Adriel Sumathipala
12/19/2018
CPSC 424: Parallel Programming Techniques

Parallelizing the Mean Shift Algorithm

Introduction:

The mean shift algorithm is a technique for obtaining the probability density maxima in a non-parametric feature space. Developed in 1975 by Fukunaga and Hostetler, the mean shift is an unsupervised machine learning technique and has applications in cluster analysis, image segmentation, and appearance-based image tracking [1]. The mean shift algorithm is used for image processing tasks such as identifying the contours of a building, segmenting tumors on medical images, and classifying objects in photos [8]. The mean shift algorithm is also used in video analysis to track moving objects and people in real-time [7].

The mean shift algorithm uses a generalized kernel density function to estimate the gradient of the probability density function at a discrete point using nearby data points. The mean shift algorithm uses gradient ascent to iteratively cluster the discrete data and find the maxima of the estimated probability density function. After convergence, the algorithm clusters the discrete data points into labeled clusters that represent the modes of the estimated density function. Unlike the traditional k-means algorithm, the mean shift algorithm does not require a pre-specified number of clusters, uses a single tunable bandwidth parameter that controls the size of the Parzen window, and works in arbitrary non-parametric feature spaces.

Algorithm Overview:

Given a set of finite, discrete data points S located within an p-dimensional Euclidean space, the initial probability density underlying the generation of these data points is unknown. Estimating this probability density is a common problem in non-parametric statistics and the mean shift algorithm relies on non-parametric methods, specifically the kernel-density estimation function and Parzen window, to obtain this estimated density.

To estimate the density of S, which is comprised of a set of n i.i.d. (identically, independently distributed) p-dimensional vectors $X_1, X_2, \ldots, X_n$. Then suppose we have the multivariate kernel density estimator, as presented by Cacoullos, where h is the bandwidth parameter controlling the Parzen window [2]:

$$\hat{f}(X)_p = \frac{1}{nh^p} \sum_{i=1}^{n} k(\frac{X - X_i}{h})$$

The kernel function $k(Y)$ has the following properties (Image Source: [1]):

$$\sup_{Y \in R^n} |k(Y)| < \infty$$

$$\int_{R^n} |k(Y)| \, dY < \infty$$

$$\lim_{\|Y\| \to \infty} \|Y\|^n k(Y) = 0$$

$$\int_{R^n} k(Y) \, dY = 1$$

Several kernel functions can be used in the mean shift algorithm including the Epanechnikov, Gaussian, and flat kernels. In this paper, the Gaussian kernel will be used which is of the form:

$$K(X) = \frac{1}{\sqrt{2\pi}}^n \exp(\frac{-1}{2} X^T X)$$

Broadly speaking, the mean shift algorithm will perform a gradient ascent on the contour of the estimated probability density function. To do this, the gradient of the density estimation function is taken:

$$\nabla \hat{f}(X)_p = \frac{1}{nh^p} \sum_{i=1}^{n} K'(\frac{X - X_i}{h})$$

To compute the solution to this gradient function, it is set equal to 0, which then yields:

$$\sum_{i=1}^{n} K'(\frac{X - X_i}{h})X = \sum_{i=1}^{n} K'(\frac{X - X_i}{h})X_i$$

Finally, the solution for the gradient ascent is reached, where $g(X) = -K'(X)$:

$$X = \frac{\sum_{i=1}^{n} g(\frac{X-X_i}{h})X_i}{\sum_{i=1}^{n} g(\frac{X-X_i}{h})}$$

As previously explained, the mean shift algorithm works by finding the maxima of an estimated probability density function in a non-parametric feature space. To find these maxima, the data points are iteratively shifted by the weighted mean of the density function, which is the solution to the gradient ascent as found above. This iterative shift is can be notated as the mean shift function, $m(X)$:

$$m(X) = \frac{\sum_{i=1}^{n} g(\frac{X-X_i}{h})X_i}{\sum_{i=1}^{n} g(\frac{X-X_i}{h})} - X$$

The mean shift algorithm then sets $X \leftarrow m(X)$ for each iteration until $m(X)$ converges. Convergence is reached once the data points no longer move by a pre-selected distance computed using the Euclidean norm. Convergence of the mean shift algorithm is generally sufficient for most applications, but has not been proved in all cases. However, convergence has been proven in high-dimensions with a finite number of data points for the Gaussian kernel, which makes it a suitable algorithm for the clustering applications demonstrated in this paper [3].

Time Complexity:

When it comes to the implementation of the mean shift algorithm, the worst-case time complexity of the algorithm must be considered. The mean shift algorithm must first iterate through all n points to shift each point, which yields an $O(n)$ time operation. Next, when shifting each point the weighted mean of the kernel density function must be computed, which is also an $O(n)$ operation. It is conceivable to improve this process by considering only the k-nearest neighbors of each point (see Future Work section). If the k-nearest neighbor lookup process is implemented using a k-d tree, the time complexity of this operation is then $O(k \log n)$, or more simply $O(\log n)$ as k is constant. An alternative approach to this process would be to use a λ-ball, wherein each point must obey the following inequality to be considered when computing the weighted mean: $\| X_i \| < \lambda$. This improves the $O(n)$ operation by a constant factor by excluding certain points when computing the weighted mean. Finally, this process must be repeated T times until convergence has been reached. This means the entire mean shift algorithm is of time-complexity:

$$O(Tn^2)$$

An improvement on this worst-case time complexity, would be to use the *k*-nearest neighbors approach. This would involve constructing a balanced k-d tree, which is an *O(n log n)* operation, to enable logarithmic lookup time. If using this k-d tree approach, the algorithm is of time complexity:

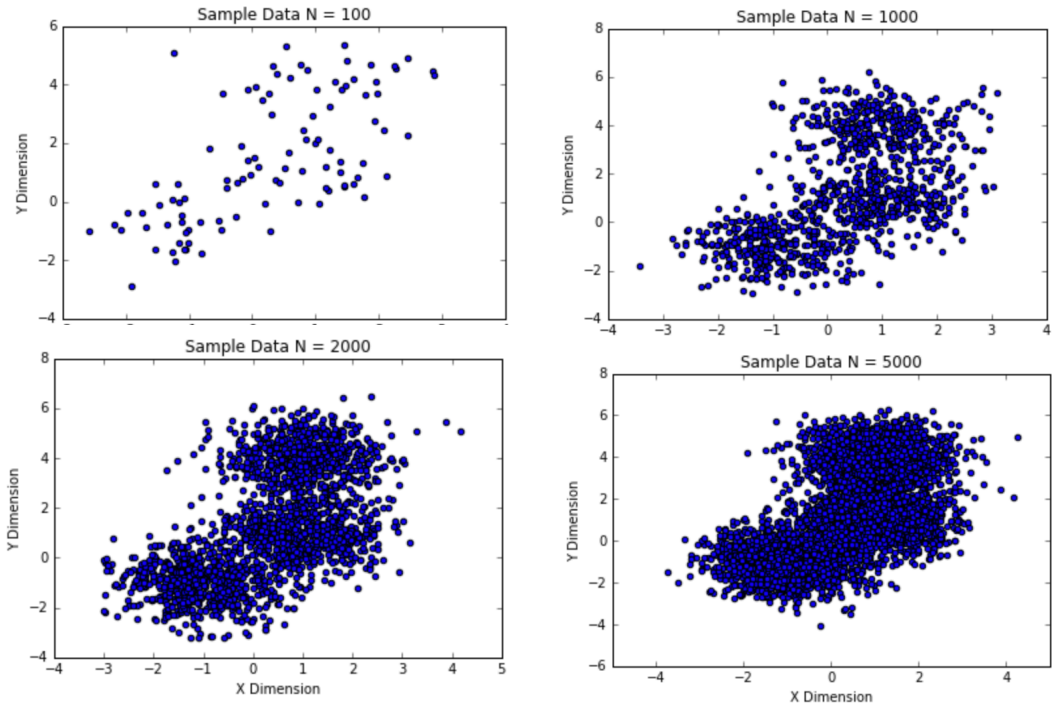
$$O(Tnk \log n)$$

However, when working in high-dimensions, the lookup time of a k-d tree will tend to *O(n)* as a result of the *curse of dimensionality* [4]. Thus, in high-dimensions, the k-d tree mean shift method will tend towards the general $O(Tn^2)$ time complexity for the mean shift algorithm.

Mean Shift Implementation:

The mean shift algorithm implemented in this paper assumes the worst possible time-complexity by iterating through each point, computing the weighted mean using all other *n* data points, and repeating this process until convergence. Speedups such as k-d trees, coarse seed binning, or a λ-ball were not employed and could be added in later development (see Future Work). Serial Python code for implementing the mean shift algorithm was adapted from M. Nedrich [5].

The general algorithm contained in the serial Python code was written in C for performance improvements and to enable parallelization via C's Open MP and MPI standard libraries. The serial C code is contained in `serial.c` and uses functions from `util.h` and `shift.h`. Randomly generated, two-dimensional data was created in Python, using the scikit-learn and a fixed seed for reproducibility. This data was written to a file read in by the C program, ensuring the Python and C programs used the same data. The Python implementation of the mean shift algorithm is contained within the files `mean_shift_runner.py, mean_shift_utils.py, mean_shift.py, and point_grouper.py`. Visualizations were made using the Matplotlib library in Python.



Randomly Generated Data with 3 Clusters
where $N = 100, 1000, 2000, 5000$

All Python code was run in the interactive Jupyter Notebook file, `Mean Shift Serial.ipynb`. The Jupyter Notebook session was hosted on a `bigmem` partition node on Yale's Omega cluster with 128GB of RAM and an Intel Xeon E5-2650 processor with sixteen cores. The notebook was accessed via an SSH pipe and internet browser. The `MeanShift()` function in the widely-used scikit-learn Python library was used as an additional comparison to the parallel methods developed in this work [6]. Python was often used as a comparison in this paper as it remains a universally popular language within the data science and machine learning communities. In this paper three parallel programming paradigms were applied: 1.) a single-node multithreaded implementation using Open MP, 2.) a multi-node implementation using MPI, and 3.) a hybrid multi-node, multithreaded implementation using both Open MP and MPI. All code was run on Yale's Omega clusters on the `-day` and `-bigmem` partitions and all C implementations described in this paper can be compiled using the included Makefile.

Open MP: Multithreaded Parallelization of the Mean Shift Algorithm

The first parallel implementation of the mean shift algorithm utilizes multithreading to boost performance. This parallel program was implemented using the

Open MP library and by modifying the `serial.c` program. All Open MP experiments were carried out on an Omega node with 128GB of RAM and a sixteen core processor.

