Scalability of a Parallel Arbitrary-Dimensional Image Distance Transform

Scott K. Pardue, Nikos P. Chrisochoides, Andrey N. Chernikov
Old Dominion University Computer Science Department
spardue@cs.odu.edu, nikos@cs.odu.edu, achernik@cs.odu.edu

Keywords: Parallel computing, parallel distance transform, distance transform, scalability, speedup, efficiency, algorithms, Euclidean Distance Transform, EDT, medical imaging, image processing

Abstract

Computing the Euclidean Distance Transform (EDT) for binary images is an important problem with applications involving medical image processing, computer vision, computational geometry, and pattern recognition. Currently, there exists a sequential algorithm of O(n) complexity developed by Maurer et al. and a parallel implementation of Maurer's algorithm developed by Staubs et al. with a theoretical complexity of O(n/p) for n voxels and p threads. In this paper, we present an efficient, scalable parallel implementation of Maurer's algorithm for large datasets with high efficiency for 16 processors.

1. Introduction

An EDT of an N-dimensional binary image is an N-dimensional matrix representing the Euclidean distance of each volume pixel (voxel) in the binary image to the closest foreground element in the binary image. Currently, the best sequential algorithm for calculating the EDT of a binary image, developed by Maurer[1], computes it in linear time. Maurer's algorithm focuses on dimensionality reduction by computing the EDT at each dimension by generating partial Voronoi diagrams for each row of voxels in each dimension. This is done by first initializing the EDT by iterating through each row of voxels in the binary image for the lowest dimension. When a foreground voxel is found, the associated voxel in the EDT is set to zero while all others are set to infinity. The voxel data of the binary image is only used for initialization. The EDT is used for the remainder of the computations. After initialization, each row of voxels for each dimension in the EDT is iterated through beginning with the lowest dimension. All voxels that are equal to infinity are disregarded. All of the remaining voxels, known as feature voxels (FV) are compared against the two closest FV to determine if the current FV intersects the row of voxels that is currently being examined. If the current FV does not intersect the current row, then this FV is disregarded. After all of the FV have been compared, the row of voxels is iterated through comparing the current Euclidean distance for the given voxel to the Euclidean distance to each FV. The lower of the two distances is the new Euclidean distance for the current voxel. The EDT of the lower dimension is used to calculate the EDT of the current dimension. The order of processing for a two dimensional image is shown in Figure 1. Each row, of the first dimension is iterated through, then each row of the second dimension is iterated through.

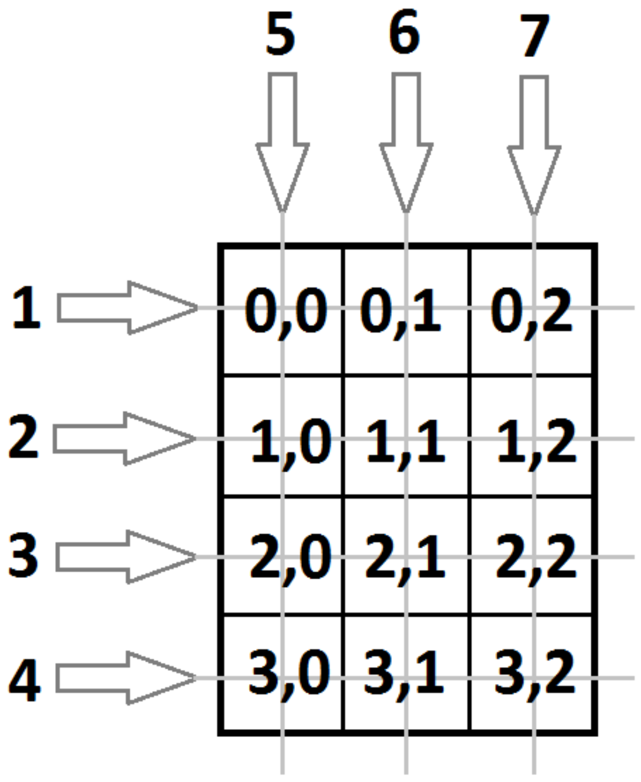

Figure 1. Order of Rows to Process

Figure 2 and Figure 3 show an example of an image of brain ventricles from the Brain Atlas[3] and its corresponding 3D distance transform, respectively.

