Decentralized Dynamic Scheduling across Heterogeneous Multi-core Desktop 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*—The recent advent of multi-core computing environments increases both the heterogeneity and complexity of managing desktop grid resources, making efficient load balancing challenging even for a centralized manager. Even with good initial job assignments, dynamic scheduling is still needed to adapt to dynamic environments, as well as for applications whose running times are not known a priori.**

In this paper, we propose new decentralized scheduling schemes that backfill jobs locally and dynamically migrate waiting jobs across nodes to leverage residual resources, while guaranteeing bounded waiting times for all jobs. The methods attempt to maximize total throughput while balancing load across available grid resources. Experimental results via simulation show that our scheduling scheme has performance competitive with an online centralized scheduler.

I. Introduction

Modern desktop machines now use multi-core CPUs to enable improved performance. However, achieving high performance on multi-core machines without optimized software support is still difficult [1], because contention for shared resources can make it hard to exploit multiple computing resources efficiently. Moreover, desktop grids that contain multi-core machines are becoming increasingly diverse and heterogeneous, so that efficient load balancing for the overall system is becoming a very challenging problem even with global status information and a centralized scheduler.

Our previous research on decentralized resource management for desktop grids has developed and evaluated efficient initial job assignment algorithms for multi-core resources [2]. However, dynamic scheduling via migration of waiting jobs is still required for the best performance because 1) stale load information propagated between machines can lead to poor initial job assignments, 2) unpredictable job completion times can change the current load situation, and 3) initial job assignment is done in a probabilistic manner, and so can be improved with additional information.

The performance of distributed scheduling in such multi-core environments can be improved by starting waiting jobs immediately, through use of residual resources on other nodes (if the job is moved) or on the same node (if the local schedule is changed). However, efficient decentralized job migration can be difficult to achieve because of limited and/or stale global state. Moreover, a job profile often has *multiple* resource requirements; a simple job migration mechanism considering only CPU usage cannot be applied to in such situations. In addition, we would also like to guarantee progress for all jobs, i.e., no job starvation.

Our contribution in this paper is a novel dynamic scheduling scheme for multi-core desktop grids. The scheme includes (1) *local scheduling*, a form of backfilling on a single node, (2) *internode scheduling*, for backfilling across multiple nodes, and (3) *queue balancing*, which proactively balances wait queue lengths. Our approach is a completely decentralized scheme that balances load and improves throughput when scheduling jobs with multiple constraints across a distributed system. We demonstrate the effectiveness of our algorithms via simulations that show that the decentralized approach performs competitively with an online centralized scheduler.

The rest of this paper is structured as follows. Section II discusses related work on various parallel job scheduling and dynamic job migration techniques for desktop grids. Section III describes the basic architecture of our peer-to-peer desktop grid system and a decentralized resource management method for multi-core machines. We present our scheduling approach in Section IV, and show simulation results in Section V. We conclude in Section VI.

II. Related Work

Backfilling [3], [4] is a commonly used scheduling method for parallel jobs, because it is straightforward but has been shown to be more effective than a first-come, first-serve (FCFS) scheduler. Backfilling opportunistically reorders jobs in the scheduling queue when a large job (meaning one with high resource requirements) at the front of the queue is unable to run immediately. The general goal of backfilling algorithms is to allow a job to bypass jobs ahead of it in the queue to be able to exploit current residual resources, but also should not delay either any other jobs (called *conservative* backfilling in the literature), or only the first job in the queue (called *EASY* backfilling). Both backfilling schemes require the job running time, which is given by the user or estimated, and inaccurate estimation is closely related to scheduling performance [5]. However, this assumption is not applicable to our heterogeneous decentralized desktop

grid, where good estimates of job running times may be very difficult to acquire.

While most previous research takes only CPU utilization into account, Leinberger et al. suggest a backfilling scheme within a single machine that allows for multiple resource requirements, such as CPU and memory [6]. That work proposed two backfilling techniques for selecting backfilled jobs, to maximize total utilization as well as to balance utilization across resources. However, those are based on the EASY backfilling criterion, which requires accurate information about job running times, therefore we cannot apply those techniques in a straightforward manner.

Leinberger et al. also proposed a load balancing scheme via job migration in computational grids, and allowed multiple resource constraints [7]. As they did for a single machine [6], they tried to balance load locally across K-resources by exchanging jobs with different resource requirements among machines to enhance throughput. However, they assumed a near-homogeneous environment, and did not consider backfilling.

For dynamic job migration techniques, much work has been done on dynamic load distribution for distributed systems [8] and on thread migration in multiprocessor machines [9]. WaveGrid [10] is a peer-to-peer based desktop grid computing system that adopts a timezone-aware job migration technique. Once a job is assigned to a host that is in a night-time zone but busy, the job is migrated to another (presumably idle) host in the night-time zone [11]. However, WaveGrid does not allow specifying resource requirements for jobs, so it is a simpler model than for our desktop computing platform, to be described in Section III-A.

