ABSTRACT

Title of dissertation: DECENTRALIZED RESOURCE ORCHESTRATION FOR HETEROGENEOUS GRIDS

Jaehwan Lee, Doctor of Philosophy, 2012

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

Modern desktop machines now use multi-core CPUs to enable improved performance. However, achieving high performance on multi-core machines without optimized software support is still difficult even in a single machine, because contention for shared resources can make it hard to exploit multiple computing resources efficiently. Moreover, more diverse and heterogeneous hardware platforms (e.g. general-purpose GPU and Cell processors) have emerged and begun to impact grid computing. Given that heterogeneity and diversity are now a major trend going forward, grid computing must support these environmental changes.

In this dissertation, I design and evaluate a decentralized resource management scheme to exploit heterogeneous multiple computing resources effectively. I suggest resource management algorithms that can efficiently utilize a diverse computational environment, including multiple symmetric computing entities and heterogeneous multi-computing entities, and achieve good load-balancing and high total system throughput. Moreover, I propose expressive resource description techniques to ac-

commodate more heterogeneous environments, allowing incoming jobs with complex requirements to be matched to available resources.

First, I develop decentralized resource management frameworks and job scheduling schemes to exploit multi-core nodes in peer-to-peer grids. I present two new load-balancing schemes that explicitly account for resource sharing and contention across multiple cores within a single machine, and propose a simple performance prediction model that can represent a continuum of resource sharing among cores of a CPU. Second, I provide scalable resource discovery and load balancing techniques to accommodate nodes with many types of computing elements, such as multi-core CPUs and GPUs, in a peer-to-peer grid architecture. My scheme takes into account diverse aspects of heterogeneous nodes to maximize overall system throughput as well as minimize messaging costs without sacrificing the failure resilience provided by an underlying peer-to-peer overlay network. Finally, I propose an expressive resource discovery method to support multi-attribute, range-based job constraints. The common approach of using simple attribute indexes does not suffice, as range-based constraints may be satisfied by more than a single value. I design a compact ID-based representation for resource characteristics, and integrate this representation into the decentralized resource discovery framework.

By extensive experimental results via simulation, I show that my schemes can match heterogeneous jobs to heterogeneous resources both effectively (good matches are found, load is balanced), and efficiently (the new functionality imposes little overhead).

DECENTRALIZED RESOURCE ORCHESTRATION FOR HETEROGENEOUS GRIDS

by

Jaehwan Lee

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
2012

Advisory Committee:
Professor Alan Sussman, Chair/Advisor
Professor Pete Keleher
Professor Atif M. Memon
Professor Jeffrey Hollingsworth
Professor Derek C. Richardson

© Copyright by
Jaehwan Lee
2012

Dedication

To my wife - Hyunjoo Park for her endless love and support.

Acknowledgments

I owe my gratitude to all the people who helped me complete this dissertation.

First and foremost I'd like to express immeasurable gratitude to my advisor, Professor Alan Sussman, who gave me an invaluable opportunity to work on challenging and extremely interesting projects over the past years. He always guided me to the right directions and encouraged me in all the time of research.

I would also like to thank my co-advisor, Dr. Pete Keleher. He always gave me clear guidelines for the right track throughout this work. I would like to thank my committee members, Dr. Ratify Memon, Dr. Derek Richardson and Dr. Jeffrey Hollingsworth for sharing their time and efforts to review the manuscript. They gave me excellent suggestions to improve the quality of my dissertation.

My lab colleagues have enriched my graduate life in many ways and deserve a special mention. I am indebted to my former college , Jik-soo Kim who was my research mentor and helped me build a foundation of my research. I was so lucky to spend most of my graduate school days with Sukhyun Song, who gave me valuable comments on my research. I also thank Beomseok Nam, Ilchul Yoon, Shang-Chieh Wu, Gary Jackson and Teng Long.

I am very grateful to my close friends, Soobum Lee, Hyunyoung Song, Sungwoo Park, Youngmin Kim, Eunhui Park, Jinhyuk Jung, Inseok Choi, Minkyung Cho, Jisun Shin, Seungjoon Lee, Hyunmo Kang, Minho Shin, Sangchul Song, Takyeon Lee, Chanhyun Kang, Youngil Kim, Kyungjin Yoo, Woomyoung Park, Eunyoung Seo, Hyuk Oh, Jonghyun Choi, Dongwoon Hahn, and Hojin Kee. I could not have

enjoyed my graduate days in Maryland without them.

Last but not least, I owe my deepest thanks to my parents. Their endless support and love motivated me in moving forward to become their proud son. Finally, I cannot express my gratitude in any words to my wife, Hyunjoo Park for her endless love, patience and support.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1
- 1.1 Motivating Applications . . . 4
- 1.2 Thesis and Contributions . . . 5
- 1.3 Outline . . . 8

2 Overview of Peer-to-Peer Grid System 9
- 2.1 Overall System Architecture . . . 9
- 2.2 Matchmaking Algorithm . . . 11
- 2.3 Advanced Load Balancing . . . 14
- 2.4 Categorical Resource Type . . . 17

3 Matchmaking Framework in a Multi-core Grid 20
- 3.1 Resource Management in a Multi-core Grid . . . 21
  - 3.1.1 Two Logical Nodes for a Multi-core Node . . . 21
  - 3.1.2 Dual-CAN Model . . . 23
  - 3.1.3 Balloon Model . . . 24
  - 3.1.4 Matchmaking for Multi-core Nodes and Multi-threaded Jobs . 26
  - 3.1.5 Model Comparison . . . 29
- 3.2 Parameterized Prediction Model for Resource Contention . . . 31
  - 3.2.1 Contention Penalty in Multi-core Nodes . . . 31
  - 3.2.2 Experimental Results . . . 33
  - 3.2.3 New Prediction Model . . . 35
- 3.3 Experiments . . . 37
  - 3.3.1 Experimental Setup . . . 37
  - 3.3.2 Experimental Results . . . 41
- 3.4 Summary . . . 54

4 Supporting Computing Element Heterogeneity 56
- 4.1 Resource Management for Heterogeneity . . . 56
  - 4.1.1 Accommodating Heterogeneous Nodes . . . 56
  - 4.1.2 Job Pushing for a Heterogeneous System . . . 58
- 4.2 Scalable Support for Heterogeneity . . . 62
  - 4.2.1 Maintenance Cost Analysis . . . 63
  - 4.2.2 Compact Heartbeat . . . 66
  - 4.2.3 Adaptive Heartbeat . . . 68
- 4.3 Experimental Results . . . 69
  - 4.3.1 Load Balancing Performance . . . 70
  - 4.3.2 Scalability and Heterogeneous Resources . . . 74
- 4.4 Summary . . . 79

5 Multi-attribute Range Search 80
5.1 Range-type Search Algorithm 80
5.1.1 ID-based Resource Representation 81
5.1.2 Multi-attribute Requirements 84
5.1.3 Implementation Choices 86
5.2 Experimental Results 89
5.2.1 Load Balancing Performance 90
5.2.2 Cost Analysis 94
5.2.3 Exact Match Workload 96
5.2.4 Stale Information 98
5.3 Summary 103

6 Related Work 105

7 Conclusions and Future Work 111
7.1 Thesis and Contributions 111
7.2 Future Work 114

Bibliography 117

List of Tables

3.1 STREAM result: normalized running time (slow down) on a multi-core machine . . . 33

4.1 Normalized Running Times . . . 60
4.2 Comparison Summary of Vanilla CAN, Compact Heartbeat and Adaptive Heartbeat . . . 69

5.1 Node Resource Capabilities & CIDs . . . 82
5.2 Nodes Resource Capabilities & CIDs: Bit String Representation . . . 86

List of Figures

2.1 Overall System Architecture 11
2.2 Matchmaking Mechanism in CAN 13
2.3 Job Pushing in the CAN 16
2.4 Matchmaking across sub-CANs: Solid arrows denote the physical routing path of job J, while dotted arrows show the logical path. . . . 17

3.1 Changes for the two logical nodes when jobs are assigned to the node 22
3.2 Dual-CAN: Node B creates Residue-node C on Secondary CAN when running a job 23
3.3 Balloon model: Node B is assigned the same job as in Figure 3.2 . . 25
3.4 Normalized Running Time vs. Number of Concurrent Jobs 34
3.5 Expected running time ratio α with respect to shared resource usage 37
3.6 Cumulative distributions for Job Turn-around time 40
3.7 Costs of Dual-CAN, Balloon and MP 42
3.8 Cumulative Distributions of the Job Waiting Time 43
3.9 Costs for Dual-CAN, Balloon and Vanilla CAN 44
3.10 Snapshots of number of jobs in the system varying job inter-arrival time 45
3.11 CDF of Job wait time varying Inter-arrival Time 46
3.12 CDF of Normalized Job wait time varying Inter-arrival Time 47
3.13 CDF of Job wait time varying Job Constraint Ratio 49
3.14 CDF of Normalized Job wait time varying Job Constraint Ratio . . . 50
3.15 Snapshots of number of jobs in the system varying job inter-arrival time 51
3.16 CDF of Job wait time under different job running time distribution . 52
3.17 Over-provisioning effect in Centralized Matchmaker 54

4.1 Recovery from a Broken Link via Heartbeats 64
4.2 Zone Splits and Take-over Nodes 64
4.3 Worst Case for Compact Heartbeat 64
4.4 CDF of Job wait time varying Inter-arrival Time 71
4.5 CDF of Job wait time varying Job Constraint Ratio 72
4.6 Broken Links under high churn 76
4.7 Scalability, measured per node per minute 77

5.1 An Integrated CAN with four nodes: lower case letters denote CIDs, and each set next to the dotted-arrow shows aggregated CID information flow 82
5.2 Integrated CAN for four nodes: Bit string approach 87
5.3 CDF of Job wait time varying Inter-arrival Time 92
5.4 CDF of Job wait time varying Job Constraint Ratio 93
5.5 CDF of number of Routing hops for matchmaking 95

5.6 Information Accuracy and System Performance Varying Heartbeat Periods: The Node Churn Rate is 5 Seconds 100
5.7 Information Accuracy and System Performance Varying Heartbeat Periods: The Node Churn Rate is 10 Seconds 101

Chapter 1

Introduction

Modern computing machines now use multi-core CPUs to enable improved performance. However, achieving high performance on multi-core machines without optimized software support is still difficult even in a single machine, because contention for shared resources can make it hard to exploit multiple computing resources efficiently. For example, Moore [1] shows that multi-core processors cannot guarantee increased performance for supercomputing applications, and that some applications may even have worse performance than with single-core nodes. In addition to the fact that multi-core machines are becoming overwhelmingly popular in grid computing systems, more diverse and heterogeneous hardware platforms (e.g. GPGPU(General Purpose computation on Graphics Processing Units) technology and Cell Processors) have emerged and begun to impact on grid computing area recently. For example, Nvidia's CUDA solution on its GPU hardware [2] can achieve tremendous computing performance for iterative scientific computation with relatively low costs [3]. Some researchers have showed experimental results using machines with Cell Processors supported by wide-spread grid platforms such as Condor [4], BOINC [5, 6] and Folding@Home [7]. Given that heterogeneity and diversity is the major trend of grid computing, we need to support these environment changes inevitably. In addition, a decentralized resource management scheme

should be employed for a reliable and scalable grid system, because a client-server based centralized approach is vulnerable to a single point of failure (lack of reliability) and has a performance bottleneck (lack of scalability). Fortunately, we have designed an effective peer-to-peer grid solution for single-core machines.

However, a simple extension of single-core approach is not effective because we have several following issues in order to get optimized performance by exploiting multiple heterogeneous computing entities.

- *Contention for shared resource* : Multiple jobs (or multi-threaded jobs) can contend for shared resources (e.g. memory), and those contentions degrade overall performance dramatically. Assigning two or more jobs in a multi-core machine without taking the contention effects into account cannot guarantee performance improvement. However, quantitative prediction for the contention effects and job allocation based on the parameterized model is not an easy problem to solve.

- *Reducing excessively frequent updates in the peer-to-peer network* : To leverage available resources effectively, advertising current resource status across nodes should be required. However, this frequent dynamic information update in peer-to-peer (P2P) system can be excessive. To provide effective dynamic update without generating excessive changes is not easy to implement efficiently.

- *Job Assignment across heterogeneous nodes* : How can we find the best node that satisfies all requirements of a job in a peer-to-peer system for better total throughput and load balance (called *matchmaking*) ? If we have multiple

choices with different resource capabilities, a policy for decision is required and becoming more difficult as distributed resources are more heterogeneous.

- *Expressiveness for heterogeneous requirements* : Heterogeneity and diversity in resources (and job requirements) need more expressive methods for description and job allocation. Our eventual goal is to provide a more generalized platform for resource description and discovery techniques in a decentralized way.

To address these issues, I have designed a new decentralized resource management framework to extend our basic P2P grid system. This framework has the following advantages. First, I have developed an elastic resource advertisement and a discovery technique on top of the current P2P overlay based system to accommodate multi-core nodes. This technique includes the efficient job allocation method to take the contention effect for shared resources into account. Second, I have developed a scalable matchmaking system for heterogeneous nodes having multiple, different kinds of computing elements. The extended matchmaking system can assign jobs to exploit heterogeneous nodes effectively while it limits the excessive communication volume. Third, this system supports more flexible formats of resource descriptions, for example, range-type resource constraints/capabilities. The range-based resource description allows allocating jobs in a more flexible and efficient way. Throughout extensive experimental results, I show that the newly extended P2P grid system can exploit multi-core nodes as well as more heterogeneous nodes in a scalable way, and effectively support more flexible resource descriptions.

1.1 Motivating Applications

Our target applications are mostly computation intensive but don't require frequent I/O access or large data access. Examples of the target applications include bio-informatics (e.g. protein folding, HIV/AIDS, neural disorders), computational finance (high frequency trading, Monte Carlo), and various scientific applications in physics/astronomy area. Some practical examples are as follows.

Bio-informatics

There are many computation intensive problems in bio-informatics. For example, Folding@home [7] project investigates how to get correct a DNA sequence in protein. The main problem is that only knowing this sequence does not tell us enough to find what the protein does and how it works. The protein should *fold* to be a particular shape in order to carry out the specific functions. This problem needs a huge amount of computation. For example, in order to simulate folding instance, it needs a microsecond interval in the simulation time scale, which requires 10,000 CPU days [8]. By leveraging volunteer desktop computing, the CPU time can be reduced significantly. Recently the scientists begin using GPGPU technology in their platform [5, 6] to improve computation performance. In addition to the protein folding problem, there are so many computation intensive projects related to HIV/AIDS, neural disorders, and cancers in the bio-informatics area.

Computational Finance

The current real-time, high-frequency trading system allows automated transactions

based on given mathematical algorithms. The issue in the machine-based trading is who can get the result quickly and send a sell or buy request to the trading market, because the algorithm itself would not be very different. To get the fastest response, exploiting distributed computational resources is very important. Parallel or distributed computation in grid-like environments can reduce computation time significantly compared to using a single machine. In addition to the distributed approach, nowadays they start using a GPGPU in a grid node to reduce computation time. As more heterogeneous hardware resources are emerging, we should take care of this trend for heterogeneity to get better results.

Scientific Computation in Physics/Astronomy

In the astrophysics area, some research problems need tremendous amounts of computation. One example of such projects is Einstein@home project [9]. The objective of this project is to search for weak astrophysical signals from spinning neutron stars. They use data from gravitational-wave detectors, radio telescopes, and Fermi gamma-ray satellites. The main computation is to solve the matching problem in a frequency domain, so computation can be easily distributed according to the search spaces. Recently they announce to support OpenCL platform [10], which mainly targets to leverage GPGPU resources.

1.2 Thesis and Contributions

In this dissertation, I support the following thesis: *decentralized resource management scheme can be employed to exploit heterogeneous multiple computing re-*

sources in grid systems. To support my thesis, I develop, apply, and evaluate a set of techniques for building an effective and scalable P2P grid system for heterogeneous nodes and jobs.

More specifically, this dissertation makes the following contributions:

1. **Effective contention-aware job scheduling for multi-core nodes**

   While the majority of CPUs now sold contain multiple computing cores, current grid computing systems either ignore the multiplicity of cores, or treat them as distinct, independent machines. The latter approach ignores the resource contention present between cores in a single CPU, while the former approach fails to take advantage of significant computing power. I provide a decentralized resource management framework for exploiting multi-core nodes in peer-to-peer grids. I present two new load-balancing schemes that explicitly account for the resource sharing and contention of multiple cores, and propose a simple parameterized performance prediction model that can represent a continuum of resource sharing among cores of a CPU.

2. **Scalable resource management framework for heterogeneous environments**

   Since GPGPU technology has emerged and begun to impact grid computing, a grid node can have multiple, different types of computing elements. Unfortunately, a straightforward extension of current P2P framework for heterogeneous nodes cannot be effective or scalable because first, multiple computing elements can be of different types so that their performance characteristics can

vary greatly, and second, the number of resource attributes increases as the node becomes more heterogeneous, so resource advertisement and discovery in our P2P framework can suffer from more communication costs. To address these issues, I provide an effective extended P2P scheme to take the node's heterogeneity into account. The new scheme makes good scheduling decisions in scenarios where both job requirements and nodes can contain multiple, possibly heterogeneous, computing elements. Moreover, I develop a set of mechanisms that limit communication cost growth incurred by heterogeneity without sacrificing failure resilience, one of the key advantages of P2P systems.

3. **Range-type resource discovery and load balancing**

   The recent trend toward heterogeneity among, and even within, computers requires new expressiveness in the way resource descriptions are created, and new sophistication in systems that attempt to match jobs with resources. To enable more expressive job description, I propose a novel resource discovery and load balancing method to support multi-attribute, range-based job constraints in a peer-to-peer grid system. The common approach of using simple attribute indexes does not suffice, as range-based constraints may be satisfied by more than a single value. I propose a compact new representation for resource characteristics, and integrate this representation into the existing decentralized resource discovery framework. In addition, our system relies on resource descriptions being aggregated and periodically disseminated. Decentralized algorithms are often sensitive to the "freshness" of this information.

However, I show that this approach is not sensitive to stale data, allowing the information movement to happen rarely enough that it imposes only insignificant overhead.

1.3 Outline

The rest of this dissertation is organized as follows. First, Chapter 2 describes our basic peer-to-peer grid platform. Chapter 3 presents efficient resource management and job scheduling schemes for multi-core nodes. We discuss a more extensive, scalable framework for more heterogeneous nodes which have multiple, different types of computing resources in Chapter 4. In Chapter 5, we describe an expressive framework for range-type resource discovery and load balancing. In Chapter 6, we present some related work on P2P grid and heterogeneous computing. Chapter 7 presents conclusions, summarizes the work and suggests possible directions for future work.

Chapter 2

Overview of Peer-to-Peer Grid System

In this chapter, we describe the basic peer-to-peer grid platform including how jobs are submitted and matchmaking is performed to an appropriate node in a decentralized way. First, we describe the basic system architecture including underlying assumptions and nodes/jobs characteristics. Second, we discuss our basic matchmaking framework based on Content-Addressable Network (CAN) [11]. Third, advanced load balancing protocols are presented using probabilistic job pushing method. Finally, we describe how to perform matchmaking when a discrete type of resource constraints that requires a singular value for that resource (i.e. exact match) are given in our CAN-based framework.

2.1 Overall System Architecture

Our system is built on a variant of an existing distributed hash table (DHT) to organize peer-to-peer structures among nodes [12, 13, 11, 14, 15, 16]. DHT uses a random hash function to map nodes in a uniform way on the space, and this randomness enables inserting and looking up a node in a simple way. In our system, every node is assigned a GUID (Global Unique Identifier) for the DHT and with a given random GUID each node can maintain a unified, organized single structure by exchanging information about pointers to only a small subset of nodes in the

system.

A job in our system is composed of a profile that describes how to compute the job result. The job profile includes the locations of input data and the executable program, its minimum resource requirements, and information about the client submitting the job. We assume that each job is independent so that no communication between jobs is needed [17]. This is a typical scenario for desktop grid computing systems, enabling many independent users to submit their jobs to a collection of resources, for embarrassingly parallel workloads [18]. The following steps outline the procedure for injecting and executing a job in the system.(See Figure 2.1)

1. A client inserts a job into the system through an arbitrary node called the *injection node*.

2. The injection node assigns a GUID to the job and initiates CAN routing of the job to the *owner node*.

3. The owner node begins the matchmaking process to find a lightly loaded node (*run node*) that meets all of the job's resource requirements.

4. Once a run node is determined, the owner node sends the job to the run node.

5. The run node inserts the job into an internal FIFO queue for job execution. Periodic heartbeat messages between the run node and owner node ensure that both are still alive. Missing consecutive heartbeats triggers a failure recovery procedure.

6. After finishing the job, the run node delivers the results to the client.

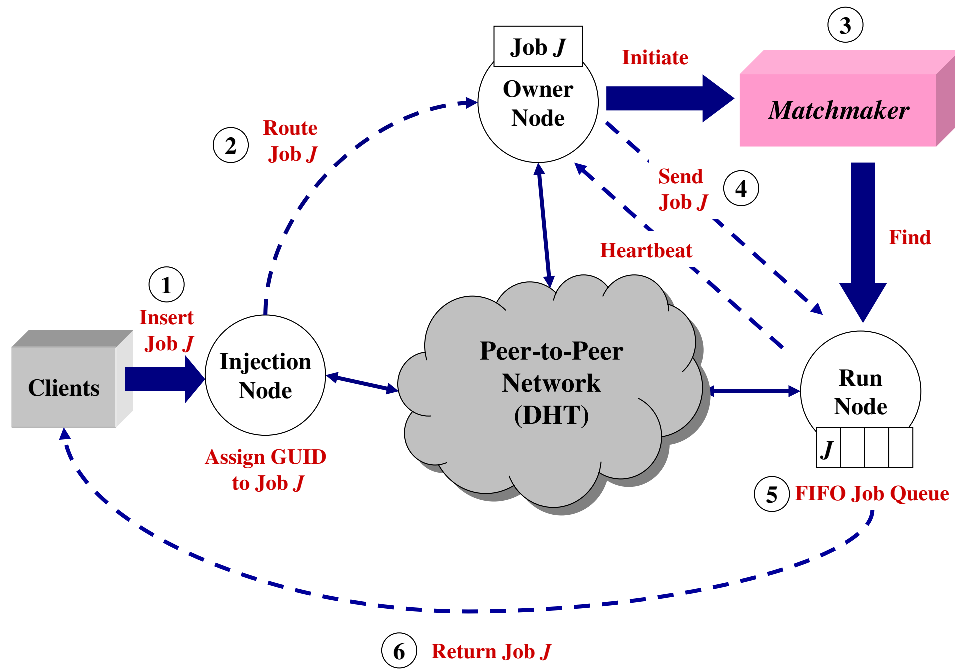


Figure 2.1: Overall System Architecture

The owner node is responsible for monitoring job status until the job finishes execution and returns the result to the client. The run node and owner node exchange soft-state heartbeat messages for recovering from voluntary departure or failure of either node. More details about our basic architecture are presented in earlier work [19, 20].

2.2 Matchmaking Algorithm

Matchmaking is the process of assigning a job to a node that both satisfies the job's requirements and is also lightly loaded. A good matchmaking algorithm should meet the following criteria:

- **Expressiveness** The matchmaking framework should be expressive enough

to specify job requirements as well as node capabilities.

- **Load Balance** Jobs should be distributed evenly across the nodes to maximize the total throughput of the system.

- **Parsimony** A node's resources should not be wasted as a result of over-provisioning.

- **Completeness** As long as the system has a node that is capable of running the job, the matchmaking algorithm must find a run node.

- **Low Overhead** The matchmaking process should not incur significant overhead.

Among various DHTs, we use a CAN to implement a peer-to-peer system in a structured way so that we can use the multiple dimensions of a CAN to represent the resource attributes of grid nodes and jobs. The original CAN design maps a node to a point in a d-dimensional space by hashing a GUID. The space is divided into non-overlapping hyper-rectangular zones that each maintain neighbor information. However, nodes' positions in the CAN may not be randomly distributed. For example, Tang et al. [21] map documents and queries into a multi-dimensional CAN space where each dimension measures the relevance of a particular index term. By this mapping, they can execute queries via a local search centered on a query's mapping location. Similarly, each dimension in our modified CAN represents the amount of a particular resource type for a node, or that resource's requirement for a job, and each node is sorted along each dimension according to the amount of

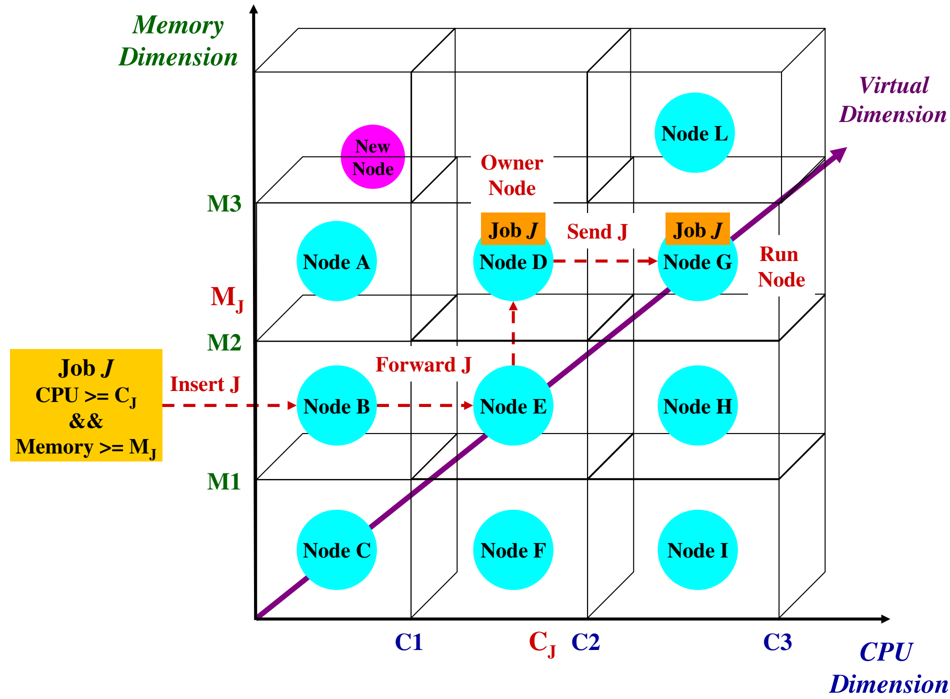


Figure 2.2: Matchmaking Mechanism in CAN

that resource in the CAN space so that we can solve the matchmaking problem as a routing problem in the CAN as in Figure 2.2.

