HARP: A Fast Spectral Partitioner

Horst D. Simon[1], Andrew Sohn[2], Rupak Biswas[3]

Abstract - Partitioning unstructured graphs is central to the parallel solution of computational science and engineering problems. Spectral partitioners, such recursive spectral bisection (RSB), have proven effective in generating high-quality partitions of realistically-sized meshes. The major problem which hindered their widespread use was their long execution times. This paper presents a new inertial spectral partitioner, called HARP. The main objective of the proposed approach is to quickly partition the meshes at runtime in a manner that works efficiently for real applications in the context of distributed-memory machines. The underlying principle of HARP is to find the eigenvectors of the unpartitioned vertices and then project them onto the eigenvectors of the original mesh. Results for various meshes ranging in size from 1000 to 100,000 vertices indicate that HARP can indeed partition meshes rapidly at runtime. Experimental results show that our largest mesh can be partitioned sequentially in only a few seconds on an SP2 which is several times faster than other spectral partitioners while maintaining the solution quality of the proven RSB method. A parallel MPI version of HARP has also been implemented on IBM SP2 and Cray T3E. Parallel HARP, running on 64 processors SP2 and T3E, can partition a mesh containing more than 100,000 vertices into 64 subgrids in about half a second. These results indicate that graph partitioning can now be truly embedded in dynamically-changing real-world applications.

1 Introduction

One of the most difficult problems to implement on a distributed memory parallel machine is a problem with a dynamically changing data structure, which requires repeated load balancing and which is coupled to an implicit computational solver [23]. This situation is typical for applications in computational fluid dynamics or computational structural mechanics, which involve grid adaptation, automatic mesh refinement or multizonal grid technologies [3]. An important aspect of the overall implementation of such dynamically changing applications, is the partitioning of the underlying grid. Mesh or graph partitioning algorithms for static grids have been extensively investigated in the last five years, and significant progress has been made both in improved heuristic algorithms, as well as in high quality software. In this paper we want to show, how a particularly successful approach for graph partitioning based on spectral algorithms can be extended to handle the dynamic case. Our goal is to combine the overall effectiveness of the spectral type partitioners in terms of reducing the cutsize of the partition, with some techniques, which use the dynamic character of the calculation to also produce a fast repartitioning of the grid.

The most general approach to mesh partitioning is to use generic combinatorial optimization techniques based on a cost function. Two methods that yield good suboptimal solutions are simulated annealing (SA) [16] and genetic algorithms (GA) [17]. SA is analogous to a method in statistical mechanics designed to simulate the slow cooling of a physical system. It works by iteratively proposing new partitions, evaluating their quality, and accepting them based on the Metropolis criterion. The method requires several user-specified parameters that makes it difficult to find good partitions in a problem-independent manner. GA are a model of machine learning which derives its behavior from the processes of evolution in nature. Such methods start with an initial population of randomly-generated partitionings. New partitionings are then generated from the current population using the natural processes of reproduction, crossover, and mutation. Individual partitionings that contribute to the minimization of an objective function are more likely to reproduce. Once again, a large number of parameters must be set for a successful partition. In general, stochastic optimization techniques when used on their own, can be slow, trapped in local minima, and depend on many application-specific parameters. However, these methods may be very useful in fine tuning an existing partition.

Another intuitive approach to mesh partitioning is to use clustering techniques. The nearest-neighbor algorithm in [19] generates initial clusters so that neighboring grid points are assigned to the same or neighboring partitions. These clusters are then modified using a boundary refinement procedure to improve the partitions. The greedy algorithm in [8] grows the first partition from a given starting point until the correct number of grid points has been included. Construction of the next partition begins from the boundary of the previous partition until the whole domain is decomposed. Despite its simplicity, it often yields partitions with low edge cuts. Since it is not a recursive process and the partitioning time is independent of the number of partitions, this algorithm is considered one of the fastest partitioners. Bandwidth reduction algorithms also belong to this class of mesh partitioning techniques. Essentially, if the mesh elements are renumbered to reduce the bandwidth of the adjacency matrix, a lexicographic decomposition of the mesh can be performed to obtain good partitions. The Reverse Cuthill-McKee (RCM) ordering scheme [5] is one of the most popular methods for bandwidth reduction; however, subdomains usually have bad aspect ratios. This problem can be reduced if the scheme is used recursively, as in recursive graph bisection (RGB) [22]. Two vertices at maximal or near-maximal distance in the graph are first determined. All other vertices are then sorted by distance from one of these extremal vertices, and partitioned to two subdomains. The RCM scheme is used

---

1 NERSC, MS 50B-4230, Lawrence Berkeley National Laboratory, Berkeley, CA 94720; simon@nersc.gov. This work was supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number 76SF00098.

2 Dept. of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ 07102; sohn@cis.njit.edu. This work is supported in part by the NASA JOVE Program, USRA RIACS, and MRJ Technology Solutions.

3 MRJ Technology Solutions, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035; rbiswas@nas.nasa.gov. This work is supported by NASA under contract NAS 2-14303.

In Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures,
Newport, Rhode Island, June 22-25, 1997.

to find the level structure, a convenient way of organizing the vertices in sets of increasing distance from one of the extremal vertices.

The class of geometry-based bisection algorithms recursively divide the mesh into two parts by exploiting its geometric properties. Recursive coordinate bisection (RCB) [22] sorts the mesh vertices according to their coordinates in the direction of the longest spatial extent of the domain. Half the vertices are then assigned to each subdomain, and the process is repeated recursively. This is a simple and intuitive technique, but one which provides poor separators as a result of excluding all graphical information. Inertial recursive bisection (IRB) [6] instead considers the inertial coordinate system, where the origin is the center of gravity of the mesh. The vertices are considered point masses with mass values set to the vertex weights. The vertices are then orthogonally projected onto the principle axis of this structure, and sorted into two sets. This technique is more expensive than RCB but generally produces much better results. IRB is especially used in conjunction with local refinement strategies such as the Kernighan-Lin (KL) heuristic [15]. Repeated pairwise exchanges are performed on an initial partition to improve the quality. A salient feature of KL is that sequences of perturbations are considered rather than single exchanges to bypass local minima.

A considerably less intuitive class of mesh partitioning algorithms are based on spectral methods. The most widely-used technique is Recursive Spectral Bisection (RSB) [22] that is derived from a graph bisection strategy [18] based on a specific eigenvector of the Laplacian matrix of the graph. In particular, the eigenvector corresponding to the second smallest eigenvalue gives some directional information about the graph. The special properties of this eigenvector have been extensively investigated by Fiedler [10]; hence, called the Fiedler vector. The computational challenge of the RSB algorithm is the efficient calculation of the Fiedler vector. RSB is regarded as one of the best partitioners due to its generality and high quality; however, the method is very expensive since it requires computing the Fiedler vector at each recursive step. The multidimensional spectral partitioning (MSP) [12] algorithm improves RSB by considering several cuts at each recursive step. For example, it can perform spectral octasection to partition a graph into eight sets using three eigenvectors. MSP requires less computations than RSB to generate the same partitions; however, they are still too slow for many applications. These algorithms are often combined with KL to improve the fine details of the partition boundaries.