III. Background

A. Overall System Architecture

In prior work, we have developed a completely decentralized peer-to-peer (P2P) desktop grid system that is both resilient to single-point failures, and provides good scalability [12]. A desktop grid system may contain heterogeneous nodes with different resource types and capabilities, e.g. CPU speed, memory size, disk space, number of cores. Jobs submitted to the grid also can have multiple resource requirements, limiting the set of nodes on which they can be run. 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) [13] distributed hash table (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 zone contains the node's coordinates within the d-dimensional space. Nodes exchange load and other information with nodes whose zones abut its own (called *neighbors*). The following steps describe how jobs are submitted and executed in the grid system.

1) A client (user) 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 initiates the process to find a lightly loaded node (*run node*) that meets all of the job's resource requirements (called *matchmaking*)
4) The run node inserts the job into an internal FIFO queue for job execution. Periodic heartbeat messages between the run node and the owner node ensure that both are still alive. Missing multiple consecutive heartbeats invoke a (distributed) failure recovery procedure.
5) After the job completes, the run node delivers the results to the client and informs the owner node that the job has completed.

The owner node monitors a job's execution status until the job finishes and the result is delivered to the client. To enable failure recovery, the owner node and the run node periodically exchange soft-state heartbeat messages to detect node failures (or a graceful exit from the system). More details about the basic system architecture can be found in Kim et al. [12].

B. Matchmaking Procedure

Matchmaking is the initial job assignment to a node that satisfies all the resource requirements of the job, and also does load balancing to find a (relatively) lightly loaded node. A good matchmaking algorithm has several desirable properties: expressiveness, load balance, parsimony, completeness, and low overhead. The matchmaking framework should be *expressive* enough to specify the essential resource requirements of the job as well as the capabilities of the nodes. It should *balance load* across nodes to maximize total throughput and to obtain the lowest job turnaround time. However, over-provisioning can decrease total system throughput, therefore the matchmaking should be *parsimonious* so as not to waste resources. *Completeness* means that as long as the system contains a node that satisfies a job's requirements, the matchmaker should find that node to run the job. Finally, the overall matchmaking process should not incur significant costs, to minimize *overhead*.

Our CAN-based decentralized matchmaking framework directly supports expressiveness and completeness with low overhead. Our previous efforts to enhance load balancing performance but be parsimonious are two-fold - employing a *virtual dimension* and using *probabilistic pushing* of jobs. The basic CAN mechanisms do not allow the multiple nodes to have the same coordinates in the multidimensional space. However, the coordinates in our CAN are determined by the amount of each resource a node has, so multiple nodes with identical resource capabilities can conflict. We address this

problem by adding another dimension (called the *virtual* dimension), which 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 evenly, 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 dimension by piggybacking load data onto the periodic heartbeat messages sent between neighbors that are used to maintain the CAN structure. After a job is routed to the node that meets its minimum resource requirements, that node chooses a dimension and a target node among its neighbors, to try 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 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, that node will pick as the run node the least loaded node among itself and its neighbors. Otherwise, the job continues to be pushed to a node with higher resource capability farther out in some dimension in the CAN. This probabilistic approach can balance load effectively, as shown in our previous work, but also minimizes over-provisioning. More details on our previous work for initial job placement can be found in Kim et al.[14].

C. Resource Management in a Multi-core Grid

Multi-core nodes may be capable of running multiple jobs simultaneously, so that the number of currently available cores and the available amount of other shared resources can vary over time for each node in the grid. Jobs also may request more than one core to express the requirements of a multi-threaded application. However, a structured DHT like our CAN can have problems with frequent changes to its structure, because it works best in a low-churn environment. To express the dynamically changing amount of available resources in each node, and to minimize the changes required to our existing CAN mechanisms, we represent dynamic resource availability by employing two logical nodes for each physical one: one that models the maximum resource available for that node (*Max-node*), and a second that models the currently unused amount of that resource (*Residue-node*) [2].

