PARALLEL COMPUTATION OF SIGNED DISTANCE FIELDS IN 2D PIXEL IMAGES

RAYMOND LIN

Abstract. Signed distance functions are functions which compute the Euclidean distance to the boundary points of a set and negates values for inputs which lie within the set. They have many applications, particularly in graphics where they may be used to manipulate surfaces and images. In these use cases, it is often useful to use a signed distance field which stores the result of the signed distance function at every point in space.

Signed distance fields for 2D images can be computed such that each pixel stores the signed distance from its center to the nearest boundary point. This can also be extended to 3D voxel spaces and higher dimensions, but as the resolution and dimension count grow, the number of points to evaluate the signed distance function at increases quickly.

This paper will explore several methods for computing a signed distance field in 2-dimensional pixel spaces quickly using parallel programming in the Julia programming language. Although higher dimensions are outside of the scope of this project, many of these techniques can easily generalize, which can be useful for 3D graphics or n-dimensional optimization problems.

In order to optimize performance of the signed distance field computation, both parallel and serial considerations will be made when comparing three different algorithms. In particular, serial performance can be significantly optimized compared to brute-force algorithms before even considering parallel optimizations, but this may come at the cost of reduced ability to parallelize the algorithm.

Key words. Signed Distance Field, Parallel Computing

AMS subject classifications. 65Y05, 68W10

1. Introduction.

Signed distance fields are widely used in a variety of fields, including graphics and robotics [4]. Given a set of interior points in a region, the signed distance field computes the shortest (Euclidean) distance from any point to the nearest boundary point, which is then negated if the point was an interior point and left non-negative for exterior points.

Given a set of interior points B, the signed distance function for a point (x, y) can be written as the following optimization problem:

$$f(x,y) = \begin{cases} \min ||(x,y) - (x',y')|| \ : \ (x',y') \in B & (x,y) \notin B \\ -\min ||(x,y) - (x',y')|| \ : \ (x',y') \notin B & (x,y) \in B \end{cases} \tag{1.1}$$

Voxel grids such as pixel images are commonly used in the context of signed distance fields. The interior and exterior points of the original shape can be represented as a binary voxel grid, with white/true values indicating interior points and black/false values indicating exterior points [3]. The boundary is the set of points demarcating adjacent black and white voxels rather than the centers of the voxels themselves - this means that the lowest absolute value a voxel can have is 0.5.

2.121	1.58	1.5	1.5	1.58	2.121
1.58	0.707	0.5	0.5	0.707	1.58
1.5	0.5	-0.5	-0.5	0.5	1.5
1.5	0.5	-0.5	-0.5	0.5	1.5
1.58	0.707	0.5	0.5	0.707	1.58
2.121	1.58	1.5	1.5	1.58	2.121

FIG. 1. *Signed distance field for a 6 × 6 pixel grid with 4 interior pixels in the center*

Calculating the signed distance field in voxel grids requires evaluating the signed distance function at every voxel. This can be quite computationally expensive as the resolution of the grid is increased, leading to a need for optimized implementations of signed distance field calculations.

This paper will evaluate implementations of signed distance field algorithms on the special case of 2D images with the goal of minimizing runtime through parallelization and improvements in serial performance. Each of these algorithms has been implemented in the Julia programming language and were benchmarked on a 6-core Intel(R) Core(TM) i7-9750H CPU.

The algorithms will be presented in serial in section 2 and expanded to parallel implementations in section 3, with more detailed descriptions in the appendix. They will then be timed for comparison in section 4. Section 5 will evaluate the algorithms and explain the differences in their performance and investigate bottlenecks common between algorithms such as shared memory.

2. Serial Algorithms. Serial implementations can be found in this file in the project repository.

2.1. Brute Force Algorithm. Although inefficient, a brute force approach to the problem of signed distance field computation provides a useful baseline and is also easier to parallelize than many other more intricate algorithms.

At a high level, the brute force approach works by computing the distance between every pair of pixels for which exactly one is an interior point and one is an exterior point, using the pixel border as the boundary.

Due to this algorithm comparing every pixel with every other pixel, it has a quadratic asymptotic runtime - for an $m \times n$ input image, this approach has an $O(m^2n^2)$ runtime.

2.2. Brushfire Algorithm. In order to improve serial performance of signed distance field calculations, graph search algorithms can be employed. An algorithm known as the brushfire algorithm, which is an extension of Dijkstra's graph search, may be employed to calculate an *unsigned distance field*[2]. This is also referred to as a Euclidean distance transform and computes the Euclidean distance from all points in the domain to the closest interior point. As this is unsigned, interior points will have a distance of zero. A second pass of the Euclidean distance transform which inverts interior and exterior points can then be subtracted from the first to yield the

signed distance field.

Within each pass calculating the Euclidean distance transform, there are two phases: seeding and propagation [6].

The seeding phase initializes the conditions for Dijkstra's search to run by making a linear pass through all pixels to find interior pixels which border exterior pixels. These pixels are marked as the points from which the propagation phase will begin from.

The propagation phase uses a modified version of Dijkstra's algorithm to compute shortest distances to the boundary in exterior pixels in order of increasing distance. In each iteration, Dijkstra's algorithm chooses the closest unexplored node via a priority queue and propagates through the node. Pixels in the image represent nodes in the graph, and the connections from each pixel to its 8 neighbors are the edges. The distances calculated by Dijkstra's would not be accurate for a Euclidean distance transform, as the sum of edge lengths will always be greater than or equal to the Euclidean distance to the boundary, but it can still be used for determining the order to explore neighboring pixels. Thus, rather than storing just the best known Euclidean distance to a pixel in the priority queue, the vector pointing from the nearest boundary point to the pixel will also be stored.

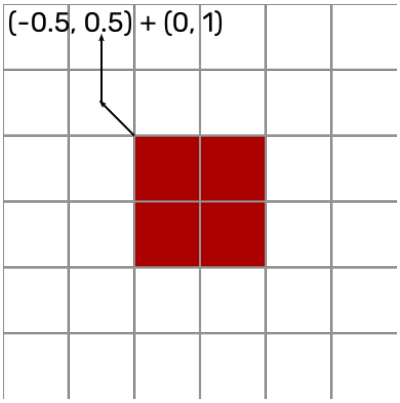

FIG. 2. *The brushfire algorithm stores the vectors from the nearest boundary point to the current pixel, allowing vector addition to be used to calculate the vector from the boundary to the pixel.*

With a priority queue such as a min-heap, the selection of the closest pixel to the boundary to explore in each iteration can be achieved in $O(\log(mn))$ time for an $m \times n$ image, so the brushfire algorithm has an asymptotic runtime of $O(mn \log(mn))$. Priority queues with faster asymptotic runtimes such as Fibonacci heaps exist, but in practice they are slower due to constant factors [1].

