A De-centralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments

Manish Arora and Sajal K. Das
Dept. of Computer Science & Engineering
The University of Texas at Arlington
Arlington, TX 76019-0015
{arora, das}@cse.uta.edu

Rupak Biswas
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035-1000
rbiswas@nas.nasa.gov

Abstract

In the past two decades, numerous scheduling and load balancing techniques have been proposed for locally distributed multiprocessor systems. However, they all suffer from significant deficiencies when extended to a Grid environment: some use a centralized approach that renders the algorithm unscalable, while others assume the overhead involved in searching for appropriate resources to be negligible. Furthermore, classical scheduling algorithms do not consider a Grid node to be N-resource rich and merely work towards maximizing the utilization of one of the resources. In this paper, we propose a new scheduling and load balancing algorithm for a generalized Grid model of N-resource nodes that not only takes into account the node and network heterogeneity, but also considers the overhead involved in coordinating among the nodes. Our algorithm is de-centralized, scalable, and overlaps the node coordination time with that of the actual processing of ready jobs, thus saving valuable clock cycles needed for making decisions. The proposed algorithm is studied by conducting simulations using the Message Passing Interface (MPI) paradigm.

1. Introduction

Computational Grids [1, 6] are typically a conglomeration of various resources with different owners, but make it possible for users to develop complex applications that access remote sites. Each of these sites (or nodes) could be a uni-processor machine, a symmetric multiprocessor cluster, a distributed memory multiprocessor system, or a massively parallel supercomputer. Each node consists of a number of heterogeneous *resources*; the heterogeneity being in the type and capability of each of its N-resources (e.g., number of processors, CPU speed, amount of memory, and so on). Perhaps the biggest advantage of a heterogeneous Grid environment over an isolated multiprocessor system is that it can offer resources to the user that are not locally available.

With the Grid becoming a viable high performance computing alternative to the traditional supercomputing environment, various aspects of effective Grid resource utilization are gaining significance. With its multitude of resources, a proper scheduling and efficient load balancing across the Grid can lead to improved overall system performance and a lower turn-around time for individual jobs. Classical load balancing algorithms [3, 5, 14, 20] address this problem by maximizing the utilization of a single resource (generally, CPU). But, the approach loses its merit for systems like the SUN Enterprise, the SGI Origin, and the IBM Regatta that offer multiple resources like shared memory, large disk farms, distinct I/O channels, and software licenses that can be independently allocated to different jobs.

Another area where classical and even recent N-resource load balancing approaches show their deficiency is in scalability—not many of them [10, 11, 12, 13, 14, 15, 18] can be scaled to the large number of processors in a Grid. This drawback is due either to the centralized approach of the algorithm [13, 18] or to the need for each node to have global system knowledge [11]. Also, most algorithms [10] either do not consider the overhead of searching for appropriate nodes or consider it to be negligible. This assumption is valid for tightly-coupled multiprocessor systems [16, 17, 19], but not for geographically distributed environments like the Grid.

The present work is targeted to the Grid model where each node is assumed to be a N-resource server and any job submitted to the Grid can be executed at any node. The only information our proposed algorithm needs before a node schedules a job is the communication latency between itself and its neighbors, thus making it fully scalable—an important consideration for a wide-area network like NASA's Information Power Grid (IPG) [2, 7]. The overhead involved in capturing the resource utilization status of a given node's neighbors before making a scheduling decision can be a major issue negating the advantages of job migration. Our algorithm therefore overlaps the time spent looking for appropriate nodes with the actual execution of the ready jobs, thus saving precious clock cycles. Also, since each Grid node (whether a single uni-processor machine or a multiprocessor system) can have its own independent scheduling algorithm, our technique does not overrule the local schedulers' job assignment policy. The class of problems we address is where jobs are computation-intensive and can be divided into totally independent sub-tasks with no communication between them.

