Parallelism in BSIM

Peicheng Tang
Yajie Wang

SUMMARY

We parallelized the Blind Structured Illumination Microscopy(BSIM) algorithm with three different parallel models: synchronized MPI, asynchronized MPI and hybrid of OpenMP and asynchronized MPI. Our implementation on Bridges achieved 5.977x speedup for synchronized MPI, 6.071x for asynchronized MPI with 8 cores, and achieved 19.404x speedup for asynchronized MPI and 10.802x speedup for hybrid model with 32 cores.

BACKGROUND

BSIM algorithm has been widely used in the biomedical field since it can reconstruct high resolution images that exceed Abbe distraction limitation with low hardware cost. The input data for BSIM are hundreds of images generated under different pattern illumination environments and obtained by microscopy. In this project, we simulated the imaging processing to generate input data. The output data for BSIM would be one image with high resolution. Because we would deal with images most of the time, we chose to represent each image as a 1D vector of doubles, and used the vector of vectors to index each image. We would mainly do matrix operations and convolutions to those images, and it could be difficult for us to write some of the math operations like Bessel functions by ourselves. We chose to utilize various libraries, mentioned in the APPROACH section, to help us with those operations.

The BSIM algorithm process can be divided into two parts: generating hundreds of estimated patterns, where each pattern is an image, based on gradient descent and calculating the output result based on the covariance between estimated patterns and input data.

In our project, we use the proximal gradient descent(PGD). The PGD part is the most computational expensive one, which often takes from ten minutes to several hours depending on experiment setting. Parallelizing gradient descent can be troublesome since we have

dependency between two iterations. Also in each iteration, the calculation of gradient for all patterns must be executed before calculating the total cost value.

Algorithm 1: Pseudocode of Proximal Gradient Descent in BSIM

```
1  function PGD Parameter: 2D Vector of Vector residual, patterns and
                           new_patterns
                           double cost_value, iter_num, step
2  initialize residual
3  while iter_num < NUM_ITERS do
4      for j iterate over all images do
5          initialize gradient as 1D vector
6          calculate gradient based on residual[j] and patterns[j]
7          calculate new_patterns[j] based on gradient, step
8      end
9      for j iterate over all images do
10         update residual[j] based on new_patterns[j]
11         cost_value += sum of each element in residual[j]
12     end
13     if cost_value_next > cost_value then
14         step/ = 2;
15     else
16         update cost_value;
17         update patterns;
18     end
19 end
```

Figure 1. Pseudocode of Proximal Gradient Descent(PGD) in BSIM that showing the data dependency for sequential version of PGD

Note that we intentionally omit detailed calculations and other unchanged inputs required in this figure to keep the figure clean, as Figure 1 aims to show the dependencies in our sequential version of PGD. *Residual* depends on the *new_patterns*, which also depends on the value of the *gradient*. We then can calculate *cost_value* based on the *residual*. In such a sequential execution, waiting for all the images to finish their processing can be really slow.

There will be potential parallel execution for the iteration over images in the code snippet since each image is independent of each other. However, even though we can run the for loop in parallel, we still need to wait for all the parallel threads to finish to obtain the cumulated *cost_value_next* and compare it with the current *cost_value*, which means that a

potential slow thread may hurt the entire performance if we simply parallel over the for loop. As a result, we decided to use MPI as our programming model. In the next section, we will discuss more details about our MPI implementation.

APPROACH

We started our project from converting Matlab code to C++. We used BOOST for the bessel function, FFTW3 for fast convolution calculation, Eigen for image vector operation and OpenCV for image show. The project runned on the Bridges machine, which supports hybrid OpenMP and MPI environments.

In our project, gradient calculation for each pattern in each iteration is independent. Thus, we partitioned the work based on patterns and let each core runned part number of patterns. However, the cost value calculation for the sequential algorithm depends on all the patterns, which means we should either follow this dependency(synchronized MPI) or break the dependency(asynchronized MPI). Furthermore, since the MPI is the coarse distribution, we can apply fine distribution in each working machine based on OpenMP(Hybrid MPI).

