[illegible]

A Latency-Tolerant Partitioner for Distributed Computing on the Information Power Grid

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

Rupak Biswas
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field, CA 94035-1000
E-mail: rbiswas@nas.nasa.gov

Abstract

NASA's Information Power Grid (IPG) is an infrastructure designed to harness the power of geographically distributed computers, databases, and human expertise, in order to solve large-scale realistic computational problems. This type of a metacomputing environment is necessary to present a unified virtual machine to application developers that hides the intricacies of a highly heterogeneous environment and yet maintains adequate security. In this paper, we present a novel partitioning scheme, called MinEX, that dynamically balances processor workloads while minimizing data movement and runtime communication, for applications that are executed in a parallel distributed fashion on the IPG. We also analyze the conditions that are required for the IPG to be an effective tool for such distributed computations. Our results show that MinEX is a viable load balancer provided the nodes of the IPG are connected by a high-speed asynchronous interconnection network.

1 Introduction

The Information Power Grid (IPG) has been developed by NASA and other collaborative partners to harness the power of geographically distributed resources. The I-WAY experiment [8] identified several classes of applications that would benefit from such an infrastructure:

- Desktop coupling to remote supercomputers to provide access to large databases and high-end graphics facilities.
- User access to sophisticated instruments through remote supercomputer connections utilizing virtual reality techniques [7].
- Remote interactions with supercomputer simulations [9, 10].

There have been numerous attempts by the research community to develop computational grid capabilities. A comprehensive survey of current technology can be found in [13]. The Condor system [17], developed to manage research studies at workstations around the world, is an example of an early success. However, Condor did not adequately deal with security issues that are important for a general computational grid implementation. Other grid-based systems that have been developed include Nimrod [1], NetSolve [4], NEOS [5], Legion [14], and CAVERN [16]. The Globus Metacomputing Infrastructure Toolkit [12] (`http://www.globus.org`) has been extremely successful in providing a portable virtual machine environment. Mechanisms exist within Globus to share remote resources, provide adequate security, and allow MPI-based message passing. Due to its general, portable, and modular nature, Globus has been chosen by NASA as the middleware to implement the IPG.

Limited studies have been performed to determine the viability of parallel distributed computing on the IPG. In one such study [2], latency tolerance and load balancing modifications were implemented in connection with a computational fluid dynamics problem to compensate for slower communication speed. Results showed that the application actually 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. 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 so that general conclusions can be drawn as to when the IPG would be suitable to solve applications of this nature.

In the past, we have investigated two different load balancing strategies with this application as the test case. The first strategy, called PLUM [19], is an architecture-independent framework geared towards adaptive numerical solutions. PLUM 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, it utilizes an effective re-mapping algorithm to minimize the required data movement. Application processing is temporarily suspended during the partitioning and data re-mapping operations. Although PLUM is designed to utilize any parallel graph partitioner, ParMeTiS [15] has proven to be the most effective.

The second approach uses a general-purpose topology-independent dynamic load balancer utilizing Symmetric Broadcast Networks (SBN) [6]. A salient feature of this SBN-based approach is that it balances processor workloads while the application is executing. Thus 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 the SBN approach have their relative merits, and that they achieve excellent load balance with minimal extra overhead.

In this paper, we propose a novel partitioning approach that optimizes the two important steps of PLUM (balancing and re-mapping) as part of the partitioning process. The goal of this partitioner, called MinEX, is different from that of most partitioners. Instead of attempting to balance the load, the objective 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 processed. MinEX is also used to experiment with latency tolerant techniques on the IPG. Results show that MinEX reduces the number of elements migrated by PLUM, and 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 approach were 36.5% and 19,446. Instead, the MinEX partitioner values were 20.9% and 30,548.

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 Computational Test Case

Many computational problems are modeled discretely as an unstructured mesh of vertices and edges. To capture evolving features, the mesh topology is frequently adapted. For an efficient parallel implementation, this requires dynamic load balancing. In other words, mesh objects will have to be reassigned after each adaptation phase to re-balance the workload among the processors. It is critical to minimize the overhead associated with re-mapping data sets, and to reduce the communication between processors at the next solution step. These goals are especially important in an IPG context where communication bandwidths between nodes are likely to be much smaller than on a single multiprocessor machine.

The computational mesh used for the experiments in this paper simulates an unsteady environment where the adapted region is strongly time-dependent. As shown 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.

