Exercises 2: Running Sequential and Parallel Programs

NMNV565: High-Performance Computing for Computational Science

Things you will learn today

1. Module system on cluster
2. Sequential DGEMM performance
   - BLAS vs. naive code
   - BLAS3 vs. BLAS2
   - Blocked/tiled versions
3. Running a parallel program on the cluster
4. Performing weak and strong scaling experiments on the cluster

Lab Setup

1. Login to Karlin cluster
2. Download the archive ex2.tar from the class Moodle site
3. scp the ex2.tar file to your chosen directory on the cluster
4. Extract the files from the archive ("`tar -xf ex2.tar`")

Optimized vs. Naive DGEMM

- We will first take a look at the performance we can obtain with BLAS routines versus attempting to write code ourselves.
- Our task will be to multiply two n x n matrices, i.e., compute C = AB.

- Look inside the directory "dgemm"
- You should see files:
  - dgemm-blas3.c: contains a call to the BLAS3 routine 'dgemm'.
  - dgemm-blas2.c: computes C one column at a time; there is a single loop over the columns of C that computes C(:; j) = AB(:; j) by a call to the BLAS2 routine 'dgemv'.
  - dgemm-naive.c: contains C = AB implemented how you would compute it by hand, using three nested loops.

The Module System

- First, we want to make sure that we are using autotuned BLAS2/3 routines, which have been optimized for the architecture we are using. The cluster already has installed the optimized BLAS library OpenBLAS.

- The cluster uses a module system by which we can load different libraries for use.

1. Type "`module avail`" to see all available modules.
2. Type "`module load openblas`" in order to load the OpenBLAS module (this will load the default version).

Running the Benchmarks

- Open and read through the files dgemm-blas3.c, dgemm-blas2.c, and dgemm-naive.c to make sure you understand what is happening in each.

- Open the Makefile and see if you can understand it

- Type "`make`"
  - This will create object files dgemm-blas3.o, dgemm-blas2.o, and dgemm-naive.o
  - It also creates executable programs called benchmark-blas3, benchmark-blas2, and benchmark-naive.
    - The benchmark programs test the Mflops/s achieved by the dgemm routine on various test case sizes.

Running the Benchmarks

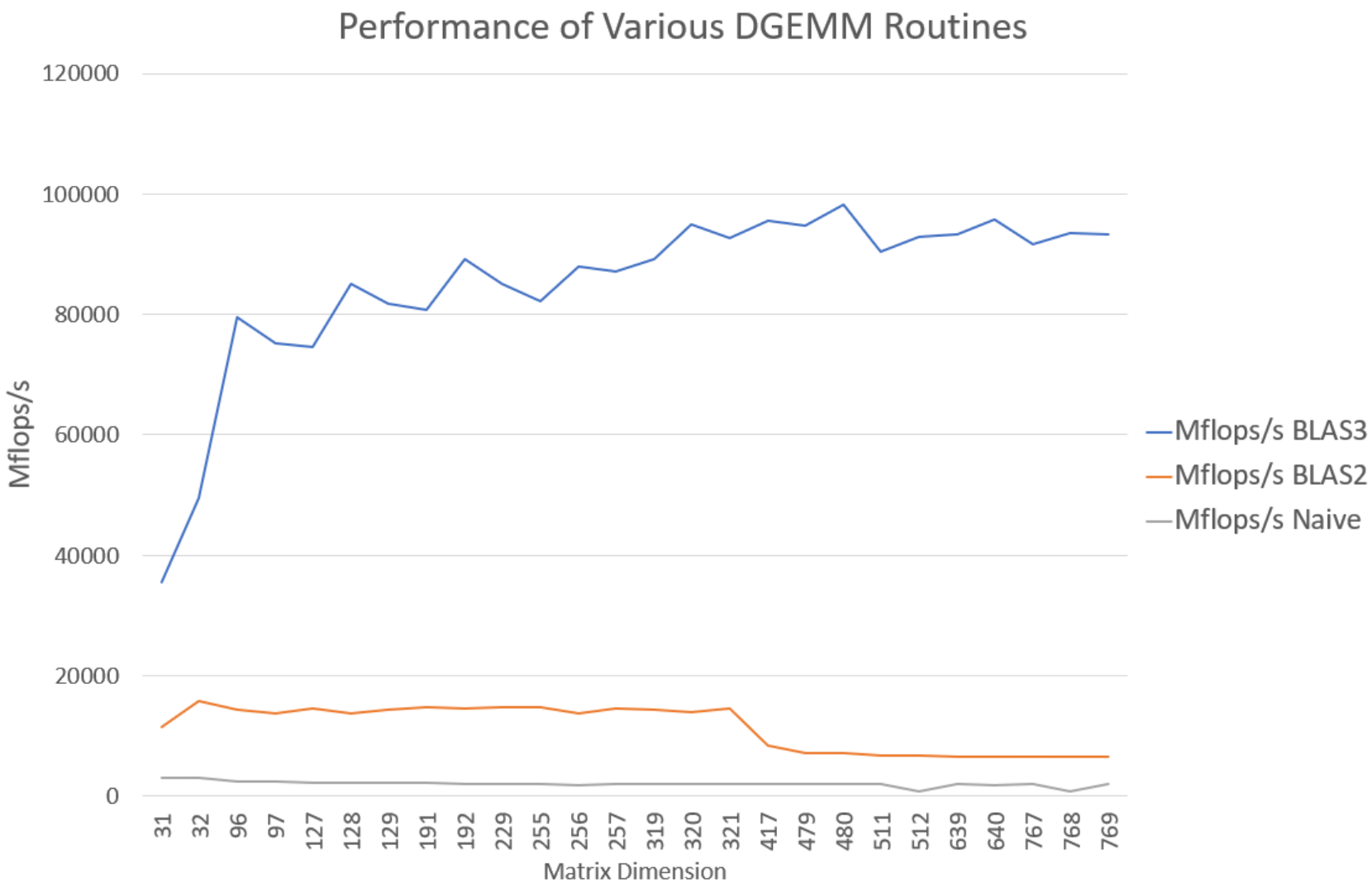
- The benchmark programs test the Mflops/s achieved by the dgemm routine on various test case sizes.