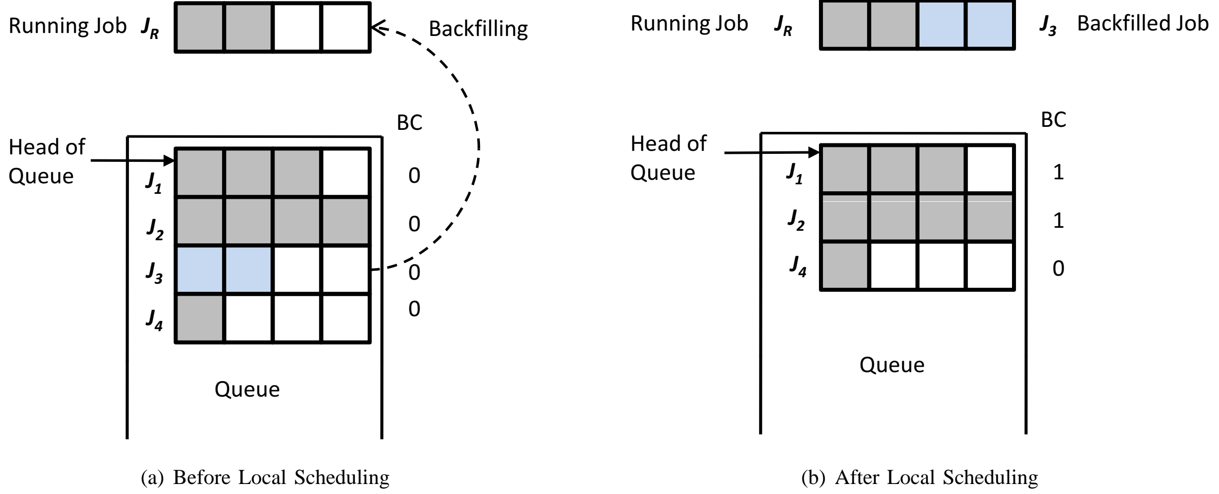
We have designed two resource management schemes, named *Balloon-Model* and *Dual-CAN*, that employ two logical nodes per physical node. Dual-CAN uses two separate CANs, one for each logical node type, so that dynamic effects due to resource changes (e.g., jobs starting or ending) in a multi-core node affect only the Secondary CAN, which contains only Residue-nodes. The number of nodes in the Secondary CAN is typically much fewer than in the Primary CAN (composed of Max-nodes), so the additional overhead for managing the Dual-CAN is not high. However, maintaining an additional CAN is not free, so we can also incorporate Residue-nodes into a single Primary CAN in a simple form, called a Balloon. A Balloon represents the currently available amount of resources for a node as a point in the CAN, and is associated with the zone that contains that point in the CAN. Therefore, the addition or removal of a Balloon due to resource availability changes for the node the Balloon represents affects at most 2 nodes in the CAN, minimizing changes to the Primary CAN. Using both static and dynamic node information in the two management schemes, a job is assigned to an appropriate node capable of running the job, preferably a node not currently running any other jobs (a *free* node). The initial job matchmaking and information aggregation schemes are similar to what was described for a single-core environment in Section III-B, except that the algorithms require information on core utilization rather than on the number of free nodes. Once a run node is determined, the job is inserted into the local queue of the node to wait to be run. The default queuing policy is first-come first-serve (FCFS), based on the time the job arrived in the system, but a node tries to run as many jobs as possible simultaneously to utilize all its available resources.

IV. A Tri-Pronged Approach

In this section we discuss the new dynamic scheduling techniques in detail. After successful initial job assignment, as described in Section III, we can still improve performance if we exploit residual resources by reordering jobs in the queue or by migrating jobs across nodes. After step 4, but before step 5 in the job submission and execution procedure described in Section III-A, a job redistribution algorithm based on current dynamic load status is invoked periodically in each node to try to improve job placement. We do not allow preemptive scheduling as in Condor [15], which causes the system to stop the currently running job and to later resume execution. Therefore there is no cost in terms of job turn-around time for job migration between nodes, since we only move jobs that have not started yet. There is some communication cost to send a job profile to a different node, but that is negligible because a job profile is not very big. The following three sections describe in detail our three methods: (1) local scheduling, (2) internode scheduling, and (3) queue balancing.

A. Local Scheduling

Local scheduling addresses selecting a waiting job from the job queue of a local node, regardless of its arrival order, constrained by the remaining available resources on the node (some resources may already be used by currently running jobs).



(a) Before Local Scheduling

(b) After Local Scheduling

Figure 1. Local Scheduling: Changing the Backfilling Counter (BC)

A key difference from other approaches is that we do not rely on job descriptions to provide running time information, which earlier backfilling algorithms use to prevent backfilled jobs from delaying other waiting jobs. We are left with the problem of preventing unconstrained backfilling from starving jobs with high resource requirements.

