Parallel Scientific Computing
Course Project
May 2, 2024

PARALLEL IMPLEMENTATION OF MAZE SOLVING ALGORITHMS

M Keerthy Vasan
ME20B095

ABSTRACT

Mazes are abstract representations of graphs in graph theory and has extended practical applications in robotics, computer science, game design, operations research, navigation and network routing. Path finding algorithms that can solve mazes for a variety of parameters like most efficient route for multiple parameters, shortest route (both in terms of distance and time taken) are continuously being devised, refined and deployed. Some of the most popular algorithms used to solve mazes include depth-first search and breadth-first search. Taking advantage of multi-core processors, these algorithms can be parallelized using OpenMP. For this project, such a parallel version of these algorithms is implemented for some large and complex mazes and metrics like runtime, speedup and efficiency are analysed.

NOMENCLATURE

The nomenclature for the symbols used in this report is given below

E	Set of edges in the graph
M	Maze image
N	Number of pixels in the side length of the square maze
N_s	Set of nodes in the graph
P	Total Number of Threads
Q	Queue
T_Q	Thread Private Queue
S	Stack
T_S	Thread Private Stack
V	Set of vertices in the graph
W	Work balance Array
N_n	Set of neighboring nodes of node n
e	Edge between two vertices in the graph
n	Node in graph
p	Pixel in the maze
p_w	White pixel
p_b	Black pixel
s	Start node in the graph
t	End node in the graph
v	Vertex in the graph

INTRODUCTION

Mazes are in some sense, a dual problem of solving graphs in general. Hence, potential algorithms that can solve mazes effectively and efficiently, can then be extended to solve graphs and their associated problems. This report first converts the maze, to a graph with nodes, connects the nodes both horizontally and vertically and finds a path (or) multiple paths (if exists) from start node to end node using depth-first search and breadth-first search algorithms.

The time complexity of creating nodes and connecting them horizontally and vertically is of the order $O(N^2)$ and the time complexity of both depth-first search and breadth-first search is O(V+ E). Other algorithms like Dijkstra's algorithm and A^* algorithms, while can be used, do not yield more benefit compared to breadth-first search due to the way the graph is constructed.

Parallel implementation of both of these algorithms, using OpenMP, provides a consistent speedup for large mazes of the order N = 6000 and can be potentially used to solve large graph theory problems in general.

The maze images used have the following nomenclature *maze_typexxxx* where *maze_type* has three types namely

i) perfect (only one path exists),

ii) braid (multiple paths exist),
iii) combo (multiple paths exists like braid but many paths have dead ends)

and *xxxx* denotes the number of pixels in the side length of the square maze ranging from 200 to 6000.

CONSTRUCTION OF MAZE

The algorithms devised in this report are designed to solve mazes, that follow some rule set. These are

i) Mazes must be square.
ii) Mazes must contain only black pixels denoting walls and white pixels denoting paths.
iii) Mazes must be surrounded by walls on all four sides except for the start node and end node.
iv) Mazes must contain a start node at the top wall and end node at the bottom wall.
v) Mazes must have at least one path from start node to end node.

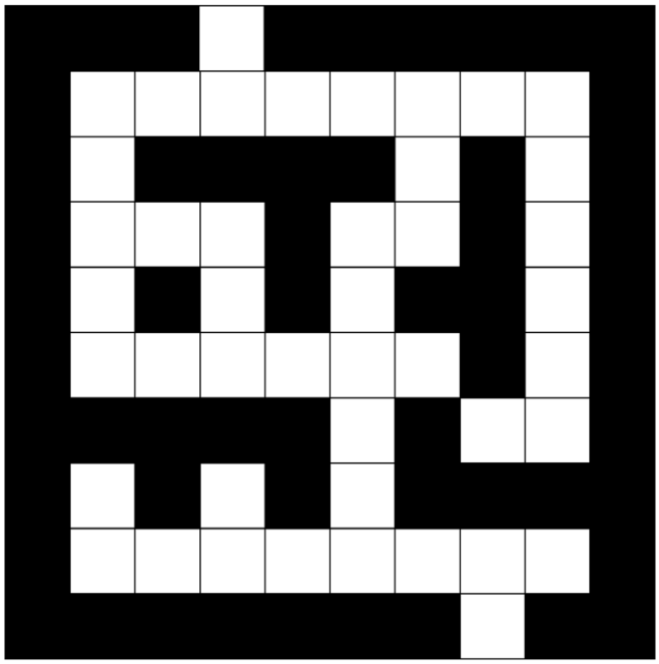


FIGURE 1: Sample maze constructed using the rule set

CONSTRUCTION OF GRAPH

The maze images are converted to graphs with nodes (an object in memory) with attributes like their position, neighbors, processed status and parent, connected by edges. The node objects although take more memory than the pixels themselves actually provide a less memory footprint in total. For example, take the combo6k image, meaning it has 6000 pixels on each side (or) 3.6×10^7 pixels in total. Approximately, half of them are white pixels (i.e.) 1.8×10^7 pixels. When the path finding algorithms are implemented directly on these white pixels, the pixels along the path also get processed, leading to increased computation time. The same maze constructed as a graph has approximately 6.1×10^6 nodes. That means without the graph structure, the maze has *2.95* times more nodes than required. The implication is that, there is a tremendous amount of memory saved by using the graph nodes which drastically undercuts the memory used by them as objects in memory compared to treating the white pixels themselves as nodes of the graph.

Construction of Nodes

Nodes are constructed based on the following algorithm. Dead-ends have 1 neighboring white pixel, T-junctions have 3 neighboring white pixels and both along a path and L-junctions have 2 neighboring white pixels. Nodes are not created on walls.

Algorithm 1: Creating nodes

Input: M
Output: Maze image with nodes

```
for p ∈ M in 1st row do
    if p is p_w then
        Create start node and mark its position.
    end

for p ∈ M in Nth row do
    if p is p_w then
        Create end node and mark its position.
    end

for j ∈ [2, N − 1] do
    for p ∈ M in jth row do
        if p is p_b then
            skip
        else
            if p has a dead-end (or) T-junction then
                Create node and mark its position.
            else if p has a L junction then
                Create node and mark its position
    end
end
```

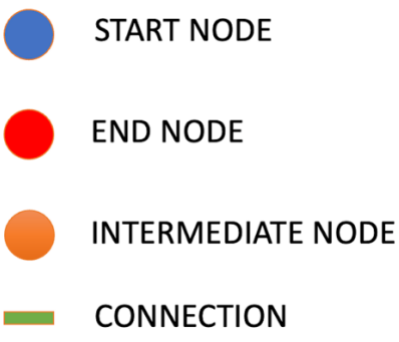
Connection of Nodes

Nodes are connected, first using a horizontal sweep and then using a vertical sweep by the following algorithm and this step's parallelization is trivial.

