P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

HW 4 - Asynchronous Communication and Hybrid MPI & OPENMP

Issued: April 11, 2022
Due Date: May 03, 2022, 10:00am

Task 1: Point-to-Point Communication Bandwidth (10 Points)

The fundamental communication mechanism in MPI are point-to-point messages. In this task, our goal is to obtain insight on what communication bandwidth we can achieve when exchanging messages between two MPI ranks (point-to-point). The data we collect will help us understand and show limitations of the underlying network hardware on our target system.

a) (5 points) Write a small MPI program to measure the bandwidth of point-to-point messages. Your program should output the measured bandwidth for message sizes starting from 2^0 byte up to 2^{24} byte. To average out noise when sending small messages, consider to design your code such that you measure a given message size multiple times in an iterative loop. To determine the bandwidth, you must compute the ratio

$$\text{communication bandwidth} = \frac{\text{total bytes transferred}}{\text{total time for transfer}}.$$

You may use `MPI_Wtime` or another suitable method to measure time. Since you are interested in a point-to-point communication, your program must require exactly two ranks.

b) (5 points) Perform two measurements on `euler.ethz.ch`:

1. Use *a single* node and map the two MPI ranks on the *same socket*.
2. Use *two* nodes and map the two ranks on *one node* each.

For each of these two configurations, measure the bandwidth as a function of the message size and visualize them in *one* log-log plot with message size on the abscissa and measured bandwidth on the ordinate. Comment your findings in a few sentences.

Hint: Have a look at the `--map-by` argument in the `mpirun` manpage. To verify your mapping, see `--report-bindings` in the same manpage. When you submit your jobs make sure you use `bsub -R fullnode` to ensure that you get full nodes for yourself. This will minimize measurement errors due to resource contention of other users on the same node.

Task 2: Compute/Transfer overlap (30 Points)

Consider an application that performs smoothing sweeps on scalar structured 3D data u using a Laplacian smoothing kernel given by

$$u_{i,j,k}^{(m+1)} = \frac{1}{12}\big(u_{i-1,j,k}^{(m)} + u_{i+1,j,k}^{(m)} + u_{i,j-1,k}^{(m)} + u_{i,j+1,k}^{(m)} + u_{i,j,k-1}^{(m)} + u_{i,j,k+1}^{(m)} + 6u_{i,j,k}^{(m)}\big), \qquad (1)$$

where $u_{i,j,k}^{(m)}$ denotes the value of u at indices i, j, k and smooth sweep m. The smoothed data is simply a weighted average of the neighboring data points. In the following subtasks, you will implement a distributed version of the serial data smoother implemented in `LaplacianSmoother.h` and `LaplacianSmoother.cpp` with main application in `main.cpp`. The distributed implementation shall use asynchronous MPI communication of ghost cell values with an MPI virtual topology and derived MPI datatypes. The goal is to analyze the performance of our algorithm with respect to latency hiding by compute/transfer overlap of asynchronous communication.

a) (10 points) Complete the code for the constructor in `LaplacianSmootherMPI.cpp`. You need to take care of three tasks:

   1. Define a virtual MPI topology for a Cartesian process layout. You may want to determine the six neighbor ranks you need to communicate with when you exchange the ghost cells.
   2. Define derived MPI datatypes that help you to communicate the ghost cells with your neighbors. Note that you can use the ghost buffers directly to receive data. You can access the data $u_{i,j,k}^{(m)}$ with `operator()(i,j,k)`. The address of the first ghost cell on the low end of the `i`-dimension is given by `&operator()(-1,0,0)`, for example.
   3. Correctly initialize the data based on the global indices i, j, k of each point (see also skeleton code).

b) (10 points) The main `sweep` method (performs one smooth step) is defined in the sequential implementation `LaplacianSmoother.cpp` and consists of four parts:

   1. Initiate asynchronous communication of ghost cells with neighboring ranks
   2. Perform smoothing operation on inner domain that is not affected by communication
   3. Synchronize communication
   4. Perform smoothing operation on boundary domain using the received ghost cells

   In this task, you implement the first and third item. The other two parts are identical to the sequential code and are reused.

   Implement these missing parts in `LaplacianSmootherMPI.cpp` Make use of the data structures that you have defined in the previous task. To check that your implementation produces the same result as the sequential version, compute the check sum in the 'report' function (see skeleton code).

c) (10 points) Implement a profiling report similar to the output of the sequential code. In your MPI code, compute the minimum, maximum and average time over all ranks for the total time spent in `sweep`, `comm_`, `smooth_inner_`, `sync_` and `smooth_boundary_`. You can see the associated time accumulators in `src/LaplacianSmoother.cpp:22`. These time measurements are the accumulated time over `sweep_count_` invocations of the `sweep` method. Write this code in `src/LaplacianSmootherMPI.cpp:38`.