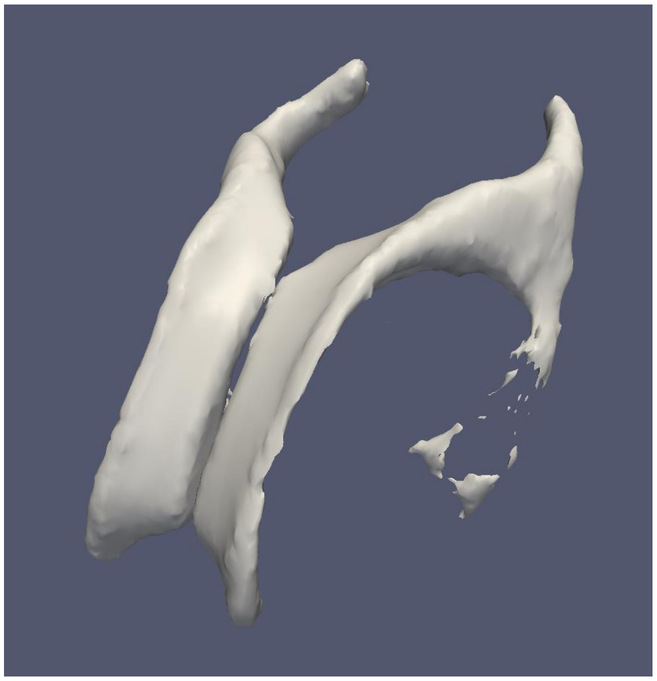

Figure 2. Image of Brain Ventricles

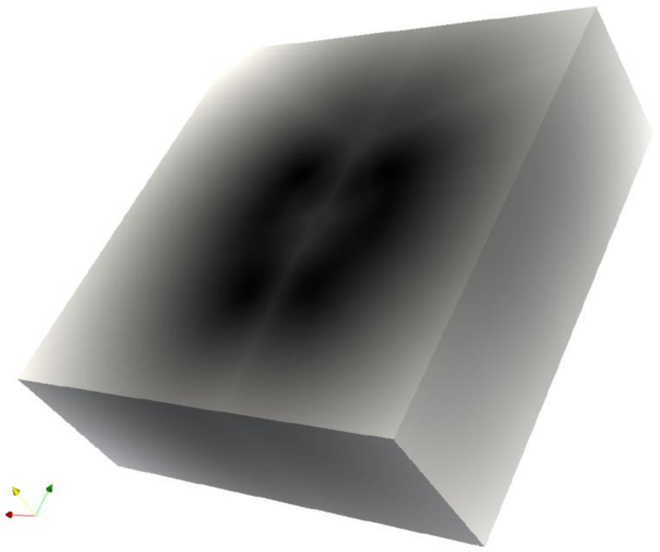

Figure 3. 3D Distance Transform of Figure 2

There is another parallel implementation of Maurer's algorithm, presented in Staubs[2] which features significant speedup for small scale problems and a low number of threads. In this paper, we present a more generalized, scalable, and efficient parallel implementation of Maurer's algorithm. Our implementation is able to provide a mean speedup of 6 times for 8 threads while Staubs[2] was only able to provide a mean speedup of 3 times with 8 threads. Also, the speedup data presented in Staubs[2] asymptotically approaches 3 times, while our implementation has a projected asymptote of 20 times for the speedup for large datasets.