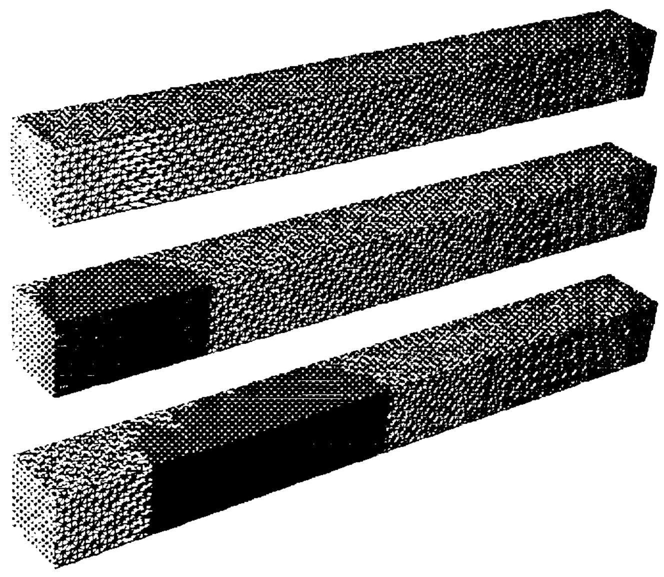

Figure 1: Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

To realistically simulate the overhead associated with an adaptive mesh computation, two weights are associated with each vertex and one weight with each edge. These weights respectively reflect the number of time units required for computation, data remapping, and communication. The total time required to process the vertices assigned to a processor p must take into account all three metrics which are defined as follows.

- **Processing Weight** (Wgt^v) is the computational cost to process a vertex v.
- **Communication Cost** ($Comm_p^v$) is the cost to interact with all vertices adjacent to v but whose data sets are not local to processor p.
- **Redistribution Cost** ($Remap_p^v$) is the overhead to copy the data set associated with v to another processor from p. Note that the redistribution cost incurred at p includes the operations of packing data and initiating transmission. The redistribution cost incurred by the processor receiving v is the sum of the communication time and the operations of unpacking and merging the data into existing data structures.

Additional metrics that will be needed in this paper 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.
- **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`.

Clearly, if the data set for v is already assigned to p, no redistribution cost is incurred, i.e. $Remap_p^v = 0$. Similarly, if the data sets of all the vertices adjacent to v are also assigned to p, the communication cost, $Comm_p^v$, is 0.

Table 1: Scalability analysis of the test application

	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 indicates the scalability of this application where the number of processors, P, is varied from 2 to 2048. The data was obtained by simulating the application (details presented 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; the second row assumes that no latency tolerance is achieved. *Maximum latency tolerance* is defined as 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 1000 processors, indicating good potential for an IPG implementation.

3 Proposed MinEX Partitioner

Previous studies with this mesh application under the PLUM framework utilized a variety of general partitioners such as ParMeTiS [15], UAMeTiS [20], DAMeTiS [20], Jostle-MS [21], and Jostle-MD [21]. 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 makes 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 this study, we optimize both communication and remapping costs by implementing a novel partitioner, called MinEX, that considers computational, communication, and data remapping costs. We also redefine the partitioning goal from producing balanced loads to minimizing **MaxQWgt**.

3.1 General Design

MinEX can be classified as a diffusive multilevel partitioner. Partitioning occurs in three steps: contraction, partitioning, and refinement. Each of these steps are discussed below:

- Similar to other multilevel partitioners, the first step in MinEX is to contract the mesh to a reasonable size. What is different, however, is the contraction procedure. Instead of repeatedly contracting the mesh in halves as is common with other multilevel partitioners, MinEX sequentially contracts one vertex at a time. The advantage to 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 would make the algorithm more flexible since the graph would 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. Total complexity would not be greater than the complexity of contracting the mesh sequentially in halves, since that would involve $O(|V/2|)$ steps. Performing all the steps would still require $O(|V/2|) + O(|V/4|) + \cdots = O(|V|)$.
- Once the mesh is sufficiently contracted, the remaining vertices are reassigned according to the criteria to be followed by the partitioning algorithm (described in detail 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 it should be reassigned. This decision employs the same criteria that is followed by the partitioning algorithm in the second step above. Each coarse vertex reassignment in effect reassigns all of the vertices the coarse vertex represents.

3.2 Partitioning Criteria

To describe the criteria for deciding whether a vertex should be reassigned from one processor to another, two metrics, `Gain` and `MinVarv`, need to be defined:

- `Gain` represents the change in `QWgtTOT` that would result from a proposed vertex move. A negative `Gain` value would indicate that less total processing is required after such a vertex move. 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:

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

  In other words, `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 will tend to move vertices away from processors that have high runtime requirements.

