ABSTRACT

Title of dissertation: PARALLEL COMPUTING WITH P2P DESKTOP GRIDS

Gary Lee Jackson II, Doctor of Philosophy, 2015

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

Tightly-coupled parallel computing is an important tool for problem solving. Structured peer-to-peer network overlays are failure-tolerant and have a low administrative burden. This work seeks to unite the two.

First, I present a completely decentralized algorithm for parallel job scheduling and load balancing in distributed peer-to-peer environments. This algorithm is useful for meta-scheduling across known clusters and scheduling on desktop grids. To accomplish this, I build on previous work to route jobs to appropriate resources then use the new algorithm to start parallel jobs and balance load across the grid. I also discuss what constitutes useful clusterings for this algorithm as well as inherent scaling limitations. Ultimately, I show that my algorithm performs comparably to one using centralized load balancing with global up-to-date information. The principal contribution of this work is that the parallel job scheduling is completely decentralized, which is not featured in previous work, and enables reliable *ad hoc* sharing of distributed resources to run parallel computations.

Second, I show how clusters of computers can be found dynamically by using an existing latency prediction technique coupled with a new refinement algorithm. Several latency prediction techniques are compared experimentally. One, based on a tree metric space embedding, is found to be superior to the others. Nevertheless, I show that it is not quite accurate enough. To solve this problem, I present a refinement algorithm for producing quality clusters while still maintaining bounds for the amount of information any given node must store about other nodes. I show that clusters derived this way have scheduler performance comparable to those chosen statically with global knowledge.

Lastly, I discuss previously undiscovered under-specifications in the Content Addressable Network (CAN) structured peer to peer system. In high-churn situations, the CAN allows stale information and changes to the overlay structure to create routing problems. I show solutions to these two problems, as well as discuss other issues that may also disrupt a CAN.

PARALLEL COMPUTING WITH P2P DESKTOP GRIDS

by

Gary Lee Jackson II

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Acknowledgments

For others, this work has been an exercise in patience with and faith in me. To that end, I owe an enormous debt of gratitude to all those who have shown both in excess.

First, I thank my advisor, Professor Alan Sussman. Alan always has time to talk to me, and has always had a calming word and a key insight for me in my moments of panic. This has always been true, from the time I considered going back to school until the present. I also thank my co-advisor, Professor Pete Keleher, who along with Alan never let me quit. In that same vein, I thank Professor Jeff Foster and Professor Mike Hicks for also never letting me quit. Twice I have walked in to a Professor's office with the intention of giving up, and twice I have been told to stick with it. There is no need for a third visit. In addition to Professor Sussman and Professor Keleher, I thank Professor Jeff Hollingsworth, Professor Adam Porter, and Professor Derek Richardson for sitting on my committee and for giving me their time to read this manuscript.

I thank my predecessors in Professor Sussman's group, including Beomsok Nam, Jik-Soo Kim, and Jaewahn Lee for paving the way for my research. In particular, I thank Sukhyun Song for his work with decentralized tree metric space embedding and for giving me so much help with using his simulator. Without the timely appearance of his work, my work would be doomed by inadequate latency prediction.

To the systems staff of both the Computer Science Department and the University of Maryland Institute of Advanced Computer Studies, I owe thanks. I know that I have had, at times, some very difficult requests. I also know that they have done their best to satisfy those requests. Hopefully I was not too burdensome, because I know what it is like to be in their shoes.

In particular, I owe my deepest gratitude to my family for their love and support. Many thanks to my mother for putting me on the right path, to my father for the inspiration he has given me, and to my brother for helping me write. My wife, especially, has had the greatest of patience to let me follow my dream, and for that I am very thankful.

Lastly, I thank God for the gifts He has given me.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CAN	Content Addressable Network
DHT	Distributed Hash Table
DLB	Dynamic Load Balancing
DTM	Decentralized Tree Metric
FCFS	First Come, First Serve
GNP	Global Network Positioning
MPI	Message Passing Interface
PBS	Portable Batch System
QB	Queue Balancing
UMIACS	University of Maryland Institute for Advanced Computer Studies

Chapter 1: Introduction

Tightly-coupled parallel computing requires low latency communication between simultaneously scheduled processes. Applications for tightly-coupled parallel computing are typically written using MPI (Message Passing Interface) or some other message passing scheme and run on purpose-built clusters or shared memory systems. They are an important tool for high performance computing. A *grid* is a collection of resources across geographical and administrative boundaries brought together to serve a purpose. Grids often provide resources tightly-coupled parallel applications, though typically not across the entire grid.

Decentralized peer-to-peer networks, such as those found in distributed hash table systems are failure tolerant and have a low administrative burden. Losing any single node will not cause the network to fail. Moreover, these networks tolerate multiple failures over time. These networks self-organize and self-heal, requiring little intervention from human operators to work.

My thesis is that **scalable parallel computing in a decentralized, distributed environment is both feasible and can be done with performance comparable to centralized systems**. To that end, I have three primary contributions:

1. Decentralized scheduling and load balancing for parallel jobs.
2. Algorithms for finding suitable clusters dynamically using latency prediction.
3. A set of improvements to the robustness of the underlying structured peer-to-peer network.

For scheduling and load balancing, I assume that users will set maximum job sizes as a matter of policy. Finding a group of nodes larger than the maximum job size is unnecessary. A grid that efficiently routes, runs and load balances parallel jobs is constructed using the maximum job size. Using this assumption, complexity in the algorithms presented here scale with maximum job size as specified in policy rather than the size of the grid.

I show ways to run and schedule tightly-coupled parallel jobs in a decentralized grid. The conjunction of these features is the novelty of this work. This work is validated with favorable comparisons to real-world job traces from real grids, achieving performance comparable to centralized load balancing with global up-to-date information.

Users describe jobs in terms of the number of nodes needed for parallel computation and a maximum latency for inter-node communication. Jobs are routed and run on resources that satisfy those requirements, as long as those resources are present in the grid. The load balancing algorithms spread load between these resources to reduce idle time and improve responsiveness.

Not all clusters are created equal. The algorithms presented here can tolerate partially overlapping clusters. However, varying cluster size and the degree of overlap powerfully affects the efficiency of clustering algorithms. I evaluate the performance consequences of these variables and discuss what constitutes good clusters for the scheduling algorithm.

The scheduling algorithms are implemented in a simulator. The experiments are based on job traces recorded in real-world computational grids[1] [1]. The results show that the new algorithms enable running parallel jobs in the decentralized peer-to-peer grid with good overall performance. Over-provisioning and load balancing are crucial to good job scheduling performance. An *over-provisioned* job is one where more resources are requested than needed. Excess requests are canceled when the job finds adequate resources and starts.

Finding optimal clusters with no foreknowledge of network structure is a difficult problem, requiring $O(n^2)$ measurements between all involved nodes. Clustering is even worse, since it is effectively a max clique problem, which is NP-complete. Instead, I find clusters using approximation, with good results.

I evaluate three different latency prediction techniques that are the work of others: Global Network Positioning (GNP) [2], Vivaldi [3], and Decentralized Tree Metric (DTM) [4]. DTM is found to be the most accurate. It also has a built-in clustering algorithm, and it is fully decentralized. However, its results are not perfect and do not necessarily satisfy the qualities possessed by good clusters for the scheduling algorithm.

I therefore use the DTM results as input to a decentralized refinement algorithm. The refinement algorithm produces clusters with performance comparable

[1]`http://www.cs.huji.ac.il/labs/parallel/workload/`

to static clusters chosen with prior knowledge of network structure. Symmetry is important for the refinement algorithm, as it allows nodes to advertise the best possible clusters without affecting the bounds on the number of other nodes with which a given node must communicate.

The Content Addressable Network (CAN) [5] is the underlying structured peer-to-peer network used for the grid. It tries to guarantee connectivity as long as two adjacent nodes do not fail within the time it takes to recover from one failure. While experimenting on this system, however, I found several situations that cause routing discontinuities that do not involve simultaneous failure. First, it allows stale information to leak back in to the topology. Second, it is vulnerable to breaking when adjacent nodes join or leave voluntarily. Effectively, this simultaneous voluntary change resembles a simultaneous failure. I show solutions to these two problems, and discuss other scaling and repair issues that are inherent to all CANs.

My goal is to construct a decentralized grid that discovers resources suitable for tightly-coupled parallel computing and exploits those resources without centralized information or control. This is a difficult problem for two reasons. First, finding exact clusters of computers without prior knowledge of their relationships is intractable. Second, because parallel job scheduling is entirely decentralized, new techniques for matching jobs to resources and scheduling those resources must be developed. I solve both of these problems with new scheduling and cluster refinement algorithms. Furthermore, I make significant improvements to the underlying peer-to-peer network.

Chapter 2: Background

2.1 P2PGrid

P2PGrid [6, 7] is a desktop grid system that uses a peer-to-peer overlay for decentralized operation. It relies on a heavily modified version of the structured peer-to-peer overlay CAN to organize nodes in to a Euclidean space based on their characteristics. It then routes jobs across this space according to their requirements, to find a node on which to run the program.

CAN as originally described is a distributed hash table (DHT) much like Chord [8], Pastry [9], and many other similar systems. However, CAN has a slight twist: Instead of mapping keys into a one dimensional space, it uses a scheme where nodes are mapped into a multidimensional Euclidean space. CAN works by dividing the given space into zones much like a *kd*-tree divides space. A given zone is associated with one node, which has mapped itself to a point inside that zone. As nodes join and leave, zones are split and merged as with the *kd*-tree. Special care is taken so that nodes can identify for themselves when they need to take over some part of a neighboring zone when the owner of the neighboring zone fails or leaves voluntarily. It is the spatial nature of CAN that is exploited for P2PGrid.

With a few compromises for robustness and performance, hashing can be ig-

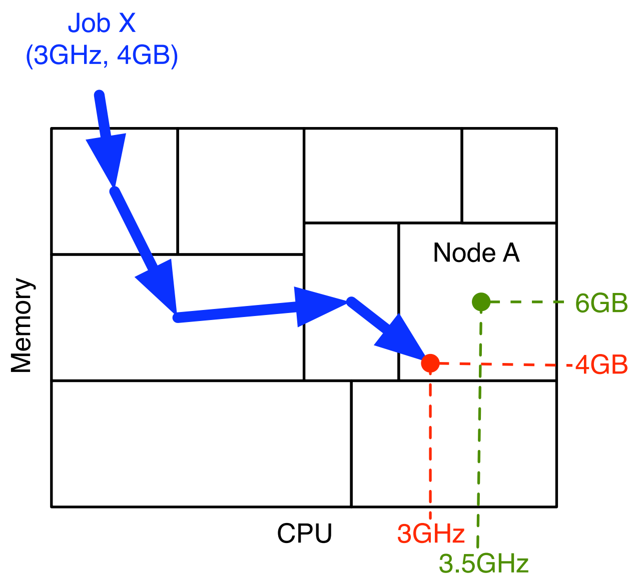


Figure 2.1: Job X is routed to Node A by always forwarding the job to the neighbor closest to the Job X requirement.

nored on key dimensions. The dimensions are instead used directly so that CAN is treated as a distributed spatial index. P2PGrid uses the attributes of a node as the dimensions, such as processor speed, memory, and available disk space.

For an example, see Figure 2.1. Suppose only processor speed and memory are indexed in the CAN. Some node in the CAN, call it Node A, has a 3.5GHz processor and 6GB of memory. A job called Job X requires a 3GHz processor and 4GB of memory. Job X can be routed from any point in the CAN to Node A by forwarding the job to the closest neighbor by Euclidean distance.

Numerous extensions have been added to this functionality to support more types of resources and to enhance the basic functionality of the system. The first is to resolve collisions between nodes with identical resource capabilities. These nodes can be distinguished with a new dimension, called the *virtual dimension*, where a node picks a random value for the point at that dimension when it joins the P2P network. This helps to spread out nodes across the CAN space and lets multiple nodes with identical resource capabilities occupy different zones. This virtual dimension is also used to distribute job load, since jobs also pick random values for the virtual dimension when they are submitted.

Support for deeper heterogeneity has also been added. For instance, support for Windows and Linux hosts simultaneously, or some other configuration of multiple software versions. It is impractical to build a CAN for these high-dimensional cases. However, since all of these values are qualitative rather than quantitative, the are used as input to a space filling curve and combine many of these attributes into a single scalar value [10]. Architecture, operating system, etc. are each mapped in to their own dimensions. These *qualitative* coordinates are coded to a point along a Hilbert space filling curve. This curve is treated as another CAN resource dimension, called the *transform dimension.* Each single value in the transform dimension represents a whole class of nodes, so all of the nodes present at a given transform dimension value are called a *subCAN*. In a sense, all the nodes in a given subCAN are on an island unto themselves for job scheduling. P2PGrid has also been extended to support multi-core processors [7] and dynamic load balancing for serial jobs [11].

The number of dimensions in CAN must limited because adding more dimensions increases the number of neighbors that a given node has [5]. This is a problem for scalability. However, it is also a problem for robustness because it increases the probability that two adjacent nodes will fail in less time than it takes to recover from one failure. Without the adaptations for parallel processing, P2PGrid uses 6 dimensions: CPU frequency, memory, disk, number of cores, virtual, and transform.

2.2 Latency Prediction

Part of the intent for this work is to dynamically discover groups of nodes close enough for tightly-coupled parallel computation. However, making and disseminating an all-to-all measurement is not scalable because it involves making and storing $O(n^2)$ individual measurements. For this reason, *latency prediction* is necessary for finding groups of close nodes. Latency prediction uses a scalable amount of information stored at any given node to make an accurate prediction of the latency between any two nodes without the need for making an all-to-all measurement.

One approach to this problem is to *embed* the nodes in a *metric space*. A metric space is a set of points M and a distance function $\delta : M \times M \to \mathbb{R}$ such that:

1. $\delta(x, y) = 0$ if and only if $x = y$
2. $\delta(x, y) = \delta(y, x)$ (symmetry)
3. $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$ (triangle inequality)

An embedding is a function e from nodes N to points in a metric space M: $e : N \to M$. The distance between nodes $a, b \in N$ is then predicted to be $\delta(e(a), e(b))$. This embedding cannot be perfect because triangle inequality violations happen in real networks [12, 13]. Suppose the actual measured distance between nodes in N is $\Delta : N \times N \to \mathbb{R}$. A good embedding minimizes the error between δ and Δ.

A well known metric space is a Euclidean n-space. A Euclidean n-space the set of all n-ary vectors $\mathbf{x} \in \mathbb{R}^n$ plus the Euclidean distance metric. The Euclidean distance metric $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is the inner product of the difference of $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$:

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}|| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Euclidean space embeddings are how two of the latency prediction techniques examined here, GNP and Vivaldi, work.

GNP [2] uses a two-part process. First, a fixed set of landmark nodes are chosen. The number of landmark nodes is small, between six and fifteen members. A minimum of six independent points are needed to define a five dimensional space, and results do not appreciably improve for more than fifteen landmarks in spaces up to seven dimensions. An all-to-all measurement is made between all landmark nodes. Coordinates in the Euclidean space are found for these nodes such that the error in the predicted distances versus the actual distances is minimized. In the cited paper, the authors used the Simplex Downhill method [14], but they claim other minimization techniques could be used as well. Once the landmark coordinates are calculated, they are made available to the network at large. The second part starts when any other node joins. The new node makes measurements to the landmark nodes. Solving the same minimization problem, the new node finds a coordinate for itself that minimizes error.

Vivaldi [3] also embeds nodes in to a Euclidean space. Unlike GNP, Vivaldi is fully decentralized in that it does not rely on landmarks or any other mechanism which gives nodes special status. Vivaldi works by simulating a physical mass-spring system, including dampening to reduce oscillation. Every node has some conception of what its own coordinate should be. Over time, nodes make measurements to other nodes. After making such a measurement, a given node will adjust its own coordinate by a small amount to minimize the error for that measurement. Vivaldi uses the absolute error for making these adjustments rather than relative error. Optionally, Vivaldi can use spherical space instead of Euclidean space to model the surface of the earth, and add an optional height vector to model local area network latency.

2.2.1 Tree-Metric Space Embedding

A *tree metric* space is a metric space that satisfies the stronger *four-point condition (4PC)*:

$$\delta(i,j) + \delta(m,n) \leq max\{\delta(i,m) + \delta(j,n), \delta(j,m) + \delta(i,n)\}$$

Figure 2.2 illustrates the 4PC, after [15]. When $m = n$, this condition is equivalent to the triangle inequality. When a space satisfies the four-point condition, it can be coded as a tree [16] where the distance between two vertices is the sum of all edge weights on the path between two vertices. Furthermore, any tree with non-negative weights constitutes a tree metric space.

Sequoia [15] is a centralized algorithm for embedding networks in to a tree metric space. When the network does not violate the 4PC, the embedding is proved to be perfect without needing to make $O(n^2)$ measurements. Furthermore, when

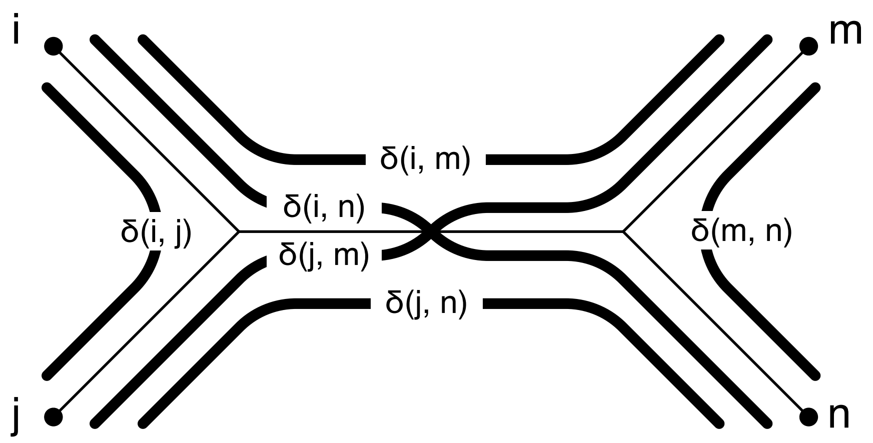


Figure 2.2: An illustration of the four-point condition (4PC).

the 4PC is relaxed in to the ϵ-4PC that allows for bounded error in 4PC violations, the error in the resulting embedding is also limited. The Sequoia designers claim that though the Internet violates the ϵ-4PC, the Internet is still close enough to a tree metric space that the embedding should still be useful for latency prediction. This claim is supported in later work [17], where the centralized Sequoia algorithm is compared favorably to Vivaldi.

Later, separate work in our group takes the Sequoia algorithm and turns it in to a decentralized, structured peer-to-peer system [4], designed to predict network bandwidth. This work was later extended for clustering [18] and centroid-finding. This work is referred to here as DTM, to distinguish it from the earlier, centralized Sequoia algorithm. DTM is trivially adapted from bandwidth to latency by using latency measures directly instead of using a transformation on the bandwidth. DTM holds key insights to improving prediction, including the use of simulated joins. A node will simulate two joins to the peer-to-peer network to find the best location for an actual join.

Chapter 3: Related Work

3.1 Peer-to-Peer Networks

This work is built on decentralized peer-to-peer (P2P) networks, where client nodes interact with each other as well as with other possible services within the system. Some popular P2P file-sharing systems are based on a centralized service or directory, such as Napster [1] and BitTorrent [2]. Still others are entirely decentralized and unstructured, where nodes interact in a chaotic but locally robust way, such as older versions of Gnutella and Freenet [19]. These are typically used for file sharing, much like the centralized P2P systems. Unstructured networks can be difficult to search efficiently. Later versions of Gnutella and other unstructured networks have started using a two-tier hierarchical approach.

DHTs are P2P systems that are also searchable. Nodes in a DHT P2P network link together in structured ways so that finding a given value is reduced to a routing problem. The idea is to use a hash function to map single search term, such as a filename, in to a key value. The node that owns that key value can be easily identified by routing across the structure of the DHT network. Chord [8], for

[1] `http://opennap.sourceforge.net/napster.txt`

[2] `http://www.bittorrent.org/beps/bep_0003.html`

instance, treats the hash key space as a ring. When a node joins a Chord, it maps its own identity in to the key space and joins in a circular linked list at its identity key position. Pastry [9] and Tapestry [20] are similar systems that use Plaxton trees [21] to structure the network. The CAN [5, 22] hashes values in to a multidimensional Euclidean space. P2PGrid is built directly on CAN, as documented in Section 2.

3.2 Parallel Job Scheduling

There is a large body of work on scheduling and running parallel jobs both on clusters. The Portable Batch System (PBS) [23], LoadLeveller, and the Load Sharing Facility are systems designed to queue, schedule, and run parallel jobs on clusters. Several techniques have been developed to improve scheduling performance, including backfill [24, 25]. Backfill requires foreknowledge of runtime length, which is not always possible on heterogeneous resources. Instead, P2PGrid relies on dynamic mechanisms for matching jobs to free resources. The utility of over-subscription in centralized scheduling has been noted in other work [26].

3.3 Grid Computing

Grid computing is coordinated computing across administrative boundaries [27]. One toolkit that provides a complete solution for grid computing is Globus [3]. Globus provides services for grids such as the Globus Resource Allocation Manager (GRAM) [28], middleware, and implementations of standard interfaces for grid

[3] `http://toolkit.globus.org/toolkit/`

computing [29]. Grids based on Globus services are not typically designed for use in a P2P decentralized system, but there is nothing preventing implementation of such a system on Globus middleware. Other Grid projects are lightweight alternatives to Globus [30] or extensions of batch queue systems to cross administrative boundaries such as with the Load Sharing Facility or the Univa Grid Engine.

3.3.1 Desktop Grids and Volunteer Computing

A *desktop grid* is a grid designed to exploit excess, unused computing resources. HTCondor [31] (formerly called Condor) is a desktop grid system. Nodes and jobs advertise themselves to a central manager. The central manager then matches jobs to resources. HTCondor supports transparent checkpointing [32, 33], parallel jobs [34], and several other features [35]. HTCondor's architecture is centralized, for the most part. Flocking [36] presents a decentralized network of HTCondor pools, but job sharing from one site is restricted to one hop: there is no general way to route a job from a busy pool across an arbitrary network to an idle pool.

Volunteer computing is a particular type of desktop grid where many individual participants compute small parts of a much larger problem. Typically, these work by distributing *work units* to individuals. Work units are solitary parts of the given problem that can be computed independently. One of the problems of volunteer computing is defending against sabotage or cheating [37]. The first two examples of volunteer computing are the Great Internet Mersenne Prime Search[4]

[4]`http://www.mersenne.org`

and `distributed.net`[5]. SETI@Home [38] is another notable volunteer computing project. SETI@Home led to the creation of BOINC [39], which provides a general framework for constructing desktop grids for volunteer computing. This style of computation is not appropriate for tightly-coupled parallel jobs.

3.3.2 Grid Scheduling

There has been some work specifically on grid scheduling. Condor-G [40] extends HTCondor to the Grid, and uses centralized matchmaking. Large scientific grids like XSEDE[6] and the Open Science Grid[7] use Condor-G for this purpose. There has also been some research in hierarchical scheduling [41]. Decentralized scheduling in grids has been done using such optimization techniques as hill climbing [42], ant colony [43], and tabu [44]. Conceptually, the work presented here borrows from some of this work in that the algorithms described are effectively a distributed hill climb in the P2P network. However, the load balancing algorithms presented here differ in that they decentralized at all levels, down to the individual node.

There have been many practical efforts in decentralized job scheduling, including the P2PGrid [6] upon which this is based. For instance, BonjourGrid [45] will schedule loosely coupled bag-of-tasks applications. The work presented here differs principally in that tightly-coupled parallel processes are targeted, where the application is obligated to use many co-scheduled resources that are related in some way, typically by latency proximity.

[5]`http://distributed.net`

[6]`https://www.xsede.org`

[7]`https://www.opensciencegrid.org`

Chapter 4: Exploiting Parallel Resources

4.1 Clustering

Before discussing routing, scheduling, and running jobs, clustering must be addressed. A *cluster* is a group of two or more nodes that can be used together for parallel processing. For tightly-coupled parallel processing, latency typically determines whether nodes can be used together. There may be other factors such as whether two nodes have access to the same software libraries or access to programmable GPUs.

Clusters may be explicitly assigned. This is the case in a conventional parallel processing environment. The knowledge of how nodes are grouped is known *a priori* as well as attributes of these groupings such as inter-node latency. In this case, clusters are disjoint and as large as possible, yet still small enough to be centrally managed. In a sense, this is the ideal situation since no extra work is necessary to advertise cluster size and latency into the grid.

