A Cluster-based Dynamic Load Balancing Middleware Protocol for Grids

Kayhan Erciyes[1], Orhan Dagdeviren[1], and Reşat Ümit Payli[2]

[1] Izmir University
Computer Engineering Department
Uckuyular, Izmir, Turkey
{kayhan.erciyes,orhan.dagdeviren}@izmir.edu.tr
[2] Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology
Indiana University-Purdue University
Indianapolis, Indiana 46202, U.S.A.
rpayli@iupui.edu

Abstract. The load distribution in a grid may vary leading to the bottlenecks and overloaded sites. We describe a hierarchical dynamic load balancing protocol for Grids. The Grid consists of clusters and each cluster is represented by a coordinator. Each coordinator first attempts to balance the load in its cluster and if this fails, communicates with the other coordinators to perform transfer or reception of load. This process is repeated periodically. We analyze the correctness, performance and scalability of the proposed protocol and show from the simulation results that our algorithm balances the load by decreasing the number of high loaded nodes in a grid environment.

Keywords : load balancing, clustering, hierarchical protocol, grid, e-science.

1 Introduction

Future e-science applications will require efficient processing of the data [4] where storages and processors may be distributed among the collaborating researchers. A computational Grid consists of heterogenous computational resources, possibly with different users, and provide them with remote access to these resources [5–7] and it is an ideal computing environment for e-science applications [8–10]. The Grid has attracted researchers as an alternative to supercomputers for high performance computing. One important advantage of Grid computing is the provision of resources to the users that are locally unavailable. Since there are multitude of resources in a Grid environment, convenient utilization of resources in a Grid provides improved overall system performance and decreased turnaround times for user jobs [11]. Users of the Grid submit jobs at random times. In such a system, some computers are heavily loaded while others have available processing capacity. The goal of a load balancing protocol is to transfer the

load from heavily loaded machines to idle computers, hence balance the load at the computers and increase the overall system performance. Contemporary load balancing algorithms across multiple/distributed processor environments target the efficient utilization of a single resource and even for algorithms targeted towards multiple resource usage, achieving scalability may turn out difficult to overcome.

A major drawback in the search for load balancing algorithms across a Grid is the lack of scalability and the need to acquire system-wide knowledge by the nodes of such a system to perform load balancing decisions. Scalability is an important requirement for Grids like NASA's Information Power Grid (IPG) [12], TeraGrid [13], EGEE [14] and NORDUGRID [15]. Some algorithms have a central approach, yet others require acquisition of global system knowledge. Scheduling over a wide area network requires *transfer* and *location* policies. Transfer policies decide *when* to do the transfer[16] and this is typically based on some threshold value for the load. The location policy [17] decides where to send the load based on the system wide information. Location policies can be *sender initiated* [18–20] where heavily loaded nodes search for lightly loaded nodes, *receiver initiated* [21] in which case, lightly-loaded nodes search for senders or *symmetrical* where both senders and receivers search for partners [22]. Some agent based and game theoretic approaches were also proposed previously [23–26]. Load balancing across a Grid usually involves sharing of data as in an MPI (Message Passing Interface) *scatter* operation as in [27, 28]. MPICH-G2, is a Grid-enabled implementation of MPI that allows a user to run MPI programs across multiple computers, at the same or different sites, using the same commands that would be used on a parallel computer [29].

In this study, we propose a dynamic and a distributed protocol based on our previous work [30] with major modifications and detailed test results to perform load balancing in Grids. The protocol uses the clusters of the Grid to perform local load balancing decision within the clusters and if this is not possible, load balancing is performed among the clusters under the control of clusterheads called the *coordinators*. We show that the protocol designed is scalable and has favorable message and time complexities.

The rest of the paper is organized as follows: In Section 2, the proposed protocol including the coordinator and the node algorithms is described with the analysis. In Section 3, the implementation of the protocol using an example is detailed and test results using Indiana University Grid environment are analyzed in Section 4 and Section 5 has the concluding remarks along with discussions.

2 The Protocol

We extend the load balancing protocol [30] for Grids to achieve a more balanced load distribution. We use the same *daisy* architecture shown in Fig. 1, which is shown to be more scalable for group communication among other well-known architectures [31]. In this architecture, coordinators are the interface points for the nodes to the ring and perform load transfer decisions on behalf of the nodes in

their clusters they represent. They check whether load can be balanced locally and if this is not possible, they search for potential receivers across the Grid same as in [30]. Additionally, by using the advantage of a daisy architecture, a token circulates the coordinator ring and can carry the global load information to all coordinators when it is needed. By using this information, coordinators can distribute the load in a more balanced way and also know the time to finish. These extensions are detailed in Section 2.1.