Δ`MinVar` is the change in the `MinVar` value after moving a vertex from one processor to another. A negative value indicates that the `MinVar` value has been reduced.

Let us now describe how the partitioning decisions are made. For each vertex, consider all edges to adjacent vertices that are assigned to other processors. Compute the `Gain` and `MinVar` values that would result from moving the given vertex to the adjacent processor. Designate the newly computed `MinVar` value as `MinVarNew` and the original `MinVar` value as `MinVarOld`. If `MinVarNew` < `MinVarOld` and `Gain`/(`MinVarOld` – `MinVarNew`) < `ThroTTle`, the proposed reassignment is considered. Note that `ThroTTle` is a user-supplied parameter. The move chosen will be the one with the smallest `Gain` value. To increase efficiency, the program utilizes a minimum heap with vertex pointers to heap locations to quickly find the best move and directly remove heap entries without having to search.

Table 2: Expected runtimes experienced based on varying `ThroTTle` values

		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

Conceptually, `ThroTTle` acts as a gate that limits increases in `Gain` based upon how much of an improvement in `MinVar` can be achieved. Table 2 indicates how varying `ThroTTle` affects the expected application runtimes (`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` value of 64 produces the lowest overall `MaxQWgt`, and that larger `ThroTTle` values improve `LoadImb`. Experiments with other network sizes using this same mesh 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 processing steps illustrate how communication and data redistribution can be reduced or eliminated.

1 Initiate send of all data sets to be redistributed.

2 Initiate send of communication data needed by adjacent processors.

3 Process vertices that are not waiting for incoming transmissions.

4 Receive and unpack any re-mapped data sets destined for this processor.

5 Receive and unpack communication data destined for this processor.

6 Repeat steps 2 through 5 until all vertices are processed.

The above logic implements a strategy where processors distribute data sets and communication data as early as possible. Servicing of internal mesh 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 set 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 Partitioning Data Structures

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

`Mesh`: The adaptive mesh whose format is $\{|V|, |E|,$ `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 (includes merged vertices)

`*VMaP` is a pointer to list of active vertices

`*VList` is a pointer to list of vertices

`*EList` is a pointer to list of edges.

`VmaP`: A list of active vertex numbers. 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 or -1 if no merge took place.

lookup is the active vertex that contains v after a series of contraction operations or -1 if no merges took place.

$*vmap$ is a pointer to the position of v in the active vertex table.

$*heap$ is the pointer to an entry in the heap that relates to vertex, v. This entry represents a potential reassignment of v. This pointer is used to be able to remove heap entries without searching.

border is a boolean flag indicating whether v is adjacent to vertices assigned to other processors.

`EList`: The list of edges in the mesh. Each edge record is defined as $\{v, \texttt{Comm}\}$ where v is the adjacent vertex and `Comm` is the communication weight associated with this edge.

`Heap`: The heap of potential vertex reassignments. Each heap record is defined as $\{\texttt{Gain}, \Delta\texttt{MinVar}, v, p\}$ which specifies the `Gain` and Δ`MinVar` 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, $(\{vertex1, vertex2\})$. These vertices are refined in reverse order from the order that they were compressed.

3.5 Graph Contraction

The partitioner selects sets of randomly chosen pairs of vertices that are assigned to the same processor. From this set, the vertex pair, (v, w), to be merged has the largest $Comm_v^w/(Remap^v + Remap^w)$ value. This formula attempts to find edges with large edge communication costs while minimizing the potential cost of data set redistribution. The motivation behind this strategy is to arrive at a contracted mesh with a small edge cut and with small costs of data distribution.

To contract a vertex, a merged vertex record is created such that the merged vertex, M, is adjacent to all vertices, other than v and w, that were originally adjacent to either of the two original vertices. The edge records corresponding to M are created accordingly. `VMap` is adjusted to contain the newly created vertex 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`.

3.6 Union/Find Algorithm

A union/find algorithm is utilized 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, the union/find algorithm will search the chain of vertices beginning with *merge* to update the *lookup* value so subsequent lookups can be done efficiently. Pseudo code describing the union/find procedure is shown 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: Pseudo code for the union/find algorithm

3.7 Partitioning of the Contracted Graph

Once the graph contraction process is complete, the partitioning can be executed. Because the number of vertices is greatly reduced, the MinEX partitioning algorithm can execute efficiently. The algorithm considers every remaining vertex of the mesh to find potential reassignments that will reduce `Gain` and `MinVar` as described in Subsection 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.