Figure 2.2 shows an example for matchmaking Job J and Node G with two resource types, CPU speed and Memory size, through routing in the CAN space. To measure CPU performance, we may use FLOPS, IPS or CPU measures for standard benchmarks, such as SPEC, instead of CPU clock speed [22, 23, 5]. We use CPU speed for CPU performance metric because it is simple for general grid users to specify and often effective [24]. The Job J is inserted into the system using its requirements as coordinates ($\{C_J, M_J\}$ and routed to the zone (owner node, Node D) containing those coordinates. Among upper and right neighbor nodes from the owner nodes in the Figure 2.2, the *least loaded* node that meets resource require-

ments will be selected as a run node; the load information is exchanged periodically between nodes piggybacked on CAN heartbeat messages. *Load balance* property is satisfied by picking the least loaded node among neighbors. *Parsimony* and *Expressiveness* follow naturally from the fact that the owner node of a job maintains the zone containing the coordinates of a job (corresponding to its minimum resource requirements), so the minimally capable nodes for a job are neighbors (or next-nearest neighbors) of the owner. Also, under the assumption that there is always at least one node capable of running a job, *Completeness* can easily be assured by the CAN routing, which in the worst case will eventually map a job to the most-capable node in the system (the node occupying the extreme corner or the edge of the CAN space).

The original CAN does not allow two nodes to have identical coordinates, but multiple nodes with the same resource capabilities can exist in the CAN. To address this problem, we add another dimension to the CAN that has randomly generated values for both nodes and jobs, called the *virtual dimension*. Therefore, multiple nodes with the same resource capabilities can be differentiated in the CAN space via the random coordinate in the virtual dimension. The randomly assigned virtual dimension value for jobs also is used to improve load balance across nodes.

2.3 Advanced Load Balancing

The simple load balancing scheme based on random virtual dimension coordinates does not always show good performance. We have improved the matchmaking

algorithm by *pushing* jobs in a probabilistic way through the CAN space, to find less loaded nodes [25]. The key idea is to push a job into under-loaded regions in the CAN space (still satisfying its resource requirements), to select a lightly loaded node from those nodes that meet the job resource requirements. To provide the required load information in the decentralized system, we first aggregate information, such as the number of nodes and average job queue length in each CAN dimension, by piggybacking onto the heartbeat messages that are used to maintain the CAN DHT. Based on that aggregated information, in the matchmaking process a node chooses a dimension and a target node to push a job that is being matched. However, before pushing the job, the node computes a stopping probability to decide whether to *stop* or *push*. If the job stops, we search for the best (most lightly loaded) run node among the node neighbors that satisfy the job resource requirements. Otherwise, the job continues to be pushed through the CAN for better load balancing. The system with the job pushing scheme balances load more effectively than the simple scheme and improves overall system throughput.

Figure 2.3 shows a simple example of job pushing in a 2-dimensional CAN. A node's coordinate is represented by a circle, and the zones for the nodes are partitioned by the dotted lines. Suppose that job J is inserted via node C with the coordinate (C_J, M_J). First, job J is routed to its coordinate (in node F's zone) via CAN routing. In this example, nodes D through I can be the run node for job J, because they all satisfy the requirements for job J. For better load balancing, the job can be pushed towards upper regions in the CAN, and that is done using aggregated load information. For example, node F has aggregated load information along both

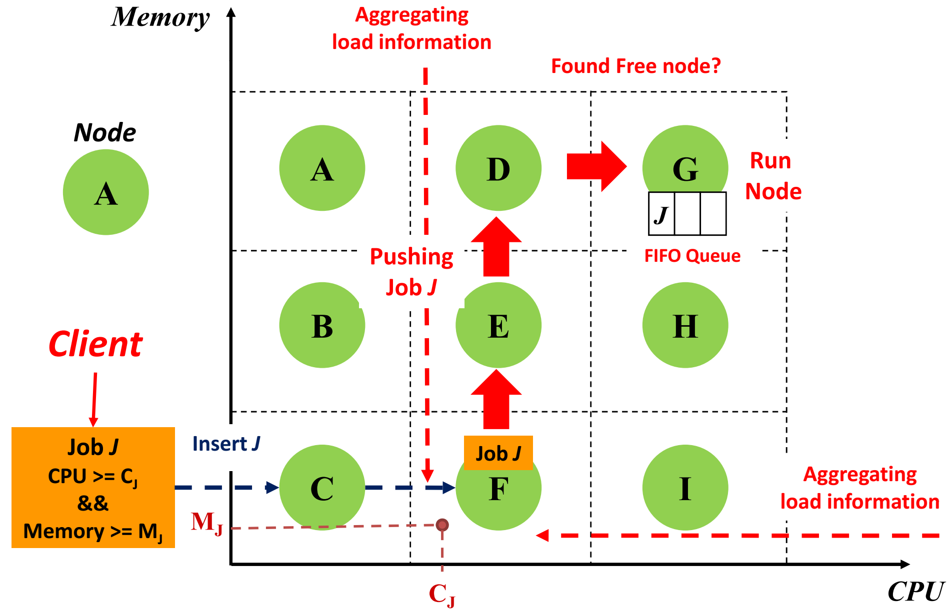


Figure 2.3: Job Pushing in the CAN

dimensions, and the job is pushed from node F to node E if the aggregated load along the *memory* dimension is less than in the *CPU* dimension. Similarly, the job can be pushed to node D from node E. But job pushing may stop at node D or E probabilistically (based on the likelihood of finding a node that can run the job immediately), though this example does not show probabilistic stopping. During the job pushing process at node D, suppose that node G is a *free-node*, meaning the node has no running or waiting jobs in its queue, so can run the job immediately. Then job pushing stops and job J is inserted in node G's waiting queue. More details on this probabilistic approach for job placement can be found in Kim et al.[25].

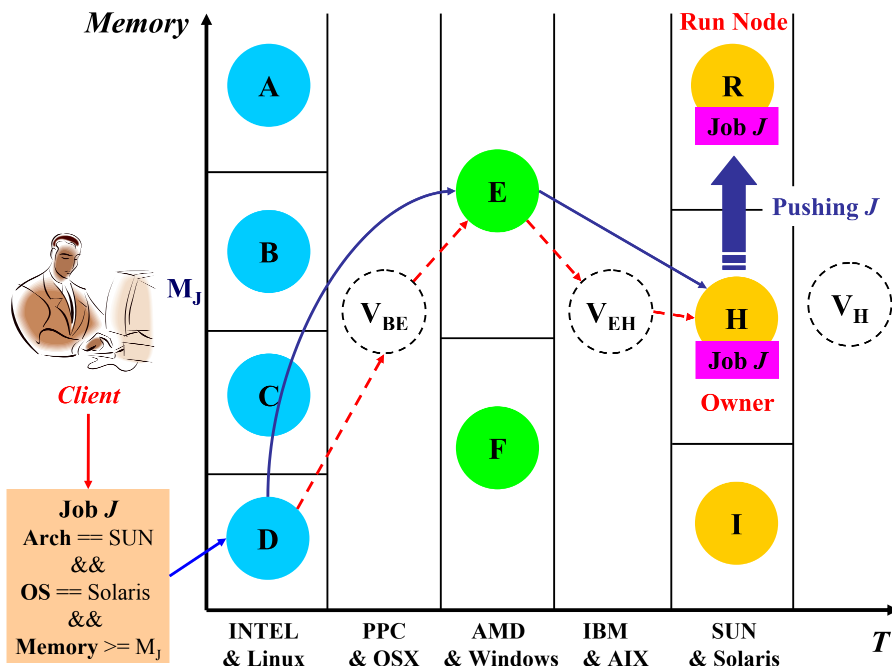


Figure 2.4: Matchmaking across sub-CANs: Solid arrows denote the physical routing path of job J, while dotted arrows show the logical path.

2.4 Categorical Resource Type

In our system, the resource requirement (or capability) for a job (or a node) can be either a *continuous* type (e.g. CPU speed $\geq$ 1GHz) or a *categorical* type (e.g. OS == "Linux"). Continuous constraints include CPU speed, memory size, disk space, etc. whereas categorical constraints include operating system type and CPU architecture. While Section 2.2 and Section 2.3 have addressed the matchmaking problem with only continuous resource types, we describe how to matchmake jobs with categorical resource constraints given as an exact match in this section. The system must be able to search for exact matches for the categorical resource types and minimum matches for the continuous resource types simultaneously, while balancing load among multiple candidate nodes.

To effectively integrate all types of resources into a CAN, first the CAN space is divided into multiple disjoint sub-spaces (called *sub-CANs*) where all of the categorical resource types in each sub-space are exactly the same, and an efficient mechanism is provided to connect the multiple sub-spaces [26]. With this design, each physical peer only is responsible for the exact region of the CAN space to which it belongs, with respect to its categorical resource specifications, and the rest of the space (unoccupied spaces) is covered by virtual peers. Since a virtual peer is not a physical node, each virtual peer can be mapped to physical peers (called manager nodes). For efficient management of sub-CANs and to simplify failure recovery, all categorical resource types are *transformed* into a single dimension (called the T dimension) using a Hilbert Space-Filling Curve [27]. We first route the given job to the sub-CAN having the same categorical resources values along the T dimension. Then the matchmaking process inside the sub-CAN is done efficiently using the previously described job pushing mechanism. To distinguish this approach from our new solution that we describe in the next chapter, we call it the *sub-CAN approach* in the rest of dissertation.

Figure 2.4 shows the overall procedure for matching a job J to a node R that meets both the categorical and continuous resource constraints of J. In the figure, V_{BE} and V_{EH} denote virtual peers covering the (PPC & OSX) and (IBM & AIX) sub-spaces respectively (i.e., no physical peer that belongs to either of these regions currently exists in the grid). During a matchmaking (routing) procedure across multiple sub-CANs, each node utilizes information about the neighbors of a virtual peer and once the request arrives at the right sub-CAN (in terms of categorical

resource types), the job pushing mechanism described in Section 2.3 is employed to achieve good load balance.

Chapter 3

Matchmaking Framework in a Multi-core Grid

In this chapter, we describe decentralized resource management and job scheduling techniques for a multi-core node in order to run multi-threaded jobs or multiple single-threaded jobs. Both to effectively utilize all available grid resources, including multi-cores, and to run jobs that request multiple cores (presumably because they are multi-threaded), we address two challenges as follows. First, when a multi-core node runs jobs but all cores are not used, the number of free cores and amount of available shared resources should be advertised to the grid system for matchmaking with future incoming jobs. Unfortunately, representing the dynamic aspects of node capability in current peer-to-peer grid systems is not straightforward. We describe a new efficient matchmaking framework for multi-core environments. Second, running concurrent jobs on a multi-core node can result in contention for shared resources such as memory, cache, etc. There is currently no simple analytical model to predict slow-down for this situation. We discuss a new straightforward approach to modeling the performance of multi-core nodes, using a *penalty factor* to account for resource contention. Third, we present extensive experiment results that show that our new approaches outperform multi-core oblivious approaches.

3.1 Resource Management in a Multi-core Grid

3.1.1 Two Logical Nodes for a Multi-core Node

To maximize the benefit of multi-core CPUs, each core should run a different job simultaneously. However, a job that requires a large amount of a shared resource, such as memory or disk space, should be able to run on the multi-core node, too. To accommodate multiple small jobs as well as a large job, matchmaking should be done based on current dynamic status information on shared resource usage within each grid node. The problem is that it is not easy to dynamically update the status of all nodes, even in a centralized desktop grid system. For example, Condor[28] uses a static resource partitioning method to utilize all cores in a node, but the partitioning of all available node resources must be configured by the grid administrator in advance. The problem is worse in structured P2P grid systems like our CAN, since they rely on low churn (nodes entering and leaving the system) to maximize throughput and minimize overhead. To minimize overhead as well as to advertise dynamic updates to the node information in our CAN based system, we introduce the idea of using *two* logical nodes to represent a single physical multi-core node.

First, we map a node to a *static* point in the CAN space, just as for a single-core node. The coordinates in the CAN space for this logical node, which we call the *Max-node*, are the maximum value for each resource type, regardless of current availability for each resource within the node. The Max-node has an internal job queue – it can be a free node or a busy node, where a free node has no waiting jobs

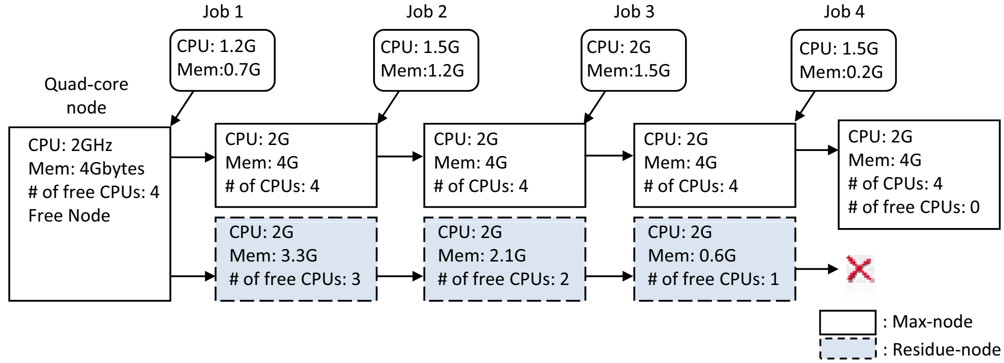
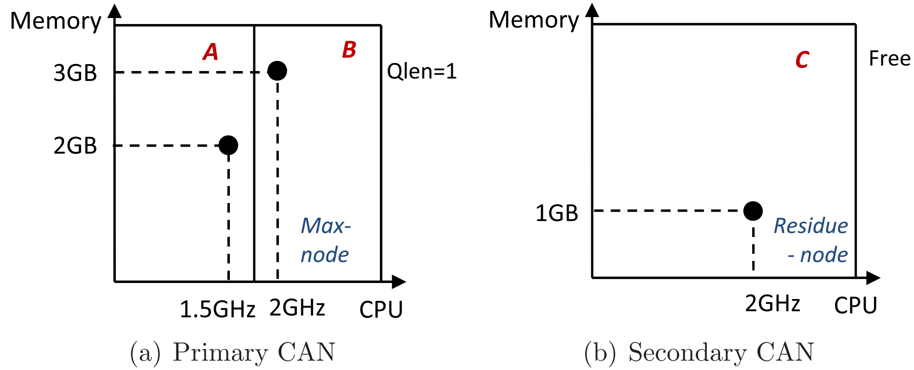


Figure 3.1: Changes for the two logical nodes when jobs are assigned to the node in its queue.

The other logical node, called the *Residue-node*, represents the currently available shared resources. The coordinates of a Residue-node may change frequently, since they represent the *dynamic* status of available shared resources in the node. (The coordinates for resource types that do not change, e.g. operating system type are those of the corresponding Max-node.) An interesting feature of a Residue-node is that it is *always* a free node; a Residue-node does not have an internal queue. If a Residue-node is assigned a job to run, it can start running the job immediately (no wait time). If a Max-node is a free node, or all CPU-cores are busy running jobs, then the Residue-node is not explicitly represented in the CAN.

Figure 3.1 shows the status of one Max-node and its Residue-node, and how to create and remove nodes on job arrival. To simplify this example, we consider only a 2-dimensional CAN. The main issue to explore is how to construct a CAN that performs matchmaking for jobs in the multi-core grid. Two approaches are described in the following sections.



(a) Primary CAN (b) Secondary CAN

Figure 3.2: Dual-CAN: Node B creates Residue-node C on Secondary CAN when running a job

3.1.2 Dual-CAN Model

The first model is *Dual-CAN*. The basic idea of Dual-CAN is to have a *Primary CAN* for the Max-nodes, and a *Secondary CAN* containing Residue-nodes. To reduce the overhead for frequent updates of dynamic node information, we separate the static node information from the dynamic information so that each CAN takes care of one type of logical nodes. Therefore, the overhead of the Primary CAN is the same as that of the original CAN. On the other hand, even though the overhead of the Secondary CAN is not negligible, the additional overhead is not large compared to the Primary CAN. This is for two reasons: the number of nodes in the Secondary CAN is much smaller than in the Primary CAN for a loaded system, and matchmaking is mainly done in the Primary CAN.

The Primary CAN contains both single-core nodes and the Max-nodes for multi-core nodes. The Primary CAN has low churn (only for nodes entering and leaving the grid) and contains one zone per physical node. Zone splitting (joining & leaving) is done the same way as in our earlier CAN. The Secondary CAN contains

the Residue-nodes for multi-core nodes. The Secondary CAN can be very dynamic because Residue-nodes may join and leave the CAN frequently, as jobs are matched to nodes and as jobs complete in the grid system, consequently requiring changes to the status of shared resources in the Residue-nodes. Figures 3.2(a) and 3.2(b) show the status of the Primary CAN and Secondary CAN, respectively, after a single-core node (with coordinates [1.5GHz, 2GB]) and a dual-core node (with coordinates [2GHz, 3GB]) join the grid and a job requiring 2GB memory and 1GHz CPU is matched to the dual-core node. Whenever a node (or a job) in the Primary CAN needs to contact the Secondary CAN, such as for a new node join or in the job pushing process, first if the node participates in the Secondary CAN, we can directly access the Secondary CAN. Second, the node will look at the information for its neighbors in the Primary CAN to see if any of them participate in the Secondary CAN. If any node in the Secondary CAN can be found in the neighbor lists, then we can use that node as an injection node to the Secondary CAN. If not, we can look up the injection node information in the Manager node, which provides a directory service for the secondary CAN so that we can access the Secondary CAN [26].

3.1.3 Balloon Model

To reduce the overhead of frequent join and leave operations for the Residue-node, we have separated the CAN into static and dynamic parts. However, the overhead from the Secondary CAN can be significant if not managed carefully. The *Balloon Model* is a method to represent a Residue-node in a single CAN. The Balloon

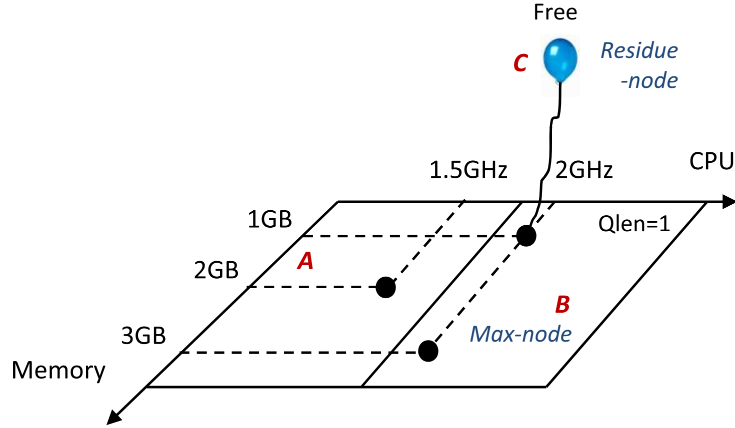


Figure 3.3: Balloon model: Node B is assigned the same job as in Figure 3.2

Model has only one CAN, which contains both single-core nodes and the Max-nodes for multi-core nodes. This is equivalent to the Primary CAN in the Dual-CAN scheme. Residue-nodes are then attached to the zone that contains the Residue-node's coordinates. This is not a separate node in the CAN. Rather, a given Residue-node associates itself with the physical CAN node that owns the zone that contains the Residue-node's coordinates. We call the Residue-node a *Balloon*, because it is attached to a physical node in the CAN space like a balloon is attached with a string to a child's wrist. A Residue-node is therefore connected to only one zone in the CAN. Therefore, creating and removing Residue-nodes is straightforward and affects only one CAN node. If a Residue-node is assigned a job to run, the Residue-node detaches itself from its current node and migrates (routes in the CAN) to a new node based on its new resource availability. This is analogous to cutting a balloon's string so it can fly off and land somewhere else. Figure 3.3 shows an example for the Balloon model, where the status of the nodes is the same as in the Dual-CAN example in Section 3.1.2.

3.1.4 Matchmaking for Multi-core Nodes and Multi-threaded Jobs

We describe new matchmaking algorithms for the Dual-CAN and Balloon models. Although we have previously developed matchmaking mechanisms that work well for single-core nodes [25], those mechanisms cannot take advantage of all the resources available in multi-core nodes. The first approach for a multi-core environment is that we add a dimension to the CAN that represents the number of cores in a node. This dimension is used to both advertise the number of cores available in a multi-core node in the CAN, and also to specify the number of cores requested for a multi-threaded job.

Information Aggregation The original CAN propagates aggregated load information along each CAN dimension, such as the number of nodes and average job queue lengths, to aid in load balancing for the matchmaking process. However, in a multi-core environment we must change the way to measure the load (= job queue length divided by number of nodes), because this measure assumes a single-core machine [25]. For multi-core nodes, we generalize queue length to measure the sum of the cores required for all jobs in a node's queue and use the total number of cores instead of the number of nodes. On a single-core machine, these generalizations retain their original meaning.

In the Dual-CAN scheme, information aggregation is performed only in the Primary CAN, because only free nodes (those with empty job queues) can exist in the Secondary CAN. Thus, information aggregation is not meaningful in the Secondary CAN. On the other hand, in the Balloon Model, the number of cores

includes the number of cores in the node as well as the number of cores in all Balloons attached to the node. However, the number of cores in a Balloon should be discounted somewhat, because contention for shared node resources can slow down jobs that are assigned to cores represented by Balloons. Section 3.2 shows in more detail how we model contention for shared resources among the cores in a node. The total number of cores in node N (denoted by $TNC(N)$) can be measured by the following equation:

$$TNC(N) = NC(N) + \sum_{b \in \mathbf{B}} \beta_b \cdot NC(b) \tag{3.1}$$

In Equation 3.1, $NC(N)$ is the number of cores in the node or Balloon N, and B is the set of all Balloons in Node N. In addition, β_b is the discount factor for Balloon b, which will be discussed in the next Section.

Pushing jobs for load balancing Job pushing starts once a job has been routed to a CAN node that meets the job's minimum resource requirements. To minimize job queue wait time, we set the priority for job assignment so that free nodes (with empty job queues) have higher priority than nodes with both free and used cores (so are already running one or more jobs), which then have higher priority than nodes with no free cores (fully busy nodes). Therefore, the job pushing procedure for the multi-core CAN is divided into three steps. First, try to find a free node in the CAN neighbors of the current node. If one or more free nodes is found, assign the job to the node having the fastest CPU(s). Otherwise, the target node and dimension

with minimum objective function is chosen.

$$F_d(u) = \frac{AI_d(u).SumOfRequiredCores}{(AI_d(u).NumberOfCores)^2} \tag{3.2}$$

In Equation 3.2, $F_d(u)$ is the objective function for the upper neighbor node (= the neighbor nodes that are farther from the origin) u in dimension d, and $AI_d(u)$ is aggregated load information for node u. That value is equal to the average core utilization divided by the number of cores. Given the target dimension, we can decide whether to stop pushing or not according to the following formula:

$$P(N) = \frac{1}{(1 + AI_{TD}(N).NumberOfNodes)^{SF}} \tag{3.3}$$

In Equation 3.3, $P(N)$ is the probability to stop at Node N, and SF is the stopping factor, which is the parameter to adjust the stopping probability [25]. In addition, $AI_{TD}(N)$ is the aggregated information at Node N for the target dimension TD. If the job stops probabilistically, a search is initiated for a node with enough free cores for the job, either in the Secondary CAN for the Dual-CAN scheme or in the Balloons of the node and its CAN neighbors for the Balloon Model, and if found the job is assigned to the fastest node of those free nodes. If a node with enough free cores is not found, the job is assigned to the best run node among the capable nodes based on a multi-core node score function:

$$F(C) = \frac{C.RequiredCores/C.NumberOfCores}{C.SpeedOfCPU} \tag{3.4}$$

Algorithm 1 Pushing for Dual-CAN

```
1: Choose target node and dimension with minimum objective function(Equation 3.2).
2: Determine stopping based on the stopping probability(Equation 3.3) for the target dimension
3: if Stop then
4:    Start pushing on the Secondary CAN and find the best candidate.
5:    Select the best candidate with minimum score (Equation 3.4) among neighbors on the Primary CAN.
6:    if A candidate in the Secondary CAN exists then
7:       Pick the run node from the Secondary CAN.
8:    else
9:       Pick the run node from the Primary CAN.
10:   end if
11: else
12:   Push the job to the target node.
13: end if
```

$F(C)$ is the score function for node C, which is computed as its core utilization divided by its CPU speed. The node with minimum score is chosen as the run node among all the candidate nodes, encoding a preference for more lightly utilized nodes with faster CPUs. Algorithm 1 outlines the job pushing procedure for the Dual-CAN. For the Balloon Model, when the job stops probabilistically at node C, we search only the CAN neighbors and their Balloons from node C and select the best node among them. In this case, a Balloon will be preferred, because a Balloon will start running the job immediately since it has adequate available resources.

3.1.5 Model Comparison

One of the main differences between the two models is the overheads imposed. In the Dual-CAN model, additional overhead consists of Residue-node *join* messages, Residue-node *leave* messages, Secondary CAN maintenance messages, and

job-pushing in the Secondary CAN. This additional cost incurred by the Secondary CAN is proportional to the total number of nodes in the CAN - the total cost to maintain the Secondary CAN is less than that of the Primary CAN because the Secondary CAN has fewer nodes than the Primary CAN. On the other hand, the Balloon model has much lower overhead than the Dual-CAN. For example, the cost to insert a new Balloon is the cost of CAN routing to the coordinate in the CAN, and the cost to remove a Balloon is a single message. On the contrary, the cost for *join* and *leave* operations for a Residue-node in the Secondary CAN is a zone split or take-over as in the conventional CAN. A more significant difference is CAN maintenance costs. For the Balloon Model, that cost is just a periodic one-way heartbeat message per Balloon to the node it is attached to. In the Dual-CAN scheme, by comparison, Residue-nodes in the Secondary CAN exchange heartbeat messages periodically with CAN neighbors.