2.3. Linear-Time Algorithm. [5] Meijster et al. devised a linear-time $O(mn)$ Euclidean distance transform algorithm for an $m \times n$ image using a 2-phase approach consisting of only linear scans. This is the fastest possible asymptotic time complexity to compute such a transform, as a pass through all pixels is required. A second pass of the Euclidean distance transform with an inversion of interior and exterior points as described in the previous section can be used to compute the signed distance field.

The Euclidean distance transform function for an interior point set B can be written as $f(x, y) = \min |(x, y) - (x', y')| \ : \ (x', y') \in B$ or equivalently,

$$f(x, y) = \min \sqrt{(x - x')^2 + (y - y')^2} \ : \ (x', y') \in B \tag{2.1}$$

Meijster et al. explain that the algorithm begins with two linear vertical passes through the image to compute the function $G(x, y) := \min |y - y'| \ : \ (x, y') \in B$, where B denotes the set of interior points. In other words, $G(x, y)$ computes the smallest distance from (x, y) to an interior point with the same x coordinate. G can be computed using these linear passes because the computations to determine $G(x, y)$ and $G(x', y')$ are independent for $x \neq x'$.

Given that $G(x, y)$ has been calculated for all (x, y) within the image bounds, the minimization problem from (2.1) can be rewritten as

$$f(x, y) = \min \sqrt{(x - x')^2 + G(x', y)^2} \tag{2.2}$$

Note that the minimization problem now only minimizes over one variable, x', and that there are no constraints on x' with regards to the set B because membership in the interior point set is already accounted for with the function G.

Similarly to the initial pair of vertical passes, two linear horizontal passes can then be done to minimize across x'. This computes the final value of $f(x, y)$ at every point. Similarly to the computation of G, this is possible because each calculation of $f(x, y)$ and $f(x', y')$ is now independent for all $y \neq y'$.

The use of only linear passes allows this algorithm to have an $O(mn)$ asymptotic runtime on $m \times n$ images; hence, it has the best theoretical runtime of the described algorithms.

3. Parallel Implementation. Parallel implementations can be found in this file in the project repository.

3.1. Brute Force Algorithm. Parallelization of the brute force approach is fairly simple because the distance calculations for each pixel are independent of every other pixel. Each pixel's distance is written to its own slot in the result matrix, which does not need to be read by other pixels, so memory race conditions are avoided. Hence, each processor can be assigned roughly n/p columns, where n is the total width of the image and p is the number of processors.

3.2. Brushfire Algorithm. Although the brushfire algorithm theoretically offers considerable improvements in serial performance, it suffers from being far less practical to parallelize. Distance calculations for pixels are dependent on the order they are explored, as determined by the priority queue. Hence, pixel distances cannot be easily computed in parallel without introducing inaccuracies.

One possible method of parallelizing Dijkstra's algorithm involves parallelizing the priority queue's operations [7], in particular, adding nodes to the queue and popping the closest node. Although previous work has shown that this method has potential [7], the implementation presents many challenges which bring this method outside of the scope of this project.

Dijkstra's algorithm is by nature highly sequential, so for this project, the signed distance field algorithm was instead parallelized by running each of the two Euclidean distance transform passes in parallel. Each Euclidean distance transform path on its own does not have any parallelized code.

3.3. Linear-Time Algorithm. The creators of the linear-time algorithm stated that the algorithm was designed with parallelization as a goal [5]. Parallelization of the algorithm is fairly simple - during each vertical scan of the image, the work done on each column is independent of other columns, which means that each processor can be assigned roughly n/p columns where n is the total columns and p is the number

of processors. Similarly, during horizontal scans, each processor can be assigned m/p rows for m total rows in the image.

The main challenge of parallelizing this algorithm comes from avoiding unnecessary memory allocations. During the second half part of this algorithm, the horizontal scans to minimize $f(x, y)$ given $G(x, y)$, arrays with size equal to the image width need to be allocated to aid with constructing the final distance field. The serial implementation preallocates a single $n \times 1$ array for an image of width n prior to the horizontal scans, but this will fail for the parallel implementation because multiple threads will attempt to read and write to the same preallocated array at the same time. Instead, for p processors, an $n \times p$ matrix is allocated for the parallel approach. Processor i will be constrained to using column i of the matrix, allowing the processors to avoid conflicts, but this comes at the cost of allocating p times as much memory to this step.

4. Comparison. In order to benchmark the real-world runtime of each of the algorithms, four sets of test images were used, with each image having variants with resolution 256^2, 512^2, 1024^2, 4096^2, and 8192^2. These images had interior regions which consisted of a 2×2 dot in the center of the image, an offset filled circle, the entire left half of the image, and randomly placed dots around the image. These cases were selected because the size and shape of the test images can have an impact on the real-world runtime of the algorithms even if the theoretical asymptotic runtimes are dependent only on the image resolution. Hence, a variety of different shapes was selected. See Appendix B for a visualization of the 256×256 variants of the images and their associated signed distance fields.

Note that some algorithms were not run on the higher resolution variants due to their poor performance.

As mentioned previously, each of these algorithms has been implemented in the Julia programming language and were benchmarked on a 6-core Intel(R) Core(TM) i7-9750H CPU.

The following graph shows the various algorithms benchmarked on the offset circle image (see Figure 8 for the original image). Note that the brute force algorithm was only run on sizes up to 512^2, while the brushfire algorithm was only run on sizes up to 4096^2.

