Load Balancing Across Near-Homogeneous Multi-Resource Servers

William Leinberger, George Karypis, Vipin Kumar, and Rupak Biswas

February 16, 2000

Accepted in 9th Heterogeneous Computing Workshop (HCW 2000)

Load Balancing Across Near-Homogeneous Multi-Resource Servers [*]

William Leinberger, George Karypis, Vipin Kumar
Army High Performance Computing and Research Center
Department of Computer Science and Engineering, University of Minnesota
{leinberg, karypis, kumar}@cs.umn.edu

Rupak Biswas
MRJ Technology Solutions, Numerical Aerospace Simulation Division
NASA Ames Research Center, Moffett Field, CA 94035
rbiswas@nas.nasa.gov

February 16, 2000

Abstract

An emerging model for computational grids interconnects similar multi-resource servers from distributed sites. A job submitted to the grid can be executed by any of the servers; however, resource size or balance may be different across servers. One approach to resource management for this grid is to layer a global load distribution system on top of the local job management systems at each site. Unfortunately, classical load distribution policies fail on two aspects when applied to a multi-resource server grid. First, simple load indices *may not recognize that a resource imbalance exists at a server. Second, classical* job selection policies *do not actively correct such a resource imbalanced state. We show through simulation that new policies based on* resource balancing *perform consistently better than the classical load distribution strategies.*

1. Introduction

An emerging model in high performance supercomputing is to interconnect similar computing systems from geographically remote sites, creating a *near-homogeneous* computational grid system. Computing systems, or servers, are homogeneous in that any job submitted to the grid may be sent to any server for execution. However, the servers may be heterogeneous with respect to their exact resource configurations. For example, the first phase of the NASA Metacenter linked a 42-node IBM SP2 at Langley and a 144-node SP2 at Ames [7]. The two servers were homogeneous in that they were both IBM SP2s, with identical or *synchronized* software configurations. However, they were heterogeneous on two counts: the number of nodes in each server, and the fact that the Langley machine consisted of thin nodes while the Ames machine had wide nodes. A job could be executed by either server without modifications, provided a sufficient number of nodes were available on that server.

The resource manager for the near-homogeneous grid system is responsible for scheduling submitted jobs to available resources such that some global objective is satisfied, subject to the constraints imposed by the local policies at each site. One approach to resource management for near-homogeneous computational grids is to provide a *global load distribution system* (LDS) layered on top of the *local job management system* (JMS) at each site. This architecture is depicted in Figure 1. The compute server at each site is managed by a local JMS. Users submit jobs directly to their local JMS which places the jobs in wait queues until sufficient resources are available on the local compute server. The global LDS monitors the load at each site. In the event that some sites become heavily loaded while other sites are lightly loaded, the LDS attempts to equalize the load across all serves by moving jobs among the sites. The JMS at each site is then responsible for the detailed allocation and scheduling of local resources to jobs submitted directly to it, as well as to jobs which are assigned to it by the global LDS. The local JMS also provides load status to the LDS to support load distribution decisions, as well as a scheduling Applications Programming Interface (API) to implement these decisions. For example, in the NASA Metacenter, a *peer-aware* receiver-initiated load balancing

[*]This work was supported by NASA grant NCC2-5268 and contract NAS2-14303, and by Army High Performance Computing Research Center (AHPCRC) cooperative agreement DAAH04-95-2-0003 and contract DAAH04-95-C-0008. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute.

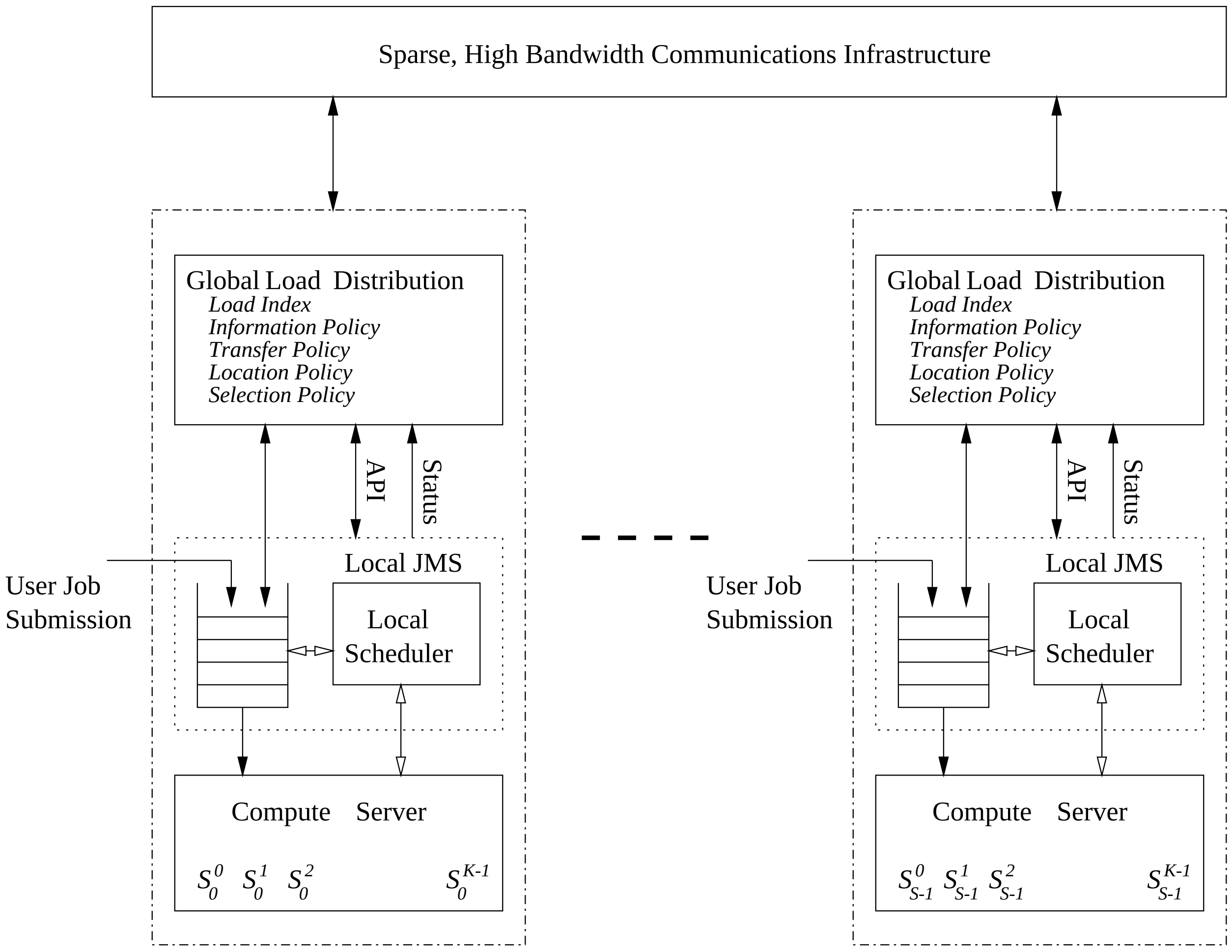


Figure 1. Near-Homogeneous Metacomputing Resource Management Architecture

algorithm was used to move work from one IBM SP2 to the other. When the workload on one SP2 dropped below a specified threshold, the peer-aware load balancing mechanism would query the other SP2 to see if it had any work which could be transferred for execution.

