Machine Problem 3

CS 484 - Parallel Programming

Due: April 19, 2024 @ 23:59

(Typeset on : 2024/03/23 at 19:09:54)

Introduction

Learning Goals

You will learn the following:

- Communication patterns of parallel 2D Van der Waals gas simulator
- Using MPI, and AMPI for distributed memory execution

Please read the entire document before beginning any part of the assignment, as some parts are interdependent.

Assignment Tasks

The basic workflow of this assignment is as follows (there are more details in the relevant sections):

- Clone your turnin repo to Campus Cluster [1]
  Your repo should be visible in a web browser and clonable with git from
  `https://gitlab.engr.illinois.edu/sp24-cs484/turnin/YourNETID/mp3.git` .
- You may need to iterate:
  - Implement the algorithms / code for the various programming tasks.
  - Build and test.
  - Check in and push your complete, correct code.
  - Benchmark on Campus Cluster as a batch job (`sbatch scripts/batch_script.slurm`).
    See `README_CAMPUS_CLUSTER.md` for details.
- Check in any benchmarking results that you wish to and final versions of code files.
- Write and check in your writeup.

1 Part I: MPI

You will implement the communication functions in `part1/solution.h` and `part1/solution.cpp`. You may modify the classes in these files to add member variables/functions or even create other helper classes, etc. Although you are allowed to create or alter other files and use them for debugging/testing purposes, they will be discarded at grading time.

1.1 Implementation

The MPI code revolves around a class named `MPISimulationBlock` defined in `part1/solution.h`, which inherits from `SimulationBlock` defined in `common/simblock.h`. The parent class `SimulationBlock` contains most of the program parameters as well as particle data, which can be accessed from `MPISimulationBlock`'s member methods. Each MPI rank contains one `MPISimulationBlock` that represents a block of the entire simulation grid. Remember that with MPI, the decomposition (i.e. number of blocks/ranks used for the simulation) is dependent on the number of CPU cores.

You may add code to the constructor and destructor bodies if want (not required). You should implement the following functions:

[1] You are free to use the Docker container uiuccs484parallelprog/cs484_student , but we will provide no support for Docker itself.

- `MPISimulationBlock::exchange_particles()`

  When this function is called, any particle outside the current block's bounds must be moved to the appropriate adjacent block. That is, it must be removed from this block's `SimulationBlock::all_particles` array (using the provided `SimulationBlock::remove_particle(int)` function) and added to one and only one appropriate recipient's `all_particles` array (either via `SimulationBlock::add_particle(phys_particle_t)` or by directly placing it in the `all_particles` array and updating `N_particles`).

  You can determine which direction a particle needs to move by calling `check_migrant_direction(particle)`. The return value of this function is an `int`, which is one of the following:

  `SimulationBlock::DIR_SELF, SimulationBlock::DIR_N, SimulationBlock::DIR_S,`

  `SimulationBlock::DIR_E, SimulationBlock::DIR_W, SimulationBlock::DIR_NE,`

  `SimulationBlock::DIR_NW, SimulationBlock::DIR_SE, SimulationBlock::DIR_SW`

  `SELF` means the particle does not need to move to another block. `N` means north, `S` means south, `NW` means northwest, and so on. You may use the provided macro `DIR_EQ` (provided in `common/simblock.h`) to check the direction:

  e.g. `DIR_EQ(SimulationBlock::DIR_N, direction)` will return true (i.e. 1) if the particle should migrate to the north neighbor.

  For the particle exchange, you would need to use MPI communication calls (`MPI_send, MPI_recv, etc.`) and proper synchronization (if necessary). Note that the receiving rank will need to know how many particles it is going to receive before posting a receive MPI call, since you want to avoid sending particles one by one over the network.

- `MPISimulationBlock::communicate_ghosts()`

  Any time a particle from `SimulationBlock::all_particles` is within a certain distance of the current block's edge, it must be communicated to the appropriate adjacent, or corner-touching `SimulationBlock` and placed in that block's `SimulationBlock::all_ghosts` array. `SimulationBlock::N_ghosts` must be set to the total number of ghost particles received in this iteration. Communicating ghost particles is very similar to exchanging particles, except that a single particle may be sent as a ghost to several adjacent blocks, and particles that are sent are *not* removed from the local `all_particles`. You can determine which direction(s) a particle needs to be sent by calling `check_ghost_direction(particle)`. Because a particle may be sent to multiple neighbors unlike in `exchange_particles()`, you should use the provided `DIR_HAS` macro:

  e.g. `DIR_HAS(SimulationBlock::DIR_N, direction)` will return true if north is one of the directions that the particle needs to be sent to.

  You should use MPI communication calls to send and receive the ghosts, similar to `exchange_particles()`.