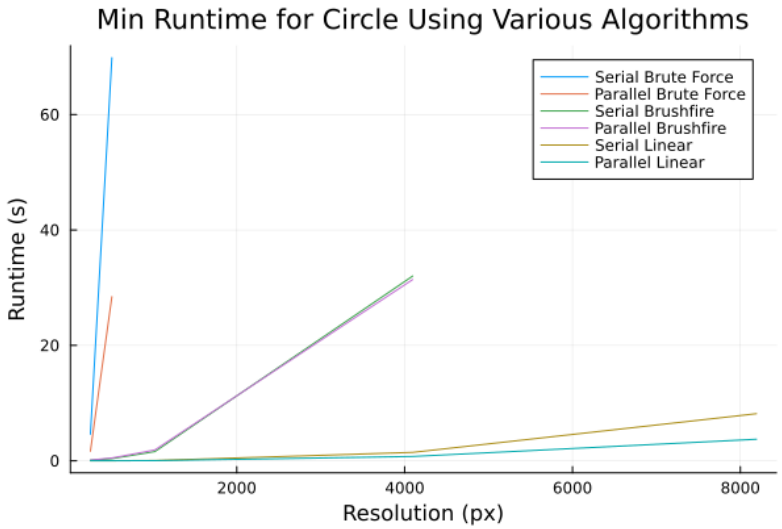

FIG. 3. *Comparison of serial and parallel implementations of algorithms on various resolutions of the circle image*

At all resolutions, the linear algorithm is faster than the brushfire algorithm, which is in turn faster than the brute force algorithm. Additionally, parallelizing improves runtime in all cases, although the amount by which it improves the runtime is dependent on the algorithm - note that the runtimes for the brushfire algorithm are almost identical between the serial and parallel implementations.

This general trend carries through to the other images as well. A plot of the runtimes for each algorithm run in both serial and parallel on each different image type can be found in Appendix C.

The next section will evaluate the differences in performance in more detail and explain the reasons for these differences as well as any discrepancies between theoretical and real performance gains from parallelization.

5. Evaluation.

5.1. Brute Force Algorithm. As seen in the plots in Appendix C, the brute force algorithm is generally unaffected by the shape of the interior points in the region, with the exception of the random dots image which takes over 120 seconds for a resolution of 512^2, while the other images all finish within 75 seconds for the same resolution. Despite running many trials and taking the minimum runtime, the random dots image proved to be consistently slower than other images in the serial case. This may be because the randomness of the dot locations forces worst-case memory writes. As the brute force algorithm iterates through every pair of points, it needs to make a memory write every time a new best distance is found. Having random dots spaced roughly evenly around the image may cause best new distances to be found more frequently during each iteration compared to a structured image, which in turn requires far more memory writes. However, the best parallel runtime of 37.78 seconds almost exactly matches the best parallel runtime of 37.46 seconds for the halves image.

In serial, the average difference in runtime between the 512^2 images and 256^2 images is $\frac{67.7824811+69.8607894+72.6243412+122.4982236}{4.8957518+4.6128594+5.0149132+7.7139157} = 14.96$ times. Given that the theoretical asymptotic runtime is $O(m^2n^2)$, this is close to the expected 16 times

theoretical slowdown.

For the 256^2 images, parallelizing the brute force algorithm produces an average speedup of $\frac{4.8957518+4.6128594+5.0149132+7.7139157}{2.377716+1.6612869+2.1323944+2.1111143} = 2.68$ times. For 512^2 images, the average speedup is $\frac{67.7824811+69.8607894+72.6243412+122.4982236}{29.5334853+28.4388328+37.461746+37.7826545} = 2.50$ times. With a 6-core processor, the theoretical maximum speedup should be on the order of 6 times. Much of the failure to meet this speedup could be attributed to the brute force algorithm only being run on inputs with relatively small sizes. These sizes may be too small for the benefits of parallelization to outweigh the increased overhead needed to distribute the task across the 6 cores. Regardless, the poor performance of the brute force algorithm means that even if further benefits from parallelization may exist, it would be far more effective to optimize serial performance first.

5.2. Brushfire Algorithm. Unlike the brute force approach, the brushfire algorithm's runtime is much more dependent on the shape of the image. The midpoint image has a best serial runtime of 27.02 seconds, whereas the halves image has a serial runtime of 60.77 seconds. Given that the brushfire algorithm has a theoretical runtime of $O(mn \log(mn))$, much of this slowdown can be attributed to the constant factors which come from the algorithm needing two passes to compute the positive and negative parts of the signed distance field. The midpoint image is almost entirely composed of exterior points, so the majority of the work done in the algorithm comes from the pass to determine exterior distances. In contrast, the halves image breaks the interior and exterior into evenly-sized regions, so both passes have to perform roughly the same amount of computation.

The differences due to division of each work in each pass is especially apparent when parallelizing brushfire by running the two passes in parallel. The following plot shows the runtime of the serial algorithm divided by the parallel algorithm for the midpoint image:

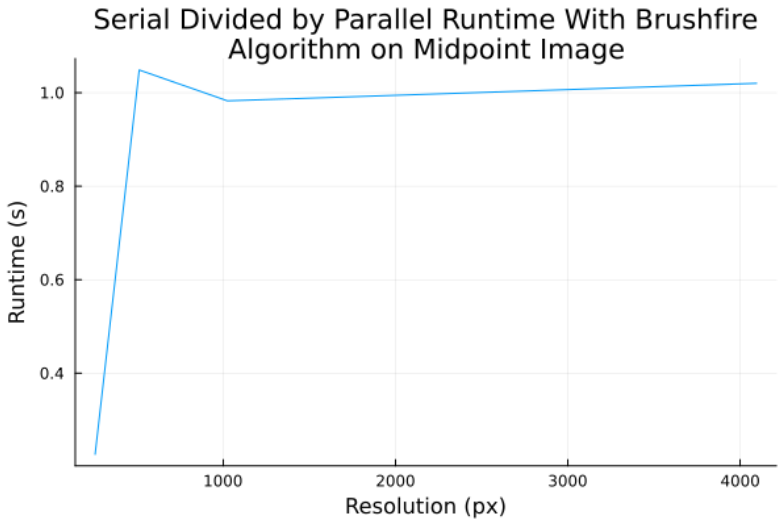

FIG. 4. *Serial divided by parallel runtime with brushfire algorithm on midpoint image*

As shown above, for the lowest resolution (256^2), the parallel runtime is considerably worse. This is to be expected, as the overhead of running the algorithm in parallel is a much higher percentage of the total work at low input sizes. For higher

resolutions, the ratio approaches 1 - the parallel and serial implementations have almost the same runtime. This is because the interior of the image is very small, so the interior pass of the brushfire algorithm takes almost no time. Hence, the majority of the work is in the exterior pass, which takes the same amount of time in both cases.

