Programming Technique and Problem Complexity - Impacts on Performance

Jason Graalum

March 16, 2019

CS 431/531 Introduction to Performance Measurement, Modeling and Analysis

Computer Science Department

Portland State University

Winter 2019

1 Introduction

For my CS531 Course Project, I explored the impact of various programming techniques on the performance of the solution to a simple problem. The problem I choose was finding the solution to the well-known Sudoku puzzle. This problem is interesting as both the complexity and size can be adjusted by either adding or removing given values or by increasing the size of the puzzle. For my project, I wrote several different programs which utilized a set of programming techniques including serial, multi-threaded, and multi-processor. This report includes overviews of the programs, descriptions of methods for testing the performance and the conclusions drawn from the tests. I begin with a summary of the testing environment including software and hardware used in the testing.

2 Test Environment

2.1 Hardware Platform

One key for measuring performance of a program is the ability to isolate the testing platform. To do this, I decided to use a simple cluster of Raspberry Pi single board computers. This gave me full control of any other programs running on the systems. The cluster included 6 systems CPI1 - CPI6.

	CPI 1	CPI2	CPI3	CPI4	CPI5	CPI6
Model Name	Raspberry Pi 3 B+ Rev 1.3	Raspberry Pi 3 B+ Rev 1.3	Raspberry Pi 3 B+ Rev 1.3	Raspberry Pi 3 B+ Rev 1.3	Raspberry Pi 3 B+ Rev 1.3	Raspberry Pi 3 B+ Rev 1.3
Hardware	BCM2835	BCM2835	BCM2835	BCM2835	BCM2835	BCM2835
Revision	a020d3	a020d3	a020d3	a02082	a020d3	a020d3
Architecture	armv71	armv71	armv71	armv71	armv71	armv71
CPUs	4	4	4	4	4	4
Thread/Core	1	1	1	1	1	1
Cores/Socket	4	4	4	4	4	4
Sockets	1	1	1	1	1	1
CPU Model	ARMv7 rev 4	ARMv7 rev 4	ARMv7 rev 4	ARMv7 rev 4	ARMv7 rev 4	ARMv7 rev 4
CPU Max MHz	1400	1400	1400	1400	1400	1400
CPU min MHz	1200	1200	1200	1200	1200	1200

Table 1: Raspberry Pi Cluster Specifications

The exact systems specifications for each are shown in table 1. An image of the cluster is shown in figure: 1.

For communication between nodes in the cluster, each Raspberry Pi was network through a D-Link DGS-108 8-port 10M/100M/1G Unmanaged Ethernet switch.

2.2 Software

I used the standard Raspian operating system which is built on the Debian Linux kernel. The Raspian version was GNU/Linux 9 Stretch and can be found here: Raspian Downloads.
The software I used included:

1. Open MPI 4.0.0
2. clang 7.0.1 $tags/RELEASE_701/final$
3. GNU back, version 4.4.21(1)-release (arm-unknown-linux-gnueabihf)