- `MPISimulationBlock::init_communication()`
- `MPISimulationBlock::finalize_communication()`

  These functions will be called by the main program before and after the simulation runs. You may use them to do whatever setup and teardown you wish, but do not call `MPI_Init()` or `MPI_Finalize()`, which our main program will do. Mostly this is for setting up/tearing down whatever variables you will need for communication. Try not to leak any memory, we may choose to test your code under **valgrind** or another memory debugger.

1.2 Compilation and Testing

1. Create a new `build` directory.
2. In `build`, run `cmake <path_to_mp3_dir>`. This will go through the system configurations and generate a Makefile.
3. Run `make`. This will compile binaries for all parts of the assignment (`bin/part1`, `bin/part2`, `bin/part3`).

This compilation process is identical for all parts of the assignment.

To test if your MPI program works, run `bin/part1 -N 100 -i 1`. This will run the simulation for 100 iterations, outputting the iteration value every iteration.

2 Part II: AMPI

2.1 Implementation

For this section, you do not need to modify/add any code, but please do read the code in `part1/main.cpp` that is wrapped with `#ifdef AMPI`. These code blocks demonstrate how load balancing can be invoked with AMPI, and will be included in the AMPI version of the program.

You do need to benchmark this code, as explained in the last section.

2.2 Testing

Run `bin/part2 +vp 4 -N 100 -i 1 +balancer GreedyRefine +isomalloc_sync`. This will run the AMPI program with 4 virtual ranks, with the GreedyRefine load balancing strategy.

3 Benchmarking

We have provide you with a batch script that will run both parts of the assignment (MPI and AMPI; Charm will be excluded this term). As always, you should run this on the campus cluster. It will vary the number of utilized CPU cores from 1 to 36, running on at most 2 physical nodes (each node has 20 CPU cores) with a fixed decomposition. The results will be stored in `writeup/benchmark_*.txt`, where `*` is the one of `mpi` and `ampi`. You should plot these results and evaluate the performance in your writeup. More specifically, explain how the performance for each version of the simulation code scales with the number of CPU cores.

4 Questions

As part of this assignment, you will need to answer multiple questions about your experiments. The repo contains a file (**mp3.answers**) to put your answers into. Each line of the file contains numbers corresponding to each question. Put your answers on corresponding lines. Do not include any extra symbols (e.g. no dots at the end of the line). **IMPORTANT**: we will be using automated tools to grade your work, so make sure you follow the described format. The answers to these questions may require multiple runs of the experiments, so start early.

4.1 Question 1

In terms of runtime, does MPI or AMPI perform better on 4 processes? Answer **with either MPI or AMPI**.

4.2 Question 2

In terms of runtime, does MPI or AMPI perform better on 36 processes? Answer **with either MPI or AMPI**.

4.3 Question 3

Which scales better, MPI or AMPI? Answer **with either MPI or AMPI**.

4.4 Question 4

What is the most probable cause of one scaling better than the other? Answer **with either A, B, or C**
A) MPI scales better since for larger number of processes the frequency of AMPI migration is too high.
B) MPI scales better since AMPI migration synchronizes processes.
C) AMPI scales better due to the effectiveness of load balancing.

5 Submission

You must commit at least the following files. These files, and only these files, will be copied into a fresh repo, compiled (if needed), and tested at grading time.

- `part1/solution.h`, `part1/solution.cpp`
- `mp3.answers`

Nothing prevents you from altering or adding any other file you like to help your debugging or to do additional experiments. This includes the benchmark code. (Which will just be reverted anyway.)

It goes without saying, however, that any attempt to subvert our grading system through self-modifying code, linkage shenanigans, etc. in the above files will be caught and dealt with harshly. Fortunately, it is absolutely impossible to do any of these things unaware or by accident, so relax and enjoy the assignment.

6 Grading Rubric

6.1 MPI

Speedup, 4 processes (weight 1 / 3):

- `Speedup` $\geq$ `2.40: 1 point`
- `Speedup` $\geq$ `2: 0.5 points`

Speedup, 9 processes (weight 1 / 3):

- `Speedup` $\geq$ `5.5: 1 point`
- `Speedup` $\geq$ `4.6: 0.5 points`

Speedup, 16 processes (weight 1 / 3):

- `Speedup` $\geq$ `11: 1 point`
- `Speedup` $\geq$ `10: 0.5 points`

Total weight for MPI: 0.6

6.2 AMPI

Speedup, 4 processes (weight 0.5):

- `Speedup` $\geq$ `1.55: 1 point`
- `Speedup` $\geq$ `1.33: 0.5 points`

Speedup, 9 processes (weight 0.5):

- `Speedup` $\geq$ `3: 1 point`
- `Speedup` $\geq$ `2.4: 0.5 points`

Total weight for AMPI: 0.2

6.3 Answered questions

1 point for each question.
Total weight for answered questions: 0.2