NGS: Service Adaptation in Open Grid Platforms

Krishnaveni Budati, Jinoh Kim, Abhishek Chandra and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota - Twin Cities
{budati, chandra, jon}@cs.umn.edu

Abstract

Large-scale donation-based distributed infrastructures need to cope with the inherent unreliability of participant nodes. A widely-used work scheduling technique in such environments is to redundantly schedule the outsourced computations to a number of nodes. We present the design and implementation of RIDGE, a reliability-aware system which uses a node's prior performance and behavior to make more effective scheduling decisions. We have implemented RIDGE on top of the BOINC distributed computing infrastructure and have evaluated its performance on a live PlanetLab testbed. Our experimental results show that RIDGE is able to match or surpass the throughput of the best BOINC configuration by automatically adapting to the characteristics of the underlying environment. In addition, RIDGE is able to provide much lower workunit makespans compared to BOINC. RIDGE is also able to produce significantly lower communication makespans for downloading clients. Collectively, the results suggest that RIDGE has great promise for service-oriented environments with time constraints.

1 Introduction

Voluntary distributed computing infrastructures have been an active area of research in the past few years, e.g. SETI@home [2]. Today, these infrastructures are being used in a diverse set of application domains such as bioinformatics [6], physics [7], and environment science [5]. BOINC [1] is a generalization of these projects that provides a computing infrastructure for utilizing donated resources.

This paper presents the design and implementation of *RIDGE (Reliable Infrastructure for Donation-based Grid Environments)*, a system designed to combine reliability and performance of the underlying infrastructure. RIDGE is built as an extension to BOINC.

We focus on two problems here, the timeliness of computation and communication performance of the nodes. These problems arise as nodes have dynamically changing workloads, may leave and join unexpectedly, and may behave maliciously. When nodes are serving data, their extreme time-varying heterogeneity in terms of different capacity, bandwidth, and latency can also compromise performance. For the first problem, we present a scheme that can dynamically adjust the degree of replication based on the current node behavior. In an earlier paper, we showed how intelligent replication can improve performance through a simulation study [9]. This paper focuses on the implementation and deployment of the proposed ideas in a live environment.

For the second problem, we present a scheme that can dynamically select the most appropriate nodes for data download based on current network dynamics. The key contribution is that clients can make independent download decisions from replicated data nodes without direct interaction and minimal state.

We have deployed a prototype of RIDGE and evaluated it on a live distributed testbed on PlanetLab [3], using the BLAST [4] bioinformatics application. The results show that RIDGE can automatically match, and in some cases surpass, the best static BOINC performance (which requires knowing the dynamics of the environment) in terms of reliability, and can achieve far better computation makespan. The results also show that communication makespan can be significantly reduced using our schemes (17-43% improvement over existing heuristics). Collectively, these results indicate that RIDGE is well-suited to service-oriented environments with time constraints.

2 System Architecture

RIDGE is implemented on top of the core BOINC architecture, and it utilizes BOINC mechanisms for workload creation, communication with worker nodes, result gathering, etc. We first briefly describe the core BOINC architecture, followed by the RIDGE enhancements and workload allocation strategy.

2.1 BOINC Architecture and Work Allocation Policy

The BOINC architecture consists of a centralized server responsible for distributing work to the worker nodes. Each unit of computation (referred to as a *"workunit"*) is replicated into a fixed number of replicas (referred to as *"tasks"*). The replication factor is a static value specified by the application writer. Results are returned by the workers to the server upon completion of each task execution, and are verified using a verification technique specified by the application designer. M-majority voting and M-first voting are the most common verification techniques used. With M-majority voting, each workunit is replicated into at least 2M-1 tasks and the workunit is said to have completed successfully if a minimum of M out of the 2M-1 results match. In M-first voting, each workunit is replicated into at least M tasks and a workunit is said to have completed successfully as soon as M results match. A key limitation of BOINC work assignment policy is that a static replication factor is used for all workunits and the assignment of tasks to worker nodes is arbitrary.