We have conducted extensive experiments using the Message Passing Interface (MPI) paradigm and by simulating the job arrival rate. We compared the quality of load balance with the ideal case (where no overheads are involved) and found that our algorithm performs remarkably well in an heterogeneous Grid environment and gives encouraging results. The remainder of this paper is organized as follows: Section 2 describes our algorithm and presents pseudo codes of the key procedures; Section 3 discusses the experimental setup that we used to test and substantiate our claims, and interprets the results; and Section 4 concludes the paper.

2. Scheduling and load balancing

Two important aspects of any wide area network scheduler are its *transfer* [4, 15] and *location* [8, 9] policies. The transfer policy decides if there is a need to initiate load balancing across the system, and is typically threshold based. Using workload information, it determines 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). The location policy selects a partner node for a job transfer transaction. In other words, it locates complementary nodes to/from which a node can send/receive workload to improve overall system performance.

Location policies can be broadly classified as *sender-initiated* [4, 21], *receiver-initiated* [4, 12], or *symmetrically-initiated* [5, 15, 19]. Sender-initiated policies are those where heavily-loaded nodes search for lightly-loaded nodes while receiver-initiated policies are those where lightly-loaded nodes search for suitable senders. Symmetrically-initiated policies combine the advantages of these two by requiring both senders and receivers to look for appropriate partners.

Load balancing policies can also be classified on the basis of how up-to-date each node's knowledge is about the state of the system. *Dynamic* [16, 17] policies make decisions based on the current system state and can rapidly adapt to workload fluctuations. On the other hand, policies that use static information and are not amenable to changes in the workload are known as *static* [3] policies. However, dynamic policies incur the overhead of communicating among the system nodes to keep them informed about the state of the system.

In this section, we describe our scheduling and load balancing algorithm for N-resource Grid environments. It is dynamic, sender-initiated, and completely de-centralized. The last feature makes it extremely scalable for Grid environments. A remarkable property of our algorithm is that it uses a smart search strategy for finding partner nodes. It also overlaps this decision making process of a node with the actual execution of ready jobs, thereby saving precious processor cycles.

2.1. Preliminaries

Before discussing the algorithm, let us introduce the concepts of *Internal* and *External* queues, which we assume exist in each Grid node. The *Internal Queue* of a node consists of the ready jobs which would be executed by this particular node only. Let τ be the time when the tasks were last mapped, $a(t_j)$ be the arrival time of task t_j, and $e(t_j)$ be the time t_j starts executing. Then, the jobs in the *Internal Queue* are those that have been mapped and scheduled to this node, and are either being executed (Eq. 1) or are ready to be executed (Eq. 2); they would never be delegated to any other node:

$$\{t_j \mid a(t_j) \leq \tau, e(t_j) \leq \tau\} \qquad (1)$$

$$\{t_j \mid a(t_j) \leq \tau, e(t_j) > \tau\} \qquad (2)$$

Instead, the *External Queue* of a node consists of jobs which have been initially submitted to this node by a user, but are yet to be mapped and scheduled for execution (Eq. 3):

$$\{t_j \mid a(t_j) > \tau, e(t_j) > \tau\} \qquad (3)$$

Let us now enumerate the key notations we will be using throughout the paper to explain our algorithm:

- P_i: Grid node i
- P_i^j: The j-th resource of P_i
- J_k: Job k
- J_k^j: Ideal requirement for the j-th resource by J_k
- $Neigh(P_i)$: Immediate neighbors of P_i
- $Comp_i(t)$: Time needed by P_i to empty its *Internal Queue* assuming no more jobs are assigned to it after time t
- $Comm_i^j$: Communication latency between P_i and P_j
- ExQ_i: Number of jobs in the *External Queue* of P_i

We assume that each Grid node has knowledge about the communication latency between itself and all of its neighbors; i.e., each node P_i knows $Comm_i^j$, $\forall j \in Neigh(P_i)$. Not only does this make the algorithm highly scalable, it also allows the network to conveniently accommodate any changes in its topology.