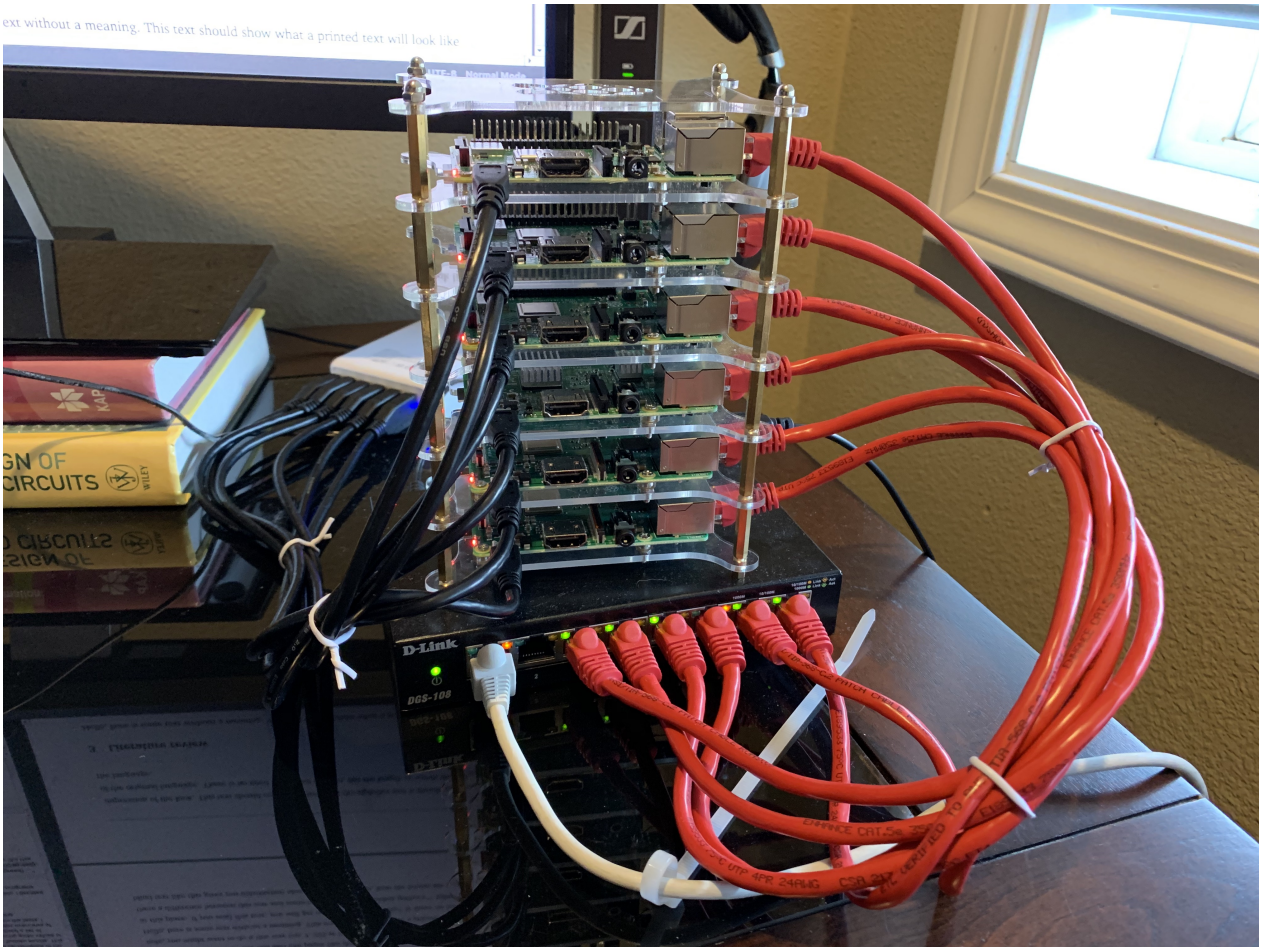

Figure 1: Raspberry Pi Cluster.

3 Solution Algorithm - Backtracking

There are many different algorithms available to find the solution to a general Sudoku puzzle. These range from brute force(Backtrack) to extremely elegant(Don Knuth's Algorithm X.) Because my interest was in exploring in performance impact of programming techniques, using an algorithm which has inherently poor performance would help highlight differences in the programs. Therefore, I chose to use the Backtracking algorithm.
The Back tracking is a recursive algorithm which follows these steps: **btSolve(puzzle):**

1. Select any empty cell in the puzzle. If no empty cells, return PASS
2. If no numbers are left to try, return FAIL.
3. Select a number to try.
4. Validate the solution
5. If the solution is valid:
   (a) Call btSolve(puzzle)
   (b) If return from btSolve is PASS, then return PASS.
   (c) If return from btSolve is FAIL, go back to step 2.
6. Go to step 2.

As will be discussed in the analysis section, this algorithm does not lend itself easily to optimization through parallel methods - i.e. multi threading or multi processing.

4 Program Descriptions and Data

As stated, I used three different programming techniques to solve the Sudoku puzzles. An overview of each is included below. Source code for each can be found at on GitHub at:
https://github.com/jasongraalum/SudoSolv.

