Due: Friday April 10, 2020

In this project you will use MPI to create parallel 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: (80% of maximum) Create a program that implements the Power Method in parallel using MPI. Base this program on the "for-loop" (i.e., the "non-CBLAS") version of one of your previous programs. If you completed either Option 1 or 2 of Project 2 you should use that as your starting code base, otherwise, use your code from Project 1. In this project you must use the `decompose1d()` function (or equivalent) so that your program can work with matrix sizes that are not strict multiples of the number of processes. *See the implementation notes below for some helpful details about this and other things.*

Test your programs on the Minor Prophets (MP) cluster and Canaan cluster using the same HDF5 datafiles for the matrix data as used for the previous projects.

Full-Credit Option: (100% of maximum) In addition to the minimum requirement, create a hybrid MPI-OpenMP version of your program: MPI will be used to communicate between the nodes, each of which will be using OpenMP to use multiple threads to work on a portion of the overall problem. To do this, use appropriate "`#pragma omp parallel for`" directives on the significant for-loops in your program – just as you did in the minimum requirement for Project 2.

Testing: Carry out testing for each version of the program you complete. At a minimum this should include running your programs multiple times for each particular run-time configuration and computing average solve times. (You might want to modify the script `http://www.cs.gordon.edu/courses/cps343/projects/gen_data.sh` used for the last project to automate much of this work.

Before starting your testing, view the manual page for `salloc` or `srun`, paying special attention to the description of the flags `--ntasks`, `--nodes`, and `--ntasks-per-node`. Use these to control the overall number of processes created and the number running on each node (remember, communication will be faster between processes running on the same node). If you completed the full-credit option, also read about the `--cpus-per-task` flag to `srun` or `salloc` so you will know now to use it to allocate a specific number of cores to each process. Experiment with different values but ensure that you don't "oversubscribe" the nodes by create more threads than cpu cores.

The node resources managed by Slurm are organized into *partitions*; you can see a list of partitions by typing `sinfo` at a terminal prompt. For example, on the MP cluster the output of `sinfo` shows

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
primary*     up   infinite     11   idle mp[02-12]
allNodes     up   infinite     12   idle mp[01-12]
admin        up   infinite     12   idle mp[01-12]
headNode     up   infinite      1   idle mp01
```

Here the default partition is denoted with a `*` and we see that it includes 11 nodes while the partition `allNodes` includes all 12 workstations. Since each of the workstations has 4 cores, this means the default partition gives you access to 44 cores while if you used the `allNodes` partition you can use all 48 cores.

On Canaan the output of `sinfo` shows

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
phys*        up   infinite     16   idle node[00-15]
chem         up   infinite      2   idle node[16-17]
allNodes     up   infinite     18   idle node[00-17]
```

The first 13 nodes in the `phys` partition each of 12 cores while the last three only have 8 cores. The total number of cores in this partition is 180. Both nodes in the `chem` partition have 16 cores, for a partition-wide total of 32 cores. Using the `allNodes` partition gives you access to all 212 cores available on the cluster's compute nodes.

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 clearly which data is local and which data is distributed across the processors and how the MPI functions are used to in the algorithm. 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

It is both important and helpful to recognize that the program(s) you write for this project, unlike the multithreaded programs in Project 2, do not start as a single execution thread, split into separate processes, and then rejoin. Instead, the MPI system will start multiple copies of your program and all of them will terminate when the work is done. No data is shared between the processes.

The file `http://www.cs.gordon.edu/courses/cps343/projects/readMatrixMPI.cc` contains a function called `readMatrixMPI()` that reads the matrix from an HDF5 file in parallel. This function uses the `decompose1d()` and so automatically handles the case where the number of matrix rows is not a multiple of the number of processes. The overall matrix dimensions

and starting and ending row numbers for the calling process are returned in parameters. The easiest way to make use of this file is to copy it into your working directory and add the statement `#include "readMatrixMPI.cc"` along with the other include statements in your program.

Pseudocode for the MPI version of the Power Method is shown below. As the matrix data is distributed across the processes, we will use A_r to indicate the portion of the matrix that the process with rank r is responsible for. Further, we assume that f_r and l_r (for "first" and "last") are first and last rows of the entire matrix that the rank r process is responsible for. Similarly, we can use $\mathbf{x}_r$ and $\mathbf{y}_r$ to indicate the portion of each of the vectors that the rank r process is responsible for. Finally, several scalar values will need to be computed using a reduction across the processes. We'll use the subscript r to indicate the rank r processes contribution to this value. For example, each process computes the inner product $\lambda_r = \mathbf{x}_r^T \mathbf{y}_r$ and then the λ_r values are combined into a new eigenvector estimate λ through a reduction operation.

$A_r, f_r, l_r \leftarrow$ read matrix from HDF5 file
$n_r := l_r - f_r + 1$
$F \leftarrow \text{allgather}(f_r)$ *array of offsets to first rows in each process*
$N \leftarrow \text{allgather}(n_r)$ *array of number of rows assigned to each process*
$\mathbf{x} := (1, 1, 1, \cdots, 1)^T$
$\mathbf{x} := \mathbf{x}/||\mathbf{x}||$
$\lambda := 0$
$\lambda_0 := \lambda + 2\epsilon$
$k := 0$
while $|\lambda - \lambda_0| \geq \epsilon$ and $k \leq M$ do
    $k := k + 1$
    $\mathbf{y}_r := A_r \mathbf{x}$
    $\lambda_0 := \lambda$
    $\lambda_r := \mathbf{x}_r^T \mathbf{y}_r$
    $\alpha_r := \mathbf{y}_r^T \mathbf{y}_r$
    $\lambda \leftarrow \text{allreduce}(\lambda_r)$
    $\alpha \leftarrow \text{allreduce}(\alpha_r)$
    $\mathbf{y}_r := \mathbf{y}_r / \sqrt{\alpha}$
    $\mathbf{x} \leftarrow \text{allgatherv}(\mathbf{y}_r, N, F)$
end while

Using MPI and OpenMP together

It is generally true that one should take advantage of the most natural and efficient means of parallelization. In a cluster constructed of SMP nodes (SMP refers to *symmetric multiprocessing*, i.e., multicore computing), it is reasonable to use OpenMP to write multithreaded

programs to run run on each node and use MPI to commutate between nodes.

One of the things we found in Project 2 was that parallelizing several key for-loops using `#pragma omp parallel for` directives yielded code that was just as fast as the code with more substantial changes to work with with a single pool of parallel threads, and took far less programmer-time! Let's take advantage of that now.

For the full-credit option in this project, you should make copy of your MPI version and

1. add the `-fopenmp` flag to the compiler command line
2. add a statement to include the `omp.h` header file
3. add appropriate `#pragma omp parallel for` directives to the key for-loops (matrix multiplication, dot product, copy vector, scale vector)

That's it! Testing this program will require careful use of flags to `srun` or `salloc` to make sure you're not oversubscribing cpu cores and to make sure that you are also reserving cpu cores so that Slurm doesn't schedule someone else's job using the same cores.