EECS587 Parallel Sodoku Solver Final Report

Yilin Jia

December 14, 2024

1 Abstract

I am going to create a parallel solver of the Sodoku game with MPI, and perform a deep analysis of the performance characteristics. The code is available at `https://github.com/tic-top/ParallelSudoku.git` and the experiment data can be found in the appendix.

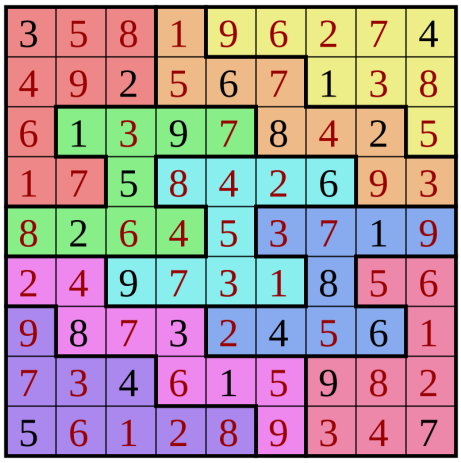

Figure 1: Sodoku sample

2 Introduction

Sudoku is a popular number puzzle that originated from a game called *Number Place*, created by American architect Howard Garns in 1979 and published in *Dell Magazine*. The puzzle was later introduced in Japan in 1984 by the company Nikoli, where it was named *Sudoku*, meaning "single number." A standard Sudoku puzzle consists of a 9x9 grid, with each row, column, and 3x3 subgrid containing the numbers 1 through 9 without repeats. The total number of valid, completed Sudoku grids is approximately 6.67×10^{21}, though not all of these

can be used as puzzles with unique solutions. Various algorithms have been developed to solve Sudoku puzzles efficiently.

1. Basic methods include *backtracking*, a depth-first search that tries all possible values for each cell and backtracks upon encountering contradictions.

2. More advanced techniques employ *constraint propagation*, using strategies like naked and hidden singles, naked pairs, and locked candidates to reduce possible values iteratively.

3. The *Exact Cover with Dancing Links (DLX)* algorithm, developed by Donald Knuth, converts Sudoku into an exact cover problem and solves it with high efficiency using "dancing links."

4. Additionally, stochastic algorithms such as genetic algorithms provide heuristic-based approaches, though they may not always guarantee a unique solution.

Together, these methods allow for efficient solving of even the most complex Sudoku puzzles.

3 Methodology

3.1 Normal DFS

The normal implementation of the Sudoku solver uses a sequential depth-first search (DFS) approach, also known as backtracking. This algorithm starts by visiting each empty cell in the Sudoku grid, attempting to fill in possible digits sequentially. If a conflict arises (i.e., the current number violates Sudoku rules), the algorithm backtracks to the previous step and tries the next possible digit. This process repeats until either a solution is found or all possibilities are exhausted.

The sequential DFS algorithm operates on a single thread and is inherently difficult to parallelize due to its reliance on a shared stack structure for recursion. Each recursive call depends on the results of the previous step, making it unsuitable for concurrent execution. Furthermore, the algorithm's use of a boolean return type to indicate solution validity limits the potential for task decomposition.

3.2 Parallel DFS

To overcome the limitations of the sequential DFS, we implemented a parallelized version of the algorithm. This approach begins with a breadth-first search (BFS) to explore the initial levels of the Sudoku solution tree. By exploring the first few empty cells, the BFS generates a set of partially solved grids, each representing a valid starting point for further exploration. These grids serve as the input for the worker threads.

Each worker thread independently performs a depth-first search from its assigned starting grid, exploring the remaining empty cells to find complete solutions. This task decomposition ensures that the work is evenly distributed among the workers, significantly reducing the overall computation time.

To minimize contention and synchronization overhead, each worker operates on its own copy of the Sudoku grid, avoiding the need for shared data structures. Additionally, a global queue is maintained to collect results, allowing efficient aggregation of all valid solutions.

The hybrid BFS-DFS approach leverages the parallelism of modern multi-core processors, achieving significant speedups compared to the sequential implementation while maintaining the correctness of the algorithm.

4 Technical Detail

This section provides details about the sequential and parallel implementations of the Sudoku solver. The primary goal is to compare the computational efficiency and scalability of the two approaches.

4.1 Sequential

The sequential implementation utilizes a backtracking algorithm to solve Sudoku puzzles. The algorithm attempts to fill empty cells by testing numbers from 1 to 9 and checking their validity based on Sudoku rules (row, column, and 3x3 subgrid constraints). If no valid number is found, the algorithm backtracks to the previous cell and tries the next possible number.