The architecture depicted in Figure 1 is conceptually identical to classical load balancing in a parallel or distributed computer with two notable exceptions. First, the compute server at each site may be a complex combination of multiple types of resources (CPUS, memory, disks, switches, and so on). Similarly, the applications submitted by the users are described by multiple resource requirements. We generalize these notions and define a K-resource server and corresponding K-requirement job. Each server S_i has K resources, $S_i^0, S_i^1, \ldots, S_i^{K-1}$. Each job J_j is described by its requirements for each resource type, $J_j^0, J_j^0, \ldots, J_j^{K-1}$. Note that the servers are still considered homogeneous from the jobs' perspective, as any job may be sent to any server for execution.

The second exception is that the physical configurations of the K resources for each server may be heterogeneous. This heterogeneity can be manifested in two ways. The *amount* of a given resource at one server site may be quite different than the configuration of a server at another site. For example, server S_i may have more memory than server S_j. Additionally, servers may have a different *balance* of each resource. For example, one server may have a (relatively) large memory with respect to its number of CPUs while another server may have a large number of CPUs with less memory.

Classical load balancing attempts to maximize system throughput by keeping all processors busy. We extend this notional goal to fully utilizing all K resources at each site. One heuristic for achieving this objective is to *match* the job mix at each server with the capabilities of that server, in addition to balancing the load across servers. For example, if a server has a large shared memory, then the job mix in the local wait queue should be adjusted by the global LDS to contain jobs which are generally memory intensive. Compute intensive jobs should be moved to a server which has a relatively large number of CPUs with respect to its available memory. The goal of the LDS is to therefore balance the total resource demand among all sites, *for each type of resource*.

This work investigates the use of load balancing techniques to solve the global load distribution problem for computational grids consisting of near-homogeneous multi-resource servers. The complexity of multi-resource compute servers along with the multi-resource requirements of the jobs cause the methods developed in past load balancing research to fail in at least two areas. First, the definition of the *load* at a given server is not easily described by a single load index. Specifically, a *resource imbalance*, in which the local job mix does not match the capabilities of the local server, is not directly detectable. This impacts the ability of the global LDS to match the workload at a site to the capabilities of the site. We propose a simple extension to a classical load index measure based on a *resource balancing heuristic* to provide this additional level of descriptive detail. Second, once a resource imbalance is detected, existing approaches to selecting which jobs to move between servers fail to actively correct the problem. We provide an analogous job selection policy, also based on resource balancing, which heuristically corrects the resource imbalance. The combination of these two extensions provides the framework for a global LDS which consistently outperforms existing approaches over a wide range of compute server characteristics.

The remainder of this paper is organized as follows. Section 2 provides an overview of relevant past research, concluding with variants of a baseline load balancing algorithm drawn from the literature. Section 3 investigates the limitations of the baseline algorithms, and provides extensions based on the resource balancing heuristic. A description of our simulation environment is given in Section 4. The performance results of our new load balancing methods as compared to the baseline algorithms is also summarized in Section 4. Finally, Section 5 provides conclusions and a brief overview of our current work in progress.

2. Preliminaries

Research related to this effort is drawn from single server scheduling in the presence of multiple resource requirements and general load balancing methods for homogeneous parallel processing systems.

Recent research in job scheduling for a single server has demonstrated the benefits of including information about the memory requirements of a job in addition to its CPU requirements [13, 14]. The generalized K-resource single server scheduling problem was studied in [10], where it was shown that simple backfill algorithms based on multi-dimensional packing heuristics consistently outperform single-resource algorithms, with increasing K. These efforts all suggest that the local JMS at each site should be multi-resource *aware* in making its scheduling decisions. This induces requirements on the global LDS to provide a job mix to a local server which maximizes the success rate of the local server.

The general goal of a workload distribution system is to have sufficient work available to every computational node to enable the efficient utilization of that node. A centralized work queue provides every node equal access to all available work, and is generally regarded as being efficient in achieving this goal. Unfortunately, the centralized work queue is generally not scalable as contention for the single queue structure increases with the number of nodes. In massively parallel processing systems where the number of nodes was expected to reach into the thousands, this was a key concern. In distributed systems, the latency for querying the central queue potentially increases as the number of nodes is increased. Load balancing algorithms attempt to emulate a central work queue by maintaining a representative workload across a set of distributed queues, one per compute node. In this paper, we investigate only the performance of load balancing across distributed queues.

Classical load balancing algorithms are typically based on a *load index* which provides a measure of the workload at a node relative to some global average, and four *policies* which govern the actions taken once a load imbalance is detected [15]. The load index is used to detect a load imbalance state. Qualitatively, a load imbalance occurs when the load index at one node is much higher (or lower) than the load index on the other nodes. The length of the CPU queue has been shown to provide a good load index on time-shared workstations when the performance measure of interest is the average response time [2, 11]. In the case of multiple resources (disk, memory, etc.), a linear combination of the length of all the resource queues provided an improved measure, as job execution time may be driven by more than CPU cycles [5].

The four policies that govern the action of a load balancing algorithm when a load imbalance is detected deal with information, transfer, location, and selection. The *information* policy is responsible for keeping up-to-date load information about each node in the system. A global information policy provides access to the load index of every node, at the cost of additional communication for maintaining accurate information [1].

The *transfer* policy deals with the dynamic aspects of a system. It uses the nodes' load information to decide when a node becomes eligible to act as a sender (transfer a job to another node) or as a receiver (retrieve a job from another node). Transfer policies are typically threshold based.

Thus, if the load at a node increases beyond a threshold T_s, the node becomes an eligible sender. Likewise, if the load at a node drops below a threshold T_r, the node becomes an eligible receiver. Load balancing algorithms which focus on the transfer policy are described in [2, 15, 16].

The *location* policy selects a partner node for a job transfer transaction. If the node is an eligible sender, the location policy seeks out a receiver node to receive the job selected by the selection policy (described below). If the node is an eligible receiver, the location policy looks for an eligible sender node. Load balancing approaches which focus on the use of the location policy are described in [8, 9].

