Programming Assignment 3:
Sparse Matrix-Vector Multiplication

Parallel and Distributed Computing

Due: Wednesday, March 23

1 Sparse Matrices

We saw in the last programming assignment that the key to a fast implementation of conjugate gradients is a fast implementation of matrix-vector multiplication. However, thus far we’ve only talked about dense matrix-vector multiplication, and, in practice, conjugate gradients is almost always used when the matrices are sparse. That is, the number of nonzero elements in the matrix is much less than the number of elements in the matrix. For example, if the matrix has order 10,000 but, on average only 10 nonzeros in each row, then we’ll use 100,000,000 doubles or 800 megabytes to store the matrix in the usual dense format. However, of the 100,000,000 doubles only 100,000 are nonzero. That is, we’ll really only be using 1/1000 of the storage allocated — i.e., 800 kilobytes of storage.

There are many solutions to the problem of wasted storage for sparse matrices. In many cases sparse matrices have some special structure, that makes it quite simple to store them efficiently. For example, tridiagonal matrices and block tridiagonal matrices arise naturally in the solution of certain types of PDE’s, and these types of matrices can be stored with little information beyond a listing of the nonzero elements. On the other hand, in many other applications (e.g., circuit simulation) the sparse matrices have no special structure. These are the types of matrices we’ll be interested in for programming assignment 3.

2 Compressed Sparse Row Format

An obvious data structure for the representation of unstructured sparse matrices is the storage of each nonzero along with both its row and column subscript. While easy to understand, this approach will store 2 ints for each floating point value, and hence double the amount of storage needed for doubles and triple the amount of storage needed for floats. A more economical alternative is called compressed sparse row format or CSR format. The idea is to store three arrays.

1. An array of floating point numbers for the nonzeros.

2. An array of ints: the column subscripts of the corresponding entries in the array of floating point values.
3. Another array of ints, which stores the subscript (in the array of floating point numbers) of the first entry in each row.

For example, suppose the matrix is

\[
\begin{pmatrix}
1 & 0 & 0 & 2 & 0 & 3 \\
4 & 5 & 0 & 0 & 0 & 0 \\
0 & 6 & 7 & 0 & 0 & 8 \\
9 & 0 & 0 & 10 & 11 & 12 \\
0 & 13 & 0 & 0 & 14 & 0 \\
0 & 0 & 0 & 0 & 0 & 15
\end{pmatrix}
\]

If the three arrays are called \textit{vals}, \textit{cols}, and \textit{rows}, respectively. Then their contents might be

\begin{align*}
\text{vals:} & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \\
\text{cols:} & \quad 0 \quad 3 \quad 5 \quad 0 \quad 1 \quad 1 \quad 2 \quad 5 \quad 0 \quad 3 \quad 4 \quad 5 \quad 1 \quad 4 \quad 5 \\
\text{rows:} & \quad 0 \quad 3 \quad 5 \quad 8 \quad 12 \quad 14 \quad 15
\end{align*}

The contents of the \(i\)th row is stored in the \textit{vals} array in locations \(\text{rows}[i], \text{rows}[i]+1, \text{rows}[i]+2, \ldots, \text{rows}[i+1]-1\)

If there are \(n\) rows, then in order for this formula to work for the last row, row \(n - 1\), we need to store the number of nonzeros in the last element of the \textit{rows} array.

Note that in our example, the elements within a row are stored in increasing order of column subscript. This isn’t required by the CSR format, but you can assume that any matrix your program works with will have this additional property. (This makes operations like converting from CSR to dense formats much easier.)

In our earlier example — order 10,000 with an average of 10 nonzeros per row — CSR format would store 100,000 doubles and 110,000 ints or 1,240,000 bytes — less than 0.2% of the storage required by the dense format.

