A Scalable 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, algorithms, Euclidean Distance Transform, 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. In algorithm development, execution time is an important factor. Parallel algorithm development also needs to focus on scalability and efficiency. 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 an asymptotical speedup of 3 times. In this paper, we present a parallel implementation of Maurer's algorithm with a theoretical complexity of O(n/p) for n voxels and p threads and an evaluated unprecedented linear speedup for large datasets.

1. Introduction

Computing the EDT of a binary image was originally performed through an exhaustive method by first iterating through the image and identifying each voxel as a background voxel or a feature voxel (FV). After the set of FVs have been identified, the image is then iterated through again and each background voxel is compared to every FV to determine the distance from the current background voxel to each of the FVs. The shortest distance is then recorded for the background voxel. This method requires an initial loop through the image and a nested loop to compute the distances for each background voxel. This approach has a complexity on average of $O(n^2+n)$ or just $O(n^2)$. In 2003, Maurer et. al[1] developed an approach to compute the EDT of a binary image in linear time through an approach of dimensionality reduction and partial Voronoi diagrams. For the approach developed by Maurer[1], each row of voxels for each dimension of the EDT is iterated through beginning with the lowest dimension and the partial Voronoi diagram for each row is generated. Using this partial Voronoi diagram, each voxel in the row is compared to each candidate Voronoi site in the row and the Euclidean distance is calculated. 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 potential candidate FVs are compared against the two closest FVs to determine if the current potential candidate FV intersects the row of voxels that is currently being examined. If the current potential candidate FV does not intersect the current row, then this potential candidate FV is disregarded. After all of the candidate FVs have been identified, the row of voxels is iterated through comparing the current Euclidean distance for the given voxel to the Euclidean distance to each candidate FV. The smaller 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.

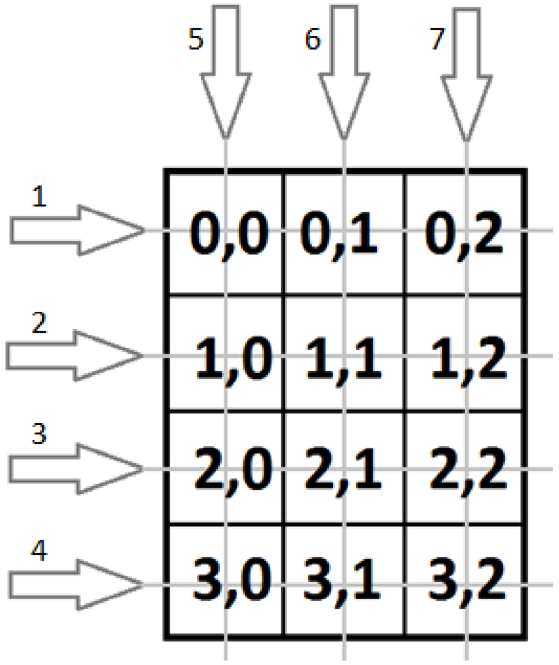

Figure 1. Order of Rows to Process

Currently, the approach developed by Maurer[1] is determined to be the best sequential approach for computing the EDT of a binary image. Using the approach developed by Maurer[1], a parallel implementation was presented by Staubs et. al[2] which featured a maximum speedup of three times. This maximum speedup was achieved using eight threads. Previously, we[3] developed a parallel implementation also using the approach developed by Maurer[1]. We[3] obtained a mean speedup of six times for eight threads along with a projected asymptote of twenty times speedup. In this paper, we present a parallel implementation of Maurer's algorithm. Our parallel implementation operates with unprecedented speedup and has a complexity of O(n/p).

2. Our Approach

Our algorithm follows the same approach developed by Maurer[1] and uses the same producer-consumer paradigm which we[3] previously implemented. However, after our experimental performance showed a logarithmic drop in efficiency with a projected asymptote of a times 20 speedup around 40 processes, we began researching the cause of the drop in performance. We examined communication time between the producer process and the consumer processes along with the accumulated wait time of the consumer processes as the number of consumer processes increased. Additionally, we monitored the call stack and system memory to determine how resources were being allocated. We also studied how long the consumer processes waited to access the shared queue of work. Based on these examinations, we discovered that the source of the drop in performance did not lie within the shared queue of work, but in how the system is managing the call stack and other resources. Additionally, other optimization techniques were utilized to minimize tasks that are performed in constant and linear time as the number of processes increases.

2.1 Load Balancing

We had originally speculated that our algorithm's efficiency dropped as the number of processes increased due to access to the shared mutex. Upon further experimentation, it was discovered that the accumulated wait times associated with accessing the shared queue of work through a single mutex increases linearly as the number of processes increases, but the individual process wait times decrease. This is partially due to the inverse relationship between units of work per process and number of processes. The important evidence to note is that the maximum accumulated wait times for each experiment increases linearly as the number of processes increases with a fixed amount of work, which is acceptable for our algorithm as it operates with a complexity of O(n/p).