Once a node becomes an eligible sender, a *selection* policy is used to pick which of the queued jobs is to be transferred to the receiver node. The selection policy uses several criteria to evaluate the queued jobs. Its goal is to select a job that reduces the local load, incurs as little cost as possible in the transfer, and has good affinity to the node to which it is transferred. A common selection policy is *latest-job-arrived* which selects the job which is currently in last place in the work queue.

The primary difference between existing load balancing algorithms and our global load distribution requirements is that our node is actually a *multi-resource server*. With this extension in mind, we define the following baseline load balancing algorithm:

- Load Index. The load index is based on the average resource requirements of the jobs waiting in the queue at a given server. This index is termed the resource average (RA) index. For our multi-resource server formulation, each resource requirement for a job in the queue represents a *percentage* of the server resource that it requires, normalized to unity. Therefore, the RA index is a relative index which can be used to compare the loads on different servers.

- Information Policy. As the information policy is not the subject of this study, we choose to use a policy which provides perfect information about the state of the global system. We assume a global information policy with instantaneous update.

- Transfer Policy. The transfer policy is threshold based, since it has been shown to provide robust performance across a range of load conditions. A server becomes a *sender* when its load index grows above the global load average by a threshold, T_s. Conversely, a server becomes a *receiver* when its load index falls below the global average by a threshold T_r.

- Location Policy. The location policy is also not the subject of this study. Therefore, we use a simple location policy which heuristically results in fast convergence to a balanced load state. In the event that the transfer policy indicates that a server becomes a sender, the location policy selects the server which currently has the least load to be the receiver. However, the selected server must also be an eligible receiver, meaning that it currently has a load which is T_r below the global average. Conversely, if the server is a receiver, the location policy selects the server which currently has the highest load that is T_s above the global average. If no eligible partner is found, the load balancing action is terminated.

- Selection Policy. A latest-job-arrived selection policy (LSP) is used to select a job from the sending server to be transferred to the receiving server. This selection policy generally performs well with respect to achieving a good average response time, but suffers from some jobs being moved excessively. Therefore, each job keeps a *job transfer count* which records the number of times it has been moved. When this count reaches a threshold T_c, the job is no longer eligible to be selected for a transfer. Jobs which are already executing are excluded from being transferred.

The sender initiated (SI), receiver initiated (RI), and symmetrically initiated (SY) algorithm variants are generated using a transfer policy which triggers a load balancing action on T_s, T_r, or both, respectively. All baseline variants use the RA load index and the LSP job selection policy.

3. Multi-Resource Aware Load Balancing Policies

In this section, we first discuss the limitations of the resource average load index, RA, and the latest-job-arrived selection policy, LSP, of the baseline load balancing algorithms for the heterogeneous multi-resource servers problem. We provide an example which illustrates where these naive strategies can fail to match the workload to the servers, resulting in local workloads which exhibit a resource imbalance. We then provide extensions to the load index and the job selection policy which strive to balance the resource usage at each server.

3.1. Limitations of RA and LSP

The resource average load index, RA, and the latest-job-arrived job selection policy, LSP, in the baseline algorithm fail in the multi-resource server load balancing context. The following discussion gives an example of these failures and provides some insight into possible new methods. Our new methods will be further discussed in Section 3.2.

In past research, the index used to measure the load on a server with respect to multiple resources consisted of a linear combination or an average of the resource requirements for the actively running jobs in a time-shared system. A corresponding index which may be applied to batch queued space-shared systems is to use the average of the total resource requirements of the jobs waiting in the wait queue. However, this may not always indicate a system state where there exists a resource imbalance, that is, the total job requirements for one resource exceeds the requirements for the other resources. Essentially, a server with a mismatched work mix will be forced to leave some resources idle while other resources are fully utilized, resulting in an inefficient use of the system as a whole.

Figure 2(a) depicts the state of the job ready queues, RQ_0 and RQ_1 for a two-server system, S_0 and S_1. Assume that each server has three resources, S_i^0, S_i^1, and S_i^2, and that the configuration for the two servers is identical, $S_0^0 = S_1^0, S_0^1 = S_1^1$, and $S_0^2 = S_1^2$. Each of the two ready queues currently has two jobs. The job which arrived latest at each server is on the top of the ready queue for that server. For example, the latest arriving job , J_L, in RQ_0 has the resource requirements $J_L^0 = 2, J_L^1 = 3$, and $J_L^2 = 2$. Note that the resource requirements for a job are given as a *percentage* of the total available in the server. The total workload for each resource, k, in a given server, S_i, is denoted as

$$W_i^k = \sum_{J_j \in RQ_i} (J_j^k), \quad 0 \le i < S, \quad 0 \le k < K.$$

The resource average load index for a given server, S_i, is then given by

$$RA_i = Avg(W_i^k), \quad 0 \le k < K.$$

In this example, $K = 3$ and $RA_0 = RA_1 = 4$.

The third queue in Figure 2(a), RQ_{Avg}, represents the global average workload for each resource in RQ_0 and RQ_1. The global average workload for resource k, is then given by

$$W_{Avg}^k = Avg(W_i^k), \quad 0 \le i < S.$$

Here, $S = 2$ and $W_{Avg}^0 = W_{Avg}^1 = W_{Avg}^2 = 4$, meaning that on average, each RQ_i has a total requirement of 4 percent for each resource. The global resource average load index is simply

$$RA = Avg(W_{Avg}^k), \quad 0 \le k < K,$$

which in this example is $RA = 4$. Server S_i is defined to be in a load balanced state as long as $RA * (1 - T_x) < RA_i < RA * (1 + T_x)$, where T_x is the transfer policy threshold, as defined in Section 2. Since $RA_0 = RA_1 = RA$, the system is believed to be in a load balanced state.

Even though the RA index indicates a balanced load, it is clear from Figure 2(a) that the job mix in RQ_0 has a higher requirement for resource S_0^1 than for resources S_0^0 and S_0^2. The result is that S_0 will probably be unable to fully utilize resources S_0^0 and S_0^2 as resource S_0^1 becomes the bottleneck. Conversely, the job mix in RQ_1 has a higher requirement for resources S_1^0 and S_1^2 than for S_1^1, resulting in an inefficient use of resource S_1^1. Therefore, the workload at each server suffers from a resource imbalance.