In this paper, we categorize the *Load* to LOW, MEDIUM and HIGH classes. The node is LOW when it can accept load from other nodes. A node is HIGH loaded when it is detected to be higher than the *upper threshold* as defined in the previous protocol [30]. The main difference is in the MEDIUM load definition. A node is MEDIUM if it has a load above the maximum limit of LOW load and can accept load from other nodes to reach the *upper threshold* , which is called MEDIUM_MAX. Thus, MEDIUM loaded nodes do not need to give loads to LOW loaded nodes since they are not overloaded. Beside the categorization of nodes, clusters can be classified as LOW, MEDIUM and HIGH as regards to the sum of all local node loads in order to get a global point of view. Based on these definitions, we state that the two main targets of the load balancing protocol is to distribute the excessive loads of HIGH loaded nodes and group of nodes which are called clusters, to LOW and MEDIUM nodes and clusters to reach a global stable load distribution and to consider the time and message complexities in this operation by eliminating unnecessary transmissions.

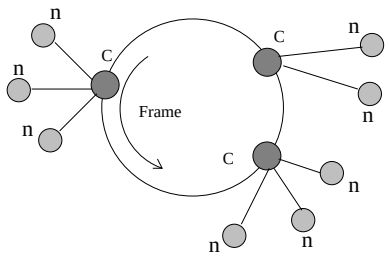

Fig. 1. The Load Balancing Model for a Grid

2.1 Coordinator Algorithm

The primary functions of the coordinator is to monitor loads of the nodes in its cluster, initiate transfer of nodes from HIGH to LOW and MEDIUM nodes locally if possible and search for LOW and MEDIUM nodes across the Grid if there are no local matches. The pseudocode is given in Alg. 1.

The coordinator of a cluster is activated periodically and starts the protocol by sending a *Coord_Poll* message to every node in its cluster. It waits to receive all *Node_State* messages from all ordinary nodes belonging to the same cluster.

Algorithm 1 Load balancing algorithm for coordinator

initially M: set of all member nodes of the coordinator the node
    C: set of all coordinators
    $load_M$: received load array of coordinator from M
    T_M: total load of $load_M$
    token_received: the token received state, initially equals to false.
    next_coordinator: next coordinator (cluster leader) on the ring.
    Legend : □ State ∧ *input_message* ⟶ actions
loop
  □ IDLE ∧ $Tout$ ⟶ **multicast** $Coord\_Poll$ to M
    current_state ← WAIT_POLL
  □ IDLE ∧ $Xfer\_Token$ ⟶ **multicast** $Coord\_Poll$ to M
    current_state ← WAIT_POLL
    token_received ← true
  □ WAIT_POLL ∧ *Node_State* ⟶ **if** all loads are received from M **then**
      call local_load_balance_procedure($load_M$)
      if T_M is MEDIUM_MAX **then multicast** End to M
        current_state ← IDLE
      else send $Xfer\_Token$ to the *next_coordinator*
        current_state ← WAIT_XFER_MES
      end if
      if *token_received*=true **then forward** $Xfer\_Token$ to the *next _coordinator*
      end if
  □ WAIT_XFER_MES ∧ $Xfer\_Token$ originated from this coordinator ⟶
    calculate global transfers similar to local_load_balance_procedure
    if no global transfer **then multicast** End to M
      current_state gets IDLE
    else multicast $Xfer\_List$ to C
      if T_M is HIGH **then** *current_state* ← WAIT_LD
      else T_M is HIGH **then** *current_state* ← WAIT_XLD
      end if
    end if
  □ WAIT_XFER_MES ∧ $Xfer\_List$ ⟶ **if** no global transfers **then multicast** End to M
    end if
    if T_M is HIGH **then** *current_state* ← WAIT_LD
    else *current_state* ← WAIT_XLD
    end if
  □ WAIT_XACK ∧ X_ACK ⟶ **send** In_ACK
    if no other transfer exists **then** *current_state* ← IDLE
    else *current_state* ← WAIT_LD
    end if
  □ WAIT_XLD ∧ X_Load ⟶ **send** In_Load
    current_state ← WAIT_INACK
  □ WAIT_XACK ∧ In_ACK ⟶ **send** X_ACK
    if no other transfer exists **then** *current_state* ← IDLE
    else *current_state* ← WAIT_XLD
    end if
end loop

