Multi-thread Maze Problem

CSCI-GA-3033 Multicore Processors: Architecture & Programming Final Reprot

Zhihao Shu(zs2264), Junzhou Liu(jl12220), Dongzhe Fan(df2362), Yang He(yh4251)

NEW YORK UNIVERSITY

Courant Institute
New York University

Abstract

A maze problem is a type of puzzle where a person needs to navigate through a complex network of paths and passages to reach a specific endpoint. The goal is to find the most efficient path from the starting point to the endpoint, without getting lost or stuck in any dead-ends. Maze problem is important in the following domains: Robotics, Computer Vision, Game design and Artificial Intelligence etc. Typically, solving a maze problem entail determining the best path from the starting point to the endpoint while avoiding dead ends and other obstacles. Usually, the solver can be split to two types: search and dynamic programming. Search includes algorithms such as DFS, BFS, double-end BFS, and A*, while DP includes various variants of shortest path algorithms, which can be similarly classified as memetic search, and BFS-like shortest path (which essentially mimics Dijkstra). In our project, we tried several algorithms: In search solutions, we tried BFS, DFS, Double-end DFS. We compared normal solution with OpenMP version solution. In Dynamic programming, we set the value of path to 1 and the obstacle as -1. Then we can apply double ended Dijkstra to figure out the shortest path. We discovered that: We can not simply parallel DFS since it is based on recursion. So we used double-end DFS, with the input maze size became larger, the speed up for OpenMP version double ended DFS has significant improvement up to 1.3 when the maze size is 3000*3000. For future work, we also tried a solution to deal with larger maze which Mimics Dijkstra to find the shortest path. The speedup could have more remarkable improvement to 2.

1 Introduction

1.1 Definition of Maze Problem

A maze problem is a type of puzzle in which the goal is to find a specific endpoint or goal by navigating through a complex network of interconnected paths or corridors, often with dead ends and obstacles. The maze can be visualized as a two-dimensional grid or as a graph, with each cell or node representing a location in the maze and each edge representing a possible path between those locations.

In our project, we generate maze with 1(obstacles) and 0(path). We use the $n*n$ matrix to present graph. We set (0, 0) as source and (n-1, n-1) as destination. The maze is as Figure1

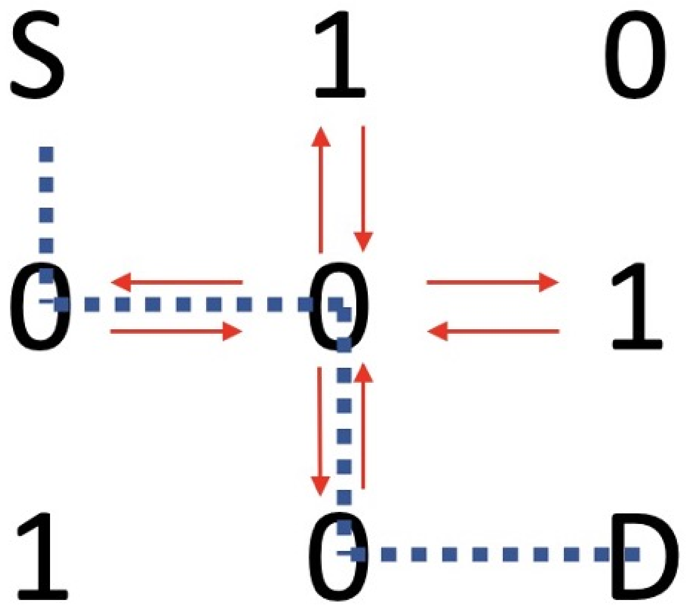


Figure 1: Maze Represnted by $n*n$ Matrix

As the diagram presented above, the path should be $(0,0) \rightarrow (1,0) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,2)$

1.2 The Importance of Solving the Maze Problem

Solving maze problem is of great significance in the following domains:

- **Robotics**: Navigating a maze is a fundamental problem in robotics, where robots must be able to navigate through unknown environments autonomously. Solving maze problems can aid in the development of algorithms that allow robots to explore and map new environments, avoid obstacles, and find the best paths to specific locations.
- **Computer vision**: Solving maze problems can also aid in the development of computer vision algorithms capable of recognizing and navigating complex scenes such as crowded streets or indoor environments with obstacles.
- **Game design**: Mazes are a classic element of game design, and solving maze problems can assist game designers in creating more challenging and engaging levels that require players to use problem-solving skills to progress.
- **Artificial intelligence**: Maze problems are commonly used as a benchmark in artificial intelligence to evaluate the performance of search and optimization algorithms. Researchers can develop more efficient and effective algorithms for a wide range of real-world problems by solving maze problems.