The matchmaking cost for the two models is also not the same. When probabilistic stopping occurs, searching for a Residue-node in the Dual-CAN is a global operation, because the Dual-CAN algorithm can traverse the entire Secondary CAN in the worst case. However, the Balloon Model searches for appropriate Balloons only among nodes that are one or two routing hops away from where stopping happens, so the search area is limited(local operation). Therefore there is a trade-off between performance and cost between the two models.

3.2 Parameterized Prediction Model for Resource Contention

3.2.1 Contention Penalty in Multi-core Nodes

One of the unresolved issues in multi-core computing is the performance effects from having two or more jobs running simultaneously, on different cores, in the same node. If two jobs frequently access a shared resource, such as cache or memory, or have large memory bandwidth requirements, one job may have to wait while another job is accessing memory. This contention for memory or other shared resources increases job running time compared to running a single job on a node. However, instead of trying to build a theoretical model for the expected job running time taking contention for shared resources into account, we employ an empirical model. The most important observation is that if two (or more) jobs running on the same node are both memory- or I/O-intensive, they will likely have longer running times than when run in isolation.

To measure worst case contention effects, we use the STREAM benchmark [29] to run multiple memory-intensive jobs on a multi-core machine. STREAM was originally developed to measure node memory bandwidth, so it runs highly memory-intensive jobs and measures average memory bandwidth and job running time. We conducted an experiment on a dual 6-core processors Linux machine running a version 2.6.18 kernel, with two AMD Opteron 6168 CPUs running at 1.9GHz with 32GB memory. We used the Linux *taskset* command to enable running a job on a specific core. First, we run a single STREAM job on one core of the multi-core machine and leave other cores idle. Second, we run multiple STREAM jobs at the

same time, one on each core, to measure memory bandwidth and overall elapsed time. Table 3.1 shows the normalized running time on a multi-core machine. The normalized running time is the running time for multiple jobs (with contention) divided by the running time for a single job (no contention). We changed the number of jobs from 2 to 6 in a single multi-core CPU to see the behavior of the slow down under contention effect. In addition, we launched 12 jobs on the two 6-core CPUs (6 jobs in each CPU) to see the worst case in a single machine. The results show that multiple extremely memory-intensive jobs on the multi-core node increase job running time by a linear factor in the number of concurrent jobs in the worst case. For example, a STREAM *Copy* operation takes 35% longer when 2 copies are run, one on each core, and 73% longer when 4 copies are run. On average, the running time for the STREAM benchmarks running on multi-core cores is approximately $(1 + n \times 0.175)$ times longer than for running only on a single core, where n is the number of concurrent jobs and less than or equal to the number of cores. Note that this is a worst case scenario for contention (in this case to shared memory), so cannot be directly generalized to a more diverse workload. A similar finding is presented in a paper by Weinberg and Snavely [30]. They measured the slowdowns of memory intensive benchmark tests varying the number of concurrent jobs in the machine, and showed that the slowdown increases linearly according to the number of concurrent jobs.

	2	4	6	12
Copy	1.3514	1.7318	1.9507	3.5064
Scale	1.3336	1.7120	1.9029	3.3285
Add	1.3547	1.7235	1.9096	3.5478
Triad	1.3665	1.7440	1.9417	3.5628
Average	1.3515	1.7278	1.9262	3.4864

Table 3.1: STREAM result: normalized running time (slow down) on a multi-core machine

3.2.2 Experimental Results

Scientific applications that are the usual target for desktop grid computing are often either CPU intensive or have mixed job requirements, with both large CPU and memory requirements. One example of prior work on performance evaluation for multi-core machines, by Alam et al. [31], performed scientific workload experiments with multi-core processors. The authors ran common scientific benchmark test programs, such as the NAS Parallel Benchmarks, the AMBER and LAMMPS molecular dynamics simulators, and the POP (Parallel Ocean Program) climate modeler in various environments using MPI (Message Passing Interface). From their experiments in the case of a single MPI task running on a dual core node and two MPI tasks running simultaneously on a dual core node, running time for two tasks is higher by 3.8% to 27% (with a mean:10.97%) than for a single task.

We have run additional experiments with the SPEC CPU2006 integer benchmark suite [32] to measure contention effects in a multi-core node. The same machine (with two 6-core CPUs) used for the STREAM benchmark is used for this experiment. We compared the running times for the entire suite for multiple cases, starting with one copy of the job on the multi-core node and increasing to 2,4,6 and

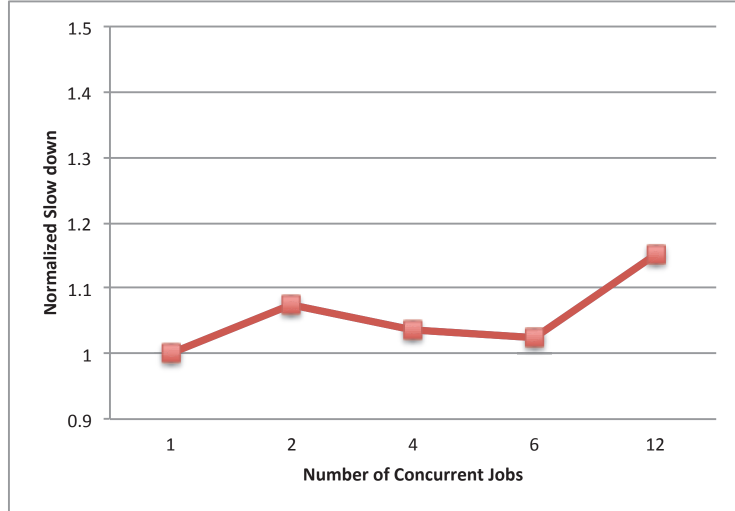


Figure 3.4: Normalized Running Time vs. Number of Concurrent Jobs

12 copies on the node. We limit the number of concurrent jobs to the number of cores in the node in this experiment. Figure 3.4 shows that the normalized running time increases somewhat as the number of concurrent jobs increases. It shows that running multiple computing-intensive jobs in a multi-core machine takes a little bit longer (with a range of 2.5% to 15%), but it is not changed a lot by the number of concurrent jobs in the system. Similarly, the experimental result in the paper by Weinberg and Snavely [30] shows that a small increase (approximately 2%) is added to the running time for multiple concurrent computing intensive jobs scenario.

Combining the results of other researchers work and our experiments show that the time penalty for running two tasks on a multi-core node due to contention can be modeled as a constant (about 10% on average). The SPEC scientific workload is extremely CPU intensive, but some performance penalty still occurs.

3.2.3 New Prediction Model

From the experimental results in the previous section, we find that most scientific jobs using multiple cores in a multi-core node run a constant factor longer than when using only a single core, and that a memory intensive job contending with other jobs on a multi-core node may incur a penalty of up to a constant multiplied by number of jobs compared to running alone, in the worst case [1]. Therefore, we have built a mathematical formulation for our multi-core performance prediction based on these results. First, if a job is not extremely memory intensive, the expected job running time increases by a constant $p\%$, where p is determined from the earlier experimental result as approximately 10%. On the other hand, if a job is memory intensive, we model a heavier contention penalty. If a job has large memory requirement, it is very likely to be memory intensive. In the extreme case, if the sum of the memory requirements for all jobs running on the node is close to the physical memory size of the node, it is likely that there will be frequent contention for access to memory, and frequent contention leads to longer job running times. Therefore, in the worst case, the job running takes longer by $(1 + n \times SDR)$ times where n is the number of concurrent jobs and SDR (Slow Down Ratio) is a constant which is equivalent to the maximum additional running time due to contention effects.

Suppose that there is a multi-core machine that has shared resources R_i for $i = 1, 2, 3, \ldots, k$, where k is the number of shared resources. Also assume there

[1] Some CPUs use hyper-threaded architectures [33, 34, 35] to maximize throughput of multi-threaded applications. A hyper-threaded architecture can also have contention effects on the internal shared caches in the CPU. However, our matchmaking model assumes that each thread in the user's job is run on a different physical core so we do not take hyper-threading techniques into account in this contention prediction model.

are $n-1$ jobs already running on the node. If the node is assigned a new job and runs it, the total amount of the i-th shared resource that is used becomes C_i. We compute *the expected running time ratio* of the multi-core over the single-core base case, called α. If the new job is not shared-resource intensive, $\alpha = 1 + p$ where p is the slow-down penalty factor for a general application. If the new job is shared-resource intensive, such that C_i is large, $\alpha = max_i\{1 + n \cdot SDR \cdot (\frac{C_i}{R_i})^m\}$ where m is a factor to adjust the steepness of the curve in the worst case performance region (see Figure 3.5). If m is large, the worst case performance region would be small but steep. If m is small, the extreme area is more broad, but not steep. Considering CPU intensive and shared-resource intensive cases, α is defined in Equation 3.5.

$$\alpha = max[1 + p, max_i\{1 + n \cdot SDR \cdot (\frac{C_i}{R_i})^m\}] \tag{3.5}$$

The *total penalty* Ω, the increased running time for all jobs because of running the new job, is shown in Equation 3.6.

$$\Omega = n \cdot max[p, max_i\{n \cdot SDR \cdot (\frac{C_i}{R_i})^m\}] \tag{3.6}$$

The discount factor β is equal to $1/(1{+}\Omega)$. It is used to discount the Residue-node in the Balloon Model so that a Balloon has lower priority than a normal node for job assignment. Figure 3.5 shows the expected running time ratio α with respect to C_i. If C_i becomes close to R_i, the running time will increase drastically, up to $1+n \times SDR$ times that of running a job all by itself using a single core of the multi-core node. Though this prediction model is based on the experimental results, a

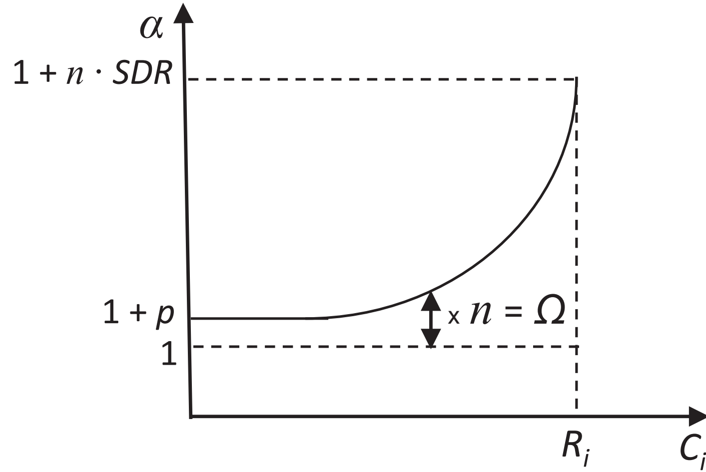


Figure 3.5: Expected running time ratio α with respect to shared resource usage

new model can predict more accurate result. However, our matchmaking framework is flexible enough to accommodate any new models if the new model can give more accurate result.

3.3 Experiments

3.3.1 Experimental Setup

To experiment with a P2P desktop environment efficiently, our simulation uses synthetic workload and resource events. We generated a sequence of events that are composed of node joins, node departures (both voluntary and from failure), and job submissions. Events are generated with the intervals between events having a Poisson distribution with arrival rate τ.

Nodes and Jobs Each node and job is assigned a resource capability or requirement, including CPU speed, memory space, disk space, and the number of

cores/processors We generated a node profile where a high percentage of nodes have relatively low resource capability and a low percentage of nodes have high resource capability. In addition, our simulations use both *clustered* and *mixed* workloads and node capabilities to support various scenarios. A clustered scenario means that a small number of distinct sets of computing nodes or jobs exist in the Grid. Within each set, all nodes or job capabilities are equivalent, but nodes or jobs differ between sets. Mixed workloads have resource capabilities randomly selected so the node or job makeup of the simulation is heterogeneous. Thus, we can have clustered or mixed nodes as well as clustered or mixed jobs.

Each job has an expected running time that has average time T and is randomly selected and uniformly distributed between $0.5T$ and $1.5T$. For these simulations, we set T to 3600 seconds (1 hour). However, our matchmaking algorithm is not aware of a job's expected running time while most parallel job scheduling algorithms assume that expected job execution time or wall time is known in advance, and utilize job execution time information in their job scheduling algorithm. In addition, if a node running a job has CPU speed faster than the CPU speed requested for the job, job running time is then shorter than the expected running time. We adjust the modeled job running time to take into account the speedup obtained by running on a faster than required node. In addition to this speed-up for CPU speed, we compute the job completion time by multiplying the expected running time by the variable α computed as described in Section 3.2, to emulate contention in a multi-core node. Finally, the communication delay between nodes for each message sent between nodes is generated from an exponential distribution

with an average of 50 milliseconds. For our multi-core model, we used 0.1 for the multi-core slow-down penalty p, and set the factor for the curve shape (m) to 4. We ran many simulations varying these parameters, and the results were always similar to those shown below.

An important feature of our simulation study is that we focus on *steady-state* performance, where the system job arrival and completion rates are approximately the same during the simulation period measured. In our simulation, in the steady state there are an average of 1000 nodes in the system, and we measure the matchmaking performance for submitting 5000 jobs with various job inter-arrival periods that depend on the workload, including whether they are clustered or mixed, and heavily or lightly constrained.

Comparison Models For comparison purposes, we also test a greedy centralized matchmaking scheme that would be very expensive to implement for a real decentralized Grid system, but gives some indication of the best performance possible for an online matchmaking algorithm. The **centralized matchmaker** exploits the current state in all the Grid nodes to assign jobs based on up-to-date global information. However, the centralized matchmaker is still run online for a fair comparison to the online decentralized algorithms. The centralized matchmaker is used only to measure load balancing performance, and does not incur any cost to collect state information from the nodes or to match jobs to the nodes. It uses a greedy job allocation policy that selects the fastest CPU with minimum load among nodes that meet the job's resource requirements. However, for optimized performance, we add

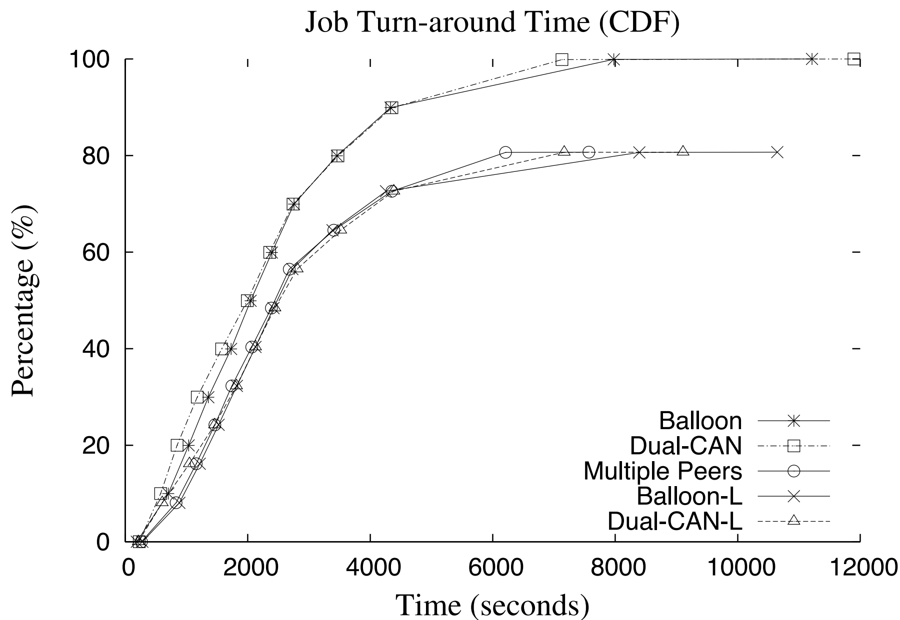


Figure 3.6: Cumulative distributions for Job Turn-around time

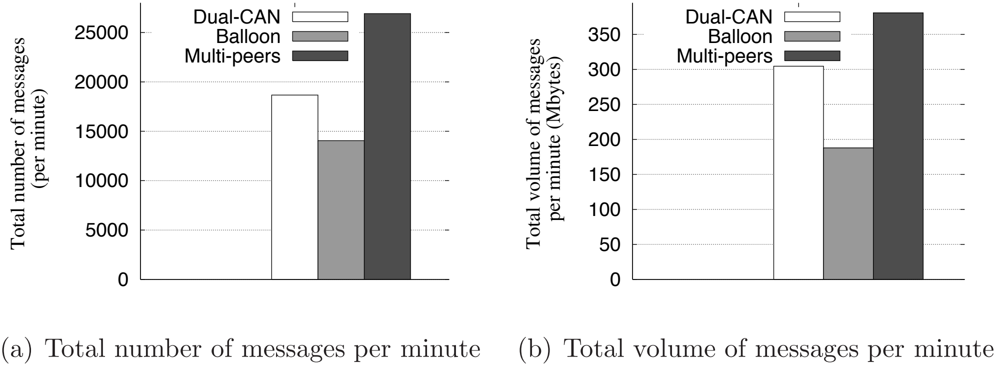
a policy on over-provisioning to our centralized matchmaker. We will discuss the relationship between performance and over-provisioning in the last paragraph of the Section 3.3.2.

A second simulation model we compare against deploys **Multiple Peers**(MP) on a single node, each responsible for one CPU/core and an equal fraction of the other node resources. MP is the core scheme of Condor's current strategy for multi-core nodes[28]. We run one peer per core and statically and equally partition all other shared resources. Note that MP cannot accommodate some jobs that have large resource requirements (even though a whole node may be able to do so), nor can MP accommodate multi-threaded jobs that require multiple cores.

3.3.2 Experimental Results

Completeness The first experiment compares the Dual-CAN and Balloon models against the Multiple Peers (MP) scheme. Figure 3.6 shows the cumulative distribution of job turn-around time for the three systems. Job turn-around time is defined as the time from when the job is injected into the system to when the job finishes running on the node to which it was assigned. We submit only single-threaded jobs to the system, to fit with the limited capability of MP. Each node has 1, 2, 4, or 8 cores. The total number of cores in the system with 1000 nodes is 1838, because the nodes with fewer cores are more frequent in the node model used. As is seen in Figure 3.6, Dual-CAN and Balloon can run all the jobs, but MP can run only about 80% of the jobs, because a job requiring a large amount of memory or disk space may fail to find a node capable of running the job because the multi-core node's resources are statically partitioned. In Figure 3.6, we do not include such failed jobs for MP. However, this comparison is somewhat unfair to the Dual-CAN and Balloon algorithms and to MP, because MP runs fewer jobs and the unmatched jobs tend to have high resource requirements, so the overall system load for MP is much lower than for the other schemes. To compensate for system load differences, we also show other results for Dual-CAN and Balloon (called Dual-CAN-L and Balloon-L, respectively), which show only the jobs that are capable of being run with MP. In Figure 3.6, Balloon-L and Dual-CAN-L show competitive performance to MP, when measuring job turn-around time.

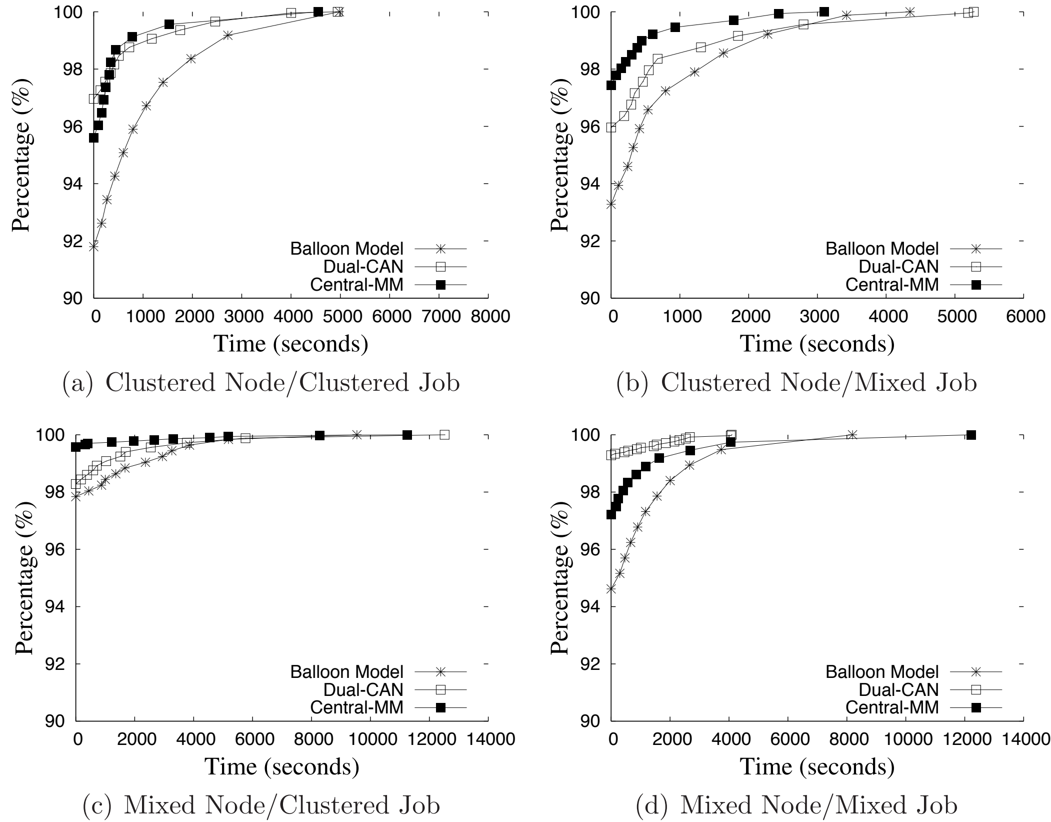
Figure 3.7 shows the total overhead from messaging across the entire system.



(a) Total number of messages per minute (b) Total volume of messages per minute

Figure 3.7: Costs of Dual-CAN, Balloon and MP

The cost metrics are the number of messages and the total volume of messages. In Figure 3.7, the overall cost for MP is significantly higher than for the other schemes. For example, the total number of messages for MP is 90% larger than for Balloon. The reason for the higher cost for MP is related to the number of peers in the system. The largest portion of the total overhead is CAN maintenance messages that neighbors exchange periodically. The total number of heartbeat messages is proportional to the number of peers in the system, so MP sends more messages because it has more peers (one per core, instead of one per node). As we discussed in Section 3.1.5, the overhead for Dual-CAN is higher than for Balloon (about 30% more messages and 60% total message volume). The results imply that the Balloon model has a cost advantage compared to Dual-CAN. In this experiment, MP cannot accommodate a job with large resource requirements, even though there exists a capable node in the system (lack of *Completeness*). Furthermore, MP has significantly higher overhead compared to our two models (lack of *Low-overhead*). Because of its limitations and its high cost, we do not further compare against MP



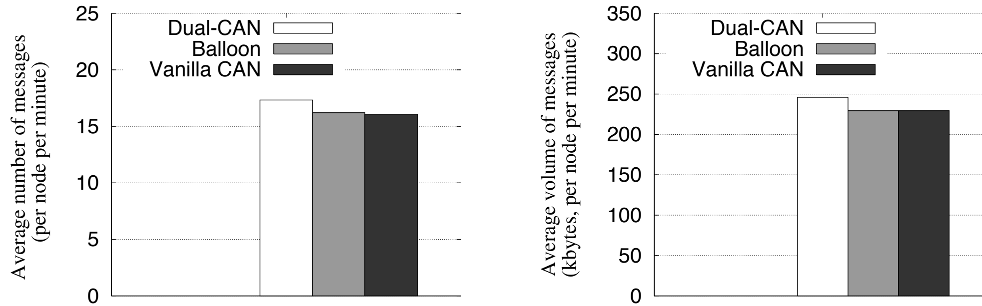
(a) Clustered Node/Clustered Job (b) Clustered Node/Mixed Job

(c) Mixed Node/Clustered Job (d) Mixed Node/Mixed Job

Figure 3.8: Cumulative Distributions of the Job Waiting Time

in the rest of experiments.

Load Balancing Figure 3.8 shows load balancing performance for our two schemes, comparing against the centralized matchmaker. We experiment with different scenarios - clustered or mixed nodes, clustered or mixed jobs, and lightly or heavily constrained jobs. In this experiment both single- and multi-threaded jobs are submitted. Figure 3.8 shows results for four different scenarios. Note that the starting point for the y-axis in the graphs is 90%, not 0, to better illustrate the differences between the matchmaking schemes. Overall, the performance of the three schemes is not much different in measuring job waiting time. These results show that our two algorithms show very competitive performance for load balancing, even compared



(a) Number of messages per node per minute (b) Volume of messages per node per minute

Figure 3.9: Costs for Dual-CAN, Balloon and Vanilla CAN

to a centralized matchmaker. Comparing our two algorithms, Dual-CAN achieves better performance than Balloon, as we discussed in Section 3.1.5

On the other hand, the overhead for the Dual-CAN is noticeably larger than for Balloon, as seen in Figure 3.9. The left graph in Figure 3.9 shows the average number of messages per node per minute, and the right graph shows the average volume of messages per node per minute. The cost for Balloon is very competitive with the cost of the vanilla single CAN. However, for example, the additional cost in message volume for the Dual-CAN is about 7%. There is therefore a trade-off in the cost and performance between the two schemes. Overall, Figure 3.8 and Figure 3.9 show that our two decentralized schemes both balance job load well, and do not add significant overhead. Because the clustered node/clustered job scenario is a common but most difficult for our matchmaking algorithm [19, 25], we will show the result of clustered scenarios in later sections.