After receiving all nodes from its cluster, it checks whether there are any local matching nodes by executing the pseudocode in Alg. 2. If there is a match, it sends $Coord\_Xfer$ message to HIGH node to initiate the local transfer. Before starting a local transfer, the coordinator node sets the load of both parties to end configuration. Also it calculates the overall cluster load by adding the loads of each node. If the cluster state is HIGH loaded, it must send or forward a $Xfer\_Token$ message to its next coordinator on the ring. If it hasn't received a $Xfer\_Token$ message yet, it sends a new $Xfer\_Token$ message to its next coordinator otherwise picks up the received $Xfer\_Token$ with the smallest $node\_id$ and forwards it. $Xfer\_Token$ message includes the overall cluster load and it is used for global load balancing.

Algorithm 2 *Local load balance procedure*

```
initially: load[i]: the load value of i^th node
input: load array
for each load[i] do
  if load[i] > MEDIUM_MAX then
    excessive_load ← load[i]-MEDIUM_MAX
    for each load[j] except j=i do
      if excessive_load = 0 then
        break the loop
      end if
      if load[j] < MEDIUM_MAX then
        if MEDIUM_MAX - load[j] >= excessive_load then
          excessive_load ← 0
          transferable_load ← excessive_load
        else if
         then
          excessive_load ← excessive_load-(MEDIUM_MAX-load[j])
          transferable_load ← MEDIUM_MAX-load[j]
        end if
      end if
      load[i] ← load[i]-transferable_load
      load[j] ← load[j]-transferable_load
    end for
  end if
  create a new transfer T(from:i, to:j, load:transferable_load)
  start the transfer T
end for
output: load array
```

After the $Xfer\_Token$ of any coordinator is circulated across the ring, the originator node broadcasts a $Xfer\_List$ message which consists of all overall cluster loads. It is important to note that the local transfers can occur in parallel with this event, since they would not affect the overall cluster loads. After receiving the $Xfer\_List$, all coordinators create the global transfers by an al-

Algorithm 3 Load balancing algorithm for node

```
1: initially current_state: the current state of the node
2:     current_state ← IDLE
3:     load: node's load
4:     c: node' coordinator
5:     Legend : □ State ∧ input_message ⟶ actions
6: loop
7:   □ IDLE ∧ Coord_Poll ⟶ send load to c
8:     if load is HIGH then current_state ← WAIT_XFER
9:     else if load is MEDIUM_MAX then current_state ← IDLE
10:    end if
11:  □ WAIT_LD ∧ Node_Load ⟶ send Xfer_ACK
12:  □ WAIT_LD ∧ End ⟶ current_state ← IDLE
13:  □ WAIT_XFER ∧ Coord_Xfer ⟶ send Node_Load
14:    current_state ← WAIT_ACK
15:  □ WAIT_XFER ∧ End ⟶ current_state ← IDLE
16:  □ WAIT_ACK ∧ Xfer_ACK ⟶ current_state ← WAIT_XFER
17: end loop
```

gorithm as in Alg. 2. If the coordinator has no global transfer or completed its global transfers and has finished all local transfers, it broadcasts an *End* message to its cluster in order to finish the load balancing operation.

2.2 Node Algorithm

The node algorithm is same as our previous protocol [30] with minor modifications. The node process replies by sending its load status to the coordinator in return for the *Coord_Poll* message from the coordinator. If its load is HIGH, it waits for initiation of transfer from the coordinator. When it receives the ($Coord\_Xfer$) message meaning transfer can be started, it sends the excessive load to the receiver sent by the coordinator and waits for an acknowledgement to confirm that the transfer was successful. The pseudocode of this process is given in Alg. 3. Different than the previous protocol, if the load of the node is LOW or MEDIUM, it will wait for a transfer from the HIGH node or wait the *End* message.

2.3 Analysis

Let us assume that messages in the algorithms given in Alg. 1, Alg. 2 and Alg. 3 will arrive in finite time from a source to destination or multiple destinations. We also assume k, d are upperbounds on the number of clusters and diameter of a cluster respectively.