2 Literature Survey

Since 1991, many scientists have tried to maze problem or related problems with different parallel algorithms. In 1991, R.Mall proposed a PMAZE algorithm to solve the parallel maze routing problem on hypercube computers[1]. This algorithm consists of three different phases: concurrent wavefront expansion, parallel backtrace and concurrent cleanup. It is targeted to run on loosely coupled multiprocessor systems. This algorithm has some limitation, it only works on some specific systems, for most common multiprocessor systems, his algorithm doesn't work well, so a better algorithm is needed for more common situations. Then in 1993, Yen and Dubash try to implement Lee's maze routing algorithm on parallel systems[2]. They firstly modifies mapping algorithm by using the concept of mirror images. Then, they propose a dynamic mapping algorithm, which is shown to give an optimal mapping in an obstacle-free grid space. Besides using high efficient parallel algorithms, some scientists also try to use advanced devices. Bashuk tries to solve maze problem with a quantum computer which can generate a quantum parallelism effect that allows the computer to be in multiple states at the same time.[3]. But his method only get speedup on devices level. Combining high efficiency algorithm and advanced device will be a better way. Kumar and Goswami try to convert maze problem into a quantum search problem[4]. By accomplishing iterative Grover's Search Algorithm, they solve the maze problem more efficiency. Eddie make some steps in machine learning areas, he/she uses cellular simultaneous recurrent neural network(CSRN)[5]. He/she creates clusters by using relevant information and also add an additional external input to improve the model result. This algorithm combines Machine learning with maze problem, which is really a good idea. However it only works for small size mazes. Both Kumar and White's algorithm has a very high complexity. When the maze complexity increases, solving a maze problem will become more difficult and time consuming. Also, training the recurrent network will take a lot more time. Then, Yuriy. V and Massimiliano give rise to a new approach to sovle large size maze problems easily[6]. They use a network of network of memristors - resistors with memory-to calculate every mid-steps simultaneously. Their algorithm can find all the paths from the start to the end and the get the shortest path by sorting them. Then Liu and Xiao try ant colony algorithm to find the shortest path in a maze[7]. They firstly introduce a parallel matrix design method and then use ant colony searching process to find the optimal path. During this process, the data are rationally organized and stored to increase the algorithm speed. Solving the maze problem is not only about find the path, but also includes how to generate a large size maze in a short time. Muthuselvi and Sridevi try to use Kruskal algorithm to generate the maze and they use OpenMP to make the algorithm be operated in parallel[8]. Making the generating process parallel will indeed save some time. However, they work only concentrate on making the generating process parallel, which is only a small part of the maze problem. In order to solve the maze problem, a high efficiency algorithm suitable for every kind of devices is still of great importance.

3 Proposed Idea

3.1 DFS with OpenMP

One of the most popular algorithm to solve the maze problem is **Depth-First Search(DFS)**. Using the idea of the back-sheet method, vanilla DFS algorithm search into four directions on one step. In our project, we try to operate these four search processes simultaneously. Vanilla DFS algorithm is implemented by recursion method. We utilize the **omp task** function to parallel this recursive process. For omp task, it will put each recursion step into the task pool, and when each thread finish its task, it will then operate the task in the task pool. We also need to use **nowait** field to ensure that the thread is also executing as a separate thread, but does not have to wait for the thread to finish executing, other threads can just execute down the line.

3.2 Double-Ended DFS with OpenMP and Dead-end Mark

The double-ended Depth-First Search (DFS) algorithm works by performing two concurrent DFS searches, one from the start point and another from the end point, until they meet or exhaust all possible paths. The algorithm uses OpenMP to parallelize the two searches, running them concurrently to potentially speed up the search process. Additionally, the exploration of neighboring cells is randomized to avoid bias in the search direction.

To define the meeting condition, the algorithm used an atomic operation which switches a shared Boolean variable to true whenever one DFS searcher meets the other one.

During the search process, the paths explored by the searchers are marked with different values (2 for the forward search, 3 for the backward search). When the two searchers meet, the algorithm concludes that a path has been found.

It also marks the meeting point and prints out where should it starts to solve the maze. Because in some cases, one searcher might go beyond the meeting points. Without specifying the start point, it may lead to ambiguous result. see example: 5.1

In case the DFS encounters a dead-end (i.e., a path that leads to no solution), it backtracks, marking the cell with the value 7 to indicate that it is part of an unsuccessful path. This approach helps differentiate between the successful path (marked with 2 and 3) and the dead-end path (marked with 7), making it easier to visualize and understand the search process when printing the maze.

