Latency Hiding in Dynamic Partitioning and Load Balancing of Grid Computing Applications*

Sajal K. Das, Daniel J. Harvey
Dept. of Computer Science & Engineering
The University of Texas at Arlington
Arlington, TX 76019-0015, USA
{das,harvey}@cse.uta.edu

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

Abstract

The Information Power Grid (IPG) concept developed by NASA is aimed to provide a metacomputing platform for large-scale distributed computations, by hiding the intricacies of a highly heterogeneous environment and yet maintaining adequate security. In this paper, we propose a latency-tolerant partitioning scheme that dynamically balances processor workloads on the IPG, and minimizes data movement and runtime communication. By simulating an unsteady adaptive mesh application on a wide area network, we study the performance of our load balancer under the Globus environment. The number of IPG nodes, the number of processors per node, and the interconnect speeds are parameterized to derive conditions under which the IPG would be suitable for parallel distributed processing of such applications. Experimental results demonstrate that effective solutions are achieved when the IPG nodes are connected by a high-speed asynchronous interconnection network.

1. Introduction

The Information Power Grid (IPG) infrastructure has been developed by NASA and other collaborative partners to harness the power of geographically distributed resources (computers, databases, and human expertise) in order to solve large-scale computational problems. Applications that would benefit from such an infrastructure include:

- Desktop coupling to remote supercomputers so as to provide access to large databases and high-end graphics facilities [9].
- User access to sophisticated instruments through remote supercomputer connections utilizing virtual reality techniques [8].
- Remote interactions with supercomputer simulations [10, 11].

Several attempts have recently been made to develop what are called *computational grid* capabilities and/or implementations [14]. For example, the Condor system [19] is developed to manage research studies at workstations around the world. However, it did not adequately deal with the security issues involved. Other grid-based systems include Nimrod [1], NetSolve [4], NEOS [5], Legion [15], and CAVERN [18]. The Globus Metacomputing Infrastructure Toolkit [13] successfully provides a portable virtual machine environment. It supports mechanisms for sharing remote resources, provides adequate security, and allows MPI-based message passing. Due to its portable and modular nature, Globus has been chosen by NASA as the middleware to implement the IPG.

So far, limited studies have been performed to determine the viability of parallel distributed computing on the IPG. In [2], latency tolerance and load balancing modifications were implemented for a CFD application to compensate for slower communication speed. Results showed that the application ran faster under Globus on two IPG nodes of four processors each than on a single tightly-coupled machine of eight processors. However, this result is clouded in that asynchronous message passing was supported over the high-speed link but not within the single platform. With a goal to make more informative conclusions, in this paper we simulate an unsteady adaptive mesh application on a wide area network. The number of IPG nodes, the number of processors per node, and the interconnect speeds are parameterized to derive conditions under which the IPG would be suitable for parallel distributed processing of such applications.

*This work was supported by NASA Ames Research Center under Cooperative Agreement Number NCC 2-5393.

Earlier, we proposed two different load balancing approaches with an unsteady adaptive mesh as the test case application. The first approach, called PLUM [21], is an architecture-independent framework which globally partitions the computational mesh after each adaptation and determines whether re-balancing the load would lead to reduced total execution time. If an improvement in the load balance can be achieved, PLUM utilizes an effective remapping algorithm to minimize the required data movement. Application processing is temporarily suspended during the partitioning and data remapping operations. Utilization of a parallel graph partitioner like ParMeTiS [17] gives effective results.

The second approach, called Symmetric Broadcast Networks (SBN) [7], gives a general-purpose topology-independent solution to dynamic load balancing. A salient feature of this approach is that it balances processor workloads while the application is running. Therefore, it is able to hide the high data migration overhead, albeit at the cost of increased interprocessor communication. Results reported in [3] indicate that both PLUM and SBN approaches have their relative merits, and that they achieve excellent load balance with minimal extra overhead.