Performance vs. System Load To see the performance under different system load, we conducted experiments varying the job inter-arrival time from 2 seconds to

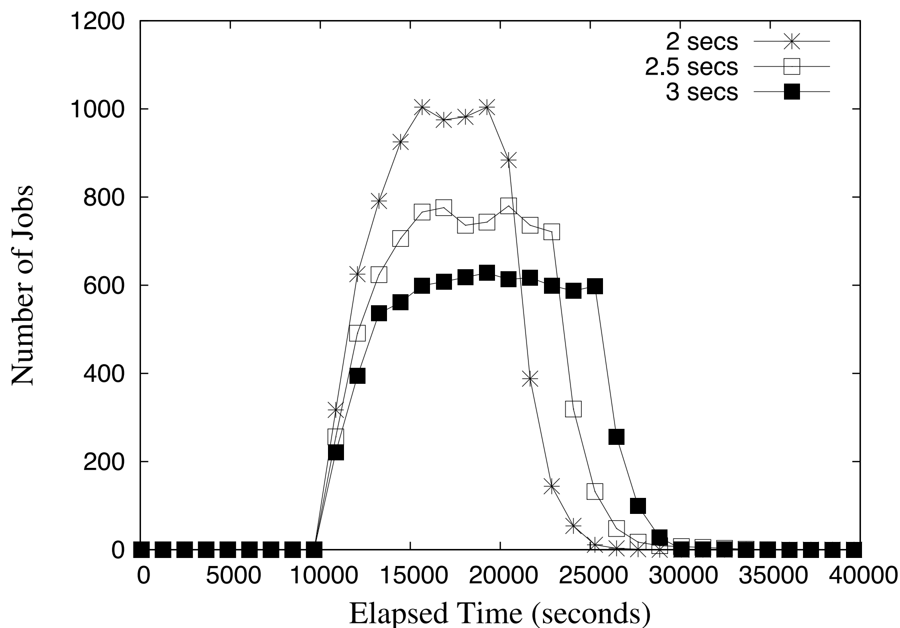
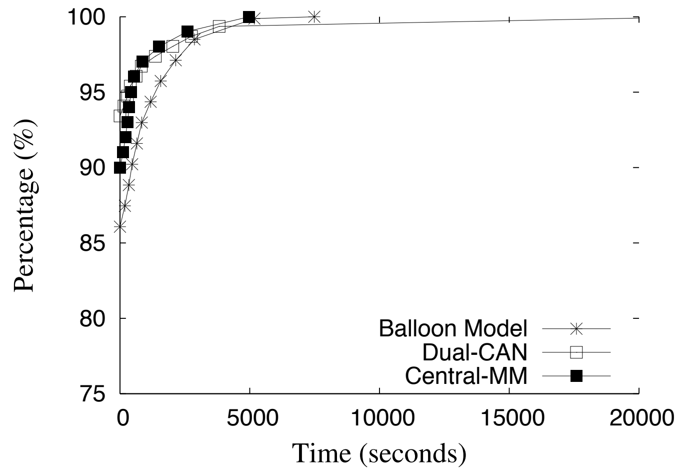


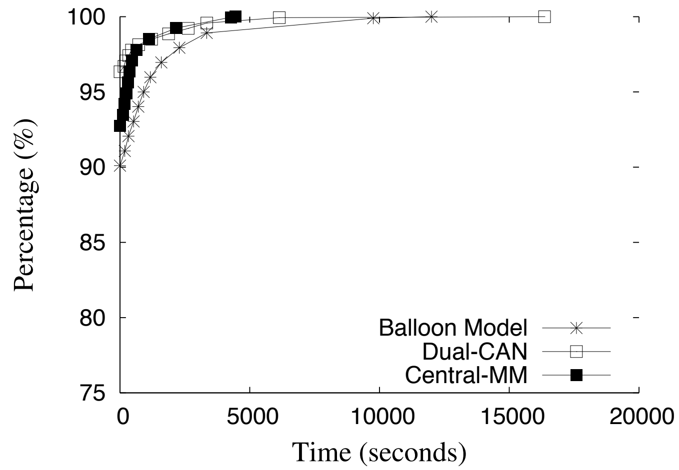
Figure 3.10: Snapshots of number of jobs in the system varying job inter-arrival time

3 seconds. As we discussed, we focus on the steady-state performance characteristics so changing job inter-arrival time should change system loads effectively. Figure 3.10 shows snapshots of the number of existing (running or waiting) jobs in the total system over the entire simulation. Note that the period from the start to 10000 seconds is the bootstrap time, so all node join the system but no jobs are inserted until 10000 seconds. In this Figure, the short (2 seconds) inter-arrival time can increase system load effectively in the steady-state.

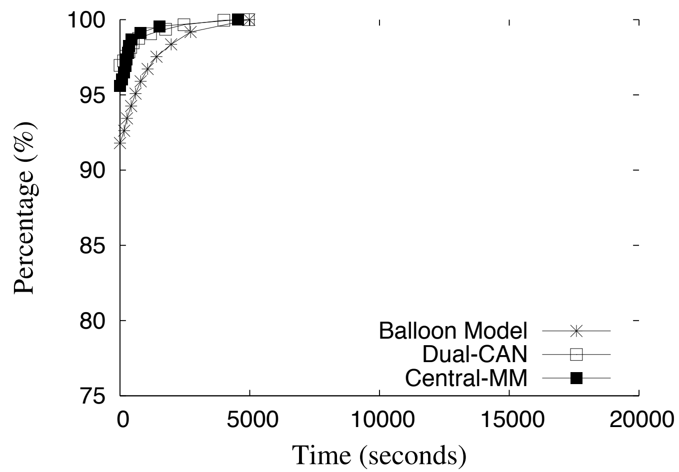
Figure 3.11 presents the job wait time distributions varying the job inter-arrival time. Though the average job wait time as well as the number of non-zero waiting time jobs increases when the inter-arrival time is short (2 seconds), the overall matchmaking performance of Dual-CAN or Balloon algorithm is competitive with the centralized matchmaker regardless of the system load. In addition to the ab-



(a) 2 seconds

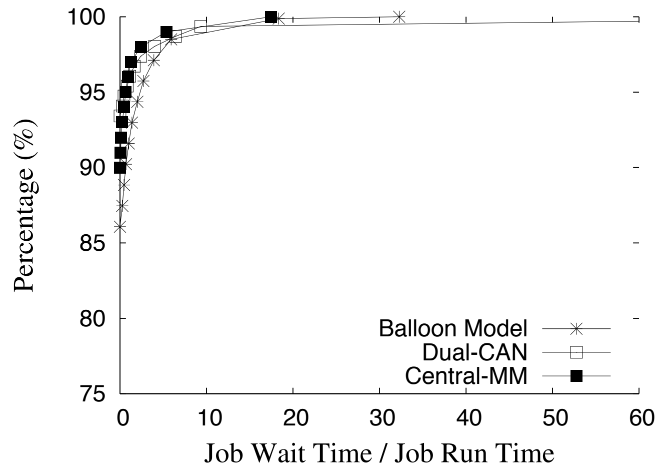


(b) 2.5 seconds

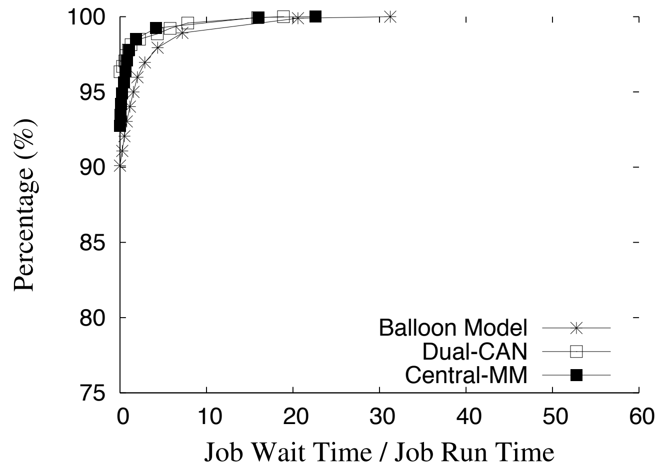


(c) 3 seconds

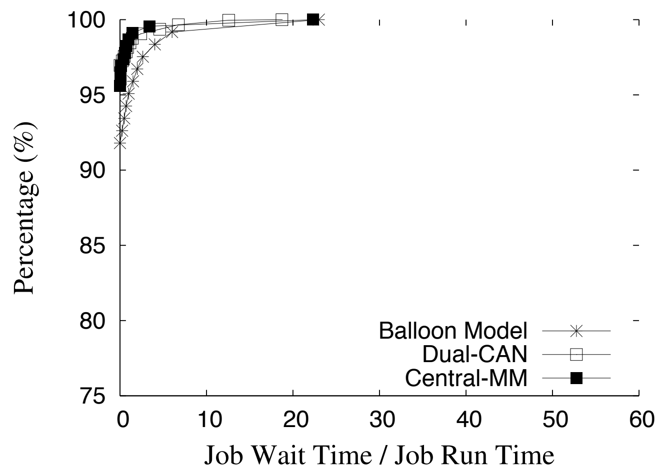
Figure 3.11: CDF of Job wait time varying Inter-arrival Time



(a) 2 seconds



(b) 2.5 seconds



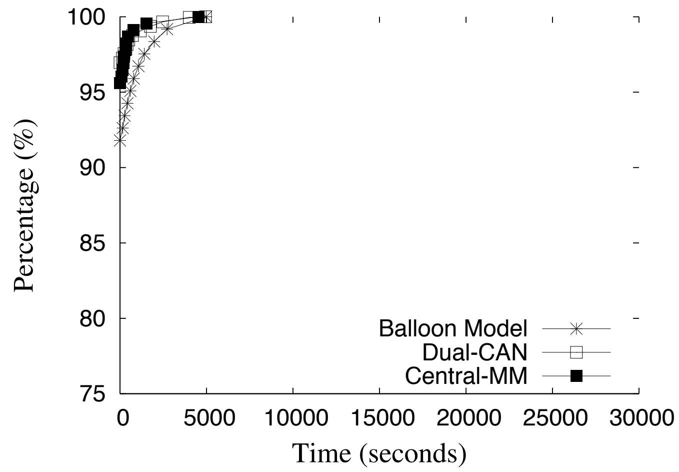
(c) 3 seconds

Figure 3.12: CDF of Normalized Job wait time varying Inter-arrival Time

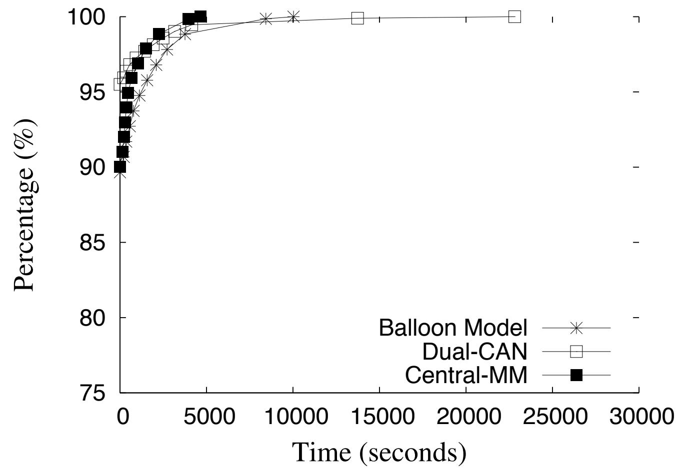
solute job wait time distribution, we plot the relative job wait time distribution in Figure 3.12. The relative (normalized) job wait time is defined as the absolute value of job running time divided by the job execution time. As our matchmaking algorithm is oblivious to the job execution time, the trend of the result in Figure 3.12 is very similar to that of Figure 3.11. Through Figure 3.11 and Figure 3.12, we conclude that load balancing performance of our matchmaking algorithm is competitive to the on-line centralized matchmaker.

Performance vs. Job Constraint Ratio A job's requirements for the continuous resource types may be omitted (meaning any amount of that resource is acceptable). We define the *job constraint ratio* as the probability that each continuous resource type for a job is specified for a given input stream of jobs. A higher job constraint ratio (called heavily constrained jobs) makes matchmaking more difficult, as highly-specified jobs are more difficult to match to nodes since fewer nodes will meet the specification. We conducted our experiments varying job constraint ratio from 30% to 70% to see how the performance changes with the job constraint ratio.

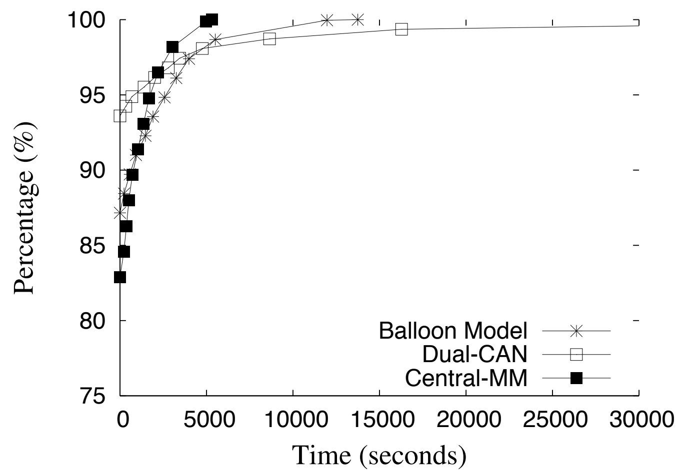
Figure 3.13 shows the cumulative distribution of the absolute job wait times, and Figure 3.14 presents the normalized job wait time distribution (i.e. the absolute job wait time divided by job running time). Both figures show that our two approaches matchmaking performance is competitive to the centralized matchmaker, though a few jobs with Dual-CAN scheme have relatively long wait times when the job constraint ratio is high (70%). When the matchmaking decision becomes difficult with higher job constraint ratio, some jobs can be assigned to nodes



(a) 30%

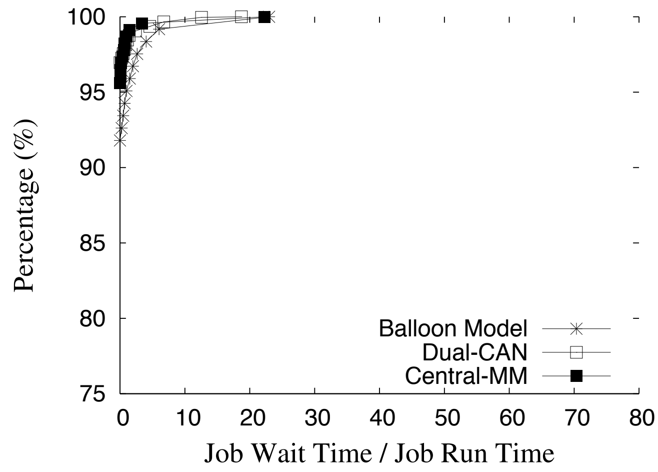


(b) 50%

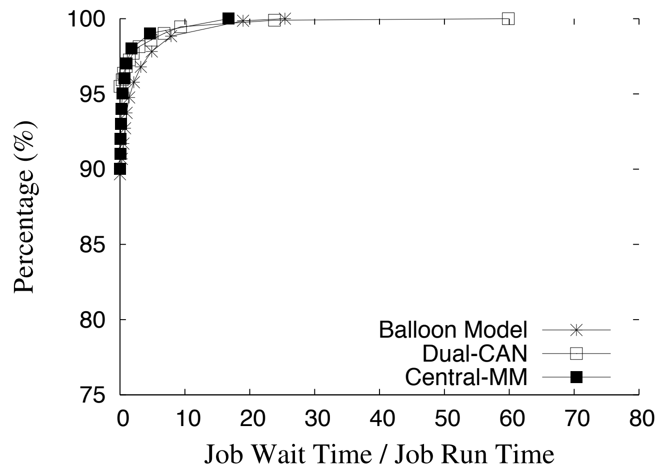


(c) 70%

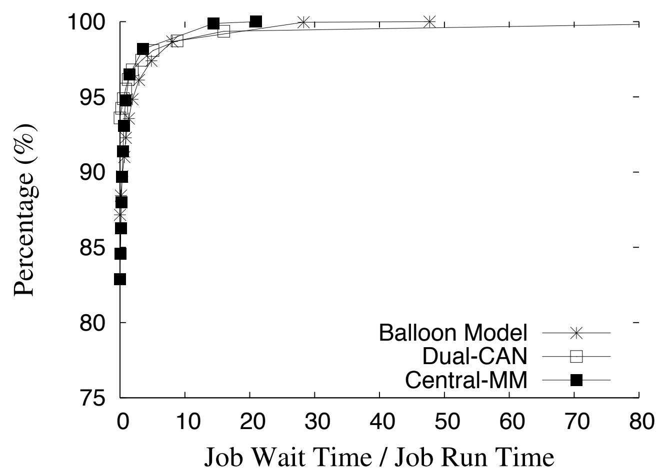
Figure 3.13: CDF of Job wait time varying Job Constraint Ratio



(a) 30%



(b) 50%



(c) 70%

Figure 3.14: CDF of Normalized Job wait time varying Job Constraint Ratio

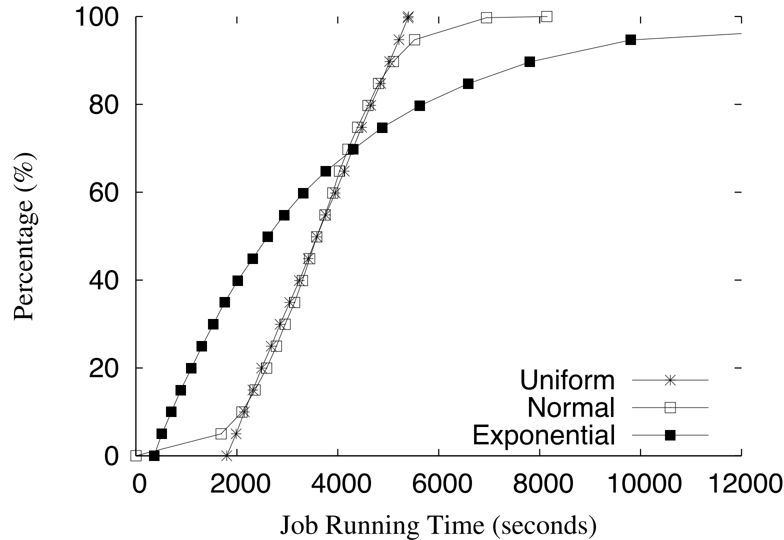
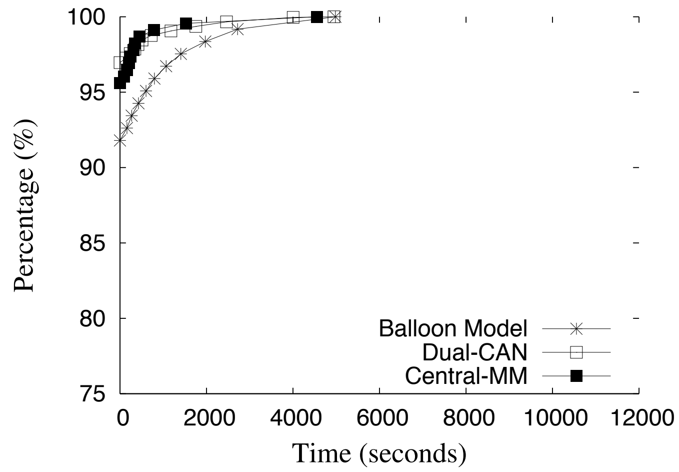


Figure 3.15: Snapshots of number of jobs in the system varying job inter-arrival time

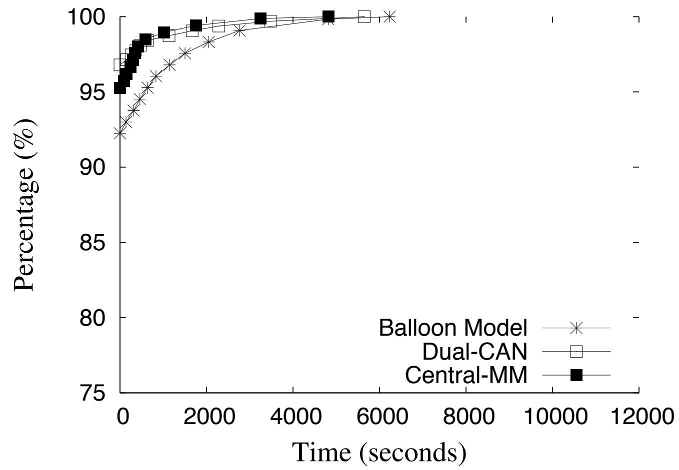
with higher load because our peer-to-peer based matchmaking algorithm relies on local search. In this case, dynamic job scheduling based on job migration among neighbor nodes can mitigate this problem [36, 37, 38].

Job Running Time Distribution So far we used workload with uniformly distributed job running time. In this simulation, we vary the job running time distribution. We use both normal distribution and an exponential distribution for job running time. Both distributions have the same average value (1 hour) as the uniform distribution. Figure 3.15 shows the job running time distributions for each case. We compare load balancing performance of three different workloads.

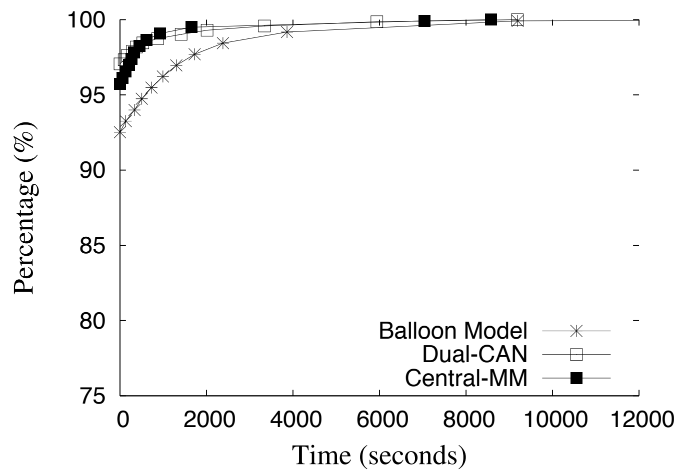
Figure 3.16 shows cumulative distribution of the absolute job wait times for three job running time distributions. Overall, the performance of Dual-CAN and Balloon model is competitive to that of the centralized matchmaker, regardless of the job running time distribution. The interesting finding is that the exponential



(a) Uniform Distribution



(b) Normal Distribution



(c) Exponential Distribution

Figure 3.16: CDF of Job wait time under different job running time distribution

distribution shows a longer tail in the graph (a few jobs show long wait times) than the others. Since variation of job running time for the exponential distribution is larger, some jobs show longer wait times even though they must only wait for a single job to complete. Dynamic load balancing techniques [36, 37, 38] can effectively handle this situation because they can modify job allocation using the dynamic load situation. In conclusion, this experiment shows that our two matchmaking algorithms show competitive performance to the centralized approach with different workload characteristics.

Over-provisioning effects in the Centralized Matchmaker Our centralized matchmaker is an on-line algorithm to maximize CPU utilization based on global load information in the system. Since the centralized matchmaker has more information compared to our peer-to-peer approach, it may have more candidate run nodes for each given job during the matchmaking process. However, selecting the most capable node as a run-node cannot guarantee optimal performance because this over-provisioning can increase the wait time of future incoming jobs that have high resource requirements. Therefore, some restrictions to avoid over-provisioning can be added to the centralized matchmaker to get better performance.

Figure 3.17 shows the load balancing performance of the centralized matchmaker, limiting the resource provisioning ratio. In this figure, "Central-MM" denotes the basic approach without any restrictions on over-provisioning, whereas "Central-MM-1.1" means that over-provisioning is limited to 10%. Therefore, a small ratio means less over-provisioning allowed. The result in Figure 3.17 shows

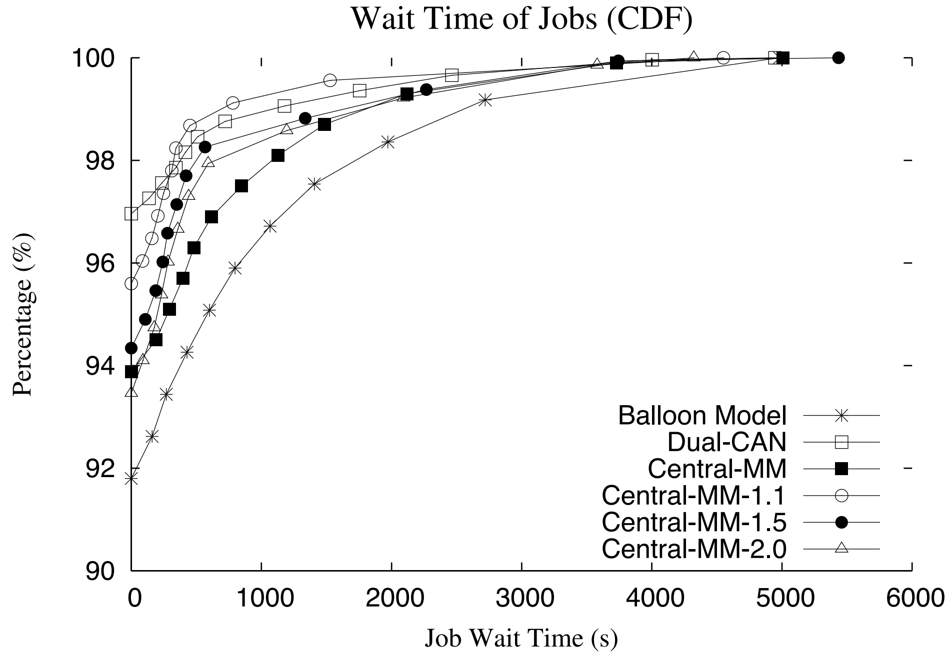


Figure 3.17: Over-provisioning effect in Centralized Matchmaker

that when the centralized matchmaker has some restrictions on over-provisioning, the load balancing performance is better than for the basic centralized matchmaker. In addition, tighter restriction on over-provisioning can improve the total system performance in this experiment. We use the most restricted centralized matchmaker (with ratio of 1.1) for the experiments in this Chapter so that we can get some hints about the maximum performance bound for our peer-to-peer approach.

3.4 Summary

In this chapter, we describe a new decentralized resource management framework for exploiting multi-core nodes in a P2P grid system. The key innovation is to use distinct logical nodes to represent the static and dynamic aspects of node utilization. We have developed two resource management schemes, the Dual-CAN

and Balloon models, and present an efficient matchmaking scheme. In addition, we present a new analytic running time model for concurrent jobs in multi-core environments.

Our experimental results show that both models perform comparably with a centralized matchmaker. Dual-CAN is able to achieve better matchmaking performance in some environments because it does a better job of exploiting residual capacity in multi-core nodes. However, the Balloon Model adds less overhead; Dual-CAN significantly increases both the number and volume of messages. Both models are more effective than the static Multiple Peers approach at running combinations of jobs with both large and small resource requirements.

However, these techniques are optimized for symmetric multi-core nodes environments, but cannot be straightforwardly extended to more heterogeneous environments where nodes have multiple computing elements of different types. The next chapter shows how to deal with such heterogeneity.

