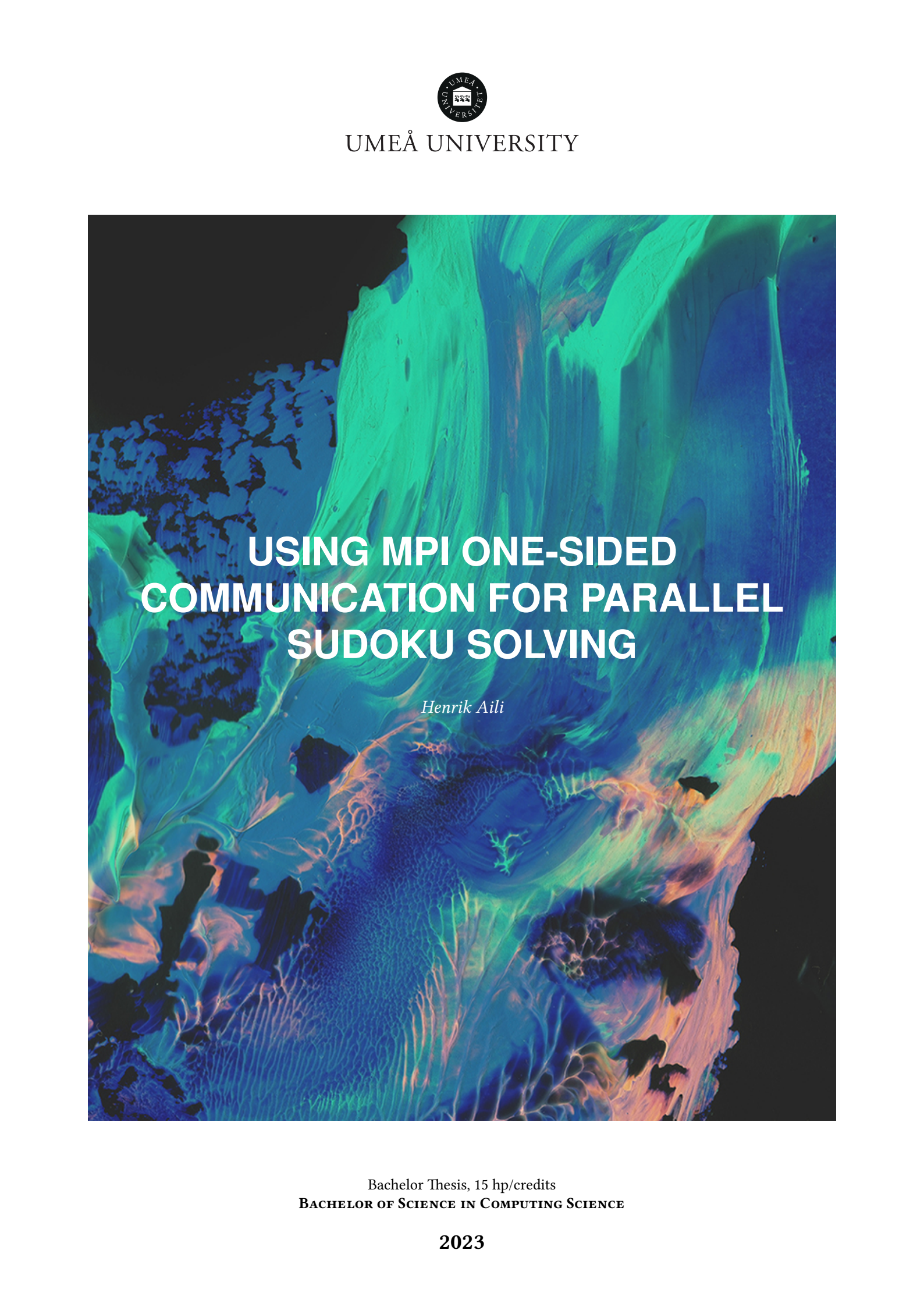

Bachelor Thesis, 15 hp/credits

Bachelor of Science in Computing Science

2023

Abstract

This thesis investigates the scalability of parallel Sudoku solving using Donald Knuth's Dancing Links and Algorithm X with two different MPI communication methods: MPI One-Sided Communication and MPI Send-Receive. The study compares the performance of the two communication approaches and finds that MPI One-Sided Communication exhibits better scalability in terms of speedup and efficiency. The research contributes to the understanding of parallel Sudoku solving and provides insights into the suitability of MPI One-Sided Communication for this task. The results highlight the advantages of using MPI One-Sided Communication over MPI Send-Receive, emphasizing its superior performance in parallel Sudoku solving scenarios. This research lays the foundation for future investigations in distributed computing environments and facilitates advancements in parallel Sudoku solving algorithms.