Algorithm 2: Connecting nodes

Input: Maze image with nodes $\in N_s$
Output: Graph with nodes

```
1   for j ∈ [2, N − 1] do
2      previous node ∈ Ns is null
3      for p ∈ M in jth row do
4         if p is pb then
5            Set previous node to null.
6         else if p is pw and is a node n ∈ Ns then
7            if previous node is null then
8               Set previous node to n.
9            else if previous node is not null then
10              Connect n & previous node.
11              Set previous node to n.
12     end
13  end
14
15  for j ∈ [1, N] do
16     previous node ∈ Ns is null
17     for p ∈ M in jth column do
18        if p is pb then
19           Set previous node to null.
20        else if p is pw and is a node n ∈ Ns then
21           if previous node is null then
22              Set previous node to n.
23           else if previous node is not null then
24              Connect n & previous node.
25              Set previous node to n.
26     end
27  end
```



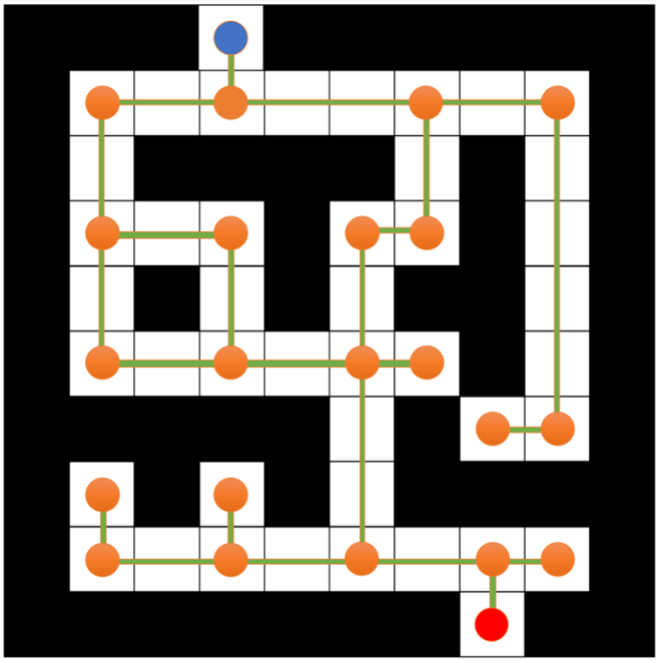


FIGURE 2: Graph created from the maze, has 23 nodes, connected by edges

DEPTH-FIRST SEARCH

Depth-first search is a graph traversal technique, where the path finding algorithm traverses the nodes in a depth wise fashion, (i.e.) going along a single path, until an end node is reached, in which case the program terminates (or) a dead end is reached, in which case the traversal traces back to the previous depth and expands from there. This algorithm is facilitated by the *Stack,* a LIFO (Last In First Out) data structure. The algorithm for depth-first search is given below

Algorithm 3: Depth – First Search

Input: Graph with nodes connected by edges
Output: Path from s to t

```
1  Add s ∈ Ns to S and mark it as visited.
2  while S is not empty do
3     Pop c ∈ Ns from S
4     for n ∈ Nc do
5        Set n's parent to c
6        Mark n as visited
7        Place n in S
8
9        if n is e then
10          Clear S and break
```

PARALLEL DEPTH-FIRST SEARCH

Parallelizing depth-first search is involving and has a few steps. First, run the serial depth first search algorithm with stack S until S has at least 8 x P, number of nodes in it. If end node is reached within this procedure, then parallel depth-first search is not initialized. Else, each thread has a private stack T_S, and there is a shared work balance array W. Each thread is allocated nodes from S in round-robin fashion and all the threads are allowed to traverse the nodes in T_S in depth-first fashion. If T_S for any thread becomes empty, that thread can take nodes from W, which is populated by extra nodes from T_S of other threads that has higher number of nodes to process. A shared Boolean variable is set to true if the end node is found and the program terminates. The race condition incurred by updating the same node in the graph by multiple threads is benign and is not a problem. This strategy is implemented in the following algorithm.

Algorithm 4: Parallel Depth-First Search

Input: Graph with nodes connected by edges
Output: Path from *s* to *t*

```
1  Perform serial DFS with S.
2  if e is found then terminate.
3  else do the following
4
5  Initialize TS for all P threads and shared W.
6  Set shared indicator Boolean I to False.
7  #pragma omp parallel num_threads(P) {
8   Divide the nodes from S to TS in a round-robin
9   while I is false do
10     if TS is empty then
11  #pragma omp critical
12  {take n ∈ Ns from W if W is non-empty.}
13        else
14         Pop c ∈ Ns from TS.
15         for n ∈ Nc do
16            Set n's parent to c.
17             Mark n as visited.
18             if |W| <= 2 & |TS| >= 8 then
19                Add n to W.
20            else
21                Place n in TS
22              if n is e then
23                 Clear TS.
24                 Set I to True.
25  #pragma omp flush // Update all P threads
26                  break
27           end
28  end
29  }
```

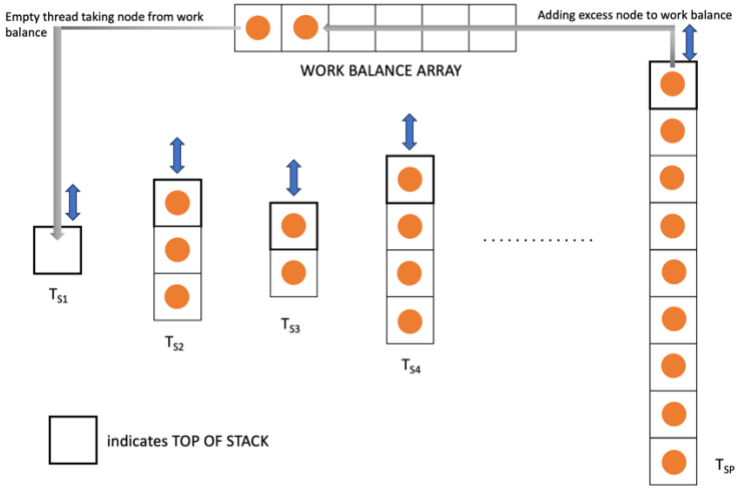


FIGURE 3: Illustration of Work balancing in Parallel Depth-First Search

BREADTH-FIRST SEARCH

Breadth-first search is a graph traversal technique, where the path finding algorithm traverses the nodes at one level first and processes them before traversing to the next level (i.e.) in a breadth wise fashion, until the end node is reached in which case the program terminates. The algorithm is facilitated by the *Queue*, a FIFO (First In First Out) data structure. The algorithm for breadth-first search is given below.

