# CSE 430: Operating Systems

Instructor: Dr. Violet R. Syrotiuk

# Covering Arrays

Design and Analysis Report

Final Report

Project 1

Author: Ankit Rathi

ASU ID: 1207543476

# Contents

# Introduction

### What are Covering Arrays?

Covering arrays are used in testing software, hardware, composite materials, biological networks, and others. They also form the basis for combinatorial methods to learn an unknown classification function using few evaluations - these arise in computational learning and classification, and hinge on locating the relevant attributes.

A covering array CA(N; t; k; v) is an N x k array where each N x t subarray contains all ordered t-sets on v symbols at least once; t is called the strength of the covering array.

The first milestone of this project is to design and analyze the C/C++ program to check if the given input file is Covering Array or not. If it is Covering Array, the Don't Care Matrix along with the position or the (r,c) co-ordinates of the don't care terms. I have developed the C program for detecting the Covering Array.

# Reading the Input file

The program has been designed to read data from the "stdin". To read the data from the "stdin", we have to put the data onto stdin while executing the program. This can be done with the command like "p1_AnkitRathi.o < 01-CA(6;2,5,2)". As per the format of the input line the first line will be in the form of N t k v, each separated by a space. After reading the first line, we have to read the next N lines, each line containing k elements. Each lines has a single space between two elements. This space is eliminated by the program, so even if there are more number of spaces then as expected, the program will still sun as it bypasses any space it encounters. After each line it reads, it encounters a new line character indicating it is now beginning to read the next line. This mentioned logic has been implemented in the code as shown below. The program outputs the results onto the console and also generates an output file in the same location as from where it is run.

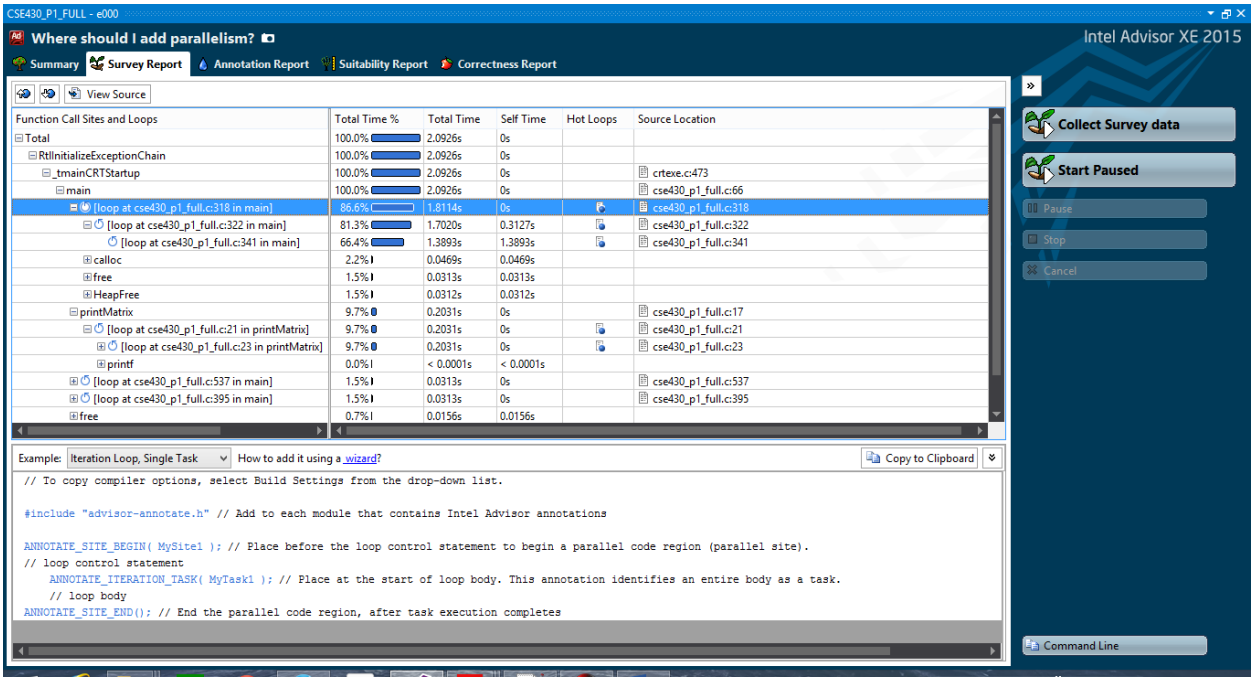
# Design and Analysis

### Survey Target

The first step to convert the serial program into the parallel version of the program is to make use of Intel Advisor XE 2015 tool. So, in the first step I am surveying the target using the release build. This is a proofing tool and here we get an idea about the part of the code that needs to be parallelized. From the survey analysis I found that there are 5 sections in the serial program that can be parallelized. This means that 5 hotspots were detected in the program.

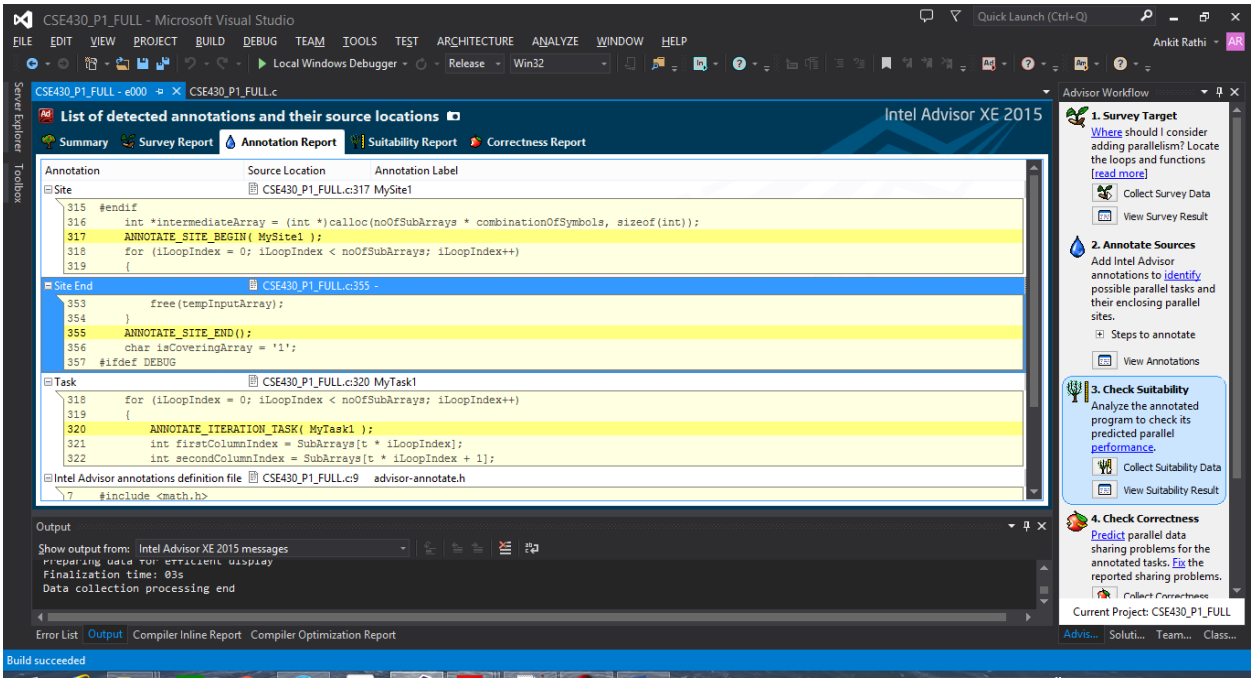
First 3 hotspots corresponds to the loop that generates the intermediate matrix. The intermediate matrix is the second matrix which is generated as was discussed in the class. This is the most time consuming part of the code. It takes about 86.6% of the time to generate the second matrix, hence this needs to be parallelized.

Fourth and Fifth hotspots corresponds to the loop to print the matrix onto the console and as well into the file. This takes up to 9.7% of the total time of execution. Hence this can also be parallelized.




## Annotate Sources

This is the second step in parallelizing the code. Here we annotate the source code to tell the Intel Advisor that this is the part of the code I will be parallelizing. Here we define the parallel sites and within the parallel sites we define the tasks that are going to be executed in parallel.

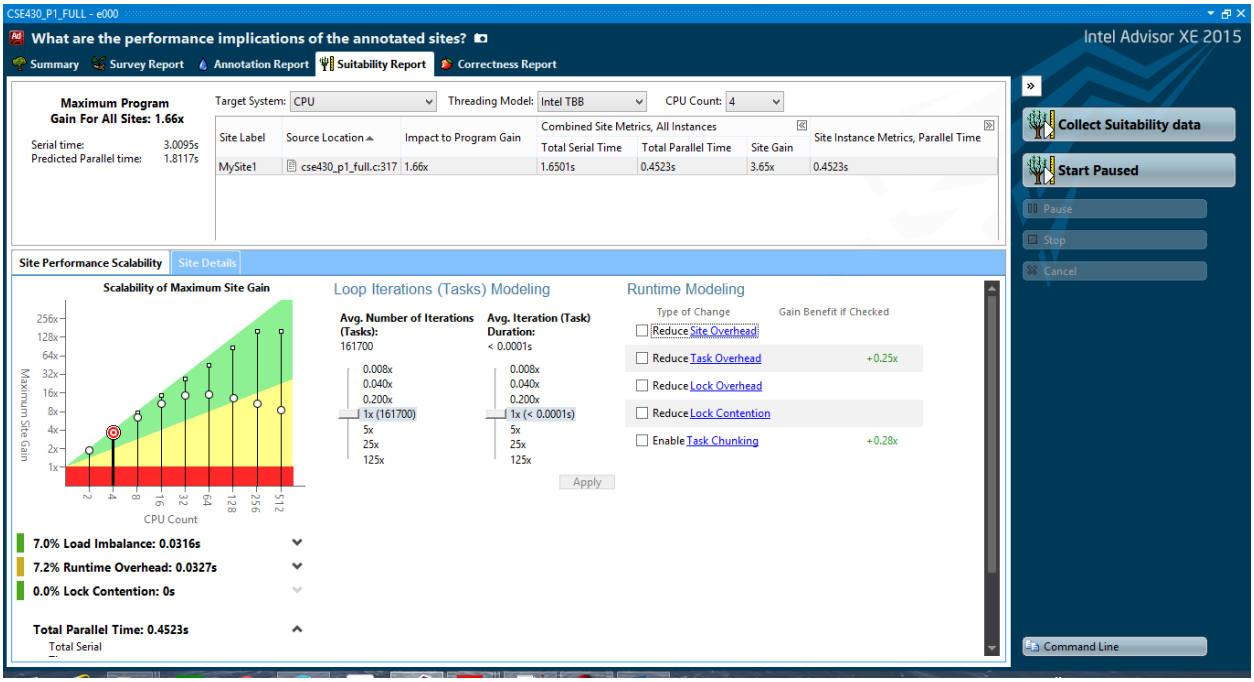


## Check Suitability

This step is used to analyze the annotated program to check its predicted parallel performance.

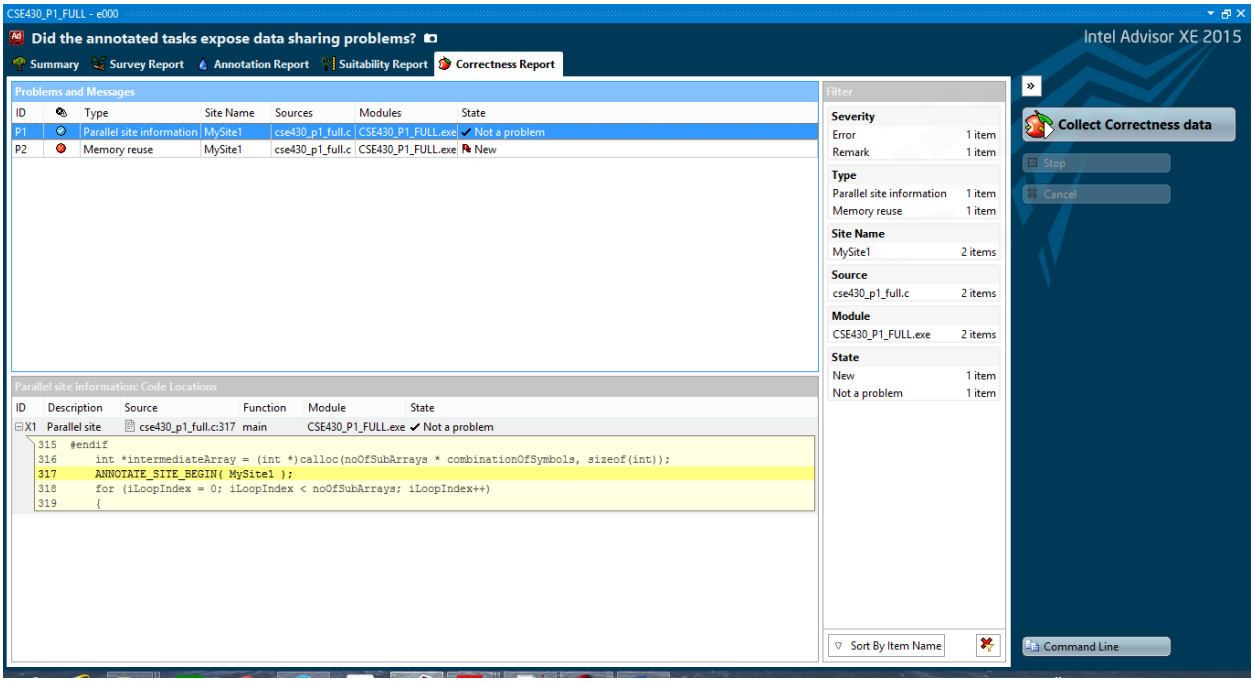
$$\frac{\text{Total Serial Time of Site}}{\text{Total Parallel Time of Site}} = \frac{1.6501}{0.4523} = 3.65\ (Maximum\ Site\ Gain)$$

$$\frac{\text{Total Serial Time}}{\text{Total Parallel Time}} = \frac{3.0095}{1.8117} = 1.66\ (Maximum\ Program\ Gain)$$

