Analysis of OpenMP and Open MPI Parallelism on the SIFT Algorithm

By Alvin Shek and Hojun Byun

Summary:

We implemented two versions of parallel SIFT algorithm, one using OpenMP and the other using Open MPI, to match similar localized features between two images. We also analyzed the difference in performance of using OpenMP and Open MPI to parallelize the code. The algorithm was run on the GHC machines (eight 3.2 GHz Intel Core i7 processors, NVIDIA GeForce GTX 1080 GPU).

Background:

The SIFT algorithm is a feature detection algorithm used to to detect and describe local features in images. It can be used to find matching features across multiple images to describe how similar the images are.

Matching features across two images is a common task in many computer vision applications. For instance, everything from robots to the newest iPhone uses disparity estimation to generate depth images from 2D images, similar to how humans estimate depth through two eyes. This disparity estimation attempts to align two slight shifted images through feature matching. Motion capture and optical flow similarly attempt to match two images and identify the overall displacement between these matching parts of image.

Key data structures in this algorithm are:

1. Image data structure which is a 2D array with data values specifying how black/white each pixel is. We created our own Image() class with various C++ operators (copy, assignment, instantiation, and most importantly subtraction) to allow us to store images and perform high-level subtractions between them. We also created interface functions to store data into OpenCV data structures (cv::Mat) to take advantage of their existing write-to-file and display functionality.
2. Various arrays and vectors that are used in the intermediary steps of the algorithm such as storing keypoints
3. We created various C++ classes that implement each of these major steps using smaller helper functions with public and private data. For instance, our LoG class allows the user to call a high-level *find_LoG_images()* that performs all intermediate steps like blurring and shrinking.

At a high level, SIFT takes in a single image and outputs the same image with feature points marked on top of them. The algorithm can be broken down into a few steps:

1. Shrinking images to allow image matching to be scale-invariant.
2. Blurring the images through convolution to smooth out noise.

3. Finding the LoG(Laplacian of Gaussian) approximations to calculate gradients in the x and y-direction.
4. Use the gradients to find keypoints, which are designed to find features that are “corner-like”.
5. Find orientation of keypoints and apply inverse rotation. This explains why SIFT is invariant to rotations.
6. Generating and distinguishing features using keypoints from each image and match said features.

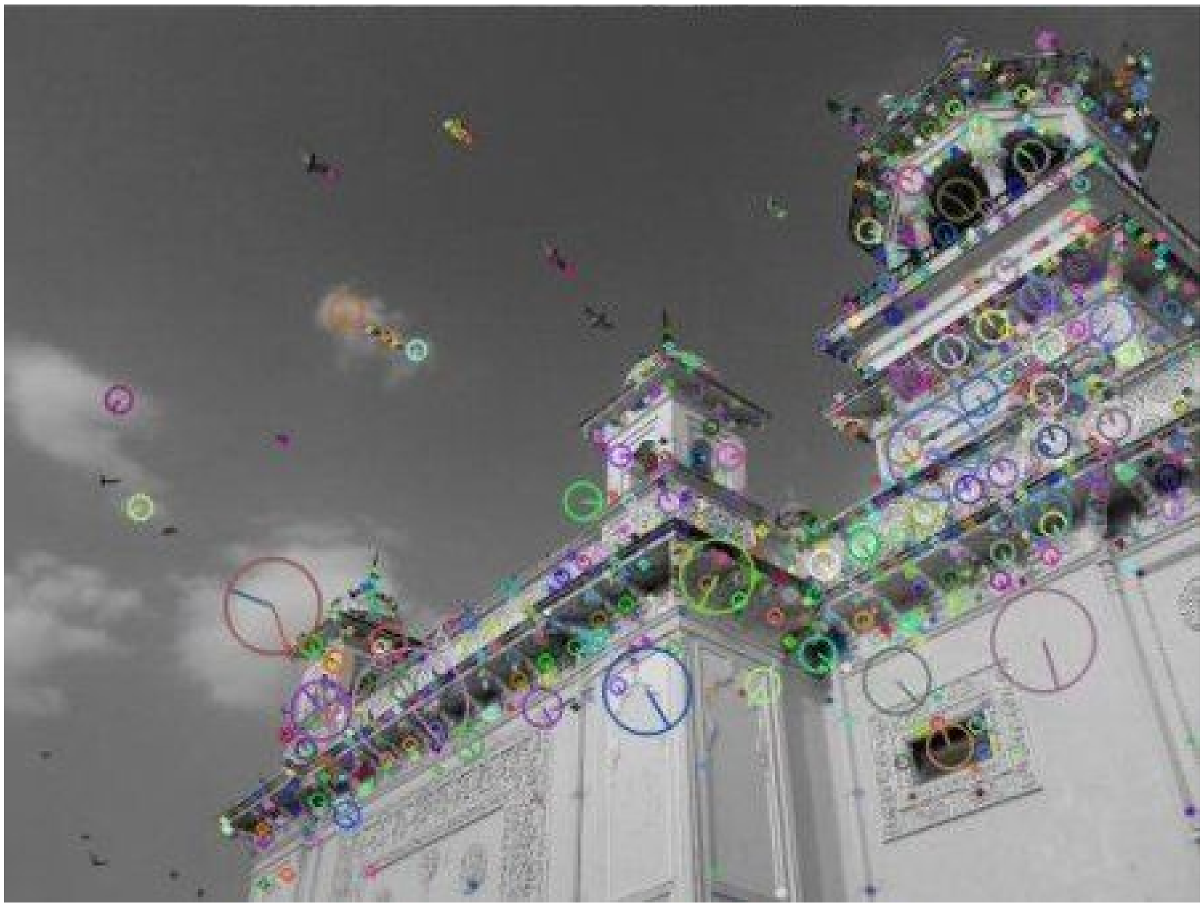
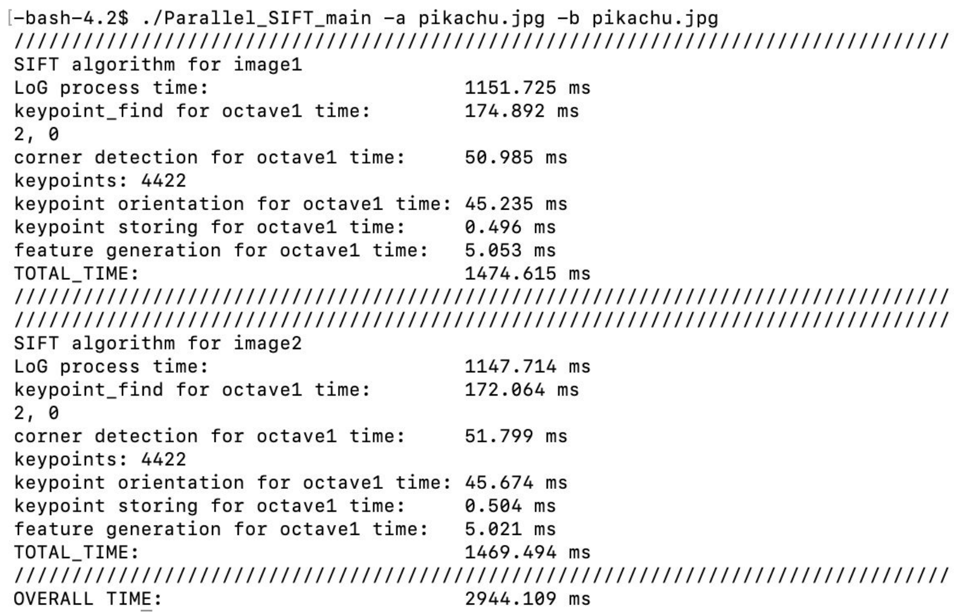