Chapter 4

Supporting Computing Element Heterogeneity

In this chapter, we present our new resource management framework and the techniques that allow us to exploit multiple computing elements with different performance characteristics. We first describe how our CAN-based matchmaking framework can be extended to express various types of heterogeneous resources, and then discuss additional mechanisms to deal with multiple resource types and different performance of computing elements. Second, we show how to make these decentralized scheduling decisions *efficiently*. Directly extending our prior matchmaking framework described in Chapter 2 to handle diverse computing elements can add greatly to the communication costs incurred by the underlying DHT. We describe a set of mechanisms that limit communication cost growth without sacrificing failure resilience, one of the key advantages of P2P systems.

4.1 Resource Management for Heterogeneity

4.1.1 Accommodating Heterogeneous Nodes

We first address the target heterogeneous environment and the system model. A node in a grid can have *multiple* computing elements (CEs), and the node can run multiple, independent multi-threaded jobs concurrently. A *CE* is a physically separated unit within a grid node, and contains a set of cores which are mainly

used for computation, such as a CPU, a GPGPU, or other types of special-purpose computing processors. In addition, the CEs can be of different types, so that their performance characteristics can vary greatly. Each CE can have independent resource capabilities, so expressing the various resource capabilities in a compact way can be challenging.

Now we discuss how to extend our existing CAN-based matchmaking framework for the heterogeneous environment in detail. For symmetric multi-core nodes, we use a 5-dimensional CAN to represent node's resource capabilities; the 5 dimensions are CPU clock speed, memory size, disk space and the number of cores, plus a random virtual dimension to distinguish nodes that are identical in resource capabilities. To advertise heterogeneous nodes in the CAN, we need additional dimensions to specify different CEs and other resources that are dedicated to those CEs. For example, if a machine has two GPUs (different CEs) in addition to a CPU, the additional required dimensions are 2 (for the two new CEs) $\times$ 3 (GPU Clock Speed, GPU Memory, number of GPU cores) = 6, so the total number of CAN dimensions required is 11 [1]. If a grid system has more heterogeneous types of nodes, the CAN will need even more dimensions to manage heterogeneity effectively. However, adding more dimensions to the CAN can incur significant system costs, which may be a potential bottleneck to system scalability. We will discuss those costs vs. the number of dimensions in the next section. However, even after we add more dimensions to the CAN, several other issues must be considered because

[1]As for the CPU resource description, the GPU clock speed can be replaced with other metrics such as FLOPs. In addition, memory bandwidth or latency in the GPU card can also be included in the performance metric for effective matchmaking [39, 40]. Section 4.2 presents how to deal with a large number of resource attributes due to this heterogeneity in the system.

of the *multiplicity* of CEs in the heterogeneous nodes, as described in the following sections.

4.1.2 Job Pushing for a Heterogeneous System

As we described in Section 2.3, *job pushing* is a mechanism to improve load balance by pushing jobs to less loaded regions in the CAN. To balance load across nodes, at each step in the matchmaking process the pushing algorithm first looks for a *free-node* among the neighbors of the current node. If the algorithm finds a free-node, this free-node will be the node to run the job. Otherwise the job will be pushed to the least-loaded region in the CAN until a free-node is found or the job stops probabilistically. We now discuss all steps involved in pushing jobs in heterogeneous environments.

Acceptable node An acceptable node is a node that can start a job's execution without waiting. A heterogeneous node can have multiple CEs, so even if one or more CEs of a node are busy running jobs, other CEs may be idle and ready to begin another job's execution. Therefore, we can use an acceptable node instead of a free-node to run a job. A node can be regarded as an acceptable node or not depending on the node's resource availability and a job's requirements, while a free-node is always a free-node regardless of a job's requirements. Therefore, the first part of the job pushing process for heterogeneous environments should be changed to look for an acceptable node instead of a free-node.

Dedicated vs. Non-dedicated CE A multi-core CPU can run multiple jobs on

separate cores concurrently; in this case, running multiple jobs can cause contention effects, which may degrade each core's performance significantly. We will call this type of CE a *non-dedicated* CE since it can run multiple jobs at the same time and multiple jobs may contend for shared resources in the CE. We previously described a performance prediction model for contention effects on non-dedicated CEs by interpolating experimental results 3. However, current GPUs (e.g., Nvidia Tesla) can run only a single job at a time (the next version of Nvidia GPUs will run multiple simultaneous jobs, but it is not yet available). We call this type of CE a *dedicated* CE. Note that a dedicated CE cannot run multiple jobs simultaneously, but can run a single multi-threaded job.

To see contention effects between CPUs and GPUs, we use SHOC (Scalable Heterogeneous Computing) benchmark suite [41], and measure the running time under various configurations and testsuits. We use 2-CPU/2-GPU machines for the experiment. Generally the jobs to use GPU need a set of a core in a CPU (for threads control) and many-cores in a GPU card (for computation). Because GPUs in the test machine are dedicated CE, we can execute a single task on a GPU card at a time, but we can run multiple jobs on different cores in a CPU. We measured the running times 1) when a single job runs, 2) when two jobs run on different GPUs and on different CPUs, and 3) when two jobs run on different GPUs but on different cores in a single CPU. Table 4.1 shows the execution times of the case 2) and 3) normalized by the case 1). From this experiment, we have found that there were no contention effects between separate CEs. Our matchmaking algorithm takes those contention effects into account for heterogeneous systems.

Testsuits	Case 2)/ Case 1)	Case 3)/ Case 1)
FFT	1.00163	1.00294
MD	1.00052	1.00064
Reduction	1.00024	1.00038
Scan	1.00087	1.00052
SGEMM	1.00568	1.01195
Sort	1.00392	1.00255
Stencil2D	1.00949	1.00205
Triad	0.99547	1.01417

Table 4.1: Normalized Running Times

Dominant CE If a job needs multiple CEs for its execution, the job may require multiple resource types for each different CE. However, most applications target a specific CE as their main computational resource, and use other CEs as secondary resources. We call this main CE the *dominant CE* of the job. For example, a job using the CUDA library may require a CPU and a GPU, but the CPU is used to control multiple threads in the GPU and the majority of the computation is done on the GPU. In this example, the GPU is the dominant CE for the job. Therefore, matchmaking for such jobs taking into account the dominant CE's requirements first may be the best way to maximize performance and balance loads evenly because the job's execution time is determined by the performance of its dominant CE. The dominant CE should be determined by the user who develops and submits the job, because the user has the best knowledge about which CE type is the most heavily used during job execution. The current matchmaking framework considers the single dominant CE case for this study. However, some applications use multiple CE types heavily [39]. In this case, the user can describe multiple dominant CEs along with weights for their expected resource usage, and the matchmaking algorithm can be

extended to use this weighted load information to choose the job pushing direction.

Job Assignment Policy If there are multiple nodes capable of running a job, we must select the best candidate as the node to run the job. The first choice is to choose a free-node. An acceptable node (but not a free-node) is ranked lower for selection than a free-node because such an assignment can incur contention effects, increasing job turnaround time. If we cannot find an acceptable node, we choose the node that minimizes a score function we now describe, that is based on the job's dominant CE. Let C denote the type of the job's dominant CE, and $CE(N, C)$ denote the C type of CE in node N. The score function for $CE(N, C)$ is defined as the core utilization divided by the clock speed of $CE(N, C)$. If $CE(N, C)$ is a dedicated CE, then the core utilization of $CE(N, C)$ is the number of running and queued jobs (Equation 4.1). If $CE(N, C)$ is a non-dedicated CE, the core utilization of $CE(N, C)$ is the required cores for running and waiting jobs divided by the number of cores in $CE(N, C)$ (Equation 4.2). These score functions prefer the least utilized node for the dominant CE type, relative to its CE clock speed.

$$F(N, C) = \frac{CE(N, C).JobQueueSize}{CE(N, C).ClockSpeed} \tag{4.1}$$

$$F(N, C) = \frac{\frac{CE(N, C).RequiredCores}{CE(N, C).NumberOfCores}}{CE(N, C).ClockSpeed} \tag{4.2}$$

Based on the discussed job assignment policy so far, the complete algorithm for matchmaking and job pushing for heterogeneous environments is described in

Algorithm 2. The job pushing procedure for heterogeneous nodes is not significantly different from the job pushing for multicore nodes 1, but the equations inside the algorithm are changed to be optimized for the heterogeneous environment. The equations in Algorithm 2 are as follows.

$$F_D(N,C) = \frac{AI_D(N,C).SumOfRequiredCores}{(AI_D(N,C).NumberOfCores)^2} \tag{4.3}$$

$$P(N) = \frac{1}{(1 + AI_{TD}(N).NumberOfNodes)^{SF}} \tag{4.4}$$

In Equation 4.3, $F_D(N,C)$ is the objective function for the neighbor node N along dimension D in terms of type of CE C, and $AI_D(N,C)$ is aggregated load information for node N's CE C. In Equation 4.4, $P(N)$ is the probability to stop at node N, and SF is the stopping factor, which is a parameter used to adjust the stopping probability [25]. $AI_{TD}(N)$ is the aggregated load information at node N along the chosen dimension TD.

Now we can perform matchmaking for heterogeneous nodes and jobs using the job pushing algorithm, which we will show balances load well. A remaining issue is the cost of the algorithm; we discuss cost and scalability in the next section.

4.2 Scalable Support for Heterogeneity

Increasing the number of dimensions in the CAN to represent additional resource requirements gives an effective method to match jobs to resources and bal-

Algorithm 2 Job Pushing for Heterogeneous jobs

```
1: Route the job in the CAN to the node whose zone contains the job's coordinate.
2: while run-node not found do
3:    Find an acceptable node(s) among neighbors.
4:    if Found an acceptable node(s) then
5:       if Found a free-node(s) among acceptable nodes then
6:          Pick the free-node with the fastest clock speed for the job's dominant
            CE.
7:       else
8:          Pick the acceptable node with the fastest clock speed for the job's domi-
            nant CE.
9:       end if
10:   else
11:      Choose a target node and dimension to minimize the objective function
         (Equation 4.3).
12:      Determine stopping based on the probability (Equation 4.4) for the target
         dimension.
13:      if Stop then
14:         Select the node with minimum score (Equation 4.1, 4.2) among neighbors.
15:      else
16:         Push the job to the target node.
17:      end if
18:   end if
19: end while
```

ance load across heterogeneous nodes. However, additional dimensions can result in higher communication costs in the CAN, mainly from heartbeat messages between neighboring CAN nodes to maintain connectivity, making the CAN less scalable. In this section, we begin with a cost analysis for the existing system with the original CAN, and suggest two approaches to reduce costs and improve scalability for heterogeneous nodes.

4.2.1 Maintenance Cost Analysis

As we discussed in Section 4.1.1, the CAN must be extended to accommodate more heterogeneous environments. However, adding more dimensions can result in

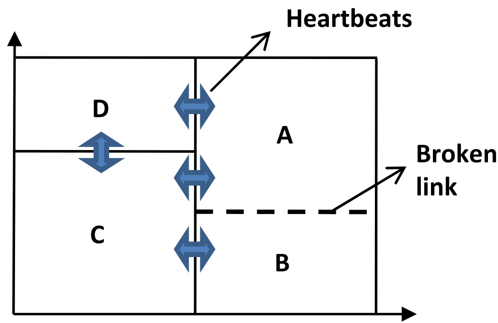


Figure 4.1: Recovery from a Broken Link via Heartbeats

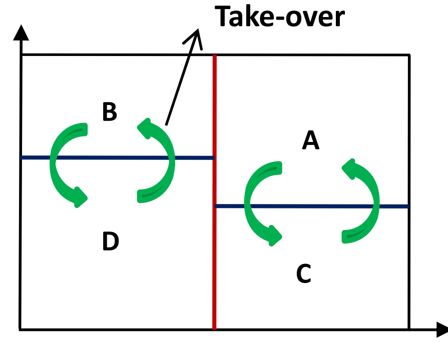


Figure 4.2: Zone Splits and Take-over Nodes

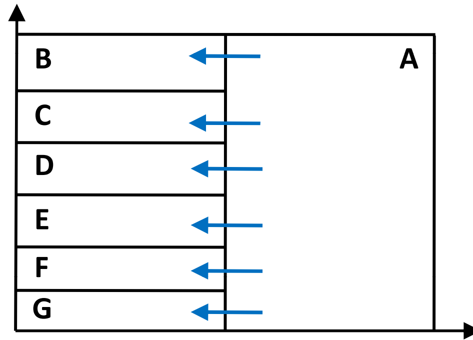


Figure 4.3: Worst Case for Compact Heartbeat

more overhead. We have two major metrics to measure costs over a fixed time period; the number of messages per node and the volume of messages per node. Therefore, we need to evaluate the relationship between the number of CAN dimensions and those costs, across all nodes in the system.

Suppose that the existing CAN, called the *vanilla* CAN to distinguish it from the enhanced CAN that is the subject of this chapter, contains d dimensions to express resource capabilities. The average number of neighbors per node in the CAN is proportional to the number of dimensions, since each node must keep information about at least two neighbors (one in each direction) along each dimension. Also, the

number of heartbeat messages for a node is proportional to its number of neighbors, because heartbeat messages are sent periodically by a node. Therefore, the number of messages per node per minute is proportional to the number of dimensions (O(d)).

However, the volume of messages is proportional to the square of the number of dimensions (O(d^2)). In the vanilla CAN, each heartbeat message must contain all the neighbor information from the sender, since complete neighbor information is needed to take over a CAN zone that is vacated when a node leaves the system voluntarily or fails, to continue to be able to route in the CAN DHT. Therefore, each heartbeat message size is proportional to the average number of neighbors of a node, so is proportional to the number of dimensions, O(d). Thus, the average message volume per node per minute is $O(d) \times O(d) = O(d^2)$. This cost analysis can also be applied to the algorithm in the original CAN [11] because the original CAN also exchanges heartbeat messages with complete neighbor information.

On the other hand, the neighbor information can be used not only for recovering from nodes leaving the system, but also can be used to recover a node's *broken links*, as shown in Figure 4.1. A broken link means that a node has missing neighbor information along an edge of its zone, even though some node already owns the zone on the other side of that edge. For example, node A in Figure 4.1 can receive node B's information from node C's heartbeat message (since C is also a neighbor of B), so node A can fix the broken link using node C's heartbeat message.

4.2.2 Compact Heartbeat

As was discussed in Chapter 2, each dimension in the CAN represents a node resource capability. Therefore, the coordinates for a node can never be changed, except along the virtual dimension. A node's zone in the CAN must include the node's coordinate, so we cannot always split a zone into equal sized zones along a dimension when it is partitioned for a node join operation, as is done for example in a quad-tree spatial data structure. The CAN partitioning algorithm is similar to that of a distributed KD-tree in a d-dimensional space, so a node should maintain its own zone split history, to enable proper zone take-over operations when a neighbor leaves the system voluntarily or fails, to maintain the CAN tree-like structure. Therefore, the take-over node for a given node is predetermined by the leaving/failing node's split history. For example, as seen in Figure 4.2, suppose that the split is done vertically first, and later splits are done horizontally. In this situation, node A and node C are take-over nodes for each other, and nodes B and D take over each other's zone if one of the nodes leaves the system or fails.

Since take-over node information is predetermined, that provides a way to reduce heartbeat message size, because the neighbor information in a heartbeat update is mainly used for take-over operations. We propose a heartbeat messaging scheme with smaller messages, called *compact heartbeat*, that sends full neighbor information in a heartbeat message only to the take-over nodes for the node sending the heartbeat (there can be more than one for some CAN configurations), while other neighbors receive only aggregated load information from the sender node. Compact

heartbeats reduce message size in most situations, since the number of take-over nodes is usually small, so that average message volume per node reduces to O(d). However, in the worst case, the size of the compact heartbeat message is still O(d^2), as shown in Figure 4.3. Node A has many neighbors and all its neighbors are take-over nodes, so node A has to send O(n) messages to all its neighbors (where n is the number of neighbors), and a message has to include all neighbor information, so is of size O(n), because all receiving nodes are take-over nodes. Therefore the messaging cost for the worst case can be O(n^2), but it is very unlikely that this situation will happen to many nodes in the CAN, so the expected heartbeat message volume is O(d).

Using compact heartbeat can reduce overhead costs, while still providing the same resilience to failure as the vanilla CAN, as long as there are no simultaneous events in the system. Such events include node joins, node leaves (voluntarily) and node failures. We have used this assumption (no simultaneous events in a heartbeat period) in our previous work to argue for the completeness of our CAN algorithm. In fact, the original CAN algorithms also assumed no simultaneous events locally to ensure correctness. Therefore, our compact heartbeat scheme achieves the same level of failure resilience as the vanilla CAN, but can greatly reduce message costs, making compact heartbeats a more scalable solution.

4.2.3 Adaptive Heartbeat

While we can assume that there will be no simultaneous events in the CAN in theory, in practice we get no such guarantee. Therefore, we must evaluate the failure resilience of our system under more general assumptions, namely that there may be multiple events in a heartbeat interval among neighbors in the CAN. If simultaneous events happen in adjacent CAN nodes, those events can create broken links for a node. As we discussed earlier, the redundant neighbor information in the vanilla CAN can fix the broken links. However, compact heartbeat messaging cannot recover from the broken link unless a take-over node's heartbeat happens to include the missing neighbor information for the broken link. In that case, the vanilla CAN is more resilient to failure than with compact heartbeats.

We propose an *adaptive heartbeat* scheme to improve failure resilience with compact heartbeat. Adaptive heartbeat is an on-demand update mechanism that is added to compact heartbeat. In the adaptive heartbeat scheme, nodes exchange heartbeats using the compact heartbeat scheme under normal circumstances. However, when a node detects a broken link on one of its edges, the node broadcasts a *full-update request* to all neighbors. A node that receives a full-update request responds to the requesting node with full neighbor information, to help the requesting node recover from the broken link. For example, in Figure 4.1, if node A finds a broken link towards node B, then node A sends a full-update request to node C and node D. Node C responds to node A with information about node B so that node A can reconstruct the broken link to node B. Therefore, adaptive heartbeat is as

	Vanilla CAN	Compact heartbeat	Adaptive heartbeat
Number of Messages	$O(d)$	$O(d)$	$O(d)$
Volume of Messages	$O(d^2)$	close to $O(d)$	close to $O(d)$
Error Resiliency (without simultaneous events)	No error	No error	No error
Error Resiliency (with simultaneous events)	Good	worse than Vanilla	close to Vanilla

Table 4.2: Comparison Summary of Vanilla CAN, Compact Heartbeat and Adaptive Heartbeat

failure resilient as vanilla CAN in many cases, but the cost for adaptive heartbeat is nearly as low as for compact heartbeat. Table 4.2 is a summary of asymptotic analysis of vanilla CAN, compact heartbeat, and adaptive heartbeat. Note that cost representation is normalized (per node per minute), so it just shows the relationship between the cost and the number of dimension ($=d$). In the next section we present simulation results comparing the three approaches.

4.3 Experimental Results

We present two sets of experimental results. The first shows the performance of our matchmaking and load balancing scheme for heterogeneous environments. We have compared job wait times with an online centralized matchmaker to confirm that our decentralized solution is comparable in performance to a centralized approach. The other experiment shows the scalability and failure resilience of our heterogeneous solution. We describe a set of experiments that varies the number of nodes and the number of CAN dimensions to measure overall system costs and compare the costs of our two approaches with the vanilla CAN.

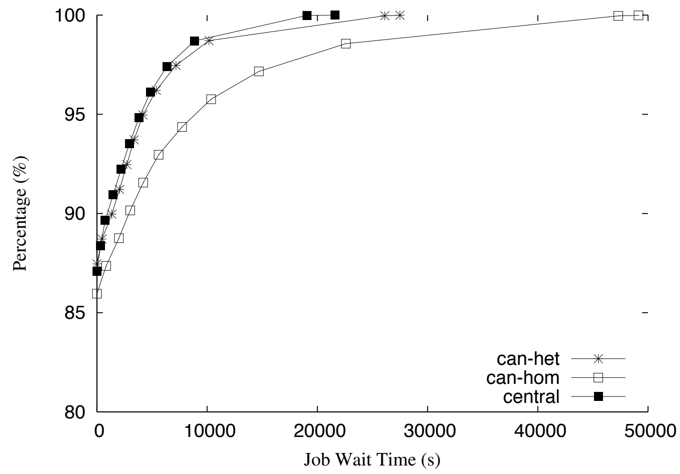
4.3.1 Load Balancing Performance

Setup

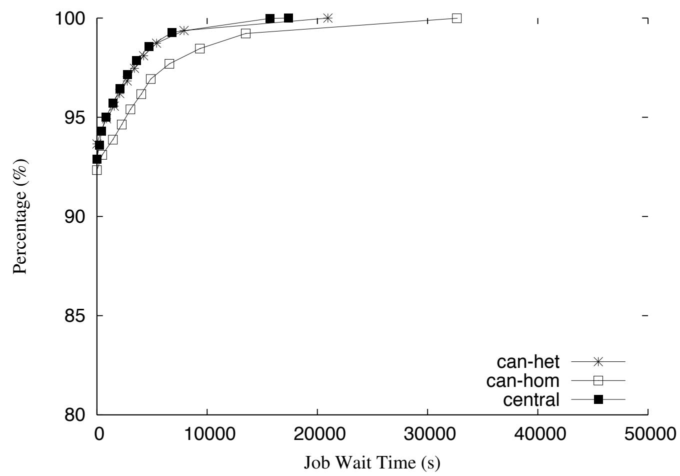
In this simulation, we used an event driven simulator described in Section 3.3 Our simulation scenario contains 1000 heterogeneous nodes, and 20,000 jobs are submitted to those nodes. The simulations are executed on an 11-dimension CAN as in the example in Section 4.1.1. Each node potentially has either a single- or multi-core CPU (1, 2, 4 or 8 cores), and may include up to two different types of GPU. The resource characteristics for a CPU are CPU clock rate, memory size, disk space, and number of cores. Each GPU has three characteristics: GPU clock rate, GPU memory, and number of GPU cores. Therefore nodes in our experiments can have up to 10 resource characteristics, although more dimensions could be added to specify other types of resources, such as memory bandwidth [42], if users desired to match on those resources.

The interval between individual job submissions follows a Poisson distribution, and we vary the average inter-job arrival times in the experiments. The inter-job arrival rate effectively determines average total system load since we run the simulations in a steady-state environment. Each job has an expected running time with an average value of 1 hour, uniformly distributed between 0.5 and 1.5 hours. However, the simulated job execution time is scaled up or down by the corresponding dominant CE's clock speed, which is specified relative to a nominal clock speed.

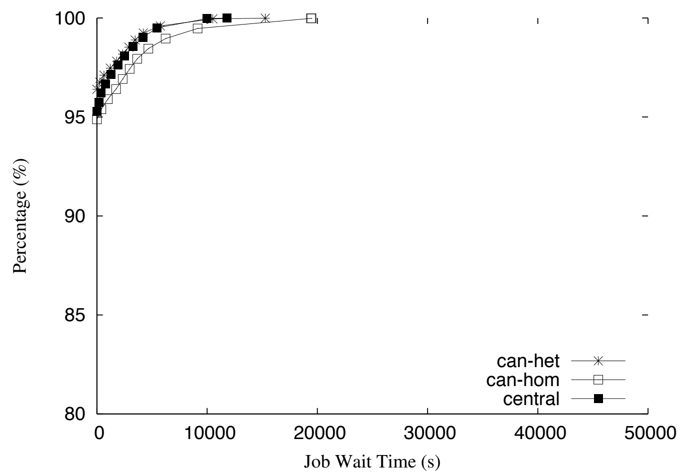
For comparison purposes, we implemented a greedy online centralized scheduler (named *central* in the graphs) as we did for multi-core cases, which assigns jobs



(a) 2 seconds

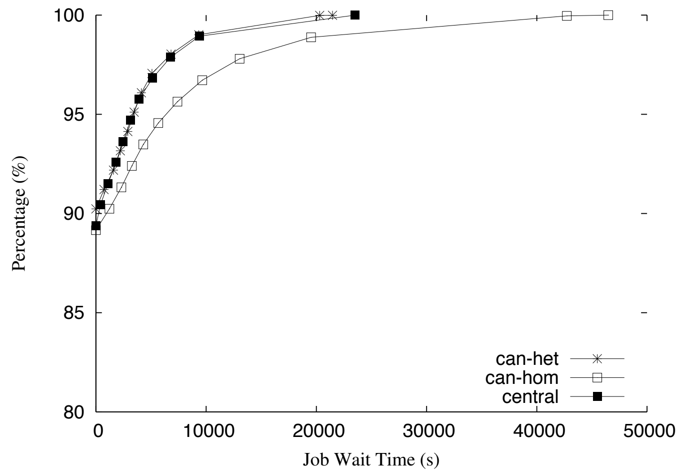


(b) 3 seconds

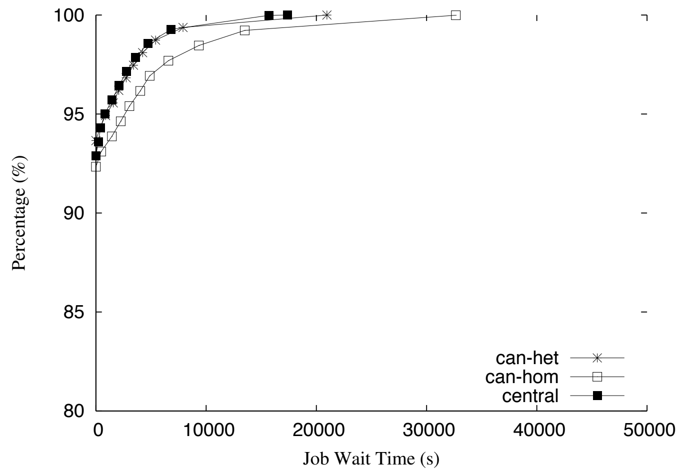


(c) 4 seconds

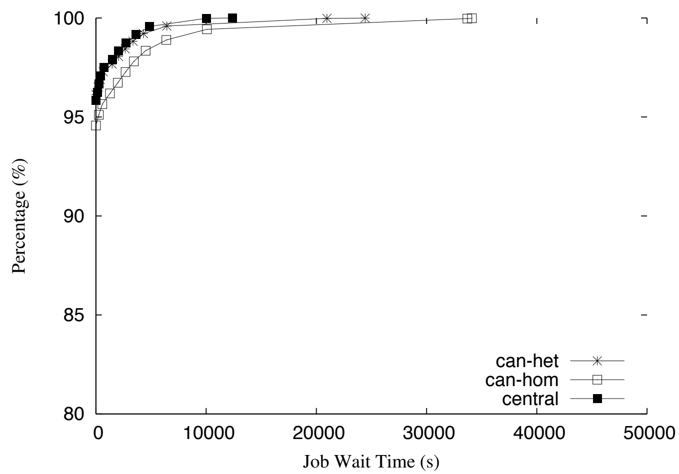
Figure 4.4: CDF of Job wait time varying Inter-arrival Time



(a) 80%



(b) 60%



(c) 40%

Figure 4.5: CDF of Job wait time varying Job Constraint Ratio

based on complete load information across all nodes. We also compare our new approach against our previous work, which is oblivious to heterogeneous resources (named *can-hom*). Because can-hom ignores various considerations described in Section 4.1, job push decisions in can-hom can lead to a poor choice for a run-node, since it is based on inaccurate aggregated information.