Let us summarize the contributions of this paper. We propose a novel partitioner, called MinEX, that optimizes the two important steps of PLUM (balancing and remapping) as part of the partitioning process. Instead of attempting to only balance the load like most other partitioners, the objective of MinEX is to minimize the total runtime of the application. This approach counters the possibility that perfectly balanced loads can still incur excessive communication and redistribution costs while the application is being processed. MinEX is also used to experiment with the latency tolerant techniques on the IPG. Our experimental results show that MinEX reduces the number of elements migrated by PLUM, and also lowers the percentage of edges cut by SBN. For example, for 32 partitions with our test case, PLUM showed an edge cut of 10.9% and redistributed 63,270 mesh elements. The corresponding values for the SBN-based approach were 36.5% and 19,446. In contrast, the MinEX partitioner values were 20.9% and 30,548 respectively. Thus MinEX attempts to optimize both communication and remapping costs, and hence is found to be an effective approach to latency hiding in dynamic load balancing for grid computing.

This paper is organized as follows. Section 2 introduces the computational application to be tested and determines its scalability. Section 3 describes the new MinEX partitioner. Section 4 describes the experimental study, analyzes the obtained results and draws conclusions as to the use of the IPG for this and similar applications. Section 5 concludes the paper.

2. Test Case Scenario

Many computational problems are often modeled as an unstructured mesh of vertices and edges. To capture evolving features, the mesh topology is also frequently adapted. For an efficient parallel implementation, this leads to dynamic load balancing in the sense that mesh objects will have to be reassigned after each adaptation phase to rebalance the workload among processors. It is critical to minimize the overhead associated with remapping data sets, and to reduce the communication between processors at the next solution step. These goals are particularly important in the IPG context where communication bandwidth between nodes are likely to be much smaller than those within a single multiprocessor machine.

The computational mesh considered for our experiments in this paper simulates an unsteady environment with a strongly time-dependent adapted region. As depicted in Fig. 1, a shock wave is propagated through an initial grid to produce the desired effect. The computational mesh is processed through nine adaptations by moving a cylindrical volume across the domain with constant velocity. Grid elements within the cylindrical volume are refined while previously-refined elements are coarsened in its wake. During the processing, the size of the mesh increases from 50,000 elements to 1,833,730 elements.

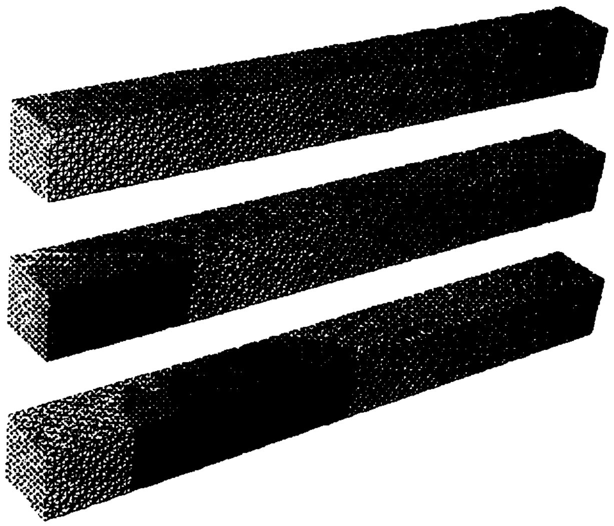

Figure 1. Initial and adapted meshes (after levels 1 and 5) for the simulated experiment.

To realistically simulate the overhead associated with an adaptive mesh computation, two weights are associated with each mesh vertex and one weight with each mesh edge. These weights respectively reflect the number of time units

	Number of Processors										
Latency	2	4	8	16	32	64	128	256	512	1024	2048
Max. Tolerance	3777	1824	1148	614	324	168	89	72	58	51	57
No Tolerance	4547	3193	1699	1033	558	302	173	123	115	109	103

Table 1. Scalability analysis of the test application.

required for computation, data remapping, and communication cost. The total time required to process the vertices assigned to a processor p must take into account all these three metrics as defined below.

Processing Weight, Wgt^v, is the computational cost to process a vertex v.