Acknowledgements

I want to thank my friends and family for their continued support, especially Emma, Ivar, Anton, and a special thanks to Leo for being my eternal lab partner throughout the programme. I also want to thank Jerry for providing the thesis topic, and guidance and support when supervising my thesis.

Contents

1 Introduction

Sudoku, initially named Number Place, is a numerical puzzle that first appeared in an American puzzle magazine in 1979. Number Place later appeared in 1984 in a Japanese magazine, which is where it got the name Sudoku [1]. Sudoku is played on a 9×9 grid, divided into nine 3×3 boxes, according to a set of rules. Each tile of the grid should be filled with an integer from $1 - 9$, where one unique integer may only be on one tile in the same row, column, and box. The Sudoku board typically comes with a set of pre-filled tiles, and the objective is to complete the board by filling in the remaining spaces [1].

1.1 Solving Sudokus Computationally

There are numerous computational methods for solving Sudoku puzzles, including various algorithms. The most commonly used algorithm is a brute force approach, which involves a depth-first search of the Sudoku grid. During this search, the algorithm backtracks whenever it reaches a non-valid solution and continues doing so until it eventually arrives at a valid solution for the puzzle.

Donald Knuth's Algorithm X and Dancing Links may also be used for solving Sudoku as made evident by Harrysson and Laestander [2]. Algorithm X and Dancing Links, are an algorithm and a technique made by Donald E. Knuth [3, p. 1]. Dancing Links, which is the basis of the technique, uses a doubly linked list, where it uncouples and recouples nodes in the list with its neighbours [3, p. 1]. Algorithm X is a non-deterministic recursive algorithm specifically designed to solve exact cover problems, which are combinatorial problems known for their NP-Complete difficulty. When implemented, Algorithm X may employ a technique called Dancing Links [3, p. 3].

1.2 Purpose of Thesis

This thesis explores the opportunities of parallelizing Knuth's Algorithm X using the Message Passing Interface (MPI) library for high performance parallel computing. MPI as the name suggests is a message passing interface which has use in distributed memory environments [4, p. 1]. Two distinct implementations of Algorithm X utilize MPI's One-Sided Communication and Send-Receive to achieve parallelism.

1.3 Research Methodology

This study employs an empirical research methodology to compare the scalability of two Sudoku solver implementations: one utilizing MPI One-Sided Communication and the other utilizing MPI Send-Receive. The effectiveness of each implementation in solving Sudoku puzzles is evaluated through experimentation and result evaluation.

1.4 Research Question

The research question in this thesis is: *Does the use of MPI One-Sided Communication result in better performance for implementing Algorithm X than using MPI Send-Recieve?*

1.5 Outline of the Thesis

The thesis is structured as follows: Section 2 provides a comprehensive review of related works to the thesis, while Section 3 covers the necessary background information on the topic and its application. In Section 4, the author presents their contributions to the research. Section 5 describes the experimental setup used to obtain the results presented in the thesis, while Section 6 presents and analyses the results obtained from the experiments conducted during the research. Section 7 provides a detailed discussion of the results obtained, including their implications and the limitations of the thesis. Finally, Section 8 concludes the thesis with a summary of the main findings and their implications along with potential directions for future research.

2 Related Works

Harrysson and Lestander proposed a solution for solving Sudoku puzzles using Algorithm X, which involves reducing Sudoku to an exact cover problem [2]. They demonstrated the effectiveness of Algorithm X as an efficient method for solving Sudoku puzzles. By transforming Sudoku into an exact cover problem, they provided valuable insights into the computational complexity of the approach. Their work serves as a foundational contribution to the application of Algorithm X in Sudoku puzzle solving and has inspired the approach taken in this thesis.

Foroutan Rad provides an array-based solution to the Dancing Links technique for solving Sudoku puzzles [5]. This approach addresses the challenge of parallelizing Dancing Links and Algorithm X due to the difficulty of sharing pointers between threads. By utilizing an array-based representation, the solution enables easier parallelization and coordination of multiple threads or processes working on the Sudoku puzzle solving task. This work offers valuable insights into enhancing the parallel efficiency and scalability of Sudoku-solving algorithms and serves as a reference for future research and developments in parallel Sudoku solving techniques.

Viksten and Mattson analyse the differences in algorithms for solving Sudoku puzzles in their study [6]. They compare three algorithms: Brute-Force, Simulated Annealing, and Dancing Links with Algorithm X. The research findings indicate that Dancing Links with Algorithm X exhibits the best performance among the evaluated algorithms for solving Sudoku puzzles [6]. This study offers current insights into the strengths and effectiveness of various algorithmic approaches in the context of Sudoku puzzle solving.