Observation 1 *In coordinator algorithm given in Alg. 1 and node algorithm given in Alg. 3, each coordinator and node starts in $IDLE$ state and should end in $IDLE$ state.*

Lemma 1. *Coordinator algorithm given in Alg. 1 is free from deadlock and starvation.*

Proof. We prove the lemma by contradiction. From Observation 1, a coordinator starts executing the Alg. 1 in $IDLE$ state and should end in $IDLE$. If there exists a deadlock or starvation, a coordinator should be any state other than $IDLE$ after executing the algorithm. We assume the contrary that a coordinator is not in $IDLE$ state at the end of the algorithm. In this case, a coordinator may be in any of $WAIT\_POLL$, $WAIT\_XFER\_MES$, $WAIT\_ACK$, $WAIT\_LD$, $WAIT\_LD$, $WAIT\_INACK$, and $WAIT\_XLD$ states. We investigate all these cases below:

- $WAIT\_POLL$: A coordinator makes a state transition from $WAIT\_POLL$ to $IDLE$ or $WAIT\_XFER\_MES$ state when it receives all $Node_State$s from cluster members. From Observation 1, since all ordinary cluster member nodes start executing the algorithm in $IDLE$ state, they reply the $Coord\_Poll$ message with $Node\_State$ as given in line 7 of Alg. 3. Thus a coordinator can not wait endlessly in this state.
- $WAIT\_XFER\_MES$: In this state, a coordinator node makes a state transition to $WAIT\_ LD$, $WAIT\_XLD$, or $IDLE$ state if it receives a $Xfer\_List$ or its own circulated $Xfer\_ Token$ message. If it receives its circulated $Xfer\_Token$ message, the coordinator multicasts a $Xfer\_List$ message to all coordinators. Since at least one of the $Xfer\_Token$ messages will be circulated the ring, a coordinator will receive its own $Xfer\_Token$ or multicasted $Xfer\_List$. Thus a coordinator will make a state transition from this state.
- $WAIT\_LD$: When a coordinator is in $WAIT\_LD$ state, it waits for In_Load message from ordinary nodes. To transfer the load, a coordinator sends $Coord\_Xfer$ message to the destination node with $HIGH$ load. When an ordinary node is $HIGH$ loaded, it will make a state transition to $WAIT\_XFER$ state and and it replies the $Coord\_Xfer$ message with a $Node\_Load$. So a coordinator will make a state transition from $WAIT\_LD$ state to $WAIT\_XACK$ since it receives the In_Load message.
- $WAIT\_ACK$: A coordinator which is in this state will make a state transition to $IDLE$ state upon receiving the last X_ACK message. A LOW or $MEDIUM$ loaded ordinary node will make a state transition to $WAIT\_LD$ state when it receives $Coord\_Poll$ message and it replies the $Node\_Load$ message with a $Xfer\_ACK$ message. So a coordinator will make a state transition to $IDLE$ state in finite time.
- $WAIT\_XLD$: This case is similar to the case of a coordinator in $WAIT\_LD$ state. When the coordinator in $WAIT\_XLD$ state receives the X_Load message it will make a state transition to $WAIT\_INACK$ state.
- $WAIT\_INACK$: When the coordinator receives the last In_ACK message, it makes a transition to IDLE state. This case is similar to the case of a coordinator in $WAIT\_ACK$ state.

  As explained from all the cases listed above, a coordinator node can not finish the algorithm in one of listed states. Thus we contradict with our

assumption. The coordinator node will be $IDLE$ state after executing the algorithm and coordinator algorithm is free from deadlock and starvation.

Lemma 2. *A coordinator will multicast End message to its cluster members before ending the execution of the algorithm.*

Proof. A coordinator starts the algorithm in $IDLE$ state and makes a state transition to $WAIT\_POLL$ state when a $Tout$ occurs. From Lemma 1, all coordinators will be in $IDLE$ state after the algorithm execution. There are three transitions to the $IDLE$ state and in each transition a coordinator multicasts *End* message.

Lemma 3. *Node algorithm given in Alg. 1 is free from deadlock and starvation.*

Proof. Assume the contrary. From Observation 1, a node should be in $IDLE$ state at the end of the algorithm. Thus we assume that an ordinary node can be in $WAIT\_LD$, $WAIT\_ACK$ or $WAIT\_XFER$ state after the execution of the algorithm. When a node is LOW or MEDIUM loaded it makes a state transition from $IDLE$ to $WAIT\_LD$ state upon receiving $Coord\_Poll$. A node in this state will eventually receive *End* message from its coordinator as proved in Lemma 2 and will make a state transition to $IDLE$ state. When a node is HIGH loaded it will make a state transition to $WAIT\_XFER$ state and transfers its load upon receiving $Node\_Load$ message. After it transfers the load, it will receive $Xfer\_ACK$ message from its coordinator and makes a state transition to $WAIT\_XFER$ state. When the coordinator sends a *End* message, this cyclic state transitions will finish and node will be in $IDLE$ state at the end. We contradict with our assumption, node algorithm is free from deadlock and starvation.