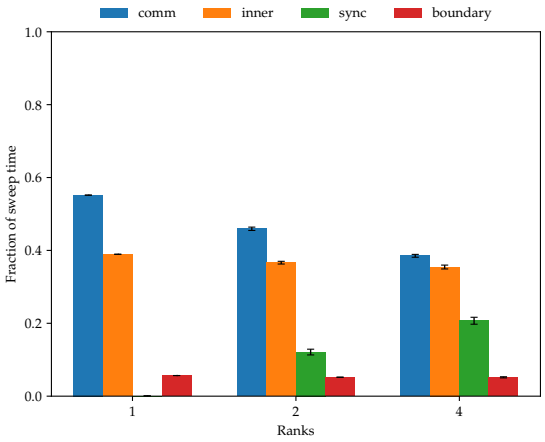

Figure 1: Example bar plot for the fraction of time per sweep spent in computation and communication.

Run your MPI code on `euler.ethz.ch` using 1, 2 and 4 ranks with 2 nodes (use `bsub -R fullnode` when you submit the jobs). Map your ranks onto CPU sockets and use 12 OpenMP threads per rank. Similar to question 1, you can use `--map-by` to accomplish this and `--report-bindings` to check your binding.

Perform this experiment for a problem size of 128^3 and 1024^3 cells per rank. For example:

```
mpirun -n 1 ./mainMPI 128 128 128 1 1 1
mpirun -n 2 ./mainMPI 128 128 128 1 1 2
mpirun -n 4 ./mainMPI 128 128 128 1 2 2
```

would run the code with 128^3 cells per rank involving 1, 2 and 4 ranks, respectively. For each of the two experiments, create a bar plot similar to fig. 1 with the data extracted from your profiling report (use the data averaged over ranks, the standard deviation is optional). Comment on your observations regarding how much time is spent in the four subtasks (`comm_`, `smooth_inner_`, `sync_` and `smooth_boundary_`) for the two cases. Can you overlap computation with communication?

Task 3: 2D Wave Equation - Hybrid MPI & OPENMP (40 points)

You are given a skeleton code that solves the following 2D wave equation

$$\frac{\partial^2 u}{\partial t^2} - c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0,$$
$$u(0,x,y) = 1 - \sin(\pi r) \cdot \exp(-r) \ , \ \frac{\partial u}{\partial t}(0,x,y) = 0,$$
$$u(t,0,y) = u(t,1,y) \ , \ u(t,x,0) = u(t,x,1) \quad \forall x, y \in [0,1] \tag{2}$$

by using central finite differences in space and explicit integration in time

$$u_{i,j}^{n+1} = 2u_{i,j}^n + c^2 \frac{\delta t^2}{h^2} (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n), \tag{3}$$

where $u_{i,j}^n = u(n\,\delta t, (i+\frac{1}{2})h, (j+\frac{1}{2})h)$. The main output of the solver is the energy functional of the wave equation, given by

$$E(t) = \frac{1}{2} \int_\Omega \left(\frac{\partial u(\mathbf{x},t)}{\partial t} \right)^2 + c^2 \left\| \nabla_{\mathbf{x}} u(\mathbf{x},t) \right\|^2 d\mathbf{x}. \tag{4}$$

The skeleton code is already parallelized with MPI. It uses a Cartesian topology and custom MPI datatypes for ghost cell exchange among different processes. Your task is to further parallelize this code with OPENMP.

a) (5 points) Before adding any OPENMP directives, overlap communication of ghost cells and computation in `Equation2D::run`. Inner cells should be updated first, while waiting for send and receive requests to complete. Then, cells at the boundary of each process need to be updated last.

b) (1 point) We would like to allow different OPENMP threads to make MPI calls, without any restrictions. To do so, change the MPI initialization and use a correct thread safety mode in `main.cpp`.

c) (19 points) Parallelize `Equation2D::run` with OPENMP. You should do this by opening **only one** parallel section. You will need to use `pragma omp barrier` to enforce thread synchronization when needed. For this subquestion, create a `pragma omp master` region and place the functions `Equation2D::derivedFunctionCalls` and `Equation2D::saveGrid` in that region. You do not need to parallelize those with OPENMP yet.

d) (10 points) Place and function `Equation2D::derivedFunctionCalls` outside of the `#pragma omp master` region and parallelize it with OPENMP. Then, check correctness of your hybrid code on Euler. Run it with different configurations of threads and MPI processes and verify that the computed energy functional is the same for all. Very small differences in the results are acceptable and should be attributed to how reductions are performed with multiple OPENMP threads and MPI ranks (the order of operations is not always the same and round-off errors may differ when running with different number of threads and/or ranks). You can also plot your results with the `plot.py` script.

e) (5 points) Run your code with pure MPI (and only one OPENMP thread). Then, use the same number of cores but rerun the code with MPI and more than one OPENMP thread.

For example, run the code first with 48 MPI processes and one thread and then run it with 4 processes and 12 threads. Which one is faster and why do you think this is the case? When performing this benchmark, comment out calls to the `saveGrid` function, as this was not parallelized with OPENMP in this homework.

Guidelines for reports submissions:

- Submit a zip file of your solution via Moodle until May 03, 2021, 10:00am.
- **Do not submit** binary files or build directories
- There are 80 available points in this homework. To get a grade of $6/6$ you need to collect 60 points. Collecting 50 points results in a grade of $5/6$ and so on.