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

HW 4 - Asynchronous Communication and Compute/Transfer Overlap

Issued: April 19, 2021
Due Date: May 03, 2021, 10:00am

Task 1: Point-to-Point Communication Bandwidth (30 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) (20 points) Write a small MPI program to measure the bandwidth of unidirectional 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) (10 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 (60 Points)

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

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

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. The particular application for this task is of secondary importance. Any application that requires neighboring data will exhibit similar characteristics like the smoothing kernel above.

In the following subtasks, you will implement a distributed version of the serial data smoother implemented in `include/LaplacianSmoother.h` and `src/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) (warm-up) We are going to use the Meson build system (https://mesonbuild.com/) to compile our code for this exercise. Especially for larger projects, it is essential that you have a reliable build system that helps you compile, link and test your code. You may be familiar with the classic `make` utility and `meson` is a modern alternative (similar to `cmake`). Unlike `cmake`, `meson` is implemented in `python` and allows to write build definitions in a `python`-like manner. To install `meson` on your system, simply fetch it from PyPI (https://pypi.org/):

```
python3 -m pip install meson ninja
```

Study the main build definitions for this exercise written in `meson.build`. In `meson`, code is built out-of-source in a separate build tree. To setup a build directory, execute the following command:

```
meson setup build_debug
```

This needs to be done only once and will configure your build. By default, debug flags are set automatically, if you want to compile optimized code you can use the `--buildtype=release` argument. The build directory will be named `build_debug`, you can give it any name you like. Once you have setup your project, you can compile your code with

```
meson compile -C build_debug
```

or you could invoke `ninja` (similar to `make`) when you are inside the `build_debug` directory. Note that you can have several build directories (e.g. one configured for debug and another configured with optimization flags) at the same time. Invoking

```
meson compile -C <build_directory>
```

will compile your code for the corresponding setup.

b) (5 points) Add a new target (`executable`) in the `meson.build` file that will compile the `mainMPI` executable. When you are done, create a build directory (if you have not done already, see the warm-up above) and compile your targets with

```
meson compile -C <build_directory>
```

You can execute your code by changing the directory into `build_directory` and run

```
./main    # execute the sequential code; generates referenceSEQ.bin
./mainMPI # execute the MPI code; generates referenceMPI.bin
```

This will execute the sequential and the MPI code. You can verify they they generate the same result by running

```
diff referenceMPI.bin referenceSEQ.bin
Binary files referenceMPI.bin and referenceSEQ.bin differ
```

If the MPI code is implemented correctly, the `diff` command will not report that the files differ and exit with return code `0`.

c) (20 points) Complete the code for the constructor in `src/LaplacianSmootherMPI.cpp:11`. You need to take care of two 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.

d) (20 points) The main `sweep` method (performs one smooth step) is defined in the sequential implementation `src/LaplacianSmoother.cpp:22` 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

```
src/LaplacianSmootherMPI.cpp:49 and
src/LaplacianSmootherMPI.cpp:56,
```

respectively. Make use of the data structures that you have defined in the previous task. You can check that your implementation produces the same result as the sequential code with

```
./main # you do not need to run this again if you already did
./mainMPI
diff referenceMPI.bin referenceSEQ.bin
```

The `diff` program should return silently with an exit code `0`. Be sure you compile your changes by calling `ninja` while inside your `build_directory`.

e) (15 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

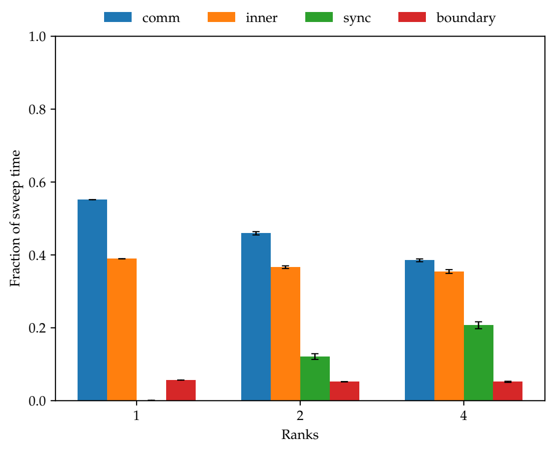

Figure 1: Example bar plot for the fraction of time per sweep spent in computation and communication.

measurements are the accumulated time over `sweep_count_` invocations of the `sweep` method. Write this code in `src/LaplacianSmootherMPI.cpp:38`.