The main communication model we applied was the master-slave model. For the gradient part, the master distributed pattern estimation work by assigning patterns to slaves, gathered cost value and controlled the next step for the iteration process based on cost value. Because we knew the total number of patterns to be calculated, we statically partitioned those patterns in equal and assigned them to slaves to achieve load balance. Meanwhile, since the data size for patterns was too large to communicate, we implemented local cost value calculation in each machine and thus the master machine contained no information about the estimated patterns. On the other hand, with such a programming model, covariance calculation which was based on estimated patterns had to be done in the slave machine as well. The slave machines would have to communicate with the master machine as shown in Figure 2. They need to locally calculate and send their partial sum to the master, retrieve the

total sum from the master, calculate and send the partial covariance. In the end, the master machine would gather all the partial covariance from all the slaves machines to obtain the final result.

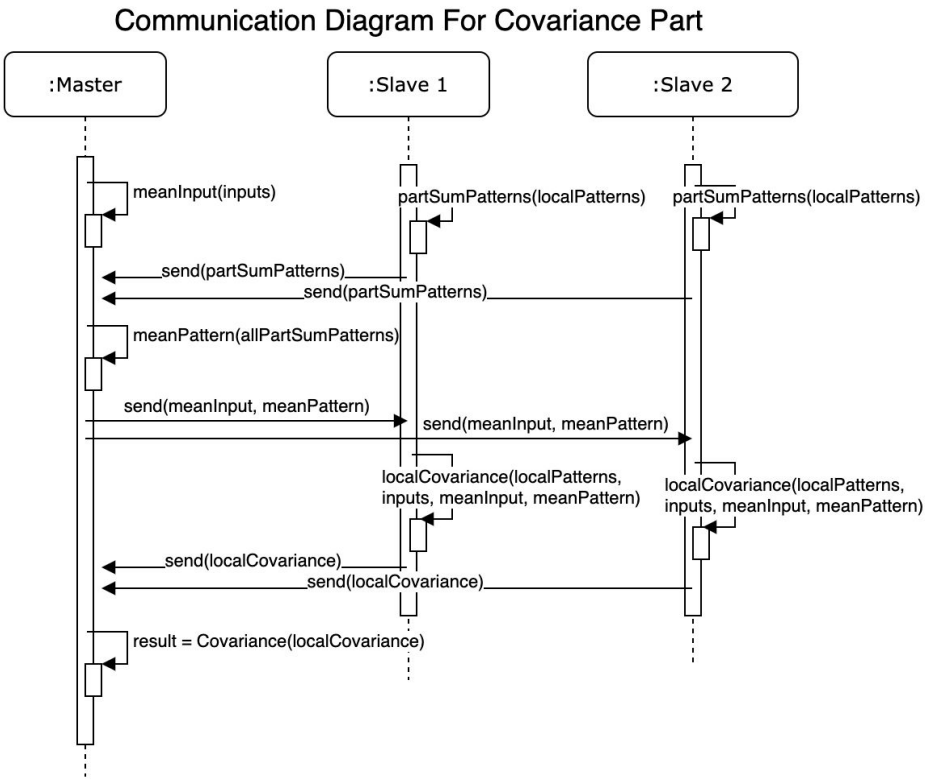

Figure 2. MPI Communication Diagram For Covariance

- Synchronized MPI

The main idea for synchronized MPI was similar to the sequential one, which was all the slave machines sharing the same update pace. the master machine waited for all the slave machines to finish the cost value update for the current iteration. If the cost value descended, it asked all slave machines to accept the result and kept going to the next iteration together. Otherwise, all the slave machines discarded the result, cut the step into half and recalculated again. This method may waste some time because the master machine should wait for all slaves, especially when the work distribution is imbalanced. This would create an implicit barrier for slave machines to continue their local work. The detailed synchronized MPI communication between the master and slaves are shown below in Figure 3.

