the distances from their respective neighbors are either 1 or the value of distance obtained for the previously chosen random node. If this holds true for all other nodes in the graph, then the communication is indeed a uniform 2D near-neighbor pattern.

Algorithm 8.1 Pseudo-code for identifying regular communication graphs

```
Input: CM_{n,n} (communication matrix)
Output: isRegular (boolean, true if communication is regular)
        dims[ ] (dimensions of the regular communication graph)
for i = 1 to n do
    find the maximum number of neighbors for any node in CM_{i,n}
end for
if max neighbors ≤ 5 then
    // this might be a case of regular 2D communication
    select an arbitrary node start_pe find its distance from its neighbors
    dist = difference between node IDs of start_pe and its top or bottom neighbor
    for i := 1 to n do
        if distance of all nodes from their neighbors == 1 or dist then
            isRegular = true
            dim[0] = dist
            dim[1] = n/dist
        end if
    end for
end if
```

Algorithm 8.1 shows the pseudo-code for identifying one possible 2D communication pattern. Currently, this algorithm can only identify a 5-point stencil pattern (left diagram in Figure 8.2). However, the algorithm can be extended trivially so that it can identify other regular patterns such as a 9-point stencil or a communication with all 8 neighbors around a node in 2D (other two diagrams in Figure 8.2). The algorithms for identifying 3D and 4D near-neighbor patterns are similar. Once the information about communicating neighbors has been extracted and identified, mapping algorithms can use it to map communicating neighbors on nearby physical

processors.

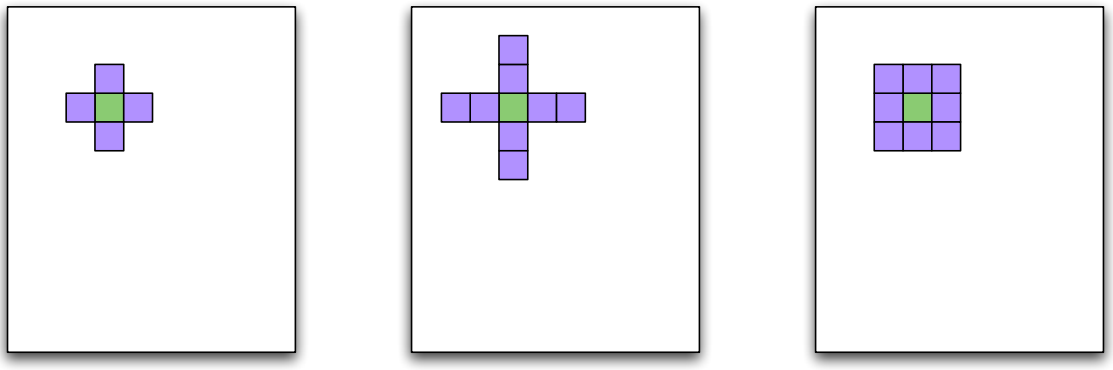


Figure 8.2: Different communication patterns in two-dimensional object graphs — a 5-point stencil, a 9-point stencil and communication with all 8 neighbors around a node

The pattern matching algorithms were tested with three production applications which are known to have regular communication: MILC [90, 91], POP [92–95] and WRF [24, 25, 96]. The communication patterns and the size of each dimension were correctly identified as shown in Table 8.1.

Application	*No. of cores*	*Dimensionality*	*Size of dimensions*
MILC	256	4-dimensional	$4 \times 4 \times 4 \times 4$
POP	256	2-dimensional	8×32
POP	512	2-dimensional	32×16
WRF	256	2-dimensional	16×16
WRF	512	2-dimensional	32×32

Table 8.1: Pattern identification of communication in MILC, POP and WRF

Figure 8.3 shows visualizations of the 2D communication graphs as output by the pattern matching library for POP and WRF running on 256 processors. The radial and tangential directions in the graph show the two dimensions of the object graph. POP has an object graph of dimensions 8×32 and WRF has an object graph of dimensions 16×16.

We have developed two sets of algorithms for mapping of applications with regular and irregular communication graphs. The next two chapters discuss these

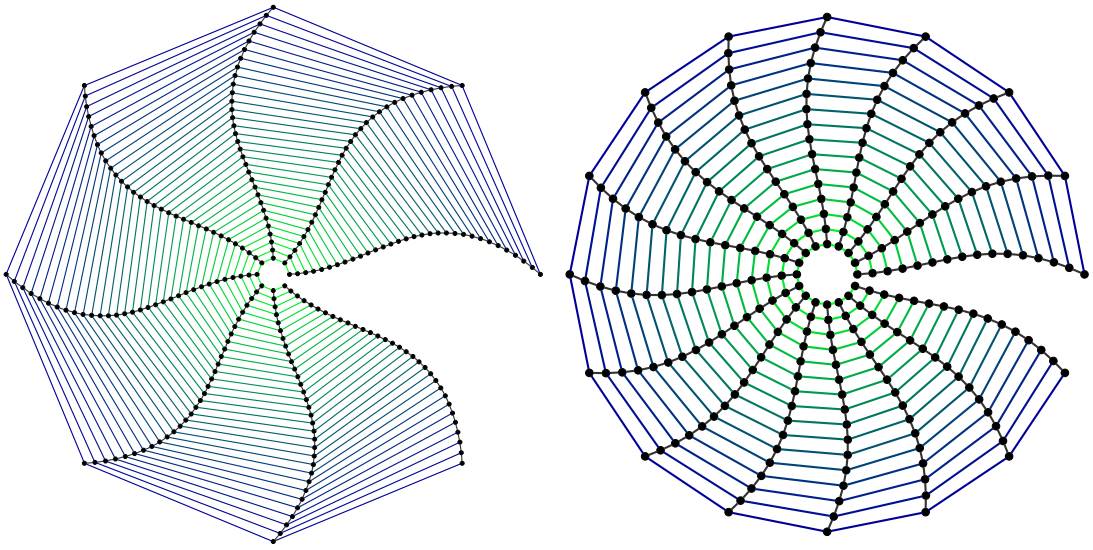


Figure 8.3: Communication graph for POP (left) and WRF (right) on 256 processors

algorithms which form the core of the mapping framework.

9 Mapping Regular Communication Graphs

This chapter focuses on the mapping of regular communication graphs to 2D and 3D mesh topologies. Several applications which perform stencil-like computations have a regular communication pattern. Examples include MILC [90, 91], POP [92–95] and WRF [24, 25, 96]. These applications can use the algorithms described in this chapter for mapping to torus topologies.

We first discuss heuristics to map 2D object graphs (grids) to 2D processor graphs (meshes), then discuss techniques for mapping them to 3D processor meshes. Finally we discuss mapping 3D object grids to 3D processor meshes and discuss performance results for some applications.

9.1 Algorithms for Mapping 2D Grids to 2D Meshes

Let us say that processes in an application have a 2D stencil-like communication pattern where each task communicates with four neighbors, two in each direction. So the communication graph for this application is a planar graph which resembles a 2D grid. We want to map this 2D grid to a 2D processor mesh. For heuristics in this chapter, we assume that all edges in the communication graph have the same weight. Further, we are targeting MPI applications, so, the number of ranks (nodes in the communication graph) is the same as the number of processors (in the mesh). Therefore, we can assume that the number of nodes in the two graphs is the same. Of course, if the object grid has the same dimensions as the processor mesh, the

best mapping is trivial. Heuristic strategies are needed when the aspect ratios are different. We describe five heuristics to map a 2D object grid to a 2D processor mesh. All of these heuristics are designed to optimize different cases and as we shall see in the results section, they perform best for grids of different aspect ratios.

Heuristic 1 - Maximum Overlap: This heuristic attempts to find the largest possible area of the object grid which overlaps with the processor mesh and maps it one-to-one. For the remaining area of the object grid and the processor mesh, we then make a recursive call to the algorithm. The intuition behind this heuristic is that we get the best hop-bytes for a large portion of the grid, although for a few nodes at the boundaries of the recursive calls, we might have longer hops. However, the hope is that the average hop-bytes for the entire object grid will be low and a few distant messages will not affect performance.

Figure 9.1 illustrates this mapping technique,i where an object grid of dimensions 9×8 is to be mapped to a processor mesh of dimensions 6×12. We first map the largest possible sub-grid with dimensions 6×8 to the processor mesh. Once this is done, a recursive call is made for the object grid of size 3×8 to be mapped onto a processor mesh of size 6×4.

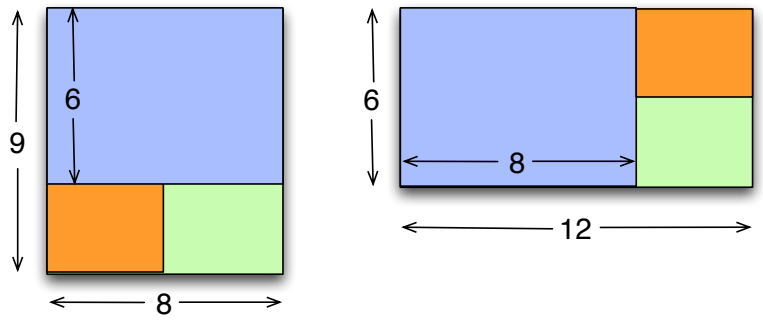


Figure 9.1: Finding regions with maximal overlap in the Maximum Overlap Heuristic (`MXOVLP`)

Algorithm 9.1 presents the pseudo-code for this heuristic (referred to as `MXOVLP` in figures and tables). O_x and O_y refer to the x and y dimensions of the object grid

and P_x and P_y refer to the x and y dimensions of the processor mesh.

Algorithm 9.1 Maximum Overlap Heuristic (`MXOVLP`) for 2D to 2D mapping

```
procedure MXOVLP(O_x, O_y, P_x, P_y)
    if O_x == P_x then
        do a one-to-one mapping and return
    end if
    if O_x > P_x then
        map the area P_x × O_y
        MXOVLP(O_x − P_x, O_y, P_x, P_y − O_y)
        copy the mapping into the main array and return
    else
        map the area O_x × P_y
        MXOVLP(O_x, O_y − P_y, P_x − O_x, P_y)
        copy the mapping into the main array and return
    end if
end procedure
```

Heuristic 2 - Maximum Overlap with Alignment: This is similar to Heuristic 1 but it tries to align the longer dimension of the object grid with that of the processor mesh (referred to as `MXOV+AL` in figures and tables). This realignment is done at each recursive call and yields a better mapping than `MXOVLP` in most cases. Heuristics 1 and 2 lead to dilation at each recursive call at the boundaries where the object grid is split during recursion. However, as per our claim in Chapter 5, as long as the average hop-bytes is low, we should obtain a good mapping. All the following heuristics (including Heuristic 1 above) also do an initial alignment of the longer dimensions of the object grid and the processor mesh.

There are some optimization possibilities for Heuristics 1 and 2 which will be explored in future work – After the mapping for recursively smaller sub-graphs is complete, at the end of each recursive call, it is possible to rotate the mapping for the sub-graph by 180 degrees or flip it. There are several possibilities at each recursive call leading to a combinatorial explosion of arrangements. Hence we will not discuss this post-rotation here.

Heuristic 3 - Expand from Corner: In this algorithm, we start at one corner of the object grid and order all other objects by their increasing Manhattan distance from the chosen corner. We also order the processors in a similar fashion starting from one corner. Then the objects are placed in order starting from the chosen corner of the processor grid (pseudo-code for `EXCO` in Algorithm 9.2). The intuition is that objects which communicate with one another will be close in the new ordering based on the Manhattan distance and hence, will get mapped nearby. Figure 9.2 shows how we start from the upper left corner of the object grid and map objects successively starting from the corresponding corner of the processor mesh.

Algorithm 9.2 Expand from Corner (`EXCO`) Heuristic for 2D to 2D mapping

procedure EXCO(O_x, O_y, P_x, P_y)
  $no_x = no_y = np_x = np_y = 0$
  for $i := 1$ to $O_x \times O_y$ **do**
    $< co_x, co_y >=< no_x, no_y >$
    $< cp_x, cp_y >=< np_x, np_y >$
    $\text{Map}[co_x][co_y] = cp_x \times P_y + cp_y$
    $< no_x, no_y > =$ `findNearest2D`(co_x, co_y, O_x, O_y)
    $< np_x, np_y > =$ `findNearest2D`(cp_x, cp_y, P_x, P_y)
  end for
end procedure

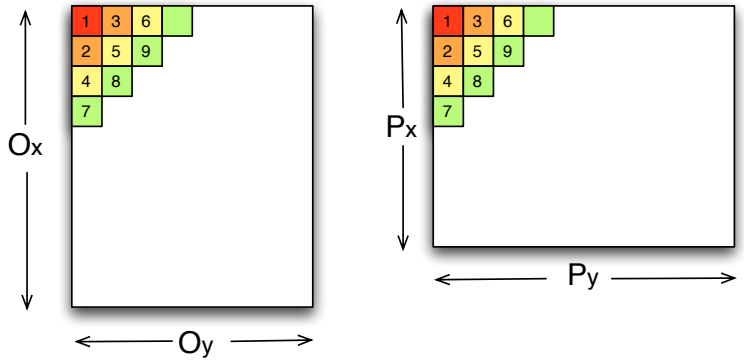


Figure 9.2: Expand from Corner (`EXCO`) Heuristic

Heuristic 4 - Corners to Center: This is similar to the Heuristic 3 but in this case, we start simultaneously from all four corners of the 2D object grid and move

towards the center. The objects are again picked based on their Manhattan distance from the corner closest to them (this heuristic is referred to as `COCE` in figures and tables). This modification to Heuristic 3 achieves better proximity for a larger number of objects since we start simultaneously from four directions. However, for certain aspect ratios, as we move closer to the center, objects may be placed farther from their communicating neighbors leading to larger hop-bytes.

Heuristic 5 - Affine Mapping: The idea is to stretch/shrink the object grid in both dimensions and align it to the processor mesh. It is expected that such a mapping will preserve the relative orientations of the objects, thereby minimizing the dilation. A destination processor is calculated for each object based on its position, (x, y) in the communication graph:

$$(x, y) \rightarrow (\lfloor P_x \times \frac{x}{O_x} \rfloor, \lfloor P_y \times \frac{y}{O_y} \rfloor) \tag{9.1}$$

Algorithm 9.3 Affine Mapping (`AFFN`) Heuristic for 2D to 2D mapping

```
procedure AFFN(O_x, O_y, P_x, P_y)
    for i := 1 to O_x do
        for j := 1 to O_y do
            af_x = ⌊P_x × i/O_x⌋
            af_y = ⌊P_y × j/O_y⌋
            < free_x, free_y > = findNearest2D(af_x, af_y, P_x, P_y)
            Map[i][j] = free_x × P_y + free_y
        end for
    end for
end procedure
```

Since the coordinates are constrained to be integers, it is possible that two objects may be mapped to the same processor. To resolve this, we use the function `findNearest2D` which returns an unused processor closest to (af_x, af_y). It should be noted that this mapping is not strictly affine since we use `findNearest` to resolve conflicts for the same processor (see Algorithm 9.3). This mapping is referred to as

AFFN in figures and tables.

A loose theoretical lower bound can be calculated for the affine algorithm based on Equation 9.1 as show below:

$$\begin{aligned}\text{Hops per byte} &= \frac{(\text{Map}[x][y] - \text{Map}[x+1][y]) + (\text{Map}[x][y] - \text{Map}[x-1][y])}{2} \\ &+ \frac{(\text{Map}[x][y] - \text{Map}[x][y+1]) + (\text{Map}[x][y] - \text{Map}[x][y-1])}{2} \\ &= \frac{1}{2} \times (\frac{P_x}{O_x} + \frac{P_x}{O_x} + \frac{P_y}{O_y} + \frac{P_y}{O_y}) \\ &= \frac{P_x}{O_x} + \frac{P_y}{O_y}\end{aligned}$$

The calculation above considers (x, y) as a point in continuous space and hence we ignore the ceiling operation applied in Equation 9.1. Based on this formula, if we are mapping an object grid of dimensions 9×8 to a processor mesh of dimensions 6×12, the lower bound is 2.167.

The next two algorithms are borrowed from literature and used for visualization and comparison with the heuristics presented in this dissertation.

Heuristic 6 - Step Embedding: This algorithm is an implementation of the step embedding technique (STEP) presented in [53]. Techniques in [53] were written to optimize chip layout and hence they try to minimize the length of the longest wire. The paper presents ways to "square up" an arbitrary rectangular grid. However, unlike our mapping algorithms, for step embedding, the number of nodes in the processor mesh can be greater than that in the object grid. We borrow the idea of visualizing the mappings for the object grids from this paper.

Heuristic 7 - Pairwise Exchanges: Several research papers in the past have used the technique of pairwise exchanges by itself or with an intelligent initial mapping [26, 27]. In this technique (PAIRS), we start with a random or default mapping,

choose two objects randomly and swap the processors they are placed on. We retain this swap if a chosen metric improves otherwise we discard it. This is continued until the improvement in the chosen metric falls below a certain threshold. The pairwise exchange algorithm with probabilistic jumps, presented in [26] has a $\mathcal{O}(n^3)$ time complexity. This algorithm is too expensive to be practical. However, in absence of the knowledge of the true optimal mapping, we use this algorithm to produce an approximation of the optimal mapping, to which other (faster) heuristic strategies can be compared. We use a simpler implementation for our purposes to obtain values for the hop-byte metric which are close to the optimal solution. Figure 9.3 (top) shows the time taken by the `PAIRS` algorithm to obtain a "good" mapping solution for a regular graph of $4,096$ nodes.

9.1.1 Time Complexity

Among the five heuristics presented in this chapter, `MXOVLP` and `MXOV+AL` visit each node in the object grid only once and decide on its mapping. Hence, they have a time complexity of $\mathcal{O}(n)$ where n is the number of objects to be mapped. However, `EXCO`, `COCE` and `AFFN` algorithms use the `findNearest2D` function, which in the worst case can take $\mathcal{O}(n)$ time. Hence the worst case running time of these three algorithms is $\mathcal{O}(n^2)$. A detailed discussion on the running time of `findNearest2D` can be found in Chapter 10.

In the era of petascale machines with hundreds of thousands of cores, it is crucial to use linear or linearithmic (that is $\mathcal{O}(n\ logn)$) running time algorithms for mapping and all presented heuristics adhere to that on the average. Figure 9.3 (bottom) presents the actual running times for the mapping algorithms when run sequentially on a dual-core 2.4 GHz processor. We can see that three algorithms (`MXOVLP`, `MXOV+AL` and `EXCO`) take less than a millisecond for obtaining the mapping of a regular graph with $65,536$ nodes. Even the `AFFN` and `COCE` algorithms take

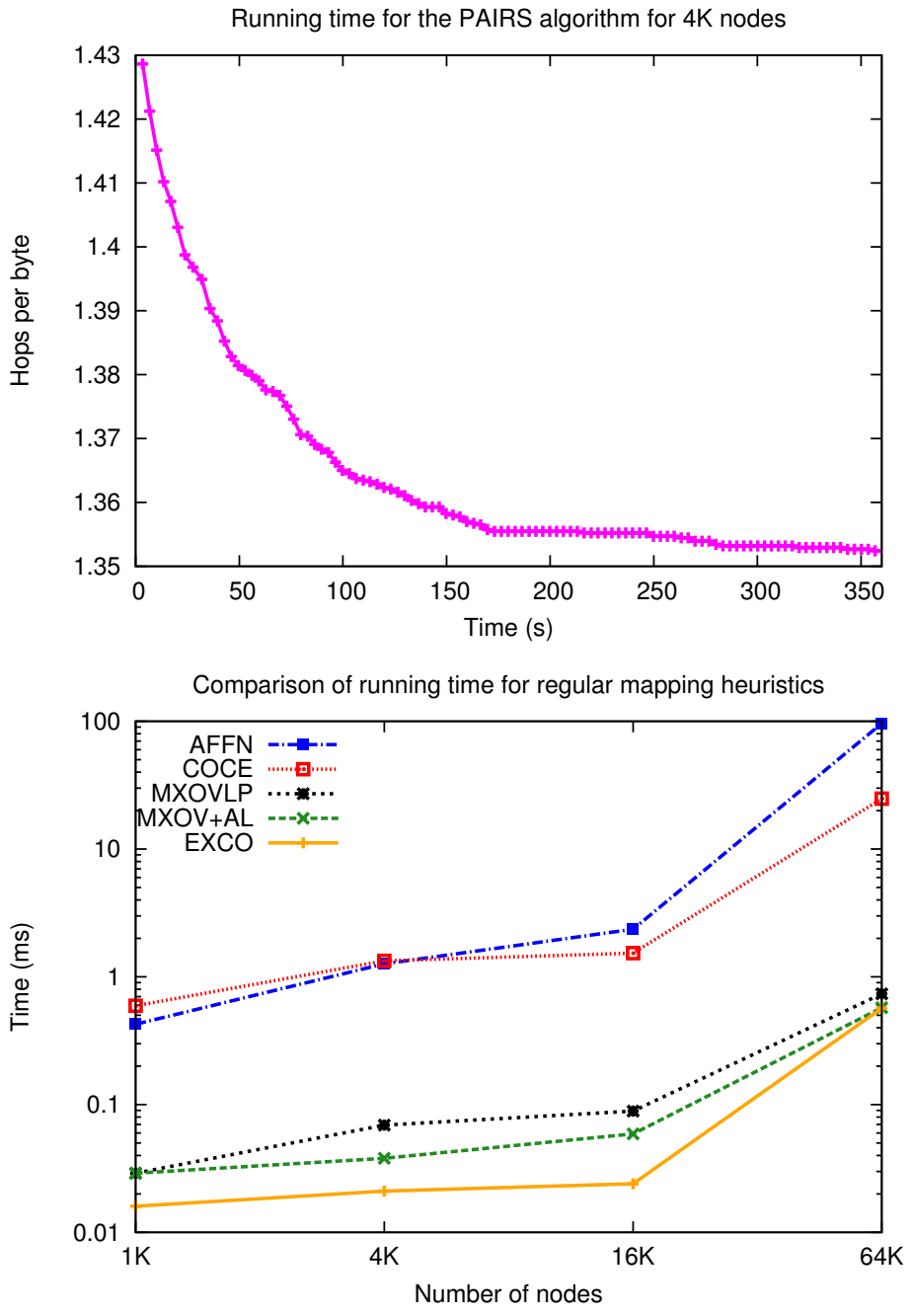


Figure 9.3: Running time for the `PAIRS` algorithm (top) and for other mapping heuristics for regular mapping (bottom)

tens of milliseconds. For parallel applications which run for days, this time spent on mapping should be negligible compared to the total execution time.

9.1.2 Quality of Mapping Solutions: Hop-bytes

This section evaluates the mapping algorithms presented above. We use the *hop-bytes* metric to compare across them. For better intuition, the graphs in this section present the average hops per byte for each algorithm. The ideal value for average hops per byte is 1 and so the closer to 1 we get, the better the mapping algorithm is. In an effort to find the hops per byte for close to optimal solutions, we also implemented the $\mathcal{O}(n^3)$ algorithm of pairwise exchanges (`PAIRS`) used in literature [26,27].

Visualization of the mapping of an object grid to a processor mesh helps understand mapping algorithms better and also helps in fixing potential problems with them. Figure 9.4 shows a step-by-step process of mapping individual rows of the object grid on to the processor mesh. We used the `STEP` algorithm for this figure which maps rows one by one. Each dot represents a node in the graph and the edges are communication arcs. The golden edges represent the horizontal communication and the green edges represent the vertical communication. In the top left corner, we have the object grid of dimensions 9×8. The 8 subsequent graphs with dimensions 12×6 show the mapping individual rows of the object grid on to the processor mesh. We can see how the *green* vertical edges are dilated in the processor mesh. Similar diagrams are used to show the mapping of individual rows for other heuristics even though they do not map graphs by rows. It helps one compare across the various mapping solutions in a graphical manner.

