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
NAS Systems Division
NASA Ames Research Center
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. Experimental results show that MinEX is an effective load balancer in a distributed IPG environment.

1. Introduction

The Information Power Grid (IPG) has been developed by NASA and other collaborative partners to harness the power of geographically distributed resources. There have also been numerous other attempts to develop computational grid capabilities. Refer to [4] for a comprehensive survey of current technology and grid-based systems. For example, Condor [7] was an early success in developing a distributed system to manage research studies at workstations around the world; however, it did not adequately deal with security issues that are important for a general computational grid implementation. On the other hand, The Globus Metacomputing Infrastructure Toolkit (`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.

With a goal to study the latency tolerance, partitioning and load balancing performance of parallel distributed computing applications on the IPG, 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 slowdowns are parameterized so that general conclusions can be drawn. Before presenting our contributions, let us summarize our previous work in this context. We have investigated a load balancing strategy, called PLUM [9], which is an architecture-independent framework geared towards adaptive numerical solutions. (It was also experimented with the above application as the test case.) PLUM globally partitions the computational mesh after each adaptation and determines whether rebalancing the load would lead to reduced total execution time. If an improvement in the load balance can be achieved, it utilizes an effective remapping algorithm to minimize the required data movement.

This paper proposes a novel partitioning approach that optimizes the two important steps of PLUM (balancing and remapping) as part of the partitioning process. The goal of this partitioner, called MinEX, is different from that of most other 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 also is able to compensate for latency tolerance on the IPG. Comparisons between MinEX and PLUM show that MinEX reduces the number of elements migrated. For example, with 32 partitions in our test case, PLUM redistributed 63,270 mesh elements in contrast to 30,548 elements when MinEX is used.

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

This paper is organized as follows. Section 2 introduces the computational application to be tested. Section 3 describes the new MinEX partitioner. Section 4 describes the experimental study, analyzes the results, and draws conclusions. 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 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 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 Figure 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.

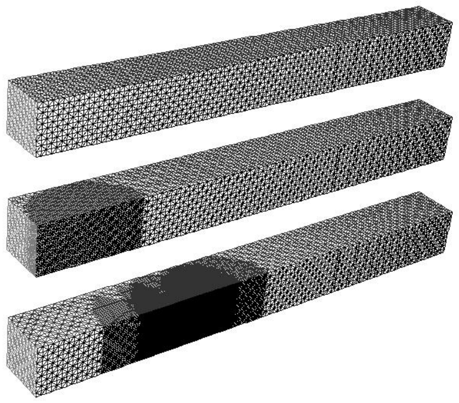

Figure 1. Initial and adapted meshes

The data used for the simulations presented in this paper were generated by an Euler solver [10] and uses tetrahedral elements [2] to represent the three dimensional mesh. A dual graph representation of the *original* mesh is used by our experiments for load balancing where each original tetrahedra is a vertex of the dual graph. An edge exists between two dual graph vertices if the elements share a face. We assume that units of computational and communication cost are equal. Mesh refinement consists of subdividing parent tetrahedral elements into two, four, or eight subelements in specified areas of the mesh. Subsequent refinements can further refine the child elements, thereby forming a refinement tree of tetrahedra for each original mesh element.

Each vertex v of the dual graph has two weights, $\mathtt{PWgt}_v$ and $\mathtt{RWgt}_v$, while each edge (v, w) has one weight, $\mathtt{CWgt}_{(v,w)}$. These weights refer respectively to processing, data remapping, and communication costs associated with processing a dual graph vertex. $\mathtt{PWgt}_v$ is the number of leaves in the refinement tree because these elements participate in the actual calculation. $\mathtt{RWgt}_v$ is the total number of elements in the refinement tree because the entire tree must be relocated when the vertex is reassigned to another processor. $\mathtt{CWgt}_{(v,w)}$ is the number of leaf faces that are adjacent to dual graph vertices v and w.

