HW: B. Parallel Computing

Due Monday, March 22, 2021 (11:59 PM EDT)

Abstract

The objective in this homework is to develop practical skills in parallel computing for computational and data science. The focus is on developing compute-intensive applications on high performance systems based on GPUs, and shared- and distributed-memory parallel architectures.

Contributors

Ignacio Illorente, David Sondak, Zhiying Xu, Dylan Randle, Hayoun Oh, Zijie Zhao, Charles Liu, Matthew Holman, Zudi Lin, Nicholas Stern and Kar-Tong Tan

Guidelines

- **The files needed to do the exercises are available for download from the course Canvas**.
- **AWS**
    - First you should have followed the Guide "First Access to AWS". It is assumed you already have an AWS account and a key pair, and you are familiar with the AWS EC2 environment.

If you are using AWS, we strongly recommend you use the same instance type for all the experiments (**t2.2xlarge**) so you can compare the performance results achieved with the different programming models and platforms

- **Performance Optimization**
    - For the performance optimization exercises you can use Cannon, a VM instance on AWS, or

your own Mac OS or Linux system with gcc or a commercial compiler.

- If you are using AWS, we recommend you use a **t2.2xlarge** instance and first follow the Guide "Performance Optimization and OpenMP on AWS" in order to set-up gcc and get familiar with its optimization flags. **Ensure you stop the instance after the homework**.
- The codes include calls to timing functions to write on screen the **elapsed execution time** of their compute part.

- **Accelerated Computing**
  - For the GPU programming exercises you can use a VM instance on AWS.
  - If you are using AWS, we recommend you use a **g3.4xlarge** instance and first follow the Guide "OpenACC on AWS" in order to install CUDA and the PGI tools, and get familiar with its OpenACC support. Ensure you stop the instance after the homework (**GPU-based Accelerated Computing instances are very expensive!**).
  - The codes include calls to timing functions to write on screen the **elapsed execution time** of their compute part. If you set the environment variable `PGI_ACC_TIME` to 1, the **GPU runtime** summarizes the time taken for data movement between the host and GPU, and computation on the GPU.
- **Shared-Memory Parallel Programming**
  - For the shared-memory parallel programming exercises you can use Cannon, a VM instance on AWS, or your own Mac OS or Linux system with gcc or a commercial compiler.
  - If you are using AWS, we recommend you use a **t2.2xlarge** instance and first follow the Guide "Performance Optimization and OpenMP on AWS" in order to set-up gcc and get familiar with its OpenMP support and automatic parallelization functionality. Ensure you stop the instance after the homework.
  - The codes include calls to timing functions to write on screen the **elapsed execution time** of their compute part.
- **Distributed-Memory Parallel Programming**
  - For the distributed-memory parallel programming exercises you can use Cannon or an MPI cluster on AWS.
  - Install a local version of MPI, by following the first section of the Guide "MPI on AWS", to test your programs on your local computer before running on a MPI cluster. Programs written and tested locally can be run on a cluster with just a few additional steps.
  - If you are using MPI on an AWS VM cluster, we recommend you use a cluster with at least two **t2.2xlarge** instances and first follow the Guide "MPI on AWS" in order to set-up the cluster and install the MPICH library. Ensure you stop the instance after the homework.
- **Submission**

Any computing experiment that cannot be replicated cannot be considered as a valid submission

- **Your performance results should be replicable[1] results. Sometimes the equivalent term Repeatability is used for this experimental property, so you should provide ALL the information of the system and the environment needed to repeat your tests.** At the beginning of each PDF answering each specific exercise you must provide, at least:
  - Specs of the system (model, number of CPUs, number of cores per cpu, clock rate, cache memory, and main memory)
  - If using a cluster, number of systems and specs of networking (latency and bandwidth)
  - Operating System (Linux distro and kernel version)
  - Compiler (name and version, and flags)
  - Libraries (name and version)
  - Any other configuration needed to replicate the experiments and achieve exactly the same result and performance?
- Upload on **Canvas** the files specified in each assignment.
- The grade on this assignment is **15% (150 points) of the final grade.**

Table of Contents

[1] The attribute **Replicability** describes the ability to repeat a computer based experiment and to come to the same results and performance.

1. Python Multiprocessing Module (20 points)

One of the simplest ways to do things in parallel in Python is through the multiprocessing module. Here, you'll reason through two simple examples that show the oddities of parallelism.

1.1. Count to Ten (10 points)

The multiprocessing module allows a pool of processes to complete a batch of jobs in parallel. The script `P11.py` demonstrates this functionality. If you run the code a number of times, you may see some unexpected results.

1. Explain these results.
2. How could this affect how we program in parallel?
3. Describe a scenario where this would be important.