3 Background

The background chapter aims to provide the necessary context for understanding the application of Algorithm X and Dancing Links in solving Sudoku puzzles. The chapter begins by introducing the core concepts of Algorithm X and Dancing Links, and their use in solving Sudoku puzzles. Additionally, it briefly explains how parallelization techniques, such as MPI One-Sided Communication and MPI Send-Receive, can be used to accelerate the solution process. While these techniques are not the primary focus of the chapter, they are introduced to provide some context for how parallelization can be used to improve the performance of Algorithm X and Dancing Links in solving Sudoku puzzles. A separate chapter focuses on explaining how these MPI techniques are utilized to accelerate the solving of puzzles using Algorithm X and Dancing Links.

3.1 Algorithm X and Dancing Links

Dancing Links is a technique based on a discovery that Knuth made on doubly linked lists [3, p. 1]. Knuth realized that links in a doubly linked can be reattached after they have been removed. Considering the two operations,

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x] \tag{3.1}$$

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x \tag{3.2}$$

(3.1) which removes x from the list, and (3.2) which put x back into the list [3, p. 1]. It is those two operations that is the heart of the Dancing Links technique.

Definition 3.1.1 (Exact Cover) *Given a family $\{S_j\}$ of subsets of a set $\{u_i, i = 1, 2, ..., t\}$, there exists a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\cup T_h = \cup S_j = \{u_i, i = 1, 2, ..., t\}$ [7, p. 95].*

Algorithm X, developed by Knuth, is a non-deterministic recursive algorithm designed to solve the exact cover problem [3, p. 3]. The exact cover problem, a combinatorial conundrum, falls under the category of NP-Complete difficulties [7, p. 95]. Algorithm X is a non-deterministic recursive algorithm created by Knuth to solve an exact cover problem [3, p. 3]. The exact cover problem is a combinatorics problem that is of NP-Complete difficulty [7, p. 95]. The definition of exact cover is seen in Definition 3.1.1. The objective of Algorithm X is to find the subfamily T_h. Algorithm X, as seen in Algorithm 1, essentially works by having a matrix A consisting of 0s and 1s that represents the exact cover problem to solve. It recursively forms subalgorithms in a tree-like manner and backtracks after failing to find a solution [3, pp. 3-4].

Algorithm 1 Algorithm X

```
1:  function X(A)                          ▷ A is a matrix of the exact cover problem
2:      if A is empty then
3:          return success
4:      else
5:          choose a column c                                   ▷ deterministically
6:          choose a row r such that A[r, c] = 1                ▷ non-deterministically
7:          include the row r in the partial solution
8:          for each j such that A[r, c] = 1 do
9:              delete column j from matrix A
10:             for each i such that A[r, c] = 1 do
11:                 delete row i from matrix A
12:             end for
13:         end for
14:     end if
15:     return X(A)          ▷ Call Algorithm X recursively on the newly reduced matrix
16: end function
```

3.1.1 Implementing Algorithm X with Dancing Links

To implement Algorithm X, a combination of Algorithm X and Dancing Links may be used, which Knuth calls Algorithm DLX [3, p. 5, 8]. In Algorithm DLX, the matrix is represented by data objects x that corresponds to a 1 in the matrix. Each object x has four links that each point in one of the cardinal directions and a link to the list header [3, p. 5]. The data objects are linked row- and column-wise [3, p. 5]. The list headers are special because in addition to the links they have a size that corresponds to the number of rows in the current column and a name used for results [3, p. 5]. There is also a "root" object which is the master of every other active list header and links the headers together into a circular list[3, p. 5].

The algorithm when implemented consists of three main functions: 1. Search seen in Algorithm 2, which is the main recursive search function, 2. Cover seen in Algorithm 5 which essentially covers a column in the matrix representation by unlinking the column similar to the operation in (3.1), 3. Uncover seen in Algorithm 6, which uncovers a column in the representation by linking the column back together using the operation seen in (3.2) [3, pp. 6-7].

Algorithm 2 Search algorithm used in DLX

```
1: function SEARCH(depth k, root header h, stack S)
2:     if R[h] = h then
3:         print solution and return
4:     else
5:         choose a column object c which has the least amount of rows
6:         Cover column c                                  ▷ see Algorithm 5 in Appendix A
7:         for each r ← D[c], D[D[c]], ...,while r ≠ c do
8:             stack_push(S,r)
9:             for each j ← R[r], R[R[r]], ...,while j ≠ r do
10:                cover column j
11:            end for
12:            SEARCH(k + 1, h, S)
13:            r ←stack_pop(S) and c ← C[r]              ▷ C[r] is the column header of r
14:            for each j ← L[r], L[L[r]], ...,while j ≠ r do
15:                uncover column j                        ▷ see Algorithm 6 in Appendix A
16:            end for
17:        end for
18:        uncover column c and return
19:    end if
20: end function
```

3.2 Solving Sudoku Puzzles with Algorithm X

To solve a Sudoku puzzle using Algorithm X, the puzzle is reduced to an Exact Cover problem. This transformation enables the application of Algorithm X for efficient Sudoku puzzle solving. The rules of Sudoku, as described in Section 1, define the constraints for the reduction [2, p. 13]. These constraints include:

- Cell: Each cell can only contain one integer between 1-9.
- Row: Each row can only contain nine unique integers in the range of 1-9.
- Column: Each column can only contain nine unique integers in the range of 1-9.
- Box: Each box can only contain nine unique integers in the range of 1-9 [2, p. 13].