The partitioning time for large meshes can be considerably reduced by contracting the graph. Multilevel algorithms reduce the size of the mesh by collapsing edges, partitioning the smaller graph, and then uncoarsening it back to obtain a partition for the original mesh. The most sophisticated schemes use a sequence of successively smaller contracted meshes, and smooth the partitions using KL during the uncoarsening phase. The multilevel implementation of RSB, called MRSB [2], calculates the Fiedler vector for the coarsest graph, and then prolongates it for the original mesh. Alternative graph contraction strategies are described in [12,25], but they all use spectral methods on the coarsest mesh. The fastest multilevel scheme to date is MeTiS [14], which claims to produce partitions that are of higher quality than those generated by spectral partitioning schemes. MeTiS uses heavy edge matching during the coarsening phase, a greedy graph growing algorithm for partitioning the coarsest mesh, and a combination of boundary greedy and KL refinement during the uncoarsening phase.

The HARP algorithm which will be discussed in this paper can be described in the context of the above approaches to graph partitioning fairly easily, as a combination of the efficiency of spectral algorithms (in terms of finding small cutsets), with the speed of IRB. A very closely related algorithm has been proposed in [4]. We will explore the relationship of HARP with spectral algorithms in section 2. In section 3 we will discuss the serial and parallel versions of HARP in more detail, and in section 4 we will present some numerical results. After a comparison to other (static) partitioning algorithms, we are going to demonstrate in section 6 the performance of HARP in the framework of an unstructured adaptive mesh refinement code for computational fluid dynamics, which solves for the flow around a helicopter blade.

2 Motivation and General Description of the Algorithm

2.1. Laplacian Eigenvectors as Euclidean Coordinates

The first important element in motivating and understanding the HARP algorithm is to take a fresh look at the geometric interpretation of the Laplacian eigenvectors. The view we take here is that the first several eigenvectors of the Laplacian matrix of a graph can be viewed as coordinates in Euclidean space. This view has been taken as early as [21], and was implicitly present in many investigations of spectral algorithms. For example spectral quadra and octasection as proposed by Hendricksen and Leland [13] can be viewed as taking the first two or three nontrivial eigenvectors of the Laplacian matrix of a graph as coordinates of the vertices of the graph in the plane or in three dimensional space. Qudrasection is then equivalent to finding a rotation and translation of the plane so that the new coordinate axis partition the vertices into four equal sets. Use of spectral coordinates makes the resulting cut sets relatively small.

Similarly, Chan, Gilbert, and Teng [4] used the Laplacian eigenvectors as Euclidean coordinates, and then performed inertial bisection with respect to this coordinate system. HARP differs from that in [4] in two ways, both related to the fact that we also consider the Laplacian eigenvalues:

(a) HARP does not a priori make a decision on the number of eigenvectors to compute. Instead, HARP compares the magnitude of the corresponding eigenvalue to the smallest nonzero Laplacian eigenvalue. Eigenvalues which have grown above a certain threshold are discarded. Our numerical results in section 4.1 indicate that even for very large graphs, a few (less than a hundred) eigenvalues are sufficient to capture the global properties of the graph. A physical analogue of this procedure is the dynamic analysis in structural engineering. It is common engineering practice to compute a few of the smallest eigenvalues and vectors of the finite element model of a large structure, and then use the subspace spanned by these few eigenvectors for an analysis of the dynamic response of the structure to wind loading or to an earthquake. HARP uses a similar heuristic argument to claim that the essential features of a graph are represented in a relatively small subspace spanned by the smallest Laplacian eigenvectors.

(b) After a set of smallest eigenvectors has been selected, HARP uses the scaled eigenvectors as coordinates. Each eigenvector is scaled by square root of the inverse of corresponding eigenvalue. We call Laplacian eigenvectors scaled in this way the spectral coordinates of the graph. In this way the eigenvector corresponding to the smallest non-zero eigenvalue, which is often called Fiedler vector, will be the most heavily weighted coordinate direction. Since the Fiedler vector has been proven to be useful for partitioning in many experiments, this scaling of the vectors results in emphasizing the most important coordinate direction for bisection.

Another way to motivate the scaling by the values is that in this way we construct the best low rank approximation to the (pseudo) inverse of the Laplacian matrix. This of course begs the question what relationship there is between the (pseudo) inverse of the Laplacian matrix of a graph and any geometric embedding in Euclidean space. There are some more involved relationships, which will be discussed in a forthcoming paper.

We have thus argued that Laplacian coordinates are a canonical way to embed a graph in Euclidean space, and that recursive inertial bisection using this new coordinate system is an effective partitioning algorithm, which combines the efficiency of RSB with the speed of recursive inertial bisection. We will demonstrate this with a set of numerical tests on some standard meshes in section 4.

2.2 Dynamic Partitioning

So far all we have constructed is yet another static partitioner and added just another new variation to the existing knowledge. In order to make this partitioner useful in the context of a dynamically changing calculation, we need to make two additional observations.

Observation 1 For many (but not all) dynamically changing calculations, the changing computational load can be easily expressed as a graph partitioning problem with dynamically changing vertex weights. For example, in a simple case of adaptive unstructured grid calculations with triangular elements, we can consider the coarsest mesh as the one to be used with a graph partitioner, all elements being weighted equally with one. If the mesh gets refined at a later stage in the calculation, we don't need to partition the refined mesh. We can equally well partition the coarse mesh, but change the vertex weights. Any refined triangle will now have the weight four (or any other weight reflective of the increased amount of calculation for the refined mesh). This implies that we would not partition across a refined element. Even though this may be suboptimal from the partitioning point of view, it is very sensible from an implementation point of view, since we do not want to split the data structures associated with a refined element across multiple processors.

There is one set of applications where this model of changing vertex weights does not apply: these are applications where topological changes occur. In the finite element world, the canonical example would be crash codes, where previously disconnected parts of a mesh may have contact and then interact. This situation is discussed in detail by Diniz et al. [7], who also present a distributed memory implementation. Our approach is not well suited to handle topological changes.

Observation 2 The success of many practical implementations of graph partitioning algorithms rests on the application of multilevel schemes, as was discussed in section 1. Multilevel schemes work, because even a very coarse approximation of the graph can given some very good general information about how to optimally partition the graph in a global sense.

Combining these observations is the foundation for the HARP algorithm for dynamic partitioning. HARP consists of two parts:

(a) Precomputation of the spectral basis. We compute once and for all a spectral basis set of eigenvectors for the coarsest mesh in a given simulation. Although this calculation may be costly, it needs to be done only once for a given mesh. Since the same geometry and the same mesh are often used over and over again for design studies, the cost of the initial eigenvector calculation can be amortized over many simulations. In our current work we perform the initial eigenvector calculation with a shift-and-invert Lanczos algorithm described in [11]. We claim that the spectral basis, even for a coarse graph, captures the essential features of the graph, and can be used for effective partitioning.

(b) Repartitioning because of dynamic changes. At any time during the simulation when the characteristics of the calculation are changing because of refinement, derefinement, adaptation, etc. we compute a new vertex weight vector corresponding to the changed computational load. We repartition the graph with recursive inertial bisection in the spectral coordinates for the coarsest mesh. The change in vertex weights will affect the load balancing and hence the distribution of partitions, but it does not affect the initially computed spectral coordinates. Hence the repartitioning step is very fast, but continues to have the spectral information available, which make repartitioning also very efficient, and comparable to spectral partitioners.

3 The HARP Algorithm

We will not discuss the precomputation phase here. This is well documented elsewhere, and we simply used a Cray library routine on the C90 to precompute the eigenvectors. Instead, we will list the execution times of the eigen solver for the meshes used in the report.

As was mentioned before, the serial version of the repartitioning is essentially equivalent to inertial recursive bisection (IRB). Our implementation follows exactly this algorithm as described in [9]. The only difference is that IRB in [9] was physically motivated, i.e. based on a physical meaningful mesh with coordinates in three dimensional Euclidean space. Here we are using spectral coordinates in a generally larger than three dimensional space, with a cut-off depending on the growth of the Laplacian eigenvalues.

