Chapter 1
A Parallel Implemetation of Multilevel Recursive Spectral Bisection for Application to Adaptive Unstructured Meshes

Stephen T. Barnard* Horst Simon†

Abstract

The design of a parallel implementation of multilevel recursive spectral bisection is described. The goal is to implement a code that is fast enough to enable dynamic repartitioning of adaptive meshes.

1 Background

The Recursive Spectral Bisection (RSB) algorithm is one of a class of recursive bisection methods for partitioning unstructured problems [4]. RSB is typically used as a preprocessing step prior to running a unstructured-mesh simulation on a massively parallel computer. Applications that change the mesh adaptively throughout the simulation, however, require a fast method to repartition "on the fly."

Finding a partition that both balances the work of all processors and minimizes interprocessor communication is an NP-hard problem. Therefore, all practical partitioning algorithms are necessarily heuristic approximations. Among these, RSB empirically provides the best partitions for a large set of problems, albeit at somewhat more run time than most other methods. RSB bisects a graph by first finding the eigenvector (the *Fiedler vector*) corresponding to the smallest non-trivial eigenvalue of the Laplacian matrix of a graph. The graph could represent, for example, the connectivity between elements in a finite-element mesh, or the connectivity between volumes in an unstructured finite-volume mesh. The vertices of the graph are reordered with the permutation induced by sorting the components of the Fiedler vector, and the graph is cut in half, with those vertices in the lower half of the new ordering in one part, and the remaining vertices in the other part.

The most straightforward implementation of RSB, which uses the Lanczos algorithm to find the Fiedler vectors, is unacceptably slow for many applications. Multilevel recursive spectral bisection (MRSB) [1] is a refinement of the algorithm that is much faster, typically by an order of magnitude or more, and has been instrumental in the acceptance of RSB. The basic idea behind MRSB is to speed up the Fiedler-vector computation by constructing a series of successively smaller contracted graphs that maintain the global structure of the original graph. The Fiedler vector of the smallest graph is found quickly with the Lanczos algorithm. That result is interpolated to the next larger graph to form an approximate Fiedler vector, which is then refined with Rayleigh Quotient Iteration using the SYMMLQ algorithm. This process is repeated until the Fiedler vector of the original graph is obtained.

*Cray Research Inc., NASA Ames Research Center, Moffett Field, CA 94035
†Computer Sciences Corp., NASA Ames Research Center, Moffett Field, CA 94035 (simon@nas.nasa.gov)

```
/* Partition matrix M into n parts.  Assume n is a power of 2. */
void partition(struct matrix *M, int n)
{ struct matrix *M0,*M1;
  if (n > 1) {
    bisect(M);                  /* set mask to {0,1} */
    M0 = copy_submatrix(M,0);   /* partition M */
    M1 = copy_submatrix(M,1);
    partition(M0,n/2);          /* partition each submatrix */
    partition(M1,n/2);
    remap(M0,M1,n/2, M);        /* remap processor assignments */
    free_matrix(M0);            /* free the submatrices */
    free_matrix(M1);
  }}
```

FIG. 1. partition.

2 A Parallel Design for MRSB

The MRSB algorithm is naturally recursive in two ways. First of all it is a recursive bisection method; and secondly, at a lower level in the control structure, it uses a recursive multilevel technique to find Fiedler vectors, and hence to determine each bisection. These two quite different uses of recursion present different challenges (and opportunities) for parallel implementation.

2.1 Recursive Bisection

Recursive-bisection partitioning algorithms have the structure illustrated by partition, the C code in Figure 1. The partition routine takes as its input a sparse matrix M and assigns rows of M to n processors. The bisect routine simply assigns rows to elements of the set $\{0, 1\}$, while the remap routine promotes processor assignments for the submatrices to processor assignments for the parent matrix. As it stands, partition works for any bisection algorithm.