We also postulate that each incoming job knows its requirements for each of the resources available at a node. In order to generalize this concept, we define N-resource jobs and N-resource nodes/servers. Each job J_k looks for a node P_i with resources P_i^0, P_i^1, ..., P_i^{N-1}, such that it meets its requirement for each resource type, J_k^0, J_k^1, ..., J_k^{N-1}. The algorithm described below would be executing on every node of the Grid.

2.2. Proposed algorithm

Whenever a job is submitted by a user to a node P_i, procedure *Main* (Fig. 1) invokes procedure *NeedForTriggering* (Fig. 2) to make a decision whether the job need to be migrated. If the job ought to be migrated to another node, a request is sent to all nodes $j \in Neigh(P_i)$, provided $2 \times Comm_i^j \leq Time_{IQ}$. This condition implies that the status request to the neighboring nodes and their responses should be received before the *Internal Queue* is emptied (denoted by $Time_{IQ}$). This strategy avoids any wastage of the node's resources; the inequality overlaps the task of looking for appropriate nodes with the actual processing of the *Internal Queue*, thus hiding the overhead.

```
Procedure Main
  Repeat forever
    If ($\alpha \leftarrow$ new job submitted)
      $Time \leftarrow$ Current System Time (CST)
      NeedForTriggering ($\alpha, Time$)
      If (NeedForTriggering returns TRUE)
        $Time_{IQ} \leftarrow$ Time to empty Internal Queue
        $\forall j \in Neigh(P_i)$
          If ($2 \times Comm_i^j \leq Time_{IQ}$)
            Request ($j, Comm_i^j, Time_{IQ}$)
        Receive ($Time_{IQ}$)
        Balance (S, R)
      End If
    End If
  End Repeat
End Main
```

Figure 1. Procedure *Main*

Refer to Figures 2 and 3 for the triggering policy we have incorporated into our algorithm. It is based on the simple heuristic that greater the load at a node, the less inclined would it be to accept future loads. Within a time window of $Comp_i(\tau)$, triggering is initiated if the traffic burst is more than admissible; however, higher the resource usage, the smaller is the traffic burst that a node will accommodate (Fig. 3).

```
Procedure NeedForTriggering ($\alpha, Time$)
  $\delta \leftarrow \delta + \alpha$     /* $\delta$ is Cummulative Load */
  If (CST$-Time \leq Comp_i(\tau)$)
    If ($\delta \geq$ Admissible Load at $\tau$)
      $\tau \leftarrow$ CST
      Return TRUE
  Else
    Commit $\delta$ to Internal Queue
    $\tau \leftarrow$ CST
    $\delta \leftarrow 0$
    Return FALSE
  End If
End NeedForTriggering
```

Figure 2. Procedure *NeedForTriggering*

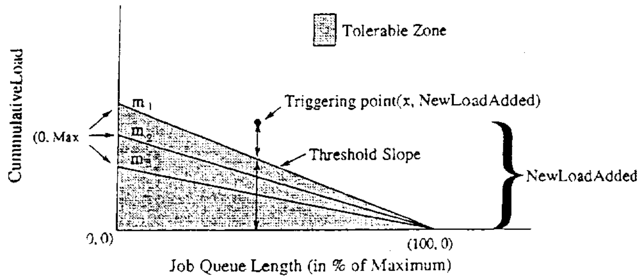

Figure 3. Value of δ when Job Queue is x% full

A node, having received a request to send the status of its resources, packs the information about their current utilization and sends it back to the requesting node along the route the request came (Fig. 4). This route is also piggybacked to the node which needs to migrate load. Besides replying to requests, a node also recursively pings its neighbors for their resource status if its database says that the total round-trip latency between the sender and its neighbor would be less than $Time_{IQ}$. This allows the time required to look for additional resources be hidden under processing.

```
Procedure Request ($i, \gamma, Time_{IQ}$)
  Create Set S
  S.Route $\leftarrow$ Route followed to reach $i$
  S.ResStatus $\leftarrow$ Current usage of
                         $\{P_i^0, P_i^1, \ldots, P_i^{N-1}\}$
  MPI_Send (S to $i$)
  $\forall j \in Neigh(P_i)$
    If ($2 \times (\gamma + Comm_i^j) \leq Time_{IQ}$)
      Request ($j, \gamma + Comm_i^j, Time_{IQ}$)
    End If
End Request
```