In order to detect this problem, we define a second load index, called resource balance (RB), which measures the resource imbalance at a given server or globally across the system. Namely, for server $S_i, 0 \le i < S$,

$$RB_i = \frac{Max(W_i^k)}{Avg(W_i^k)} = \frac{Max(W_i^k)}{RA_i}, \quad 0 \le k < K.$$

Similarly,

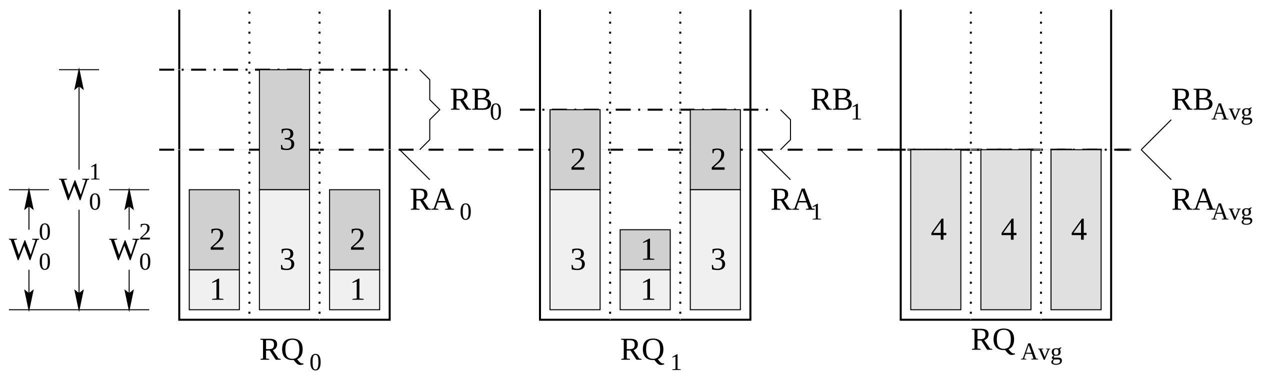
$$RB = \frac{Max(W_{Avg}^k)}{Avg(W_{Avg}^k)} = \frac{Max(W_{Avg}^k)}{RA_{Avg}}, \quad 0 \le k < K.$$

Heuristically, the RB index of a server measures how balanced the job mix is with respect to their different resource requirements. If the total resource requirements are all the same, then the local RB measure is unity, since $Max(W_i^k) = Avg(W_i^k)$. This corresponds to the case where the workload is *matched* to the server. The global RB is a measure of how well the total work in the system matches the capabilities of all the servers in the system. The goal of the load balancing algorithm is to move each server towards this global balanced resource level. In Figure 2(a), $RB_0 = 6/4$ or 1.5, while $RB_1 = 5/4$ or 1.25. Since $RB = 4/4$ or 1.0, the two servers recognize the existence of a resource imbalanced state.

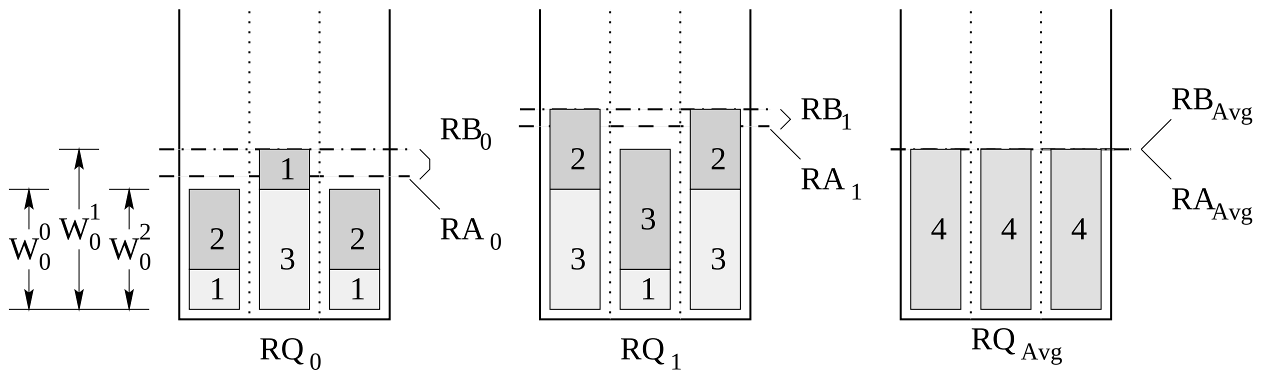
Once a resource imbalance is detected, the load balancing policies must actively correct the imbalance. Figure 2(b) shows the result of using the LSP policy to adjust the resource imbalance. Server S_0 sends its latest job to S_1, while S_1 sends its latest job to S_0. Note that the resource balance index improves on both servers, with $RB_0 = 4/3.33$ or 1.2, while $RB_1 = 5/4.66$ or 1.07. However, the resource balance could have been improved even further, as shown in Figure 2(c), by transferring the jobs which best balance the workload at both servers. We refer to this heuristic policy as the balanced job selection policy or BSP.

3.2. Resource Balancing Algorithms

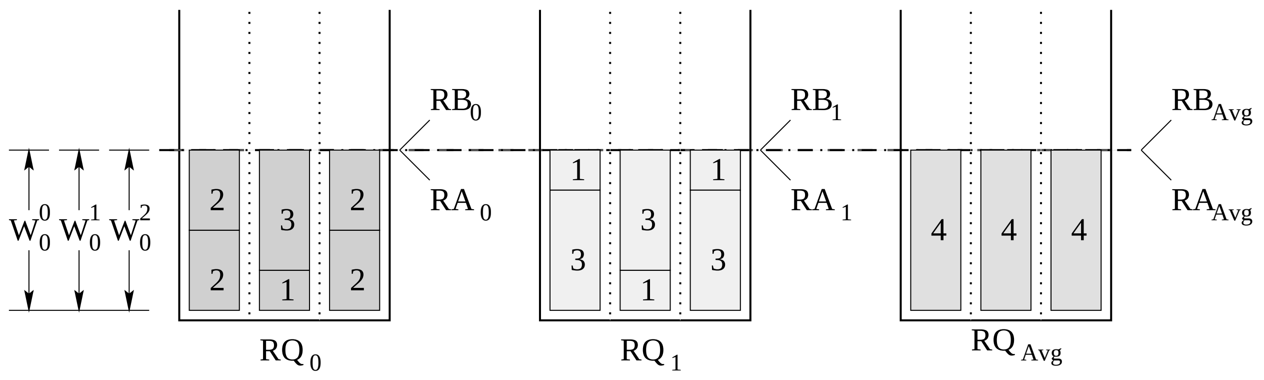
In the following discussion, we extend the baseline load balancing algorithm with the heuristic RB load index and the BSP job selection policy. In general, the goal of these extensions is to move the system to a state where the load is balanced across the servers *and* the job mix at each server matches the resource capabilities provided by that server. These extensions are described below.



(a) Comparison of RA and RB Load Index Measures



(b) Result of Latest Job Selection Policy (LSP)



(c) Result of Balance Job Selection Policy (BSP)

Figure 2. Limitations of RA and LSP

Sender Initiated, Balanced Selection Policy: SI_BSP. The baseline sender initiated algorithm, SI, is extended to SI_BSP by modifying the selection policy as follows. The fact that the load balancing action was triggered by the condition that the load index, RA, of a given server was above the global average implies that it has more work than at least one other server. Thus, this heavily loaded server needs to transfer work to another server. The BSP policy selects the job for transfer (out) which results in the best resource balance of the local queue. Note that transferring a job may actually *worsen* the resource imbalance, but we proceed nonetheless so that the overall excess workload can be reduced. Also, the resource balance at the receiving server may worsen as well. However, the receiving server currently has a workload shortage, so it may be executing less efficiently anyway.