Figure 1. An example of the end product of the SIFT Algorithm, marking the relevant key points of an image.

Approach:

We parallelized the SIFT algorithm with both OpenMP and MPI by parallelizing the first three steps in the algorithm: convolution, the LoG approximations that call convolution multiple times, and finding of the keypoints. We chose to focus on parallelizing these two steps as we noticed that with the initial sequential implementation, a vast majority of the time was spent on these three steps as seen below:



```
[-bash-4.2$ ./Parallel_SIFT_main -a pikachu.jpg -b pikachu.jpg
/////////////////////////////////////////////////////////////////////////////////
SIFT algorithm for image1
LoG process time:                     1151.725 ms
keypoint_find for octave1 time:       174.892 ms
2, 0
corner detection for octave1 time:    50.985 ms
keypoints: 4422
keypoint orientation for octave1 time: 45.235 ms
keypoint storing for octave1 time:    0.496 ms
feature generation for octave1 time:  5.053 ms
TOTAL_TIME:                           1474.615 ms
/////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////
SIFT algorithm for image2
LoG process time:                     1147.714 ms
keypoint_find for octave1 time:       172.064 ms
2, 0
corner detection for octave1 time:    51.799 ms
keypoints: 4422
keypoint orientation for octave1 time: 45.674 ms
keypoint storing for octave1 time:    0.504 ms
feature generation for octave1 time:  5.021 ms
TOTAL_TIME:                           1469.494 ms
/////////////////////////////////////////////////////////////////////////////////
OVERALL TIME:                         2944.109 ms
```

Figure 2. Output of timed sections of the algorithm.

Sequential Algorithm:

We chose to make our SIFT algorithm from scratch as we wanted the option to inject parallelism of every step of the algorithm. Although we completed the algorithm and could generate relevant features of images, our implementation was not good that mapping out similarities between two different images. However, as this project is for a parallel computations and architecture class and not a computer vision class, we proceeded to make do and focus on the speedup of our algorithm.

Our benchmark test was of our sequential SIFT algorithm ran twice on the same image, which correctly mapped the features of one image to the same feature of the other image. Our benchmark

In order to pinpoint which functions that need to be optimized, we used the execution time of all the intermediary steps of the sequential algorithm that was shown previously. With this insight, we see that most of our focus would need to go towards the blurring/convolution function as that takes up majority of the execution time of the program.

Open MPI:

Diving into the specifics, we first parallelized image shrinking. When an image is shrunk in half, the total number of pixels is divided by 4. Each pixel of the shrunk image is an average of four pixels from the original size image. This is shown in the example below:

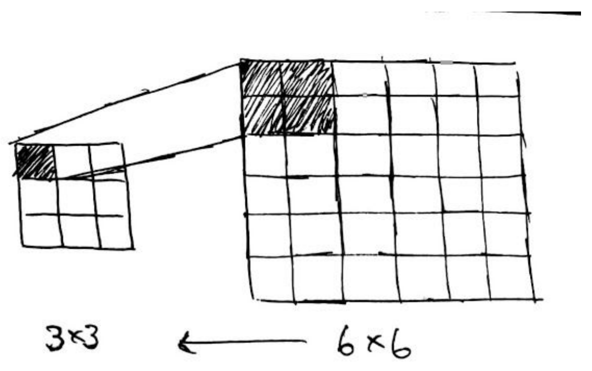


Figure 3. Example image of how scaling works.