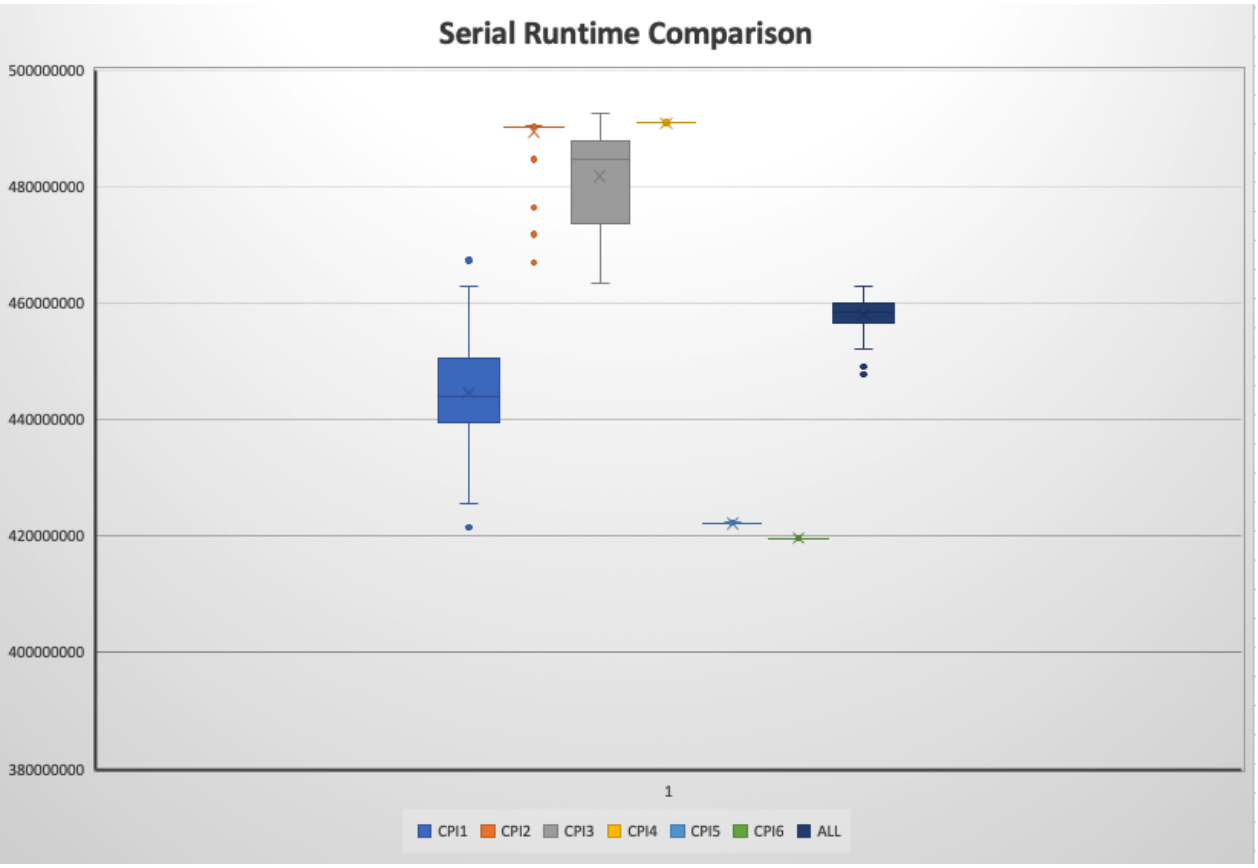

Figure 2: Serial Run-time Comparison

4.1 Serial

The serial solution is identical to the standard algorithm included above. It ran on a single core and solved for the first possible solution. Even if the puzzle had multiple solutions, only the first was found and returned. The source code of the solver is shown in the Appendix A.

4.2 Multi-threaded

As was quickly discovered, the Backtrack algorithm does not lend itself easily to parallel optimization. The one area that is straight forward is in the validation routine. To check if a given solution(partial or complete) is valid, three unique sections need to be checked for duplicate numbers: Rows, Columns and Blocks. In the threaded version of the code, I simply created a thread for each of theses for a total of three threads. The expected advantage is not large as the time spent in the verification routine is only a slightly greater than 50% of the total program time.