Sender Initiated, RB Index, Balanced Selection Policy: SI_RB_BSP. The SI_RB_BSP algorithm extends the SI_BSP algorithm by including the RB load index, and modifying the transfer and selection policies as follows. First, the transfer policy triggers a load balancing action based on RA *or* RB. If the action is based on RA, SI_RB_BSP proceeds as SI_BSP. However, if the action is based only on RB, the selection policy is further modified over that used for SI_BSP. The job which *positively* improves the resource balance of the local queue the most is selected for transfer (out). If no such job is found, no action occurs.

Receiver Initiated, Balanced Selection Policy: RI_BSP. The baseline receiver initiated algorithm, RI, is extended to RI_BSP in a fashion complementary to SI_BSP.

Receiver Initiated, RB Index, Balanced Selection Policy: RI_RB_BSP. The RI_RB_BSP algorithm extends the RI_BSP algorithm in a fashion complementary to SI_RB_BSP.

Symmetrically Initiated, Balanced Selection Policy: SY_BSP. The baseline symmetrically initiated algorithm, SY, is extended to SY_BSP as follows. If the transfer policy triggers a send action, SY_BSP proceeds as SI_BSP. Alternatively, if the transfer policy triggers a receive action, SY_BSP proceeds as RI_BSP.

Symmetrically Initiated, RB Index, Balanced Selection Policy: SY_RB_BSP. The SY_RB_BSP algorithm extends the SY_BSP algorithm as follows. If the action is based on RA, SY_RB_BSP proceeds as SY_BSP. However, if the action is based only on RB, then SY_RB_BSP performs both send *and* receive actions using methods identical to SI_RB_BSP and RI_RB_BSP. Heuristically, this maintains the RA index but improves the RB index.

4. Experimental Results

The baseline and extended load balancing algorithms were implemented on a simulated system that is described in Section 4.1. The experimental results are summarized in Section 4.2.

4.1. System Model

The simulation system follows the general form of Figure 1. The server model, workload model, and performance metrics are discussed below.

Server Model. A system with 16 servers was used for the current set of experiments. A server model is specified by the amount of each of the K resource types it contains and the choice of the local scheduler. For all simulations, the local scheduler uses a backfill algorithm with a resource balancing job selection criteria [10]. To our knowledge, this is the best performing local scheduling algorithm for the multi-resource single server problem. At this point, we assume that the jobs are *rigid*, meaning that they must receive the required resources before they can execute. We also assume that the execution time of a job is the same on any server. Simulation results are reported for a value of $K = 8$.

Two independent parameters were used to specify the degree of heterogeneity across the servers in the simulated system. First, within a single server, the *server resource correlation*, S_{rc}, parameter specifies how the resources of a given server are balanced. This represents the *intra-server* resource heterogeneity measure. For example, assume each server has two resources, CPUs and memory. If a correlation value of about one were specified, then a server with a large memory would also have a large number of CPUs. Conversely, if a correlation value of about negative one were used, then a server with a large memory would probably have a low number of CPUs. Finally, a correlation value near zero implies that the resource sizes within a given server are unrelated. The value of the resource correlation ranged from 0.15 to 0.85 in the simulations (our simulator is capable of generating S_{rc} values in the range $-1.0/(K-1) < S_{rc} \leq 1.0$).

The second parameter is the *server resource variance*, S_{rv}, which is used to describe range of sizes for a single resource which may be found across all of the servers. This represents the *inter-server* heterogeneity measure. Again, a resource variance about one implies that the number of CPUs found in server S_i will be approximately the same as the number of CPUs found in server S_j for $0 \leq i, j < S$. In general, a resource variance of $S_{rv} = V$ implies that the server S_i with the largest amount of a resource k has V times as much of that resource as the server S_j which has the smallest amount of that resource. All other servers have some amount of resource k between S_i^k and S_j^k. The value of the resource variance ranged from 1.2 to 8.0 for our experiments.

Workload Model. The two main aspects of the simulated workload are the generation of multi-resource jobs and the job arrival rate. Recent studies on workload models have focused primarily on a single resource — the number of CPUs that a job requires. Two general results from these studies show that the distribution of CPU requirements is generally hyperexponential, but with strong discrete components at powers of two and squares of integers [3]. An additional study investigated the distribution of memory requirements on the 1024 processor CM-5 at Los Alamos National Laboratory. The conclusion was that memory requirements are also hyperexponentially distributed with strong discrete components. Additionally, there was a weak correlation between the CPU and memory requirements for the job stream studied [4].

We generalize these results to a K-resource workload as follows. The multiple resource requirements for a job in the job stream are described by two parameters. The kth resource requirement for job j, J_j^k, is drawn from a hyperexponential distribution with mean $\overline{X}_k$. Additionally, the correlation between resource requirements within a single job, J_{rc} is also specified. A single set of workload parameters was used for all of the initial simulations reported here, in which $\overline{X}_k = 0.15, 0 \leq k < K$, and the resource correlation $J_{rc} = 0.25$. Essentially, the average job requires 15% of each resource in an average server, and its relative resource requirements are near random.

Figure 3(a) shows the single resource probability distribution used for the workload. Note that the probability for small resource requirements is reduced over a strictly exponential distribution. We justify this modification by noting that small jobs are probably not good candidates for load balancing activity as they do not impact the local job scheduler efficiency significantly (except to improve it). Figure 3(b) shows the joint probability distribution for a dual resource $(K = 2)$ system. In general, the joint probability distribution shown in Figure 3(b) is nearly identical for all pairs $(i, j), 0 \leq i, j < K$, of resources in the job stream. This workload model has also been used to study multi-resource scheduling on a single server [10].

The job arrival rate generally affects the total load on the system. A high arrival rate results in a large number of jobs being queued at each server, while a low arrival rate reduces the number of queued jobs. For our initial simulations, we selected an arrival rate that resulted in an average of 32 jobs per server in the system. As each job arrives, it is sent to a server selected randomly from a uniform distribution ranging from 0 to $S - 1$. A final assumption is that the nature of the workload model impacts only the absolute values of the system performance, not the relative performance of the algorithms under study.