Load Balancing Performance

Figure 4.4 shows matchmaking and load balancing performance in the heterogeneous grid system compared to central and can-hom, where we vary average job inter-arrival times from 2 seconds to 4 seconds. Lower job inter-arrival time means a heavily loaded system, and higher job inter-arrival time results in a lightly loaded system. The figure shows cumulative distributions for job wait times, where wait time is measured from when a job is placed on a run-node after matchmaking to when the job starts executing. Note that the Y axis starts at 80% to better see the difference among the three matchmaking schemes. Overall, the performance of the decentralized scheme is not much different from the centralized solution, as measured by job waiting time, regardless of job inter-arrival time. However, when the system becomes more loaded, the performance gap between our heterogeneous scheme (named *can-het*) and can-hom becomes larger. This means that can-hom cannot balance load very well when the system gets heavily loaded.

Figure 4.5 shows load balancing performance versus job constraint ratio, i.e., load balance versus difficulty in matching jobs to nodes. The job constraint ratio can also affect load balancing performance because a higher job constraint ratio

makes the matchmaking problem more difficult. Similar to the results for varying job inter-arrival time, when the job constraint ratio is low (i.e. 40%), the three schemes show similar performance, while higher job constraint ratios can lead can-hom to misdirect jobs to heavily-loaded nodes. However, the heterogeneous scheme shows performance competitive to the centralized matchmaker for all job constraint ratios.

From these simulations, we confirm that our matchmaking and load balancing performance is competitive to the online centralized matchmaker, and better than the approach for homogeneous environments.

4.3.2 Scalability and Heterogeneous Resources

Setup

To test the scalability and failure resilience of our algorithms for heterogeneous environments, we have experimented with 5, 8, 11 and 14 dimensional CANs with 500, 1000 and 2000 nodes, respectively. In the initial stage of each experiment, n nodes join the system sequentially. After that, node *join* and node *leave* events occur with equal probability, so that the number of nodes in the system converges to a dynamic equilibrium. The time gap between events (join or leave) in the second stage of the experiment is either longer than a heartbeat period (to ensure no multiple simultaneous events), or shorter than a heartbeat period (to see the effects of multiple simultaneous events). We ran simulations for the vanilla CAN, with compact heartbeats, and with adaptive heartbeats for each configuration.

Failure Resilience

To measure behavior in the presence of multiple simultaneous failures, we tracked the number of broken links over time in the system at a given point in time. First, we ran simulations for three approaches (vanilla CAN, compact heartbeat and adaptive heartbeat) under the assumption that *no multiple events* occur in a heartbeat period. No broken links occurred in the entire experiment for all three approaches in that case. This result shows that both compact and adaptive heartbeat are failure-resilient just like vanilla CAN if no simultaneous events happen during a heartbeat period.

We ran another set of experiments with multiple events within a heartbeat period. This scenario implies high churn, meaning that nodes are joining and leaving frequently, to the extent that failures (broken links) may not be repaired even by the end of an experiment.

Figure 4.6 shows the change in the number of broken links over time for the 11-dimensional CAN. Note that the X axis begins at 10000 seconds, because there are no broken links in the initial part of the experiment. We see that the number of broken links increases as time elapses, and then mostly levels out, because irreparable links accumulate and these accumulated errors may cause additional failures. However, all three schemes appear to have reached steady-state behavior.

The figure shows that: 1) vanilla CAN shows the most failure-resilience (meaning the fewest broken links), 2) compact heartbeat is the least failure-resilient, achieving its performance gains at the expense of approximately 70% more link failures in this experiment, and 3) adaptive heartbeat is better at recovering from

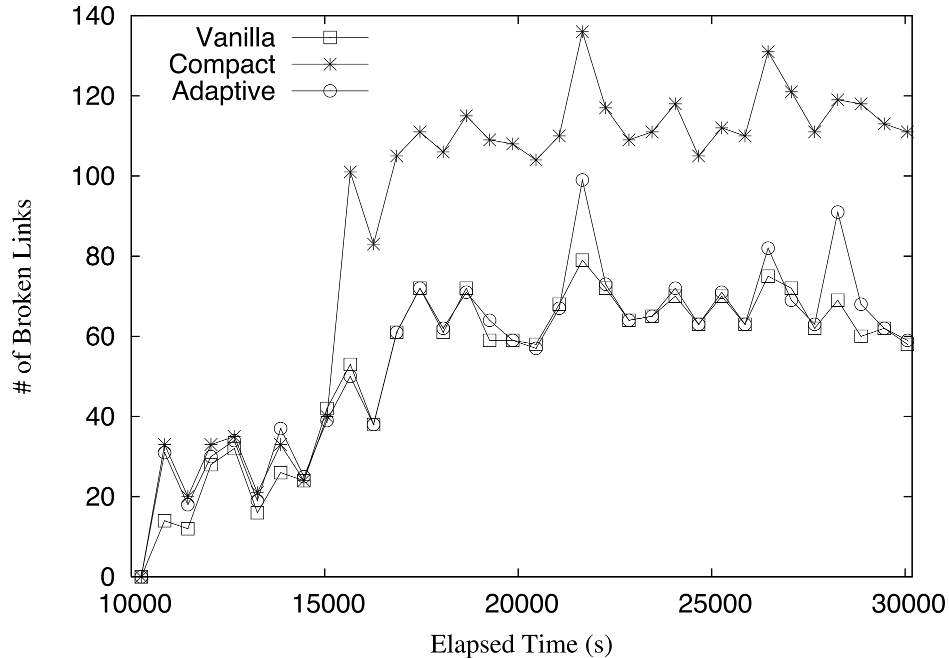


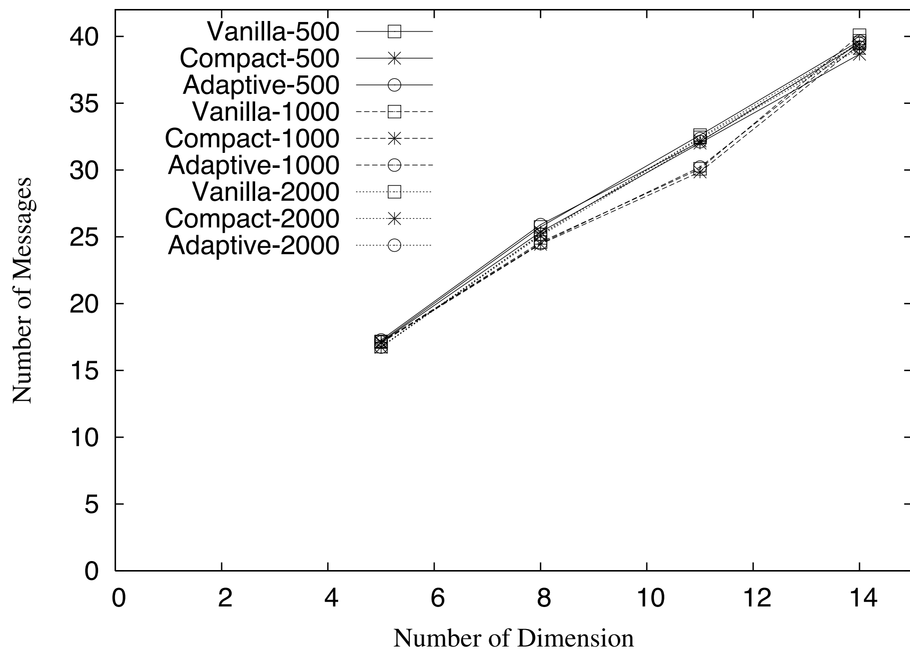
Figure 4.6: Broken Links under high churn

failures than compact heartbeat, and performs very close to vanilla CAN. We have conducted a number of experiments varying the parameters for this experiment with qualitatively similar results.

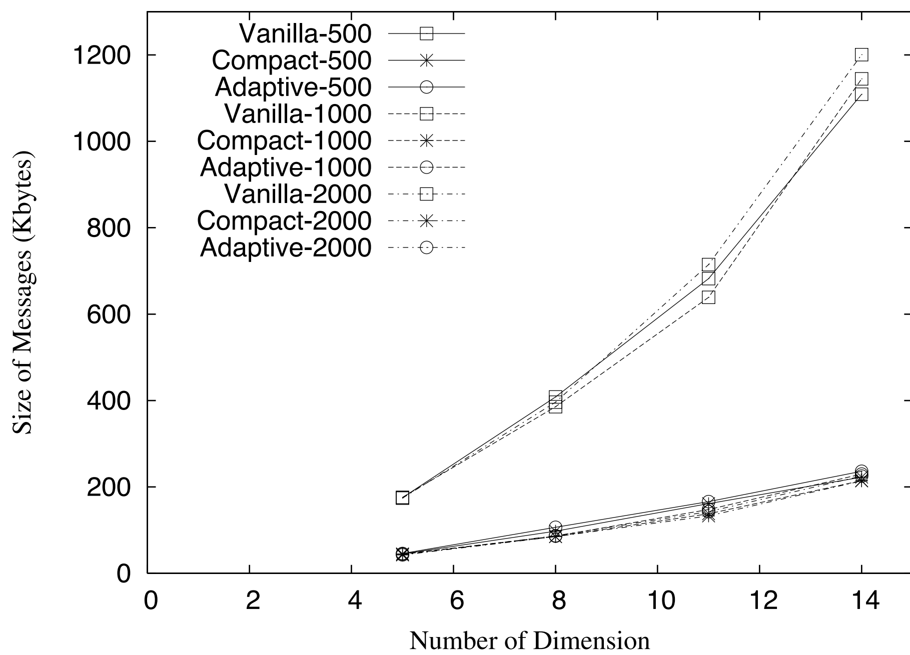
We conclude that adaptive heartbeat is comparable in resilience to failure to vanilla CAN even under high churn.

Scalability

As discussed in Section 4.2, we claim that our compact and adaptive heartbeat schemes are more scalable than vanilla CAN, as measured by messaging costs. To confirm this claim, we have conducted experiments with various numbers of nodes and dimensions, and measured the costs for heartbeat messages. Figure 4.7 shows the cost for varying numbers of nodes and CAN dimensions. Each sub-figure shows how the number of messages or the volume of messages increases as the number



(a) Number of Messages



(b) Size of Messages

Figure 4.7: Scalability, measured per node per minute

of CAN dimensions increases. Note that the number of messages and the volume of messages in Figure 4.7 are average values (i.e., per node per minute). Each

line shows the result of a set of configurations for each mechanism (vanilla CAN, compact heartbeat, and adaptive heartbeat) and the number of nodes, denoted by the number after the dash in the legend. For example, Vanilla-1000 denotes the result for the vanilla CAN mechanism with 1000 nodes.

The number of messages per node per minute (Figure 4.7(a)) is proportional to the number of dimensions because compact heartbeat reduces message length, not the number of messages. Moreover, we can see that the adaptive heartbeat does not incur additional overhead compared to compact heartbeat; in fact, it is difficult to tell the differences among the results from the three schemes. The results also are mostly insensitive to the number of nodes in the system, since all messaging is only to a node's neighbors in the CAN.

In Figure 4.7(b), the message sizes for the vanilla CAN increase with $O(d^2)$, but for compact and adaptive heartbeats show close to a linear increase, as expected. The decreased message volume would become more important for even larger numbers of CAN dimensions, from additional node resource types. Thus our compact and adaptive heartbeat algorithms are more scalable than the vanilla CAN. In addition, note that the message volume does not increase regardless of the number of nodes in the system, which means that the message cost is perfectly scalable with system size.

4.4 Summary

In this chapter, we propose a decentralized resource management and job scheduling scheme that exploits diverse computing elements in heterogeneous environments. By considering features of heterogeneous nodes, i.e., multiple, different types of computing elements in a single node, our solution effectively utilizes diverse kinds of computing elements and provides an efficient matchmaking technique to satisfy numerous resource constraints due to heterogeneity. We have confirmed via extensive simulations that our proposed scheme shows load balancing performance competitive to an ideal centralized approach, and better than our prior approach ignoring heterogeneity.

However, support for matchmaking jobs with many resource constraints in heterogeneous environments can cause the overall system to scale poorly. We have analyzed the system costs required to maintain the underlying CAN DHT, with respect to the complexity of the job resource requirements and found that the messaging cost is $O(d^2)$ in the number of dimensions for the existing system. We have described more scalable solutions to reduce the costs to $O(d)$ without sacrificing system resilience to node failure. Via comprehensive simulations, we have shown that our compact and adaptive heartbeat methods can greatly reduce message costs, while still being as resilient to failures as the original DHT.

Chapter 5

Multi-attribute Range Search

In this chapter, we describe a flexible resource discovery technique that effectively supports matchmaking for multi-attribute, range-based categorical constraints. Specifically, our approach accommodates cases where 1) a categorical resource attribute can be satisfied with multiple values as a form of disjunction (e.g. $OS == Ubuntu10.04 \vee 10.04.1 \vee 10.04.02 \vee 10.04.03$), and 2) any values for a categorical attribute can satisfy the job (called a *don't care* condition, e.g. $OS == *$, for example, a job running on a Java virtual machine does not care about the operating system). Second, we show that our approach is not very sensitive to stale information under situations of high node churn by extensive simulations.

5.1 Range-type Search Algorithm

We describe our new approach for matching resources to jobs with range-based categorical resource constraints. If we use a *transform dimension*(see Section 2.4) for categorical type resources, jobs are routed to a specific sub-CAN first, so jobs with range-type constraints for categorical resources cannot be distributed evenly across sub-CANs. Our new approach differs from previous work in the use of only a single CAN to represent both continuous and categorical resources. More specifically, only continuous resource types are mapped onto CAN dimensions. Instead

of using a transform dimension for categorical resource types, each node has a specific categorical identifier (CID) that encodes the node's values for all categorical resource types. Use of CIDs enables range-based searches and load balancing with a straightforward and efficient architecture.

5.1.1 ID-based Resource Representation

We first present how to represent a node's (or job's) categorical resources with the CID method. Suppose that there is a (bi-jective) mapping function f from n-dimensions to 1 dimension. The input of this function f is a set of n categorical resource values; the output is a CID describing a specific combination of input categorical resource values. Therefore, if we know the CIDs of a node and a job, we can directly determine whether the node meets the job's categorical constraints (details about how to do are described in later).

Through the CID approach, we can separate the categorical resource representation from the CAN dimensions. Every node joins a single CAN and is allocated a zone in the CAN based on its continuous resource capabilities; categorical resources are not considered for the node's zone in the CAN. Each node uses its CID to express its categorical resource capabilities, and the CID information is propagated along each dimension towards the CAN origin as part of the periodic CAN update mechanism, so that aggregated CID information can be used for the matchmaking. We discuss the details of the information propagation in Section 5.1.3, and the matchmaking algorithm based on CID information in Section 5.1.2.

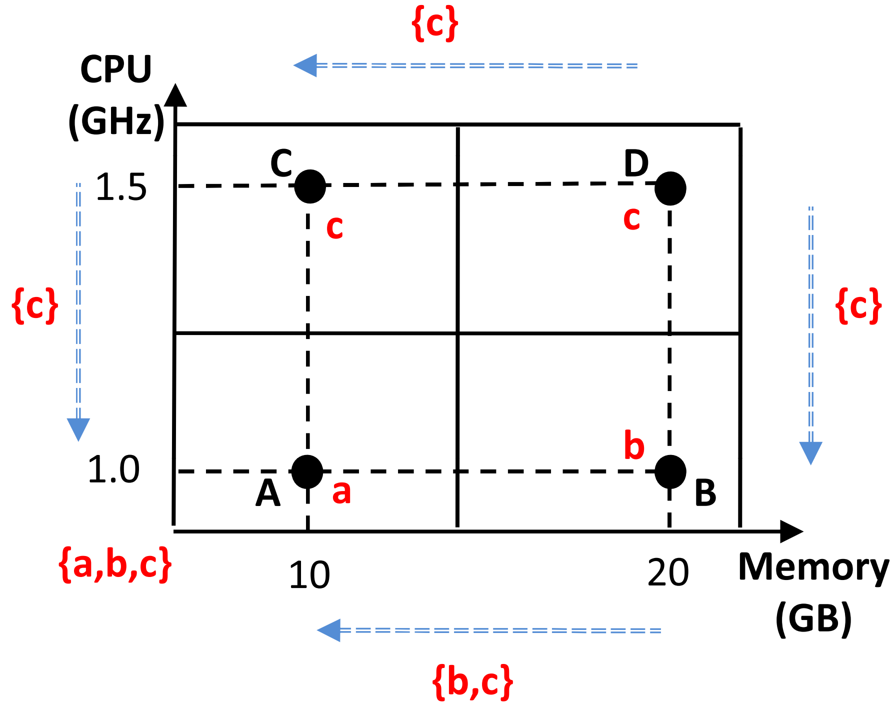


Figure 5.1: An Integrated CAN with four nodes: lower case letters denote CIDs, and each set next to the dotted-arrow shows aggregated CID information flow

An example shows how to construct a single CAN embedding both continuous and categorical resource information using the CID approach. Table 5.1 shows the categorical resource values for four nodes, as well as its CIDs. For this example, we assume that there are two continuous resource types, memory size (GB) and CPU speed (GHz), and two types of categorical resources, for example CPU architecture and OS. Node D in Table 5.1 has the same CID as Node C because the categorical resource values of the two nodes (MIPS & Linux) are the same.

	Memory	CPU	Archi	OS	CID
Node A	10	1.0	Intel x86	Linux	a
Node B	20	1.0	Intel x86	Windows	b
Node C	10	1.5	MIPS	Linux	c
Node D	20	1.5	MIPS	Linux	c

Table 5.1: Node Resource Capabilities & CIDs

Figure 5.1 shows the CAN built from the four nodes shown in Table 5.1. A node's zone contains its coordinate, which is the node's *continuous* resource capabilities. Each node delivers its aggregated CID information (abbreviated as $ACID$) for *categorical* resources towards the CAN origin along each dimension, so that each node has information about what kinds of CIDs for nodes exist in the outer regions of the CAN. Equation 5.1 defines the ACID, where $AC(n)$ is the ACID of node n, $C(n)$ is the CID of node n, d is the number of dimensions in the CAN, and $UN_i(n)$ is the set of direct neighbors farther from the origin (the *upper* neighbors) of node n for dimension i.

$$AC(n) = C(n) \cup \bigcup_{i=1}^{d} \left(\bigcup_{k \in UN_i(n)} AC(k) \right) \tag{5.1}$$

The ACID for a node is the **union** of the sets of CIDs of nodes farther from the origin along all dimensions, along with the node's own CID. For example, in Figure 5.1 $AC(D)$ is a just $C(D) = \{c\}$, because node D is the outermost node in both CAN dimensions. $AC(B)$ is $C(B) \cup AC(D) = \{b, c\}$, so node B knows node D's CID along the CPU dimension. In the same way, $AC(A)$ is $C(A) \cup AC(B) \cup AC(C) = \{a, b, c\}$, so node A knows the CIDs of all nodes by information aggregation. But ACID can grow as the number of possible combinations of categorical resources; we will revisit this issue in later section.

5.1.2 Multi-attribute Requirements

If job requirements are given as a range of values for one or more categorical resources, we can represent the job's categorical requirements as a set of CIDs. In fact, this representation is also a union over all acceptable categorical resource combinations for each job. Since job requirements may not be specified as an exact match (i.e., a range), a job can have multiple acceptable combinations for categorical resource constraints, so the number of elements in the CID set for a job can be greater than 1, whereas the CID set of a node has only a single element. To test whether a node can meet a job's categorical resource requirements, we need to perform a *set membership check*. That operation determines whether a job's categorical requirements are met by the ACID of the node. If the set membership check returns true, that means that one or more nodes satisfying the job's constraints exist in the *upper* region of the CAN from that node (meaning the regions farther from the origin).

The detailed matchmaking process using the CID method works as follows. First, a job is routed to a point that satisfies its minimum *continuous* resource requirements. We can then check whether each upper neighbor has a node capable of running the job in its upper region, looking at *categorical* resources constraints by comparing the neighbor's ACID and the job's CID. If there is no capable node for a particular dimension, we do not attempt to push the job along that dimension. If multiple dimensions have capable nodes, we can push this job along the least loaded dimension, as measured by aggregated *load* information, as is done

for continuous resources. The detailed description of the job pushing algorithm is shown in Algorithm 3. The objective function (Equation 4.3), the probability function (Equation 4.4) , and the score functions (Equation 4.1, Equation 4.2) in Algorithm 3 are described in Chapter 4.

Algorithm 3 Matchmaking/Job Pushing for efficient multi-attribute, range-based categorical resource requirements

```
1: Transform the job's categorical resource requirements into a CID(s) form using
   the mapping function.
2: Route the job in the CAN to the node containing the job's continuous resource
   coordinates.
3: while a run-node is not found do
4:    Find a set of acceptable nodes among neighbors satisfying continuous resource
      constraints.
5:    if Found an acceptable node(s) then
6:       if Found a subset of acceptable nodes satisfy job's categorical requirements
         (check the CID) then
7:          Pick the acceptable node with the fastest clock speed.
8:       end if
9:    else
10:      Choose a target node and dimension to minimize the objective function
         (Equation 4.3)
11:      Check existence of capable nodes for that direction in terms of categorical
         requirements based on ACID.
12:      while An capable node does not exist for that direction do
13:         Choose the next target node/dimension based on the objective function,
            and check existence of capable nodes for categorical resources again.
14:      end while
15:      Determine stopping based on the probability (Equation 4.4) for the target
         dimension.
16:      if Stop then
17:         Select the node with minimum score (Equation 4.1 or Equation 4.2)
            among neighbors.
18:      else
19:         Push the job to the target node.
20:      end if
21:   end if
22: end while
```

5.1.3 Implementation Choices

There are many ways to represent a unique ID and support union/set membership checks for sets of IDs efficiently. This section proposes a practical method based on bit strings. Assume that the number of possible combinations of categorical resource values is M. The CID (output of mapping function f) can be represented by an M bits string that is filled with all zeros except a 1 in only one bit position. Therefore, a CID is distinguished by a position in an M-bits string which is set to '1'. Table 5.2 shows a bi-jective mapping and possible CIDs in strings that are 4 bits long.

For CID information aggregation, each node sends a bitwise-OR value of its CID and the ACIDs for its upper regions. A bitwise-OR operation for these bit strings is a union operation over the sets. The location of the bit that is set in a CID is unique for each combination of categorical resource values, so bitwise-OR operations produce the complete set of combinations of categorical resource values for a node and all the nodes in its upper region. Figure 5.2 shows an example of the CID representation and ACID propagation using the bit string approach.

To allow matching multiple categorical resource requirements for a job, we can use a CID to represent the job's categorical requirements as follows. Each bit in

	Memory	CPU	Archi	OS	CID
Node A	10	1.0	Intel x86	Linux	0001
Node B	20	1.0	Intel x86	Windows	0010
Node C	10	1.5	MIPS	Linux	0100
Node D	20	1.5	MIPS	Linux	0100

Table 5.2: Nodes Resource Capabilities & CIDs: Bit String Representation

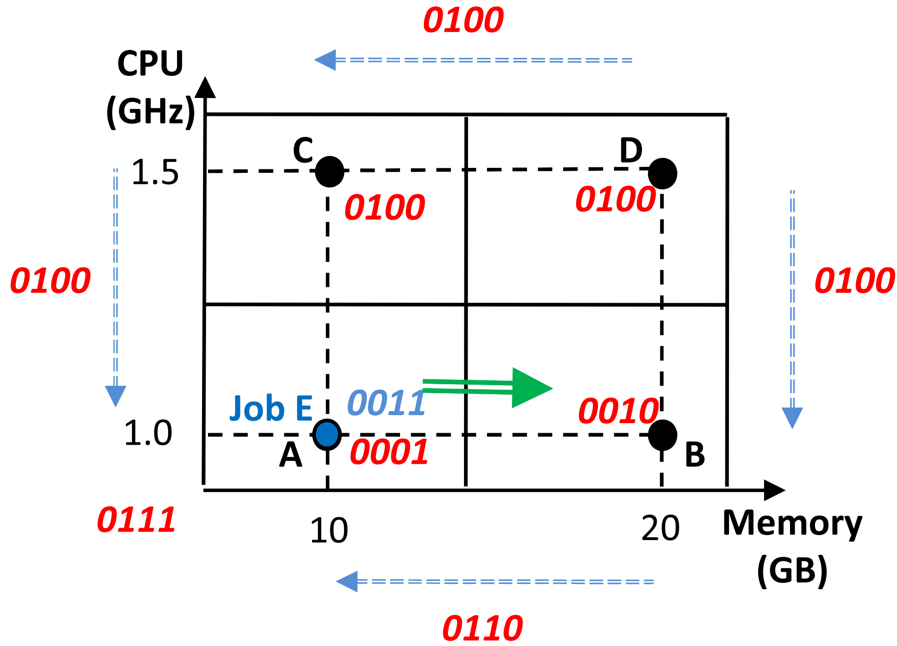


Figure 5.2: Integrated CAN for four nodes: Bit string approach

the CID for a job needs to be set if the corresponding combination of categorical resource values satisfies the job's requirements. Therefore, if a job has a wide range of acceptable categorical resource constraints, then the number of '1' bit in its CID becomes large. This representation is also essentially a union operation across all acceptable combinations for the job. For example, if a job E requires the Intel x86 platform but does not care about the OS, then $C(E)$ is 0011 from Table 5.2, because both Linux & Intel x86 and Windows & Intel x86 meet the job's constraints so $C(E) = 0001|0010 = 0011$.