Theorem 1. *Our hierarchial dynamic load balancing protocol is free from deadlock and starvation.*

Proof. Our protocol consists of coordinator and node algorithm. From Lemma 1 and Lemma 3 we proved that both algorithms are free from deadlock and starvation, thus theorem holds true.

Theorem 2. *The total number of messages for acquiring the global load knowledge at the worst case is k(1+(k-1)/2).*

Proof. A $Xfer\_Token$ must circulate across the ring and must be transmitted back to originator node. $Xfer\_Token$ with the smallest id will circulate, others will be dropped. So, at the worst case arrangement of coordinators, the sum of 1 to (k-1) $Xfer\_Token$ messages, which is equal to k*(k-1)/2, are transmitted. After originator node receives its token, it broadcasts k $Xfer\_List$ messages. Thus totally k(1+(k-1)/2) messages are needed.

Corollary 1. *For the global load distribution algorithm, circulation of the token in daisy architecture eliminates at least half of messages with compared to broadcasting.*

Proof. When each coordinator broadcasts its load, totally (k-1)*k messages are needed. On the other side, from theorem 2, circulation of token requires k*(k-1)/2 messages at the worst case.

Corollary 2. *The time for acquiring the global load knowledge is $O(dk)$ where T is the average message transfer time between adjacent nodes.*

Proof. As shown in Theorem 2, at the worst case, k*(k-1)/2 messages are needed for token circulation. However, since messages are transmitted in parallel, at the worst case (2k-1)*T time needed for token circulation. After originator node receives its token, it broadcasts k $Xfer\_List$ messages in Td time. Totally ((2k-1)+d)T time is needed. Thus, the time for acquiring the global load knowledge is $O(k)$.

Theorem 3. *The time for a single load transfer is between $4dT+L$ and $(4d+1)T+L$ where L is the actual average load transfer time.*

Proof. A node transfers its state to the coordinator in d steps in parallel with the other nodes and assuming there is a match of LOW-HIGH nodes in the local cluster, the coordinator will send X_Load message to the HIGH node in d steps. Then there will be L time for the actual load transfer. The HIGH and LOW(or MEDIUM) nodes also perform a final handshake to confirm delivery of load in $2d$ steps. The total minimum time for load transfer is then the sum of all of these steps which is $4dT+L$. In the case of a remote receiver, only 1 $Coord\_Xfer$ message will be transmitted resulting in $(4d+1)T+L$ time.

Corollary 3. *The total number of messages exchanged for a single load transfer is $O(d)$.*

Proof. As shown by Theorem 3, the maximum total number of messages required for a remote receiver will be $(4d+1)T+L$. Thus, the message complexity for the a single load transfer of the algorithm is $O(d)$.

3 Performance Evaluation

We implemented the load balancing algorithm with MPI library. We carried out our tests on Indiana University's BigRed Cluster. Indiana University's BigRed is a distributed shared-memory cluster, consisting of 768 IBM JS21 Blades, each with two dual-core PowerPC 970 MP processors, 8GB of memory, and a PCI-X Myrinet 2000 adapter for high-bandwidth, low-latency MPI applications. In addition to local scratch disks, the BigRed compute nodes are connected via gigabit Ethernet to a 266TB GPFS file system, hosted on 16 IBM p505 Power5 systems. In our all experiments, each measurement is averaged with three identical measurement scenarios. We firstly measure the runtimes of the algorithm for varying nodes and cluster sizes. The node numbers are selected from 3 to 30. Since each node has 4 cores, the core numbers are varying from 12 to 120. The cluster sizes are selected as 3, 4 and 6. Fig. 2 shows that our algorithm is

