MS Scholarly Paper: Decentralized Resource Management for Multi-core Desktop Grids

Jaehwan Lee
Department of Computer Science, University of Maryland
jhlee@cs.umd.edu

Abstract

The majority of CPUs now sold contain multiple computing cores. However, current desktop grid computing systems either ignore the multiplicity of cores, or treat them as distinct, independent machines. The former approach ignores the resource contention present between cores in a single CPU, while the latter approach fails to take advantage of significant computing power.

We propose a decentralized resource management framework for exploiting multi-core nodes in peer-to-peer grids. We present two new load-balancing schemes that explicitly account for the resource sharing and contention of multiple cores, and propose a simple simulation model that can represent a continuum of resource sharing among cores of a CPU. We use simulation to confirm that our two algorithms match jobs with computing nodes efficiently, and balance load during the lifetime of the computing jobs.

1 Introduction

The majority of CPUs now sold contain multiple computing cores. However, even though multiple cores can often handle multiple tasks in parallel, overall performance with unoptimized software support can be poor. For example, Moore [13] 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. Given that it is not easy to utilize multi-core environments even on a single host, it is not surprising that there has been little research on integrating multi-core desktop machines into desktop grids. Both to effectively utilize all available grid resources, including cores, and to run jobs that request multiple cores (presumably because they are multi-threaded), we address the problem of resource management for multi-core grids.

There are many challenges in multi-core grids. 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 ideally 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. 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 for this situation.

In this paper, we make the following contributions. First, we propose two new dynamic resource management schemes, Dual-CAN and the Balloon Model, which account for multi-core machines in a peer-to-peer (P2P) 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 available amount of resources. Second, we develop a scheme for efficiently mapping jobs to multi-core nodes within the new resource management frameworks. We extend our existing single-core approaches to aggregating information and representing global grid state, and we describe a "job-pushing" algorithm that efficiently balances load in both single-core and multi-core machines. Third, we suggest a simple approach to modeling the performance of multi-core nodes, using a *penalty factor* to account for resource contention. Finally, we describe simulation results that show that our new approaches outperform multi-core oblivious approaches.

The rest of the paper is organized as follows. Section 2 describes the Content-Addressable Network (CAN) [15]-based desktop grid system within which we implement our new techniques. We present the distributed resource management scheme in Section 3. Section 4 presents the analytic model for concurrent jobs in multi-core nodes, and Section 5 shows our performance results. Section 6 summarizes related work, and we conclude in Section 7.

2 Background

2.1 Overall System Architecture

Our system is built on a variant of an existing distributed hash table (DHT). We use a CAN to maintain a peer-to-peer system in a structured way. The original CAN maps a node to a point in a d-dimensional space by hashing a GUID (Global Unique Identifier). The space is divided into hyper-rectangular zones that each maintains neighbor information. In our CAN, each dimension represents the amount of a particular resource type for a node, or that resource's requirement for a job, so that we can solve the matchmaking problem as a routing problem in the CAN.

A job in our system is composed of the data and a profile that describes how to compute the job result. The job description 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. This is a typical scenario for desktop grid computing systems. The following steps outline the procedure for injecting and executing a job in the system.

1. A client inserts a job into the system through an arbitrary node called the *injection node*.
2. The injection node 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. 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.
5. After finishing the job, the run node delivers the results to the client.

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 Kim et al. [9, 10]

2.2 Matchmaking Algorithm

Matchmaking is the process of assigning a job to a node that 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.

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.

The simple load balancing scheme based on random virtual dimension coordinates does not show 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 in the space [8]. 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 among those nodes that meet the job resource requirements. To provide the required load information in the decentralized system, we 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 (lightest load) 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. More details about our work on improved decentralized load balancing can be found in Kim et al. [8]