Loop Parallelization:

The mean shift algorithm is not easily parallelizable, as iterations of the algorithm depend on previously shifted values to achieve convergence. If the iterations were independent, this would be an embarrassingly parallel problem. However, two parts of the algorithm can be parallelized using multithreaded programming. The first part that can be parallelized is the computation of the weighted mean given a discrete data point. This part of the program must iterate over all the data points, compute the Euclidean distance, weight each data point using the Gaussian kernel, and sum the weighted data points and individual weights. This portion of the code can be parallelized using the loops directive in Open MP. The for loops directive has been implemented in the `omp_shift()` function in `shift.c` and this function is called in the `for-openmp.c` file.

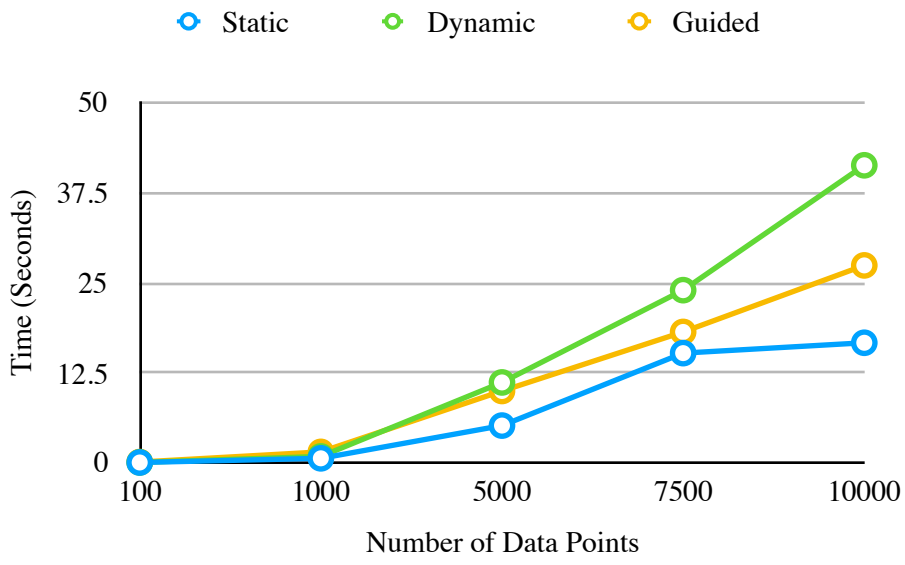
However, there exists two significant limitations to using the loop directive in the mean shift algorithm. The first is that the number of iterations of the for loop cannot be computed at compile time, because the number of iterations required for convergence is unknown. Open MP loops work the fastest when the number of iterations is known at compile-time rather than computing this on-the-fly at runtime. The second limitation of the loops parallelization is that sums of the weighted data points ($\Sigma\ \text{weight}_i \cdot X_i$) and weights themselves ($\Sigma\ \text{weight}_i$) are shared variables and need to increase during each iteration of the for loop. One simple method of ensuring these values are calculated correctly is to insert a critical region in the for loop, ensuring that only one thread updates the weights at a time. However, this will lead to greater overhead to manage write access across threads and will essentially serialize this part of the code by allowing only a single add operation at a time. A better solution, implemented in this work, would be to make private sum variables for each thread, parallelize the for loop, and have a critical region after the for loop to add the private sum variables from each thread to the shared sum variable, thereby performing a reduction operation. This leads to much greater parallelization across the threads, as the threads can compute their private sum values in parallel.

After implementing the looping paradigm describe above in `for-openmp.c`, the optimal number of OMP threads was found via experimentation on a 10,000 2-D point dataset:

Number of Threads	Time (Seconds)
1 Thread	193.4
2 Threads	97.5
4 Threads	49.6
8 Threads	27.0
16 Threads	16.7

As seen above, the best performance was achieved by using sixteen threads, giving each core its own thread. To further optimize performance, three different scheduling methods were tested—static (equal, round-robin distribution), dynamic (default chunk size of 10), and guided:

Number of Points	Static	Dynamic	Guided
N = 100	0.079711	0.093094	0.142414
N = 1000	0.657626	0.919810	1.550554
N = 5000	5.199637	11.184161	9.996244
N = 7500	15.249079	23.981243	18.187671
N = 10000	16.685938	41.300936	27.416899



The static scheduling performed and scaled the best. This is because static scheduling requires the least amount of overhead and the load balance is even across loop iterations. In the case of the mean shift algorithm, each loop iteration requires roughly the same amount of work, and thus dynamically assigning loop iterations requires greater overhead without leading to improved load balancing. From the chart, is also apparent that the mean shift algorithm is an $O(n^2)$ operation. Dynamic scheduling scaled the worst because the chunk size was fixed at 10, meaning the program had to distribute more loop iterations, leading to greater overhead as n grew.

Task Parallelization:

With regards to multithreaded programming, the second way to parallelize the mean shift algorithm is to parallelize the task of computing the weighted means at each data point. While iterations of the algorithm are not independent, computing the mean shift of a single point during a given iteration is an independent computation. Thus, the mean shift algorithm can be parallelized using the task paradigm in Open MP. The task parallelization strategy is as follows:

1.) A parallel region (with proper shared and private variables) is created outside the while loop, which contains the stopping condition for convergence.

2.) A single thread then generates tasks for the given iteration. Each task computes the mean shift for a single data point and each task is an $O(n)$ operation. These updated data points are stored in a temporary buffer.