2.2 RIDGE Scheduling Framework

RIDGE replaces the default BOINC workload allocation policy with a *Reputation-based scheduling* technique [9]. The idea behind this technique is to collect reliability ratings of individual worker nodes and use this information to group them together more intelligently and thus increase throughput while meeting the desired success-rate. The basic idea is that a node's reliability rating is based on the number of 'timely' and 'correct' task executions performed in the past relative to the total number of tasks allocated to it. Using these values, it is possible to determine effective redundancy groups, both in size and in worker composition. More details of this technique can be found in [9]. While the original algorithms are designed for M-majority voting, we have extended them to work for M-first voting in this paper. The RIDGE server employs these scheduling algorithms and is driven by the following key parameters:

- *Target Success-Rate*: It is defined as the minimal success-rate desired from the system and is specified as a value in the range 0-1.
- *Execution-Threshold*: It is defined as the maximum time that a task execution is allowed to take for it to be considered 'timely'.
- *Scheduling-Threshold*: The number of workers for which the RIDGE scheduler waits for before running the scheduling algorithm. In this paper, we use a threshold of 1 to enable a fair comparison of RIDGE to vanilla BOINC.
- *MinClients*: The minimum number of workers that a workunit should be scheduled to.
- *MaxClients*: The maximum number of workers that a workunit should be scheduled to.

There is a tradeoff between the desired success-rate and throughput. The higher the replication factor, the greater is the success-rate achieved, while there may be a drop in throughput.

2.2.1 Component Architecture

Scheduler: The scheduler is responsible for forming redundancy groups of worker nodes based on their reliability ratings, and assigning a workunit to each redundancy group. Before each scheduling instance, the scheduler waits for the *Scheduling Threshold* number of workers to arrive at the server. Once it has enough workers to proceed, the scheduler obtains the reliability ratings for the available workers from the reputation manager. It then runs a Reputation-based scheduling algorithm [9] to form the redundancy groups and assigns tasks to the worker nodes.
Reputation Manager: The reputation manager maintains the reliability ratings of the worker nodes. The scheduler uses these reliability ratings in making its scheduling decisions. The reputation manager is also responsible for updating the reliability ratings of worker nodes when a workunit is validated: a node's rating may be increased or decreased based on the outcome of the validation [9].
Validator: This is a part of the BOINC core architecture (while the others are not). The validator initiates the validation process when the required number of results for a workunit arrive at the server, and determines if an agreement is achieved.
Re-Scheduler: When a validation fails, the re-scheduler decides the number of additional tasks to be created for the failed workunit As a default, the re-scheduler creates one additional task incrementally for the failed workunit.

2.2.2 RIDGE Workflow

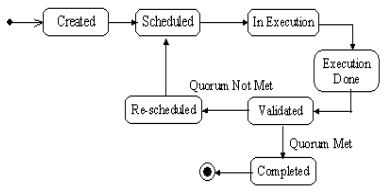


Figure 1: Workunit Life-Cycle

Figure 1 illustrates the workflow in the RIDGE framework through the life-cycle of a workunit. Workunits are

created and put in the RIDGE database. Additional workunits are created as the work queue empties to maintain a minimum workpool size at the server. Worker nodes arrive at the RIDGE server requesting work. The request handler informs the scheduler about the arrival of a worker node and blocks the worker node until the scheduler is ready to allocate work. When the number of available workers meets the scheduling threshold, the scheduler performs the allocation of workunits. In our framework, priority is given by default to partially completed workunits whose tasks are assigned to the most reliable available workers. The remaining workers in the worker queue are then grouped into redundancy groups and each group is given tasks of one workunit to execute. At this point, the workunit transits to the *scheduled* state. Once the workunit is scheduled, the worker nodes in its associated group pick up their assigned tasks and start executing them. The workunit is now *in execution*.

When the minimum number of results for a workunit that are required for validation have arrived, the *validation* process is triggered to verify the results using the validation scheme. If the validation succeeds, then the workunit is considered to be complete, otherwise, the workunit needs to be *re-scheduled*. The re-scheduler then incrementally creates new tasks for this workunit which are eventually allocated by the scheduler.