Redistribution Cost, $Remap_p^v$, is the overhead to copy the data set associated with v from p to another processor. This cost incurred at p includes operations like data packing and initiating transmission. The redistribution cost incurred by the processor receiving v is the sum of the communication cost and the operations of unpacking and merging the data into existing data structures. Clearly, if the data set for v is already assigned to p, no redistribution cost is incurred.

Communication Cost, $Comm_p^v$, is the cost to interact with all vertices adjacent to v but whose data sets are not local to p. Thus, if the data sets of all the vertices adjacent to v are also assigned to p, the communication cost, $Comm_p^v$, is 0.

We also use six additional metrics which are defined below.

Weighted Queue Length, QWgt(p), is the total cost to process the vertices assigned to p. It is defined as:

$$\texttt{QWgt}(p) = \sum_{v \text{ assigned to } p} (Wgt^v + Comm_p^v + Remap_p^v).$$

Total System Load, QWgtTOT, is the sum of QWgt(p) over all processors. This metric is used in Section 3.2 to decide whether it is appropriate to reassign a vertex from one processor to another.

Heaviest Load, MaxQWgt, is the maximum value of QWgt(p) over all processors, and indicates the total time required to process the application.

Lightest Load, MinQWgt, is the minimum value of QWgt(p) over all processors, and indicates the workload of the most lightly-loaded processor.

Average Load, AvgQWgt, is QWgtTOT/P, where P is the total number of processors.

Load Imbalance Factor, LoadImb, represents the quality of the partitioning and is defined as MaxQWgt / AvgQWgt.

Table 1 shows the scalability of our test application where P is varied from 2 to 2048. The data was obtained by simulating the application (details in Section 4). Each column reflects non-dimensionalized MaxQWgt values in thousands. The first row of the table assumes that maximum latency tolerance is achieved, while the second row assumes that no latency tolerance is achieved. By *maximum latency tolerance*, we mean the ability to utilize all available processors to overlap communication and redistribution costs. Further explanations are provided in Section 3. Table 1 shows that this application can scale to over 128 processors with linear speedup, and therefore is a good candidate for an IPG implementation.

3. MinEX: A New Partitioner

Previous studies with this mesh application under PLUM utilized a variety of general partitioners such as ParMeTiS [17], UAMeTiS [22], DAMeTiS [22], Jostle-MS [23], and Jostle-MD [23]. Note that UAMeTiS, DAMeTiS, and Jostle-MD are diffusive schemes designed to modify existing partitions to produce a processor allocation; whereas PMeTiS and Jostle-MS are global partitioners which make no assumptions about the original mesh distribution. Although all these partitioners achieve good load balance while minimizing communication overhead, they fail to consider the cost of moving data between processors. A unique feature of PLUM is to address this drawback through the use of an efficient heuristic procedure for redistributing data to assigned processors.

In the following, we design, implement, and analyze a novel partitioner, called MinEX, that optimizes computational, communication, and data remapping costs. We also redefine the partitioning goal from producing balanced loads to minimizing MaxQWgt. No direct comparisons with other existing partitioners mentioned above are possible because MinEX also considers the data redistribution cost while partitioning the computational mesh.

3.1. Design Principles

MinEX can be classified as a diffusive multilevel partitioner. Diffusive algorithms [6] utilize an existing partition as a starting point instead of partitioning from scratch. The multi-level approach, originally introduced in [16], partitions the graph in three steps — contraction, partitioning, and refinement — each of which is described below.

Similar to other multilevel partitioners, the first step in MinEX is to contract the mesh to a reasonable size. However, instead of repeatedly contracting the mesh in halves,

