CSE 415: Introduction to Parallel Computing
Spring 2023, Homework 7
Due 11:59 pm, Friday, April 28th

Important note: *Please use a word processing software (e.g., MS Word, Mac Pages, Latex, etc.) to type your homework. Follow the submission instructions at the end to turn in an electronic copy of your work.*

1) [40 pts] MPI point-to-point vs collectives

Consider the trapezoidal rule for numerical integration (see relevant course slides and/or the textbook).

a) Implement two distributed memory parallel versions of the trapezoidal rule: i) using point-to-point communications, and ii) using collective communications. See the skeleton code and comments provided there for guidance.

b) Time your implementations of the trapezoidal rule using various numbers of sampling points (i.e., n = 1000, 100000, or 10M) and MPI processes (i.e., p = 1, 2, 5, 10, 20, 100, 250). How does the performance of your point-to-point and collective communication versions compare? What are the speedups and efficiencies? Based on the data you collected, what could you say about the scalability of your implementation? **Note:** Maximum number of cores that one is allowed to allocate at once is capped at 512 per HPCC policies.

2) [60 pts] Code Breaking

Encryption (and how to crack it) is one of the most fundamental concepts in cybersecurity. Data Encryption Standard (DES) is a widely used encryption method that entails the use of a 56-bit secret key for encrypting and decrypting data. Better security standards such as the Advanced Encryption Standard (AES) that offer much wider key lengths have been gradually replacing DES since 2001.

In this problem, you will work on a brute-force cracking of encrypted messages using MPI. While DES-encryption can be broken in a matter of days using resources at HPCC, this is certainly not practical for our purposes. For this reason, we will look at a simple 32-bit encryption algorithm that relies on XOR operations along with bit rotations. The file encrypter.c implements this 32-bit encryption algorithm sequentially. Similarly, in the file codebreaker.c, you are given a sequential code-breaking algorithm that simply tries all 2^{32} keys in a brute force manner to decrypt a file encrypted by the code in encrypter.c. While searching over the keys, the validation for each candidate is performed by comparing the words in the decrypted message against those in our small dictionary.

To encrypt a message, we have provided you with encrypt.c. After compiling encrypt.c, you can encrypt a text file as follows:

```
gcc encrypt.c -o encrypter

./encrypter inp1.txt <int key>
```

The first argument is the name of the text file you want to cipher. The second argument is the key value.

The encrypted message is saved into a new file with the ".ecp" extension. Note that, for debugging purposes, you'd probably want to experiment with smaller key values or key values that reside at the beginning of your partitions in the parallel version (for faster runs). Below, we explain how you could parallelize the brute search linear search.

Your task is to parallelize the brute force algorithm given in codebreaker.c file in a few different ways, and analyze the performance of the resulting implementation. There are two general possibilities for partitioning this problem:

• **Static partitioning** where the division of the workload is done a priori based on a predetermined mechanism,

• **Dynamic partitioning** where the division and assignment of the workload are done at run-time, permitting load balancing to be performed, albeit at the cost of a more complicated solution. We will only be looking at *static partitioning* techniques.

a) Parallelize codebreaker.c using block partitioning of the set of possible keys. Your code should follow the following outline:

i) Root process should read the encrypted message and broadcast it to all processes,

ii) Based on the total #processes, each process should determine its own set of trial keys,

iii) When a process discovers the right key, it should print the success message and write the decrypted message into a file. In addition, it should notify all other processes so that the MPI program can be terminated immediately and correctly.

Hint: This can be achieved by each process posting an immediate receive operation before trying their own keys. Then the process that discovers the correct key sends notifications to all others about program completion.

iv) *Do not change the dictionary, the program output message, and/or the output file name.* Your program will be tested by providing a set of random (but properly encrypted) messages (of length < 1000).

Hint: You may find the commented-out printf statements useful in understanding or debugging your parallel codes. Make sure to comment out any printf statements that you uncomment before submission. For initial testing, run your parallel codes with a single process only, and try to decrypt messages encrypted with small key values (i.e. key = 5, 10, 100).

b) Analyze the scalability of your implementation in **part a** on a single node of the intel18 cluster. Note that as opposed to regular problems, search algorithms can achieve *super-linear*, *linear* or *no* speedup compared to a sequential program (but the expected speed-up will still be linear). Demonstrate each of these possible scenarios by carefully choosing the encryption key and the number of processes (make sure to include examples using up to all 40 processes on a single intel18 cluster node).

c) Find and implement a partitioning strategy that overcomes the zero (or sublinear) speedup pitfall of the implementation in **part a**. Test your new code (version 2) on your examples from part b to show that your new partitioning strategy does indeed overcome the no (or sublinear) speedup pitfall.