3 Evaluation

In this section, we evaluate the RIDGE framework and present a comprehensive performance comparison of RIDGE against vanilla BOINC. We first describe our experimental setup along with the metrics used, followed by the evaluation results.

3.1 Experimental Setup

We have deployed BOINC/RIDGE on PlanetLab [3]—a shared distributed infrastructure consisting of donated machines. Our Grid consists of 120 nodes which serve as the worker nodes. The BOINC/RIDGE server runs on a dedicated machine outside the PlanetLab infrastructure. We used the BLAST (Basic Local Alignment Search Tool) [4] bioinformatics application as our test application. In our setup, BLAST is run as a BOINC project by writing a BOINC-specific wrapper around it. Each workunit consists of a BLAST database file and an input sequence that has to be compared with each sequence in the database file. BLAST performs the sequence comparison and generates an output file result which is returned to the server. We have used a standard BLAST database file igSeqNt, with sizes of 28MB and 55MB for our experiments. The input sequence was a randomly selected sequence from the database file and is of length 770 bytes. M-first voting is used as the verification technique. To isolate the impact of RIDGE vs. BOINC scheduling, we have disabled 'Re-scheduling' in the initial results presented (later, we re-enable it). Thus, in our first set of experiments, a workunit whose validation is not successful for the first time is deemed to have failed and is discarded from the work queue. Each experiment is run for 2 hours and is repeated 3 times to smooth the effects of the underlying load fluctuations in PlanetLab.

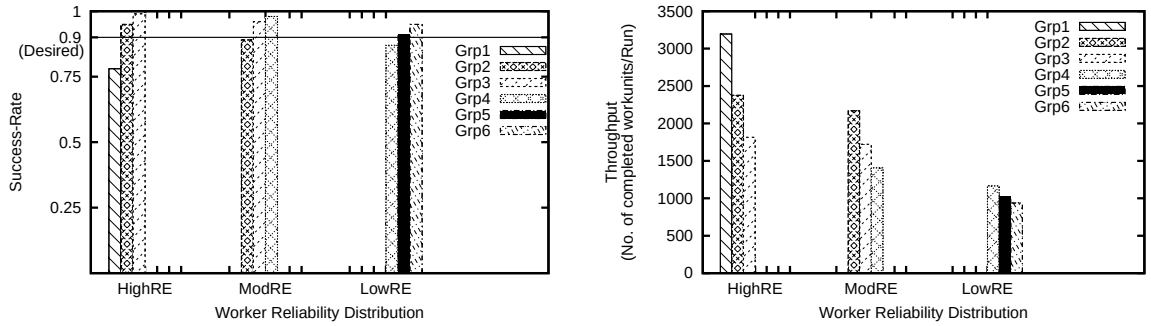
3.2 Timeliness Evaluation of Reliability

In this section, we evaluate the performance of BOINC against RIDGE w.r.t. the timeliness of workers in an environment where getting work done within certain time-constraints is the primary objective. Here, we assume that every worker is 100% reliable w.r.t. correctness and hence the reliability of a worker reduces to the probability that it returns a result in a 'timely' manner. The timeliness of a task is determined by an *'Execution-Threshold'* parameter, which is defined as the maximum task execution time beyond which a task is considered late and discarded. The ultimate goal is to use these reliability ratings to do sophisticated scheduling to support deadline-oriented service environments.

Since all nodes are assumed to be correct in this scenario, a workunit is said to be completed as soon as one scheduled task returns within the 'Execution-Threshold' time. In other words, M-first voting with M=1 is used as the verification technique. Since M is just 1, now comparably higher success-rates could be achieved for smaller replication factors, and hence, we use a desired success-rate of 0.90 for these experiments.