For example, suppose all clusters are known at grid creation. This is *static* clustering. Each node gets a list of all other nodes in its cluster along with the nominal latency for intra-cluster traffic. Each node can then advertise itself into the resource grid and start parallel jobs in the cluster. This is not much different than

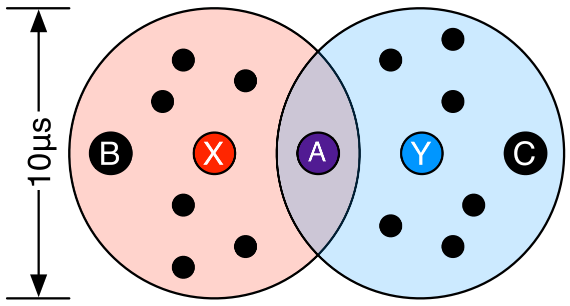


Figure 4.1: Each node is responsible for its own cluster. In this case, nodes X and Y have formed clusters that partially overlap. Node X clusters with node A and node B. Similarly, node Y clusters with node A and node C. However, no cluster of $8\mu s$ diameter can contain both B and C.

using a central resource manager for each cluster. However, the peer-to-peer grid is more fault tolerant at the cost of potential intra-cluster scheduling inefficiency.

On the other hand, clusters may not be known ahead of time. Instead they must be found by making observations (measurements). This type of clustering is *dynamic*. Dynamic clustering is further discussed in Chapter 5.

Resource management algorithms must be prepared to deal with two issues whether the clustering is static or dynamic. First, there is *overlap*. With dynamic clustering each node has a different view of the overall network. This leads to situations where dynamically formed clusters overlap. This scenario is less likely in static clustering, but it may arise temporarily when nodes have inconsistent views of which members of clusters are available. The resource management algorithms must tolerate such overlaps. Figure 4.1 illustrates how this situation arises in dynamic clustering.

Second, there is no bound on the number of nodes within a given latency radius, presenting a scaling problem. For instance, there may be many nodes packed tightly into a single physical location, such as a large machine room. Thus there may be so many nodes in that once place such that it would be prohibitively expensive to manage all the information in a peer-to-peer network. There must be no single grouping of nodes that exceeds some reasonable bound. Furthermore, since clusterings may overlap, a single node can be overwhelmed if it belongs to too many different clusters or otherwise needs to store state for and communicate with too many other nodes. This is where the fundamental assumption comes in to play: *Maximum job size, measured by number of nodes, is a matter of policy.* This yields a fixed constant by which clusterings can be made, since clusters larger than those required by the largest jobs are unnecessary. When clusters are explicitly assigned, size and membership bounds can be guaranteed without difficulty. However, if clustering is dynamic and based on network observations, extra steps must be taken to ensure those bounds are maintained. Dynamic clustering is discussed later in Chapter 5.

4.1.1 Bad Clusters

Static and dynamic clusterings have two potential flaws. These problems cause underutilization, sometimes in serious ways, but they do not cause the scheduling algorithm to fail. Clusters incorrectly identified as too small to service the workload can cause load imbalance. Clusters that overlap can cause the resource management algorithms to interfere with themselves when there are differences between cluster sizes.

A cluster is too small when it is much smaller than the resources that are available. This can arise when the static allocation is poor, when latency estimation fails, or when enforcing cluster membership bounds in dynamic clustering causes resources to be grouped into small clusters. In this case, many nodes may be physically capable of large running jobs, but only a few clusters are made available to run those jobs. Thus, nodes that are placed in small clusters are idle when they do not need to be.

Interference is a somewhat deeper problem that happens when small jobs scheduled on artificially small clusters interfere with the execution of large jobs. For example, suppose there are two clusters X and Y. Cluster X is large. Cluster Y is small, such that only a small fraction of the incoming jobs can run there. Cluster Y is comprised in part by nodes that are also a member of cluster X. Since Y advertises a small cluster size, small jobs may be pushed into that cluster by dynamic load balancing. However, those small jobs will delay the scheduling of larger jobs in cluster X, because now resources from cluster X are being used to run small jobs that could run elsewhere instead. Some level of interference may be unavoidable. In situations where cluster Y is a strict subset of cluster X, the interference is wholly unnecessary. *Symmetry*, a way to avoid this overlap in dynamically discovered clusters while maintaining bounds on cluster size and membership, is discussed further in Chapter 5.

4.2 Routing, Running, and Load Balancing

Matchmaking is the process of routing a job with particular requirements to resources that meet or exceed those requirements. The existing P2P matchmaking scheme matches job to resources by turning the matching problem into a routing problem. Jobs specify their minimum resource requirements, which constitute a point in the CAN resource space. In the simplest job matchmaking case, jobs are routed through the CAN to the node that owns the zone enclosing that point. For parallel computations, parallel resources in the resource space must also be expressed. The CAN dimensions are extended so that the existing job routing mechanisms are used without other changes.

As per Section 4.1, each node is aware of some set of nodes with which it can run tightly-coupled parallel jobs and also knows the nominal intra-cluster latency. However, these latency values cannot be used directly in the CAN dimensions. Instead, the inverse latency is used to indicate that lower latency clusters have a higher capability than higher latency clusters, thus satisfying the assumptions of the job routing algorithm to send jobs on to a minimally capable resource. As a special case, singleton nodes that are part of no cluster can advertise a cluster size of one and a zero inverse latency. Thus the relationship between singletons and clusters is maintained: singletons are always less capable resources both in size and latency than bigger clusters.

A parallel job is then described with latency and size requirements and routed using the CAN routing algorithm. For an example, see Figure 4.2. Node A is part

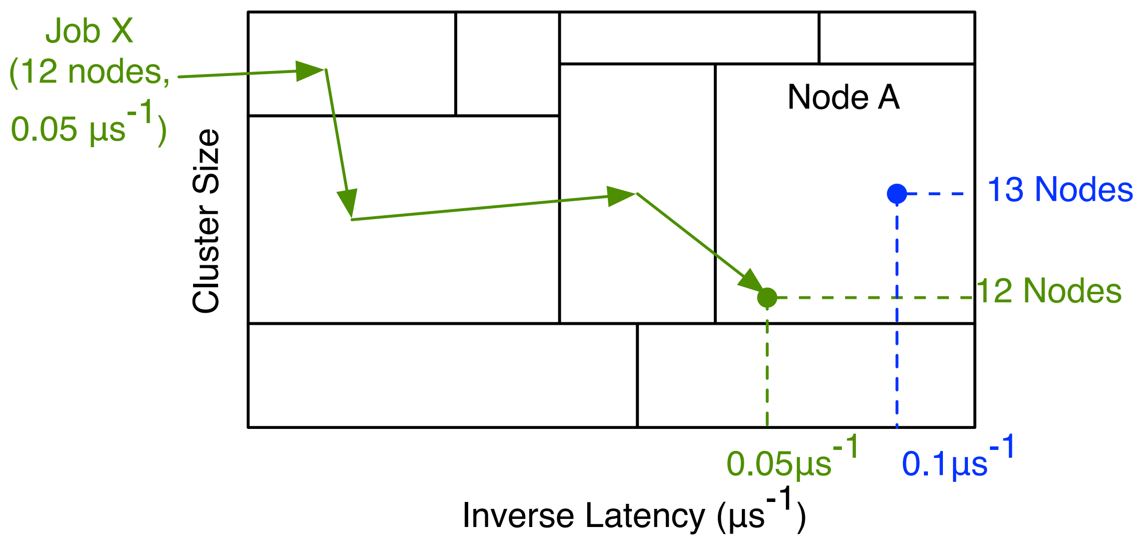


Figure 4.2: Job X is routed to node A by always forwarding the job to the neighbor closest to the job X requirement. Latency is expressed as inverse latency so that lower latencies result in a higher capability.

of a cluster of 13 nodes no more than $10\mu s$ from one another. Job X requires 12 nodes and $20\mu s$ inter-node latency. The job is routed from any point in the resource space to node A by forwarding the job to the closest neighbor by Euclidean distance. Other resource dimensions are omitted here, such as memory and CPU frequency, for clarity.

Starting execution of the parallel job, however, is considerably more complicated than starting a single node job. After the parallel job is routed, many tasks need to be started and managed on different nodes so that the parallel job can run to completion. To this end, a *representative task* is started on the routing end point node.

The representative task is a pseudo-task that does no processing itself and is not counted for job queue length in the node that it resides on. Its sole purpose is to manage the parallel computation from start to finish. The representative task starts immediately at the end point of the job routing, regardless of the queue state and whether or not the node is running a computational task. The representative task dispatches *dependent tasks* to the nearby neighbor nodes. Dependent tasks are the actual runnable tasks that do the work of parallel computation. Each one represents the participation that a single node may put in to the parallel job. This number of dependent tasks can vary from exactly the number of nodes needed for the task up through dispatching dependent tasks to all available nearby nodes (in latency terms). Minimizing job wait time requires some over-provisioning. This is shown experimentally in Section 4.3.

Dependent tasks are almost identical to single-node tasks in P2PGrid and participate in the task queue and the run state of the node in very similar ways. The number of dependent tasks is determined by the job size and the amount of over-provisioning used. For instance, with 2-fold over-provisioning, twice as many dependent tasks as needed are submitted for the parallel job to run. When nodes are chosen for dependent tasks, those that are minimally capable of meeting the job requirements and are not busy are preferred. If candidate nodes are otherwise equivalent, dependent tasks are sent to nodes chosen randomly from among the candidates. For total over-provisioning, a dependent task is submitted to every node available in the latency neighborhood. Every dependent task is associated with its corresponding representative task.

Dependent tasks communicate their state to the representative task. For example, they report to the representative when they arrive at the head of the node's queue and are ready to start computation. When enough dependent tasks are ready to execute, the representative instructs each ready dependent task to begin parallel computation. Any other dependent tasks are canceled (removed from the queues on their nodes) by the representative. As each dependent task finishes its computation, it notifies the representative. When all dependent tasks have finished and reported, the job is complete, and results can be returned to the submitting client. Job queues are kept in order sorted by submit time, with ties broken by the value of a hash on job properties. Though queues are sorted consistently, deadlocks are still a concern. When parallel dependent task with an older submit time arrives at a node where a newer dependent task is ready to start but not yet running, the newer dependent task is killed to prevent deadlock. There must be a certain level of trust between grid participants about not dishonestly manipulating submit times to gain advantage or there must be some scheme to enforce honesty across the grid. The discussion of such a scheme is out of the scope of this work.

Although this static scheduling scheme is satisfactory for running jobs eventually, its performance can be improved. For instance, a single long-running serial job can prevent a parallel job that requires an entire cluster of nodes from running. All nodes except the one with the serial job will sit idle, since the dependent tasks that are ready to run are all waiting for the signal to proceed from the representative. Such waiting can be very inefficient. I have found that dynamic load balancing (DLB) and Queue Balancing (QB) can help reduce wait times considerably.

DLB and QB have been widely used in scheduling algorithms. However, particular care must be taken to make them work in the decentralized parallel environment. The dynamic scheduling techniques work on the principal of moving jobs from a node where the job is scheduled to another capable node that may be able to run the job sooner. The difficulties lie in deciding which jobs get moved, and where. Under DLB, waiting jobs get moved to resources capable of running the job immediately if they are available. I have also added queue balancing to DLB, which moves jobs to resources with shorter queues when free resources are not available.

CAN topology strongly affects the efficiency of these scheduling enhancements. If a node only has two or three direct neighbors, which may happen in highly homogeneous networks, then the ability to find separate clusters of nodes is limited. This problem is mitigated somewhat by allowing nodes to probe indirect neighbors in the CAN as well as direct neighbors for their capabilities and availability. This information is already available through the neighbor data used for P2P network maintenance, so does not incur much additional message traffic. In the following algorithmic descriptions, "neighbors" may refer to direct neighbors only or both direct and indirect neighbors in the CAN.

Before describing the load balancing algorithms, several terms must be defined:

Resource coordinate: A coordinate in the CAN space that expresses capability. This can include resources such as processor frequency, memory capacity, nodes available for parallel processing, and the inverse radius of the parallel processing neighborhood. A given node in the network has a resource coordinate. Some values are fixed, such as processor frequency and memory. Others, like the ones used for matching parallel jobs, can change as other nodes leave and join the grid as well as if the network itself changes, changing latency measurements. For instance, a Node A can have coordinate (3.5GHz, 4GB, 32 nodes, $1/10\mu s^{-1}$), meaning the node has a 3.5GHz processor, 4GB of memory, and 32 nodes in the local latency neighborhood with a radius of $10\mu s^{-1}$ that are available for parallel computing.

Residue coordinate: A point in the CAN space that expresses the amount of free resources available. At some point in time, Node A's residue coordinate might be (3.5GHz, 4GB, 24 nodes, $1/10\mu s^{-1}$). So, Node A might be free, but eight nodes in its neighborhood are unavailable for computation. Therefore a 32-node job should not be sent to Node A if it can be avoided, since Node A will not be able to run the job immediately.

Requirement coordinate: To route jobs, the resources that the job requires must first be described. A job must be routed to a node such that all requirements are less than or equal to the node's capabilities. For instance, a job J may have a requirement of (3.5GHz, 2GB, 8 nodes, $1/10\mu s^{-1}$). Node A would then be a valid target for J.

Runnable Task: A serial job or a dependent task. Contrast with the representative task, which does not occupy computational resources to do work.

Queue Position: The number of runnable tasks ahead of a given runnable task in the job queue on a node. Runnable tasks that are already executing have a queue position of zero.

Average Queue Position (averageQP): The average queue position for all dependent tasks in a parallel process, or just the queue position for a single node task.

Queue Length: The number of runnable tasks on a given node.

Maximum Queue Length (maxQL): For all members of the cluster viewed by a node M, this is the largest queue length. This represents a worst case scenario: Any parallel job started on M will have dependent tasks with a queue position of no more than the maximum queue length.

Minimum Coordinate: For some set of coordinates M, m is the minimal coordinate when such that there are no nodes m' in M such that $m'.coordinate < m.coordinate$. The $<$ relation is defined such that for every dimension i, $a_i < b_i$. There may be coordinates such that neither $m < m'$ or $m' < m$, but for all other m'', $m < m''$ and $m' < m''$. In this case, the choice between m and m' is random between the two.

DLB can alleviate problems that arise from limited information in the initial job scheduling, thus improving resource utilization and decreasing job wait times. DLB moves idle jobs from busy resources to idle resources capable of running the jobs. DLB for the P2PGrid has already been investigated for the serial job case [11]. However, care is needed to make DLB work for parallel jobs, since parallel jobs are distributed across multiple nodes. Moving a parallel job from one cluster to another involves coordinating the move between the representative task and all dependent tasks.

The DLB algorithm is described in Algorithm 1. This algorithm runs periodically on each node. For simplicity, only parallel jobs are addressed in this description. The algorithm attempts to send non-running jobs to free resources. Condition c0 in the algorithm ensures that node m can accommodate the job immediately. This information may be stale when the algorithm runs, but works in

Algorithm 1 dlb(n): Dynamic Load Balancing on node n

```
1:  for all $j$ in idle representative tasks and serial jobs do
2:      for all $m$ in neighbors of $n$ do
3:          let c0 := $j$.requirement $\leq$ $m$.residue
4:          if c0 then
5:              add $m$ to candidate set $S$
6:          end if
7:      end for
8:      if $S$ is not empty then
9:          let $m \in S$ such that $m$ is minimum
10:         send $j$ to $m$
11:         break from the enclosing loop
12:     end if
13: end for
```

practice. The algorithm ensures progress for a job by only allowing the job to get moved once. The algorithm stops after moving a single job, since overwhelming a free resource is undesirable.

DLB scales in the CAN since it only uses information that is already stored at each node: the periodic update mechanism for DLB uses direct and indirect neighbor information already used by the CAN to maintain routing connectivity. The CAN itself may scale poorly in some circumstances, by developing a pathological topology such that a single node has $O(n)$ neighbors, but that is a problem with the CAN itself and not with the scheduling algorithm.

The QB algorithm is described in Algorithm 2. This algorithm is very similar to the DLB algorithm, with two notable exceptions. First, condition c0 stipulates that candidate nodes be capable of running the job (requirement), but it is not necessary that they be able to run the job immediately (residue). Second, condition c1 ensures that the process is moved to a better queue position. Since the maximum queue length at the candidate is shorter than the current average queue position, the new average queue position would not be any worse than the current average queue position.

QB requires a little more bookkeeping traffic than DLB. First, each representative task must collect the queue position from all dependent tasks. The message traffic cost is bound by the maximum cluster size, since there cannot be more dependent tasks that nodes in a cluster. The messages themselves are of constant size. Each individual node sends its queue length information to any other nodes that manage a cluster containing the individual node. The outgoing number of messages

Algorithm 2 qb(n): Queue Balancing on node n

```
1:  for all $j$ in idle representative tasks and serial jobs do
2:      for all $m$ in neighbors of $n$ do
3:          let c0:= $j$.requirement $\leq$ $m$.coordinate
4:          let c1:= $m$.maxQL< $j$.averageQP$-1$
5:          if c0 $\wedge$ c1 then
6:              add $m$ to candidate set $S$
7:          end if
8:      end for
9:      if $S$ is not empty then
10:         let $m \in S$ such that $m$ is minimum
11:         send $j$ to $m$
12:         break from the enclosing loop
13:     end if
14: end for
```

is bound by the cluster membership limit, and the message size is constant. On the receiving end, the number of messages that a given node receives is also bound by the maximum cluster size. Ultimately, since the message size is very small and the number of messages is bound, this results in very little overhead necessary to balance load.

4.3 Simulations

Experimental evidence suggests that DLB and QB can improve job scheduling performance. The experiments described here use a modified version of the detailed event simulator employed in previous work on the P2PGrid [6, 7, 11]. The version used for these experiments implements the cluster size and network latency dimensions, parallel job management, dynamic load balancing, and queue balancing.

Three different input workloads from two separate grids are used. To validate decentralized scheduling in the grid, slices from two traces from the Parallel Workloads Archive[1] [1] are used. The first validation slice is a 30,000 job trace from the SHARCNET grid. SHARCNET is a heterogeneous grid consisting of 10 clusters of 32 to 768 nodes each, for 2091 nodes total. This slice is selected such that the difference between the first and the last submission time is minimized. The second validation slice consists of more than 70,000 jobs from the busiest month traced from the DAS2 grid. DAS2 is a homogeneous grid consisting of five clusters, four of which have 32 nodes and largest has 72, for 200 nodes total. This subset contains

[1]`http://www.cs.huji.ac.il/labs/parallel/workload/`

periods of both heavy and light use.

To test the algorithms with high load and grid heterogeneity, the SHARCNET dataset was filtered and compressed to increase the number of parallel jobs and reduce the time between job submissions. This subset has all non-parallel jobs filtered out, sliced down to the 1,000 jobs such that the time between the first and last submissions is minimized. Then the submit times are compressed so that all 1,000 jobs are submitted over a period of 643 seconds:

1. I had the original 30k job data set, “slice”, which I used for validation. This was selected so that the difference in submit time between the first and last jobs was minimized. In other words, it was the 30k jobs submitted over the shortest interval (about 12 hours).

2. I wanted to come up with a comparable data set with only parallel jobs so I:

    (a) Filtered out all serial jobs.

    (b) Selected the 30k parallel jobs where the difference between submit times was minimized (much like the slice data set).

    (c) Compressed the interarrival times of this new 30k job set down to the same interval as the slice data set.

3. Unfortunately, this 30k job dataset became unwieldy and un-simulatable: simulations would run for a couple days then crash. This was an integer overrun limitation of the simulator. So I just used the first 1k jobs from that data set.

This data set is usable and useful for three reasons. First, it is an otherwise unmodified subsequence of parallel jobs. Second, it is compressed to a shorter interval than any sequence of 1000 parallel jobs appears in the original input. There is a sequence of 1000 parallel jobs submitted over 804 seconds in the original input whereas the intense data set is submitted over about 643 seconds. Thirdly, it is far more diverse than that sequence submitted over 804 seconds, which was from 1000 nearly identical jobs submitted to the same cluster.

In the first three experiments, clustering is explicitly set to use the clusters from the original data sets. That is, ten clusters from SHARCNET and the five clusters from DAS2 are modeled as they appeared in the real systems. In the remaining experiment, the clustering is varied to show how different clustering approaches can affect performance.

For all experiments, *bounded slowdown* [24] is the metric. Bounded slowdown quantifies the queuing delay relative to the total run time of the job. However, it uses a lower bound for job run time, in this case 10 seconds, to ameliorate the effects of very short jobs, preventing them from distorting the average slowdown. Bounded slowdown is computed as follows, where s_i is the slowdown for the ith job in the trace: $s_i = 1 + \frac{Q_i}{max(R_i,10)}$.

In this equation, Q_i is the amount of time that the job spends in the queue and R_i is the run time of the job. Unlike many conventional parallel batch queuing systems, the algorithms presented here do not require a maximum job run time. For this reason, techniques that can provide a great benefit to job scheduling, like backfill [24, 25], are not applicable in this scenario. This is due in part because estimating run time is difficult in a heterogeneous environment. However, it is expected that a maximum run time is set as a matter of policy so that jobs that run too long can be killed.

The graphs with experimental results are all cumulative distribution functions (CDF) of the bounded slowdown. The x-axis is the bounded slowdown, and the y-axis is the fraction complete. Except where noted, the y-axis spans between 0.5 or 0.7 and 1, since the slowdown for most jobs is very close to 1. Also, the x-axis spans 0 to 5000, though tails will extend past 5000. Where this happens, it is noted.

4.3.1 Validation

The first experiment compares the performance of different configurations of the simulated system to the actual turnaround times observed in the real grid from the workload, where each cluster was scheduled separately. Total over-provisioning is used in this experiment, where dependent tasks are sent to every single node in the cluster regardless of cluster size and job requirement. This experiment is performed twice, once for SHARCNET and once for DAS2. In both experiments, load balancing, static scheduling, and the results observed in the actual trace are compared. For each of these experiments, three different things are compared:

QB+DLB: Decentralized job scheduling with both queue balancing and dynamic load balancing active.

Static: Decentralized job scheduling where scheduling is left entirely to the initial job routing.

Actual: The result of the actual observed job trace.

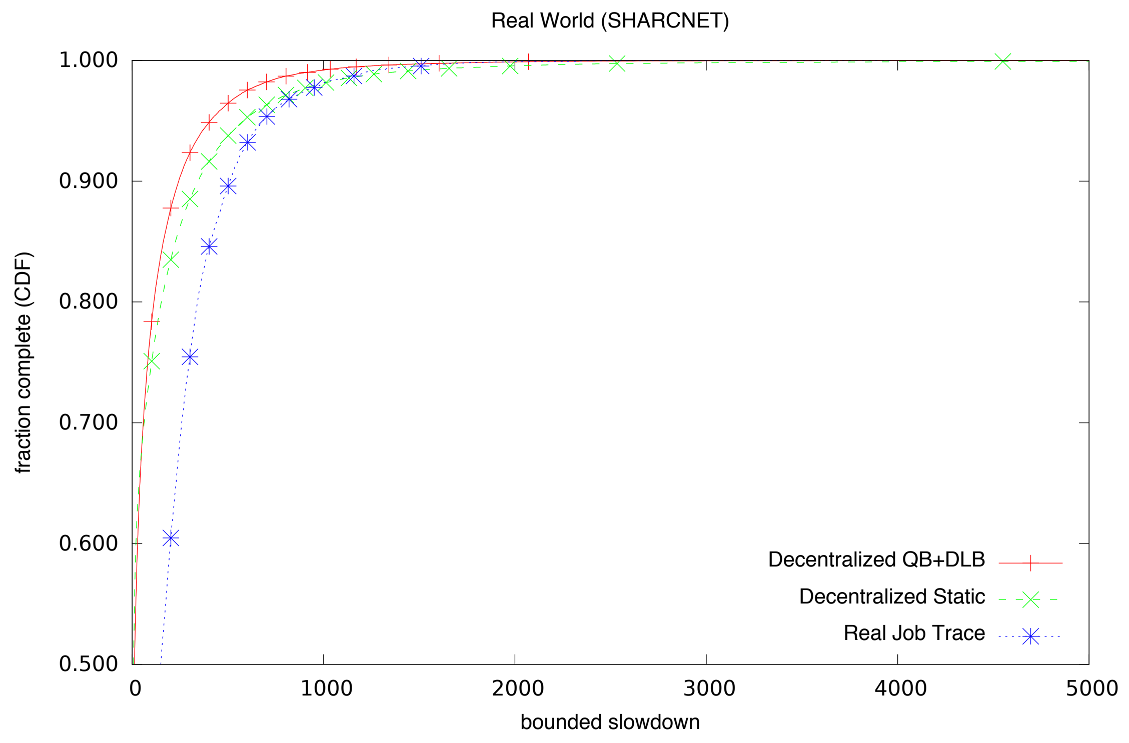


Figure 4.3: Performance for a trace of 30,000 jobs from the SHARCNET grid. Decentralized QB+DLB extends almost to 8,000, and Decentralized Static extends past 35,000.

