Final Report: Parallel QR Factorization

Jiarui Zhang (jiaruiz), Wangshu Li (wangshul)

1. Project Web Page URL

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

2. Summary

We implemented a sequential version and three parallel versions (OpenMP, OpenMPI, CUDA) of the QR factorization algorithm. Among the three parallel implementations, OpenMP performs the best, achieving 7x speedup on a GHC machine using 8 cores and achieving 41x speedup on a Bridges-2 machine using 64 cores.

3. Background

In linear algebra, QR factorization is a decomposition of a matrix A into a product A = QR of an orthogonal matrix Q and an upper triangular matrix R. QR factorization is often used to solve the least squares problem and is the basis for a particular eigenvalue algorithm, the QR algorithm.

The key data structures used in the algorithm are 2-D array (representing matrix) and 1-D array (representing vector). The key operations are floating point addition and multiplication used for norm computation, dot product, and matrix multiplication. The algorithm's input is a non-singular matrix A and the outputs are an orthogonal matrix Q and an upper triangular matrix R. The sudo code is shown below.

```
Algorithm 5: QR Decomposition.
  input  : A non-singular matrix $\mathbf{A}$ of size $n \times n$
  output: Orthogonal matrix $\mathbf{Q}$, upper triangular form $\mathbf{U}$
  Function QRFact($A$) is
      $N \leftarrow$ order of the square matrix $A$;
      $\mathbf{u} \leftarrow \mathbf{0}_{N \times N}$, $\mathbf{R} \leftarrow \mathbf{0}_{N \times N}$;
      for $j \leftarrow 0$ to $n-1$ do
          $\mathbf{w}_j \leftarrow A[:, j]$;

          /* Squared norm of w_j                                  */
          sum $\leftarrow 0$;
          for $k \leftarrow 0$ to $n-1$ do
              sum $\leftarrow$ sum $+\mathbf{w}_j[k] \times \mathbf{w}_j[k]$;
          $|\mathbf{w}_j|^2 \leftarrow$ sum;

          /* Compute the coefficients R[i][j]                     */
          sum $\leftarrow 0$;
          for $i \leftarrow 0$ to $j-1$ do
              $R[i][j] \leftarrow \mathbf{w}_j \cdot u[i]$;
              sum $\leftarrow$ sum $+R[i][j] \times R[i][j]$;
          $R[j][j] \leftarrow (|\mathbf{w}_j|^2 -$ sum $)^{1/2}$;

          /* Compute vector u[j] using fwd substitution           */
          sumvec $\leftarrow \mathbf{0}_{1 \times N}$;
          for $i \leftarrow 0$ to $j-1$ do
              sumvec $\leftarrow$ sumvec $+ R[i][j] \times u[i]$;
          $u[j] \leftarrow (\mathbf{w}_j -$ sumvec $)/R[j][j]$;
      $\mathbf{Q} \leftarrow \mathbf{u}^T$;
      return $\mathbf{Q}, \mathbf{R}$;
```

As we can see, the whole algorithm is computationally expensive, since it involves many matrix computations. Since this is an iterative algorithm and there are many for loops in the algorithm, it can benefit from parallelization. Based on the outer `for` loop, the algorithm computes the result matrices column by column. Therefore, we can break down the workload column by column and let each thread or process compute a certain amount of columns. Parallelizing over cells is also possible due to the inner `for` loops and the inner loops are amenable to SIMD execution. There is one dependency in this program. When computing the result of a column, it needs the results from the previous columns, which can be seen from the `for i ← 0 to j -1 do` lines below. This dependency leads to some synchronization costs which can decrease the performance of parallel versions of this algorithm.

4. Approach

4.1 OpenMP

In OpenMP, there are some built-in primitives for parallel programming. In this project, we first used `#pragma omp parallel for schedule(dynamic)` over the outer `for` loop. In this way, OpenMP dynamically assigns each iteration of work to a thread, which balances the workload. Inside each iteration, we also added `#pragma omp parallel` to parallel the inner loops. However, in doing so, we need to be aware of the dependencies between the inner loops. Some computations cannot begin until previous calculations have completed. Therefore, we added `#pragma omp barrier` where necessary to ensure correctness.