We applied data parallelism to parallelize the shrinking operation for three scales: half, quarter, and eighth versions of the original image all in one clean function call by the user, *perform_shrinks(half, quarter, eighth)*. We allocated work in *allocate_shrink_work_mpi()* by calculating the total number of pixels for a downsized image and assigning an even number of pixels to each task. The processes would scale up these downsized indexes to access the original size image pixels. If there are M pixels and N tasks, process one would handle pixels [0, M/N - 1], process two would handle pixels [M/N, 2*M/N-1], and so on. Only the last thread would receive the remainder. Processes need to generate these assignments themselves since they need to know the index range of other processes' work to properly store their shared results. Of course, every MPI application requires developers to not only manually assign work to each process, but also handle message passing. Since each process needed a copy of all three shrinked images, each process would scatter its results to all other processes and gather all other processes' results.

We wrote helper functions to handle these two operations: *send_to_others()* and *receive_from_others()*, which loop through each process(besides itself) and perform non-blocking send and receive respectively. This use of non-blocking message passing calls for some synchronization, which is why we wrote a separate *mpi_barrier()* function that loops through each process and calls *MPI_Wait()*. Since a process shouldn't send and receive results from itself, the index of our MPI_Request* array corresponding to this entry would be undefined, causing segfaults when using *MPI_Waitall()*. Separating synchronization also allowed us to make parts of our program task-parallel.

Our original sequential shrink function was modified heavily to perform any scale of shrinking rather than multiple calls to shrink_half. To handle contiguous pixel assignment, we break down our nested for-loops over rows and columns into just pixel index. We created our own range typedef as a pair of integers to map the start and end of a process's index assignment.

Our next step at parallelizing was the Gaussian blur convolution. A blur convolution involves taking the weighted average of a pixel's neighbors including that pixel and storing the result into that pixel. Since a Gaussian distribution is continuous, the probabilities on the distribution never quite reach zero, but we discretize this as a binomial distribution so we can use a finite-sized N x N kernel for convolution. Our GaussianBlur class converts variances used in gaussian distributions to generate binomial distributions using Pascal’s triangle, and with this we can perform approximate blurring where the weight of all pixels must sum to one to not lose brightness of pixels.

$$\begin{array}{ccccccc|l}
 & & & & & & 1 & \sum \Longrightarrow 1 \\
 & & & & & \frac{1}{2} & \frac{1}{2} & \sum \Longrightarrow 1 \\
 & & & & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & \sum \Longrightarrow 1 \\
 & & & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} & \sum \Longrightarrow 1 \\
 & & \frac{1}{16} & \frac{1}{4} & \frac{3}{8} & \frac{1}{4} & \frac{1}{16} & \sum \Longrightarrow 1 \\
 & \frac{1}{32} & \frac{5}{32} & \frac{10}{32} & \frac{10}{32} & \frac{5}{32} & \frac{1}{32} & \sum \Longrightarrow 1 \\
\frac{1}{64} & \frac{6}{64} & \frac{15}{64} & \frac{20}{64} & \frac{15}{64} & \frac{6}{64} & \frac{1}{64} & \sum \Longrightarrow 1
\end{array}$$

Figure 4. Description of a binomial distribution in estimating gaussian distribution.

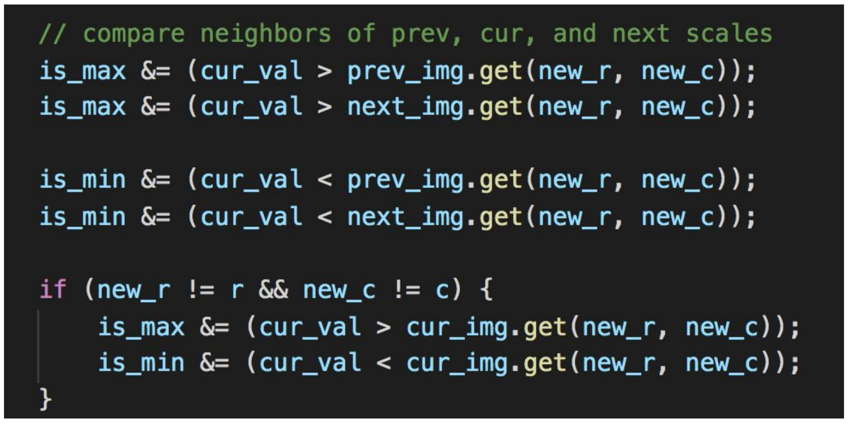
From the above description, a naive implementation would be to loop through all pixels in a kernel, but this takes O(M^2) for an M x M kernel. Rather, convolution can be separated into two separate one-dimensional convolutions for overall O(2M) computation. Our implementation performs vertical convolution first, and uses this to compute horizontal convolution. We parallized this by assigning dividing up rows and columns of an image to each process. The following summarizes the work done by each process:

1. Vertical convolution for assigned rows
2. Non-blocking send results
3. Non-blocking receive other processes’ results
4. Barrier ensuring all results are received
5. Horizontal convolution for assigned columns
6. Non-blocking send and receive
7. Barrier

The next step was to parallelize finding keypoints. Keypoints are crucial to comparing images: intuitively, we as humans compare entire objects in images, but SIFT compares specific pixels and some information about their surroundings.

The first step of keypoint-finding is to check whether a given pixel is an extrema. SIFT performs this check not only for a pixel’s neighbors, but neighbors of other images(which are different blurred versions). The same data-parallelism as image shrinking is applied here, where processes are assigned a certain range of pixels to perform this extrema check. In fact, our function to allocate work, *allocate_work_pix_mpi()*, is basically repeated three times for the shrinking work allocation. Our high-level send and receive helper functions make this parallelization easy.