Results from the SHARCNET experiment are in Figure 4.3. In this case, load balanced results are superior to the static results and both are superior to the actual results. This experiment shows that a decentralized grid can outperform scheduling individual clusters separately because jobs can be move from a loaded cluster to one that is free.

Results from the DAS2 experiment are in Figure 4.4. This job trace is not particularly intensive, so the figure only shows a small portion of the CDF. The decentralized results are nearly identical, with 15.8 mean slowdown for DLB+QB and 16.0 mean slowdown for static scheduling. In contrast, the real world results

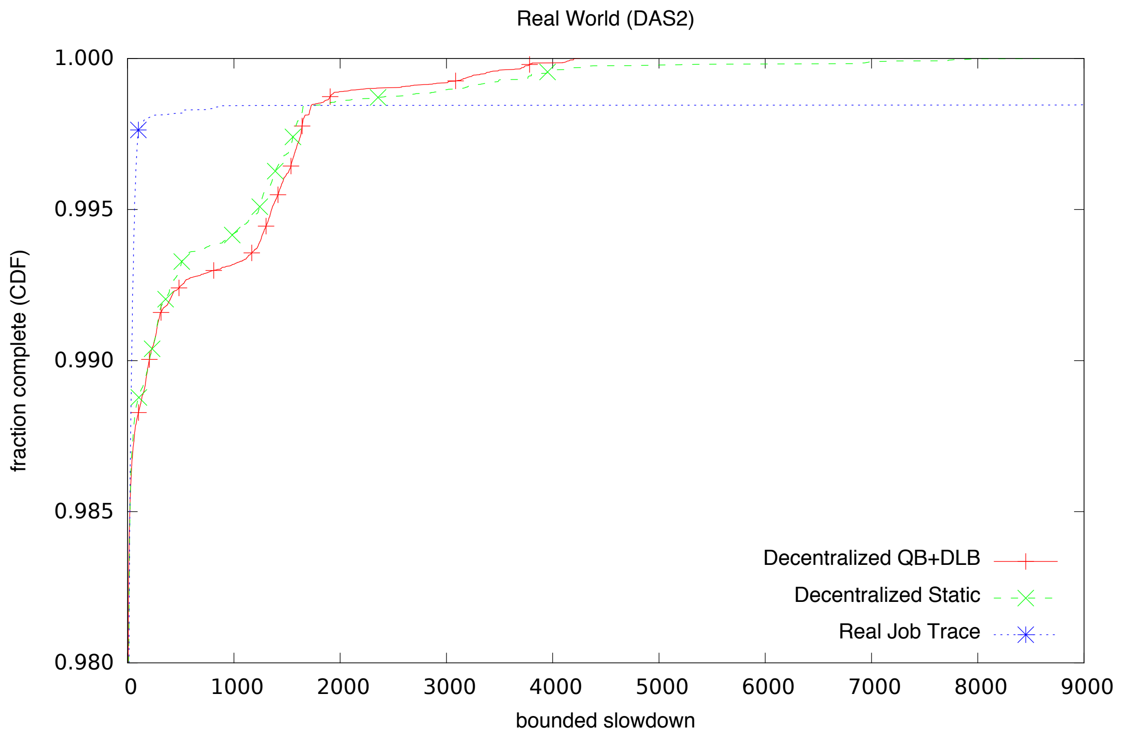


Figure 4.4: Performance for a trace of about 70,000 jobs from the DAS2 grid over the course of one month. In this case, the y-axis only covers the top 2% since most jobs start immediately. The real job trace slowdown extends past 65,000.

had a mean slowdown of 30.2. However, the maximum slowdown is 4,275, 8,647, and 65,707 for DLB+QB, static, and the real world trace. Thus, the load balanced and the static scheduled results are better than the real trace because jobs can move between clusters. Long running serial jobs block access to the single large cluster in the DAS grid. As a result, DLB and QB do not make much of a difference versus static scheduling. *Preemption* of jobs to resources with lesser capabilities may solve this issue.

4.3.2 Heterogeneity and High Load

In this experiment I increase the load beyond what might be seen in a real system, then observe how the scheduling algorithms do under high load. The previous experiment showed that the decentralized grid approach is superior to scheduling on individual clusters, but the load in those cases is not particularly intense, so does not truly stress the scheduler. In this experiment, I use a modified version of the SHARCNET data set. To obtain the input data set from the base SHARCNET data set, I filtered out all non-parallel jobs, took a 1000-job period of intense activity, and time compressed the submission window down to 12 minutes while leaving the job run times the same. Figure 4.5 is a heatmap of parallelism versus job run time. This heat map uses color to express frequency for a given combination of Job Run Time and Job Size. In this case, the most frequent combination of run time and size is for jobs running between 32 and 63 seconds on 2 to 3 nodes. There are 70 such jobs.

Figure 4.6 are the results for static scheduling, QB, DLB, and QB+DLB. Static scheduling performs very poorly under high load. Using QB and DLB in various combinations performs much better, with the best performance obtained from applying both QB and DLB with a mean slowdown of 77.9, just edging out QB alone (83.8) and DLB (91.5). For the rest of the experiments, I use QB+DLB.

The results are also compared to that of an idealized First Come, First Serve scheduler (FCFS), shown in Figure 4.7. The idealized FCFS scheduler has global, up-to-date information about the state of all nodes in the grid and a central job

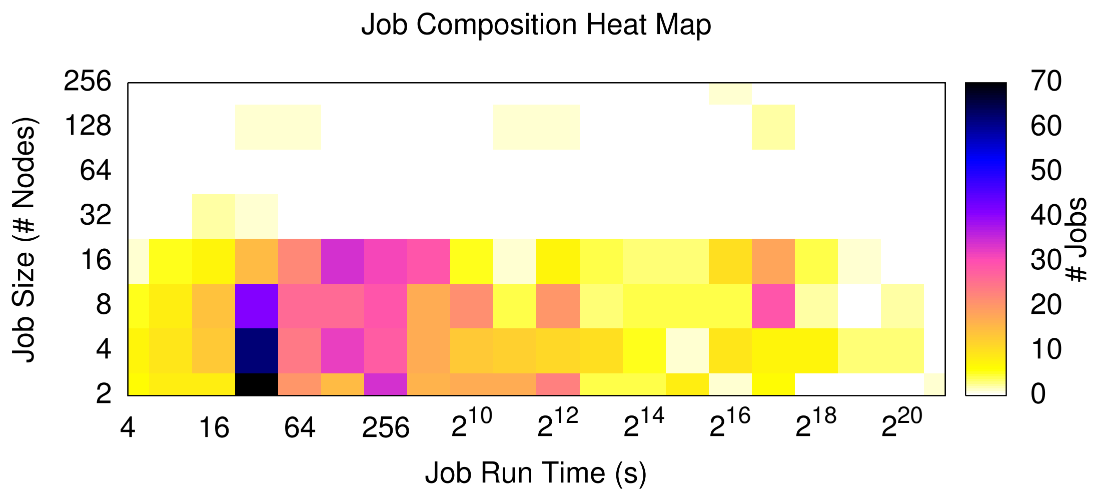


Figure 4.5: A heatmap for the composition of the intense job load showing parallelism (# Nodes) versus run time.

queue. This FCFS scheduler assigns jobs to free resources as the resources become available, and it will allow out of order execution when running a newer job will not delay an older job. For instance, it will run a small job on a small cluster though there is a large job waiting ahead of the small job. The global results have a mean slowdown of 61.4 and the decentralized scheduling has a mean slowdown of 77.9. I show that decentralized scheduling performance is comparable to global scheduling, which is the most important result from this work. I do not intend to show any kind of performance advantage for decentralized job scheduling over centralized job scheduling, only that the benefits of a decentralized system can be enjoyed without a large loss in efficiency.

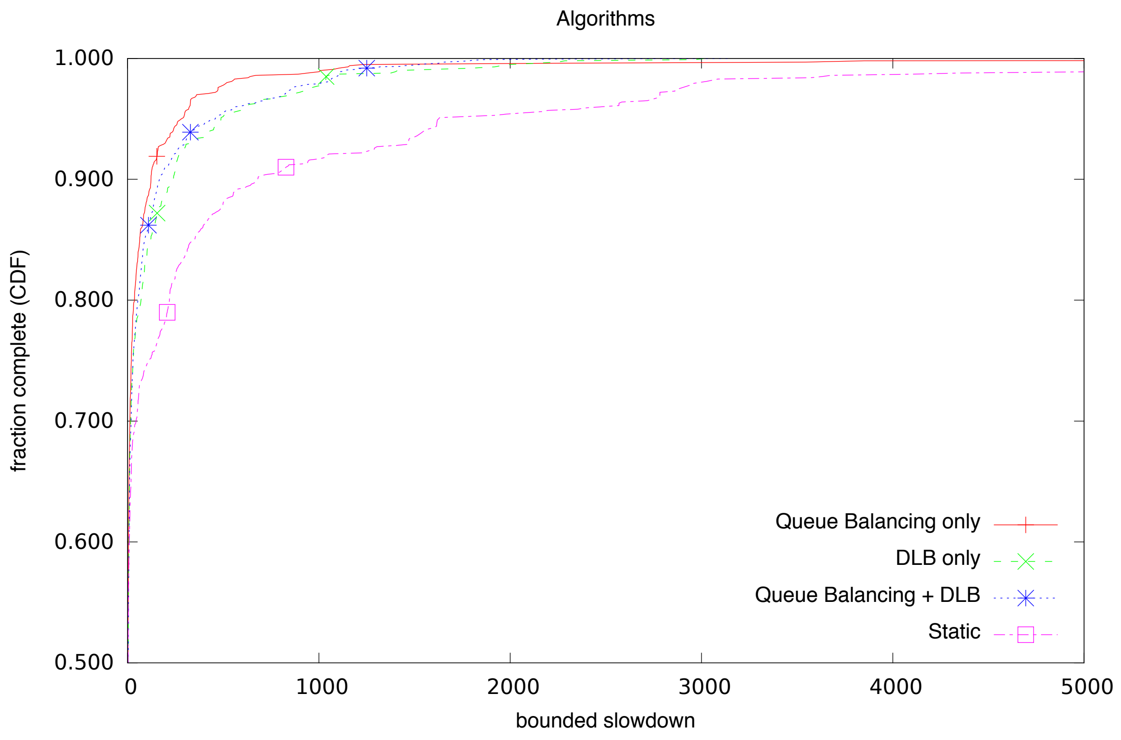


Figure 4.6: Comparing different load balancing algorithms on an intense data set of 1,000 jobs. Static scheduling is very poor, but adding in various combinations of the load balancing algorithms improve performance considerably. The best combination is with both QB and DLB enabled. The QB and Static results extend past 18,000 and 48,000.

4.3.3 Over-provisioning

One of the questions discussed in this chapters is to what extent over-provisioning of dependent tasks is necessary or useful. To answer this question, the level of over-provisioning is varied from none, where no extra dependent tasks are submitted, through total over-provisioning, where dependent tasks are sent to all nodes in a cluster. I use the high-load data set from Section 4.3.2 for this comparison, and

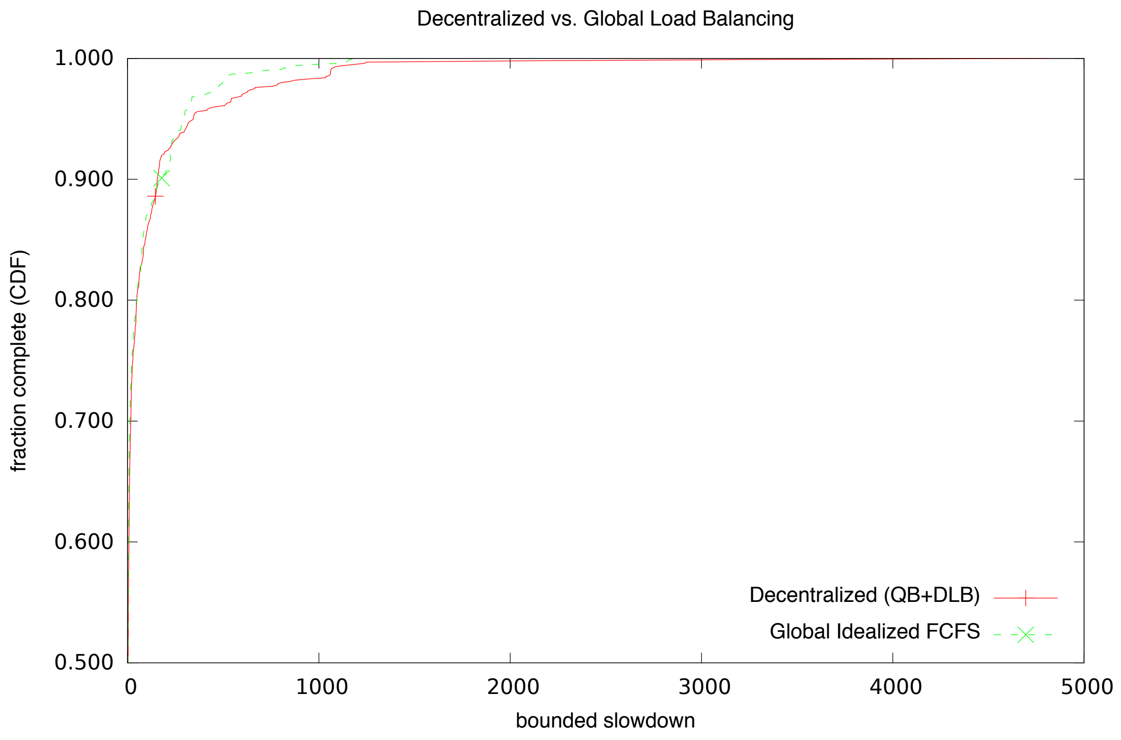


Figure 4.7: Comparing decentralized scheduling to an idealized FCFS scheduler.

use the QB+DLB dynamic scheduling strategy. Figure 4.8 are the results of over-provisioning at total, 8-fold, 1.5-fold, and none. More over-provisioning makes a big difference in the results up to a point, with 8-fold over-provisioning performing slightly better than total over-provisioning. The intuition here is that total over-provisioning here might distort the metrics used for initial job routing and load balancing, since there may be many dependent tasks that never run thus mischaracterizing the actual load on a node. However, the result is that the advantage of over-provisioning outweighs this distortion.

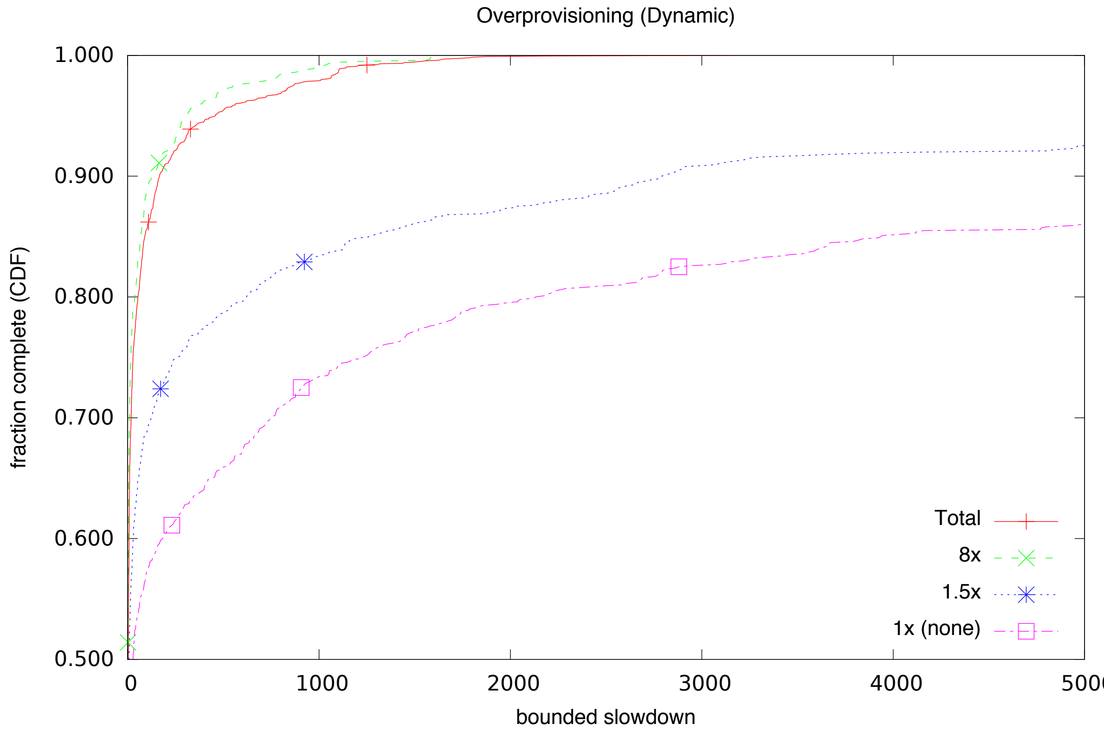


Figure 4.8: Comparing varying levels of over-provisioning. The level of over-provisioning used in running jobs is critical to scheduler performance. No provisioning and 1.5x extend past 55,000 and 28,000.

4.3.4 Clusterings

As noted in Section 4.1, performance can depend on clustering. Several different clusterings of the SHARCNET grid are modeled to evaluate the effects of clustering:

ACTUAL: Clusters are explicitly assigned according to how they appear in the real system. This is the clustering used in the other experiment subsections.

DISJOINT-n: The original groupings are divided into disjoint clusters of n nodes. When the cluster size is not divisible by n, the remaining nodes are put into an under-sized cluster. Each node then individually advertises a cluster of size n (or the remainder). For example, suppose there is an original cluster of 768 nodes. In DISJOINT-64, nodes 1 to 64 are placed in one cluster, nodes 65 to 128 in another cluster, and so forth. Node 1 will advertise a cluster of size 64, and schedule onto nodes 1 to 64. Similarly, node 2 will advertise a cluster of size 64, and schedule on to nodes 1 to 64. The DISJOINT clusterings test the consequences of dividing large clusters in to smaller pieces.

OVERLAP-n: Each individual node is assigned a moving window of n nodes. There is significant overlap between individual node views. For example, suppose there is an original cluster of 768 nodes. In OVERLAP-64, node 1 advertises a cluster of size 64 and runs jobs on nodes 1 to 64. Similarly, Node 2 advertises a cluster of size 64 and runs jobs on nodes 2 to 65. This continues up to node 768, which advertises a cluster of size 64 and runs jobs on nodes 768 and 1 to 63. The OVERLAP clusterings test the consequences of overlap.

The intense data set from Section 4.3.2 is used for these experiments. Where the jobs are too large for the available clusterings, the results are not counted. At most, only 7 jobs out of 1000 are discounted for that reason.

Results from all experiments are summarized in Table 4.1. Disjoint clusterings are compared in Figure 4.9. Surprisingly, the ACTUAL clustering performs the worst. This is due to the interaction between over-provisioning and the way that parallel jobs start. Interestingly, DISJOINT-128 performs best. Results from overlapping clusterings, shown in Figure 4.10, are similar in that all of the results are roughly comparable, with ACTUAL as the best, OVERLAP-64 as the worst,

Clustering	Mean	Median	Mode	Maximum
ACTUAL	77.9	4	2	3150
DISJOINT-64	68.2	2	2	25147
DISJOINT-128	33.3	2	2	5219
DISJOINT-256	51.3	3	2	2910
DISJOINT-512	39.1	3	2	5039
OVERLAP-64	55.2	2	2	4936
OVERLAP-128	31.9	2	2	7048
OVERLAP-256	40.4	3	2	5035
OVERLAP-512	38.7	3	2	2514

Table 4.1: Bounded slowdown statistics for all clustering results.

and DISJOINT-128 very close to ACTUAL.

4.3.5 Summary

There are three important conclusions to take from these experiments. First, dynamic load balancing is important. There is no way to predict how long a job would run, and user estimates in a heterogeneous environment are futile. Thus, though two resources may have equal queue lengths to begin with, imbalance can develop over time as one resource completes shorter jobs. Therefore dynamic load balancing is critical for moving jobs from the loaded resource to the free resource. Dynamic load balancing is critical and will continue to be critical in any environment

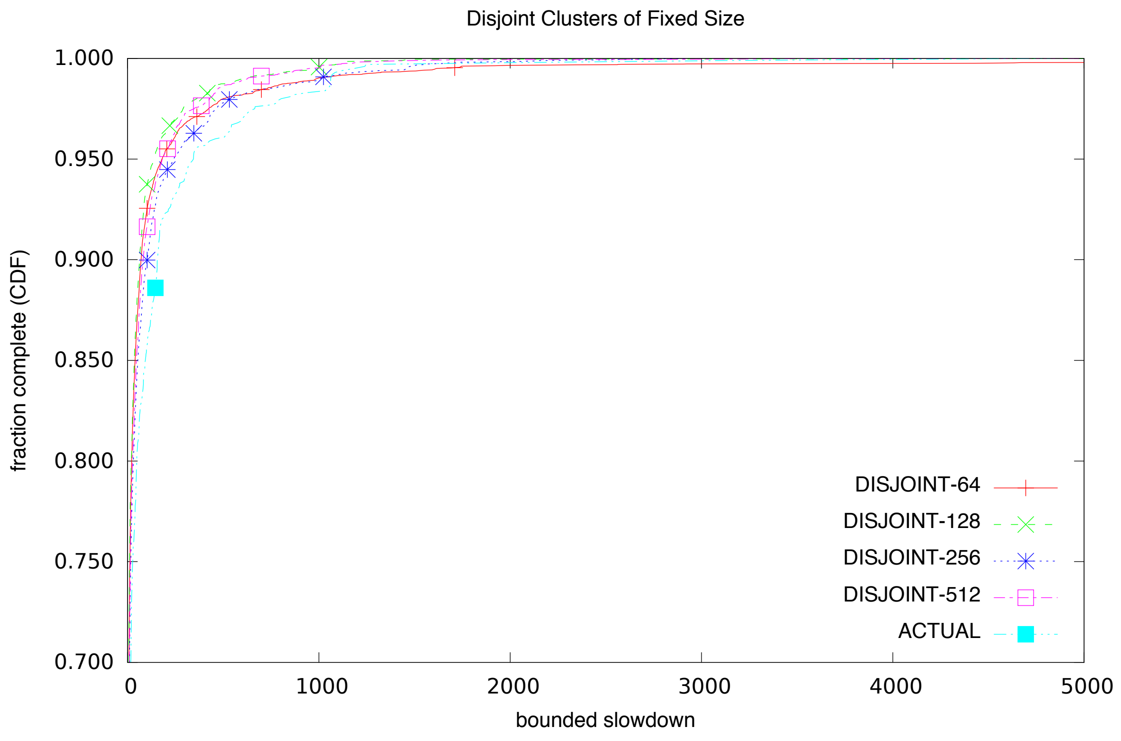


Figure 4.9: Comparing disjoint clusters. Some results extend past a slowdown of 5000, as per Table 4.1.

where job run time is unpredictable.

Second, over-provisioning is important. It limits the inefficiency of a job sitting idle on many nodes, waiting for the last task to start. It also helps jobs start roughly in the order they were submitted. Over-provisioning might be complemented or even supplanted by a backfill scheme, discussed as future work in Section 7.1.