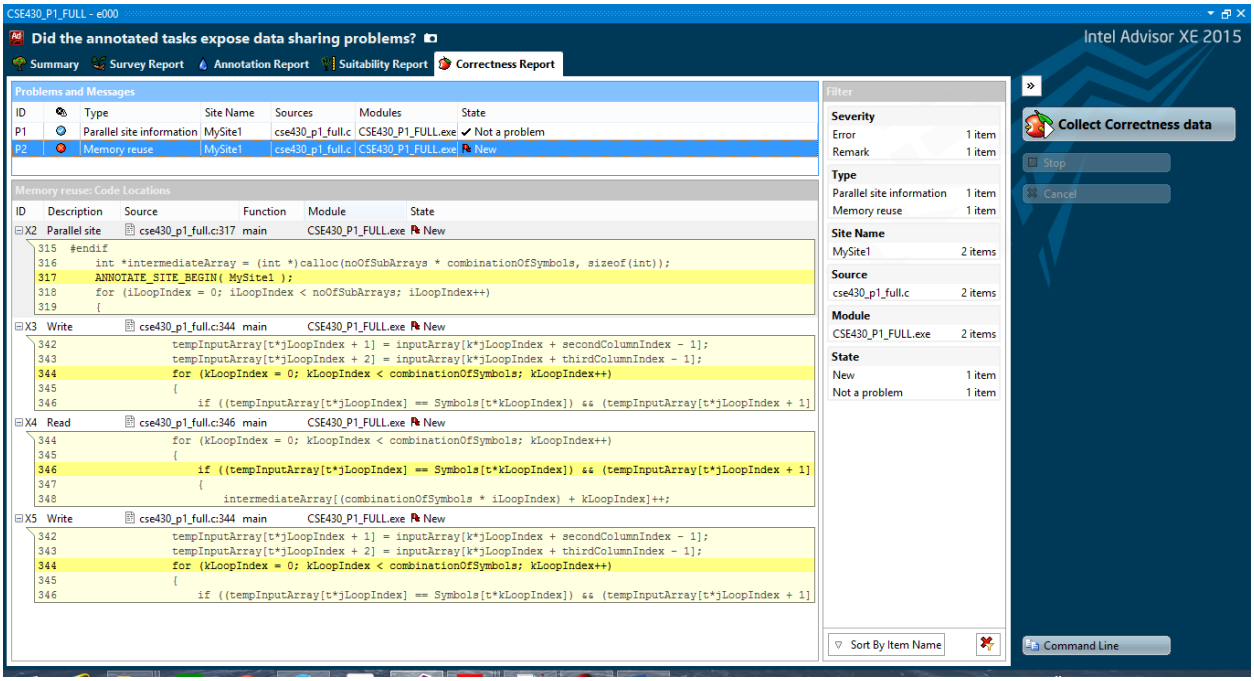



## Check Correctness

This step is used to predict parallel data sharing problems for the annotated tasks. Here we can find various kinds of problems that may have been induced into the code after the code is parallelized. Some of the issues include race conditions, Memory reuse, Data Communication.




As it can be seen from the above screenshot that one memory reuse issue has been reported after the code with annotations is assumed to be the parallel version of the code. This error needs to be handled and we need to make the variable as shared in the omp declarations so that the parallel running threads share the same variable without creating any memory reuse and data race conditions.




Add Parallel Framework

This is the last step in the Intel Advisor to parallelize the code. Here the annotations that had been added before to analyze the parallel code now needs to be replaced by the actual code to convert it into the parallel code.

```
327
328  #pragma omp parallel  num_threads(4) shared(noOfSubArrays, SubArrays, t, N, inputArray, combinationOfSymbols, Symbols, intermediateArray)
329      {
```

As shown in the above snapshot, the #pragma omp parallel for has been added to make the code run in parallel. The keyword **parallel** is to indicate the below piece of code will run in parallel mode and the keyword **for** is used to indicate that the each iterations of loop is assigned to the threads and they executed by individual threads.

### Introducing Threads

Num_threads(4) is used to introduce 4 threads into the parallel site. The 4 threads will divide the work among themselves.

The part of the program which generates the intermediate matrix, or the second matrix as per the class discussion is the code that needs to be parallelized. It consists of 3 nested loops. I have tried to parallelized the outermost loop. As parallelizing the outermost loop will let the inner loops to run by each thread hence the entire code appears to be running in parallel. Load Balancing among the threads can be done by making use of the schedule (dynamic, 50). Here instead of dynamic, other methods of scheduling can also be used such as runtime, static and guided.

```
329      {
330  #pragma omp for  private(iLoopIndex, jLoopIndex, kLoopIndex) schedule(dynamic, 50)
331          for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
332          {
```

I have defined chuck_size=50, so the compiler will give 50 iteration to each thread and after the first chunk of iterations is finished by the thread, it requests for the next work to be assigned to it.

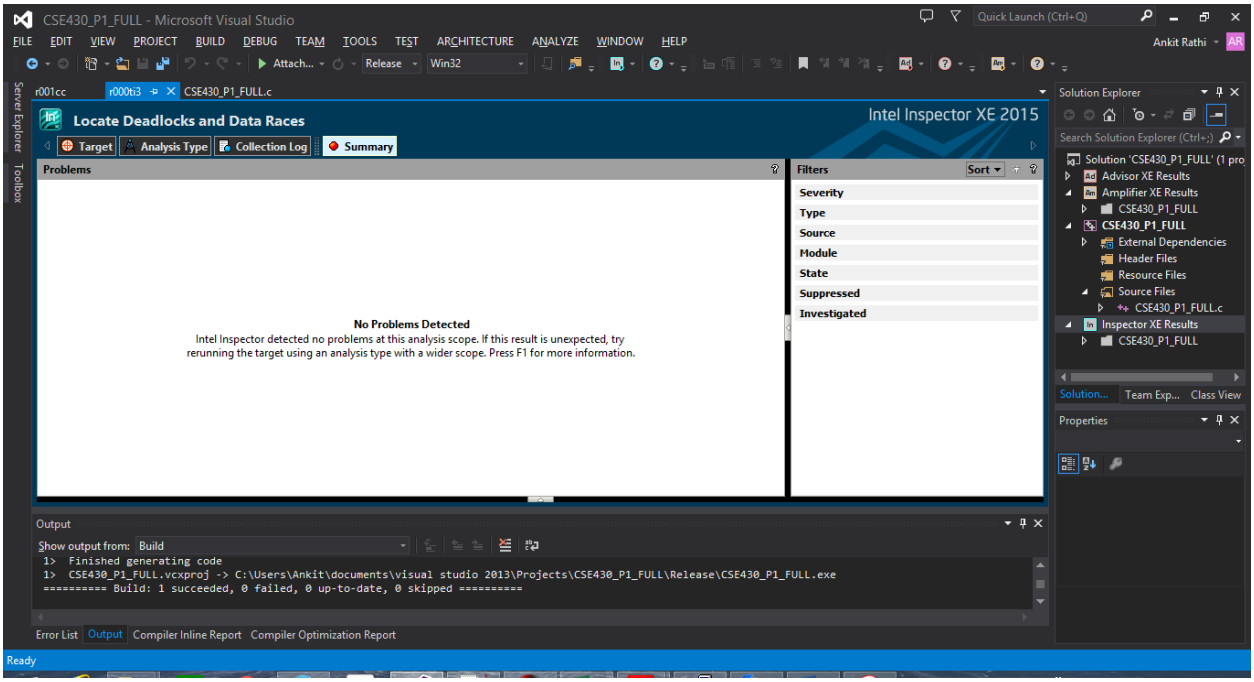
### Load Balancing

The load balancing as discussed in the previous section is done by using schedule (dynamic, 50). Here I have given chunk_size=50.

## Locate Deadlocks and Data Races

### Errors that Arose

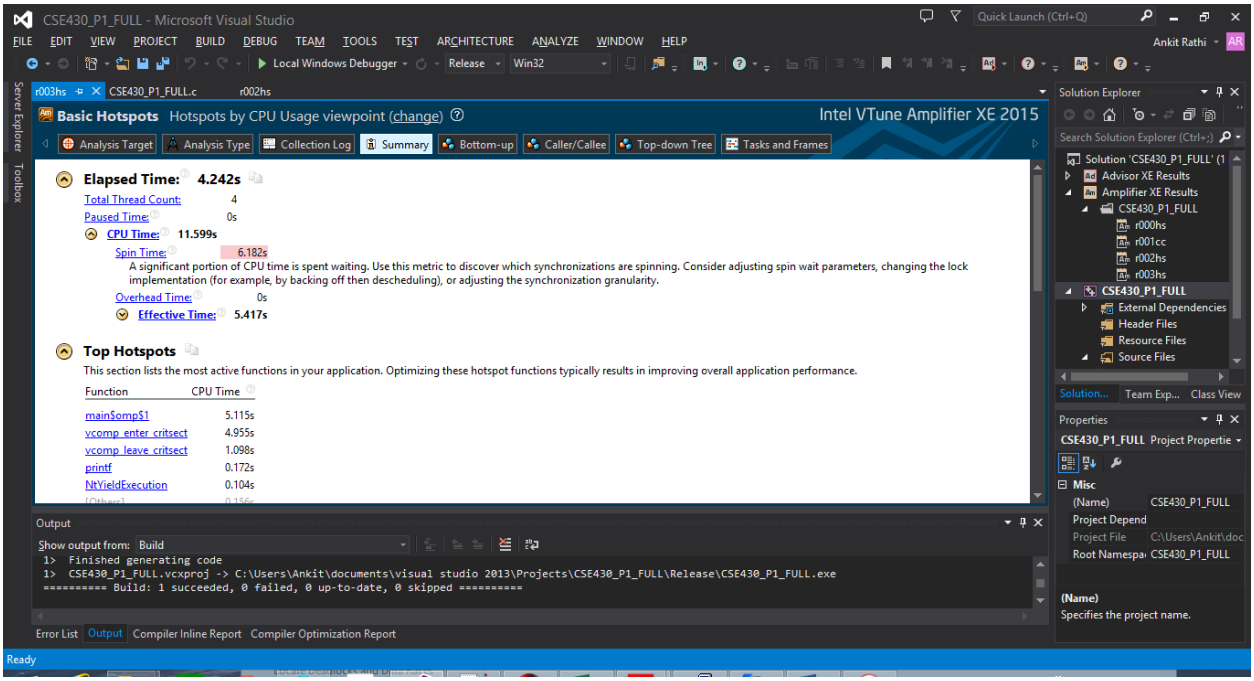
There were problems related to data race conditions in the code, after making the code parallel. This is because some the shared variables were read and written at the same time by multiple threads leading to wrong output. This was handled by making those variables shared and some variables which each thread creates for itself. These are declared as private in the omp parallel for declaration.

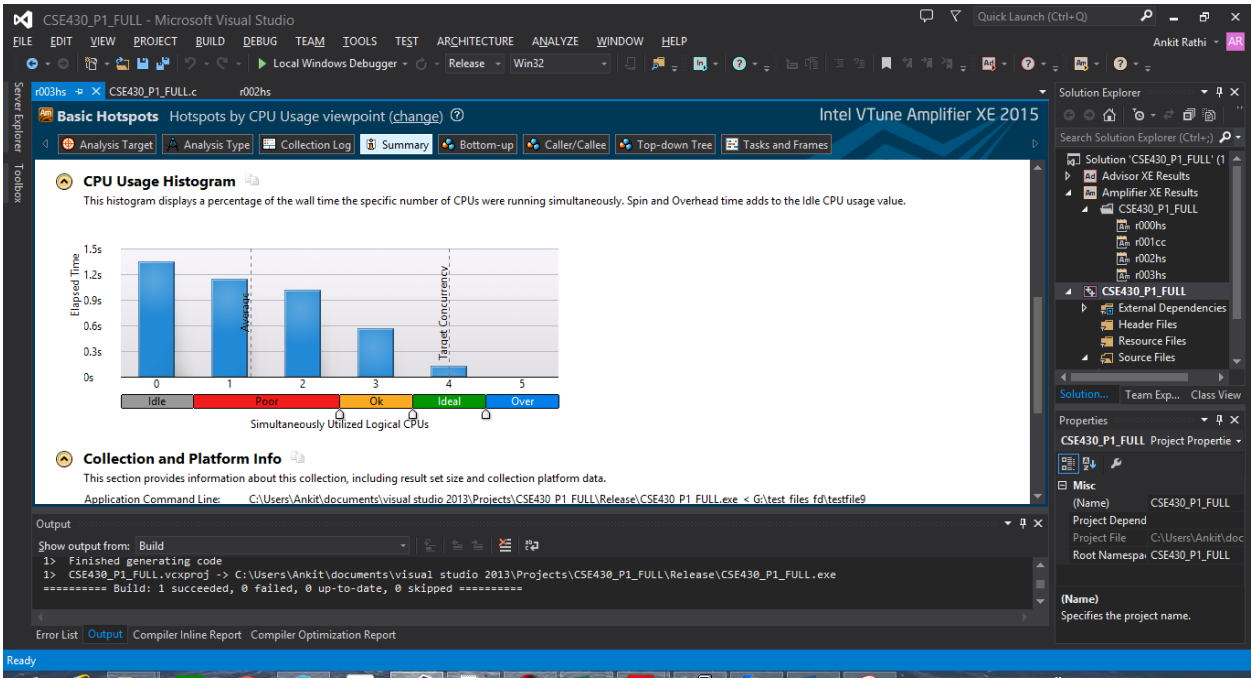



# Improving the speedup of Parallel program

## Hotspot Analysis

Hotspot analysis is used to identify the most time consuming source code. As per the report generated most time consuming part of the code is waiting at the critical section.



