MinEX: A Latency-Tolerant Dynamic Partitioner for Grid Computing Applications

Sajal K. Das [a], Daniel J. Harvey [b], Rupak Biswas [c,*]

[a] *Dept. of Computer Science & Engineering, The University of Texas at Arlington, Arlington, TX 76019, USA*

[b] *Dept. of Computer Science, Southern Oregon University, Ashland, OR 97520, USA*

[c] *NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA 94035, USA*

Abstract

The Information Power Grid (IPG) being developed by NASA is 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 infrastructure 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 latency-tolerant 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. The number of IPG nodes, the number of processors per node, and the interconnect speeds are parameterized in a simulation experiment to derive conditions under which the IPG would be suitable for solving such applications. Experimental results demonstrate that MinEX is an effective load balancer for the IPG when the nodes are connected by a high-speed asynchronous interconnection network.

Key words: Information Power Grid, adaptive computations, partitioning, dynamic load balancing, latency tolerance

* Corresponding author. Tel.: +1-650-604-4411; Fax: +1-650-604-3957.
Email addresses: `das@cse.uta.edu` (Sajal K. Das), `harveyd@sou.edu` (Daniel J. Harvey), `rbiswas@nas.nasa.gov` (Rupak Biswas).

1 Introduction

NASA and its collaborative partners are actively developing the Information Power Grid (IPG) [20] to harness the vast collection of their geographically distributed resources (computers, databases, and human expertise). Current engineering and research status of the IPG project is available at `http://www.ipg.nasa.gov`. One of the primary benefits of the IPG will be to facilitate the efficient solution of large-scale computational problems by providing a scalable, adaptive, and transparent environment that is both ubiquitous and uniformly accessible through a convenient interface. Some other areas that would benefit from such a nationwide infrastructure include:

- Desktop coupling to remote resources so as to provide access to large databases and high-end graphics facilities [10].
- User access to sophisticated instruments through remote connections utilizing virtual reality techniques [9].
- Remote interactions with parallel and distributed supercomputer simulations [11,12].

The IPG is one of several approaches to develop what are called *Computational Grid* [1] (in short, Grid) capabilities and/or implementations [16]. For example, Condor [23] was an early success in developing a distributed system to manage research studies at workstations around the world. However, it does not adequately deal with the security issues that are important for a general Grid implementation. Other Grid-based systems include Nimrod [1], NetSolve [4], NEOS [6], Legion [17], and CAVERN [22]. The Globus Metacomputing Infrastructure Toolkit [15] 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.

Till date, only a few limited studies have been performed at NASA Ames Research Center to determine the viability of large-scale parallel and distributed computing on the IPG [2,13]. In [2], latency tolerance and load balancing modifications were implemented for a computational fluid dynamics (CFD) application to compensate for the slower communication speed between two IPG computers (nodes). Results showed that the application actually ran faster under Globus on two 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 wide area network but not within the single platform. The results presented in [13] demonstrated the feasibility of parallel distributed computing on homogeneous IPG testbeds, but performance was significantly affected by increased communication times. The paper concluded that poorer connectivity and larger latencies due to geographical separation in a realistic IPG environment could further impact

[1] Not to be confused with computations on discretization grids.

overall performance.

With a goal to make more informative conclusions regarding the latency tolerance and load balancing performance of parallel distributed computational applications on the IPG, in this paper we simulate an unsteady adaptive mesh problem on a wide area network. The number of nodes, the number of processors per node, and the interconnect speeds between nodes are all parameterized to derive general conditions under which such an infrastructure would be suitable for parallel distributed processing of this class of applications.

In our previous work, we have developed two different load balancing techniques for dynamic irregular applications. The first strategy, called PLUM [25], is an architecture-independent framework which globally partitions the computational mesh after each adaptation and determines whether rebalancing the workload would reduce the total execution time. If an improvement in the load balance can be achieved, PLUM utilizes one of several remapping algorithms 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 [21] gives extremely 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 the SBN-based method 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 have their relative merits, and that they achieve excellent load balance with minimal extra overhead.

