Supporting Computing Element Heterogeneity in P2P Grids

Jaehwan Lee, Pete Keleher and Alan Sussman
UMIACS and Department of Computer Science
University of Maryland
College Park, MD, USA
{*jhlee, keleher, als*}*@cs.umd.edu*

***Abstract*—We propose resource discovery and load balancing techniques to accommodate computing nodes with many types of computing elements, such as multi-core CPUs and GPUs, in a peer-to-peer desktop grid architecture. Heterogeneous nodes can have multiple types of computing elements, and the performance and characteristics of each computing element can be very different. Our scheme takes into account these diverse aspects of heterogeneous nodes to maximize overall system throughput.**

However, straightforward methods of handling diverse computing elements that differ on many axes can result in high overheads, both in local state and in communication volume. We describe approaches that minimize messaging costs without sacrificing the failure resilience provided by an underlying peer-to-peer overlay network. Simulation results show that our scheme's load balancing performance is comparable to that of a centralized approach, that communication costs are reduced significantly compared to the existing system, and that failure resilience is not compromised.

I. Introduction

Modern desktop machines now rely on multi-core CPUs to provide high performance, though exploiting these cores effectively is still difficult even on a single machine, and even when the cores are homogeneous. The situation is growing more complicated as diverse heterogeneous hardware platforms (e.g., GPGPU (General Purpose computation on Graphics Processing Units) technology) have emerged and begun to impact desktop computing. For example, Nvidia's CUDA solution can achieve tremendous computing performance for iterative scientific computation with relatively low costs [1].

Grid computing must support these trends toward heterogeneity and diversity. In addition, resource management should be decentralized for a reliable and scalable system, because a centralized approach is vulnerable to a single point of failure and may create performance bottleneck. We have previously developed peer-to-peer (P2P) grid solutions for single-core and homogeneous multi-core machines [2]. However, ours and other grid research on multi-core environments does not efficiently exploit machines with heterogeneous computing elements.

The main focus of this paper is to accommodate heterogeneous nodes in a P2P grid, and we first address the target heterogeneous environment and system model. A node in a grid can have *multiple* computing elements (CEs), and the node can run multiple, independent multi-threaded applications (a *job*, in grid terminology) 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.

Our main contributions in this paper are two-fold. First, we describe a decentralized P2P system that makes good scheduling decisions in scenarios where both job requirements and nodes can contain multiple, possibly heterogeneous, computing elements. These scheduling decisions are made in the context of a distributed, decentralized desktop grid system. We are aware of no prior grid work that accommodates heterogeneous CE's to the extent described here.

Second, we show how to make these decentralized scheduling decisions *efficiently*. Directly extending prior work to handle diverse CEs can add greatly to the communication costs incurred by the underlying P2P system (a distributed hash table, or *DHT*). We describe a set of mechanisms that limit communication cost growth without sacrificing failure resilience, one of the key advantages of P2P systems.

The rest of the paper is organized as follows. Section II describes the basic architecture of our P2P desktop grid system. Section III discusses our decentralized resource management technique for heterogeneous environments. We present our approach to enable a scalable heterogeneous system in Section IV and show experimental results in Section V. We describe related work in Section VI. Finally we conclude in Section VII.

II. Background

A. Overall System Architecture

In prior work, we have developed a completely decentralized P2P desktop grid system that is both resilient to single-point failures and provides good scalability [3], [4], [2]. A desktop grid system may contain nodes with different

resource types and capabilities, e.g., CPU speed, memory size, disk space, number of cores. Jobs submitted to the grid can also have multiple resource requirements, limiting the set of nodes on which they can be run (called a job's *run node*). We assume that every job is independent, meaning that there is no communication between jobs. To build the P2P grid system, we employ a variant of a Content-Addressable Network (CAN) [5] DHT, which represents a node's resource capabilities and a job's resource requirements as coordinates in a d-dimensional space. Each dimension of the CAN represents the amount of that resource, so that nodes can be sorted according to the values for each resource. A node occupies a hyper-rectangular zone that does not overlap with any other node's zone, and the entire multi-dimensional space is covered by the zones for all nodes currently in the system. The zone for a node always contains the node's coordinates within the d-dimensional space. Nodes exchange load and other information in periodic heartbeat messages with nodes whose zones abut its own, called *neighbors*, to maintain the DHT structure. More details about the basic system architecture can be found in Kim et al. [4].