2. Our Approach

Our algorithm follows the same dimensionality reduction approach and partial Voronoi diagram generation developed by Maurer[1] to efficiently calculate the EDT for the image. Our algorithm is generalized to compute the EDT for any given number of dimensions. This is done by representing the image as a single array in row-major order. The row-major representation is used instead of column-major representation because the elements are accessed by iterating the index of the current dimension. Row-major order would produce a benefit in access time over column-major order for computing the EDT of a row, while column-major order would produce a benefit for computing the EDT of a column. The array is laid it contiguously in memory which results in better cache performance for iterating over a row or column for row-major or column-major organization, respectively, of the array representation of the binary image.

For our implementation, we are utilizing the POSIX Threads Programming library. The library includes a pthread datatype and a mutex datatype which we required to implement our algorithm. A pthread is the POSIX version of a thread. A thread is an individual collection of instructions to be executed. Without the use of threads, a program runs sequentially, one instruction after another, waiting for each instruction to terminate before beginning the next instruction. With threads, multiple instructions may be executed simultaneously. A mutex is the name given to the semaphore to control access to a common resource. When a thread locks a mutex, then that thread has exclusive control of that resource until the controlling thread releases the resource by unlocking the mutex. If a thread tries to gain control of a mutex that is already owned, or locked, by another thread, then the calling thread waits until the resource becomes available. We also use the thread controls wait, signal, and broadcast. When a thread waits, the thread pauses its execution until it receives a signal. If multiple threads are waiting, then all thread may resume execution through the use of a broadcast.

2.1 Load Balancing

A common problem in parallel algorithms is load balancing. In order to maximize speedup, idle time must be minimized. We have achieved this by utilizing the producer-consumer paradigm in that the main program creates the consumer threads and creates the work for the consumer threads. The total amount of work is stored in W_d, a one-dimensional array, while the indices of each dimension are stored in $D_{d,w,i}$, a three-dimensional array. The total amount of work for a given dimension can be calculated with a single multiplicative summation if the current dimension is excluded:

$$\left[\prod_{i=0}^{d-1} N_i\right] * \left[\prod_{i=d+1}^{n} N_i\right]$$

Where N_i is the number of rows for dimension i. For each dimension i, there are N_i number of rows. For calculating the EDT, each row of the current dimension must be iterated through for all other dimensions. Consider the example of a 3x4x2 matrix, (width of 3, height of 4, and a depth of 2). For the lowest dimension, we calculate the number of rows that need to be processed by multiplying the number of rows in all other dimensions. So the number of rows that need processing for the lowest dimension would be 4*2=8, and the number of rows the need processing for the next dimension would be 3*2=6, and 3*4=12 for the highest dimension.
Once the work has been generated for the current dimension by the producer, a signal is sent to the consumers which then retrieve work from the front of the queue and compute the EDT by iterating through the indices of the current dimension while keeping the indices of the other dimensions fixed based on the values retrieved from $D_{d,i}$. This approach provides better load-balancing than statically allocating work before processing begins. When a consumer thread checks the queue of work and the queue is empty, the consumer thread goes into a wait state. Once all threads have reached the wait state, a signal is sent to the producer. If the producer has finished generating the work queue for the next dimension, the producer broadcasts to the consumer threads to begin processing the next dimension. A barrier cannot be used because we cannot guarantee that the producer will have produced the work queue necessary for the next dimension by the time the current dimension has finished computing. While the consumer threads process the current dimension, the producer is generating the work queue. This also helps eliminate thread idle time by not waiting until the work queue for all dimensions is generated. This approach is possible because the only task dependency (other than the dependency of the consumer for the producer in generating the work queue) is that the EDT for current dimension is dependent on the previous dimension. Each row in the current dimension is independent of all other rows in the current dimension.