In this paper, we propose a novel partitioner, called MinEX, that optimizes the two important steps of PLUM (namely, balancing and remapping) as part of the partitioning process. Instead of attempting to merely balance the load and reduce the runtime interprocessor communication 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 with minimal communication can still incur excessive redistribution costs for adaptive applications. MinEX is also used to experiment with latency tolerant techniques for the IPG. Our experimental results show that MinEX reduces the workload migrated by PLUM, and lowers the communication cost over partitions generated by SBN. For example, for 32 partitions with our test case, PLUM showed an edge cut (reflecting the communication overhead) of 10.9% and redistributed 63,270 mesh elements. The corresponding numbers for the SBN-based approach were 36.5% and 19,446. In contrast, the MinEX partitioner values were 20.9% and 30,548, respectively, while maintaining comparable load balance. Thus, MinEX attempts to optimize both communication and remapping costs, and can be an effective latency hiding technique in dynamic load balancing for Grid computing applications. A preliminary version of this paper appeared in [8].

The remainder of this paper is organized as follows. Section 2 introduces the dynamic irregular computational application used as the test case for our experiments, and describes the various graphs and metrics that model the problem. Section 3 presents the new MinEX partitioner and gives implementation details. Performance results are reported and analyzed in Section 4. Finally, Section 5 summarizes our key conclusions as to the viability of MinEX and the IPG for this class of applications.

2 Preliminaries

In this section, we describe our computational test case, and the various graphs and metrics utilized to model the problem and evaluate MinEX.

2.1 Computational test case

Many computational problems are often modeled discretely 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 requires dynamic load balancing in the sense that mesh objects usually have to be reassigned after each adaptation phase to rebalance the workload among the processors. It is critical to minimize the overhead associated with remapping data sets, and to reduce the communication between processors during the subsequent solution step. These goals are particularly important in the context of the IPG where communication bandwidth between nodes are likely to be much smaller than those within a single node (i.e., 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 tetrahedral grid to produce the desired effect. This grid 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.

2.2 Graph models

In our experiments, a dual graph representation of the initial mesh is used for load balancing where each original tetrahedron is a vertex of the graph. An edge exists between two dual graph vertices if the corresponding elements share a face. Mesh refinement consists of subdividing parent tetrahedral elements into two, four, or eight subelements in specified regions of the mesh. Subsequent refinements can further split the child elements, thereby forming a refinement tree of tetrahedra for each original mesh element.

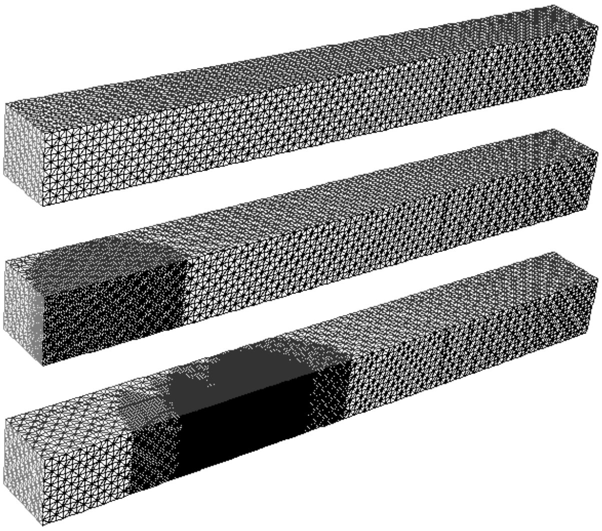

Fig. 1. Initial and adapted meshes (after levels 1 and 5) for the simulated experiment.

To realistically simulate the overhead associated with such an adaptive mesh computation, weights are associated with the vertices and edges of the dual graph. Each vertex v has two weights, $Pwgt_v$ and $Rwgt_v$, while each edge (v, w) has one weight, $Cwgt_{(v,w)}$. These weights respectively model the associated computational processing, data remapping, and runtime interprocessor communication costs. $Pwgt_v$ is proportional to the number of leaves in the refinement tree because only those elements participate in the actual calculation. However, $Rwgt_v$ is proportional to the total number of elements in the refinement tree because the entire tree must be relocated when the root is reassigned to another processor. Finally, $Cwgt_{(v,w)}$ depends on the number of leaf faces between dual graph vertices v and w.