To avoid job starvation or unreasonably long waiting times, we employ a backfilling counter (BC) value for every job, with an initial value of zero. We then allow only a job with a BC equal to or greater than that of the job at the head of the queue to backfill. After backfilling a job, all other jobs that were ahead of the backfilled job in the queue have their BC incremented. Therefore, the BC for a job is the number of jobs that have bypassed the job in the waiting queue. This BC does not allow unlimited backfilling from jobs behind a given job in the queue, so that every job can begin execution without waiting too long, as will be shown in the experiments in Section V. Figure 1 shows how backfilling occurs and how the backfilling counters of jobs change after backfilling. In Figure 1, the shaded slots show the number of required cores for jobs on the quad-core machine, and Job J_3 is backfilled to use two free cores in the machine.

If multiple jobs are candidates to be backfilled, we must choose the best job to run for better utilization. We use the *Backfill Balanced* algorithm [6] to rank jobs, and choose the one whose product of *balance measure* (BM) and *fullness measure* (FM) is the minimum. BM and FM are defined as follows:

$$BM = \frac{max_k(S^k + R_j^k)}{\frac{\sum_{k=1}^{K}(S^k+R_j^k)}{K}} = \frac{MaximumUtilization}{AverageUtilization} \quad (1)$$

$$FM = 1 - \frac{\sum_{k=1}^{K}(S^k + R_j^k)}{K} = 1 - AverageUtilization \quad (2)$$

where K is the number of resources (or requirements), S^k is normalized utilization for resource k ($1 \leq k \leq K$, $0 \leq S^k \leq 1$), and R_j^k is job j's normalized requirement for resource k ($0 \leq R_j^k \leq 1$). BM measures unevenness across utilization of multiple resources, and FM measures how much resources are under-utilized on average. Therefore, lower BM and FM imply better balanced resource utilization and better average utilization, respectively.

B. Internode Scheduling

Internode scheduling is an extended version of local scheduling; the target node for backfilling can be the neighbors in the CAN. A node backfilling counter (NBC), which is the BC of the job at the head of the node's waiting queue, is used to prevent jobs with large resource requirements from long waits in the queue and from starvation. Only jobs whose BC is equal to or less than the NBC of the target node can be migrated. Figure 2 shows how BC works with NBC for internode scheduling. Job J_4 in the center node can be run on a free core either in the left or in the right node, but J_4 cannot be migrated to the left node because the NBC of the left node is greater than the BC of J_4. However, J_4 can be moved to the right node because the NBC of the right node is less than the BC of J_4.

While local scheduling is only a change to the job execution order within the queue on a node, internode scheduling must decide 1) which node initiates job migration (are jobs *pushed* away from a heavily loaded node or *pulled* to a lightly loaded node), 2) which node should be the sender (or receiver) of a job, and 3) which job should be migrated.

In the PUSH scheduling model the job sender initiates the migration process. First the sender node tries to match every job in its queue with residual free resources in its neighbor nodes in the CAN. That is possible because every node knows its neighbor nodes' resource capabilities and

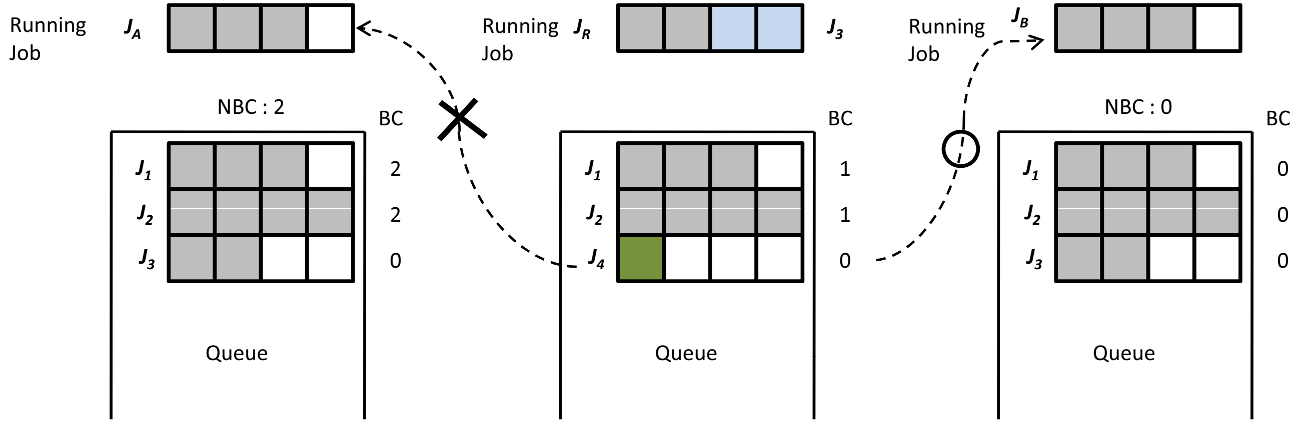


Figure 2. Internode Scheduling: J_4 cannot be moved to the left node because of BC

recent load information. If a job can be run on multiple neighbors, the sender sends it to the node that has minimum objective function value as follows.

$$f_{Inter-PUSH} = BM \cdot FM \cdot \frac{1}{CPU_{SPEED}} \tag{3}$$

To prefer the fastest node among neighbors, the objective function also includes an inverse term for CPU speed. Before sending a job profile, there is a simple confirming handshake process between a sender and a potential receiver to avoid inappropriate job migration because the potential receiver information may not be up-to-date at the sender.

