Milestone Report: Parallel QR Factorization

Jiarui Zhang (jiaruiz), Wangshu Li (wangshul)

1. Project Web Page URL

https://liwangshu.github.io/QRparallel/

2. Schedule

Week 1 (11/7 - 11/13)	(Project proposal due) Research on ideas to parallel algorithm	Completed
Week 2 (11/14 - 11/20)	Baseline sequential implementation of QR factorization, profile code to figure out bottlenecks	Completed
Week 3 (11/21 - 11/27)	Implement OpenMP and OpenMPI versions of QR factorization	Completed
Week 4 (11/28 - 11/30)	(Milestone Report due) Write milestone report (Wangshu, Jiarui)	Completed
Week 4 (11/30 - 12/4)	Create CUDA baseline code for QR factorization (Wangshu)	TBD
Week 5 (12/5 - 12/8)	Enhance CUDA parallel code to achieve reasonable speedup (Jiarui)	TBD
Week 5 (12/8 - 12/11)	Analyze the results and write final reports (Wangshu, Jiarui)	TBD
Week 6 (12/12 - 12/18)	Submit final report & Demo (Wangshu, Jiarui)	TBD

3. Summary

We created two folders: the src folder and the test folder. In the src folder, we wrote four cpp files: one sequential version, one OpenMP version, and two OpenMPI versions of QR factorization. In the test folder, we wrote two files: one for testing sequential and OpenMP implementations, and the other for testing OpenMPI versions. The test files will also output the execution time of each call of QR factorization functions. We also created a Makefile to compile those files by using `make` or `make mpi`. The codes can be run on GHC machines and can generate some results which will be described in the Preliminary Results section.

3. Goals and Deliverables

Plan to achieve: We plan to implement the sequential version of QR factorization first, and then parallelize it in three different ways, including using CUDA threads in GPU, OpenMP and MPI. Then we will pick one with the best performance and analyze it. We will also plot some graphs to show the speedup of different parallel programming models over the sequential version as the visualization result of our demo.

Hope to achieve: There are some existing parallel QR factorization algorithms. We will further optimize one of our parallel programming models and hope our model can achieve similar or better performance compared to the existing algorithms on the same type of machines.

We have reached most of the “Plan to achieve” goals. We believe we can finish the remaining “Plan to achieve” goals and produce all the deliverables as expected. For the “Hope to achieve” goals, we will try our best to optimize our solutions further and hope to achieve an ideal performance compared to the existing algorithms.

4. Poster Session

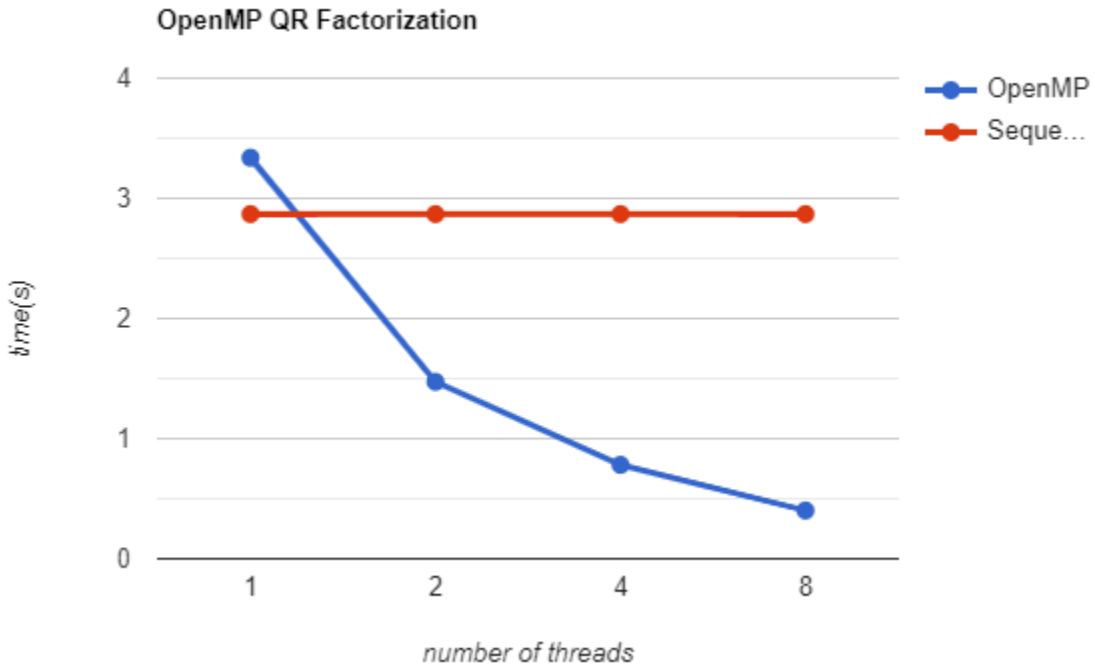
We will show some graphs at the poster session. Specifically, the graphs are about the speedup of several parallel programming models, including OpenMP, MPI and CUDA,

over the sequential implementation. Also, we will show why some of the programming models are better than others in terms of scalability and (possibly) locality.