### 3 Matrix-Vector Multiplication

In this assignment you’ll be implementing sparse matrix-vector multiplication when the matrix is stored using the CSR format and the vectors are stored in the usual (dense) format. So suppose \(A\) is stored in CSR format using the arrays \textit{vals}, \textit{cols}, and \textit{rows}, as above. Then the following algorithm can be used to multiply \(Ax\) and store the result in \(y\):

\begin{verbatim}
int i, j, k;

for (i = 0; i < n; i++) {
    y[i] = 0;
    for (k = rows[i]; k < rows[i+1]; k++) {
        j = cols[k];
        y[i] += vals[k]*x[j];
    }
}
\end{verbatim}
In order to understand this algorithm observe that in iteration $i$, we’re forming the dot product of the $i$th row of $A$ with $x$. Since entries of $A$ other than those stored in $\text{vals}$ are zero, we don’t need to worry about any components of $x$ other than those corresponding to nonzeros in row $i$. Furthermore, since the nonzero elements in the $i$th row of $A$ are stored in consecutive entries of $\text{vals}$ running from $\text{row}[i]$ to $\text{row}[i+1]$, we can “dot” the $i$th row with $x$ by simply multiplying these elements of $\text{vals}$ by the corresponding elements of $x$.

Note that this algorithm assumes that the number of nonzeros is stored in $\text{rows}[n]$.

Also note that if we’re solving a nonsingular linear system (e.g., using CG), there will be no row of zeroes. So even though the algorithm is fine without this assumption, you can assume there is no row of zeroes in the matrices in this assignment.

4 Parallelization

How can we parallelize this? Since the matrix is stored by rows, most programs partition the matrix by rows. If we’re using Pthreads, an obvious approach is to assign one row to each thread. This algorithm has no critical sections. That is, the threads are completely independent of each other, and the algorithm affords as much parallelism as the system can provide. From a practical point of view, though, the cost of starting thousands of threads on most systems will be quite high. So this approach is usually impractical.

An obvious alternative is to use one of the familiar schemes: a block, cyclic, or block-cyclic partition of the rows. A possible drawback to this approach is that the number of nonzeros in each row may vary dramatically, and it could happen that one process/thread was given much more work than another.

An alternative that might seem more promising would be to assign, as nearly as possible, the same number of nonzeros to each process/thread. A drawback to this is that finding the optimal assignment is NP-complete. However, there are some simple heuristics that do reasonably well. Here’s one.

1. Sort the rows in decreasing order of number of nonzeros.

2. Assign the rows to the processes/threads using an iterative scheme:
   (a) Assign the first row to process/thread 0.
   (b) Using the sorted order assign subsequent rows to the process/thread with the smallest total number of nonzeros.

This guarantees that the maximum difference in the number of nonzeros between any two processes/threads will be at most the largest number of nonzeros in a row.

A third scheme assigns an exponentially decreasing number of rows to the threads. For example, we might initially assign $r = k/p$ rows to each thread where $k \leq n/2$. If we keep a marker variable $v$ indicating the first available row, then threads can help themselves to new rows when they run out of rows. The number of rows a thread will take will initially be $r/2$, then $r/4$. When $r/2^s \leq 1$, the method will set $r = 1$, and the thread will continue taking 1 row until all the rows are exhausted.
For example, suppose $n = 1000$, $k = 500$, and $p = 5$. Then each thread will initially get $r = k/p = 100$ rows. When a thread finishes multiplying its initial 100 rows, it will get $r/2 = 50$ rows, then 25, then 12, then 6, then 3, then 1, and it will continue getting 1 until all the rows have been multiplied.

An obvious drawback to the second and third schemes is that there will be overhead in assigning the rows. However, if the assignment is saved for use in an iterative solver, the overhead can be amortized over many iterations.

A second complication is that if you’re in a distributed memory environment (e.g., MPI), then the Allgatherv of $x$ may be more efficient with a fairly complex communication scheme that involves derived datatypes, since it may make sense to only gather the elements of $x$ corresponding to nonzeros in a process’s assigned rows. However, for assignment 3, you’ll be using Pthreads, so this won’t be an issue.

5 Details

For assignment 3, you’ll be implementing the three schemes just outlined, and evaluating them. You only need to parallelize the partitioning scheme and the matrix-vector multiplication. I/O and setup can be serial. You can get extra credit by devising additional (effective) schemes or by making the above schemes run fast. You can also get extra credit by implementing one or more of the schemes in MPI and/or OpenMP.

Input to the program will come from the command line and (possibly) two files. The command line will have the form

```
$ csr <thread_count> <matrix_order> <method> 
    [matrix_filename] [vector_filename] [n]
```

The arguments in angle brackets (<...>) are required. The arguments in square brackets ([...]) are optional.

The method argument will be ‘1’, ‘2’, or ‘3’. This indicates which method should be used for parallelization. If you implement additional methods, be sure to document how to use them.