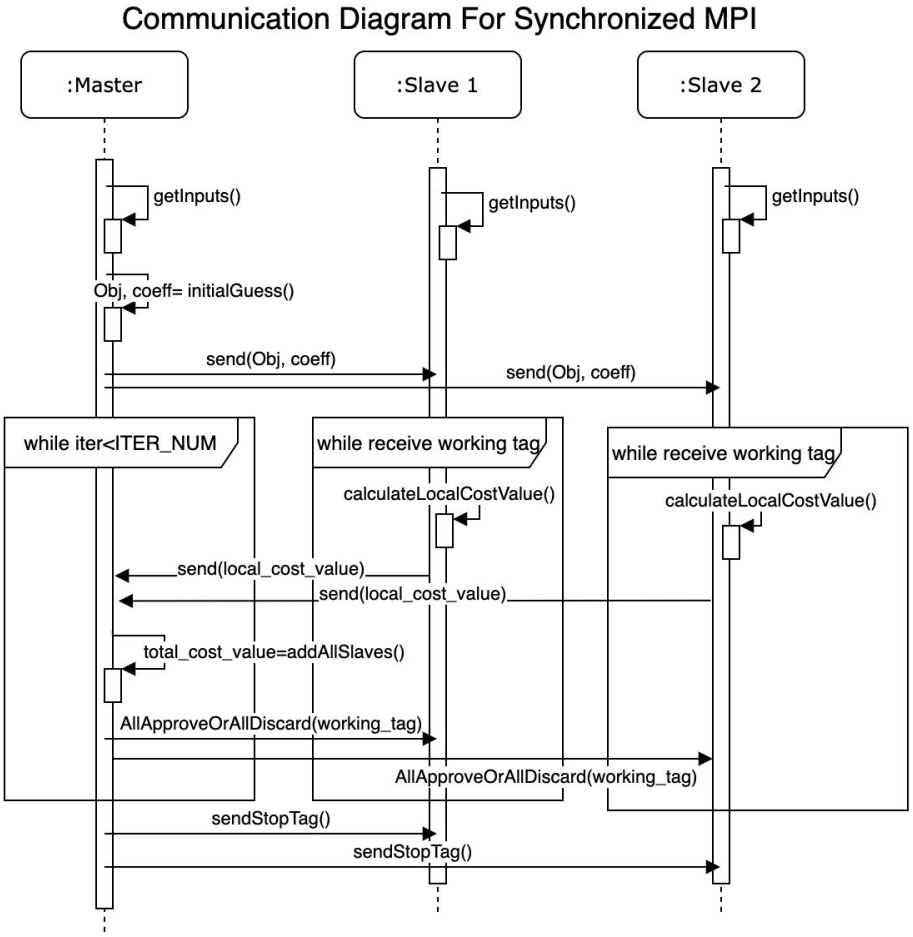

Figure 3. Communication Diagram for Synchronized MPI

- Asynchronized MPI

The asynchronized MPI is used to break the barrier at the end of each iteration for the master machine. Figure 4 shows the overall communication in this setting. Unlike the synchronized MPI, each slave machine could run at a different iteration pace during running time and master recorded iteration pace for each slave machine. Besides that, the master machine also stored partial cost value for each slave and total cost value for the last iteration for each slave. Each time one slave sent its own partial cost value, Master updated the total cost value for this iteration for this slave. If this cost value descended, Master asked this slave machine to accept the result and kept going to the next iteration. Otherwise, this slave machine discarded the result, cut the step into half and recalculated again. In short, we

removed the barrier in synchronized version by keeping track of the number of iterations for the slaves in the master machine with additional memory overhead, but allowing each slave machine to run at their own pace. However, there would still be an unavoidable barrier before calculating the covariance because it depended on all estimated patterns.

