# Sparse Matrix-Vector Multiplication with the Message Passing Interface

18:337, Lab 5

April 9, 1996

In this lab you will learn how to write message-passing parallel programs using MPI. The program will test the effect of partitioning an unstructured mesh among processors. You are responsible for writing the parallel sparse matrix-vector multiplication subroutine. Your subroutine will be used in two applications:

- **Iterative Sparse Linear Solver**: The conjugate gradient method is one of the most efficient iterative solvers for symmetric positive definite linear systems. The main kernel of a conjugate gradient solver is the computation of the product of a sparse matrix and a vector. Your parallel subroutine will be used in our conjugate gradient solver. You will perform some experiments to evaluate the efficiency of the partition that we have generated. In this application, you will get to try simple methods for sparse matrix iterative methods, and more complex but more efficient methods.
- **Spectral Partitioning**: As an interesting application of sparse matrix-vector multiplication, you will apply your program in a simple power method that approximate the second smallest eigenvalue/vector of the Laplacian graph of the input unstructured mesh. You then can find a bisection of the mesh using the spectral partitioning technique.

  **Remark:** Practical spectral partitioning implementations normally do not use the power method to approximate small eigenvectors. A more efficient procedure uses Lanczos's method, which converges much faster. In fact, the core computation in Lanczos's method is also sparse matrix-vector multiplication. Thus, your program can also be used as part of a parallel Lanczos implementation. Recently, several more efficient methods have been proposed. They use the multi-level approach to speed-up the approximation.

## 1 Getting Started.

First, please follow these steps to login to the IBM SP2 and set up the necessary environment for doing MPI programming:

1. First, login to your athena account.

2. Login to the SP2 at MIT, using the telnet command as follows:

   - telnet sp2n9.mit.edu,
   - telnet sp2n10.mit.edu,
   - telnet sp2n11.mit.edu, or
   - telnet sp2n12.mit.edu

3. Once you are logged in to the SP2, copy all the files from `~edelman/lab5` into your directory by typing

   ```
   cp ~edelman/lab5/* .
   ```

   (the period is part of the command).

   The files you will get include some source code that you will modify, an executable, a makefile, and some data files.

4. The code in `solver_incomplete.c` is an iterative solver (partially completed) for a medium sized sparse matrix problem (4253 equations and 12289 nonzeros). First, run the executable that we've made from a complete version of the code. To run this, type

   ```
   poe solver
   ```

   If there is some complaint about "too many processors," type `setenv MP_PROCS 4` and try again.

   As you see, the code reports on the progress of the iterations, and the time to solve the problem.

## 1.1 About the iterative solver.

The program `solver_incomplete.c` is a message passing program that implements a simple iterative solver for sparse systems of linear equations. One part of the solver is missing, and you will have to fill it in. (It's a C program. If you don't know C, this is would be a good time to learn! Team up with another student who does.)

The program implements an iterative conjugate gradient solver. The conjugate gradient algorithm finds the solution $x$ of a linear system $Ax = b$ using three types of operations in every iteration: multiplying a vector $p$ by the matrix $A$, computing dot products of two vectors, and computing linear combinations of vectors.

The matrix $A$ that we use is the Laplacian matrix of a certain undirected graph plus an identity matrix. For every edge $(i, j)$ in the graph, the corresponding element $A_{ij} = -1$. The other off-diagonal elements of $A$ are all zero. Since the graph is undirected, $A$ is symmetric. Each diagonal element is equal to 1 plus the number of nonzero elements in that row. The program reads the matrix from a file that contains the number of edges of the graph and then the list of $(i, j)$ pairs defining the edges, which determines the locations of the nonzeros of $A$.

To test the conjugate-gradient solver, the program generates a right hand side $b$, and uses the zero vector for the initial guess $x$ in the iterative algorithm.