Inertial recursive bisection involves several components: The original eigenvector evec[v][n], where n is the number of eigenvectors of the grid and v is the number of vertices. Given the original n eigenvectors, the inertial center center[n] of the unpartitioned vertices will be computed, and in turn the inertial matrix inertia[n][n]. Inertial center center[n] needs n components each of which bears the inertial distance between the vertices and the center. Inertia[n][n] indicates how far the n inertial vectors are away from each other. The following algorithm briefly outlines HARP.

```
for (i=0; i<log(npart); i++) { /* npart = total # of partitions */
   for (j=0; j<2^i; j++) {
   1    Find an inertial center of the unpartitioned vertices
   2    Construct an inertial matrix using the inertial vector
   3    Symmetrize the inertial matrix
   4    Find the eigenvectors of the inertial matrix
   5    Project the vertex coordinates
        on the dominant inertial direction (eigenvector 0)
   6    Sort the projected coordinates
   7    Divide the unpartitioned vertices into two sets
        according to the sorted values
   }
}
```

Specifically, each step of the inner loop can be implemented as follows:

```
for (i=0; i<v; i++)          /* find inertial center */
   for (j=0; j<n; j++) center[j] = center[j] + evec[i][j];
for (i=0;i<v;i++) {          /* compute the inertial distance */
   for (j=0;j<n;j++)
      for (k=0;k<n;k++)
         inertia[j][k] = inertia[j][k] +
            (evec[i][j] – center[j]) * (evec[i][k] – center[k]);
for (i=0;i<n;i++)            /* symmetrize the inertial matrix */
   for (j=i+1;j<n;j++) inertia[j][i] = inertia[i][j];
inertial_eigenvector[n] =
   compute the dominant eigenvector of inertia[n][n];
for (i=0; i<v; i++)          /* project */
   for (j=0; j<n; j++)
      key[j] = key[j] + evec[i][j] * inertial_eigenvector[j];
sort key in an ascending order using float radix sorting;
split the sorted key into half;
place the two partitions each into an appropriate place.
```

The steps listed above are only for presentation purposes. Numerous steps are missing from the steps as they will unnecessarily complicate the understanding of the overall organization. Two routines of TRED2 and TQLI are used to find eigen vectors. They are derived from EISPACK, the eigen system subroutine package. TRED2 subroutine reduces a real symmetric matrix to a symmetric tridiagonal matrix using and accumulating orthogonal similarity transformations. TQLI subroutine finds the eigenvalues and eigenvectors of a symmetric tridiagonal matrix by the QL method. A 32-bit float radix sorting is used in the sorting step. We have written this routine from scratch. The float radix sorting is based on IEEE floating point standard, where bits 0..22 are significand, the bits 23..30 are exponent, and the bit31 is the sign bit. The radix of eight bits (the bucket size of 256) is used in the implementation.

Before we discuss the performance of HARP, we shall briefly identify how each of the above steps performs in terms of execution time. The most time consuming step is the inertial matrix computation step, which consists of three nested loops. The second most time-consuming step is sorting. It appears that the eigen solver can be a major bottleneck but it turned out trivial. For small problem size of below 10,000 vertices, the eigen solver can be of significance. However, for large problem sizes, the solver is a fraction of the overall computation time. We list some plots in Fig. 1 to show the distribution of the individual steps.

The results in Fig. 1 indicate that the majority of the times is spent on computing the inertial matrix of the unpartitioned vertices. Again, the second most time consuming step is the sorting step which occupies approximately 20%. There is a slight difference for the two grids. For a larger grid, the sorting time increases. As we shall come back to this issue later, the main target of parallel HARP is therefore the inertial computation time.

A parallel version of HARP has been designed and implemented on SP-2 [1] and T3E [20]. Two types of parallelism are used: loop level parallelism and recursive parallelism. The primary objective of reporting the parallel version in this paper is to demonstrate that HARP can be effectively parallelized and used in parallel environments. Significant performance improvement is expected in the near future. Porting a working SP-2 version of HARP to T3E was not straight forward due to some difference in machine architecture and compiler. Readjustment and even recoding of some functions were needed especially for floating point radix sorting. The details of parallel HARP are not included in this report. Instead, we will list some experimental results in the following sections.

Two of the five modules of HARP have been parallelized to date. In iteration 0, all the eight processors work together to find the inertial center of the unpartitioned vertices. This step is the most expensive since it involves all the unpartitioned vertices and their original eigenvectors in order to find their relative position in M-dimensional space. In comparison, the second step of finding the eigen vectors of the inertial matrix of dimension M is relatively trivial for large meshes and is therefore not parallelized. The third step, where the vertex coordinates of the unpartitioned vertices are projected onto the major inertial direction (corresponding to eigenvector 0) is somewhat expensive, and has also been parallelized. Sorting is still done sequentially in the current parallel version of HARP. The final step, where the unpartitioned vertices are divided into two sets, requires a negligible amount of time and is thus not parallelized. The most time-consuming modules of parallel HARP are to find the inertial matrix of the unpartitioned vertices, to project them onto the dominant inertial direction, and to sort the projected coordinates. This can be seen from the histograms in Fig. 2.

The current parallel version parallelizes only the inertial matrix construction and the projection modules. These still require 31% and 17% of the total time, respectively. Sorting is done sequentially in the current version, and constitutes more than 47% of the total partitioning time. The sorting module will be parallelized in the future that will result in significant performance improvement. There is also scope for substantial improvement in the first step where blocking send/receive commands are used.

4 Results

Thsoe test meshes and their characteristics are first listed in the section. The interplay betwwen the number of eigenvectors and the partition quality is explained using 128 partitions, followed by the realtionship between the number of partitions and partition quality

4.1 Test meshes and experimental settings

To verify the performance of HARP, we have done substantial experimentation over the last three years. The IBM SP-2 installed at

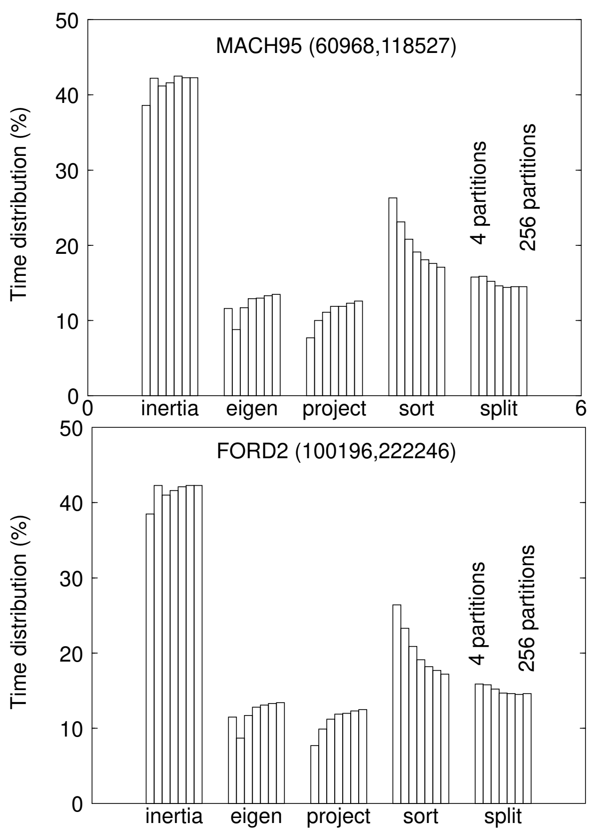

Figure 1: Time distribution on a single processor SP2.