Considering the constraints, the Sudoku puzzle can be represented as a binary matrix of size 729×324 [2, p.17]. This matrix encapsulates the relationships between the Sudoku cells, rows, columns, and boxes, forming the foundation for applying Algorithm X to solve the puzzle efficiently.

3.3 Message Passing Interface

As briefly mentioned earlier in Section 1.2, MPI is a standardized communication protocol used for message passing between parallel processes. It enables the exchange of data through the sending and receiving of messages with the use of an Application Programming Interface (API) [8, p. 22]. MPI is extensively used in high-performance computing for parallel programs that utilize distributed memory machines [8, p. 22], where multiple computing nodes

collaborate within a cluster. The standardized nature of MPI offers the advantage of predefined operations for communication and computation, making it a favored choice in the field of high-performance computing [8, p. 22].

In MPI, communicators are used to define groups of processes that can communicate with each other. Communicators serve as the entities that enable processes to interact and coordinate their operations. [8, pp. 23-24]. The default communicator MPI_COMM_WORLD includes all the processes in a parallel program [8, p. 24]. In MPI, there are several basic primitives, with ”send” and ”receive” being the two most commonly used in this thesis. The ”send” primitive in MPI allows for the transmission of an array of data to a specified destination process. Conversely, the ”receive” primitive enables a process to accept data from a specified source process and store it in an array [8, p. 26]. A small example MPI program in C using send and receive is seen in Listing 3.1

Listing 3.1: Example MPI program in C using the send and recieve primitives

```
1
2  #include <stdio.h>
3  #include <mpi.h>
4
5  int main(int argc, char* argv[]) {
6      //number of processors and process rank
7      int p, my_rank, sum;
8      int my_arr[10];
9
10     //Initialise MPI
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &p)
13     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
14
15     if (my_rank == 0) {
16         //Calculate the partial sums of the 10 first natural numbers
17         for (int i = 0; i < 10; i++) {
18             my_arr[i] = (i*(i+1))/2;
19         }
20         //Send the array of partial sums to rank 1
21         MPI_Send(&my_arr, 10, MPI_INTEGER, 1, 0, MPI_COMM_WORLD);
22     }
23     else {
24         //Recieve an array with 10 integers from rank 0
25         MPI_Recv(&my_arr, 10, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, NULL);
26     }
27
28     //Waits for all processes to reach this code
29     MPI_Barrier(MPI_COMM_WORLD);
30
31     sum = 0;
32     for (int i = 0; i < 10; i++) {
33             sum += my_arr[i]
34     }
35     fprintf(stderr "Sum of 10 first partial sums: %d \n my rank = %d\n", sum
           , my_rank);
36
37
38     MPI_Barrier(MPI_COMM_WORLD);
39
40     MPI_Finalize();
41
42     return 0;
43 }
```

3.3.1 One-Sided Communication

MPI supports one-sided communication through Remote Memory Access (RMA) operations [9, p. 4386]. This allows processes to directly access and modify the memory of other processes without explicit send and receive operations [9, p. 4386]. MPI RMA utilizes memory windows, representing contiguous regions of memory that are accessible to a specific set of processes within a communicator [9, p. 4387]. The creation of windows is a collective operation involving all processes within the input communicator, achieved through the MPI_Win_create function call [9, p. 4387].

The MPI RMA protocol includes three main operations for communication: MPI_Put, MPI_Get, and MPI_Accumulate [9, p. 4387]. However, in this thesis, only the MPI_Put and MPI_Get operations are utilized. MPI_Put puts data into the memory window of the target process, allowing it to be accessed by the target process [9, p. 4387]. Similarly, MPI_Get gets data from the memory window of the target process and stores it into the memory window of the calling process [9, p. 4387]. The communication operations occur within a synchronization epoch, and they are completed at the end of the epoch [9, p. 4387].

3.4 Scalability

Scalability in software refers to the ability of a parallel program to efficiently utilize additional resources as the problem size or the number of processors increases. It is based upon speedup, which quantifies the performance improvement achieved by scaling up the computational resources [10]. Speedup, is defined as:

$$S = \frac{T_s}{T_p},$$

where T_s is the execution time of the sequential program and T_p is the execution amount of the parallel program [11, p. 125]. Efficiency, is defined as:

$$E = \frac{S}{p},$$

where S is the speedup of the parallel program, and p is the number of processes/threads [11, pp. 125-126].