BONUS:
[20 pts] MPI/OpenMP parallelization

On a multi-core architecture, good use of system resources can be achieved by first starting a single process for each physical socket in the system (on intel18, there are two sockets (or chips) per node), and then spawning as many threads as the number of cores per socket to fully exploit the multiple cores in that system. This is called MPI/OpenMP hybrid parallelization.

- Implement an MPI/OpenMP parallel version of the numerical integration code you developed in problem 1.
- Analyze the performance of the MPI/OpenMP parallel code with respect to the MPI-only parallel version. Do you observe better, worse or similar performance? Why do you think this is the case?

Instructions:

- **Cloning git repo.** You can clone the skeleton codes through the git repo. Please refer to *"Homework Instructions"* under the *"Reference Material"* section on D2L.

- **Submission.** Your submission will include:
  - A pdf file named "HW7*_yourMSUNetID*.pdf", which contains your answers to the non-implementation questions, and report and interpretation of performance results. o All source files used to generate the reported performance results. Make sure to use the exact files names listed below:
    - trapezoid.c (for problem 1)
    - codebreaker.c (for problem 2a)
    - codebreaker_v2.c (for problem 2c)
    - trapezoid_hybrid.c (for the bonus problem)

These are indeed the default names in the git repository, so do not change the directory structure or file names.

To submit your work, please follow the directions given in the "Homework Instructions" under the "Reference Material" section on D2L. Make sure to strictly follow these instructions; otherwise you may not receive proper credit.

- **Discussions.** For your questions about the homework, please use the Slack discussion forum for the class so that answers by the TA, myself or one of your classmates can be seen by others.

- **Compilation and execution.** You can use any compiler with MPI (and OpenMP) support. The default compiler environment at HPCC is GNU. You can compile an MPI parallel code as follows:

```
module load OpenMPI

mpicc codebreaker.c -o decrypter
```

The resulting MPI parallel binary must be executed using mpirun (for example using 4 processes):

```
srun –n 4 ./decrypter encrypted_filename.ecp
```

- For hybrid MPI/OpenMP compilation, you need to provide the –fopenmp flag to the mpicc compiler. Remember to set the OMP_NUM_THREADS environment variable, when necessary.

- **Measuring your execution time properly.** The MPI_Wtime() or omp_get_wtime() commands will allow you to measure the timing for a particular part of your program (see the skeleton codes). *Make sure to collect 3-5 measurements and take their averages while reporting a performance data point.*

- **Executing your jobs.** You should develop, compile and test your programs on the dev nodes at HPCC. However, on the dev-nodes there will be several other programs running simultaneously, and your measurements will not be accurate. After you make sure that your program is bug-free and executes correctly on the dev-nodes, the way to get good performance data for different programs and various input sizes is to use the interactive or batch execution modes. Please consult HPCC's wiki pages for details (https://wiki.hpcc.msu.edu/display/ITH/Job+Scheduling+by+SLURM). *Note that jobs may wait in the queue to be executed for a few hours on a busy day, thus plan accordingly and do not wait until the last day.*

- **Batch job script.** Batch jobs are convenient, especially if you would like to collect data for several runs (this may still take a few hours to complete, but at least you do not have to sit in front of the computer). Note that you can execute several runs for your programs with different input values in the same job script – this way you can avoid submitting and tracking several jobs. An example job script:

```
#!/bin/bash -login
#SLURM resource allocations

# change to the working directory where your code is located
cd <your batch submission dir>

# call your executable with different no. of processes/threads
srun –n 1 ./decrypter inp1.ecp
srun –n 14 ./decrypter inp1.ecp
srun –n 28 ./decrypter inp1.ecp
# if necessary, list more jobs here…

# all output will be written to the default job stdout stream. #
if you are trying to debug your code, make sure to print into the
# stderr stream, using fprintf(stderr, "…", …); for instance.
# If desired, you can redirect individual results to
# a file, i.e.,
# srun –n 1 ./decrypter inp1.ecp >& decrypt_p1.out
```