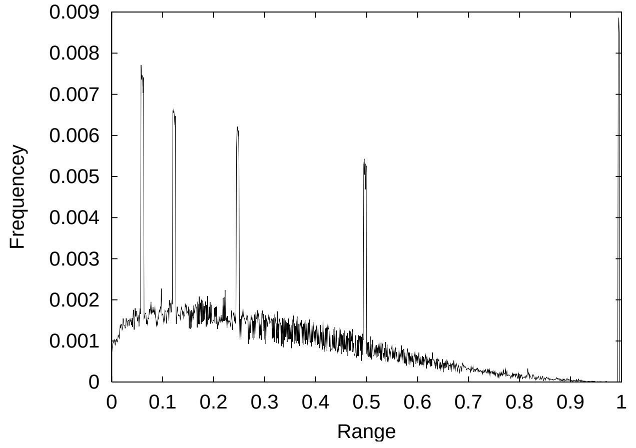
Performance Metrics. A single performance metric, *throughput*, is our current method for evaluating these algorithms. Throughput is measured as the total elapsed time from when the first job arrives to when the last job departs.

4.2. Simulation Results

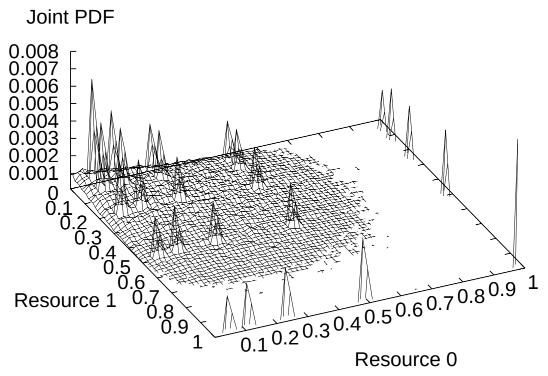
Our initial simulation results are depicted in Figures 4(a)–(f). Recall that load balancing algorithms essentially try to mimic a central work queue from which any server can select jobs as its resources become available. Therefore, the performance results for the load balancing algorithms are normalized against the results of a system with a central work queue. For each graph in the figure, the x axis represents the server resource variance parameter, S_{rv}, as described previously, while the y axis represents the throughput of the algorithms, normalized to the throughput of the central queue algorithm. The following paragraphs summarize these results.

Impact of the Resource Balancing Policies. Figures 4(a)–(c) depict the performance of the sender initiated, receiver initiated, and symmetrically initiated baseline and extended algorithms, normalized to the performance of the central queue algorithm. For these experiments, $K = 8$ and $S_{rc} = 0.50$ (resources within a server are very weakly correlated). In comparing the performance of the baseline and the extended algorithms, we see that the extended variants consistently out-perform the baseline algorithm from which they were derived. The addition of the intelligent job selection policy, BSP, provides a 5–10% gain in the SI_BSP, RI_BSP, and SY_BSP algorithms over the SI, RI, and SY algorithms, respectively. Moreover, the addition of the RB load index and associated transfer policy further increases these gains for SI_RB_BSP, RI_RB_BSP, and SY_RB_BSP.

Effects of Server Resource Correlation, S_{rc}. The jobs which arrive at each server may or may not have a natural affinity for that server. For example, if a server has a large memory and a few CPUs, a job which is memory intensive has a high affinity for that server. However, a job which is CPU intensive has a low affinity to that server. For a job stream with a fixed intra-job resource correlation, J_{rc}, the probability that an arriving job has good affinity to a server increases as S_{rc} increases. A larger natural affinity increases the packing efficiency of the local schedulers, improving the throughput. Figures 4(d)–(f) compare



(a) Single Resource Probability Distribution



(b) Dual Resource Joint Probability Distribution, Correlation=0.25

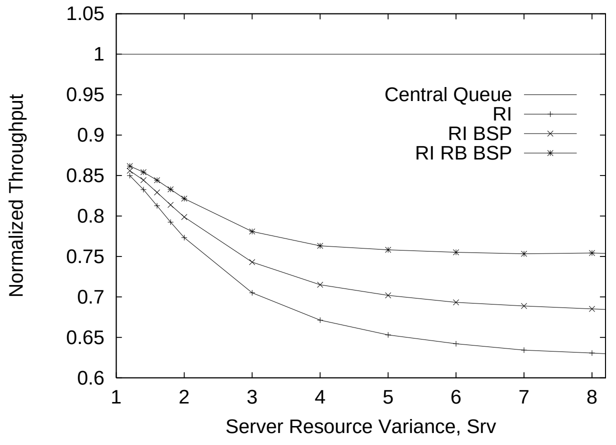
Figure 3. Multi-Resource Workload Model

the performance of the RI_RB_BSP, SI_RB_BSP, and the SY_RB_BSP algorithms, over the range of server resource correlation values, $S_{rc} = \{0.15, 0.50, 0.70\}$. Generally, as the value of S_{rc} increases, the performance of the load balancing algorithms also improve, due to an increased probability of natural affinity.

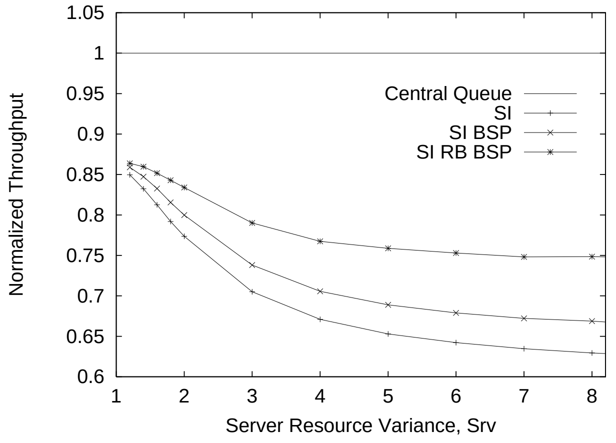
The SI_RB_BSP algorithm performs slightly better than RI_RB_BSP at low values of S_{rc} as seen in Figure 4(d). However, RI_RB_BSP begins to outperform SI_RB_BSP at higher values of S_{rc}, as seen in Figures 4(e) and 4(f). At low values of S_{rc}, the SI variant can actively transfer out jobs with low affinity, which occur with high probability, while the RI variant can only try to correct the affinity of their total workload. Higher values of S_{rv} magnify this problem. Therefore, the performance advantage goes to the SI variant. For higher values of S_{rc}, the probability of good job-server affinity is also higher. When accompanied by higher S_{rv}, the system may be thought of as having some larger servers and some smaller servers, with good job affinity to any server. In this case, the throughput of the system is driven by the efficiency of the larger servers. In the SI variant, the smaller servers will tend to initiate load balancing actions, by sending work to the larger servers. So while the smaller servers may execute efficiently, the larger servers may not. However, in the RI variant, the larger servers will tend to initiate load balancing, and intelligently select which work to receive from the smaller servers. Now, the larger servers will tend to execute more efficiently. For this reason, the performance advantage goes to the RI variant.