The approach to parallelism here is to partition the graph and map the subsets of vertices to the processors. The elements of all vectors and the rows of $A$ that correspond to a vertex subset are mapped to the same processor as that subset; so, every processor stores a subset of the elements of each vector and the corresponding rows of $A$. A good mapping of the graph of the matrix onto 16 processors, computed using Matlab's Geometric Toolbox, has been stored in the file `airfoil_map.p16`. The program reads in this mapping, and modifies it for the number of processors actually being used. The solver also has the ability to use a random mapping of the graph. You can specify the partition by giving the program a run time argument `PARTITION_FROM_FILE` (the default) or `PARTITION_RANDOM`. For example

```
poe solver PARTITION_RANDOM
```

# 2 Matrix-Vector Multiplication with Allgatherv

Since ours is a Laplacian matrix, when a processor computes its elements of $Ax$, it must multiply each of its vector elements $x_i$ by the corresponding diagonal element $A_{i,i}$ and then subtract $x_j$ for each of the neighboring vertices $j$, that is for each $j$ such that $A_{i,j} = -1$. Some of these neighboring $x_j$ values may belong to other processors.

The program is missing the code for matrix-vector multiplication. You should fill in the code for the subroutine multiply_by_A_allgatherv which multiplies a given vector $x$ by the matrix $A$. Computing the elements of $Ax$ that belong to a processor requires some elements of $x$ that belong to other processors. You should implement the subroutine by collecting all of the vector $x$ on all the processors using the MPI subroutine MPI_Allgatherv.

Here is what you need to know about the data structures that the program uses. See Figure 1 for an illustration.

- The global ints *nprocs* and *my_rank* are the number of processors and the processor number of the processor the code is running on. Look for the calls to MPI_Comm_size and MPI_Comm_rank that compute these values.
- The number of vector elements in each processor is stored in *my_number_of_vertices*.
- The diagonal elements that correspond to the vector section owned by the processor are stored in an array of doubles called *D*.
- The rows of $A$ are stored in an array, called *my_vertices*, of *Vertex*structures.
- Each *Vertex* structure contains a *VertexId* structure that specifies the global label (name) of that row, the processor that owns it, and the index of that row in *my_vertices*. The row structure also contains the number of nonzero elements in the row (neighbors in the underlying graph), and an array of neighbors, each of which is a *VertexId* structure.

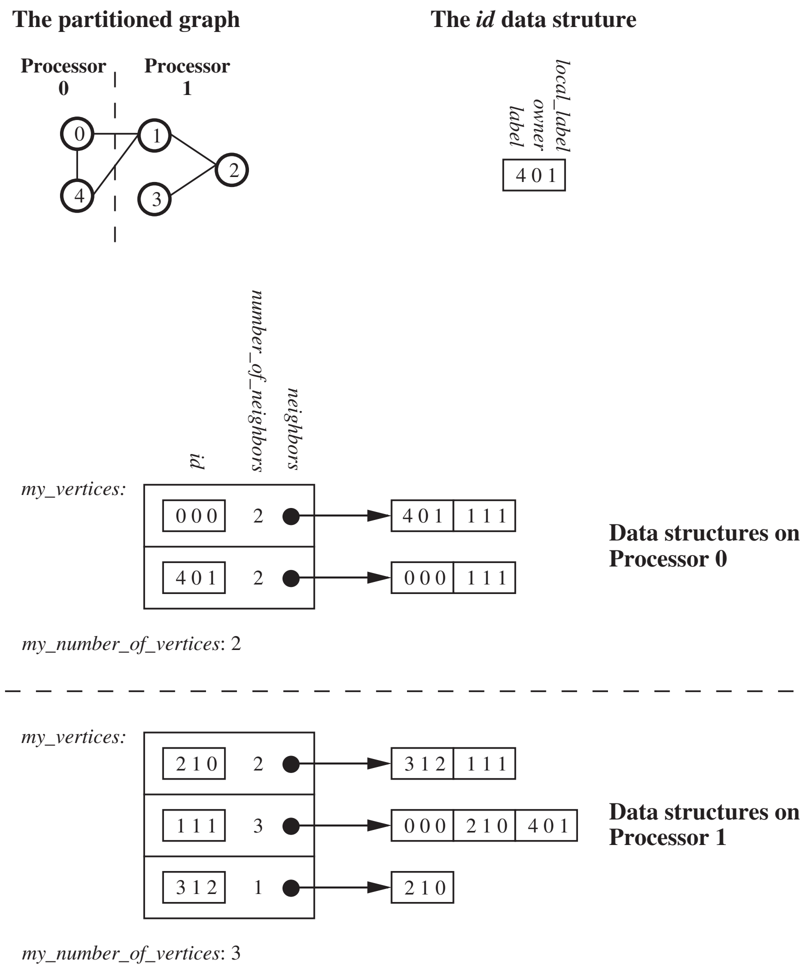


Figure 1: Local data structures for a distributed sparse graph.