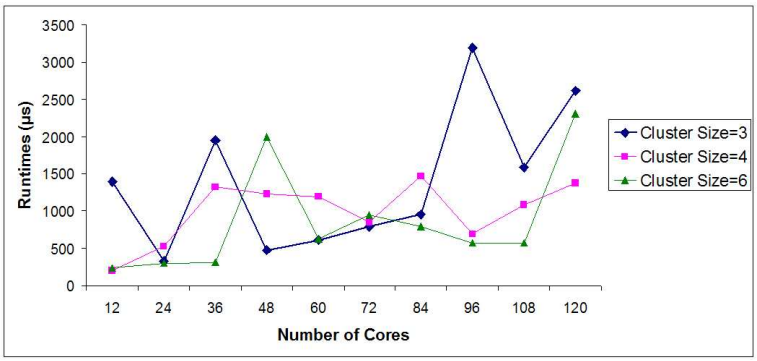

Fig. 2. Runtimes of the Algorithm

scalable when increasing core numbers from 12 to 120 and varying cluster sizes as 3,4 and 6. Since the nodes in the BigRed Cluster are identical, the cluster formation is simply achieved by considering the rank of the cores given by MPI. For example the cores 0, 1 and 2 are in the same the cluster in a system which is divided to the clusters having 3 cores each.

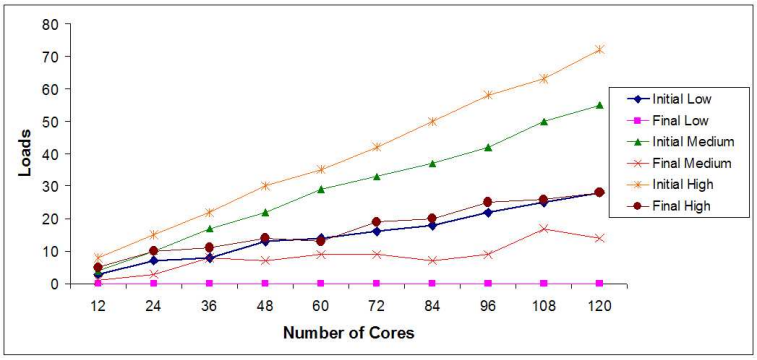

Fig. 3. Number of High Loaded Cores w.r.t Total System Load

One important goal of the algorithm is to decrease the number of high loaded nodes in the system as much as possible. For the system with 4 cores in each cluster, we prepare low, medium and high loaded test scenarios randomly with different seeds. The loads are represented with the integer values. Between 5 and 20, each 5 integer interval belongs a load type low, medium and high respectively. Fig. 3 shows the number of high loaded cores in initial state, before applying the algorithm, and in final state, after applying the algorithm. In the low loaded systems, all of the high loaded cores give their excessive loads to the other cores. In the medium loaded and high loaded systems more than half of the high loaded

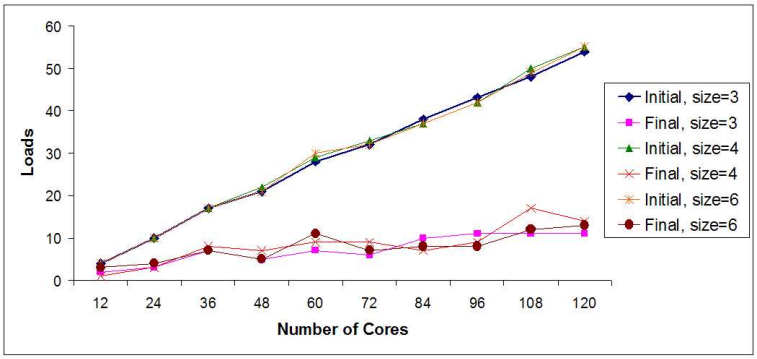

Fig. 4. Number of High Loaded Cores w.r.t Cluster Size

cores become medium loaded as shown in Fig. 3. We also fix the system load as medium and change the cluster sizes to measure the number of high loaded cores with respect to various clustering schemes. Measurements in Fig. 4 shows that the curves in different clustering scenarios are similar which shows that our algorithm achieves both inter-cluster and intra-cluster balancing.