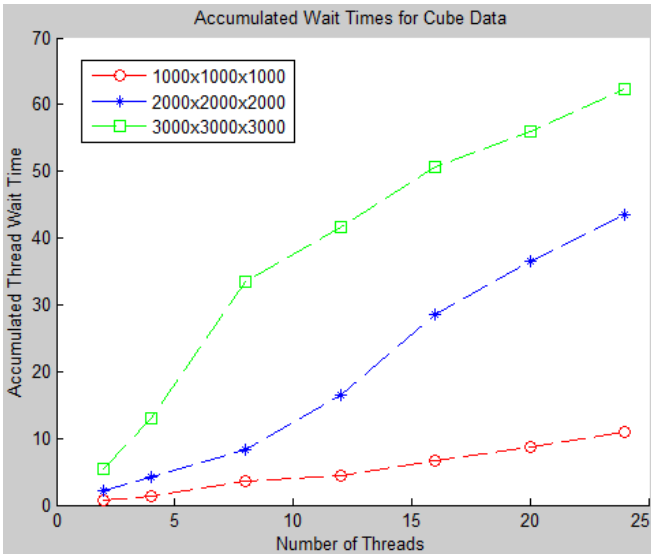

Figure 2. Graph of Wait Times for Cube Data

Results for the accumulated wait times for threads for three cubes (1000x1000x1000, 2000x2000x2000, 3000x3000x3000) are depicted in Figure 2. The accumulated wait times for the 4000x4000x4000 and 5000x5000x5000 cube are not shown for graphical scaling reasons as the data plotted from the 4000x4000x4000 cube would render the data from the shown cubes as flat lines and the 5000x5000x5000 cube would render the data from the 4000x4000x4000 cube as a flat line.

2.2 Elimination of Recursion

The task of creating the shared queue of work was originally handled recursively in order to provide an algorithm which can compute the EDT of an image independent of the number of dimensions and the length of rows for each dimension. In programming, when a method is invoked, a new stack frame is added to the call stack. The call stack contains all of the stack frames of the active methods currently executing while a stack frame contains all of the data associated with the particular stack frame's method. The stack frame data includes input variables, local variables, and a reference to where the method will return control after the method terminates. Originally most programming languages chose not to support recursion because each stack frame was identified by the name of the method in the call stack. This causes recursion to be impossible as there would be no way to distinguish between stack frames because the identifiers would be identical. For programming languages that support recursion, each invocation of a method is identified by an address which is independent of the method name. This allows for multiple instances of a method to exist by allowing the call stack to contain unique stack frames for identical method invocations. However, this poses a problem for recursive methods because the call stack has a limited amount of memory allocated to itself by the system. If the call stack exceeds the memory size limitation, then the program will crash. For recursive methods that generate a large number of subsequent recursive calls or are deeply recursive, the probability of a crash due to stack overflow is increased.

The following pseudocode shows the structure of a simplified version of the two recursive function responsible for generating the shared queue of work. d is an integer representing the current dimension that work is being defined for, c is a dimension iterator, and N is the set of the length of the row size for the given dimension. The computeEDT function is originally invoked with an input of the number of dimensions minus one. The computeEDT_R function is tail recursive, meaning that a new stack frame does not need to be added to the call stack because most modern compilers interpret tail recursive functions as GOTO statements. However, the computeEDT function is not tail recursive and a new stack frame is needed. Additionally, each call to computeEDT where d is not zero generates N_d new calls to computeEDT. Each call to computeEDT generates a call to computeEDT_R. Each call to computeEDT_R where d is not equal to c generates N_c new calls to computeEDT_R.

```
computeEDT(d)
BEGIN
if(d != 0)
  for 0..Nd
    allocate subsections of work queue
    computeEDT(d-1)
  end for
end if
computeEDT_R(0, d)
END
```

```
computeEDT_R(c, d)
BEGIN
if(c == d)
  create work
else
  for i..Nc
    initialize subsections of work queue
    computeEDT_R(c+1, d)
  end for
end if
END
```

Even though the computeEDT_R function is tail recursive, the compiler should not be relied on to optimize tail recursive approaches. Additionally, not all compilers support tail recursion, and not all languages support recursion. A switch to iterative methods was chosen to improve optimization during the task of generating the shared queue of work, eliminate stack overflows caused by recursion, and define an approach that can be implemented independently of programming language.