To implement this parallel approach, the following steps were taken:

1. Use OpenMP sections: OpenMP sections are used to parallelize the two DFS calls. By using #pragma omp parallel sections, the code within the curly brackets is executed concurrently by different threads.

Listing 1: Parallelized Depth-First Search in Maze

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        dfs(startX, startY, endX, endY, ROWS, COLS, maze, found, 2, 3);
    }
    #pragma omp section
    {
        dfs(endX, endY, startX, startY, ROWS, COLS, maze, found, 3, 2);
    }
}
```

2. Atomic variables: The found variable is declared as an atomic variable (atomic¡bool¿ found(false);) to ensure that it is accessed and modified atomically by different threads. This prevents race conditions, where multiple threads access or modify the variable simultaneously, leading to inconsistent results.

3. Marking the maze: In the DFS function, the me and other variables are used to mark the maze cells visited by the current thread and the other thread, respectively. The values 2 and 3 are used for this purpose. In addition, a backtracking path cell is marked with a 7.

4. Path checking: During the DFS, if the current thread encounters a cell marked by the other thread or reaches the other thread's start/end point, it means a path has been found, and the found atomic variable is set to true. This signals the other thread to stop searching, as the path has already been discovered.

The choice of using OpenMP sections allows for a straightforward parallelization of the bidirectional search approach. The sections ensure that both DFS calls are executed concurrently, while atomic variables and the marking mechanism provide a way to handle synchronization between the threads. The bidirectional search can potentially reduce the search space and increase the speed of finding a solution, making it a suitable choice for parallelizing the problem.

4 Experiments Setup

4.1 Maze Generator

We use python code to generate the mazes using a randomized Depth-First Search algorithm. We use Numpy library to initialize a matrix. It starts from the upper-left corner and iteratively carves a path through the maze grid while maintaining a stack of visited cells. When there are no more possible moves from the current cell, it backtracks using the stack. The generated maze is then written to a text file. The produced maze must have an odd number size.

4.2 Base-line Algorithm

We implement vanilla DFS algorithm as the base-line to solve maze problem. Due to the limitation of the stack memory, we adjust the stack memory by using the **ulimit** command before the experiment. In order to solve the problem on a large size maze(large than $1000 * 1000$), we set the stack memory as 512MB by using **ulimit -s 512000**.

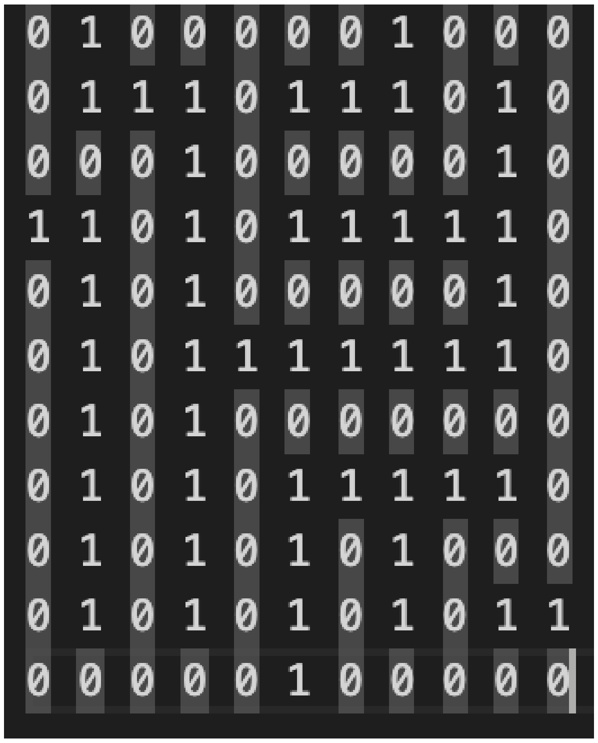
4.3 Experimental Environment Configuration and Operation Process

We select to run our code on CIMS machines. The specific environment is to do the compilation and execution on crunchy1 machines. First, we generate the required maze matrix using the maze_generator.py file and write it to the corresponding txt file. Then, since the default version on the cims server is 4.8.5, we switch to version 9.2 using the command "module load gcc-9.2". Next, as the DFS algorithm used involves recursion and has certain requirements for stack memory as mentioned above, we repeatedly tested and found that 512MB is suitable for maze matrices up to $3000 * 3000$ in size. Therefore, we use "ulimit -s 512000" to increase our stack memory. Finally, we use "g++ -fopenmp dfs.cpp -o dfs" and "g++ dfs.cpp -o dfs" to generate the corresponding OpenMP version and sequential version of the compiled files, respectively. We then use the generated compiled files to read the previously generated maze matrix and perform the final experimental operation. This allows us to obtain the time required to find a path out of the maze under different algorithms, facilitating subsequent analysis and comparisons.

5 Experiments and Analysis

5.1 Example Input and Output

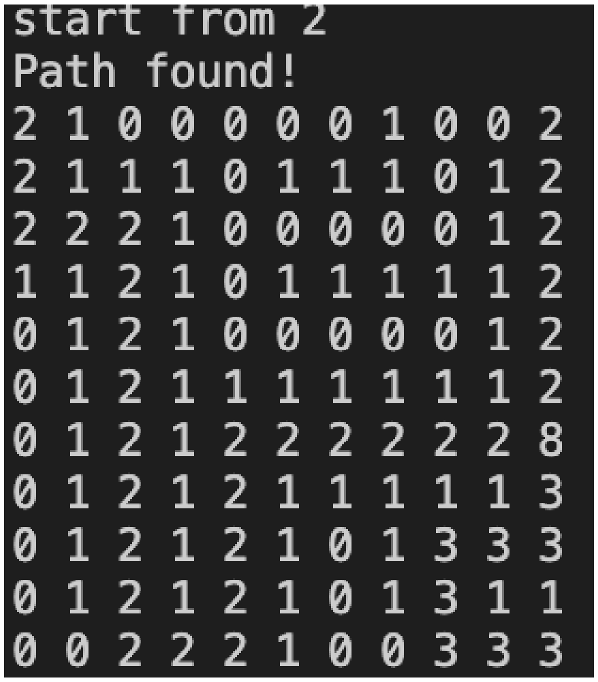
The generated maze is a squared maze, with '0' as path, and '1' as wall. The example input is as Figure2