Algorithm 5: Breadth-First Search

Input: Graph with nodes connected by edges
Output: Path from *s* to *t*

```
1  Add s ∈ Ns to Q and mark it as visited.
2  while Q is not empty do
3     Pop c ∈ Ns from Q
4     for n ∈ Nc do
5        Set n's parent to c
6        Mark n as visited
7        Place n in Q
8
9       if n is e then
10          Clear Q and break
```

PARALLEL BREADTH-FIRST SEARCH

Parallelizing breadth-first search is involving and has a few steps. First, run the serial breadth first search algorithm with queue Q until Q has at least 8 x P, number of nodes in it. If end node is reached within this procedure, then parallel breadth-first search is not initialized. Else, each has thread has private queue T_Q, and there is a shared work balance array W. Each thread is allocated nodes from Q in round-robin fashion and all the threads are allowed to traverse the nodes in T_Q in breadth-first fashion. If T_Q for any thread becomes empty, that thread can take nodes from W, which is populated by extra nodes from T_Q of other threads that has higher number of nodes to process. A shared Boolean indicator variable I is set to true if the end node is found and the program terminates. The race condition incurred by updating the same node in the graph my multiple threads in benign and is not a problem. This strategy is implemented in the following algorithm.

Algorithm 6: Parallel Breadth-First Search

Input: Graph with nodes connected by edges
Output: Path from *s* to *t*

```
1  Perform serial BFS with Q.
2  if e is found then terminate.
3  else do the following
4
5  Initialize TS for all P threads and shared W.
6  Set shared indicator Boolean I to False.
7  #pragma omp parallel num_threads(P) {
8   Divide nodes from Q to TQ in a round-robin
9   while I is false do
10      if TQ is empty then
11  #pragma omp critical
12  {
13     take n ∈ Ns from W if W is non-empty.
14    }
15        else
16         Pop c ∈ NS from TQ.
17         for n ∈ Nc do
18            Set n's parent to c.
19             Mark n as visited.
20             if |W| <= 2 & |TQ| >= 8 then
21                Add n to W.
22            else
23                Place n in TQ.
24             if n is e then
25                 Clear TQ.
26                 Set I to True.
27  #pragma omp flush // Update all P threads
28                 break
29          end
30  end
31  }
```

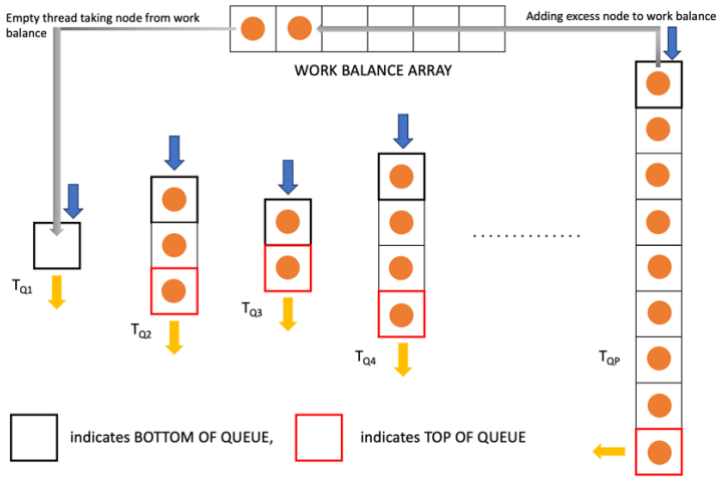


FIGURE 4: Illustration of work balancing in Parallel Breadth-First Search

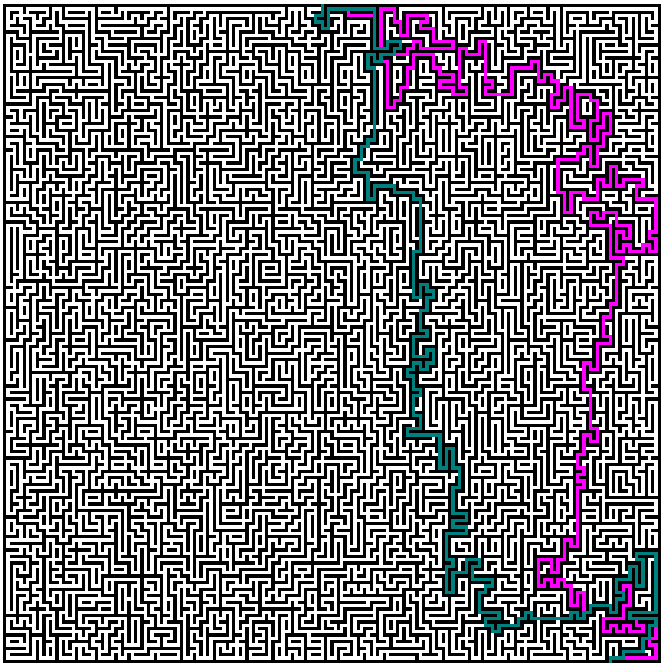


FIGURE 5: Maze (braid200) solved using serial depth-first search (magenta) and serial breadth-first search (teal)

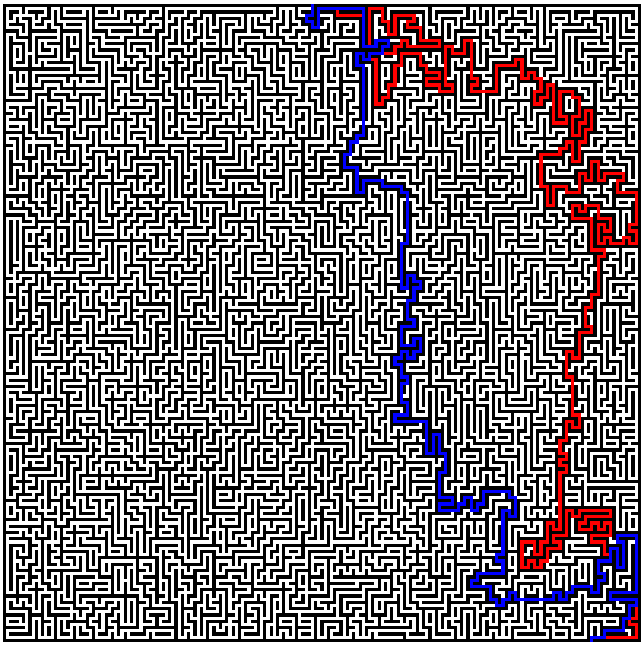


FIGURE 6: Maze (braid200) solved using parallel depth-first search (red) and parallel breadth-first search (blue)

RESULTS AND DISCUSSIONS

Input Images: Maze images have *N* as 200, 400, 2000, 4000, 6000. Mazes can be of different types namely, perfect, braid and combo.

In Fig 5. and Fig 6. both the serial and parallel version of breadth first search traverse through 193 nodes. The serial depth-first search traverses through 303 nodes while the parallel version traverses through 313 nodes.