Given the input of a three-dimensional cube with a row size of x, the number of additional computeEDT calls totals x^2+x while the number of additional computeEDT_R calls totals $3x^2+2x$. In developing a sustainable and robust algorithm suitable for usage for exascale computing, recursion was replaced with an iterative approach, utilizing the structure of the tasks. Each task involves a process iterating through a row of voxels and computing the EDT of the given row based on the current values of the in-progress EDT. The term row is used generically to refer to a linear set of voxels parallel to an axis. Each task has one dimension that is iterated over while all other dimensions remain fixed. Examining the set of tasks, we notice that each row must be iterated over for each dimension. This allows us to use combinatorics, a branch of mathematics focusing on countable structures, to generate all of the tasks in an iterative fashion.

2.3 Inline Methods

For methods that are repetitively called in immediate succession, a large portion of runtime is wasted for creating a new stack frame, pushing the stack frame to the call stack, passing the input variables, allocating and initializing local variables, returning the output variables, releasing control back to the calling method, releasing memory space for the local variables, and popping the stack frame from the call stack. By modifying the algorithm to replace the successive method invocations with a while loop eliminates the system tasks of allocating, pushing, and popping stack frames along with eliminating the constant allocation and deallocation of local variables. Using an inline approach, local variables are allocated outside of the while loop and initialized inside of the loop for each iteration of the method. The removeEDT method is invoked when there are three FV identified. Each of the three FV has its corresponding Voronoi area computed to determine which Voronoi areas intersect the current row of the image. The removeEDT method has six input parameters and three local variables. Converting the removeEDT method to an inline representation of the method eliminates six copy assignments and three allocation and deallocation instructions per method invocation. Given a row of size x, the maximum number of removeEDT calls totals x-2 (at least three FV are needed to call removeEDT). This accumulates to approximately 6x copy instructions and 3x allocation and deallocation instructions eliminated per row.

2.4 Memoization

Memoization is an optimization technique which involves storing the results of computationally expensive calculations so that the result can be reused when the inputs are repeated. This allows for the computationally expensive calculations to not have to be repeated. Our algorithm requires the indices of the row to be used twice: once for constructing the partial Voronoi diagram and again for computing the EDT. Since our algorithm uses dimension generalization, the following calculation must be evaluated for each voxel:

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

where i is the index being computed, d is the current dimension that the EDT is being calculated in, k is the dimension iterator, N is the set of the length of the row size for the given dimension, and $D_{d,w}$ is the set of the current dimension indices. The parameter w is fixed for the calculation and only represents a single unit of work for computing the EDT for a given row. Storing the resulting index for each calculation during the construction of the partial Voronoi diagram allows for the index to be reused during the calculation for the EDT for the given row of voxels. Additional memoization is implemented by only calculating the offset portion of the index calculation once. The offset is given by the two summations of the index calculation:

$$offset = \left[\sum_{k=0}^{d-1}\left(\prod_{j=0}^{k-1} N_j\right) * D_{d,w,k}\right] + \left[\sum_{k=d+1}^{n_d}\left(\prod_{j=0}^{k-1} N_j\right) * D_{d,w,k}\right]$$

Note that the reason why this calculation can be extracted and stored is that the offset is independent of the indices being calculated. The offset is only dependent on the set of the current dimension indices, $D_{d,w}$ and the set of the length of the row size for the dimensions, N. From the offset, the set of indices can be calculated from:

$$index_i = offset + \left[\left(\prod_{j=0}^{d-1} N_j\right) * i\right]$$

For a three-dimensional image, storing the set of indices reduces the calculations by 50% and first calculating the offset reduces the initial calculation of indices by 80% because there are five multiplicative summations that must be calculated (four of which are used to calculate the offset).Without these two methods, ten multiplicative summations must be computed. With these methods of memoization, only one multiplicative summation needs to be computed, reducing the total computation time for index calculations by 90%.

2.5 Barriers

Our original approach did not utilize barriers because the shared queue of work is generated for a dimension by the single producer process while the lower dimension's EDT is computed by the consumer processes and there is no guarantee that the producer process will finish generating the work before the consumer processes finish computing the EDT. Instead of a barrier, the use of signals, broadcasting, and wait statements are utilized. Once a consumer process has computed the EDT for all of its rows, it increments a global variable and waits for a signal via broadcast from the producer process. Each consumer process checks the incremented global variable to determine if all of the consumer processes are finished computing. If all of the consumer processes are finished, then the final consumer process sets a flag to represent that the consumer processes are finished computing and then sends a signal to the producer process. A flag must be set in the event that the producer process is not finished producing the shared queue of work for the next higher dimension so that when the producer process is finished generating the work, then the flag can be checked because the producer process will not be waiting on a signal from the final consumer process resulting in the signal from the final consumer process to be ignored. If the flag is set (showing that the consumer processes are finished computing) then the producer immediately broadcasts a signal to the consumer processes to start computing the EDT of the higher dimension. If the flag is not set, then the producer process waits for a signal from the final consumer process before the producer process broadcasts the signal to the consumer processes. Both the flag and signal from the final consumer process are needed for the two cases: first where the consumer processes complete before the producer process and second where the producer process completes before the consumer processes.