```
0 1 0 0 0 0 0 1 0 0 0
0 1 1 1 0 1 1 1 0 1 0
0 0 0 1 0 0 0 0 0 1 0
1 1 0 1 0 1 1 1 1 1 0
0 1 0 1 0 0 0 0 0 1 0
0 1 0 1 1 1 1 1 1 1 0
0 1 0 1 0 0 0 0 0 0 0
0 1 0 1 0 1 1 1 1 1 0
0 1 0 1 0 1 0 1 0 0 0
0 1 0 1 0 1 0 1 0 1 1
0 0 0 0 0 1 0 0 0 0 0
```

Figure 2: Example Input

The example output will show the start point and notify whether a path is found. Also, it will print a matrix. The example output is as Figure3



```
start from 2
Path found!
2 1 0 0 0 0 0 1 0 0 2
2 1 1 1 0 1 1 1 0 1 2
2 2 2 1 0 0 0 0 0 1 2
1 1 2 1 0 1 1 1 1 1 2
0 1 2 1 0 0 0 0 0 1 2
0 1 2 1 1 1 1 1 1 1 2
0 1 2 1 2 2 2 2 2 2 8
0 1 2 1 2 1 1 1 1 1 3
0 1 2 1 2 1 0 1 3 3 3
0 1 2 1 2 1 0 1 3 1 1
0 0 2 2 2 1 0 0 3 3 3
```

Figure 3: Example Output

5.2 Results

As we are using the DFS algorithm to find the path out of the maze, we have found that there is a special case where some maze matrices may have a path to the exit on the first path selected at the beginning. This extreme case may be found by the DFS_DoubleEnd algorithm after multiple iterations, causing the regular sequential DFS algorithm to be faster than the former, with some degree of error. Therefore, we conducted ten different experiments on mazes of size $1000 * 1000$ to verify the stability of the algorithm used and to exclude the aforementioned special cases. The Figure4 shows the performance of different algorithm versions on ten maze matrices of size $1000 * 1000$. From the figure, it can be seen that the performance of each algorithm is relatively stable, without any outliers.

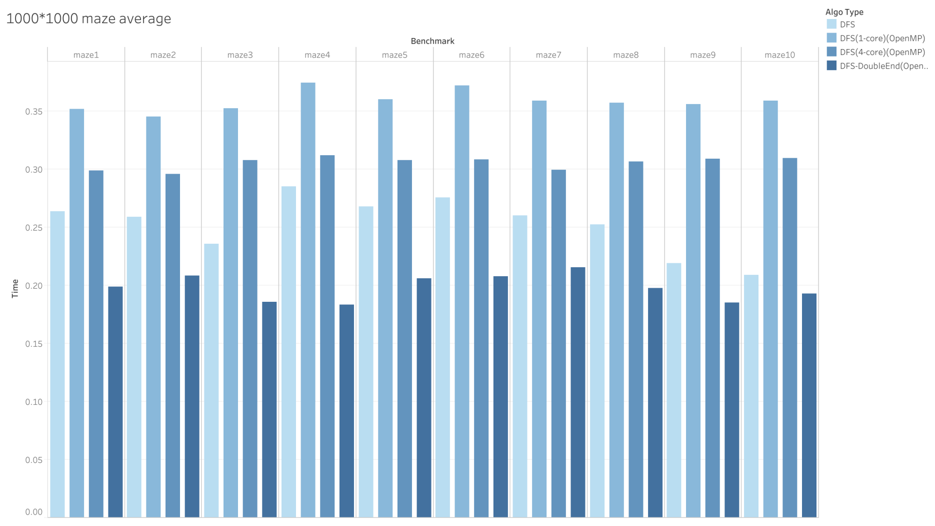


Figure 4: 1000*1000 Maze Average

Based on the above experiments, we conducted nine sets of control experiments on mazes of sizes $10*10$, $100 * 100$, $200 * 200$, $500 * 500$, $800 * 800$, $1000 * 1000$, $1500 * 1500$, $2000 * 2000$, and $3000 * 3000$. To eliminate random errors, we generated 10 different mazes of the same size for each maze size, and then took the average of the results. The experimental results are shown in the Table1 and Fingure5 below. From the data and the line chart in the table, it can be seen that as the size of the maze increases, the time required by each algorithm also increases gradually, and the performance differences become more apparent. Overall, the DFS_DoubleEnd algorithm has the lowest time growth rate and the best performance compared to the other three algorithms. DFS_Sequential is the second best, followed by DFS_4core and DFS_1core, respectively.