Figures 9.5 and 9.6 present mappings of some representative object grids to processor meshes using the different heuristics. The six 2D grids in each figure illustrate mappings of the object grid onto the processor mesh based on the six heuristic algorithms: `MXOVLP, MXOV+AL, EXCO, COCE, AFFN` and `STEP`. For Figure 9.5, We can

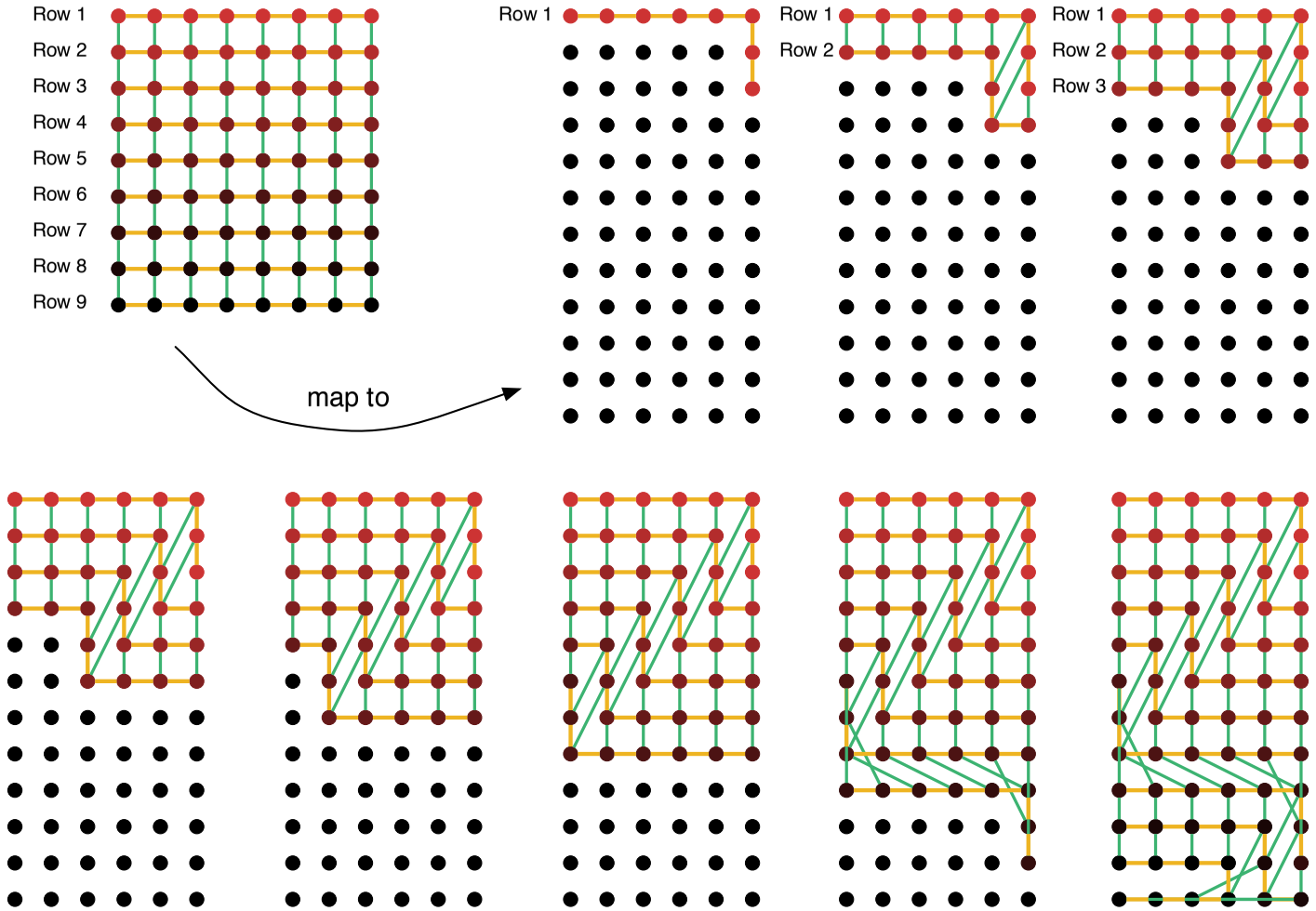


Figure 9.4: Mapping of a 9×8 object grid to a 12×6 processor mesh using the `STEP` algorithm

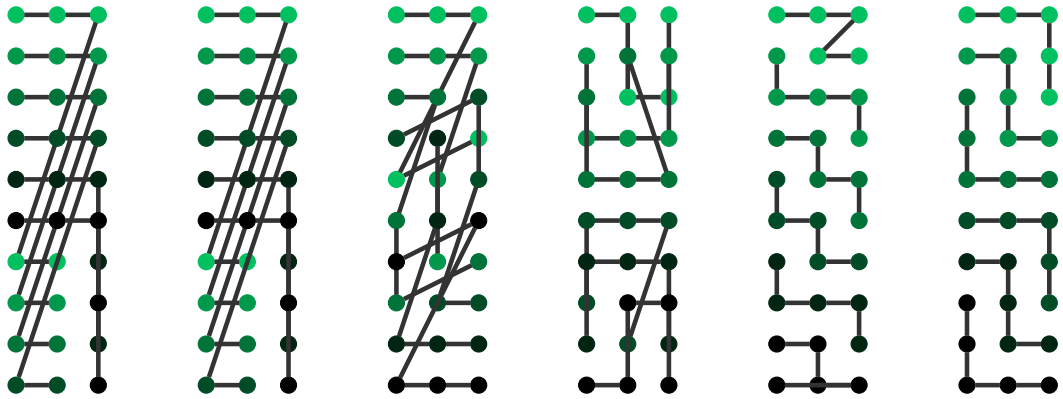


Figure 9.5: Mapping of a 6×5 grid to a 10×3 mesh using `MXOVLP`, `MXOV+AL`, `EXCO`, `COCE`, `AFFN` and `STEP` respectively

see that the first three heuristics stretch some edges significantly while the rest try to minimize both hop-bytes and maximum dilation. The mapping of the 9×8 object grid can be found in Appendix B.

Figure 9.7 compares the total theoretical hops for the different algorithms assuming that communication is 2D near-neighbor and regular. The representative object grids and processor meshes were chosen so as to cover a wide range of aspect ratios. The maximum overlap with alignment heuristic (`MXOV+AL`) gives the best solution in most cases. The `AFFN` heuristic which does a affine inspired mapping also performs quite well. In the case of mapping of a 100×40 grid to a 125×32 mesh, `AFFN` does considerably better than the other algorithms.

9.2 Algorithms for Mapping 2D Grids to 3D Meshes

Some of the largest and fastest supercomputers in the Top500 list today have a 3D torus or mesh interconnect. So, in order to use our mapping algorithms on these real machines, we need to develop algorithms to map 2D communication graphs

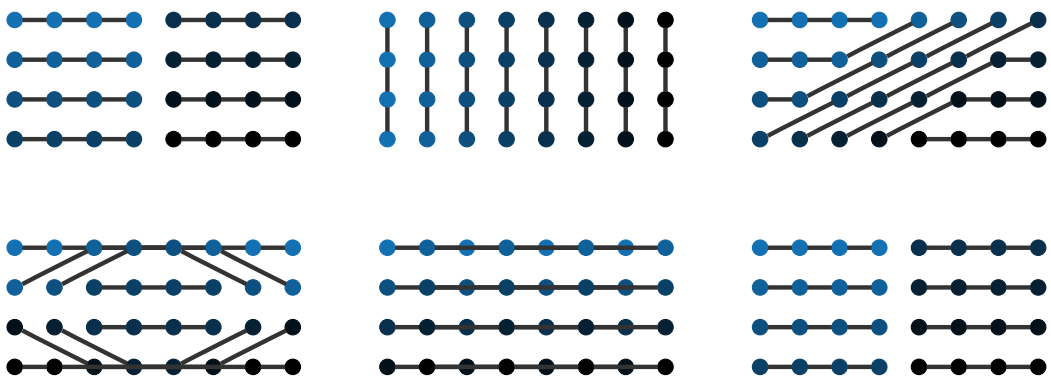


Figure 9.6: Mapping of a 8×4 grid to a 4×8 mesh using `MXOVLP, MXOV+AL, EXCO, COCE, AFFN` and `STEP` respectively

to 3D processor topologies. We now present algorithms for mapping 2D grids to 3D processor meshes. Some algorithms presented in this section use the 2D to 2D mapping algorithms developed in the previous section.

Heuristic 1: Stacking: The general idea is to use the algorithms developed in the previous section for mapping 2D object grids to 2D processor meshes. We find the longer dimension of the 2D object grid and split the object grid along it, into several smaller grids (subgrids). The number of subgrids equals the length of the smallest dimension of the 3D processor mesh. We then take the first subgrid and map it onto a plane perpendicular to the smallest dimension of the 3D processor mesh. The mapping framework chooses the best heuristic to map the 2D subgrid to the 2D processor mesh. A simple translation is used to map the remaining subgrids to other planes of the processor mesh. For example if we wish to map a 32×8 object grid onto a $8 \times 8 \times 4$ processor mesh, we split the longer dimension (32) into 4 pieces (the smallest dimension of the processor mesh) and then map a 8×8 object subgrid to a 8×8 processor mesh.

Heuristic 2: Folding: If the processor topology is a 3D mesh, then in the previous

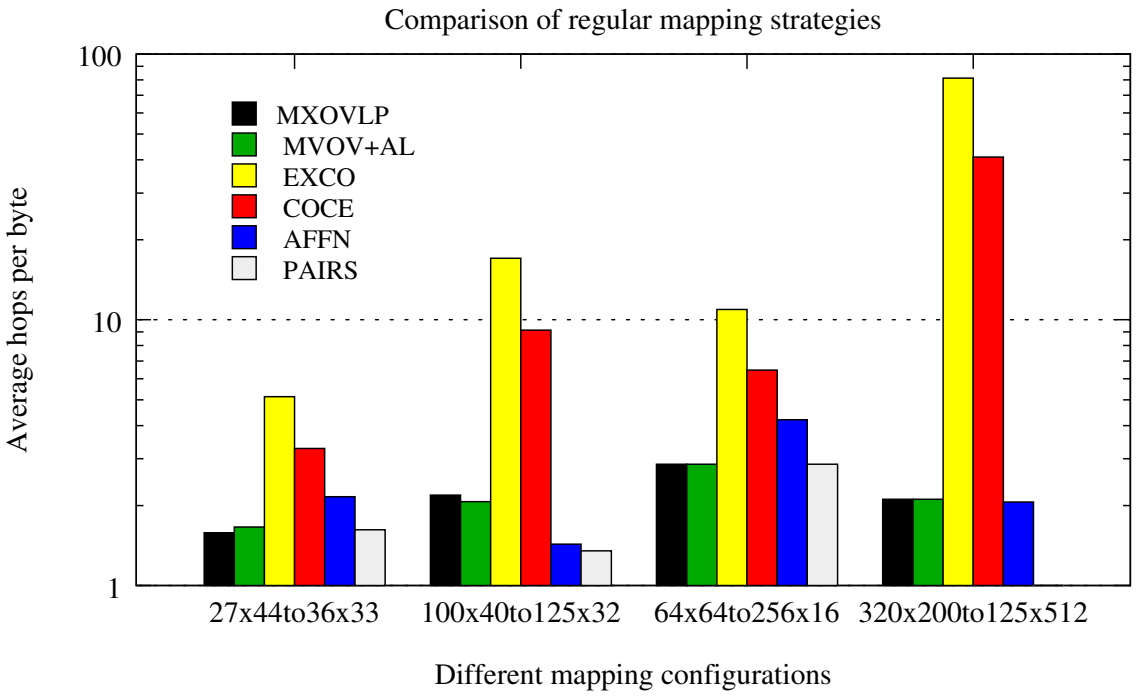


Figure 9.7: Hop bytes compared to the lower bound for different techniques

heuristic, elements at the boundaries of the subgrids are separated by a large distance in the processor mesh (as shown in Figure 9.8). To avoid this, we use a more general strategy where we fold the 2D object grid like an accordion folder and place the folded parts perpendicular to the smallest dimension. To achieve this, once we obtain the mapping for the first subgraph, instead of a simple translation, we flip the mapping for every alternate subgraph by 180 degrees. Figure 9.8 illustrates the folding technique to map a 2D grid to a 3D torus or mesh.

Heuristic 3: Space Filling Curve: A space filling curve is a continuous functions whose domain is the unit interval [0,1]. These curves (discovered by Peano in 1890 [97, 98] and generalized later on) can be used to fill the entire n-dimensional Euclidean space (where n is a positive integer). We use a space filling curve to map the 2D object grid to a 1D line and another space filling curve to map the 1D line to 3D processor mesh. Space filling curves preserve locality and hence we expect the dilation and hop bytes to be small under this construction.

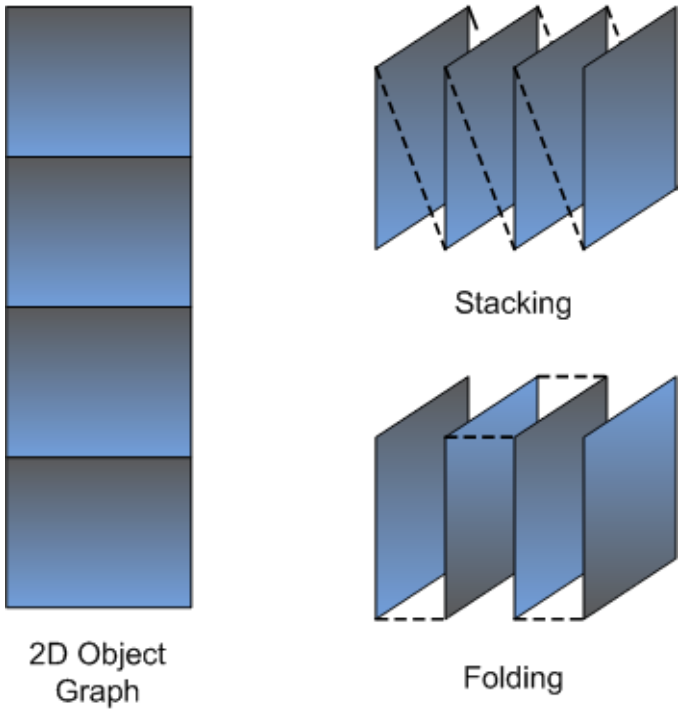


Figure 9.8: Stacking and Folding

9.3 Algorithms for Mapping 3D Grids to 3D Meshes

Several algorithms discussed in Section 9.2 above can be extended for the mapping of 3D object grids to 3D mesh topologies. Here, we discuss the extension of the affine mapping heuristic to 3D.

Affine Mapping: Similar to the 2D to 2D affine inspired mapping, we can do a mapping by using affine translations in all three dimensions (see Algorithm 9.4).

$$(x, y, z) \rightarrow (\lfloor P_x \times \frac{x}{O_x} \rfloor, \lfloor P_y \times \frac{y}{O_y} \rfloor, \lfloor P_z \times \frac{z}{O_z} \rfloor) \tag{9.2}$$

Since the coordinates are constrained to be integers, it is possible that two objects may be mapped to the same processor. To resolve this, we use the function `findNearest3D` which returns an unused processor closest to the desired unavailable

processor. Similar to the 2D affine mapping, a loose lower bound for the 3D affine heuristic is:

$$\text{Hops per byte} = \frac{P_x}{O_x} + \frac{P_y}{O_y} + \frac{P_z}{O_z}$$

Algorithm 9.4 Affine Mapping (`AFFN`) Heuristic for 3D to 3D mapping

```
procedure AFFN(O_x, O_y, O_z, P_x, P_y, P_z)
    for i := 1 to O_x do
        for j := 1 to O_y do
            for k := 1 to O_z do
                af_x = ⌊P_x × i/O_x⌋
                af_y = ⌊P_y × j/O_y⌋
                af_z = ⌊P_z × k/O_z⌋
                af = (af_x, af_y, af_z)
                free = findNearest3D(af, P_x, P_y, P_z)
                Map[i][j][k] = free
            end for
        end for
    end for
end procedure
```

9.4 Application Studies

Using the algorithms developed in the previous sections, we attempted topology aware mapping of two applications: a 2D Stencil benchmark in MPI and the Weather Research and Forecasting (WRF) program. All performance runs were done on the IBM Blue Gene/P machines at Argonne National Laboratory. We use `TXYZ` mapping as the default mapping. This means that MPI ranks are mapped in order on the four cores of a node first, then along the increasing X dimension, then Y and Z respectively.

9.4.1 2D Stencil

Many scientific applications [24, 92, 93] have a communication structure similar to a five-point stencil. We use a representative code where there is some computation followed by the exchange of data with neighbors in every iteration. Each element communicates with four neighbors in the 2D grid.

First, we study the weak scaling behavior of the application. In other words, the amount of computation and communication on each processor remains the same as we scale to more and more processors. The effect of congestion in the network, however, may be different with varying partition sizes (number of processors) owing to the difference in topology and corresponding mapping. We compare the performance of the default mapping with the "folding" scheme presented in Section 9.2.

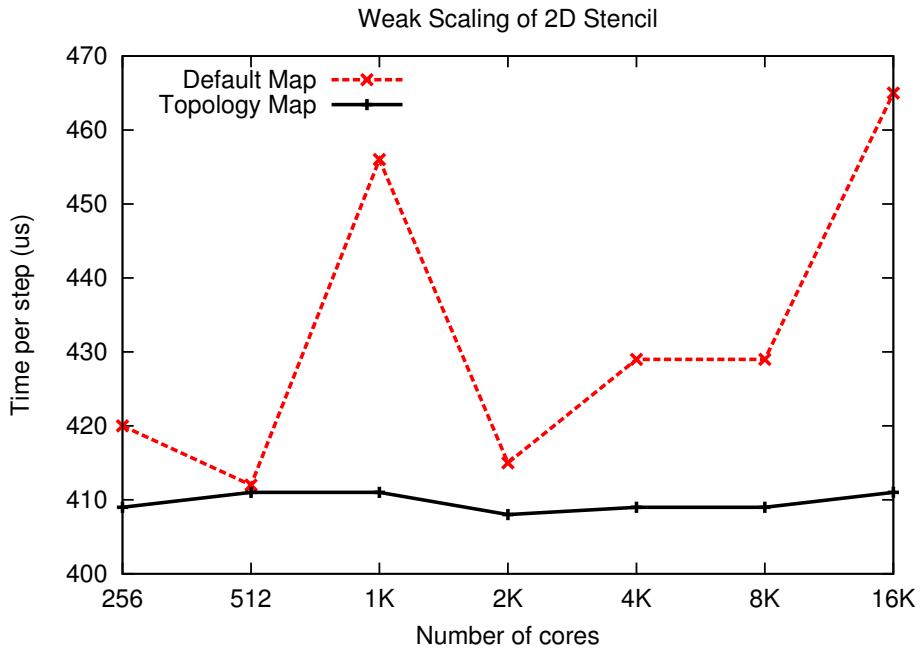


Figure 9.9: Weak Scaling experiment results for 2D Stencil

The results from this experiment are presented in Figure 9.9. Each processor holds 64×64 doubles and exchanges 4 messages containing 64 doubles (first and last rows and columns). Topology aware mapping leads to performance improvements

for most processor counts, the maximum being 11% at 16,384 cores. A peculiar observation is the reduction of time per step from 1,024 to 2,048 cores for the default `TXYZ` mapping. This is because the torus links become available only at 2,048-core and larger allocations. The availability of these links reduces the congestion significantly. We observe that as the application is run on increasing number of cores, the congestion in the network and hence the time per step keeps increasing for the `TXYZ` Mapping. On the other hand, using the topology aware mapping, the time per iteration for all the runs remains practically unchanged. We conclude that topology aware mapping can lead to performance benefits and improve scaling behavior for this class of applications.

It is interesting to compare the results in Figure 9.9 with the improvement in hops per byte (shown in Figure 9.10) in each of the above cases. It is evident that decreasing the hops per byte ratio leads to decreased congestion in the network. As demonstrated by the 1,024 and 16,384-core runs, a larger reduction in hop-bytes translates into larger performance improvements from topology aware mapping.

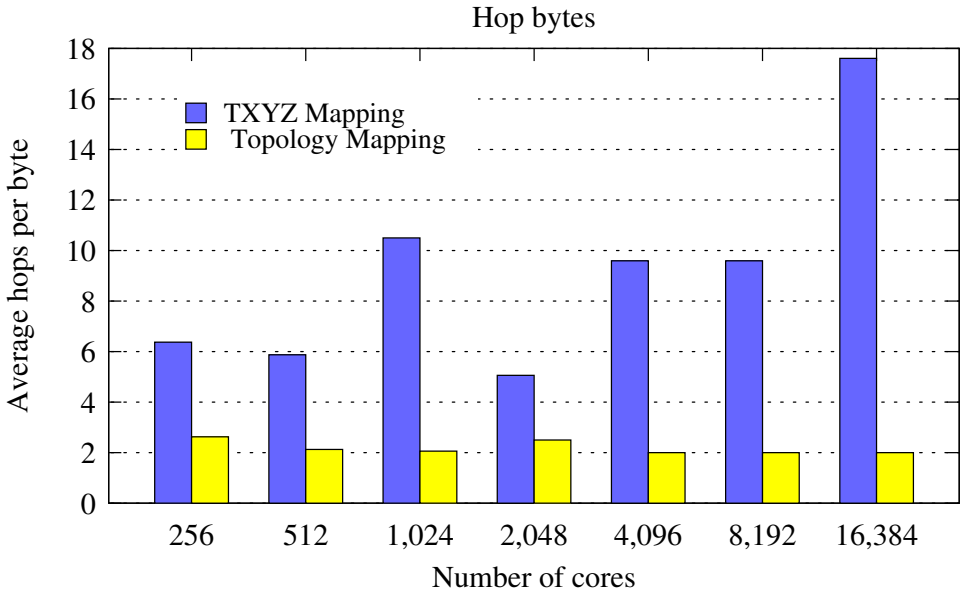


Figure 9.10: Average Hops per byte for 2D Stencil

We also study the effect of varying the ratio of computation and communication

for this application. This is motivated by the following factors:

- Different supercomputers have varying processing and communication characteristics. For example, comparing the network bandwidth available per floating point operation, BG/P can transfer 0.375 bytes per flop whereas Cray XT4 can transfer 1.357 bytes per flop to the network. The general trend for building faster supercomputers is increasing the number of cores per node and using faster processors. As a result, computation will tend to be faster and communication is likely to be a bottleneck.

- The same application may use a bigger stencil (e.g. a nine-point stencil) for the purpose of achieving faster convergence at the cost of doubling the communication and communication per time step. Increased communication may lead to an increased congestion in the network.

The ratio of computation versus communication was varied by increasing the message size while keeping the amount of computation constant. The results for this experiment are presented in Figure 9.11. On the X-axis, we have increasing message sizes for a fixed amount of computation. This experiment was run on two partitions of dimensions $8\times4\times8$ and $8\times8\times16$. Notice that the default mapping leads to significant congestion in the network leading to long delays and a considerable increase in the running time of the application. However topology aware mapping minimizes congestion and reduces the message delivery times, especially for large message sizes. The performance improvements for 8 KB messages at $1,024$ cores and $4,096$ cores are 66% and 53% respectively.