For the set-membership operation, we compute the bitwise-AND of the node's CID and the job's CID. If the output of the bitwise-AND has at least 1 bit set, then the node satisfies the job's categorical resource requirements. Similarly, we can check whether an upper region has (at least) one node that meets the job's

requirement using the bitwise-AND operation and the ACID for the node. The set membership check can also be used in the job pushing process described in the previous section. For example, in Figure 5.2 if job E's continuous requirements are (10GB, 1.0GHz), the job E is initially routed to node A's zone. At node A, job E may be pushed to node B or node C because both nodes are located in the upper region of node A, without considering categorical resource constraints, but job E can be pushed to node B only because node C and the upper region of node B do not have capable nodes that meet the categorical resource requirements. In more detail, node B's ACID is 0101 and Node C's ACID is 0100. Bitwise-AND of those ACIDs with the job's CID (0011) are 0001 and 0000, respectively. Since the output of the bitwise-AND of node B's ACID and job E's CID has one '1' bit, job E can be pushed to node B.

Note that this representation and set membership check do not produce any false positives, thus every decision on the job pushing process is accurate so that we can support *completeness* in our matchmaking process. One major concern is the bit string length - we are using an M bit string to represent the M different combinations of categorical constraints. Generally the combinations of the categorical resource types for the machines in the P2P grid system do not vary across all such combinations. Even in the worst case, where every node has different categorical resource capabilities (not a likely scenario), the bit length really needed is the number of nodes in the system. However, if the length of the bit string really matters, we can use a Bloom-filter [43] scheme, or another compression technique to reduce bit length at the cost of decreasing accuracy (but with Bloom filters we would get

to choose how to make that trade-off).

One problem for the CID approach and the sub-CAN approach is when a new value for a categorical resource type is needed after the CAN construction. For example, if a new version of the OS is released after the CAN structure is built, we need a dynamic update to assign new bit(s) to accommodate a new version in the CIDs. We have assumed that the mapping function for categorical values (i.e. Table 5.2 in the example) has been pre-determined before CAN construction so that every node can maintain the same mapping information for categorical resource types. However, if we need to add a new bit for a new value for a categorical resource type, the update can be done in the same way as the information propagation algorithm. We first pick a few nodes by choosing random coordinates, and update the CID mapping function in the nodes to add a new bit. The newly updated nodes can then propagate the new information for categorical resource values to neighbor nodes by heartbeat messages, but now a node has to not only incorporate its categorical resource values into the CID, but also update the meaning of the bits in the CID. We can assume that such updates to add a categorical value do not happen frequently, so the system overheads due to the updates would be negligible.

5.2 Experimental Results

In this section, we present two sets of experimental results. First, we show the performance and costs of matchmaking and load balancing using the CID approach. As a baseline, we compare the performance of using CID against both an online cen-

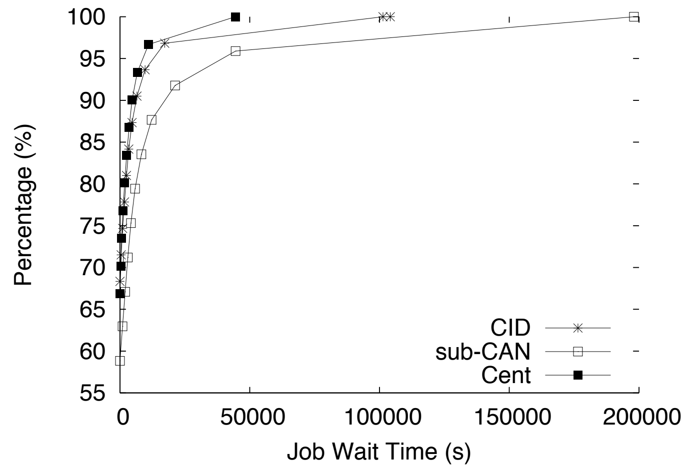
tralized matchmaker and the earlier sub-CAN approach described in Section 2.4, as measured by job wait times, to show that the decentralized CID solution is comparable in performance to a centralized approach and much better than the sub-CAN method. Another set of experiments shows how performance of the CID approach is affected by staleness of the aggregated information communicated across the CAN We have measured the accuracy of the aggregated CID and load information, and its effects on job wait times, as we increase the interval between updates passed between neighboring nodes.

5.2.1 Load Balancing Performance

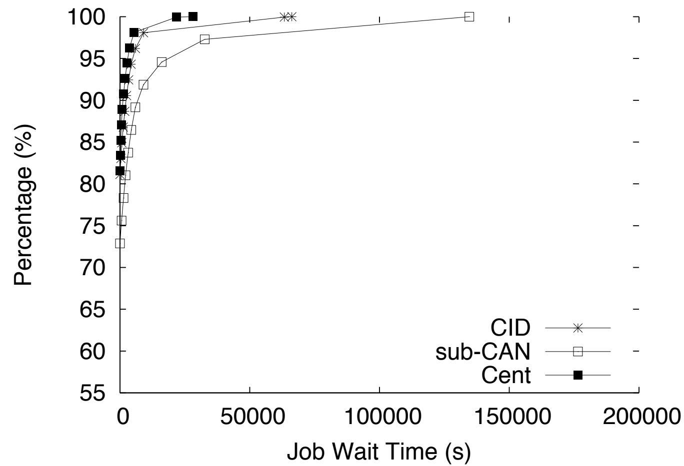
The first experiment looks at load balancing performance when jobs' requirements for a categorical resource type are given as a range. We used the same configurations described in previous chapters (Chapter 3, Chapter 4). The simulation scenario contains 1000 heterogeneous nodes and 20,000 jobs are submitted to those nodes. The job workload about execution time and continuous resource characteristics is the same as that in the previous experiments in Section 3.3.1. We compare our CID approach against the online centralized scheduler (denoted as *Cent* in the graphs) as well as our previous work, which is based on the sub-CAN approach described in Section 2.4 (denoted as *sub-CAN* in the graphs). Sub-CAN is optimized to support exact matches for categorical resource types, but it cannot support multi-attribute range-type job requirements efficiently, because a sequential search along the transform dimension is needed to find all the sub-CANs that match

the job requirements.

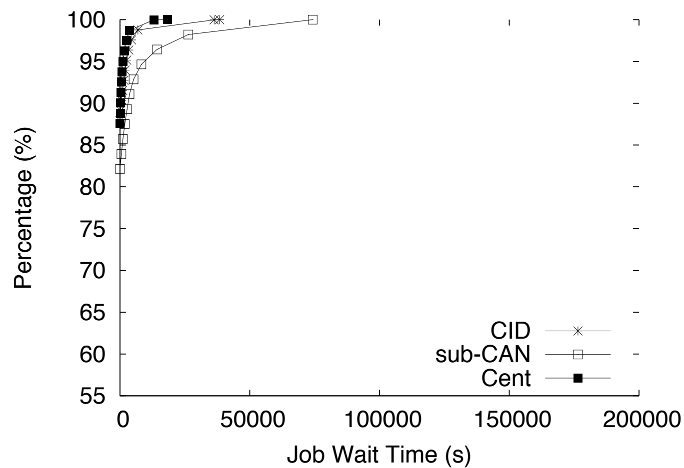
Nodes have four kinds of categorical resource characteristics, and each categorical resource can have two values. Therefore, nodes can have 16 distinct categorical resource combinations (i.e. there can be 16 sub-CANs in the sub-CAN scheme), and the number of nodes in each sub-CAN is approximately equal. We submitted exact match jobs as well as range-type jobs for categorical resources. In these experiments, 62.5% jobs are exact-match cases, and the rest of the jobs have a "don't care" categorical constraint. Therefore, if we assign jobs without carefully considering load balancing across sub-CANs, total matchmaking performance can be very poor. Figure 5.3 shows load balancing performance compared to the online central scheduler (Cent) and the sub-CAN approach, where we vary average job inter-arrival times from 3 seconds to 5 seconds. Lower job inter-arrival time means a heavily loaded system, and higher job inter-arrival time results in a more lightly loaded system overall. The figure shows cumulative distributions for job wait times, where wait time is measured from when a job is placed on a run-node after matchmaking to when the job starts executing. Note that the Y axis starts at 55% to better see the difference among the three matchmaking schemes. Overall, the performance of the CID scheme is not much different from that of the centralized solution, as measured by job wait times, regardless of job inter-arrival time. Sub-CAN shows long tails in the graphs for all case, which means that a few nodes have many waiting jobs due to load imbalances across sub-CANs. Overall, our decentralized CID solution shows competitive performance to Cent, and performs better than the sub-CAN approach over different system loads.



(a) 3 seconds

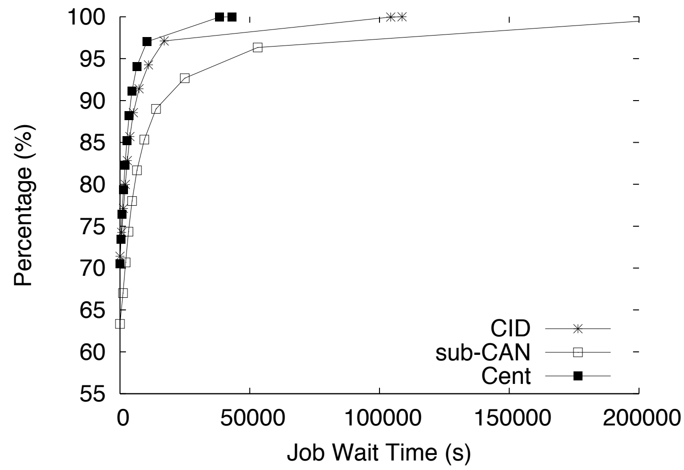


(b) 4 seconds

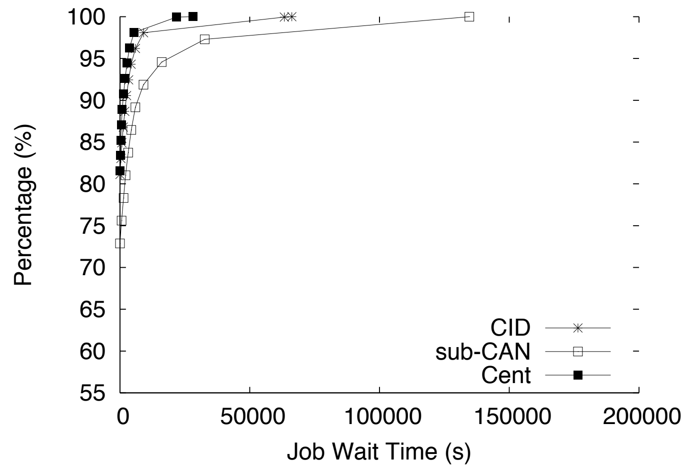


(c) 5 seconds

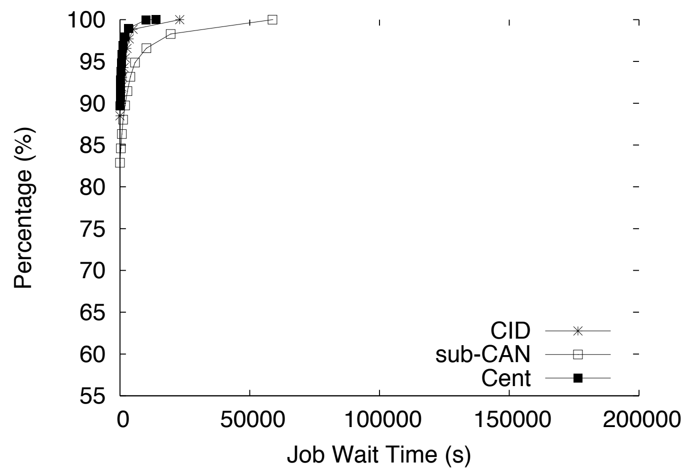
Figure 5.3: CDF of Job wait time varying Inter-arrival Time



(a) 80%



(b) 60%



(c) 40%

Figure 5.4: CDF of Job wait time varying Job Constraint Ratio

Figure 5.4 shows load balancing performance versus job constraint ratio, i.e., load balance versus difficulty in matching jobs to nodes. The job constraint ratio can also affect load balancing performance because a higher job constraint ratio makes the matchmaking more difficult. Similar to the results with varying job inter-arrival time, when the job constraint ratio is low (i.e. 40%), CID shows very similar performance to Cent, while higher job constraint ratios can lead CID to misdirect jobs to heavily-loaded nodes because there are fewer nodes capable of running each job in the system, so peer-to-peer matchmaking with limited global information can experience difficulty in finding less loaded nodes. In addition, sub-CAN shows overall load imbalance because it fails to balance load effectively across sub-CANs.

From these simulations, we confirm that our matchmaking and load balancing performance is competitive to the online centralized matchmaker for this scenario, and effectively supports workloads with multi-attribute, range-based categorical resource requirements, while the sub-CAN approach does not perform well for that type of workload.

5.2.2 Cost Analysis

We evaluate two aspects of our system cost. We first gathered statistics on messaging overhead among nodes to maintain our peer-to-peer system during the simulation period, and compared our CID approach with the sub-CAN for all experiments. Overall, we do not see any significant difference between the two methods. Since heartbeat messages to maintain connectivity of the CAN overlay network are

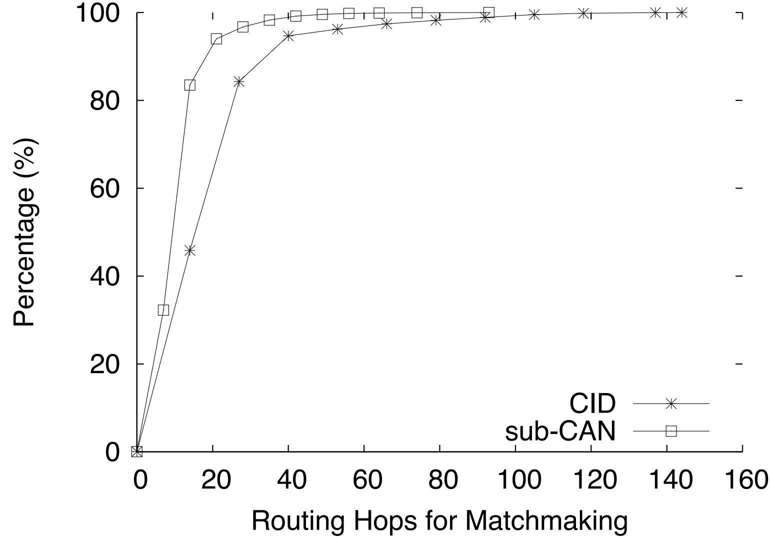


Figure 5.5: CDF of number of Routing hops for matchmaking

the major portion of the total messaging costs, and heartbeat costs are determined by the structure of the CAN (mainly the number of dimensions) [44], the similarity of the messaging costs for the CID and sub-CAN approaches is not surprising.

The other metric is the number of hops for the matchmaking process for a job. Figure 5.5 shows cumulative distributions of the number of hops for the matchmaking process for both approaches. We show only one of the experimental results (60% job constraint ratio, 4 seconds inter-arrival time), but other overall distributions for the other experiments are similar to those presented. This result indicates that the CID approach needs more hops to find the run-node on average as well as in the worst-case, compared to sub-CAN. The main reason for this difference is that the sub-CAN approach has fewer nodes in a sub-CAN than for CID; for example, in this experiment, there are 250 nodes in each sub-CAN for the sub-CAN approach, while all 1000 nodes are in a single CAN for CID. The number of hops for job pushing is affected by the number of nodes in a CAN - in the worst case, we may have to

search all the nodes along a particular dimension to find a free node to run on when the system is highly loaded (meaning all capable nodes are already running jobs), so in general the search is limited by the number of nodes along a dimension in the CAN. Therefore, we can see from Figure 5.5 that the number of hops for CID is approximately two times greater than that for sub-CAN in the worst case. However, for 90% of the matches the number of hops is less than 40, so we can conclude that CID approach is comparable in terms of the matchmaking cost in practice. Moreover, the matchmaking time is very small compared to the job wait time and the job running time, so will not greatly increase the total job turn-around time in practice.

5.2.3 Exact Match Workload

If there is no range specified in a job's requirements (meaning an exact match is needed), and there are a large number of unique combinations of categorical resources types, then the job can be directed to a single sub-CAN for that approach, whereas the CID approach does not show any benefits from its integrated approach. Therefore in a situation like exact matches with many sub-CANs, the load balancing performance of CID may be worse than for the sub-CAN approach. In addition, the number of hops for matchmaking increases with the fraction of capable nodes in the CAN in terms of categorical values. That is because there is a lower probability of encountering capable nodes among neighbors during the job pushing process in matchmaking, since neighbors can have different values for their categorical resources in the CID approach. Therefore, we examine the most difficult scenario for

the CID approach and compare the performance and cost of the CID and sub-CAN approaches.

In this experiment, only exact-match jobs are submitted to the system with equal probability for having each combination of categorical resource values (other configuration and job workload are the same as those for the previous experiment). Overall, the performance of CID, sub-CAN, and Cent does not look significantly different, although Cent and sub-CAN perform slightly better than the CID approach. We conducted the simulations varying inter-arrival times and jobs constraints ratio, but the results are similar, so we omit the graphs. In terms of routing hops for matchmaking, CID has many more routing hops for matchmaking, because the number of node in the entire CAN is much greater than in a single sub-CAN case. Even with this effect, the median values for the number of hops are not very different, and in the worst case the number of routing hops is bounded by the system size. Furthermore, the matchmaking time is always very small compared to the job wait time and the job running time, so the matchmaking cost does not contribute much to overall job turn-around time.

From these results, we conclude that even under unfriendly conditions, the CID approach performs competitively to the online centralized matchmaker and to the sub-CAN approach, which has been optimized to perform exact matches for categorical resource types, and that the overall costs remain reasonable.

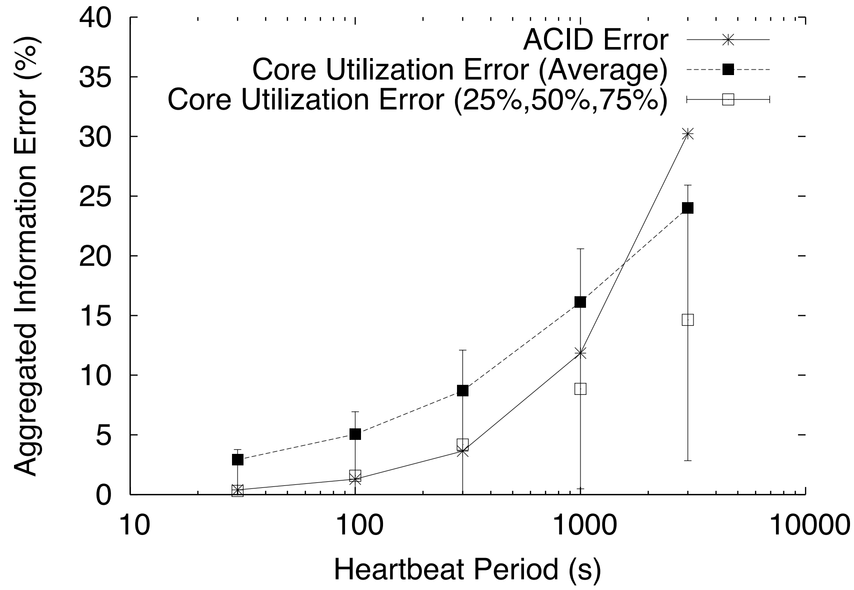
5.2.4 Stale Information

Our job pushing algorithm based on CID depends on aggregated information in two ways: First, ACID information should be propagated in a downward direction toward the origin of the CAN so the CAN can use it to find capable nodes, for matchmaking for categorical resources. Propagated ACID information cannot be sent to all nodes in the system instantly. It takes time to update any changes in categorical resources among non-neighbor nodes (e.g, because nodes arrive or leave the system) because information updates are piggybacked onto the periodical heartbeats between neighbors used to maintain CAN connectivity. Therefore, the delay in ACID information propagation may cause a failure in matchmaking even when a capable node exists in the system, thereby causing loss of the *completeness* property Second, load information, such as the number of nodes, the number of cores in the nodes and the number of jobs in the nodes can be aggregated and propagated along each dimension in the CAN to give some hints about the load distributions across nodes in the CAN. Such aggregated load information can impact the performance of load balancing. Therefore, staleness of aggregated information can affect matchmaking performance both in terms of completeness as well as for load balancing.

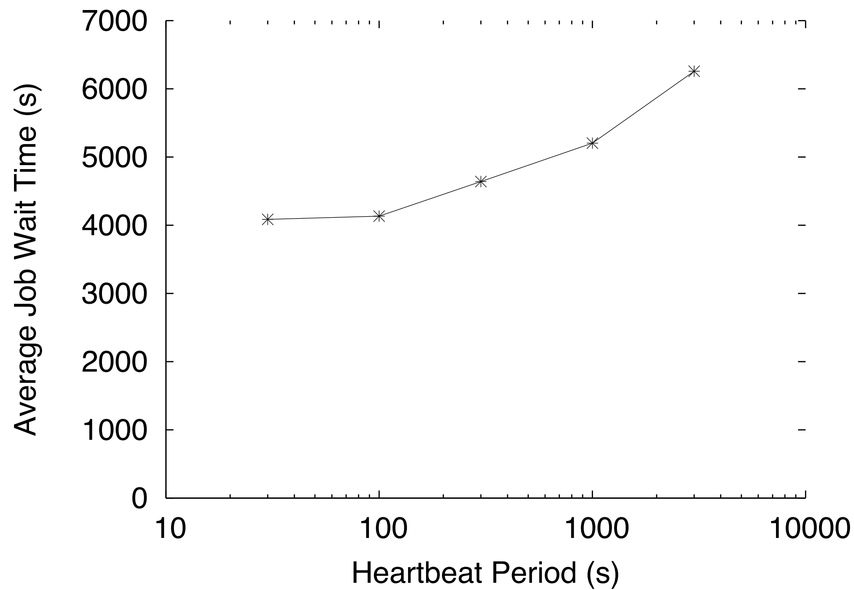
Staleness of the aggregated information is determined by the heartbeat interval, so we conducted experiments changing the heartbeat interval and used two metrics to evaluate performance. As a direct measure to see how stale the aggregated information is, we compute the accuracy of the aggregated information as

follows. We capture a snapshot of the entire system every five minutes during the simulation and compute the correct ACID and load information for each node based on the up-to-date information in the snapshot. We then compare the correct information from the snapshot with the current aggregated information at each node at some point in time, and compute how many bits in the ACID bit string in each node are wrong (for ACID), or how far the current load information is from the correct value (for load information). The correct values in the snapshot are used only for checking the accuracy of the aggregated information, not for matchmaking jobs. The second metric for performance is an indirect effect of staleness of aggregated information. We measured the job wait time distributions with varying heartbeat periods to see the effects of overall system performance as the staleness of the aggregated information varied.

Setup The experiments are configured as follows. Initially 1000 nodes with 4 different combinations of categorical resource join the system. The node distribution for the categorical resource types is uniform (25 % for each combination of categorical resource types). In the second stage, 1000 Jobs are submitted sequentially with a Poisson distribution for job inter-arrival time (10 seconds on average), and the average running time of submitted jobs is 120 minutes. Concurrently a new node joins or one of the nodes leaves the system in intervals that also follow a Poisson distribution with either a 5 or 10 second average between events (called *node churn rate*). The node churn rate is faster than or equal to the job inter-arrival rate in this simulation so that each job can do matchmaking based on some stale information



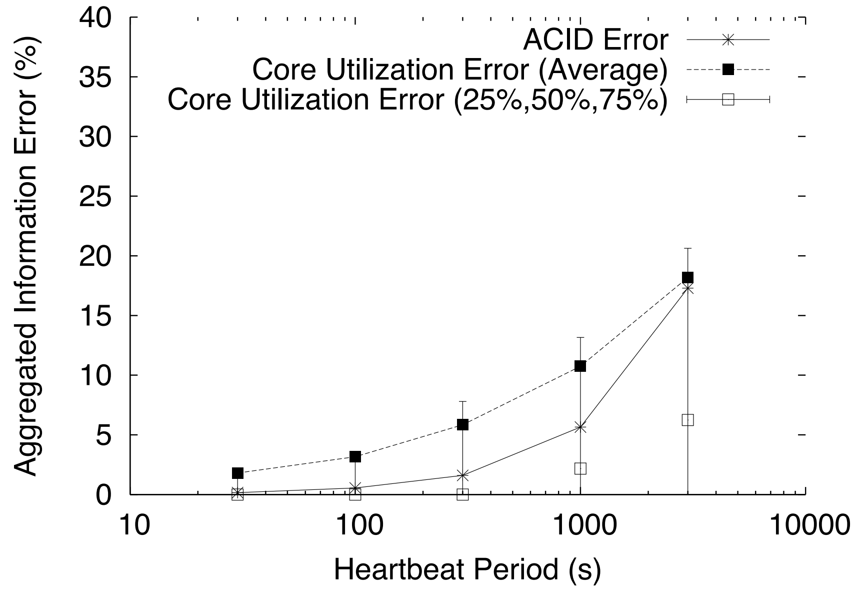
(a) Error of Core Utilization and Correctness of ACID



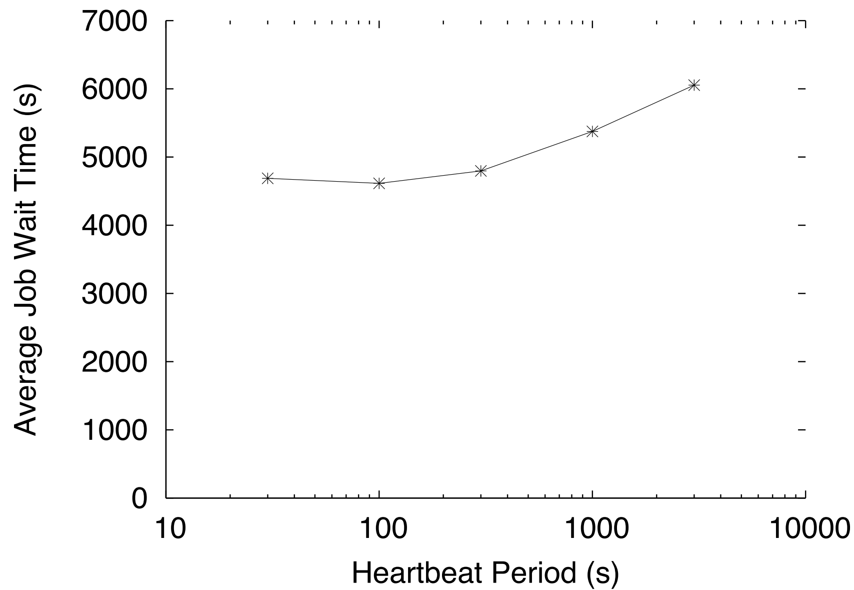
(b) Averages of Job Wait Time

Figure 5.6: Information Accuracy and System Performance Varying Heartbeat Periods: The Node Churn Rate is 5 Seconds

for nodes in the system. All jobs (or nodes) in the second stage specify particular categorical resource requirements (or capabilities), which are chosen from the four combinations used in the first stage. Therefore, only 25% of nodes that join in the



(a) Error of Core Utilization and Correctness of ACID



(b) Averages of Job Wait Time

Figure 5.7: Information Accuracy and System Performance Varying Heartbeat Periods: The Node Churn Rate is 10 Seconds

initial stage can execute the incoming jobs and 75% of nodes are only exchanging heartbeat messages. The ratio of nodes having different categorical values can increase the information propagation delay among actual capable nodes, so having a

large number of different categorical values for the nodes in the system makes it easier to see the effects of increasing the heartbeat interval. The purpose of this experiment is to see the effects from information staleness for long heartbeat intervals, so we fixed this ratio for this experiment. We increased the heartbeat interval from 30 seconds (the default value used in earlier experiments on load balancing) to 300, 1000, and 3000 seconds and measured the correctness of ACID and the accuracy of (aggregated) core utilization of the upper region for periodic snapshots, and job wait time distributions for all jobs submitted in the second stage.