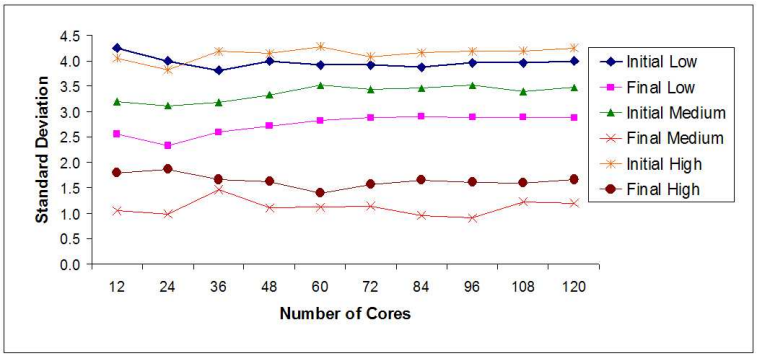

Fig. 5. Standard Deviations w.r.t Total System Load

Lastly, we measure the standard deviations of the loads before and after applying the algorithm in Fig. 5. In medium and especially high loaded systems, number of high loaded cores are much greater than the number of high loaded cores in low loaded systems. Generally, the standard deviations after applying the algorithm are smaller than half of the values before applying the algorithm in high and medium loaded systems as seen Fig. 5. In low loaded systems, since the number of transfers are smaller, the standard deviations after applying the algorithm are approximately 70% of the initial values. When we fix the system as medium loaded and vary the cluster size, the standard deviations after applying

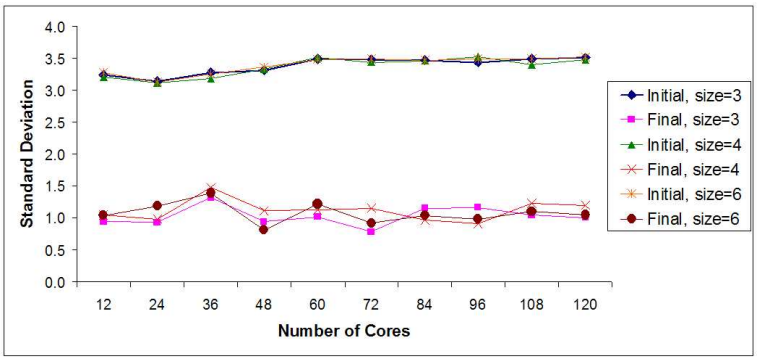

Fig. 6. Standard Deviations w.r.t Cluster Size

the algorithm are smaller than half of the initial values as seen in Fig. 6. This shows that the algorithm behaves stable under varying cluster sizes.

4 Conclusions

We showed the design and implementation of a protocol for dynamic load balancing in a Grid. The Grid is partitioned into a number of clusters and each cluster has a coordinator to perform local load balancing decisions and also to communicate with other cluster coordinators across the Grid to provide inter-cluster load transfers, if needed. Our results confirm that the load balancing method is scalable and has low message and time complexities. We have not addressed the problem of *how* the load should be transferred, and there are many research studies on this subject but we have tried to propose a protocol that is primarily concerned on *when* and *where* the load should be transferred. In fact, it may be possible not to transfer the load at all by employing copies of a subset of processes across the nodes in the Grid. The load transfer would require sending of some initial data only in this case. This is a future direction we are investigating.

The coordinators may fail and due to their important functionality in the protocol, new coordinators should be elected. Also, a mechanism to exclude faulty nodes from a cluster and add a recovering or a new node to a cluster are needed. These procedures can be implemented using algorithms as in [32] which is not discussed here. Our work is ongoing and we are looking into using the proposed model for real-time load balancing where scheduling of a process to a Grid node should be performed to meet its hard or soft deadline. The other area of concern discussed above would be the keeping the replicas of a subset of important and frequently used processes at Grid nodes to ease load transfer.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ACI-0338618l, OCI-0451237, OCI-0535258, and OCI-0504075. This research was supported in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative of Indiana University is supported in part by Lilly Endowment, Inc. This work was also supported in part by Shared University Research grants from IBM, Inc. to Indiana University.

References

1. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
2. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
3. National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov
4. Scholl, T., Bauer, B., Gufler, B., Kuntschke, R., Reiser, A., Kemper, A. Scalable Community-driven Data Sharing in E-science Grids. Elsevier Future Generation Computer Systems 25(3), 290–300 (2009)
5. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. Journal of High Performance Computing Applications 15(3), 200–222 (2001)
6. Foster, I.: What is the Grid? A Three Point Checklist. Grid Today (1)6, (2002)
7. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
8. Carlson, A., Bohringer, H., Scholl, T., Voges, W.: Finding Galaxy Clusters using Grid Computing Technology, In: Proc. of the IEEE Intl. Conf. on e-Science and Grid Computing, IEEE, Washington (2007)
9. Enke, H., Steinmetz, M, Radke, T. Reiser, A., Roblitz, T., Hogqvist, M.: AstroGrid-D: Enhancing Astronomic Science with Grid Technology, In: Proc. of the German e-Science Conference (2007)
10. Catlett, C.: The Philosophy of TeraGrid: Building an Open, Extensible, Distributed TeraScale Facility. In: Proc. of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE, Washington (2002)
11. Arora, M., Das, S., K.: A De-centralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments. In: Proc. of Int. Conf. Parallel Processing Workshops, IEEE, Washington (2002)
12. Johnston, W. E., Gannon, D., Nitzberg, B.: Grids as Production Computing Environments : The Engineering Aspects of NASA‘s Information Power Grid. In: Proc. Int. Sym. High Performance Distributed Computing, pp. 197–204, IEEE, Washington (1999)
13. TeraGrid, https://www.teragrid.org/
14. EGEE, http://public.eu-egee.org/
15. NORDUGRID, http://www.nordugrid.org/
16. Eager, D. L., Lazowska, E. D., Zahorjan, J.: A Comparison of Receiver-initiated and Sender-initiated Adaptive Load Sharing. ACM SIGMETRICS Performance Evaluation Review 6(1), 53–68 (1986)