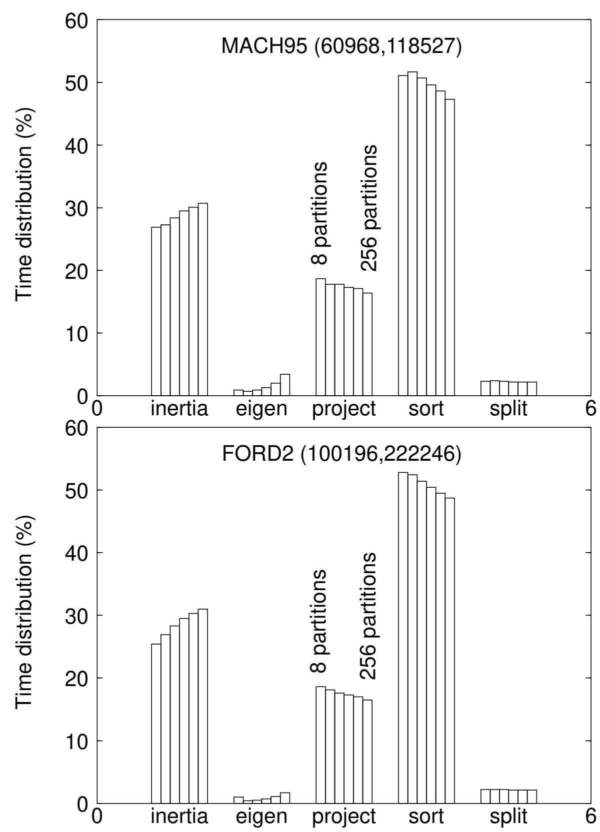

Figure 2: Time distribution on an 8-processor SP2.

	SPIRAL	LABARRE	STRUT	BARTH5	HSCTL	MACH95	FORD2
Type, 2D or 3D	2D	2D	3D	2D	3D	3D	3D
Number of vertices V	1200	7959	14,504	30,269	31,736	60,968	100,196
Number of edges E	3191	22,936	57,387	44,929	142,776	118,527	222,246

Table 1: Characteristics of the seven test meshes.

NASA Ames Research Center and the Cray T3E installed at NERSC, Lawrence Berkeley Laboratory are used in this study. While the main emphasis of this report is on the evaluation of the new HARP algorithm, we will briefly present some parallel results in the context of dynamically-changing adaptive mesh computations.

Seven different two- and three-dimensional test meshes are used in this study. They varied in size from 1200 vertices to more than 100,000 vertices. Table 1 shows the characteristics of the test meshes. SPIRAL is a very small toy grid which is a long chain geometrically arranged in a spiral. This mesh has no computational significance other than to serve as a difficult test case for partitioners. STRUT is a three-dimensional mesh used in civil engineering problems for structural analysis. BARTH5 is a dual graph for a four-element airfoil. HSCTL is a 3-dimensional mesh for a high-speed civil transport configuration. MACH95 is a tetrahedral mesh around a helicopter rotor blade. FORD2 is a surface mesh of a Ford car.

Table 2 lists the precomputation times of the eigen solver for the test meshes on a C90. The eigenvectors are computed in the precomputation stage. Once they are computed, they are used over and over again for the next experiments.

Test meshes	10 eigenvectors		20 eigenvectors		100 eigenvectors	
	mem	time	mem	time	mem	time
SPIRAL	0.3	0.54	0.4	0.98	0.6	4.71
LABARRE	2.1	4.25	2.2	6.25	3.5	29.73
STRUT	3.9	8.50	4.2	17.26	6.5	55.63
BARTH5	7.6	15.40	8.2	22.04	13.0	104.03
HSCTL	9.1	23.11	9.8	29.48	14.8	144.93
MACH95	39.2	192.68	40.5	209.56	50.1	687.89
FORD2	26.7	60.25	28.7	84.39	44.6	386.52

Table 2: Precomputation times on Cray C90, performed once and for all. (mem = memory size in mega words; time in seconds.)

We note from the table that the eigenvector computation times are not substantial considering that they are done once and only once for the lifetime of the meshes. The maximum memory usage is also limited to 50 mega words on Cray C90. It should be noted that the eigensolver time does not linearly increase as the number of eigenvectors increases. For example, the solving time of Ford2 is 60 seconds for 10 eigenvectors. When the number of eigenvectors is increased to 100, the solving time is increased slightly more than 6 times. This relatively slow rate of increase indicates that solving more than 100 eigenvectors is not prohibitively expensive if such number of eigenvectors is desired. As we will shows shortly, we find that 10 eigenvectors are suitable for our purposes.

Two parameters characterize the performance of all graph partitioning algorithms: the number of cut edges C and the total partitioning time T. Throughout this report, we will compare these parameters whenever appropriate.

We have performed three types of experiments. First, we identify the partition quality in terms of the number of eigenvectors that are used. Results do not depend on whether the serial or the parallel version of HARP is used. The experiment is thus performed on a single processor. Both the number of cut edges and the execution time will be presented to identify the trade-off between partition quality and execution time. Second, we identify the partition quality across different grids when the number of eigenvectors remain fixed. This experiment is also independent of sequential or parallel settings. It is thus performed on a single processor. Third, we run the parallel version of HARP on more than one processor. Partition quality remains unchanged from that for the serial version. Only the execution time will therefore be investigated.

Several other parameters are used throughout the study: V is the number of vertices, E is the number of edges, M is the number of eigenvectors of the original grid, P is the number of processors, and S is the number of sets (or partitions). The words sets and partitions are used interchangeably throughout this paper.

4.2 Number of eigenvectors and partition quality

Figure 3 illustrates the effect of the number of eigenvectors used on the partition quality and the execution time for 128 partitions. Both the number of edges cut and the execution time are normalized by their respective values when using only one eigenvector. It is clear that the solution quality improves for all the meshes except SPIRAL as the number of eigenvectors is increased. There is a drastic change

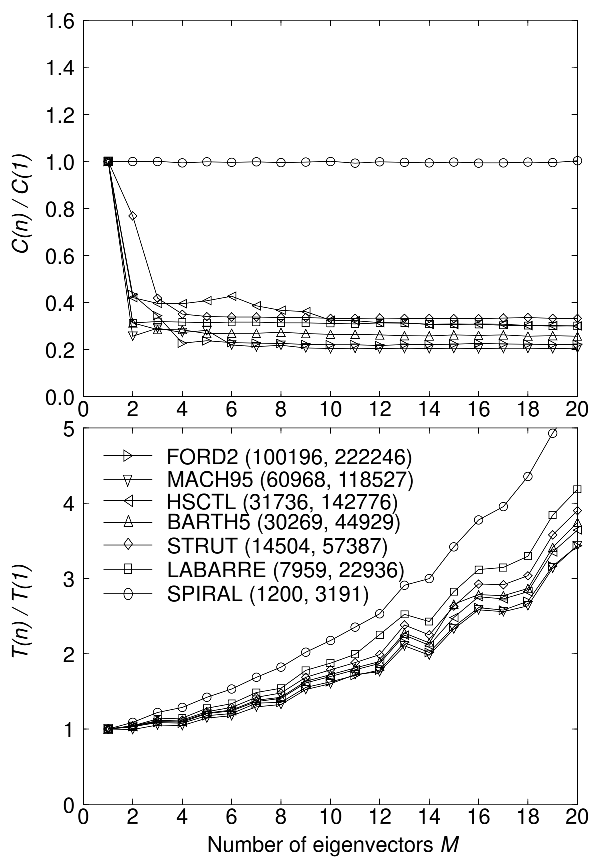

Figure 3: Effect of the number of eigenvectors on the number of cut edges and execution time for 128 sets.

when two eigenvectors are used instead of one. A gradual improvement is noticed for up to 10 eigenvectors. There is very little reduction in the number of cut edges beyond M=10. The reason that the partition quality for SPIRAL remains essentially unchanged is because it is geometrically a spiral in cartesian coordinates. However, in eigenspace, it is a long chain and its spectral property can be captured with only one eigenvector.