Lastly, clusterings matter. Limiting cluster size is beneficial for performance. This prevents individual jobs from backing up an entire group of resources while it waits for more resources to become free. Surprisingly, overlap also has a positive effect on performance. I speculate that overlap helps performance by creating diversity in the number of different arrangements that form clusters. For instance, in a

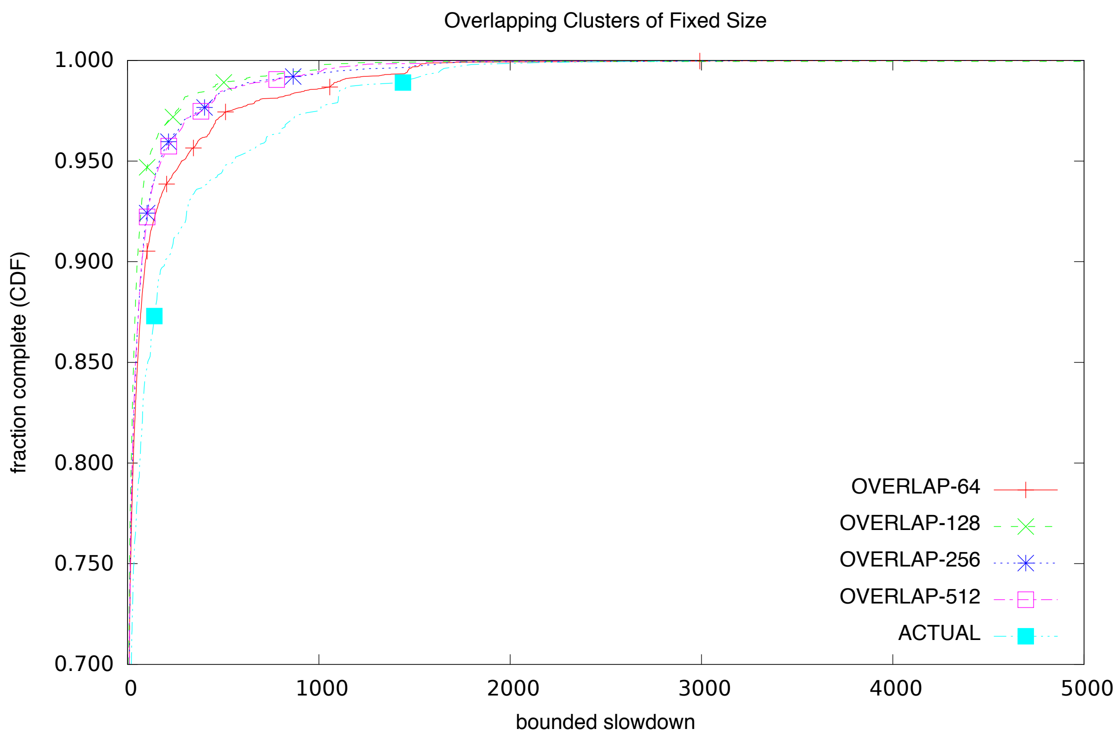


Figure 4.10: Comparing overlapping clusters. Some results extend past a slowdown of 5000, as per Table 4.1.

768-node grid divided in to 64-node disjoint clusters, there are only 12 arrangements in any possible partitioning. However, if the rolling overlap is used as detailed for the experiments, there are 768 different arrangements, one for each node. When cluster workload is fragmented by jobs that require fewer nodes than cluster size, it may be easier to find some cluster that does have adequate free resources when dynamically balancing load. As a hypothetical example, suppose there are 12 63-node jobs running on that 768-node grid. In a disjoint clustering, there are 12 idle nodes, but none of them are in the same cluster. In an overlapped clustering, depending on how the jobs were started, those 12 nodes may be found together and available for parallel computing. The relationship between over-provisioning, cluster size, and job load is still not completely understood.

Chapter 5: Dynamic Clustering

For reasons stated in Chapter 4, clusters need to meet the following three requirements for the scheduling algorithm to do well:

- Good Size: clusters must be as large as possible without being too big. That is, clusters must be as close to the size determined by the user policy, but no larger.
- Bounded Membership: The number of clusters to which a given node belongs must be bounded. Every node is responsible for communicating information about itself to and from the rest of the cluster. To preserve scalability, the number of clusters, and thus the maximum number of nodes with which a given node must communicate, must be bounded.
- Accurate: nodes that fall inside of a given cluster must satisfy the clustering criterion. For instance, nodes that are too far apart could cause a parallel computation to become unusably slow.

The end game of this clustering is to bound the number of nodes to which any given node must communicate while also supporting efficient parallel job scheduling. Here, I will show that an approximate solution to dynamic clustering is possible using a good latency estimation technique, efficient clustering in the latency estimation, and a refinement algorithm to ensure that clusterings have the three properties. The contributions here are the comparative evaluation of three latency estimation techniques and the refinement algorithm.

Unfortunately, an exact solution to clustering is impractical for several reasons. Creating the latency graph requires making and storing $O(n^2)$ measurements for the all-to-all map where n is the number of nodes in the grid. This is not scalable distributed system, but the problem is far worse.

Suppose making and storing an all-to-all map is practical: for the set of all nodes $x, y \in V$, the exact distance between them $\Delta(x, y)$ is known. A cluster meeting some cluster criterion c, then, is a of nodes such that the measurements between all nodes in the cluster are less than c. The problem can be phrased this way: graph $G = (V, E)$ where the edges E are all pairs of nodes (x, y) such that $\Delta(x, y) < c$. In this formulation, clusters are cliques in the graph G. Maximum clusters are maximum cliques, the finding of which is known to be NP-Complete [46]. As per Chapter 4, the scheduling algorithm only needs cliques of a maximum size k. Finding cliques of fixed size k is known to be $O(|V|^k)$ [47]. Thus, finding exact cliques of useful size is also not practical. Instead, an approximation of some sort is necessary.

Fortunately, although networks can be arbitrary, in practice they still have some structure that can be used for approximation. For instance, there has been some attempt to predict latency in networks by embedding nodes in to a metric space. Clearly this would not be a perfect solution, since triangle inequality violations happen in networks. However, it might be good enough so that the properties of the space can be used to efficiently approximate clique finding.

Three latency prediction techniques are evaluated here, including GNP [2], Vivaldi [3], and embedding in a tree metric space (DTM) [4]. These are detailed in Chapter 2. Tree metric space embedding has been found here to be far superior to the others for latency estimation. Despite this superiority, DTM still has mispredictions that cause problems in clustering on that space. These results are discussed further in Section 5.2.

5.1 Refinement

The distributed tree metric space embedding found to be the best predictor of latency also has an associated clustering algorithm that works on the data structure used for prediction [4]. That work is concerned with bandwidth prediction. Here it is trivially adapted to latency prediction by using latency directly as the distance measure. The clustering algorithm finds clusters that have a maximum internal latency of less than or equal to some number. For instance, it might find clusters of 10μs or less, which would capture clusters on dedicated high-performance networks like InfiniBand. This algorithm yields a good first approximation, satisfying the first property: good size. However, those clusters must be refined to ensure that clusters satisfy the other two properties: accuracy and bounded membership. As observed in the experiments, the latency prediction is not perfect and can yield predictions where a handful of nodes are predicted to be much closer together than their actual distance. Furthermore, the clustering algorithm makes no attempt to bound membership. To solve this problem, a refinement algorithm is presented here that takes any clustering approximation and yields usable *job-ready clusters*.

The overall flow of the refinement algorithm is as follows:

1. Limiting Cluster Membership: The first approximation clustering step finds clusters that have the maximum size. However, this does nothing to ensure that a given node is a member of a bounded number of clusters. Thus, the first approximation clusters must be analyzed and pared down to ensure that the membership bound holds.

2. Filtering: Ensure that nodes predicted to be close are actually close. This guarantees the accuracy requirement.

3. Symmetry: The previous two steps may result in clusters that are too small and overlap in ways that hurt job scheduling. For instance, as described in Chapter 4, overlapping clusters can cause interference in the scheduling algorithm. The cluster that any given node finds for itself may not be the largest one to which it belongs. However, because a node is aware of all the other clusters to which it belongs, it can advertise the largest such cluster instead of the one it identified. Symmetry effectively relaxes cluster membership in a harmless way to ensure that individual nodes advertise and use the largest possible clusters. This step helps to meet the good size requirement as well as eliminate pathological overlap.

There is one critical change to the initial clustering algorithm. By default, the number of nodes aggregated from one node to the next is equal to the desired maximum cluster size. However, this can lead to situations where a few nodes are included in many clusters. The Limiting Cluster Membership step eliminates hot-spots like this in the final clustering, but this means the cluster quality is degraded. To counter this, more nodes than are necessary for the maximum cluster size are aggregated to form larger clusters. A random subset is taken from the large cluster to reduce the size down to the maximum cluster size. In practice, aggregating twice as many nodes is enough.

For clarity, the components of the refinement algorithm are explained as a serialized, synchronous process. It is assumed that all of the steps happen globally and in-order. Furthermore, it is assumed that each step itself is serialized. The algorithm can be turned in to a distributed asynchronous process, and this is detailed in Section 5.1.4.

5.1.1 Limiting Cluster Membership

Once the first approximation clustering is complete, each node has a concept of the cluster it can advertise. This group of nodes the *candidate cluster*. Now each node must ensure that no node in its candidate cluster does not belong to too many other clusters. At the end of this process, each node will produce an *unfiltered cluster*. To this end, a node will generate and send a list of solicit messages to each member of its candidate cluster. A node that sends solicitations is the *suitor*. At the end of the process, a node should be a member of no more than M clusters. Like the maximum cluster size N, M is a value set by policy. No assumptions about the order of the arrival of these messages are made.

Algorithm 3 solicit(from, to, last): Solicitation Message Handling

```
from is the suitor
to is the recipient
last indicates whether this is the last solicitation that a recipient will receive
to.suitors.append(from)
if to.suitors.size > $M$ then
    to.suitors.sort(criteria)
    suitors.end.delete
end if
if last then
    for all suitor $\in$ suitors do
        accept(to, suitor)
    end for
end if
```

The solicit Algorithm 3 illustrates how these messages are processed. The soliciting node *from* is added to a candidate node's list of suitors. The suitors are then sorted according to some criterion. If the suitor list is too long, the node deemed least desirable by the sorting criterion is deleted. Finally, once all solicitations are received, the candidate node sends accept messages to all suitors in the list.

Three different sorting criteria are investigated here: FCFS, global ordering, and greedy. FCFS is essentially no sort. Suitors are accepted in solicitation order until the maximum cluster membership is achieved. Global ordering sorts nodes based on a consistent hash of the suitor identity. The intent of global ordering is to emulate an ordering of solicitation messages across all candidate nodes with the constraint that only constant size information about suitors is stored. The greedy strategy relies on updates from suitors about accepted and rejected solicitations to adjust how suitors are sorted so that nodes join clusters that are more likely to be large. This strategy is somewhat more complicated. In order, it gives sorting precedence to the following measures, where the first is the most important and the subsequent ones are used to break ties:

1. Suitors with a higher ceiling size on their respective clusters. That is, the total number of advertisements minus the number of rejections. Thus, preference is given to potentially large clusters first.

2. If two nodes have equal ceiling, the strategy prefers clusters that have more accepted solicitations.

3. Finally, if all other factors are equal, it uses a consistent hash as per the global ordering heuristic to break ties.

At the end of this process, any given node x will be a member of no more than M clusters. For the rest of this work, it is assumed that $M = N$. Recall that N is a fixed value set by policy. If all M clusters are disjoint except for x, then x knows about no more than $M * (N - 1)$ other nodes. This is important for the filtering step.

5.1.2 Filtering

Though the latency prediction algorithm is very good, it is still not perfect. Situations where two distant nodes appear close together in the estimation tree have been observed experimentally. For this reason, a measurement between every pair of nodes in a given cluster must be made. Thanks to cluster membership limitation, any given node only needs to make measurements to $M * (N - 1)$ other nodes.

Once all measurements are made, each individual node needs to filter out nodes that are too far away. Effectively, after making the all-to-all measurement within the unfiltered cluster, an individual node looks for the maximum set of neighbors that are actually close to itself and each other. Unfortunately, this is the clique problem, which is NP-complete: A cluster is a graph where each node in the cluster corresponds to a node in the graph, and there is an edge between two nodes only if the measured distance falls inside the cluster diameter. Fortunately, the results in practice are either all entirely within the cluster diameter or entirely outside. These are easy properties to check. Failing that, efficient exact solutions exist for smaller graph sizes [48]. This step produces a *filtered cluster* at each node.

5.1.3 Symmetry

The cluster limitation process can still yield undersized clusters. However, the cluster that a node finds may not be the largest cluster to which the node belongs. Through the cluster limitation process, all nodes are aware of all clusters to which they belong. So instead of scheduling tasks on to its own detected cluster, any node can schedule instead the largest cluster to which it belongs, effectively adopting the larger cluster as its own. This largest cluster is the job-ready cluster that the node will advertise in to the CAN.

This can effectively increase the number of clusters to which a node belongs. However, it does not increase the number of nodes with which a given node must regularly communicate. A detailed proof of this property is in Appendix A. That being said, symmetry is not free. Though the number of nodes with which a given node might communicate does not increase, the number of nodes that may potentially send tasks to a node will.

5.1.4 Distributed Algorithm

Making the algorithm distributed follows the conventional process of dividing the parts in to a state machine. *Heads* form clusters by soliciting nodes and filtering out false positives. *Clients* limit the number of clusters they join and measure inter-node latency. Every node acts as a head as well as a client.

Clients have four states: IDLE, WAIT, MEASURE, and STEADY. Figure 5.1 illustrates the state machine. The IDLE state represents the initial state of the client before it has joined any peer-to-peer overlays. When the client is ready to start joining clusters, it enters the WAIT state. Clients in the WAIT state collect solicitations and updates from heads. When a client enters the WAIT state, it starts a timeout. Every time a valid new solicitation appears, the timeout is reset. The client arranges its internal list of suitors as per the synchronous algorithm.

This may include sending rejection messages to heads. When the timeout triggers, the client sends accept messages to heads that remain on the list of suitors and transitions to the MEASURE state. If no solicitations have arrived, the client merely resets the timeout and remains in the WAIT state. In the MEASURE state, a client measures inter-node latency to all nodes in its unfiltered cluster, and then forwards this information to any concerned head. The client responds to new solicitations received during the MEASURE state by instructing the suitor to defer the solicitation to a later time. Finally, once all measurements are complete and the client receives the final confirmation message from each head, the client transitions to the STEADY state. A new solicitation received in the STEADY state will cause the client to go back to the WAIT state if the client has not already committed to its full allocation of clusters. Once committed to a head, a client will not abandon that cluster until the head leaves voluntarily or a fault is detected by timing out.

Heads have five states: IDLE, COLLECT, SOLICIT, FILTER, and STEADY. Figure 5.2 illustrates the state machine for heads. Similar to the client IDLE state, the head IDLE state represents the initial state of the head. The head is in the COLLECT state while it gathers information from the distributed tree metric space and forms its own first-approximation cluster. When the head receives no new information about close neighbors for a fixed number of seconds, it considers the network stable. Empirically, a thirty second timeout has been found to be adequate when looking for clusters of up to 64 nodes. Then it runs the clustering algorithm and enters the SOLICIT state by sending solicitations to all clients in its discovered cluster. As the head receives accept and deny messages from solicited clients, it

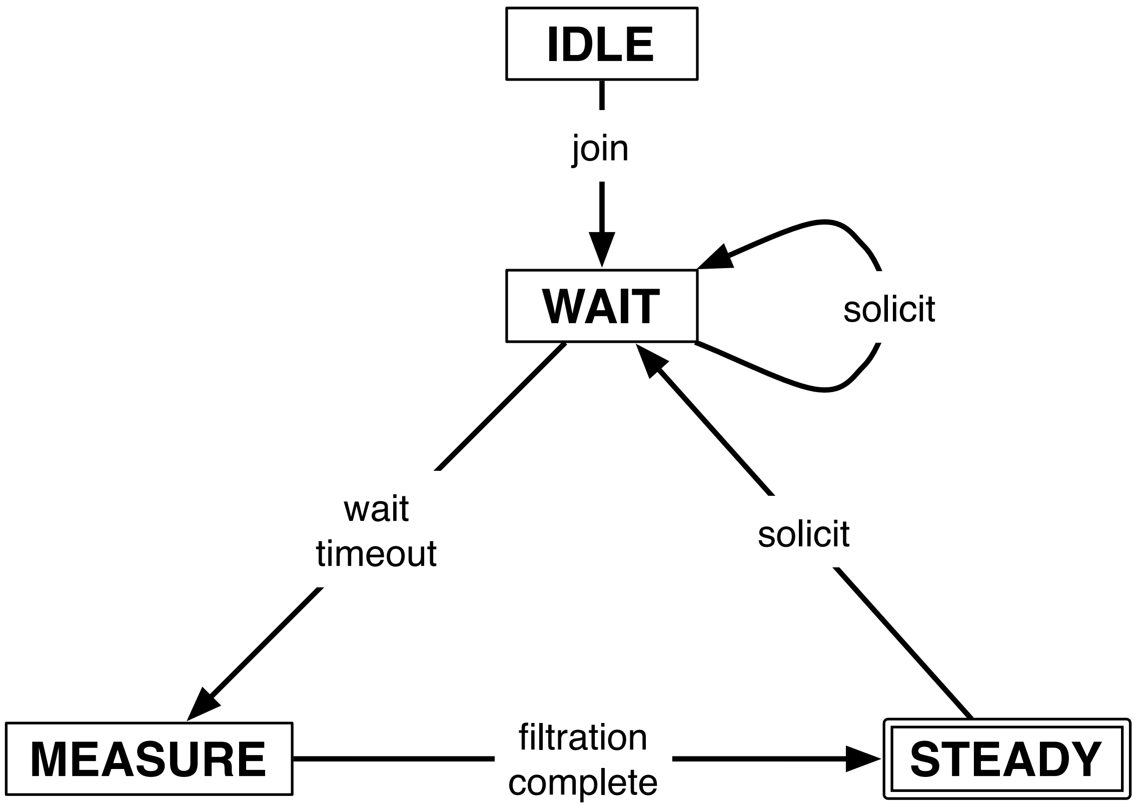


Figure 5.1: State machine for clients.

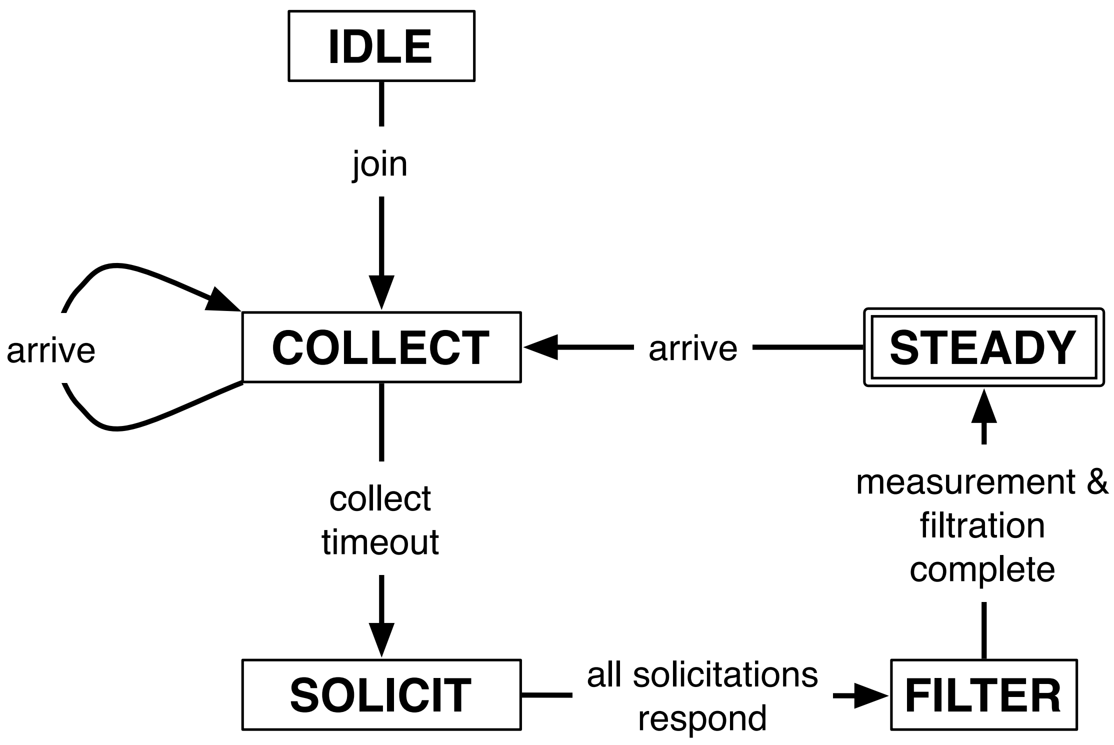


Figure 5.2: State machine for heads.

sends updates to any clients which have not denied the solicitation. Once the head has heard either way from all clients, it enters the FILTER state and waits until all accepted nodes have replied with their latency measurements. The head uses the filtering heuristic to find the largest clique that includes itself. This clique forms the job-ready cluster. Finally, the head sends a final confirmation message to all clients in the job-ready cluster, denial messages to any clients outside the cluster, and enters the STEADY state. Once in the STEADY state, the head can react to new client arrivals observed since the head entered the SOLICIT state, and may cause the head to re-enter the COLLECT state if the head does not have a maximum size cluster.

An important aspect to all of this is change. Networks are not static entities.

Nodes leave voluntarily or fail unexpectedly. These departures may be permanent or transient. New nodes can be brought online, sometimes close enough to exist in the same cluster as existing resources. The interconnection latencies themselves change as network equipment is added, removed, or upgraded. The distributed refinement algorithm presented here is an online algorithm, which adapts to these changes as they happen. However, the refinement algorithm is ultimately dependent on the information it gets from the prediction framework. That is, the prediction framework itself must also adapt to change. For DTM, the suggested technique is to make actual measurements over time to several other participants in the DTM overlay. If the measurements drift too much, the node itself must voluntarily leave and rejoin the DTM to maintain prediction quality.

5.2 Experiments

The experiments in this chapter have two purposes. First, existing latency prediction techniques are evaluated for their accuracy and their limitations are discussed. Two datasets are used to evaluate latency prediction: the UMIACS and SHARCNET latency datasets. Second, the refinement algorithm for dynamic clustering is compared to static clustering both with and without symmetry.

The measure used for gauging the accuracy of prediction techniques is their *relative error*. Where the actual distance between two nodes is $\Delta(x, y)$ and the predicted distance is $\delta(x, y)$, the relative error is $|(\Delta(x, y) - \delta(x, y))/\Delta(x, y)|$. The techniques are compared in terms of relative error percentile. For the nth percentile,

$n\%$ of the pairs have a relative error less than the stated relative error. For instance, the DTM technique has a relative error of 0.37 at the 99th percentile. This means that 99% of pairs had a relative error of less than 0.37. The charts are all the cumulative distribution of relative error.

5.2.1 UMIACS Latency Dataset

The first is a real dataset collected from the computing facilities of the University of Maryland Institute for Advance Computer Studies (UMIACS) by the author. This dataset is referred to here as *UMIACS latency.* It consists of 151 nodes divided in to several clusters. Its small size and few resources for tightly-coupled parallel computing limit its utility for modeling parallel scheduling in a grid. However, there is nothing synthetic about this dataset, and all measurements were made using MPI. For this reason, it is useful for discussing the latency prediction techniques themselves. The collection of this dataset is detailed in Appendix B.

5.2.2 SHARCNET Latency Dataset

The second dataset is synthetic, intended to emulate a realistic network configuration for the SHARCNET grid. This dataset is referred to here as *SHARCNET latency.* As discussed in the previous chapter, the SHARCNET grid consists of 2091 nodes allocated to 10 clusters of varying size and capability. This grid is important because it has a satisfactory job load dataset for scheduling, thus giving a basis for comparison to the static clustering explored in the previous chapter.

Three supplementary datasets were used to create this all-to-all latency dataset. I made a subset of a PlanetLab dataset [49] such that every node in the subset had at least one measurement to another node in the PlanetLab set. From this subset, I chose a smaller subset randomly with one PlanetLab node corresponding to one of the clusters modeled in the final dataset. The PlanetLab latency measures, then, were used to model the inter-cluster latency in the network. For instance, between any two nodes not in the same cluster, I modeled the latency using the corresponding pair in the PlanetLab network. These measurements were on the order of tens of *milliseconds*. I created two datasets to model intra-cluster latency. For modeling the clusters connected with Myrinet-2000, I made an all-to-all MPI latency measurement of a cluster using that same type of interconnect. These measurements were consistently between 7 and 8 *microseconds*. For the single large cluster connected with Gigabit Ethernet, I used MPI measurements made between nodes connected with that technology instead. These were consistently between 100 and 150 *microseconds*. The resulting synthetic model has many decimal order of magnitude differences between the highest and lowest latencies. This accurately reflects the latency structure of a geographically distributed grid. These huge differences in latency measures will become important when discussing the utility of various latency prediction techniques.

5.2.3 Latency Prediction

For finding resources in a decentralized grid, latency prediction should also be decentralized. Vivaldi is a well-known decentralized latency prediction technique, and its source is freely available, so it was the one initially used in the development of the decentralized scheduler. The other fully decentralized technique compared here is the DTM, which has previously been used to predict bandwidth. Finally, GNP is another well-known latency prediction technique. It uses fixed landmarks, so it is not fully decentralized. However, it forms a basis of comparison since it may outperform decentralized techniques.