4.2 OpenMPI

In OpenMPI, processes do not share the input matrix and the output matrices. Instead, they must communicate explicitly to send variables to and receive variables from other processes. As we mentioned in the previous section, when a process is computing the result of a column, it needs the computation results from the previous columns. That is where communication comes from. Therefore, we distributed the work evenly to all processes and let processes with lower pid send their results to processes with higher pid so that processes with higher pid can complete their computations.

In our first version of OpenMPI implementation, we let each process compute results column by column, the same as before. When computing each column, each process waits until receiving all the columns computed by the previous processes in order to finish its computation. To send and receive asynchronously, we used 1-d buffers to store the results. This approach leads to too much idle time for each process to wait for the messages, performing even worse than the sequential implementation.

In order to optimize the performance, we came up with the second version of OpenMPI implementation, where we had each process compute all of its columns horizontally. Specifically, we removed the outer `for` loop, and let each process compute all parts of all columns that do not require communication. Then, each time it receives a column message from another process, it handles all the columns that require that message. To achieve this, we used 2-d buffers to store the results sent by the previous workers. The reason for this is that the current process has more than one column to compute, and every column needs the previous results to finish computation. Even if some columns cannot finish all computations, we can still store some temporary results to the buffers and continue the computation later after all previous columns have been received.

Compared to version1, version2 removes the outer loop and reduces all workers' idle time because all columns can be processed at the same time. This method improves the performance significantly, achieving some speedup over the sequential version.

4.3 CUDA

CUDA is most suitable for independent workload that does not require synchronization or communication, such as SAXPY. In this project, if we let each thread compute some columns on GPU, it is difficult for threads in a thread block communicate with threads in another thread block. Therefore, we parallelized over cells in our CUDA version of QR factorization. Specifically, for each iteration in the outer loop, we launched a kernel function that parallelized the computation on GPU. Each thread is responsible for a cell with a specific row number and column number in the output matrices. We also defined some variables in the block-shared memory for threads in a thread block to use, and added `__syncthreads()` statement to synchronize between threads in the same way as the barrier in OpenMP. By using these approaches, we can ensure the correctness of this CUDA implementation.

5. Results

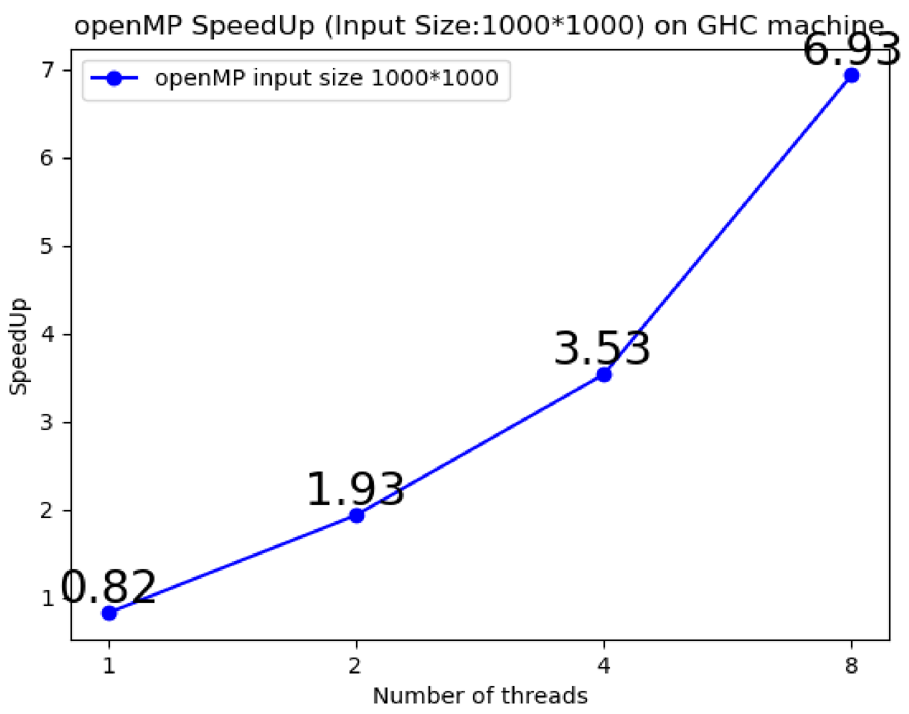
Our project is to parallelize the QR factorization. We measured the performance of our parallel algorithm by computing the speedup over the sequential version.