On the other hand, for the PULL model, a receiver node tries to obtain a job from its CAN neighbors so as not to waste its available resources. However, the node does not have all information on the queued jobs' resource requirements in its neighbors to minimize neighbor update message sizes, so the node invokes a *PULL-Request* message to the node with maximum queue size among its neighbors. When a node receives a PULL-Request message, it checks whether any of its waiting job can be backfilled onto the requesting node, and if the job's BC is equal to or less than the NBC of the pulling node. If so, the job is migrated to the receiver and starts running. If there are multiple candidate jobs in the waiting queue, then the job that has minimum objective function value ($BM \cdot FM$, as above), is selected. If there is no candidate job, then the requesting node gets a *PULL-Reject* message and continues to look for another potential sender with maximum queue length among neighbors not contacted recently.

C. Queue Balancing

The local scheduling and internode scheduling algorithms find and execute a job using residual free resources in a node, meaning that only jobs that can start running immediately will be moved. However, if the load across nodes is skewed, meaning that job queue lengths vary greatly, a more pro-active *queue balancing* scheme may improve load distribution, and overall throughput, across heterogeneous nodes.

Our grid model allows for multiple resource types to be specified for a node, therefore defining and measuring load is more complex than for a single resource type. First, we set the maximally loaded resource among the **K** available resources as the *Load* of a node, and our algorithm minimizes the total sum of the *Loads* among neighbors, and also balances *Load* across the nodes [7]. We define a W_i^k, *normalized load* for *Resource* k of Node i by:

$$W_i^k = \textstyle\sum_{J_j \in Queue_i} (R_j^k), 1 \leq k \leq K \tag{4}$$

where J_j is Job j, R_j^k is the k-th normalized resource requirement for J_j, and $Queue_i$ is the job queue for node i. The *normalized load* of *Node* i, L_i is given by

$$L_i = Max(W_i^k), 1 \leq k \leq K \tag{5}$$

The PUSH and PULL job migration models can be used for queue balancing, as they were for internode scheduling. For PUSH, a node i computes normalized load (L_i) for itself and for its neighbors. If L_i is the locally maximum value among all its neighbors, then node i checks its queue to find candidate jobs for migration that reduce L_i if the (candidate) job is moved. When there are multiple candidate jobs, the algorithm selects the job and the receiver node that minimize an objective function if the job is moved to the neighbor. The objective function is defined by:

$$f_{QB-PUSH} = TLL \cdot MLL \tag{6}$$

$$TLL = \sum_{i \in Neighbors} L_i \tag{7}$$

$$MLL = Max(L_i), i \in neighbors \tag{8}$$

where TLL is the total local load and MLL is the maximum local load. This policy is used to minimize total load in

neighbors as well as to balance load across neighbors with the goal of maximizing system throughput through efficient utilization of node resources.

The PULL model is similar to the PUSH model, except that the node with a locally non-zero minimum normalized load among equal or less capable neighbors will initiate the PULL process from the most loaded node among its neighbors. Since zero local load means a free node, internode scheduling handles that case. The reason that we consider only equal or less capable neighbors is the following. If a highly capable node (which has a large amount of resources) is locally minimum-loaded, the node is likely to pull a large job, so small jobs cannot get benefits from this algorithm. On the other hand, if a node that has a relatively small amount of resources is locally minimum-loaded, call it **N**, the potential sender may not have a job that the pulling node is capable of running. In this case, node **N** cannot obtain a job to run, and other less loaded nodes also cannot pull a job because node **N** is still the locally minimally loaded node. Therefore, PULL requests may not occur frequently enough to help with load-balancing as compared to the PUSH scheme. Thus, to pull small jobs as well as to attempt frequent pull requests, we choose the PULL node as the least loaded node among equal or less capable neighbors.