17. Kumar, V. , Garma, A., Rao, V.: Scalable Load Balancing Techniques for Parallel Computers. Elsevier Journal of Parallel and Distributed Computing 22(1), 60–79, (1994)
18. Liu, J., Saletore, V. A.: Self-scheduling on Distributed Memory Machines. Proc. of Supercomputing, 814–823 (1993)
19. Lu, K., Subrata, R., Zomaya, A. Y.: An efficient load balancing algorithm for heterogeneous grid systems considering desirability of grid sites, IPCCC 06, IEEE, Washington (2006)
20. Shah, R., Veeravalli, B. Misra, M.: On the Design of Adaptive and Decentralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments. IEEE Trans. on Parallel and Distributed Systems 18(12), 1675–1686 (2007)
21. Lin, H., Raghavendra, C. S.: A Dynamic Load-balancing Policy with a Central Job Dispatcher. IEEE Trans. on Software Engineering 18(2), 148–158 (1992)
22. Feng, Y., Li, D., Wu, H., Zhang, Y.: A Dynamic Load Balancing Algorithm based on Distributed Database System. In: Proc. of Int. Conf. High Performance Computing in the Asia-Pasific Region, pp. 949–952, IEEE, Washington (2000)
23. Cao, J.: Self-organizing Agents for Grid Load Balancing. In Proc of Fifth IEEE/ACM International Workshop on Grid Computing 388-395, Washington (2004)
24. Liu, J., Jin, X., Wang, Y.: Agent-based Load Balancing on Homogeneous Minigrids: Macroscopic Modeling and Characterizatio., IEEE Trans. on Parallel and Distributed Systems 16(7), 586–598 (2005)
25. Wang, J., Wu, Q.-Y., Zheng, D., Jia, Y.: Agent based Load Balancing Model for Service based Grid Applications. Computational Intelligence and Security 1 486–491 (2006)
26. Subrata, R., Zomaya, A. Y.: Game-Theoretic Approach for Load Balancing in Computational Grids. IEEE Transactions on Parallel and Distributed Systems 19(1), 66–76 (2008)
27. Genaud, S., Giersch, A., Vivien, F.: Load-balancing Scatter Operations for Grid Computing. Parallel Computing 30(8), 923–946 (2004)
28. David, R., Genaud, S., Giersch, A., Schwarz, B., Violard, E.: Source Code Transformations Strategies to Load-Balance Grid Applications. In: GRID 2002. LNCS, vol. 2536, pp. 82–87, Springer-Verlag, Berlin (2002)
29. Karohis, N. T., Toonen, B., Foster, I.: MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface. Journal of Parallel and Distributed Computing 63(5), 551–563 (2003)
30. Erciyes, K., Payli, R., U.: A Cluster-based Dynamic Load Balancing Middleware Protocol for Grids. In: Advances in Grid Computing EGC 2005. LNCS, vol. 3470, pp. 805–812, Springer-Verlag, Berlin (2005)
31. Erciyes, K., Dagdeviren, O., Payli, U.: Performance Evaluation of Group Communication Architectures in Large Scale Systems using MPI. In: GADA 06. LNCS, vol. 4276, pp. 1422–1432, Springer-Verlag, Berlin (2006)
32. Tunali, T., Erciyes, K., Soysert, Z.: A Hierarchical Fault-Tolerant Ring Protocol For A Distributed Real-Time System. Special issue of Parallel and Distributed Computing Practices on Parallel and Distributed Real-Time Systems 2(1), 33–44 (2000)