The execution time, on the other hand, keeps increasing as the number of eigenvectors increases. For 20 eigenvectors, the execution time has increased almost four-fold. There is a clear trade-off between the solution quality and the execution time. In fact, we reach a point of diminishing returns beyond a certain number of eigenvectors. The partition quality improves only slightly at the cost of significantly higher execution time.

Table 3 shows the absolute number of edge cuts and the execution time for MACH95. The execution times are for a single processor of an SP2. The table clearly indicates that increasing the number of eigenvectors is beneficial for the partition quality. However, doing so will significantly increase the partitioning time. This time and quality trade-off has been observed for other meshes.

4.3 Number of partitions and partition quality

In the previous section, we examined the relationship between the number of eigenvectors used and the partition quality for 128 partitions across the seven meshes. In this section, we look at how the number of eigenvectors affects the quality in terms of number of partitions. Figure 4 presents the number of cut edges and the execution time for two meshes: HSCTL and FORD2.

Four observations can be made from the results in Fig. 4. First, the partition quality improves as the number of partitions increases. Second, when the two meshes are cross-compared, the larger meshes

# of partitions	Edge cuts							Execution time						
	1 EV	2 EVs	4 EVs	6 EVs	8 EVs	10 EVs	20 EVs	1 EV	2 EVs	4 EVs	6 EVs	8 EVs	10 EVs	20 EVs
2	817	817	817	817	817	817	817	0.186	0.193	0.202	0.223	0.249	0.298	0.614
4	2442	1657	1657	1657	1657	1657	1657	0.360	0.372	0.390	0.433	0.484	0.583	1.214
8	5734	3283	3514	3773	3733	3728	3786	0.543	0.553	0.580	0.647	0.724	0.871	1.823
16	12312	5020	5431	5770	5693	5685	5784	0.729	0.741	0.777	0.867	0.970	1.166	2.442
32	25441	8443	8710	8827	8662	8145	7866	0.920	0.927	0.973	1.084	1.213	1.460	3.073
64	51651	13495	13404	12577	12818	10798	10741	1.110	1.117	1.173	1.309	1.469	1.769	3.735
128	72512	18542	19743	15874	15822	14803	14930	1.304	1.298	1.368	1.538	1.730	2.089	4.483
256	74109	28059	28798	21405	21870	20204	20118	1.491	1.483	1.571	1.782	2.018	2.489	5.260

Table 3: Effects of the number of eigenvectors on edge cuts and execution time for MACH95 on a single-processor SP-2.

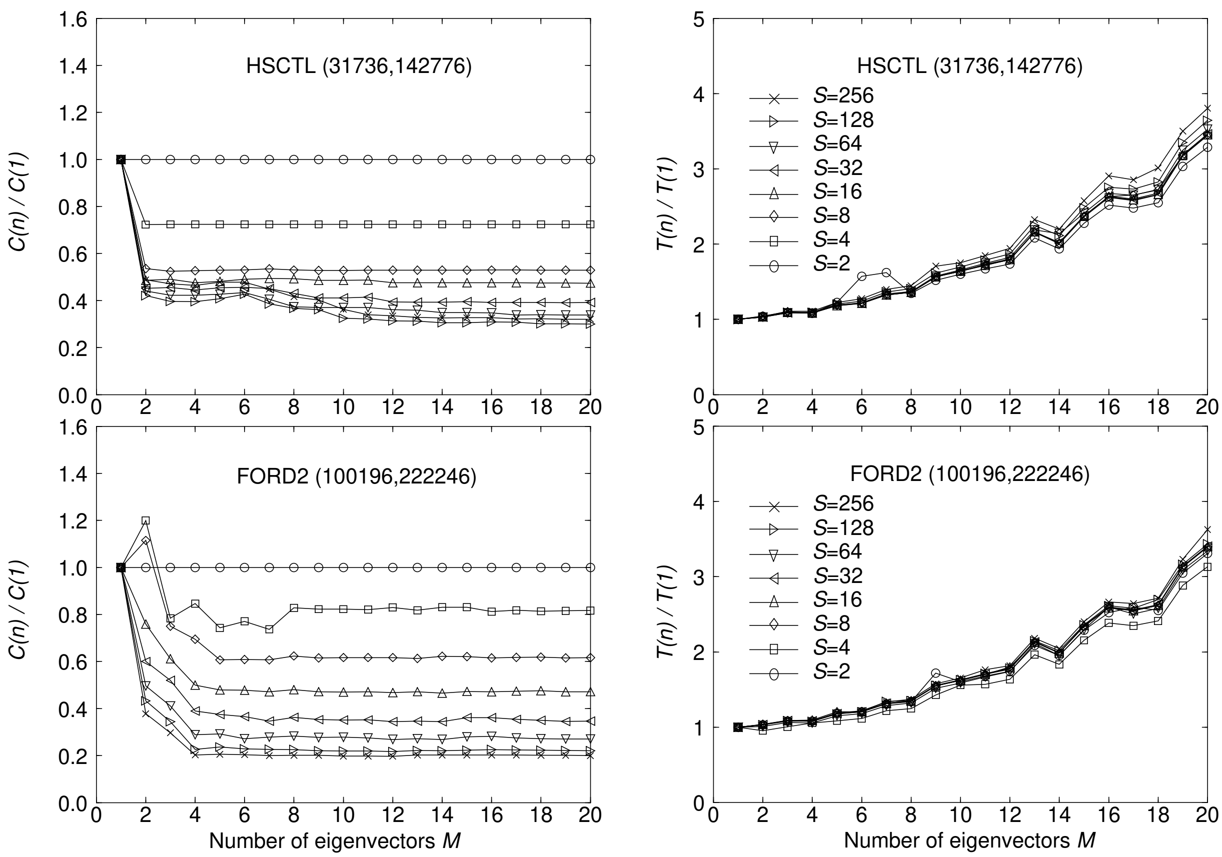

Figure 4: Effects of the number of eigenvectors on edge cuts and execution time for different number of partitions.

shows greater improvement in quality with more partitions. This is because we have more fine-grained control on how the partitions are generated. Third, the conclusions about partition quality versus the number of eigenvectors that were drawn from Fig. 3 for 128 partitions hold true for any number of partitions. Fourth, it should be noted that the nature of the normalized execution time does not change across different meshes. Contrary to the expectation of increased execution time, larger meshes tend to give lower execution time as the number of eigen vectors increases. Furthermore, as the number of eigen vectors increases, the execution times tend to settle in, resulting in less fluctuation.

5 Comparative Performance of HARP

The performance of both serial and parallel HARP is analyzed in the section. The serial version of HARP is first compared against another serial mesh partitioner. Preliminary results of parallel HARP are presented to demonstrate that HARP can be effectively parallelized on large-scale distributed-memory multiprocessors. The HARP results are based on 10 eigen vectors.