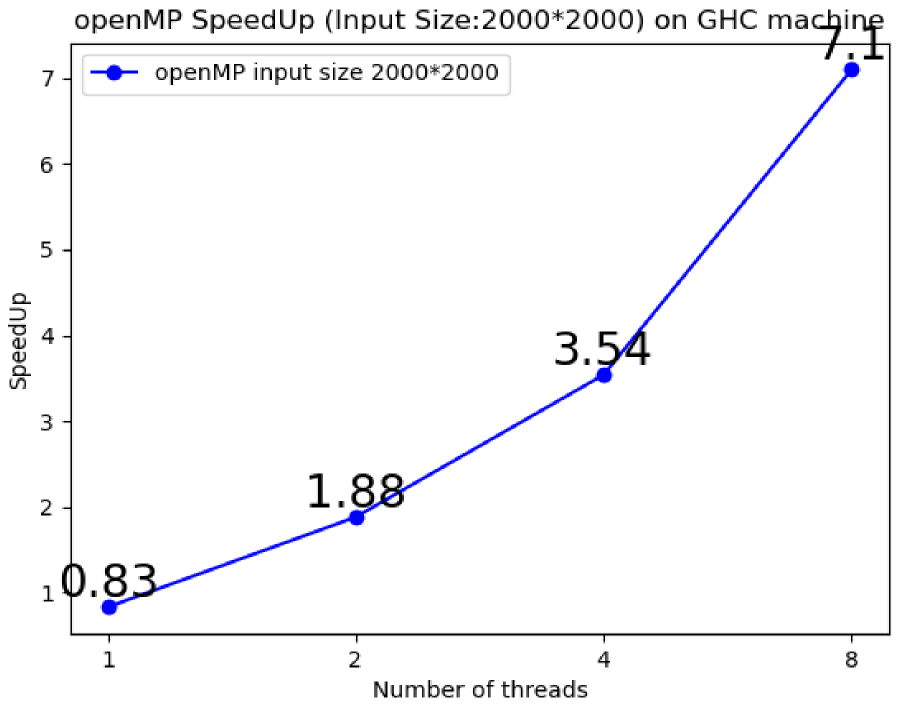
We have three different input sizes: 1000*1000, 2000*2000, 3000*3000. We use the test.cpp file to generate a random matrix with floating point numbers of a specific input size, call our parallel programming function, get the result back, and finally test the correctness and performance (time used to execute the algorithm)

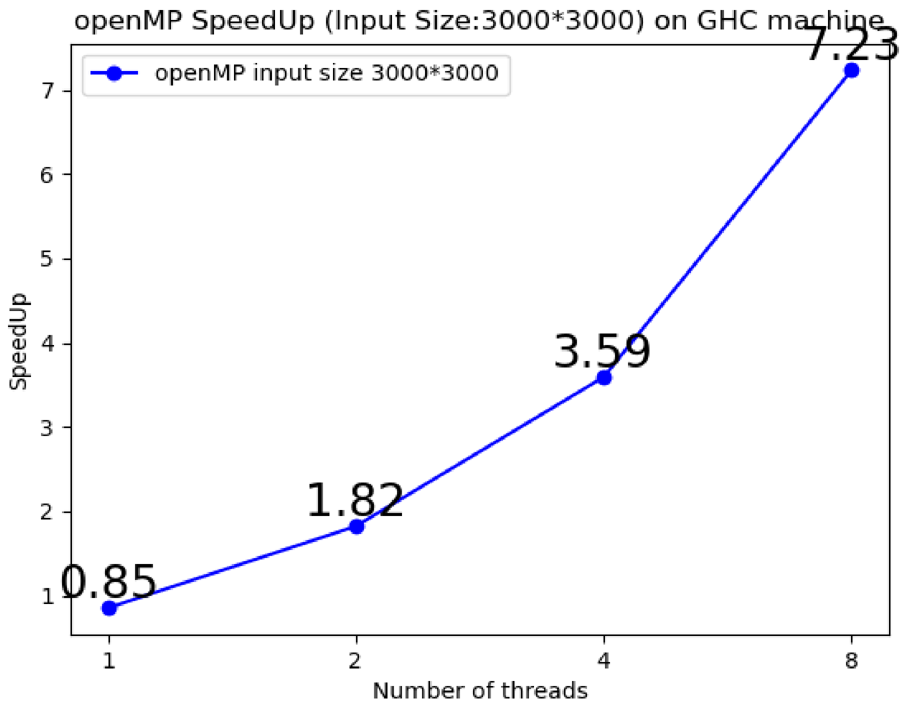
We've completed three parallel programming models for QR factorization, including OpenMP, OpenMPI, and CUDA.