To realistically predict performance on a variety of distributed architectures, a configuration graph is also utilized. Vertices in this graph represent a cluster of processors. For the sake of the experiments presented in this paper, we assume that all processors in a cluster are homogeneous and that there is a constant bandwidth for intra-cluster communication. Each vertex in the configuration graph has an associated weight $\mathtt{Proc}_c \geq 1$ representing the processing slowdown factor for a cluster relative to the others. Likewise, edges $\mathtt{Connect}_{(c,d)} \geq 1$ represent the interconnect slowdown factor when a processor in cluster c communicates with a processor in cluster d. If $c = d$, $\mathtt{Connect}_{(c,c)}$ represents the slowdown associated with communication between processors in the same cluster c. Note that if $\mathtt{Proc}_c$ or $\mathtt{Connect}_{(c,d)}$ is unity, there is no slowdown (represents the most efficient connection in the network).

The following metrics 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.

- **Processing Weight** ($\mathtt{Wgt}^v$) is the computational cost to process vertex v assigned to a processor in cluster c:

$$\mathtt{Wgt}^v = \mathtt{PWgt}_v \times \mathtt{Proc}_c.$$

- **Communication Cost** ($\mathtt{Comm}_p^v$) is the cost to interact with all vertices adjacent to v but whose data sets are not local to processor p (assuming that v is assigned to p). Vertex w is adjacent to v, while c and d are the clusters respectively associated with the processors assigned to v and w:

$$\mathtt{Comm}_p^v = \sum_{w \notin p} \mathtt{CWgt}_{(v,w)} \times \mathtt{Connect}_{(c,d)}.$$

- **Redistribution Cost** ($\mathtt{Remap}_p^v$) 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 cost of unpacking and merging the data into existing data structures:

$$\mathtt{Remap}_p^v = \begin{cases} \mathtt{RWgt}_v \times \mathtt{Connect}_{(c,d)} & \text{if } c \neq d \\ 0 & \text{if } c = d \end{cases} .$$

  Assume that c is the cluster to which vertex v is assigned, and d is the cluster associated with the processor to which v is to be relocated.

  If the data set for v is already assigned to p, no redistribution cost is incurred, i.e. $\mathtt{Remap}_p^v = 0$. Similarly, if the data sets of all the vertices adjacent to v are also assigned to p, the communication cost, $\mathtt{Comm}_p^v$, is 0.

Additional metrics used in this work are defined below:

- **Weighted Queue Length** ($\mathtt{QWgt}(p)$) is the total cost to process the vertices assigned to p:

$$\mathtt{QWgt}(p) = \sum_{v \text{ assigned to } p} (\mathtt{Wgt}^v + \mathtt{Comm}_p^v + \mathtt{Remap}_p^v).$$

- **Total System Load** ($\mathtt{QWgtTOT}$) is the sum, over all procesors, of $\mathtt{QWgt}(p)$.
- **Heaviest Load** ($\mathtt{MaxQWgt}$) is the maximum value of $\mathtt{QWgt}(p)$ over all processors, and indicates the total time required to process the application.
- **Lightest Load** ($\mathtt{MinQWgt}$) is the minimum value of $\mathtt{QWgt}(p)$ over all processors, and indicates the workload of the most lightly-loaded processor.
- **Average Load** ($\mathtt{AvgQWgt}$) is $\mathtt{QWgtTOT}/P$, where P is the total number of processors.
- **Load Imbalance Factor** ($\mathtt{LoadImb}$) represents the quality of the partitioning and is $\mathtt{MaxQWgt}/\mathtt{AvgQWgt}$.

3. Proposed MinEX Partitioner

MinEX can be classified as a diffusive multilevel partitioner. The multi-level approach, originally introduced in [5], partitions a graph in three steps: contraction, partitioning, and refinement. Diffusive algorithms [3] utilize an existing partition as a starting point instead of partitioning from scratch. MinEX is unique in that it redefines the partitioning goal to minimizing $\mathtt{MaxQWgt}$ rather than balancing processing cost among partitions.