As it can be seen both the serial and parallel versions of breadth-first solution, yield the same solution, but the depth-first solution results differ. But, it is to be noted that both the parallel algorithms produce inconsistent results when run multiple times. The above images are the best ones after multiple trials. This behaviour can be attributed to the independent running of thread private stacks (or) queues that may update the same node multiple times. However, the difference in the number of nodes especially in breadth-first search is very small and thus can be attributed as being a *nearly* parallel breadth-first search. Preventing such benign race conditions using omp critical makes the algorithm serial and pretty much gives up all the potential advantages of parallelism, since the goal is, to just solve the maze, in the least time possible

The running time for parallel depth-first search follows the expected trend of reduction in running time as *N* increases. But there is an anomaly for *N* = 4000 in parallel breadth-first search. The running time increases after *P* = 4. Since, it is a perfect maze, only one path exists from *s* to *t* and the overhead of running all the extra threads, outweighs the advantages of parallelism. In case of the speedup of *N* = 2000 of the parallel depth-first search, the reasoning here is that the first few paths taken by the serial depth-first already finds a path from *s* to *t*, hence negating the advantages of parallelism. However, *N* = 6000, shows a positive trend in speedup with increase in *P*, for both serial and parallel versions of these traversal algorithms, which is what we expect, an algorithm that works for large problem sizes. Other sizes, follow an inverted trend, where after some *P*, the overhead of parallelism outweighs the advantage of reduction in running time.

An almost downward trend can be observed for efficiency of both the parallel depth-first and breadth-first search, indicating that the program is not very efficient.

DEPTH – FIRST SEARCH

NUMBER OF THREADS (P)

SIDE LENGTH OF MAZE (N)	2	4	6	8	10	SERIAL
200	6.898	4.964	5.216	5.521	5.573	10.973
400	24.568	12.476	10.044	10.531	13.032	29.474
2000	201.228	269.14	280.68	228.167	222.959	200.023
4000	1461.96	1239.89	1254.32	1189.89	1166.57	2106.12
6000	5051.48	3694.51	3388.71	3077.22	2058.75	8158.89

Time taken is in milliseconds

TABLE 1: Time taken for depth-first search for different maze sizes and number of threads