2.2 Dimension Generalization

Once the binary image is represented in row-major form as a single array, the indices for iterating through a given dimension can be computed using the stored size information in N and the current fixed dimensions stored in $D_{i,w}$. We were able to generalize the calculations to compute the indices for a given dimension using a summation of multiplicative summations. These computations are few, even in the case of large dimensions, compared to the task of generating and querying the partial Voronoi diagram and therefore do not have a significant effect on the complexity of the algorithm. Given the current dimension and the indices of the fixed dimension, $D_{d,i}$, the indices of the current dimension are calculated during the construction of the partial Voronoi diagram by

$$\begin{aligned} index_i = & \left[\sum_{k=0}^{d-1}\left(\prod_{j=0}^{k-1} N_j\right) * D_{d,w,k}\right] \\ & + \left[\left(\prod_{j=0}^{k-1} N_j\right) * i\right] \\ & + \left[\sum_{k=d+1}^{n_d}\left(\prod_{j=0}^{k-1} N_j\right) * D_{d,w,k}\right] \end{aligned}$$

This formula works by calculating the offset for each dimension. To calculate the offset for a dimension, the summation of the multiplicative summation of all dimensions is calculated. The multiplicative summation of a dimension is represented by multiplying the size of all lower dimensions to produce an offset. This offset represents the number of voxels that comprise one set of the given dimension. This offset represents the first index of the first set of the dimension. To find the first index of the N-th set of a dimension, the offset is multiplied by N. For our algorithm, the N-th set of the fixed dimensions is given by $D_{d,w,i}$ while the N-th set of the current dimension is given by i. It is important to store these indices for future use while querying the partial Voronoi diagram.

3. Experimental Performance

We have tested our implementation on a machine with 40 CPUs and 126G of memory. We generated cube images with dimensions of 500x500x500, 1000x1000x1000, and 1250x1250x1250 and ran each with a literal interpretation of the algorithm presented in Maurer[1] sequentially and with our implementation using 2, 4, 8, 16, 24, 32, and 40 threads. A cube with dimensions 1250x1250x1250 is used as our largest test case because our machine does not have enough memory to process larger cubes.

The image data that is being processed are cubes, so the amount of work can be represented by $3*n^2$ where n is the number of rows in each dimension. One unit of work is one row of voxels for a dimension. For our 500x500x500 cube image, there are 750,000 tasks; 1000x1000x1000: 3,000,000 tasks; 1250x1250x1250: 4,687,500 tasks. As the number of threads increases, efficiency decreases. The point at where the efficiency starts decreasing rapidly depends on the problem size. This is due to thread idle time and overhead costs for accessing the mutex to retrieve work. The thread idle time is minimized through dynamic load balancing. Through the user of a mutex, dynamic load balancing is made possible. However, for implementations that use static allocation of tasks, thread idle time would increase due to improper load balancing and there would be minimal overhead costs. For larger problems, using a mutex to minimize thread idle time results in better performance and an overall higher efficiency. Results are depicted in Figure 4 (Speed Up) and Figure 5 (Efficiency).