To predict performance on a variety of distributed architectures, a configuration graph is utilized. Each vertex in this fully-connected graph represents a tightly-coupled cluster of processors, while edges denote cluster interconnects. For the experiments reported here, we assume that all processors in a single cluster (node) are homogeneous and that there is a constant bandwidth for intra-cluster communication. Vertex c in the configuration graph has weight $Proc_c \geq 1$ that represents the processing slowdown factor for the corresponding cluster. Similarly, the edge weight $Conn_{(c,d)} \geq 1$ represents the interconnect slowdown factor when a processor in cluster c communicates with a processor in another cluster d. If $c = d$, $Conn_{(c,c)}$ is the slowdown associated with communication between processors in the same cluster c. Note that if any of these weights are unity, there is no slowdown (ideal conditions).

2.3 Metrics

The following three metrics respectively measure the number of time units required for computation, communication, and remapping. The total time required to process the elements assigned to processor p in cluster c must take into account all of them.

- Processing cost: Wgt_v^p is the computational cost to process dual graph vertex v assigned to processor p which is in cluster c:

$$Wgt_v^p = Pwgt_v \times Proc_c.$$

- Communication cost: $Comm_v^p$ is the communication cost to interact with all vertices w adjacent to v whose data sets are not local to p (assuming that v is assigned to p):

$$Comm_v^p = \sum_w Cwgt_{(v,w)} \times Conn_{(c,d)},$$

where c and d are the clusters containing the processors to which v and w are respectively assigned. Obviously, if the data sets of all vertices adjacent to v are also assigned to p, then $Comm_v^p = 0$.

- Redistribution cost: $Remap_v^p$ is the overhead to copy the data set associated with v to another processor from p:

$$Remap_v^p = Rwgt_v \times Conn_{(c,d)},$$

where c and d are the clusters containing the source and destination processors for v. Note that the redistribution cost incurred at p includes the operations of packing data and initiating transmission, while the cost incurred by the processor receiving v is the sum of the communication time and the cost of unpacking and merging the data into existing data structures. Clearly, $Remap_v^p = 0$ if the data set for v is already assigned to p.

Five additional metrics used in this work are defined below.

- Weighted queue length: $Qwgt^p$ is the total cost to process all vertices v assigned to p:

$$Qwgt^p = \sum_v \ (Wgt_v^p + Comm_v^p + Remap_v^p).$$

- Total system load: $QwgtTot$ is the cost to process the entire application:

$$QwgtTot = \sum_p \ Qwgt^p.$$

- Heaviest load: $MaxQwgt$ indicates the total time required to process the application:

$$MaxQwgt = \max_p \ Qwgt^p.$$

- Lightest load: $MinQwgt$ indicates the workload of the most lightly-loaded processor:

$$MinQwgt = \min_p \ Qwgt^p.$$

- Load imbalance factor: $LoadImb$ represents the quality of the partitioning:

$$LoadImb = P \times MaxQwgt/QwgtTot,$$

where P is the total number of processors in the configuration graph.

3 MinEX: A new partitioner

Previous studies with the test application (described in Section 2.1) under PLUM utilized a variety of general-purpose partitioners such as ParMetis [21], UAMetis [26], DAMetis [26], Jostle-MS [27], and Jostle-MD [27]. Note that UAMetis, DAMetis, and Jostle-MD are diffusive schemes designed to modify existing partitions to produce a processor allocation, whereas ParMetis and Jostle-MS are global from-scratch partitioners which make no assumptions about the original distribution of the mesh. 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 for redistributing data to assigned processors.

In the following subsections, 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 workloads to minimizing the $MaxQwgt$ metric. No direct comparisons with other existing partitioners mentioned above were feasible since MinEX also considers the data redistribution cost while partitioning the computational mesh.

3.1 General design principles

MinEX can be classified as a diffusive multilevel partitioner. Diffusive algorithms [5] utilize an existing partition as a starting point instead of partitioning from scratch. The multilevel approach, originally introduced in [19], partitions a graph in three steps: contraction, partitioning, and expansion — 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 as is common with other multilevel partitioners, 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, the contraction step requires $O(|V|)$ substeps which is asymptotically equal to the complexity of contracting the mesh sequentially in halves.

Once the mesh is sufficiently contracted, the remaining vertices are reassigned according to the partitioning criteria described in Section 3.3.

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

3.2 Latency tolerance

Our MinEX partitioner can interact via a user-defined function to accommodate any latency tolerance that a mesh application may possess. The following steps illustrate how the application can be programmed so that MinEX eliminates (or at least reduces) communication and data redistribution costs.

Step 1: Initiate send of all computational data sets that are to be redistributed to other processors.