While parallelizing another part, we realized that extrema finding has low arithmetic intensity: one simple inequality check per memory access. We thought we could speed up the gradient calculation process by combining this with the extrema-finding section by storing the accessed pixel values into local variables and using them for gradient and magnitude calculation. The below series of “get” operations demonstrates this low arithmetic intensity.



```
// compare neighbors of prev, cur, and next scales
is_max &= (cur_val > prev_img.get(new_r, new_c));
is_max &= (cur_val > next_img.get(new_r, new_c));

is_min &= (cur_val < prev_img.get(new_r, new_c));
is_min &= (cur_val < next_img.get(new_r, new_c));

if (new_r != r && new_c != c) {
    is_max &= (cur_val > cur_img.get(new_r, new_c));
    is_min &= (cur_val < cur_img.get(new_r, new_c));
}
```

Figure 5. Spinnet from the *get_maxes* function

However, we realized that this optimization doesn’t work because the gradient calculation step uses the results of the extrema-finding step, which masks out pixels that aren’t extrema. Overall, we stuck with the more simple idea of keeping gradient-finding separate. This step finds the gradient at each pixel and its corresponding magnitude, which are stored in arrays of the same size as the image. The same data-parallelism is performed here as shrinking.

One challenging part, however, was to perform sharing of keypoints. Keypoints are stored if both their horizontal and vertical gradients exceed a threshold, and thus the number of keypoints and overall amount of data shared between processes cannot be calculated beforehand. To solve this, we statically allocate an array of same size as image, but in each location store a keypoint’s 1-D index. An example array is shown below.

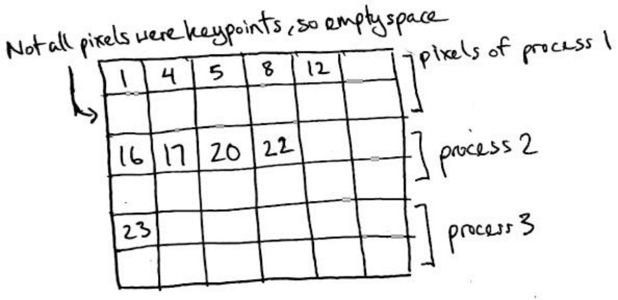


Figure 6.

With this strategy, static allocation of space to store keypoints can still be done. Not all space will be used, and a process can use the MPI_Get_count() function to actually check how many keypoints were sent by a process.

Overall, the last steps of keypoint orientation assignment involved the same data-parallelism approach. Overall, not much task-parallelism was achievable since one step would depend on the results of previous steps. Data parallelism was the natural approach to take since all pixels typically have the same operations performed on them. MPI overall had some nuances that we got around using some quick fixes. For instance, messages to the same recipient should have unique ids so they don't get mixed up, and this would happen in gradient, magnitude, and keypoint value-sharing. We fixed this with a simple enum to create three unique ids, one for each data group.

We had some attempts not mentioned above where we tried to parallelize memory loads and operations, but these ideas didn't work because overall produced minimal to worse speedup. First, a horizontal convolution has high cache efficiency, but a vertical has the opposite for row-major traversal of pixels. As we traverse across the columns in a row, we access completely new rows, some of which may be loaded into cache, but still be cold misses when reaching the same column of the next row. The image below shows this problem:

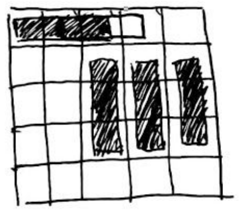


Figure 7. Vertical convolutions with row-major traversal has low cache efficiency.

To solve this, we tried a hacky approach where processes would be assigned entire columns rather than rows, and thus need to traverse across the rows for a single column. This sounds like bad cache performance, but when loading a block of data around a pixel into cache, we get the same cache reuse as when traversing horizontally and doing horizontal convolutions. We stored these results in a "flipped" array, where an index = column * rows_per_column + row. This way, storage of these results into an array has better cache efficiency in accessing contiguous memory. However, this approach ended up having many tricky indexing bugs especially in sending data between processes, and we ended up getting worse performance.

Even though we have better cache hit efficiency when storing values into a flipped array, we eventually need to store them into a correct row-major image. Here, we are left with the dilemma of whether to load row-major and store column-major or do the reverse, and both ways have the same issue of either loading or storage having bad performance. We also ultimately realized that even if we can reuse the values of adjacent pixels, we need to constantly write them to new variables since each convolution will scale those pixels with different weights.

OpenMP:

There are some OpenMP specific changes that had to be made. First, for OpenMP, we tried using the *#pragma omp parallel for* construct to parallelize the for various loops of the algorithm. This meant that in order to get the best parallelization possible, we needed to use loop unrolling to ensure that the OpenMP construct could parallelize the biggest loop it possibly can. Loop unrolling was necessary as parallelizing at the higher levels of the algorithm and function calls took higher priority than trying to inject parallelism on the lower ends. So we loop unrolled all of the functions that stood to benefit from having the annotation.

Second, we could inject some task-parallelism by calling some of the functions with the *#pragma omp parallel ... single ... task* or *#pragma omp parallel ... sections* annotations so that different threads can work on certain functions in parallel. Using annotations that injected task-parallelism was used on the higher-level functions.