The implementation accepts input as an 81-character string representing the Sudoku puzzle, where '0' denotes an empty cell. The input is converted into a 9x9 integer grid for processing. The backtracking approach guarantees correctness but has an exponential time complexity of $O(9^k)$, where k is the number of empty cells.

4.2 Parallel: 1 Master and (p-1) Servants Arch

To enhance performance, the Sudoku solver is parallelized using the Message Passing Interface (MPI). A master-worker model is adopted, where:

- Master Process: Manages a task queue of partially solved Sudoku boards. It distributes tasks to worker processes and listens for results or failure messages.
- Worker Processes: Receive a Sudoku board, attempt to solve it using a depth-first search (DFS), and return the solution or a failure signal to the master.

Tasks are generated by expanding the first empty cell of the Sudoku board into multiple possibilities. Each possibility creates a new board state, which

is added to the task queue. This approach balances the distribution of the workload among workers while ensuring completeness.

Communication is implemented using `MPI_Send` and `MPI_Recv`, with custom tags for task distribution, solution reporting, and termination. The master process dynamically assigns new tasks to idle workers, ensuring efficient resource utilization. If a solution is found, all processes are terminated immediately to save computational resources.

The performance of sequential and parallel implementations is evaluated using real-time measurements. Execution time is measured using the `std::chrono` library for the sequential version and `MPI_Wtime` for the parallel version. Metrics such as speedup and parallel efficiency are analyzed to assess the scalability of the parallel solver.

4.3 Key Code Snippets

The following snippets illustrate the core logic of the implementations:

Sequential Backtracking Algorithm:

```
bool solveSudoku(int grid[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (grid[i][j] == 0) {
                for (int val = 1; val <= 9; val++) {
                    if (isValid(grid, i, j, val)) {
                        grid[i][j] = val;
                        if (solveSudoku(grid)) return true;
                        grid[i][j] = 0;
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

Master Task Distribution in Parallel Solver:

```
void distributeTasks(queue<vector<int>> &tasks, int p, double *end) {
    int activeWorkers = p;
    while (activeWorkers > 0) {
        MPI_Status status;
        for (int w = 1; w <= p; w++) {
            if (tasks.empty()) {
                MPI_Send(NULL, 0, MPI_INT, w, TAG_NO_MORE_TASK, MPI_COMM_WORLD);
                activeWorkers--;
            } else {
```