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. Also initiate send of communication data needed by adjacent processors.

Step 3: Process vertices that are not waiting for any incoming transmissions.

Step 4: Receive and unpack any remapped computational data sets destined for processor p.

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 computational and communication data as early as possible. Internal vertices can then be serviced while waiting for expected incoming messages. As information is received, additional communications can be initiated and vertices processed. The most optimistic view 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. To analyze the effect of latency tolerance on our test application, experiments simulating both possibilities are described in Section 4.

3.3 Partitioning criteria

The criteria for deciding whether a vertex should be reassigned from one processor to another is based on two metrics: *Gain* and $MinVar$. These are obtained as follows:

- *Gain* represents the change in $QwgtTot$ that would result from a proposed vertex move. A negative value indicates 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.

- $MinVar$ measures the variance of processor workloads from that of the most lightly-loaded processor. It is computed using the workload for each processor p and the smallest load over all processors:

  $$MinVar = \sum_{p} (Qwgt^p - MinQwgt)^2.$$

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

The partitioning decisions are made as follows. For each vertex v, consider all adjacent vertices assigned to other processors. Compute the $Gain$ and $MinVar$ values that would result from moving v to each of these adjacent processors. The vertex moved is the one with the smallest $Gain$, and satisfies $\Delta MinVar < 0$ and $-Gain/\Delta MinVar < ThroTTle$, where $ThroTTle$ is a user-supplied parameter. To increase efficiency, we use a minimum heap with pointers to vertex locations in order to rapidly find the best migration and directly remove entries without searching.

Conceptually, $ThroTTle$ acts as a gate that limits increases in $Gain$ based upon how much of an improvement in $MinVar$ can be achieved. Table 1 shows how varying $ThroTTle$ affects the expected application runtime ($MaxQwgt$) and load balance quality ($LoadImb$), assuming maximum latency tolerance. The $MaxQwgt$ entries are non-dimensionalized values in thousands, and were obtained by running the experiments described in Section 4. Table 1 results are for a network of $P = 32$ 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. Observe that $ThroTTle = 64$ 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 $MaxQwgt$.

3.4 Data structures

We give here a brief description of the data structures used for implementing the multilevel MinEX partitioner:

$Mesh$**:** The adaptive mesh, represented as
$\{|V|, |E|, vTot, *VMap, *VList, *EList\}$ where $|V|$ is the number of active vertices, $|E|$ is the number of edges, $vTot$ is the total vertex count (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

Table 1
Expected runtime and load balance quality with maximum latency tolerance for varying *ThroTTle* values and $P = 32$

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

list of edges.

VMap: The list of active vertices (those that have not been compressed during multilevel partitioning).

VList: The complete list of vertices. Each vertex v is represented as $\{Pwgt_v, Rwgt_v, |e|, *e, merge, lookup, *vmap, *Heap, border\}$ where $Pwgt_v$ is the computational cost to process v, $Rwgt_v$ is the redistribution cost to copy the data set associated with v, $|e|$ is the number of edges incident on v, $*e$ is a pointer to the first incident edge (subsequent edges are stored contiguously), *merge* is the vertex that was merged with v during a contraction operation (set to -1 if not merged), *lookup* is the active vertex that contains v after a series of contractions (set to -1 if not merged), $*vmap$ is a pointer to the position of v in *VMap*, $*Heap$ is a pointer to v's heap entry 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: The list of edges in the mesh. Each vertex v in *VList* points to its first edge in $EList$ using $*e$. Each edge record is defined as $\{w, Cwgt_{(v,w)}\}$ where w is a vertex adjacent to v and $Cwgt_{(v,w)}$ is the communication weight associated with this edge.

$Heap$**:** The heap of potential vertex reassignments. Each heap record is defined as $\{Gain, \Delta MinVar, v, p\}$ which specifies the $Gain$ and $\Delta MinVar$ that would result from reassigning vertex v to processor p. The min-heap is keyed by the $Gain$ value.

$Stack$**:** The stack of collapsed edges, (v, w). These pushed edges are refined in an order reversed from the one in which they were compressed. This graph contraction technique is described in the next section.

3.5 Graph contraction

MinEX randomly selects a set of adjacent vertex pairs that are assigned to the same processor. From this set, the vertex pair (v, w) that has the largest $Cwgt_{(v,w)}/(Rwgt_v + Rwgt_w)$ 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 as well as a small data distribution cost.