Third, the LoG step required the usage of local vectors to store the x and y-gradients across pixels per convolution call, which could not be used if the convolution was broken up into multiple sections and used by several smaller convolution functions. So we had to create another shared vector called *temp_temp* that could serve as a temporary vector between all the different convolution sections. This shared vector was important because along the border of each convolution section, the x and y-gradients could not be accurately calculated if one convolution section did not have access to the calculation done on the edge of another convolution section. These convolution sections would then be called using the *#pragma omp parallel ... single ... task* annotation so that only a single thread would be able to work on a convolution section.

Fourth, we tried out several memory load methods such as loading blocks of image memory at a time to reduce cache misses but could not get that to work correctly by the deadline so we decided to drop it like we did for MPI.

For OpenMP, we followed a similar structure of making separate functions for shrinking instead of calling the shrink_half function multiple times on previous shrink_half outcomes. These *shrink_half*, *shrink_quarter*, and *shrink_eighths* would be called parallel to each other, but in series with the *create_blur* function per scaled-down versions of the images by grouping the shrink and blur functions with *#pragma omp section*. This ensured that each call to *create_blur* would not be hindered by calls of other shrink functions that the one it needed.

We also parallelized the convolution section by dividing up the workload of convolving an entire image into separate sections so that we can inject data-parallelism into the algorithm. We tried out different numbers of convolution sections to divide into so because cost of OpenMP's overhead of launching more threads to perform smaller convolutions in parallel vs. the performance gain from having those parallel convolutions were unknown. The net gain would need to be tested. Within these convolution sections, we also tried injecting *#pragma omp parallel for* annotations to see if the loops inside after unrolling them, which would have yielded speedup.

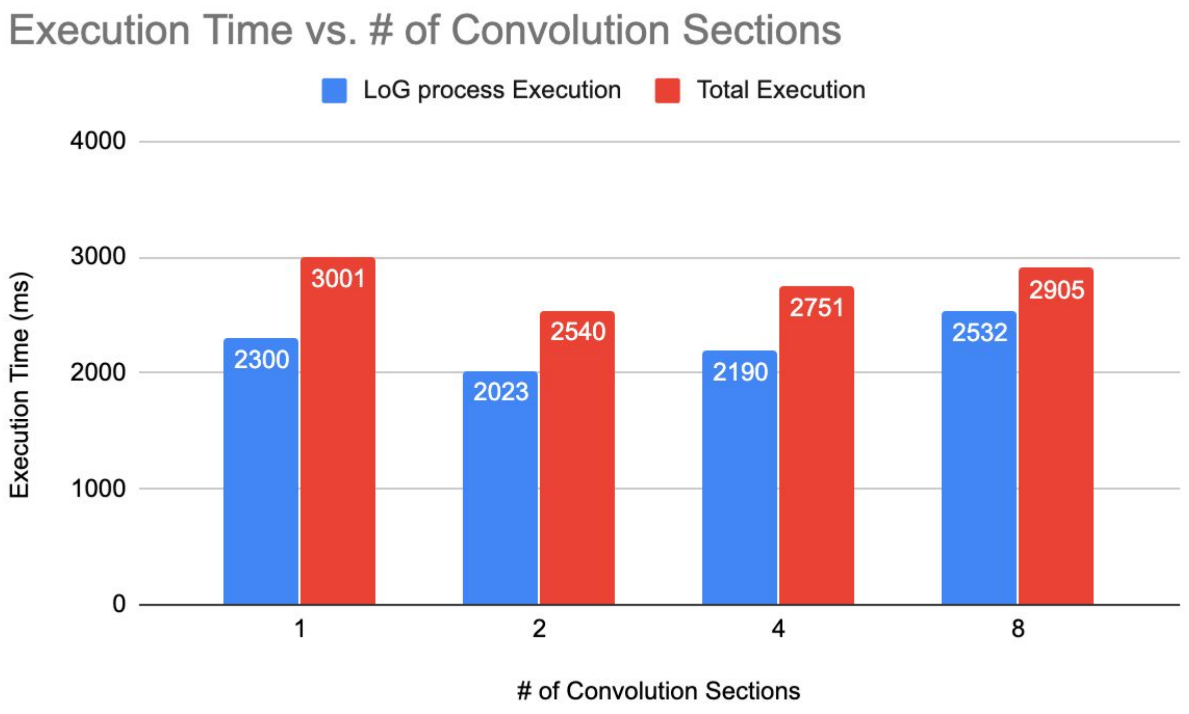
Results:

We set out to achieve at least 3x speed-up for both OpenMP and MPI implementations. However, we could only achieve this goal for the MPI version for certain functions with certain image sizes. The speed-up was measured by comparing the overall execution time (the time taken to run the SIFT algorithm on both images and the image comparison) of the sequential version to the two types of parallel algorithms.

OpenMP:

This is the average runtime of the code over 10 runs with different number of convolution sections:
(Note: This table shows the differences of execution using different number of convolution sections after all other forms of parallelisms have been applied such as using "omp tasks" to run the blurring algorithms that use convolution in parallel. This means that the # of convolution sections correspond to the number of threads used for that run.)