5.1 OpenMP

Below are some plots we drew to show the speedup of the OpenMP model on GHC machines with different input sizes, the baseline is the single-threaded CPU code.

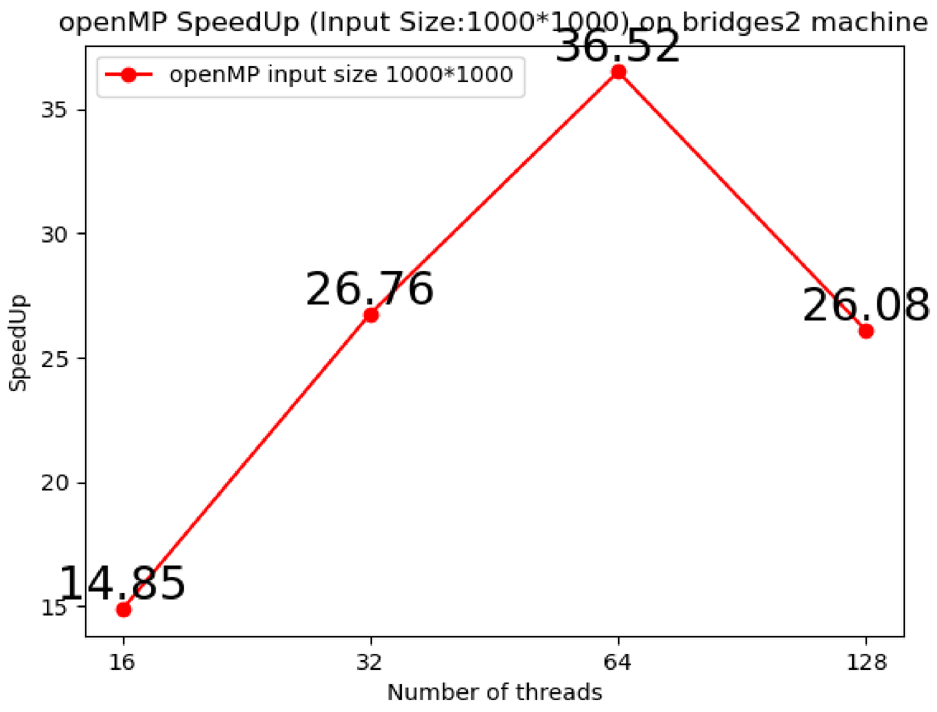


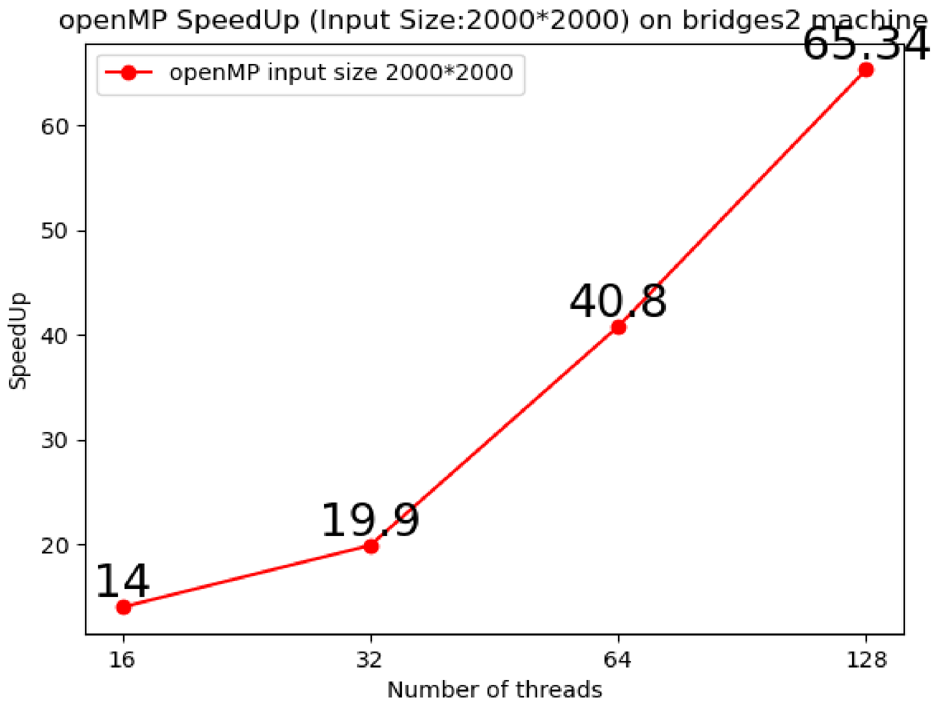

openMP SpeedUp (Input Size:2000*2000) on GHC machine
openMP input size 2000*2000
SpeedUp
Number of threads
1
2
4
8
1
2
3
4
5
6
7
0.83
1.88
3.54
7.1

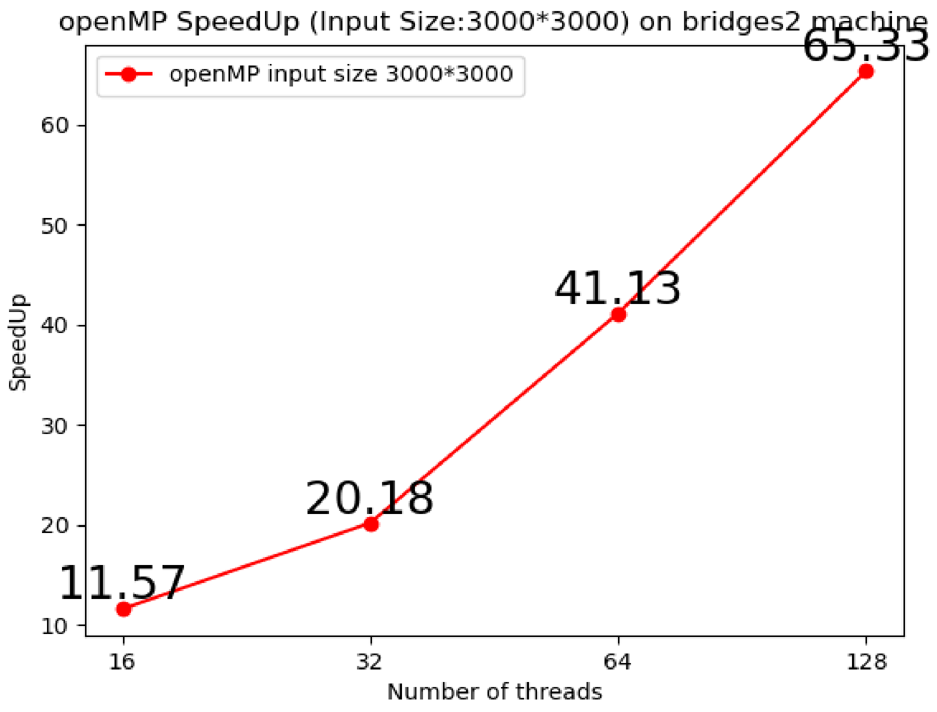

openMP SpeedUp (Input Size:3000*3000) on GHC machine
openMP input size 3000*3000
SpeedUp
Number of threads
1
2
4
8
1
2
3
4
5
6
7
0.85
1.82
3.59
7.23

We can see from the plots that the speedup increases almost linearly (We achieved ~7x speedup over sequential version when there are 8 threads on GHC machine). Also, different workloads only exhibit a slightly different execution behavior, the larger input size, the higher speedup(although the improvement is negligible).