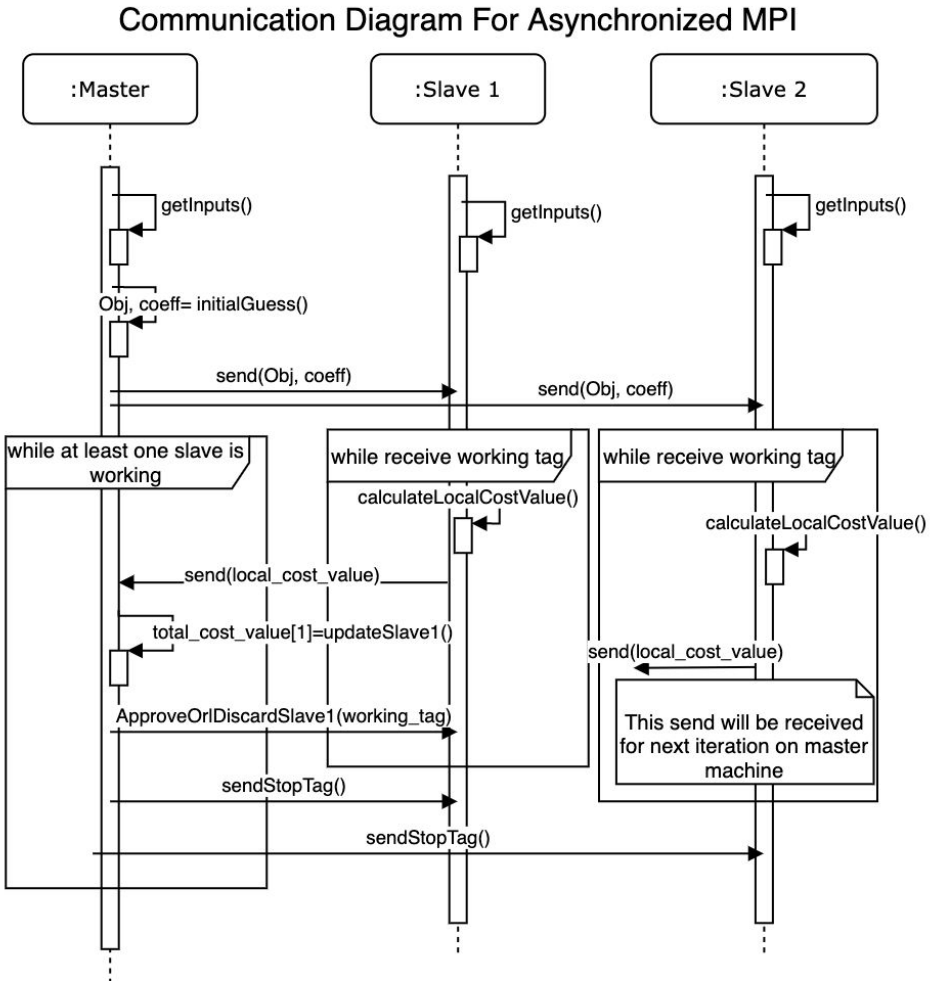

Figure 4. Communication Diagram for Asynchronized MPI

- Hybrid of OpenMP and Asynchronized MPI

As mentioned in the early section, we could parallelize the loops every time we need to iterate over the patterns since all the individual patterns are independent of each other. We believed that we could use OpenMP to help parallelize those for loops. Building on the

asynchronized MPI method, we added OpenMP clauses to run those for loops in parallel. We also used reduction for cost value calculation.

RESULTS

Experimental Setup

The inputs for the BSIM algorithm are images observed by microscopy structure under each illumination pattern. Therefore, if we have N illumination patterns, we will have N input images. Since here we had no hardware support to generate inputs from experiment, we simulated the microscopy imaging processing to get all the requests.