B. Matchmaking Procedure

Matchmaking is the initial assignment of a job to a node that satisfies all the resource requirements of the job, and also does load balancing to find a lightly loaded node. Basic matchmaking can be solved as a routing problem in our CAN, because every node in the CAN is sorted according to its resource capability along each dimension. Therefore, once the job is routed to its coordinate, all nodes with zones further from the origin than that point in the CAN will satisfy the job's requirements.

However, this basic matchmaking method can have load balancing issues. Our efforts to enhance load balancing are two-fold, employing a *virtual dimension* and using *probabilistic pushing* of jobs. The virtual dimension is a separate dimension in addition to the real resource dimensions that has a random value assigned to differentiate multiple nodes with the same capabilities. The random value in the virtual dimension also helps distribute jobs across nodes, so improves load balance. However, using the virtual dimension does not always achieve good load balance.

We have improved the basic matchmaking algorithm to improve load balance by *pushing* jobs into less loaded regions in the CAN in a probabilistic way. We aggregate global load information along each CAN dimension by piggybacking load data onto the heartbeat messages used to maintain connectivity in the CAN. After a job is routed to the node that minimally meets its resource requirements, that node chooses a dimension and a target node among its neighbors to find a path to a more lightly loaded region in the CAN. The decision process to push the job employs the periodically updated aggregate load information along each dimension. However, before pushing the job, the node

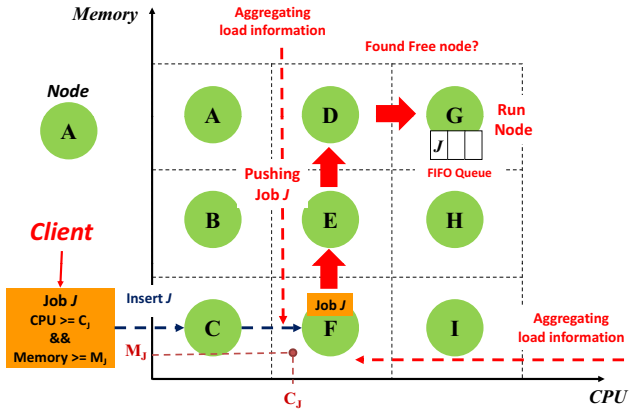


Figure 1. Job Pushing in the CAN

computes a stopping probability based on known load information in outer regions of the CAN to determine whether the job is to be pushed or not. If a job stops at a node, the node will pick the least loaded node among itself and its neighbors to run the job. Otherwise, the job continues to be pushed to a node with higher resource capability farther out in some dimension in the CAN. Once the run node is determined by job pushing, the job is inserted in the FIFO queue in the run node and waits for its execution to begin.

Figure 1 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 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 initial job placement can be found in Kim et al.[3].

III. Resource Management for Heterogeneity

In this section we present our new resource management framework, and the techniques that allow us to exploit multiple CEs with different performance characteristics. We

first describe how our CAN can be extended to express various types of heterogeneous resources, and then discuss additional mechanisms to deal with multiple resource types and asymmetric performance of CEs.

A. Accommodating Heterogeneous Nodes

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, available 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 (Clock Speed, GPU Memory, number of GPU cores) = 6, so the total number of CAN dimensions required is 11. 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 of the *multiplicity* of CEs in the heterogeneous nodes, as described in the following sections.

B. Job Pushing for a Heterogeneous System

As we described in Section II, *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 are busy running jobs, other CEs may be idle so may be able to begin another job's execution without delay. 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 simultaneously; 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 in Lee et. al. [2]. 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. We have conducted extensive experiments on contention effects between different CEs, such as CPUs and GPUs, and have found that there were no significant contention effects between separate CEs (those results are not shown because of space limitations). Our matchmaking algorithm takes those contention effects into account for heterogeneous systems.

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. We determine the dominant CE for a job based on the job resource requirements. If a job has requirements for multiple CEs, we pick the CE requiring the most of these other resources (e.g. memory, number of cores, etc.) as the job's dominant CE because the job needs more overall computational resources for that CE.

