An introduction to the EML Linux cluster and the basics of parallel programming

April 3, 2013

Note: my examples here will be silly toy examples for the purpose of keeping things simple and focused on the parallelization approaches.

1 Overview of parallel processing computers

There are two basic flavors of parallel processing (leaving aside GPUs): distributed memory and shared memory. With shared memory, multiple processors (which I'll call cores) share the same memory. With distributed memory, you have multiple nodes, each with their own memory. You can think of each node as a separate computer connected by a fast network.

1.1 Distributed memory

Parallel programming for distributed memory parallelism requires passing messages between the different nodes. The standard protocol for doing this is MPI, of which there are various versions, including *openMPI*.

However, the EML cluster is not currently set up to allow for message passing between nodes, so I won't cover MPI here.

1.2 Shared memory

For shared memory parallelism, each core is accessing the same memory so there is no need to pass messages. But programmers writing parallel code need to be careful that activity on different cores doesn't mistakenly overwrite places in memory that are used by other cores. This generally won't be an issue for most EML users who are simply using built-in parallel capabilities in Matlab and Stata.

Some of the shared memory parallelism approaches that we'll cover are:

1. threaded calculations in Matlab and Stata
2. multicore functionality in Matlab
3. general purpose threaded C programs using openMP

Threading Threads are multiple paths of execution within a single process. Using *top* to monitor a job that is executing threaded code, you'll see the process using more than 100% of CPU. When this occurs, the process is using multiple cores, although it appears as a single process rather than as multiple processes. In general, threaded code will detect the number of cores available on a machine and make use of them. However, you can also explicitly control the number of threads available to a process.

For most threaded code (that based on the openMP protocol), the number of threads can be set by setting the OMP_NUM_THREADS environment variable. E.g., to set it for four threads in bash:

```
export OMP_NUM_THREADS=4
```

Matlab and Stata are exceptions to this. Threading in Matlab can be controlled in two ways. From within your Matlab code you can set the number of threads, e.g., to four in this case:

```
feature('numThreads', 4)
```

To use only a single thread, you can use 1 instead of 4 above, or you can start Matlab with the singleCompThread flag:

```
matlab -singleCompThread ...
```

Threading in Stata can be controlled as follows from within your Stata code, setting four threads in this case:

```
set processors 4
```

2 Running jobs on the cluster and the queueing system

The EML cluster has 8 nodes, with each node having 32 cores. Most clusters and supercomputers have many more nodes but often fewer cores per node. So the cluster mainly set up for shared memory parallelism.

To submit a job to the cluster, you need to create a simple shell script file containing the commands you want to run. E.g., the script, named *job.sh*, say, may simply contain:

```
matlab -nodisplay -nodesktop -singleCompThread < sim.m > sim.out
```

To submit your job:

```
$ qsub job.sh
```

Or to submit to the high priority queue (a user can only use 12 cores at a given time in the high priority queue, but jobs take precedence over any in the low priority queue).

```
$ qsub -q high.q job.sh
```

I/O intensive jobs For jobs that read or write a lot of data to disk, it's best to read/write from the local hard disk rather than to your home directory on the EML fileserver.

Here's an example script that illustrates staging your input data to the local hard disk on the cluster node:

```
if [ !  -d /tmp/myusername/ ]; then mkdir /tmp/myusername; fi
cp ~/projectdir/inputData /tmp/myusername # this positions an input file on the local hard disk of the node that the job is on
matlab -nodisplay -nodesktop -singleCompThread < sim.m > sim.out
# your Matlab code should find inputData in /tmp/myusername and write output to /tmp/myusername
cp /tmp/myusername/outputData ~/projectdir/.
```

Note that you do NOT have to stage your data from the local hard disk. By default files will be read from and written to the directory from which you submitted the job, which will presumably be somewhere in your EML home directory. If you are reading or writing big files, you may want to stage data to the local disk to increase performance (though if I/O is a small part of your job, it may not matter). And if I/O is a big part of your job (i.e., gigabytes and particularly tens of Gb of I/O), we DO ask that you do do this.

You can also stage the data on and off the local disk using an interactive job that accesses the specific cluster node, but in advance, you won't know what node your job will start on. (You can request a specific node, but we don't recommend that.)

