COLFAX RESEARCH | SOFTWARE EXPERTISE EVALUATION (SEE)

# MPI String Vibration

Download lab file

---

## File list

- Makefile
- main.cc
- L.h
- L.cc
- worker.cc*

(* files need to be submitted)

---

## IMPORTANT NOTICE

There was a bug with one of the lab files that has been corrected. If you had downloaded the lab files before Jan 30th 2019, please download the lab files again. Submitting the old version of the lab files will result in a verification error.

## Instructions

In this lab, you will be working with an MPI application that simulates a vibrating string with a non-uniform linear density evolving in time using the Finite Difference Method (FDM). The string is modeled by its displacement from the center line, *d(x, t)*, where x is the position along the string, and t is the timestep. *x* then evolves as per the wave equation (see *Wikipedia* for a little background on the physics of the problem). The non-uniform linear density of the string results in varying values of the wave velocity as a function of position along the string, characterized by *L(x)*. In order to apply the FDM, the

string is discretized into `n_segments` segments. At each timestep, the objective is to solve for *d(x, t+1)*.

While a detailed exposition of using the FDM to solve this problem is beyond the scope of the lab, you can find more information at the *Wikipedia* page on the FDM. We can boil the workload down to an iterative solver with two buffers, `d_t1` and `d_t`:

```
for(int t = 0; t < n_steps; t++) {
#pragma omp parallel for simd
  for(long i = 1; i < n_segments-1; i++) {
    const float L_x = L(alpha,phase,i*dx);
    d_t1[i] = L_x*(d_t[i+1] + d_t[i-1])
              +2.0f*(1.0f-L_x)*(d_t[i])
              - d_t1[i];
  }
  float* temp = d_t1; d_t1 = d_t; d_t=temp;
}
```

`d_t1` stores *d(x,t+1)* (the "next" position) and `d_t` stores *d(x,t)* (the currect position). Note that we do not have to simulate the ends of the string at `i = 0` and `i = (n_segments - 1)` because we assume that they are fixed. After the timestep, we make `d_t1` the new current position, `d_t`. We also recycle the `d_t` buffer for the new `d_t1`. In other words, we swap the two buffers.

The code above is for a single process. To distribute the workload into multiple processes, we divide the string equally among the processes and have each process work on it's piece of the string.

```
const long start_segment = segments_per_process*((long)rank)   +1L;
const long last_segment  = segments_per_process*((long)rank+1L)+1L;
  // ...
for(int t = 0; t < n_steps; t++) {
#pragma omp parallel for simd
  for(long i = start_segment; i < last_segment; i++) {
    // ... work on segments
  }
  float* temp = d_t1; d_t1 = d_t; d_t=temp;
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
                &d_t[1], segments_per_process, MPI_FLOAT,
                MPI_COMM_WORLD);
}
```

Here `segments_per_process` is the number of segments that each MPI process must compute. After each time step, we use the `MPI_Allgather` command to synchronize the current string displacement across the processes. This turns out to be inefficient because we are transfering far more data than necessary. To see this, perform a quick scaling test on the Colfax cluster, distributing the

computation across varying numbers of nodes (try with 1, 2, and 4 nodes). You can reach the same conclusion by doing a quick back-of-the-envelope calculation of the data transfer requirements.

Each position, `d_t1[i]`, is only dependent on the immediate neighbors, `d_t1[i+1]` and `d_t1[i-1]`. So instead of synchronizing the entire string, it is sufficient to only synchronize the segments neighboring the `start_segment` and `end_segment`.

Modify "worker.cc" so that only the immediate neighbors are transferred after each time step. Remember to gather the entire string to rank 0 process at the very end.

---

## Hints

Remember that the MPI commands are blocking by default. If you start an *MPI_Send* operation on every process at once, your application will hang.

---

## Running app:

The grading script uses the following command to run the application.

```
% mpirun -host localhost -np 4 ./app $ALPHA
```

---

## Grading

1. Compile: The code compiles without error (0 points)
2. Verification: The code generates the correct output (0 points)
3. Performance: The application completes in under 0.9 seconds on the Xeon Phi 7210 system (1.0 points)

---

Last transaction TID: 29151
Your grade has been sent to the course platform.

Solution: 选取文件 未选择文件

Resubmit

| TID | Grade | Feedback | Submitted file | Submission time |
|---|---|---|---|---|
| 29151 | 100 * | Compilation: PASSED<br>Verification: PASSED<br>Performance Check: PASSED<br>Submitted code completed in 0.635075s | Submitted solution | 2020/12/18 13:02:34 |
| 29150 | 0 | Application timed out after 90 seconds | Submitted solution | 2020/12/18 11:10:03 |
| 29149 | 0 | Compilation: PASSED<br>Verification: FAILED<br>The result did not match the reference result | Submitted solution | 2020/12/18 10:59:11 |
| 29145 | 0 | Compilation: PASSED<br>Verification: PASSED<br>Performance check: FAILED<br>The computation must be completed under 0.9s. Submitted code completed in 12.555342s | Submitted solution | 2020/12/17 14:25:53 |

Legend:

* - your best grade for this lab so far

x - not yet submitted to the course platform. Grade transfer may take up to 5 minutes. To re-check the status, reresh this page

Coursera Learners:

- Coursera only remembers your highest score. If you re-attempt the lab and get a lower score, it will not be reflected in Coursera
- If your best grade does not get reflected in Coursera for more than 10 minutes, please let the system administrators know at labs@colfaxresearch.com