To collapse the edge (v, w), a merged vertex M is generated. The edges incident on M are created 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 $Stack$. The entire process is repeated until the graph is sufficiently contracted.

This contraction procedure is implemented using a set Union/Find algorithm so that edges of unmerged vertices remain unchanged. For example, if an unmerged vertex is adjacent to v, accesses to its $EList$ will check whether v has been merged. If it has, $lookup$ will 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 $lookup$, so that subsequent queries can be done efficiently. The pseudo code describing the Union/Find procedure is given in Fig. 2.

procedure $Find\,(v)$
if $(merge\ =\ -1)$ **return** (v)
if $((lookup \neq -1)$ **and** $(lookup \leq vTot))$
  then return $(lookup\ =\ Find\,(lookup))$
  else return $(lookup\ =\ Find\,(merge))$

Fig. 2. Pseudo code for the Union/Find algorithm.

3.6 Partitioning the contracted graph

The partitioning is performed when the graph contraction process is complete. MinEX partitioning is efficient because the number of vertices is greatly reduced. The algorithm considers every vertex of the coarse mesh to find potential reassignments that will reduce $Gain$ and $MinVar$ as described in Section 3.3. All potential vertex reassignments are added to the min-heap, and executed in heap order. After each reassignment, the heap is adjusted to reflect the new partition.

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 $lookup$ values are also adjusted in $VList$. The list $VMap$ of active vertices 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, it is checked whether a partition can be improved by reassigning v or w. When reassignments are made, adjacent border vertices are also considered.

4 Performance results

In the experimental study presented below, two extreme cases are considered. The first is the most optimistic view in which processing activity can entirely hide the data set redistribution and communication latency. The second case, on the other hand, is the most pessimistic view where no latency tolerance can be achieved.

The MinEX partitioner was executed with the computational test case (described in Section 2.1) that simulates an adaptive mesh calculation. A variety of system configurations was evaluated. Individual runs model networks with varying number of processors (P), number of IPG nodes/clusters (C), $ThroTTle$ values, and interconnect slowdowns (I). In our experiments, P ranged from 2 to 2,048, C from 1 to 8, $ThroTTle$ was varied to find the optimal value for minimizing runtime, and I simulated high- and low-bandwidth cluster interconnections.

Based on performance studies reported in [14,24], typical communication latencies and bandwidth slowdowns from integrated clusters to configurations connected through a high-speed interconnect are in the range of 3 to 100. Wide area network connections are typically 1,000 to 10,000 times slower than the internal intraconnects of a single cluster. In our experiments, we normalized the intra-cluster communication speed to unity. Simulations of inter-cluster communication assumed slowdown factors of 3, 10, 100, and 1,000. To simplify

Table 2
Expected runtime ($MaxQwgt$) without and with latency tolerance for varying interconnect slowdowns, $P = 32$, and $ThroTTle = 64$

		Interconnect slowdowns			
Case	Clusters	3	10	100	1000
No latency tolerance	1	507	507	507	507
	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
Max. latency tolerance	1	306	306	306	306
	2	305	469	763	3941
	3	324	548	2386	12705
	4	328	680	3297	21888
	5	359	768	4369	33092
	6	357	856	5044	52668
	7	371	893	5480	61079
	8	376	1048	5721	61321

the analysis, we also assumed that individual processors are homogeneous and divided as evenly as possible among the clusters.

Table 2 shows results of experimental runs analyzing the effect of varying numbers of clusters and interconnect speeds, for $P = 32$ homogeneous processors and $ThroTTle = 64$. The interconnect speeds indicate the slowdown factor relative to the intra-cluster communication speed. Results are presented both when no latency tolerance is achieved, and also with maximum latency tolerance. To be consistent with Table 1, runtimes are shown as non-dimensionalized values in thousands. The following conclusions can be drawn from these experiments.

As the interconnect speed is reduced, the slowdown experienced by utilizing additional clusters increases dramatically. For example, the runtime metric with no latency tolerance as shown in Table 2 is 4,102 when two clusters and an interconnect slowdown of 1,000 is assumed; however, the metric is 93,566 when eight clusters are assumed. Thus, performance deteriorates by almost a factor of 22.8. Instead, if we consider an interconnect slowdown of 3, the performance degradation is only 1.3. The same pattern also holds true when maximum latency tolerance is assumed.