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 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 2). These score functions prefer the least utilized node for the dominant CE type, relative to its CE clock speed.

$$F(N,C) = \frac{CE(N,C).JobQueueSize}{CE(N,C).ClockSpeed} \quad (1)$$

$$F(N,C) = \frac{\frac{CE(N,C).RequiredCores}{CE(N,C).NumberOfCores}}{CE(N,C).ClockSpeed} \quad (2)$$

The complete algorithm for matchmaking and job pushing for heterogeneous environments is described in Algorithm 1. The equations in Algorithm 1 are as follows.

$$F_D(N,C) = \frac{AI_D(N,C).SumOfRequiredCores}{(AI_D(N,C).NumberOfCores)^2} \quad (3)$$

$$P(N) = 1/(1 + AI_{TD}(N).NumberOfNodes)^{SF} \quad (4)$$

In Equation 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, $P(N)$ is the probability to stop at node N, and SF is the stopping factor, which is a parameter used to adjust the stopping probability [3]. $AI_{TD}(N)$ 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.

Algorithm 1 Job Pushing for Heterogeneous jobs

```
Route the job in the CAN to the node containing the job's coordinate.
while run-node not found do
    Find an acceptable node(s) among neighbors.
    if Found an acceptable node(s) then
        if Found a free-node(s) among acceptable nodes then
            Pick the free-node with the fastest clock speed for the job's dominant CE.
        else
            Pick the acceptable node with the fastest clock speed for the job's dominant CE.
        end if
    else
        Choose a target node and dimension to minimize the objective function (Equation 3).
        Determine stopping based on the probability (Equation 4) for the target dimension.
        if Stop then
            Select the node with minimum score (Equation 1, 2) among neighbors.
        else
            Push the job to the target node.
        end if
    end if
end while
```

IV. Scalable System 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 balance 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.

A. Maintenance Cost Analysis

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

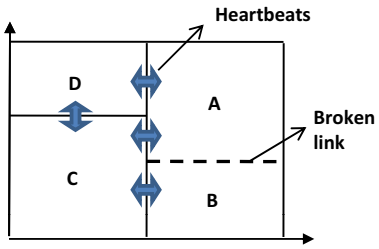


Figure 2. Recovery from a Broken Link via Heartbeats

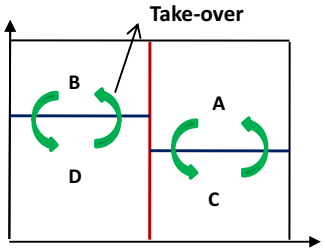


Figure 3. Zone Splits and Take-over Nodes

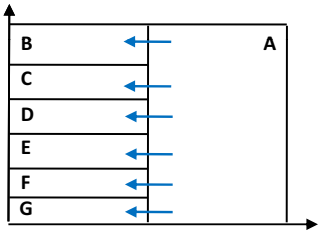


Figure 4. Worst Case for Compact Heartbeat

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 [5] 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 2. 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 2 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.

B. Compact Heartbeat

As was discussed in Section II, 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 3, 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. 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.

C. 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 the broken link happens to be to a take-over node. 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 2, if node A finds a broken link towards 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 node B's information. Therefore, adaptive heartbeat is as failure resilient as vanilla CAN in many cases, but the cost for adaptive heartbeat is nearly as low as for compact heartbeat.

V. 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.

A. Load Balancing Performance

Setup We used an event driven simulator that simulates the CAN construction, as well as matchmaking algorithms. We used a synthetic workload to model a typical grid resource configuration and a heterogeneous set of jobs. 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 like the example in Section III-A. Each node potentially has a single-/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 [6], if users desired to match on those resources.

Although a job may specify requirements for all 10 distinct resource types, any of them may be omitted (meaning any amount of that resource is acceptable). We define the *job constraint ratio* as the probability that each resource type for a job is specified for a given input stream of jobs. A higher job constraint ratio makes matchmaking more difficult, as highly-specified jobs are more difficult to match to nodes since fewer nodes will meet the specification. In addition, a high percentage of the nodes and jobs have relatively low resource capabilities and requirements, and a low percentage of the nodes and jobs have high resource capabilities and requirements, respectively, which is a common node capability distribution in grid environments.