At figure 3 shows, the Pthreaded version of the program is much slower(almost 3x slower) that the

serial version. This is due to the simple use of threading to run the verification setup as 3 parallel jobs. This speed up does not cover the overhead of the thread creation. As pointed out during my presentation, creating a thread pool prior to calling the threads may provide some speed up.

4.3 Multi-processor

The mulit-processor solution is where I spent the majority of my effort. I learned quite a bit about MPI programming and the difficulty of managing the interactions between the various processes. Overall, MPI showed the most promise with a runtime nearly equivalent to the serial version. I had begun working on a progam to generate all solutions to a given Sudoku puzzle as this would(I thought and still think) would lend its self much better to a parallel approach. The trouble I ran into was synchronizing the various processes so that they didn't duplicate work or overwrite previous results.I would have liked to spend more time and effort analyzing and expanding the code, but alas, I ran out of time.

5 Conclusions

I had a great time with all the programming I did, which can be found on my GitHub page. However, the amount of programming got in the way of the analysis, so I feel the conclusions I can draw are a bit weak due to lack of analysis.

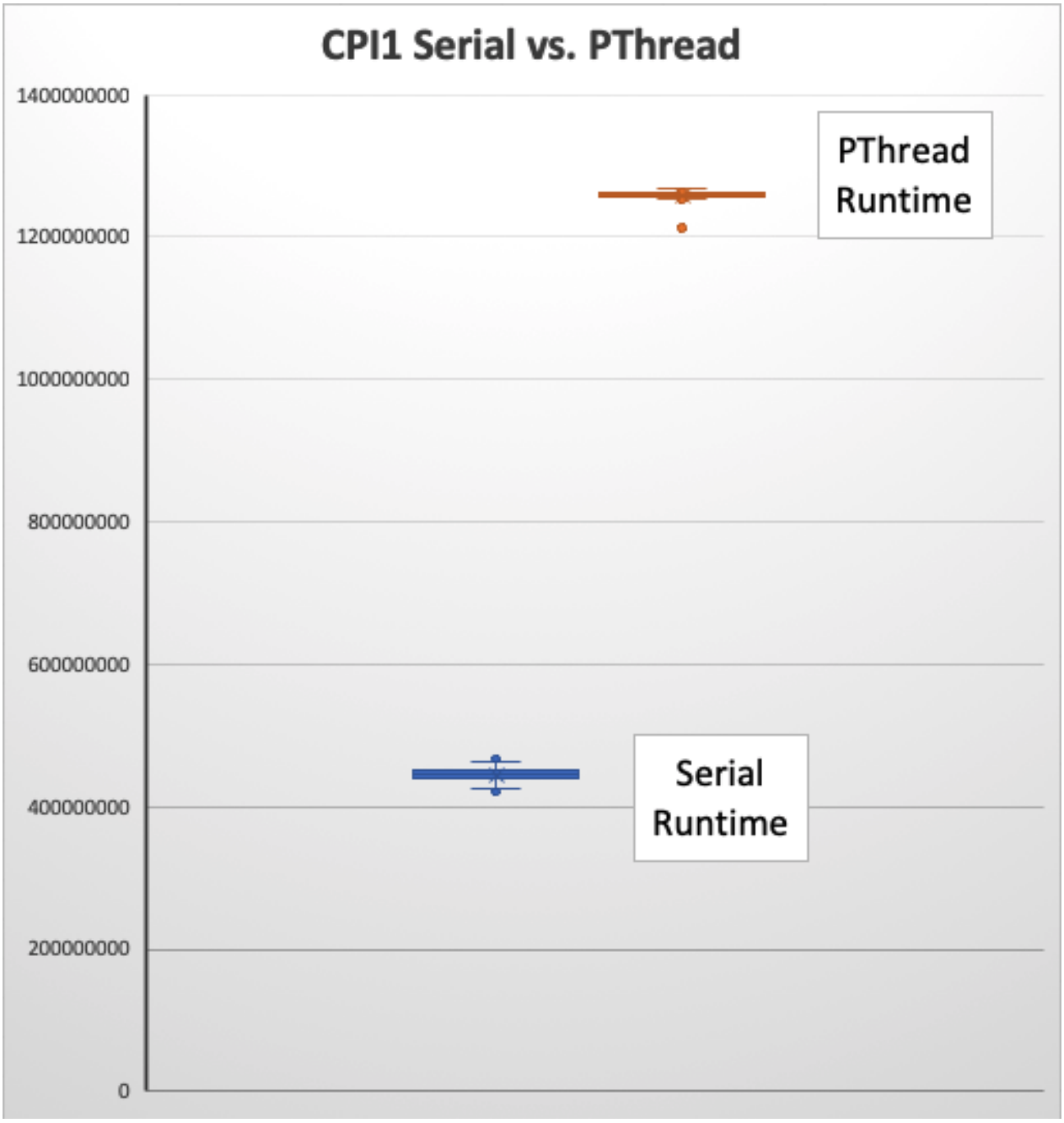

Figure 3: Serial vs Pthread Run-time Comparison

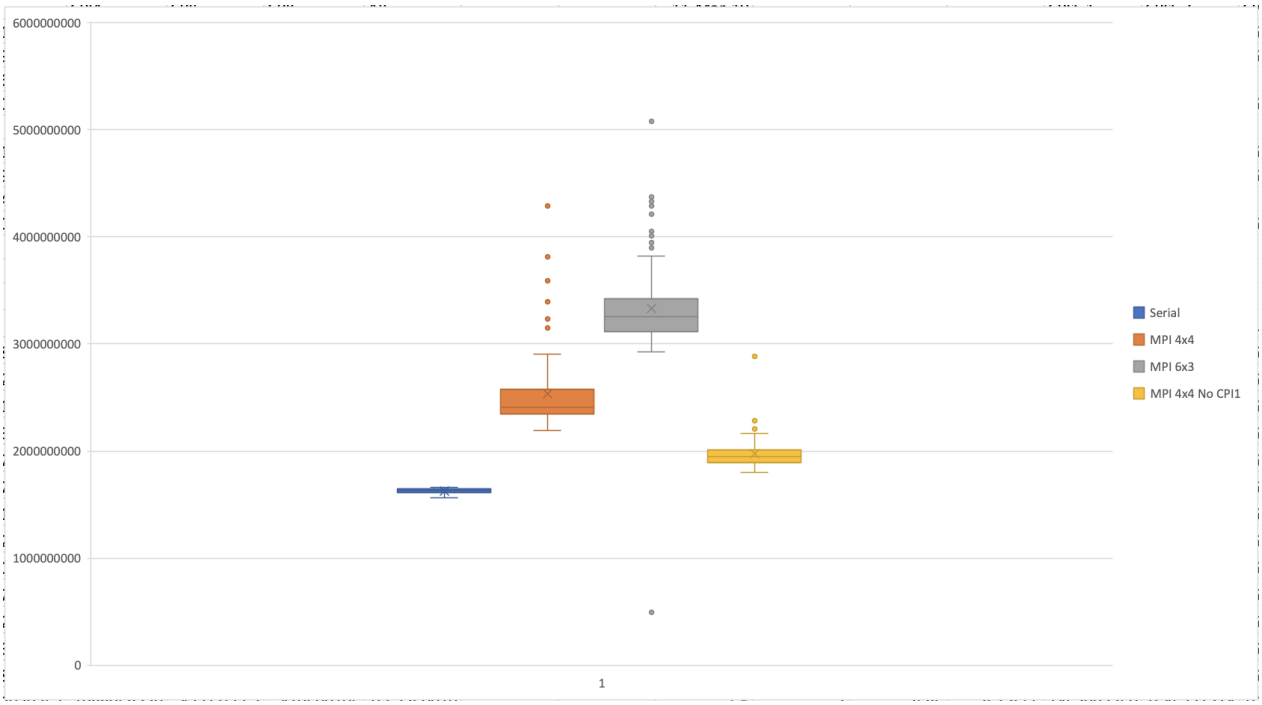

Figure 4: MPI Runtime Comparison

Appendix A: Serial Backtrack Algorithm Code

```
#include "sudoSolvers.h"
#include "sudoSolvUtils.h"

int btSolve(Puzzle * p)
{
    return(addGuess(p, 0, 0));
}

int addGuess(Puzzle * p, int x, int y)
{
    // If cell already has a value
    if(getCell(p, x, y) != 0) {
        int new_x, new_y;
        new_x = (x + 1) % p->degree;
        new_y = new_x == 0 ? (y + 1) : y;

        // If x and y are the last cell, we have a solution
        if(new_y == p->degree) return(verifyPuzzle(p));

        // Call addGuess. If it returns greater than 0, we have a solution
        // If it returns less than 0, this guess_val isn't valid
        return(addGuess(p, new_x, new_y) > 0);
    }

    Cell guess_val = 1;
    do {
```

```
        // Try a number
        setCell(p, x, y, guess_val);

        // If valid, increment cell indexes and call addGuess
        if(verifyPuzzle(p) > 0) {
            int new_x, new_y;
            new_x = (x + 1) % p->degree;
            new_y = new_x == 0 ? (y + 1) : y;

            // If x and y are the last cell, we have a solution
            if(new_y == p->degree) return(1);

            // Call addGuess. If it returns greater than 0, we have a solution