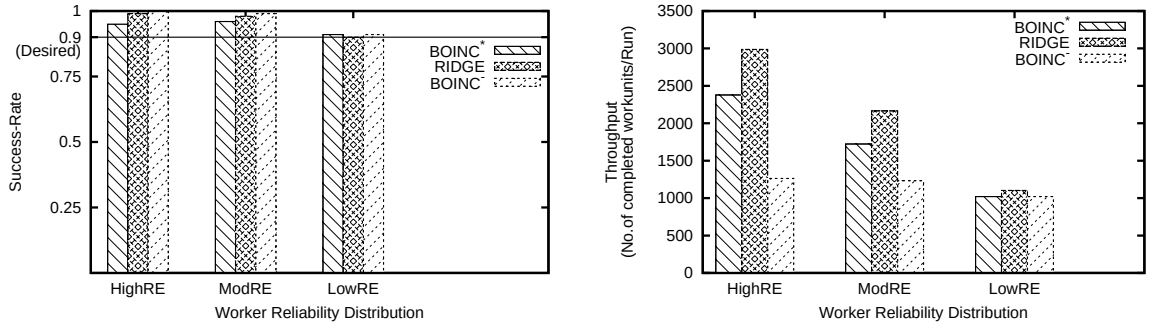
3.2.1 Emulation of different Reliability Environments

In this set of experiments, we use the actual timeliness values of the results returned by nodes in our PlanetLab testbed to determine their reliability ratings. To emulate different reliability environments, we used different values of Execution-Threshold, so that higher values of Execution-Threshold corresponded to more reliable environments and vice-versa. We use an Execution-Threshold of 120s, 180s and 240s respectively. We refer to these distributions as LowRE, ModRE and HighRE respectively. We observed an interesting fact that most of the nodes are either highly reliable or unreliable w.r.t. a given Execution-Threshold and there are a very small fraction of nodes with reliabilities in the range 0.2-0.9. This implies that given an Execution-Threshold, learning the reliability of nodes is indeed useful since the node reliabilities are relatively stable over time intervals in the order of a few hours.



(a) Success-Rate (b) Throughput

Figure 2: Comparison of different BOINC configurations.



(a) Success-Rate (b) Throughput

Figure 3: Comparison of RIDGE with BOINC optimal and conservative configurations.

3.2.2 Performance of BOINC

In this section, we evaluate the performance of BOINC for various fixed replication factors for different reliability environments discussed above. The replication factor is varied from a minimum of 1 to a maximum of 6, to determine the optimal replication factor for a desired success-rate.

Figures 2(a) and 2(b) show the performance of BOINC for different replication factors, for different reliability environments. We observe that the optimal replication factor values in this case are 2, 3, and 5 respectively for HighRE, ModRE and LowRE with corresponding (success-rate, throughput) combinations of (0.95, 2378), (0.96, 1723) and (0.91, 1020) respectively, and that RIDGE equals or surpasses the best static replication factor. Since the underlying distribution may not be known a priori, a conservative application designer might operate at a fixed replication factor of 5, to get a minimal success-rate of 0.90 for all reliability environments.

3.2.3 BOINC vs. RIDGE Comparison

We now compare the performance of RIDGE and BOINC. Each run of BOINC was set at 2 hrs, while RIDGE was run for 3 hrs with 1 hr for the learning period. The RIDGE server is configured with a *Target Success-Rate* of 0.90, and *MinClients* and *MaxClients* set to 1 and 5 respectively.

Performance Comparison: Figures 3(a) and 3(b) illustrate the performance comparison of BOINC*, RIDGE and BOINC$^-$ (that uses a replication factor of 5). We observe that RIDGE meets the desired success-rate of 0.90 in all three environments. Also, from the Throughput comparison graph, we notice that RIDGE in fact has higher throughput than BOINC*. This is because RIDGE can form groups of size 1 which is not possible in BOINC and RIDGE does fast serial scheduling.