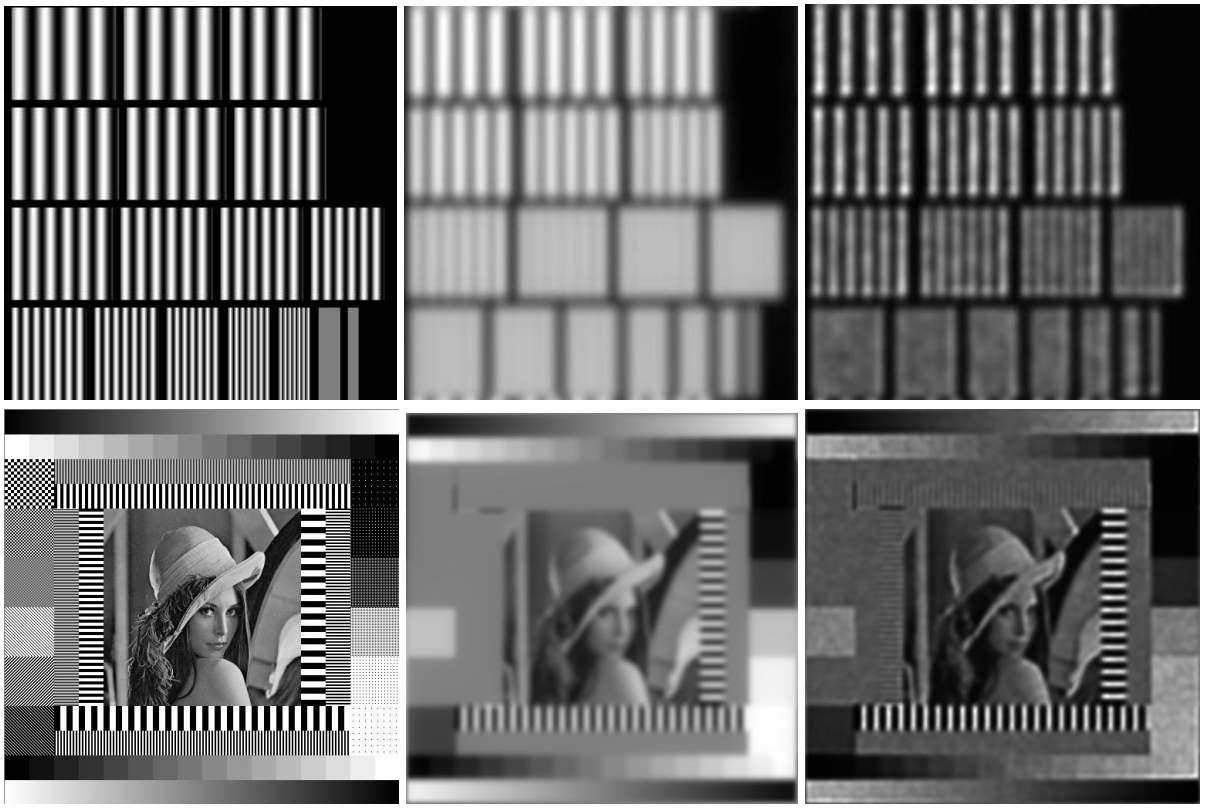

Figure 5. Reconstruction results under a hybrid parallel method with 7 MPI tasks and 4 OpenMP threads per MPI task. a(1)-a(3) are images of 256 by 256 images. b(1)-b(3) are images of 512 by 512 images. a(1) and b(1) for objective. a(2) and b(2) for the widefield result. a(3) and b(3) for the reconstruction result.

In our project, we first used a 256 by 256 image, which was particularly used for resolution measurement, with 384 patterns to obtain more parallel results with shorter running time and smaller memory consumption. We also tried a 512 by 512 normal image

with 2048 patterns to check the scalability of our system for large dataset. Figure 5 shows the reconstruction result under a hybrid parallel method with 7 MPI tasks and 4 OpenMP threads per MPI task. The top row shows the result of 256 by 256 images and the bottom row shows the result of 512 by 512 images. From left to right are original objectives that contain all high frequency information, widefield results with low resolution and reconstruction results with high resolution. This indicated that our parallel implementation could successfully reconstruct the high resolution result with high quality in terms of expected application outcome.

Performance Analysis

As we tried to parallel the whole reconstruction process, we would measure the execution time involving both gradient and covariance calculation. We considered the execution time of our sequential version as our baseline and calculated the speedup of our parallelized version accordingly. For the following performance result and analysis, if not specified, the results are from 256 by 256 images only.