Figure 4. Procedure *Request*

Figure 5 shows the pseudo code for procedure *Receive*. The sender waits for time $Time_{IQ}$ to get replies from the nodes that have been queried for the status of their resources.

```
Procedure Receive($Time_{IQ}$)
  While ($Time \leq Time_{IQ}$)
    MPI_Receive (S)
  End While
  R ← Number of replies
  Return R
End Receive
```

Figure 5. Procedure *Receive*

Figure 6 shows our procedure to schedule the jobs soon after $Time_{IQ}$ elapses. Without loss of generality, we can assume that $0.0 \leq P_i^j, J_k^j \leq 1.0$, $0 \leq j \leq N-1$. Let M_i^k be a *match* variable which defines the number of resources in node P_i that fulfill the requirements of job J_k. If $bool(J_k^j \leq P_i^j)$ is 1 and $bool(J_k^j > P_i^j)$ is 0, then we can formally define M_i^k as

$$M_i^k \leftarrow \sum_{j=0}^{N-1} bool(J_k^j \leq P_i^j) \qquad (4)$$

Clearly, $0 \leq M_i^k \leq N$. Now, let us define matrices T and C, and vector V, as described in steps 1, 2, and 3 of procedure *Balance* (Fig. 6). Intuitively, the u-th row and k-th column of T gives the number of resources in node P_i that meets the requirements of job J_k; the k-th entry of V gives the number of nodes which satisfy the minimum requirements of J_k; and element $C_{u,j}$ denotes that there is a common node that satisfies the requirements of both J_u and J_j, and that there might be a conflict while scheduling them.

Another possible scenario is when the *set* of nodes that satisfy the requirements of J_u is a subset of the set which satisfies the requirements of J_j; in such cases, giving preference to J_j might leave J_u with no viable option. To avoid such cases, our algorithm first schedules jobs that have the fewest choices. $T_{(u,min(V_j))}$ in step 4.1 of Fig. 6 corresponds to the job J_{min} that has the minimum number of nodes it can be mapped to. The variable z indicates the node P_z to which J_{min} can be delegated. Step 4.2 checks matrix C and, in case there is another job that can be mapped to P_z, chooses a different z for J_{min}, if possible. Finally, J_{min} is mapped and scheduled to P_z. This mechanism continues until all jobs have been scheduled or until no more can be mapped because of the lack of resources.

3. Experimental study

Here we describe the metrics used to gauge the performance of our scheduling and load balancing algorithm, the setup we had for our experiments, the simulation results that were obtained, and the conclusions we can draw from them.

3.1. Performance metrics

We analyze the performance of our algorithm using a parameter called *Normalized Performance*, η (defined in Eq. 5). Basically, η is the effectiveness of the load balancing strategy. It is a comprehensive metric as it considers both the initial load balance as well as the load balancing overheads:

$$\eta = \frac{T_{no} - T_{my}}{T_{no} - T_{lb}} \qquad (5)$$

Here, T_{no} is the time to completely process all the jobs on a uniprocessor machine; T_{lb} is the time required by one processor divided by the total number of processors, thus providing the runtime with ideal load balancing; and T_{my} is the time needed by our algorithm to balance the load and execute all the jobs. Clearly,

$$\text{if } T_{my} \rightarrow T_{lb}, \quad \text{then } \eta \rightarrow 1 \qquad (6)$$

$$\text{if } T_{my} \rightarrow T_{no}, \quad \text{then } \eta \rightarrow 0 \qquad (7)$$

These two conditions imply that higher the value of η, the better is the load balancing; the ideal case being $\eta = 1$.

3.2. Experimental setup

The experimental results reported in this paper were obtained by using an MPI implementation of our proposed algorithm. It is worth mentioning here that the various parameters of our algorithm were varied following a Poisson distribution. Their respective mean values are given in Table 1.