There are two types of scaling in parallel programming: strong scaling and weak scaling [11, p. 127]. A strong scalable program maintains constant efficiency without increasing the problem size, while a weak scalable program maintains constant efficiency as the problem size increases with the number of processes [11, p. 127].

4 Parallelizing Algorithm DLX

This section consists of the main contributions and intricacies found when parallelizing Algorithm DLX using MPI. It aims to provide the details needed for the reader to understand how the parallel version works and be able to implement it themselves. The section begins with the core of the parallel implementation and then continues on with the intricacies of using MPI One-Sided Communication and MPI Send-Recieve.

The core of the parallel version of Algorithm DLX is the serial algorithm as seen in Algorithm 2, since the two parallel implementations relies on the serial implementation of the algorithm. Using both MPI One-Sided Communication and Send-Recieve, it was possible to reduce the Algorithm to a producer-consumer problem for both implementations which each have their own intricacies. The parallelization of Algorithm DLX introduces the concept of granularity, which pertains to the number of Sudoku puzzles assigned to each unit of work.

4.1 MPI One-Sided Communication

In Section 3.3.2, it is discussed that MPI One-Sided Communication rely on shared memory windows. Each MPI process has its own window, allowing all processes within the communicator to access it. By utilizing these memory windows, a specific process (in this case, root/0) can extract and store Sudoku puzzles from the provided file. Subsequently, other processes can retrieve one puzzle at a time to work on, utilizing the serial Algorithm DLX.

However, to ensure work is not duplicated, synchronization among the processes is required. This is achieved by utilizing an integer stored in a second memory window on the root process. The integer serves as a counter that each process increments whenever it acquires a new Sudoku puzzle from the root process. The One-Sided Communication procedure is depicted in Algorithm 3.

4.2 MPI Send-Recieve

In Section 3.3.1, it is discussed that MPI Send-Recieve relies on message passing. Each MPI process within the communicator can send messages to each other. Through the use of message passing, the processes establish a coordinator-worker relationship, with process 0 acting as the coordinator and assigning new Sudoku puzzles to the worker processes upon receiving a request.

When a worker finishes its work, it sends a message to the coordinator process with its process rank and then waits to receive new work from the coordinator. When the coordinator receives a work request, it sends the next Sudoku puzzle in order and increments its index counter until it reaches the number of Sudoku puzzles to solve. Finally, when the total number of Sudoku puzzles is reached, the coordinator sends a termination message to the workers, signalling the completion of all tasks. The Send-Recieve procedure is depicted in Algorithm 4.

Algorithm 3 MPI One-Sided Communication Algorithm DLX

```
1:  function SUDOKU_INDEX(mpi_win W)
2:      lock_window(W)
3:      index ← get(W)
4:      if index ≥ num_sudoku then
5:          unlock_window(W)
6:          return -1
7:      end if
8:      put(index+1, W)
9:      unlock_window(W)
10:     return index
11: end function
12:
13: function GET_SUDOKU(mpi_win W, index I)
14:     lock_window(W)
15:     sudoku ← get(W, I)
16:     unlock_window(W)
17:     return index
18: end function
19:
20: function OSC_DLX(mpi_win SW, mpi_win IW, dlink dl)
21:     I ← 0
22:     while (I ← SUDOKU_INDEX(IW)) ≠ −1 do
23:         sudoku ← GET_SUDOKU(SW, I)
24:         REMOVE_CLUES(dl, sudoku) ▷ Removes the pre-filled squares from the exact cover
    problem
25:         SEARCH(dl)
26:         RESET_DLINK(dl)
27:     end while
28: end function
```

Algorithm 4 MPI Send-Recieve Algorithm DLX

```
1: function ROOT_LOOP(sudokus S, total_sudoku T, num_procs N)
2:     index ← 0
3:     while index < T do
4:         worker ← recv(any)
5:         send(S[index], worker)
6:         index++
7:     end while
8:     for i in N do
9:         send(−1,worker)
10:    end for
11: end function
12:
13: function WORKER_LOOP(dlink dl, my_rank n)
14:    sudoku S
15:    while true do
16:        send(n,0)
17:        S ← recv(0)
18:        if S = −1 then
19:            return
20:        end if
21:        REMOVE_CLUES(dl, sudoku) ▷ Removes the pre-filled squares from the exact cover problem
22:        SEARCH(dl)
23:        RESET_DLINK(dl)
24:    end while
25: end function
```

5 Experimental Setup

This chapter covers the validation of the research question through a series of experiments. It discusses the experimental setup employed and provides details about the test data used for these experiments. It begins by covering the experiments performed, along with the test data used, finishing with the system used for performing the experiments.