Submission
- `P11.pdf`: Answers and discussion

1.2. How Much Faster? (10 points)

In `P12.py`, the `burnTime` has been changed to simply sleep for a parameterized amount of time.

1. Using the `Pool.map` functionality, call `burnTime` 16 times in parallel using 4 processes and 16 times in serial using a standard loop. Use `time.time()` to determine how many seconds each takes and use various sleep times (ranging from 10^{-6} to 100 seconds) for each timing.
2. Plot the ratio of serial to parallel execution time against sleep time.
3. Try to explain the trend you observe.
4. Is it possible that a parallel program could take longer than it's serial version? If so, under what conditions does this occur?

Submission
- `P12.pdf`: Answers (including the plot) and discussion
- `P12.py`: Source code

2. Performance Optimization (25 points)

2.1. Optimization of Basic Codes (10 points)

The aim of this exercise is to compile two codes with an optimization flag and to try different optimization techniques.

- `dotprod_serial.c` performs a simple dot product calculation

  Use the following to compile `dotprod_serial.c`:

  ```
  gcc -DUSE_CLOCK dotprod_serial.c timing.c -o dotprod_serial
  ```

- `jacobi1d.c` is a serial implementation of a 1D Poisson problem using Jacobi iteration. The basic syntax of the serial executable is

  ```
  jacobi1d [ncells] [nsteps] [fname]
  ```

  The `ncells` and `nsteps` arguments define the number of cells in the Poisson discretization and the number of steps in the Jacobi iteration, respectively. The argument `fname` is the name of a file to which the final solution vector is written. Each row in this file consists of the coordinate for a point in the discretization and the corresponding solution value; for example, after 1000 steps on a 100000-cell discretization, we have

  ```
  0 0
  1e-05 5e-13
  2e-05 1e-12
  ...
  ```

  If `ncells` or `nsteps` are omitted, the default value of 100 is used. If the file name `fname` is omitted, the program simply prints out the timings, and does not save the solution.

  Use the following command to compile `jacobi1d`

  ```
  gcc -DUSE_CLOCK jacobi1d.c timing.c -o jacobi1d
  ```

If you use any aggressive optimization, you should check the correctness of your implementation by comparing the solution output files to the solution output files from the code with no optimization.

Submission

- `P21.pdf`: Report the improvements in elapsed execution time when using **separately** `-O0`, `-O1`, `-O2`, `-O3`, `-Os` and `-Ofast` flags, and 2x and 4x loop unrolling technique on a single core for both codes. Results should be presented in tabular form. For example, each column should correspond to a different optimization flag or technique and each row will be the code being run (either `jacobi1d.c` or `dotprod_serial.c`). The entries of the table should be the run times returned by the timing function provided in `timing.c`. For `jacobi1d.c`, use 10^8 cells and 100

steps.
- `P21a.c`: Source code for the version of `dotprod_serial.c` with one level of loop unrolling
- `P21b.c`: Source code for the version of `jacobi1d.c` with loop unrolling

Note: we assume n is even.

2.2. Optimization of Matrix Multiplication (15 points)

This exercise is intended to show how the reuse of data that has been loaded into cache by previous instructions can save time and thus increase the performance of your code.

`seq_mm.c` is a simple code that performs a 1,500 by 1,500 matrix multiplication. Develop a new version of the code that uses blocking to improve its temporal locality.

Use the following command to compile `seq_mm`

```
gcc -DUSE_CLOCK seq_mm.c timing.c -o seq_mm
```

Submission
- `P22.pdf`: Report with replicability information (see Submission note above), the improvements in elapsed execution time when using **separately** `-O0`, `-O3`, loop unrolling, blocking and unrolling/blocking. Please report your results in tabular form with columns corresponding to optimization flags or techniques as appropriate
- `P22.c`: Source code for the combined version of the code with loop unrolling and blocking. In your source code, please add comments to highlight where you have applied unrolling and blocking

Note: for "loop unrolling/blocking", unroll the blocking version OR unroll the innermost layer for x2 or x4.

3. Accelerated Computing (35 points)

3.1. Acceleration of Basic Code (20 points)

The goal of this exercise is to familiarize you with OpenACC and make your first accelerated code. Develop an OpenACC version of the jacobi code used in section 2.1. Your task is to parallelize the `for` loops in the function `jacobi` using OpenACC. You should do this by copying over the corresponding code from `jacobi1d.c` and putting OpenACC `#pragma`s in appropriate places. You should develop two versions:

A. One with a copy of data in each iteration and
B. One with a single copy of data.

1. Run both codes with `ncells` equal to 10^k, for k = 6, 7, and 8, and 100 steps. Plot the execution times of both codes and the sequential execution (with no acceleration) as a function of k. Consider the elapsed time provided by the application and the GPU runtime provided by the driver.
2. Run the second, enhanced version of the code, with `ncells` of 10^8, 100 steps and multiple values (32, 64, 128, 256, 512, 1024) for the number of threads per block (vector length). Which value for the number of threads is faster and why?