maze_size	DFS	DFS(1-core)	DFS(4-core)	DFS-DoubleEnd
$10 * 10$	0.0084	0.0086	0.0087	0.015
$100 * 100$	0.0122	0.0123	0.0124	0.0162
$200 * 200$	0.0196	0.024	0.021	0.0218
$500 * 500$	0.067	0.0934	0.0806	0.0628
$800 * 800$	0.1486	0.2412	0.2062	0.1294
$1000 * 1000$	0.2523	0.3568	0.3065	0.1976
$1500 * 1500$	0.6007	0.7630	0.6566	0.4686
$2000 * 2000$	0.9544	1.3802	1.2174	0.7286
$3000 * 3000$	2.4858	3.0418	2.6662	1.7572

Table 1: Run Time of Different Algorithm

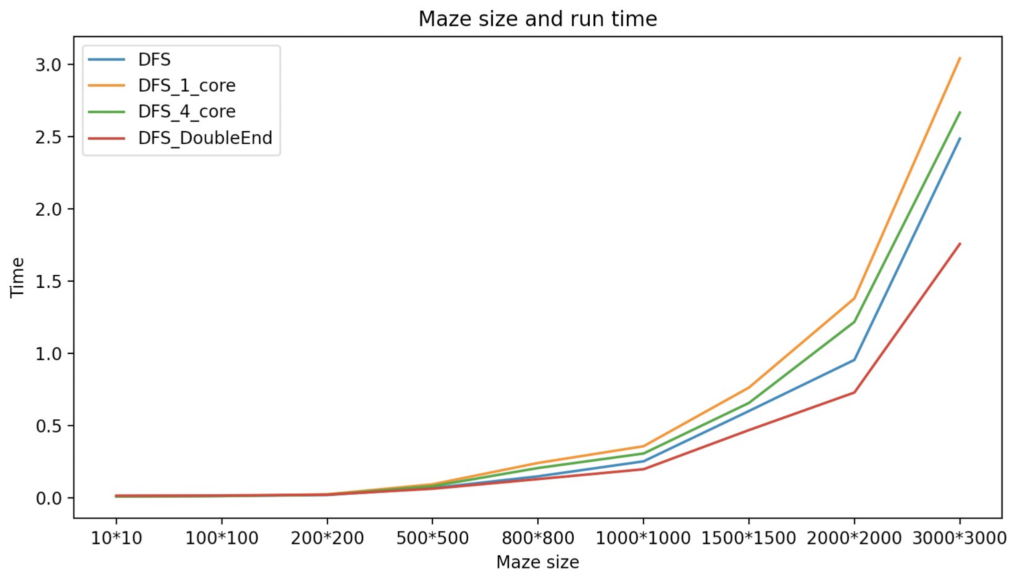


Figure 5: Run Time of Different Maze Size

In order to measure the efficiency and performance improvement of parallel computing on mazes of different sizes in the above experiments, we can calculate the Speedup value by comparing the time taken to execute the tasks using a single processor or core and the time taken to execute the tasks using multiple processors or cores. Through the Speedup value, we can understand the effect and performance improvement of parallel computing. The specific results are shown in the Figure6:

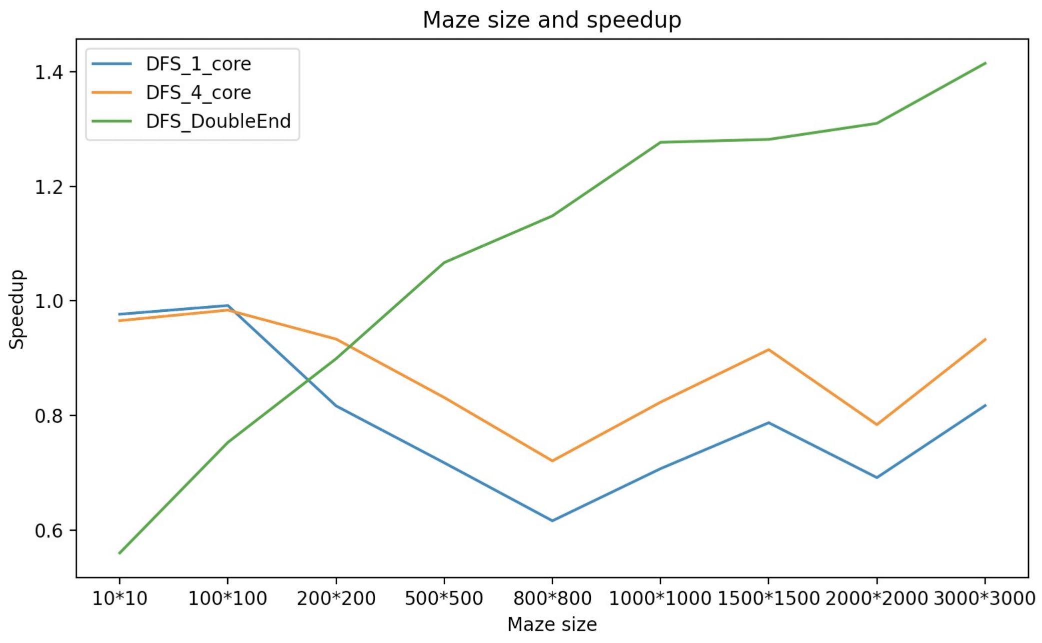


Figure 6: Speedup of Different Algorithm

Finally, based on the above data, we calculated the performance improvement efficiency of DoubleEnd, using the DFS_Sequential's experiment time as the baseline. As the size of the maze matrix being tested increases (from $10 * 10$ to $3000 * 3000$ in the current experiment), the efficiency of DoubleEnd's performance improvement has gradually increased from negative optimization to about 30%.The improvement effect is quite significant and consistent with our expected results. The specific performance is shown in the Figure7.

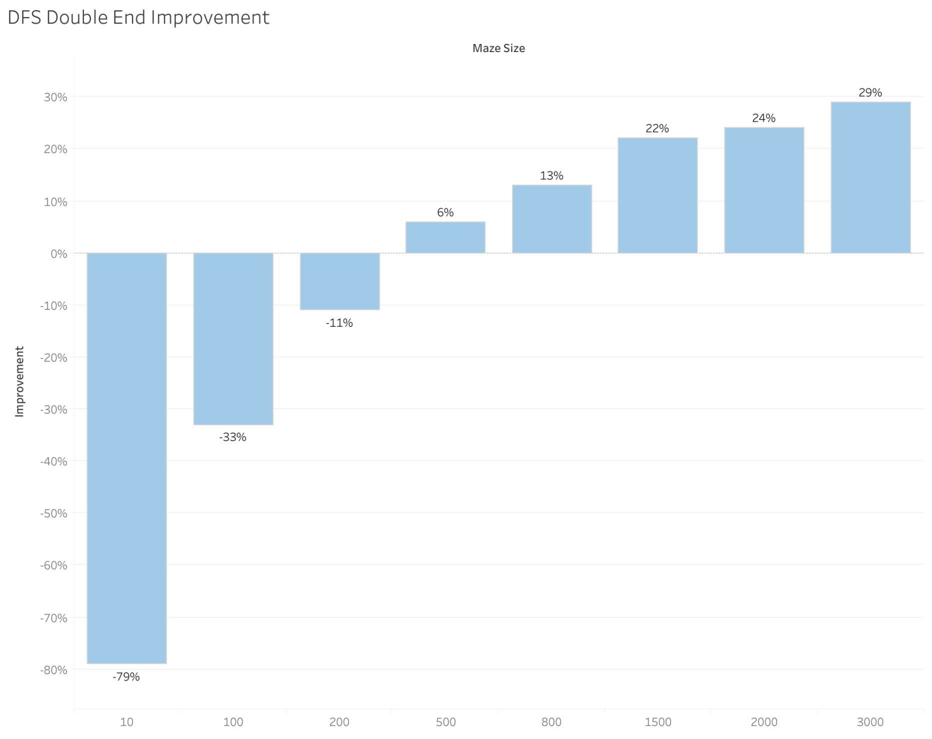


Figure 7: Double-End DFS Improvement

5.3 Analysis

5.3.1 DFS

As the result in Table1 and Figure5&6, we noticed that the total run time of DFS with openMP is always longer than sequential DFS. This is because that DFS can not be parallelized[9]. DFS algorithm is actually a P-complete in thr $NC \neq P$ conjecture, where NC means Nick's Class. The class NC is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors[10]. $NC \neq P$ means that this problem can not be solved in parallel. Therefore parallelizing DFS is impossible, which gives rise to the bad performance of the DFS algorithm implemented with openMP.