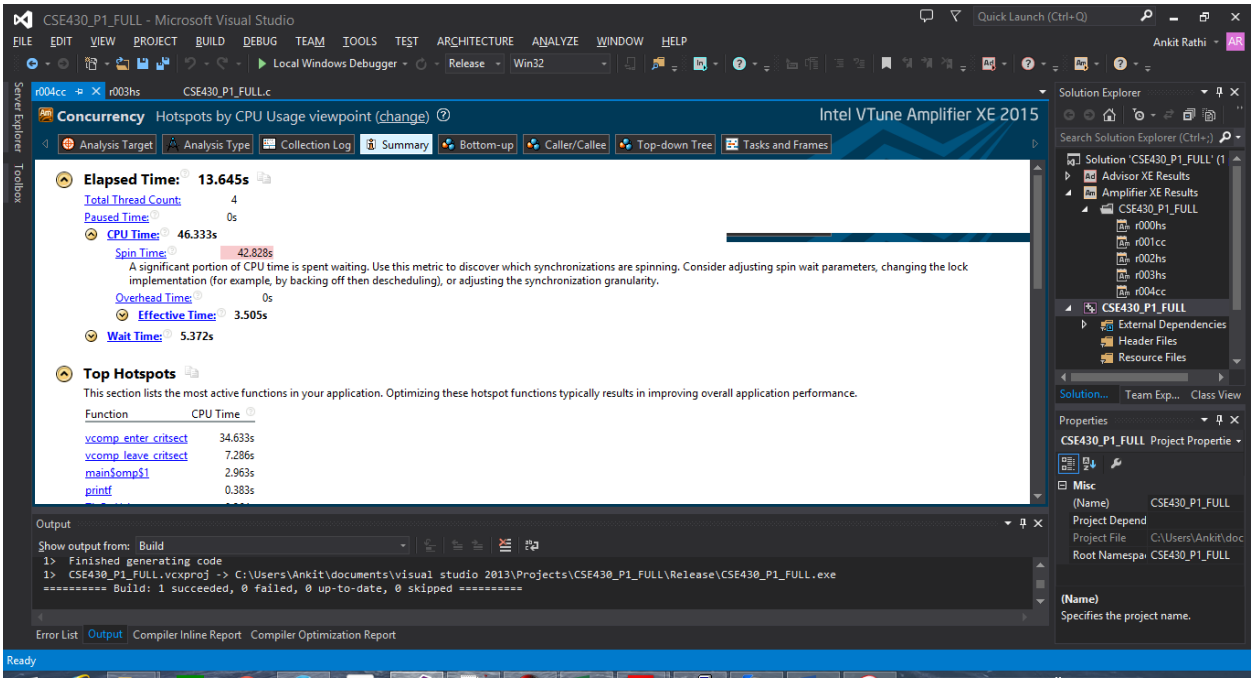


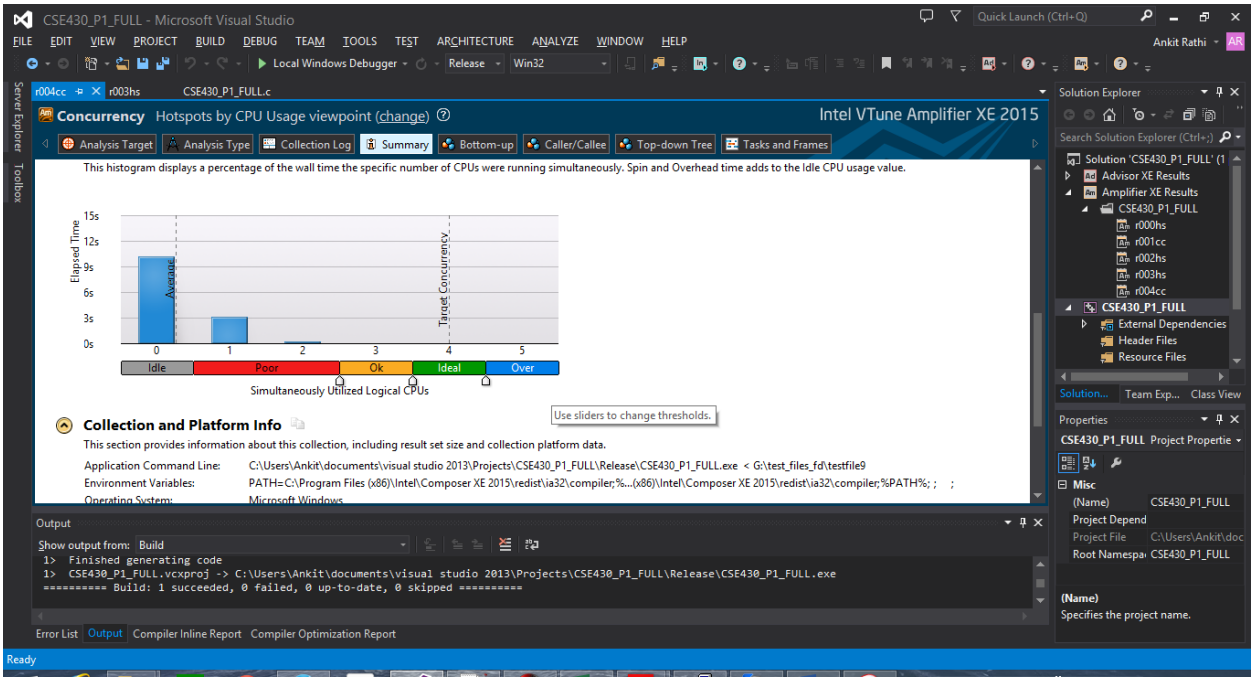

From the above screenshots it is very clear that lot of time is spent waiting for critical section. This can be improved by making use of atomic increments instead of using #pragma omp critical section by __sync_fetch_and_add() function.

```
//#pragma omp critical
                        //intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex]++;
                        __sync_fetch_and_add(&intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex], 1);
                    }
                }
            }
```

## Concurrency Analysis

Concurrency Analysis is used to analyze how the application is using the available logical CPUs, discover where parallelism is incurring synchronization overhead and identify the potential candidates for parallelization. In this kind of analysis, we use the user mode sampling and trace collections.



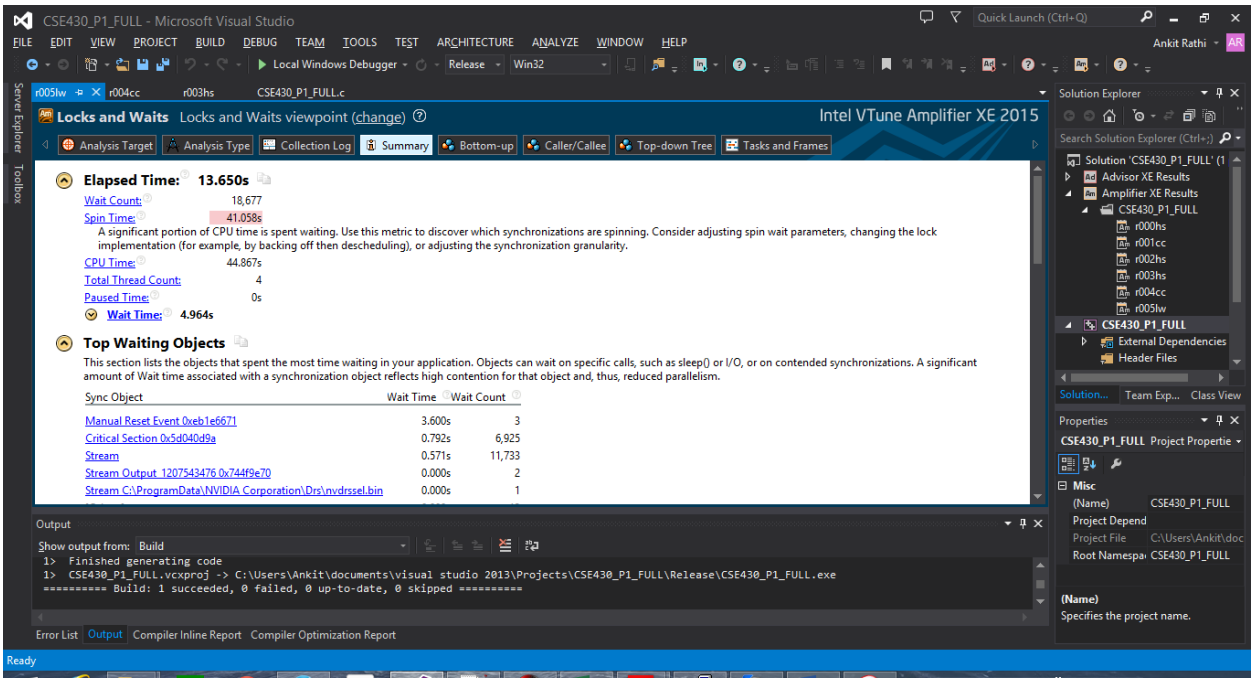



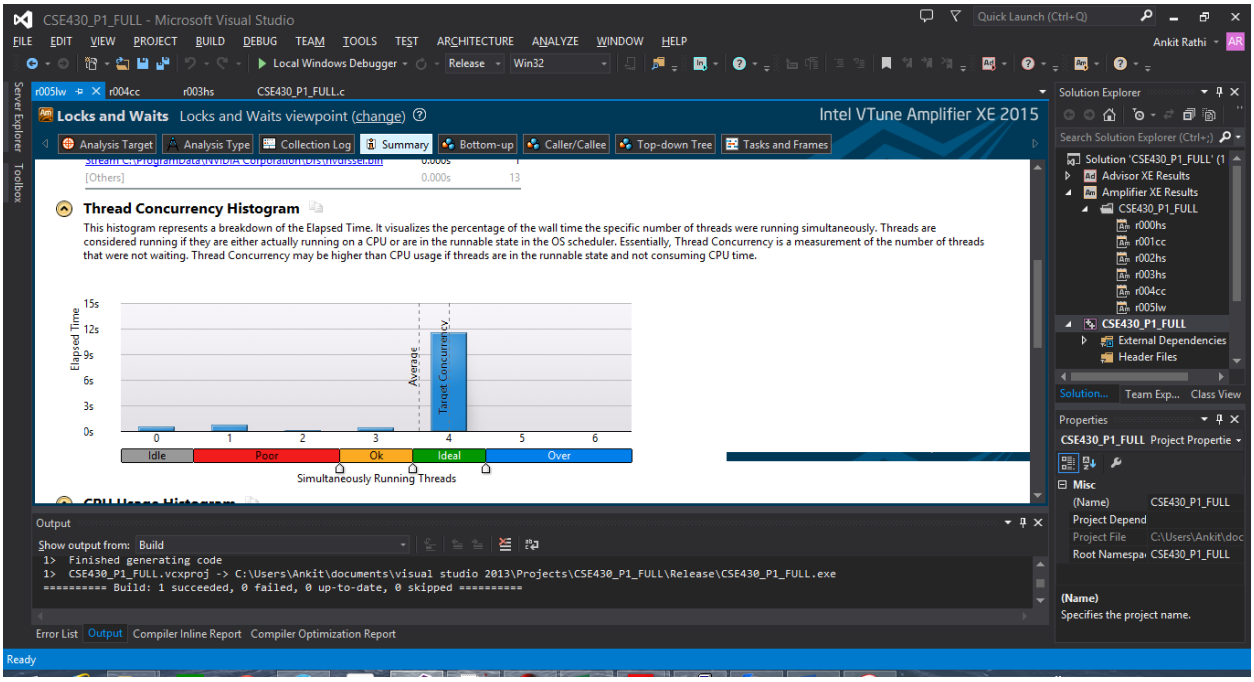

From the above screenshots it is quite evident that concurrency analysis is also indicating that lot of time is spent waiting for critical section. Hence we need to modify this critical section. This can be improved by making use of atomic increments instead of using #pragma omp critical section by __sync_fetch_and_add() function.

```
361 //#pragma omp critical
362                     //intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex]++;
363                     __sync_fetch_and_add(&intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex], 1);
364                 }
365             }
366         }
```

## Locks and Waits

This analysis is used to determine which part of the application is waiting for synchronization objects or I/O operations and it can also be used to analyze how these wait periods affect applications performance.