In order to test the scalability of our program, we also ran our program on Bridges-2 machines. Bridges-2 machines also have more cores. Below are some plots we drew to show the speedup of the OpenMP model on Bridges-2 machines with different input sizes. The baseline is still the single-threaded CPU code.




openMP SpeedUp (Input Size:2000*2000) on bridges2 machine
openMP input size 2000*2000
SpeedUp
Number of threads
16
32
64
128
14
19.9
40.8
65.34

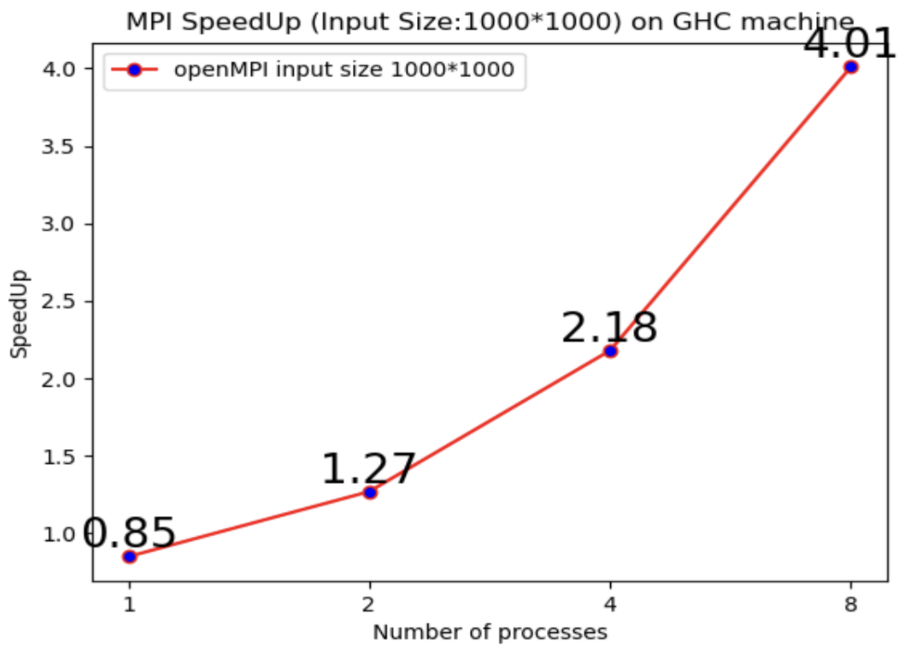

openMP SpeedUp (Input Size:3000*3000) on bridges2 machine
openMP input size 3000*3000
SpeedUp
Number of threads
16
32
64
128
11.57
20.18
41.13
65.33

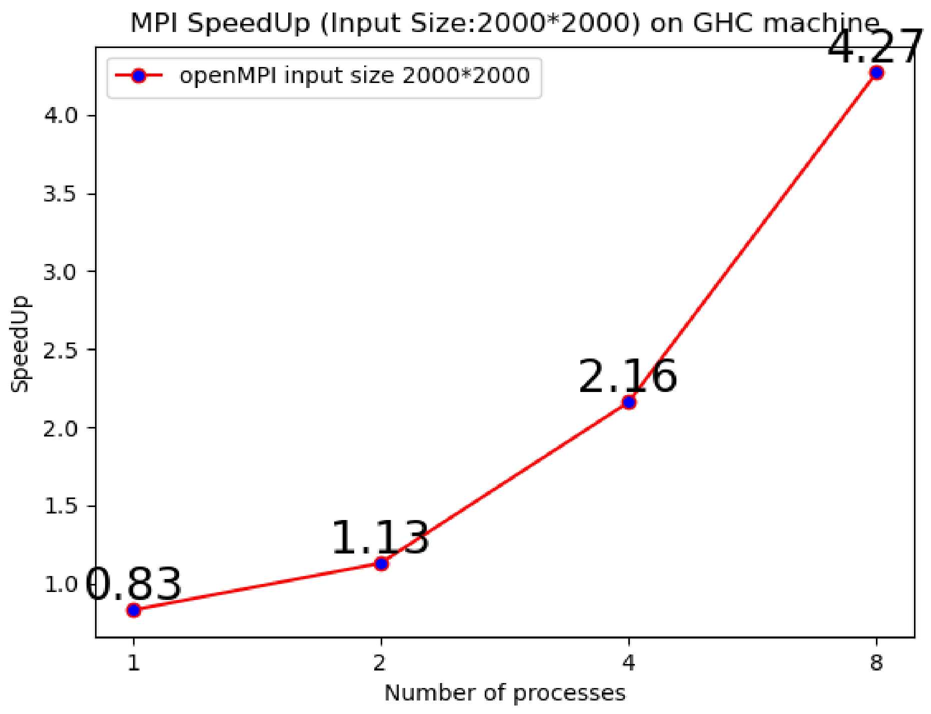
We achieved ~41x speedup with 64 threads and ~66x speedup with 128 threads when input size is 2000*2000 or 3000*3000. We also observed that there is a significant performance drop down when the number of threads is equal to 128 with input size 1000*1000. This is because the input size is not large enough to keep all 128 threads busy, the time of context switching between threads dominates the total execution time.