3.8 Refinement

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

4 Experimental Study

The partitioner MinEX was executed with actual application data to simulate mesh processing for a variety of system configurations. Individual runs simulates networks with a particular number of processors (P), number of 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 high-speed cluster interconnections and low-speed wide area network connections.

Based on performance studies [11, 18], typical communication latency and bandwidth slowdowns from integrated clusters to configurations with clusters 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. For these experiments, we have assumed that the intra-cluster communication speed to be 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.

4.1 Summary of Results

Table 3(a) and 3(b) show 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 results presented in Tables 1 and 2, runtimes are shown in thousands. Table 3(a) charts the experimental results when no latency tolerance is achieved; Table 3(b) assumes maximum latency tolerance.

Table 3: Expected runtime in thousands of units for varying clusters and interconnect speeds ($P = 32$)

	Interconnect Speeds			
Clusters	3	10	100	1000
1	473	473	473	470
2	763	926	926	4781
3	952	1354	1354	23769
4	989	1602	1602	25040
5	1021	1649	1649	53912
6	1091	1717	1717	86068
7	989	2116	2116	105570
8	968	2178	2178	93566

(a) No latency tolerance

	Interconnect Speeds			
Clusters	3	10	100	1000
1	306	306	306	306
2	305	499	860	5355
3	324	600	2431	14357
4	328	717	3373	33092
5	359	768	4369	51816
6	357	929	5120	62032
7	371	893	5873	62059
8	376	1048	5721	61321

(b) Maximum latency tolerance

The following conclusions can be drawn from the experiments.

- As the interconnect speed slows, the slowdown experienced by utilizing additional clusters increases dramatically. For example, the runtime metric in Table 3(a) is 4,781 when two clusters and an interconnect slowdown of 1000 is assumed. However, the runtime metric is 93.566 when eight clusters are assumed. The ratio, $93,566/4,781 \approx 19.57$. If we consider the interconnect slowdown of 3. the ratio between two clusters and eight clusters is $968/763 \approx 1.26$ which is a much smaller value. The same pattern holds true in Table 3(b)

- For 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 has to improve by at least one or two orders of magnitude before this approach could be useful. Under current technology. applications would have to have minimum communication and data-set re-mapping for low-speed wide area networks to be practical interconnects.

- Latency tolerant algorithms show larger runtime gains when more clusters are utilized. This can be verified by comparing the same rows from Table 3(a) and Table 3(b). The rows that correspond to more clusters show greater latency tolerance runtime gain. The same cannot be said when analyzing columns of the tables where interconnect slowdowns are varied. Latency tolerance runtime gains remain relatively constant in this case. We can also conclude that regardless of whether one or eight clusters of processors are employed, latency tolerance algorithms will always be beneficial to reducing expected application runtimes.

- For our application. Globus could be a viable approach if a high-speed interconnect (slowdown factor between 3 and 10) between clusters is utilized. The first column of Tables 3(a) and 3(b) comparing 1 and 8 clusters with an interconnect slowdown factor of 3, respectively, show a slowdown factor of 2.04 and 1.22. Similarly, the second column of the tables with an interconnect slowdown factor of 10 show slowdown factors of 4.60 and 3.35, respectively. These factors being smaller than the number of clusters indicate a speedup from when one cluster of $\frac{1}{8}$ the number of processors are used.

5 Conclusions

This paper 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. We also analyzed the conditions that are required for the IPG to be an effective tool for such distributed computations. Our results demonstrate that MinEX is a viable load balancer provided the nodes of the IPG are connected by a high-speed asynchronous interconnection network. An area of further research includes mathematical analysis of latency tolerance or slowdowns based on the interconnect speed, numbers of clusters employed, and the topology of the mesh.

Acknowledgements

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

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*, to appear.

[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] S. Das, D. Harvey, and R. Biswas, "Parallel processing of adaptive meshes with load balancing," *27th Conference on Parallel Processing*, 1998, 502–509.

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

[8] 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.

[9] 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.

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

[11] I. Foster and N. Karonis, "A grid-enabled MPI: Message passing in heterogeneous distributed computing systems." *ACM/IEEE SC98 Conference*, 1998.

[12] I. Foster and C. Kesselman. "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*. 11 (1997) 115–128.

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

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.