		ThroTTle values								
Metric	Clusters	0	1	3	4	16	32	64	128	200k
MaxQWgt	1	1993	1427	348	312	291	300	306	312	324
	2	1847	1142	748	467	320	304	305	318	345
	3	2035	1801	674	556	375	331	324	326	382
	4	1868	1516	761	639	412	352	328	371	425
	5	1834	1626	835	767	438	373	359	343	400
	6	2081	1579	898	825	481	391	357	361	427
	7	1884	1279	1032	758	505	383	371	369	414
	8	1944	1451	1102	834	531	434	376	380	435
LoadImb	1	7.05	5.09	1.23	1.11	1.01	1.00	1.00	1.00	1.00
	2	8.54	4.16	2.74	1.81	1.26	1.14	1.04	1.00	1.00
	3	7.15	6.40	2.50	2.11	1.41	1.19	1.05	1.02	1.01
	4	6.63	5.41	2.82	2.40	1.58	1.26	1.07	1.03	1.01
	5	6.53	5.78	3.06	2.83	1.66	1.30	1.11	1.02	1.01
	6	7.31	5.58	3.25	2.99	1.81	1.40	1.08	1.02	1.01
	7	6.68	4.61	3.74	2.80	1.84	1.33	1.10	1.03	1.00
	8	6.90	5.15	3.92	3.05	1.94	1.43	1.13	1.06	1.00

Table 2. Expected runtime and load balance quality for varying ThroTTle values.

MinEX sequentially contracts one vertex at a time. The advantage of this approach is that a decision can be made each time a vertex is later refined as to whether it should be assigned to another processor. This makes the algorithm more flexible since the graph does not have to be doubled in size before this decision could be made. If $|V|$ is the number of vertices in the mesh, contraction requires $O(|V|)$ steps which is asymptotically no larger than that of contracting the mesh sequentially in halves. Once the mesh is sufficiently small, the remaining vertices are reassigned according to the partitioning criteria described in Section 3.2.

The mesh is expanded back to its original size through a refinement process. As each vertex is refined, a decision is made as to whether or not it should be reassigned. This decision employs the same partitioning criteria used by the partitioning algorithm in the previous step. Each coarse vertex reassignment in effect reassigns all of the computational vertices that the coarse vertex represents.

3.2. Partitioning Criteria

The criteria for deciding whether a vertex should be reassigned from one processor to another, is based on two metrics: Gain and MinVar. Gain represents the change in QWgtTOT that would result from a proposed vertex move. A negative Gain would indicate that less total processing is required after such a vertex reassignment. The partitioning algorithm favors vertex moves with negative or small Gain values that reduce or minimize overall system load.

MinVar is computed using the workload (i.e. QWgt(p)) for each processor p and the smallest load of any processor (MinQWgt) in accordance with the following formula:

$$\texttt{MinVar} = \sum_{p} (\texttt{QWgt}(p) - \texttt{MinQWgt})^2.$$

Basically, MinVar computes the variance of processor workloads from that of the most lightly-loaded processor. The objective is to initiate vertex moves that lower this value. Since processors with large QWgt(p) values will have large MinVar components, this criteria tends to move vertices away from processors that have high runtime requirements. ΔMinVar is the change in the MinVar value after moving a vertex from one processor to another. A negative value indicates that MinVar has been reduced.

Let us now describe how the partitioning decisions are made. For each vertex, v, consider all edges to adjacent vertices that are assigned to other processors. Compute the Gain and MinVar values that would result from moving v to each of the adjacent processors. The move involves the adjacent vertex that has the smallest value of Gain as long as $\Delta\texttt{MinVar} < 0$ and $-\texttt{Gain}/\Delta\texttt{MinVar} < \texttt{ThroTTle}$, where ThroTTle is a user-supplied parameter. To increase efficiency, the program utilizes a minimum heap with pointers to vertex locations to quickly find the best move and directly delete entries without searching.

Conceptually, ThroTTle acts as a gateway that limits increases in Gain based upon how much of an improvement in MinVar can be achieved. Table 2 shows how varying ThroTTle values affects the expected application runtime (MaxQWgt) and load balance quality (LoadImb). The MaxQWgt entries are non-dimensionalized values in thousands. These results were obtained by running the experiments described in Section 4. Table 2 assumes a network of 32 homogeneous processors distributed over one to eight IPG nodes (clusters). The inter-cluster interconnect speed is assumed to be a third of the intra-cluster speed. Results show that a `ThroTTle` of 64 produces the lowest overall `MaxQWgt`, and that larger `ThroTTle` values improve `LoadImb`. Experiments with other network sizes using this same application have shown that `ThroTTle` generally converges at values between P and $2P$. Note also that for large values of `ThroTTle`, better `LoadImb` does not necessarily imply lower `MaxQWgt`.