#of Convolution Sections	LoG process Execution time (ms)	Total Execution time	Speedup
1	2300	3001	1x
2	2022	2540	1.18x
4	2190	2752	1.09x
8	2532	2904	1.03x



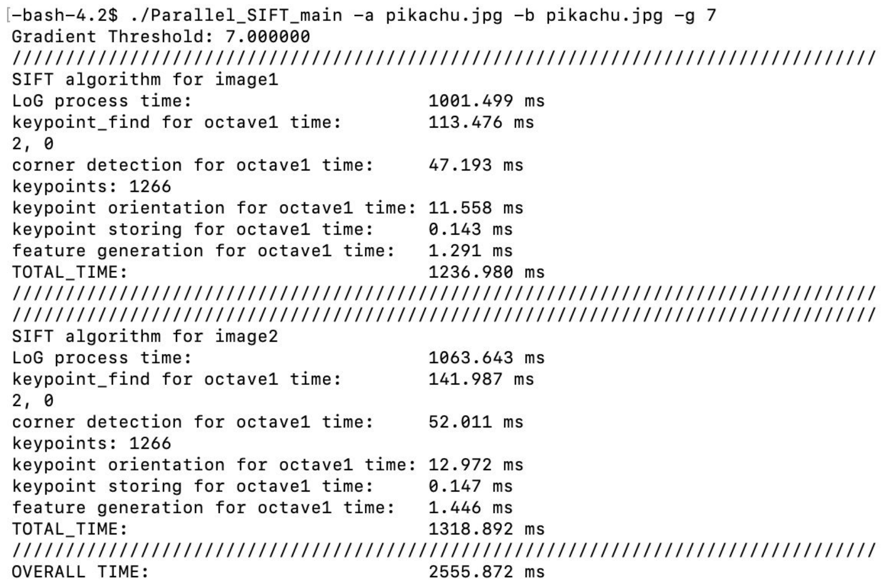
Using this table, we can see that the best speedup using OpenMP to create multiple sections of convolution is actually achieved by creating only two sections.

The speedup was also measured across different images of different sizes to see if the speedup of an implementation was depended on the image size
(Note: This table shows the differences of execution time of images of different sizes with the fully parallelized OpenMP algorithm using two convolution sections, using omp sections on blurring, and using omp parallel for to find keypoints.)

Image Size	Sequential Total Execution time (ms)	Parallel Total Execution time (ms)	Speedup
700 x 700	3160	2540	1.24x
1396 × 1414	12469	10567	1.17x
1000 x 800	6431	5407	1.18x

As seen here, there were no significant differences in speedup when using different images, so the image size had no impact on the speedup.

Here is the final time division of execution time



```
-bash-4.2$ ./Parallel_SIFT_main -a pikachu.jpg -b pikachu.jpg -g 7
Gradient Threshold: 7.000000
////////////////////////////////////////////////////////////////////////////////
SIFT algorithm for image1
LoG process time:                       1001.499 ms
keypoint_find for octave1 time:         113.476 ms
2, 0
corner detection for octave1 time:      47.193 ms
keypoints: 1266
keypoint orientation for octave1 time: 11.558 ms
keypoint storing for octave1 time:      0.143 ms
feature generation for octave1 time:    1.291 ms
TOTAL_TIME:                             1236.980 ms
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
SIFT algorithm for image2
LoG process time:                       1063.643 ms
keypoint_find for octave1 time:         141.987 ms
2, 0
corner detection for octave1 time:      52.011 ms
keypoints: 1266
keypoint orientation for octave1 time: 12.972 ms
keypoint storing for octave1 time:      0.147 ms
feature generation for octave1 time:    1.446 ms
TOTAL_TIME:                             1318.892 ms
////////////////////////////////////////////////////////////////////////////////
OVERALL TIME:                           2555.872 ms
```

Figure 8. Break down of the execution time of the final OpenMP implementation

The figure above shows that most of the speedup was from the LoG process by the parallel convolutions, with some contributions from the *find_keypoint* function. It still stands to show that there is much desired speedup of the LoG processes as the difference from sequential to parallel is still very small. More than 80% of the execution time is still being spent on the LoG processes. The other parts of the algorithm are not in need of much parallelization as parallelizing those values would only yield very small increases in performance.

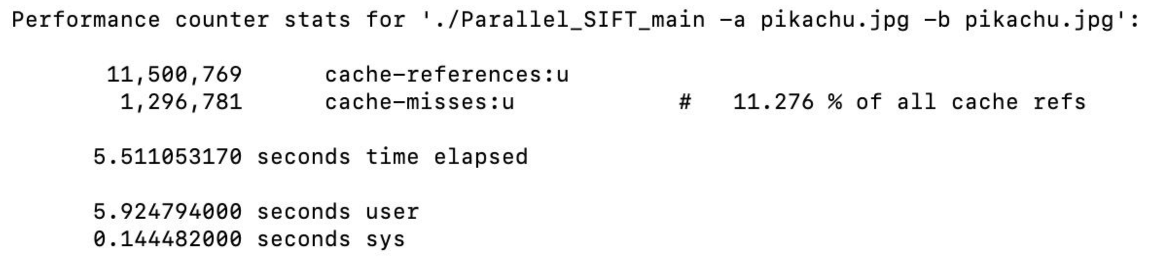
There is a lot left unfulfilled by our OpenMP implementation, allowing us to reflect on what went wrong and what is hindering speedup. Here are several factors that limited our speedup with OpenMP:

1. Too much overhead.

Ahmdahl’s law explains that the maximum speedup that we can get is determined by not only how much a program can be parallelized, but by the sequential portion of the program that cannot be parallelized. This means that our speedup is bound by the slowest part of the program and in our case that is for the convolution section. More specifically, the speedup is bounded by the execution time of the convolution done to the original image, which is four times bigger than the second biggest image that has convolution done to it which is the half-scaled image. So the strategy of breaking up the convolution of the original into several different sections was necessary, but the actual result says otherwise. By breaking up the convolution into several sections, the program’s overhead of launching more threads to convolve those sections increases, diminishing the amount of speedup that is possible. Even though there is no synchronization required as each thread would be editing different sections of a shared data structure, the overhead from launching several threads into doing the convolution limit the speedup and even cause a decrease in performance. This is evident on our experiment with using different number of convolution sections, which correspond to the number of threads being used. The more we increased the number of sections, the worse our speedup became.