- Look at the job-all.sh script

- Submit the job-all.sh script to the SLURM queueing system.
  - ("`sbatch job-all.sh`")
- Once the job is finished, look at the output.

- What can you say about the relative performance of these methods?

Example


Performance of Various DGEMM Routines
120000
100000
80000
60000
40000
20000
0
Mflops/s
31
32
96
97
127
128
129
191
192
229
255
256
257
319
320
321
417
479
480
511
512
639
640
767
768
769
Matrix Dimension
Mflops/s BLAS3
Mflops/s BLAS2
Mflops/s Naive

Blocked DGEMM

- We might try to improve the performance of the naive dgemm routine by blocking. The file dgemm-blocked.c does just this, computing the matrix multiply in blocks of size b x b.

1. Open dgemm-blocked.c and make sure you understand it. Notice that the block size parameter is set by the #define BLOCK SIZE command.
   - It is set to 1 as a default. Your job will be to see if there is a better block size.
2. Type "`cat /proc/cpuinfo`" to see information about the cpus on the login node. Based on the cache size, determine what the theoretical optimal block size should be. Set this as the block size in dgemm-blocked.c
3. Open the Makefile and add commands in order to compile dgemm-blocked.o and link to create an executable called "benchmark-blocked". Then type "`make`" to create the benchmark-blocked executable.
4. Run the benchmark-naive script and the benchmark-blocked script, both on the login node for comparison. What do you observe? Try playing around with the blocksize a bit (but don't spend too much time here).
   - (Hint: try making it much smaller, something less than 100)

Running a Parallel Program

- The SLURM system uses parameters that tell it how to run the parallel program - how many nodes to use, how many cpus per node to use, etc. See standard SLURM documentation, and the Karlin cluster documentation for information about the available hardware.

-N 5

--ntasks-per-node=8

allocates 5 full nodes, and srun will run 8 mpi jobs per node (i.e., using a total of 40 cpus).

-N 5

-n 40

--ntasks-per-node=8

says to allocate 40 cpus, use exactly 5 nodes, and spread the allocation so that each node has 8 jobs (8+8+8+8+8).

-N 5

-n 40

will again allocate 40 cpus over 5 nodes, but the distribution to nodes will be up to the system (you can get, for example, 12+12+12+2+2).

-n 40

you will get an allocation like 12+12+8+8, so the minimal number of nodes to get 40 cpus.

Inner Product computation

- cd to the innerprod directory (cd ../innerprod)
- The program inner-mpi.c computes an inner product between two vectors
  - Look at the file

- inner-mpi must be run with 2 input parameters. The first is the size of the vectors on which to compute the inner product and the second is the number of times to repeat the experiment (I suggest setting this to something > 1 so your data will be more smooth, but this is up to you).

- Note that for this simple program the size of the vector must be divisible by the number of MPI processes you use.

Compiling the MPI Program

- We want to compile the MPI program. For this we need to load another module:

  Type "`module load openmpi`"

- To compile, we will use "mpicc" instead of "gcc".