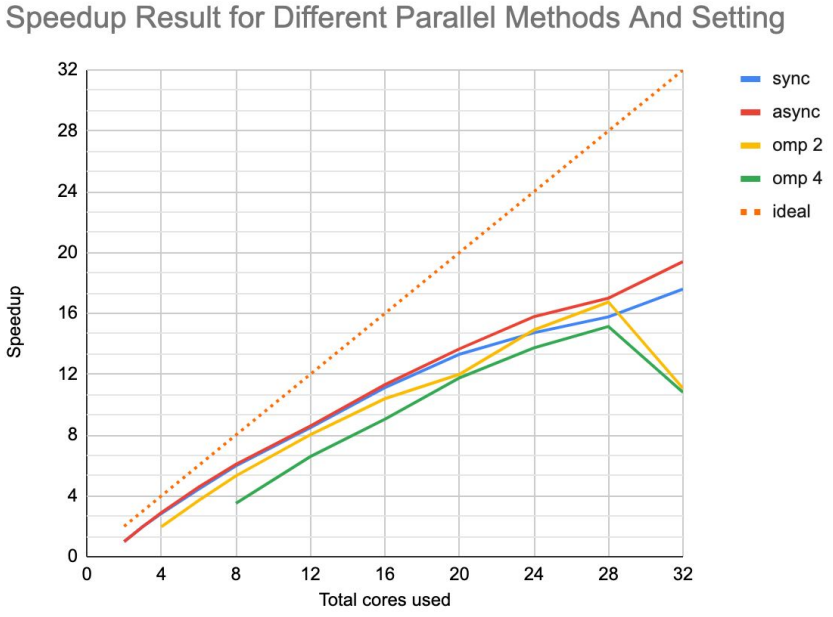

Figure 6. Speedup for Different Parallel Methods and Settings

Figure 6 shows the speedup of different parallel methods and settings in this project. *Sync* and *async* represent the pure MPI parallel tasks with different numbers of cores, while *omp 2* and *omp 4* represent the number of OpenMP threads for each MPI task. For example, when we have 16 cores used for *omp 2*, it means we launched 8 MPI tasks with 2 OpenMP threads for each MPI task. We have achieved our original goal to have 4x speedup, in fact 6.07x for asynchronized MPI, with 8 cores compared to the Matlab version. Besides that, we also encounter some unexpected behaviors.

We did not achieve the ideal perfect speedup for several reasons. In our master-slave model, the master machine will not do useful computation work. It only helps gather and distribute data to slave machines. In addition, as the number of cores increased, the communication cost also increased, especially for the covariance part where we needed to send image data back and forth. Moreover, if the master machine does not respond to the slaves in time, the slave machines will be idle waiting for instructions from the master.

- Synchronized vs Asynchronized MPI

# Cores	1*	2	3	4	6	8	12	16	20	24	28	32
Sync(s)	352.243	353.845	177.531	124.248	79.377	58.933	41.479	31.674	26.467	23.916	22.334	20.022
Async(s)	352.243	353.033	178.252	121.537	76.98	58.022	40.993	31.144	25.775	22.321	20.72	18.153

Table 1. Execution Time for Synchronized vs Asynchronized MPI Speedup over Baseline.
* Since we have the master-slave model for MPI, we can't run the code with 1 core. Time recorded in the table is our baseline.

Our asynchronized MPI implementation is generally a little bit faster than the synchronized version. However, the amount of speedup achieved over the synchronized version seems to be negligible. We believe that this happens because of our input data structure. During the simulation, each pattern contains the same number of speckles with different distributions, which leads to the microscopy system generating similar input data as shown in Figure 7. By applying the same set of operations (fconv, matrix) to those similar input data, we would have similar computational intensity among all the slave machines.

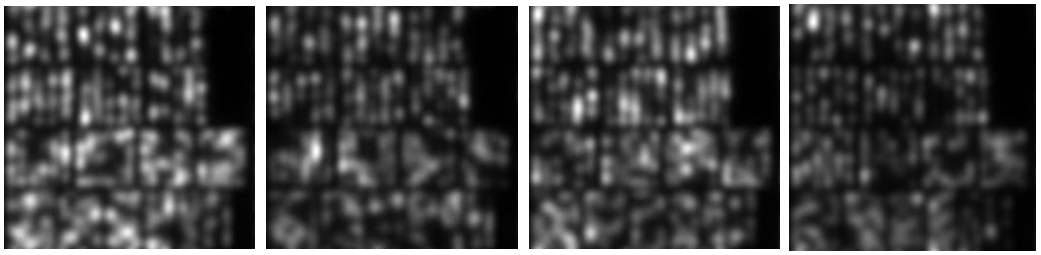

Figure 7. Sample Inputs Generated by Simulation(Normalized to 0-255 for jpg image show)

With static balanced work distribution, each slave machine would finish the computation at around the same time. In this case, the benefit of asynchronized MPI that slave machines do not need to wait for each other, diminishes with the similar computation time among slaves.

- Asynchronized MPI vs Hybrid

As shown in Figure 6, with the same cores used, Asynchronized MPI has better speedup compared to the Hybrid one. We initially expected that the Hybrid model could take the benefit of parallelizing the for loop when calculating the gradient and covariance. However, we missed the fact that we could only evenly assign OpenMP threads on each MPI task while no OpenMP operation needed on our master machine. Besides, when the slave machines send and receive messages, the OpenMP threads for the slaves will also be idle as well. In this situation, although we may benefit from parallelizing the loop, we suffer greatly from those idle threads, which could do work in the pure MPI setting.