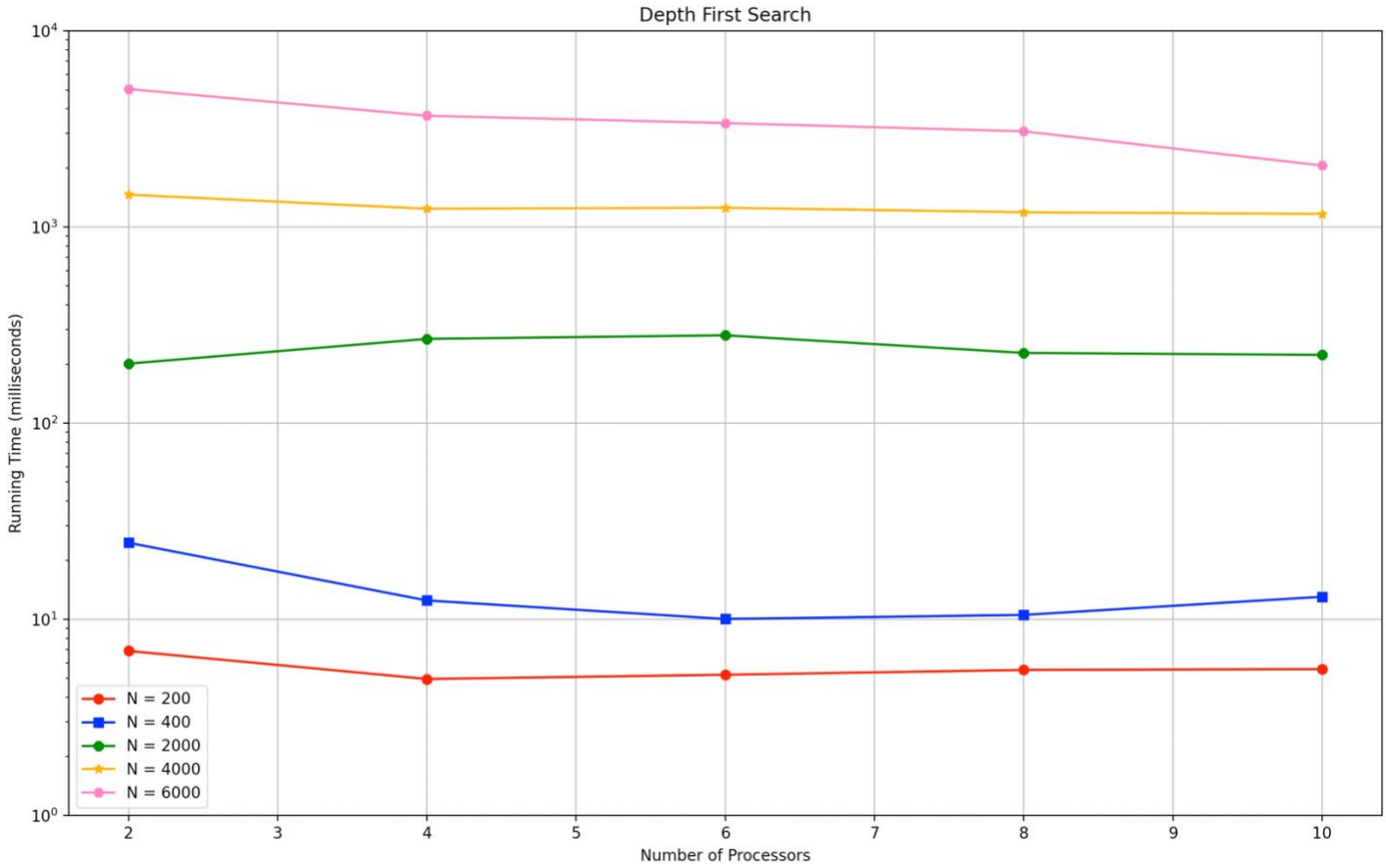
BREADTH – FIRST SEARCH

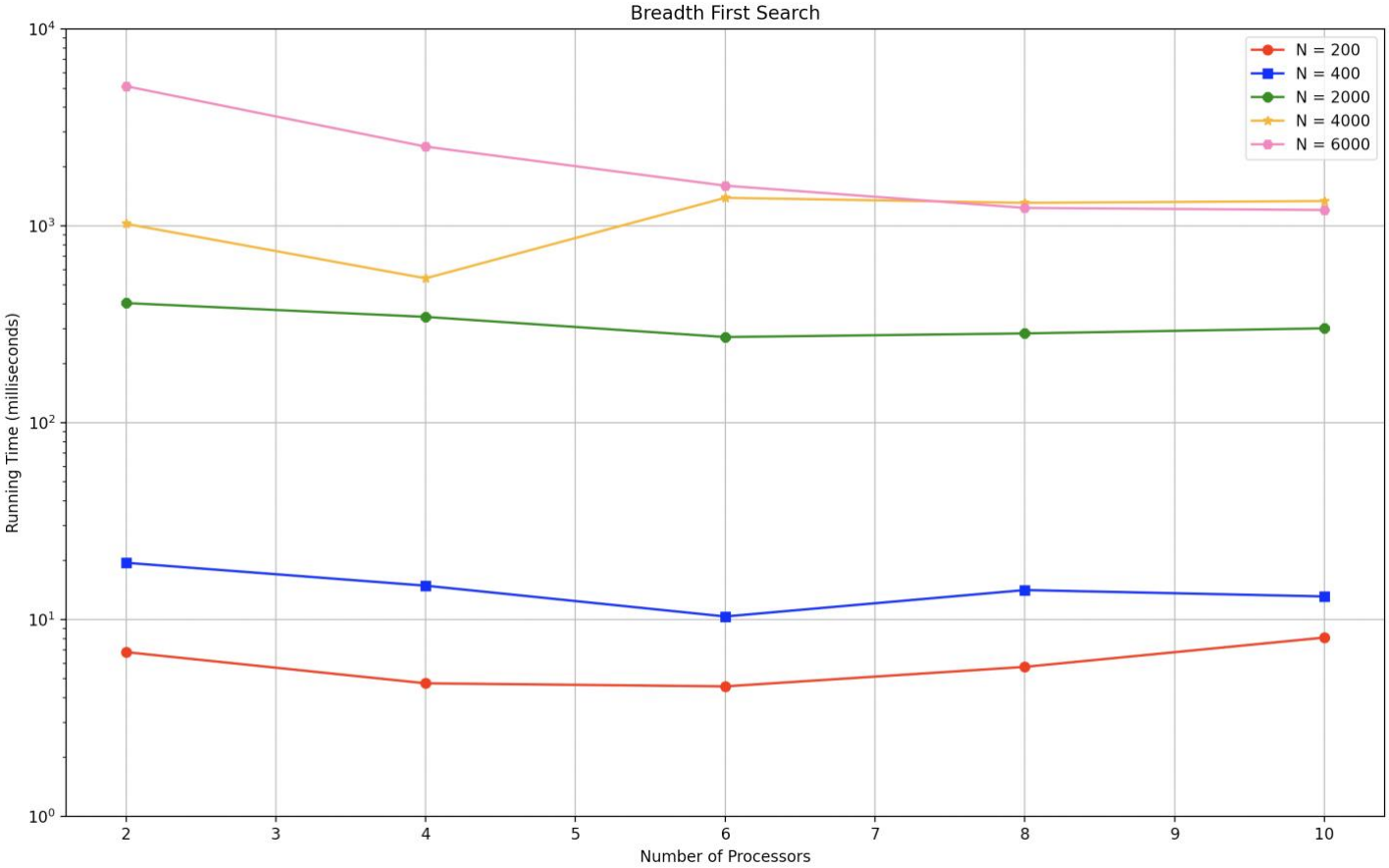
NUMBER OF THREADS (P)

SIDE LENGTH OF MAZE (N)	2	4	6	8	10	SERIAL
200	6.832	4.745	4.578	5.751	8.101	8.884
400	19.433	14.863	10.369	14.121	13.104	31.661
2000	405.434	345.095	272.54	284.42	301.899	508.545
4000	1024.7	541.69	1389.37	1309.21	1337.33	1968.58
6000	5125.14	2527.82	1600.78	1234.43	1204.99	8850.73