Submission
- `P31.pdf`: Answer the questions and discuss the implementations
- `P31a.c`: First version source code
- `P31b.c`: Second version source code, with vector lengths specified

3.2. Acceleration of Matrix Multiplication (15 points)

Develop an OpenACC version of `seq_mm.c` and compare it with the optimized sequential execution of section 2.2.

Submission
- `P32.pdf`: Discuss the implementation, its performance and the default block grid created by the system
- `P32.c`: Source code

4. Shared-Memory Parallel Processing (35 points)

4.1. Parallelization of Basic Code (20 points)

The goal of this exercise[2] is to familiarize you with OpenMP and make your first parallel code. Develop an OpenMP version of the jacobi code used in section 2.1 by completing `jacobi1d_omp.c`. Your task is to parallelize the `for` loops in the function jacobi using OpenMP. You should do this by copying over the corresponding code from `jacobi1d.c` and putting OpenMP `#pragma`s in appropriate places. You should start by putting a `#pragma parallel` inside the outer loop in order to start a parallel section, and then parallelize the inner loops with `#pragma` for directives.

You should do two experiments:

1. Run your parallel codes on a system with at least 4 cores with `ncells` equal to 10^k, for k = 5, 6, 7, and 8, and 100 steps. Plot the ratio of your parallel run times to the serial code run time as a function of k. Calculate the Speed-up and explain why it increases or decreases with the problem size. Use -Ofast to compile the code in all cases.
2. Run your codes with `ncells` of 10^8 and and 100 steps on 2-8 cores and produce a speedup plot. Explain why it increases or decreases with the number of cores.

You should check the correctness of your implementation by comparing the solution output files to the solution output files from the serial code.

Submission

- `P41.pdf`: Explain your results in terms of a simple performance model that takes into account the time spent on both computation (which is proportional to the number of points per processor) and communication/synchronization.
    - Hint: No need to present a mathematical model. Just describe using words. You can think about concepts such as granularity and synchronization overhead.
- `P41.c`: Complete source code

4.2. Scheduling Policies (15 points)

The goal of this exercise is to illustrate the impact of the scheduling algorithm in performance. The code `mandelbrot_omp.c` generates an ASCII Portable Pixel Map (PPM) image of the Mandelbrot set using OpenMP.

Run the parallel code and take the execution time for a growing number of threads. Record the timing in a table. Try with the three scheduling algorithms. To change the schedule, you can either change the environment variable with `export OMP_SCHEDULE=type` where `type` can be any of `static`, `dynamic`, or `guided`. Alternatively, you can change the schedule in the source code as `omp parallel for schedule(type)`.

Note: if you change the environment, then you don't have to change anything inside the code.

[2] http://www.cs.cornell.edu/~bindel/class/cs5220-f11/hw2.html

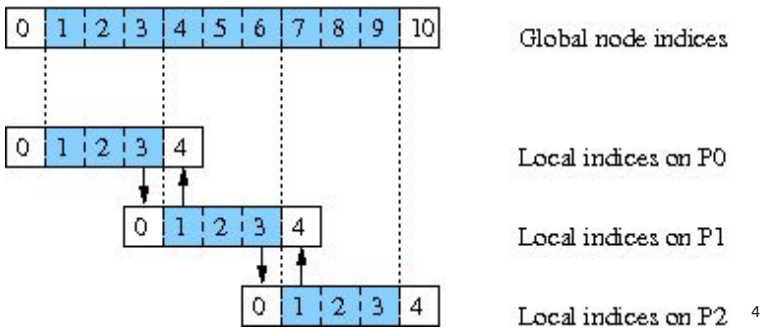
Submission

- `P42.pdf`: with replicability information (see Submission note above), Explain your results, what is the scheduling that provides the best performance? What is the reason for that?
- `P42.c`: Complete source code

5. Distributed-Memory Parallel Processing (35 points)

5.1. Parallelization of Basic Code (20 points)

The goal of this exercise[3] is to familiarize you with MPI and make your first parallel code. Develop a MPI version of the jacobi code used in section 1.1 by completing `jacobi1d_mpi.c`. The structure of `jacobi1d_mpi.c` is somewhat more complicated. In this code, no processor holds the entire solution vector; instead, each processor has a local piece of the vector, and these pieces overlap slightly at the ends. No variable is directly updated by more than one processor; the overlap is simply to accommodate *ghost cells*. As a concrete example, suppose we discretize the interval [0,1] into 10 cells of size 0.1. Then there are 11 points in our mesh, including the endpoints subject to boundary conditions. If we partition these points among three processors, we have the following picture:



[4]

In this picture, the active variables are colored blue and the white boxes at the end correspond to storage for ghost cells or boundary data. For example, consider the storage on the second processor (P1). Entries 1, 2, and 3 in this processor's local vector correspond to entries 4, 5, and 6 in the global indexing of variables. However, in order to update these entries, P1 must know about the entries 3 and 7 in the global vector. P1 has storage for these variables in 0 and 4 of its local vector; but rather than computing these variables, P1 must be sent the information from the neighboring processor. Thus, at each step,

- P0 sends entry 3 of its local vector to P1 and receives entry 4;
- P1 sends entry 1 of its local vector to P0 and receives entry 0;
- P1 sends entry 3 of its local vector to P2 and receives entry 4;
- P2 sends entry 1 of its local vector to P1 and receives entry 0

[3] http://www.cs.cornell.edu/~bindel/class/cs5220-f11/hw2.html

[4] In fact, the provided skeleton code splits global indices in a particular way and - as some of you might have noticed - the picture for your execution of *n=10* will look more like this: https://harvard-iacs.github.io/2021-CS205/homeworks/HWB/guide/indices_table.png.
This does not result in a perfectly even distribution, but will be reasonably efficient when the size of n is sufficiently large. Your portion of the code will also not be affected, so you can disregard this particularity.

In `jacobi1d_mpi.c`, the code that implements this exchange of ghost cell information is left out. You should fill this code in. Your code should be correct for any number of processors, not just two or three. The solution uses `MPI_Send`, and `MPI_Recv` in two phases: one that sends data to the processor to the left, and one that sends data to the processor to the right. However, if you would prefer to use a different organization, you are welcome to do so.

You should do the following experiments:

1. Run your code with 1 to 4 tasks on one node with `ncells` equal to 10^8 and 100 steps.
2. Run your code with 8 tasks on two nodes with `ncells` equal to 10^k, for k = 5, 6, 7, and 8 and 100 steps. Plot the ratio of your parallel run times to the serial code run time as a function of k. Calculate the Speed-up and explain why it increases or decreases with the problem size. Make sure to change your instance type to t2.2xlarge when running k = 8.
3. Run your codes with `ncells` of 100,000,000 and 100 steps with 2-8 tasks on two nodes and produce a speedup plot. Explain why it increases or decreases with the number of cores.
4. Compare the performance results with those obtained with OpenMP.

You should check the correctness of your implementation by comparing the solution output files to the solution output files from the serial code.

Submission

- `P51.pdf`: with replicability information (see Submission note above), Explain your results in terms of a simple performance model that takes into account the time spent on both computation (which is proportional to the number of points per processor) and communication/synchronization
- `P51.c`: Complete source code. In your source code, please insert comments to explain what you are using each of the MPI functions for

5.2. Hybrid Parallel Processing (15 points)

OpenMP by itself is constrained to a single node. In the world of very large scale infrastructures, the motivation for using OpenMP is possible performance gains when combined with MPI. Develop a hybrid version of `jacobi1d_mpi.c` by including OpenMP directives to parallelize the two main execution loops. Include one `#pragma` for each loop. Compile and execute the code on multiple nodes and threads per node. Compare execution time in these three scenarios:

- SMP Nodes
  - Single MPI task launched per node
  - Parallel Threads share all node memory
  - For example: 4 threads/task and 1 task/node on system based on nodes with one 4-core CPU
- SMP Sockets (CPUs)
  - Single MPI task launched on each socket (CPU)
  - Parallel Thread set shares socket (CPU) memory, e.g. 4 threads/socket
  - For example: 2 threads/task and 2 tasks/node on system based on nodes with two 2-core CPUs
- No Shared Memory (all MPI)
  - Each core on a node is assigned an MPI task
  - For example: 1 thread/task and 4 tasks/node on system based on nodes with 4 cores

If you are using a 2-node MPI cluster on AWS you can emulate the three scenarios by running 4 threads/task and 1 task/node in the SMP Nodes mode; 2 threads/task and 2 tasks/node in the SMP Sockets (CPUs) mode, and 1 thread/task and 4 tasks/node in the No Shared Memory mode.

Submission

- `P52.pdf`: with replicability information (see Submission note above), Discussion of the results
- `P52.c`: Complete source code. Please insert comments in your source code to explain the type of parallelization you are applying