The experiment focuses on evaluating the scalability of MPI One-Sided Communication in contrast to MPI Send-Receive. The experiment involves running parallel implementations of the two approaches with a fixed file and an increasing number of Sudokus while varying the core count along with varying granularity. For each core count and grain size, the experiment is performed 10 times, and the runtime of each run is recorded. Finally, the average runtime is calculated for each core count, enabling the determination of the speedup and efficiency of the two implementations.

For the aforementioned experiment, the test data comprises text files containing numerous Sudoku puzzles gathered from various sources on the web. The Sudoku puzzle files adhere to the formatting depicted in Listing 5.1.

Listing 5.1: An example of a Sudoku puzzle file with 1000 puzzles

```
1000
.5..83.17...1..4..3.4..56.8....3...9.9.8245....6....7...9
....5...729..861.36.72.4
2.6.3......1.65.7..471.8.5.5......29..8.194.6...42...1...
.428..6.93....57.....13.
..45.21781...9..3....8....46..45.....7.9...128.12.35..4..
.....935..6.8.7.9.3..62.
```

The system used for performing the experiment is a login server available for students and employees at the Institution of Computing Science, Umeå University used for programming and other use cases. The server used has the following specifications:

- **Hardware**
  - Processor: AMD EPYC 7702P 64 cores/128 threads @2.0-3.35GHz
  - Memory: 128 GB
  - Disk: 2x500 GB SSD
- **Software**
  - Debian GNU/Linux 11 (bullseye) x86_64
  - openmpi(3.1.3-11)
  - gcc (4:10.2.1-1)

6 Results

The purpose of this chapter is to present the results obtained from the experiment. As a reminder, the research question guiding this study is: "*Does the utilization of MPI One-Sided Communication result in better scalability for implementing Algorithm X compared to using MPI Send-Receive?*"

Figure 1 illustrates the speedup of the MPI One-Sided Communication implementation with different grain sizes, ranging from 1 to 64 puzzles per unit of work. The graph shows a generally linear speedup trend, with a slight decrease in speedup as the number of cores increases. Notably, when the grain size is 1, there is a more significant drop-off in speedup from around 50 cores onwards.

Figure 2 illustrates the speedup of the MPI Send-Recieve implementation with different grain sizes, ranging from 1 to 64 puzzles per unit of work. The graph shows a generally linear speedup trend, with a slight decrease in speedup as the core count increases. Notably, when the grain size is 1, there is a more significant drop-off in speedup from around 34 cores onwards.

Figure 3 illustrates the speedup of the optimal grain sizes for MPI One-Sided Communication and MPI Send-Recieve. The graph demonstrates that the performance of the two implementations is comparable, with Send-Recieve exhibiting slightly lower performance than One-Sided Communication.

Figure 4 illustrates the efficiency of the optimal grain sizes for MPI One-Sided Communication and MPI Send-Recieve. The graph shows a relatively constant efficiency until 30 cores, after which it gradually decreases.