Table 1. Variables used in the experiments

Variables	Mean	Simulated by
Processing Power Requirements	2–16	50 floatingpoint multiplications per unit
Memory Requirements	2–16	1KB of memory allocated & freed per unit
I/O Requirements	2–16	1KB of data written to disk per unit
Network Latency	5–11	sleep(3) per unit
Node Degree	5	number of neighboring nodes

```
Procedure Balance(S, R)
1.   Using $S$, define matrix $T$ of dimensions $ExQ_i \times R$ where $T_{u,k} \leftarrow M_u^k$
2.   Define vector $V$ of dimension $ExQ_i$ where $V \leftarrow \sum_{u=1}^{R} bool(T_{u,k} = N)$, $1 \leq k \leq ExQ_i$
3.   Define matrix $C$ of dimensions $ExQ_i \times ExQ_i$ where
             $C_{l,k} \leftarrow C_{k,l} \leftarrow 1$, if $T_{k,j} = T_{l,j} = N$; 0, otherwise; $1 \leq l, k \leq ExQ_i$, $1 \leq j \leq R$
4.   Repeat until (no more jobs can be mapped)
4.1      $z \leftarrow u \mid T_{(u,min(V_j))} = N$, $1 \leq u \leq R$, $1 \leq j \leq ExQ_i$
4.2      If ($C_{(min(V_j),k)} = 1$, $1 \leq k \leq ExQ_i$)
           Choose another $z$, if possible
4.3      Assign $J_{min(V_j)}$ to node $P_z$, $1 \leq j \leq ExQ_i$
4.4      Remove row $min(V_j)$, $1 \leq j \leq ExQ_i$ and column $z$ from $T$
End Balance
```

Figure 6. Procedure *Balance*

Experiments were conducted for three different values of *Max* (15, 20, and 25) (see Fig. 3), and repeated for 1-, 2-, and 3-resource nodes. The following three inequalities give the relationships between the m_i's, where each m_i refers to the slope of the line joining the co-ordinates (0, *Max*) and 100, 0) (Fig. 3):

$$m_1, m_2, m_3 < 0 \quad (8)$$

$$m_1 < m_2 < m_3 \quad (9)$$

$$\mid m_1 \mid > \mid m_2 \mid > \mid m_3 \mid \quad (10)$$

3.3. Simulation results

We have conducted extensive experiments to evaluate the performance of our algorithm and help us substantiate our approach. Figures 7 through 9 illustrate the results obtained from the study.

To verify that our algorithm works well for completely heterogeneous systems, we divided the experiments into three groups. The *first* set of experiments was run on systems where heterogeneity was in the capabilities of the N-resources of a node; thus, the communication latency between all neighboring nodes was constant. The *second* set involved keeping the node capability constant and varying only the communication latency between the nodes. Finally, the *third* set of experiments combined the above two approaches, thereby exposing a totally heterogeneous setup to various load conditions (that were varied by changing the job arrival rate and the load associated with each job). Each set of experiments was repeated for 1-, 2-, and 3-resource nodes. The objective was to evaluate the algorithm thoroughly by taking various scenarios of heterogeneity into consideration.

Results for the first set (where only the capabilities of the N-resources of a node are varied while keeping all other factors unchanged) are summarized by the graphs in Fig. 7. The horizontal axis represents the *Mean Node Capacity* of the network which can be defined as the mean value used for the capacity of each of the resources in a node (all resource having the same mean). Increasing the resource capability of the nodes without changing the job resource requirements effectively reduces the granularity of the latter. As depicted, any increase in node capability increases η. However, as the threshold slopes (m_i's) become steeper, η decreases. This is because the frequency of triggering the load balancing algorithm is reduced.

In the second set of experiments, the *Mean Node Capacity* was held constant while varying the communication latency. The results presented in Fig. 8 show that η decreases with increasing communication cost. As in the previous set, the algorithm performs best when the absolute value of the threshold slope is the smallest (m_3 in this case).