Resource Utilization Comparison: Table 1 illustrates the Resource Utilization of BOINC* with that of RIDGE for the three reliability environments. An interesting observation is that the Group-Size of RIDGE is *less than* that of BOINC*. This is because, depending on the value of *Threshold-Time*, there is a high percentage of very highly reliable workers in

	Group Size		Quorum Size	
Distr	BOINC*	RIDGE	BOINC*	RIDGE
HighRE	1.67	1.46	1.05	1.01
ModRE	2.31	1.89	1.12	1.03
LowRE	3.68	3.22	1.41	1.08

Table 1: BOINC* vs RIDGE Resource Utilization

all three reliability environments. RIDGE actually has the option to create groups of size exactly 1 using such highly reliable workers, thus, lowering the average group-size. Depending on the reliability of the environment, there may be large number of such single-worker groups, having a positive performance impact, despite the minor overhead in RIDGE due to oversized groups or performance overheads as discussed before. However, forming such small groups is not possible for BOINC*, since it operates at a fixed replication factor for all the workunits (with an optimal replication factor of at least 2 in all cases).

3.3 Evaluation for Service-Oriented Environments

In this section, we evaluate how BOINC and RIDGE perform in service-oriented environments. We characterize such environments by a high-level unit of work, a *service request*, that is defined as a set of workunits. A request is said to be completed when all its constituent workunits are completed successfully. For this set of experiments, we enable the 'Re-scheduling' component of BOINC and RIDGE, so that a workunit that has failed in its first validation is not discarded, but is re-scheduled until it is successfully completed.

Since the optimal BOINC configuration BOINC* has already been identified, we compare only the performance of BOINC* against RIDGE. To emulate 'Service Request' behavior, each set of consecutive workunits in the workpool are bundled to model a 'Service Request'. The performance comparison is shown for two reliability environments, HighRE and LowRE. The results for ModRE are similar and are omitted due to space constraints.

Makespan Comparison: Request makespan is a key metric in a service-oriented environment since a service request is not complete until all of its component workunits are complete. Figures 4(a) and 4(b) show the request makespan for BOINC* and RIDGE as the number of workunits per request is varied from 1 to 8. We observe that as the request size is increased, the makespan for BOINC* increases much more rapidly when compared to RIDGE. This is explained by the way BOINC and RIDGE schedule and re-schedule work. As mentioned, randomization in scheduling is one factor. Another is that when a validation fails, BOINC puts the additional task in the workpool and no explicit preference is given to the pending tasks. However, since RIDGE gives preference to pending work compared to new work, RIDGE achieves better requests makespans. This is another factor that supports RIDGE in a service-oriented environment. Our results also indicate that RIDGE not only minimizes the makespan but also maintains the request throughput.

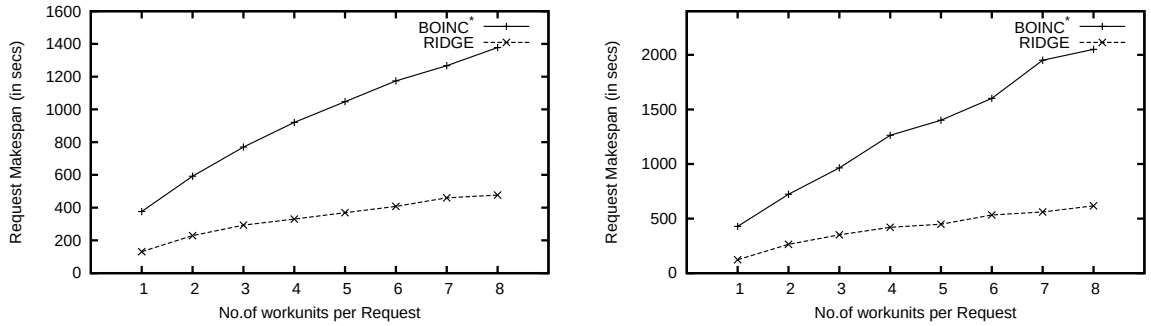
4 Collective Data Download

We now consider the complementary problem of concurrent downloading by a number of compute clients working on the same service request in RIDGE. This challenge is complicated by the extreme time-varying heterogeneity of the volunteer Grid as data servers have widely different capacity, bandwidth, and latency with respect to a downloading client. Simultaneous downloading from central data servers can lead to bottlenecks due to capacity and geographic constraints. Since worker nodes can be dispersed world-wide, the download times of some distant and poorly connected nodes might overwhelm the overall execution time of the service request. Replicating to a few data servers can achieve more efficient data access, but still suffers from the problems of scalability and fault-tolerance if static replicas are used. Caching at the compute workers can help, but it is not well-suited to handle the problems of node churn and dynamic data generation. To address these problems, we assume that the data is highly replicated across a data network and that clients make local decisions to select a server for download.