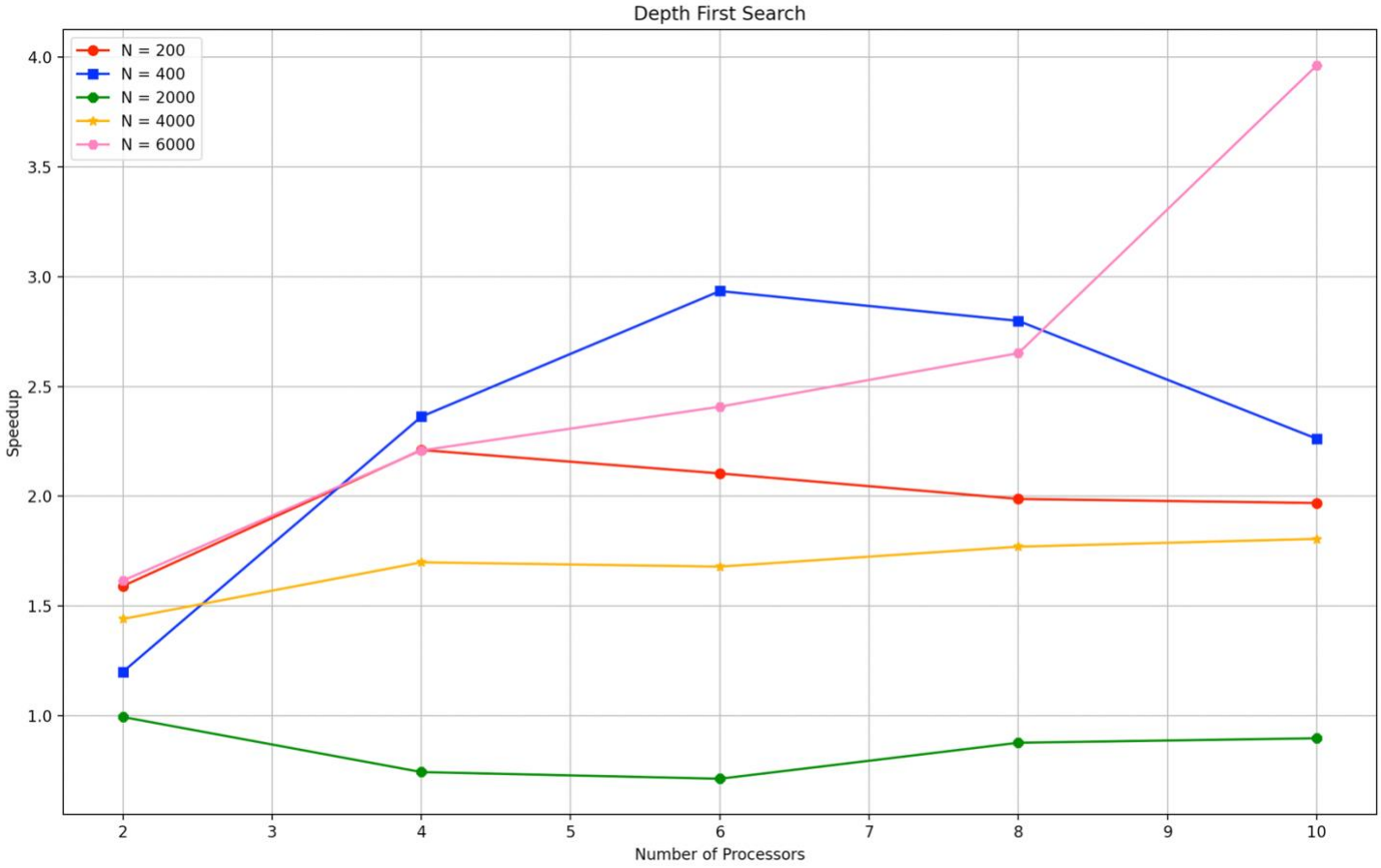
Time taken is in milliseconds

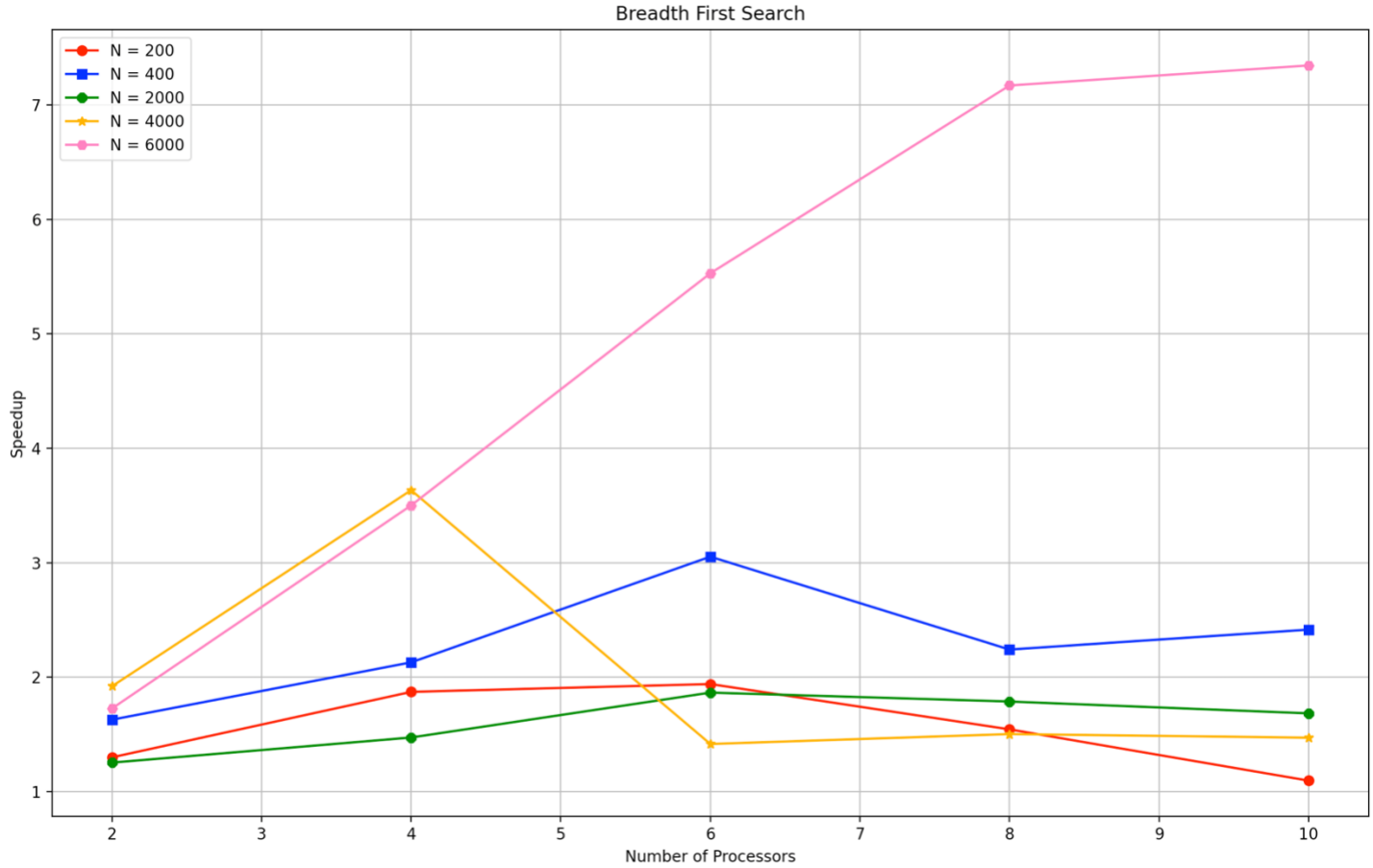
TABLE 2: Time taken for breadth-first search for different maze sizes and number of threads



