CME 213, Introduction to parallel computing
Eric Darve, William Jen
Winter 2020

Stanford University

Homework 5—due Friday March 13, 11:00pm

Total number of points: 100 (+ 0 bonus). Late day policy: 2 late days with a 10% grade penalty.

In this programming assignment, you will use the OpenMPI implementation of the Message Passing Interface (MPI) standard to implement the Dekel–Nassimi–Sahni (DNS) matrix-matrix multiplication algorithm. In the process, you will learn how to use point-to-point (P2P) communication, collective communication, cartesian topologies, and general MPI program flow and design. You must turn in your own copy of the assignment as described below. You may discuss the assignment with your peers, but you may not share answers. Please direct your questions about the assignment to Canvas.

The DNS Matrix-Matrix Multiplication Algorithm Although we will briefly introduce the DNS algorithm here, you should read the in-depth explanation of the algorithm in Chapter 8.2.3 of the textbook Introduction to Parallel Computing by Grama et al.[1] The book is accessible through Stanford SearchWorks.

Assume we have two square matrices A and B of size $n \times n$. Let $C = AB$ where $C_{ij} = \sum_{k=0}^{n-1} A_{ik}B_{kj}$. We have three dimensions i, j, and k. We can compute a single C_{ij} element in parallel by distributing A_{ik} and B_{kj} across n processes. Each of those processes will compute a local product $A_{ik}B_{kj}$. Then, we can reduce across all of these processes to obtain the element C_{ij}. Ignoring communication costs, this reduction-based approach takes $O(\log n)$ time. Since C has n^2 elements and we need n processes to compute each element in parallel, we need a total of n^3 processes.

Now that we have described the idea behind the DNS algorithm, let's add the communication back in. Assume our processes are arranged in a $n \times n \times n$ cube. Each process has coordinate (i, j, k), and all processes on the $k = 0$ plane are initialized with A_{ij} and B_{ij}. The DNS algorithm thus proceeds as follows:

- All $(i, j, 0)$ processes send their A_{ij} element to rank (i, j, j) and their B_{ij} element to rank (i, j, i).
- Each (i, j, j) process with the A_{ij} element then broadcasts A_{ij} along the j axis, so now each $(i, *, j)$ process have A_{ij}. Similarly, each (i, j, i) process broadcasts B_{ij} along the i dimension, so now all $(*, j, i)$ processes have B_{ij}.
- Each (i, j, k) process now has A_{ik} and B_{kj}. We now compute $A_{ik}B_{kj}$.
- We reduce along the k dimension to obtain the final C_{ij} result in the $(i, j, 0)$ rank.

The above description assumed we perform one multiplication per process. In this assignment, you will implement an improved version where we have fewer processes and each process multiplies a smaller, square matrix.

[1] https://searchworks.stanford.edu/view/5375111

Virtual Machines We will be using VM instances on the Google Cloud Platform. We have provided two scripts called `create_mpi8.sh` and `create_mpi64.sh` in the starter code to set up the correct virtual machines for this homework. You will have all the necessary libraries and tools pre-installed on the VM.

Use the `mpi8` virtual machine when implementing the DNS algorithm. Once you have the DNS algorithm working, use the `mpi64` to collect runtime data. In order to use `mpi64` VM, you will need to increase the quotas **CPUs (all regions)** and **CPUs** for us-west1 to 64. You can edit these quotas at the IAM & Admin Quota page.

Starter Code The starter code is composed of the following files (* means the file will *not* be submitted by our script):

- `*dnsmmm.cpp` — This file runs and tests your DNS implementation. Do not modify.
- `dns.h` — This file implements the DNS algorithm. You will need to modify this file.
- `*serialmmm.cpp` — This file outputs timings for naive matrix-matrix multiplication and an OpenMP matrix-matrix multiplication implementation. Do not modify.
- `*util.h` — This file contains helper functions such as serial matrix-matrix multiplication functions. Do not modify.
- `*Makefile` — `make` will build both `dnsmmm` and `serialmmm`.
- `*create_mpi8.sh` — This script will create the mpi8 VM. Use this VM when implementing and testing your DNS implementation is correct.
- `*create_mpi64.sh` — This script will create the mpi64 VM. Use this VM when collecting runtimes for your DNS implementation. **Make sure you have increased your CPU quotas before running this script**. See the Virtual Machines section for additional information.

Note The files in the starter code contain some additional information about the implementation in the form of comments. Read them carefully.

Running the program Type `make` to compile the code. Once this is done, you can test your DNS implementation by:

```
mpirun -mca btl ^openib -n [P] ./dnsmmm [N]
```

`P` is the number of processes to launch and `N` is the size of matrix, i.e. test using `NxN` matrices.

The program will output the time taken to run your parallel implementation as well as the L_2 error between your DNS output and the serial implementation. Typical error ranges should be on the order of $\left[10^{-7}, 10^{-5}\right]$.

Search for `TODO` in `dns.h` to see where you need to implement code.

Later in this homework, you will be collecting runtime for the DNS algorithm. To make it easier to collect this data, we have provided an additional Makefile target called `dnsmmm_timing` that skips the DNS verification. You can run it as follows:

```
mpirun -mca btl ^openib -n [P] ./dnsmmm_timing [N]
```

Additionally, you will also need to run the `serialmmm` executable to obtain timings for the naive matrix-matrix multiplication and the OpenMP implementation. You can invoke the executable as follows:

```
./serialmmm [-so]
```

where `-s` stands for serial and `-o` stands for OpenMP. You can run both flags at the same time, i.e. `./serialmmm -so`. This program will output a table of timings for whichever implementations you have enabled.