5. Preliminary Results

We've completed two parallel programming models for QR factorization, including OpenMP and MPI.

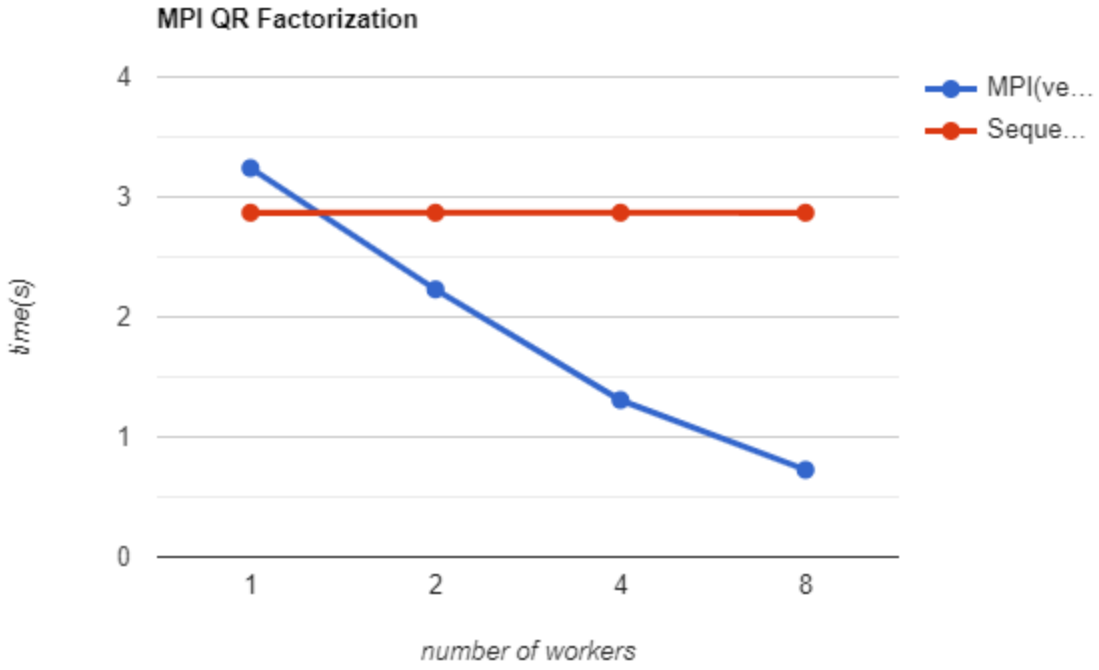
For OpenMP, we applied **#pragma omp parallel for schedule(dynamic) num_threads()** to parallelize the program. We also added necessary barriers inside the for-loop when there are some computation dependencies to ensure the correctness of the result. Below is a plot we drew to show the scalability of the OpenMP model, we can see from the plot that the time needed to finish the program decreases almost linearly as the number of threads increases.



For MPI, we implemented two versions. Both of them have the same idea: we want to partition the work to several workers, each of which are responsible for the same

number of columns. Workers with larger pid need to wait for the results computed by workers with smaller pid due to the nature of QR factorization.

In version1, we let every worker wait until they receive all the columns computed by the previous workers asynchronously. We used a **1-d** buffer to store the results. This does not scale at all and its performance is even worse than the sequential version due to the overhead required to execute the MPI. Instead, we came up with version2, which uses a **2-d** buffer to store the columns computed by the previous workers once it's received. The reason why we used a **2-d** buffer is the current worker has more than one column to compute, every column needs the previous results before computation. **Even if some columns can not finish all computation, we can still store some temporary results to the buffer and continue the computation after all previous columns have been received**. Compared to version1, **version2 allows the program to do some computations and stores at the same time as it receives the results from previous workers**. After receiving all the results from previous workers, the current worker has much less work to do compared to version1 because most of the work was done during the receive phase. Here is a plot we drew to show the scalability of the MPI model(version2), we can see from the plot that the time needed to finish the program decreases almost linearly as the number of workers increases.



6. Concerns

Currently, we haven't implemented CUDA version yet, we are not sure about if this is difficult to implement or not. We are hoping resources are available on GHC machines so that we can start work on the CUDA version soon.

Also, we find out that the locality in our program(both sequential and parallel) is not good. The nature of QR factorization requires us to process the input matrix column-wise. But we all know that data in different columns does not stay contiguous in memory/cache so the locality is not good. We want to find out a way to do QR factorization row-wise to have better locality. Maybe just transpose the matrix during computation and transpose it back after computation. We are not sure about how to do it right now but we will take this into consideration in the future.