```
                vector<int> task = tasks.front(); tasks.pop();
                MPI_Send(&task[0], 81, MPI_INT, w, TAG_SEND_TASK, MPI_COMM_WORLD);
            }
        }
    }
}
```

5 Result

5.1 Benchmark

To evaluate the performance of our algorithms, we implemented both the sequential backtracking algorithm and its parallel version using MPI. We conducted a performance analysis using a subset of Sudoku puzzles. Specifically, we selected 744 samples with varying difficulty levels from a dataset containing 3 million Sudoku puzzles, sourced from Kaggle.

5.2 Dataset Description

The dataset comprises 3 million Sudoku puzzles and their corresponding solutions, with difficulty levels ranging from beginner-friendly to puzzles that challenge even experienced solvers. Most puzzles in the dataset have between 23 and 26 clues, with the minimum and maximum number of clues being 19 and 31, respectively. While it is known that 17 clues are the theoretical minimum for a valid, uniquely solvable Sudoku puzzle, such puzzles are exceptionally rare and are not included in this dataset.

Each row of the dataset provides the puzzle, its solution, the number of clues, and an estimated difficulty rating. The difficulty rating is calculated using an automated solver, based on the average search tree depth over ten attempts. In particular, 43% of the puzzles have a difficulty rating of zero, which means that they can be solved using simple scanning techniques. The highest difficulty rating in the dataset is 8.5.

The puzzles were generated using Blagovest Dachev's Sudoku generator and solver, a reliable tool for generating a wide range of Sudoku challenges.

5.3 Sampled Dataset

From the original dataset, we extracted a smaller subset of 7034 puzzles to ensure coverage at different difficulty levels. The sample data set is available here. This tinier data set was carefully curated to allow for detailed performance evaluation across a spectrum of complexities.

In the following, we provide a visual representation of the difficulty distribution in our sampled dataset, showing the relationship between difficulty ratings and the number of samples.

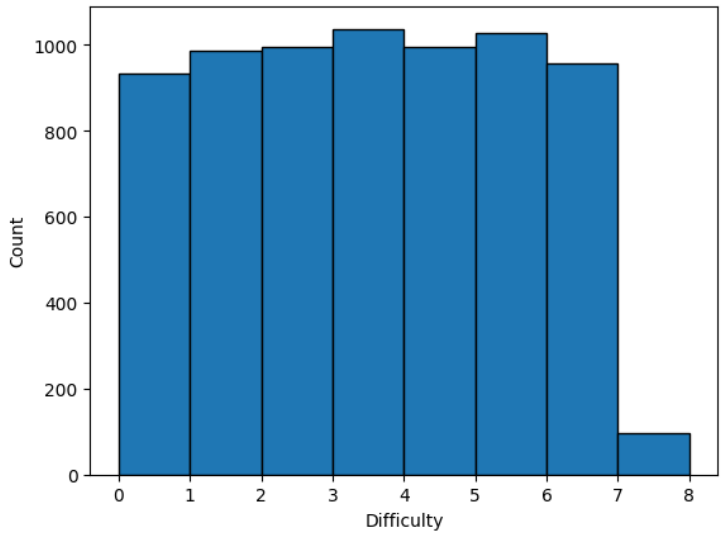

Figure 2: Difficulty distribution vs. number of samples in the tinier dataset.

6 Experimental Analysis

In this experiment, we designed and implemented a distributed Sudoku solver leveraging parallel computing techniques. The aim was to evaluate its performance across different Sudoku difficulty levels, with a specific focus on **runtime**, **speedup**, and **efficiency**. Experiments were conducted using 1, 2, 4, 8, 16, and 32 cores, and the solver's performance was tested on puzzles of varying difficulties. The following sections present the analysis of runtime, speedup, and efficiency based on the experimental results.

6.1 Runtime Analysis

We first evaluate the runtime of the Sudoku solver under different core counts and difficulty levels. Figure 3 shows the runtime for solving Sudoku puzzles with varying core numbers. From the results, we observe the following:

- For **low-difficulty puzzles** (e.g., difficulty levels 1 and 2, with more clues), increasing the core count results in limited performance improvement. These puzzles are relatively simple, and the overhead of parallelization outweighs the benefits.