2.1 Submitting jobs: To 'pe' or not to 'pe'

2.1.1 Non-parallel jobs

Threading in Matlab needs to be controlled explicitly by the user. If you are running in Matlab, you are **REQUIRED** to start Matlab with the singleCompThread flag:

```
matlab -singleCompThread ...
```

Alternatively, from within your Matlab code you can do

```
feature('numThreads', 1)
```

If you are using Stata in a non-parallel job, you should run your job with `stata` and not `stata-mp`.

Apart from Matlab and Stata, if you have not written any explicit parallel code, you can submit your job as above. By default the system limits any threaded jobs, including calls to the BLAS, to one core by having OMP_NUM_THREADS set by default to one.

2.1.2 Parallel jobs

If you want your job to use more than one core (either by using threaded code or by explicitly parallelizing your code), you need to do the following. First, submit your job with the "`-pe smp X`" flag to qsub, where X is the number of cores (between 2 and 32) that you want. E.g.,

```
$ qsub -pe smp 4 job.sh
```

Next in your code, make sure that the total number of processes multiplied by the number of threads per process does not exceed the number of cores you request, by following the guidelines below. Note that the NSLOTS environment variable is set to the number of cores you have requested via -pe smp, so you can make use of NSLOTS in your code.

Matlab jobs

1. To use **more than one thread within a single process**, you should including the following Matlab code in your script:
   `feature('numThreads', str2num(getenv('NSLOTS')));`
   This will make available as many threads as cores that you have requested.

2. To **run multiple processes** in a *parfor* job, you should set the number of workers to $NSLOTS in *matlabpool()*. (Template code is provided in the next section.) By default it appears that Matlab only uses one thread per worker, so the result should be that your job uses no more cores than you have requested. Note: if you're interested in combining threading with *parfor*, email consult@econ.berkeley.edu, and we can look into whether there is a way to do this.

Matlab limits you to at most 12 cores per job, so there is no point in requesting any more than that. If you want to use more cores, this may be possible via the Parallel Computing Toolbox; email consult@econ.berkeley.edu, and we can look into this.

Stata/MP jobs If you want to use multiple cores in Stata, you need to invoke `stata-mp` rather than `stata`. By default, Stata/MP uses eight cores. Our license does not allow you to use more, so there is no point in requesting more than eight cores via -pe smp. If you request fewer than eight cores, then your Stata/MP job should set the number of cores used to be equal to NSLOTS.

To hard-code the number of cores into your Stata code (in this case selecting four cores) you can do:

```
set processors 4
```

One way to automate this, rather than hard-coding the number of cores into your code is to start your Stata script as:

```
stata -b do myStataCode.do $NSLOTS
```

And at the start of your Stata code file, include these lines:

```
args ncores
set processors `ncores'
```

Other kinds of jobs

1. To use **more than one thread within a single process**, set OMP_NUM_THREADS to NSLOTS in your script:

   ```
   export OMP_NUM_THREADS=$NSLOTS
   ```

   This will make available as many threads as cores that you have requested.

2. To **run multiple processes** via explicit parallelization in your code, but **with a single thread per process**, you need to create only as many processes as you have requested. We'll see various ways for doing this below when using C. Since OMP_NUM_THREADS defaults to one on the cluster, only a single thread per process will be used.

3. Finally, to **run multiple processes with more than one thread per process**, you need to make sure that the total number of threads across all processes is no more than the number of cores you have requested. Thus if you want to have H threads per process and your code starts P processes, you should request $H \times P$ cores via -pe smp and you should then set OMP_NUM_THREADS to H.

2.1.3 Reserving multiple cores

If the queue is heavily loaded and you request many slots via the -pe smp syntax, that many slots may not come open all at once for a long time, while jobs requesting fewer cores may slip in front of your job. You can request that the queue reserve (i.e., accumulate) cores for your job, preventing smaller jobs from preempting your job. To do so, include "-R y" in your qsub command, e.g.,

```
$ qsub -pe smp 20 -R y job.sh
```

2.2 Monitoring jobs

You can get information about jobs using *qstat*.

```
$ qstat # shows your jobs
$ qstat -u "*" # shows everyone's jobs
$ qstat -j pid # shows more information about a single job
$ qdel pid # deletes a single job
$ qdel -u username # deletes all your jobs
```

Among other things, qstat will show you what node the jobs is running on. If you want to be able to monitor it via top, you can log on to that node in an interactive session as described next.