V. Experiments

A. Experimental Setup

We used a synthetic workload to model a typical grid resource configuration and a heterogeneous set of jobs. Our simulation scenario contains 1000 multi-core nodes (each with 1, 2, 4 or 8 cores), and 5000 jobs submitted to those nodes. Each node and each job is given multiple resource capabilities or requirements, respectively, for CAN resource dimension such as CPU speed, memory size, disk space and the number of cores. A high percentage of the nodes (and jobs) have relatively low resource capabilities (requirements), and a low percentage of nodes (jobs) have high resource capabilities (requirements). Moreover, we simulated both *clustered* and *mixed* workloads and node capabilities to cover a wide range of grid scenarios. *Clustered* means that a small number of distinct sets of computing nodes (or jobs) with the same resource capabilities (requirements) are available (submitted), while a *mixed* environment has various sets of nodes (jobs) with randomly assigned capabilities (requirements). In addition, job requirements can be omitted by users. If users do not specify a requirement for a resource, the matchmaking process does not take resource requirements into account, so that the number of nodes capable of running the job can be large. We define the *Job Constraint Ratio* as the probability that each resource type for a job is specified.

The interval between individual job submissions follows a Poisson distribution, with varying average inter-job arrival times in the experiments. Each job has an expected running time with average value T, uniformly distributed between $0.5T$ and $1.5T$, with T = 3600 seconds, running on a canonical node with a normalized CPU speed of 1. The simulated job running time is then scaled up or down by the node CPU speed relative to the canonical node.

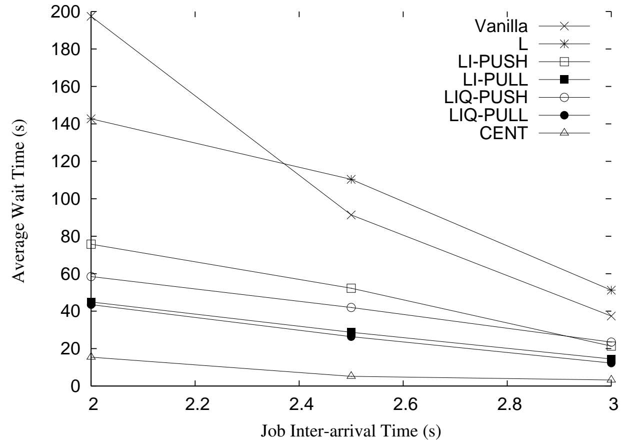


Figure 3. Average Job Wait Time: Clustered Nodes/Clustered Jobs

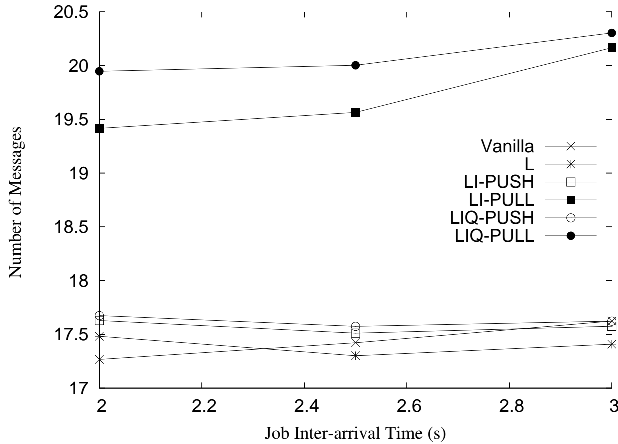
We compare our schemes to a greedy online centralized scheduler, which has a single queue and assigns jobs based on the complete global grid load information. Such a scheme would be very expensive in a real system, but gives some indication of the best possible performance for our decentralized system. The centralized scheduler is used only to measure load balancing performance, and does not incur any (communication) costs to obtain load status information from all nodes.

To measure the performance of a long running desktop grid system, and 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 achieve 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. However, we did not test very lightly loaded systems, because those are not very interesting for measuring dynamic scheduling performance.

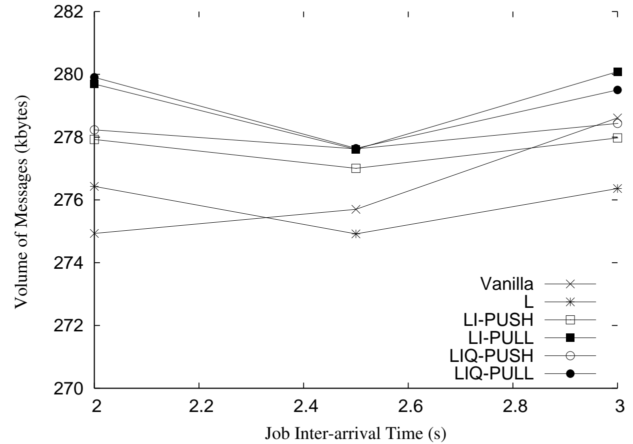
B. Experimental Results

Figure 3 compares the performance of our proposed dynamic scheduling schemes with the centralized scheduler, for varying job inter-arrival times. To see the effects of the different schemes, we run the simulation with various combinations of our methods as follows:

1) **L**: local scheduling only
2) **LI**: local scheduling + internode scheduling
3) **LIQ**: all three methods (local scheduling + internode scheduling + queue balancing).