  Type "`mpicc -o inner-mpi inner-mpi.c`"

- Start an interactive bash session by typing, e.g.,

```
srun -N 2 -p express3 --tasks-per-node 8 -t1:00:00 --pty /bin/bash -i
```

- This will give you 2 nodes with 8 tasks per node, so you can run up to 16 MPI processes.
- You need to again type "`module load openmpi`" on this new node
- (note: on a heterogeneous cluster like this, it might be a good idea to compile again once on the node where you will run the code)

Running the MPI Program

- To run the MPI program, we use the "mpirun" command.
- For example,

```
mpirun -N 1 -n 2 ./inner-mpi 1000000 100
```

This will compute an inner product with vectors of size 10000000 using 2 MPI processes, and report timings averaged over 100 runs.

You can type "exit" to exit the interactive bash session

Job scripts

- As an alternative to using an interactive session (or if the interactive session isn't working), you can use batch scripts and submit your jobs to the SLURM system
  - I have included an example of this for you for reference
    - see job-strong.sh

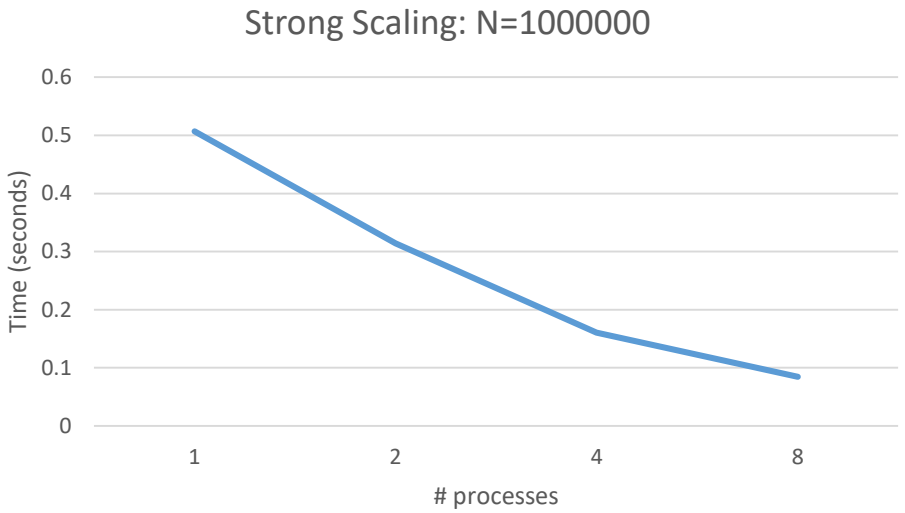
Scaling Experiments

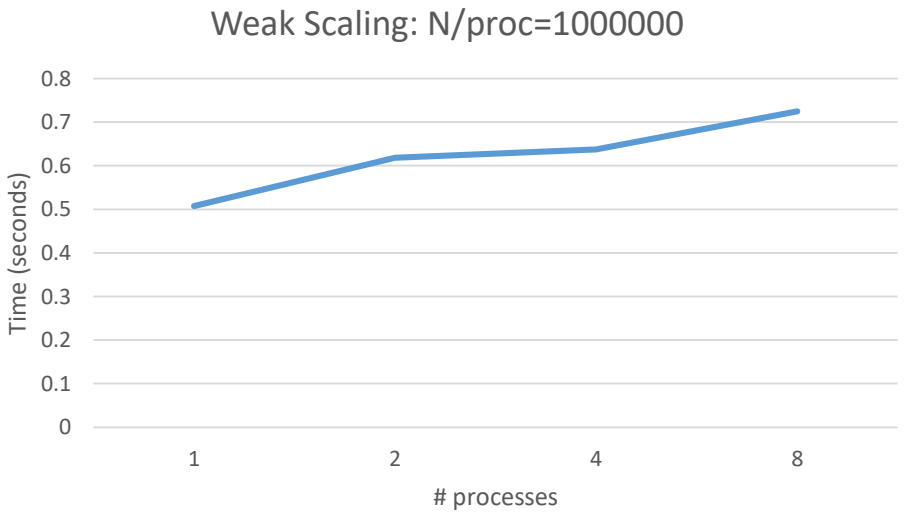
- Your task is to create two plots: a strong scaling plot and a weak scaling plot.

1. For the strong scaling plot, you should first pick the different numbers of processors you will test and make sure that your vector size is divisible by all these numbers.
2. For the weak scaling experiments, you should first choose a fixed problem size per processor. Then scale the number of processors and problem size together.

- Make sure the problems sizes are big enough that you can see the expected behavior (but not so big that they take a very long time to run!)

Example Expected Output





Summary

- By following the instructions, you should have generated 3 files:
  - 1 text file of your job comparing DGEMM performance
  - 1 plot of strong scaling for the inner-mpi program
  - 1 plot of weak scaling for the inner-mpi program