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 corresponding rows in Table 2. 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 1,000. In contrast, results with two clusters indicate gains of 2.4, 1.8, 1.6, and 1.0, respectively, for the same interconnect slowdowns. These 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 columns in Table 2). For example, with an interconnect slowdown of 1,000, the runtime improvement factors when 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 factors are 1.6, 1.8, 2.1, 2.0, 2.1, 2.0, 2.1, and 2.1. In this case, results somewhat 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 class of applications, the IPG could be a viable environment if a high-speed interconnect (slowdown factor between 3 and 10) between clusters is available. Results in Table 2 comparing one and eight clusters with an interconnect slowdown of 3 show runtime deterioration factors of 1.24 and 2.04 with and without latency tolerance, respectively. Similar comparisons for an interconnect slowdown of 10 show deterioration factors of 3.65 and 4.60. These factors, being smaller than the number of clusters, indicate a relative speedup when the number of clusters increases.

Table 3 presents simulation results when the total number of processors P is varied, for interconnect slowdown $I = 3$ and $ThroTTle = 2P$. Both the expected runtime ($MaxQwgt$) and the load balance quality ($LoadImb$) with and without latency tolerance are reported, but only for one and eight clusters. The performance ratio between the two cluster configurations remains roughly constant across all processor counts. So too does the ratio between maximum and zero latency tolerance. Note that generally near optimal results are obtained when $ThroTTle$ is set to a value of $2P$; however, when $ThroTTle = 3P$ for $P = 1024$ and $C = 8$, $MaxQwgt$ and $LoadImb$ both improve to 188 and 7.22, respectively (compared to 214 and 8.95 as reported in Table 3).

Table 3 also demonstrates the scalability of our test application. The benefits of using more processors begins to decrease beyond $P = 128$ as is evident from the $MaxQwgt$ values. The quality of load balance also deteriorates

Table 3
Expected runtime and load balance quality without and with latency tolerance for varying number of processors, $I = 3$, and $ThroTTle = 2P$

		$MaxQwgt$		$LoadImb$	
Case	Processors	$C=1$	$C=8$	$C=1$	$C=8$
No latency tolerance	2	4526		1.00	
	4	2922		1.00	
	8	1568	2518	1.00	1.01
	16	910	1493	1.00	1.17
	32	507	968	1.01	1.48
	64	276	563	1.05	1.69
	128	169	405	1.19	2.42
	256	131	253	1.66	2.80
	512	111	214	2.47	4.69
	1024	105	214	4.16	8.95
	2048	102	170	7.47	14.33
Max. latency tolerance	2	3782		1.00	
	4	2014		1.00	
	8	1089	1245	1.00	1.00
	16	589	661	1.00	1.00
	32	306	376	1.00	1.13
	64	158	246	1.01	1.39
	128	85	176	1.05	1.98
	256	73	124	1.60	2.77
	512	61	103	2.47	4.14
	1024	55	95	4.04	7.79
	2048	60	86	8.14	13.43

rapidly when $P > 256$. This is because the problem size for our test application does not increase with P but is fixed at 50,000 elements (see Section 2.1).

Finally, to evaluate the effectiveness of MinEX versus the case where no partitioning is done, additional experiments were conducted. These results are reported in Table 4 where the settings are identical to those in Table 3 except that the MinEX partitioner was not invoked. As expected, the quality

Table 4
Expected runtime and load balance quality without and with latency tolerance for varying number of processors, $I = 3$, $ThroTTle = 2P$, and no partitioning

		MaxQwgt		*LoadImb*	
Case	Processors	$C=1$	$C=8$	$C=1$	$C=8$
No latency tolerance	2	6621		1.80	
	4	5434		2.57	
	8	3624	4712	2.81	2.70
	16	2825	3739	3.22	3.34
	32	1725	2207	3.37	3.45
	64	964	1294	3.50	3.78
	128	663	868	4.10	4.50
	256	407	524	4.38	4.87
	512	392	503	8.17	8.77
	1024	353	431	13.54	13.45
	2048	247	304	17.74	17.80
Max. latency tolerance	2	6561		1.81	
	4	5125		2.63	
	8	3142	3142	2.97	2.95
	16	1832	1910	3.10	3.05
	32	1036	1265	3.27	3.52
	64	560	776	3.41	3.93
	128	364	516	4.07	4.69
	256	205	328	4.24	5.35
	512	198	317	7.93	9.53
	1024	178	281	13.08	14.61
	2048	128	199	17.56	19.23