3.1. General Design

The partitioning steps of MinEX are discussed below.

Similar to other multilevel partitioners, the first step in MinEX is to contract the mesh to a reasonable size. 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, making the algorithm more flexible. If $|V|$ is the number of vertices in the mesh, contraction requires $O(|V|)$ steps 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 criteria followed by the partitioning algorithm (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 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 additional metrics, $\mathtt{Gain}$ and $\mathtt{MinVarv}$, need to be defined:

- $\mathtt{Gain}$ represents the change in $\mathtt{QWgtTOT}$ that would result from a proposed vertex move. A negative value would indicate that less processing is required after such a move. The partitioning algorithm favors vertex moves with negative or small $\mathtt{Gain}$ values that reduce or minimize overall system load.
- $\mathtt{MinVar}$ is computed using the workload ($\mathtt{QWgt}(p)$) for each processor p and the smallest load of any processor ($\mathtt{MinQWgt}$) in accordance with the following formula:

$$\mathtt{MinVar} = \sum_p (\mathtt{QWgt}(p) - \mathtt{MinQWgt})^2.$$

  In other words, $\mathtt{MinVar}$ 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 $\mathtt{QWgt}(p)$ values will have large $\mathtt{MinVar}$ components, this criteria will tend to move vertices away from processors that have high runtime requirements. $\Delta\mathtt{MinVar}$ is the change in $\mathtt{MinVar}$ after moving a vertex from one processor to another. A negative value indicates that the $\mathtt{MinVar}$ value has been reduced.

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

Table 1. `MaxQWgt` **values for varying** `ThroTTle`

	ThroTTle values						
Clusters	0	3	16	32	64	128	∞
1	1993	348	291	300	306	312	324
2	1847	748	320	304	305	318	345
3	2035	674	375	331	324	326	382
4	1868	761	412	352	328	371	425
5	1834	835	438	373	359	343	400
6	2081	898	481	391	357	361	427
7	1884	1032	505	383	371	369	414
8	1944	1102	531	434	376	380	435

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 indicates how varying `ThroTTle` affects the expected application runtimes (`MaxQWgt`). The `MaxQWgt` entries are non-dimensionalized values in thousands, and were obtained by running the experiments described in Section 4. Table 1 assumes a network of 32 homogeneous processors distributed over one to eight IPG nodes. The inter-cluster interconnect is assumed to have a third of the bandwidth of the intra-cluster interconnects. Results show that a `ThroTTle` value of 64 produces the lowest overall `MaxQWgt`.

3.3. Partitioning Data Structures

We describe here the data structures used by the MinEX partitioner to perform its multilevel algorithm.

`Mesh` The adaptive mesh whose format is $\{|V|, |E|, \texttt{vTot}, *\texttt{VMaP}, *\texttt{VList}, *\texttt{EList}\}$ 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 list of edges.

`VmaP` The list of active vertices (those that have not been compressed through multilevel partitioning).

`VList` The complete list of vertices. Each vertex, v, is defined by a `VList` record as $\{\texttt{PWgt}, \texttt{RWgt}_v, |e|, *e, mrg, lkup, *vmap, *heap, bdy\}$ where `PWgt` is the computational cost to process v, $\texttt{RWgt}_v$ is the redistribution cost to copy the data associated with v, $|e|$ is the number of adjacent edges associated with v, $*e$ is a pointer to the first edge associated with v (subsequent edges are stored contiguously), mrg is the vertex that was merged with v during a contraction operation (-1 if not merged), $lkup$ is the active vertex that contains v after a series of contractions (-1 if not merged), $*vmap$ is a pointer to the position of v in the active vertex table, $*heap$ is a pointer to the heap entry that relates to v and represents a potential reassignment of v, and bdy 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 $\{w, \texttt{CWgt}_{(v,w)}\}$ where w is an adjacent vertex to v and $\texttt{CWgt}_{(v,w)}$ 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 v to processor p. The min-heap is keyed by the `Gain` value.