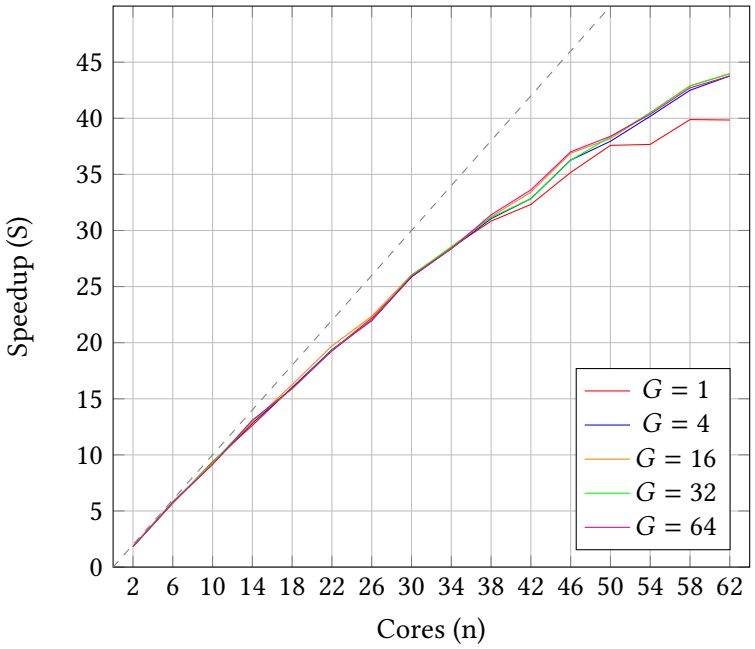

Figure 1: Speedup of varying grain-size using MPI One-Sided Communication

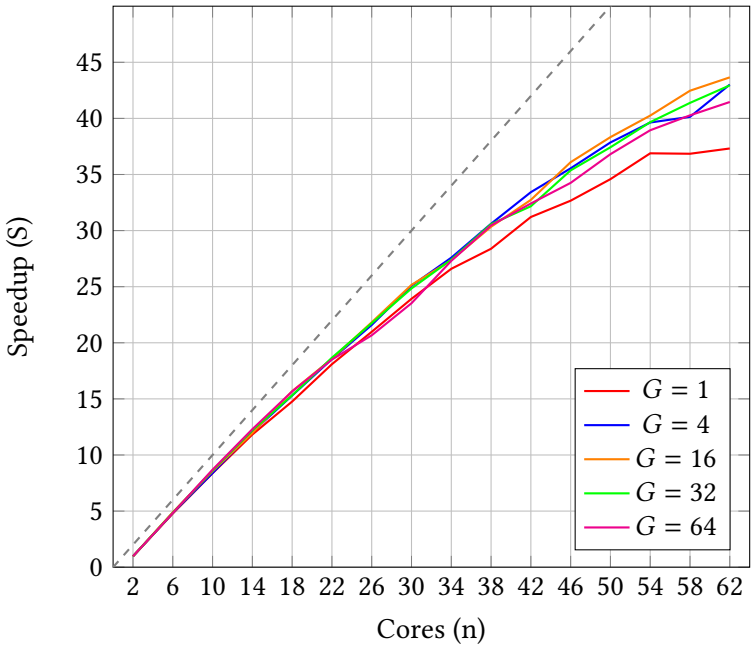

Figure 2: Speedup of varying grain-size using MPI Send-Receive

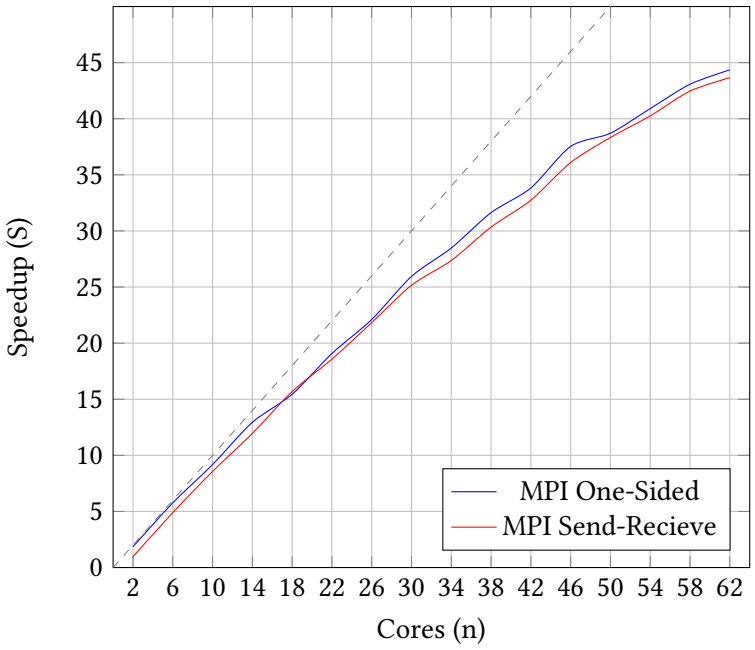

Figure 3: Speedup of the optimal grain sizes for the two implementations

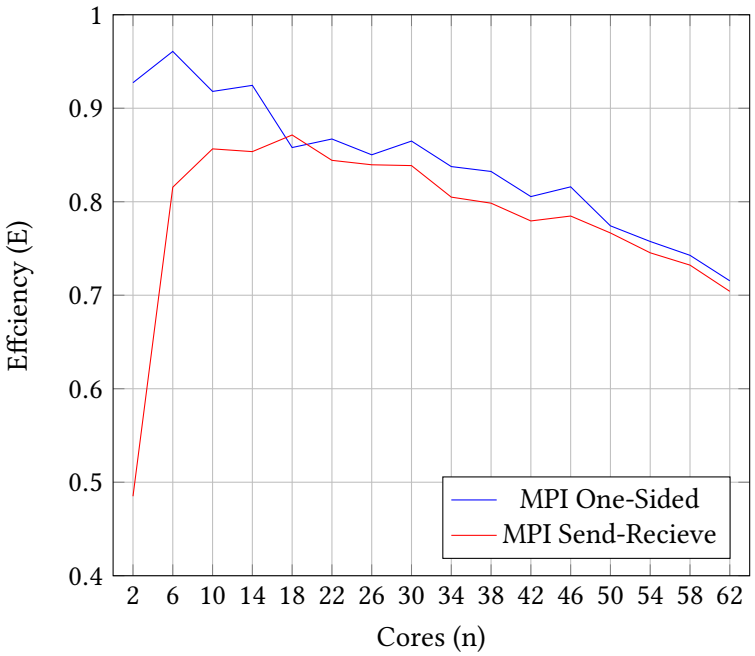

Figure 4: Efficiency of the optimal grain sizes for the two implementations

7 Discussion

One-Sided Communication. This outcome aligns with expectations, as the Send-Receive implementation relies on a root process responsible for sending Sudoku puzzles to the other working cores. This means that the Send-Receive implementation essentially has one less core working on solving the puzzles. In hindsight, the performance difference between the two implementations with optimal grain size is neck and neck, but ultimately Send-Receive exhibits slightly worse performance. This suggests that the extra core that One-Sided Communication has over Send-Receive due to Send-Receive needing a coordinator process outweighs the benefits of the Send-Receive implementation. The obtained results imply that MPI One-Sided Communication offers superior scalability compared to Send-Receive, indicating that the former is better suited for handling increased parallelism.