of load balance is severely affected, even for small numbers of processors. The expected runtimes also increase by about a factor of two. Other interconnect and cluster combinations showed significant improvements as well when using MinEX (but the results are not reported here due to length restrictions). For example, with $I = 100$ and $C = 4$, MinEX reduces the expected runtime from 17,752 to 3,297 if maximum latency tolerance occurs. Similarly, if no latency tolerance is possible, the improvement in runtime is from 18,323 to 3,667. In both cases, MinEX improves the runtime by approximately a factor of five.

5 Conclusions

In this paper, we presented a novel latency-tolerant partitioner, called MinEX, that is suitable for adaptive mesh applications executed in a parallel distributed fashion on NASA's Information Power Grid (IPG). MinEX not only balances processor workloads but also minimizes data movement and runtime communication, and can account for expected latency tolerance in the application. Our simulation results demonstrated that MinEX is a viable load balancer provided the IPG nodes are connected by a high-speed asynchronous interconnection network. Otherwise, applications would have to have little runtime communication and data set remapping overhead for low-speed wide area networks to be practical interconnects. For details on other applications, see [18]. Implementing a parallel version of MinEX and conducting a rigorous mathematical analysis are part of our future work. Finally, real distributed experiments using Globus are planned to complement the results presented in this paper.

Acknowledgements

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

References

[1] D. Abramson, R. Sosic, J. Giddy, and R. Hall, Nimrod: A tool for performing parametised simulations using distributed workstations, in: *Proceedings 4th IEEE Symposium on High Performance Distributed Computing* (Washington, DC, 1995) 112–121.

[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, in: *Proceedings 7th Symposium on the Frontiers of Massively Parallel Computation* (Annapolis, MD, 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, *International Journal of Supercomputer Applications*, 11 (1997) 212–223.

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

[6] J. Czyzyk, M.P. Mesnier, and J.J. Moré, The Network-Enabled Optimization System (NEOS) server, Preprint MCS-P615-1096, Argonne National Laboratory, Argonne, IL, 1996.

[7] S.K. Das, D.J. Harvey, and R. Biswas, Parallel processing of adaptive meshes with load balancing, *IEEE Transactions on Parallel and Distributed Systems*, to appear.

[8] S.K. Das, D.J. Harvey, and R. Biswas, Latency hiding in dynamic partitioning and load balancing of Grid computing applications, in: *Proceedings 1st International Symposium on Cluster Computing and the Grid* (Brisbane, Australia, 2001) 347–354.

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

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

[11] 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.

[12] T.L. Disz, M.E. Papka, M. Pellegrino, and R. Stevens, Sharing visualization experiences among remote virtual environments, in: *Proceedings International Workshop of High Performance Computing for Computer Graphics and Visualization* (Swansea, UK, 1995) 217–237.

[13] M.J. Djomehri, R. Biswas, R.F. Van der Wijngaart, and M. Yarrow, Parallel and Distributed computational fluid dynamics: Experimental results and challenges, in: *Proceedings 7th International Conference on High Performance Computing* (Bangalore, India, 2000) Springer-Verlag LNCS 1970, 183–193.

[14] I. Foster and N. Karonis, A grid-enabled MPI: Message passing in heterogeneous distributed computing systems, in: *Proceedings Supercomputing'98*, (Orlando, FL, 1998).

[15] 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`.)

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

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

[18] D.J. Harvey, Load balancing techniques for distributed processing environments, Ph.D. Thesis, The University of Texas at Arlington, Arlington, TX, 2001.

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

[20] W.E. Johnston, D. Gannon, and B. Nitzberg, Grids as production computing environments: The engineering aspects of NASA's Information Power Grid, in: *Proceedings 8th International Symposium on High Performance Distributed Computing* (Redondo Beach, CA, 1999) 197–204.

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

[22] J. Leigh, A.E. Johnson, and T.A. 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.

[23] M.J. Litzdow, M. Livny, and M.W. Mutka, Condor — a hunter of idle workstations, in: *Proceedings 8th International Conference of Distributed Computing Systems* (San Jose, CA, 1988) 104–111.

[24] 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, Hanover, NH, 1996.