            // If it returns less than 0, this guess_val isn't valid
            if(addGuess(p, new_x, new_y) > 0)
                return(1);
        }
        guess_val++;
        if(guess_val > p->degree) {
            setCell(p, x, y, 0);
            return(-1);
        }
    } while(1);
}
```

Appendix B: Multi-threaded Verification Function

```
int verifyPuzzle()
{
    void *row_result, *col_result, *block_result;

    pthread_t *tid_row, *tid_col, *tid_block;
    tid_row = (pthread_t *)malloc(sizeof(pthread_t));
    tid_col = (pthread_t *)malloc(sizeof(pthread_t));
    tid_block = (pthread_t *)malloc(sizeof(pthread_t));

    // Check rows
    // For all rows
    // For all columns 0 to PDEGREE-1
    // For all columns
    pthread_create(tid_row, NULL, checkPuzzleRows, NULL);

    // Check columns
    // For all columns
    // For all rows 0 to degree-1
    // For all rows
    pthread_create(tid_col, NULL, checkPuzzleCols, NULL);

    // Check subblocks
```

```
    // Need to optimize this.
    // Loop for each subblock - there are PDEGREE subblocks
    pthread_create(tid_block, NULL, checkPuzzleBlocks, NULL);

    pthread_join(*tid_row, &row_result);
    pthread_join(*tid_col, &col_result);
    pthread_join(*tid_block, &block_result);

    int result = ((*(int *)row_result) + (*(int*)col_result) + (*(int *)block_result));

    return(result-6);
}
```