3.3. Latency Tolerance

The following steps illustrate how communication and data redistribution can be reduced or eliminated.
Step 1: Initiate send of all data sets to be redistributed.
Step 2: For each edge (v, w), where the data set for vertex v is local to processor p and the data set for vertex w is local to another processor q, initiate send of communication data. The metric $Comm_p^{(v,w)}$ represents the cost of this communication. Also initiate send of communication data needed by adjacent processors.
Step 3: Process vertices that are not waiting for incoming transmissions.
Step 4: Receive and unpack any remapped data sets destined for this processor.
Step 5: Receive and unpack communication data destined for this processor.
Step 6: Repeat Steps 2 through 5 until all vertices are processed.

These steps implement a strategy where processors distribute data sets and communication data as early as possible. The processing of internal vertices can then take place while waiting for expected incoming messages. As data sets and communication data are received, additional communications can be initiated and vertices processed. The most optimistic expectation of this strategy is that the processing activity can entirely hide the data redistribution cost and communication latency. At the other extreme, the most pessimistic view is that no latency tolerance is achieved. Experiments simulating both views to analyze the effect of latency tolerance on our test application are described in Section 4.

3.4. Data Structures

The following data structures are used by the MinEX partitioner to perform its multilevel algorithm:

- `Mesh`: The adaptive mesh has the format
  $\{|V|, |E|$, `vTot`, `*VMaP`, `*VList`, `*EList`$\}$ where
  $|V|$ is the number of active vertices in the mesh,
  $|E|$ is the number of edges in the mesh,
  `vTot` is the total number of vertices (including merged vertices),
  `*VMaP` is a pointer to the list of active vertices,
  `*VList` is a pointer to the complete list of vertices, and
  `*EList` is a pointer to the list of edges.
- `VmaP`: A list of active vertices. None of these vertices have been compressed through multilevel partitioning.
- `VList`: A complete list of vertices. Each vertex, v, is defined by a `VList` record as
  $\{Wgt,\ Remap_p,\ |e|,\ {*e},\ merge,\ lookup,\ {*vmap},\ {*heap},\ border\}$ where
  Wgt is the computational cost to process v,
  $Remap_p$ is the redistribution cost to copy the data set associated with v to another processor from p,
  $|e|$ is the number of adjacent edges associated with v,
  $*e$ is a pointer to the first edge associated with v (subsequent edges are stored in contiguous memory locations),
  $merge$ is the vertex that was merged with v during a contraction operation (set to -1 if no merge took place),
  $lookup$ is the active vertex that contains v after a series of contraction operations (set to -1 if no merges took place),
  $*vmap$ is a pointer to the position of v in the active vertex table,
  $*heap$ is a pointer to the heap entry that relates to vertex, v, and represents a potential reassignment of v,
  and $border$ is a boolean flag indicating whether v is adjacent to vertices assigned to other processors.
- `EList`: A list of edges in the mesh. Each record is defined as $\{w, Comm_{(v,w)}\}$ where (v, w) is an edge and $Comm_{(v,w)}$ is the associated communication weight. Vertex v has an entry in `VList` and edges are located using the $*e$ pointer.
- `Heap`: The heap of potential vertex reassignments. Each heap record is defined as $\{$`Gain`, `ΔMinVar`, v, $p\}$ which specifies the `Gain` and `ΔMinVar` that would result from reassigning vertex v to processor p. The min-heap is keyed by the `Gain` value.
- `Stack`: The stack of compressed vertex pairs, (v_1, v_2). These vertices are refined in reverse order from the order that they were compressed. This graph contraction technique is described below.

3.5. Graph Contraction

The partitioner selects sets of randomly chosen pairs of vertices that are assigned to the same processor p.