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



(b) Volume (total bandwidth) of messages (per minute per node)

Figure 4. Costs - Messages per minute per node

In each scenario for LI and LIQ, the PUSH and PULL mechanism are both tested (denoted by LI(Q)-PUSH/PULL). **Vanilla** denotes the scenario with only the initial job matchmaking described in Section III-C and no dynamic scheduling, while **CENT** denotes the results of the greedy centralized scheduler.

In Figure 3, the average job wait time is shown, to measure overall load balancing performance. LIQ-PULL shows the best performance (shortest average wait time) among our schemes, and has performance competitive with the centralized scheduler.

In general, the PULL mechanisms are better than the PUSH ones in a highly loaded system, because in the PULL schemes idle nodes more aggressively try to find jobs to run. On the other hand, in experiments not shown, we have seen that PUSH shows better performance in a lightly loaded system. However, the initial matchmaking algorithm balances load well enough to handle light loads, so the PUSH-based job migration technique does not outperform a PULL-based scheme when dynamic load balancing is really needed to improve throughput. This result is consistent with the observations on the characteristics of PUSH and PULL load balancing schemes described by Demers et al. [16] that suggest using pull (or a combination of push and pull) over push to spread information across distributed database systems. Moreover, the performance of LI is similar to that of LIQ, i.e. internode scheduling provides major benefits in performance, and queue balancing provides some additional benefit. We vary the job inter-arrival time from a heavily to a lightly loaded system (from 2 to 3 seconds), but the results do not vary much, except in one case – local scheduling shows worse performance than Vanilla for the 2 second inter-arrival time. Local scheduling cannot guarantee increased performance because it cannot guarantee that it does not increase the wait time of other jobs - that is the difference between our local scheduling algorithm and a conservative backfilling algorithm.

Figure 4 shows the cost for each scenario, measuring the number of messages and the total bandwidth (volume) of messages per minute per node. Note that the Y-axis in each figure in Figure 4 does not start at zero, to better see the differences between the scenarios. In Figure 4(a), Local scheduling and the two PUSH-based methods do not generate significant additional messages compared to Vanilla. On the other hand, despite the improved load balance performance, PULL has a higher cost than PUSH, because PULL requires more messages to search its neighbors iteratively for jobs to run. Even though PULL generates more messages, it is still desirable compared to the other schemes because of its better load balancing performance, since the absolute number of messages is still very low (only about one message per node every three seconds, in a highly loaded system). Moreover, the volume of messages in each scenario is very similar to that of Vanilla, because the size of messages for job migration is small (See Figure 4(b)). The two PULL schemes also do not increase the message volume much, because the iterative PULL trials require only small messages. Therefore, the additional cost for our migration schemes is very small compared to Vanilla.

We have run various combinations of scenarios with clustered and mixed nodes and jobs, but we show only the results for clustered nodes and clustered jobs, because the results (performance competitive to the centralized scheduler) are basically similar for other combinations and we believe that this is the most common scenario for many desktop grids.

Figure 5 presents the average wait time when the job constraint ratio is varied from 20%-80%. In this experiment, the average job inter-arrival time is 3 seconds, and we model both clustered nodes and clustered jobs. The results are generally similar to the previous experiment, regardless of the

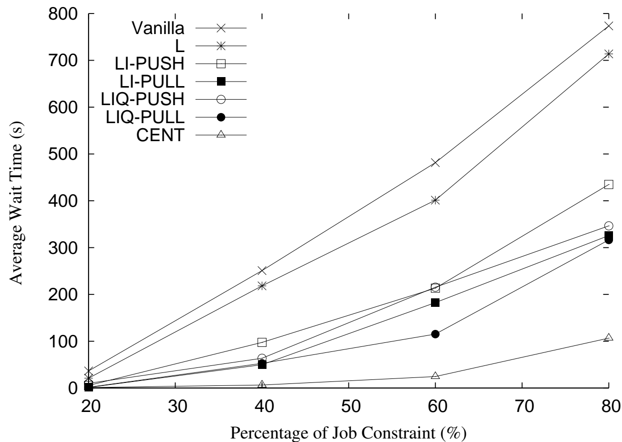


Figure 5. Average Job Wait Time, Changing Job Constraint Ratio

percentage of job constraints, i.e. PULL is better than PUSH, LIQ is similar to LI, and LIQ-PULL is competitive to CENT, except that LIQ-PULL gets worse when the job constraint ratio is very high (e.g., 80%). That is because it becomes difficult to find nodes that are capable of running jobs with many resource constraints in the local neighborhood of the node, as jobs get more highly constrained. Because the number of capable nodes for a specific job becomes smaller with more constraints, local job migration does not help much.