5.2.3.1 UMIACS Latency Dataset

The first experiment compares these latency prediction techniques, GNP, Vivaldi, and DTM, using the UMIACS latency dataset. The experimental setup for Vivaldi is as follows:

- Embedding in to the default 5D Euclidean space.
- 128,000 rounds of measurement and adjusting. This value was established by running the Vivaldi simulator with several different values for the number of rounds, with the intention of finding the point where more rounds have little effect on increased accuracy. The result of this benchmark is in Table 5.1. The value chosen here is larger than $151^2 = 22,801$, so far more than n^2 measurements are being made here in each Vivaldi simulation.

Rounds	Relative Error at 99th %ile
4000	13.86
8000	11.85
16000	4.10
32000	1.36
64000	0.69
128000	0.69
256000	0.67
512000	0.65
1024000	0.67

Table 5.1: Relative error at the 99th percentile of the cumulative distribution of the error for a given number of rounds. This table was used to select the number of rounds used when running Vivaldi for comparison with GNP and DTM.

- Default dampening (0.25). Dampening proportionally affects the amount that a point will move after a round.
- 100 different seeds for the pseudo-random number generator, to ensure that any particular seed does not produce particularly good or bad results.

The experimental setup for GNP is as follows:

- Embedding in to a 5D Euclidean space, comparable to Vivaldi.
- Ten different landmark selections of six fixed landmarks each, chosen at ran-

Rounds	50th %ile	95th %ile	99th %ile	maximum
DTM	0.019	0.209	0.492	21.75
VIVALDI	0.122	0.467	0.701	6.31
GNP	0.235	0.891	0.995	29.66

Table 5.2: Relative error at several percentiles for DTM, GNP, and Vivaldi on the UMIACS dataset. The x-axis extends from 0 to 2, but the tails extend in some cases to almost 30.

dom.

Finally, the experimental setup for DTM is as follows:

- Ten different randomly-selected join orders. Join order plays a critical role in the resulting estimation trees.
- Ten different seeds for the pseudo-random number generator.

The results are in Table 5.2 and Figure 5.3. Table 5.2 shows the relative error at the 50th, 95th, and 99th percentiles. Figure 5.3 shows the cumulative distribution function for relative error for each of the latency estimation techniques. In this case, Vivaldi does a better job than GNP in the small network of 151 nodes. However, DTM is clearly superior to either of the Euclidean space embeddings.

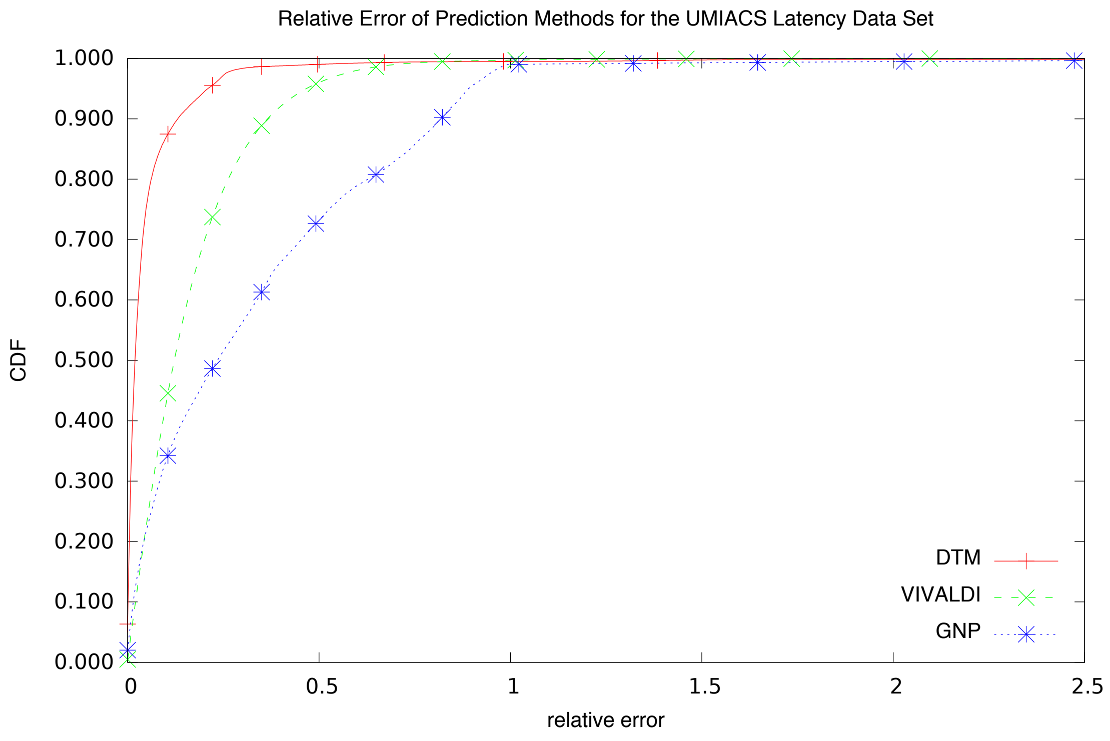


Figure 5.3: Comparing relative error for DTM, GNP, and Vivaldi on the UMIACS dataset.

5.2.3.2 SHARCNET Latency Dataset

Next, the three techniques are compared using the synthetic SHARCNET latency dataset. An important distinction here is that the SHARCNET latency dataset is composed of ten different all-to-all maps, so each technique needs to be tested against all ten of these arrangements.

Vivaldi setup:

- Embedding in to the default 5D Euclidean space.
- Two different numbers of rounds were used to exercise Vivaldi. First, 4,096,000 rounds are used to give Vivaldi the best possible results. Again, this value is

determined empirically using the same process as the number of rounds used for the UMIACS dataset (see Table 5.3). This is close to n^2 rounds, so this process is not realistically scalable. These results are called VIVALDI-n^2. Second, the DTM experiments made 117,941 measurements, on average. This value of rounds are used to model more realistic latency prediction, effectively giving Vivaldi the same number of opportunities to make adjustments that DTM uses. Since DTM makes a number of measurements proportional to $n \log n$, these results are called VIVALDI-$n \log n$.

- Default dampening (0.25). Dampening proportionally affects the amount that a point will move after a round.
- Ten different seeds for the pseudo-random number generator, to ensure that any particular seed does not produce particularly good or bad results. With the ten different inputs, this results in 100 total runs for each value of the number of rounds.

GNP setup:

- Embedding in to a 5D Euclidean space, comparable to Vivaldi.
- 6 fixed landmarks, chosen at random.
- 10 different landmark selections. With the 10 different inputs, this results in 100 total runs.

DTM setup:

Rounds	50th %ile	95th %ile	99th %ile
32000	0.55	16888.64	23511.47
64000	0.39	16918.67	24360.85
128000	0.14	8797.83	14023.01
256000	0.08	1050.46	2443.66
512000	0.07	398.50	945.29
1024000	0.02	224.46	584.25
2048000	0.02	190.05	448.44
4096000	0.01	97.00	238.90
8192000	0.01	84.21	241.49
16384000	0.01	81.02	235.71
32768000	0.01	84.16	239.48
65536000	0.01	75.73	232.91

Table 5.3: Relative error at several percentiles for a given number of rounds in the Vivaldi simulation. This table was used to select the number of rounds used when running Vivaldi for comparison with GNP and DTM on the SHARCNET latency dataset.

- 10 different randomly-generated join orders.
- With the 10 different inputs, this results in 100 total runs.

The results are in Table 5.4 and Figure 5.4. Table 5.4 shows the relative error at the 50th, 95th, and 99th percentiles. Figure 5.4 shows the cumulative distribution

Rounds	50th %ile	95th %ile	99th %ile	maximum
DTM	0.00	0.11	0.37	10,289.74
GNP	0.19	1.94	4.65	84,287.44
VIVALDI-n^2	0.01	97.00	238.90	8,077.81
VIVALDI-$n \log n$	0.20	5,031.89	13,483.07	89,768.45

Table 5.4: Relative error at several percentiles for DTM, GNP, and the two Vivaldi variations. The y-axis extends from 0.7 to 1.0. The x-axis extends from 0 to 100, but the tails extend in some cases to almost 90,000.

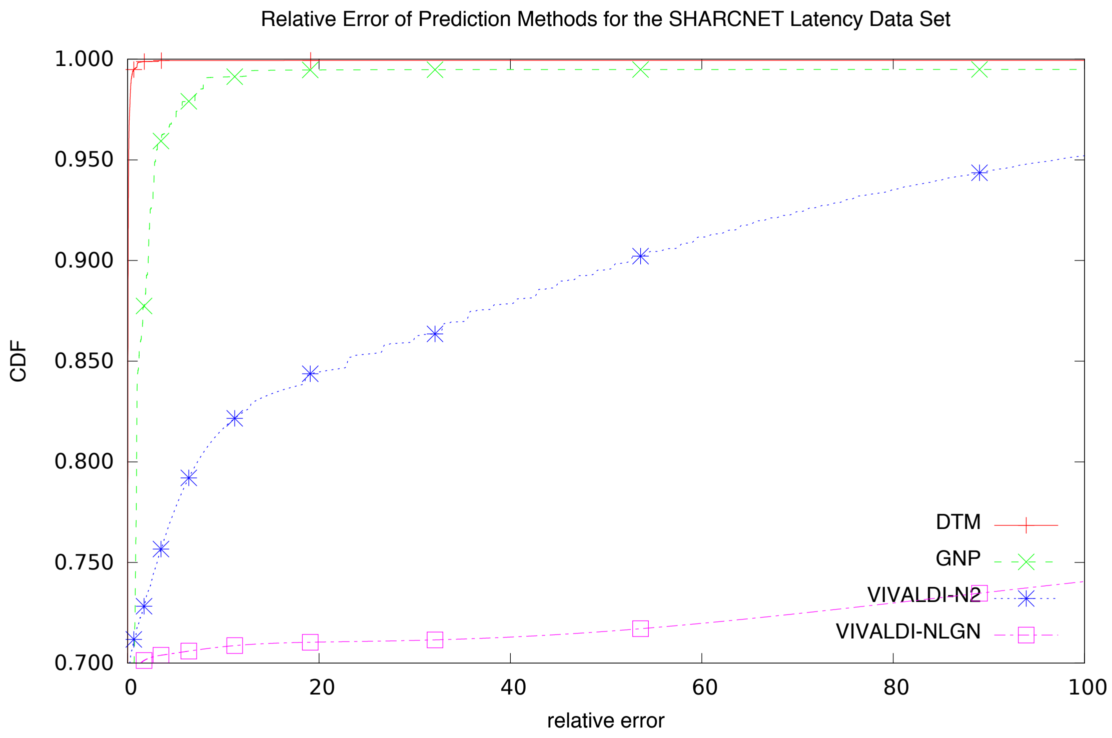


Figure 5.4: Comparing relative error for DTM, GNP, and the two Vivaldi variations.

function for relative error for each of the latency estimation techniques.

DTM is clearly the superior prediction technique. This technique did not do any worse at the 50th, 95th, and 99th percentiles than it did with the smaller UMIACS dataset, suggesting that prediction accuracy will scale with the size of the network. GNP is usable, but does not have the same persistence of quality with the larger dataset. Vivaldi, however, struggles mightily beyond the 70th percentile. The reason for this is fairly straightforward: Vivaldi makes adjustments to coordinates based on absolute error multiplied by the dampening. This is a serious problem when there are order of magnitude latency differences between close neighbors and distant nodes. If, for a given node, there are many more distant nodes than close nodes, then most measurements are against distant nodes. As a result, it becomes impossible for Vivaldi to keep close nodes together since any adjustment against a distant node overwhelms locality for close neighbors. Adjusting the dampening only helps so much. Effectively, the latency distance between close nodes is consistently overestimated by a large margin. This makes the actual results for clustering much worse than portrayed in these graphs: Finding clusters is almost impossible because close nodes are rarely predicted as such. GNP also overestimates latency, but the magnitude of the error is not as bad. For this reason, DTM is the technique chosen to use as input for clustering and refinement.

The premise of GNP and Vivaldi are that, because computers themselves are embedded in a reasonable approximation of a Euclidean space, then using a Euclidean space to predict their relative positions is a useful concept. Euclidean space is a metric space and, as such, requires that the triangle inequality hold. In contrast, DTM uses a tree-metric space to predict distance. In a tree-metric space, the more restrictive four-point condition must hold. Every violation of the triangle inequality is also a violation of the four-point condition. Some violations of the four-point condition do not violate the triangle inequality. Despite this fact, DTM outperforms GNP and Vivaldi, sometimes by a very wide margin. This suggests that embedding in a Euclidean space may actually be a poor model for predicting distances in the Internet.

5.2.3.3 Error in DTM

Bad over-estimation of latency in DTM is caused by *partitioning.* Partitioning happens when two nodes that are close in latency end up in different parts of the prediction tree. Some nodes make a wrong turn and get added to the prediction tree far away from where other nodes in that same cluster are added. These errors are the result of four-point condition violations that cannot be captured by the underlying embedding. Simulated joins, as discussed in the background section on DTM, mitigate the problem, but it cannot prevent all partitioning. This is why join order and initial join location have a strong effect on the resulting accuracy of the prediction tree. The consequence for overestimation is underutilization. Because clusters are sometimes split up in to smaller pieces, a larger job may not fit where it otherwise should be able to.

Bad under-estimation of latency happens when nodes are misplaced in the prediction tree, also due to four-point inequality violations. A node is inserted in to the prediction tree such that the distance from the leaf vertex representing the node to the closest interior node of the prediction tree is far too small. This results in a chain of nodes being inserted down the branch of the prediction tree until the distances between the interior nodes sum up to a reasonable number. Then the prediction tree can grow as normal from that point forward. This typically results in false clusters where none of the nodes involved have an inter-node latency useful for tightly-coupled parallel computing. If a parallel job gets scheduled on a cluster with such an anomaly, it could cause all involved nodes to grind to a virtual halt while the job struggles with an inter-node latency measured in milliseconds instead of microseconds.

Both of these problems arise due to the imperfect embedding. If all of the distances in the input satisfy the four-point condition, then they can be perfectly embedded in the prediction tree. However, the input does not satisfy this condition, leaving cases where the embedding fails.

Several techniques are tried where the input is altered to get the best results:

- Rounding: Distances are rounded to the nearest power of two when they are greater than 64μs. This is an attempt to eliminate smaller violations of the four-point condition happening at the large scale. It is not important that the predictions are precise at that scale, since they are not useful for tightly-coupled parallel processing.

- Lowest-Quartile Filtering: Instead of using the average of all measurements between a given pair of nodes, average only those in the lowest quartile. Filtering the input data is an established method of improving prediction results [49]. Removing high latency measurements takes out anomalous situations where the measurements were hung up for any number of reasons, including extra traffic over intervening network connections, high load on either end, or other causes of high latency. These anomalous measurements can distort the average measure, which is why they are discarded in this case.

- Rejoining: After the initial join, nodes leave the DTM prediction tree then rejoin. Early embedding errors might be fixed by removing the nodes responsible for the early errors and re-inserting them in to a more mature tree. This is different than the multiple simulated join strategy in that rejoins do not occur until after all participating nodes have already joined and are participating fully in the tree overlay. Rejoining has the drawback of intentionally introducing churn in to a decentralized peer to peer system, so this is not necessarily a good strategy. It also uses global knowledge to wait until all nodes are joined to the tree, and then serializes departures and rejoins. A decentralized approach where nodes make individual decisions about when to rejoin may not get the same results.

These three variations are compared to the original runs in Table 5.5 and Figure 5.5. The results are mixed. Rejoining clearly has the best upper bound on error, topping out at around 2. This is an amazing result, considering that the others have a maximum error on the order of 10^4. This strategy eliminates both partitioning and underestimation errors. It lags behind the others at the 50th, 95th, and 99th percentiles. The others are all roughly the same.

However, this does not tell the whole story. The primary purpose of latency

DTM Variation	50th %ile	95th %ile	99th %ile	maximum
Lowest Quartile	0.00	0.089	0.35	8,077.81
Rejoin	0.00	0.28	0.50	2.06179
Round	0.00	0.090	0.31	9,329.09
Original	0.00	0.11	0.37	10,289.7

Table 5.5: Relative error at several percentiles plus the maximum relative error for the variations on DTM.

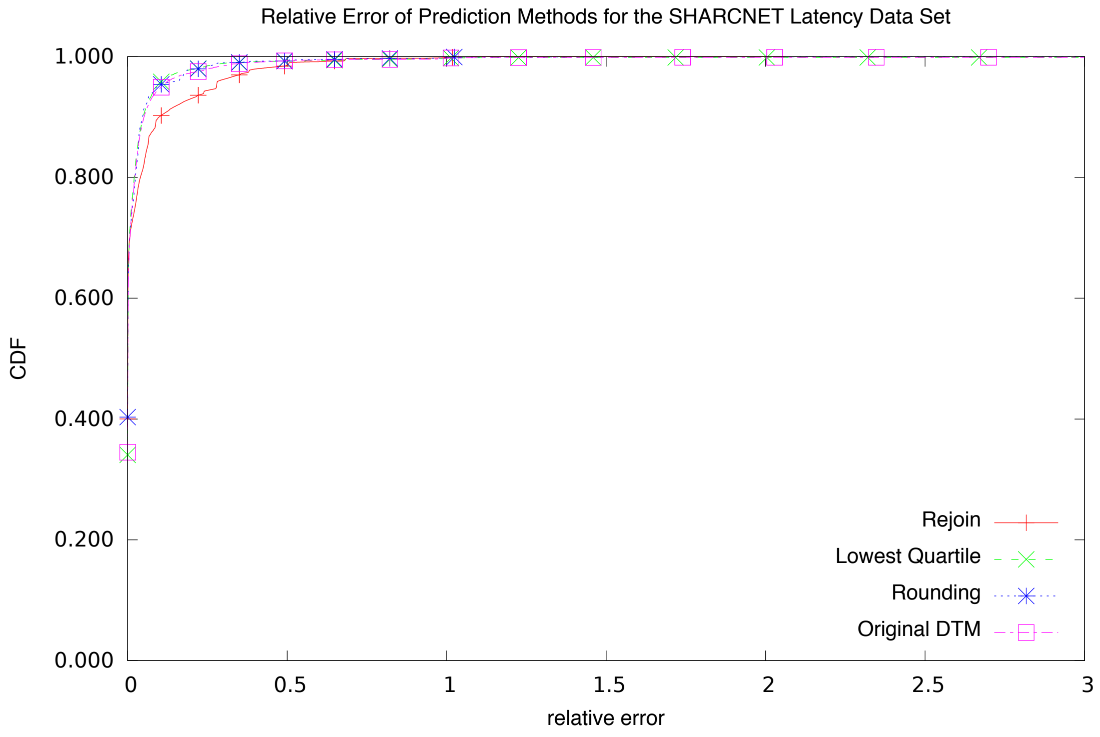


Figure 5.5: Comparing relative error for variations on DTM. The x-axis ranges from 0 to 3, but the tails for Lowest Quartile, Round, and Original extend out to about 10^5.

DTM Variation	Positive Error	Negative Error
Lowest Quartile	0.99×10^{-5}	0.030
Rejoin	0	0.022
Round	1.0×10^{-5}	0.013
Original	1.4×10^{-5}	0.022

Table 5.6: Error rates for positive and negative errors at the 10μs boundary. That is, the error rates for predicting a given distance as less than or equal to 10μs when it is greater (positive error), or predicting a given distance as more than 10μs when it is less than or equal to that value (negative error). Since clustering is done at fixed latency diameters, this gives a good estimate about how often errors will appear in clusters.

prediction is to find clusters of nodes that are close together. So, the real question is not how good are predictions over all, but how good are they when the nodes are close together? Table 5.6 shows error rates for positive and negative errors at the 10μs boundary. The value 10μs is used because all high-performance interconnects measured here have latency under 10μs: The measured average latency for Myrinet and InfiniBand are 7.2μs and 6.6μs. In this case, only latencies from Myrinet are used in the simulated SHARCNET grid. Setting the value to 10μs allows for a modest amount of overestimation.

The positive error column indicates the rate of underestimation, when a distance is predicted to be less than or equal to 10μs though it is greater. The negative error column indicates the rate of overestimation, when a distance is predicted to be greater than 10μs when it is less than or equal to that value. Since clustering is done at fixed latency diameters, this gives a good idea about how often errors will appear in clusters. For all but Rejoin, about 20 underestimations would be expected in the simulated grid consisting of 2091 hosts. This is one reason the refinement algorithm is absolutely necessary, since prediction without positive errors cannot be guaranteed.

5.2.4 Cluster Refinement

The most direct way to examine the utility of the refinement algorithm is to compare scheduler performance on dynamically discovered versus the static clusters from Chapter 4. The experimental setup for this experiment is very similar. The job trace is the time-compressed SHARCNET trace used in most experiments from Chapter 4. Both the queue balancing and dynamic load balancing heuristics were used in all simulations for this experiment. Dynamic clustering is a two step process: First, approximate clusters are found using the DTM simulator with the clustering algorithm enabled. Unfortunately, cluster size was limited to 64 nodes, here, due to the limitations of simulation. That is, finding clusters any larger than 64 members in a grid of 2091 nodes were infeasible to simulate on a single computer using the DTM simulator: Running the DTM simulator takes about 1 week for 64-node clusters, and would take about 4 weeks for 128 node clusters. Second, the output of that simulation is used as input for a simulator implementing the distributed refinement

Clustering	Mean	Median	Mode	95th %ile	99th %ile	Maximum
DISJOINT-64	68.2	2	2	180	992	25147
OVERLAP-64	55.2	2	2	285	1114	4936
DYN-64	313	7	2	2392	6820	19632
DYN-SYM-64	86.4	3	2	207	1739	17957

Table 5.7: Bounded slowdown statistics for static and dynamic clustering arrangements.

algorithm. The refinement algorithm was run both with and without symmetry, to illustrate the necessity of that step. So, in addition to the DISJOINT-64 and OVERLAP-64 clusterings, there are now two more clusterings to examine: DYN-64 and DYN-SYM-64. DYN-64 is the result of dynamic clustering and refinement without symmetry. DYN-SYM-64 is the result of dynamic clustering and refinement with symmetry. The clusters resulting from the refinement algorithm were then used as input to the scheduling simulator.

Results from the experiment are in Table 5.7 and Figure 5.6. DISJOINT-64, OVERLAP-64, and DYN-SYM-64 all have comparable performance, with the static clusterings having a slight but not unexpected edge on DYN-SYM-64. DYN-64, in contrast, performs very poorly. This highlights the necessity of the symmetry step to finding useful clusterings for the scheduling algorithm.

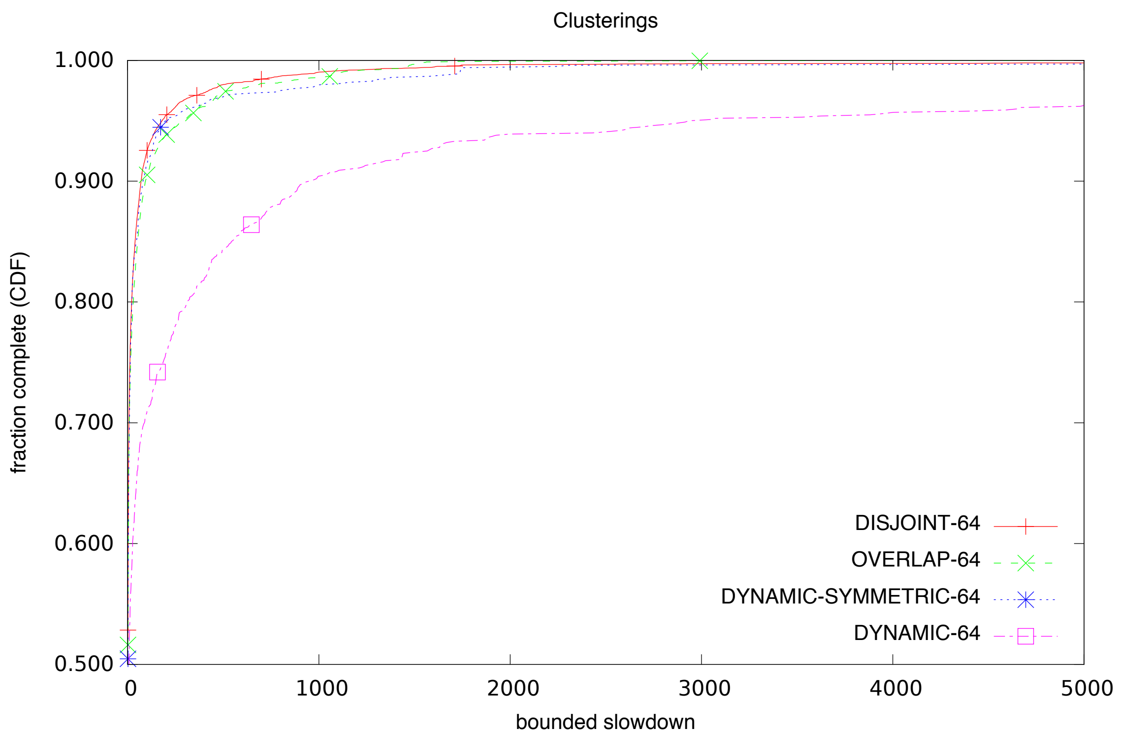


Figure 5.6: Comparing dynamic clustering to static clustering. The slowdown for most jobs is very close to 1, so the y-axis only spans 0.5 to 1. Some results extend past a slowdown of 5000, as per Table 5.7.