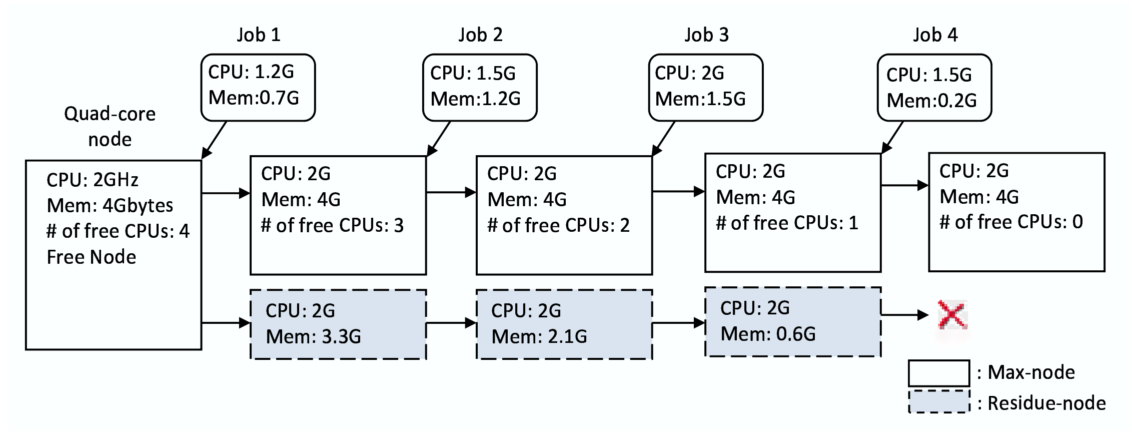


Figure 1. Changes for the two logical nodes when jobs are assigned to the node

3 Resource Management in a Multi-core Grid

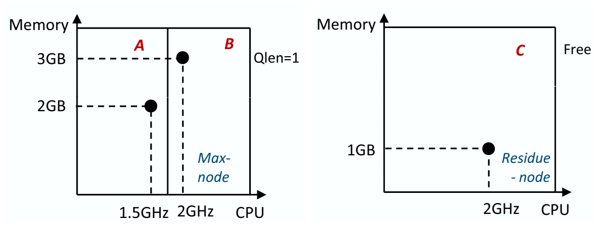
3.1 Two Logical Nodes for a Multi-core Node

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[19] 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 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. 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 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 2. Dual-CAN: Node B creates a Residue-node C on the Secondary CAN

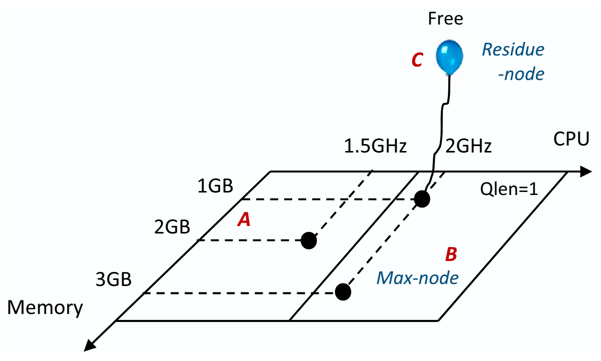


Figure 3. Balloon model: Node B is assigned the same job as in Figure 2

3.2 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 2(a) and 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.

3.3 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 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 shows an example for the Balloon model, where the status of the nodes is the same as in the Dual-CAN example in Section 3.2.

3.4 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 [8], 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 As we discussed earlier, 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 queue length and number of nodes, because those measures assume a single-core machine. 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 4 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:

$$TNC(N) = NC(N) + \sum_{b \in \mathbf{B}} \beta_b \cdot NC(b) \tag{1}$$

In Equation 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.

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

Algorithm 1 Pushing for Dual-CAN

```
1: Choose target node and dimension with minimum objective function(Equation 2).
2: Determine stopping based on the stopping probability(Equation 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 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
```

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.

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

In Equation 2, $F_d(u)$ is the objective function for the upper neighbor node 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:

$$P(N) = \frac{1}{(1 + AI_{TD}(N).NumberOfNodes)^{SF}} \tag{3}$$

In Equation 3, $P(N)$ is the probability to stop at Node N, and SF is the stopping factor, which is the parameter to adjust stopping probability pattern [8]. In addition, $AI_{TD}(N)$ is the aggregated information at Node N for upper (chosen) 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 nodes score function:

$$F(C) = \frac{C.RequiredCores/C.NumberOfCores}{C.SpeedOfCPU} \tag{4}$$

$F(C)$ 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.

3.5 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 cost is less than that of the Primary CAN because Secondary CAN membership is a subset of the Primary CAN members, but may not be negligible. 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.

4 Simulation Model for a Multi-core Node

4.1 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 shared 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 [17] to run two memory-intensive jobs on a dual-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-core Linux machine running a version 2.6.9 kernel, with an Intel Core2 Duo E6550 CPU running at 2.33GHz with 2GB memory. We compiled the STREAM source code with both gcc version 3.4.6 and Intel icc version 10.1. 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 dual-core machine and leave the other core idle. Second, we run two STREAM jobs at the same time, one on each core, to measure memory bandwidth and overall elapsed time. The results show that two extremely memory-intensive jobs on the dual-core node increase job running time by a factor of 2 in the worst case. For example, a STREAM *Copy* operation takes 39.2 ms on a single core with the other core idle, but 85.8 ms when 2 copies are run, one on each core. On average, the running time for the STREAM benchmarks running on two cores is 2.09 times longer than for running only on single core with *gcc*, and 2.04 times longer with *icc*. 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.