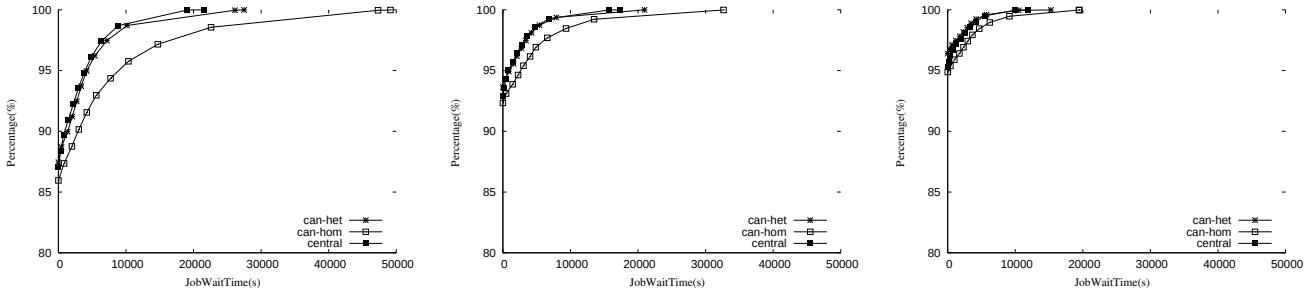
The interval between individual job submissions follows a Poisson distribution, and we vary the average inter-job arrival times in the experiments. Each job has an expected running time with an average value of 1 hour, uniformly distributed between 0.5 and 1.5 hours. However, the simulated job execution 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 (denoted as *central* in the graphs), which assigns jobs based on complete load information across all nodes. Such a scheme would be very expensive in a real system, but can give some indication of the best possible performance for our decentralized system. Note that the implementation of central does not necessarily represent the most globally efficient assignment, to provide a fair comparison to our online decentralized algorithms. Though it assumes perfect information, central greedily assigns a job to the most capable node, possibly assigning jobs to nodes that are over-provisioned.

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

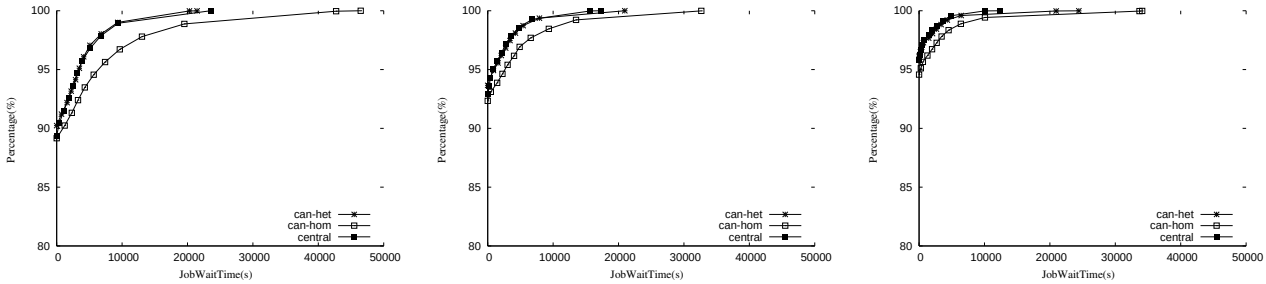
To avoid startup and cleanup anomalies, we run the simulations in a steady-state environment. Steady-state means that the job arrival and departure rates are similar, so that the system achieves a dynamic equilibrium state during the simulation period, with the system neither highly overloaded, nor very underloaded. Therefore, the inter-job arrival rate effectively determines average total system load.

Load Balancing Performance Figure 5 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



(a) 2 seconds (b) 3 seconds (c) 4 seconds

Figure 5. CDF of Job wait time varying Inter-arrival Time



(a) 80% (b) 60% (c) 40%

Figure 6. CDF of Job wait time varying Job Constraint Ratio

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 (denoted by *can-het*) and can-hom becomes larger. This means that can-hom cannot balance load very well when the system gets heavily loaded.

Figure 6 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.

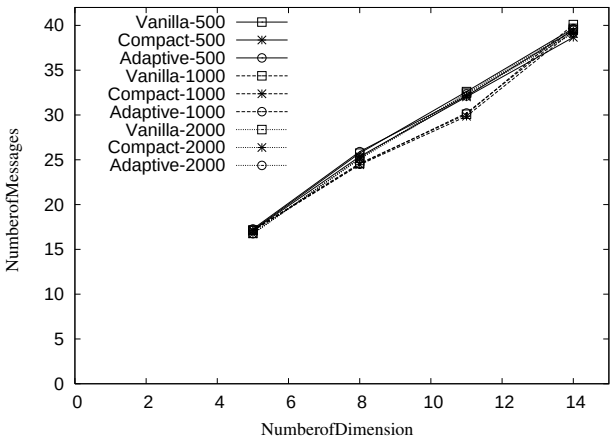
From these simulations, we confirm that our matchmaking and load balancing performance is competitive to the online centralized matchmaker, and better than the approach for homogeneous environments.

