Due: Friday March 20, 2020

In this project you will use OpenMP to create multithreaded versions of the programs you wrote for the first project, perform performance testing for each version, and write a report documenting your results.

Minimum Requirement: (70% of maximum) Make appropriate use of the "`#pragma omp parallel for`" directive to speed up the for-loop version of your program from Project 1.

Option 1: (90% of maximum) In addition to the minimum requirement, create two more programs that each use a single "`#pragma omp parallel`" directive. One of the programs should be based on the for-loop program from Project 1 while the other should be based on the CBLAS version. In contrast to the minimum requirement program, these versions will start one set of threads which will work collaboratively to carry out the power method and then terminate when the work is complete. For this option you may assume that the matrix dimension is a multiple of the number of threads so that each thread will be responsible for the same number of matrix rows as every other thread. Note that your program should enforce this restriction; if the matrix dimensions are not multiples of the number of threads, the program should display a message to that effect and quit.

Option 2: (100%) Extend your Option 1 programs so that they each allow for matrix dimensions that are not a multiple of the number of threads. For this option, each thread will need to determine which rows of the matrix it will work with.

Testing: Carry out testing for each version of the program you complete. At a minimum this should include running your programs multiple times and computing average solve times. (You might want to use the script `http://www.cs.gordon.edu/courses/cps343/projects/gen_data.sh` to automate much of this work. You should read the comments in the script as you will probably need to make some changes to make it works with your programs). You might also consider determining the number of floating point operations required in the Power Method and use this to compute gigaflop rates.

Project Report: To complete your work on this project, write a short report following the guidelines in `http://www.cs.gordon.edu/courses/cps343/projects/writing.pdf`. As you write the implementation section you may assume that the reader knows and understands the Power Method, but you will need to explain why you choose to place the OpenMP directives where you did, especially if you completed Options 1 or 2. Be sure your introduction clearly indicates which options you completed, and your discussion presents your results and documents your testing procedure.

Parallel implementation hints and suggestions

For the minimum requirement you will only need to make minor changes most of which will be inserting appropriate "`#pragma omp parallel for`" directives.

Options 1 and 2 will require significantly more work. You should specify a single parallel thread body with something like

```
#pragma omp parallel // may want default(none) and then shared() here
{
  // determine thread number and total number of threads
  const int num_threads = omp_get_max_threads();
  const int rank = omp_get_thread_num();

  //
  // code to determine the first and last rows of the matrix this
  // thread will work with; it is also helpful to compute the number
  // of these rows with something like length = last - first + 1.
  // (this is the code that differentiates Option 2 from Option 1)
  //

  // main power method loop
  while ( fabs( lambda - lambda_old ) > tol && num_iter <= max_iter )
  {
    ...
  }
}
```

Note that this includes the power method's main loop. You'll need to think carefully about shared variables and critical sections. You'll also need to use "`#pragma omp barrier`" to synchronize the threads at several points and use "`#pragma omp critical`" to protect any critical regions. Depending on your approach, you may also need to use the directives "`#pragma omp master`" or "`#pragma omp single`." The following discussion should help you think about how to create the body of each thread.

Suppose we have a parallel machine with p processors. Following the PCAM design approach, we begin by partitioning the problem into many small individual tasks. We can partition the matrix A into individual rows where $\mathbf{a}_i$ is a $1 \times n$ row vector that is the i^{th} row of A. Then the matrix A and the product $\mathbf{y} = A\mathbf{x}$ can be written

$$A = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \\ . \\ . \\ . \\ \mathbf{a}_n \end{bmatrix}, \qquad \mathbf{y} = A\mathbf{x} = \begin{bmatrix} \mathbf{a}_1\mathbf{x} \\ \mathbf{a}_2\mathbf{x} \\ \mathbf{a}_3\mathbf{x} \\ . \\ . \\ . \\ \mathbf{a}_n\mathbf{x} \end{bmatrix}$$

where the i^{th} entry of $\mathbf{y}$ is computed via an *inner product*:

$$y_i = \mathbf{a}_i \mathbf{x} = \sum_{j=1}^{n} a_{ij} x_j.$$

Each of these products is a task that can be computed in parallel. If n is larger than the number of threads p we can assign multiple tasks to each thread during the agglomeration and mapping phases.

Each task uses a single row of A and the entire vector $\mathbf{x}$ in order to compute a single y_i. Every task must then make its y_i value to all other tasks since the entire vector $\mathbf{y}$ is normalized to become the vector $\mathbf{x}$ for the next iteration.

One obvious way to agglomerate and map is to group tasks together based on the rows of A they work with (and therefore the portion of $\mathbf{y}$ they work with). The rows may be interleaved or consecutive, but it's probably most natural for a group of tasks to work with consecutive rows of A. If there are p threads then the matrix A will be partitioned into individual row blocks A_i, $i = 1, \ldots, p$ where each block has roughly n/p rows (except in Option 1 where each block will have exactly n/p rows). The product $\mathbf{y} = A\mathbf{x}$ then can be written as

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{y}_p \end{bmatrix} = \begin{bmatrix} A_1\mathbf{x} \\ A_2\mathbf{x} \\ \cdot \\ \cdot \\ \cdot \\ A_p\mathbf{x} \end{bmatrix}$$

and thread i can compute $\mathbf{y}_i = A_i\mathbf{x}$ independently of the other threads.