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?

Hint: Make sure to run these experiments with an optimized executable. That is, create a build directory with `meson setup build_opt --buildtype=release` and compile the code with `meson compile -C build_opt`.

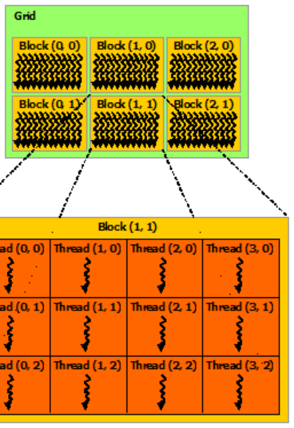

Figure 2: A 2D CUDA kernel launch grid with 2D thread blocks.

Task 3: Introduction to CUDA GPU kernels (10 Points)

The Compute Unified Device Architecture (CUDA) is a programming language extension that allows to program Nvidia General Purpose Graphics Processing Units (GPGPUs). Its core abstractions implemented in the language extension are hierarchies of thread groups, shared memories and barrier synchronization, concepts that you are familiar with from previous lectures. The remainder of the upcoming lectures will focus on GPU programming in CUDA C/C++ (CUDA also offers Fortran extensions), where in this task we will look at the very basic CUDA syntax to get started.

a) (5 points) A compute task on the GPU is performed by an entity called a *kernel*. When computing on a GPU, the goal is to divide the computational domain into smaller pieces that can execute *independently* from each other. Independent means effectively data parallel. A problem for which this division is not trivial is hard to efficiently implement on a GPU.

   In CUDA jargon this division is achieved by launching a kernel on a *grid* where the elements that compose the grid are called *thread blocks*. The GPU schedules thread blocks for execution, where all threads in a thread block run simultaneously on the GPU and each executes the kernel code. The launch grid and thread blocks can be either 1D, 2D or 3D. The CUDA extension provides built-in variables that give you access to the thread coordinates within the kernel. These are

   ```
   blockIdx, blockDim, threadIdx,
   ```

   where each of them has a field `.x`, `.y`, `.z` to access the coordinate in the corresponding dimension. An example of a 2D launch grid is shown fig. 2. An example application where you could use such a launch grid would be a computation involving matrices (recall the similar blocking techniques we studied for such problems using SIMD intrinsics on a CPU).

   A GPU kernel in CUDA C/C++ is nothing more than a function with an additional execution space specifier such that the compiler knows what to do with it. The main execution space specifiers in CUDA are

   ```
   __global__ // GPU kernel
   __device__ // function only callable on the GPU
   __host__   // function only callable on the host CPU
   ```

   An example GPU kernel could look like:

```
__global__ void my_kernel(float *array)
{
    // do something with array
}
```

You are given the following C function

```
void kernel(const float *A, const float *B, float *C,
  const int N)
{
    for (int i = 0; i < N; ++i) {
        C[i] = A[i] + B[i];
    }
}
```

Convert this function into a CUDA GPU kernel that can run on an arbitrary 1D launch grid with arbitrary 1D thread blocks.

Hint: The loop is fully data parallel, i.e., the accessing pattern in the loop only depends on `i`. You will need `blockIdx`, `blockDim` and `threadIdx`. Make sure you do not access addresses that are out of bounds.

b) (5 points) A kernel is called from the host by using an *execution configuration syntax* specified by

`kernel<<<grid_dim, block_dim>>>(arguments);`

where `grid_dim` specifies the dimension of the kernel launch grid and `block_dim` specifies the dimension of the thread blocks. In the most general form these are variables of type `dim3`[1] but you can also pass integers in the case of 1D configurations.

Complete the following code to launch your kernel from the previous task with thread blocks that contain 128 threads each.

```
int main(void)
{
    float *A, *B, *C;
    const int N = 1000;
    // memory allocation
    const int n_threads = ; // number of threads per block
    const int n_blocks  = ; // number of blocks in grid
    // launch your GPU kernel here

    return 0;
}
```

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

[1]see https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dim3