9.4.2 WRF Experiments

WRF stands for the Weather Research and Forecasting Model [24]. This code is a next-generation mesoscale numerical weather prediction system that is being

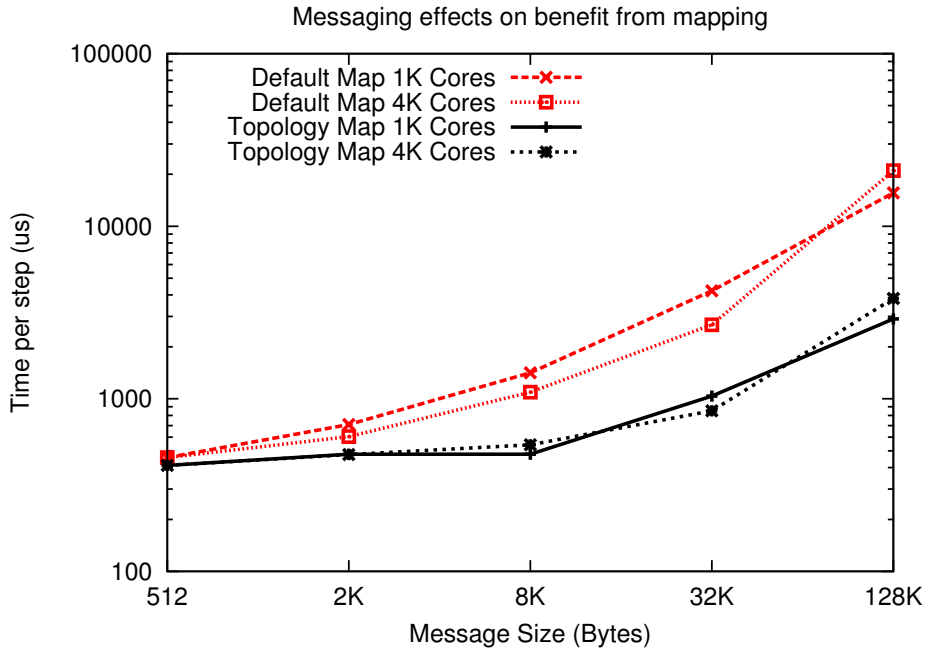


Figure 9.11: Effect of percentage of communication on benefit from mapping for 2D Stencil

designed to serve operational forecasting and atmospheric research. For our experiments, we used the weather data from the 12 km resolution case over the Continental U.S. (CONUS) domain on October 24, 2001. The benchmark simulates the weather for 3 hours using the data from a restart file. We used profiling tools to obtain the communication graph of WRF which was given as input to the mapping framework. The pattern matching algorithm (Section 8.1) found that WRF has a 2D near-neighbor communication pattern. Although developers of WRF would already know this, we used the pattern matching tool to validates its efficacy. Besides, the specific object graph, *i.e.* which rank communicates with which, still needs to be derived in absence of direct information from the application. The communication graph for WRF is a 2D grid with dimensions 16×16 on 256 processors, 32×32 on $1,024$ processors and so on.

Based on the pattern matching findings, the framework output mapping files which were passed as an option to the job scheduler on BG/P. WRF was run in the

SMP mode (using 1 process per node) because it uses OpenMP to create threads on each node. We compare the mappings with the default `XYZ` mapping on BG/P using average hops per byte. The hops per byte are obtained by using IBM's HPCT profiling tools [86]. Figure 9.12 shows the actual weighted hops (hops per byte averaged over all MPI ranks) obtained from profiling data for WRF. The corresponding percent improvements in application performance at each core count are listed above the bars. The empirically obtained values correlate strongly with the calculated theoretical hops. It is clear that topology aware mapping of MPI ranks to physical processors is successful in decreasing the average hops per rank for all the applications. The average hops for WRF reduce by 64% on $1,024$ and $2,048$ nodes which is quite significant and should lead to a dramatic drop in the load on the network.

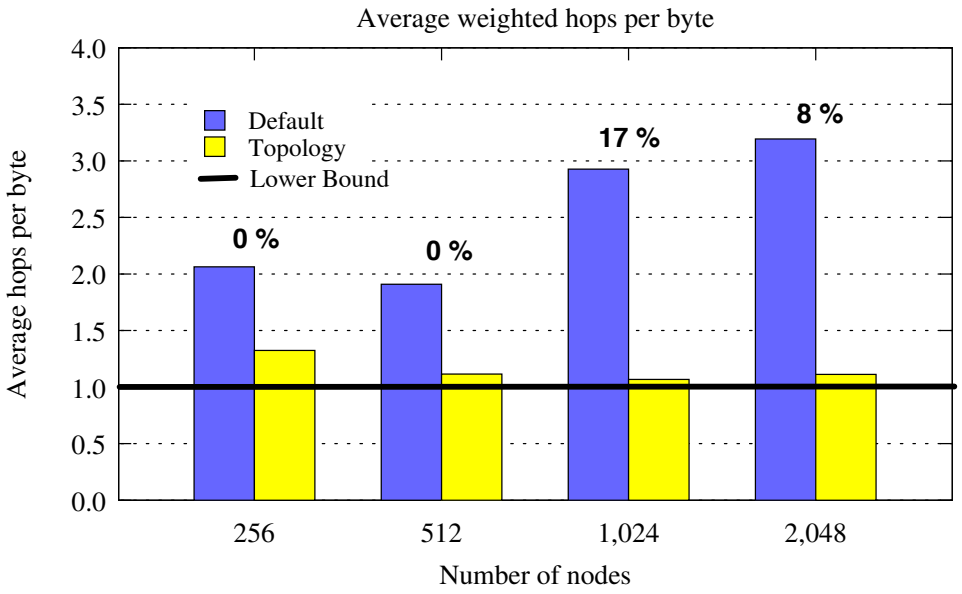


Figure 9.12: HPCT data for actual hops per byte for WRF

Using topology aware mapping, we were able to bring the average hops per rank close to 1 (Figure 9.12). This suggests that most MPI ranks are sending messages only 1 hop away and we should see performance improvements. When running on 256 nodes, we were able to reduce the average hops by 35% and the time spent in

`MPI_Wait` by 4%. This leads to a 2% reduction in overall communication time. This does not lead to any overall performance improvement. The results on 512 nodes are similar in spite of the reduction in hops. However, when we ran WRF on $1,024$ nodes, the average hops per byte reduced by 64% and communication time reduced by 45%. At $1,024$ nodes, communication is roughly 45% of the total time and hence we see an overall performance improvement of 17%. On $2,048$ nodes, there is similar improvement in hops and we obtain a performance improvement of 8%. This information is summarized in Table 9.1 below. Such performance improvements can be quite significant for the overall completion time of long running simulations. We expect that the gains when running WRF on large installations will be even more.

		Percentage Reduction		
Nodes	**WRF Mesh Dimensions**	*Hops*	*Comm. Time*	*Total Time*
256	16×16	35.9%	2%	0%
512	32×16	41.6%	2%	0%
1024	32×32	63.5%	45%	17%
2048	64×32	65.1%	−12%	8%

Table 9.1: Percentage reduction in average hops per byte, communication time and total time using topology aware mapping

It is important to keep in mind that the performance of a parallel application is a complex function of various factors. The routing protocols, latency tolerance of the application (see Chapter 11) and fraction of time spent in communication can affect performance in varying degrees. Hence, a reduction in hop-bytes and a corresponding improvement in the communication behavior of an application may not always lead to an overall performance improvement. However, learning from the WRF results, it is reasonable to expect that as we run on larger partitions, communication time will be a significant fraction of the overall execution time and hence the benefits from topology aware mapping will increase.

10 Mapping Irregular Communication Graphs

Parallel applications with irregular but static communication constitute another class of communication graphs. Unstructured mesh applications are a classic example of such patterns. Unlike applications with regular patterns, for applications of this class, we cannot exploit the regularity of the graphs and simpler embedding onto the regular 2D or 3D topology of the machine. This chapter discusses heuristic techniques for mapping irregular communication graphs to regular topologies.

We will discuss heuristics for two different cases of irregular graphs –

1. There is no information about the physics behind the application, from which the communication graphs were obtained. In this scenario, we use heuristics which exploit the neighbor relations between different nodes. The heuristics make no assumptions about patterns in the communication graph. However, if the domain is known to have a geometric structure, we can try to infer the geometric arrangement of the tasks, analogous to pattern matching for regular graphs.

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 with unstructured meshes, 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.

The heuristics presented in this chapter are applicable for mapping to 2D, 3D or even higher dimensional processor topologies. This is facilitated by the general idea on which all algorithms in this chapter are based: At each step, they pick 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 an efficient function to find the nearest available processor for a specific topology, the heuristics are generally applicable. Let us begin by looking at an implementation of the function for finding the nearest available processor in a 2D mesh.

10.1 Finding the Nearest Available Processor

If the "desired" processor for an object is not available, we place the object on the nearest available processor. Some algorithms such as affine mapping might lead to hits for the same processor for two objects and in this case, one of the objects has to be placed on a nearby processor. If we can implement a function which returns the nearest available processor given 1) a processor to start from, and 2) a table of which processors are available, we can use the algorithms for any topology.

Algorithm 10.1 shows the code for finding the nearest available processor in a two-dimensional mesh. We start from 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 code for doing the same in a 3D mesh looks very similar (see Algorithm B.1 in Appendix B). We can also extend this code

to 2D and 3D tori by extending the area under consideration on both sides to include wraparound links.

Algorithm 10.1 Finding the nearest available processor in 2D

```
procedure FINDNEAREST2D(x, y, freeProcs)
    diameter = P_x + P_y − 2
    if isAvailable2D(x, y, freeProcs) then
        return < x, y >
    end if
    for d := 1 to diameter do
        for i := (−1) × d to d do
            j = d−abs(i)
            r_x = x + i
            r_y = y + j
            if withinBounds2D(r_x, r_y) && isAvailable(r_x, r_y, freeProcs) then
                return < r_x, r_y >
            end if
            r_x = x + i
            r_y = y − j
            if withinBounds2D(r_x, r_y) && isAvailable(r_x, r_y, freeProcs) then
                return < r_x, r_y >
            end if
        end for
    end for
end procedure
```

The spiraling implementation presented above has a worst case time complexity of $\mathcal{O}(n)$. Hence, if `findNearest2D` is called for each node during mapping, it leads to a worst-case time complexity of $\mathcal{O}(n^2)$ for the mapping algorithm. Figure 10.1 shows the running time for the algorithm when it is called from one of the mapping algorithms (`AFFN`) for irregular graphs. Note that, time on the y-axis is in milliseconds and is on a logarithmic scale. 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.

However, in practice it is possible to keep the running time of this function constant by keeping a list of the available processors when their number drops

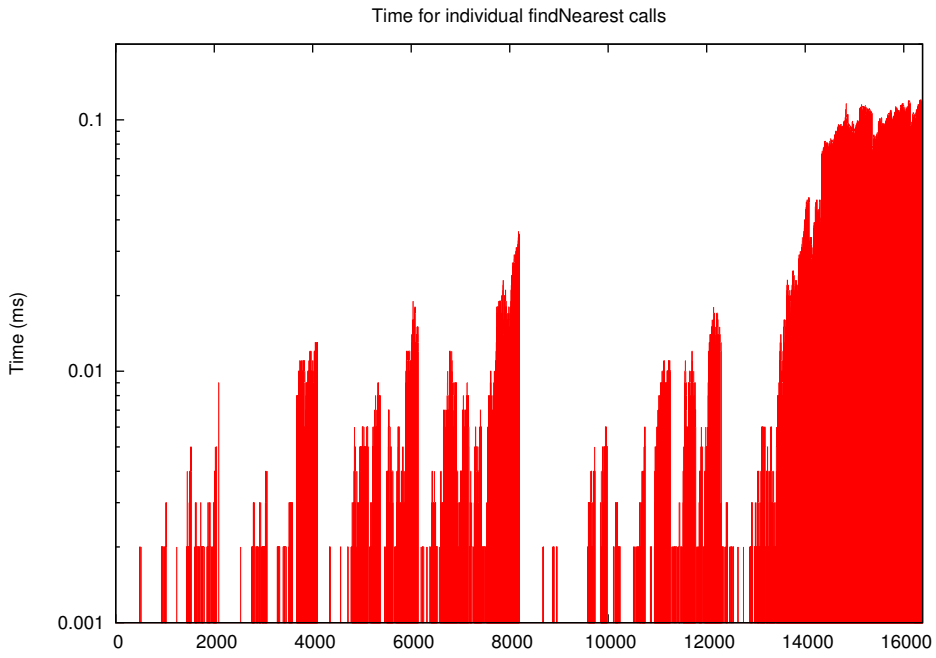
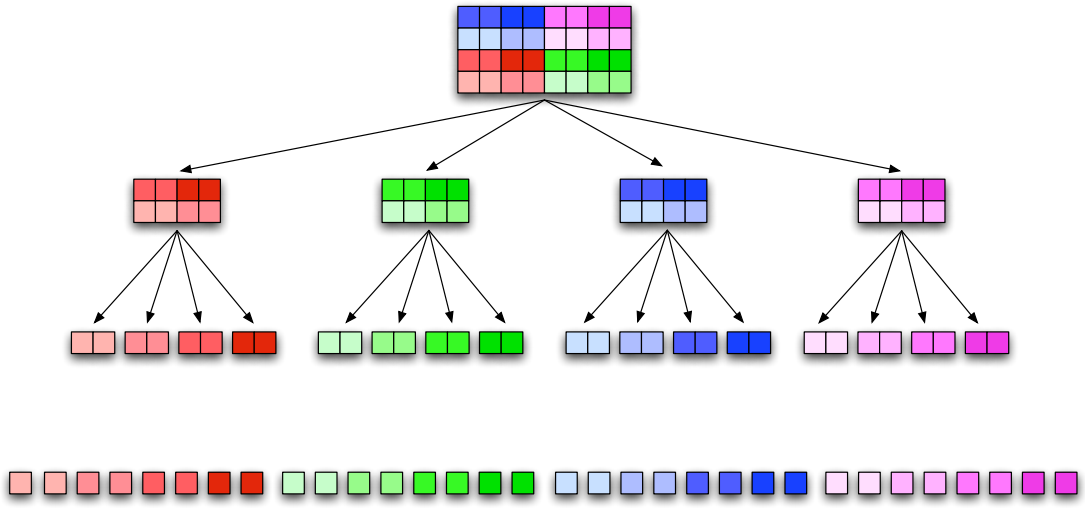


Figure 10.1: Execution time for 16,384 consecutive calls to the spiraling algorithm for `findNearest` from the `AFFN` algorithm for irregular graphs

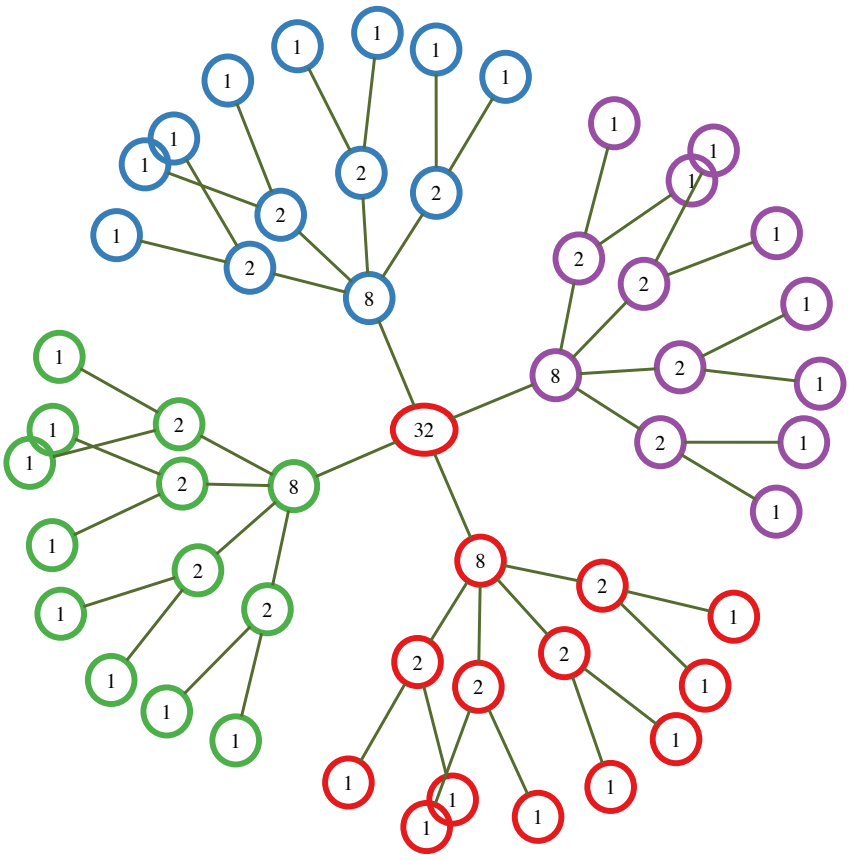
below a certain threshold. Using a quadtree data structure (octree in case of 3D), we think that the asymptotic time complexity of this algorithm can be reduced to $\mathcal{O}(logn \times logn)$, although we have not proved this yet. The next section describes the quadtree implementation of the algorithm.

10.1.1 Quadtree: An Alternative to Spiraling

We can build a quadtree representing the two-dimensional 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 $\mathcal{O}(log_4 n)$. At each node, the number of available processors is maintained along with information about the subdomain of the mesh controlled by it. Figure 10.2(a) shows the division of a 2D



(a) Recursive bisection of the mesh



(b) Quadtree view of the 2D mesh

Figure 10.2: Representation of a 2D mesh of processors of dimensions 4×8 as a quadtree

processor mesh into a quadtree and Figure 10.2(b) shows its tree representation with the label at each node indicating the number of available processors (initial view).

To find the nearest available processor, we start at the leaf which 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 $\mathcal{O}(log_4 n)$. When looking for a nearest available processor we start from a leaf and traverse all the way to the root (which takes $\mathcal{O}(log_4 n)$). 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 $\mathcal{O}(log_4 n \times log_4 n)$ by avoiding unnecessary traversals. Figure 10.3 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). In the next section, we compare the two implementations of `findNearest` when they are deployed in actual mapping algorithms.

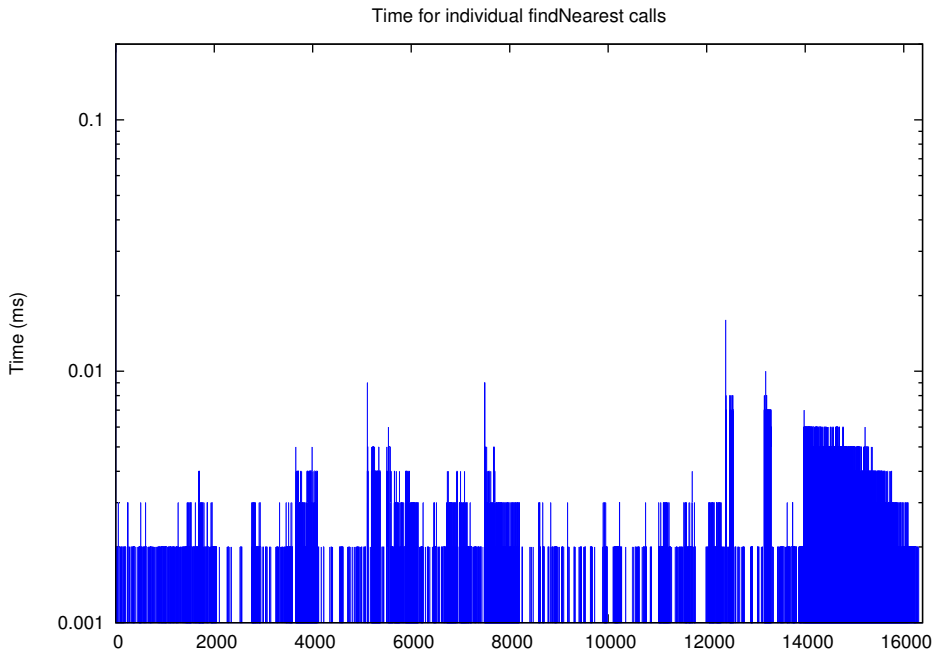


Figure 10.3: Execution time for 16,384 calls to the quadtree implementation for `findNearest` for the `AFFN` algorithm for irregular graphs

10.1.2 Comparison between Spiraling and Quadtree

The problem encountered with the spiraling implementation was that towards the end, when there were few available processors, it took a very long time to find a suitable processor. Comparing Figure 10.3 with the one for spiraling (Figure 10.1), we can see that we are successful in avoiding that problem. In fact, towards the end we are able to avoid visiting more and more nodes since most of the subtrees are empty, thereby leading to a decrease in the execution time. On the average calls to the quadtree implementation take much less time than the spiraling implementations (1.8 μs versus 16 μs).

Table 10.1 compares the execution time for the two implementations for a synthetic case. We start with an empty (all processors available) 2D mesh and look for processors around a certain processor, making them unavailable as we find them. The total time for finding all the processors one by one is recorded and tabulated.

The first three columns represent the case where we look for processors close to $(0, 0)$ and the remaining three 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 one corner, the speedup over the spiraling implementations is nearly 23 times for $16,384$ nodes.

	Start from one corner			Start from center		
Cores	**Spiraling**	**Quadtree**	**Speedup**	**Spiraling**	**Quadtree**	**Speedup**
1024	5.45	0.83	6.6	3.88	1.67	2.3
2048	20.92	2.44	8.6	13.09	5.02	2.6
4096	67.34	5.24	12.9	44.13	12.49	3.5
8192	304.64	16.61	18.3	198.12	38.05	5.2
16384	1005.82	44.23	22.7	676.52	98.48	6.9

Table 10.1: Comparison of execution time (in ms) for spiraling and quadtree implementations of `findNearest`

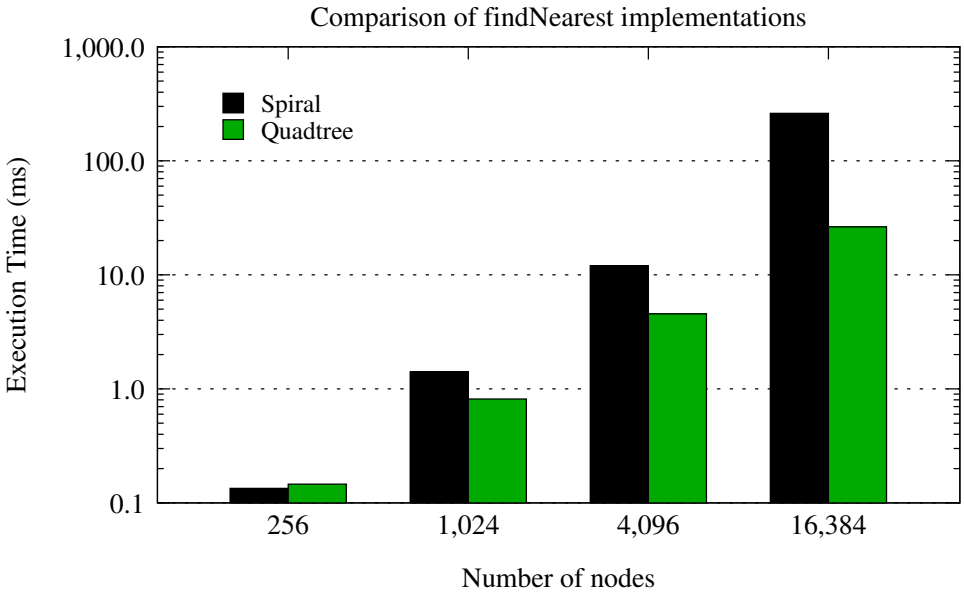


Figure 10.4: Comparison of execution time for spiraling and quadtree implementations when invoked from the `AFFN` mapping algorithm

Let us try to see the impact of using spiraling and quadtree in one of the mapping algorithms. We noticed that the `AFFN` heuristic for irregular graphs (similar to

the `AFFN` heuristic for regular graphs) takes a significant amount of time. This algorithm will be discussed later in this chapter but we use it as our test case for the performance of the `findNearest` function. Figure 10.4 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 which uses a quadtree is nearly 10 times faster than the one which uses spiraling.

Now we will discuss heuristics for two different classes of scenarios, depending on whether we do or do not have coordinate information associated with the communication graph. In section 10.3, we will try to employ some graph layout algorithms to obtain the coordinate information if it is not present.