The optimal case for parallelization occurs when the interior and exterior regions are roughly the same size. The half image satisfies these conditions, and as a result, the parallel implementation is able to perform much better:

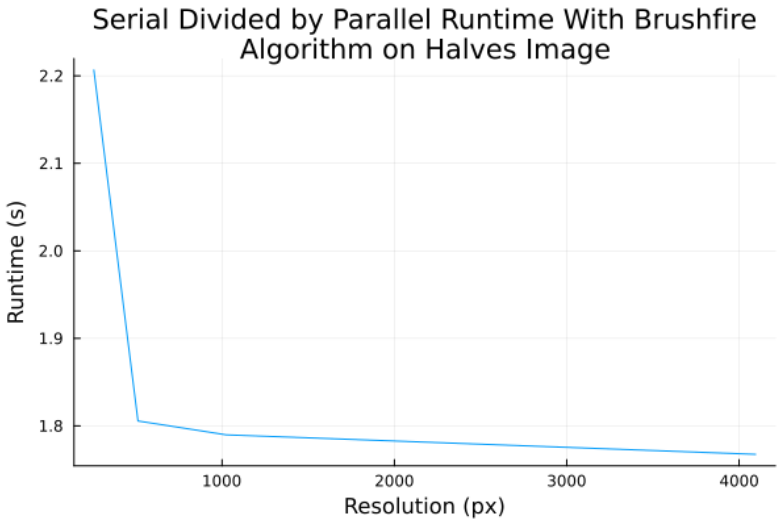

FIG. 5. *Serial divided by parallel runtime with brushfire algorithm on halves image*

The parallel approach completes in close to half the time as the serial approach at all tested image resolutions. In serial, the two brushfire passes will take the same amount of time to complete but must run sequentially. Since the parallel runtime is constrained by which of the two passes is slower, the passes having equivalent runtimes maximizes the performance gain.

5.3. Linear-Time Algorithm. As expressed by Meijster et al., the linear-time algorithm is designed to be easily parallelizable and its runtime is generally independent of the contents of the input image [5]. This is generally supported by the results of the algorithm benchmarking. For 8192^2 images, the serial implementation of the linear-time algorithm runs in around 8 seconds for each of the input images, while the parallel implementation runs in just under 4 seconds. The average difference in runtime between 512^2 and 8192^2 images in serial is a factor of $\frac{7.5354068+7.2247885+7.928729+8.162382}{0.0150689+0.013887+0.0142074+0.0125212} = 554.04$. This is about double the expected theoretical time slowdown of a factor of 256. This may be due to the amount of time taken by memory allocations becoming a larger percentage of the total runtime with higher input sizes. As show in Appendix D, allocating an 8192^2 array takes 0.989 seconds, while allocating a 512^2 array takes only 0.000714 seconds, which is a difference of a factor of 1385. As a percentage of the original runtime in serial, allocating the 512^2 array takes about 5.1% of the total runtime, while allocating the 8192^2 array takes about 12.8% of the total runtime.

Parallelizing the linear-time algorithm does improve runtime, but not as much as the theoretical 6× speedup from having 6 cores:

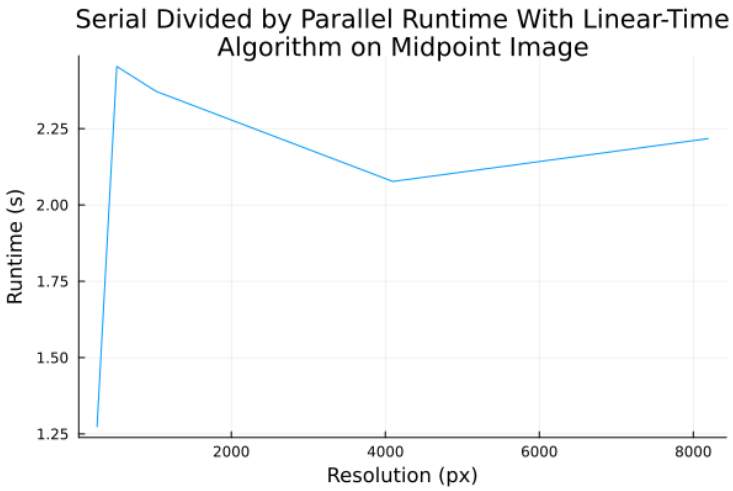

FIG. 6. *Serial Divided by Parallel Runtime With Linear-Time Algorithm on Midpoint Image*

Much of this can again be attributed to memory as a bottleneck. In the second phase of the linear-time algorithm, the horizontal scan, the parallel algorithm needs to allocate p times as much memory since there are now p processors. As shown in Figure 25 in Appendix D, this allocation takes a considerable amount of time.

Additionally, in the parallel implementation, each core will need to read and write to their own data structures in memory. While these data structures are not shared between cores, the frequent operations on different locations in memories will reduce the number of cache hits, which worsens runtime. Nevertheless, this algorithm is still the fastest by far on all input sizes, and parallelizing it generally decreases runtime by half across sufficiently large inputs.

6. Conclusion. As discussed in the above sections, the linear-time algorithm performs best, followed by the brushfire algorithm and then the brute force algorithm in both the serial and parallel cases. Parallelizing generally improves performance given a large enough input, but factors such as memory allocations and the nature of the original serial algorithm can reduce the effectiveness of parallelization.

Despite the difficulty in parallelization and the dependence on input shape and not just input dimensions, the brushfire algorithm is still considerably faster than the brute force algorithm in both serial and parallel. This makes the algorithm more practical than brute force despite its shortcomings and highlights the importance of serial performance as a precursor to obtaining good parallel performance. Furthering this point, the serial implementation of the linear-time algorithm outperforms the parallel implementations of the other algorithms.

Further improvements to serial performance may be difficult since the linear-time algorithm already achieves the best possible theoretical asymptotic complexity, but a faster parallel algorithm could be possible with more work in investigating how to reduce and/or optimize memory accesses.

REFERENCES

[1] C. E. R. R. L. S.-C. CORMEN, THOMAS H.; LEISERSON, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 3rd ed., 2001.