5.3.2 Double-end DFS

The speedup achieved by Double-Ended DFS compared to the other methods in section 5.2 can be attributed to its bidirectional search strategy. This approach effectively splits the search space into two smaller parts, which reduces the amount of time needed to find a solution.

Reduced search space: In a bidirectional search, you start exploring the graph from both the source and the target nodes simultaneously. This means that you only have to explore half the search space from each end before the two searches meet. This effectively reduces the overall search space, leading to a faster search process.

Early termination: Double-Ended DFS can stop as soon as the two separate searches meet. This can happen much earlier than in a unidirectional DFS, which has to search the entire path from the source to the target node. Early termination greatly contributes to the speedup observed in Double-Ended DFS.

Exploiting graph structure: In some cases, the structure of the graph can be exploited to speed up the search further. For example, if one end of the search encounters a dead-end or a part of the graph with few connections, the search from the other end can continue, increasing the chances of finding a solution sooner.

Parallelism: Bidirectional search can take advantage of parallelism if multiple processor cores are available. Since each search (from the source and the target) can run independently, they can be executed in parallel, further speeding up the process.

The Speedup5 and Figure7 observed in the results is more pronounced as the maze size increases. This is because larger mazes usually have longer and more complex paths, and the bidirectional nature of Double-Ended DFS becomes more beneficial in these cases. The reduction in search space and the potential for early termination play crucial roles in making Double-Ended DFS a faster algorithm for solving large mazes compared to the other methods.

6 Conclusion

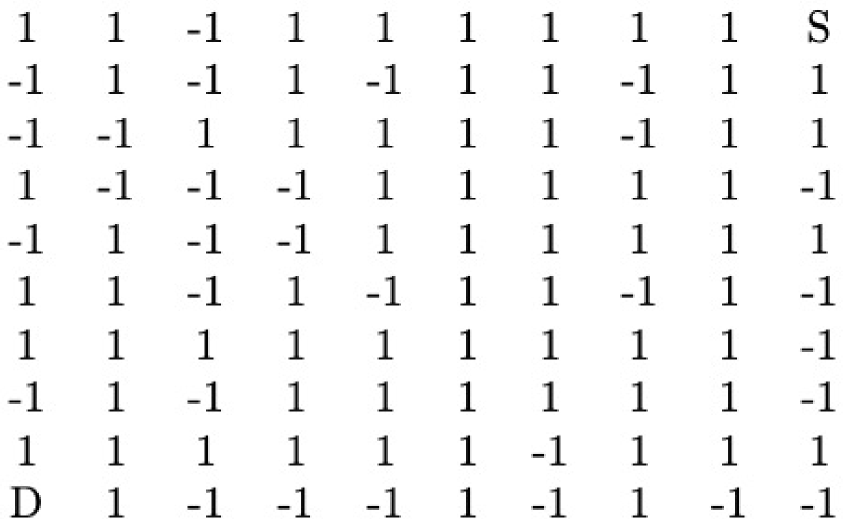
1. When the size of the maze is tiny, the speedup of all the parallel algorithm is not obvious. With the input maze size became larger, the speed up for OpenMP version double-end DFS has significant improvement up to 1.3 when the maze size is 3000*3000. The vanilla DFS with openMP nearly doesn't have any improvement no matter how many cores are used.