10.2 Strategies for General Communication Graphs

In the most general case, 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 which exploit the neighbor relations between different nodes. The heuristics below make no assumptions about patterns in the communication graph. They also 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 nearby 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 and tables.

Algorithm 10.2 Breadth First Traversal (`BFT`) Heuristic

```
procedure BFT(commMatrix, P_x, P_y)
    // start with a random node start and a processor p and place start on p
    queue.push(start)
    Map[start] = p
    while !queue.empty() do
        start = queue.pop()
        p = Map[start]
        for i in neighbors_start do
            if Map[i] == NULL then
                queue.push(i)
                q = findNearest2D(p, P_x, P_y)
                Map[i] = q
            end if
        end for
    end while
end procedure
```

Algorithm 10.2 shows the pseudo-code for the `BFT` heuristic. 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 is good and places objects on the processor mesh in an organized way, each call to `findNearest2D` takes constant time.

Figure 10.5 shows a graph containing 90 nodes. This graph was obtained from a CHARM++ benchmark which does unstructured mesh computations on 2D meshes. 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 [99]. Each node might typically contain 100 to $10,000$ triangles. Figure 10.6(a) shows the default mapping of this graph on to a 2D processor grid of dimensions 15×6. Showing all the edges between the nodes of the communication graph gives us an idea of which algorithms stretch the edges more.

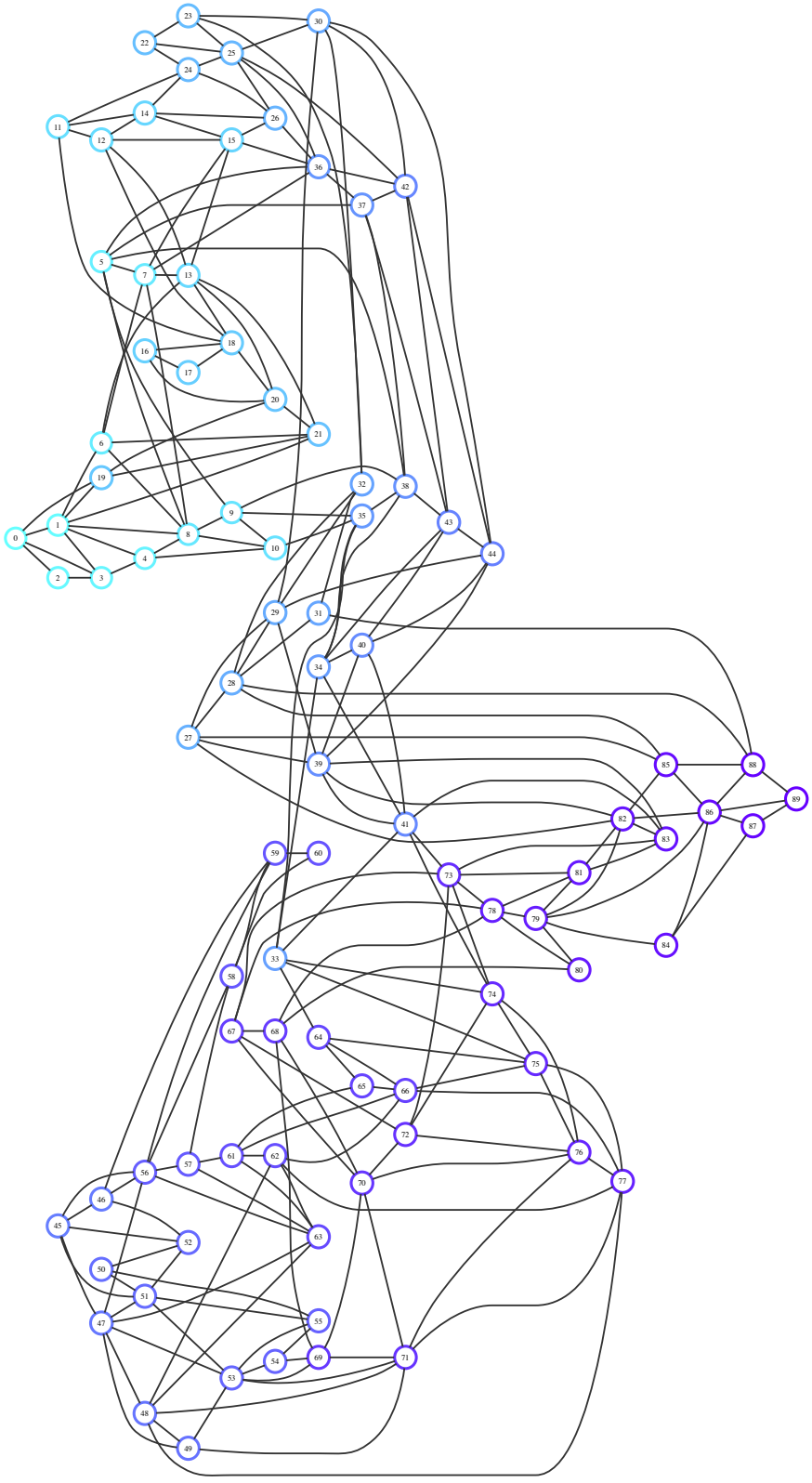
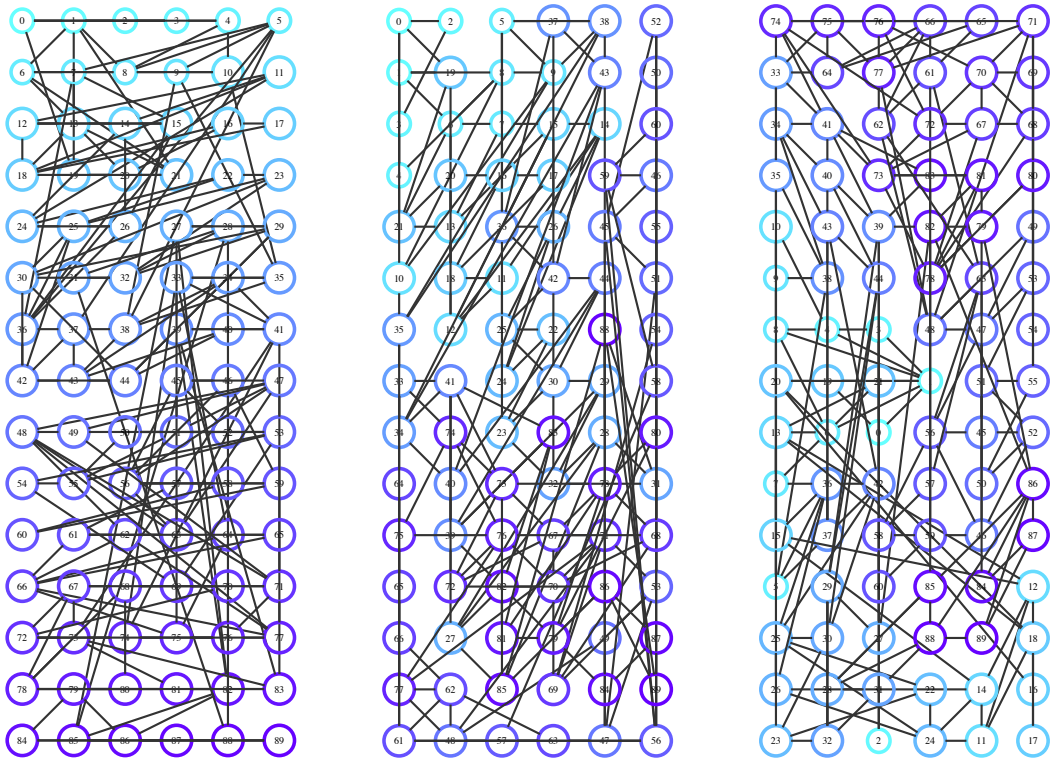


Figure 10.5: Irregular graph with 90 nodes

We can also see if there are certain areas of the processor mesh which are getting swamped by communication. Figure 10.6(b) shows the mapping of the irregular graph using the `BFT` heuristic on to a 2D mesh of dimensions 15×6.



(a) Default Mapping, Hops per byte: 3.07 (b) `BFT`, Hops per byte: 3.16 (c) `MHT`, Hops per byte: 2.75

Figure 10.6: Mapping of an irregular graph with 90 nodes using the (a) Default mapping, (b) `BFT`, and (c) `MHT` algorithm to a 15×6 grid

Heuristic 2 - Max Heap Traversal: This is an optimization over Heuristic 1. Here, we start with the node which 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, we give preference to nodes which have the maximum number of neighbors that have already been placed.

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 and tables. Figure 10.6(c) shows the mapping of the irregular graph shown earlier using the `MHT` heuristic on to a 2D mesh of dimensions 15×6.

10.3 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. This is analogous to pattern matching for regular graphs. Graph layout algorithms assign coordinates to each node for a layout of planar graphs using force-directed graph algorithms [100, 101]. We observed that graph layout algorithms created graphs which matched the actual geometry of the meshes quite well.

To infer the coordinates of nodes in a graph, we use the graphviz library [3], 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. Figure 10.7 shows the geometry inferred by the graphviz library for an irregular graph of 90 nodes (shown earlier in Figure 10.5). Neato implements the algorithms developed by Kamada and Kawai [100]. Once we have coordinates for the nodes from the graphviz library, we can use any of the algorithms discussed in the next section.

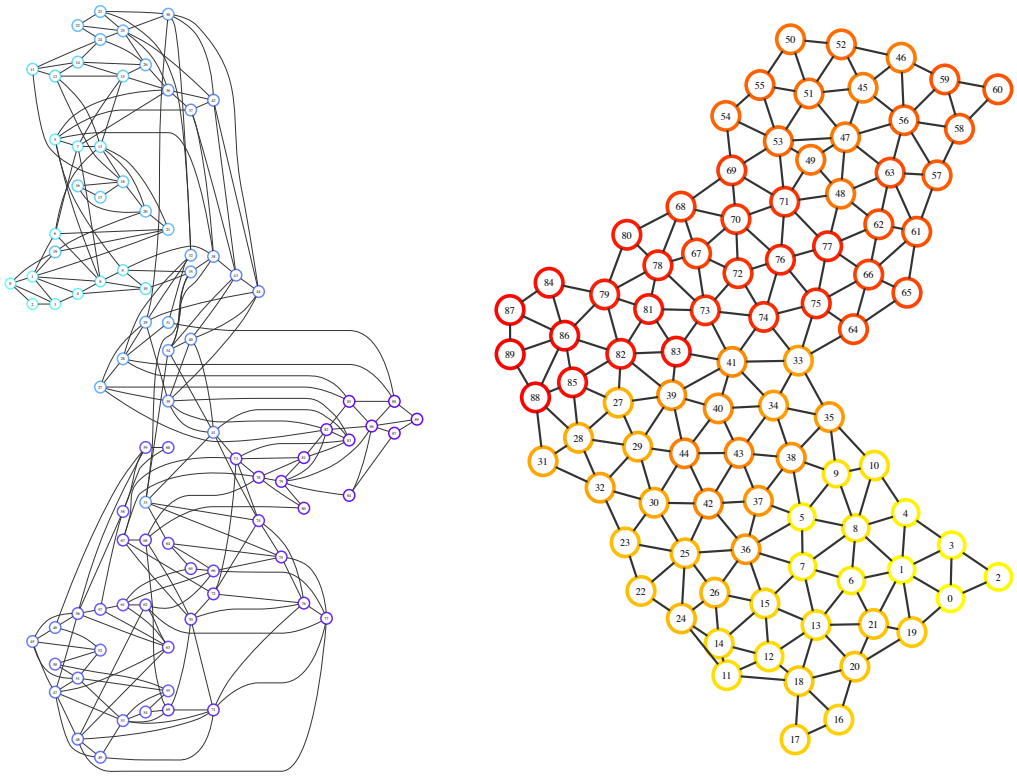


Figure 10.7: Using the graphviz library to infer the spatial structure of an irregular graph with 90 nodes

10.4 Strategies for Graphs with Coordinate Information

Quite often, when using unstructured meshes to simulate fractures in planes or solids which 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 1 - Affine Mapping: In this case, we try to do a geometric placement of the object graph on to the processor graph based on physical coordinate informa-

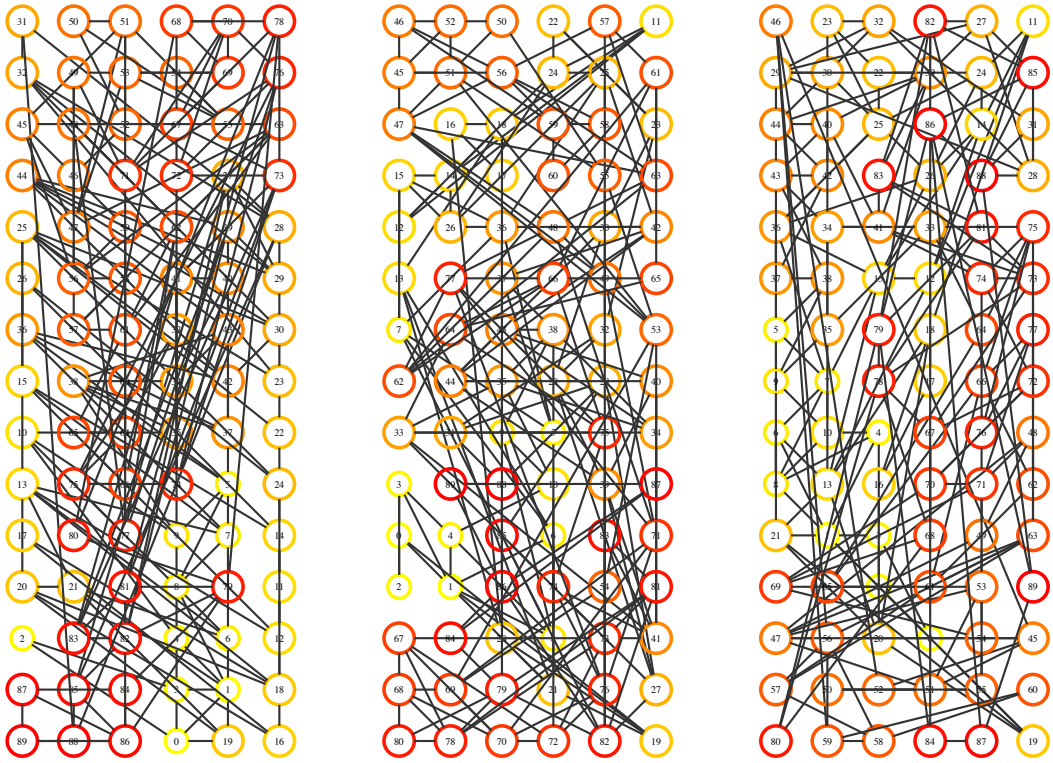
tion associated with each node. Each node in the graph has X and Y coordinates which are coordinates of the centroid for all triangles of the underlying mesh in that particular node. Based on its coordinates, if more than one node gets mapped to the same processor, all subsequent nodes after the first 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 object graph and may or may not give the best solutions depending on its aspect ratio. This algorithm (pseudo code in Algorithm 10.3) is referred to as `AFFN` in figures and tables.

Algorithm 10.3 Affine Mapping (`AFFN`) Heuristic

procedure AFFN($commMatrix, coordInfo, P_x, P_y$)
// let min_x, max_x, min_y and max_y denote the maximum x and y coordinates associated with any node
for $i := 1$ to Num_{nodes} **do**
$af_x = \lfloor P_x \times \frac{x - min_x}{max_x - min_x} \rfloor$
$af_y = \lfloor P_y \times \frac{y - min_y}{max_y - min_y} \rfloor$
$< free_x, free_y > =$ `findNearest2D`(af_x, af_y, P_x, P_y)
Map$[i][j] = free_x \times P_y + free_y$
end for
end procedure

Heuristic 2 - Corners to Center: Another heuristic that is similar to the ones we developed for regular graphs, starts from four corners of the object 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 the previous section to choose the mapping of the remaining nodes. We can either do a breadth first traversal from each node or we can do a max heap traversal and place



(a) `AFFN`, Hops per byte: 2.87 (b) `COCE`, Hops per byte: 2.89 (c) `COCE+MHT`, Hops per byte: 3.00

Figure 10.8: Mapping of an irregular graph with 90 nodes using the (a) `AFFN`, (b) `COCE`, and (c) `COCE+MHT` algorithms to a grid of dimensions 15×6

nodes with the maximum mapped neighbors first. `COCE` refers to the algorithm which uses the `BFT` heuristic for mapping the remaining nodes after the corners have been placed. `COCE+MHT` refers to the algorithm which uses the `MHT` heuristic for the remaining nodes. Figures 10.8 (a), (b) and (c) present the mapping of the same 90-node graph shown previously, using the `AFFN`, `COCE` and `COCE+MHT` mapping algorithms respectively.

10.5 Comparison of Strategies for 2D Irregular Graphs

This section compares the mapping heuristics discussed in the three sections above for mapping of 2D irregular graphs to 2D processor meshes. We compare the running time for different algorithms and also compare them based on achieved hop-bytes.

10.5.1 Time Complexity

The time complexity of the mapping algorithms depends on the running time of the `findNearest` implementation, that 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 $\Theta(logn)$ and inserts new elements in the heap which takes $\Theta(logn)$ for each insert. Every node in the graph is only inserted once and hence the total time complexity for the algorithm is $\mathcal{O}(n\ logn)$. The `COCE` heuristic takes linear time since it visits each node only once and needs constant time to place it. The `COCE+MHT` algorithm which uses the max heap technique takes $\mathcal{O}(n\ logn)$. The affine mapping heuristic also visits each node once and therefore has a linear running time. The running times for these algorithms are summarized in Table 10.2 below.

If we assume that `findNearest` has an average case time complexity of $\mathcal{O}(log_4 n \times log_4 n)$, then an additional $\mathcal{O}(n\ (log_4 n)^2)$ 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) are not practical. Figure 10.9 shows the actual running times for the mapping algorithms when run sequentially on a dual-core 2.4 GHz processor. We can see that `BFT` and `COCE` take less than 10 ms for a $16,384$-node graph.

	Time Complexity	
Algorithm	**Constant**	**Logarithmic**
BFT	$\mathcal{O}(n)$	$\mathcal{O}(n\ (log_4 n)^2)$
MHT	$\mathcal{O}(n\ logn)$	$\mathcal{O}(n\ (log_4 n)^2)$
COCE	$\mathcal{O}(n)$	$\mathcal{O}(n\ (log_4 n)^2)$
COCE+MHT	$\mathcal{O}(n\ logn)$	$\mathcal{O}(n\ (log_4 n)^2)$
AFFN	$\mathcal{O}(n)$	$\mathcal{O}(n\ (log_4 n)^2)$

Table 10.2: Time complexity for different mapping algorithms for irregular graphs assuming constant and logarithmic running time for `findNearest`

10.5.2 Quality of Mapping Solutions: Hop-bytes

We now compare the mapping heuristics discussed earlier using irregular graphs of varying sizes. These are mapped on two-dimensional 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 from the pairwise exchanges technique (PAIRS). We start with the solution obtained by MHT and do pairwise swaps until we have a "reasonable" value for hops per byte. Figure 10.10 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. The other three heuristics (BFT, COCE and AFFN) do not perform as well. Another heuristic we discussed was an extension an COCE where we start with four corners of the mesh but then choose the subsequent nodes to be mapped from a max heap. This heuristic (COCE+MHT) compares favorably with the MHT algorithm. It is interesting that MHT performs quite well even though it does not use coordinate information associated with the nodes.

Next we compare the effects of varying the aspect ratios of the processor graph (keeping the total number of processors constant). Table 10.3 presents the hops per

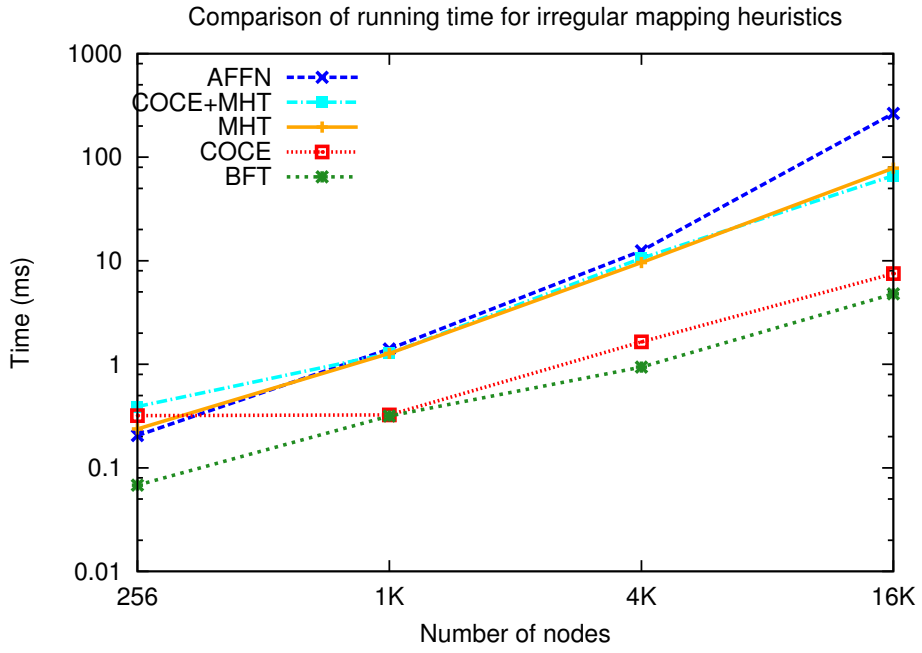


Figure 10.9: Running time for different irregular mapping heuristics

byte for mapping a $1,024$-node graph using various heuristics. The dimensions of the processor grid are varied from 512×2 to 32×32. A general trend which can be observed 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, only the `BFT` heuristic does better than the default mapping. Even for a mesh of 128×8, `BFT` is the best.

Mesh	Default	`BFT`	`MHT`	`COCE`	`COCE+MHT`	`AFFN`
512×2	11.97	11.67	22.97	15.86	14.85	14.70
256×4	6.93	6.57	11.00	8.69	8.28	7.87
128×8	5.02	4.37	6.27	5.45	4.95	4.77
64×16	4.93	4.03	4.59	4.45	4.02	3.92
32×32	6.28	4.87	3.23	4.66	3.29	4.55

Table 10.3: Average hops per byte for mapping of a $1,024$ node graph to meshes of different aspect ratios

The `MHT` heuristic, which performed the best for different mesh sizes in Figure 10.10, only works well when the shape of the processor mesh is close to a square.