From this set, the vertex pair, (v, w), that has the largest $Comm_{(v,w)}/(Remap_p^v + Remap_p^w)$ value is merged. This formula attempts to find edges with large communication costs while minimizing the potential data redistribution overhead. The motivation behind this strategy is to arrive at a contracted mesh with a small edge cut and a small data distribution cost.

To contract a vertex v, a merged vertex record, M, is created and the edge (v, w) is collapsed. The edges of M are generated by utilizing the edge lists of vertices v and w. VMap is adjusted to contain M and to remove v and w; $|V|$ is decremented and vTot is incremented; $|E|$ is increased by the number of edges created for M; and the pair (v, w) is pushed onto Stack.

This contraction procedure is implemented using a set union/find algorithm so that edges of existing vertices can remain unchanged. For example, if an existing vertex is adjacent to v, accesses to its EList record will check whether v has been merged. If it has, *lookup* will be accessed to quickly find the appropriate merged vertex. If *lookup* is not current (i.e., *lookup* > vTot), the union/find algorithm will search the chain of vertices beginning with *merge* in order to update the *lookup* value, so that subsequent lookups can be done efficiently. Pseudo code describing the union/find procedure is given in Fig. 2.

```
Procedure Find (v)
If (merge == −1) Return v
If (lookup != −1) And (lookup <= vTot)
    Then Return lookup = Find (lookup)
    Else Return lookup = Find (merge)
```

Figure 2. The union/find algorithm.

3.6. Partitioning the Contracted Graph

Once the graph contraction process is complete, the partitioning can be performed. Because the number of vertices is greatly reduced, the MinEX algorithm can execute very efficiently. The algorithm considers every remaining vertex of the mesh to find potential reassignments that will reduce Gain and MinVar as described in Section 3.2. All potential vertex reassignments are added to the min-heap. Actual reassignments are executed in heap order. As a reassignment is executed, the heap is adjusted to reflect the new partition status.

3.7. Graph Expansion

The graph is restored to its original size by expanding pairs of vertices in an order reversed from which they were merged. The Stack data structure controls the order. As pairs of vertices, (v, w), are refined, merged edges and vertices are deallocated. The *merge* and *lookup* vertex numbers are also adjusted in the vertex table. The VMap table is updated to delete the merged vertex, M, and to add v and w; $|V|$ is incremented and vTot is decremented; and $|E|$ is decreased by the number of edges created for M. After each refinement, a decision is made as to whether a partition can be improved by reassigning v or w. When reassignments are made, adjacent border vertices are also considered.

4. Performance Results

The MinEX partitioner was executed with actual application data to simulate an adaptive mesh computation for a variety of system configurations. Individual runs model networks with a particular number of processors P, number of IPG nodes/clusters C, ThroTTle values, and interconnect speeds I. In our experiments, P was varied from 2 to 2048, C was varied from 1 to 8, ThroTTle was varied to find the optimal value for minimizing runtime, and I was varied to simulate both high-speed cluster interconnects and low-speed wide area network connections.

Based on performance studies reported in [12, 20], typical communication latency and bandwidth slowdown from integrated clusters to configurations connected through a high-speed interconnect are in the range of 3 to 100. Wide area network connections are 1,000 to 10,000 times slower than the internal intra-connects of a single cluster. In our experiments, we have assumed that the intra-cluster communication speed is normalized to a value of 1. Simulations of inter-cluster communication assumed slowdown factors of 3, 10, 100, and 1,000. To simplify the analysis, we have assumed that individual processors are homogeneous and divided as evenly as possible among the clusters.

Table 3 shows results of experimental runs analyzing the effect of varying numbers of clusters and interconnect speeds, assuming $P = 32$ homogeneous processors. The interconnect speeds indicate the slowdown factor relative to the intra-cluster communication speed. To be consistent with Tables 1 and 2, runtimes are shown as non-dimensionalized values in thousands. Table 3(a) charts the experimental results when no latency tolerance is achieved, while Table 3(b) assumes maximum latency tolerance. The following conclusions can be drawn from the experiments.