- Experiment with larger input

As we mentioned, we also tried 512 by 512 images with 2048 patterns to see how our code can scale with larger data size. We reached similar results on large input with less than 8 MPI tasks for three parallel methods. However, we found that when we assigned more than 8 MPI tasks, we would have out-of-memory issues for some processors on Bridges. It made sense after we did some calculations of our problem size. Our input in total needs $512 \times 512 \times 2048 \times 8 = 4GB$ of memory. Our implementation keeps a local copy of all the

inputs for each slave machine to avoid data communication, and thus, when the processor scales, we may not have enough memory in hardware to store all the intermediate variables. (On Bridges, 1 node containing 28 cores has 128 GB of RAM.) Our proposed solution to this problem is that we could probably store those data in the filesystem and read from file when we need to use it. This can slow down our program as we would have additional file I/O's. We do not implement this solution due to time constraints and the scope of this project.

- Deeper Analysis

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 19.68     50.53    50.53                             __gnu_cxx::__enable_if<std::__is_scalar<double>::__v
double>(double*, unsigned long, double const&)
 14.20     86.99    36.45                             std::vector<double, std::allocator<double> >::operat
 10.62    114.27    27.28    97154     0.00     0.00  matrixScalarMul(std::vector<double, std::allocator<d
 10.57    141.42    27.15    39553     0.00     0.00  fconv2(std::vector<double, std::allocator<double> >,
  8.10    162.21    20.79    57984     0.00     0.00  matrixSub(std::vector<double, std::allocator<double>
  6.97    180.10    17.89    79106     0.00     0.00  fft2(std::vector<double, std::allocator<double> >, d
  6.39    196.50    16.40                             std::remove_reference<std::vector<double, std::alloc
std::allocator<double> >&>(std::vector<double, std::allocator<double> >&)
  5.49    210.59    14.09    39553     0.00     0.00  circshift_double(double*, double*, int, int, int, in
  5.32    224.25    13.66    38784     0.00     0.00  matrixEleMul(std::vector<double, std::allocator<doub
  3.44    233.09     8.85    59137     0.00     0.00  sumImage(std::vector<double, std::allocator<double>
  2.97    240.73     7.63    19200     0.00     0.00  matrixAdd(std::vector<double, std::allocator<double>
  2.93    248.25     7.52    39553     0.00     0.00  ifft2(double (*) [2])
  1.36    251.75     3.50    19584     0.00     0.00  matrixAbs(std::vector<double, std::allocator<double>
```

Figure 8. Gprof Result of Sequential version

From Figure 8, we can see that assigning value to vector and vector initialization took most of the time in our program, even longer than time required to run convolution. In order to achieve better performance, we could try to use float instead of double to save time for memory allocation and memory space in sacrifice of the precision and maybe convergence of the gradient descent. Meanwhile, we wrote our own naive version of some simple matrix operations like add, we did not change it to use the Eigen library since we believed that such change would benefit both the sequential and the parallel version of our code, but we decided to focus on how we could parallel the code and changes would benefit the parallel version only. We also did not use the MPI and OpenMP features available in FFTW and Eigen libraries for the same reason. We believed that if we utilized the two libraries better, we could speed up both the sequential and the parallel version.

REFERENCES

1. Ströhl, Florian and C. Kaminski. “Frontiers in structured illumination microscopy.” (2016).
2. Yeh, Li-Hao et al. “Structured illumination microscopy with unknown patterns and a statistical prior.” Biomedical optics express 8 2 (2017): 695-711
3. Zinkevich, Martin et al. “Parallelized Stochastic Gradient Descent.” NIPS (2010).
4. Zinkevich, Martin et al. “Slow Learners are Fast.” NIPS (2009).
5. https://dvicini.github.io/projects/bscthesis/vicini_bsc_thesis.pdf
6. http://www.fftw.org/
7. http://eigen.tuxfamily.org/index.php?title=Main_Page
8. https://www.openmp.org//wp-content/uploads/HybridPP_Slides.pdf

LIST OF WORK

Sequential Code:

1. Microscope Imaging Process Simulation(inputs): Yajie
2. Eigen, Boost and matrix operation: Peicheng
3. FFTW and convolution: Yajie
4. BSIM Reconstruction: Both

Parallel Code:

1. Synchronized MPI Coding: Both
2. Asynchronized MPI Coding: Both
3. Hybrid OpenMP + MPI Coding: Peicheng

Collecting Result:

1. Experiment on 256 by 256 images with 384 patterns: Both
2. Experiment on 512 by 512 images with 2048 patterns: Yajie