[2] S. S. W. W. M. Q.-H. M. Delong Zhu, Chaoqun Wang and R. Garg, *Vdb-edt: An efficient euclidean distance transform algorithm based on vdb data structure*, (2021), https://arxiv.org/pdf/2105.04419.pdf.
[3] S. Gustavson and R. Strand, *Anti-aliased euclidean distance transform*, (2010), https://www.sciencedirect.com/science/article/pii/S0167865510002953?via%3Dihub.
[4] Z. T. E. G. J. N. Helen Oleynikova, Alexander Millane and R. Siegwart, *Signed distance fields: A natural representation for both mapping and planning*, (2016), https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/128029/eth-50477-01.pdf.
[5] A. M. J. Roerdink and W. Hesselink, *A general algorithm for computing distance transforms in linear time*, (2000), http://fab.cba.mit.edu/classes/S62.12/docs/Meijster_distance.pdf.
[6] TKMI-kyon, *Computing the signed distance field*, (2020), https://tkmikyon.medium.com/computing-the-signed-distance-field-a1fa9ba2fc7d.
[7] Z. Ye, *An implementation of parallelizing dijkstra's algorithm*, https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ye-Fall-2012-CSE633.pdf.

Appendix A. Source Code. Project code can be found at this link. Serial implementations of algorithms can be found here, while parallelized versions are here.

Appendix B. Test Images. In each image, white pixels represent interior points while black pixels represent exterior points. For the visualizations of signed distance fields, increasing red brightness indicates increasing positive distance, while increasing green brightness indicates increasing negative distance. Blue is used to mark the boundary points at which distance is 0.

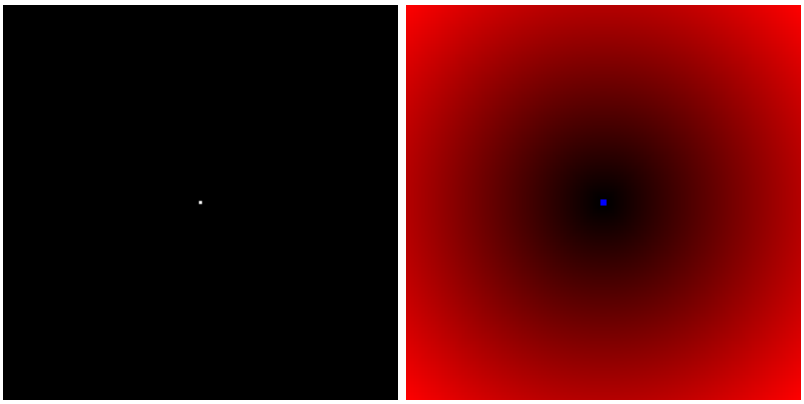

Fig. 7. *A single centered dot (left) and its associated signed distance field (right)*

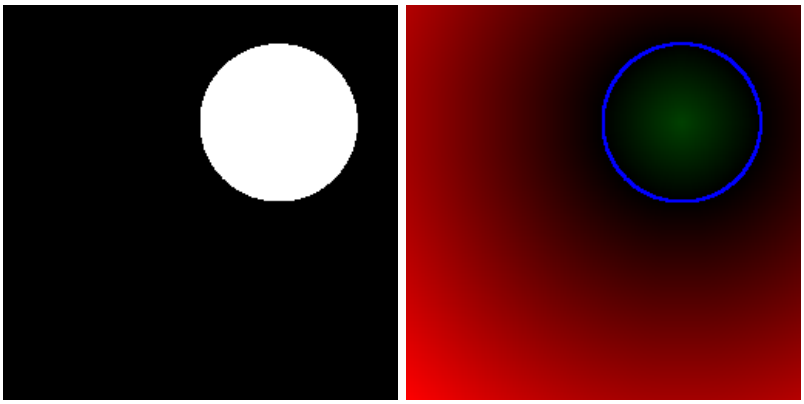

FIG. 8. *An offset circle (left) and its associated signed distance field (right)*

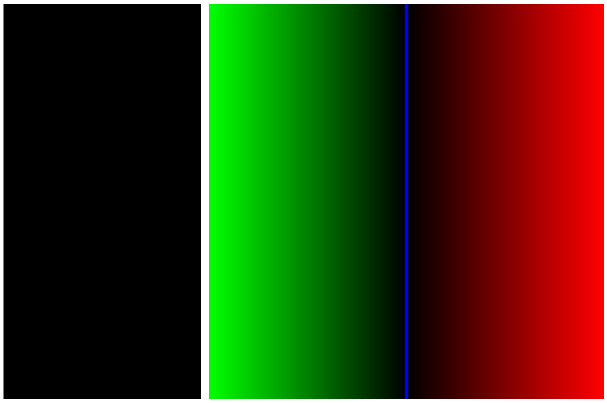

FIG. 9. *A region filling the left half of the image (left) and its associated signed distance field (right)*

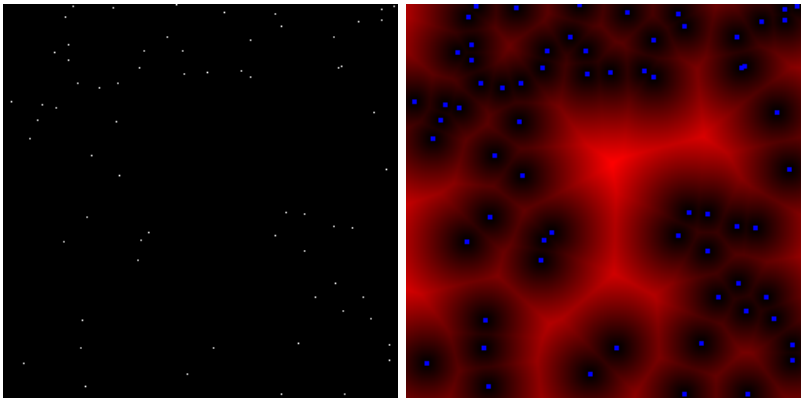

FIG. 10. *A set of randomly positioned dots (left) and its associated signed distance field (right)*

Appendix C. Benchmarks on Various Test Images.