We would like to offer two hints. First, it is likely that you will need to perform some setup operations before the subroutine is first called. For example, you might need to broadcast to every processor the number of rows stored on every other processor. (Initially that information is stored only on the processor whose rank is zero in a vector called *vertices_per_owner*.) Second, remember that if, after you collect them, the elements of $x$ are not in increasing global index order, then you need to do something to find specific neighbors.

You will need to use the following MPI subroutines:

- MPI_Allgatherv. The prototype for the subroutine is

  int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void* recvbuf, int *recvcounts, int *displs,
  MPI_Datatype recvtype, MPI_Comm comm).

  This subroutine collects a section of a vector from every processor and broadcasts the entire vector to all the processors. Each processor sends a section which is stored at address *sendbuf* and contains an array of *sendcount* elements of datatype *sendtype*. Valid datatypes include MPI_INT, MPI_DOUBLE, and so on. A useful datatype is MPI_BYTE, which can be used with an appropriate value of *count* to send structures. The entire vector is stored in address *recvbuf*. An important aspect of this subroutine is that every processor must know how many elements each other processor contributes, and where in the vector each section belongs. The arrays *recvcounts* and *displs*, which must be initialized and identical on all the processors, must contain the number of elements each processor contributes, and the displacement in recvbuf that marks the beginning of each section. These arrays are of size *nprocs*. *Comm* specifies a communicator, which in this program should be set to MPI_COMM_WORLD.

- MPI_Bcast. This subroutine sends information from one processors to all the other processors. The prototype for the subroutine is

  int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
  MPI_Comm comm).

  The subroutine sends the buffer pointed to by *buf* in processor *root* to all the other processors. The buffer is assumed to contain an array of *count* objects of type *datatype*. *Comm* should be set to MPI_COMM_WORLD.

Both MPI_Bcast and MPI_Allgathrv are collective communication routines, which means that nothing happens until *all* the processors reach and execute the call to the routine!

You can compile the program by typing `make`. You can run the program by typing `poe solver`, and you can pass runtime arguments to the program by passing them to `poe`. For example, `poe solver PARTITION_RANDOM` will run the program with a random data partitioning.

Does your program speed up as the number of processors is increased? The program runs by default on 4 processors. To change this to $n$, type

```
procs n
```

before the `poe` command. You can use 1, 2, 4, or 8 processors.

# 3 Matrix-Vector Multiplication with Point-to-Point Communication

A more sophisticated way to implement the matrix-vector multiplication subroutine is using the point-to-point MPI communication subroutines MPI_Send and MPI_Recv. This version of the algorithm should be implemented in the subroutine multiply_by_A_sendrecv. This is a more sophisticated algorithm, but not necessarily more complicated. Do it first without trying for message vectorization, eliminating redundant communication, or reindexing. Just send a one item message for each edge that connects to a neighbor on another processor. In a scheme like this, a processor will do a receive for every send, because the graph is undirected.

To use the new matrix-vector multiplication subroutine, you need to specify the argument MATVEC_SENDRECV to `poe solver`.

How does this version compare in performance to the allgatherv version?

You will need to use the following MPI subroutines:

- MPI_Send. The prototype for the subroutine is

  ```
  int MPI_Send(void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm).
  ```

  The subroutine sends the buffer pointed to by *buf* to processor *dest*. The buffer is assumed to contain an array of *count* objects of type *datatype*. *Tag* is used by the receiving processor, which can choose to receive messages with specific tags. *Comm* should be set to MPI_COMM_WORLD.

- MPI_Recv. The prototype for the subroutine is

  ```
  int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,MPI_Status *status).
  ```

  The subroutine receives one message into the buffer pointed to by *buf*. The message must be from processor *source*, unless *source* is MPI_ANY_SOURCE, in which case a message from any processor can be received. The message must carry the specified *tag*, unless *tag* is MPI_ANY_TAG, in which case a message with any tag can be received. The meaning of *count* and *datatype* is identical to their meaning in MPI_Send. *Comm* should be set to MPI_COMM_WORLD. *Status* contains two fields, MPI_SOURCE and MPI_TAG that store the source and tag of the message. This may be useful if MPI_ANY_SOURCE or MPI_ANY_TAG are specified. You may retrieve them by using the contents of *status.MPI_SOURCE* and *status.MPI_TAG*. (Our implementation does not use the return status.)