2.3 Interactive jobs

In addition to submitting batch jobs, you can also run a job from the command line on a node of the cluster.

```
qrsh -q interactive.q
```

You can then use the node as you would any other Linux compute server, but jobs are limited to 30 minutes. The primary use of this functionality is likely to be prototyping code and debugging. However, note that cluster nodes are set up very similarly to the other Linux compute servers, so most code should run the same on the cluster as on any other EML Linux machine.

In some cases, you may want to specify a particular node, such as for data transfer to a particular node or to monitor a specific batch job. In this case (here for node eml-sm01), do:

```
qrsh -q interactive.q -l hostname=eml-sm01
```

Once on the node you can transfer files via *scp*, use *top* to monitor jobs you have running, etc.

Note that the NSLOTS environment variable is not set in interactive jobs.

3 Matlab

3.1 Threaded operations

Many Matlab functions are automatically threaded, so you don't need to do anything special in your code to take advantage of this. Just make sure to follow the instructions in the previous section about setting the number of threads based on the number of cores you have requested.

3.2 Parallel for loops

To run a loop in parallel in Matlab, you can use the *parfor* construction. Note that this only makes sense if the iterations operate independently of one another. If you're doing this on the cluster, you must set the *matlabpool* size to the number of slots requested via the *-pe smp* flag to your qsub submission. (It appears that Matlab only uses one thread per worker.)

```
nslots = str2num(getenv('NSLOTS')); # if running on the cluster
# nslots = 8; # otherwise choose how many cores you want to use
matlabpool(nslots);
# matlabpool open local nslots # alternative
parfor i = 1:16
     c(:,i) = eig(rand(1000));
end
matlabpool close
```

3.3 Matlab Parallel Computing Toolbox functionality

You can also explicitly program parallelization, managing the individual parallelized tasks. I won't provide much detail here, but here are a few of the key functions.

- You can use *createJob()* and related functions (*createTasks(), submit(), wait(), fetchOutputs()*) to send different chunks of code to different cores.
- The *batch()* function allows you to execute a chunk of code as a separate process.

4 Stata/MP

Stata/MP has parallelized a number of its algorithms and routines; details are available here. The parallelization happens behind the scenes, and as long as you use `stata-mp`, you will be taking advantage of this without any changes to your code. As mentioned previously, one can set the number of cores, which is set to eight by default.

Using *top* to monitor a Stata/MP job, if parallelized code is being used, you'll see that top reports more than 100% CPU usage, indicating that Stata/MP is using threading.

5 Threaded linear algebra and the BLAS

The BLAS is the library of basic linear algebra operations (written in Fortran or C). A fast BLAS can greatly speed up linear algebra relative to the default BLAS on a machine. Some fast BLAS libraries are Intel's MKL, AMD's ACML, and the open source (and free) openBLAS (formerly GotoBLAS). The default BLAS on the EML Linux cluster is ACML and on the Linux compute servers is openBLAS. All of these BLAS libraries are threaded - if your computer has multiple cores and there are free resources, your linear algebra will use multiple cores, provided your pro-

gram is linked against the specific BLAS and provided OMP_NUM_THREADS is not set to one. Note that OMP_NUM_THREADS is set to one on the cluster.

Matlab threads certain linear algebra and other calculations by default (but uses a different mechanism than openMP), so if you're running Matlab and monitoring *top*, you may see a process using more than 100% of CPU.

To use threaded BLAS calls in a function you compile from C/C++, just make your usual BLAS or Lapack calls in your code. Then link against BLAS and Lapack (which on our system will automatically link against openBLAS). Here's an example C++ program (*testLinAlg.cpp*) and compilation goes like this (the R and Rmath links are because I use R's rnorm function):

```
g++ -o testLinAlg testLinAlg.cpp -I/usr/share/R/include -llapack
-lblas -lRmath -lR -O3 -Wall
```

If you'd like to link against ACML's BLAS and LAPACK (not necessary on the cluster, where ACML is the default):

```
g++ -o testLinAlg testLinAlg.cpp -I/usr/share/R/include -L/opt/acml5.2.0/
-lacml_mp -lgfortran -lgomp -lrt -ldl -lm -lRmath -lR -O3 -Wall -fopenmp
```