For the final set of experiments, we vary the input load for a setup which has a heterogeneous mix of resource capability and communication latency. This was repeated for 1-, 2-, and 3-resource job specification for a 3-resource node. Figure 9 shows that the execution time decreases as we get more specific about job requirement.

4. Conclusions

In this paper, we presented a highly de-centralized, distributed, and scalable algorithm for scheduling tasks and load balancing resources in heterogeneous Grid environments. Our algorithm takes into consideration the overheads of coordination and communication between the Grid nodes which were assumed to be N-resource servers that varied in their respective capacities across resources. The goal was to assign each node a job which would utilize its resources in the best possible manner, thus providing an effective scheduling and resource management strategy. We introduced a new load balance triggering policy based on the endurance of a node reflected by its current queue length. Also, our algorithm overlaps the time needed for

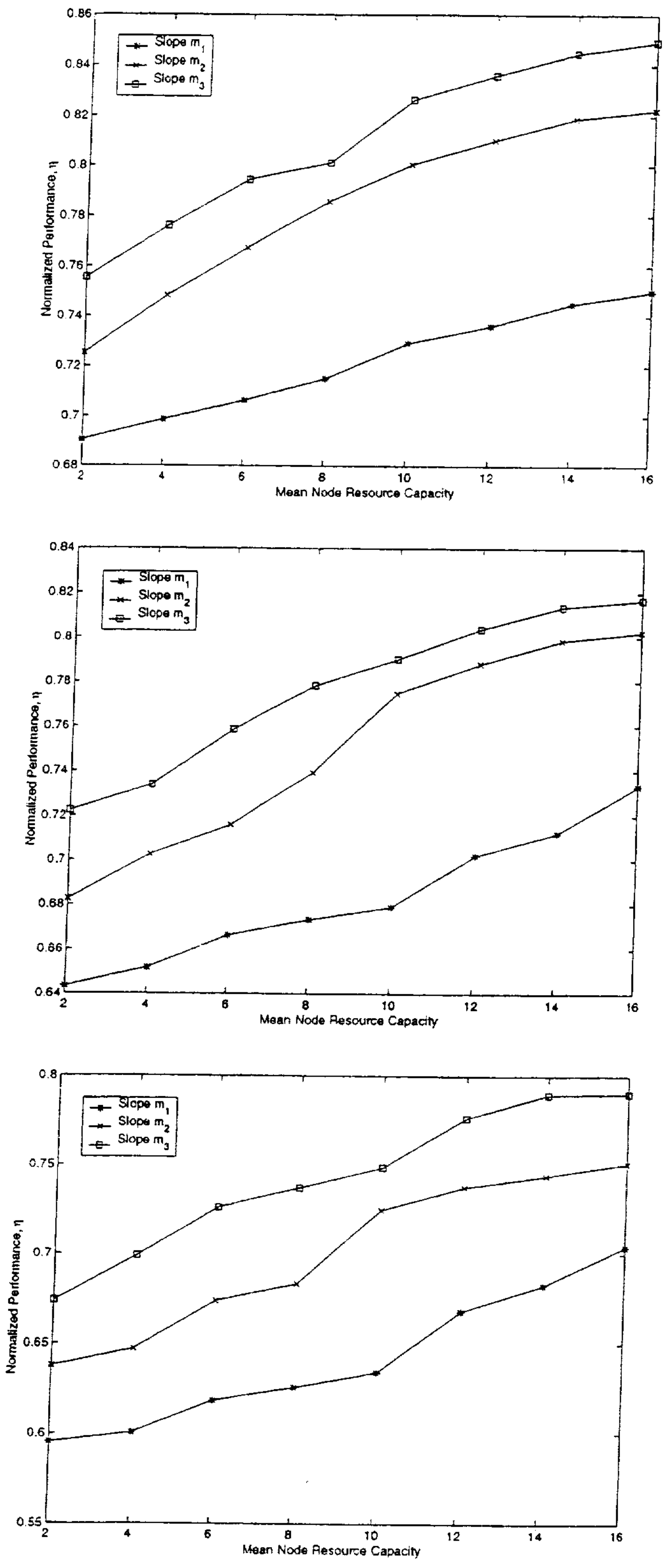

Figure 7. *Normalized Performance* (η) **vs.** *Mean Node Capacity* **for 1-, 2-, and 3-resource nodes (top to bottom)**