Question 1

(10 points) We first need to rearrange our process into a 3D Cartesian topology and initialize communicators to talk within that topology. Implement the `initialize_topology` and `mesh_info_free` functions.

Question 2

(45 points) Implement the `dns_multiply` function.

Question 3

(15 points) Compute the heap-allocated memory of the DNS algorithm as a function of the number of processes p and the matrix dimension n. Do not include the memory allocated for the source matrices and the final output matrix.

Question 4

(15 points) On the mpi64 VM, use the `dnsmmm_timing` executable to collect the runtime T_p of the DNS algorithm for matrix sizes from $n = 576$ to $n = 2880$ in increments of 96 for $p = [8, 27, 64]$. Run the `serialmmm` executable to collect timings for naive and OpenMP matrix-matrix multiplication. Plot the runtimes on a semilog y plot. Briefly comment on the observed trends.

Question 5

(15 points) The approximate parallel run time taken for this algorithm on a hypercube network is

$$T_p \approx \left(\frac{n}{q}\right)^3 + 3t_s \log q + 3t_w \left(\frac{n}{q}\right)^2 \log q$$

- n is the matrix dimension
- q is the topology dimension. In other words, our p processes are arranged in a $q \times q \times q$ cube.
- t_s is the time required to create a connection for data transfer.
- t_w is the time required to send a word across the network. This is typically inversely proportional to the available bandwidth between a pair of nodes.

Let p be the total number of processes which is also equal to q^3. Derive the following:

- T_p in terms of n and p
- The iso-efficiency of the DNS algorithm. Compare against the naive matrix-matrix multiplication algorithm.

  Hint. Chapter 5.4 in the Grama textbook describes iso-efficiency in greater detail.
- The growth rate of p as the problem size grows to maintain the same efficiency.

A Advice and hints

- Read the DNS algorithm in the textbook thoroughly and run through the algorithm by hand with a small matrix size (e.g. 4×4). As you implement each step, verify you get the expected matrices in each process by printing out its contents. The following debug code may prove useful within `dns_multiply`:

```
for (int i = 0; i < mesh_info.num_procs; i++)
{
  if (i == mesh_info.myrank)
  {
    printf("Rank = %d | Coord: (%d, %d, %d)\n", mesh_info.myrank,
           mesh_info.mycoords[0], mesh_info.mycoords[1], mesh_info.mycoords[2]);
    print_sq_matrix(YOUR_MATRIX_HERE, matrix_dim);
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

- Chapter 6 in the Grama textbook goes into more detail about common MPI functions and communicators.
- To better understand isoefficiency, refer to Chapter 5.4 in the Grama textbook.

B MPI Quick Reference

MPI is very extensive and thus it can be difficult to know which functions you may need. We have compiled a list of functions and constants that may be helpful in this homework.

Function/Constant	Description
`MPI_Comm_size`	Gets the number of ranks in the given communicator.
`MPI_Comm_rank`	Gets the rank of the current process in the given communicator.
`MPI_Comm_free`	Releases an allocated communicator.
`MPI_COMM_WORLD`	The default communicator that talks to all ranks.
`MPI_Cart_create`	Creates a cartesian topology.
`MPI_Cart_sub`	Creates a subtopology from a given cartesian topology.
`MPI_Cart_rank`	Gets this process' rank in the given cartesian topology.
`MPI_Cart_coords`	Gets this process' coordinates in the given cartesian topolgy.
`MPI_Scatterv`	Gives each rank in the specified communicator a chunk of the given data. The user specifies how much data to distribute to each rank.
`MPI_Gatherv`	Dual of `MPI_Scatterv`: Takes chunks of data from each rank in the given communicator and places it into the specified destination rank.
`MPI_Send`	Blocking send to a specified rank in a given communicator.
`MPI_Recv`	Blocking receive from the specified rank in a given communicator.
`MPI_Bcast`	Sends the given data from the specified rank to all ranks in the given communicator.
`MPI_Reduce`	Reduces the given data according to the given MPI operation (e.g. `MPI_SUM`) and places the result into another buffer on a specified rank.
`MPI_SUM`	Sum operation for `MPI_Reduce`.
`MPI_Barrier`	Wait for all processes in the given communicator to reach this statement. Similar to CUDA's `__syncthreads()`.
`MPI_FLOAT`	MPI datatype representing floating point values.

C Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file and upload this file on Gradescope.

2. Make sure your code compiles on Google Cloud VM and runs. To check your code, we will run:
   `$ make dnsmmm`
   This should produce one executable: `dnsmmm`

3. The homework should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.

4. Copy your submission files to `cardinal.stanford.edu`. The script `submit.py` will copy only the files below to a directory accessible to the CME 213 staff. Only these files will be copied. Any other required files (e.g., Makefile) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files for your own debugging purposes, but make sure you test it with the default test files before submitting. Also, do not use external libraries, additional header files, etc, that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   `dns.h`

5. To submit, type:

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw5 <directory with your submission files>
```

6. You can submit at most 10 times before the deadline; each submission will replace the previous one.