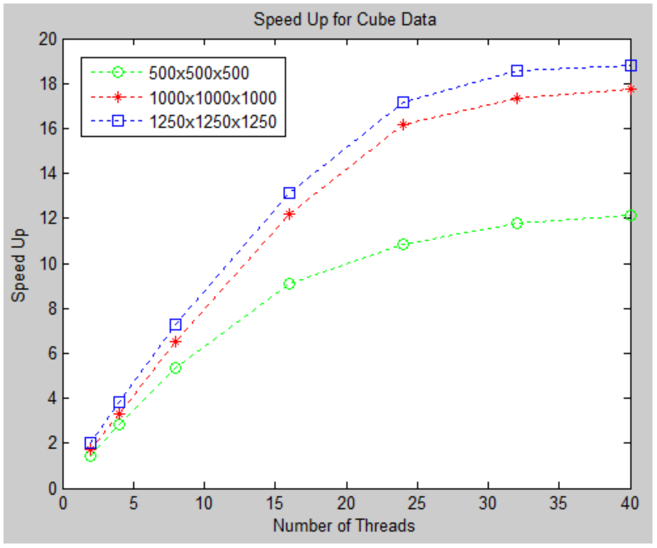

Figure 4. Graph of Speed Up for Cube Data

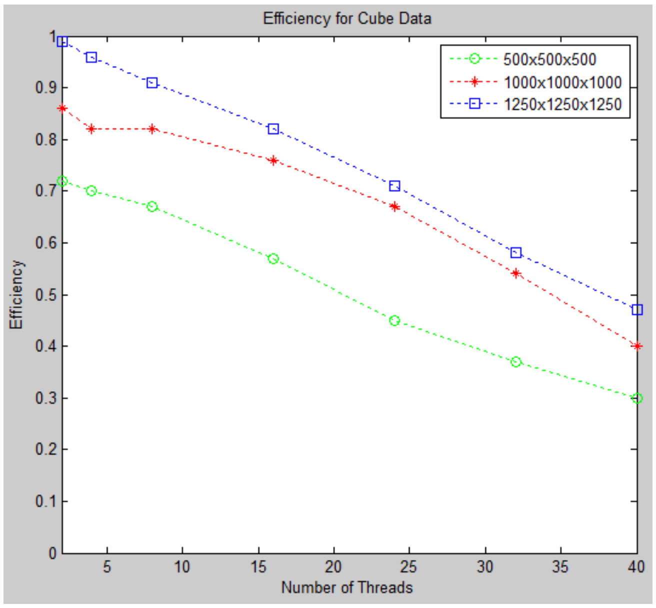

Figure 5. Graph of Efficiency for Cube Data

3.1 Performance Comparisons

The difference between our implementation and the implementation presented in Staubs[2] is that our implementation is more scalable in regards to problem size and number of threads. Our implementation is able to provide a mean speedup of 6 times for 8 threads while Staubs[2] was only able to provide a mean speedup of 3 times with 8 threads on a machine with 4 processors and 2 GB of memory. Also, the speedup data presented in Staubs[2] asymptotically approaches 3 times, while our implementation has a projected asymptote of 20 times for the speedup for large datasets. Using our current machine, we were able to measure mean speedups of 19 times using 40 threads.

3.2 Work In Progress

This implementation is currently still a work in progress as we plan to test our implementation on larger machines and larger datasets. We also plan to further analyze and modify our algorithm to improve the efficiency. Limitations of our implementation include a drop in performance when using a larger number of threads. This is most likely due to the mutex required to access the shared queue of work as the efficiency drops even for larger problem sizes. Another possible adaptation of our implementation that we will examine and evaluate is the master-worker paradigm where the master will communicate to each thread which tasks need to be generated and processed as opposed to our current implementation of the producer-consumer paradigm where the producer generates the tasks and the consumers process them.

The master-worker paradigm will decrease the need for the mutex and alleviate some startup costs on the main program (i.e. the producer in this case). Another possible, but smaller adaptation, would be to create multiple queues of work and multiple mutexes for accessing the work queues. Since the efficiency drops rapidly after 16 threads, one possibility would be to have the number of queues equal to the ceiling of the number of threads divided by 16. Each thread would be designated to an initial queue and when the queue becomes empty, those threads may request work from the other queue(s).

4. Conclusions

Our introduction of a more scalable Parallel Euclidean Distance Transform algorithm implementation will allow for larger datasets to be processed more efficiently and with more processing power. Also, because our algorithm operates on any dimension, we are able to provide an efficient and extendable algorithm for many different image processing applications to utilize without the need to modify our algorithm.

5. Acknowledgements

We would like to thank the Old Dominion University Information Technology Services for making available the computing resources used in our evaluation.

6. References

[1] Calvin R. Maurer, Jr., Rensheng Qi, and Vijay Raghavan. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. IEEE Trans. Pattern Anal. Mach. Intell., 25(2):265–270, 2003. http://dx.doi.org/10.1109/TPAMI.2003.1177156. (document), 1, 2, 3

[2] Staubs R., Fedorov A., Linardakis L., Dunton B., Chrisochoides N. Parallel N-Dimensional Exact Signed Euclidean Distance Transform. 2006 Sep. http://www.insight-journal.org/browse/publication/123.

[3] Talos I-F., Jakab M., Kikinis R., Shenton M.E. SPL-PNL Brain Atlas. SPL-PNL March;