An efficient way to implement partition on a distributed-memory MIMD parallel computer is illustrated in Figure 2. Initially, a "task-team" of processors is formed, each of which holds some number of rows of the matrix. All processors participate in determining the first bisection, then they split into two smaller task-teams. These task teams are responsible for partitioning the two submatrices determined by the first bisection after the appropriate data has been copied. Recursive splitting proceeds down to task teams of single processors. Task teams are asynchronous: after a task team is split its two children can execute independently because all barriers and other global operations are restricted to a particular task team.

2.2 Recursive Multilevel Fiedler-Vector Computation

MRSB bisects by sorting the Fiedler vector, which is computed recursively as illustrated by the C code in Figure 3. If the matrix is small the Fiedler vector is found with the Lanczos algorithm; otherwise, the matrix is contracted, the Fiedler vector of the smaller matrix is found recursively, the result is interpolated to the original matrix and improved with the Rayleigh Quotient Iteration algorithm. Unlike partition, the recursive computation of fiedler occurs over a single task team, as shown in Figure 4.

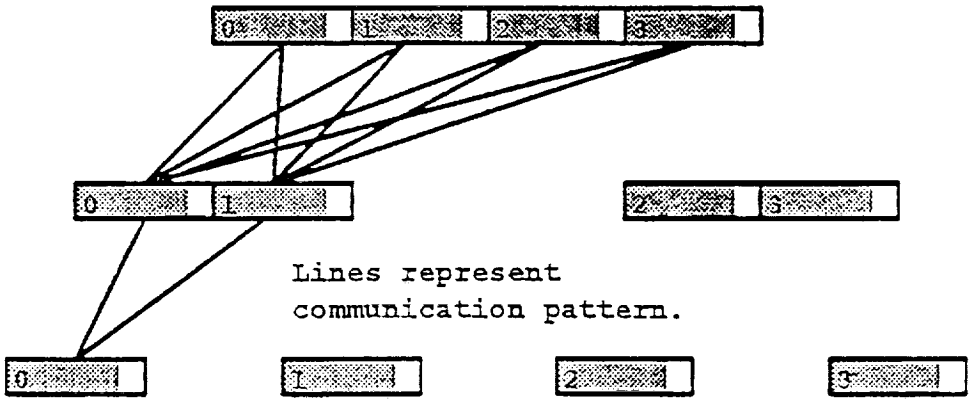

FIG. 2. *MIMD Recursive Bisection with Asynchronous Task Teams*

```
/* Find the Fiedler vector of the sparse Laplacian matrix M */
void fiedler(struct matrix *M)
{ struct matrix *M_c;
  if (M->neqns <= min_neqns) {  /* M is small enough */
    lanczos(M);                    /* use Lanczos */
  else                           /* M is too big */
      M_c = contract(M);           /* contract matrix M */
      fiedler(M_c);                /* find the Fiedler vector */
      interpolate(M_c,M);          /* interpolate to M */
      rqi(M);                    /* refine estimate with RQI */
    }
}
```