Through the simulations, we have confirmed that our dynamic scheduling and load balancing schemes perform competitively with the centralized approach, and do not have a high cost, measuring the number of messages and the volume of messages needed to perform the dynamic scheduling.

VI. Conclusion

We have proposed three distinct decentralized dynamic scheduling schemes for P2P desktop grids. *Local scheduling* uses a variant of backfilling to leverage residual resources in a single machine. *Internode scheduling* extends local scheduling across machines by migrating jobs when that will allow a job to run immediately. *Queue balancing* proactively balances job waiting queues across machines to avoid highly skewed queues, to overlap job migration with other system activities.

Our combined approach improves total system throughput, by improving load balance across heterogeneous nodes and lowering average job queue wait times. The algorithms also avoid starvation of large jobs through the backfilling counter mechanism. Through extensive simulation, we have confirmed that our proposed schemes show performance competitive with that of a greedy online centralized algorithm, and do not incur high messaging costs. In addition, we note that our algorithms based on PUSH and PULL job migration mechanisms show performance consistent with that from previous work described in the literature.

In future work, we will extend our decentralized resource management scheme to accommodate asymmetric multiprocessors (e.g. multi-core machines equipped with GPGPUs (General Purpose Graphics Processing Units)), to extend into even more heterogeneous environments. We are also implementing our schemes in a real testbed grid to characterize its behavior and performance, in cooperation with researchers from the Maryland Astronomy department.

References

[1] S. Moore, “Multicore is bad news for supercomputers,” *IEEE Spectrum*, vol. 45, no. 11, pp. 15–15, Nov. 2008.

[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*. Atlanta, Georgia, USA: IEEE Computer Society Press, 2010.

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

[4] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka, “The EASY – LoadLeveler API project,” in *IPPS ’96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1996, pp. 41–47.

[5] P. Keleher, D. Zotkin, and D. Perkovic, “Attacking the bottlenecks in backfilling schedulers,” *Cluster Computing: The Journal of Networks, Software Tools and Applications*, vol. 3, 2000.

[6] W. Leinberger, G. Karypis, and V. Kumar, “Job scheduling in the presence of multiple resource requirements,” in *Supercomputing ’99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1999, p. 47.

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

[8] N. G. Shivaratri, P. Krueger, and M. Singhal, “Load distributing for locally distributed systems,” *IEEE Computer*, vol. 25, no. 12, pp. 33–44, 1992.

[9] M. Becchi and P. Crowley, “Dynamic thread assignment on heterogeneous multiprocessor architectures,” in *CF’06: Proceedings of the 3rd Conference on Computing Frontiers*. New York, NY, USA: ACM, 2006, pp. 29–40.

[10] D. Zhou and V. Lo, “Wavegrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system,” in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE Computer Society Press, April 2006.

[11] ——, “Wave scheduler: Scheduling for faster turnaround time in peer-based desktop grid systems,” in *Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2005.

[12] 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.

[13] 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.

[14] 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 the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC-16)*, Jun. 2007.

[15] A. Roy and M. Livny, “Condor and Preemptive Resume Scheduling,” *Grid Resource Management: State of the Art and Future Trends*, pp. 135–144, 2003.

[16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, “Epidemic algorithms for replicated database maintenance,” in *PODC ’87: Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1987, pp. 1–12.

Author Biographies

Jaehwan Lee is a Ph.D. student in the Computer Science Department at the University of Maryland, College Park. His current research interests include high performance computing, peer-to-peer Grid computing for multi-core and heterogeneous environments, and general distributed system related to network and security. Before his study in Maryland, he was a researcher at Korea Telecom (KT) Labs for five years. His research focused on public wireless network based on IEEE 802.11/16, such as Quality of Service (QoS) enhancement, fast hand-off, nation-wide network management and AAA/security issues. He received his B.S.(1998) and M.S.(2000) in Electrical Engineering from Seoul National University, Korea. He was a recipient of the General Electric (GE) Scholarship and the Korean Government Scholarship for Electric Power Industry during 1996-1998 and 2005-2007, respectively.

Pete Keleher received a Ph.D. in computer science from Rice University in 1995. He is currently an Associate Professor in the Computer Science Department at the University of Maryland. Professor Keleher’s primary interests are in the design and analysis of distributed computing infrastructure, distributed security infrastructure, and communication performance.

Alan Sussman is an Associate Professor in the Computer Science Department at the University of Maryland. His research interests include Grid computing, Peer-to-Peer (P2P) systems, high performance database and I/O systems, coupled multiphysics simulations, and compilers and runtime environments for distributed and parallel systems. He received his Ph.D. in computer science from Carnegie Mellon University and his B.S.E. in Electrical Engineering and Computer Science from Princeton University.