Impact of Server Resource Variation, S_{rv}. As the resource variation, S_{rv}, increases in the graphs of Figure 4, the throughput of the load balancing algorithms drops relative to the central queue algorithm. This is due to the fact that the average job size (size of the jobs resource requirements) is not taken into account when selecting jobs for transfer. At higher server resource variances, some servers have a very small amount of one or more resources. However, the average job size ending up on the servers with small resource capacities is the same as those ending up on the larger servers. The small size of the resources in these servers, relative to the average resource requirement of the arriving jobs, causes packing inefficiencies by the local scheduler, due to job size *granularity*. In the case of a centralized queue, the servers with small resource capacities are more likely to find jobs with smaller resource requirements. In short, simply balancing the workload resource characteristics is not sufficient. Other workload characteristics must also be emulated in the local queues, such as the average job requirements relative to the server resource capacities. This is a topic in our current work in progress and is briefly discussed in Section 5.

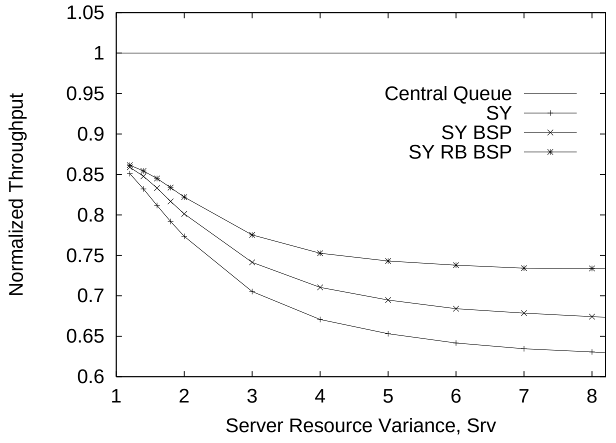
Central Queue vs. Load Balancing. A final observation may be drawn from the graphs in Figure 4. Even when the servers are all similarly configured (e.g. $S_{rv} \sim 1$ and $S_{rc} \sim 1$), there is a consistent performance gap of 15% for all baseline and extended load balancing algorithms with respect to the central queue algorithm. This is due to the fact that even if the load balancing algorithms are successful in balancing the load, the local scheduler at each server may not be able to find a job in its *local* queue to fill idle resources, even if such a job exists in the queue of a different server. Closing this gap is the subject of our current work and is briefly discussed in Section 5.



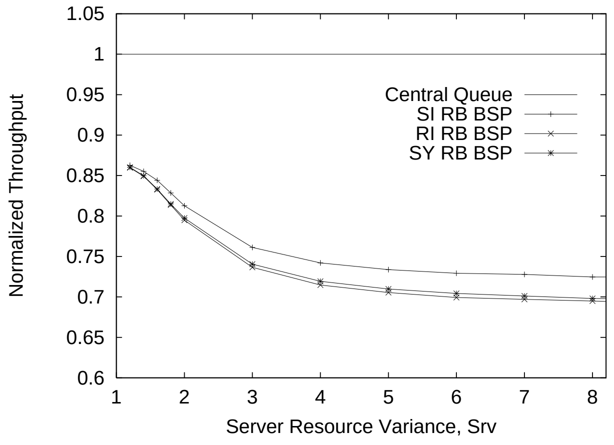
(a) Receiver Initiated Variants



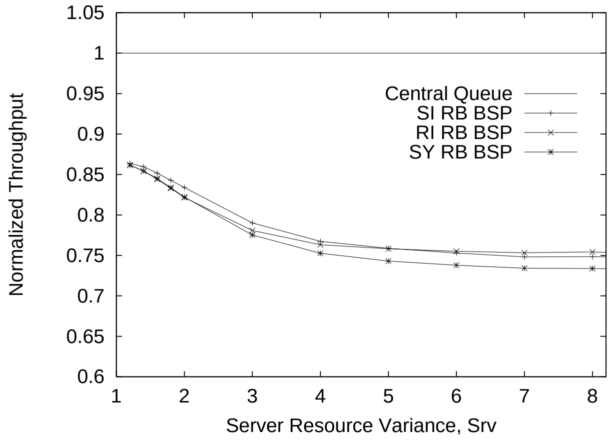
(b) Sender Initiated Variants



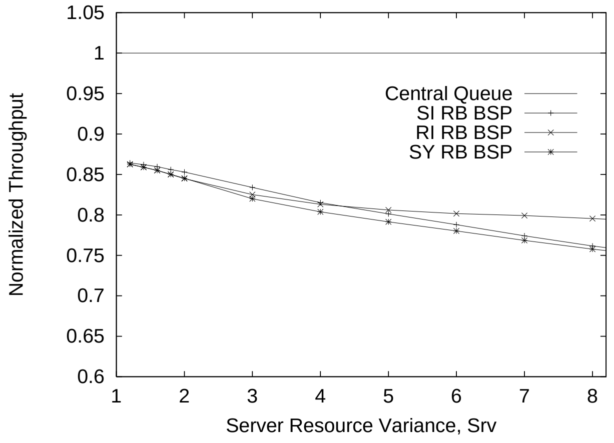
(c) Symmetrically Initiated Variants



(d) Server Resource Correlation: Src=0.15



(e) Server Resource Correlation: Src=0.50



(f) Server Resource Correlation: Src=0.70

Figure 4. Baseline and Extended Algorithm Performance Comparison

5. Summary and Work in Progress

In this paper, we defined a workload distribution problem for a computational grid with near-homogeneous multi-resource servers. First, servers in the grid have multiple resource capacities, and jobs submitted to the grid have requirements for each of those resources. Additionally, the servers are homogeneous in that any job submitted to the grid can be executed by any of the servers, but heterogeneous in their various resource configurations. We next investigated a load balancing approach to workload distribution for this grid. We showed how previous baseline load balancing policies for single resource systems failed to maintain a workload at each server which had a good affinity towards that server. We then proposed two orthogonal extensions based on the concept of resource balancing. The basic idea of resource balancing is that the local scheduler is more effective in utilizing the resources of the local server, if the total relative resource requirements of all jobs in a local work queue match the relative capacities of the server. Our simulation results show that our policy extensions provided a consistent 5–15% increase in system throughput performance over the baseline load balancing algorithms.

However, there is still significant room for improvement in the workload distribution approach. First, as the resource variance between servers grows, additional workload characteristics, beyond the total resource balance, must be taken into account when evaluating the workload for a given server. Specifically, we noted that the granularity of jobs in a local queue impacts the performance of the smaller servers. Intuitively, small jobs should be sent to small servers, and large jobs should be sent to large servers. Here, a large job is one that generally has large resource requirements, and a large server is one that generally has large resource capacities. Note that the size of a job is relative to the size of the server to which it is being compared. Our current work in progress is investigating refinements to the load balancing policies which improve the affinity of the local workload to the local server. Note that these investigations apply to single resource servers as well.