If the optional arguments `matrix_filename` and `vector_filename` are present, the input should come from these files. The matrix file will have the form

```
Matrix order  
Number of nonzeroes  
List of nonzero entries  
List of corresponding column subscripts  
List of initial row subscripts
```

Elements in the lists will be separated by white space. Note that the three lists are just the compressed sparse row format. The vector file will just be a list of all the entries (including zeroes) in the vector $x$. All floating point values will be doubles.

If the optional arguments `matrix_filename` and `vector_filename` are not present, the program should generate a random matrix and vector. Note that you should not use the C
library function `rand`. It does a very poor job of generating random numbers. The `random` function is much better.

You may want to allow additional *optional* command line arguments that specify such information as the total number of nonzeroes, the standard deviation of the number of nonzeroes in each row, the blocksize in method 1 and the parameter $k$ in method 3. (The default for these values should be $n/p$ in method 1 and $n/2$ in method 3.) Be sure to document any optional arguments.

Also note that either both filenames should be present or neither should be present.

The output of the program should include the runtime of the matrix-vector multiplication together with any overhead associated with the parallelization. If the final command line argument is not present, the product vector $y = Ax$ should be printed. If the final argument is present — the character ‘n’ — then the output of the product vector should *not* be shown.

6 Development

You may want to implement the first method with a block partition of the rows first. It’s essentially trivial in Pthreads, and you can concentrate on other issues.

You may want to write a function that will print your CSR matrices in dense format. Such a function will make it much easier to check the correctness of your randomly generated matrix and your matrix-vector multiplication.

For development you can use a node of the cluster to get four independent threads. (Note that it makes no sense to use multiple nodes of the cluster: only one node will be used, regardless of the number of threads.) For timing you may want to use one of our 8-core systems, grolsch, spaten, or stella. (Note that these are not identical systems: grolsch has 4 dual-core Opterons, while spaten and stella have 2 quad-core Xeons.)

7 Pthreads

The matrix multiplication described in methods 1 and 2 requires no coordination among the threads: the main thread reads or generates $A$ and $x$. It then starts the threads. Since the threads are writing to different locations in the product vector $y$, there won’t be a race condition. There can be false sharing, but this can only affect performance, not correctness.

For the third method, there will need to be coordination among the threads. Checking and updating the marker variable which indicates the first available row will introduce a race condition. This can be easily handled with a mutex.

For timing, you may need to synchronize the threads. (This will depend on whether the timing is started by the main thread or the other threads.) Linux supports Pthreads barriers with the functions `pthread_barrier_init`, `pthread_barrier_wait`, and `pthread_barrier_destroy`. If you’re developing on a system that doesn’t support Pthreads barriers, you can use the function `My_barrier` in the program [http://cs.usfca.edu/peter/cs625/code/pth_trap_time.c](http://cs.usfca.edu/peter/cs625/code/pth_trap_time.c) on the class website. (Note that in order to use this function, you’ll need to initialize and destroy a condition variable and a mutex in the main thread.)
8 Documentation

Follow the standard rules for documentation. Describe the purpose and algorithm, the parameters and variables, and the input and output of each routine.

If you come up with any alternative schemes for implementing CSR matrix-vector multiplication, be sure to give a careful description of how they work.

9 Grading

Correctness will be 80% of your grade. Does your program correctly implement the required methods for sparse matrix-vector multiplication? Does it accept input and command line options in the required format? Does it output the correct information?

Quality of solution will be 15% of your grade. Is the implementation reasonably efficient? Is the program well-designed and easy-to-read?

Static features such as source format and documentation will be 5% of your grade?

Well-implemented and well-documented additional, efficient methods that use Pthreads are worth 5 points each. Efficient MPI methods are also worth 5 points each. OpenMP methods are worth 2 points each.

The fastest program (sum of the runtimes of the three methods) will get 10 points extra credit. The next two programs will get 5 points each.

10 Collaboration

You may discuss all aspects of the program with your classmates. However, you should never show any of your code to another student, and you should never look at anyone else’s code — regardless of its source.

11 Submission

The program is due on Wednesday, March 23. By 10 am, you should have copied your source files, makefiles, and any additional files to your cs625/prog3 SVN directory. Put a hardcopy of your source code in my mailbox in Harney 545 by 2 pm.