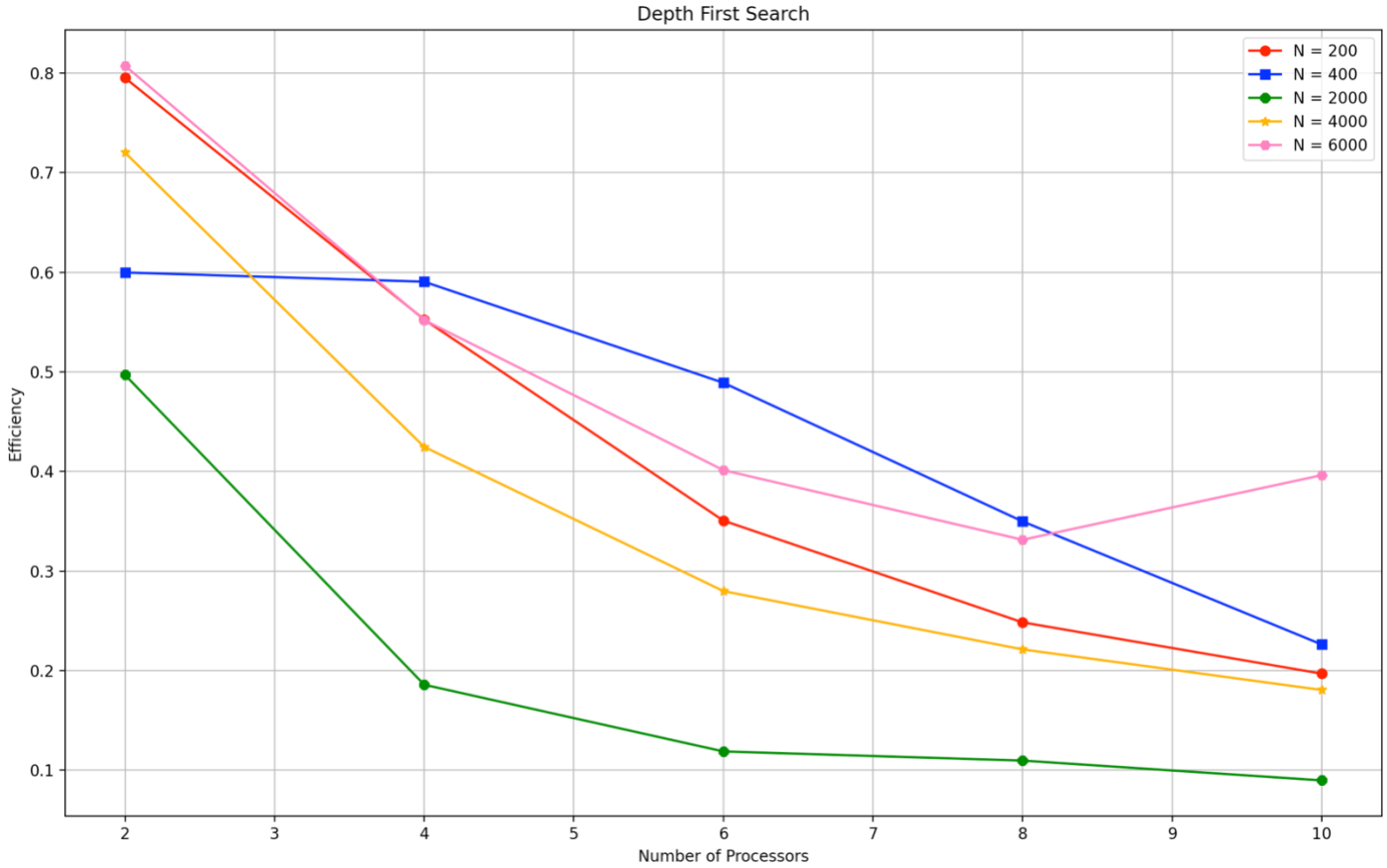


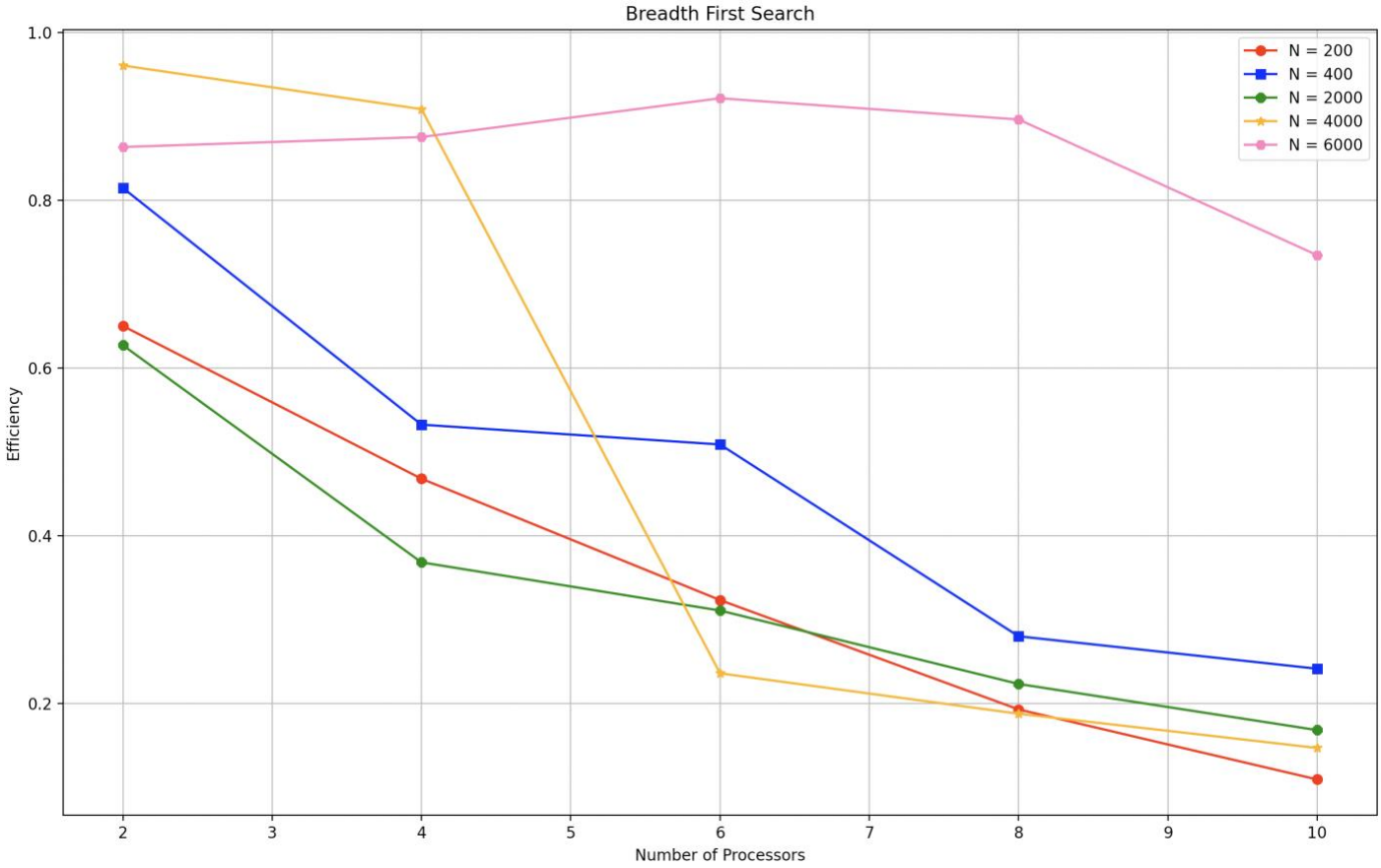
FIGURES 7 & 8: Running time vs *P* for different *N*, for parallel depth-first search and parallel breadth-first search respectively





FIGURES 9 & 10: Speedup vs *P* for different *N*, for parallel depth-first search and parallel breadth-first search respectively





FIGURES 10 & 11: Efficiency vs *P* for different *N*, for parallel depth-first search and parallel breadth-first search respectively