2. Cache misses

Through the use of “perf”, a performance measuring tool, we learned that our OpenMP program has around 13% of all cache assess being cache misses on average. This did not bode well for the section algorithm that relied heavily on its memory accesses and in the end became a bottleneck for the speedup.



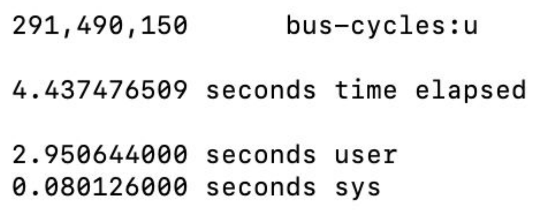
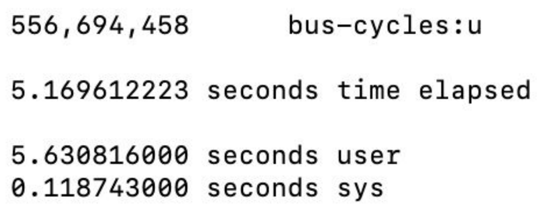
```
 Performance counter stats for './Parallel_SIFT_main -a pikachu.jpg -b pikachu.jpg':

        11,500,769      cache-references:u
         1,296,781      cache-misses:u            #   11.276 % of all cache refs

       5.511053170 seconds time elapsed

       5.924794000 seconds user
       0.144482000 seconds sys
```

Figure 9. Output of a pref examining cache references and cache misses

3. Memory Traffic

We also used perf to determine how much the bus was being used. Here are the results:

Sequential:	OpenMP:

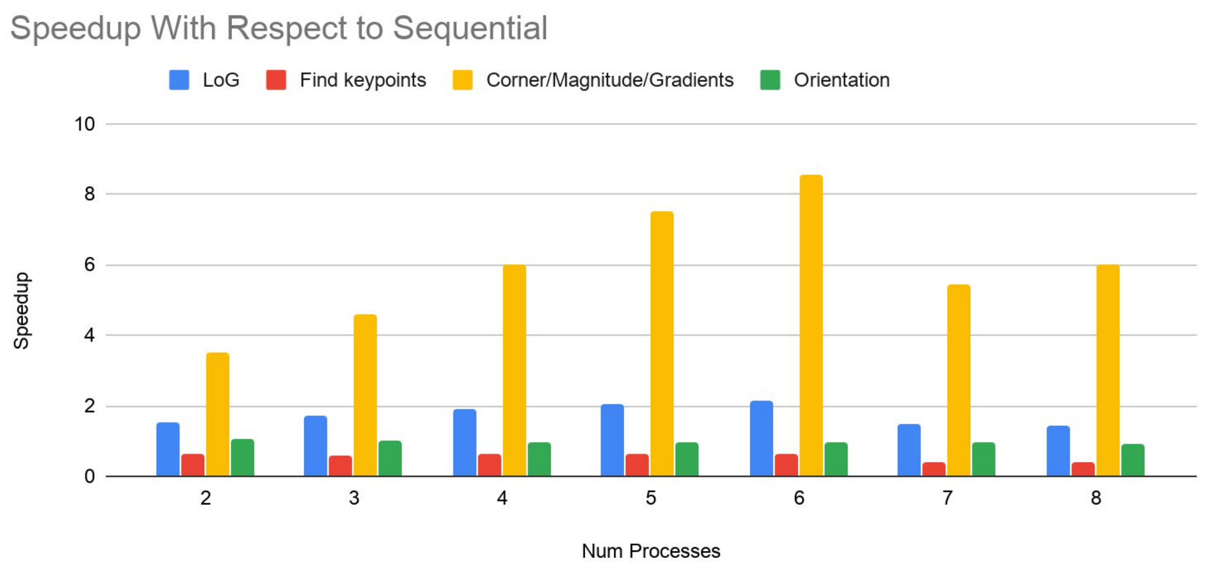
 `291,490,150      bus-cycles:u`
`4.437476509 seconds time elapsed`
`2.950644000 seconds user`
`0.080126000 seconds sys`
	 `556,694,458      bus-cycles:u`
	`5.169612223 seconds time elapsed`
	`5.630816000 seconds user`
	`0.118743000 seconds sys`

As seen here, the sequential implementation around half the bus cycles than the OpenMP version, which implies that it has half the amount of bus traffic than the OpenMP version. This could explain the poor performance as the amount of bus cycles indicate that the memory traffic is also a bottleneck. The more traffic on the bus, the need for memory coherency traffic also increases between cores. While the CPU used in GHC machines uses a large inclusive shared L3 cache that filters coherency traffic between cores, having double the amount of memory traffic would have a severe hit to the performance of the OpenMP version.

4. Sequential Nature
   Throughout the algorithm, there exists sections of code that are sequential by nature. For example, all the steps 3-5 of SIFT are all sequential because they rely on the output of the previous step to generate their outcomes. In the *find_corners_gradients, find_keypoint_orientations, store_features, store_keypoints* functions in the file that deals with finding keypoints and generating features out of them, all of those functions needed to have specific ordering in which they are called, eliminating the possibility of using *#pragma omp parallel task* or *#pragma omp parallel sections*. Even further, these functions needed to have the arrays/vectors they edit in their for loops be in a specific order to keep correctness from sequential to parallel, eliminating the possibility of using *#pragma omp parallel for* as well.