As the interconnect speed is reduced, the slowdown experienced by utilizing additional clusters increases dramatically. For example, the runtime metric in Table 3(a) is 4,102 when two clusters and an interconnect slowdown of 1000 is assumed; however, the metric is 93,566 when eight clusters are assumed. Thus, performance deteriorates by almost a factor of 22.8. If we consider an interconnect slowdown of 3, the performance degradation is only 1.3. The same pattern holds true in Table 3(b).

	Interconnect Speeds			
Clusters	3	10	100	1000
1	473	473	473	473
2	728	863	1228	4102
3	755	1168	2783	18512
4	791	1361	3667	25040
5	854	1649	5677	53912
6	915	1717	8521	76169
7	956	1915	10958	80568
8	968	2178	11492	93566

(a) No latency tolerance

	Interconnect Speeds			
Clusters	3	10	100	1000
1	287	287	287	287
2	298	469	763	3941
3	322	548	2386	12705
4	328	680	3297	21888
5	336	768	4369	33092
6	345	856	5044	52668
7	352	893	5480	61079
8	357	1048	5721	61321

(b) Maximum latency tolerance

Table 3. Expected runtime for varying cluster sizes ($P = 32$) and interconnect speeds.

For the mesh application considered, Globus over low-speed networks such as the Internet is not a viable approach assuming current technology. In fact, the interconnection speed must improve by at least an order of magnitude before this approach could be useful. At present, applications would have to have little runtime communication and data set remapping for low-speed wide area networks to be practical interconnects.

We can compare the effectiveness of latency tolerant algorithms to those without latency tolerance, by measuring runtimes of each approach as the number of clusters and interconnect speeds are varied. The performance improvements using latency tolerance increase dramatically as the number of clusters increases. This can be verified by comparing the same rows from Tables 3(a) and 3(b). For example, consider the results with eight clusters. The runtime improvements comparing latency tolerant algorithms to those with no latency tolerance are factors of 2.7, 2.1, 2.0, and 1.5, respectively, for interconnect slowdowns of 3, 10, 100, and 1000. In contrast, results with two clusters indicate gains of 2.4, 1.8, 1.6, and 1.0, respectively, for the same interconnect slowdowns. Results clearly demonstrate that utilizing more clusters give greater runtime improvement when employing latency tolerance.

The same is also true when the interconnect slowdowns are varied (this can be analyzed by comparing the corresponding table columns). For example, with an interconnect slowdown of 1000, the improvements in runtime by utilizing latency tolerance are 1.6, 1.0, 1.5, 1.1, 1.6, 1.4, 1.3, and 1.5, respectively, for one to eight clusters. On the other hand, with an interconnect slowdown of 10, the corresponding improvements are 1.6, 1.8, 2.1, 2.0, 2.1, 2.0, 2.1, and 2.1. In this case, results surprisingly demonstrate that latency tolerance has a bigger payoff when interconnect slowdowns are smaller. Additional investigations are required to verify/counter this observation.

For our test application, Globus could be a viable approach if a high-speed interconnect (slowdown factor between 3 and 10) between clusters is utilized. Results in Tables 3(a) and 3(b) comparing one and eight clusters with an interconnect slowdown of 3 show runtime deterioration factors of 2.04 and 1.24, respectively. Similar comparisons for an interconnect slowdown of 10 show deterioration factors of 4.60 and 3.65, respectively. These factors, being smaller than the number of clusters, indicate a relative speedup when the number of clusters increases.

5. Conclusions

We presented a latency-tolerant partitioner, called MinEX, that not only balances processor workloads but also minimizes data movement and runtime communication, for adaptive mesh applications that are executed in a parallel distributed fashion on the IPG. Additional future experiments that are planned will test MinEX performance in the context of different application classes and devise metrics to compare it with other popular partitioning schemes. We also analyzed the conditions that are required for the IPG to be an effective tool for such distributed computations. Our results demonstrated that MinEX is a viable load balancer provided the IPG nodes are connected by a high-speed asynchronous interconnection network. We are currently implementing a parallel version of MinEX. An area of further research includes mathematical analysis of latency tolerance and performance slowdowns based on the interconnect speed, the numbers of clusters employed, and the topology of the mesh.