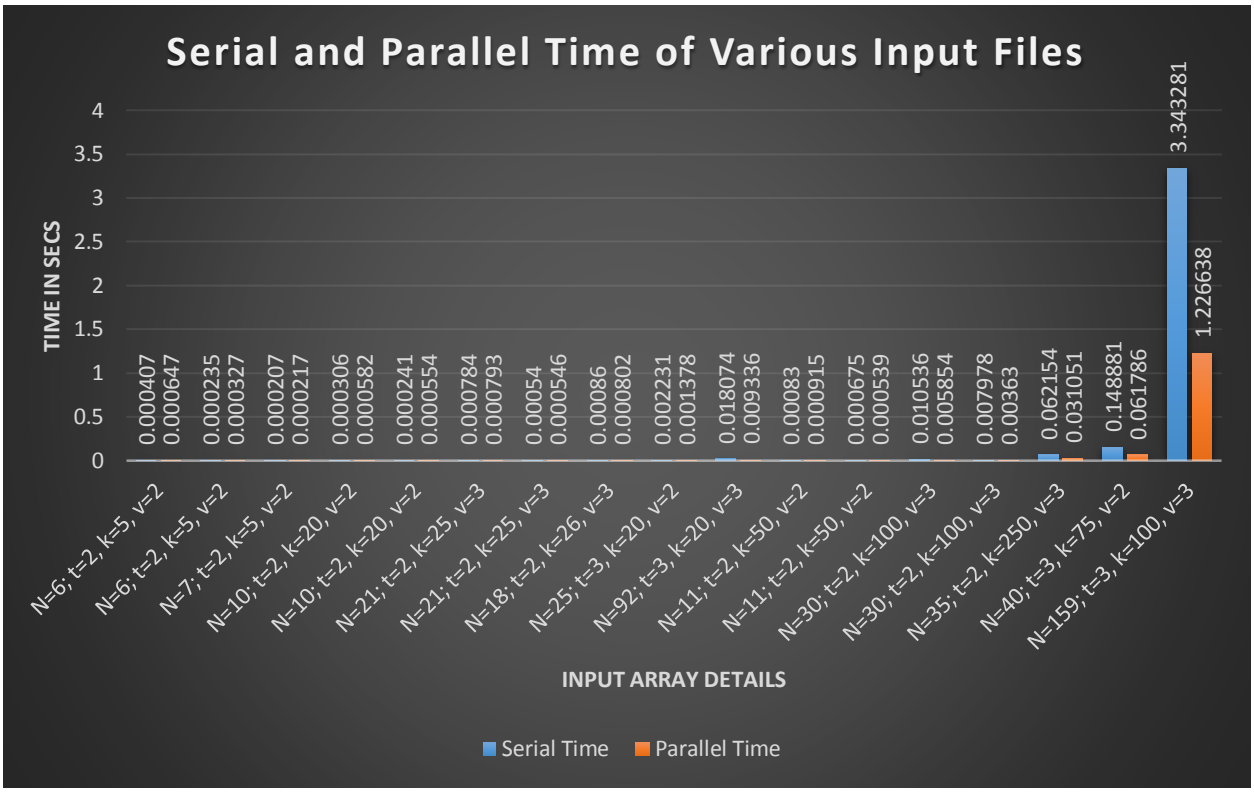

This screenshot also indicates a lot of time being spent in critical section.

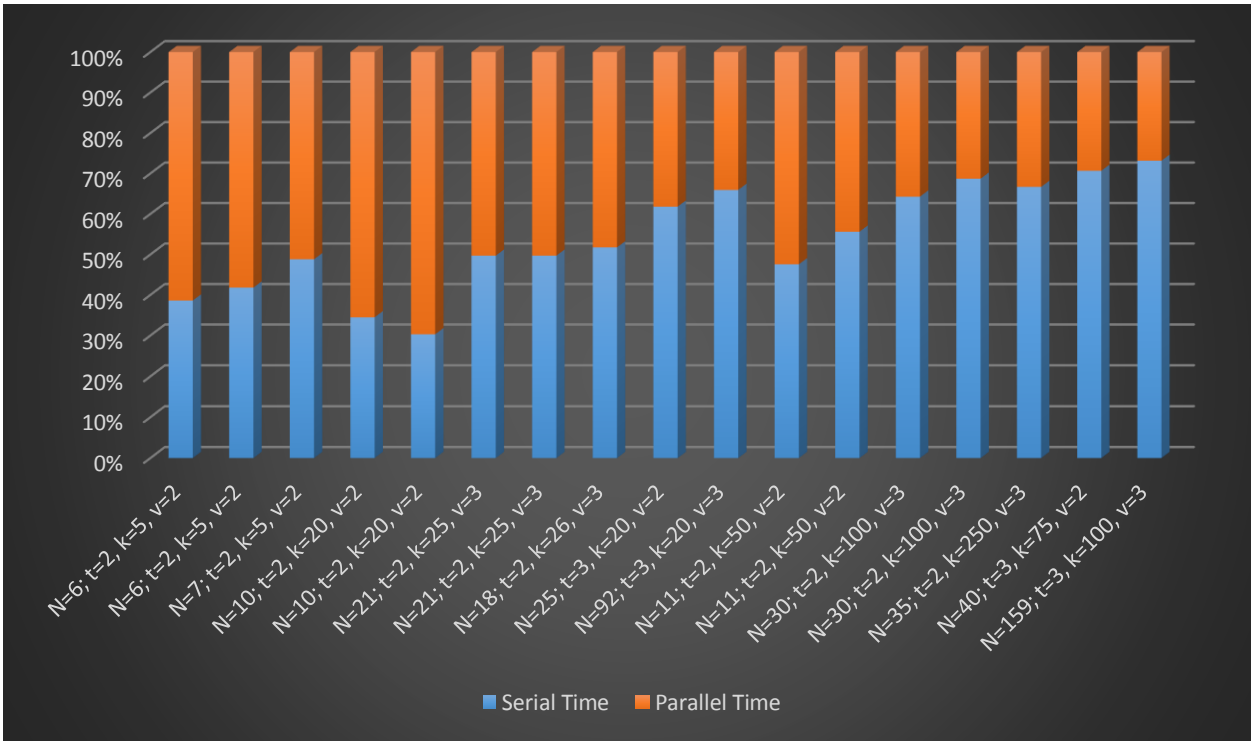
## Running on ASURE: num_threads(4)

Both Serial and the parallel program have been copied to the ASURE. After they have been copied to ASURE, we need to run program on the sample input files. This is done with the help of shell script file. The shell script file consists of the commands to execute the serial and parallel program serially. The output obtained after running shell script is recorded into the table as presented below.

| Inputs | k choose t | Combination | Serial Time | Parallel Time | SpeedUP |
|---|---|---|---|---|---|
| N=6; t=2, k=5, v=2 | 10 | 4 | 0.000407 | 0.000647 | 0.629057 |
| N=6; t=2, k=5, v=2 | 10 | 4 | 0.000235 | 0.000327 | 0.718654 |
| N=7; t=2, k=5, v=2 | 10 | 4 | 0.000207 | 0.000217 | 0.953917 |
| N=10; t=2, k=20, v=2 | 190 | 4 | 0.000306 | 0.000582 | 0.525773 |
| N=10; t=2, k=20, v=2 | 190 | 4 | 0.000241 | 0.000554 | 0.435018 |
| N=21; t=2, k=25, v=3 | 300 | 9 | 0.000784 | 0.000793 | 0.988651 |
| N=21; t=2, k=25, v=3 | 300 | 9 | 0.00054 | 0.000546 | 0.989011 |
| N=18; t=2, k=26, v=3 | 325 | 9 | 0.00086 | 0.000802 | 1.072319 |
| N=25; t=3, k=20, v=2 | 1140 | 8 | 0.002231 | 0.001378 | 1.619013 |
| N=92; t=3, k=20, v=3 | 1140 | 27 | 0.018074 | 0.009336 | 1.935947 |
| N=11; t=2, k=50, v=2 | 1225 | 4 | 0.00083 | 0.000915 | 0.907104 |
| N=11; t=2, k=50, v=2 | 1225 | 4 | 0.000675 | 0.000539 | 1.252319 |
| N=30; t=2, k=100, v=3 | 4950 | 9 | 0.010536 | 0.005854 | 1.799795 |
| N=30; t=2, k=100, v=3 | 4950 | 9 | 0.007978 | 0.00363 | 2.197796 |
| N=35; t=2, k=250, v=3 | 31125 | 9 | 0.062154 | 0.031051 | 2.001675 |
| N=40; t=3, k=75, v=2 | 67525 | 8 | 0.148881 | 0.061786 | 2.409624 |
| N=159; t=3, k=100, v=3 | 161700 | 27 | 3.343281 | 1.226638 | 2.725565 |

As it can be seen from the tabulated values, only for some input files we are not able to achieve speedup greater than 1. Most of the values have serial time more than parallel time. This after analysis can be explained by the fact that as the input size is increasing, the speedup also increases. Therefore when N increases, we are able to achieve speedup greater than 1. Speedup > 1 is marked in green and speedup < 1 is marked in red.