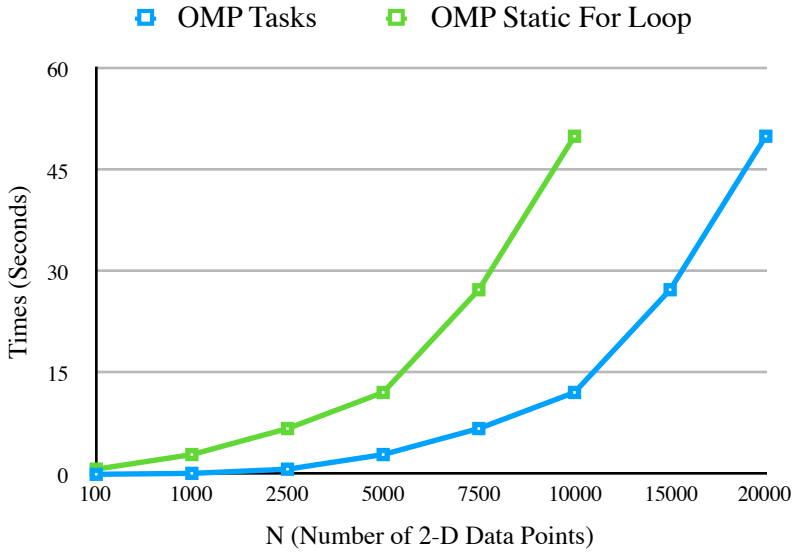
3.) There is a `taskwait` condition at the end of every iteration to ensure the mean shift at every point has completed before moving on to the next iteration.

4.) The pointers to the temporary buffer and buffer containing points to be shifted are switched and task generation is repeated until convergence.

The resultant program can be found in `tasks-openmp.c`. As with the loops program above, the first step was to find the optimal number of threads. The tests below were computed on a dataset of 10,000 2-D points. As we can see below, the optimal number of threads was sixteen, which lead to the greatest CPU usage and parallelization:

Number of Threads	Time (Seconds)
1 Thread	192.5
2 Threads	96.63
4 Threads	48.35
8 Threads	24.21
16 Threads	12.15

Using sixteen threads, the program was tested for scalability. The tasks parallelization strategy outperformed the static for loop parallelization strategy as seen below. This is because the tasks parallelization gives each thread a greater amount of work than the static for loop parallelization. The task parallelization requires each thread to do $O(n)$ work, while the static for loop program assigns each thread $O(n / p)$ work, where p is the number of threads. In certain cases, such as GPU matrix multiplication, assigning threads more work can improve performance by reducing the overhead required to distribute work across the threads.



Loops and Tasks:

The above approaches each parallelized a different part of the mean shift algorithm—the loops program parallelized the computation of the weighted means and the tasks parallelized the computation of mean shifting each point. These parallelization techniques are not mutually exclusive and address different components of the mean shift algorithm. Thus, the tasks and loop parallelization approaches can be combined as implemented in `for-tasks-openmp.c`. Loop parallelization was implemented in this

program using static scheduling, the best scheduling routine as shown earlier. As usual, the optimal number of threads was determined to be sixteen:

Using sixteen threads, the scalability of the program was tested. Below, it is clear the program using only tasks outperforms the program with both tasks and loops. This is because the program with only tasks has far less overhead than the program with both tasks and loops. Because the number of loop iteration cannot be computed at compile-time, the loop parallelization requires a substantial amount of overhead. Thus, the overhead of managing allocation for both loops and tasks at runtime outweighs any performance gains from increased parallelization.

MPI: Multi-Node Parallelization of the Mean Shift Algorithm

Multi-node parallelization works by distributing computation across several nodes via a high-speed data network. In this work, the Message Passing Interface (MPI) was investigated as a means to parallelize the mean shift algorithm. The parallelization strategy for the multi-node paradigm was as follows:

1.) Each node, belonging to a cluster of p nodes, reads in a local copy of the entire dataset.

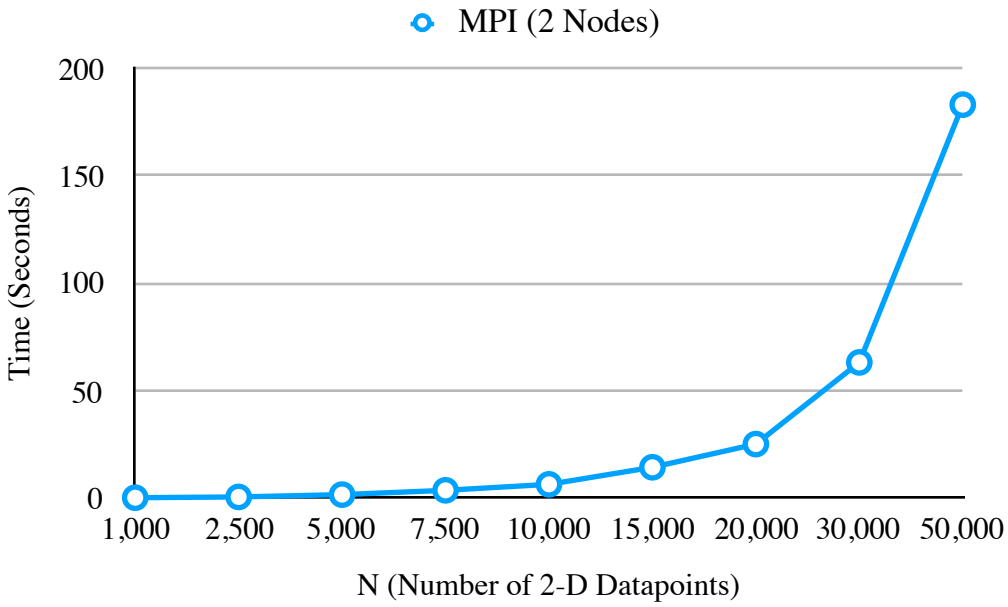
2.) The root node creates a second copy of the entire dataset. The root node then partitions these n data points across the p nodes into n / p chunks using the Scatterv collective operation.