2. Double-end DFS is a high efficiency algorithm when dealing with large size maze. However, vanilla DFS is hard to be parallelized because of the $NC \neq P$ conjecture.

7 Future Work

The performance of double ended DFS (OpenMP version) is better than original version. We suppose each end can be optimized in one thread (Two threads total) with OpenMP. We think about how to get better performance which could utilize more cores and make it more parallel. After our research, we discovered an idea and managed to replicate and improve his experiments.

The maze generator for this algorithm is a little bit different. The size of maze is $10000 * 10000$. We set the source of the maze to (0, 9999) and we set the destination of the maze to (9999, 0). We present 1 as the path and -1 as the wall. The example maze is as Figure8.



1	1	-1	1	1	1	1	1	1	S
-1	1	-1	1	-1	1	1	-1	1	1
-1	-1	1	1	1	1	1	-1	1	1
1	-1	-1	-1	1	1	1	1	1	-1
-1	1	-1	-1	1	1	1	1	1	1
1	1	-1	1	-1	1	1	-1	1	-1
1	1	1	1	1	1	1	1	1	-1
-1	1	-1	1	1	1	1	1	1	-1
1	1	1	1	1	1	-1	1	1	1
D	1	-1	-1	-1	1	-1	1	-1	-1

Figure 8: Example Maze

The general idea for this parallel algorithm is using an algorithm based on double ended BFS to simulate Dijkstra, and since the map weights are 1, we used the idea of pipelining for the simulation, and finally using multi-threading and using data parallelism in each stage of the pipeline. We designed the experiment to three types: 2. Single ended algorithm 3. Parallel double ended algorithm.

1. **Sequential double ended algorithm**

   Use regular Dijkstra algorithm to find the shortest path.

2. **Single ended algorithm**

   Specifically, it is equivalent to Dijkstra's algorithm starts by simultaneously putting the starting point S and the ending point D in the priority queue and setting their distances both to 0. In this way, the points out of the queue will form two connected sets, one containing S and one containing D. The two sets keep expanding, and when the two "touch", it must be possible to find the two points located on the shortest path from all the pairs of points in contact along the outer edge of the two sets. Note that the two sets are non-intersecting, and it is forbidden for one side to expand the existing nodes of the other side, so the number of contacted point pairs is $O(n)(n = 10000)$.