5.1 Serial performance of HARP

The HARP results are compared with the MeTiS2.0 multilevel partitioner. All HARP results in this section are based on 10 eigenvectors, and are denoted as HARPα. Two parameters are used for comparison: number of edge cuts and partitioning time. All execution times are based on a single-processor SP2. Tables 4 and 5 show the absolute numbers of edge cuts and execution times on SP2.

Table 6 shows the execution times of HARP on a Cray T3E. We find from the table that the T3E results are comparable to SP2 results listed in Table 5. The difference in the execution results comes from the machine's absolute performance and compiler optimization. SP2 consists of Power2 processors which can issue up to six instructions per clock while T3E consists of DEC Alpha 21164 processors which can issue up to four instructions per clock. The higher superscalar capability coupled with wider memory bandwidth has contributed to the higher performance on SP2.

# of sets	Spiral	Labarre	Strut	Barth5	Hsctl	Mach95	Ford2
2	0.005	0.036	0.069	0.144	0.151	0.288	0.477
4	0.010	0.081	0.152	0.313	0.331	0.643	1.052
8	0.017	0.125	0.227	0.479	0.501	0.997	1.621
16	0.025	0.168	0.298	0.635	0.665	1.342	2.188
32	0.037	0.215	0.366	0.782	0.818	1.664	2.748
64	0.056	0.268	0.442	0.928	0.971	1.975	3.266
128	0.089	0.340	0.534	1.086	1.132	2.280	3.761
256	0.149	0.441	0.656	1.281	1.324	2.609	4.270

Table 6: Execution times of HARPα in seconds on a single-processor T3E, using 10 eigenvectors.

Figure 5 plots the ratio of HARPα to MeTiS2.0. Figure 5(a) shows that HARPα gives partitions that are of poorer quality than MeTiS2.0. We find that the maximum overall difference is between 30% and 40%. It should be noted however that the HARPα results are based on 10 eigenvectors.

The execution times shown in Fig. 5(b) indicate that HARPα is more than twice as fast as MeTiS2.0. As we shall discuss in the next section, this is precisely the purpose of developing HARP. Since dynamically-changing computations require rapid runtime mesh repartitioning, this fast algorithm is perfectly suitable for our purposes. The fact that the partition quality is somewhat poor is not a major concern when dealing with adaptive computations. Since repartitioning has to be performed fairly frequently, it is more important to decrease the partitioning time than reducing the number of cuts.

5.2 Parallel performance of HARP

The main target of a preliminary version of parallel HARP is the step that computes the inertial matrix of the unpartitioned vertices. This module has been parallelized, as well as the projection step. A brief profile of the execution times for the individual modules for the sequential and parallel versions of HARP are shown in Figs. 1 and 2. The sorting step is the most expensive module in parallel HARP as it requires almost half the total execution time. Our next step, therefore, is to parallelize the sorting step.

# of sets	SPIRAL		LABARRE		STRUT		BARTH5		HSCTL		MACH95		FORD2	
	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2
2	9	9	169	144	82	82	109	86	1484	576	817	815	324	379
4	29	29	423	325	539	528	296	201	1958	1322	1657	1623	911	817
8	67	65	759	530	1027	1005	513	381	3180	2393	3731	3161	1826	1303
16	151	145	1150	864	1970	1939	855	588	5770	4371	5687	4600	3062	2146
32	301	290	1775	1381	3757	3261	1315	985	9652	6970	8664	6128	4732	3203
64	623	589	2667	2132	6879	4947	2012	1561	15896	10306	11557	8467	7561	4928
128	1234	985	4093	3227	8723	7287	3186	2427	22454	15102	15001	10981	11318	7616
256	2156	1526	6140	4806	13263	10551	4954	3672	34980	21857	20954	13966	17425	11332

Table 4: Comparison of the number of cut edges for varying number of partitions. The HARPα results are based on 10 eigenvectors. The MeTiS results are based on version 2.0.

# of sets	SPIRAL		LABARRE		STRUT		BARTH5		HSCTL		MACH95		FORD2	
	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2	HARPα	MeTiS2
2	0.011	0.02	0.043	0.10	0.103	0.19	0.149	0.28	0.157	0.48	0.298	0.79	0.488	1.18
4	0.013	0.03	0.078	0.22	0.137	0.42	0.286	0.60	0.300	1.00	0.583	1.62	0.989	2.40
8	0.020	0.05	0.118	0.33	0.208	0.65	0.429	0.88	0.451	1.84	0.871	2.42	1.424	3.59
16	0.029	0.11	0.161	0.50	0.279	0.92	0.578	1.21	0.605	2.24	1.166	3.17	1.899	4.78
32	0.042	0.14	0.207	0.70	0.355	1.22	0.776	1.59	0.765	2.93	1.460	4.29	2.377	5.92
64	0.062	0.21	0.261	0.90	0.437	1.65	0.920	2.08	0.926	3.76	1.769	5.46	2.865	7.50
128	0.098	0.28	0.332	1.18	0.536	2.17	1.057	2.70	1.104	4.90	2.089	6.77	3.371	9.23
256	0.164	0.45	0.441	1.56	0.670	2.87	1.257	3.29	1.315	5.97	2.489	8.23	3.901	11.35

Table 5: Comparison of the execution times in seconds on a single-processor SP2. The HARPα results are based on 10 eigenvectors.

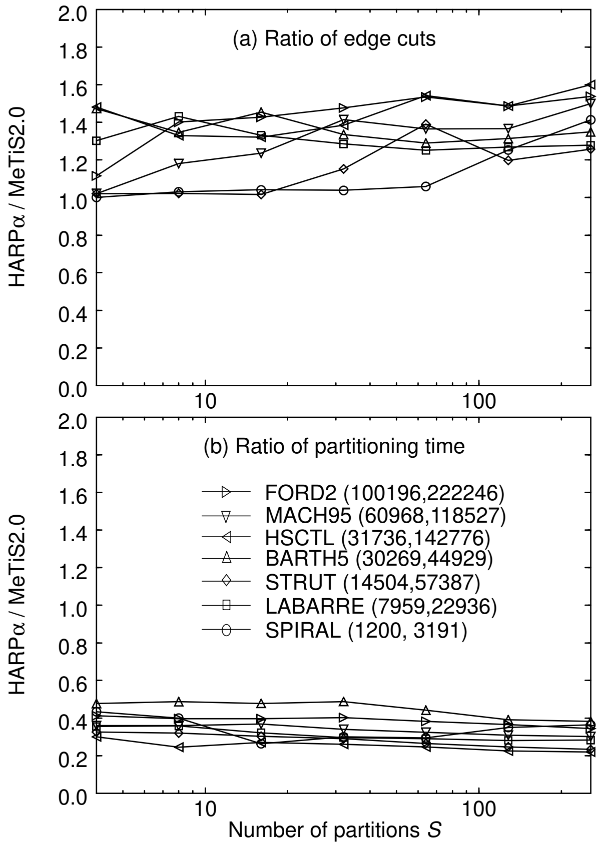

Figure 5: Comparison between HARPα and Metis2.0 on SP-2 in terms of edge cuts and execution time.

Execution times on up to 64 processors of an SP2 and T3E are presented in Tables 7 and 8 when parallel HARPα is applied to the two largest test meshes. For a given number $P > 1$ of processors, the meshes were partitioned into 2^0P, 2^1P, ..., 256 subgrids. For comparison, the times for the serial version of HARPα are also shown for up to 256 partitions. As indicated earlier, the current parallel implementation can be vastly improved. The main purpose of presenting these results here is to demonstrate that HARP can be effectively parallelized.