3.) Each node than works on mean shifting its chunk of the overall data and the weighted means of the kernel density function is then calculated using the entire dataset. This approach works if the dataset is i.i.d.—if the data are not, convergence (regardless of implementation) is not guaranteed. Furthermore, if the data are not i.i.d. this means there is no underlying probability density function, which is the entire basis for the mean shift algorithm.

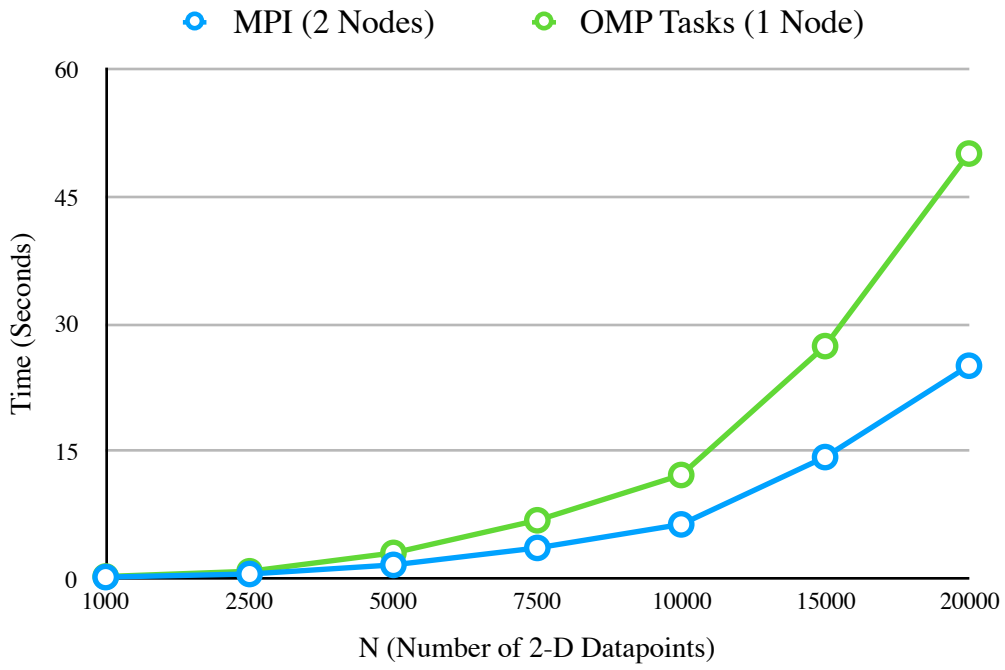
4.) Once convergence has been reached across all nodes, the shifted chunks are then collected using Gatherv collective operation on the root node. The points are then grouped into clusters and labelled as usual.

This approach was implemented in the file `mpi.c` and can be deployed using `mpi.sh`. This program was tested on two nodes, each with 128GB of RAM and a sixteen core processor. The executable was deployed using SBATCH with the parameter –

`ntasks` set to 32, thereby providing a thread to each core. This experimental setup was then tested for scalability:



The MPI implementation demonstrates good scalability and is substantially faster than any previous Open MP program. The MPI program was run on two of the same class of nodes used to run the Open MP programs and, under this setup, the MPI program appears to run roughly twice as fast. Generally, this MPI implementation is about *p*-times faster than the single-node Open MP tasks implementation:



MPI + Open MP: Hybrid Parallelization of the Mean Shift Algorithm

The final paradigm investigated in this work combines the previous two approaches—MPI and Open MP—into a single implementation. This hybrid paradigm aims to exploit parallelism at both the thread and node level. Two programs were developed for this purpose: one that uses Open MP loops and another that uses Open MP tasks.

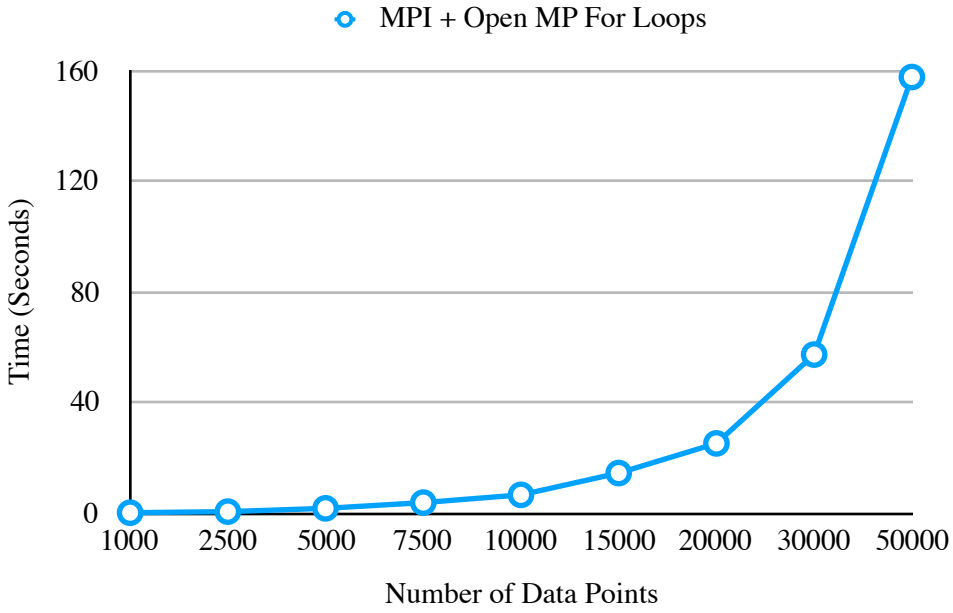
MPI + Open MP Loops:

The first hybrid program is implemented in `mpi-for.c` and can be run using `mpi-for.sh`. This program is identical to the MPI implementation, except that the for loops used to calculate the weighted kernel means are executed in parallel instead of serially. The for loop parallelization was statically scheduled with round-robin distribution to minimize overhead. As with the MPI program, this implementation was deployed on two nodes, each with 128GB of RAM and a sixteen core processor. Before scalability can be assessed, the optimal number of MPI processes and CPUs per process, which is equivalent to the `OMP_NUM_THREADS` variable, must be found. This was tested using a dataset of 5,000 2-D points:

MPI Processes	Time (Seconds)
2 processes, 16 cpus/process	4.41
4 processes, 8 cpus/process	1.71
8 Processes, 4 Cpus/process	4.34
16 Processes, 2 Cpus/process	5.30
32 Processes, 1 cpu per process	5.35

As seen above, the optimal combination was four MPI processes with eight CPUs per process. On one hand, increasing the number of MPI processes might lead to greater parallelization. However, having more MPI processes means greater communication overhead when calling the Scatterv and Gatherv collective operations. Moreover, because of the randomly distributed nature of the data, the data on some of the processes will converge faster than data on other processes. Having more processes can lead to worse load balancing in this regard, increasing the probability that some nodes converge rapidly while other nodes have outlier data that take longer to converge. Thus, four MPI processes represents the optimal tradeoff in terms of load balancing and overhead. With eight CPUs per process, this ensure that all cores across the two nodes have a single thread to work on. Having more CPUs per process rather than fewer is beneficial as it

ensures there are enough threads available to work on statically allocated for loop iterations. Using four MPI processes, the scalability of the MPI + Open MP for loop iteration program was tested using the two-node setup:



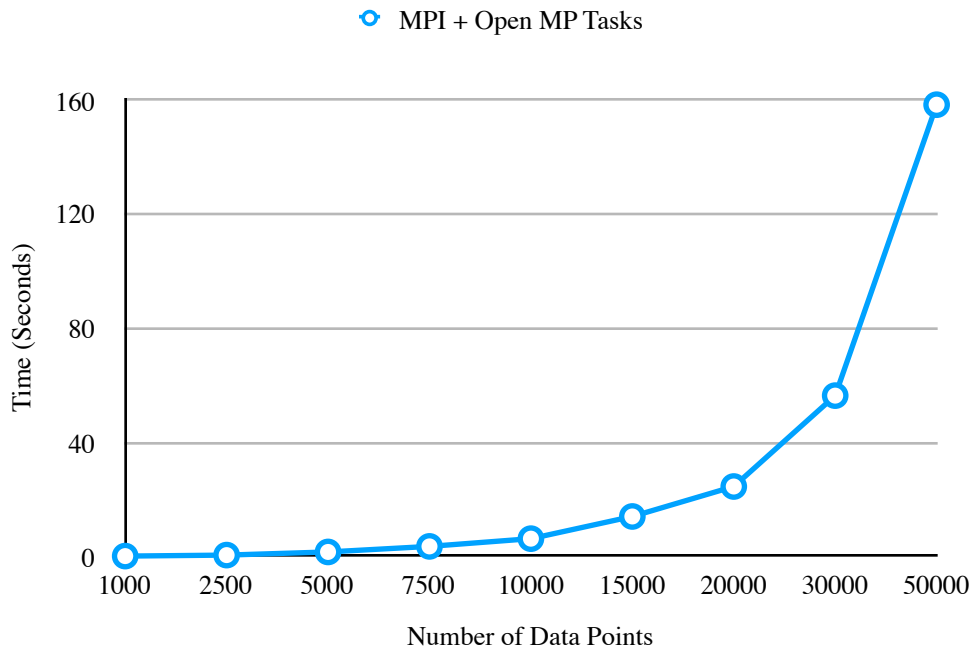
The program exhibits the expected $O(n^2)$ scalability and, especially at large values of n, the hybrid program runs substantially faster than the MPI-only program. The hybrid program provides a larger speedup over the MPI-only program at larger n values because there are more loop iterations for Open MP to parallelize and thereby improve performance.

MPI + Open MP Tasks:

The second hybrid implementation developed in this work used MPI and the tasks construct in Open MP. In this implementation, each node is handed a certain a number of data points and tasks are generated to compute the mean shift for each data point for every iteration. This implementation is contained within `mpi-tasks.c` and `mpi-tasks.sh`. As with the loop hybrid program, the optimal number of MPI processes and CPUs per process were determined using the two-node setup:

MPI Processes	Time (Seconds)
4 MPI Processes, 8 CPUs/process	1.13
8 MPI Process 4 CPUs/process	2.35
16 MPI Processes, 2 CPUs/process	2.29
32 MPI Processes, 1 CPU/process	2.77

The combination of four MPI processes and eight CPUs per process produced the best performance, likely for the same reasons discussed above with the loop hybrid implementation. Scalability was then assessed using four MPI processes:



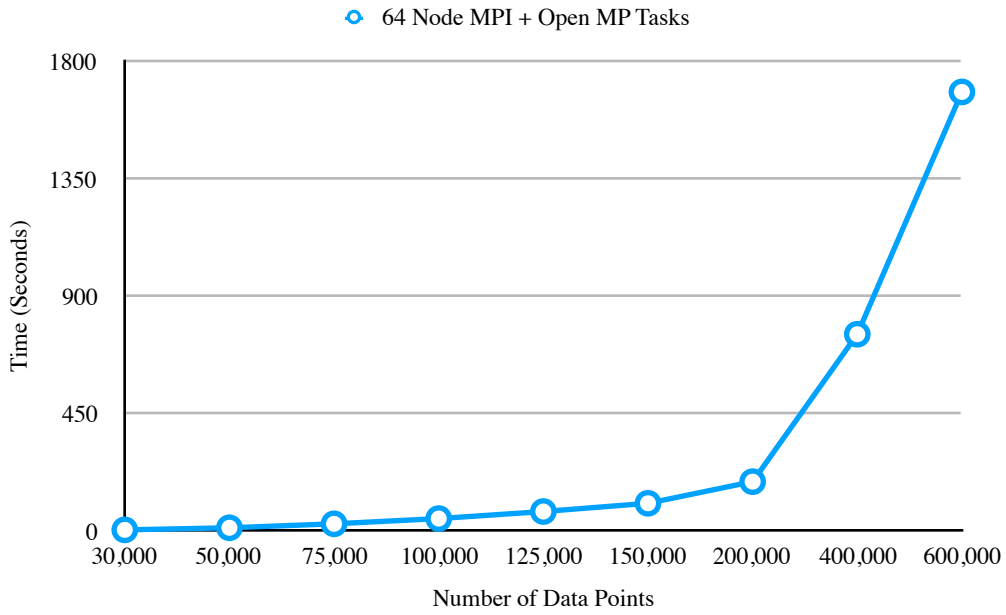
As we can see above, the performance of the hybrid tasks program is very similar to the hybrid loops program. This is in contrast to the earlier finding that tasks outperformed for loop. It is unclear why tasks and loops now perform equivalently when used in a hybrid MPI-Open MP program, but it might have something to do with how Open MP loops and tasks are implemented by MPI processes.

Scaling Up

To test and characterize the limits of scalability, the hybrid tasks + MPI program was run on 64 nodes, each with eight cores and 32 GB of RAM providing a total of 512 cores with 2,048 GB of RAM. As usual, the optimal number of MPI processes and CPUs per process needed to be found. The experiments below were computed using a randomly-generated dataset of 15,000 2-D points:

MPI Processes	Time (Seconds)
1 CPU/process	2.77
2 CPUs/process	2.29
4 CPUs/process	1.13
8 CPUs/process	2.35

As we can see, using 128 MPI processes with 4 CPUs per process yields the optimal results. This is an interesting result as when working with the sixteen core processors, using eight cores per process, or half the available cores per process, provided optimal performance. In the case of nodes with eight cores, it again appears that using half the available cores per process yields the best results. This is likely because of the load balancing and overhead considerations discussed earlier. After determining the optimal number of MPI processes, the results from the large-scale, 64 node cluster are shown below:



First, it is clear that the MPI multi-node implementation enables solving massive big data mean shift clustering problems that are impossible to solve on a single-node in a reasonable timeframe. Even with Open MP parallelization, a single-node would be unable to compute the mean shift algorithm on 600,000 2-D data points in a realistic timeframe. The quadratic nature of the mean shift algorithm underlies the importance of parallelization when scaling to massive datasets. Computing a dataset of 30,000 points took 57.3 seconds when computing on two sixteen core nodes, while computing this same dataset on sixty four eight core nodes took 4.4 seconds. Second, the algorithm scales as expected, with the usual $O(n^2)$ scaling. Finally, the algorithm, distributed across 64 nodes, computed the correct solution and correctly identified the proper clusters.

Conclusion:

In this paper, the mean shift algorithm was parallelized through several implementations. First, the single-node multithreading paradigm was implemented using

the Open MP framework's task and loops constructs, leading to substantial speed improvements, especially for the tasks-based program. Next, the multi-node paradigm was implemented using MPI, producing substantial parallelization benefits while still computing the correct solution. Finally, the multi-node, multithreading paradigm was implemented using a hybrid MPI and Open MP program, tested with both Open MP loops and tasks. This program had the best performance and successfully scaled to a large 64 node configuration allowing the computation of massive datasets of up to 600,000 data points.

Future Work:

This work advanced a number of parallel implementations for the mean shift algorithm, and there are many possible improvements that can lead to additional performance improvements. To maintain code simplicity and maximize time spent on developing parallel implementations, some standard techniques to improve the mean shift algorithm were omitted in this work. One such improvement would be to employ bin seeding. This is a standard technique in clustering that leads to substantial performance improvements and does not require any special modifications when moving from serial to parallel implementations. Bin seeding represents initial kernel locations as discretized bins of points, with the coarseness of the bins determined by the bandwidth parameter. This allows fewer seeds to be initialized, substantially improving the convergence of the algorithm in earlier iterations.

Another improvement in the algorithm would be to estimate the bandwidth using cross-validation. This is an $O(n^2)$ operation and overall runtime can be reduced by using smaller sample sizes when computing bandwidths. Empirically estimating the bandwidth can lead to improved clustering results. In many, but not all cases, using cross-validation to find the optimal bandwidth can improve overall performance as the time spent finding the proper bandwidth is less than the time saving from faster convergence given an optimal bandwidth. This is particularly true when selecting a large bandwidth value that leads slower convergence. With regards to bandwidth, another possible improvement would be to use an *adaptive bandwidth*, which sets a unique bandwidth h_i for every datapoint x_i such that $h_i = || x_i - x_{i,k} ||$, where $x_{i,k}$ is the k-nearest neighbor and the norm can be the L1 or L2 norm. This can lead to faster convergence and avoid a fixed bandwidth that is too small, leading to poor clustering, or too large, leading to slow convergence.