7.1 Limitations

From my perspective, I perceive several limitations in the conducted study. Firstly, the experiment only focuses on a single file of sudoku puzzles, which might not capture the full range of scenarios or variations in puzzle complexity. Additionally, the study does not explore the impact of the number of clues in the puzzles. These limitations suggest that further experimentation with diverse puzzle datasets and exploring different levels of puzzle complexity would provide a more comprehensive understanding of the scalability of the parallel implementations.

8 Conclusion

In conclusion, this thesis aims to explore the impact of MPI One-Sided Communication on the scalability of a parallel Algorithm X in contrast to MPI Send-Receive. Through a series of experiments and comprehensive analysis, the findings consistently demonstrate the superior scalability of MPI One-Sided Communication over MPI Send-Receive, both in terms of efficiency and performance. The experimental results reveal that MPI One-Sided Communication exhibits enhanced scalability, enabling more effective utilization of computational resources and improved parallel algorithm execution. These outcomes highlight the potential of MPI One-Sided Communication as a valuable approach for achieving better scalability in parallel computing applications.

8.1 Future Work

For future research, it is recommended to explore the impact of the number of clues in the puzzles on the performance of the parallel implementations. Investigating whether using non-blocking MPI Send-Receive can mitigate the performance difference and potentially improve the efficiency of the Send-Receive implementation would be valuable. Additionally, it would be beneficial to investigate the suitability of MPI Send-Receive and MPI One-Sided Communication for multi-node environments. This would provide valuable insights into their performance in distributed computing scenarios and help determine their effectiveness and scalability across multiple nodes. Such investigations would contribute to a deeper understanding of the behavior and optimization of these communication mechanisms in parallel algorithms, paving the way for improved parallelization strategies in the future.

Bibliography

[1] R. Wilson, 2022. [Online]. Available: https://www.britannica.com/topic/sudoku (visited on 04/14/2023).

[2] M. Harrysson and H. Laestander, *Solving sudoku efficiently with dancing links*, 2014.

[3] D. E. Knuth, "Dancing links," *arXiv preprint cs/0011047*, 2000.

[4] D. W. Walker and J. J. Dongarra, "Mpi: A standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[5] A. Foroutan Rad, *Efficient array for solving sudoku problem*, 2018. [Online]. Available: https://umu.diva-portal.org/smash/record.jsf?pid=diva2%3A1278842&dswid=1031.

[6] H. Viksten and V. MATTSSON, *Performance and scalability of sudoku solvers*, 2013.

[7] R. M. Karp, *Reducibility among combinatorial problems, complexity of computer computations (re miller and jw thatcher, editors)*, 1972.

[8] F. Nielsen and F. Nielsen, "Introduction to mpi: The message passing interface," *Introduction to HPC with MPI for Data Science*, pp. 21–62, 2016.

[9] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the mpi 3.0 one-sided communication interface," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 17, pp. 4385–4404, 2016.

[10] X. Li. "Scalability: Strong and weak scaling." (2018), [Online]. Available: https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/ (visited on 06/02/2023).

[11] P. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011, ISBN: 9780123742605 0123742609.

A Extra Algorithms

Algorithm 5 Cover function used in Search

```
1: function Cover(column c)
2:     Set L[R[c]] ← L[c] and R[L[c]] ← R[c]
3:     for each i ← D[c], D[D[c]], ...,while i ≠ c do
4:         for each j ← R[j], R[R[j]], ...,while j ≠ i do
5:             set U[D[j]] ← U[c] and D[U[j]] ← D[J]
6:             set S[C[J]] ← S[C[j]] − 1          ▷ S is the number of rows in the column
7:         end for
8:     end for
9: end function
```

Algorithm 6 Uncover function used in Search

```
1: function Uncover(column c)
2:     for each i ← D[c], D[D[c]], ...,while i ≠ c do
3:         for each j ← R[j], R[R[j]], ...,while j ≠ i do
4:             set S[C[J]] ← S[C[j]] + 1
5:             set U[D[j]] ← j and D[U[j]] ← j
6:         end for
7:     end for
8:     Set L[R[c]] ← c and R[L[c]] ← c
9: end function
```

UMEÅ UNIVERSITY