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

[26] 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.

[27] 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.

Author Biographies

SAJAL K. DAS received his Ph.D. in Computer Science in 1988 from the University of Central Florida, Orlando. Currently, he is a Professor of Computer Science and Engineering, and also the Founding Director of the Center for Research in Wireless Mobility and Networking (CReWMaN) at the University of Texas at Arlington (UTA). Prior to 1999, he was on the faculty of Computer Science at the University of North Texas (UNT), Denton, where he served as the Director of the Center for Research in Parallel and Distributed Computing (CRPDC). Dr. Das is a recipient of the UNT Student Association's Honor Professor Award in 1991 and 1997 for best teaching and scholarly research; UNT's Developing Scholars Award in 1996 for outstanding research; and UTA's Outstanding Senior Faculty Research Award in Computer Science in 2001. He has been a Visiting Scientist at the Council of National Research in Pisa and a Visiting Professor at the Indian Statistical Institute in Calcutta. His current research interests include mobile computing, wireless networks, QoS in wireless multimedia and mobile Internet, parallel algorithms, grid computing, performance modeling and simulation. He has published over 180 research papers in these areas, delivered numerous invited talks, directed several funded projects, and holds four US patents in wireless mobile networks. He received Best Paper Awards at ACM MSWIM-2000, MobiCom'99, and PADS'97. Dr. Das serves on the Editorial Boards of JPDC, PPL, and PAA, and has guest-edited special issues of ACM Wireless Networks, JPDC, and IEEE Transactions on Computers. He served as General Chair of WoWMoM-2000 and PDC-WNMC-2001, General Vice-Chair of MobiCom-2000, HiPC-2000, and HiPC-2001, General Co-Chair of MASCOTS'98, Founding TPC Chair of WoWMoM'98 and WoWMoM'99, Program Vice-Chair of HiPC'99, and TPC member of numerous IEEE and ACM conferences. He is also a member of the IEEE TCPP Executive Committee.

DANIEL J. HARVEY obtained his Ph.D. in Computer Science from the University of Texas at Arlington (UTA) in 2001. Currently, he is an Assistant Professor in the Department of Computer Science at Southern Oregon University, Ashland. From 1992 to 2001, he was an Assistant Professor at Dallas Baptist University. He was nominated for the 2001 UTA Outstanding Doctoral Research Student Award. His research interests include parallel programming, load balancing, and cluster computing. Dr. Harvey has extensive experience in industry and was President of an automotive aftermarket value-added reseller for 13 years. He has considerable consulting experience working with firms such as Dunn & Bradstreet and Honeywell. In 1999, under a NASA summer grant, he conducted research at NASA Ames Research Center, Moffett Field, California. He is also active in athletics, completing his 5th marathon at Boston in 1997. He established the Dallas Baptist University cross-country and track programs in 1996, and served as their Head Coach through 2001.

RUPAK BISWAS received his Ph.D. in Computer Science from Rensselaer Polytechnic Institute, Troy, New York, in 1991, and is currently a Senior Computer Scientist with NASA Ames Research Center, Moffett Field, California. In the past, he was a Senior Research Scientist with Computer Sciences Corporation and with Veridian/MRJ, and a Staff Scientist with the Research Institute for Advanced Computer Science (RIACS), all at NASA Ames. He is the Group Lead of the Algorithms, Tools, and Architectures (ATA) Group that performs research in computer science technology for high-performance scientific computing. The ATA Group is part of the NASA Advanced Supercomputing (NAS) Division of NASA Ames. Dr. Biswas has published over 85 technical papers in journals and major conferences, given several invited talks at home and abroad, and guest-edited several journal special issues. He received the Best Paper Award for significant research contributions at Supercomputing'99, and the Best Student Paper Award at Supercomputing-2000. He was awarded the NASA Contractor Council Excellence Award in 1993 for his work on developing an automatic mesh adaptation procedure for unstructured meshes. He was a co-recipient of the 2001 NASA Group Achievement Award as a member of the Information Power Grid (IPG) Group which made outstanding contributions to the agency's mission. His current research interests are in dynamic load balancing for NUMA and multithreaded architectures, analyzing and improving single-processor cache performance for irregular applications, scheduling strategies for heterogeneous distributed multi-resource servers in the IPG, and the scalability and latency analysis of key NASA algorithms and applications.