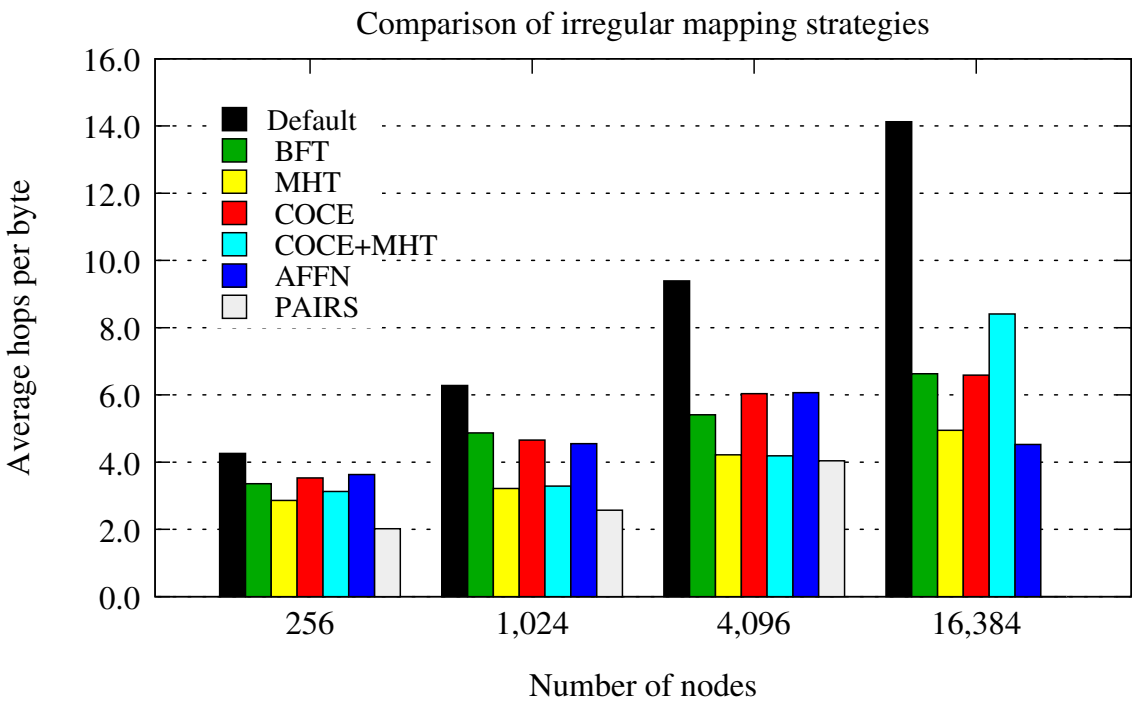


Figure 10.10: Hop-bytes for mapping of different irregular graphs to meshes of different sizes

For other aspect ratios, `BFT` is a good heuristic. We can also see that placing four corners of the mesh first (in the `COCE+MHT` heuristic) helps and this heuristic does very 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 are the "best" depending on a situation and that the time taken for the algorithms is short, it may be worthwhile to try all of them, and choose the best.

10.6 Application Studies

The previous sections 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 an unstructured mesh computation application to 3D torus partitions.

10.6.1 ParFUM Benchmark: Simple2D

Simple2D is a ParFUM benchmark which performs unstructured mesh computations on 2D meshes. ParFUM [23] is a framework within CHARM++ for writing unstructured mesh applications. We used the mapping algorithms discussed above to map the objects in Simple2D to 3D tori.

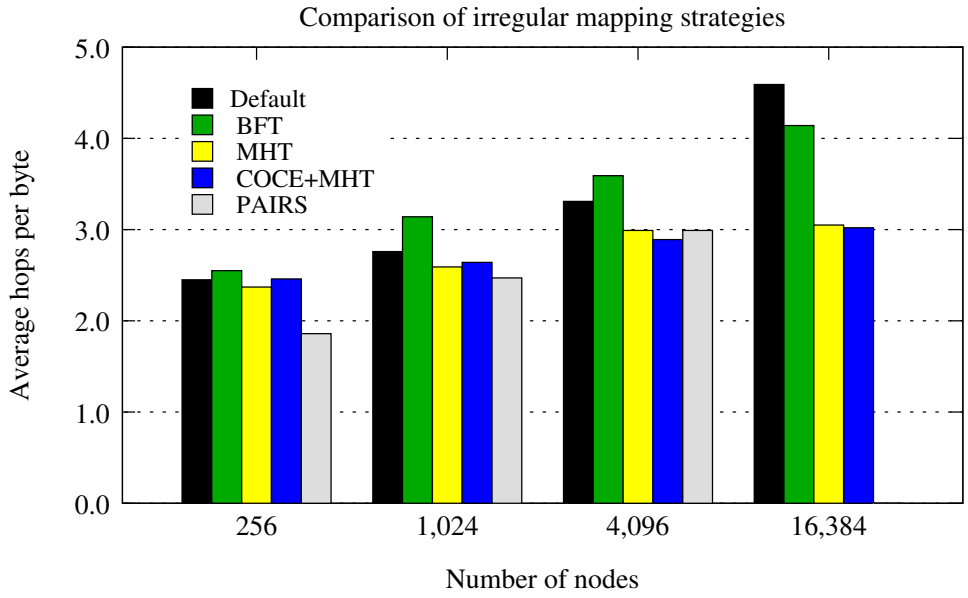


Figure 10.11: Hop-bytes for mapping of different irregular graphs to meshes of different sizes

Figure 10.11 presents the hop-bytes for mapping of this unstructured mesh application 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 10.5 and were performed with a virtualization ratio of 1. 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 results in similar average hops per byte on large number of processors.

An important observation is that the default mapping is much better than what a random or initial mapping would obtain for the hops per byte values. Let us analyze a particular case to get a better idea. Assuming that the 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 10.11, the hops per byte value 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 by their IDs to the processors. This suggests that the IDs of the objects 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 partitioning algorithm [99] which is used for partitioning the unstructured mesh into a graph of objects. METIS tries to preserve communication properties of the graph by numbering communicating objects with nearby IDs. As a result, we do not see much improvement in hops per byte over the default mapping using the heuristics presented (especially for smaller partitions.) The small improvement in hops per byte over the default mapping does not translate into actual performance improvements when running Simple2D on Blue Gene/P and we will analyze this further as part of future work.

11 Virtualization Benefits

Parallelizing an application consists of two tasks: 1. decomposition of the problem into a large number of sub-problems to facilitate efficient parallelization to thousands of processors, 2. mapping of these sub-problems on to physical processors to ensure load balance and minimum communication. Object-based decomposition separates the two tasks and gives independent control over both of them. This chapter discusses the benefits of virtualization, achieved through overdecomposition, in mitigating contention and facilitating topology aware mapping.

In MPI and most other parallel paradigms, the number of tasks into which the problem is decomposed is equal to the number of physical processors used. The Charm++ runtime allows the application developer to decompose the problem into objects (or virtual processors or VPs), whose number can be much larger than the number of processors (P). This overdecomposition of work (VP much greater than P) is called processor virtualization. The runtime does a default mapping of the objects that can be overridden by the user.

The following sections describe how virtualization helps in mitigating contention and facilitating mapping and in addition, how the Charm++ runtime makes the automatic mapping framework more useful.

11.1 Reducing Impact of Network Contention

Virtualization through overdecomposition can reduce the performance impact of network contention in two ways:

Reducing Contention Overdecomposition can help in reducing contention by scheduling different objects over time and preventing a burst of messages on the network. The network can get flooded with messages if all processors are sending their data to others at the end of an iteration. When there are multiple objects on each processor, they would finish their iterations at different times, therefore, sending messages progressively and avoiding congestion on the network.

Mitigating the Effect of Contention: Having multiple objects on a processor that can be scheduled dynamically as needed leads to an overlap of computation and communication (Figure 11.1). When a certain object is waiting for messages to arrive, before it can proceed with computation, another object on the same processor can be scheduled to do work if it is ready. If we can ensure that there is always some work to be scheduled, the processor would not have to wait for messages. In the event that certain messages are delayed because of contention, overdecomposition increases the tolerance for increased latencies compared to when no overlap is possible.

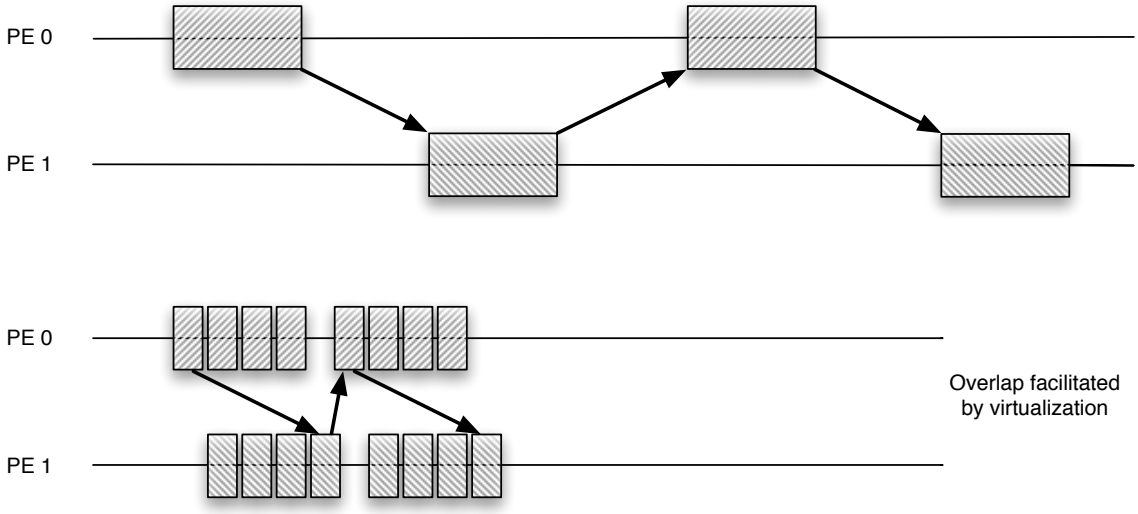


Figure 11.1: Overlap of computation and computation increases tolerance for communication delays

Thus, virtualization helps by (a) spreading the communication over time, thereby reducing the chance of contention in the network and (b) if there is still contention (and associated delays), it helps by reducing the sensitivity of the application to the delays.

11.1.1 Experimental Evidence

Further evidence for this can be seen by experiments we performed with a 3D Stencil computation. In a seven-point 3D Stencil, the 3D data array is divided among a 3D array of objects. The number of objects in case of CHARM++ can be much more than the number of processors. We ran a 3D Stencil program on the Blue Gene/L machine at Watson in CO and VN mode. The results are presented in Tables 11.1 and 11.2 below.

	32768 chares		**4096 chares**	
Cores	**RR**	**TO**	**RR**	**TO**
512	23.06	20.49	29.11	24.86
1024	11.54	10.23	17.42	14.78
2048	6.66	5.29	8.29	7.74
4096	3.15	2.82	4.05	3.43
8192	1.68	1.51	-	-
16384	0.89	0.86	-	-
32768	-	-	-	-

Table 11.1: Execution time (in milliseconds) of 3D Stencil on Blue Gene/L (CO mode) for different number of chares per processor (RR: Round-robin, TO: Topology aware)

As we can see (in Table 11.1), using more chares or objects per core gives better performance for all core counts. When comparing the round robin columns at different core counts, it is evident that having more chares per core leads to more overlap, which minimizes the wait for messages and therefore yields better performance. It is important to remember that having more chares leads to blocking of data and hence

better cache locality in most cases. This can also contribute to better performance when using more VPs per PE.

We would also expect that the cases with no overlap would benefit more using topology aware mapping than those where overlap through virtualization can handle delays. For example, when running in CO mode on 4,096 cores, we obtain a performance improvement of 15% from topology aware mapping when using one VP per PE and only 10% when using 8 VPs per PE. Similar results are observed when running in VN mode (Table 11.2).

	32768 chares		**4096 chares**	
Cores	**RR**	**TO**	**RR**	**TO**
512	62.15	51.09	78.44	47.08
1024	30.72	25.22	38.48	29.86
2048	23.24	11.23	19.49	17.10
4096	9.08	5.73	9.34	9.81
8192	4.02	3.25	4.63	5.25
16384	2.07	1.95	-	-
32768	2.47	1.23	-	-

Table 11.2: Execution time (in milliseconds) of 3D Stencil on Blue Gene/L (VN mode) for different number of chares per processor (RR: Round-robin, TO: Topology aware)

11.2 Facilitating Topology Aware Mapping

Virtualization also facilitates topology aware mapping by providing additional degrees of freedom since there are more objects which can be mapped. The Charm++ runtime in particular, helps in automating the process of mapping by offering runtime instrumentation support and dynamic load balancing.

11.2.1 Additional Degrees of Freedom

Virtualization leads to division of the work on a given processor into multiple objects. This division can depend on the type of work and on the temporal nature of when the work is performed. Hence, creating virtual processors divides objects by type of computation and by different phases in the application. OpenAtom is a very good example of division of computation into multiple chare arrays (see Chapter 7).

Multiple chare arrays which perform different kinds of computation and communication at different points in time gives the mapping framework more flexibility for placing the objects (see Figure 7.1 for details). Figure 11.2 shows a Projections [102] timeline view of the various phases in a time step of OpenAtom. Different phases of an iteration (showing interactions between different chare arrays) can be seen in different colors. At the processor level, OpenAtom presents a scenario where a given processor communicates with different sets of processors in different phases. Since we decompose the work on each processor into multiple objects, we have a flexibility to map these objects based on their communication with other objects. In case of conflicting communication patterns with different sets of objects in different phases, overdecomposition gives us flexibility of mapping which would not be available otherwise.

11.2.2 Instrumentation and Dynamic Load Balancing

There are two limitations when using the automated mapping framework with MPI applications:

1. There is a lack of sophisticated profiling tools for MPI which can provide the communication graph of an MPI application at runtime. Hence, the communication graph from a given run is used offline to develop mapping solutions.

2. As an effect of 1. and also because of lack of support for migrating MPI ranks

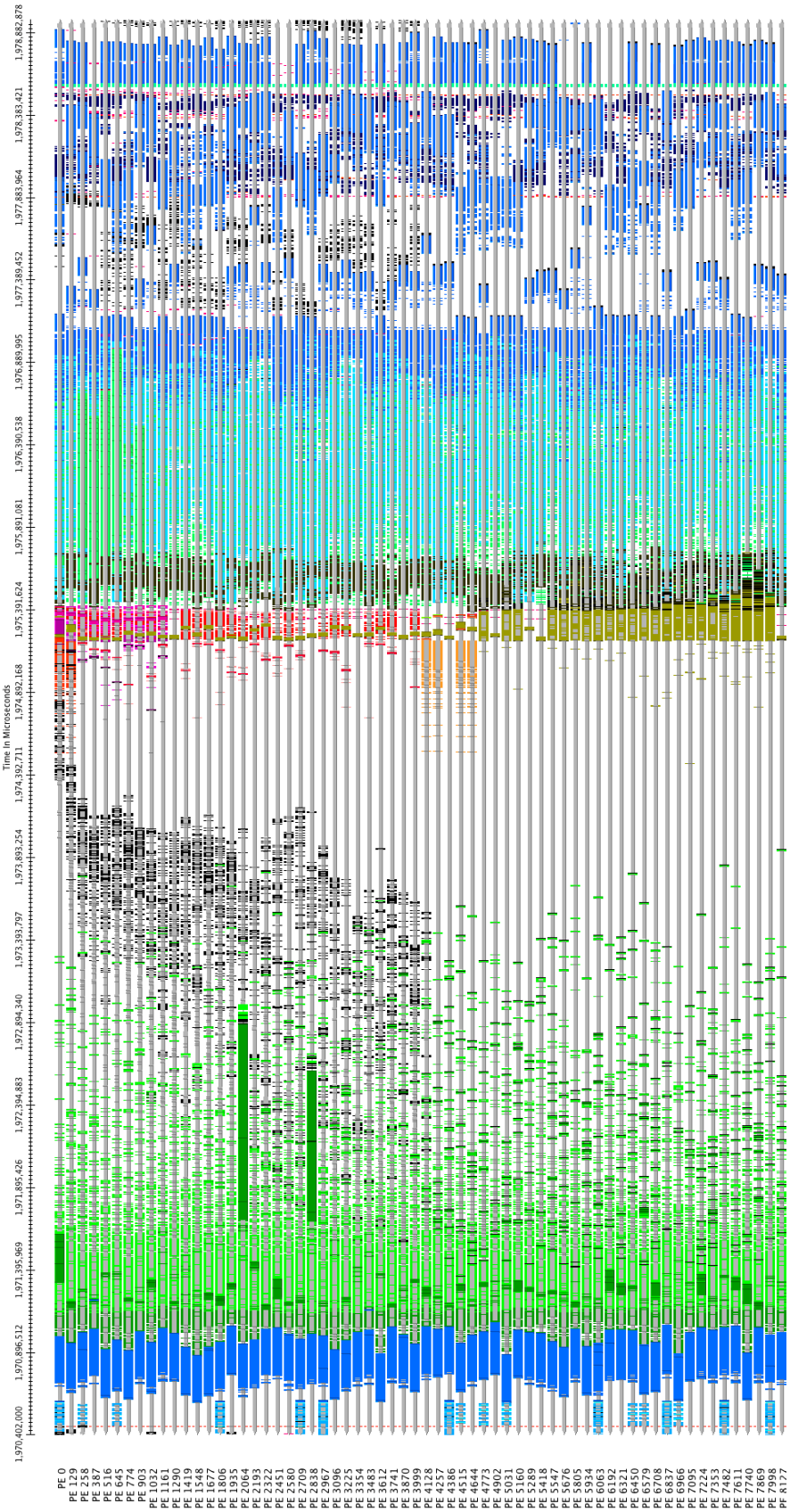


Figure 11.2: Decomposition of the physical system into chare arrays (only important ones shown for simplicity) in OpenAtom

at runtime, mapping can only be changed in a subsequent run and only at start-up.

Because of these limitations, the mapping framework is useful only for those MPI applications where the communication graph can be assumed to be the same across two different runs (at least for the same number of processors) and if it can be assumed that the communication graph does not change during a run. However, there are several MPI applications which fall in this category.

CHARM++ applications do not suffer from the limitations above. The CHARM++ runtime instruments the application and hence the information is available during a run to enable runtime remapping (in other words, communication-aware load balancing). Also, CHARM++ and Adaptive MPI [103] support runtime migration of objects and hence if the communication graph changes as the application is running, we can reassign the objects during the run. The results for the ParFUM benchmark in Chapter 10 have been obtained by an integration of the automatic mapping framework with the mapping/load balancing modules of the CHARM++ runtime.

12 Scalable Mapping and Load Balancing

Most of the mapping algorithms developed in the automatic mapping framework are sequential. When used at runtime, such as with Charm++, they work by gathering the instrumented communication graph on one processor, running the mapping algorithm, and then scattering the decisions to the relevant processors so they can migrate the objects to the new target processors. Machines with hundreds of thousands of cores will require million-way parallelism and our mapping algorithms should be able to handle such scales. The scaling bottlenecks, when we have a large number of objects and processors are the following:

- **Communication bottleneck:** If the loads and communication edges for all objects are collected on a single processor, there are millions of messages being sent to that processor simultaneously which can become a communication bottleneck. After the load balancing decisions have been made, they have to be sent to all the objects which again leads to communication problems.

- **Memory constraints:** A single processor has a limited amount of memory directly connected to it. At large scales, it becomes infeasible to store the entire communication graph on a single processor.

- **Processing power:** If the load balancing decisions are made by a single processor, load balancing might take a very long time depending on the number of objects and the running time of the algorithm.

Hierarchical mapping by forming smaller groups and assigning a master rank in each group to calculate the mapping decisions is one technique to improve scal-

ability [104, 105]. It mitigates all of the problems mentioned above but does not remove the bottlenecks completely. Also, hierarchical algorithms do not have global load and communication information and hence the load balancing solutions might be inferior compared with the centralized schemes. Another possible approach is to parallelize the mapping process completely by allowing each object to decide its destination processor. Fully distributed strategies [106–109] that use information from topological neighbors only are not adequate because they are too slow to disperse the load, in absence of global information. Hence, we want each object's mapping to have some sense of the global load and communication information. This chapter discusses completely distributed mapping and load balancing strategies (with global information).

In this chapter, the mapping problem is generalized to topology aware load balancing. The problem can be stated as follows: there are n objects to be placed on p processors (n is much greater than p). Each object (assigned to some processor initially) has a different computational load and communicates with some other objects. The goal is to achieve close to optimal load balance and optimize communication considering the topology as well. The problem in this most general form is NP-hard. Centralized and hierarchical solutions to this problem collect the loads of all objects on one or a few processors which are responsible for balancing the load. For completely distributed strategies, we assume that the "knowledge" of the computational loads of objects are distributed and each object should make its decision by itself. The strategies discussed in this chapter are relevant in the context of CHARM++ programs because we can have multiple objects per physical processor. They are also applicable to other MPI applications such as multi-bock codes that do explicit application-level load balancing.

We discuss load balancing solutions for a few scenarios. Two specific cases are:

1. We have a one-dimensional array of objects which communicate in a ring

pattern (i.e. object i communicates with $i-1$ and $i+1$). We want to map these objects on to linear array of processors. We use this case to illustrate the basic technique in a simpler context.

2. The second scenario is where we have a two-dimensional array of objects where each object communicates with two immediate neighbors in its row and two in its column. We wish to map this group of objects on to a 2D mesh of processors.

12.1 Mapping of a 1D Ring

Problem: Load balancing a 1D array of v objects which communicate in a ring pattern to a 1D linear array of p processors.

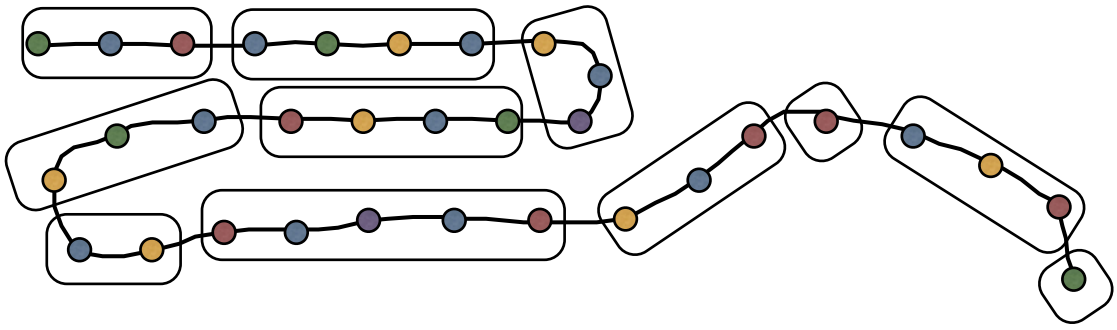


Figure 12.1: The solution of the 1D ring load balancing problem involves finding the right places to split the ring for dividing among the processors

Solution: We want to map these objects on to processors while considering the load of each object and the communication patterns among the objects. In order to optimize communication, we want to place objects next to each other on the same processor as much as possible and cross processor boundaries only for ensuring load balance. We assume that the IDs of objects denote the nearness in terms of who communicates with whom. Hence the problem reduces to finding contiguous groups

of objects in the 1D array such that the load on all processors is nearly the same (see Figure 12.1).

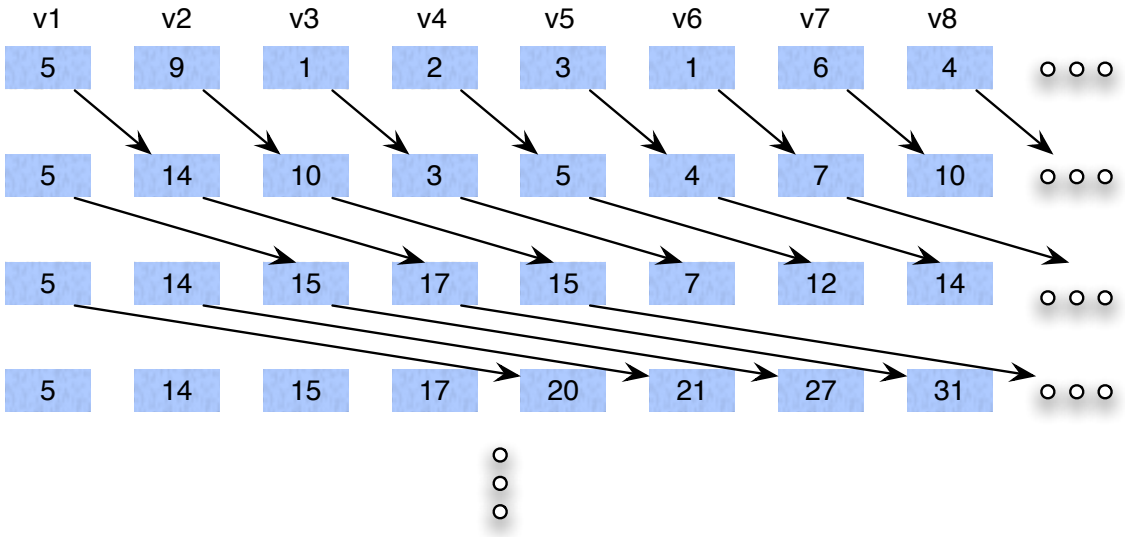


Figure 12.2: Prefix sum in parallel to obtain partial sums of loads of all objects up to a certain object

We arrange the objects virtually by their IDs and perform a prefix sum in parallel between them based on the object loads. At the conclusion of a prefix sum, every object knows the sum of loads of all objects that appear before it (Figure 12.2). Then, the last object broadcasts the sum of loads of all objects so that every object knows the global load of the system. Each object i can calculate its destination processor (d_i), based on the total load of all objects (L_v), prefix sum of loads up to it (L_i), its load (l_i) and the total number of processors (p), by this equation,

$$d_i = \lfloor p * \frac{L_i - l_i/2}{L_v} \rfloor \tag{12.1}$$

In summary, every object decides its destination processor in parallel through a parallel prefix operation and then migrate to the respective processors.