Appendix C: Multi-processor Solution Function

Some inconsequential code has been elided for brevity.

```
int main(int argc, char **argv)
{
    .
    .
    int numProcs, procId;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &procId);

    Puzzle *p;

    char *inFilename;
    int repeats=1;

    // mpiPuzzle Datatype
    int puzzleBlocksCount = 4;
    int puzzleBlocksLength[4] = {1,1,1,1};
    MPI_Datatype puzzleTypes[4] = {MPI_UNSIGNED, MPI_UNSIGNED, MPI_UNSIGNED, MPI_UNSIGNED};
    MPI_Aint puzzleOffsets[4];
    MPI_Datatype mpiPuzzle;
    puzzleOffsets[0] = offsetof(Puzzle, degree);
    puzzleOffsets[1] = offsetof(Puzzle, setCount);
    puzzleOffsets[2] = offsetof(Puzzle, getCount);
    puzzleOffsets[3] = offsetof(Puzzle, cell);
    MPI_Type_create_struct(puzzleBlocksCount, puzzleBlocksLength, puzzleOffsets, puzzleTypes ,&
    MPI_Type_commit(&mpiPuzzle);

    // mpiCell Datatype
    MPI_Datatype mpiCell;
    MPI_Type_create_resized(MPI_UNSIGNED, 0, sizeof(Cell), &mpiCell);
```

```
MPI_Type_commit(&mpiCell);

if(procId == 0) {
    if(argc != 3)
    {
        printf("Usage: sudoSolv <n repeats> <starting puzzle>\n");
        exit(-1);
    }

    repeats = atoi(argv[1]);
    inFilename = argv[2];

    printf("Repeats: %d\n", repeats);
    printf("Input file: %s\n", inFilename);

    time_data = (double *)malloc(sizeof(double)*repeats);
}
.
.
if(procId != 0)
{
    p = (Puzzle *)malloc(sizeof(Puzzle));
    p->degree = 0;
    p->cell = NULL;
    p->setCount = 0;
    p->getCount = 0;
}

for(int repeat = 0; repeat < repeats; repeat++) {
    if(procId == 0) {
        // Setup and print starting puzzle
        p = loadPuzzle(inFilename);
        if (p == NULL) {
            printf("Error loading puzzle. Exitting.\n");
        }
        //printPuzzle(p);
    }

    clock_gettime(CLOCK_REALTIME, &ts_realtime_start);
    //
    // Tranfer Puzzle to all other processes
    //
    MPI_Bcast(p, 1, mpiPuzzle, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    // Transmit puzzle cell contents
    int cell_count = p->degree*p->degree;
    if(procId != 0) p->cell = (Cell *)malloc(sizeof(Cell)*cell_count);
    MPI_Bcast(p->cell,cell_count, mpiCell, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
```

```
        results[repeat] = btMPISolve(p, 0, 0, numProcs, procId);
        .
        .
        int totalSetCount;
        MPI_Reduce((void *)&p->setCount,(void*)&totalSetCount, 1,
                            MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        int totalGetCount;
        MPI_Reduce((void*)&p->getCount,(void*)&totalGetCount, 1,
                            MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  .
  .
    }

    if(procId == 0) {
        clock_gettime(CLOCK_REALTIME, &ts_totaltime_end);
        for(int repeat = 0; repeat < repeats; repeat++) {
            printf("%d,\t%d,\t%d,\t%d,\t%lf\n", repeat, results[repeat], sets[repeat], gets
                    [repeat], time_data[repeat]);
        }

        printf("Total time: %lf\n", (double)(1e9*(ts_totaltime_end.tv_sec-ts_totaltime_start.tv
        //printPuzzle(p);
    }
    free(p->cell);
    free(p);
    MPI_Finalize();
    return 0;
}


#include "sudoMPISolvers.h"
#include "sudoMPISolvUtils.h"

int btMPISolve(Puzzle * p, int row, int col, int numProcs, int procId)
{
    int new_x, new_y;
    int * proc_res = (int*)malloc(sizeof(int));
    int *all_results = (int*)malloc(sizeof(int)*numProcs);

    // Get next empty cell, if no empty cell, return pass
    while(row < p->degree && col < p->degree && getCell(p, row, col) != 0) {
        row = (row + 1) % p->degree;
        col = (row == 0) ? (col + 1) : col;
    }
    if (col == p->degree) return(1);
    // setCell and Verify
    setCell(p, row, col, procId+1);
    *proc_res = verifyPuzzle(p);
    setCell(p, row, col, 0);

    // Gather all results
```

```
    MPI_Allgather(proc_res,1,MPI_INT,all_results,1,MPI_INT, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    // Pick first passing result
    for(int i = 0; i < numProcs; i++)
    {
        // setCell
        if(all_results[i] > 0) {
            setCell(p, row, col, i+1);
            int next_proc_res = btMPISolve(p, row, col, numProcs, procId);

            // If solver return is pass, return pass
            if(next_proc_res > 0) return(1);
            setCell(p, row, col, 0);
        }
        // If solver return is fail, try next passing result
    }
    return(0);
}
```