References

[1] D. Abramson, R. Sosic, J. Giddy, and R. Hall, "Nimrod: A tool for performing parameterized simulations using distributed workstations," *4th IEEE Symposium on High Performance Distributed Computing*, 1995.

[2] S. Barnard, R. Biswas, S. Saini, R. Van der Wijngaart, M. Yarrow, and L. Zechtzer, "Large-scale distributed computational fluid dynamics on the Information Power Grid using Globus," *7th Symposium on the Frontiers of Massively Parallel Computation*, 1999, 60–67.

[3] R. Biswas, S.K. Das, D.J. Harvey, and L. Oliker, "Parallel dynamic load balancing strategies for adaptive irregular applications," *Applied Mathematical Modelling*, 25 (2000) 109–122.

[4] H. Casanova and J. Dongarra, "NetSolve: A network server for solving computational science problems," Technical Report CS-95-313, University of Tennessee, 1995.

[5] J. Cryzyk, M. Meznier, and J. More, "The Network-Enabled Optimization System (NEOS) server," Preprint MCS-P615-0996, Argonne National Laboratory, 1996.

[6] G. Cybenko, "Dynamic load balancing for distributed-memory multiprocessors," *Journal of Parallel and Distributed Computing*, 7 (1989) 279–301.

[7] S.K. Das, D. Harvey, and R. Biswas, "Parallel processing of adaptive meshes with load balancing," *27th International Conference on Parallel Processing*, 1998, 502–509. (Extended version currently under revision for *IEEE Transactons on Parallel and Distributed Systems*.)

[8] T. Defanti, M.D. Brown, and R. Stevens, "Virtual reality over high-speed networks," *IEEE Computer Graphics and Applications*, 16 (1996) 42–43.

[9] T. Defanti, I. Foster, M. Papka, R. Stevens, and T. Kuhlfuss, "Overview of the I-Way wide area visual supercomputing," *International Journal of Supercomputer Applications*, 10 (1996) 123–130.

[10] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann, "Remote engineering tools for the design of pollution control systems for commercial boilers," *International Journal of Supercomputer Applications*, 10 (1996) 208–218.

[11] T. Disz, M. Papka, M. Pellegrino, and R. Stevens, "Sharing visualization experiences among remote virtual environments," *International Workshop of High Performance Computing for Computer Graphics and Visualization*, Springer-Verlag, 1995, 217–237.

[12] I. Foster and N. Karonis, "A grid-enabled MPI: Message passing in heterogeneous distributed computing systems," *Supercomputing'98*, 1998.

[13] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, 11 (1997) 115–128. (Also at `http://www.globus.org`.)

[14] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

[15] A. Grimshaw, W. Wulf, and the Legion team, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, 40 (1997) 39–45.

[16] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[17] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," Technical Report 96-036, University of Minnesota, 1996.

[18] J. Leigh, A. Johnson, and T. DeFanti, "CAVERN: A distributed architecture for supporting scalable persistence and interoperability in collaborative virtual environments," *Virtual Reality Research, Development and Applications*, 2 (1997) 217–237.

[19] M. Litzdow, M. Livny, and M.W. Mutka, "Condor — a hunter of idle workstations," *8th International Conference of Distributed Computing Systems*, 1988, 104–111.

[20] S. Nog and D. Kotz, "A performance comparison of TCP/IP and MPI on FDDI, fast Ethernet, and Ethernet," Technical Report PCS-TR95-273, Dartmouth College, 1996.

[21] L. Oliker and R. Biswas, "PLUM: Parallel load balancing for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, 52 (1998) 150–177.

[22] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *Journal of Parallel and Distributed Computing*, 47 (1997) 109–124.

[23] C. Walshaw, M. Cross, and M. Everett, "Parallel dynamic graph partitioning for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, 47 (1997), 102–108.