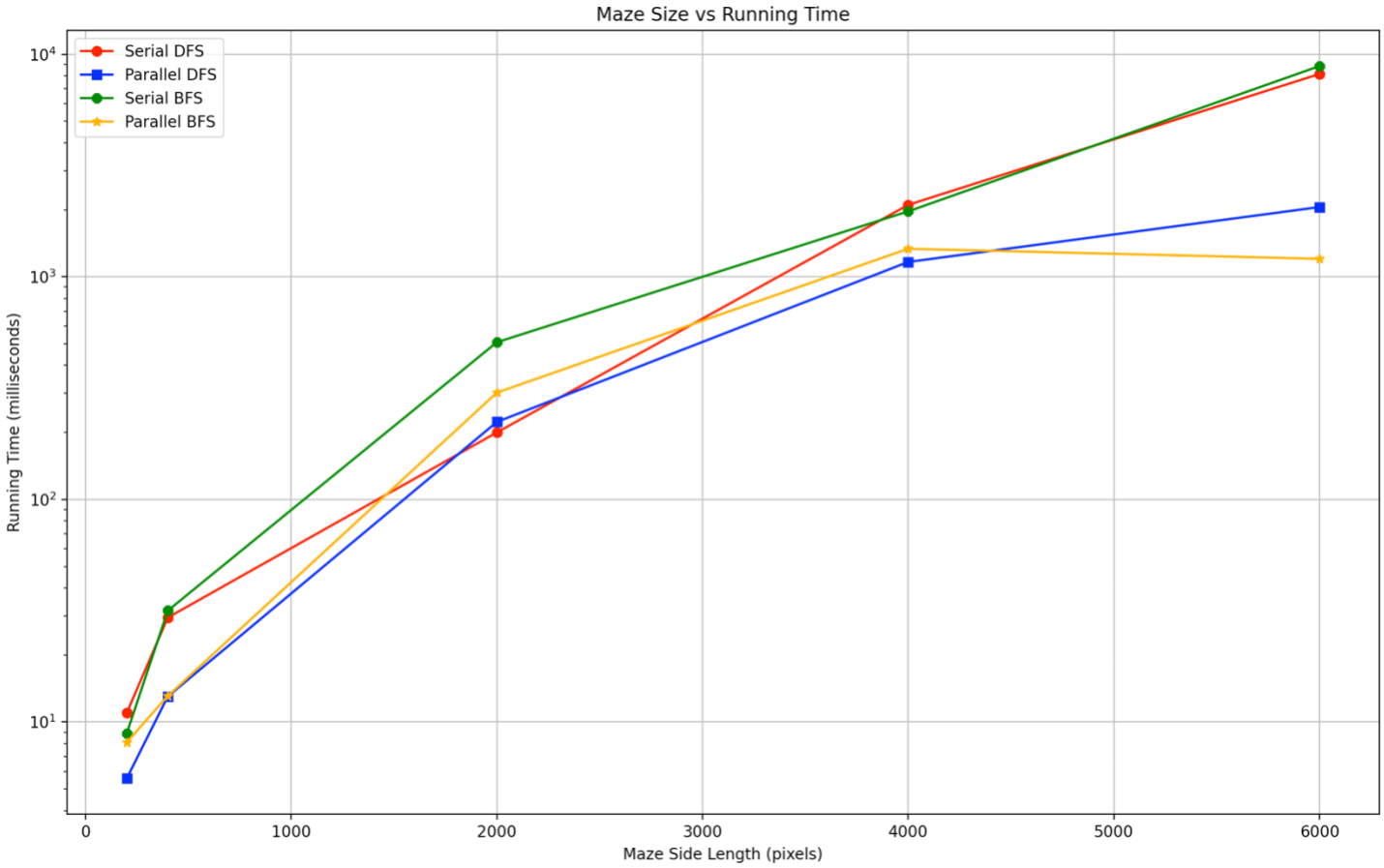


FIGURE 12: Efficiency vs *N*, for different path traversal algorithms

CONCLUSION

As part of this report, parallel versions of popular path traversal algorithms like depth-first search and breadth first search are presented, that can be run using multiple CPU Cores, for faster execution times in large mazes.

ACKNOWLEDGEMENT

I want to thank Dr. Kameswararao Anupindi for his sincere support.

ADDITIONAL REFERENCES

1. Breadth-First and Depth-First Search from www.geekforgeeks.org

2. Computerphile's Maze Solving Video for the inspiration to this project.

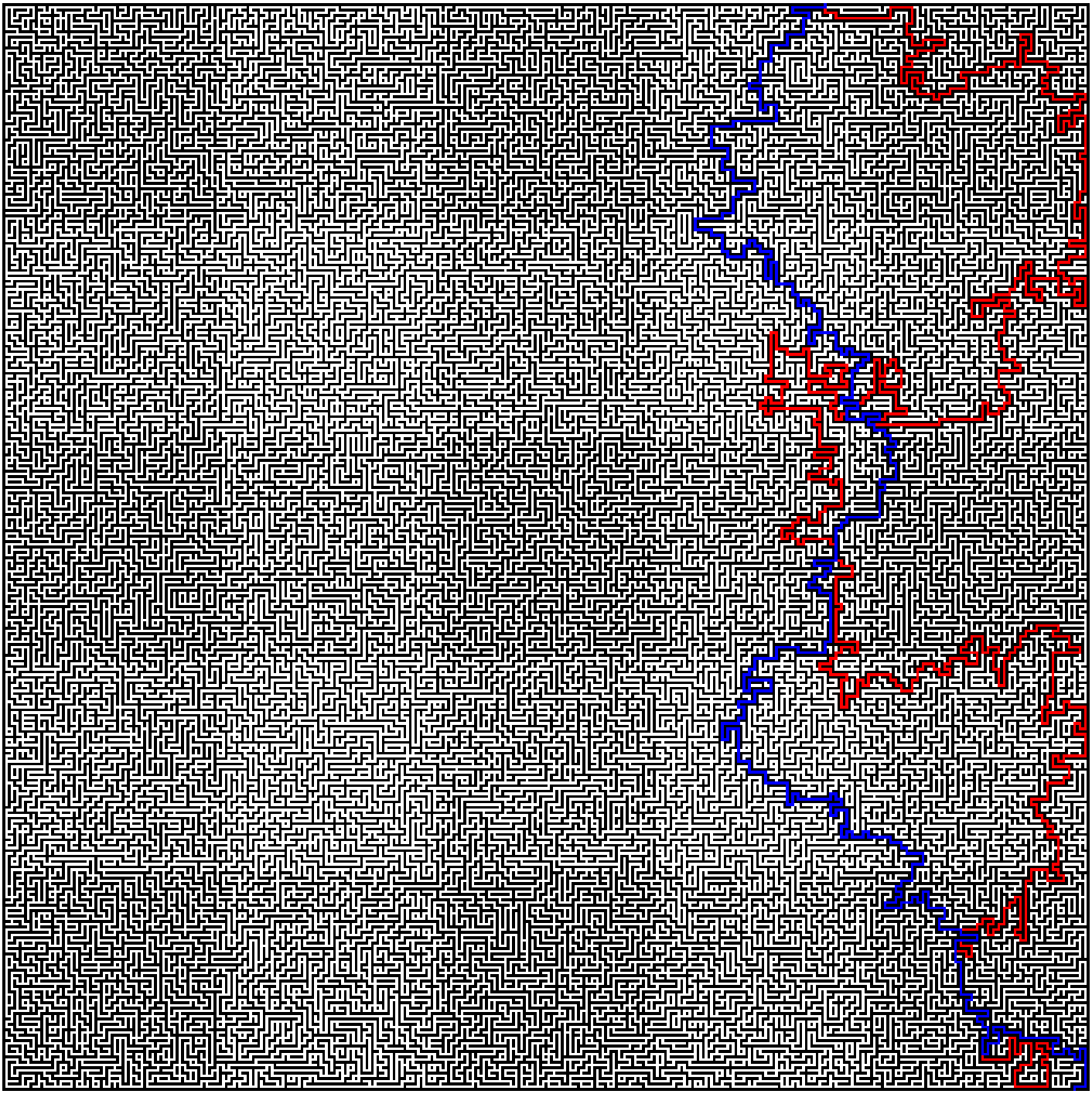


FIGURE 13: Maze (combo400) solved using parallel depth-first search (red) and parallel breadth-first search (blue)