B. Scalability and Heterogeneous Resources

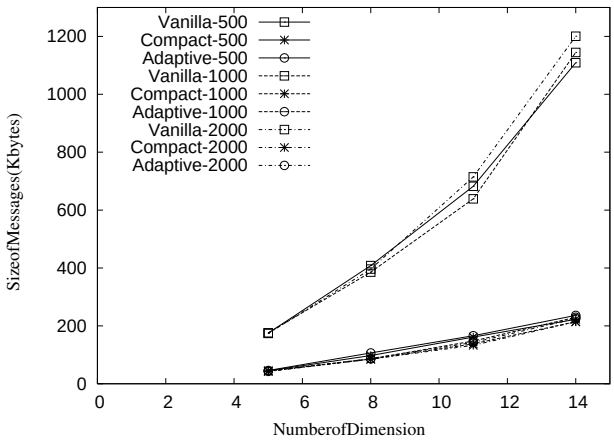
Setup To test the scalability and failure resilience of our algorithms for heterogeneous environments, we have experimented with 5, 8, 11 and 14 dimensional CANs with 500, 1000 and 2000 nodes, respectively. In the initial stage of each experiment, n nodes join the system sequentially. After that, node *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 First, none of the approaches suffers from broken links when there are no simultaneous events (failures). 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 7 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,



(a) Number of Messages



(b) Volume of Messages

Figure 8. Scalability, measured per node per minute

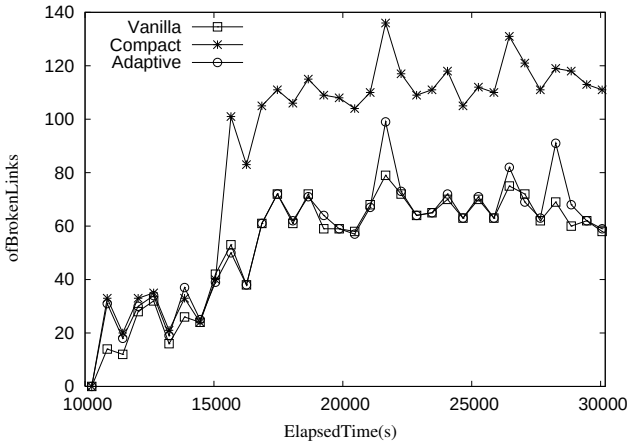


Figure 7. Broken Links under high churn

all three schemes appear to have reached steady-state behavior (the experiment continued past 30,000 seconds without qualitative changes).

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 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 IV, 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 8 shows the cost for varying numbers of nodes and CAN dimensions. Each subfigure shows how the number of messages or the volume of messages increases as the number of CAN dimensions increases. Note that the number of messages and the volume of messages in Figure 8 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 8(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 algorithms. 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 8(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.

VI. Related Work

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 [7]. 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 [8]. 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 [9]. 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.

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 [10]. One practical project to exploit desktop GPUs is GPUGRID.net [11]. 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. [12]. 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.

VII. Conclusion

In this paper, we have proposed a decentralized resource management scheme that exploits diverse computing elements in heterogeneous computing 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 CEs 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 better 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 extensive simulations.

We are currently implementing our decentralized matchmaking framework in a real testbed experiment to characterize its behavior and performance in cooperation with researchers from the Maryland Astronomy department.

References

[1] R. Linderman, "Architectural Considerations for a 500 TFLOPS Heterogeneous HPC," in *2010 19th International Heterogeneity in Computing Workshop*, Apr. 2010.

[2] J. Lee, P. Keleher, and A. Sussman, "Decentralized resource management for multi-core desktop grids," in *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium*, 2010.

[3] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman, "Using Content-Addressable Networks for Load Balancing in Desktop Grids," in *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Jun. 2007.

[4] J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman, "Resource Discovery Techniques in Distributed Desktop Grid Environments," in *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing - GRID 2006*, Sep. 2006.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," in *Proceedings of the ACM SIGCOMM Conference*, Aug. 2001.

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

[7] 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.

[8] B. Chun, B. Y. Zhao, and J. 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.

[9] B. Awerbuch and C. Scheideler, "Towards a scalable and robust DHT," *Theory of Computing Systems*, vol. 45, pp. 234–260, 2009.

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

[11] "GPUGRID.net," Available at *http://www.gpugrid.net*.

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