4.2 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. [1], 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 simulations with the SPEC CPU2006 integer benchmark suite [16] to measure the running time increase from contention in a dual core node. The same machine used for the STREAM benchmark is used for this simulation. The SPEC integer test suite was compiled with both *gcc* and *icc*, and we compared the running times for the entire suite for two cases, with one copy of the job on the dual core node and with two copies on each core. For *gcc*, the running time for the job increases by 6% on average while the running time increases by 10% with *icc*.

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

4.3 Our Simulation Model

From the experimental results in the previous section, we find that general scientific jobs using both cores in a dual-core node run 10% longer than when using only a single core, and that a memory intensive job contending with another job on a dual-core node may incur a penalty of up to a factor of 2 compared to running alone, in the worst case. Therefore, we have built a mathematical formulation for our multi-core simulation 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 10%. On the other hand, if a job is memory

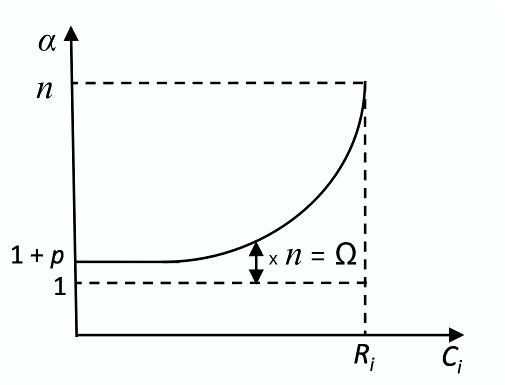


Figure 4. Expected running time ratio α with respect to shared resource usage

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 time can increase by n times where n is the number of concurrent jobs.

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 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, $\alpha = max_i\{n \cdot (\frac{C_i}{R_i})^m\}$ where m is a factor to adjust the worst case performance region. 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 5.

$$\alpha = max[1 + p, max_i\{n \cdot (\frac{C_i}{R_i})^m\}] \tag{5}$$

The *total penalty* Ω, the increased running time for all jobs because of running the new job, is shown in Equation 6.

$$\Omega = n \cdot \left(max[1 + p, max_i\{n \cdot (\frac{C_i}{R_i})^m\}] - 1 \right) \tag{6}$$

The discount factor β is equal to 1/(1+Ω). 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 4 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 n times that of running a job all by itself using a single core of the multi-core node.

5 Experiments

5.1 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.

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. Jobs in the workload have an additional characteristic, namely whether they are heavily or lightly constrained. More specifically, for each job the requirement for each resource can be constrained or not (meaning that any node can satisfy the requirement).

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, 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 4, 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.

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.

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[19]. We run one peer per core and

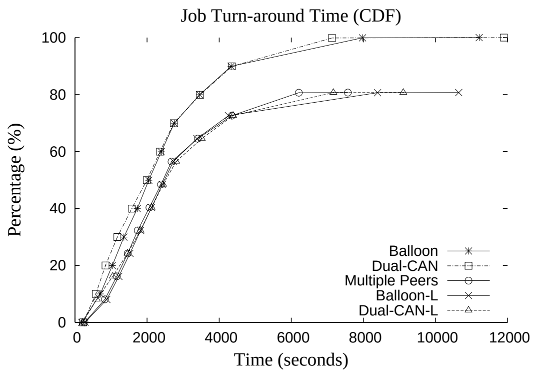
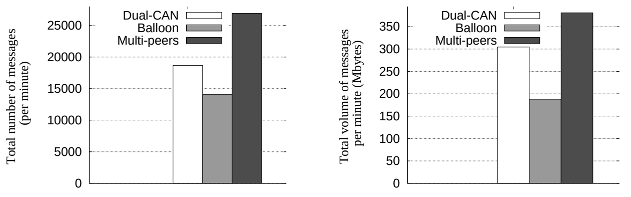


Figure 5. Cumulative distributions for Job Turn-around time



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

Figure 6. Costs of Dual-CAN, Balloon and MP

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.

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.

5.2 Experimental Results

Completeness The first experiment compares the Dual-CAN and Balloon models against the Multiple Peers (MP) scheme. Figure 5 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. We show only the result for clustered nodes and lightly constrained jobs, because it is the most common scenario in Desktop Grid. However, the results for other combinations are similar. As is seen in Figure 5, 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 5, 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 5, Balloon-L and Dual-CAN-L show competitive performance to MP, when measuring job turn-around time.