That program is rather old-fashioned and doesn't take advantage of C++. One can also use the Eigen C++ template library for linear algebra that allows you to avoid the nitty-gritty of calling Lapack Fortran routines from C++. Eigen overloads the standard operators so you can write your code in a more natural way. An example is in *testLinAlgEigen.cpp*. Note that Eigen apparently doesn't take much advantage of multiple threads even when compiled with openMP, but in this basic test it was competitive with openBLAS when openBLAS was restricted to one core.

```
g++ -o testLinAlgEigen testLinAlgEigen.cpp -I/usr/include/eigen3
-I/usr/share/R/include -lRmath -lR -O3 -Wall -fopenmp
```

Fixing the number of threads (cores used)

In general, if you want to limit the number of threads used, you can set the OMP_NUM_THREADS variable. This can be used in the context of R or C code that uses BLAS or your own threaded C code, but this does not work with Matlab. In the UNIX bash shell, you'd do this as follows (e.g. to limit to 3 cores) (do this before starting R):

```
export OMP_NUM_THREADS=3 # or "setenv OMP_NUM_THREADS 1" if using
csh/tcsh
```

Alternatively, you can set OMP_NUM_THREADS as you invoke R:

```
OMP_NUM_THREADS=3 R CMD BATCH --no-save job.R job.out
```

OMP_NUM_THREADS is set by default to 1 for jobs submitted to the cluster.

Speed and threaded BLAS

Warning: In many cases, using multiple threads for linear algebra operations will outperform using a single thread, but there is no guarantee that this will be the case, in particular for operations with small matrices and vectors. Testing with openBLAS suggests that sometimes a job may take more time when using multiple threads; this seems to be less likely with ACML. This presumably occurs because openBLAS is not doing a good job in detecting when the overhead of threading outweights the gains from distributing the computations. You can compare speeds by setting OMP_NUM_THREADS to different values. In cases where threaded linear algebra is slower than unthreaded, you would want to set OMP_NUM_THREADS to 1.

More generally, if you have an embarrassingly parallel job, it is likely to be more effective to use the fixed number of multiple cores you have access to to split along the embarrassingly parallel dimension without taking advantage of the threaded BLAS (i.e., restricting each process to a single thread).

Therefore I recommend that you test any large jobs to compare performance with a single thread vs. multiple threads. Only if you see a substantive improvement with multiple threads does it make sense to have OMP_NUM_THREADS be greater than one.

6 OpenMP for C: compilation and parallel for

It's straightforward to write threaded code in C and C++ (as well as Fortran). The basic approach is to use the *openMP* protocol. Here's how one would parallelize a loop in C/C++ using an *openMP* compiler directive. As with *foreach* in R, you only want to do this if the iterations do not depend on each other.

```
// testOpenMP.cpp
#include <iostream>
using namespace std;

// compile with:  g++ -fopenmp -L/usr/local/lib
//                  testOpenMP.cpp -o testOpenMP

int main(){
  int nReps = 20;
  double x[nReps];
  #pragma omp parallel for
  for (int i=0; i<nReps; i++){
```

```
    x[i] = 0.0;
    for ( int j=0; j<1000000000; j++){
      x[i] = x[i] + 1.0;
    }
    cout << x[i] << endl;
  }
  return 0;
}
```

We would compile this program as follows

```
$ g++ -L/usr/local/lib -fopenmp testOpenMP.cpp -o testOpenMP
```

The main thing to be aware of in using *openMP* is not having different threads overwrite variables used by other threads. In the example above, variables declared within the `#pragma` directive will be recognized as variables that are private to each thread. In fact, you could declare 'int i' before the compiler directive and things would be fine because *openMP* is smart enough to deal properly with the primary looping variable. But big problems would ensue if you had instead written the following code:

```
int main(){
  int nReps = 20;
  int j;  // DON'T DO THIS !!!!!!!!!!!!!
  double x[nReps];
  #pragma omp parallel for
  for (int i=0; i<nReps; i++){
    x[i] = 0.0;
    for (j=0; j<1000000000; j++){
      x[i] = x[i] + 1.0;
    }
    cout << x[i] << endl;
  }
  return 0;
}
```

Note that we do want x declared before the compiler directive because we want all the threads to write to a common x (but, importantly, to different components of x. That's the point.