- For **high-difficulty puzzles**, especially with fewer clues (e.g., difficulty level 7 with only 20 clues), the runtime significantly decreases as the number of cores increases. This indicates that the parallel solver effectively utilizes additional cores for more computationally intensive problems.

Overall, our solver demonstrates its greatest advantage on **high-difficulty puzzles**, particularly for sparse puzzles with fewer clues.

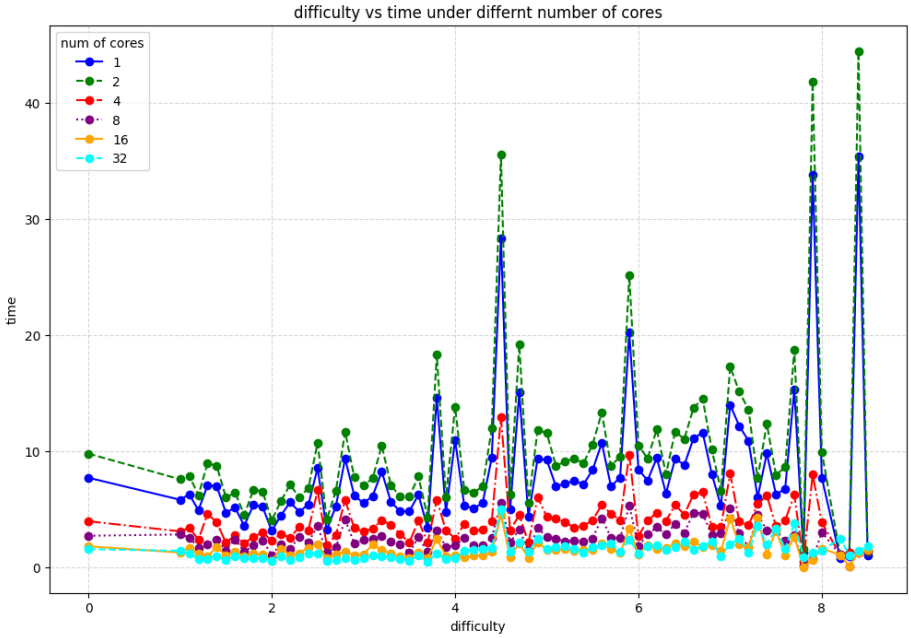

Figure 3: Runtime comparison for different core counts and Sudoku difficulties.

6.2 Speedup Analysis

Figure 4 shows the **speedup** achieved by the distributed Sudoku solver relative to the single-core baseline. Speedup is defined as:

$$\text{Speedup} = \frac{\text{Runtime with 1 core}}{\text{Runtime with } p \text{ cores}}$$

The following trends can be observed:

- For **low-difficulty puzzles**, the speedup is minimal and occasionally sub-optimal due to parallelization overhead. The simplicity of these puzzles means that additional cores provide negligible benefits.
- For **high-difficulty puzzles** (e.g., difficulty level 7), the speedup is significantly higher and scales well with the number of cores. This indicates that the parallelization strategy is effective for these computationally intensive tasks.

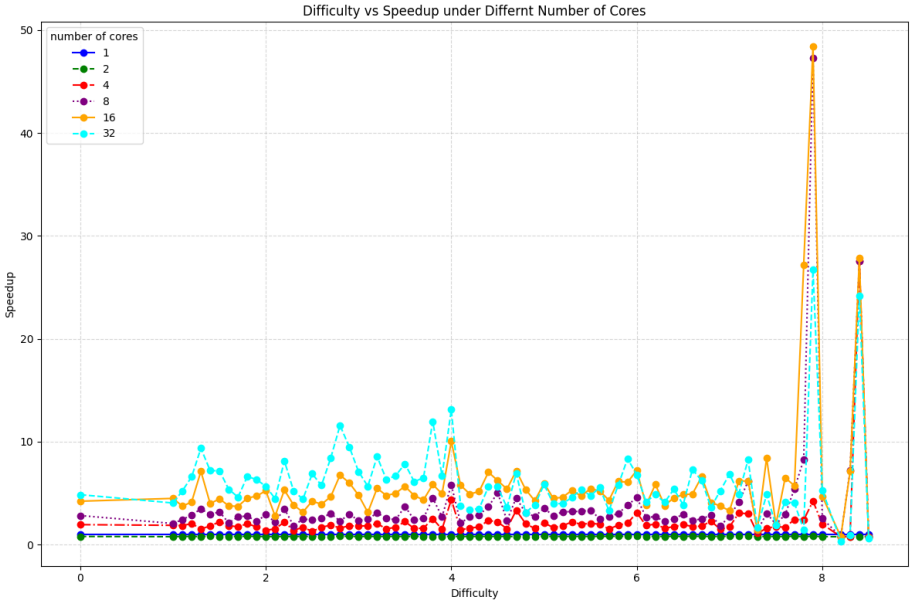

Figure 4: Speedup achieved with different core counts for varying Sudoku difficulties.

6.3 Efficiency Analysis

Efficiency measures how effectively computational resources are utilized and is defined as:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of cores}}$$

Figure 5 presents the efficiency of the solver. We note the following:

- For **low-difficulty puzzles**, efficiency drops as the number of cores increases. The parallel overhead dominates because these puzzles require less computational effort.
- For **high-difficulty puzzles**, efficiency remains high even with 32 cores. The increased computational complexity allows the solver to utilize additional cores effectively, achieving good parallel efficiency.

This demonstrates that our solver is well-suited for **challenging Sudoku puzzles**, where the benefits of parallel computing outweigh the costs of overhead.

6.4 Conclusion and Future Directions

In this experiment, we have successfully demonstrated the performance of a distributed Sudoku solver, evaluating its effectiveness in solving puzzles of varying

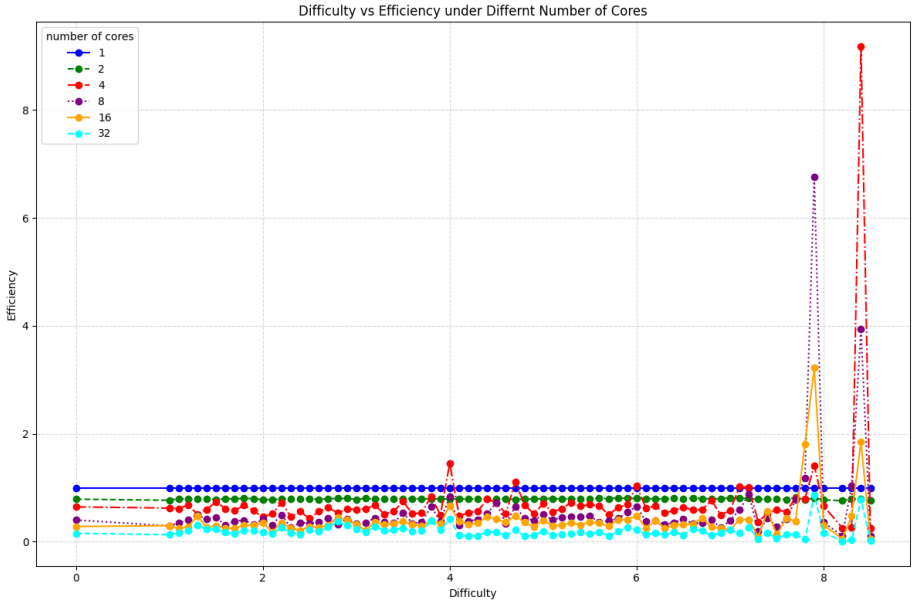

Figure 5: Efficiency comparison with different core counts for varying Sudoku difficulties.

difficulty levels using different core configurations. The results showed that the solver performs well on high-difficulty Sudoku puzzles, particularly those with fewer clues (e.g., difficulty level 7), where parallelization provides significant speedup and efficiency gains.

However, despite the promising results for standard 9x9 Sudoku puzzles, there are several important avenues for future exploration:

1. **Larger Sudoku Grids (e.g., 16x16, 32x32)**: While our current implementation is focused on the standard 9x9 Sudoku grid, the scalability of the solver for larger grids, such as 16x16 or 32x32, remains a crucial challenge. These larger grids involve significantly higher computational complexity and a greater number of possible combinations, which could further push the solver's parallelization capabilities. Exploring the performance of the solver on these larger grids would help assess its potential in handling more complex puzzle types and allow for an in-depth study of the solver's ability to scale efficiently across multiple cores.

2. **Killer Sudoku**: Killer Sudoku is a variant of traditional Sudoku that adds additional constraints based on sum regions (known as "cages"). These additional constraints make Killer Sudoku puzzles much more difficult to solve. The solver could be extended to handle this variant, which would require incorporating new algorithms for constraint propagation and optimization. Investigating the performance of the parallel solver

on Killer Sudoku would provide a valuable comparison to the standard Sudoku solver and could lead to improvements in constraint-based solving techniques.

3. **Other Challenging Variants**: There are numerous other Sudoku variants and related combinatorial problems that could benefit from parallelized solving techniques. Some of these include:

   - **Hyper Sudoku**: An extension of the standard Sudoku where certain subgrids must also satisfy additional constraints.
   - **Samurai Sudoku**: A puzzle composed of several overlapping grids, where the solver must handle multiple interdependent grids simultaneously.
   - **Diabolical or Extreme Difficulty Sudoku**: Puzzles that are specifically designed to be extraordinarily difficult, with sparse clues and complex solution patterns.

   Additionally, we could explore other combinatorial problems with similar characteristics to Sudoku, such as:

   - **KenKen**: A puzzle involving arithmetic operations within a grid, which introduces additional constraints.
   - **Magic Squares**: A mathematical puzzle that could also benefit from parallel approaches to constraint satisfaction.

4. **Optimizing Parallelization Techniques**: Although the solver demonstrated good performance with up to 32 cores, there remains room for improvement in the efficiency of parallelization. Future work could focus on more advanced parallel algorithms, such as dynamic load balancing, more efficient data partitioning, and fine-tuning communication patterns between processors to minimize overhead.

In conclusion, while our distributed Sudoku solver has proven effective for standard Sudoku puzzles, there are significant opportunities to enhance its capabilities and explore more complex problems. By extending the solver to larger grids, tackling new puzzle variants like Killer Sudoku, and optimizing parallelization techniques, we can further advance the field of parallel problem solving and push the boundaries of what is possible in combinatorial problem-solving.

7 Acknowledgement

The bellowing link are the resources for dataset and baseline algorithms.
`https://www.kaggle.com/datasets/rohanrao/sudoku`
`https://en.wikipedia.org/wiki/Sudoku`
`https://github.com/dhhruv/Sudoku-Solver?tab=readme-ov-file`

A Speedup Table

Table 1: Difficulty vs Speedup under different number of cores

Difficulty	Core 1	Core 2	Core 4	Core 8	Core 16	Core 32
4.000000	1.000000	0.792318	4.376729	5.814139	10.036709	13.121658