   In fact, we only need MAX_VAL + 1 bucket (MAX_VAL indicates the maximum weight) to form a ring. To process each bucket, we first need to remove the nodes from the bucket, update the neighbors, and then place the updated neighbor nodes into some of the MAX_VAL buckets that follow.

3. **Parallel double ended algorithm**

   Based on the sequential algorithm, it is sufficient to parallelize the data of the nodes to be processed in the distance bucket (or pipeline) above. One thing to keep in mind is the synchronization issue, along with the efficiency. Using "pragma omp for" automatically splits the data traversal, I allocate MAX_VAL+1 private bucket for each thread, which reduces the public memory writes and thus the cache misses. after the computation, the private bucket is merged into the public bucket using "prgama omp critical" and "pragma omp single".

The result of each algorithm is as Table2

test-case	case01	case02	case03
result	20369	-1	20371
single-seq	2.59	2.72	2.76
double-seq	3.22	3.79	3.25
double-omp	2.18	1.89	2.35

Table 2: Results of Each Algorithms

References

[1] R. Mall and L. Patnaik, “Parallel maze routing on hypercube computers,” *Computer-Aided Design*, vol. 23, no. 6, pp. 454–459, 1991.

[2] I.-L. Yen, R. Dubash, and F. Bastani, “Strategies for mapping lee’s maze routing algorithm onto parallel architectures,” in *[1993] Proceedings Seventh International Parallel Processing Symposium*, pp. 672–679, 1993.

[3] M. A. Bashuk, “Solving a maze with a quantum computer,” 2003.

[4] N. Kumar and D. Goswami, “Quantum algorithm to solve a maze: Converting the maze problem into a search problem,” 2013.

[5] E. White, “Clustering for improved learning in maze traversal problem,” 2009.

[6] Y. V. Pershin and M. D. Ventra, “Solving mazes with memristors: A massively parallel approach,” *Physical Review E*, vol. 84, oct 2011.

[7] Y. Liu and Y. Xiao, “Parallel solution of maze optimal path based on ant colony algorithm,” in *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pp. 1826–1829, Atlantis Press, 2013/03.

[8] R. Muthuselvi and P. Sridevi, “Parallelization of maze generation and solving in multicore using openmp,” 2016.

[9] Q. F. Stout, “Rethinking maze algorithms: Serial, parallel, power constrained parallel,” 2019.

[10] `https://en.wikipedia.org/wiki/NC_(complexity)`.