We can also be explicit about what is shared and what is private to each thread:

```
int main(){
  int nReps = 20;
  int i, j;
  double x[nReps];
  #pragma omp parallel for private(i,j) shared(x, nReps)
  for (i=0; i<nReps; i++){
    x[i] = 0.0;
    for (j=0; j<1000000000; j++){
      x[i] = x[i] + 1.0;
    }
    cout << x[i] << endl;
  }
  return 0;
}
```

As discussed, when running on the cluster, you are required to use the *-pe smp* parallel environment flag and to set OMP_NUM_THREADS to NSLOTS .

7 OpenMP for C/C++: more advanced topics

The goal here is just to give you a sense of what is possible with *openMP*.

The OpenMP API provides three components: compiler directives that parallelize your code (such as `#pragma omp parallel for`), library functions (such as *omp_get_thread_num()*), and environment variables (such as OMP_NUM_THREADS)

OpenMP constructs apply to structured blocks of code.

Here's a basic "Hello, world" example that illustrates how it works:

```
// helloWorldOpenMP.cpp
#include <stdio.h>
#include <omp.h> // needed when using any openMP functions
//                                like omp_get_thread_num()

void myFun(double *in, int id){
// this is the function that would presumably do the heavy lifting
}
```

```
int main()
{
   int nthreads, myID;
   double* input;
   /* make the values of nthreads and myid private to each thread */
   #pragma omp parallel private (nthreads, myid)
   { // beginning of block
      myID = omp_get_thread_num();
      printf("Hello, I am thread %d\n", myID);
      myFun(input, myID);  // do some computation on each thread
      /* only master node print the number of threads */
      if (myid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
   } // end of block
   return 0;
}
```

The parallel directive starts a team of threads, including the master, which is a member of the team and has thread number 0. The number of threads is determined in the following ways - here the first two options specify four threads:

1. `#pragma omp parallel NUM_THREADS (4)` // set 4 threads for this parallel block
2. `omp_set_num_threads(4)` // set four threads in general
3. the value of the OMP_NUM_THREADS environment variable
4. a default - usually the number of cores on the compute node

Note that in #pragma omp parallel for, there are actually two instructions, 'parallel' starts a team of threads, and 'for' farms out the iterations to the team. In our parallel for invocation, we have done it more explicitly as:

```
#pragma omp parallel
#pragma omp for
```

We can also explicitly distribute different chunks of code amongst different threads:

```
// sectionsOpenMP.cpp
#pragma omp parallel // starts a new team of threads
{
   Work0(); // this function would be run by all threads.
   #pragma omp sections // divides the team into sections
   {
      // everything herein is run only once.
      #pragma omp section
      { Work1(); }
      #pragma omp section
      {
         Work2();
         Work3();
      }
      #pragma omp section
      { Work4(); }
   }
} // implied barrier
```

Here Work1, {Work2 + Work3} and Work4 are done in parallel, but Work2 and Work3 are done in sequence (on a single thread).

If one wants to make sure that all of a parallized calculation is complete before any further code is executed you can insert

`#pragma omp barrier`

Note that a `#pragma for` statement includes an implicit barrier as does the end of any block specified with `#pragma omp parallel`

You can use '`nowait`' if you explicitly want to prevent threads from waiting at an implicit barrier: e.g., `#pragma omp parallel sections nowait` or `#pragma omp parallel for nowait`

One should be careful about multiple threads writing to the same variable at the same time (this is an example of a race condition). In the example below, if one doesn't have the `#pragma omp critical` directive two threads could read the current value of '`sum`' at the same time and then sequentially write to sum after incrementing their local copy, which would result in one of the increments being lost. A way to avoid this is with the `critical` directive (for single lines of code you can also use atomic instead of critical:

```
// see criticalOpenMP.cpp
```

```
double sum = 0.0;
double tmp;
#pragma omp parallel for private (tmp, i) shared (sum)
for (int i=0; i<n; i++){
   tmp = myFun(i);
   #pragma omp critical
   sum += tmp;
}
```