`Stack` The stack of collapsed edges, $(\{v_1, v_2\})$. The pushed edges are refined in an order reversed from the one that they were compressed.

3.4. 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 $\texttt{CWgt}_{(v,w)}/(\texttt{RWgt}_v + \texttt{RWgt}_w)$ value is merged. This formula attempts to find edges with large communication weights while minimizing the potential cost of data set redistribution. The motivation for this strategy is to arrive at a contracted mesh with a small edge cut and with small data distribution cost.

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

3.5. Union/Find Algorithm

The contraction is implemented using a set Union/Find algorithm so that edges of existing vertices remain unchanged. For example, if an existing vertex is adjacent to v, accesses to its `EList` will check whether v has been merged. If it has, $lkup$ will quickly find the appropriate merged vertex. If $lkup$ is not current ($lkup > \texttt{vTot}$), the Union/Find algorithm will search the chain of vertices beginning with mrg to update $lkup$, so subsequent lookups can be done efficiently. Pseudo code describing the Union/Find algorithm is given in Fig. 2.

```
procedure $Find\ (v)$
if ($mrg\ ==\ -1$) return ($v$)
if ($lkup\,! =\ -1$) and ($lkup\ <=\ \texttt{vTot}$)
    then return ($lkup\ =\ Find\ (lkup)$)
    else return ($lkup\ =\ Find\ (mrg)$)
```

Figure 2. Union/Find algorithm pseudo code

3.6. Partitioning the Contracted Graph

The partitioning is performed when the graph contraction process is complete. Partitioning is efficient because the number of vertices is greatly reduced. The algorithm considers every remaining mesh vertex 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, and executed in heap order. After each reassignment, the heap is adjusted to reflect the new partition.

3.7. Refinement

The graph is restored to its original size by expanding pairs of vertices in reverse order 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 mrg and $lkup$ 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, 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. Experimental Study

In the experimental study that we present, two cases are considered. The first case is the most optimistic view in which processing activity can entirely hide the data set and communication latency. The second case is the most pessimistic view where no latency tolerance can be achieved.

MinEX was executed with actual application data to simulate mesh processing for a variety of system configurations. Individual runs simulate networks with varying numbers of processors (P), numbers of clusters (C), `ThroTTle` values, and interconnect slowdowns (I). P ranged from 2 to 2048, 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 [8], typical communication latencies and bandwidth slowdowns from integrated clusters to configurations with clusters connected through a high-bandwidth interconnect are in the range 3 to 100. Wide area network connections are 1,000 to 10,000 times slower than the internal intraconnects of a single cluster. For these experiments, we have assumed the intra-cluster communication slowdowns to be normalized to a value of unity. 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 evenly among the clusters.

We present our results in Tables 2 and 3 for $P = 32$. To be consistent with results in other tables of this paper, runtimes are shown in thousands of units. Table 2 charts the experimental results when no latency tolerance is achieved, while Table 3 assumes maximum latency tolerance.

Table 2. Expected runtimes (no tolerance)

	Interconnect Slowdowns			
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

Table 3. Expected runtimes(max. tolerance)

	Interconnect Slowdowns			
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

The following conclusions can be drawn:

- With greater interconnect slowdowns, the runtimes increase dramatically as additional clusters are used. For example, first compare the runtime metrics in the Table 2 for 2 and 8 clusters when an interconnect slowdown of 1000 is assumed. A slowdown ratio of $93,566/4,102 \approx 22.80$ is shown. Next, consider the same rows for an interconnect slowdown of 3. Now the slowdown ratio is $968/728 \approx 1.32$, which is a much smaller value. The same pattern holds true in Table 3.