5.2.5 Summary

These experiments show three important points. First, that DTM is very accurate compared to the prediction techniques that use a Euclidean embedding. I started off using Vivaldi, which was adequate for small simulated grids (200 nodes). I had built up support for Euclidean space embeddings, including support for using CAN as a decentralized index for querying in the embedded space. However, when scaling up to 2091 nodes for the SHARCNET dataset, Vivaldi was found to be wholly inadequate and all that work was discarded. The cause for this issue is the use of absolute error for adjusting coordinates, as discussed in Section 5.2.3.2. Second, despite the much better accuracy from DTM, underestimation cannot be wholly prevented. Parallel processes scheduled on nodes that are not actually close in terms of latency can cause parallel processes to run unusably slow. No latency estimation technique guarantees the absence of underestimation. Thus, along with bounding other information that a node is required to keep about other nodes, underestimation motivates the need for the refinement algorithm. Thirdly, scheduling on dynamically discovered clusters is comparable to statically assigned clusters provided that symmetry is used so that the best possible clusters are advertised.

Chapter 6: CAN: Improvements and Observations

In the course of working with CAN, I have run afoul of several situations where the CAN algorithms as first described in Ratnasamy et al. [5] and later fleshed out in the associated thesis [22] are under-specified. I discovered these problems by pushing CAN with high churn activity. My work is not the first time CAN has been revisited and improved. In particular, the split list data structure [6] is a great advancement in improving the robustness of CAN.

Before proceeding, some CAN terms need to be defined:

- **Space** The CAN space is the multidimensional range of coordinates indexed by the CAN. The space is Euclidean, and the coordinate system is Cartesian. The space is finite, where each dimension has a lower and upper bound. In the original CAN description, the dimensions are meaningless, merely ranges to which values can be hashed. In P2PGrid, each dimension represents some computational capability, such as RAM or the number of cores in a computer.

- **Axis-aligned half-space** An axis-aligned half-space is an ordered tuple $(d, p, b) \in \mathbb{N} \times \mathbb{R} \times \{<, \geq\}$. The dimension d is a dimension in the CAN space, p is a point in that dimension, and b is the relation determining whether p is an upper or lower bound. That is, the axis-aligned half-space (d, p, b) is all points $\mathbf{x} \in \mathbb{R}$

such that $x_d < p$ or $x_d \geq p$, depending on the bound. All half-spaces used here are axis-aligned, so axis-aligned half-spaces will be referred to simply as half-spaces.

- **Zone** A zone is an axis-aligned bounding box in the CAN space. For instance, if the space is two-dimensional, then each zone is a filled rectangle. Ideally, the gamut of the CAN space is filled with disjoint zones such that each zone is bordered on all sides by other zones or the bounds of the space. However, if there are node failures, there can be gaps or overlaps. A zone is a convex, fully-enclosed intersection of half-spaces. Each of these half-spaces forms a zone face. For instance, $(0, 0, \geq) \cap (0, 1.0, <) \cap (1, 0, \geq) \cap (1, 1.0, <)$ is the unit square in two dimensions. This zone can also be expressed more succinctly as an ordered set of ranges: $\{[0, 1.0), [0, 1.0)\}$.

- **Node** A node is a single computer participating in the CAN. Each node is represented by a coordinate in the space. In the original CAN description, the coordinate is randomly determined. In P2PGrid, the point is a resource coordinate representing a node's computational capabilities. Each node participating in the CAN has a zone that contains the coordinate. Nodes are aware of their neighbors, which are nodes that control zones next to the node's own zone. These are the node's direct neighbors. Nodes are also aware of their neighbors' neighbors, also called their indirect neighbors. A node may have a neighbor that is both a direct neighbor and an indirect neighbor.

- **Split** When a new node joins the CAN, it is routed across the CAN to the zone that contains the new node's coordinate. The zone containing the coordinate splits in to two new zones, each one containing one of the nodes' coordinates. The splitting node's new zone is the old zone intersected with a half-plane that contains the splitting node's coordinate. The new node's zone is the old zone intersected with the opposing half-plane that contains the new node's coordinate. All neighbors are notified of the change. The old node copies its split list to the new node. Then each node updates its split list, which is defined next, to record the addition of the new half-plane. This half-plane at the end of the split list is now the most recent entry. For instance, suppose the unit square zone splits in to two zones at 0.5 along the zero dimension. This split forms zones $A_z = (0, 0, \geq) \cap (0, 0.5, <) \cap (1, 0, \geq) \cap (1, 1.0, <)$ and $B_z = (0, 0.5, \geq) \cap (0, 1.0, <) \cap (1, 0, \geq) \cap (1, 1.0, <)$, controlled by nodes A and B. Figure 6.1 illustrates how new node is routed to its point in the CAN space, and a zone splits to complete the join.

- **Split List** A zone's split list is the list of half-spaces indicating the sequence of zone splits that occurred to result in the current zone. For instance, in the split unit square example from the previous definition, $(0, 0.5, <)$ will be pushed on to the split list for zone A_z.

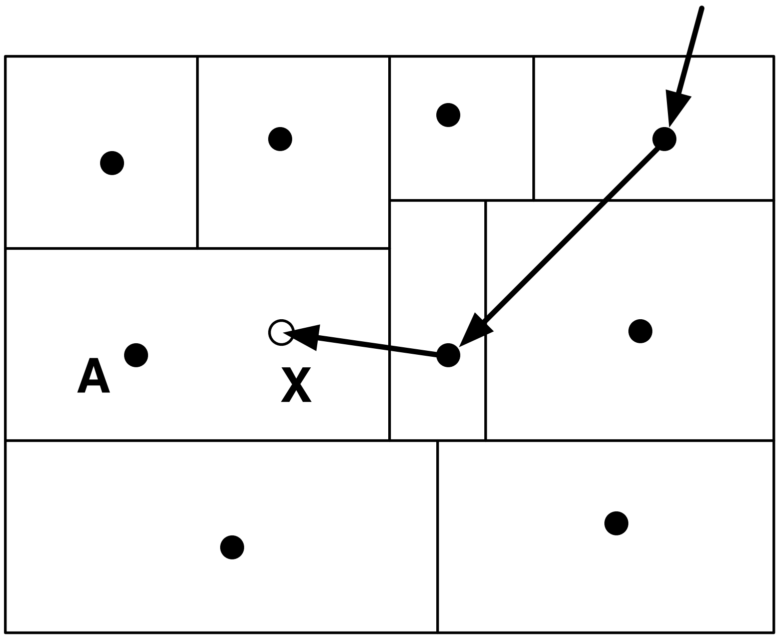
- **Takeover Face** The takeover face for a given zone is the face that will disappear when that zone's controlling node departs and its zone merges with neighbors. The takeover face is the most recent entry on the split list. A node must notify any neighbors next to the takeover face that they are responsible for taking over the node's zone. From the unit square example, node A controlling zone A_z must inform node B that it takes over for A_z. A node may be responsible for taking over all or part of more than one adjacent zones. A node cannot tell for itself whether it should take over for an adjacent node.

- **Merge** A zone controlled by a node presently in the CAN merges with a zone controlled by a departed node. The departed node may have left intentionally or by failure. A present node knows the other zones with which it should merge, since the nodes controlling those zones indicate their most recent split based on their own split lists. During the merge, the present node deletes the opposing face from its own split list. This face may not be the most recent entry in the split list. A departed zone may have more than one zone take over. Figure 6.2, for instance, indicates how two zones take over for one that left.

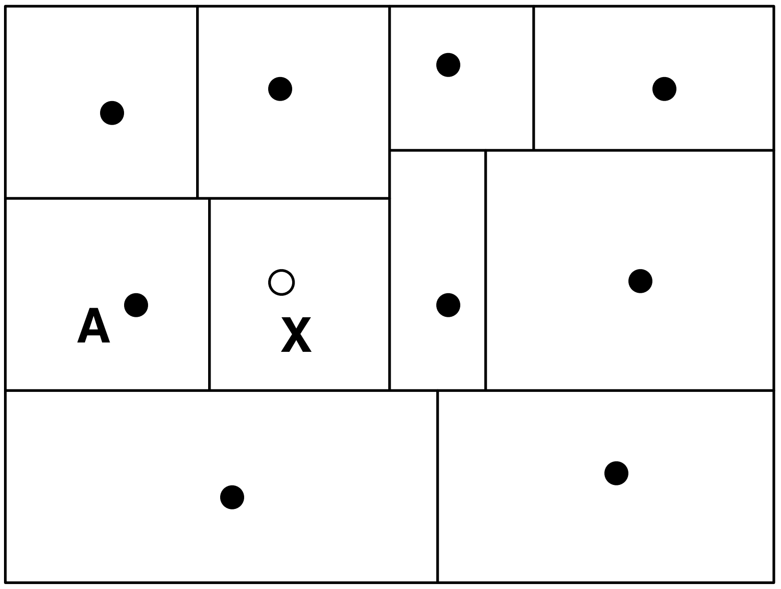
- **Neighborhood** A neighborhood is the set of all direct and indirect neighbors for a given node except itself (because a node is usually its own indirect neighbor). The neighborhood for node A is referred to as N_A.

6.1 Stale Zone Information

Nodes periodically construct and send updates to their neighbors. These updates contain information about the node itself. They also contain a node's own neighbor information so that the update recipient can construct indirect neighbor

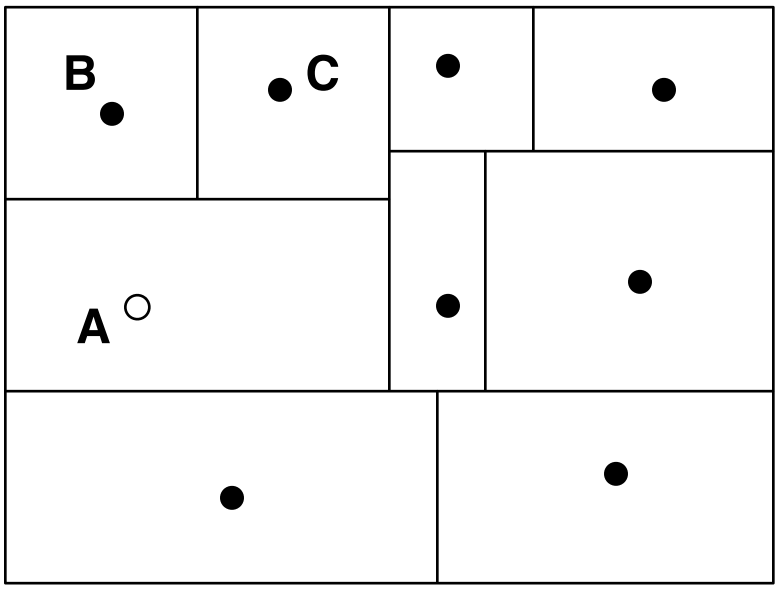


(a) New node X is routed to the zone that contains its point, owned by node A.

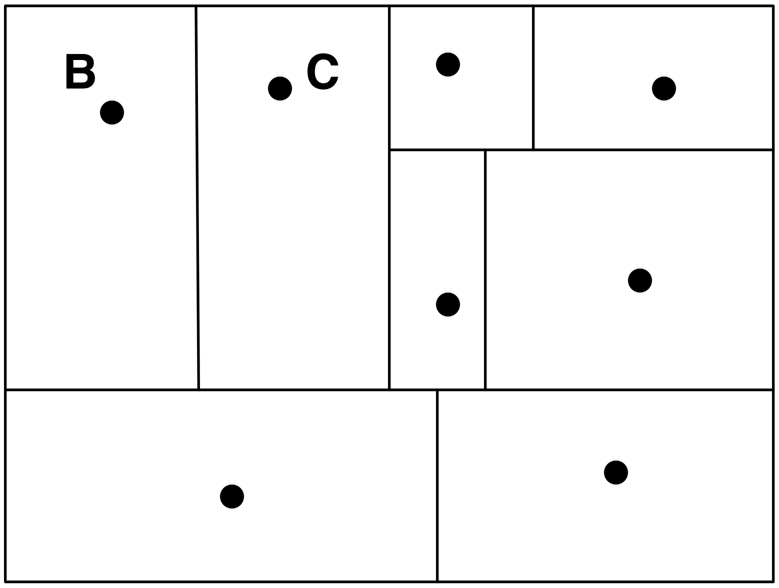


(b) The zone splits, leaving node A in one part and X in the other. All neighbors of A and X are notified of the topology change.

Figure 6.1: Node X joins the CAN by splitting the zone owned by node A.



(a) Node A departs intentionally or otherwise. All neighboring nodes, including nodes B and C, are notified by A directly or indirectly when A fails and times out that the zone formerly controlled by A is now vacant.



(b) The most recent split for A was along its upper bound in the y-dimension. Thus, the nodes on the other side of that split, being B and C, should take over. They take over their respective parts of the zone formerly belonging to A, then notify all new neighbors. Nodes B and C have the indirect neighbor information necessary to repair the topology.

Figure 6.2: Node A leaves the CAN, then nodes B and C take over by zone merge.

information. Messages can be received in any order and there are no guarantees about information freshness. This means that stale information can leak back in to the network through updates about indirect neighbors.

Many inconsistencies in the CAN topology are caused by stale information. Messages can be misrouted in to loops or lost. Or worse, nodes can incorrectly take over for others based on stale information and create unrecoverable overlaps.

The following example demonstrates how this can happen, as illustrated in Figure 6.3. At time t_0, two nodes A and B are in the CAN. Node A is the current takeover node for B. At time t_1, node B splits in to B and B', such that A and B are still adjacent. Node B sends an update to node A, but it will not arrive until t_4. B then immediately leaves voluntarily. A gets the notification for all this activity relatively quickly, including information about the role of B' as the current take-over node for B, by time t_2. Node B' takes over for B and informs its new neighbor A at t_3 Finally, the update from B finally shows up at t_4. Node A has forgotten about B, so it re-adds zombie node B as a neighbor. It has no way of deciding whether zombie B or B' is the correct neighbor along that face. A attempts to send an update to B, but then notices that it is not there. As far as A knows, it should take over for the zombie node, so it does and an overlap is created with B' at t_5. This situation is improbable, but it can be generated in high-churn situations. It is not the only mode of failure: stale information can leak back in to the CAN through indirect neighbor information. For instance, A might get the stale information about B through an update from another node adjacent to both A and B. Thus, merely ensuring that in-band messages arrive in order is insufficient since indirect neighbor information acts as an out-of-band message medium.

The solution is fairly straightforward. Each node A keeps an instance number (or serial number) that gets incremented when a node splits or merges. When a node updates a neighbor, it passes along this instance number. Each node retains a record of the instance number for each direct and indirect neighbor. Nodes also retain a record of recently departed neighbors. In the previous example, the erroneous situation is averted by noticing that the update from B originated before the deletion instance, so A simply discards the late update from B.

6.2 Concurrency

CAN uses indirect neighbor information to recover from failures. When a node takes over a vacated zone, it uses neighbor information delivered from the former owner of the zone to populate the new direct neighbor information. For this reason, CAN is able to recover from a single node failing in a given region of the CAN space. When two events happen *simultaneously*, they both occur in the same short interval of time required to identify and recover from a single failure. This is dependent on values set by policy, such as the update frequency between neighbors, as well as values outside the control of implementation, such as the message latency. Since CAN only retains one level of indirect neighbor information, CAN may not be able to recover from two adjacent nodes failing simultaneously. Similarly, CAN tolerates joins and voluntary departures as long as the joins and departures are not adjacent.

Unfortunately, when many joins and departures happen simultaneously, the

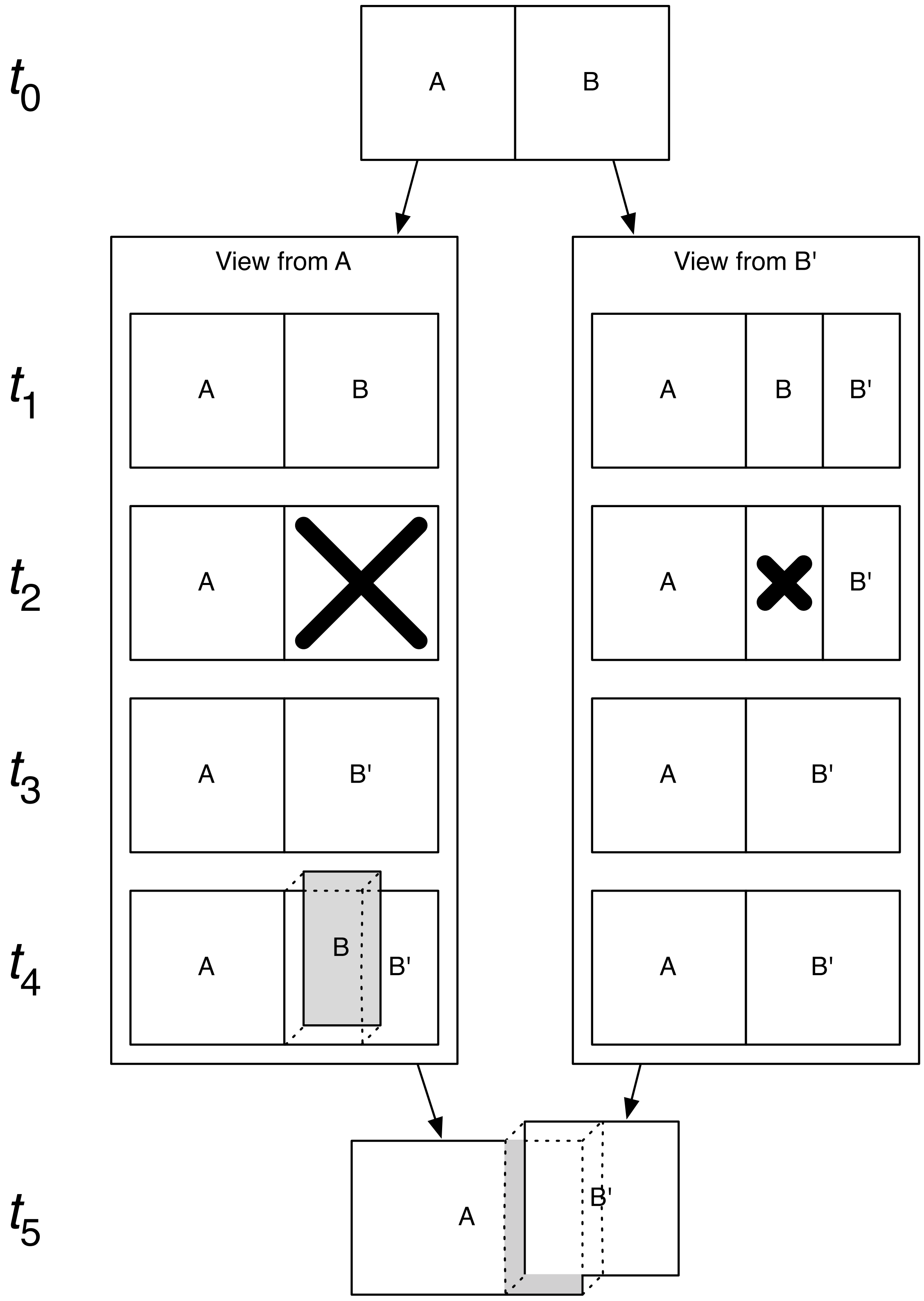
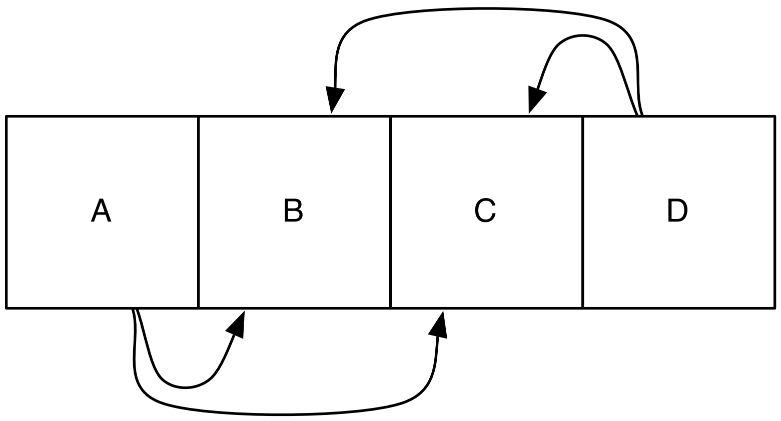


Figure 6.3: Node A overlaps node B' due to stale information arriving from a moribund B.

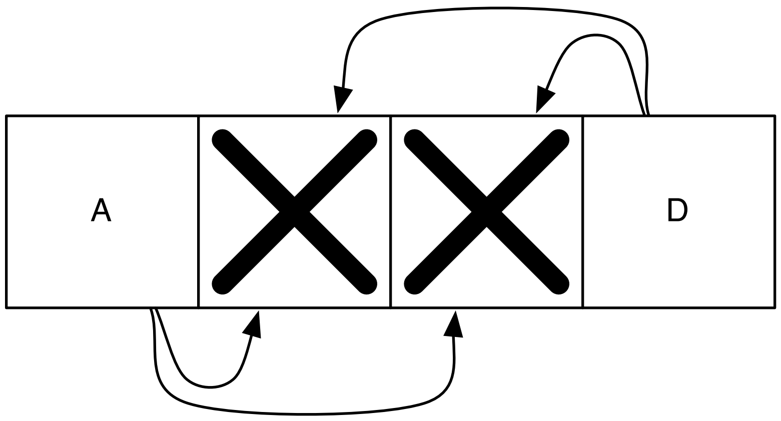
indirect neighbor information that CAN carries may still not be enough to patch up direct neighbor information. When two adjacent nodes voluntarily leave, the situation may resemble a simultaneous failure especially if the nodes should mutually take over for one another. In other words, since the zones point at one another to merge, when they both leave there is no zone left responsible for taking over the resulting gap. Similarly, when two adjacent zones split, certain local topologies may prevent the new zones from discovering one another even if they are directly adjacent. Effectively, voluntary splits and merges are almost as bad as node failure.

For example, a simultaneous departure can cause a routing discontinuity as illustrated in Figure 6.4. Suppose there are four nodes, A, B, C, and D. They control four zones arranged in a row, A_z, B_z, C_z, and D_z with A_z adjacent to B_z, B_z adjacent to C_z, and C_z adjacent to D_z. Suppose nodes B and C voluntarily leave simultaneously. Nodes A and D have enough information to take over zones B_z and C_z. However, they do not have enough connectivity information to recover, leaving a routing discontinuity on the border of the zones now owned by A and D: they cannot find one another. The same result would happen if B and C failed unexpectedly. However, since B and C are leaving voluntarily, I will show how this problem can be mitigated. A similar situation can arise when two nodes arrive at adjacent nodes at the same time, so I solve these two related problems with one solution.

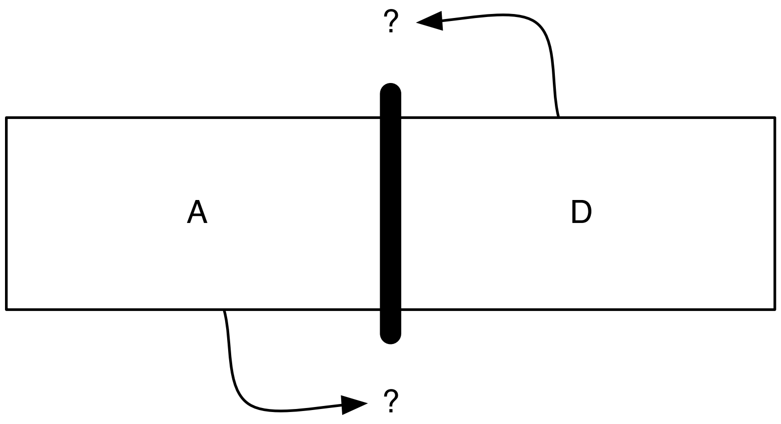
Fortunately, timing of voluntary changes can be controlled: a given change can be delayed until the local neighborhood is stable. When two nodes make changes, they may interfere with one another if they are direct neighbors or if they both share



(a) The condition of the CAN before B and C leave, including pointers to the other nodes of which A and D are aware.