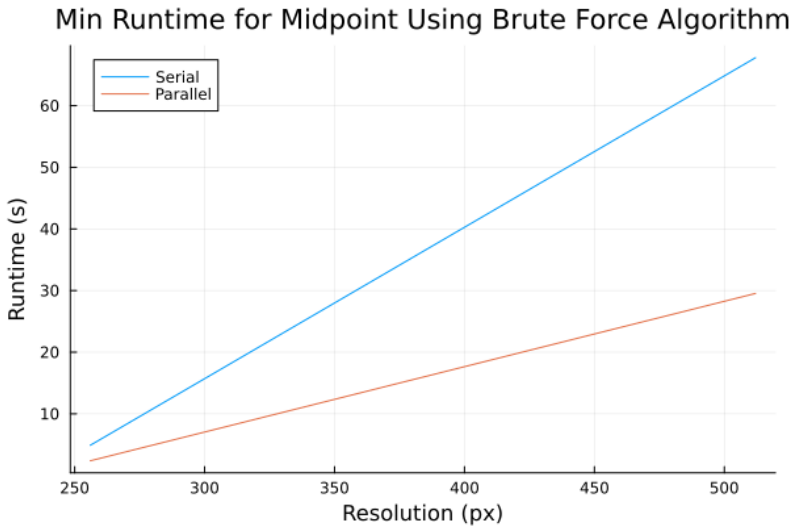

FIG. 11. *Runtimes for midpoint image with brute force algorithm. Note that only the* 256^2 *and* 512^2 *sizes were used.*

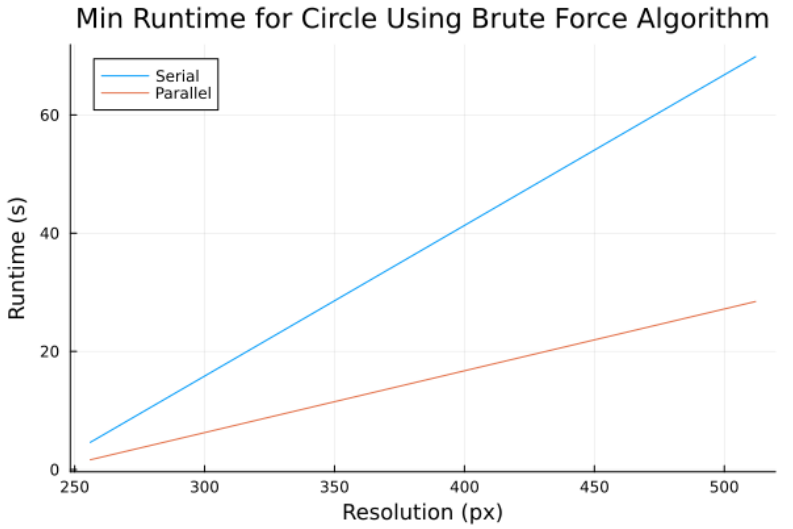

FIG. 12. *Runtimes for circle image with brute force algorithm. Note that only the* 256^2 *and* 512^2 *sizes were used.*

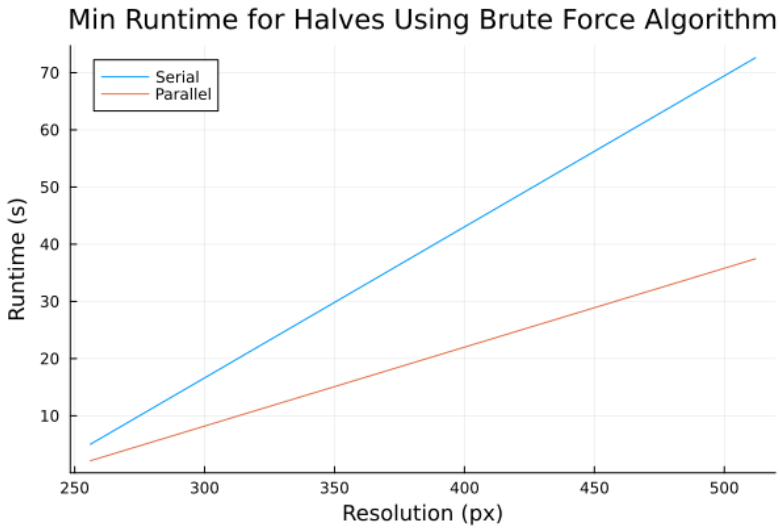

FIG. 13. *Runtimes for halves image with brute force algorithm. Note that only the* 256^2 *and* 512^2 *sizes were used.*

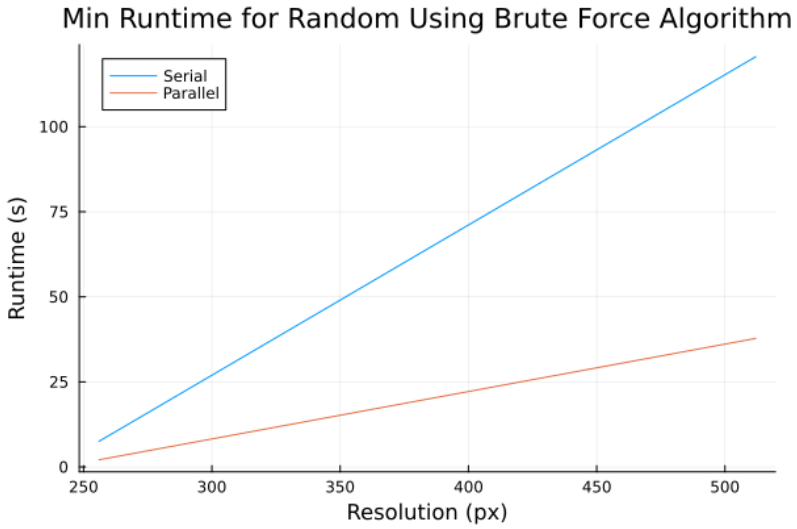

FIG. 14. *Runtimes for random dots image with brute force algorithm. Note that only the* 256^2 *and* 512^2 *sizes were used.*