As a result of the shortened producer process time due to the elimination of recursion, and the reduced computation time for each consumer process from the result of memoization techniques and the switch to inline methods, we made the decision to have the producer complete the work generation task completely for all dimensions before the consumer processes began computing. This decision was made because the time for the entirety of the consumer processes to increment the completion counter and execute wait statements, the producer process to check the completion flag or receive the wait signal, and the producer process to broadcast to the collection of consumer processes becomes computationally expensive as the number of consumer processes increases. Additionally, the time for the producer process to complete the work generation task is negligible now that recursion has been eliminated. This decision then allowed us to utilize barriers, which are highly optimized control structures which cause all consumer processes to wait at the barrier until all consumer processes have reached the barrier. This approach has been proven to be faster than the wait, signal, and broadcast paradigm which we were using before.

3. Experimental Performance

We have tested our implementation on a machine containing 2 Intel Xeon E5-2697 v2 processors with 2.7 GHz and 12 physical cores each and a NVIDIA Tesla K40c GPU with 2880 cores. The machine also contains 768 GB of 1600MHz DDR3L memory. We generated cube images with dimensions of 1000x1000x1000, 2000x2000x2000, 3000x3000x3000, 4000x4000x4000, and 5000x5000x5000 ran each with our implementation using 2, 4, 8, 12, 16, 20, and 24 threads. The cube images that we generated had randomized FVs for 1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, and 99% of the voxels. Additionally, we also created cube images with a special case of a single FV in the corner, or the lowest dimension's first row's first voxel. On our current machine, the maximum sized image we are able to generate is a cube of 5000x5000x5000 voxels, totaling 125 billion voxels. We have achieved linear speedup for all of our test cases, as depicted in Figure 3.

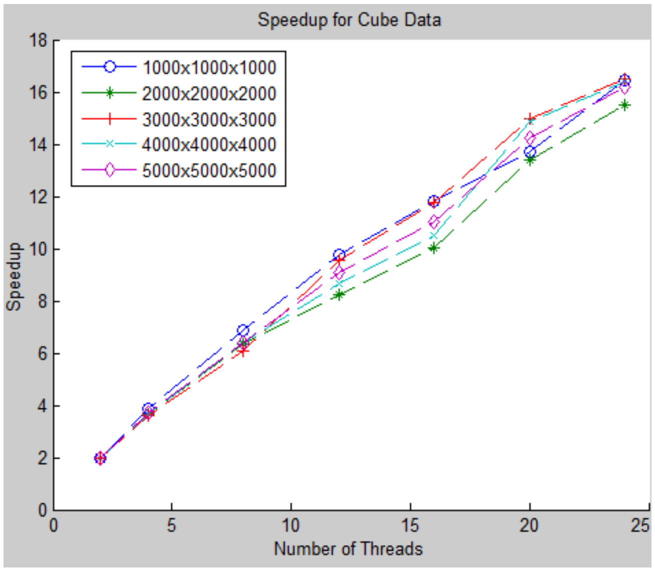

Figure 3. Graph of Speedup for Cube Data

The percentage of FVs in the image did not have a significant effect regarding the overall computation time. The minimum and maximum computation times for each percentage of FVs for each of the cube images sizes for number of processes only varied by five percent of the average computation time for each cube image. The average computation time was computed by the grouping of number of threads and cube size. We have also made significant improvements in not only speedup, but also overall execution time.

4. Conclusions

Our introduction of a linearly scalable parallel Euclidean Distance Transform algorithm approach will allow for larger datasets to be processed with a high number of processes without experiencing a significant loss of performance. Our approach does not utilize recursion, which is language dependent, and also avoids the possibility of stack overflow errors. Our approach operates on any dimension for any row length size for each dimension, which allows us to provide a generalized, efficient, and extendable algorithm for exascale computing.

5. Future Work

We plan to extend our work to handle anisotropic EDTs as well as signed EDTs in the future. We do not foresee any significant effect to the overall scalability of the algorithm with the introduction of anisotropic and/or signed EDTs.

6. Acknowledgements

We thank the Old Dominion University Computer Science Department for making available the computing resources used in our evaluation.

This work was supported in part by the NSF grant CCF-1139864.

7. 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] Pardue S., Chrisochoides N., Chernikov A. Scalability of a Parallel Arbitrary-Dimensional Image Distance Transform. 2014 Apr. https://crtc.cs.odu.edu/pub/papers/conf_147.pdf.