(b) Nodes B and C leave simultaneously.



(c) Nodes A and D have taken over for B and C. However, they are not aware of each other so there is a routing discontinuity along the shared face of their respective zones.

Figure 6.4: Nodes B and C leave a CAN simultaneously causing a routing discontinuity.

a direct neighbor. Thus, if one or both changes are delayed by different amounts of time, the changes can be serialized locally. More distant changes are independent, thus there is no need to globally serialize changes.

My solution is based on the first phase of Egalitarian Paxos [50], which is a decentralized version of Paxos [51]. Like Paxos, Egalitarian Paxos is a consensus agreement protocol where multiple nodes to safely agree on a state change. However, Egalitarian Paxos does so without the need for any centralized master.

Overall, a node that seeks to make a change will propose that change to its neighbors, both direct and indirect. If there are no conflicts and the neighbor state is consistent with the view of the changing node, the node will commit the change. If a conflict is detected, where changes to the CAN fabric would interfere with each other, one or both proposing nodes will issue a no-op and retry after a random duration. This conflict detection protocol has the added benefit of avoiding a voluntary change adjacent to a node failure.

The state that any given node is responsible for is its own coordinate, zone, and takeover face. A node's direct neighbors hold replicas of this information for failure recovery. When a zone splits or merges, that state changes and the node must communicate this state change information to its neighborhood. Effectively, a node's neighborhood is the *replica set* for its state. There are two differences between Egalitarian Paxos and the algorithm described here. First, the command leader and its replica set must agree to the change, and not just a majority. Second, if there is a conflict detected, then the command must be changed to a no-op and committed. This is because the state changes described here also change replica sets. Thus, nodes receive conflicting commands in different orders so there is no way to resolve these dependencies.

The purpose of the conflict detection protocol is to maintain routing connectivity in the absence of simultaneous adjacent failure. However, this precludes liveness for join and leave requests. There is no bound on the rate of incoming join and leave requests. Since join and leave requests can happen at any node and at any time, the conflict detection protocol must be observed every time a command is retried. Thus, there could be pathological situations where commands conflict every time they are proposed. When a conflict is detected, commands are delayed by a random duration. However, if new commands arrive too frequently, the conflict detection protocol will reach saturation and progress will stop. The protocol can be augmented so that older commands are executed first, but there's nothing that can prevent a stream of new commands from interfering with old commands.

There are three possible commands:

- SPLIT When a new node joins, a participating node must split its zone so that the new node can obtain a zone. The new node is A. The participating node that controls the zone where the coordinate for A lies is the proposing node L. The change is proposed to the neighborhood N_L of node L. If the split cannot happen immediately, A must retry the entire join process at a later time. It cannot restart the split at L, since the CAN may have changed while A waited to rejoin.

- MERGE When a node departs, participating nodes will merge their zones with the departed node's zone. Call departing node L, which is also the proposing node. The change is proposed to the neighborhood N_L of node L. If the merge cannot happen immediately, L must retry at a later time.

- NO-OP A node may start a SPLIT or a MERGE. However, if other commands interfere or a failure prevents command completion, a no-op will be committed instead. Nodes do not propose the NO-OP command.

Every node has some bookkeeping to support the serialization protocol: As discussed previously, this includes the node's coordinate, zone, and takeover face. It also includes the instance number, which is incremented when the node may split or merge voluntarily as well as when it takes over for another failed node. The instance number for node A is referred to here as i_A. Every node also keeps track of information about its neighbors, including a neighbor's coordinate, zone, takeover face, and instance number. In addition to this information, command state information is added. On node A, the information relevant to the serialization algorithm that it stores about node B is kept in $\mathrm{node}_A[B]$. So, $\mathrm{node}_A[B].\mathrm{state}$ stores the command state and $\mathrm{node}_A[B].\mathrm{instance}$ stores the latest instance number of B, as far as A is aware. By default, the command state is STABLE, indicating that the node is now quiescent as far as the CAN is concerned. In addition, node A also stores some additional information about itself. The command itself is stored in $\mathrm{node}_A[A].\gamma$ and holds a value of SPLIT or MERGE. Node A also stores a reply from every node to which a command proposal is sent as $\mathrm{node}_A[A].\mathrm{reply}$. For instance, A stores the reply from node B in $\mathrm{node}_A[A].\mathrm{reply}[B]$. This is initialized to UNDEF, but takes on the value YES or NO depending on whether the replying node has detected a conflict or the replying node has timed out.

The conflict detection detects when conditions are perfect for voluntary changes. If any kind of conflict or failure is detected, it does its best to do nothing: all nodes that may have been notified of a change will eventually treat the command as a no-op. Any failure recovery of CAN connectivity is handled through the regular CAN mechanisms, and nodes continue to send periodic updates to their neighbors. This differs from the typical function of Egalitarian Paxos, where it guarantees progress even if some replicas fail. When the command-initiating node fails, no attempt to do the extended prepare is done. Instead, the normal CAN failure recovery starts, and any intermediate state left by the conflict detection protocol is cleaned up then. Suppose A starts the conflict detection protocol, then immediately fails. When neighbors notice that A has failed, they will remove any references to it, including any outstanding command from that node, thus allowing subsequent voluntary commands from other nodes to proceed. Furthermore, when a neighbor's instance number is updated at a node, that node will also remove any outstanding command information.

A description of when and why the various parts of the conflict detection protocol are called is as follows:

- **Request** When a node L intends to split due to a new arriving node or merge with a neighbor because that node intends to depart, it invokes Request to start the conflict detection protocol. Request detects any known disruption that prevents the command from completion. These disruptions include existing conflicting commands, discontinuities, and other irregularities in the neighborhood. If so, Request rejects the invocation. Otherwise, Request sends out Propose messages to neighbors, indicating the intent of proposing node L. Request is detailed in Algorithm 4.

- **Propose** The Propose message is received on some neighbor R by a proposing node L. Propose stores relevant information about the proposing node. It also returns a ProposeOK message with the disposition of node R: whether or not the proposed command has conflicts. Propose is detailed in Algorithm 5.

- **ProposeOK** ProposeOK is received on the proposing node L from some neighbor R. Responses to the proposal are stored by L. If enough Responses have been received, L starts the commit. If any single node has responded NO, then the proposed command is changed to a no-op. ProposeOK is detailed in Algorithm 6.

- **ProposeTimeout** ProposeTimeout is triggered on the proposing node L when some neighbor R has not responded to a Propose message after a period of time. This is treated as a NO response from the node R so that the outstanding proposal can be committed. ProposeTimeout is detailed in Algorithm 7.

- **CommitPhase** CommitPhase is called on proposing node L when all neighbors have responded to the proposal or timed out. CommitPhase executes the command locally and sends a Commit message to all participating nodes. CommitPhase is detailed in Algorithm 8.

- **Commit** Commit is received on some neighbor R from proposing node L. Commit executes the command and updates R's knowledge of the state of L to STABLE so new proposals can proceed. Commit is detailed in Algorithm 9.

- **Cancellation or rejection** When a command fails, it needs to be retried after a random period of time. For instance, rejection can signal to a new arriving node that it needs to retry the join after a period of time. A command may fail because:

  - A conflicting command is detected.

  - A participating node times out in its respond to the proposing node.

  - A participating node fails.

  When a request is canceled or rejected, the initiating node waits a random duration before retrying the command.

The overall flow of the conflict detection protocol is described in Figure 6.5. Node A tries to join at node L. Node L sends proposals to all nodes in N_L. When all involved nodes return a positive result, the command can be committed and A joins the CAN. In contrast, node L' proposes a command to some node in N_L, where that set and $\mathrm{N}_{L'}$ overlap. The proposal message arrives after the proposal from L but before the command from L is committed, so that neighbor returns a

Algorithm 4 Request(γ, t) on L

```
1: if there is any node $Q$ such that $\text{node}_L[Q].\text{state} \neq \text{STABLE}$ or there is an inconsistency in the known neighborhood then
2:   reject the request
3: else
4:   $\text{node}_L[L].\gamma := \gamma$
5:   $\text{node}_L[L].\text{state} := \text{BUSY}$
6:   $\text{node}_L[L].\text{instance} := \text{node}_L[L].\text{instance} + 1$
7:   $\text{node}_L[L].\text{reply}[L] := \text{YES}$
8:   for all $R \in \text{N}_L$ do
9:     $\text{node}_L[L].\mathit{reply}[R] := \text{UNDEF}$
10:    send Propose($i_L$) to $R$
11:  end for
12: end if
```

Algorithm 5 Propose(i_L) on R from L

```
1: if $\text{node}_R[Q].\text{state} \neq \text{STABLE}$ for any $Q$ or takeover($R$) is failed then
2:   $r_R := \text{NO}$
3: else
4:   $r_R := \text{YES}$
5: end if
6: $\text{node}_R[L].\text{state} := \text{BUSY}$
7: $\text{node}_R[L].\text{instance} := i_L$
8: send ProposeOK($r_R, i_L$) to $L$
```

Algorithm 6 ProposeOK(r_R, i) on L from R

```
1: if $i = \text{node}_L[L].\text{instance}$ then
2:     $\text{node}_L[L].\text{reply}[R] := r_R$
3:     if YES $= \text{node}_L[L].\text{reply}[Q]$ for all $Q \in \text{N}_L$ then
4:         CommitPhase($\text{node}_L[L].\gamma, \text{node}_L[L].\text{instance}$)
5:     else if $\text{node}_L[L].reply[Q] \neq$ UNDEF for all $Q \in \text{N}_L$ then
6:         cancel($\text{node}_L[L].\gamma$)
7:         CommitPhase(NO-OP, $\text{node}_L[L].\text{instance}$)
8:     end if
9: end if
```

Algorithm 7 ProposeTimeout(R) on L when R times out on a ProposeOK

```
1: $\text{node}_L[L].\text{reply}[R] :=$ NO
2: if $\text{node}_L[L].reply[Q] \neq$ UNDEF for all $Q \in \text{N}_L$ then
3:     cancel($\text{node}_L[L].\gamma$)
4:     CommitPhase(NO-OP)
5: end if
```

Algorithm 8 CommitPhase(γ) on L when all involved nodes have responded or timed out

```
1: $\text{node}_L[L].\text{state} :=$ STABLE
2: execute($\gamma$)
3: for all $R \in \text{N}_L$ do
4:     send Commit($\gamma, i_L$) to $R$
5: end for
```

Algorithm 9 Commit(γ, i) on R from L

```
1: if $i = \text{node}_R[L].\text{instance}$ then
2:     $\text{node}_R[L].\text{state} := \text{STABLE}$
3:     execute($\gamma$)
4: end if
```

NO result. Consequently, L' commits a NO-OP instead of its originally requested command.

In summary, basic CAN handles joins, voluntary departures, and failures when they do not interfere with one another. When those events are simultaneous and adjacent, routing holes, overlaps and other erroneous configurations appear in the CAN. The conflict detection protocol presented here, however, prevents simultaneous joins and voluntary departures.

6.3 Outstanding Problems

Despite the improvements, there are still two outstanding problems with CAN. First, CAN can experience unbounded data structure growth. For instance, suppose node A has coordinate $(1, 1)$ and is responsible for a zone with boundaries $\{[0, 2), [0, \top)\}$ Node B has coordinate $(3, 1)$ and is responsible for a zone with boundaries $\{[2, \top), [0, \top)\}$. Suppose an arbitrary number of nodes arrive with coordinates of the form $(3, x)$, where x is any valid y-coordinate. In this case, node A can end up with all of those nodes as neighbors. This is a problem with CAN in all forms.

Second, CAN might develop routing discontinuities due to simultaneous ad-

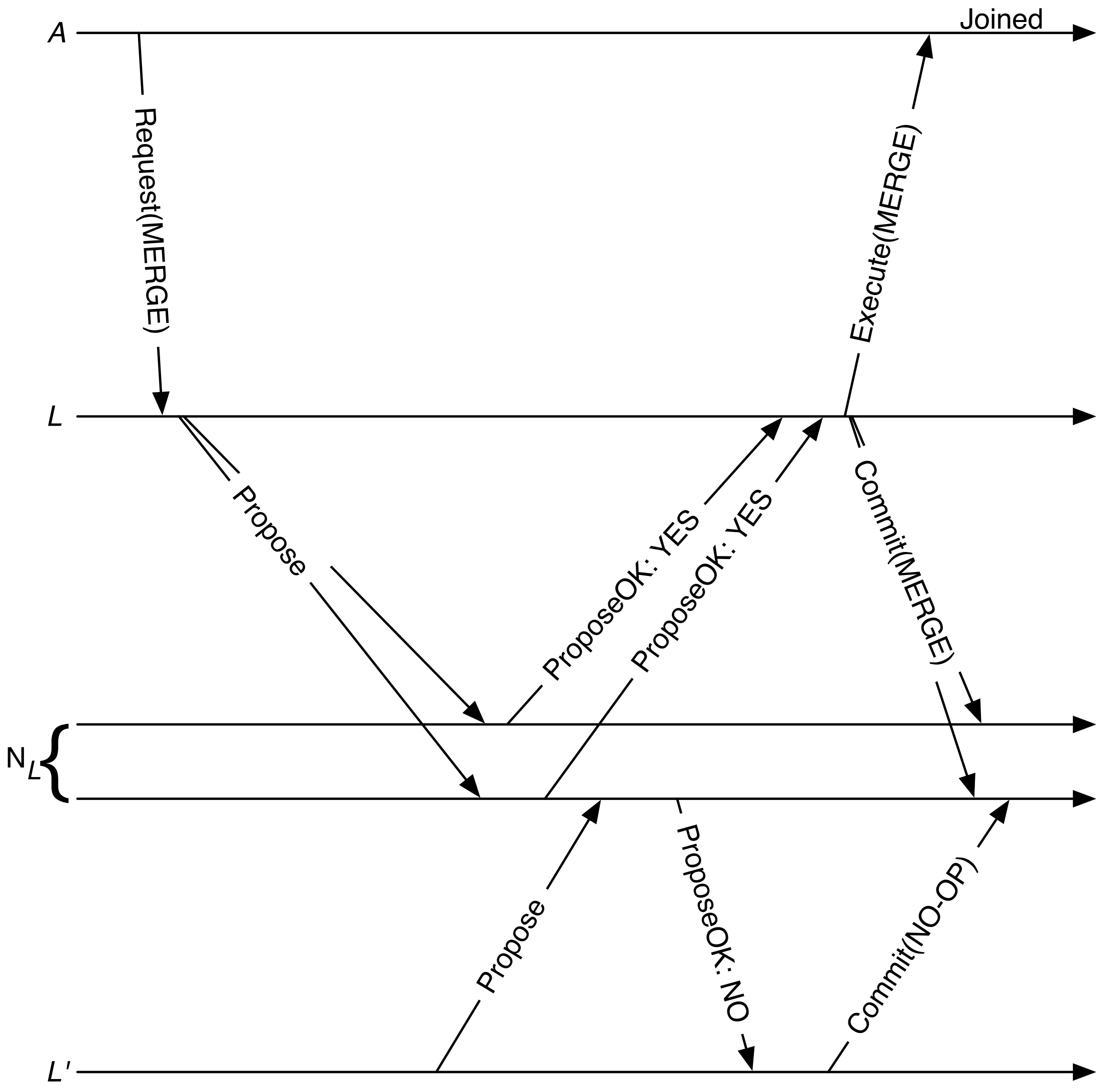


Figure 6.5: Operation of the conflict detection protocol for a successful MERGE and when a conflict is detected. Conflict detection works much the same way for SPLIT.

jacent node failures. There is no way to repair these routing discontinuities. One possible solution is to require nodes that experience an irreparable routing flaw along a boundary to depart voluntarily. Nodes will continually depart until the discontinuity is swallowed up and disappears inside a zone. However, this is impractical because it may incur the entire network leaving if the discontinuity is along the original zone split boundary. Both of these issues are addressed in Section 7.1.

Chapter 7: Conclusion

Several important contributions are explored here. First, a completely decentralized scheduler for tightly-coupled parallel jobs is demonstrated. This scheduler performs comparably to a globally load balanced scheduler. Among other things, over-provisioning and load balancing techniques are crucial to good performance. The experiments are based on actual job traces rather than synthetic inputs. Criteria for successful and efficient application of this scheduler have also been established: Clusters must be larger than the average job but not too large, and certain pathological overlaps in clusters must be limited.

Second, a way to form clusters dynamically with no foreknowledge of network structure is demonstrated. Several techniques for predicting latency in networks are tested. I show empirically that DTM produces the best predictions and also functions in a decentralized manner. However, even this is not good enough to use directly. Errors in the output can cause jobs to fail. Also, the clustering algorithm does not provide the guarantees needed to ensure cluster quality for the scheduling algorithm. For these reasons, I present a decentralized refinement algorithm for forming high-quality clusters. The refinement algorithm includes a symmetry step, which allows individual nodes to advertise larger clusters without adversely affecting the cluster quality bounds. The performance of dynamic clustering is comparable to that of static clustering when symmetry is enabled.

Third, improvements to the robustness of the underlying structured peer-to-peer network are detailed. CAN has problems with stale information leaking back in and corrupting the overlay connectivity information. CAN also has inadequate mechanisms for preventing adjacent arrivals and departures from becoming routing holes. Both of these problems were observed in practice, and both of them are solved here.

My thesis is that **scalable parallel computing in a decentralized, distributed environment is both feasible and can be done with performance comparable to centralized systems**. I have developed algorithms for decentralized parallel scheduling algorithms and dynamically forming clusters of nodes. These algorithms are evaluated experimentally and shown to satisfy the thesis.

7.1 Future Work

7.1.1 Scheduler Study and Improvements

Some things are still not well understood in the scheduler. For instance, the relationship between over-provisioning, cluster size, and job load is not well understood. More experiments exploring this relationship are necessary.

The scheduler has room for improvement. An obvious change is to increase the reach of the scheduler, making more options visible when load balancing. Presently, the scheduling algorithms look at direct neighbors and indirect neighbors. This is done to contain the per-node costs both in terms of memory use and messages. With these costs in mind, the number of other nodes the scheduler looks at while load balancing might be increased in a controlled way. Hopefully this will perform closer to a scheduler with global knowledge.

Other improvements could be made by changing the way the jobs are run in the first place. For instance, backfill [24, 25] is a common technique for improving throughput in batch scheduled systems. Analysis of the experimental logs reveals that as much as 5% of the time that nodes could be using to do work is wasted while waiting for parallel jobs to start. However, backfill is not appropriate for this scheduler because estimated run time information is not available. *Speculative backfill* [52], on the other hand, may be appropriate.

Speculative backfill is backfill where the estimated run time is unknown or inaccurate. Like regular backfill, jobs are started out-of-order. However, if the backfilled job would cause an earlier job to be delayed, the backfilled job is preempted. Speculative backfill is not appropriate for all jobs. For instance, jobs that consume limited resources or have other unrecoverable side effects cannot be backfilled. As long as those jobs are marked as such, speculative backfill should be useful.

Speculative backfill may be particularly useful due to the way that parallel jobs are started. When a parallel job is scheduled, dependent tasks are enqueued on many more nodes than are necessary to satisfy the job. When enough dependent tasks become runnable, the other idle, enqueued tasks are canceled and the parallel job runs. An individual task may become runnable immediately, then wait for a long time before the rest of the necessary tasks become available. While this task waits, the node cannot process any other tasks. Speculative backfill fixes this by running other tasks out of the queue until the job owning the task at the head of the queue is ready to run. At that point, any other running task is preempted so that the original task can participate in the parallel computation.

The criterion for clusters has been latency. However, bandwidth is also an important consideration. In particular, *bisection bandwidth* is important, or the aggregate bandwidth from half of the nodes in the cluster to the other half of the nodes. Some effort should be put in to probing bisection bandwidth for the purpose of advertisement in to the CAN. Measuring simple point-to-point bandwidth may be useful for determining whether individual nodes have a high bandwidth connection to other individual nodes. However, point-to-point tests alone are inadequate, since the underlying network infrastructure may have bottlenecks that only appear when bandwidth is tested in aggregate.

7.1.2 Yggdrasil: Finding Similar Nodes Nearby

There is no way to find nodes that have similar resources in a single latency prediction tree. This is a problem. For example, suppose there is a cluster of 1000 low-spec nodes and 10 high-spec nodes. A high-spec node has more RAM, more cores, and is generally more capable than a low-spec node. A high-spec node can run both high-spec and low-spec jobs. However, all nodes participate in the same low-latency network. If the high-spec and low-spec nodes are participating in the same latency prediction overlay, then a given high-spec node may only detect low-spec nodes as near neighbors. Recall that this is due to the maximum job size policy that affects the clustering algorithm. Thus high-spec nodes might not find one-another to form a cluster. Indeed, depending on the composition of the network, finding like nodes may be improbable.

To solve this problem, nodes join type-specific latency overlays as per Chapter 4. This preserves the different node types so that high-spec nodes can find one-another and form clusters for high-spec jobs. However, this solution fails to serve networks of limited heterogeneity where scheduling large parallel jobs across all node types is desirable. It also fails to serve highly heterogeneous networks, such as a departmental desktop network. In that case, heterogeneous clusters must be formed to run parallel jobs at all.

Fortunately, I have conceived a solution for this problem. All nodes join a single latency prediction overlay called *Yggdrasil*. In the default formulation of the distributed prediction tree, the k-nearest neighbors with a given diameter limit are aggregated to each node. In Yggdrasil, instead the k-nearest *unique* neighbors are aggregated. Yggdrasil discovers the types of nodes that are in the local latency neighborhood.

Using this information, an individual node can then add itself to smaller type-specific latency prediction overlays. For instance, a high-spec node from the previous example normally adds itself to the high-spec type overlay. However, it discovers the low-spec node type through Yggdrasil and can add itself to the low-spec overlay, as well. This does not guarantee finding all possible close node types if there are more types than can be aggregated through Yggdrasil. Node type information is unbounded in size so such a guarantee is not a desirable trait.

Two adjustments must be made to the algorithms from Chapters 5 and 4. First, the application of symmetry in clustering is not as obvious. A node cannot merely advertise the largest cluster to which it belongs. For instance, a high-spec node from the previous example may find a larger cluster in the low-spec overlay. Instead, a node must advertise the largest cluster to which it belongs such that the minimal capability in that cluster matches that node. A high-spec node will advertise the largest high-spec cluster to which it belongs. A low-spec node will advertise the largest low-spec cluster to which it belongs which may include high-spec nodes. There may also be some need to give preference to one cluster over another when they contain different proportions of heterogeneous nodes.

Second, over-provisioning during the parallel job start needs to be revisited. By default, starting a low-spec job on a mixed cluster would mean low-spec tasks sent to every node including high-spec nodes. This is necessary when the job is so large that it must be run on at least one high-spec node. However, this is undesirable when high-spec jobs are present, since only high-spec nodes can run those jobs. This means a high-spec job could be blocked from running by a low-spec job even when there are low-spec nodes available. Instead, a job should probably only be over-provisioned to groups of least capable nodes until the number of required tasks is met. This is not as straightforward as sending tasks to low-spec nodes then sending tasks to high-spec nodes if not enough low-spec nodes are available because the resource space is a partial ordering and not a total ordering. For instance, suppose there are two other node types as well, called mid-spec$_1$ and mid-spec$_2$. That low-spec <

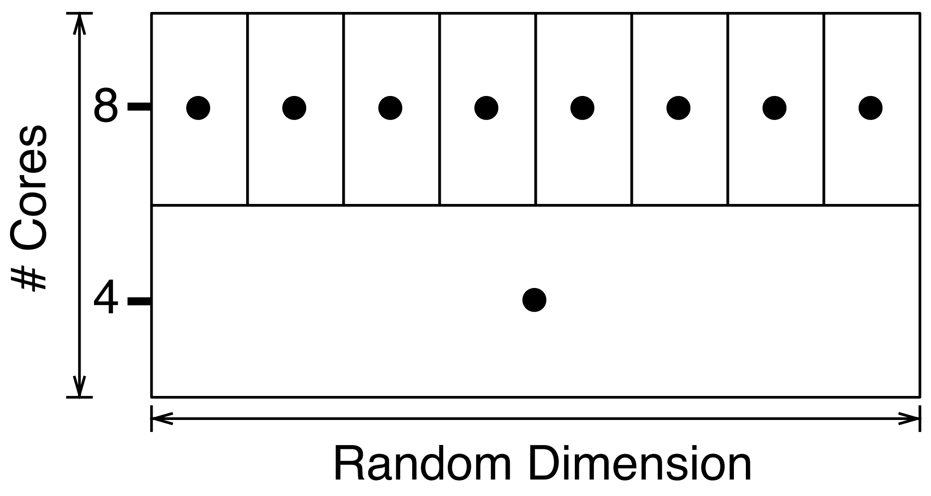
high-spec has already been established. Suppose the relationships between low-spec, high-spec, and the mid-spec types are defined such that mid-spec$_i$ $<$ high-spec and low-spec $<$ mid-spec$_i$ for $i = 1, 2$. The mid-spec types are defined such that mid-spec$_1$ $\not<$ mid-spec$_2$ and mid-spec$_2$ $\not<$ mid-spec$_1$. In this case, on which nodes should the dependent tasks go if they will not fit on the low-spec nodes in a given cluster? This is a question that this future work should answer.