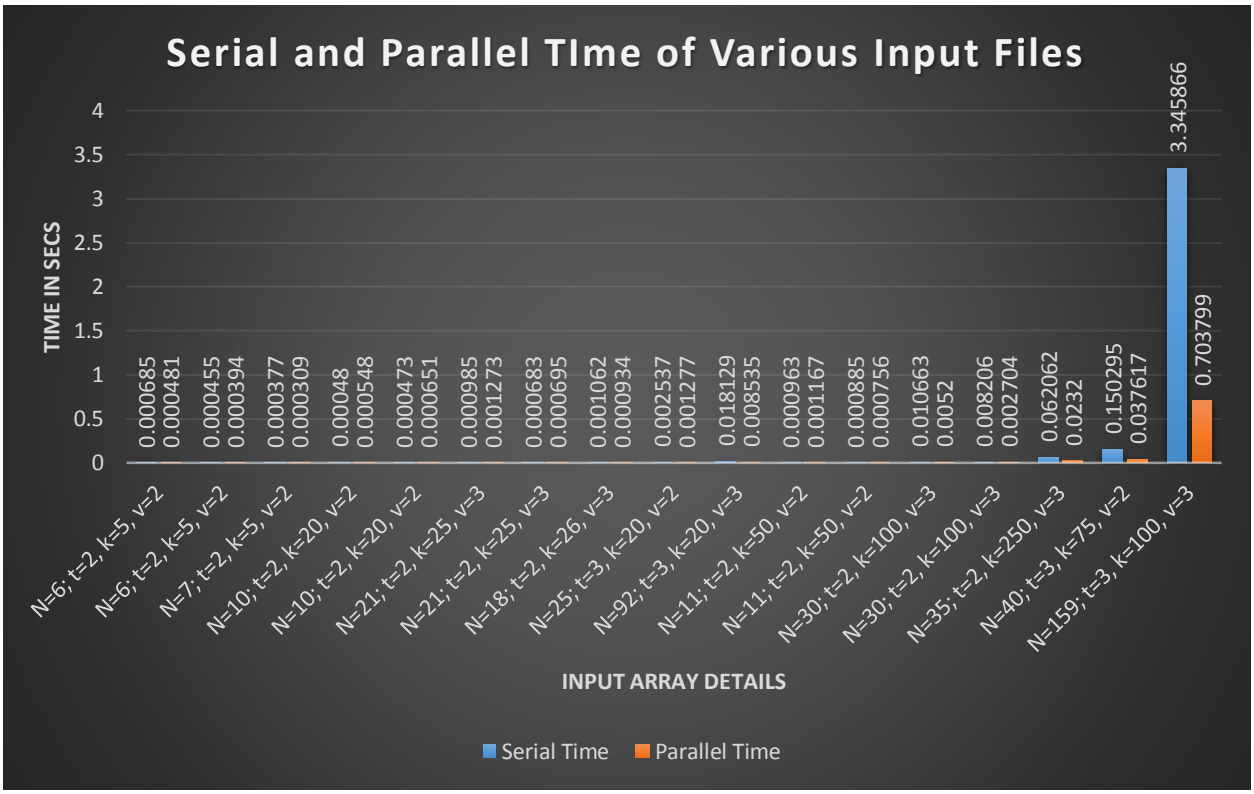


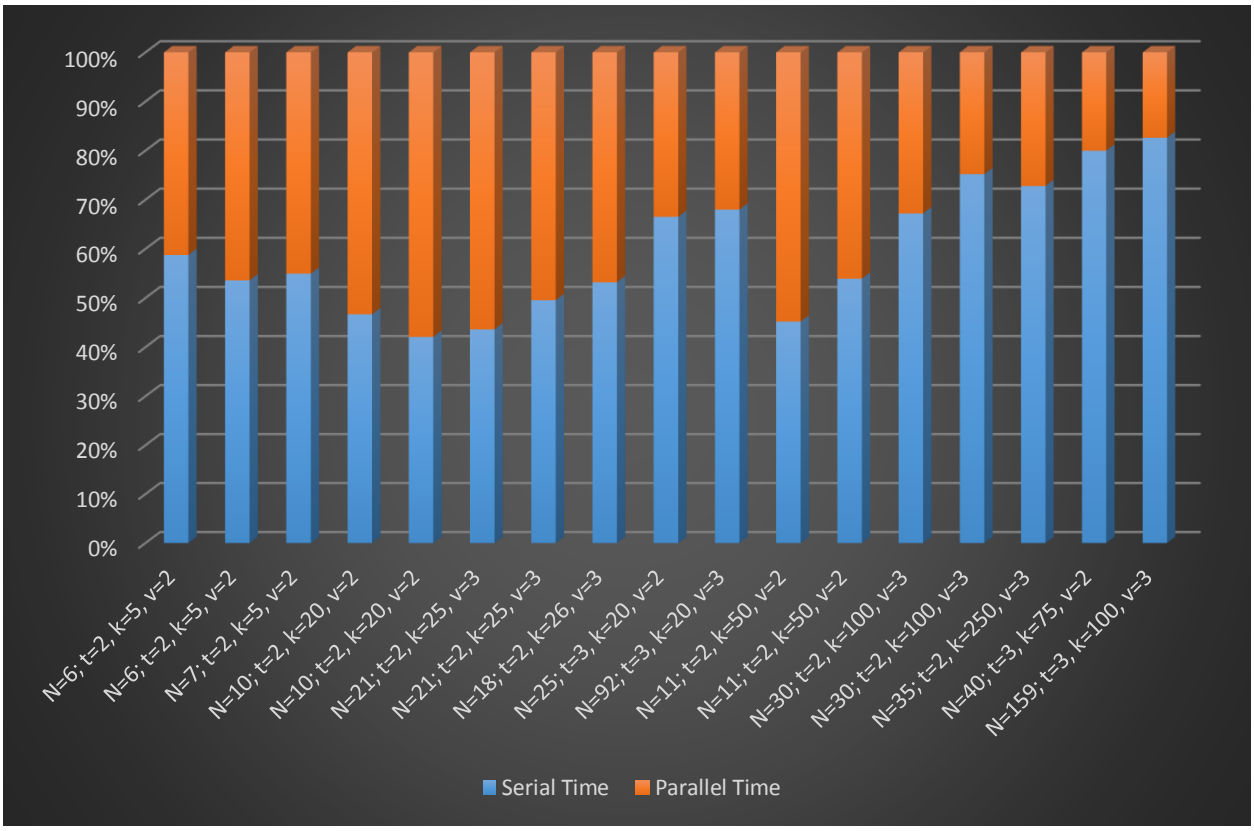




## Running on ASURE: num_threads(8)

| Inputs | k choose t | Combination | Serial Time | Parallel Time | SpeedUP |
|---|---|---|---|---|---|
| N=6; t=2, k=5, v=2 | 10 | 4 | 0.000685 | 0.000481 | 1.424116 |

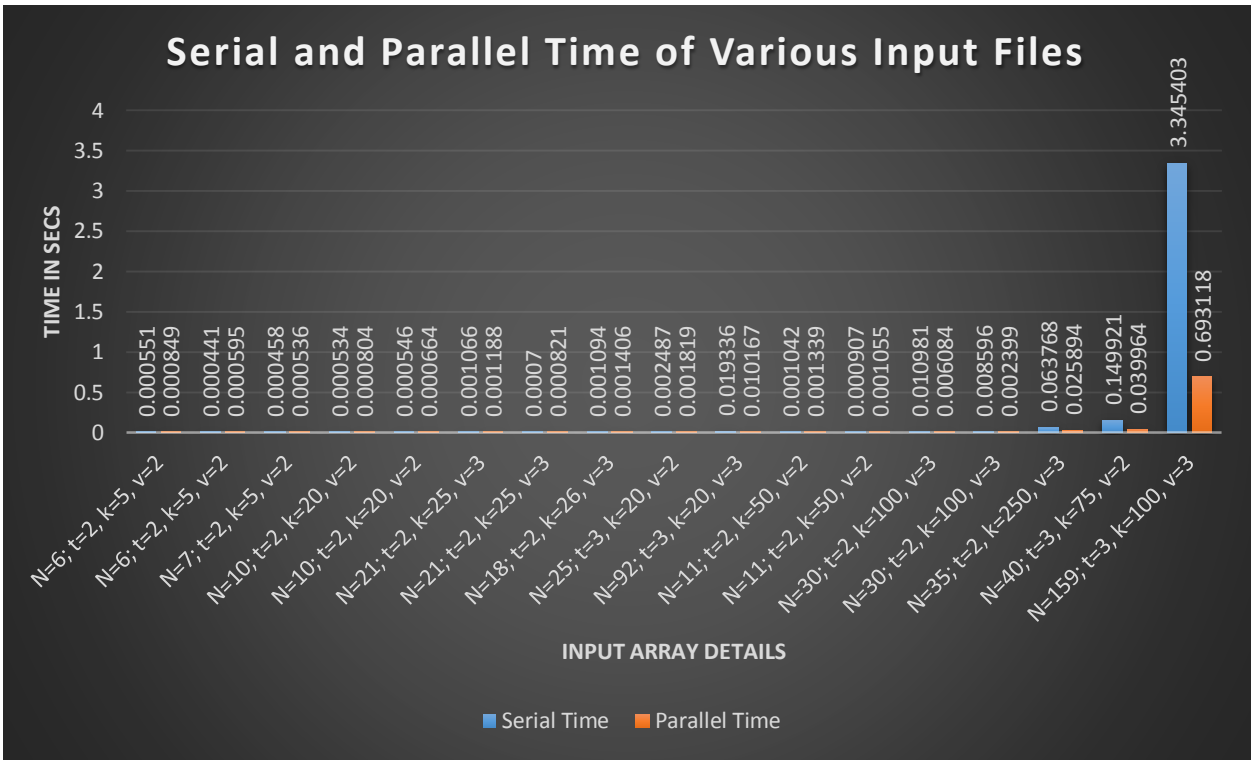
| | | | | | |
|---|---|---|---|---|---|
| N=6; t=2, k=5, v=2 | 10 | 4 | 0.000455 | 0.000394 | 1.154822 |
| N=7; t=2, k=5, v=2 | 10 | 4 | 0.000377 | 0.000309 | 1.220065 |
| N=10; t=2, k=20, v=2 | 190 | 4 | 0.00048 | 0.000548 | 0.875912 |
| N=10; t=2, k=20, v=2 | 190 | 4 | 0.000473 | 0.000651 | 0.726575 |
| N=21; t=2, k=25, v=3 | 300 | 9 | 0.000985 | 0.001273 | 0.773763 |
| N=21; t=2, k=25, v=3 | 300 | 9 | 0.000683 | 0.000695 | 0.982734 |
| N=18; t=2, k=26, v=3 | 325 | 9 | 0.001062 | 0.000934 | 1.137045 |
| N=25; t=3, k=20, v=2 | 1140 | 8 | 0.002537 | 0.001277 | 1.986688 |
| N=92; t=3, k=20, v=3 | 1140 | 27 | 0.018129 | 0.008535 | 2.124077 |
| N=11; t=2, k=50, v=2 | 1225 | 4 | 0.000963 | 0.001167 | 0.825193 |
| N=11; t=2, k=50, v=2 | 1225 | 4 | 0.000885 | 0.000756 | 1.170635 |
| N=30; t=2, k=100, v=3 | 4950 | 9 | 0.010663 | 0.0052 | 2.050577 |
| N=30; t=2, k=100, v=3 | 4950 | 9 | 0.008206 | 0.002704 | 3.034763 |
| N=35; t=2, k=250, v=3 | 31125 | 9 | 0.062062 | 0.0232 | 2.675086 |
| N=40; t=3, k=75, v=2 | 67525 | 8 | 0.150295 | 0.037617 | 3.995401 |
| N=159; t=3, k=100, v=3 | 161700 | 27 | 3.345866 | 0.703799 | 4.754008 |

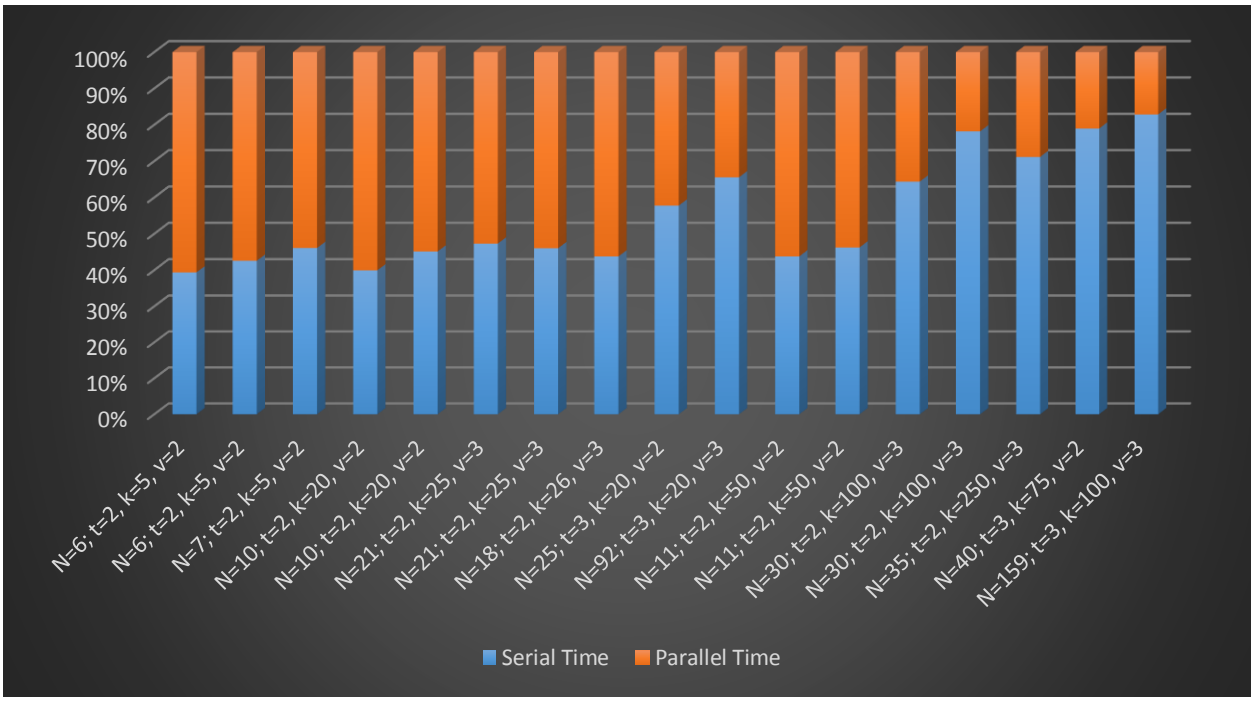





## Running on ASURE: num_threads(16)

| Inputs | k choose t | Combination | Serial Time | Parallel Time | SpeedUP |
|---|---|---|---|---|---|
| N=6; t=2, k=5, v=2 | 10 | 4 | 0.000551 | 0.000849 | 0.648999 |
| N=6; t=2, k=5, v=2 | 10 | 4 | 0.000441 | 0.000595 | 0.741176 |
| N=7; t=2, k=5, v=2 | 10 | 4 | 0.000458 | 0.000536 | 0.854478 |
| N=10; t=2, k=20, v=2 | 190 | 4 | 0.000534 | 0.000804 | 0.664179 |
| N=10; t=2, k=20, v=2 | 190 | 4 | 0.000546 | 0.000664 | 0.822289 |
| N=21; t=2, k=25, v=3 | 300 | 9 | 0.001066 | 0.001188 | 0.897306 |
| N=21; t=2, k=25, v=3 | 300 | 9 | 0.0007 | 0.000821 | 0.852619 |
| N=18; t=2, k=26, v=3 | 325 | 9 | 0.001094 | 0.001406 | 0.778094 |
| N=25; t=3, k=20, v=2 | 1140 | 8 | 0.002487 | 0.001819 | 1.367235 |
| N=92; t=3, k=20, v=3 | 1140 | 27 | 0.019336 | 0.010167 | 1.901839 |
| N=11; t=2, k=50, v=2 | 1225 | 4 | 0.001042 | 0.001339 | 0.778193 |
| N=11; t=2, k=50, v=2 | 1225 | 4 | 0.000907 | 0.001055 | 0.859716 |
| N=30; t=2, k=100, v=3 | 4950 | 9 | 0.010981 | 0.006084 | 1.804898 |
| N=30; t=2, k=100, v=3 | 4950 | 9 | 0.008596 | 0.002399 | 3.58316 |
| N=35; t=2, k=250, v=3 | 31125 | 9 | 0.063768 | 0.025894 | 2.462655 |
| N=40; t=3, k=75, v=2 | 67525 | 8 | 0.149921 | 0.039964 | 3.751401 |
| N=159; t=3, k=100, v=3 | 161700 | 27 | 3.345403 | 0.693118 | 4.8266 |