Results Figure 5.6 shows the accuracy of the core utilization and the correctness of the CID, as well as the average job wait time as the heartbeat interval varies, when the node churn rate is 5 seconds. In the figure, *ACID error* is the number of incorrect ACIDs divided by the total number of ACIDs in the system, and *Core Utilization Error* is defined as the absolute difference between the correct core utilization and the actual core utilization in each node. Core utilization is the number of used (and required) cores of running (and waiting) job(s) divided by the number of cores in the nodes in the upper region in the CAN. We plot ACID errors and statistics of core utilization, including mean, median, 25% and 75% values, varying the heartbeat interval. Note that the X axis (heartbeat period) in the figure is a log scale for a better view of the points. The graph shows that more than 90% of the ACID information is correct and that the average error of core utilization is less than 10% when the heartbeat period is reasonably short (i.e. less than 5 minutes), and overall system performance, as measured by job wait time, does not decrease

significantly. However, when the heartbeat period becomes longer (i.e. 50 minutes), the error ratio is relatively higher, and performance decreases (job wait time increases), although 50 minutes for a heartbeat interval is relatively long compared to the average job running time of 120 minutes. We conclude that the CID approach is not very sensitive to the staleness of the aggregated information in practice, and overall performance is not affected by the small errors from delayed information propagation, as long as the update interval is not too long (e.g., less than 5 minutes). Figure 5.7 shows the same results when the interval for the node churn rate is 10 seconds on average, which is a lower churn rate than for the previous experiment. Though the correctness of ACID and the accuracy of core utilization are improved because of the lower node churn rate, the overall trend of the results is the same as for the higher churn system.

5.3 Summary

We have described and evaluated a new approach to decentralized matchmaking of jobs that include multi-attribute, range-type categorical resource constraints in their job descriptions. We encode these range constraints in job descriptions into *categorical identifiers* (CIDs), and match these CIDs with resource information to find good matches. We show via simulation that our scheme supports range-type search on job descriptions and balances loads effectively across heterogeneous nodes.

Moreover, our system relies on resource descriptions being aggregated and periodically disseminated across a P2P overlay network. Decentralized algorithms

are often sensitive to the freshness of the aggregated information. We show that our approach is tolerant of relatively stale resource information, allowing the system to aggregate and disseminate information only at large intervals. Further, we show that this insensitivity persists even under situations of high node churn.

Chapter 6

Related Work

This chapter introduces related research on P2P grid technology to exploit multi-core or heterogeneous resources and to support expressive resource description and discovery on various P2P platforms.

P2P Grid System

There is a great deal of previous research on grid computing based on peer-to-peer architectures, but these do not necessarily share all of our goals. For example, unstructured P2P frameworks [45, 46, 47] employ *Time-to-Live* message timeouts, so they cannot have our desired completeness property because they may fail to find a node capable of running a job even though one exists in the grid. There have also been studies on encoding resource information using a DHT hash function for resource discovery [48, 49, 50]. However this line of research has problems with load-balancing and expressiveness, because the hash function cannot distribute similar nodes evenly across the system. Moreover, these systems do not take multi-core capabilities into account. Even centralized systems such as Condor [4] and BOINC [5], suggest treating each core as a separate entity in the grid, handle a multi-core node as a collection of virtual independent nodes.

Performance Prediction Model for Multi-core Nodes

For simulating multi-core nodes, Amdahl's Law [51, 52] for multi-core machines is

proposed in [53]. Amdahl's Law and other speedup models for multi-core machines are discussed for scalable computing [54], However, using these laws requires knowing characteristics of a specific job, such as the fraction that is parallelizable and its sequential execution performance, so they cannot be applied to our high-level simulations for desktop grid computing. In addition, these laws do not include performance degradations due to memory or other resource contention.

Scalable and Robust P2P Overlay System

There has been a great deal of work on robust and scalable structured peer-to-peer systems. For example, Gummadi et al. described the relationship between failure resilience and the geometric shape of various DHTs [55]. While they conclude that ring geometry is the most robust, the ring shape cannot support our required semantics for resource representation. Chun et al. showed that smart selection of neighbors can improve the performance and robustness of a DHT containing heterogeneous nodes [56]. They used a cost function that takes into account network proximity and node capacity to choose the best neighbors. However, they did not consider scalability in heterogeneous environments. Awerbuch and Scheideler provided a theoretical foundation for robustness and scalability of DHTs [57]. They developed a generalized model, analyzed its theoretical properties and evaluated the model in a high-churn environment. Their proposed scheme is robust so can deal with large numbers of join-leave events in a short period of time, but they did not describe the detailed protocols that are needed for a practical system.

GPU, Heterogeneous nodes and Grid Computing

There have been some efforts to exploit heterogeneous machines, especially GPGPUs, in desktop grid computing environments. For example, the BOINC system has begun to support GPGPU computing so that users can run scientific applications on a GPU platform [58]. One practical project to exploit desktop GPUs is GPUGRID.net [59]. This project intends to solve molecular simulations on top of BOINC. However, the project mainly targets specific GPU machines, not more heterogeneous resources, and its scheduling and load balancing algorithms are centralized, which is different from our purely decentralized approach. Perhaps the closest work to ours on scheduling and resource management for heterogeneous environment was done by Kotani et al. [60]. They focused on how to detect and exploit idle cycles in GPU machines and proposed a simple matchmaking framework. However, the framework depends on a central resource broker, which is very different from our completely decentralized approach.

Multi-attribute Range-query using Indexing

There has been some prior work on range queries into multi-dimensional datasets over peer-to-peer systems. One popular approach is to construct a distributed indexing overlay over the nodes in the P2P network for efficient data lookup. Earlier projects include SCRAP and MURK [61]. SCRAP uses a space-filling curve to integrate multi-dimensional data into a single index while preserving locality. However, the locality of the space-filling curve cannot be guaranteed for high-dimensional data. MURK is a variant of a distributed, multi-dimensional KD-tree (like our CAN), but its indexing scheme is also not optimized for high-dimensional data. Inspired by

MURK, DiST [62] supports optimized range queries by building a complete KD-tree index composed of partial, distributed indexes across P2P nodes. However, DiST targets stable and less dynamic systems, such as distributed databases for fast queries into spatial data. Similarly, other efforts to extend well-known multi-dimensional trees for decentralized environments have been proposed to support efficient range queries, such as RT-CAN (R-tree over CAN) [63] and a distributed quad-tree index [64]. VBI-tree [65] is a more general peer-to-peer framework for multi-dimensional indexing. That framework suggests distributing a virtual multi-dimensional binary tree across P2P nodes by adding more routing links between sibling nodes in the tree. Even though an additional overlay can support efficient range queries across distributed nodes, maintaining the DHT and decentralized indexing concurrently can be expensive for P2P systems, especially in high churn situations.

Multi-attribute Range-query on P2P Overlay System

Another approach for multi-dimensional range queries on a P2P system is to leverage generic DHTs by compressing index information into the key used by the DHT. For example, m-LIGHT [66] indexes a spatial KD-tree and transforms the index into a key using a naming function. PRISM [67] uses reference vectors to encode a multi-dimensional index over a DHT. Similarly, Li et al. propose to support multi-dimensional range queries in sensor networks by mapping a high-dimensional space to a 2-dimensional space [68]. Both systems partition a virtual multi-dimensional space, label each hyperspace with a bit-string and map (compress) bit-strings to a

1 or 2 dimensional space so that a multi-dimensional query can search in the space efficiently. One simple multi-attribute search scheme embedded in a DHT is Mercury [69]. Mercury builds individual indexes over a Chord DHT [15] to support queries for each different attribute. The system uses the key space of the Chord to represent each resource characteristic, similar to our CAN-based system, and the authors suggest load balancing techniques to resolve a skewed data population caused by the non-random key space. However, their scheme to build separate indexes may not be efficient for range queries into high-dimensional data, because their target workload is to find entities in a 3-D space, and they cannot support wild-card type range-queries efficiently. SWORD [70] uses Mercury's idea for its underlying distributed architecture, and support a set of optimized techniques and services for users for wide-area multi-attribute resource discovery, but SWORD is inefficient for wild-card queries on a high-dimensional space. One recent effort for elastic cloud storage is ecStore [71]; it supports efficient range-queries and transactional operations by leveraging a peer-to-peer structure to build a scalable and reliable system. We generalize ecStore's elastic range-query functions for the storage system to support flexible resource discovery. Even though there has been work to support multi-dimensional range queries on P2P systems, they all lack at least one key feature from our system requirements. Our proposed scheme finds the best (least loaded) node among all candidates that meet the constraints, whereas the schemes from the related work search nodes that meet the range constraints but do not address load balancing issues.

Parallel Job Scheduling under Multiple Resource Requirements Though parallel job scheduling schemes have different assumptions and usage models, some of underlying techniques are shared with our P2P based matchmaking for heterogeneous environments. Backfilling [72, 73] is a commonly used scheduling method for parallel jobs, because it is straightforward but has been shown to be more effective than a first-come, first-serve(FCFS) scheduler. Backfilling schemes require the job running time, which is given by the user or estimated, and inaccurate estimation is closely related to scheduling performance [74]. However, this assumption is not applicable to our heterogeneous decentralized desktop grid, where good estimates of job running times may be very difficult to acquire.

While most previous research takes only CPU utilization into account, Leinberger et al. suggest a backfilling scheme within a single machine that allows for multiple resource requirements, such as CPU and memory [38]. That work proposed two backfilling techniques for selecting backfilled jobs, to maximize total utilization as well as to balance utilization across resources. However, those are still based on the EASY backfilling criterion, which requires accurate information about job running times, therefore we cannot apply those techniques in a straightforward manner. They also proposed a load balancing scheme among nodes via job migration in computational grids with multiple resource constraints [37]. As they did for a single machine [38], they tried to balance loads locally across K-resources by exchanging jobs with different resource requirements among machines to enhance throughput. However, they assumed a near-homogeneous environment, and did not consider backfilling in that work.

Chapter 7

Conclusions and Future Work

In this chapter, I conclude this dissertation by reviewing the thesis and its contributions and present some directions for future work.

7.1 Thesis and Contributions

In this dissertation, I supported the following thesis: *a decentralized resource management scheme can be employed to exploit heterogeneous multiple computing resources in grid systems*. The goal of this work was to investigate the problem of building a scalable infrastructure for exploiting multi-core and various kinds of heterogeneous grid resources in an efficient way. Such infrastructure must be decentralized, robust, scalable, and expressive while efficiently allocating application instances to available resources throughout the system. The main contributions made by this dissertation include:

1. **Effective contention-aware job scheduling for multi-core nodes** First, I have proposed two new dynamic resource management schemes, Dual-CAN and the Balloon Model, which account for multi-core nodes in a peer-to-peer grid. The key idea behind both schemes is to use distinct logical peers to represent a single physical multi-core node. One logical peer represents the static and maximum node capability, and the other expresses the current avail-

able amount of resources. I have developed a scheme for efficiently mapping jobs to multi-core nodes within the new resource management frameworks. I extended our existing single-core approaches to aggregating information and representing global grid state, and we described a "job-pushing" algorithm that efficiently balances load in both single-core and multi-core machines.

Second, I suggested an approach to modeling the performance of multi-core nodes, using a *penalty factor* to account for resource contention. Based on data from real experiments using benchmark tests and from the literature, we developed a simple equation to predict the contention effect by interpolating all data. Via extensive simulation results, I show that our new approach outperforms multi-core oblivious approaches, and shows performance competitive to an online centralized algorithm.

2. **Scalable resource management framework for heterogeneous environments**

   I proposed a decentralized resource management scheme that exploits diverse computing elements in heterogeneous environments. By considering features of heterogeneous nodes, i.e., differing numbers of computing elements as well as diversity of computing element types, our matchmaking and load balancing solution is better optimized to accommodate various computing elements across nodes with different performance characteristics and capabilities. We have confirmed via extensive simulations that our proposed scheme shows load balancing performance competitive to an online centralized approach, and bet-

ter than our previous scheme that ignored heterogeneity.

However, supporting heterogeneous jobs and nodes in a system where resources are mapped to dimensionality can cause the overall system to scale poorly. We have analyzed the system costs required to maintain the underlying CAN DHT with respect to the complexity of the job resource requirements, and found that the messaging cost is $O(d^2)$ in the number of dimensions for the prior system. We have described more scalable solutions to reduce the costs to $O(d)$ without sacrificing system resilience to node failures, and have confirmed these properties via simulation.

3. **Range-type resource discovery and load balancing**

   I developed a new resource discovery and load balancing method to support multi-attribute, range-based job constraints in a peer-to-peer grid system. Handling these jobs is difficult because decentralized resource discovery and load balancing across multiple sets of resource constraints become very complex, but in a structured way. Our scheme encodes the categorical resource types of each node into a standard set representation, such as a bit string, and propagates that information across nodes in the system with fixed length messages. We combine this encoding scheme with the existing job pushing method for continuous resource types, to provide a flexible, integrated matchmaking solution. We have used simulation to show that our approach has load balancing performance competitive to an online centralized approach, and effectively supports range-type job constraints without increasing overall costs

significantly. Even in difficult environments, our approach efficiently performs range queries with load balance comparable to the more complicated sub-CAN approach, without significant overhead.

Moreover, our matchmaking algorithm relies on asynchronous information aggregation and dissemination among nodes, leading to the possibility of basing decisions on stale information. We have used simulation to show that the performance of our new algorithm is resilient to delays in propagating aggregated information, even with high node churn and infrequent updates.

7.2 Future Work

We foresee many possible extensions to the work presented in this dissertation. Although I have focused on performance improvement and optimization to exploit heterogeneous grid resources, there are many opportunities which have not been explored from various perspectives.

Real System Experiments

So far, we have tested our decentralized matchmaking framework in a simulator. Simulation allows us to see the results under various environments, so it is easy to show the behaviors of our system. Our research group has made some efforts to develop our ideas for real implementation so that we can run real workloads provided by our collaborators in Astronomy department in the University of Maryland. Once we can move our simulator to the real system, we can investigate more practical problems which could not be shown in a simulation environment.

Energy Efficient Resource Management and Job Scheduling

In a data-center environment, the cost to pay electricity bills becomes significant concern compared to the cost to maintain facility and purchase new hardware. Therefore, energy-efficient resource management and job scheduling are hot topics in a current cloud computing area. Grid computing should support this trend to reduce costs of computing resources so that people can voluntarily donate their idle computing cycles without worrying about electricity costs. Our job scheduling decision has been optimized to maximize total system throughput and/or CPU utilization in each node. In addition to focusing on utilization of computing resources, we can add some parameters related to energy consumption in the objective function in the matchmaking algorithm.

Running Dependent Jobs and Checkpointing

In my thesis work, I have assumed that there are no dependencies between jobs. However, if a job needs the output of another job as its input, then we have to consider how to organize those dependent jobs in an organized way. For example, there should be a coordinator such as Condor's DAGMan [75] which can manage and monitor the overall execution flows of dependent jobs. In addition, if the size of the first job's output is large, then we should allocate dependent jobs considering network proximity and bandwidth.

In our current formulation of the problem, if a run-node fails suddenly, then we have to restart the job execution from the scratch. However, if a run-node checkpoints the interim result or job's execution status periodically, the job can

execute from the recent checkpoint time [4]. By adding checkpointing feature, our P2P grid system can enhance its robustness.

Bibliography

[1] S.K. Moore. Multicore is bad news for supercomputers. *IEEE Spectrum*, 45(11):15–15, November 2008.

[2] CUDA. Available at *http://www.nvidia.com/CUDA*.

[3] Richard Linderman. Architectural Considerations for a 500 TFLOPS Heterogeneous HPC. In *2010 19th International Heterogeneity in Computing Workshop*, April 2010.

[4] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[5] David Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, November 2004.

[6] David P. Anderson, Carl Christensen, and Bruce Allen. Designing a Runtime System for Volunteer Computing. In *Proceedings of the 2006 IEEE/ACM SC06 Conference*, November 2006.

[7] Folding@Home. Available at http://folding.stanford.edu.

[8] Vijay S. Pande, Ian Baker, Jarrod Chapman, Sidney P. Elmer, Siraj Khaliq, Stefan M. Larson, Young Min Rhee, Michael R. Shirts, Christopher D. Snow, Eric J. Sorin, and Bojan Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, 2003.

[9] B. P. Abbott and et al. Einstein@home search for periodic gravitational waves in early s5 ligo data. *Phys. Rev. D*, 80:042003, Aug 2009.

[10] OpenCL. Available at *http://www.khronos.org/opencl/*.

[11] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM Conference*, August 2001.

[12] Michael J. Freedman, Eric Freudenthal, and David Mazi. Democratizing content publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'2004)*, March 2004.

[13] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[14] Antony Rowstran and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[15] 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 Conference*, August 2001.

[16] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.

[17] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freud. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2), November 1999.

[18] Alexandru Iosup, Catalin Dumitrescu, and Dick Epema. How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing - GRID 2006*, September 2006.

[19] Jik-Soo Kim, Beomseok Nam, Peter Keleher, Michael Marsh, Bobby Bhattacharjee, and Alan Sussman. Resource Discovery Techniques in Distributed Desktop Grid Environments. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing - GRID 2006*, September 2006.

[20] Jik-Soo Kim, Beomseok Nam, Michael Marsh, Peter Keleher, Bobby Bhattacharjee, Derek Richardson, Dennis Wellnitz, and Alan Sussman. Creating a Robust Desktop Grid using Peer-to-Peer Services. In *Proceedings of the 2007 NSF Next Generation Software Workshop (NSFNGS 2007)*, March 2007.

[21] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.

[22] Nicholas Coleman, Rajesh Raman, Miron Livny, and Marvin Solomon. Distributed Policy Management and Comprehension with Classified Advertisements. Technical Report UW-CS-TR-1481, University of Wisconsin - Madison Computer Sciences Department, April 2003.

[23] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, July 1998.

[24] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[25] Jik-Soo Kim, Peter Keleher, Michael Marsh, Bobby Bhattacharjee, and Alan Sussman. Using Content-Addressable Networks for Load Balancing in Desktop Grids. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, June 2007.

[26] Jik-Soo Kim, Beomseok Nam, Michael Marsh, Peter Keleher, Bobby Bhattacharjee, and Alan Sussman. Integrating Categorical Resource Types into a P2P Desktop Grid System. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (GRID 2008)*, September 2008.

[27] Jonathan Lawder. Calculation of Mappings Between One and n-dimensional Values Using the Hilbert Space-filling Curve. Technical Report BBKCS-00-01, Birkbeck College, August 2000.

[28] Todd Tannenbaum. What's New in Condor? What's coming up? In *2008 EU Condor Week,*, October 2008. Available at http://www.oliba.uab.es/CondorWeek2008/.

[29] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Available at *http://www.cs.virginia.edu/stream/*.

[30] J. Weinberg and A. Snavely. Symbiotic space-sharing on sdsc's datastar system. In *The 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '06)*, St. Malo, France, June 2006.

[31] Sadaf R. Alam, Richard F. Barrett, Jeffery A. Kuehn, Philip C. Roth, and Jeffrey S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *Proceedings of the IEEE International Symposium on Workload Characterization 2006 (IISWC '06)*, October 2006.

[32] SPEC (Standard Performance Evaluation Corp. SPEC CPU 2006. Available at *http://www.spec.org*.

[33] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6, 2002.

[34] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.

[35] W. Magro, P. Peterson, and S. Shah. Hyper-threading technology: Impact on compute-intensive workloads. *Intel Technology Journal*, 6, 2002.

[36] Jaehwan Lee, P. Keleher, and A. Sussman. Decentralized dynamic scheduling across heterogeneous multi-core desktop grids. In *19th International Heterogeneity in Computing Workshop (HCW'10)*, pages 1 –9, april 2010.

[37] W. Leinberger, G. Karypis, V. Kumar, and R. Biswas. Load balancing across near-homogeneous multi-resource servers. In *Proceedings of the 9th Heterogeneous Computing Workshop, 2000. (HCW 2000)*, pages 60–71, 2000. Appears with the Proceedings of IPDPS 2000.

[38] William Leinberger, George Karypis, and Vipin Kumar. Job scheduling in the presence of multiple resource requirements. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 47, New York, NY, USA, 1999. ACM.

[39] Travis Desell, Anthony Waters, Malik Magdon-Ismail, Boleslaw K. Szymanski, Carlos A. Varela, Matthew Newby, Heidi Newberg, Andreas Przystawik, and David Anderson. Accelerating the milkyway@home volunteer computing project with gpus. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 276–288, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] Condor-how to manage GPUs. Available at *https://condor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToManageGpus*.

[41] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*, pages 63–74, New York, NY, USA, 2010. ACM.

[42] F. Guim, I. Rodero, J. Corbalan, and M. Parashar. Enabling GPU and Many-Core Systems in Heterogeneous HPC Environments Using Memory Considerations. In *Proceedings of 2010 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 146 –155, sept. 2010.

[43] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[44] Jaehwan Lee, P. Keleher, and A. Sussman. Supporting computing element heterogeneity in P2P grids. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 150 –158, sept. 2011.

[45] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, 33(4-5):275–288, 2007.

[46] Adriana Iamnitchi and Ian Foster. A Peer-to-Peer Approach to Resource Location in Grid Environments. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, pages 413–429. Kluwer Academic Publishers, 2004.

[47] Carlo Mastroianni, Domenico Talia, and Oreste Verta. A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis. In *Proceedings of the European Grid Conference (EGC)*, February 2005.

[48] Adeep S. Cheema, Moosa Muhammad, and Indranil Gupta. Peer-to-peer Discovery of Computational Resources for Grid Applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID 2005)*, November 2005.

[49] Rohit Gupta, Varun Sekhri, and Arun K. Somani. CompuP2P: An Architecture for Internet Computing using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1306–1320, November 2006.

[50] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[51] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[52] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[53] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[54] Xian-He Sun, Yong Chen, and Surendra Byna. Scalable Computing in the Multicore Era. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Programming*, September 2008.

[55] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM conference*, 2003.

[56] Byunggon Chun, Ben Y. Zhao, and John D. Kubiatowicz. Impact of neighbor selection on performance and resilience of structured P2P networks. In *Proceedings of 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[57] Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust DHT. *Theory of Computing Systems*, 45:234–260, 2009.

[58] Use your GPU for scientific computing (BOINC). Available at *http://boinc.berkeley.edu/gpu.php*.

[59] GPUGRID.net. Available at *http://www.gpugrid.net*.

[60] Y. Kotani, F. Ino, and K. Hagihara. A Resource Selection System for Cycle Stealing in GPU Grids. *Journal of Grid Computing*, 6:399–416, 2008.

[61] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One Torus Rule them All: Multi-dimensional Queries in P2P Systems. In *Proceedings of the 7th International Workshop on Web and Databases*, June 2004.

[62] Beomseok Nam and Alan Sussman. DiST: Fully decentralized indexing for querying distributed multidimensional datasets. In *Proceedings of IPDPS 2006*, 2006.

[63] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 591–602, New York, NY, USA, 2010. ACM.

[64] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16:165–178, April 2007.

[65] H.V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *Proceedings of the 22nd International Conference on Data Engineering, 2006 (ICDE '06)*, page 34, april 2006.

[66] Yuzhe Tang, Jianliang Xu, Shuigeng Zhou, and Wang chien Lee. m-LIGHT: Indexing multi-dimensional data over dhts. In *29th IEEE International Conference on Distributed Computing Systems, 2009 (ICDCS '09)*, pages 191 –198, june 2009.

[67] O. D. Sahin, A. Gulbeden, F. Emekci, D. Agrawal, and A. El Abbadi. Prism: indexing multi-dimensional data in P2P networks using reference vectors. In *In Proceedings of the 13th annual ACM international conference on Multimedia (MULTIMEDIA '05)*, pages 946–955. ACM Press, 2005.

[68] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 63–75, New York, NY, USA, 2003. ACM.

[69] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM Conference*, August 2004.

[70] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing, 2005 (HPDC-14)*, pages 113 – 124, july 2005.

[71] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *Proc. VLDB Endow.*, 3(1-2):506–514, September 2010.

[72] D. Feitelson and A. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proceedings of the International Parallel Processing Symposium (IPPS)*. IEEE Computer Society Press, 1998.

[73] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The EASY – LoadLeveler API project. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47, London, UK, 1996. Springer-Verlag.

[74] Pete Keleher, Dmitry Zotkin, and Dejan Perkovic. Attacking the bottlenecks in backfilling schedulers. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 3, 2000.

[75] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 11, pages 299–336. John Wiley, 2003.