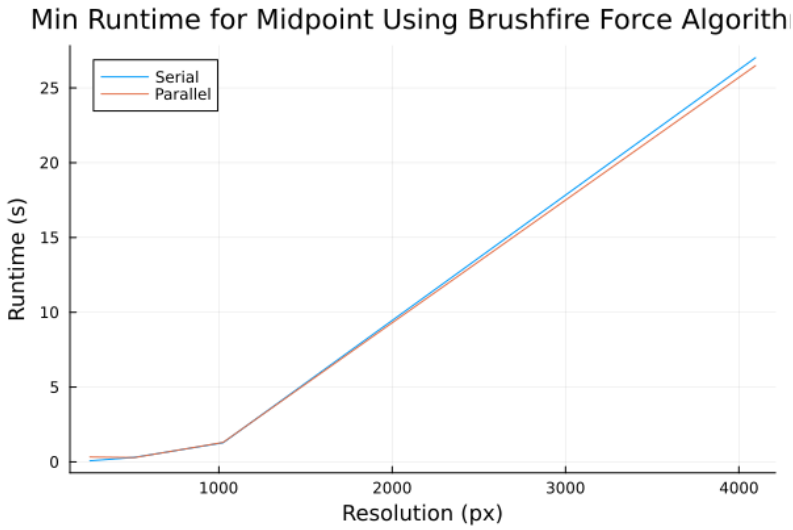

FIG. 15. *Runtimes for midpoint image with brushfire algorithm. Note that only up to* 4096^2 *sized-images were used.*

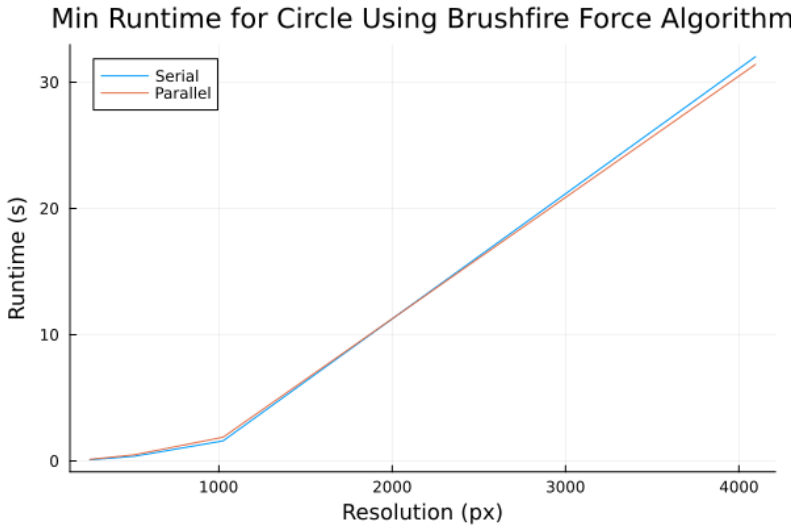

FIG. 16. *Runtimes for circle image with brushfire algorithm. Note that only up to* 4096^2 *sized-images were used.*

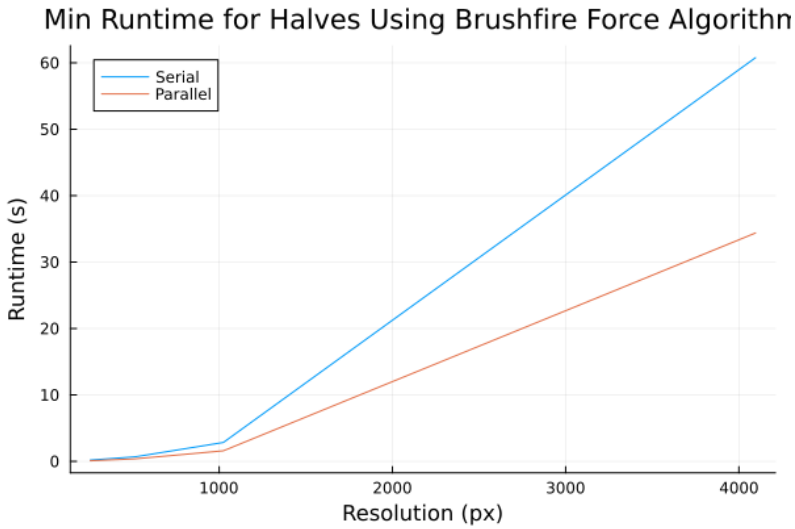

FIG. 17. *Runtimes for halves image with brushfire algorithm. Note that only up to* 4096^2 *sized-images were used.*

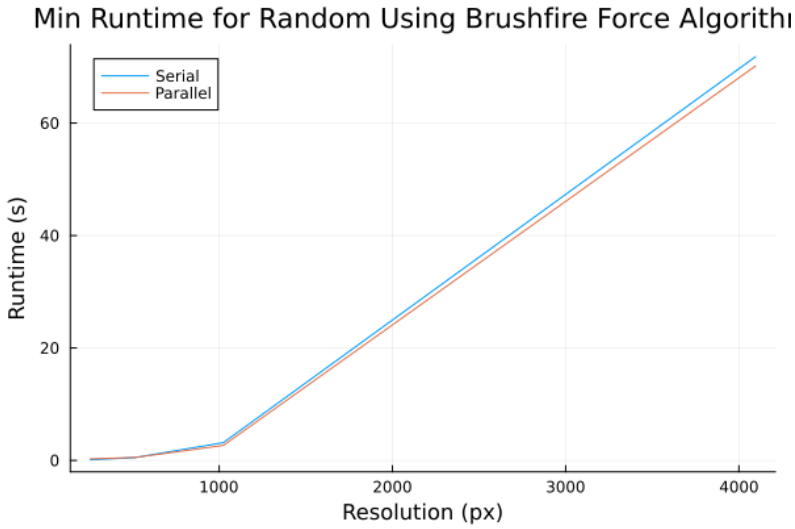

FIG. 18. *Runtimes for random dots image with brushfire algorithm. Note that only up to* 4096^2 *sized-images were used.*