The estimated eigenvalue is computed $\lambda = \mathbf{x}^T\mathbf{y}$. Note that this can also be computed in pieces with

$$\lambda = \mathbf{x}_1^T\mathbf{y}_1 + \mathbf{x}_2^T\mathbf{y}_2 + \cdots + \mathbf{x}_p^T\mathbf{y}_p \qquad \text{or} \qquad \lambda = \lambda_1 + \lambda_2 + \cdots + \lambda_p$$

where $\lambda_i = \mathbf{x}_i^T\mathbf{y}_i$ and $\mathbf{x}_i$ the same portion of $\mathbf{x}$ that $\mathbf{y}_i$ is of $\mathbf{y}$. Thus each λ_i can be computed simultaneously and then accumulated into λ using a critical section.

Finally, $\mathbf{x}$ must be updated from $\mathbf{y}$ and normalized. One way to do this is to have each thread

- compute $z_i = \mathbf{y}_i^T\mathbf{y}_i$ and accumulate this into a shared variable, say z (you should probably use more descriptive names).
- divide every element of $\mathbf{y}_i$ by $\sqrt{z}$.
- copy $\mathbf{y}_i$ to $\mathbf{x}_i$ – now every thread has access to the updated vector $\mathbf{x}$.

Helps and Hints

Using OpenMP

To adapt your program to use OpenMP you will need to

- compile with the `-fopenmp` flag,
- include the header file `omp.h`,
- insert appropriate `#pragma omp` directives, and
- think carefully about what variables should be shared between threads.

In particular, you will need to ensure that after multiple threads work together to update a value (e.g. λ) or a vector (e.g. $\mathbf{x}$) there is a barrier or other form of synchronization so that no threads use it until the update is complete.

Computing partition sizes

It is frequently the case in partitioning domains that we need to compute starting and ending indices in an array for a particular part of the partition. This is straightforward when the number of domain elements is a multiple of the number of partition parts. For example, if a 100-element array is to be partitioned into four parts in a balanced way, each part should have one quarter of the elements. In this case we would have

Part	Start	End	Length
0	0	24	25
1	25	49	25
2	50	74	25
3	75	99	25

The computations here are simple:

- the length of each part is just the total array length divided by the number of parts.
- the starting index of the i^{th} part is i times the part length.

If you attempt Option 2, you will need some way to assign an unequal number of rows to the threads. In the general case, the array length will not be a multiple of the number of parts so we will have some slight imbalance but still want to have all parts be almost the same size. For example, if we wanted to split a 120-element array into 4 parts we might find the following:

Part	Start	End	Length
0	0	30	31
1	31	61	31
2	62	91	30
3	92	121	30

Suppse there are four threads and we want to split a 122-element data structure into four parts, one for each thread. If thread i calls the function listed below with

```
decompose1d( 122, 4, i, &s, &e );
```

then s and e will have the start and end values shown in the table above corresponding to part i.

By the way, this is a useful routine, worth keeping around somewhere as you might have occasion to use it in the future.

```
/*
 * Computes the starting and ending displacements for the ith
 * subinterval in an n-element array given that there are m
 * subintervals of approximately equal size.
 *
 * Input:     int n     - length of array (array indexed [0]..[n-1])
 *            int m     - number of subintervals
 *            int i     - subinterval number
 *
 * Output:    int* s    - location to store subinterval starting index
 *            int* e    - location to store subinterval ending index
 *
 * Suppose we want to partition a 100-element array into 3 subintervals
 * of roughly the same size.  The following three pairs of calls find
 * the starting and ending indices of each subinterval:
 *   decompose1d( 100, 3, 0, &s, &e );  (now s =  0, e = 33)
 *   decompose1d( 100, 3, 1, &s, &e );  (now s = 34, e = 66)
 *   decompose1d( 100, 3, 2, &s, &e );  (now s = 67, e = 99)
 *
 * The subinterval length can be computed with e - s + 1.
 *
 * Based on the FORTRAN subroutine MPE_DECOMP1D in the file
 * UsingMPI/intermediate/decomp.f supplied with the book "Using MPI"
 * by Gropp et al.  It has been adapted to use 0-based indexing.
 */
void decompose1d( int n, int m, int i, int* s, int* e )
{
    const int length  = n / m;
    const int deficit = n % m;
    *s =  i * length + ( i < deficit ? i : deficit );
    *e = *s + length - ( i < deficit ? 0 : 1 );
    if ( ( *e >= n ) || ( i == m - 1 ) ) *e = n - 1;
}
```