Second, there is a persistent performance gap between a central queue approach to workload distribution and our load balancing algorithms. Our conjecture is that even if the load is perfectly balanced, restricting a server, S_i, to execute jobs only from its local queue will increase the percentage of time that some of S_i's resources remain idle, when there may be a job in the queue of a different server, S_j, which would fit in to the idle resources of server S_i. These effects were noted in our simulations in that even when the servers were all nearly identical, and an equal load was being delivered to each server, the system throughput was still significantly below the performance of the central queue algorithm. Load balancing schemes were limited to about 85% of the throughput of the central queue scheme at all tested values of S_{rv} and S_{rc}, as seen in Figures 4(a)–(f).

We are further motivated to look at a more centralized approach by real-world computational grids, such as NASA's Information Power Grid (IPG) [6]. The current implementation of the IPG uses services from the Globus toolkit to submit jobs, query their status, and query the state of the grid resources. Globus uses a centralized directory structure, the Metacomputing Directory Service (MDS) to store information about the status of the metacomputing environment and all jobs submitted to the grid. Information in the MDS is used to assist in the placement of new jobs onto servers with appropriate resources within the grid. While this approach is currently being used in the IPG, there are questions about the scalability of such a centralized structure. For example, can a central structure handle hundreds of sites and thousands of jobs? How about fault tolerance? Our current work in progress is therefore investigating compromises between a single central queue and completely distributed queues. The general concept is to keep work close to the servers where it will most likely execute, and move work to a specific server when needed. Recent research in dynamic matching and scheduling for heterogeneous computing systems use similar approaches, along with heuristics for matching a job to idle server resources [12]. Our work in progress attempts to combine the centralized nature of current mapping approaches with our resource-balanced workload affinity approach.

6. Author Biographies

William (Bill) Leinberger is a Ph.D. student and Research Fellow in the Department of Computer Science and Engineering at the University of Minnesota. He received a BS in Computer and Electrical Engineering from Purdue University in 1984. His thesis covers topics in scheduling in the presence of multiple resource requirements. His other research interests include resource management for computational grids, and general topics in the area of high-performance computing architectures. Bill is currently on an educational leave from General Dynamics Information Systems, Bloomington, MN, where he has held positions as a hardware engineer, systems engineer, and systems architect in the area of special-purpose processing systems.

George Karypis is an assistant professor at the department of Computer Science and Engineering at the University of Minnesota. His research interests spans the areas of parallel algorithm design, data mining, applications of parallel processing in scientific computing and optimization, sparse matrix computations, parallel preconditioners, and parallel programming languages and libraries. His recent work has been in the areas of data mining, serial and parallel graph partitioning algorithms, parallel sparse solvers, and parallel matrix ordering algorithms. His research has resulted in the development of software libraries for serial and parallel graph partitioning (METIS and ParMETIS), hypergraph partitioning (hMEITS), and for parallel Cholesky factorization (PSPASES). He has coauthored several journal articles and conference papers on these topics and a book title ”Introduction to Parallel Computing” (Publ. Benjamin Cummings/Addison Wesley, 1994). He is a member of ACM, and IEEE.

Vipin Kumar is the Director of Army High Performance Computing Research Center and Professor of Computer Science at the University of Minnesota. His current research interests include high performance computing, parallel algorithms for scientific computing problems, and data mining. His research has resulted in the development of the concept of isoefficiency metric for evaluating the scalability of parallel algorithms, as well as highly efficient parallel algorithms and software for sparse matrix factorization (PSPACES), graph partitioning (METIS, ParMETIS), VLSI circuit partitioning (hMETIS), and dense hierarchical solvers. He has authored over 100 research articles, and coedited or coauthored 5 books including the widely used text book ”Introduction to Parallel Computing” (Publ. Benjamin Cummings/Addison Wesley, 1994). Kumar has given numerous invited talks at various conferences, workshops, national laboratories, and has served as chair/co-chair for many conferences/workshops in the area of parallel computing and high performance data mining. Kumar serves on the editorial boards of IEEE Concurrency, Parallel Computing, the Journal of Parallel and Distributed Computing, and served on the editorial board of IEEE Transactions of Data and Knowledge Engineering during 1993-97. He is a Fellow of IEEE, a member of SIAM, and ACM, and a Fellow of the Minnesota Supercomputer Institute.

Rupak Biswas is a Senior Research Scientist with MRJ Technology Solutions at NASA Ames Research Center. He is the Task Leader of the Algorithms, Architectures, and Applications (AAA) Group that performs research into technology for high-performance scientific computing. The AAA Group is part of the Numerical Aerospace Simulation (NAS) Division of NASA Ames. Biswas has published over 70 technical papers in major journals and international conferences in the areas of finite element methods, dynamic mesh adaptation, load balancing, and helicopter aerodynamics and acoustics. His current research interests are in dynamic load balancing for NUMA and multithreaded architectures, scheduling strategies for heterogeneous distributed resources in the IPG, mesh adaptation for mixed-element unstructured grids, resource management for mobile computing, and the scalability and latency analysis of key NASA algorithms and applications. He is a member of ACM and the IEEE Computer Society.

References

[1] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5):340–353, May 1986.

[2] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[3] D. G. Feitelson. Packing schemes for gang scheduling. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 65–88. Springer-Verlag, New York, 1996. LNCS.

[4] D. G. Feitelson. Memory usage in the lanl cm-5 workload. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 78–94. Springer-Verlag, New York, 1997. LNCS.

[5] D. Ferrari and S. Zhou. An empirical investigation of load indicies for load balancing applications. In *Proc. 12th Intl. Symposium on Computer Performance Modeling, Measurement, and Evaluation*, pages 515–528. North-Holland, Amsterdam, 1987.

[6] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[7] J. P. Jones. Implementation of the NASA Metacenter: Phase 1 report. Technical report, NASA Ames Research Center, October 1997. Technical Report NAS-97-027.

[8] L. V. Kale. Comparing the performance of two dynamic load distribution methods. In *Proc. Intl. Conference on Parallel Processing*, pages 77–80, August 1988.

[9] V. Kumar, A. Gramma, and V. Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994.

[10] W. Leinberger, G. Karypis, and V. Kumar. Job scheduling in the presence of multiple resource requirements. In *Supercomputing ’99*, November 1999.

[11] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *Proc. ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[12] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th IEEE Heterogeneous Computing Workshop (HCW’99)*, April 1999.

[13] C. McCann and J. Zahorjan. Scheduling memory constrained jobs on distributed memory computers. In *Proc. ACM SIGMETRICS Joint Intl. Conference on Measurement and Modeling of Computer Systems*, pages 208–219, 1996.

[14] E. W. Parsons and K. C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. Technical report, Computer Systems Research Institute, University of Toronto, October 1995.

[15] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.

[16] M. Y. Wu. Symmetrical hopping: A scalable scheduling algorithm for irregular problems. *Concurrency: Practice and Experience*, 7(7):689–706, October 1995.