- We can compare the effectiveness of latency tolerant algorithms to algorithms without latency tolerance, by measuring runtimes of each approach as the number of clusters and interconnect slowdowns are varied. The relative improvements, from algorithms without latency tolerance to algorithms with latency tolerance, are greater when more clusters are employed. This can be verified by comparing the same rows from Tables 2 and 3. For example, consider the case of 6 clusters. The difference in runtimes are $915 - 345 = 570$, $1717 - 856 = 861$, $8521 - 5044 = 3477$, and $76169 - 52668 = 23501$, respectively, for interconnect slowdowns of 3, 10, 100, and 1000. In contrast, the case with 2 clusters indicate improvements of $728 - 298 = 430$, $863 - 469 = 394$, $1228 - 763 = 465$, and $4102 - 3941 = 161$ for the same interconnect slowdowns. In general, the rows that correspond to more clusters show greater runtime improvements when employing latency tolerance. The same cannot be said when analyzing columns of the two tables where interconnect slowdowns are varied. For example, with an interconnect slowdown of 100, the improvements are $473 - 287 = 186$, $1228 - 763 = 465$, $2783 - 2386 = 397$, $3667 - 3297 = 370$, $5677 - 4369 = 1308$, $8521 - 5044 = 3477$, $10958 - 5480 = 5478$, and $11492 - 5721 = 5771$, respectively, for clusters 1 to 8. Instead, with an interconnect slowdown of 10, the corresponding runtime improvements are $473 - 287 = 186$, $863 - 469 = 394$, $1168 - 548 = 620$, $1361 - 680 = 681$, $1649 - 768 = 881$, $1717 - 856 = 861$, $1915 - 893 = 1022$, and $2178 - 1048 = 1130$. In this case, a clear pattern cannot be established.
- IPG solutions with this application are viable if high-bandwidth interconnects (slowdowns between 3 and 10) are utilized. Consider the columns of Tables 2 and 3 with an interconnect slowdown factor of 3. When comparing runtimes for 1 and 8 clusters, slowdowns of 2.04 and 1.24 are shown. Similarly, with an interconnect slowdown factor of 10, the slowdown factors are 4.60 and 3.65. These factors being smaller than the number of clusters indicate a speedup the number of clusters increases.
- To evaluate the effectiveness of MinEX versus the case where no partitioning was done, additional experiments were conducted. Consider the case where the interconnect slowdown factor is 100, and 4 clusters are used. If latency tolerance is employed, the partitioner reduced runtime from 17752 to 3297. Similarly, if no latency tolerance is employed, the improvement in runtime was from 18323 to 3667. Both show partitioner improvements in runtime by approximately a factor of 5. Other interconnect/cluster combinations show significant improvements as well.

5. Conclusions

In this paper, we have presented a latency-tolerant partitioner, MinEX, that is designed specifically for distributed computing environments such as the IPG. With adaptive mesh applications, MinEX effectively balances processor workloads, minimizes data movement and runtime communication, and can account for expected latency tolerance. An ongoing area of research is to formally compare MinEX to other popular partitioners such as PMeTiS [6] using the classical N-body application [1]. Real experiments with unstructured meshes on distributed clusters using Globus are also planned to complement the results presented in this paper.

References

[1] J. Barnes and P. Hut, “A hierarchical $O(N \log N)$ force calculation algorithm,” *Nature*, 324 (1986) 446–449.

[2] R. Biswas and R.C. Strawn, “A new procedure for dynamic adaption of three-dimensional unstructured grids,” *Applied Numerical Mathematics*, 13 (1994) 437–452.

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

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

[5] B. Hendrickson and R. Leland, “A multilevel algorithm for partitioning graphs,” Tech. Report SAND93-1301, Sandia National Laboratories, 1993.

[6] G. Karypis and V. Kumar, “Parallel multilevel k-way partitioning scheme for irregular graphs,” Tech. Report 96-036, University of Minnesota, 1996.

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

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

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

[10] R.C. Strawn and T.J. Barth, “A finite-volume Euler solver for computing rotary-wing aerodynamics on unstructured meshes”, *Journal of the American Helicopter Society*, 38 (1993) 61–67.