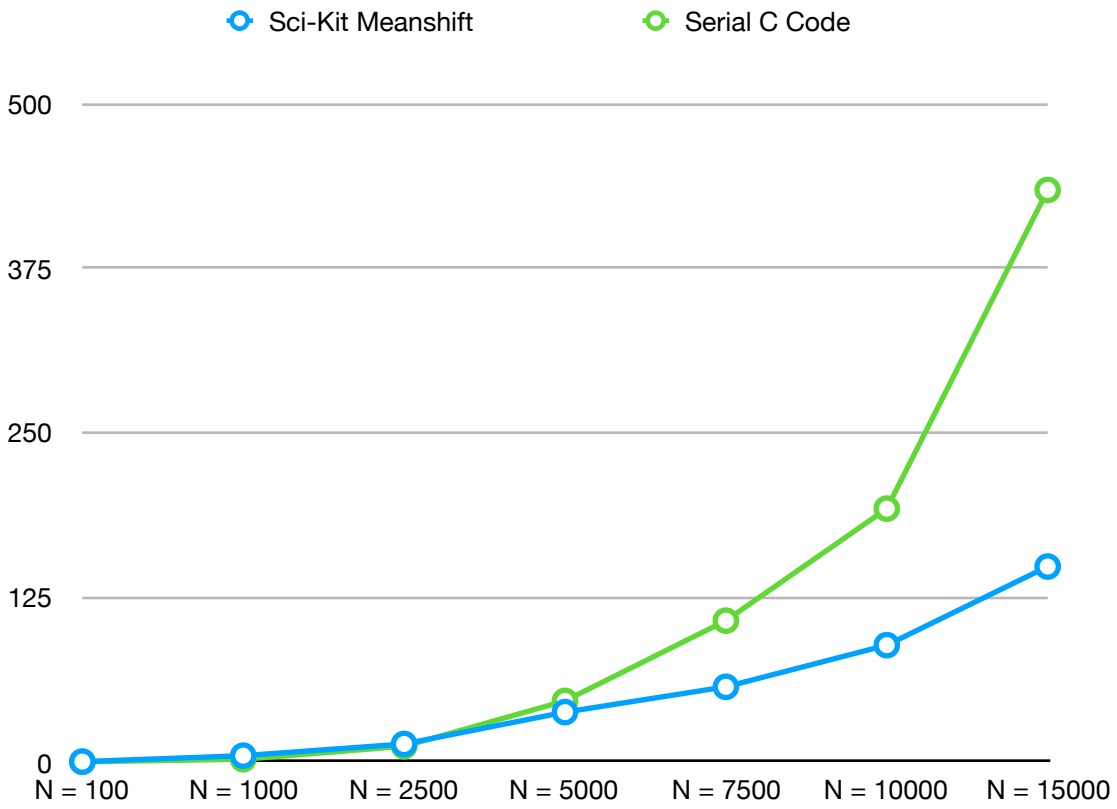
As discussed earlier, another improvement to the algorithm would be to use a k-d tree and use k-nearest neighbor lookup to only consider the k-nearest points. In lower dimensions, this allows the mean shift algorithm to tend towards $O(n \log n)$, but in higher

dimensions the algorithm will revert to quadratic time complexity as a result of the curse of dimensionality [4]. As seen in the benchmark results for the scikit-learn implementation (see Appendix), using the k-d trees with 2-D points leads to improved scalability.

A final improvement to this work would be to improve load balancing for the MPI program. In experimentation, it was observed that some nodes converged faster than other nodes and required less time to compute their chunk of data. This is because some nodes will be given data that have more outliers and these data points will require more time to converge. It is conceivable to improve load balancing by giving idle nodes that have finished converging some of the outlier data points. However, one consideration here is that the time for node-to-node communication might eliminate any benefits from parallelizing remaining computation for outlier data points.

Appendix

Benchmarking



The Scikit-Learn implementation uses both parallelization and a k-d tree, which is why the sci-kit learn implementation scales at O(n log n) rather than the serial C code which scales at O(n ^2)

Works Cited

[1] K. Fukunaga, L. Hostetler. "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32-40, January 1975.

[2] T. Cacoullos. "Estimation of a multivariate density," *Ann. Inst. Statis. Math.*, vol. 18, pp. 179-189, 1966.

[3] A. Ghassabeh, Youness."A sufficient condition for the convergence of the mean shift algorithm with Gaussian kernel," *Journal of Multivariate Analysis,* March 2015.

[4] R. Marimont, M. Shapiro. "Nearest Neighbour Searches and the Curse of Dimensionality," *IMA J Appl Math,* August 1979.

[5] M. Nedrich. "Simple implementation of mean shift clustering in python", *Github*, December 2017. Retrieved from https://github.com/mattnedrich/MeanShift_py.

[6] C. Lee, A. Gramfort, G. Varoquaux, M. Sorbaro. "Sci-kit Learn Mean Shift Implementation", *Github*, November 2018. Retrieved from https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/cluster/mean_shift_.py.

[7] Z. Wen, Z. CAI, "Mean Shift Algorithm and Its Applications In Tracking of Objects", *Proceedings of the Fifth International Conference on Machine Learning and Cybernetics*, August 2006.

[8] D. Comaniciu, P. Meer. "Mean Shift Analysis and Applications". Retrieved from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=790416.