FIG. 3. *Find Fiedler Vector.*

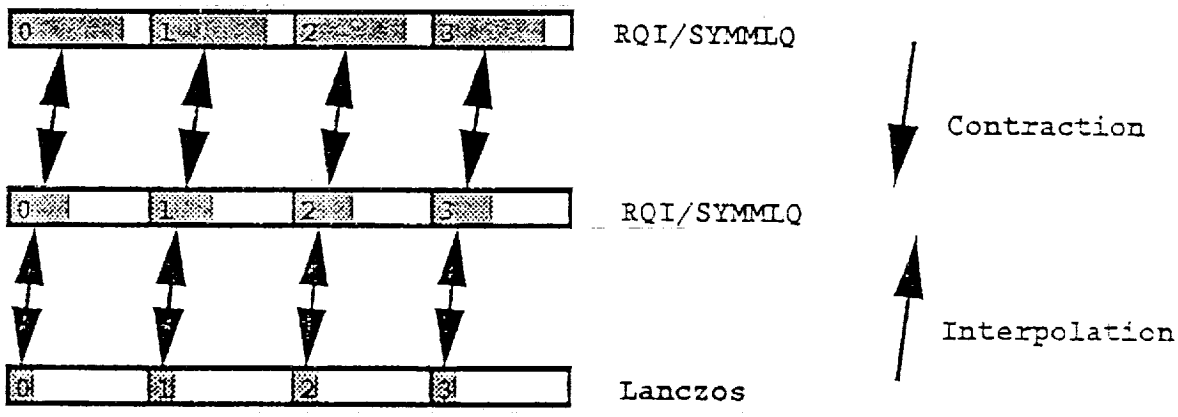

FIG. 4. *Recursive Multilevel Fiedler Computation*

We now examine the feasibility of implementing a parallel version of each step of this procedure:

- The contraction step builds a smaller graph by selecting a maximal independent set of vertices and then connecting them with a breadth-first search through the original graph. (Two vertices in the maximal independent set are connected in the contracted graph if and only if there is a shortest path in the original graph that does not contain a vertex in the maximal independent set.)
  - Luby [2] has shown how to find a maximal independent set efficiently in parallel. (Finding a maximal independent set was for a time thought to be difficult to do in parallel, although the "greedy" serial algorithm is trivial.)
  - The breadth-first search (to connect members of the maximal independent set) can be implemented with independent processes for every vertex, simply using a low-level "add-half-edge" routine that adds a single off-diagonal entry to the Laplacian matrix. The basic idea is to grow neighborhoods from each member of the maximal independent set, adding "half edges" when neighborhoods intersect, and terminating when all vertices have been expanded.
- The interpolation step (a simple prolongation operator) is easy, especially on a shared memory system.
- RQI (Rayleigh Quotient Iteration) is easy to implement in parallel because it only depends on BLAS1 and BLAS2 operations.
- The Lanczos algorithm is more difficult, though not impossible, to parallelize, but this step requires only a very small amount of work because the final contracted graph is tiny. Mapping the Lanczos computation to one processor is a workable (if ugly) solution. Another possibility we intend to investigate is to use Davidson's method, which is easily parallelized.

2.3 Other implementation issues.

The new C implementation of MRSB takes an object-oriented approach. The code is greatly simplified by using a C structure to encapsulate the data structures necessary for representing a graph and its associated Laplacian matrix. The client supplies the initial graph and optionally an array of vertex weights, then MRSB finds a mapping from vertices to processors, using a number of internal buffers not of interest to the client. The adjacency structures are represented as linked lists, which provide an efficient and flexible data structure for managing the unbounded and somewhat unpredictable requirements of the contraction operation.

There are two additional operations that must be parallelized that we haven't discussed: sorting and finding connected components. Several parallel algorithms are possible. In any case, these operations require such a small proportion of the total work that efficiency is not a serious concern.

2.4 Shared Memory vs. Message Passing Implementations

Unlike structured-grid codes, unstructured MPP applications must support highly irregular patterns of communication. Every processor holds pointers not only to local data, but also to data on a variable number of neighboring processors. The essential problem is how to dereference these remote pointers efficiently, with no redundant communication. On a shared-memory system such as the T3D this is scarcely more difficult than dereferencing pointers to local data: if a reference is remote the data is merely read from the remote memory, with no need for the processor that "owns" the data to do anything. A message-passing architecture, however, requires an elaborate preprocessing step to set up data structures that can be used for remote scatter/gather operations.

Sparse matrix-vector multiplication illustrates this problem. A reasonably efficient shared-memory version is only slightly more complex than a serial version. The only way to write a sparse matrix-vector multiply of roughly the same code complexity on a message-passing system, however, requires global reductions of complexity $\mathcal{O}(n \log m)$, where n is the number of rows and m is the number of processors. This leads to poor scaling as shown in Figure 5. To be sure, this is not the most efficient way to do matrix-vector multiplication with message-passing: the point is that the only *simple* way is poor.