## Implementation: Serial Code

```
#define _CRT_SECURE_NO_WARNINGS
//#define DEBUG
#define CONSOLE

#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <string.h>
#include <time.h>

#define OUTPUT_FILE_PATH "Output_1207543476"
#define INPUT_FILE_MODE "r"
#define OUTPUT_FILE_MODE "a+"

FILE *ofp;

/* Return 1 if the difference is negative, otherwise 0.  */
int timeval_subtract(struct timeval *result, struct timeval *t2, struct timeval *t1)
{
  long int diff = (t2->tv_usec + 1000000 * t2->tv_sec) - (t1->tv_usec + 1000000 * t1->tv_sec);
  result->tv_sec = diff / 1000000;
  result->tv_usec = diff % 1000000;

  return (diff<0);
}

void printMatrix(char *arr, int row, int column)
{
	int i = 0, j = 0;
	for (i = 0; i < row; i++)
	{
		for (j = 0; j < column; j++)
		{
			if (arr[(i*column) + j] < '9')
			{
				//fprintf(ofp, "%c\t", arr[(i*column) + j]);
#ifdef CONSOLE
				printf("%c\t", arr[(i*column) + j]);
#endif
			}
			else
			{
				//fprintf(ofp, "%d\t", ((int)arr[(i*column) + j])-48);
#ifdef CONSOLE
				printf("%d\t", ((int)arr[(i*column) + j])-48);
#endif
			}
		}
		//fprintf(ofp, "\n");
#ifdef CONSOLE
		printf("\n");
#endif
	}
}

int factorial(int i)
{
	if (1 == i)
	{
		return 1;
	}
	return factorial(i - 1) * i;
}

int combination(int n, int r)
{
	int num = 1;
	int i = 0;
	for (i = n; i>(n - r); i--)
	{
		num = num * i;
	}
```

```
	return num / factorial(r);
}

int main(int argc, char **argv)
{
	int N, t, k, v;

	struct timeval timeBegin, timeEnd, timeDiff;
	gettimeofday(&timeBegin, NULL);

	ofp = fopen(OUTPUT_FILE_PATH, OUTPUT_FILE_MODE);
	if (NULL == ofp)
	{
		fprintf(stderr, "Opening output file failed.\n");
#ifdef CONSOLE
		printf("Opening output file failed.\n");
#endif
		exit(0);
	}
	scanf("%d %d %d %d", &N, &t, &k, &v);
	//fprintf(ofp, "=====================================\n");
	//fprintf(ofp, "Serial ::: N=%d; t=%d, k=%d, v=%d\n", N, t, k, v);
	//fprintf(ofp, "=====================================\n");
#ifdef CONSOLE
	printf("=====================================\n");
	printf("Serial ::: N=%d; t=%d, k=%d, v=%d\n", N,t,k,v);
	printf("=====================================\n");
#endif

	char *inputArray = (char *)calloc(N * k, sizeof(char));

	int pos = 0;
	char c = '0';
	while (EOF != c)
	{
		c = getchar();
		if (' ' != c)
		{
			if ('\n' != c)
			{
				inputArray[pos++] = c;
			}
		}
	}
#ifdef DEBUG
	fprintf(ofp,"Input Matrix\n");
	printMatrix(&inputArray[0], N, k);
#endif
	int combinationOfSymbols = (int)pow(v, t);
#ifdef DEBUG
	fprintf(ofp,"Total Combination Of Symbols = %d\n", combinationOfSymbols);
#endif
	char *elements = (char *)calloc(v, sizeof(char));
	char *Symbols = (char *)calloc(combinationOfSymbols * t, sizeof(char));
	int iLoopIndex = 0, jLoopIndex = 0, kLoopIndex = 0, lLoopIndex = 0;
	if (2 == v)
	{
		int iCount = 0;
		elements[0] = '0';
		elements[1] = '1';
		if (2 == t)
		{
			for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
			{
				for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
				{
```

```
                                        for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                                        {
                                                if (0 == (kLoopIndex % t))
                                                {
                                                        Symbols[iCount] = elements[iLoopIndex];
                                                }
                                                else
                                                {
                                                        Symbols[iCount] = elements[jLoopIndex];
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
                else if (3 == t)
                {
                        elements[0] = '0';
                        elements[1] = '1';
                        elements[2] = '2';
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < v; kLoopIndex++)
                                        {
                                                for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
                                                {
                                                        if (0 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[iLoopIndex];
                                                        }
                                                        else if (1 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[jLoopIndex];
                                                        }
                                                        else if (2 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[kLoopIndex];
                                                        }
                                                        iCount++;
                                                }
                                        }
                                }
                        }
                }
        }
        else if (3 == v)
        {
                int iCount = 0;
                elements[0] = '0';
                elements[1] = '1';
                elements[2] = '2';
                if (2 == t)
                {
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                                        {
                                                if (0 == (kLoopIndex % t))
                                                {
                                                        Symbols[iCount] = elements[iLoopIndex];
                                                }
                                                else
```

```
                                                {
                                                        Symbols[iCount] = elements[jLoopIndex];
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
                else if (3 == t)
                {
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < v; kLoopIndex++)
                                        {
                                                for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
                                                {
                                                        if (0 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[iLoopIndex];
                                                        }
                                                        else if (1 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[jLoopIndex];
                                                        }
                                                        else if (2 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[kLoopIndex];
                                                        }
                                                        iCount++;
                                                }
                                        }
                                }
                        }
                }
        }
        int iCount = 0;
        int i = 0;
#ifdef DEBUG
        fprintf(ofp,"Symbols are\n");
        for (i = 0; i<combinationOfSymbols * t; i++)
        {
                if (0 == (iCount % t))
                {
                        fprintf(ofp, "%d-->\t", (i / t + 1));
                }
                iCount++;
                fprintf(ofp, "%c", Symbols[i]);
                if (0 == (iCount % t))
                {
                        fprintf(ofp, "\n");
                }
        }
#endif
        int noOfSubArrays = combination(k, t);
#ifdef DEBUG
        fprintf(ofp, "Total Number of Sub Arrays = %d\n", noOfSubArrays);
#endif
        int *SubArrays = (int *)calloc(noOfSubArrays * t, sizeof(int));
        if (2 == t)
        {
                int iCount = 0;
                for (iLoopIndex = 1; iLoopIndex <= k; iLoopIndex++)
                {
                        for (jLoopIndex = iLoopIndex + 1; jLoopIndex <= k; jLoopIndex++)
```

```
                {
                        for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                        {
                                if (0 == (iCount % t))
                                {
                                        SubArrays[iCount] = iLoopIndex;
                                }
                                else
                                {
                                        SubArrays[iCount] = jLoopIndex;
                                }
                                iCount++;
                        }
                }
        }
        else if (3 == t)
        {
                int iCount = 0;
                for (iLoopIndex = 1; iLoopIndex <= k; iLoopIndex++)
                {
                        for (jLoopIndex = iLoopIndex + 1; jLoopIndex <= k; jLoopIndex++)
                        {
                                for (kLoopIndex = jLoopIndex + 1; kLoopIndex <= k; kLoopIndex++)
                                {
                                        for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
                                        {
                                                if (0 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = iLoopIndex;
                                                }
                                                else if (1 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = jLoopIndex;
                                                }
                                                else if (2 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = kLoopIndex;
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
        }
        iCount = 0;
#ifdef DEBUG
        fprintf(ofp,"SubArrays are\n");
        for (i = 0; i<(noOfSubArrays * t); i++)
        {
                if (0 == (iCount % t))
                {
                        fprintf(ofp, "%d-->\t(", (i / t + 1));
                }
                fprintf(ofp, "%d,", SubArrays[i]);
                iCount++;
                if (0 == (iCount % t))
                {
                        fprintf(ofp, ")\n", (i / t + 1));
                }
        }
#endif
        int *intermediateArray = (int *)calloc(noOfSubArrays * combinationOfSymbols, sizeof(int));
        for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
        {
                int firstColumnIndex = SubArrays[t * iLoopIndex];
```

```
                int secondColumnIndex = SubArrays[t * iLoopIndex + 1];
                int thirdColumnIndex = SubArrays[t * iLoopIndex + 2];
                char *tempInputArray = (char *)calloc(N*t,sizeof(char));
                for (jLoopIndex = 0; jLoopIndex < N; jLoopIndex++)
                {
                        if (2 == t)
                        {
                                tempInputArray[t*jLoopIndex] = inputArray[k*jLoopIndex + firstColumnIndex - 1] ;
                                tempInputArray[t*jLoopIndex + 1] = inputArray[k*jLoopIndex + secondColumnIndex - 1];
                                for (kLoopIndex = 0; kLoopIndex < combinationOfSymbols; kLoopIndex++)
                                {
                                        if ( (tempInputArray[t * jLoopIndex] == Symbols[t*kLoopIndex]) && (tempInputArray[t *
jLoopIndex + 1] == Symbols[t*kLoopIndex + 1]))
                                        {
                                                intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex]++;
                                        }
                                }
                        }
                        else if (3 == t)
                        {
                                tempInputArray[t*jLoopIndex] = inputArray[k*jLoopIndex + firstColumnIndex - 1];
                                tempInputArray[t*jLoopIndex + 1] = inputArray[k*jLoopIndex + secondColumnIndex - 1];
                                tempInputArray[t*jLoopIndex + 2] = inputArray[k*jLoopIndex + thirdColumnIndex - 1];
                                for (kLoopIndex = 0; kLoopIndex < combinationOfSymbols; kLoopIndex++)
                                {
                                        if ((tempInputArray[t*jLoopIndex] == Symbols[t*kLoopIndex]) &&
(tempInputArray[t*jLoopIndex + 1] == Symbols[t*kLoopIndex + 1]) && (tempInputArray[t*jLoopIndex + 2] == Symbols[t*kLoopIndex + 2]))
                                        {
                                                intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex]++;
                                        }
                                }
                        }
                }
                free(tempInputArray);
        }
        char isCoveringArray = '1';
#ifdef DEBUG
        fprintf(ofp, "Intermediate Matrix\n");
#endif
        for (iLoopIndex = 0; iLoopIndex < noOfSubArrays ; iLoopIndex++)
        {
                for (jLoopIndex = 0; jLoopIndex < combinationOfSymbols; jLoopIndex++)
                {
#ifdef DEBUG
                        fprintf(ofp, "%d\t", intermediateArray[combinationOfSymbols * iLoopIndex + jLoopIndex]);
#endif
                        if (0 == intermediateArray[combinationOfSymbols * iLoopIndex + jLoopIndex])
                        {
                                isCoveringArray = '0';
                        }
                }
#ifdef DEBUG
                fprintf(ofp, "\n");
#endif
        }
        if ('0' == isCoveringArray)
        {
                //fprintf(ofp, "Serial::: No, It is Not a Covering Array\n");
#ifdef CONSOLE
                printf("Serial::: No, It is Not a Covering Array\n");
#endif
        }
        else
        {
                //fprintf(ofp, "Serial::: Yes, It is a Covering Array\n");
#ifdef CONSOLE
```

```
                printf("Serial::: Yes, It is a Covering Array\n");
#endif
        }

        if ('1' == isCoveringArray)
        {
                int firstColumnIndex = 0;
                int secondColumnIndex = 0;
                int thirdColumnIndex = 0;

                char *finalArray = (char *)calloc(N*k, sizeof(char));
                memset(finalArray, '0', (N*k*sizeof(char)));
                for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
                {
                        for (jLoopIndex = 0; jLoopIndex < combinationOfSymbols; jLoopIndex++)
                        {
                                int positionOfOnes = 0;
                                int p1_row = 0;
                                int p1_column = 0;
                                if (1 == intermediateArray[combinationOfSymbols * iLoopIndex + jLoopIndex])
                                {
                                        if (2 == t)
                                        {
                                                positionOfOnes = combinationOfSymbols * iLoopIndex + jLoopIndex;
                                                p1_row = iLoopIndex;
                                                p1_column = jLoopIndex;
#ifdef DEBUG
                                                fprintf(ofp, "%d(%d,%d),", positionOfOnes, p1_row, p1_column);
                                                fprintf(ofp, "{%d,%d}", SubArrays[t*iLoopIndex], SubArrays[t*iLoopIndex +
1]);
                                                fprintf(ofp, "{%c,%c}\n", Symbols[t*jLoopIndex], Symbols[t*jLoopIndex + 1]);
#endif
                                                firstColumnIndex = 0;
                                                secondColumnIndex = 0;
                                                char *tempInputArray = (char *)calloc(N*t, sizeof(char));
                                                int jLoopIndex_tempInputArray = 0;
                                                int tempIndex = 0;
                                                for (jLoopIndex_tempInputArray = 0; jLoopIndex_tempInputArray < N;
jLoopIndex_tempInputArray++)
                                                {
                                                        firstColumnIndex = SubArrays[t * iLoopIndex];
                                                        secondColumnIndex = SubArrays[t * iLoopIndex + 1];
                                                        tempInputArray[t*jLoopIndex_tempInputArray] =
inputArray[k*jLoopIndex_tempInputArray + firstColumnIndex - 1];
                                                        tempInputArray[t*jLoopIndex_tempInputArray + 1] =
inputArray[k*jLoopIndex_tempInputArray + secondColumnIndex - 1];
                                                }
                                                for (tempIndex = 0; tempIndex < N; tempIndex++)
                                                {
#ifdef DEBUG
                                                        fprintf(ofp, "%c%c==%c%c\t", Symbols[t*jLoopIndex],
Symbols[t*jLoopIndex + 1], tempInputArray[t*tempIndex], tempInputArray[t*tempIndex + 1]);
#endif
                                                        if ((tempInputArray[t*tempIndex] == Symbols[t*jLoopIndex]) &&
(tempInputArray[t*tempIndex + 1] == Symbols[t*jLoopIndex + 1]))
                                                        {
#ifdef DEBUG
                                                                fprintf(ofp, "Found at Row %d", tempIndex + 1);
#endif
                                                                finalArray[k*tempIndex + firstColumnIndex - 1]++;
                                                                finalArray[k*tempIndex + secondColumnIndex - 1]++;
                                                        }
                                                        else
                                                        {
#ifdef DEBUG
                                                                fprintf(ofp, "Not Found at Row %d", tempIndex + 1);
```

```
#endif
                                                                }
#ifdef DEBUG
                                                                fprintf(ofp, "\n");
#endif
                                                        }
#ifdef DEBUG
                                                        printMatrix(finalArray, N, k);
#endif
                                                        free(tempInputArray);
                                                }
                                                else if (3 == t)
                                                {
                                                        positionOfOnes = combinationOfSymbols * iLoopIndex + jLoopIndex;
                                                        p1_row = iLoopIndex;
                                                        p1_column = jLoopIndex;
#ifdef DEBUG
                                                        fprintf(ofp, "%d(%d,%d),", positionOfOnes, p1_row, p1_column);
                                                        fprintf(ofp, "{%d,%d,%d}", SubArrays[t*iLoopIndex], SubArrays[t*iLoopIndex
+ 1], SubArrays[t*iLoopIndex + 2]);
                                                        fprintf(ofp, "{%c,%c,%c}\n", Symbols[t*jLoopIndex], Symbols[t*jLoopIndex +
1], Symbols[t*jLoopIndex + 2]);
#endif
                                                        firstColumnIndex = 0;
                                                        secondColumnIndex = 0;
                                                        thirdColumnIndex = 0;
                                                        char *tempInputArray = (char *)calloc(N*t, sizeof(char));
                                                        int jLoopIndex_tempInputArray = 0;
                                                        int tempIndex = 0;
                                                        for (jLoopIndex_tempInputArray = 0; jLoopIndex_tempInputArray < N;
jLoopIndex_tempInputArray++)
                                                        {
                                                                firstColumnIndex = SubArrays[t * iLoopIndex];
                                                                secondColumnIndex = SubArrays[t * iLoopIndex + 1];
                                                                thirdColumnIndex = SubArrays[t * iLoopIndex + 2];
                                                                tempInputArray[t*jLoopIndex_tempInputArray] =
inputArray[k*jLoopIndex_tempInputArray + firstColumnIndex - 1];
                                                                tempInputArray[t*jLoopIndex_tempInputArray + 1] =
inputArray[k*jLoopIndex_tempInputArray + secondColumnIndex - 1];
                                                                tempInputArray[t*jLoopIndex_tempInputArray + 2] =
inputArray[k*jLoopIndex_tempInputArray + thirdColumnIndex - 1];
                                                        }
                                                        for (tempIndex = 0; tempIndex < N; tempIndex++)
                                                        {
#ifdef DEBUG
                                                                fprintf(ofp, "%c%c%c==%c%c%c\t", Symbols[t*jLoopIndex],
Symbols[t*jLoopIndex + 1], Symbols[t*jLoopIndex + 2], tempInputArray[t*tempIndex], tempInputArray[t*tempIndex + 1],
tempInputArray[t*tempIndex + 2]);
#endif
                                                                if ((tempInputArray[t*tempIndex] == Symbols[t*jLoopIndex]) &&
(tempInputArray[t*tempIndex + 1] == Symbols[t*jLoopIndex + 1]) && (tempInputArray[t*tempIndex + 2] == Symbols[t*jLoopIndex + 2]))
                                                                {
#ifdef DEBUG
                                                                        fprintf(ofp, "Found at Row %d", tempIndex + 1);
#endif
                                                                        finalArray[k*tempIndex + firstColumnIndex - 1]++;
                                                                        finalArray[k*tempIndex + secondColumnIndex - 1]++;
                                                                        finalArray[k*tempIndex + thirdColumnIndex - 1]++;
                                                                }
                                                                else
                                                                {
#ifdef DEBUG
                                                                        fprintf(ofp, "Not Found at Row %d", tempIndex + 1);
#endif
                                                                }
#ifdef DEBUG
```

```
                                                                fprintf(ofp, "\n");
#endif
                                                        }
#ifdef DEBUG
                                                        printMatrix(finalArray, N, k);
#endif
                                                        free(tempInputArray);
                                                }
                                        }
                                }
                        }
                        //Converting 0 to * for printing in the final Matrix
                        for (iLoopIndex = 0; iLoopIndex < N; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < k; jLoopIndex++)
                                {
                                        if ('0' == finalArray[k*iLoopIndex + jLoopIndex])
                                        {
                                                finalArray[k*iLoopIndex + jLoopIndex] = '*';
                                        }
                                        else
                                        {
                                                finalArray[k*iLoopIndex + jLoopIndex] = inputArray[k*iLoopIndex + jLoopIndex];
                                        }
                                }
                        }
                        //fprintf(ofp, "Don't Care Matrix\n");
#ifdef CONSOLE
                        printf("Don't Care Matrix\n");
#endif

                        printMatrix(finalArray, N, k);

                        //Printing the co-ordinates of Don't Care positions
                        //fprintf(ofp, "Don't Care position\n");
#ifdef CONSOLE
                        printf("Don't Care position\n");
#endif
                        iCount = 0;
                        for (iLoopIndex = 0; iLoopIndex < N; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < k; jLoopIndex++)
                                {
                                        if ('*' == finalArray[k*iLoopIndex + jLoopIndex])
                                        {
                                                iCount++;
                                                //fprintf(ofp, "(%d,%d)\t", iLoopIndex + 1, jLoopIndex + 1);
#ifdef CONSOLE
                                                printf("(%d,%d)\t", iLoopIndex + 1, jLoopIndex + 1);
#endif
                                        }
                                }
                        }
                        //fprintf(ofp,"\nTotal Number of Don't Cares :: %d\n",iCount);
#ifdef CONSOLE
                        printf("\nTotal Number of Don't Cares :: %d\n",iCount);
#endif
                        free(finalArray);
                }

                free(inputArray);
                free(elements);
                free(Symbols);
                free(SubArrays);
                free(intermediateArray);
```

```
	gettimeofday(&timeEnd, NULL);

	timeval_subtract(&timeDiff, &timeEnd, &timeBegin);

#ifdef CONSOLE
	printf("Serial Time:::%ld.%06ld secs\n", timeDiff.tv_sec, timeDiff.tv_usec);
#endif
	fprintf(ofp, "%ld.%06ld\t", timeDiff.tv_sec, timeDiff.tv_usec);

	fclose(ofp);
	return 0;
}
```