7.1.3 Decoupling CAN Topology from Grid Membership

The Achilles' Heel of CAN as described in Kim et al. [6] and passed on to my work is that all nodes that participate in the grid also participate in the CAN topology. This introduces two problems in to the CAN. First, nearby nodes with identical capabilities in a cluster network are likely to appear next to one another in the CAN itself. CAN makes no guarantees about recoverability when two adjacent nodes fail simultaneously. However, if the site hosting those two nodes were to go offline, then that is exactly what would happen. Second, the number of neighbors that a given node may have is unbounded. Where coordinates are hash values such as in the original CAN formulation, this is unlikely to occur. However, where the coordinates correspond to resource capability, many nodes can stack up next to one another in the virtual dimension and adjacent to another single node.

Figure 7.1a shows an example of this pathological behavior. There is a single node with four cores. There are an arbitrary number of nodes with eight cores. If the four core node joins first followed by the eight core nodes, the four core node

(a) A pathological CAN topology.



(b) A better CAN topology.

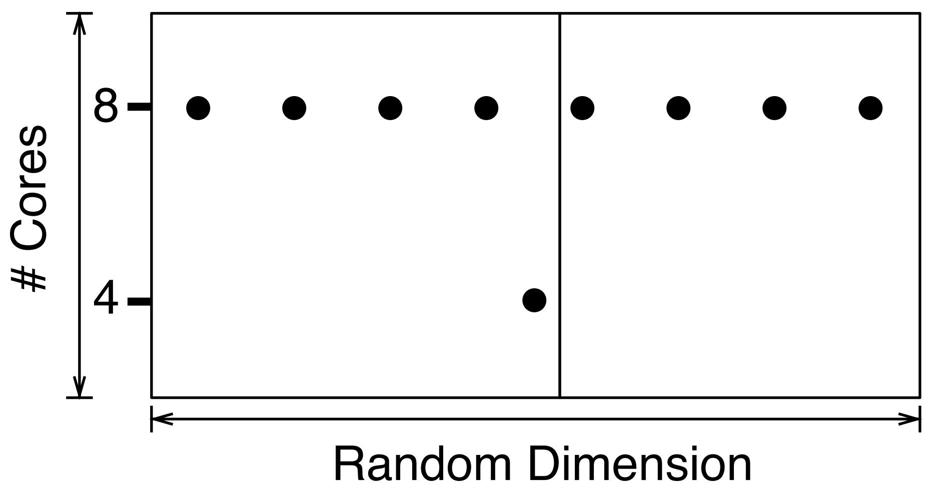


Figure 7.1: There is no bound on the number of neighbors a given zone may have. In the default formulation of CAN, this is not a problem because coordinates are hashes. However, when the CAN is treated as an actual spatial index where zones are tied to node resource coordinates, it becomes probable that scaling problems occur. For instance, in Figure 7.1a, a single four-core node owns a zone neighboring an arbitrary number of zones owned by eight-core nodes. This is fixed in Figure 7.1b by treating nodes as points stored in zones.

can wind up with an arbitrary number of eight core neighbors.

In the current formulation of CAN, a node's zone is tied to its own resource coordinate. My solution instead is to decouple the CAN topology from grid membership. CAN is treated like a distributed spatial index where the zones exist to satisfy the demands of the index rather than fixed to node resources. Individual nodes join the CAN as points associated with the enclosing zone rather than splitting a zone for every single join. Zones are split when the number of nodes gets too high, much like a conventional bucketed *k*-*d* tree. By this method, the grid becomes an induced hierarchy where all nodes are dependent on a smaller number of zone-controlling nodes.

For example, the pathological behavior from the previous example might instead appear like the grid in Figure 7.1b. The resource space is divided in to two zones. The participating grid nodes are distributed evenly between the two zones.

Chapter A: Symmetry

From Chapter 5, Symmetry is the idea that a node can advertise the largest cluster to which it belongs, rather than the cluster that it finds for itself. Dynamically discovered clusters need symmetry for good performance. However, it must be shown that symmetry does not increase the storage or communication burden on nodes, undoing the work from limiting cluster membership. What follows is a formal statement and proof of this property.

Definitions:

- $G = (V, E)$: the complete graph such that every node in the grid corresponds to a vertex in V, and the edges E are the actual latency distances between each node.
- $C : V \rightarrow \mathcal{P}(V)$: C is an injective function on V to subsets of V. C is a *clustering* of G.
- *Cluster size bound* N: Maximum number of vertices in a cluster. If $x \in V$, then $|C(x)| \leq N$.
- *Cluster membership bound* M: Maximum number of clusters to which a vertex can belong. For any $x \in V$, $|\{y|x \in C(y)\}| \leq M$. Not all clusterings have

the cluster membership bound, but clusterings yielded by the Limiting Cluster Membership from Chapter 5 do.

- *Awareness* $A(C, x)$: The set of all nodes in all clusters to which x belongs: $A(C, x) = \bigcup_{z|x \in C(z)} C(z)$ This is called awareness because the node corresponding to vertex x must make latency measurements against all other nodes in $A(C, x)$.

- *Neighbor awareness bound* $A(C, x) \leq M \cdot N$: Maximum number of other vertices for which a given vertex is aware. Clusterings with the cluster size bound N and the cluster membership bound M have this property because any vertex x is in no more than M clusters of up to N size.

- *Symmetry operation* $C' = \text{sym}(C, x, y)$ The symmetry operation transforms C in to C' such that $C' = C$

  $C'(x) = C(y)$. This means that for all $i \in V, i \neq x$, $C(i) = C'(i)$.

Lemma A.1. *Suppose clustering C has the cluster size bound N. Then any symmetry operation* $\text{sym}(C, x, y)$ *also has the cluster size bound N.*

Proof. Since the cluster size bound is guaranteed in C and $C'(x) = C(y)$ by definition, $|C'(x)| = |C(y)| \leq N$. Thus, the cluster membership bound is preserved for $C'(x)$. No other mappings are different between C and C'. Therefore the cluster size bound is preserved in C'. $\square$

Lemma A.2. *Suppose clustering C has the neighbor awareness bound $M \cdot N$. Then any symmetry operation* $\text{sym}(C, x, y)$ *also has the neighbor awareness bound $M \cdot N$.*

Proof. When $|V| \leq 2$, this is trivially true. In the case that $|V| > 2$:

Let be any $a \in V$ such that $a \neq x$ and $a \neq y$. It suffices to show that $|A(C', a)| \leq |A(C, a)|$ because that demonstrates that the awareness for any node did not grow, and thus the overall bound holds. There are four cases to consider:

- $a \in C(x) \wedge a \in C(y)$:

$$\begin{aligned}
A(C',a) &= \bigcup_{z \mid a \in C'(z)} C'(z) \\
&= \left(\bigcup_{\substack{z \mid a \in C'(z) \\ \wedge z \neq x \\ \wedge z \neq y}} C'(z) \right) \cup C'(x) \cup C'(y) \\
&= \left(\bigcup_{\substack{z \mid a \in C'(z) \\ \wedge z \neq x \\ \wedge z \neq y}} C'(z) \right) \cup C(y) \cup C(y) \\
&= \left(\bigcup_{\substack{z \mid a \in C'(z) \\ \wedge z \neq x \\ \wedge z \neq y}} C'(z) \right) \cup C(y) \\
&= \left(\bigcup_{\substack{z \mid a \in C(z) \\ \wedge z \neq x \\ \wedge z \neq y}} C(z) \right) \cup C(y) && C(z) = C'(z) \text{ for all } z \neq x \\
&= \bigcup_{\substack{z \mid a \in C(z) \\ \wedge z \neq x}} C(z) && a \in C(y) \\
&\subseteq \left(\bigcup_{\substack{z \mid a \in C(z) \\ \wedge z \neq x}} C(z) \right) \cup C(x) \\
&= \bigcup_{z \mid a \in C(z)} C(z) \\
&= A(C,a) && \text{definition of } A(C,a)
\end{aligned}$$

Thus, $A(C',a) \subseteq A(C,a)$.

- $a \in C(x) \wedge a \notin C(y)$

$$\begin{aligned}
A(C', a) &= \bigcup_{z | a \in C'(z)} C'(z) && \\
&= \bigcup_{\substack{z | a \in C'(z) \\ \wedge z \neq x}} C'(z) && a \notin C(y) \text{ and } C(y) = C'(x) \\
&= \bigcup_{\substack{z | a \in C(z) \\ \wedge z \neq x}} C(z) && C(z) = C'(z) \text{ for all } z \neq x \\
&\subseteq \left(\bigcup_{\substack{z | a \in C(z) \\ \wedge z \neq x}} C(z) \right) \cup C(x) && \\
&= \bigcup_{z | a \in C(z)} C(z) && \\
&= A(C, a) && \text{definition of } A(C, a)
\end{aligned}$$

- $a \notin C(x) \wedge a \in C(y)$

$$\begin{aligned}
A(C', a) &= \bigcup_{z | a \in C'(z)} C'(z) && \\
&= \bigcup_{z | a \in C(z)} C(z) && C(z) = C'(z) \text{ for all } z \neq x \\
&= A(C, a) && \text{definition of } A(C, a)
\end{aligned}$$

- $a \notin C(x) \wedge a \notin C(y)$ Vertex a is unaffected in C', therefore $A(C, a) = A(C', a)$.

Thus, $A(C', a) \subseteq A(C, a)$. Therefore $|A(C', a)| \leq |A(C, a)|$. □

Theorem A.3. *Suppose there is an ordered sequence of ordered pairs* $\{(x_0, y_0), (x_1, y_1), ...(x_k, y_k)\}$ *such that every pair is in* $V \times V$*. Furthermore, suppose that these pairs are applied to an initial clustering* C *such that:* $C_0 = \text{sym}(C, x_0, y_0), C_1 = \text{sym}(C_0, x_1, y_1), ..., C_k =$

$\text{sym}(C_{k-1}, x_k, y_k)$. *If C has bounded membership and neighbor awareness, then C_k will also have bounded membership and awareness.*

Proof. Proof is by induction. Assume that C has membership bound N and neighborhood awareness bound $M \cdot (N-1)$ by the premise of the proof. Lemmas A.1 and A.2 state that applying a sym operation on a clustering with the membership and neighbor awareness bound will yield another clustering that holds those same bounds. Therefore, since the initial clustering is bounded and an application of symmetry to a bounded clustering results in a bounded clustering, the result of recursive application of any number of sym operations to the original clustering C will also be bounded. $\square$

The first approximation clustering has the membership bound M. The cluster membership limitation only removes nodes from clusterings, so it yields a clustering with membership bound M and neighborhood awareness bound $M \cdot (N-1)$. Filtering only removes nodes from clusterings, so the input to the symmetry step also has membership bound M and neighborhood awareness bound $M \cdot (N-1)$. Thus, symmetry will also produce a clustering with those bounds.

The application of symmetry in the online, decentralized algorithm is not ordered. However, it is equivalent to an ordered application of symmetry like in the premise of Theorem A.3 because there are no circular dependencies. If node x uses symmetry to adopt $C(y)$ as its cluster, y has access to cluster larger than $C(x)$, which may be $C(y)$ or some other even larger cluster $C(z)$. Furthermore, nodes continue to maintain the clusters that they discovered themselves independently of the cluster that they are advertising. For instance, $C(y)$ is still available to x even though y may be advertising some other $C(z)$. Thus, at any given time, the global clustering state is equivalent to applying the sym operation in the order of a topological sort of the clustering dependencies.

Chapter B: Collecting the UMIACS Latency Data Set

This appendix discusses the challenges faced when collecting the UMIACS Latency Data Set. Most of these challenges were overcome. Some of them, however, were not.

The data set consists of two all-to-all maps of the UMIACS Condor pool. First, a ping measurement was made across all nodes. Ping is a well-understood protocol, and one with which making measurements is very easy: Coordination is largely one-sided. Only one end needs to start the measurement by sending ICMP echo request packets. The operating system on the other end will automatically respond with ICMP response packets. However, ping does not reflect the performance that a parallel application might see. Second, an MPI measurement was made across all nodes. This was done with the `osu_latency` point-to-point tool from the OSU microbenchmark suite [1]. The underlying MPI implementation was the popular OpenMPI [2], since OpenMPI supports network heterogeneity out of the box and node heterogeneity with nothing more than an extra configuration flag.

On its face, collecting an all-to-all data set should not be too difficult for a small network. For instance, the UMIACS Condor pool consists of 163 nodes.

[1] `http://mvapich.cse.ohio-state.edu/benchmarks/`

[2] `http://www.openmpi.org`

This means only 13203 individual measurements need to be made to get at least one measurement between every pair of nodes. These measurements needed to be made very carefully, however. First, the nodes on each end of the measurement should be quiescent. Load on a node can interfere with latency measurements. Second, only one pair can be measured at a time, since measurements can interfere with one another by using the same switches and network links. Moreover, parallel measurements could harm performance on a network being actively used by others. After that, the measurement itself must work. This is trivial for ping, which should always work. However, making MPI work proved to be a challenge, especially when getting MPI to not merely work but work well.

The first challenge was to find a way to exclusively co-schedule a pair of nodes for measurement. Condor has no facility for co-scheduling arbitrary pairs of nodes. Condor can schedule parallel jobs, but that is restricted to fixed clusters and this Condor pool was not configured for parallel scheduling. Instead, a separate custom central scheduling and recording service was created to match exclusively scheduled nodes to one another and record their measurements. Seizing control of a node within the policy confines of Condor was not trivial. Condor enables multiple jobs to share a multi-core node by allocating multiple slots per node. In this pool, one slot is statically allocated for every core, and it is assumed that jobs will not use more than one core. Dynamic slots are also possible, where a slot is allocated when a job is scheduled according to the number of cores needed by a given job. This would have made the task trivial, but dynamic slots are not available in the UMIACS Condor pool. Instead, a number of jobs equal to the number of slots are submitted to each node in the Condor pool. As each of these jobs start, they coordinate with the others on that same host until all slots are occupied by idle measurement jobs. Then, one of these jobs contacts the central service, notifying the central service that this particular node is available for measurement. After a period of one hour, these jobs yield by terminating and resubmitting themselves to ensure that jobs from other users can proceed. At the central service, controlled nodes are matched with a preference for pairs that have not yet been measured. When a measurement is complete, the result is reported back to the central service. Both nodes are then free to be matched, again.

For several reasons, making measurements with MPI was difficult. These challenges came in three forms: Getting OpenMPI to work, getting around policy choices to actually start MPI measurements, and finally getting InfiniBand[3] to work correctly. There were two problems with OpenMPI itself. First, to capture library incompatibilities, OpenMPI and the OSU microbenchmarks had to be compiled separately on every single host. Even within a cluster, there were operating system version differences that required this extra work. Second, OpenMPI attempts to use all available interfaces for a given interconnect to maximize bandwidth. However, the operating system[4] configures a virtual IP interface for virtual machine hosting, so it had to be explicitly excluded when starting a measurement.

To start a measurement, OpenMPI requires starting a process called `orted` on all participating nodes. In a cluster managed with conventional resource man-

[3]A high-bandwidth, low-latency network interconnect.

[4]RedHat Enterprise Linux (RHEL)

agement like Torque, SGE, etc., the underlying resource management system will manage this when an MPI process is started. Otherwise, OpenMPI uses `ssh` or `rsh` to start processes remotely. Conveniently, Condor provides an ssh-to-job facility. However, it requires having an account on both ends, which was not possible for policy reasons. That being said, Condor still lets arbitrary processes run and listen on non-privileged ports. So, a fake `rsh` interface was created, with just enough capability to run `orted` on the remote end. Unfortunately, it is hard to guarantee that *ad hoc* solutions like this are not vulnerable to exploitation.

In addition to many hosts without any special hardware, two clusters in the Condor pool are connected internally with InfiniBand. Ideally, both of these could be useful for parallel computation. However, some problems were encountered when trying to use them. On both clusters, there were limits on the amount of shared memory that could be locked. This prevented using InfiniBand as the transport. So, systems staff was notified and an adequate solution was found. There was a policy set through Pluggable Authentication Modules (PAM) so that logins through `ssh` and jobs started through Torque had the correct limit. However, Condor does not use PAM, so processes spawned by Condor did not observe that policy. That policy had to be implemented by hand in to the Condor `init` script. This change was satisfactory on the cluster designed from the start as a parallel computer. However, InfiniBand on the other cluster was being used merely as fast a TCP/IP transport (IP over IB). Unfortunately, the InfiniBand driver supplied by the OS vendor is inadequate for Remote Direct Memory Access (RDMA), which is necessary for efficient MPI over InfiniBand. Staff cannot update the driver to a working one unsupported by the vendor, since the cluster is working as intended. On the other end, the OS vendor will not fix the driver. As a result, only one useful cluster is in the dataset making it much less useful for exploring meta-scheduling.

Making a measurement should be easy. However, it is not for a variety of reasons both technical and policy-related. It was hard to make these measurements despite having intimate knowledge of the systems administration and systems programming challenges. This poses some questions: How hard would it be for someone else? How much effort would it take to get an automated, decentralized system past these problems? How realistic is it that latent parallel resources can be exploited, when a problem as simple as a broken driver distributed by a vendor can derail efficient parallel communication? I had hoped that my work would be useful in the future for detecting *ad hoc* clusters where efficient RDMA technologies like RDMA over Converged Ethernet (RoCE) and iWARP are available.

Bibliography

[1] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967 – 2982, 2014.

[2] T.S.E. Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 170–179 vol.1, 2002.

[3] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM '04: Proceedings of the 2004 Conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2004. ACM.

[4] Sukhyun Song, P. Keleher, B. Bhattacharjee, and A. Sussman. Decentralized, accurate, and low-cost network bandwidth prediction. In *Proceedings 2011 IEEE INFOCOM*, pages 6 –10, April 2011.

[5] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[6] Jik-Soo Kim, Beomseok Nam, Peter Keleher, Michael Marsh, Bobby Bhattacharjee, and Alan Sussman. Resource discovery techniques in distributed desktop grid environments. In *GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 9–16, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Jaehwan Lee, Pete Keleher, and Alan Sussman. Decentralized resource management for multi-core desktop grids. In *Proceedings of the 24th International Parallel & Distributed Processing Symposium*. IEEE Computer Society Press, April 2010.

[8] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[9] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[10] Jik-Soo Kim, Beomseok Nam, M. Marsh, P. Keleher, B. Bhattacharjee, and A. Sussman. Integrating categorical resource types into a p2p desktop grid system. In *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 284–291, Sept 2008.

[11] Jaehwan Lee, Pete Keleher, and Alan Sussman. Decentralized dynamic scheduling across heterogeneous multi-core desktop grids. In *Proceedings of the 19th Internations Heterogeneity in Computing Workshop (HCW2010)*. IEEE Computer Society Press, April 2010. Appears with Workshop Proceedings of IPDPS 2010.

[12] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson. The end-to-end effects of internet path selection. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 289–299, New York, NY, USA, 1999. ACM.

[13] Han Zheng, EngKeong Lua, Marcelo Pias, and TimothyG. Griffin. Internet routing policies and round-trip-times. In Constantinos Dovrolis, editor, *Passive and Active Network Measurement*, volume 3431 of *Lecture Notes in Computer Science*, pages 236–250. Springer Berlin Heidelberg, 2005.

[14] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[15] Ittai Abraham, Mahesh Balakrishnan, Fabian Kuhn, Dahlia Malkhi, Venugopalan Ramasubramanian, and Kunal Talwar. Reconstructing approximate tree metrics. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 43–52, New York, NY, USA, 2007. ACM.

[16] Peter Buneman. A note on the metric properties of trees. *Journal of Combinatorial Theory, Series B*, 17(1):48 – 50, 1974.

[17] Venugopalan Ramasubramanian, Dahlia Malkhi, Fabian Kuhn, Mahesh Balakrishnan, Archit Gupta, and Aditya Akella. On the treeness of Internet latency

and bandwidth. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, pages 61–72, New York, NY, USA, 2009. ACM.

[18] Sukhyun Song, Pete Keleher, and Alan Sussman. Searching for bandwidth-constrained clusters. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 655–664, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[20] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.

[21] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 311–320, New York, NY, USA, 1997. ACM.

[22] Sylvia Paul Ratnasamy. *A Scalable Content-addressable Network*. PhD thesis, 2002. AAI3082374.

[23] Robert L. Henderson. Job scheduling under the portable batch system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[24] A. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, IPPS '98, pages 542–, Washington, DC, USA, 1998. IEEE Computer Society.

[25] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12:529–543, 2001.

[26] Gladys Utrera, Julita Corbalan, and Jess Labarta. Scheduling parallel jobs on multicore clusters using cpu oversubscription. *The Journal of Supercomputing*, 68(3):1113–1140, 2014.

[27] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[28] Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, London, UK, 1998. Springer-Verlag.

[29] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM conference on Computer and communications security*, CCS '98, pages 83–92, New York, NY, USA, 1998. ACM.

[30] Mathilde Romberg. The unicore grid infrastructure. *Sci. Program.*, 10:149–157, April 2002.

[31] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[32] Michael Litzkow and Miron Livny. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of the Winter 1992 USENIX Conference*, pages 283–290, San Francisco, CA, January 1992.

[33] Todd Tannenbaum and Michael Litzkow. Checkpointing and migration of unix processes in the Condor distributed processing system. *Dr Dobbs Journal*, February 1995.

[34] Jim Pruyne and Miron Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, Townsend, TN, May 1994.

[35] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[36] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Future Gener. Comput. Syst.*, 12(1):53–65, May 1996.

[37] Luis F.G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561 – 572, 2002. Best papers from Symp. on Cluster Computing and the Grid (CCGrid2001).

[38] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. Seti@home-massively distributed computing for seti. *Computing in Science Engineering*, 3(1):78–83, Jan 2001.

[39] David P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop*

on Grid Computing, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[40] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, HPDC '01, pages 55–66, Washington, DC, USA, 2001. IEEE Computer Society.

[41] Joanna Kolodziej and Samee Ullah Khan. Multi-level hierarchic genetic-based scheduling of independent jobs in dynamic heterogeneous grid environment. *Inf. Sci.*, 214:1–19, December 2012.

[42] Qingjiang Wang, Yun Gao, and Peishun Liu. Hill climbing-based decentralized job scheduling on computational grids. In *Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences - Volume 1 (IMSCCS'06) - Volume 01*, IMSCCS '06, pages 705–708, Washington, DC, USA, 2006. IEEE Computer Society.

[43] Stefka Fidanova and Mariya Durchova. Ant algorithm for grid scheduling problem. In *Proceedings of the 5th international conference on Large-Scale Scientific Computing*, LSSC'05, pages 405–412, Berlin, Heidelberg, 2006. Springer-Verlag.

[44] C. Fayad, J.M. Garibaldi, and D. Ouelhadj. Fuzzy grid scheduling using tabu search. In *IEEE International Fuzzy Systems Conference, 2007.*, pages 1 –6, July 2007.

[45] H. Abbes, C. Cerin, and M. Jemni. Bonjourgrid as a decentralised job scheduler. In *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pages 89 –94, dec. 2008.

[46] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[47] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness ii: On completeness for w[1]. *Theoretical Computer Science*, 141(12):109 – 131, 1995.

[48] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Comput. Oper. Res.*, 38(2):571–581, February 2011.

[49] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, pages 299–311, 2007.

[50] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[51] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[52] Dejan Perkovic and Peter J. Keleher. Randomization, speculation, and adaptation in batch schedulers. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.