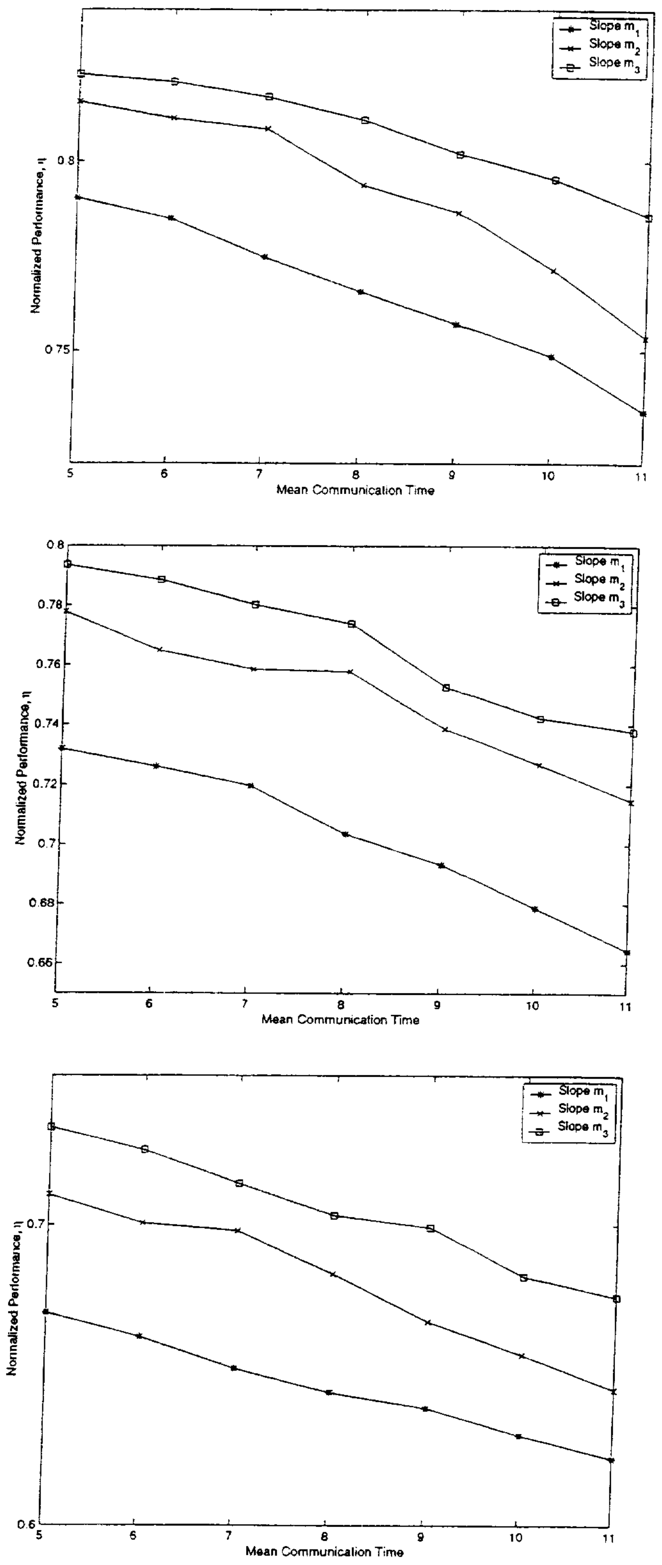

Figure 8. *Normalized Performance* (η) **vs.** *Mean Communication Time* **for 1-, 2-, and 3-resource nodes (top to bottom)**