- MPI_Iprobe. This subroutine is not absolutely necessary, but you may find it useful. It checks whether a message is available. If you call MPI_Recv and no appropriate message is available, your program will block until one arrives (on the processor that issued the call). The prototype for the subroutine is

int MPI_Iprobe(int source, int tag, MPI_Comm comm,int *flag,
MPI_Status *status).

The subroutine sets *flag* to TRUE if a message with appropriate tag and source is available. If it is, then *status* has the same information about the waiting message as it would if that message were received.

# 4 Performance Studies of Parallel CG

Compare the performance of your two matrix-vector multiplication algorithms. Use both the data partition from the file and the random data partition. Which combination is the fastest? How are they affected by increasing or decreasing the number of processors? Is it worthwhile to partition the vectors so as to maximize data locality? Which version is best on random data partitions? Can you explain this?

The executable `solver` has the same options as your code. It contains one version of the matrix-vector multiplication code that pulls out all the stops: message vectorization, redundant communication removal, and local reindexing.

To use this version, type

```
poe solver MATVEC_SENDRECV_VECTORIZE_VERTICES
```

You can look at the code in `edelman/teng/solver.c`. How fast is it?

Note that the number of floating point operations per iteration of our algorithm is $12n+2m$, where $n = 4,253$ is the number of equations and $m = 12,289$ is the number of nonzeros in $A$. How fast is this code running, in megaflops?

Is this problem big enough for the SP2? Does the code run efficiently on one processor? (The peak speed of an SP2 processor is over 200 Mflops).

# 5 Spectral Partitioning

To use the spectral partitioning approach, we need to compute an "approximation" of the eigenvector of the Laplacian matrix of the graph that has the smallest non-zero eigenvalue $\lambda$. Recall the Laplacian matrix $L$ of an undirected graph is an $n$ by $n$ matrix such that for every edge $(i,j)$ in the graph, the corresponding element $L_{ij} = -1$. The other off-diagonal elements of $A$ are all zero. The diagonal entries of $L$ are the degrees of the vertices of the graph.

If $u = (u_1, ..., u_n)$ is the eigenvector of $L$ whose eigenvalue is equal to $\lambda$, then $\sum_i u_i = 0$ and $Lu = \lambda u$. One way to approximate $u$ is to use the power method. Let $B = 2*\Delta*I - L$, where $\Delta$ is the largest degree of the graph. Then, $Bu = (2\Delta - \lambda)u$. So $u$ is still an eigenvector of $B$. In fact the largest eigenvalue of $B$ is $2\Delta$ and $\lambda$ is the second largest eigenvalue of $B$.

Suppose $v = (v_1, ..., v_n)$ is a vector that is perpendicular to the all-1's vector, i.e., $\sum_i v_i = 0$. Then $Bv$ is also perpendicular to the all-1's vector. Hence, we can approximate $u$ by compute $B^k v$, for a randomly chosen or a carefully chosen vector $v$. We know that when $k$ goes to infinity, then $B^k v$ is becoming parallel to $u$, and hence we get an approximation.

In our assignment, we will use this power method to compute an approximation to $u$. Clearly, we will use your matrix-vector multiplication subroution.

When a processor computes its elements of $Bx$, it must multiply each of its vector elements $x_i$ by the corresponding diagonal element $B_{i,i}$ and then add $x_j$ for each of the neighboring vertices $j$, that is for each $j$ such that $L_{i,j} = -1$. Some of these neighboring $x_j$ values may belong to other processors.

Like the CG part of the assignment, we provide several initialization functions which assign values to $D$, the diagonal entries, $x$, the initial vector. The final program will save the approximate eigenvector in a file called "fiedler.airfoil1". You can then start your matlab and type "plot_spect;" to see the plot of your spectral bisection.

The file here is `spectral_incomplete.c`.

# 6 Makefile

Please change the file names in the Makefile to be consistent with your program.