12.1.1 Complexity Analysis

Let us compare the running time and memory requirements for the centralized versus completely distributed load balancing algorithms. Let us assume that there are v objects (or VPs) to be placed on p physical processors.

In the centralized scheme, one processor stores all the information and hence the memory requirements are proportional to the number of objects, v. In the distributed case, each processor stores information about its objects which is v/p. In the centralized case, each processor sends a message to one processor with its load information, which leads to p messages of size v/p each. On the other hand, in the parallel prefix there are $logv$ phases and v messages of constant size are exchanged in each phase. These comparisons are summarized in Table 12.1 below.

Strategy	**Centralized**	**Distributed**
Memory	$\mathcal{O}(v)$	$\mathcal{O}(v/p)$
Number of messages	$\mathcal{O}(p)$	$\mathcal{O}(v\ logv)$
Communication Volume	$\mathcal{O}(v)$	$\mathcal{O}(v\ logv)$
Decision Time	$\Omega(v)$	$\mathcal{O}(1)$

Table 12.1: Time complexity comparison of centralized and distributed load balancing algorithms for a 1D ring

In the centralized case, if we assume that the fastest algorithm for load balancing will have a linear running time, the time complexity for the decision making can be said to have an asymptotic lower bound of $\Omega(v)$. In the distributed case, since every object makes its own decisions, the algorithm's decision time is constant. Hence, even though there is more communication for completely distributed algorithms, we get huge savings in memory usage and decision time, which will be necessary when v is very large. Also, the serial bottleneck at processor 0, which must receive $\mathcal{O}(p)$ messages in the centralized case, is removed.

12.2 Mapping of a 2D Stencil

Problem: Mapping a 2D array of objects which communicate in a stencil-like pattern, to a 2D mesh of processors.

Solution 1: In this case, the assumption is that object $(i,\ j)$ communicates with four other objects: $(i,\ j-1)$, $(i,\ j+1)$, $(i-1,\ j)$ and $(i+1,\ j)$. And as in the previous case, we want to optimize communication and balance computational load. Again, non-trivial cases that are of interest are when the aspect ratios of the object mesh and the processor mesh are different.

One possible solution is to do a parallel prefix sum in every row and column. Each object can then decide the x and y coordinates of its destination processor based on the the loads it has from the two prefix sum operations it participated in and the total loads of its row and column. However, this might lead to collisions for the same destination processor and load imbalance, so we might need another step of neighborhood load balancing afterwards. In neighborhood load balancing, processors with less than average work would request work from their neighbors.

Solution 2: Another solution is to use a parallel prefix in 1D as we did in the previous section. To do this, we have to linearize the objects in some fashion. Space filling curves can be used to map the 2D object grid to a 1D line [97, 98]. Space filling curves preserve the neighborhood properties of the objects in 2D. Figure 12.3 shows the linearization of an object grid of dimensions 32×32 and a processor grid of dimensions 8×8.

Having linearized the object grid, we can perform a parallel prefix on the 1D array of objects and obtain a destination processor for each object. This processor number is the linearized index of each processor if we create a space filling curve for the 2D processor mesh. Hence, based on this linearized index, we can obtain

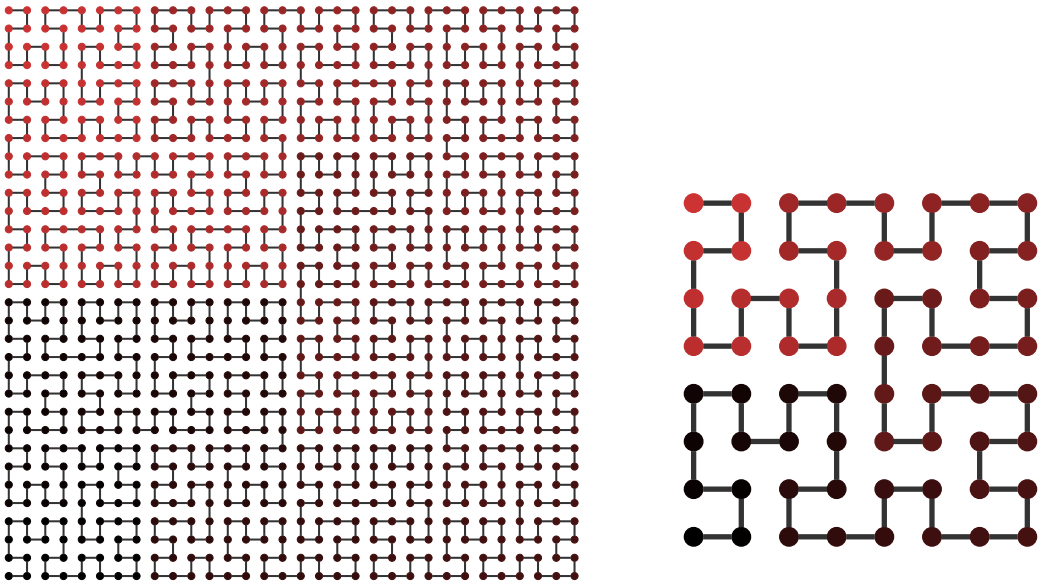


Figure 12.3: Hilbert order linearization of an object grid of dimensions 32×32 and a processor mesh of dimensions 8×8

the x and y coordinates of the processor by decoding, using the same logic used for generating space filling curves. The hope is that nearby objects in the 2D object grid end up close to one another on the 2D processor mesh.

12.2.1 Performance Results

The two solutions mentioned above for 1D and 2D (both using a one-dimensional parallel prefix) were implemented in Charm++. Table 12.2 shows the time it takes for distributed load balancing of 1 and 4 million chares with a 2D communication pattern.

Cores	*1 million chares*	*4 million chares*
4096	1.81	12.96
16384	6.77	9.59

Table 12.2: Time (in seconds) for distributed load balancing

Distributed load balancing for 1 million chares take only 2 and 7 seconds on

4,096 and 16,384 cores of XT5. For 4 million chares, the distributed algorithm takes about 13 seconds on 4,096 cores and 10 seconds on 16,384 cores. Table 12.3 shows the improvement in hops per byte when using the distributed topology aware algorithms compared to a random and default mapping. Five trials were done generating random loads for 1 million objects running on 4,096 cores. Since, there are several thousand objects on each core, the ideal hops per byte is significantly less than one. The load balancers succeed in reducing the average hops per byte significantly.

Mapping	**Random**	**Default**	**Topology**
Trial 1	42.65	3.76	0.11
Trial 2	42.66	3.76	0.12
Trial 3	42.67	3.76	0.12
Trial 4	42.67	3.76	0.12
Trial 5	42.66	3.76	0.12

Table 12.3: Reduction in hops per byte using distributed topology aware load balancing for 1 million objects on 4,096 cores

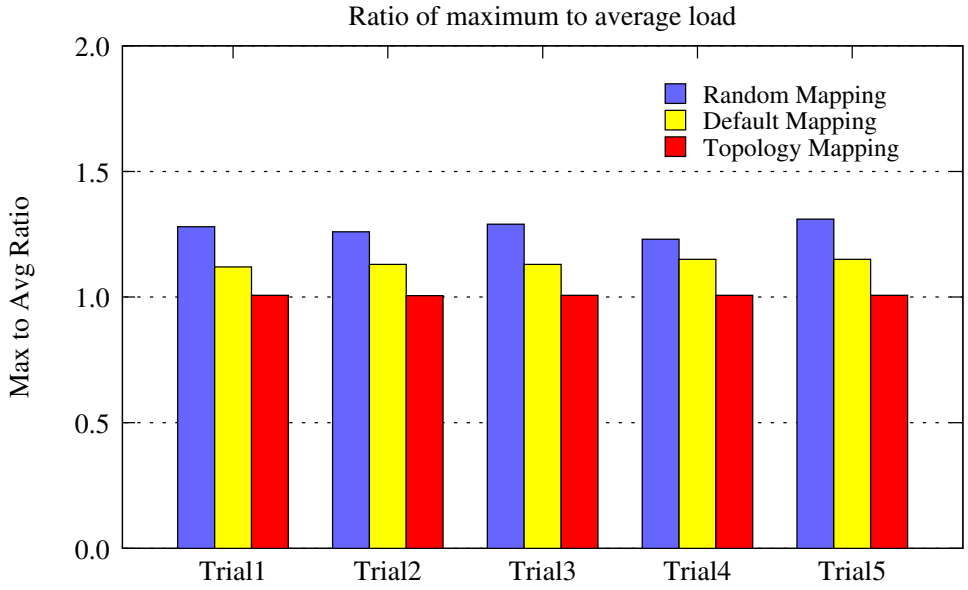


Figure 12.4: Performance of the distributing load balancers in terms of bringing the maximum load on any processor closer to the average

Figure 12.4 shows the improvement in the ratio of maximum to average load

using the load balancers describe above. Ideally, the ratio of maximum to average load should be 1 denoting perfect load balance. We can see in the figure that the load balancers succeed in bringing the value very close to 1. In summary, for very large problems, distributed algorithms can be fast, use much less memory than centralized algorithms and avoid communication bottlenecks at a single processor. Hence, they will be indispensable when running applications on large allocations of petascale and exascale machines.

13 Conclusion and Future Work

The research conducted as a part of this dissertation has reestablished the importance of topology aware mapping in improving application performance and scaling for current day supercomputers. Our MPI benchmarks show that in spite of wormhole routing, contention can severely affect message latencies. Machines with high interconnect bandwidth such as Cray XTs are not spared by this. We have demonstrated performance improvements of up to two times for production scientific codes such as OpenAtom and NAMD running on the largest contemporary machines.

This dissertation makes several important contributions. We claim, aided by micro-benchmark results that *hop-bytes* is a good evaluation metric for mapping algorithms. *Maximum dilation*, which was considered as the more important metric in VLSI design and early parallel computing work, is not as important on parallel machines though it can still impact performance in certain cases.

The dissertation presents a framework for automatic mapping of parallel applications with regular and irregular communication graphs on parallel machines. The two steps involved in automating the process are: 1. obtaining the communication graph and identifying regular communication patterns, 2. intelligent and fast heuristic solutions for mapping applications based on their communication graphs. A suite of mapping heuristics has been developed for regular and irregular communication graphs. We compare different algorithms for their running time and evaluate their performance based on the hop-bytes metric. Using this mapping framework, we demonstrate performance improvements for synthetic benchmarks and scientific applications. This automatic mapping framework will save much effort on the part

of application developers to generate mappings for their individual applications.

We also discuss the benefits of virtualization in general and that of the Charm++ runtime in particular, in handling and mitigating contention and facilitating the process of mapping. We present scalable distributed load balancing techniques which will be required for future petascale and exascale machines. We show that these algorithms can run faster, reduce memory requirements and minimize communication bottlenecks on one processor.

There are several directions for future research based on this work. The automatic mapping framework can be enhanced with more sophisticated algorithms for pattern matching and topology aware mapping. The pattern matching algorithms only identify communication with a small number of neighbors. Other cases for consideration are complex communication patterns such as many-to-many (FFT being one example) and multicasts. The mapping framework does not handle simultaneous multiple communication patterns in an application (such as in OpenAtom.) For MPI applications, the framework does not handle changes in the communication graph between runs and with time. Handling such cases requires more support from the MPI runtime which might happen in the future. We need runtime instrumentation and migration of ranks in an MPI application to handle changing communication graphs.

Obtaining tighter lower bounds for a clearer picture of the optimality of the mapping algorithms is another direction for research. A simulation framework such as BigSim [110] would be a useful tool for this. We can vary message latencies and contention in a simulation to see the impact and benefit from topology aware mapping strategies. Comparing with other algorithms in the literature (both past and recent) would also give a better understanding of the efficiency of these algorithms.

The work presented on distributed load balancing is in its preliminary stages. We chose two specific communication scenarios and implemented strategies for them.

For this work to be generally useful, algorithms for other regular patterns and irregular communication patterns would have to be developed (which would be more challenging.)

References

[1] C. Catlett and *et al.* TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In Lucio Grandinetti, editor, *HPC and Grids in Action*, Amsterdam, 2007. IOS Press.

[2] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator, 1996.

[3] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30:1203–1233, 1999.

[4] Cray Inc. Scalable Computing at Work: Cray XT4 Datasheet. `www.cray.com/downloads/Cray\_XT4\_Datasheet.pdf`, 2006.

[5] Cray Inc. Cray XT Specifications. `http://www.cray.com/Products/XT/Specifications.aspx`, 2009.

[6] N. R. Adiga, G. Almasi, , Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, and A. A. Bright. An Overview of the Blue Gene/L Supercomputer. In *Supercomputing 2002 Technical Papers*, Baltimore, Maryland, 2002. The Blue Gene/L Team, IBM and Lawrence Livermore National Laboratory.

[7] IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), 2008.

[8] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[9] James W. Dolter, P. Ramanathan, and Kang G. Shin. Performance analysis of virtual cut-through switching in harts: A hexagonal mesh multicomputer. *IEEE Trans. Comput.*, 40(6):669–680, 1991.

[10] Dilip D. Kandlur and Kang G. Shin. Traffic Routing for Multicomputer Networks with Virtual Cut-Through Capability. *IEEE Trans. Comput.*, 41(10):1257–1270, 1992.

[11] Ronald I. Greenberg and Hyeong-Cheol Oh. Universal wormhole routing. *IEEE Transactions on Parallel and Distributed Systems*, 08(3):254–262, 1997.

[**12**] Prasant Mohapatra. Wormhole routing techniques for directly connected multicomputer systems. *ACM Comput. Surv.*, 30(3):374–410, 1998.

[**13**] Loren Schwiebert and D. N. Jayasimha. On measuring the performance of adaptive wormhole routing. *hipc*, 00:336, 1997.

[**14**] Mohamed Ould-Khaoua and Hamid Sarbazi-Azad. An analytical model of adaptive wormhole routing in hypercubes in the presence of hot spot traffic. *IEEE Transactions on Parallel and Distributed Systems*, 12(3):283–292, 2001.

[**15**] 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.

[**16**] 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.

[**17**] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[**18**] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[**19**] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[**20**] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[**21**] Steve Otto Marc Snir and etc. *MPI: The Complete Reference*, volume 1. The MIT Press.

[**22**] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[**23**] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235, September 2006.

[**24**] Michalakes, J., J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The Weather Research and Forecast Model: Software Architecture and Performance. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, October 2004.

[**25**] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers. A description of the advanced research wrf version 2. Technical Report Technical Note NCAR/TN-468+STR, June 2005.

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

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

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

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

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

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

[**32**] 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.

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

[**34**] 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.

[**35**] M. Muller and Michael Resch. PE mapping and the congestion problem in the T3E. In *Proceedings of the Fourth European Cray-SGI MPP Workshop*, Garching, Germany, 1998.

[**36**] Eduardo Huedo and Manuel Prieto and Ignacio Martín Llorente and Francisco Tirado. Impact of PE Mapping on Cray T3E Message-Passing Performance. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 199–207, London, UK, 2000. Springer-Verlag.

[**37**] 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.

[**38**] Deborah Weisser, Nick Nystrom, Chad Vizino, Shawn T. Brown, and John Urbanic. Optimizing Job Placement on the Cray XT3. *48th Cray User Group Proceedings*, 2006.

[**39**] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2/3), 2005.

[**40**] G. Almasi, C. Archer, J. G. Castanos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3), 2005.

[**41**] IBM System Blue Gene Solution. Blue Gene/P Application Development Redbook. `http://www.redbooks.ibm.com/abstracts/sg247287.html`, 2008.

[**42**] George Almasi and Siddhartha Chatterjee and Alan Gara and John Gunnels and Manish Gupta and Amy Henning and Jose E. Moreira and Bob Walkup. Unlocking the Performance of the Blue Gene/L Supercomputer. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 57. IEEE Computer Society, 2004.

[**43**] Kei Davis and Adolfy Hoisie and Greg Johnson and Darren J. Kerbyson and Mike Lang and Scott Pakin and Fabrizio Petrini. A Performance and Scalability Analysis of the Blue Gene/L Architecture. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 41. IEEE Computer Society, 2004.

[**44**] Francois Gygi, Erik W. Draeger, Martin Schulz, Bronis R. De Supinski, John A. Gunnels, Vernon Austel, James C. Sexton, Franz Franchetti, Stefan Kral, Christoph Ueberhuber, and Juergen 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.

[**45**] 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.

[**46**] 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.

[**47**] Hao Yu, I-Hsin Chung, and Jose Moreira. Topology mapping for Blue Gene/L supercomputer. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 116, New York, NY, USA, 2006. ACM.

[**48**] Tarun Agarwal, Amit Sharma, and Laxmikant 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.

[**49**] Blake G. Fitch, Aleksandr Rayshubskiy, Maria Eleftheriou, T. J. Christopher Ward, Mark Giampapa, and Michael C. Pitman. Blue matter: Approaching the limits of concurrency for classical molecular dynamics. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM Press.

[**50**] Leonid Oliker, Andrew Canning, Jonathan Carter, Costin Iancu, Michael Lijewski, Shoaib Kamil, John Shalf, Hongzhang Shan, Erich Strohmaier, Stephane Ethier, and Tom Goodale. Scientific Application Performance on Candidate PetaScale Platforms. In *Proceedings of IEEE Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[**51**] Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé. Optimizing communication for Charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*, 2010.

[**52**] 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.

[**53**] Aleliunas, R. and Rosenberg, A. L. On Embedding Rectangular Grids in Square Grids. *IEEE Trans. Comput.*, 31(9):907–913, 1982.

[**54**] Ellis, J.A. Embedding rectangular grids into square grids. *Computers, IEEE Transactions on*, 40(1):46–52, Jan 1991.

[**55**] Melhem, Rami G. and Hwang, Ghil-Young. Embedding Rectangular Grids into Square Grids with Dilation Two. *IEEE Trans. Comput.*, 39(12):1446–1455, 1990.

[**56**] Jack Dongarra and P Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical Report UT-CS-05-544, University of Tennessee, Dept. of Computer Science, 2005.

[**57**] C. Walshaw, M. Cross, M. G. Everett, S. Johnson, and K. Mcmanus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980, pages 121–126. Springer, 1995.

[**58**] Kevin McManus, Mark Cross, Chris Walshaw, Steve Johnson, and Peter Leggett. A scalable strategy for the parallelization of multiphysics unstructured mesh-iterative codes on distributed-memory systems. *Int. J. High Perform. Comput. Appl.*, 14(2):137–174, 2000.

[**59**] C.E. Leiserson. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10), October 1985.

[**60**] Top500 supercomputing sites. `http://top500.org`.

[**61**] M.Blumrich, D.Chen, P.Coteus, A.Gara, M.Giampapa, P.Heidelberger, S.Singh, B.Steinmacher-Burow, T.Takken, and P.Vranas. Design and Analysis of the Blue Gene/L Torus Interconnection Network. *IBM Research Report*, December 2003.

[**62**] Jud Leonard, Avi Purkayastha, Matt Reilly, and Tushar Mohan. The software interface for a cluster interconnect based on the kautz digraph. *Cluster Computing, IEEE International Conference on*, 0:187–193, 2007.

[**63**] Nitin Godiwala, Jud Leonard, and Matthew Reilly. A network fabric for scalable multiprocessor systems. *High-Performance Interconnects, Symposium on*, 0:137–144, 2008.

[**64**] 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.

[**65**] V. Salapura, K. Ganesan, A. Gara, M. Gschwind, J.C. Sexton, and R.E. Walkup. Next-Generation Performance Counters: Towards Monitoring Over Thousand Concurrent Events. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 139 – 146, April 2008.

[**66**] M. E. Tuckerman. Ab initio molecular dynamics: Basic concepts, current trends and novel applications. *J. Phys. Condensed Matter*, 14:R1297, 2002.

[**67**] Pasquarello A, Hybertsen MS, and Car R. Interface structure between silicon and its oxide by first-principles molecular dynamics. *Nature*, 396:58, 1998.

[**68**] De Santis L and Carloni P. Serine proteases: An ab initio molecular dynamics study. *Proteins*, 37:611, 1999.

[**69**] A. M. Saitta, P. D. Soper, E. Wasserman, and M. L. Klein. Influence of a knot on the strength of a polymer strand. *Nature*, 399:46, 1999.

[**70**] Rothlisberger U, Carloni P, Doclo K, and Parinello M. A comparative study of galactose oxidase and active site analogs based on QM/MM Car Parrinello simulations. *J. Biol. Inorg. Chem.*, 5:236, 2000.

[**71**] Haye MJ, Massobrio C, Pasquarello A, and Car R. Structure of liquid GexSe1-x at the stiffness threshold composition. *Phys. Rev. B*, 58:R14661, 1998.

[**72**] Blase X, Charlier JC, De Vita A, and Car R. Structural and electronic properties of composite BxCyNz nanotubes and heterojunctions. *Appl. Phys. A*, 68:293, 1999.

[**73**] V. Musolino, A. Selloni, and R. Car. Structure and dynamics of small metallic clusters on an insulating metal-oxide surface: Copper on MgO(100). *Phys. Rev. Lett.*, 83:3242, 1999.

[**74**] J. J. Mortensen and M. Parrinello. A density functional theory study of a silica-supported zirconium monohydride catalyst for depolymerization of polyethylene. *J Phys. Chem. B*, 104:2901, 2000.

[**75**] A. Pasquarello, I. Petri, P. S. Salmon, O. Parisel, R. Car, E. Toth, D. H. Powell, H. E. Fischer, L. Heim, and A. E. Merbach. First solvation shell of the Cu(II) aqua ion: Evidence for fivefold coordination. *Science*, 291:856, 2001.

[**76**] M. Boero, M. Parrinello, S. Huffer, and H. Weiss. First principles study of propene polymerization by ziegler-natta hetergeneous catalysis. *J. Am. Chem. Soc.*, 122:501, 2000.

[**77**] Kaupp M, Rovira C, and Parrinello M. Density functional study of O-17 NMR chemical shift and nuclear quadrupole coupling tensors in oxyheme model complexes. *J Phys. Chem. B*, 104:5200, 2000.

[**78**] A.D. Becke. Density-functional exchange-energy approximation with correct assymptotic behavior. *Phys. Rev. A*, 38(6):3098–3100, (1988).

[**79**] C. Lee, W. Yang, and R.G. Parr. Development of the Calle-Salvetti correlation energy into a functional of the electron density. *Phys. Rev. B*, 37(2):785–789, (1988).

[**80**] N. Troullier and J.L. Martins. Efficient pseudopotentials for plane wave calculations. *Phys. Rev. B*, 43(3):1993–2006, (1991).

[**81**] Klaus Schulten, James C. Phillips, Laxmikant V. Kale, and Abhinav Bhatele. Biomolecular modeling in the era of petascale computing. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 165–181. Chapman & Hall / CRC Press, 2008.

[**82**] L. V. Kalé, Milind Bhandarkar, and Robert Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 251–261, 1998.