## Implementation: Parallel Code

```
#define _CRT_SECURE_NO_WARNINGS
//#define DEBUG
#define CONSOLE


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <omp.h>


#define OUTPUT_FILE_PATH "Output_1207543476"
#define INPUT_FILE_MODE "r"
#define OUTPUT_FILE_MODE "a+"

FILE *ofp;

void printMatrix(char *arr, int row, int column)
{
	int i = 0, j = 0;
	for (i = 0; i < row; i++)
	{
		for (j = 0; j < column; j++)
		{
			if (arr[(i*column) + j] < '9')
			{
				//fprintf(ofp, "%c\t", arr[(i*column) + j]);
#ifdef CONSOLE
				printf("%c\t", arr[(i*column) + j]);
#endif
			}
			else
			{
				//fprintf(ofp, "%d\t", ((int)arr[(i*column) + j]) - 48);
#ifdef CONSOLE
				printf("%d\t", ((int)arr[(i*column) + j]) - 48);
#endif
			}
		}
		//fprintf(ofp, "\n");
#ifdef CONSOLE
		printf("\n");
#endif
	}
}

int factorial(int i)
```

```
{
	if (1 == i)
	{
		return 1;
	}
	return factorial(i - 1) * i;
}

int combination(int n, int r)
{
	int num = 1;
	int i = 0;
	for (i = n; i>(n - r); i--)
	{
		num = num * i;
	}
	return num / factorial(r);
}

int main(int argc, char **argv)
{
	int N, t, k, v;

	double before, after;
	before = omp_get_wtime();

	ofp = fopen(OUTPUT_FILE_PATH, OUTPUT_FILE_MODE);
	if (NULL == ofp)
	{
		fprintf(stderr, "Opening output file failed.\n");
#ifdef CONSOLE
		printf("Opening output file failed.\n");
#endif
		exit(0);
	}
	scanf("%d %d %d %d", &N, &t, &k, &v);
	//fprintf(ofp, "=======================================\n");
	//fprintf(ofp, "Parallel ::: N=%d; t=%d, k=%d, v=%d\n", N, t, k, v);
	//fprintf(ofp, "=======================================\n");
#ifdef CONSOLE
	printf("=======================================\n");
	printf("Parallel ::: N=%d; t=%d, k=%d, v=%d\n", N, t, k, v);
	printf("=======================================\n");
#endif

	char *inputArray = (char *)calloc(N * k, sizeof(char));

	int pos = 0;
	char c = '0';
	while (EOF != c)
	{
		c = getchar();
		if (' ' != c)
		{
			if ('\n' != c)
			{
				inputArray[pos++] = c;
			}
		}
	}
#ifdef DEBUG
	fprintf(ofp, "Input Matrix\n");
	printMatrix(&inputArray[0], N, k);
#endif
	int combinationOfSymbols = (int)pow(v, t);
#ifdef DEBUG
```

```
	fprintf(ofp, "Total Combination Of Symbols = %d\n", combinationOfSymbols);
#endif
	char *elements = (char *)calloc(v, sizeof(char));
	char *Symbols = (char *)calloc(combinationOfSymbols * t, sizeof(char));
	int iLoopIndex = 0, jLoopIndex = 0, kLoopIndex = 0, lLoopIndex = 0;
	if (2 == v)
	{
		int iCount = 0;
		elements[0] = '0';
		elements[1] = '1';
		if (2 == t)
		{
			for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
			{
				for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
				{
					for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
					{
						if (0 == (kLoopIndex % t))
						{
							Symbols[iCount] = elements[iLoopIndex];
						}
						else
						{
							Symbols[iCount] = elements[jLoopIndex];
						}
						iCount++;
					}
				}
			}
		}
		else if (3 == t)
		{
			elements[0] = '0';
			elements[1] = '1';
			elements[2] = '2';
			for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
			{
				for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
				{
					for (kLoopIndex = 0; kLoopIndex < v; kLoopIndex++)
					{
						for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
						{
							if (0 == (lLoopIndex % t))
							{
								Symbols[iCount] = elements[iLoopIndex];
							}
							else if (1 == (lLoopIndex % t))
							{
								Symbols[iCount] = elements[jLoopIndex];
							}
							else if (2 == (lLoopIndex % t))
							{
								Symbols[iCount] = elements[kLoopIndex];
							}
							iCount++;
						}
					}
				}
			}
		}
	}
	else if (3 == v)
	{
		int iCount = 0;
```

```
                elements[0] = '0';
                elements[1] = '1';
                elements[2] = '2';
                if (2 == t)
                {
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                                        {
                                                if (0 == (kLoopIndex % t))
                                                {
                                                        Symbols[iCount] = elements[iLoopIndex];
                                                }
                                                else
                                                {
                                                        Symbols[iCount] = elements[jLoopIndex];
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
                else if (3 == t)
                {
                        for (iLoopIndex = 0; iLoopIndex < v; iLoopIndex++)
                        {
                                for (jLoopIndex = 0; jLoopIndex < v; jLoopIndex++)
                                {
                                        for (kLoopIndex = 0; kLoopIndex < v; kLoopIndex++)
                                        {
                                                for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
                                                {
                                                        if (0 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[iLoopIndex];
                                                        }
                                                        else if (1 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[jLoopIndex];
                                                        }
                                                        else if (2 == (lLoopIndex % t))
                                                        {
                                                                Symbols[iCount] = elements[kLoopIndex];
                                                        }
                                                        iCount++;
                                                }
                                        }
                                }
                        }
                }
        }
        int iCount = 0;
        int i = 0;
#ifdef DEBUG
        fprintf(ofp, "Symbols are\n");
        for (i = 0; i<combinationOfSymbols * t; i++)
        {
                if (0 == (iCount % t))
                {
                        fprintf(ofp, "%d-->\t", (i / t + 1));
                }
                iCount++;
                fprintf(ofp, "%c", Symbols[i]);
                if (0 == (iCount % t))
```

```
                {
                        fprintf(ofp, "\n");
                }
        }
#endif
        int noOfSubArrays = combination(k, t);
#ifdef DEBUG
        fprintf(ofp, "Total Number of Sub Arrays = %d\n", noOfSubArrays);
#endif
        int *SubArrays = (int *)calloc(noOfSubArrays * t, sizeof(int));
        if (2 == t)
        {
                int iCount = 0;
                for (iLoopIndex = 1; iLoopIndex <= k; iLoopIndex++)
                {
                        for (jLoopIndex = iLoopIndex + 1; jLoopIndex <= k; jLoopIndex++)
                        {
                                for (kLoopIndex = 0; kLoopIndex < t; kLoopIndex++)
                                {
                                        if (0 == (iCount % t))
                                        {
                                                SubArrays[iCount] = iLoopIndex;
                                        }
                                        else
                                        {
                                                SubArrays[iCount] = jLoopIndex;
                                        }
                                        iCount++;
                                }
                        }
                }
        }
        else if (3 == t)
        {
                int iCount = 0;
                for (iLoopIndex = 1; iLoopIndex <= k; iLoopIndex++)
                {
                        for (jLoopIndex = iLoopIndex + 1; jLoopIndex <= k; jLoopIndex++)
                        {
                                for (kLoopIndex = jLoopIndex + 1; kLoopIndex <= k; kLoopIndex++)
                                {
                                        for (lLoopIndex = 0; lLoopIndex < t; lLoopIndex++)
                                        {
                                                if (0 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = iLoopIndex;
                                                }
                                                else if (1 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = jLoopIndex;
                                                }
                                                else if (2 == (iCount % t))
                                                {
                                                        SubArrays[iCount] = kLoopIndex;
                                                }
                                                iCount++;
                                        }
                                }
                        }
                }
        }
        iCount = 0;
#ifdef DEBUG
        fprintf(ofp, "SubArrays are\n");
        for (i = 0; i<(noOfSubArrays * t); i++)
        {
```

```
                if (0 == (iCount % t))
                {
                        fprintf(ofp, "%d-->\t(", (i / t + 1));
                }
                fprintf(ofp, "%d,", SubArrays[i]);
                iCount++;
                if (0 == (iCount % t))
                {
                        fprintf(ofp, ")\n", (i / t + 1));
                }
        }
#endif
        int *intermediateArray = (int *)calloc(noOfSubArrays * combinationOfSymbols, sizeof(int));

#pragma omp parallel  num_threads(16) shared(noOfSubArrays, SubArrays, t, N, inputArray, combinationOfSymbols, Symbols,
intermediateArray)
        {
#pragma omp for  private(iLoopIndex, jLoopIndex, kLoopIndex) schedule(dynamic, 50)
                for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
                {
                        int firstColumnIndex = SubArrays[t * iLoopIndex];
                        int secondColumnIndex = SubArrays[t * iLoopIndex + 1];
                        int thirdColumnIndex = SubArrays[t * iLoopIndex + 2];
                        char *tempInputArray = (char *)calloc(N*t, sizeof(char));
                        for (jLoopIndex = 0; jLoopIndex < N; jLoopIndex++)
                        {
                                if (2 == t)
                                {
                                        tempInputArray[t*jLoopIndex] = inputArray[k*jLoopIndex + firstColumnIndex - 1];
                                        tempInputArray[t*jLoopIndex + 1] = inputArray[k*jLoopIndex + secondColumnIndex - 1];
                                        for (kLoopIndex = 0; kLoopIndex < combinationOfSymbols; kLoopIndex++)
                                        {
                                                if ((tempInputArray[t * jLoopIndex] == Symbols[t*kLoopIndex]) &&
(tempInputArray[t * jLoopIndex + 1] == Symbols[t*kLoopIndex + 1]))
                                                {

        __sync_fetch_and_add(&intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex], 1);
                                                }
                                        }
                                }
                                else if (3 == t)
                                {
                                        tempInputArray[t*jLoopIndex] = inputArray[k*jLoopIndex + firstColumnIndex - 1];
                                        tempInputArray[t*jLoopIndex + 1] = inputArray[k*jLoopIndex + secondColumnIndex - 1];
                                        tempInputArray[t*jLoopIndex + 2] = inputArray[k*jLoopIndex + thirdColumnIndex - 1];
                                        for (kLoopIndex = 0; kLoopIndex < combinationOfSymbols; kLoopIndex++)
                                        {
                                                if ((tempInputArray[t*jLoopIndex] == Symbols[t*kLoopIndex]) &&
(tempInputArray[t*jLoopIndex + 1] == Symbols[t*kLoopIndex + 1]) && (tempInputArray[t*jLoopIndex + 2] == Symbols[t*kLoopIndex + 2]))
                                                {

        __sync_fetch_and_add(&intermediateArray[(combinationOfSymbols * iLoopIndex) + kLoopIndex], 1);
                                                }
                                        }