4.100000	1.000000	0.795373	1.431533	2.101214	5.766232	3.773324
4.200000	1.000000	0.789070	1.611015	2.678694	4.909315	3.391212
4.300000	1.000000	0.796653	1.703850	2.892915	5.165198	3.494357
4.400000	1.000000	0.790549	2.365340	3.672740	7.037051	5.614314
4.500000	1.000000	0.797191	2.183977	5.049600	6.220767	5.610268
4.600000	1.000000	0.795687	1.546861	2.328880	5.398040	3.612288
4.700000	1.000000	0.784590	3.333395	4.545965	7.175495	6.934611
4.800000	1.000000	0.791531	2.056647	3.083098	5.374288	3.124289
4.900000	1.000000	0.790483	1.550651	2.736934	4.254224	3.837450
5.000000	1.000000	0.800550	2.109246	3.521562	5.971482	5.828193
5.100000	1.000000	0.798905	1.650842	2.823052	4.487280	3.969536
5.200000	1.000000	0.792258	1.843175	3.179680	4.600402	3.988570
5.300000	1.000000	0.798113	2.187294	3.208394	5.272580	4.578557
5.400000	1.000000	0.795864	1.976572	3.227278	4.711932	5.370409
5.500000	1.000000	0.793584	2.062817	3.346460	5.436699	4.749012
5.600000	1.000000	0.804268	1.983381	2.517344	5.212746	5.534644
5.700000	1.000000	0.799563	1.509107	2.705542	4.306527	3.299721
5.800000	1.000000	0.810077	1.889827	3.027741	6.150551	5.772787
5.900000	1.000000	0.803421	2.078041	3.808143	6.057254	8.344741
6.000000	1.000000	0.802983	3.098986	4.579531	7.215161	6.803300
6.100000	1.000000	0.798349	1.852441	2.638167	3.834417	4.148603
6.200000	1.000000	0.794134	1.995577	2.711529	5.844370	4.868636
6.300000	1.000000	0.795170	1.591734	2.248716	3.801005	4.103464
6.400000	1.000000	0.801789	1.740030	2.478898	4.545327	5.381865
6.500000	1.000000	0.798191	1.927935	2.961208	4.853679	3.873834
6.600000	1.000000	0.808477	1.768813	2.343647	4.915954	7.311047
6.700000	1.000000	0.800130	1.782787	2.497057	6.604952	6.205288
6.800000	1.000000	0.793286	2.285478	2.869910	4.094444	3.622319
6.900000	1.000000	0.798344	1.493139	1.804671	3.746569	5.223347
7.000000	1.000000	0.810562	1.731791	2.745498	3.292890	6.854834
7.100000	1.000000	0.804425	3.054376	4.114421	6.172353	4.925935
7.200000	1.000000	0.802186	3.005352	6.149509	6.190965	8.249769
7.300000	1.000000	0.787439	1.106831	1.325789	1.414603	1.672384
7.400000	1.000000	0.792361	1.580291	3.049018	8.423869	4.864759
7.500000	1.000000	0.790761	1.766766	1.976489	1.971339	1.888667
7.600000	1.000000	0.784109	1.669237	2.910966	6.492468	4.168836
7.700000	1.000000	0.817704	2.444042	5.432456	5.723713	4.033800
7.800000	1.000000	0.780354	2.390654	8.251613	27.212766	1.421111

difficulty	core1	core2	core4	core8	core16	core32
0.000000	1.000000	0.788661	1.937123	2.821185	4.217062	4.847977
1.000000	1.000000	0.768384	1.871252	2.047587	4.488592	4.041836
1.100000	1.000000	0.797778	1.843394	2.428019	3.729018	5.221579
1.200000	1.000000	0.791618	2.041087	2.866058	4.143003	6.585217
1.300000	1.000000	0.789179	1.532169	3.466990	7.126633	9.360751
1.400000	1.000000	0.800092	1.796034	2.930357	3.971171	7.184305
1.500000	1.000000	0.782183	2.216979	3.172863	4.454552	7.177420
1.600000	1.000000	0.797628	1.839668	2.108634	3.760105	5.377176
1.700000	1.000000	0.795220	1.727799	2.690143	3.713437	4.609977
1.800000	1.000000	0.804606	2.035061	2.787453	4.516997	6.613416
1.900000	1.000000	0.796376	1.716531	2.228946	4.711054	6.326303
2.000000	1.000000	0.780748	1.378931	2.906122	5.230399	5.657634
2.100000	1.000000	0.782858	1.545293	2.196914	2.766198	4.463449