Three key observations can be made from these results. First, the parallel code shows modest speedup as the number of processors increases while keeping the total number of partitions unchanged. For example, the speedups are about 5.5X, 6.5X, and 7.6X on 64 processors for 64, 128, and 256 partitions, respectively. These are very preliminary results for the parallel version of HARP and significant improvement is expected in the near future. Second, the partitioning time increases less than linearly with the number of partitions for a fixed number of processors. In fact, when 16 processors are used, the partitioning time for 256 partitions is only 20% more than that for 16 partitions. With more and more processors, the partitioning time actually seems to become independent of the number of partitions.

Third, the partitioning time gradually decreases with the number of processors when the ratio of the number of partitions to the number of processors is held constant. This can be observed by scanning diagonally across the entries in Tables 7 and 8. For example, on the SP-2, the time to partition the FORD2 grid into four subgrids on one processor is 0.989 secs but only 0.528 secs for 256 subgrids on 64 processors. Similar results were observed for all the other grids. The relative reduction in the partitioning time with increasing number of processors is more pronounced as the ratio of the number of subgrids to the number of processors increases. This is because when $S > P$, there is no communication after log P iterations. These results and observations demonstrate that HARP will remain a viable partitioner on massively-parallel systems.

6 HARP in the Dynamic Load Balancer JOVE

The primary application of HARP is to dynamically partition adaptive grids at runtime [3]. The motivation for HARP originated from the context of load balancing unstructured adaptive grid computa-

# of processors	MACH95								FORD2							
	2	4	8	16	32	64	128	256	2	4	8	16	32	64	128	256
1	0.298	0.583	0.871	1.166	1.460	1.769	2.089	2.489	0.488	0.989	1.424	1.899	2.377	2.865	3.371	3.901
2	0.250	0.370	0.498	0.625	0.756	0.889	1.036	1.200	0.411	0.609	0.818	1.024	1.234	1.448	1.671	1.912
4	•	0.324	0.381	0.446	0.511	0.577	0.649	0.732	•	0.532	0.627	0.730	0.835	0.940	1.053	1.172
8	•	•	0.337	0.363	0.396	0.429	0.466	0.508	•	•	0.553	0.595	0.648	0.701	0.755	0.815
16	•	•	•	0.332	0.343	0.359	0.377	0.398	•	•	•	0.544	0.559	0.586	0.616	0.644
32	•	•	•	•	0.328	0.328	0.338	0.349	•	•	•	•	0.532	0.535	0.550	0.563
64	•	•	•	•	•	0.322	0.324	0.325	•	•	•	•	•	0.523	0.518	0.528

Table 7: Partitioning times on an IBM SP2. • indicates not applicable.

# of processors	MACH95								FORD2							
	2	4	8	16	32	64	128	256	2	4	8	16	32	64	128	256
1	0.288	0.643	0.997	1.342	1.664	1.975	2.280	2.609	0.477	1.052	1.621	2.188	2.748	3.266	3.761	4.270
2	0.373	0.554	0.733	0.906	1.070	1.227	1.385	1.552	0.614	0.906	1.195	1.484	1.773	2.037	2.292	2.547
4	•	0.498	0.586	0.673	0.753	0.830	0.905	0.988	•	0.818	0.959	1.107	1.250	1.379	1.506	1.631
8	•	•	0.512	0.555	0.596	0.634	0.673	0.713	•	•	0.843	0.913	0.983	1.047	1.107	1.168
16	•	•	•	0.493	0.514	0.533	0.552	0.575	•	•	•	0.817	0.849	0.882	0.913	0.943
32	•	•	•	•	0.474	0.484	0.494	0.505	•	•	•	•	0.780	0.796	0.813	0.827
64	•	•	•	•	•	0.459	0.464	0.469	•	•	•	•	•	0.758	0.766	0.773

Table 8: Partitioning times on a Cray T3E. • indicates not applicable.

tions on distributed-memory machines [23,24]. The dynamic load balancing framework JOVE is described in [23] and its impact on adaptive grid computations are reported in [24]. The framework employs dual-graph representation. CFD flow solvers usually solve for the solution variables at the vertices of the computational mesh. A parallel implementation requires a partitioning of the computational mesh such that each element belongs to a unique partition. Communication is required across faces that are shared by adjacent tetrahedral elements residing on different processors. Hence for the purposes of partitioning, we consider the dual of the original CFD mesh such as MACH95.

The tetrahedral elements of the CFD mesh are the vertices of the dual graph. An edge exists between two dual graph vertices if the corresponding elements share a face in the original mesh. A graph partitioning of the dual graph thus yields an assignment of tetrahedra to processors. Each dual graph vertex has two parameters associated with it. The computational weight, w_{comp}, is a measure of the workload for the corresponding element of the CFD mesh. The communication weight, w_{comm}, measures the cost of moving the element from one processor to another. The connectivity pattern and the w_{comp} determine how dual graph vertices should be grouped to form partitions that minimizes the disparity in the partition weights. The w_{comm} determine how partitions should be assigned to processors such that the cost of data movement is minimized.

The most significant advantage of using a dual graph is that its complexity and connectivity remains *unchanged* during the course of an adaptive computation. This is because the vertices of the dual graph correspond to the elements of the initial CFD mesh. The partitioning and load-balancing times therefore depend only on the initial problem size. New grids obtained by mesh adaption are translated to the two weights, w_{comp} and w_{comm}, for every element in the initial CFD mesh.

To put HARP in the dynamic load balancing perspective, we demonstrate HARP at work using a set of snap shots taken in real world situations. In particular, we use four helicopter meshes derived from MACH95. The initial mesh has 60968 tetrahedral elements and 78343 edges. As the simulation progresses, mesh refinement (coarsening) takes place, resulting in the change in mesh size. Table 9 shows the change in the number of vertices, edges, and elements over three refinements. The initial mesh size and their respective values are listed in the first row.

adaption number	# of elements (weight)	# of edges	16 partitions		256 partitions	
			cuts	time	cuts	time
0	60968	78343	5685	1.024	20204	2.176
1	179355	220077	5229	1.024	18191	2.177
2	389947	469607	4833	1.023	15536	2.177
3	765855	913412	4539	1.021	14039	2.178

Table 9: Runtime behavior of Mach95 over three mesh adaptions.

After the first adaption, the size has grown to 179355 elements and 220077 edges. In each adaption, an element can be refined up to 8 smaller elements. After the three adaptions, the mesh size has grown to 765855 elements, which is an order of magnitude larger than the initial mesh. Runtime load balancing is indispensable when such mesh adaption is implemented on a distributed-memory multiprocessor. It is highly likely that some processors will have a very large number of elements while some perhaps have little change since mesh refinement tends to be localized over time. Table 9 also presents an important feature of HARP in JOVE, where the number of edge cuts decreased from 5685 to 4539 even if the mesh size has grown more than an order of magnitude.

The dual-graph approach employed in the dynamic load balancing framework JOVE allows the mesh size to grow but the complexity of mesh partitioning remain *fixed*. Timing results in Table 9 clearly show that the mesh partitioning times are essentially fixed. Again, the reason is because HARP is applied to the dual mesh which maintains the initial mesh structure but changes the weight of the original elements.

The mesh partitioner HARP as well as the load balancing framework JOVE is currently being applied to rotorcraft fluid dynamics to study of helicopter wake systems. Several plans are currently underway to apply JOVE and HARP, including simulations of deep submicron semiconductor modeling and computational nano-technology at the Numerical Aerospace Simulation of NASA Ames Research Center and NERSC at Lawrence Berkeley Laboratory.