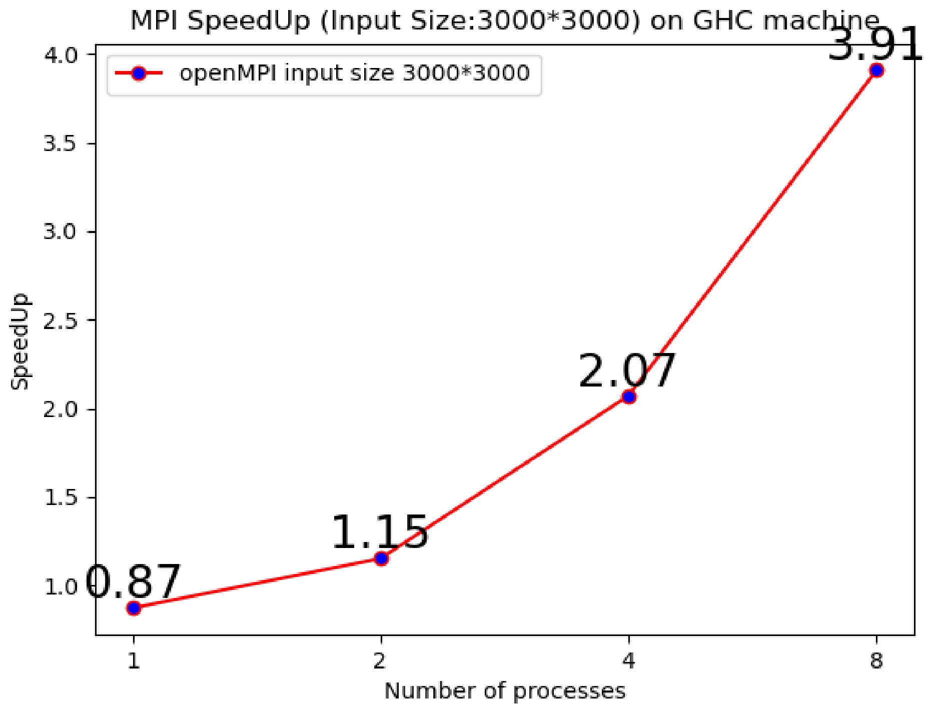
We can conclude that when input size grows, it is unlikely for us to achieve perfect linear speedup due to the data dependencies in our algorithm. In other words, some threads remain idle when waiting for the necessary computation results from other threads.

5.2 OpenMPI

As discussed in the previous section. We implemented two versions of the OpenMPI models. Here are some plots we drew to show the scalability of the MPI model (version2) on GHC machines with different input sizes.




MPI SpeedUp (Input Size:2000*2000) on GHC machine
openMPI input size 2000*2000
SpeedUp
Number of processes
1
2
4
8
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.83
1.13
2.16
4.27


MPI SpeedUp (Input Size:3000*3000) on GHC machine
openMPI input size 3000*3000
SpeedUp
Number of processes
1
2
4
8
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.87
1.15
2.07
3.91

As shown in the graphs, the speedup of openMPI model is worse than openMP, we achieved ~4x speedup when we have 8 processes (nearly half of the number of total processes)