2.200000	1.000000	0.789223	2.175168	3.480941	5.307714	8.139517
2.300000	1.000000	0.796636	1.381459	1.810099	3.836918	5.204818
2.400000	1.000000	0.793128	1.682367	2.490792	3.156494	4.413876
2.500000	1.000000	0.797512	1.286940	2.389777	4.230640	6.879225
2.600000	1.000000	0.786177	1.679108	2.560498	3.913827	5.754645
2.700000	1.000000	0.798218	1.892486	3.039580	4.654915	8.416669
2.800000	1.000000	0.801130	1.611827	2.271915	6.756487	11.579824
2.900000	1.000000	0.800887	1.823856	2.969680	6.005806	9.501628
3.000000	1.000000	0.784483	1.790346	2.328117	4.801155	7.087623
3.100000	1.000000	0.802457	1.832646	2.468755	3.131502	5.669280
3.200000	1.000000	0.786888	2.024046	3.078803	5.475806	8.559679
3.300000	1.000000	0.796739	1.540277	2.550183	4.762933	6.315873
3.400000	1.000000	0.794877	1.675682	2.408887	4.986119	6.700391
3.500000	1.000000	0.793204	2.242971	3.712581	5.669478	7.851992
3.600000	1.000000	0.800341	1.554922	2.395836	4.759963	6.114142
3.700000	1.000000	0.792982	1.599376	2.552369	4.339437	6.490512
3.800000	1.000000	0.797409	2.510271	4.523858	5.837848	11.976761
3.900000	1.000000	0.792650	1.489873	2.694504	4.933026	6.685643
7.900000	1.000000	0.808972	4.228999	47.312106	48.395848	26.712367
8.000000	1.000000	0.778400	1.983171	2.535496	4.700344	5.289360
8.200000	1.000000	0.758590	0.769019	0.755856	0.766210	0.333598
8.300000	1.000000	0.794187	0.798578	7.221429	7.119718	0.941341
8.400000	1.000000	0.796472	27.527216	27.548638	27.852085	24.196856
8.500000	1.000000	0.771592	0.763958	0.763958	0.702404	0.576226

B Efficiency Table

Table 2: Difficulty vs Efficiency under different number of cores

Difficulty	Core 1	Core 2	Core 4	Core 8	Core 16	Core 32
0.000000	1.000000	0.788661	0.645708	0.403026	0.281137	0.156386
1.000000	1.000000	0.768384	0.623751	0.292512	0.299239	0.130382
1.100000	1.000000	0.797778	0.614465	0.346860	0.248601	0.168438
1.200000	1.000000	0.791618	0.680362	0.409437	0.276200	0.212426
1.300000	1.000000	0.789179	0.510723	0.495284	0.475109	0.301960
1.400000	1.000000	0.800092	0.598678	0.418622	0.264745	0.231752
1.500000	1.000000	0.782183	0.738993	0.453266	0.296970	0.231530
1.600000	1.000000	0.797628	0.613223	0.301233	0.250674	0.173457
1.700000	1.000000	0.795220	0.575933	0.384306	0.247562	0.148709
1.800000	1.000000	0.804606	0.678354	0.398208	0.301133	0.213336
1.900000	1.000000	0.796376	0.572177	0.318421	0.314070	0.204074
2.000000	1.000000	0.780748	0.459644	0.415160	0.348693	0.182504
2.100000	1.000000	0.782858	0.515098	0.313845	0.184413	0.143982
2.200000	1.000000	0.789223	0.725056	0.497277	0.353848	0.262565
2.300000	1.000000	0.796636	0.460486	0.258586	0.255795	0.167897
2.400000	1.000000	0.793128	0.560789	0.355827	0.210433	0.142383
2.500000	1.000000	0.797512	0.428980	0.341397	0.282043	0.221910
2.600000	1.000000	0.786177	0.559703	0.365785	0.260922	0.185634
2.700000	1.000000	0.798218	0.630829	0.434226	0.310328	0.271505
2.800000	1.000000	0.801130	0.537276	0.324559	0.450432	0.373543
2.900000	1.000000	0.800887	0.607952	0.424240	0.400387	0.306504
3.000000	1.000000	0.784483	0.596782	0.332588	0.320077	0.228633
3.100000	1.000000	0.802457	0.610882	0.352679	0.208767	0.182880
3.200000	1.000000	0.786888	0.674682	0.439829	0.365054	0.276119
3.300000	1.000000	0.796739	0.513426	0.364312	0.317529	0.203738