[**83**] Abhinav Bhatelé and Laxmikant V. Kalé. Quantifying Network Contention on Large Parallel Machines. *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)*, 19(4):553–572, 2009.

[**84**] 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.

[**85**] Sameer Kumar, Chao Huang, Gengbin Zheng, Eric Bohm, Abhinav Bhatele, James C. Phillips, Hao Yu, and Laxmikant V. Kalé. Scalable Molecular Dynamics with NAMD on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):177–187, 2008.

[**86**] H. Wen and S. Sbaraglia and S. Seelam and I. Chung and G. Cong and D. Klepacki. A Productivity Centered Tools Framework for Application Performance Tuning. In *QEST '07: Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems*, pages 273–274, Washington, DC, USA, 2007. IEEE Computer Society.

[**87**] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Detecting Patterns in MPI Communication Traces. *Parallel Processing, International Conference on*, 0:230–237, 2008.

[**88**] Darren J. Kerbyson and Kevin J. Barker. Automatic Identification of Application Communication Patterns via Templates. In *In Proc. Int. Conf. Parallel and Distributed Computing Systems (PDCS)*, Las Vegas, NV, August 2005.

[**89**] Nikhil Bhatia, Fengguang Song, Felix Wolf, Jack Dongarra, Bernd Mohr, and Shirley Moore. Automatic experimental analysis of communication patterns in virtual topologies. *Parallel Processing, International Conference on*, 0:465–472, 2005.

[**90**] MILC Collaboration. MIMD Lattice Computation (MILC) Collaboration Home Page. `http://www.physics.indiana.edu/~sg/milc.html`.

[**91**] Claude Bernard, Tom Burch, Thomas A. DeGrand, Carleton DeTar, Steven Gottlieb, Urs M. Heller, James E. Hetrick, Kostas Orginos, Bob Sugar, and Doug Toussaint. Scaling tests of the improved Kogut-Susskind quark action. *Physical Review D*, (61), 2000.

[**92**] J. K. Dukowicz and R. D. Smith. Implicit free-surface method for the Bryan-Cox-Semtner ocean model. *Journal of Geophysics Research*, 99:7991–8014, April 1994.

[**93**] John K. Dukowicz, Richard D. Smith, and Robert C. Malone. A Reformulation and Implementation of the Bryan-Cox-Semtner Ocean Model on the Connection Machine. *Journal of Atmospheric and Oceanic Technology*, 10(2):195–208, April 1993.

[**94**] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. In *Proceedings of the 11th annual international conference of the Center for Nonlinear Studies on Experimental mathematics : computational issues in nonlinear science*, pages 38–61. Elsevier North-Holland, Inc., 1992.

[**95**] Kirk Bryan. A numerical method for the study of the circulation of the world ocean. *Journal of Computational Physics*, 135(2):154–169, 1997.

[**96**] The weather research & forecasting model website. `http://wrf-model.org`.

[**97**] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, 1890.

[**98**] D. Hilbert. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[**99**] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, 1996.

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

[**101**] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21:1129–1164, March 1991.

[**102**] 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.

[**103**] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[**104**] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[**105**] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, San Diego, California, USA, September 2010.

[**106**] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load balancing policies for dynamic applications. In *IEEE Concurrency*, pages 7(1):22–31, 1999.

[**107**] Anna Ha'c and Xiaowei Jin. Dynamic load balancing in distributed system using a decentralized algorithm. In *Proc. of 7-th Intl. Conf. on Distributed Computing Systems*, April 1987.

[**108**] L. V. Kalé. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, St. Charles, IL, August 1988.

[**109**] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, September 1993.

[**110**] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.

Appendix A

In this appendix, we present figures which depict the mapping of different object grids to two-dimensional processor meshes using the various mapping algorithms for regular graphs presented in Chapter 9.

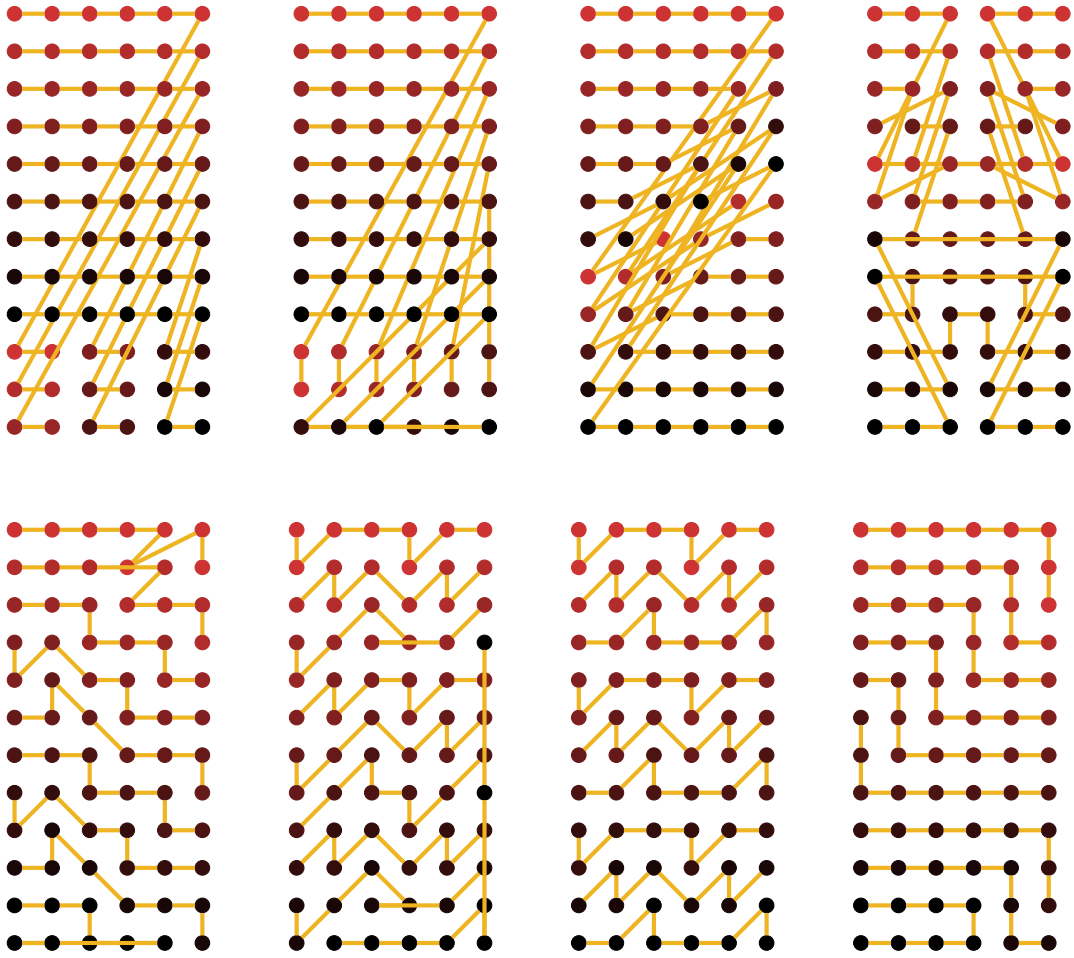


Figure A.1: Mapping of a 9×8 grid to a 12×6 mesh using `MXOVLP`, `MXOV+AL`, `EXCO`, `COCE`, `AFFN` (and two other variations of it), and `STEP` respectively

Figure A.1 shows the mapping of an object grid of dimensions 9×8 to a processor mesh of dimensions 12×6 for the various algorithms. Figure A.2 shows the mappings for the same grid with vertical edges (in green).

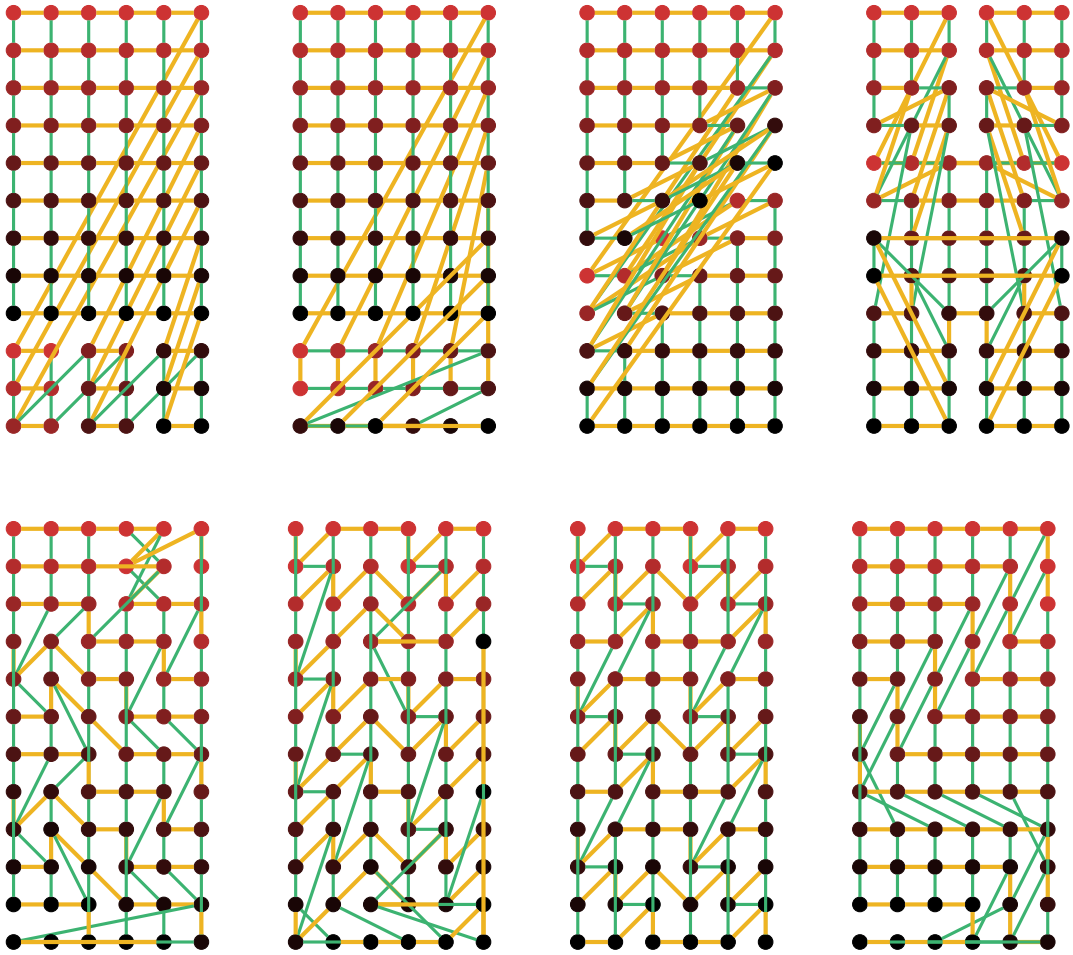


Figure A.2: Mapping of a 9×8 grid to a 12×6 mesh using `MXOVLP`, `MXOV+AL`, `EXCO`, `COCE`, `AFFN` (and two other variations of it), and `STEP` respectively (vertical edges also shown)

The mapping figures presented above should help in understanding of the mapping algorithms presented earlier and give some idea of the average hop-bytes and maximum dilation for each. Figures A.3 and A.4 show the mappings for an object grid of dimensions 14×6 to a processor mesh of dimensions 7×12 for the various algorithms.

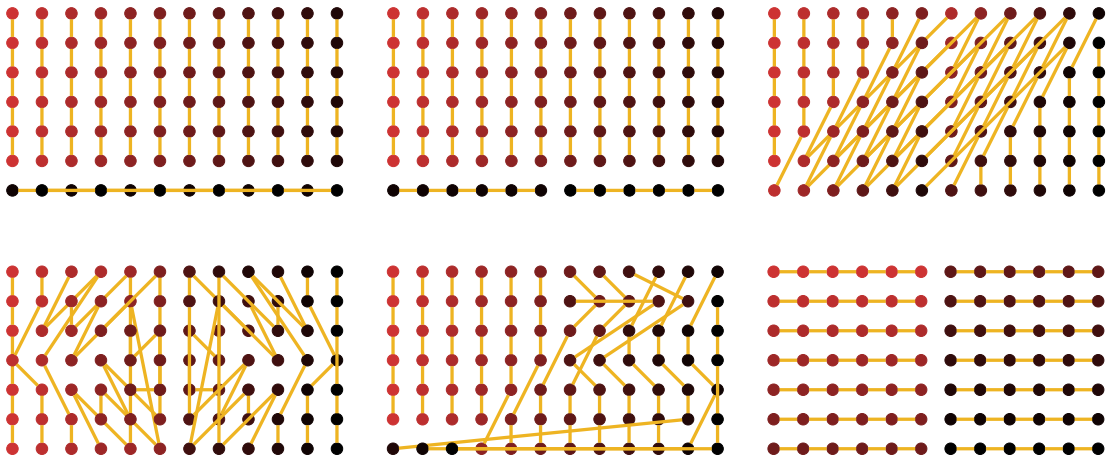


Figure A.3: Mapping of a 14×6 grid to a 7×12 mesh using `MXOVLP`, `MXOV+AL`, `EXCO`, `COCE`, `AFFN` and `STEP` respectively

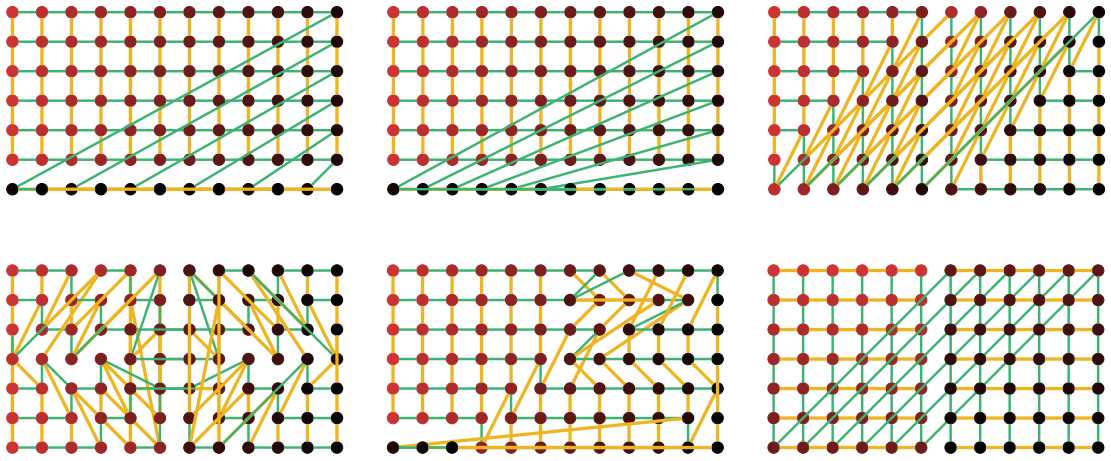


Figure A.4: Mapping of a 14×6 grid to a 7×12 mesh using `MXOVLP`, `MXOV+AL`, `EXCO`, `COCE`, `AFFN` and `STEP` respectively (vertical edges also shown)

Appendix B

The first part of this appendix contains pseudo code and detailed implementations of the spiraling and quadtree implementations for `findNearest` function. This function was introduced in Chapter 9 and discussed in some detail in Chapter 10.

Algorithm B.1 Finding the nearest available processor in 3D

```
procedure findNearest3D(p_x, p_y, p_z, freeProcs)
    diameter1 = proc_x + proc_y + proc_z − 3
    if isAvailable3D(p_x, p_y, p_z, freeProcs) then
        return < p_x, p_y, p_z >
    end if
    for d1 := 1 to diameter1 do
        for i := (−1) × d1 to d1 do
            diameter2 = d1−abs(i) −1
            for d2 := 0 to diameter2 do
                for j := (−1) × d2 to d2 do
                    k = d2−abs(j)
                    r_x = p_x + i
                    r_y = p_y + j
                    r_z = p_z + k
                    if withinBounds3D(r_x, r_y, r_z) && isAvailable(r_x, r_y, r_z, freeProcs)
then
                        return < r_x, r_y, r_z >
                    end if
                    r_x = p_x + i
                    r_y = p_y + j
                    r_z = p_z − k
                    if withinBounds3D(r_x, r_y, r_z) && isAvailable(r_x, r_y, r_z, freeProcs)
then
                        return < r_x, r_y, r_z >
                    end if
                end for
            end for
        end for
    end for
end procedure
```

The pseudo code above (Algorithm B.1) depicts the algorithm for finding the nearest available processor in a 3D mesh. The algorithm for doing the same in 2D was presented in Chapter 10.

```
64  int findNearest2D(int px, int py, int dimX, int dimY,
65                            int **freeprocs, int* rx, int* ry) {
66    int i, j, d, from, to;
67    int diameter = dimX + dimY - 1;
68    if( isAvailable2D(px, py, freeprocs) ) {
69      *rx = px; *ry = py;
70      return 1;
71    }
72
73    for(d=1; d<diameter; d++) {
74      // choose the x-coordinate first
75      if(d > px) from = (-1)*px; else from = (-1)*d;
76      if(d > dimX-1-px) to = dimX-1-px; else to = d;
77      for(i=from; i<=to; i++) {
78        // y-coordinate has two possible choices based on the x
79        j = d - abs(i);
80
81        *rx = px + i;
82        *ry = py + j;
83        if( withinBounds2D(*rx, *ry, dimX, dimY) &&
84            isAvailable2D(*rx, *ry, freeprocs) ) return 1;
85
86        *rx = px + i;
87        *ry = py - j;
88        if( withinBounds2D(*rx, *ry, dimX, dimY) &&
89            isAvailable2D(*rx, *ry, freeprocs) ) return 1;
90      }
91    }
92    *rx = -1; *ry = -1;
93    return -1;
94  }
```

Figure B.1: C implementation of the `findNearest2D` function

Figure B.1 shows the C code for the spiraling implementation of the pseudo code (see Algorithm 10.1 in Chapter 10) for `findNearest2D`. This code has optimizations

to avoid unnecessary spiraling and hence is different in some ways from the pseudo code presented earlier. When choosing the X coordinate, we carefully avoid going out of bounds of the processor mesh. This saves much time spent in spiraling.

Figure B.2 shows the C implementations of two helper functions used by the spiraling implementation of `findNearest2D`. The function `withinBounds2D` is used to determine if a certain coordinate falls within the processor mesh and is a valid processor or not. The function `isAvailable` returns whether a certain processor is vacant.

```
20 inline int isAvailable2D(int x, int y, int **freeprocs) {
21   if(freeprocs[y][x] < 0)
22     return 1;
23   return 0;
24 }
25
26 inline int withinBounds2D(int x, int y, int dimX, int dimY) {
27   if(x >= 0 && x < dimX && y >= 0 && y < dimY)
28     return 1;
29   return 0;
30 }
```

Figure B.2: C implementations of the `withinBounds2D` and `isAvailable` helper functions

Chapter 10 also discussed a quadtree implementation for finding the nearest available processor. Figure B.3 shows the `QuadTree` class and Figure B.4 shows the implementation of the `findNearest` function using a quadtree. Each node stores the extent of the subgrid it controls and the number of available processors in the subtree below it at any given time. The `findNearest` function uses two helper functions: `withinSubTree` to determine if a given processor is within its subtree and `hopsToSubTree` which returns the smallest distance from a given processor to any processor in the subgrid.

Figure B.5 shows a quadtree for a processor mesh of dimensions 8×32 and the

```
class QuadTree {
  public:
    QuadTree();
    QuadTree(int _ox, int _oy, int _bx, int _by, QuadTree *_parent);
    ~QuadTree();

    inline int getOrigX()           { return origX; }
    inline int getOrigY()           { return origY; }
    inline int getBoundX()          { return boundX; }
    inline int getBoundY()          { return boundY; }
    inline QuadTree* getParent()    { return parent; }
    inline int numAvailable()       { return numFreeNodes; }

    bool withinSubTree(int x, int y);
    int hopsToSubTree(int x, int y);
    void findNearest(int x, int y);

  private:
    int origX;          // x coordinate of lower left
    int origY;          // y coordinate of lower left
    int boundX;         // extent of node in x
    int boundY;         // extent of node in y
    QuadTree **child;   // leaf nodes will have this as NULL
    QuadTree *parent;   // root node will have this as NULL
    int numFreeNodes;   // number of free processors
};
```

Figure B.3: The `QuadTree` class: member variables and function declarations

number of free processors at each node initially. The next set of figures present the mapping of yet another irregular graph of 128 nodes (Figure B.6). Figures B.7 to B.9 present the mapping of this mesh using the various algorithms for irregular graphs. The values for hops per byte are also presented.