4.1 System Model

In our system model, we incorporate a compute network as well as a data network to host large-scale services that operate on large datasets, require significant computation, and are accessed by remote end-users, e.g. BLAST. In our model, the *compute network* consists of worker nodes that provide CPU cycles for computation. The *data network* is responsible for transmitting data from the data generation or storage locations to the worker nodes that operate on the data. Our compute network is BOINC [1], and our data network is Pastry [8].

Formally, the compute network consists of compute nodes $\{c_1, c_2, ...\}$, and the data network is composed of data servers $\{d_1, d_2, ...\}$. All the data files required for computation are assumed to be replicated across multiple servers in the data network, with R_i denoting the set of replicas for a file F_i. For scheduling purposes, a job J is decomposed into a set of tasks or workunits $U_i (0 < i \leq n)$, where each U_i requires one (or more) files F_i. The scheduler assigns a workunit U_i to a set of worker nodes W_i, each of which



(a) HighRE Makespan

(b) LowRE Makespan

Figure 4: Comparison of Request Makespan for different reliability environments

then attempts to download the associated file F_i from one of its replicas.

To download the file, each node $w_{ij} \in W_i$ queries the data network for a set of servers holding the file F_i, along with their current state. The server state might include attributes such as the server capacity, its roundtrip latency from the worker node, etc. We define a candidate set C_{ij} for a worker node to be the set of replicas that the data network returns in response to the query.

The size and composition of the candidate set is a function of the degree of replication, the time-out values used to search for replicas, and is dependent on the type of data network employed as well as the location of the worker node. The worker then uses a *server selection heuristic* select a server s_{ij} from the candidate set for the actual download.

Minimizing makespan is key as the service request will not be complete until all tasks are finished. Since data download or communication is a key component of the workunit execution time, we define the *communication makespan* to be the maximal download time for a workunit U_i:

$$makespan = \max_{w_{ij} \in W_i} \{T_{ij}\},$$

where, T_{ij} is the download time for worker $w_{ij} \in W_i$.

4.2 Server Selection Heuristics

We investigated different metrics that affect the efficiency of data downloading. Based on the impact of these metrics, we present heuristics for selecting data servers in our environment. A key requirement of our model is to minimize the overall makespan of a service request, and not to simply minimize the individual download times at each worker independently.

Proximity has been employed as a network metric of choice in several domains ranging from routing in overlay networks to nearest server selection in the Internet. In general, proximity refers to the network distance between hosts and can be measured in terms of roundtrip latency between the hosts, using TCP roundtrip times or ICMP echo packets. According to conventional wisdom, proximity is the dominant factor in predicting data download performance. As a result, proximity information is collected by many data network infrastructures.

We conducted experiments on a 43-node PlanetLab slice to determine the parameters that affect download performance. Several measures are explored, and we find strong correlations not only between RTT and download performance but also between network bandwidth and download performance. We derived a *cost function* $f_{i,j}$ that is used by a worker i to quantify the desirability of a server j for data download:

$$f_{i,j} = \alpha_j \cdot rtt_{i,j}, \quad (1)$$

where $rtt_{i,j}$ is the RTT between the worker and the server, and α_j is a weight used to incorporate other server parameters, defined as follows:

$$\alpha_j = e^{(k_j/bw_j)}, \quad (2)$$

where bw_j is the bandwidth of the server, and k_j is a (server-dependent) constant that incorporates parameters such as load and concurrency.