3.400000	1.000000	0.794877	0.558561	0.344127	0.332408	0.216142
3.500000	1.000000	0.793204	0.747657	0.530369	0.377965	0.253290
3.600000	1.000000	0.800341	0.518307	0.342262	0.317331	0.197230
3.700000	1.000000	0.792982	0.533125	0.364624	0.289296	0.209371
3.800000	1.000000	0.797409	0.836757	0.646265	0.389190	0.386347
3.900000	1.000000	0.792650	0.496624	0.384929	0.328868	0.215666
4.000000	1.000000	0.792318	1.458910	0.830591	0.669114	0.423279
4.100000	1.000000	0.795373	0.477178	0.300173	0.384415	0.121720
4.200000	1.000000	0.789070	0.537005	0.382671	0.327288	0.109394
4.300000	1.000000	0.796653	0.567950	0.413274	0.344347	0.112721
4.400000	1.000000	0.790549	0.788447	0.524677	0.469137	0.181107
4.500000	1.000000	0.797191	0.727992	0.721371	0.414718	0.180976
4.600000	1.000000	0.795687	0.515620	0.332697	0.359869	0.116525
4.700000	1.000000	0.784590	1.111132	0.649424	0.478366	0.223697
4.800000	1.000000	0.791531	0.685549	0.440443	0.358286	0.100784
4.900000	1.000000	0.790483	0.516884	0.390991	0.283615	0.123789

Table 2 – Continued from previous page

Difficulty	Core 1	Core 2	Core 4	Core 8	Core 16	Core 32
5.000000	1.000000	0.800550	0.703082	0.503080	0.398099	0.188006
5.100000	1.000000	0.798905	0.550281	0.403293	0.299152	0.128050
5.200000	1.000000	0.792258	0.614392	0.454240	0.306693	0.128664
5.300000	1.000000	0.798113	0.729098	0.458342	0.351505	0.147695
5.400000	1.000000	0.795864	0.658857	0.461040	0.314129	0.173239
5.500000	1.000000	0.793584	0.687606	0.478066	0.362447	0.153194
5.600000	1.000000	0.804268	0.661127	0.359621	0.347516	0.178537
5.700000	1.000000	0.799563	0.503036	0.386506	0.287102	0.106443
5.800000	1.000000	0.810077	0.629942	0.432534	0.410037	0.186219
5.900000	1.000000	0.803421	0.692680	0.544020	0.403817	0.269185
6.000000	1.000000	0.802983	1.032995	0.654219	0.481011	0.219461
6.100000	1.000000	0.798349	0.617480	0.376881	0.255628	0.133826
6.200000	1.000000	0.794134	0.665192	0.387361	0.389625	0.157053
6.300000	1.000000	0.795170	0.530578	0.321245	0.253400	0.132370
6.400000	1.000000	0.801789	0.580010	0.354128	0.303022	0.173609
6.500000	1.000000	0.798191	0.642645	0.423030	0.323579	0.124962
6.600000	1.000000	0.808477	0.589604	0.334807	0.327730	0.235840
6.700000	1.000000	0.800130	0.594262	0.356722	0.440330	0.200171
6.800000	1.000000	0.793286	0.761826	0.409987	0.272963	0.116849
6.900000	1.000000	0.798344	0.497713	0.257810	0.249771	0.168495
7.000000	1.000000	0.810562	0.577264	0.392214	0.219526	0.221124
7.100000	1.000000	0.804425	1.018125	0.587774	0.411490	0.158901
7.200000	1.000000	0.802186	1.001784	0.878501	0.412731	0.266122
7.300000	1.000000	0.787439	0.368944	0.189398	0.094307	0.053948
7.400000	1.000000	0.792361	0.526764	0.435574	0.561591	0.156928
7.500000	1.000000	0.790761	0.588922	0.282356	0.131423	0.060925
7.600000	1.000000	0.784109	0.556412	0.415852	0.432831	0.134479
7.700000	1.000000	0.817704	0.814681	0.776065	0.381581	0.130123
7.800000	1.000000	0.780354	0.796885	1.178802	1.814184	0.045842
7.900000	1.000000	0.808972	1.409666	6.758872	3.226390	0.861689
8.000000	1.000000	0.778400	0.661057	0.362214	0.313356	0.170625
8.200000	1.000000	0.758590	0.256340	0.107979	0.051081	0.010761
8.300000	1.000000	0.794187	0.266193	1.031633	0.474648	0.030366
8.400000	1.000000	0.796472	9.175739	3.935520	1.856806	0.780544
8.500000	1.000000	0.771592	0.254653	0.109137	0.046827	0.018588