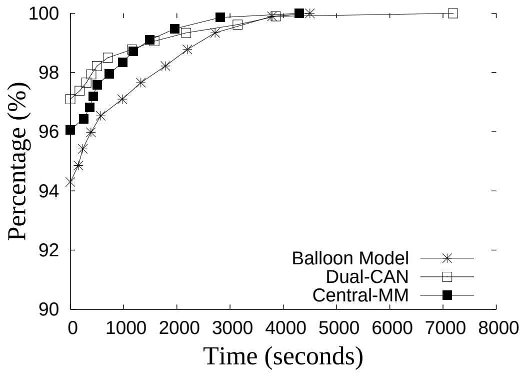
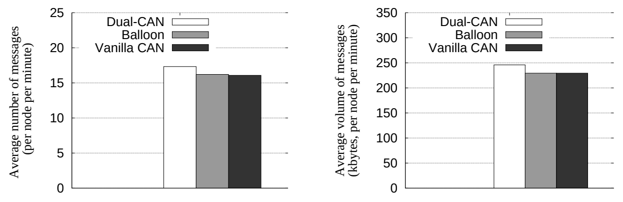


Figure 7. Cumulative Distributions of the Job Waiting Time



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

Figure 8. Costs for Dual-CAN, Balloon and Vanilla CAN

Figure 6 shows the total overhead from messaging across the entire system. The cost metrics are the number of messages and the total volume of messages. In Figure 6, 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 3.5, 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 in the rest of experiments.

Load Balancing Figure 7 shows load balancing performance for our two schemes, comparing against the centralized matchmaker. We experiment with eight 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 7 shows results for the clustered node, clustered and lightly constrained jobs scenario. Results for other scenarios are similar, but are not shown due to space limitations. 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 to a centralized matchmaker. More specifically, the Dual-CAN has more jobs with no waiting time than the other two schemes. The reason why Dual-CAN can perform better than the centralized matchmaker is that the centralized matchmaker runs a greedy algorithm so always selects the fastest node in the entire system if it is not busy, so that job are sometimes over-provisioned. Over-provisioning may increase wait times for jobs submitted later that have high resource requirements. Our decentralized algorithms are not greedy, but attempt to minimize over-provisioning and balance load via the pushing mechanism. Comparing our two algorithms, Dual-CAN achieves better performance than Balloon, as we discussed in Section 3.5

On the other hand, the overhead for the Dual-CAN is noticeably larger than for Balloon, as seen in Figure 8. The left graph in Figure 8 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% in the clustered node, clustered job scenario. There is therefore a trade-off in the cost and performance between the two schemes. Overall, Figure 7 and Figure 8 show that our two decentralized schemes both balance job load well, and do not add significant overhead.

6 Related Work

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 [3, 7, 12] employ *Time-to-Live* message timeouts, 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 [4, 5, 14]. 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 [11] and BOINC [2], suggest treating each core as a separate entity in the grid, handle a multi-core node as a collection of virtual independent nodes.

For simulating multi-core nodes, Amdahl's Law for multi-core machines is proposed in [6]. Amdahl's Law and other speedup models for multi-core machines are discussed for scalable computing [18], 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.

7 Conclusion

In this paper we have proposed 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 experiment 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 large and small resource jobs.

The work described here is limited to initial job placements in the desktop grid. Our future work will include active methods of explicitly tracking contention between multiple cores, and dynamic job-stealing methods that will remap jobs after the initial assignment. We are currently implementing the approaches described in this paper in our prototype peer-to-peer grid system, and plan to evaluate the approach with several large-scale scientific applications from our collaborators in Astronomy.

References

[1] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. 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)*, Oct. 2006.

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

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

[4] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer Discovery of Computational Resources for Grid Applications. In *Proceedings of GRID 2005*, Nov. 2005.

[5] R. Gupta, V. Sekhri, and A. K. Somani. CompuP2P: An Architecture for Internet Computing using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1306–1320, Nov. 2006.

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

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

[8] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using Content-Addressable Networks for Load Balancing in Desktop Grids. In *Proceedings of HPDC 2007*, June 2007.

[9] 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 GRID 2006*, Sept. 2006.

[10] J.-S. Kim, B. Nam, M. Marsh, P. Keleher, B. Bhattacharjee, D. Richardson, D. Wellnitz, and A. Sussman. Creating a Robust Desktop Grid using Peer-to-Peer Services. In *Proceedings of the 2007 NSF Next Generation Software Workshop (NSFNGS 2007)*, Mar. 2007.

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

[12] C. Mastroianni, D. Talia, and O. Verta. A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis. In *Proceedings of the European Grid Conference (EGC)*, Feb. 2005.

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

[14] 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 (HPDC-14)*, July 2005.

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

[16] SPEC CPU 2006. Available at http://www.spec.org.

[17] STREAM. Available at http://www.cs.virginia.edu/stream/.

[18] X.-H. Sun, Y. Chen, and S. Byna. Scalable Computing in the Multicore Era. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Programming* , Sept. 2008.

[19] T. Tannenbaum. What's New in Condor? What's coming up? In *2008 Condor Week,*, Apr. 2008. Available at http://www.cs.wisc.edu/condor/PCW2008.