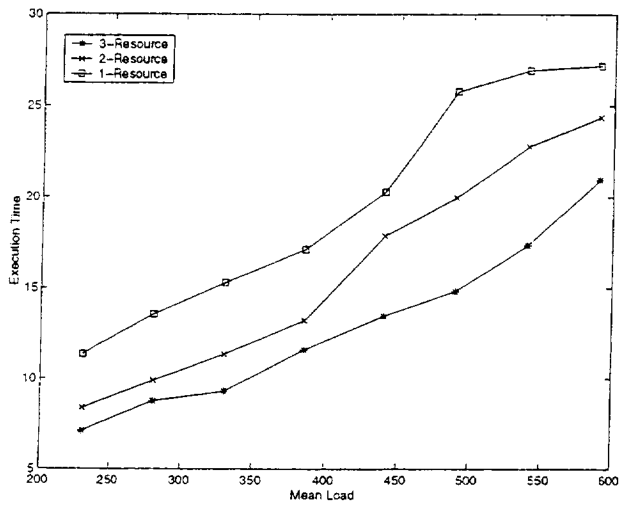

Figure 9. *Execution Time* (T_{my}) **vs.** *Average Load*

various communication overheads with that of executing the jobs already committed to the nodes, making the effective time for overheads virtually zero. The algorithm has been discussed in detail with pseudo codes being provided for all the major modules of the algorithm.

To substantiate our claims, a comprehensive experimental study was conducted using the Message Passing Interface (MPI) paradigm. Heterogeneity in resource capabilities and communication latency was maintained while repeating the set of experiments for 1-, 2-, and 3-resource jobs and nodes. The *Normalized Performance* parameter was 0.79 for 3-resource nodes and as high as 0.85 for 1-resource nodes. These excellent performance levels could be attained only by overlapping the various overheads with the actual execution of the jobs.

Acknowledgements

The work of the first two authors was partially supported by NASA Ames Research Center under Cooperative Agreement Number NCC 2-5395.

References

[1] Global Grid Forum. http://www.gridforum.org.

[2] Information Power Grid. http://www.ipg.nasa.gov.

[3] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributing Computing*, 61(6):810–837, June 2001.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6(1):53–68, 1986.

[5] [illegible]. Feng, D. Li, H. Wu, and Y. Zhang. A dynamic load balancing algorithm based on distributed database system. In *Proc. 4th Intl. Conf. on High Performance Computing in the Asia-Pacific Region*, pages 949–952, Beijing, China, May 2000.

[6] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, CA, 1999.

[7] W. E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of NASA's Information Power Grid. In *Proc. 8th Intl. Symp. on High Performance Distributed Computing*, pages 197–204, Redondo Beach, CA, August 1999.

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

[9] V. Kumar, A. Grama, 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 *Proc. Supercomputing*, Portland, OR, November 1999.

[11] W. Leinberger, G. Karypis, V. Kumar, and R. Biswas. Load balancing across near-homogeneous multi-resource servers. In *Proc. Heterogeneous Computing Workshop*, pages 60–71, Cancun, Mexico, May 2000.

[12] H. Lin and C. Raghavendra. A dynamic load-balancing policy with a central job dispatcher (LBC). *IEEE Trans. Software Engineering*, 18(2):148–158, February 1992.

[13] J. Liu and V. A. Saletore. Self-scheduling on distributed-memory machines. In *Proc. Supercomputing*, pages 814–823, Portland, OR, November 1993.

[14] A. Rajagopalan and S. Hariri. An agent based dynamic load balancing system. In *Proc. Intl. Symp. Autonomous Decentralized Systems*, pages 164–171, 2000.

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

[16] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.

[17] T. Tzen and L. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Trans. Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[18] J. Weissman. *Scheduling Parallel Computations in a Heterogeneous Environment.* PhD thesis, Dept. of Computer Science, Univ. of Virginia, August 1995.

[19] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel and Distributed Systems*, 9(4):979–993, September 1993.

[20] Y. Zhang, H. Kameda, and S.-L. Hung. Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems. *Proceedings of IEEE: Computers and Digital Techniques*, 144(2):100–106, March 1997.

[21] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. Software Engineering*, 14(9):1327–134[illegible], September 1988.