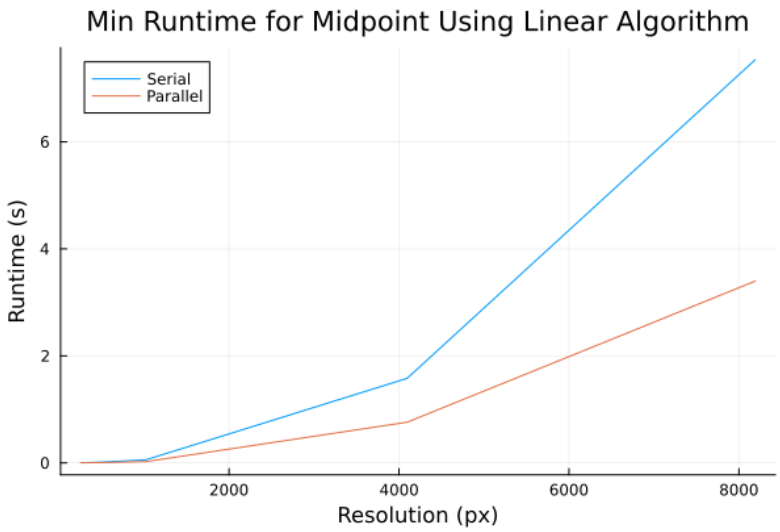

FIG. 19. *Runtimes for midpoint image with linear-time algorithm.*

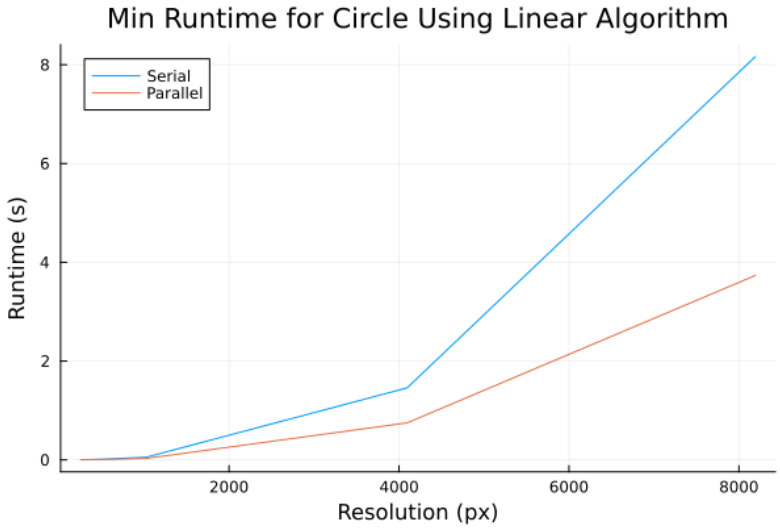

FIG. 20. *Runtimes for circle image with linear-time algorithm.*

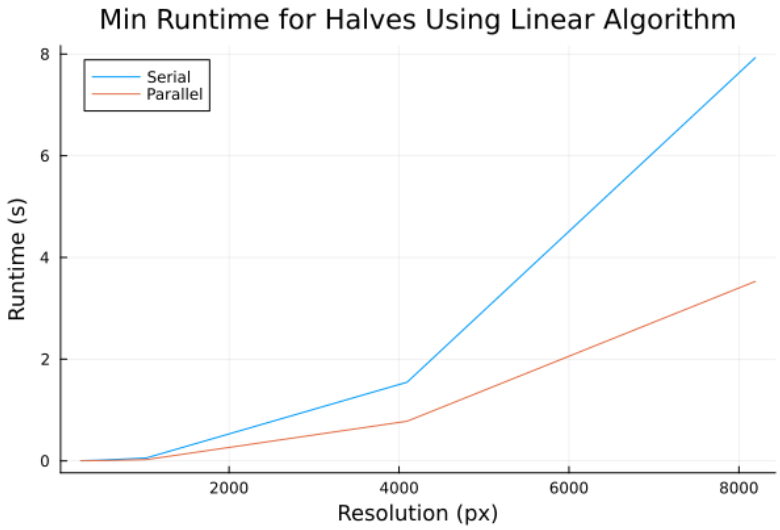

FIG. 21. *Runtimes for halves image with linear-time algorithm.*

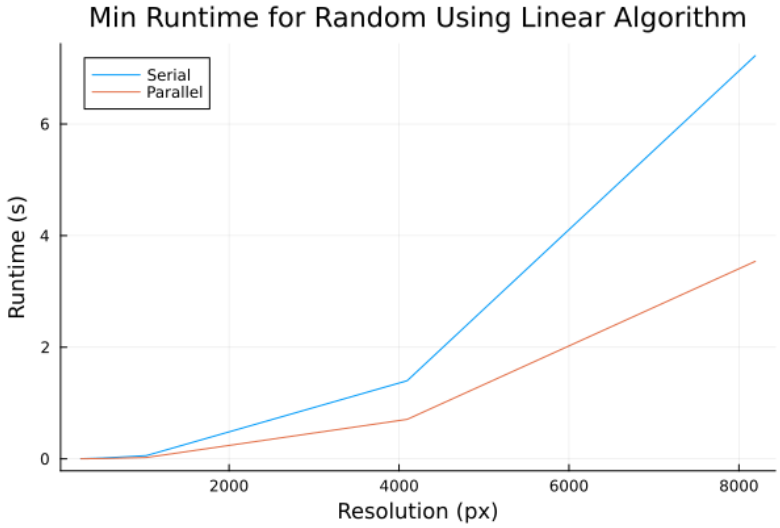

FIG. 22. *Runtimes for random dots image with linear-time algorithm.*

Appendix D. Memory Allocation Comparisons.

```
In [111]: 1 @time res = zeros(8192, 8192)
          0.989006 seconds (2 allocations: 512.000 MiB, 81.06% gc time)
```

FIG. 23. *Memory allocation time for 8192 × 8192 matrix*

```
@time res = zeros(512, 512)

  0.000714 seconds (2 allocations: 2.000 MiB)
```

FIG. 24. *Memory allocation time for 512 × 512 matrix*

```
@time Array{Int64}(undef, 8192)

  0.000013 seconds (2 allocations: 64.047 KiB)

8192-element Vector{Int64}:
        2102358876240
          17179869186
        2102358876742
           4294967296
        2102358876464
           4294967296
 -3088297230065764943
 -7407773142686400899
 -2021661474613068434
 -2763852922032132937
        2102358876320
          17179869184
 -5477222396993773743
                    ⋮
    29273400161599642
 -1657324216195717120
      256186224541696
    24769799426932824
                    0
       82463376998400
     2814749934878795
   3906362902780578304
   3688503268875778336
     9981362530422784
        2103005285072
        2101670563360
```

```
@time zeros(Int64, (8192, Threads.nthreads()))

  0.001417 seconds (2 allocations: 768.047 KiB)

8192×12 Matrix{Int64}:
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 ⋮              ⋮              ⋮
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
```

```
0.001417 / 0.000013

109.00000000000001
```

FIG. 25. *Memory allocation time for* $8192 \times p$ *array for* $p = 1, 6$