We have implemented a "logarithmically parallel" version of MRSB. Suppose we have n processors, where n is a power of two and the processors are numbered $\{0, 1, ...n-1\}$. First copy the graph to all processors. The first bisection is computed by processor 0 using the serial multilevel code. The results of this bisection are sent to processor $n/2$, and then processor 0 and processor $n/2$ compute the bisections of the two new subgraphs. This is repeated recursively until the graph is fully partitioned.

This simple method is inefficient because the the computation of the first bisection involves only one processor, the next bisection involves only two, and so on. Nevertheless, the results are encouraging. The table shows the run times in seconds to partition a graph with 15606 vertices and 45878 edges into 4 parts (npart) and 128 parts. The "Multilevel Serial" times were obtained on a Silicon Graphics Indigo 2 workstation, the "Multilevel T3D" times were obtained on a Cray T3D using the logarithmically parallel method described above, and the "RSB CM5" time was obtained on a 128-vector-unit CM5 using the CMSSL routine (parallel single-level RSB).

The multilevel algorithm on a single workstation outperforms the 128-processor CM5, demonstrating the effectiveness of the multilevel approach. The processors of the T3D

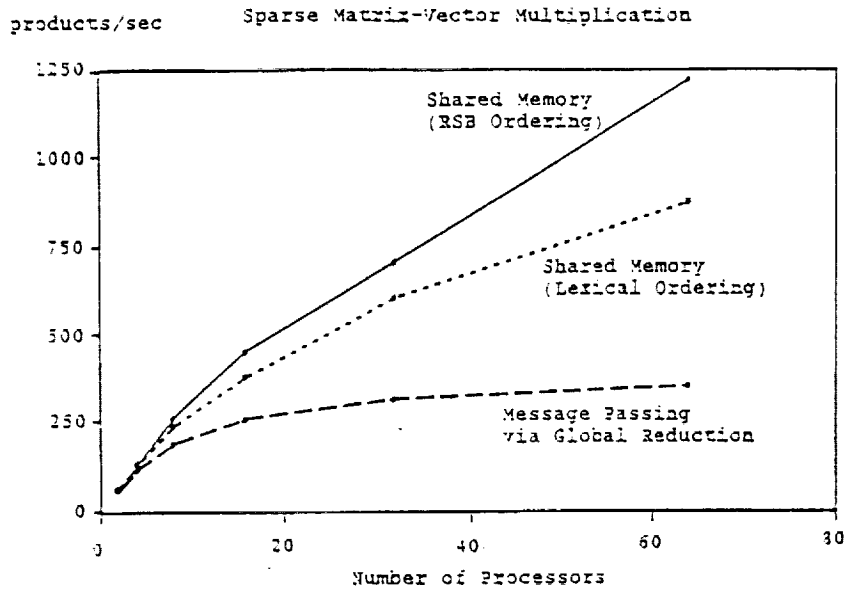

FIG. 5. *Performance of Matrix-Vector Multiplication*

TABLE 1
RSB Partitioning Run Times

npart	Multilevel Serial	Multilevel T3D	RSB CM5
4	4.2	3.8 (4 processors)	
128	14.4	5.0 (128 processors)	18.3

have approximately the same performance as the Indigo 2 processor (150 MHz), so the 4-processor T3D is only slightly faster. Computing higher order partitionings on the T3D is nearly free, however, because the parallelism rapidly becomes more effective as the number of partitions increases.

3 Conclusions

An efficient parallel implementation of MRSB to support repartitioning adaptive meshes is feasible. Recursive bisection in general can be implemented with recursive asynchronous task teams. Coding the recursive multilevel technique for finding Fiedler vectors, including the difficult contraction operation, is facilitated by the shared-memory architecture of the T3D. Preliminary testing suggests a substantial improvement in speed over currently available serial and parallel spectral partitioners.

References

[1] S. Barnard and H. Simon, *Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience, Vol. 6, No. 2, 101-117 (1994).

[2] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., Vol. 15, No. 4, November 1986.

[3] H. Simon, *Partitioning unstructured problems for parallel processing*, Comput. Syst. Eng., 2(2/3), 135-148 (1991).