                                }
                        }
                        free(tempInputArray);
                }
        }
        char isCoveringArray = '1';
#ifdef DEBUG
        fprintf(ofp, "Intermediate Matrix\n");
#endif
        for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
        {
                for (jLoopIndex = 0; jLoopIndex < combinationOfSymbols; jLoopIndex++)
```

```
                {
#ifdef DEBUG
                        fprintf(ofp, "%d\t", intermediateArray[combinationOfSymbols * iLoopIndex + jLoopIndex]);
#endif
                        if (0 == intermediateArray[combinationOfSymbols * iLoopIndex + jLoopIndex])
                        {
                                isCoveringArray = '0';
                        }
                }
#ifdef DEBUG
                fprintf(ofp, "\n");
#endif
        }
        if ('0' == isCoveringArray)
        {
                //fprintf(ofp, "Parallel::: No, It is Not a Covering Array\n");
#ifdef CONSOLE
                printf("Parallel::: No, It is Not a Covering Array\n");
#endif
        }
        else
        {
                //fprintf(ofp, "Parallel::: Yes, It is a Covering Array\n");
#ifdef CONSOLE
                printf("Parallel::: Yes, It is a Covering Array\n");
#endif
        }

        if ('1' == isCoveringArray)
        {
                int firstColumnIndex = 0;
                int secondColumnIndex = 0;
                int thirdColumnIndex = 0;

                char *finalArray = (char *)calloc(N*k, sizeof(char));
                memset(finalArray, '0', (N*k*sizeof(char)));
                for (iLoopIndex = 0; iLoopIndex < noOfSubArrays; iLoopIndex++)
                {
                        for (jLoopIndex = 0; jLoopIndex < combinationOfSymbols; jLoopIndex++)
                        {
                                int positionOfOnes = 0;
                                int p1_row = 0;
                                int p1_column = 0;
                                if (1 == intermediateArray[combinationOfSymbols * iLoopIndex + jLoopIndex])
                                {
                                        if (2 == t)
                                        {
                                                positionOfOnes = combinationOfSymbols * iLoopIndex + jLoopIndex;
                                                p1_row = iLoopIndex;
                                                p1_column = jLoopIndex;
#ifdef DEBUG
                                                fprintf(ofp, "%d(%d,%d),", positionOfOnes, p1_row, p1_column);
                                                fprintf(ofp, "{%d,%d}", SubArrays[t*iLoopIndex], SubArrays[t*iLoopIndex +
1]);
                                                fprintf(ofp, "{%c,%c}\n", Symbols[t*jLoopIndex], Symbols[t*jLoopIndex + 1]);
#endif
                                                firstColumnIndex = 0;
                                                secondColumnIndex = 0;
                                                char *tempInputArray = (char *)calloc(N*t, sizeof(char));
                                                int jLoopIndex_tempInputArray = 0;
                                                int tempIndex = 0;
                                                for (jLoopIndex_tempInputArray = 0; jLoopIndex_tempInputArray < N;
jLoopIndex_tempInputArray++)
                                                {
                                                        firstColumnIndex = SubArrays[t * iLoopIndex];
                                                        secondColumnIndex = SubArrays[t * iLoopIndex + 1];
```

```
                                                tempInputArray[t*jLoopIndex_tempInputArray] =
inputArray[k*jLoopIndex_tempInputArray + firstColumnIndex - 1];
                                                tempInputArray[t*jLoopIndex_tempInputArray + 1] =
inputArray[k*jLoopIndex_tempInputArray + secondColumnIndex - 1];
                                        }
                                        for (tempIndex = 0; tempIndex < N; tempIndex++)
                                        {
#ifdef DEBUG
                                                fprintf(ofp, "%c%c==%c%c\t", Symbols[t*jLoopIndex],
Symbols[t*jLoopIndex + 1], tempInputArray[t*tempIndex], tempInputArray[t*tempIndex + 1]);
#endif
                                                if ((tempInputArray[t*tempIndex] == Symbols[t*jLoopIndex]) &&
(tempInputArray[t*tempIndex + 1] == Symbols[t*jLoopIndex + 1]))
                                                {
#ifdef DEBUG
                                                        fprintf(ofp, "Found at Row %d", tempIndex + 1);
#endif
                                                        finalArray[k*tempIndex + firstColumnIndex - 1]++;
                                                        finalArray[k*tempIndex + secondColumnIndex - 1]++;
                                                }
                                                else
                                                {
#ifdef DEBUG
                                                        fprintf(ofp, "Not Found at Row %d", tempIndex + 1);
#endif
                                                }
#ifdef DEBUG
                                                fprintf(ofp, "\n");
#endif
                                        }
#ifdef DEBUG
                                        printMatrix(finalArray, N, k);
#endif
                                        free(tempInputArray);
                                }
                                else if (3 == t)
                                {
                                        positionOfOnes = combinationOfSymbols * iLoopIndex + jLoopIndex;
                                        p1_row = iLoopIndex;
                                        p1_column = jLoopIndex;
#ifdef DEBUG
                                        fprintf(ofp, "%d(%d,%d),", positionOfOnes, p1_row, p1_column);
                                        fprintf(ofp, "{%d,%d,%d}", SubArrays[t*iLoopIndex], SubArrays[t*iLoopIndex
+ 1], SubArrays[t*iLoopIndex + 2]);
                                        fprintf(ofp, "{%c,%c,%c}\n", Symbols[t*jLoopIndex], Symbols[t*jLoopIndex +
1], Symbols[t*jLoopIndex + 2]);
#endif
                                        firstColumnIndex = 0;
                                        secondColumnIndex = 0;
                                        thirdColumnIndex = 0;
                                        char *tempInputArray = (char *)calloc(N*t, sizeof(char));
                                        int jLoopIndex_tempInputArray = 0;
                                        int tempIndex = 0;
                                        for (jLoopIndex_tempInputArray = 0; jLoopIndex_tempInputArray < N;
jLoopIndex_tempInputArray++)
                                        {
                                                firstColumnIndex = SubArrays[t * iLoopIndex];
                                                secondColumnIndex = SubArrays[t * iLoopIndex + 1];
                                                thirdColumnIndex = SubArrays[t * iLoopIndex + 2];
                                                tempInputArray[t*jLoopIndex_tempInputArray] =
inputArray[k*jLoopIndex_tempInputArray + firstColumnIndex - 1];
                                                tempInputArray[t*jLoopIndex_tempInputArray + 1] =
inputArray[k*jLoopIndex_tempInputArray + secondColumnIndex - 1];
                                                tempInputArray[t*jLoopIndex_tempInputArray + 2] =
inputArray[k*jLoopIndex_tempInputArray + thirdColumnIndex - 1];
                                        }
```

```
                                                for (tempIndex = 0; tempIndex < N; tempIndex++)
                                                {
#ifdef DEBUG
                                                        fprintf(ofp, "%c%c%c==%c%c%c\t", Symbols[t*jLoopIndex],
Symbols[t*jLoopIndex + 1], Symbols[t*jLoopIndex + 2], tempInputArray[t*tempIndex], tempInputArray[t*tempIndex + 1],
tempInputArray[t*tempIndex + 2]);
#endif
                                                        if ((tempInputArray[t*tempIndex] == Symbols[t*jLoopIndex]) &&
(tempInputArray[t*tempIndex + 1] == Symbols[t*jLoopIndex + 1]) && (tempInputArray[t*tempIndex + 2] == Symbols[t*jLoopIndex + 2]))
                                                        {
#ifdef DEBUG
                                                                fprintf(ofp, "Found at Row %d", tempIndex + 1);
#endif
                                                                finalArray[k*tempIndex + firstColumnIndex - 1]++;
                                                                finalArray[k*tempIndex + secondColumnIndex - 1]++;
                                                                finalArray[k*tempIndex + thirdColumnIndex - 1]++;
                                                        }
                                                        else
                                                        {
#ifdef DEBUG
                                                                fprintf(ofp, "Not Found at Row %d", tempIndex + 1);
#endif
                                                        }
#ifdef DEBUG
                                                        fprintf(ofp, "\n");
#endif
                                                }
#ifdef DEBUG
                                                printMatrix(finalArray, N, k);
#endif
                                                free(tempInputArray);
                                        }
                                }
                        }
                }
                //Converting 0 to * for printing in the final Matrix
                for (iLoopIndex = 0; iLoopIndex < N; iLoopIndex++)
                {
                        for (jLoopIndex = 0; jLoopIndex < k; jLoopIndex++)
                        {
                                if ('0' == finalArray[k*iLoopIndex + jLoopIndex])
                                {
                                        finalArray[k*iLoopIndex + jLoopIndex] = '*';
                                }
                                else
                                {
                                        finalArray[k*iLoopIndex + jLoopIndex] = inputArray[k*iLoopIndex + jLoopIndex];
                                }
                        }
                }
                //fprintf(ofp, "Don't Care Matrix\n");
#ifdef CONSOLE
                printf("Don't Care Matrix\n");
#endif
                printMatrix(finalArray, N, k);
                //Printing the co-ordinates of Don't Care positions
                //fprintf(ofp, "Don't Care position\n");
#ifdef CONSOLE
                printf("Don't Care position\n");
#endif
                iCount = 0;
                for (iLoopIndex = 0; iLoopIndex < N; iLoopIndex++)
                {
                        for (jLoopIndex = 0; jLoopIndex < k; jLoopIndex++)
                        {
                                if ('*' == finalArray[k*iLoopIndex + jLoopIndex])
```

```
                                        {
                                                iCount++;
                                                //fprintf(ofp, "(%d,%d)\t", iLoopIndex + 1, jLoopIndex + 1);
#ifdef CONSOLE
                                                printf("(%d,%d)\t", iLoopIndex + 1, jLoopIndex + 1);
#endif
                                        }
                                }
                        }
                        //fprintf(ofp, "\nTotal Number of Don't Cares :: %d\n", iCount);
#ifdef CONSOLE
                        printf("\nTotal Number of Don't Cares :: %d\n", iCount);
#endif
                        free(finalArray);
                }

                free(inputArray);
                free(elements);
                free(Symbols);
                free(SubArrays);
                free(intermediateArray);

                after = omp_get_wtime();

#ifdef CONSOLE
                printf("Parallel Time:::%8.6f secs\n", (float)(after - before));
#endif
                fprintf(ofp, "%8.6f\n", (float)(after - before));

                fclose(ofp);
                return 0;
}
```