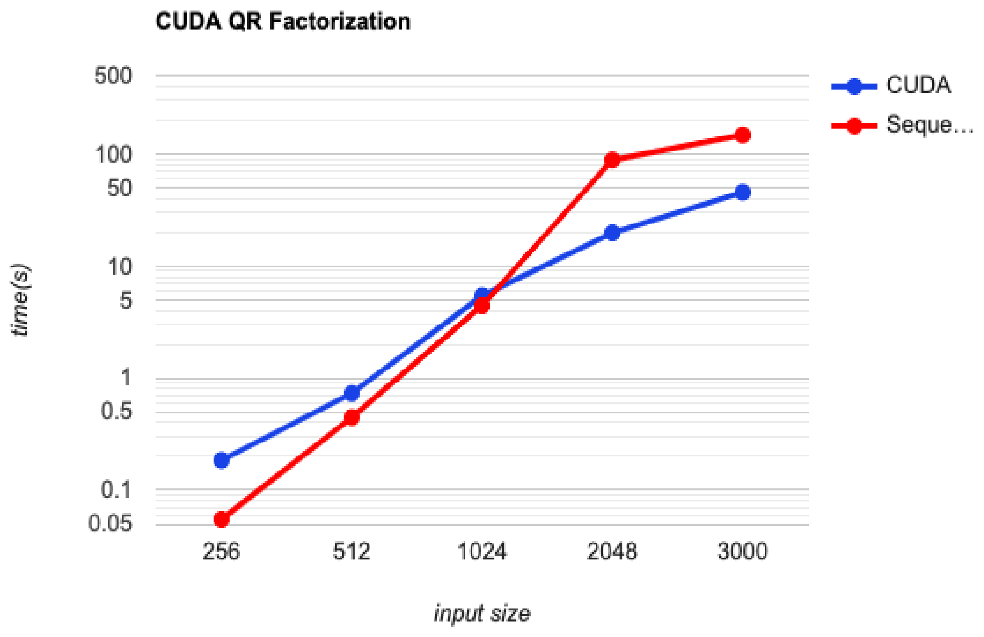
Also, different workloads only exhibit a slightly different execution behavior. Those differences are trivial and negligible.

Communication cost and synchronization overhead limit the speedup of the openMPI model. Each process needs to send its computation results to other processes with higher pid. For those processes with higher pid, they need to wait until they receive all the computation results from previous processes before computing their own columns. Although we optimized our algorithm by using 2-d buffers when receiving previous columns to achieve better workload balance and workflow pipelining, those synchronization overheads are inherent so they cannot be reduced significantly.

5.3 CUDA

As discussed in the previous section, we also used CUDA to parallelize the algorithm. The granularity in the CUDA version is a single cell, which is different from OpenMP and OpenMPI. We parallelized over cells because threads in different thread blocks cannot communicate with each other in CUDA so there is no way (or difficult to implement) to let some of them wait until previous threads complete their tasks.

Here is a plot we drew to show the performance of the CUDA model with different input sizes.



We can see from this plot that CUDA performs even worse than the sequential version when input size is small, but it performs better and better when input size grows. The reason for this is that arithmetic intensity grows as input size increases, meaning that the system spends more time in useful floating point arithmetic.

But even the input size is large enough(3000*3000), the speedup is only around 2, which is much worse than openMP and openMPI. Here are several reasons why CUDA performs badly. First, in order to guarantee the correctness of the result, each column needs to launch the kernel function separately; those launches introduce some overhead. Second, cuda is a great programming model for independent works, but due to the nature of QR factorization, there exists a lot of data dependencies, which drop the performance down significantly.

We also break down the execution time to several components: copy to GPU, kernel, copy to CPU. Take input size 1000*1000 as an example, time for copying to GPU is 2.174ms, time for kernel function is 4846.414ms, and time for copying the result back to CPU is 1.757ms. The major part of the total execution time is kernel function. Copying data back and forth is trivial compared to the kernel.

Above all, OpenMP has the best performance. In this particular case, shared address space model has the minimum communication cost. Also, it does not require any synchronization between threads because there are no concurrent writes when reading the previous columns. CUDA in nature is not a good choice for dependent works. For openMPI, the inherent communication cost can not be reduced by optimizing the algorithm.

6. Reference

- Chunawala, Q. (2022, November 4). *Fast algorithms for solving a system of linear equations*. Baeldung on Computer Science. Retrieved December 15, 2022, from https://www.baeldung.com/cs/solving-system-linear-equations

7. Work distribution

Wangshu Li and Jiarui Zhang equally contribute to this project. The total credit for the project should be distributed 50%-50%.