CSE/CMSE 822: Parallel Computing
Fall 2017, Homework 5
Due 11:59 pm, Nov 15, 2017

*This homework is 20% of your **homework grade**.*

***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) [50 pts] Parallel Weight Balancing with OpenMP Tasks

Suppose you have a set of weights $w_1,w_2,\dots,w_n$, where each weight is a real number. You are asked to divide the weights into two sets such that the difference of the sum of weights in these two sets is minimum. For example, assume that {12, 23, 45, 99} is the set of given weights. This set can be divided into two sets as {12, 23, 45} and {99}, which minimizes the difference of the sum of weights between them. (99 - (12+23+45) = 19 is the minimum possible difference).

This is an NP-hard problem, so the only way to find the exact solution is to enumerate all possible combinations and choose the one which gives the minimum difference between the two sets. Therefore you need to create all subsets of the given set of weights. The pseudocode for generating subsets using backtracking method is given below (as well as in the skeleton code provided):

```
// S: a vector of integers
// i: index in set S where we are constructing the current subset
GenerateSubset(i, ans, S) {
   if (i == S.size()) {
     // a new subset is identified. update if it is a better solution
     // than what we have so far
     return;
   }
   GenerateSubset(i+1, ans, S); // generate subsets w/out element i
   ans.push_back(S[i]); // add element i to the subset
   GenerateSubset(i+1, ans, S); // generate subsets with element i in them
}
```

i) [25 pts] Using the provided code, first complete the sequential version of the weight balancing program. Then implement an OpenMP parallel version using tasks — which are ideally suited for parallelizing recursive programs. Each recursive call to the GenerateSubset can be denoted as a new OpenMP task.

ii) [25 pts] Reporting Performance

- Experiment with different number of weights such as 15, 20, 25 and 30.
- For your OpenMP parallel version, experiment with different #threads, i.e. 2, 4, 8, 14, 20, 28.
- Plot the speedups for your parallel code with respect to the serial code. Explain your observations.
- Note: The skeleton code provided expects the #weights and #threads to be provided as command line arguments. For the serial version, you still need to provide the number of threads, but it essentially will not have been used.

Hints:

- Keep in mind that there will be 2^n number of subsets for a set of n elements. So you should refrain from saving all subsets, as this will require huge memory space. Instead you can keep a current best solution, and update your solution when you identify a partitioning which is better the current one. For initialization, you can start by setting the best solution to be empty and the global minimum to an arbitrarily large value.
- How can you update your current solution? Assume that the sum of all given weights is SUM. Note that the best solution will have the minimum difference between the sum of its weights and SUM/2. So, for each generated subset, you can simply compute the difference between your generated subset's sum and (SUM/2) and update your solution if it is less than your current best.
- Note that creation of too many tasks may slowdown a task-parallel OpenMP code. OpenMP's task directive allows conditional creation of tasks using the "if" clause. You can use this functionality to improve your implementations performance.

3) [60 pts] Code Breaking

Cybersecurity is an important topic, and 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 has 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 file codebreaker.c, you are given a sequential code breaking algorithm which simply tries all 2^{32} keys in a brute force manner to decrypt a file encrypted by 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.

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 is done at run-time, permitting load balancing to be performed, albeit at the cost of a more complicated solution.
We will only be looking a *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 having each process set up an immediate receive operation prior to starting the loop for trying the keys. Then the process that discovers the key sends individual messages to all others to notify them about program completion. Clearly, all processes must be periodically

(not very frequently though) testing on the immediate receive to see if any other process has had success.

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) using the given dictionary. Two sample input files (inp1.txt and inp2.txt) are provided for your convenience.

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 small number of processes only, and try to decrypt messages encrypted with small key values (i.e. key = 10, 100, 1000).

b) Analyze the scalability of your implementation in **part a** on a single node of the intel16 cluster. Note that as opposed to regular problems, search algorithms can achieve *super-linear*, *linear* or *zero* 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 28 processes on a single intel16 cluster node).

c) Identify and implement a partitioning strategy that overcomes the zero (or sublinear) speedup pitfall of the implementation in **part a**. Test version 2 of your codebreaker.c software on your examples from part b to show that your new partitioning strategy does indeed overcome the zero (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 intel16, 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 (obviously, when the threads created are OpenMP threads).

- Implement an MPI/OpenMP parallel version of the codebreaker.c program that you developed in question 3.
- Analyze the performance of the MPI/OpenMP parallel code with respect to the MPI only parallel version. Do you observe better or worse performance? Explain your observations.

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 "HW5_*yourMSUNetID*.pdf", which contains your answers to the non-implementation questions, and report and interpretation of performance results.
    - All source files used to generate the reported performance results. Make sure to use the exact files names listed below:
        - generate_subset.cpp (for the first problem)
        - codebreaker.c (for problem 2)
        - codebreaker_v2.c (for problem 2)
        - codebreaker_hybrid.c (for the bonus problem)

        These are the default names in the git repository. It is essential that you 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 group 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 the provided Makefile for compilation. Simply type "make" in the directory where the Makefile is located at.

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

    ```
    mpirun –n 4 ./decrypter encrypted_filename
    ```

    For hybrid MPI/OpenMP compilation in the bonus problem, you must provide the –fopenmp flag to the mpicc compiler (see the CFLAGS variable used by mpicc in the Makefile). Also 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.** 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. *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 for the last day of the assignment.*

    i) **Interactive queue.** Suggested options for starting an interactive queue on the **intel16** cluster is as follows:

    ```
    qsub -I -l nodes=1:ppn=28,walltime=00:30:00,feature=intel16 -N myjob
    ```

The options above will allow exclusive access to a node for 30 minutes. If you ask for a long job, your job may get delayed. Note that default memory per job is 750 MBs, which should be plenty for the problems in this assignment. But if you will need more memory, you need to specify it in the job script.

ii) **Batch job script.** Sometimes getting access to a node interactively may take very long. In that case, we recommend you to create a job script with the above options, and submit it to the queue (this may still take a couple hours, but at least you do not have to sit in front of the computer). Note that you can execute several runs of 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
#PBS -l walltime=00:15:00,nodes=1:ppn=28,feature=intel16
#PBS -j oe
#PBS -N subset

# change to the working directory where your code is located
cd ~/cse491-{$USER}/homework/5
cd $PBS_O_WORKDIR

# call your executable with different no. of threads
./subset_seq  15 1
./subset_tasks  15 2
./subset_tasks  15 4
# list more jobs here…
# all output will be written to the default job stdout.
# if desired, you can redirect individual results to
# a file. i.e., ./subset_tasks  15 4 > subset_w15_t4.out
```