We compared the execution time for the spiraling and quadtree implementations of `findNearest` for the mesh presented in Figure B.6 also. Similar to previous results, we see significant performance improvements in execution time when using the quadtree implementation (Figures B.10 and B.11).

```
197 void QuadTree::findNearest(int x, int y) {
198   int hops;
199   bool withinSubtree;
200   QuadTree *ret, *chret;
201
202   if(child == NULL && numFreeNodes > 0) {
203     hops = abs(x-origX) + abs(y-origY);
204     if(hops < bestHops) {
205       bestHops = hops;
206       bestNode = this;
207     }
208   } else {
209     // look at all my non-NULL children if they do not contain the
210     // node within them and if they have some free processors and if
211     // their hops might be better than the current
212     if(numFreeNodes > 0) {
213       if( (child[0] != NULL) && !(child[0]->withinSubTree(x, y))
214                     && (child[0]->hopsToSubTree(x, y) < bestHops) )
215         child[0]->findNearest(x, y);
216
217       if( (child[1] != NULL) && !(child[1]->withinSubTree(x, y))
218                     && (child[1]->hopsToSubTree(x, y) < bestHops) )
219         child[1]->findNearest(x, y);
220
221       if( (child[2] != NULL) && !(child[2]->withinSubTree(x, y))
222                     && (child[2]->hopsToSubTree(x, y) < bestHops) )
223         child[2]->findNearest(x, y);
224
225       if( (child[3] != NULL) && !(child[3]->withinSubTree(x, y))
226                     && (child[3]->hopsToSubTree(x, y) < bestHops) )
227         child[3]->findNearest(x, y);
228     }
229     if(parent != NULL && withinSubTree(x, y))
230       parent->findNearest(x, y);
231   }
232 }
```

Figure B.4: Implementation of the `findNearest` function using a quadtree

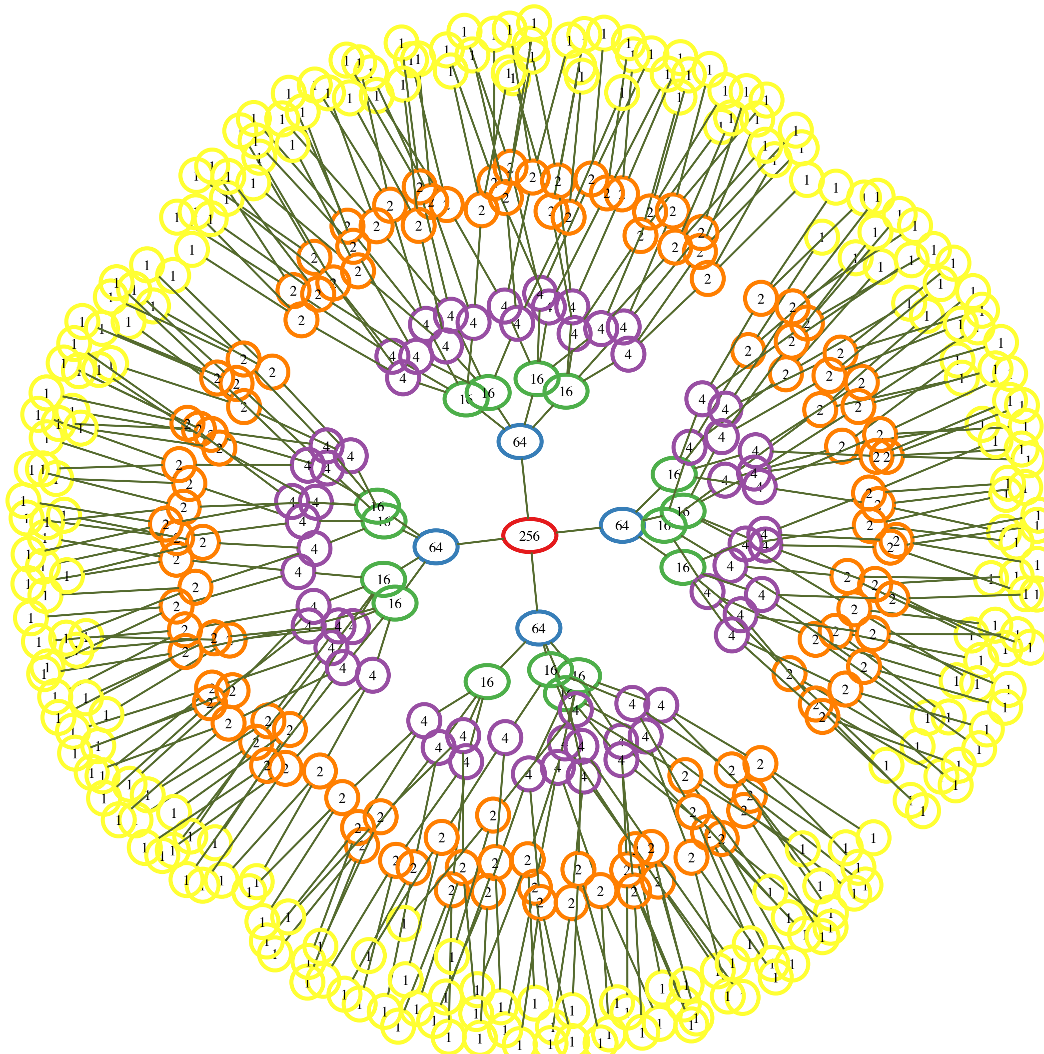


Figure B.5: Representation of a 2D mesh of processors of dimensions 8×32 as a quadtree

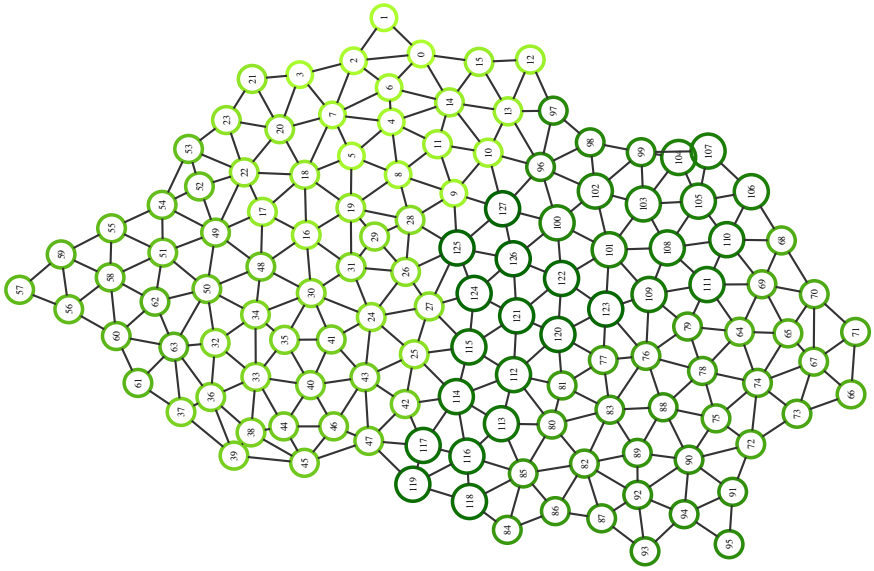
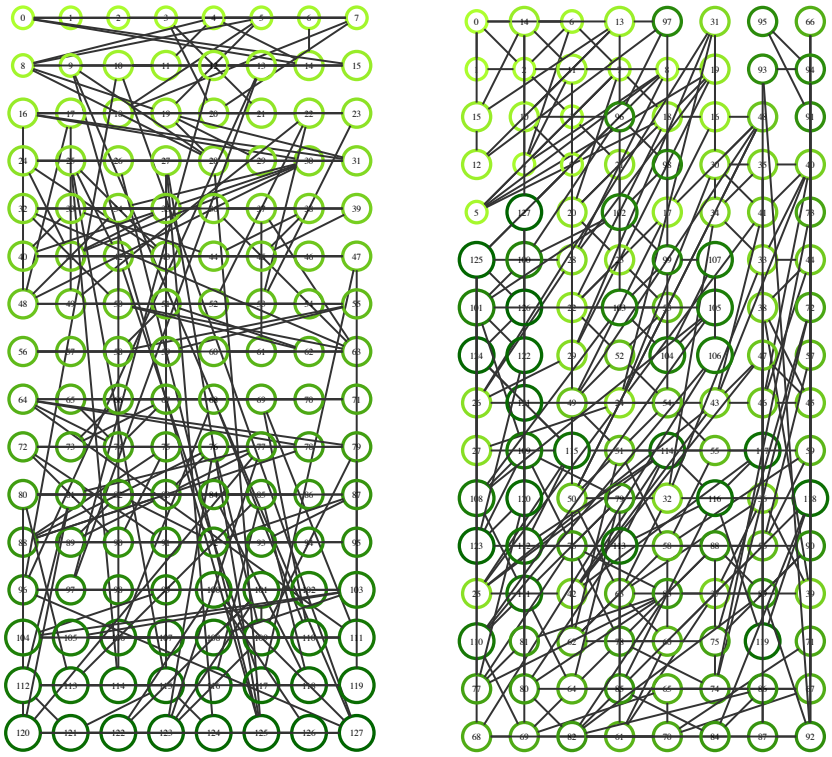


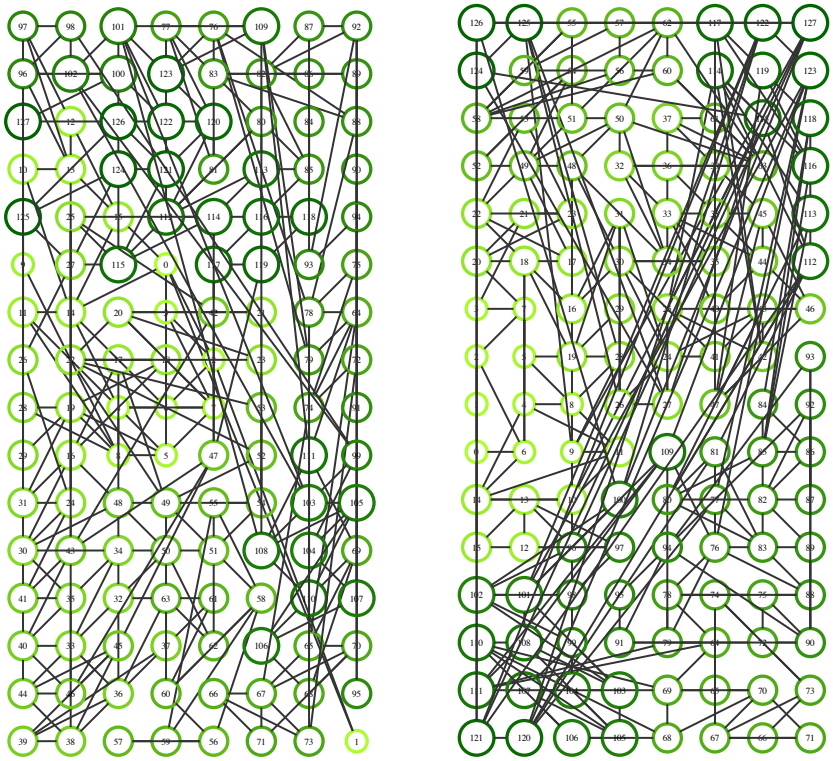
Figure B.6: Irregular graph with 128 nodes



(a) Default Mapping, Hops per byte: 3.22

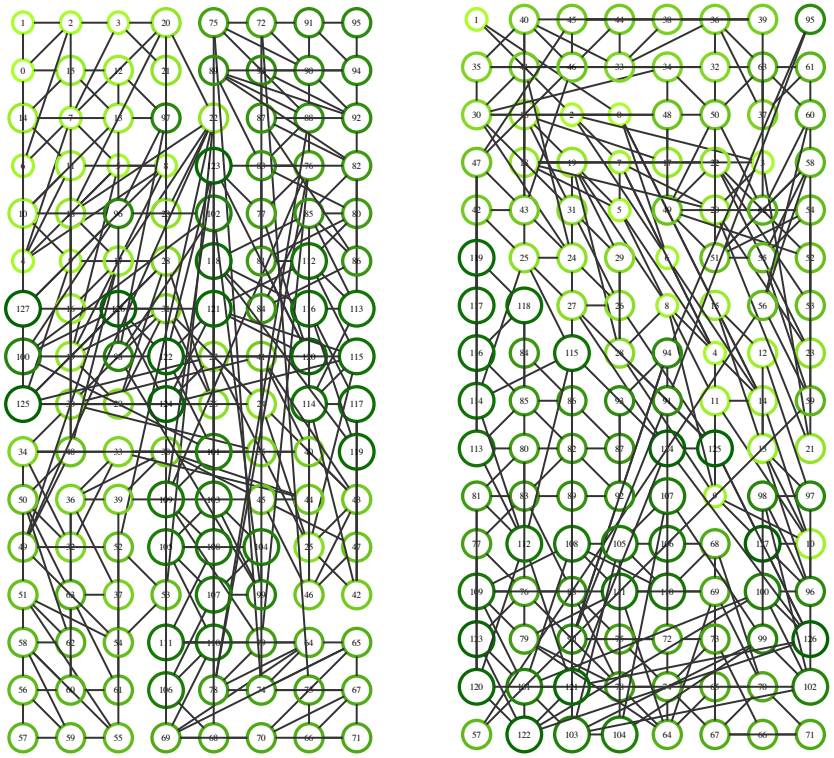
(b) `BFT`, Hops per byte: 3.53

Figure B.7: Mapping of an irregular graph with 128 nodes using the (a) Default mapping, and (b) `BFT` algorithm to a grid of dimensions 16×8



(a) `MHT`, Hops per byte: 2.83 (b) `AFFN`, Hops per byte: 3.02

Figure B.8: Mapping of an irregular graph with 128 nodes using the (a) `MHT`, and (b) `AFFN` algorithm to a grid of dimensions 16×8



(a) `COCE`, Hops per byte: 2.90 (b) `COCE+MHT`, Hops per byte: 2.73

Figure B.9: Mapping of an irregular graph with 128 nodes using the (a) `COCE`, and (b) `COCE+MHT` algorithm to a grid of dimensions 16×8

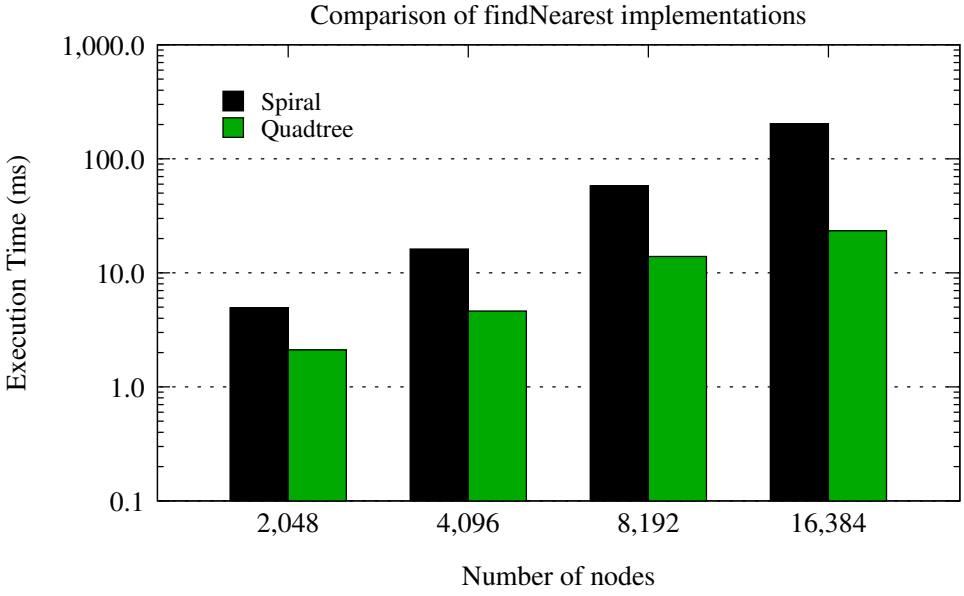


Figure B.10: Comparison of execution time for spiraling and quadtree implementations when invoked from the `AFFN` mapping algorithm

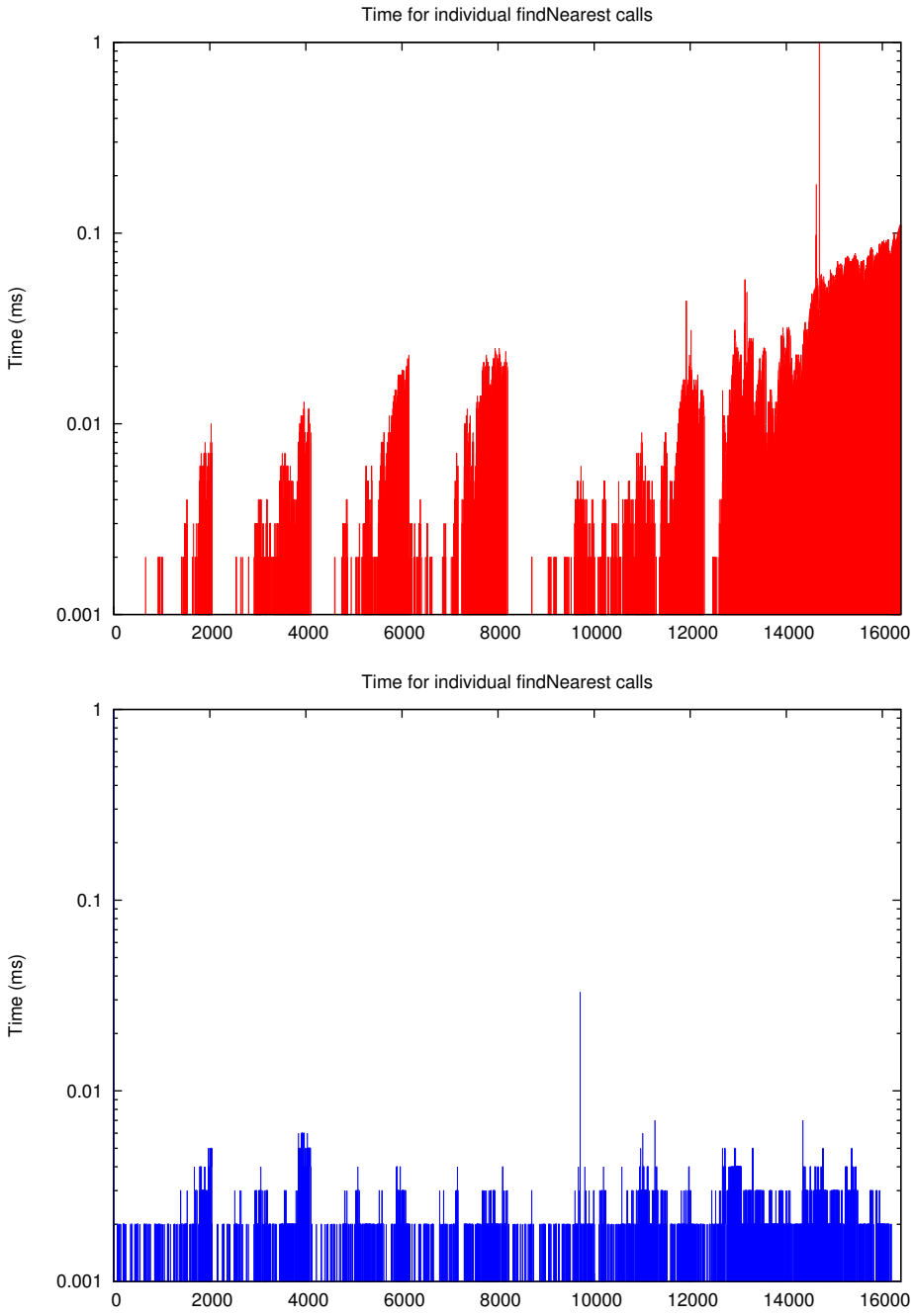


Figure B.11: Execution time for 16,384 calls to the spiraling (top) and quadtree (bottom) implementations for `findNearest` for the `AFFN` algorithm for irregular graphs

Vita

Office Address:
4103, Siebel Center for Computer Science
201, North Goodwin Avenue, MC-258
Urbana, IL 61801-2302
url: `www.bhatele.org`

Residential Address:
505, East White Street
Apartment 06
Champaign, IL 61820-1810
email: `bhatele@illinois.edu`

Abhinav Bhatelé

Educational Qualifications

University of Illinois at Urbana Champaign
Doctor of Philosophy in Computer Science August 2010
Dissertation Title: Automating Topology Aware Mapping for Supercomputers
Dissertation Adviser: Laxmikant V. Kalé
Master of Science in Computer Science December 2007
Thesis Title: Application-specific Topology-aware Mapping and Load Balancing for three dimensional torus topologies
Thesis Adviser: Laxmikant V. Kalé

Indian Institute of Technology, Kanpur
Bachelor of Technology in Computer Science and Engineering May 2005
Thesis Title: Compiler Algorithm Language (CAL): An Interpreter and Compiler
Thesis Adviser: Sanjeev K. Aggarwal

Research Experience

Research Assistant, Parallel Programming Lab, Illinois 2005-2010

- Adviser: Laxmikant V. Kalé
- Topology Aware Mapping: Working towards building on an automated topology-aware mapping framework (dissertation proposal)
- Load Balancing for MD Applications: Implemented topology-aware load balancers for a production MD application called NAMD
- Static Topology-aware Mapping: Designed and implemented mapping strategies for a chemistry application, OpenAtom

Intern, IBM T J Watson Research Center, NY, USA May-July 2007

- Worked in the Blue Gene Software Group. Devised topology sensitive load balancing algorithms to be applied to Charm++ programs in general and NAMD.

Intern, IBM T J Watson Research Center, NY, USA May-July 2006
- Worked at the Advanced Computing Technology Center. Developed a tool for automatic and detailed profiling of programs at finer levels.

Summer Intern, INRIA Labs, Nancy, France May-July 2004
- Worked with Prof. Stephan Merz in the MOSEL group. Developed a GUI for a model checker, TLC and a theorem prover, Xprove.

Teaching Experience

Instructor, Computer Architecture I (CS231), Department of Computer Science, Illinois, Summer 2008 and 2009
- Full responsibility for the course, worked with a TA
- Prepared and gave lectures, awarded final grades
- Used i>clickers to enhance pedagogical technique for the first time in a computer science course at Illinois

Instructor, Data Structures and Algorithms, Summer course at IIT Kanpur, 2005
- Prepared and gave lectures, created and graded homeworks and exams

Selected Publications

1. Abhinav Bhatele, Eric Bohm and Laxmikant V. Kale, Optimizing communication for Charm++ applications by reducing network contention, *accepted for publication in Concurrency and Computation: Practice and Experience (EuroPar special issue)*, 2010

2. Abhinav Bhatele, Lukasz Wesolowski, Eric Bohm, Edgar Solomonik and Laxmikant V. Kale, Understanding application performance via micro-benchmarks on three large supercomputers: Intrepid, Ranger and Jaguar, *International Journal of High Performance Computing Applications (IJHPCA)*, 2010 *url*: `http://hpc.sagepub.com/cgi/content/abstract/1094342010370603v1`

3. Abhinav Bhatele and Laxmikant V. Kale, Quantifying Network Contention on Large Parallel Machines, *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing), Vol. 19 Issue 4, Pages 553-572*, 2009

4. Abhinav Bhatele, Eric Bohm and Laxmikant V. Kale, A Case Study of Communication Optimizations on 3D Mesh Interconnects, *Proceedings of Euro-Par (Topic 13 - High Performance Networks)*, 2009

5. Abhinav Bhatele, Laxmikant V. Kale and Sameer Kumar. Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications, *Proceedings of 23rd ACM International Conference on Supercomputing*, 2009

6. Abhinav Bhatele, Laxmikant V. Kale, An Evaluative study on the Effect of Contention on Message Latencies in Large Supercomputers, *Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS)*, 2009

7. Abhinav Bhatele, Laxmikant V. Kale, Benefits of Topology-aware Mapping for Mesh Topologies, Parallel Processing Letters *(Special issue on Large Scale Parallel Processing), Vol. 18, Issue 4, Pages 549-566*, 2008

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

9. Abhinav Bhatele, Laxmikant V. Kale, Application-specific Topology-aware Mapping for Three Dimensional Topologies, *Proceedings of Workshop on Large-Scale Parallel Processing (held as part of IPDPS '08)*, 2008
10. Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, Glenn Martyna, Fine grained parallelization of the Car-Parrinello ab initio MD method on Blue Gene/L, *IBM J. Res. Dev., Volume 52, No. 1/2*, 2007
11. Sameer Kumar, Chao Huang, Gengbin Zheng, Eric Bohm, Abhinav Bhatele, Jim Phillips, Gheorghe Almasi, Hao Yu, Laxmikant V. Kale, Achieving Strong Scaling with NAMD on Blue Gene/L, *IBM J. Res. Dev., Volume 52, No. 1/2*, 2007

Awards and Achievements

- Feng Chen Memorial **Best Paper Award**, Dept. of CS, Illinois, 2010
- Teacher Scholar Certificate, Center of Teaching Excellence, Illinois, 2010
- ACM/IEEE **George Michael Memorial HPC Fellowship Award**, 2009
- Selected for Doctoral Showcase at Supercomputing Conference (SC), Portland, 2009
- **Distinguished Paper Award**, Euro-Par, Amsterdam, Netherlands, 2009
- **David J. Kuck Outstanding M.S. Thesis Award**, Dept. of CS, Illinois, 2009
- 3rd Prize for **Best Graduate Poster** at the *ACM Student Research Competition* at Supercomputing Conference (SC), Austin, TX, 2008
- Selected for the **TCPP PhD Forum** at IPDPS, Miami, FL, 2008
- Nominated among the six best B. Tech. projects in the Computer Science department, IIT Kanpur, 2005
- Awarded the **Student Benefit Fund Scholarship** for excellent performance in academics in 2002
- Awarded the **Academic Excellence Award** at IIT Kanpur for the year 2001-2002

Professional Service

- Reviewer, IJHPCA 2010
- Mentoring two undergraduate students working for my advisor, 2008-2010
- Mentor for the WCS Mentoring Program, Dept. of CS, Illinois, 2009 and 2010
- Reviewer, ICPP 2009
- Mentor for Undergraduate Research Lab (CS498la), Dept. of CS, Illinois, Spring 2009
- Volunteer for the Grad Recruitment weekends, Dept. of CS, Illinois, 2008 and 2009
- Student Volunteer, Supercomputing, Austin, TX 2008
- Helped in organization of Charm++ Workshops, 2007 and 2008

Relevant Courses

Graduate – Advanced Computer Architecture, Formal Methods of Computation, Parallel Programming Methods, Programming Languages and Compilers, Advanced Topics in Compiler Construction, Social Computing, Improving your Research Skills

Undergraduate – Advanced Compiler Optimizations, Computer Architecture, Compilers, Computer Networks, Operating Systems, Algorithms II, Theory of Computation, Data Structures and Algorithms, Discrete Mathematics

Programming Skills

Languages: Charm++, C, C++, JAVA, MPI, OpenMP, VHDL, Ocaml
Platforms: Most flavors of Windows, MacOS and Linux
Tools: Lex, Yacc, LaTeX, Make, Perl