7 Summary

Computational science and engineering problems involve runtime mesh partitioning when implemented on distributed-memory multiprocessors. We have presented in this paper a fast spectral partitioner, called HARP, which can quickly partition realistically-sized meshes while maintaining the partition quality of spectral partitioners such as recursive spectral bisection. To demonstrate the effectiveness of HARP, we have selected various 2D and 3D meshes with the size of up to 100,196 vertices. Both the serial and parallel versions of HARP have been implemented on two distributed-memory platforms, IBM SP-2 and Cray T3E, installed respectively at NASA Ames and NERSC of Lawrence Berkeley Laboratory.

Several types of experiments have been performed to find the effects of the number of eigenvectors on partition quality, the trade-off of the number of eigenvectors with respect to the partition quality and computation time, and the fast partitioning capabilities in the context of dynamically changing mesh adaption. We have identified that the larger meshes tend to show higher partition quality for more partitions due to the fine-grained control on how partitions are generated. The partition quality has improved as the number of eigenvectors increases at the expense of increased computation time. We have also observed that the partition quality improves as the number of partitions increases.

The performance of HARP has been compared against other partitioners such as MeTiS2. Experimental results have indicated that the execution times of HARP are three to four times faster than MeTiS 2.0. The solution quality of HARP, on the other hand, is poorer than MeTiS2. We find that the overall difference is between 30% to 40%. It should be noted that the HARP results are based on 10 eigenvectors. The fact that the partition quality is somewhat poor is not a major concern when dealing with adaptive computations. Since partitioning has to be performed fairly frequently, it is more important to reduce the partitioning time than the number of edge cuts.

The parallel version of HARP has been implemented in Message Passing Interface. It can run on any platform which supports MPI. The sole purpose of the preliminary parallel version is to demonstrate that the serial HARP can be effectively parallelized on distributed-memory machines. The most time-consuming step of the partitioner has been parallelized and its effects have been significant in terms of execution time. The largest mesh among those we used is FORD2 for modeling a Ford car with 100,196 vertices and 222,246 edges. Parallel HARP has shown to partition FORD2 into 256 partitions in 0.5 sec on 64 processors.

The T3E version of HARP has been implemented in MPI. If HARP were implemented in SHMEM with which T3E performs best, the performance of HARP can be further improved. Regardless of the paradigm used for implementation, parallel HARP can further reduce the current partitioning time since less than half the individual modules of HARP are parallelized in the preliminary version. Our immediate plan is to parallelize the sorting step, which is currently the most time consuming step. The MPI version will be converted to a SHMEM version in the near future.

The primary application of HARP is to dynamically partition adaptive grids. In this respect, we have put HARP to work in the dynamic load balancing framework JOVE. Four snap shots of a helicopter blade mesh called MACH95 have been drawn from real-world applications to test the capability of HARP. After three mesh adaptions, the mesh has grown from 60,968 to 765,855 vertices. The mesh partitioning times, on the other hand, have remained constant because of the dual graph approach. We have also found that the number of edge cuts decreased from 5685 to 4539 even if the mesh size has grown more than an order of magnitude. This fixed partitioning times and the decrease in edge cuts have indicated that graph partitioning can now be truly embedded in dynamically-changing real-world applications.

Acknowledgments

Andrew Sohn thanks Youngbae Kim and William Saphir of NERSC for helping to port HARP on T3E while visiting NERSC in January 1997. The full version of this report is available at http://www.cs.njit.edu/sohn/papers. The results of applying HARP to partition SPIRAL, BARTH5, HSCTL, and MACH95 into varying numbers of subdomains are also available at the web site. The partitions are false color coded. These pictures are shown only to give a qualitative flavor of the new partitioner.

References

1. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, SP2 system architecture, in *IBM Systems Journal* Vol. 34, No. 2, 1995.
2. S. T. Barnard and H. D. Simon, Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, *Concurrency: Practice and Experience*, Vol. 6, 1994, pp. 101-117.
3. R. Biswas and R. Strawn, A new procedure for dynamic adaption of three-dimensional unstructured grids, *Applied Numerical Mathematics* 13 (1994) 437-452.
4. T. Chan, J. Gilbert, and S. Teng. Geometric spectral partitioning. Xerox PARC Technical Report, January 1995.
5. W. Chan and A. George, A linear time implementation of the reverse Cuthill-McKee algorithm, *BIT*, Vol. 20, 1980, pp. 8-14.
6. J. De Keyser and D. Roose, Grid partitioning by inertial recursive bisection, Report TW 174, Katholieke Universiteit Leuven, Belgium, 1992.
7. P. Diniz, S. Plimpton, B. Hendrickson and R. Leland, Parallel algorithms for dynamically partitioning unstructured grids, in *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, 1995, pp.615-620.
8. C. Farhat, A simple and efficient automatic FEM domain decomposer, *Computers and Structures* 28, 1988, pp. 579-602.
9. C. Farhat, S. Lanteri, and H. Simon, TOP/DOMDEC: A software tool for mesh partitioning and parallel processing, *Computing Systems in Engineering* 6, February 1995, pp.13 - 26.
10. M. Fiedler, A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory, *Czechoslovak Mathematics Journal* 25, 1975, pp. 619-633.
11. R. Grimes, J. Lewis and H. Simon, A shifted block lanczos algorithm for solving sparse symmetric generalized eigenproblems, *SIAM J. on Matrix Analysis and Applications* 15, 1994, pp.228 - 272.
12. B. Hendrickson and R. Leland, A multilevel algorithm for partitioning graphs, Report SAND93-1301, Sandia National Laboratories, Albuquerque, NM, 1993.
13. B. Hendrickson and R. Leland, Multidimensional spectral load balancing, Report SAND93-0074, Sandia National Laboratories, Albuquerque, NM, 1993.
14. G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, Report 95-035, University of Minnesota, Minneapolis, MN, 1995.
15. B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *The Bell System Technical Journal*, Vol. 49, 1970, pp. 291-308.
16. S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vechhi, Optimization by simulated annealing, *Science*, Vol. 220, 1983, pp. 671-680.
17. S. Khuri and A. Baterekh, Genetic algorithms and discrete optimization, *Methods of Operations Research*, Vol. 64, 1991, pp. 133-142.
18. A. Pothen, H.D. Simon, and K.-P. Liou, Partitioning sparse matrices with eigenvectors of graphs, *SIAM Journal of Matrix Analysis and Applications*, Vol. 11, 1990, pp. 430-452.
19. P. Sadayappan and F. Ercal, Nearest-neighbor mapping of finite element graphs onto processor meshes, *IEEE Transactions on Computers*, Vol. 36, 1987, pp. 1408-1424.
20. S. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1996.
21. H. D. Simon, Visualization of sparse matrix algorithms, *IBM Europe Workshop*, Oberlech, Austria, August 1990.
22. H. D. Simon, Partitioning of unstructured problems for parallel processing, *Computing Systems in Engineering*, Vol. 2, 1991, pp. 135-148.
23. A. Sohn, R. Biswas, and H. Simon, A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors, in *Proc. of the 8th ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, June 1996, pp.189-192.
24. A. Sohn, R. Biswas, and H. Simon, Impact of load balancing on unstructured adaptive computations for distributed-memory multiprocessors, in *Proc. of the 8th IEEE Symposium on Parallel and Distributed Processing*, New Orleans, Louisiana, October 1996, pp.26-33.
25. R. Van Driessche and D. Roose, A graph contraction algorithm for the fast calculation of the Fiedler vector of a graph, in *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, 1995, pp. 621-626.