Open MPI:

Overall, we achieved our target 3x speedup for certain functions with certain image sizes.



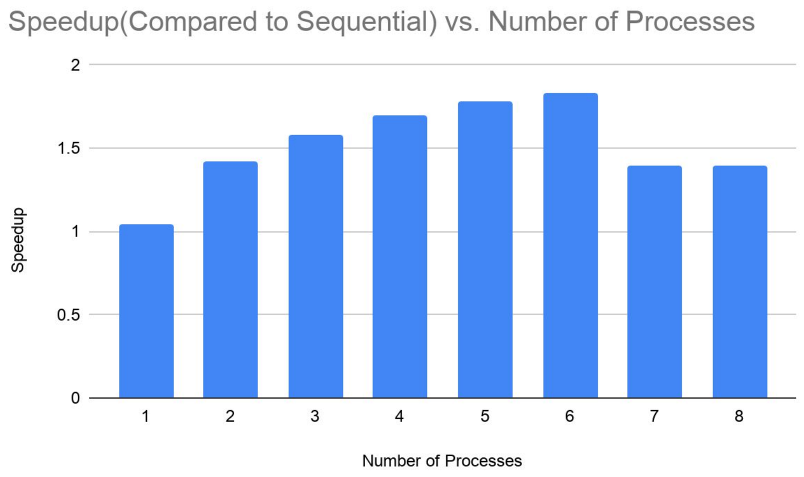
The above chart shows speedup of various functions compared to the sequential version(single CPU) as a function of number of parallel processes. The magnitude and gradient calculation function achieved high speedup whereas other functions like find finding keypoints actually performed worse.

Speedup also varied with image size as the amount of work per thread increased.
For example, here are the speedups (relative to sequential) of the blur convolution function when run on different sized images of the same image:

Image Size	Convolution Blur Speedup
Full	2.16
Half	2.09
Quarter	2.07
Eighth	1.94

As you can see above, while minor, blur speedup still decreases as image size decreases. This is caused by the decrease work per thread and thus time savings as the ratio of work to setup overhead decreases. Such overhead includes assigning all the work to other tasks as well as the communication overhead of sharing results between tasks.

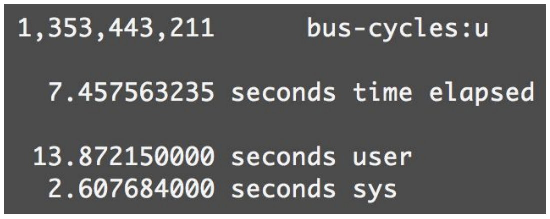
The communication overhead becomes clear when we compare speedup to number of processes. Much of the functions we parallelized had a low arithmetic intensity: simply comparing pixel values and performing simple subtraction operations. Due to the high overhead of communication, even with non-blocking, our program could only gain speedup with number of processes until a certain point, as shown in the graph below:



Here above, we can see that speedup increases quickly initially from one to two processes, but starts to slow in growth with number of processes. Eventually, at six processes, speedup is maximized before speedup suddenly drops at seven and eight threads.

Also looking at our first graph of speedup for various functions, we noticed that our parallel implementation of keypoint finding actually performed worse than the sequential by around 50%. This is disappointing since this was one of our minor attempts at task and data parallelism where we perform extrema finding between two sets of three images. We allow the processes to immediately start on the next extrema finding after performing non-blocking send and receive of the previous set's results. We speculate that this is caused by the low arithmetic intensity as we are only performing an inequality check for every memory load of a pixel from an image. With the memory overhead of assigning work, the parallel version would overall perform worse.

We also used perf to examine the memory traffic of our MPI version as well, and in this case realized that there was a high communication overhead. But compared to the OpenMP version, the bus communication is part of using a Message Passing Interface. While we do not have what constitutes a "good amount of communication through the bus" that indicates efficient bus traffic amount, we feel that having six times the amount of bus traffic is not a good sign and definitely has a lot of unnecessary information movement.



```
1,353,443,211      bus-cycles:u

  7.457563235 seconds time elapsed

 13.872150000 seconds user
  2.607684000 seconds sys
```

Overall:

Comparing the two different methods we used to parallelize the code, we much prefer using Open MPI than OpenMP. This is because OpenMP felt like a black box that we put our code into, hoping to get some amount of speedup without much of an explanation as to why certain techniques failed to achieve the results that we wanted. Open MPI yielded much more consistent results and lead to better iterative progress between implementing types of parallelism of the functions. The ability to inject parallelism through pointers and messages made for better control over our code that OpenMP could not offer.

References:

- *aishack.in*, aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/.

- Mordvintsev, Alexander, and Abid K. Revision. *Introduction to SIFT (Scale-Invariant Feature Transform)*, opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html.

- Kumar, R. , Muknahallipatna, S. and McInroy, J. (2016) An Approach to Parallelization of SIFT Algorithm on GPUs for Real-Time Applications. *Journal of Computer and Communications*, **4**, 18-50. doi: 10.4236/jcc.2016.417002.
- "Computer Vision: Feature detection and m." UW CSE vision faculty, courses.cs.washington.edu/courses/cse455/09wi/Lects/lect6.pdf?fbclid=IwAR3F3Vs3sK0a2BtInWO8M5_NFSknCf7q857_rewX8qFlzxqN-uzb4BU0gyo.

Work done:

Hojun 45%, Alvin 55%

Hojun	**Alvin**
Sequential pseudo code	Sequential implementation
Sequential debugging	Sequential debugging
OpenMP implementation	Open MPI implementation
Open MPI debugging	Open MPI debugging
Report	Report