We define three heuristics for server selection that use different values for k_j:

- BW-ONLY: Uses $k_j = constant$. We use $k_j = 1$ in our experiments.
- BW-LOAD: Uses $k_j = load_j$, where $load_j$ is the 5-minute average system load on the server.
- BW-CAND: Uses $k_j = num\_cand_j$, where num_cand_j is the number of times the servers has responded as a candidate within the last 15 seconds.

The heuristic BW-ONLY uses only the RTT and the bandwidth metrics for selecting a server, while the other heuristics BW-LOAD and BW-CAND also use average system load and concurrency information respectively. BW-CAND uses the number of times the server has responded as a candidate window. In the experiments, we set the time window to 15 seconds. Using the heuristic BW-CAND, servers which have responded as a candidate several times recently are penalized, because they are more likely to be selected by multiple workers, and hence to be concurrently serving data in the near future.

4.3 Performance Evaluation

4.3.1 Experimental Testbed and Methodology

To evaluate the various server selection heuristics described in the previous section, we conducted experiments on a set of randomly selected PlanetLab nodes geographically distributed across the globe. We conducted each of our experiments as follows: data files are distributed over the data network at the beginning of each experiment, and then data queries are generated for downloading these data files. For each data query, a set of worker nodes are selected randomly to request the same designated file concurrently. Table 2 shows the various experimental scenarios we created. The scenarios differ in some of the parameters above, as well as the specific set and number of nodes that were used.

4.4 Comparison of Server Selection Heuristics

Figure 5 compares the various server selection heuristics for Concurrency (C)=5 and Data Size (D)=2MB, using the aggregated results of all the experiments that used C=5 and D=2MB. Figure 5(a) plots the average download time and makespan respectively for the various heuristics. The first observation we make from the figure is that the bandwidth-based heuristics perform much better that proximity-based server selection in terms of both the average as well as the makespan. Moreover, the gaps in performance are larger in the case of makespan ($\sim$30-45%) than in mean download time ($\sim$20-30%). This result is also seen from Figure 5(b) that plots the CDF of the download completion times. As seen from the figure, 10% of PROXIM queries take more than 60 seconds to complete, while the bandwidth-based heuristics take less than 40 seconds to complete 90% of their queries. Moreover, these heuristics finish most of their queries within around 100 seconds, while about 5% percent of queries are unfinished for PROXIM selection. Thus, this result implies that *using bandwidth in addition to proximity produces better performance, not only in terms of individual download, but also in overall makespan.*

Another observation we make from Figures 5(a) and 5(b) is that BW-CAND shows the best results for both mean download time and makespan. In the case of makespan, BW-CAND gains over 40% compared to PROXIM, while BW-ONLY and BW-LOAD show 30-40% gains. Figure 5(b) shows the CDF of the completion times of all the queries. This result implies that *incorporating concurrency in addition to bandwidth improves the performance even further.*

The basic reason why the bandwidth-based heuristics outperform proximity-based selection is that they can exclude extremely slow servers. In our experiments, the participating hosts are almost uniformly distributed through the bandwidth ranges as shown in the Figure 5(c): nearly 10% of the hosts have a bandwidth under 1Mbps, 50% of the hosts have under 30Mbps, and upper 10% hosts have over 80Mbps bandwidth. By penalizing low bandwidth servers, the bandwidth-based heuristics can select servers with better bandwidth, even though they may be a little further from the worker node.

Table 3: Server Bandwidth Distribution

Class	Low $< 1Mbps$	Medium $1 - 10Mbps$	High $> 10Mbps$
EX-1	5%	26%	67%
EX-2	12%	6%	82%
EX-3	0%	24%	76%
EX-4	0%	24%	76%

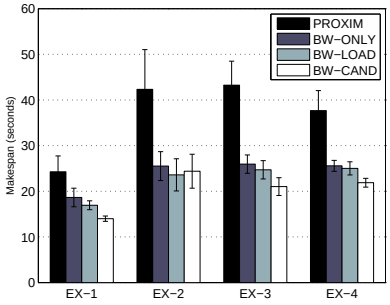
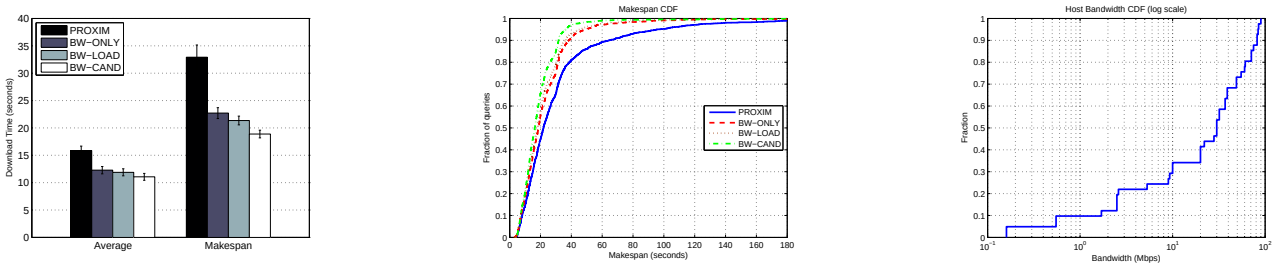


Figure 6: Performance comparison for individual experiments (C=5, D=2MB): PROXIM is the worst and BW-CAND is the best in all cases except EX-2, where BW-CAND is comparable to BW-ONLY and BW-LOAD.

The reason BW-CAND performs the best can be found in the bandwidth distribution of servers as shown in Table 3. Here, we classify hosts in three categories: low, medium, and high bandwidth, based on their bandwidth values. All of the bandwidth-based heuristics can penalize

Table 2: Experiments setup

Experiments	Nodes	Replication	Candidate	Concurrency	Data Size	Queries
EX-1	19	10	5	5	2M	690
EX-2	33	10	5	5	256K,512K,1M,2M	> 200
EX-3	29	10	5	5,10,15	2M	> 250
EX-4	29	10	5	5	256K,512K,1M,2M	> 450



(a) Mean vs. Makespan (b) Completed Query CDF (c) Server Bandwidth

Figure 5: Performance comparison of different heuristics (C=5, D=2MB, Num Queries>1800): (a) shows the average download time and makespan, (b) shows the CDF of download completions, and (c) shows the bandwidth distribution of data servers.

low-bandwidth servers (i.e. those with less than 1Mbps), but may not penalize medium-bandwidth servers (i.e. those between 1Mbps and 10Mbps). In fact, BW-ONLY might not penalize such medium-class servers because the weight value α_j is likely to stabilize beyond 1Mbps, due to its exponential relation to bandwidth (Equation 2). In addition, if the average load is low on these medium-class hosts (close to 1), BW-LOAD also does not penalize them. In contrast, BW-CAND can penalize these servers, if too many clients try to select them, thus leading to higher values of recent candidate set queries. Thus, BW-CAND is able to provide better performance for such servers, by proactively preventing overloads from happening, while BW-LOAD is able to react only to past observed load. Unlike other experiments, EX-2 shows all the heuristics to have similar performance. This can be explained by the fact that EX-2 has only 6% medium-class servers (as seen from Table 3), whereas other experimental scenarios have more than 20% medium-class servers, thus reducing the differentiation opportunity for BW-CAND. However, note that BW-CAND does not perform any worse than other heuristics even under these conditions.

References

[1] 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)*, 2004.

[2] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11), 2002.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the Fifth Symposium on Networked Systems Design and Implementation (NSDI'04)*, 2004.

[4] BLAST. http://www.ncbi.nlm.nih.gov/blast.

[5] Climate Prediction Network. `http://www.climateprediction.net/`.

[6] Folding@home distributing computing project. `http://folding.stanford.edu`.

[7] PPDG: Particle Physics Data Grid. `http://www.ppdg.net`.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-pe er systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.

[9] J. Sonnek, M. Nathan, A. Chandra, and J. Weissman. Reputation-Based Scheduling on Unreliable Distributed Infrastructures. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, July 2006.