**PARALLEL AND DISTRIBUTED COMPUTING LAB**

**REPORT**

**NAME:** S Shyam Sundaram

**REG NO:** 19BCE1560

**PROGRAMMING ENVIRONMENT:** OpenMP

**PROBLEM:** Profiling

**DATE:** 27th September, 2021

**HARDWARE CONFIGURATION:**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU NAME | | : | Intel core i5 – 1035G1 @ 1.00 Ghz |
| Number of Sockets: | | : | 1 |
| Cores per Socket | | : | 4 |
| Threads per core | | : | 1 |
| L1 | Cache size | : | 320KB |
| L2 | Cache size | : | 2MB |
| L3 | Cache size (Shared): | | 6MB |
| RAM | | : | 8 GB |

**STATEMENT**

The performance of a piece of code is evaluated by looking at metrics such as time taken by each function to complete its task. This can give us better insights into which parts of the code can be parallelised in order to try making it more efficient. For this experiment, a sample C program is written with multiple functions that include: Matrix Addition, Matrix Multiplication, Matrix Subtraction, Prefix sum, checking if a matrix is Diagonal or not and checking if a matrix is symmetrical or not. Each of these are then subject to functional, line and hardware profiling. The same C program is used for functional and line profiling. For hardware, a parallelized code for matrix multiplication is used.

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include<time.h>

#define M 1000

#define N 1000

#define I 1000

#define V 10000000

int\*\* initArray(int r,int c,int u,int l)

{

    srand(time(0));

    int\*\* arr=(int\*\*)malloc(r\*sizeof(int\*));

    for(int i=0;i<r;++i)

        arr[i]=(int\*)malloc(c\*sizeof(int));

    if(u!=l && l!=0 && u!=0)                            //initializes each element to a random array if lower (l) and uper (u) vales are different

    for(int i=0;i<r;++i)

        for(int j=0;j<c;++j)

            arr[i][j]=(rand()%(u-l+1))+l;

    if(l==0 && u==0)                                    //initialize elements to 0 if l and u are 0

    for(int i=0;i<r;++i)

        for(int j=0;j<c;++j)

            arr[i][j]=0;

    return arr;

}

void freeArray(int\*\* arr,int r)

{

    for(int i=0;i<r;++i)

    free(arr[i]);

    free(arr);

}

void MatAdd()

{

    //init array

    printf("Starting Matrix addition...\n");

    printf("Initialising arrays...\n");

    int\*\* a=initArray(M,N,100,1);

    int\*\* b=initArray(M,N,100,1);

    int\*\* c=initArray(M,N,1,1);

    printf("Performing addition...\n");

    for(int i=0;i<M;++i)

        for(int  j=0;j<N;++j)

            c[i][j]=a[i][j]+b[i][j];

    printf("Finished addition...\n");

    printf("Freeing Arrays...\n\n");

    freeArray(a,M);

    freeArray(b,M);

    freeArray(c,M);

}

void MatMul()

{

    //init array

    printf("Starting Matrix Multiplication...\n");

    printf("Initialising arrays...\n");

    int\*\* a=initArray(M,I,100,1);

    int\*\* b=initArray(I,N,100,1);

    int\*\* c=initArray(M,N,0,0);

    printf("Performing Multiplication...\n");

    for(int i=0;i<M;++i)

        for(int j=0;j<N;++j)

            for(int k=0;k<I;++k)

                c[i][j]+=a[i][k]\*b[k][j];

    printf("Finished Multiplication...\n");

    printf("Freeing Arrays...\n\n");

    freeArray(a,M);

    freeArray(b,M);

    freeArray(c,M);

}

void MatSub()

{

    //init array

    printf("Starting Matrix Subtraction...\n");

    printf("Initialising arrays...\n");

    int\*\* a=initArray(M,N,100,1);

    int\*\* b=initArray(M,N,100,1);

    int\*\* c=initArray(M,N,1,1);

    printf("Performing subtraction...\n");

    for(int i=0;i<M;++i)

        for(int  j=0;j<N;++j)

            c[i][j]=a[i][j]-b[i][j];

    printf("Finished subtraction...\n");

    printf("Freeing Arrays...\n\n");

    freeArray(a,M);

    freeArray(b,M);

    freeArray(c,M);

}

void isDiagonal()

{

    //init array

    printf("Starting isDiagonal...\n");

    int flag=1;

    printf("Initialising array...\n");

    int\*\* a=initArray(N,N,100,1);

    printf("Examining...\n");

    for(int i=0;i<N;++i)

    {

        for(int  j=0;j<N;++j)

        if(i!=j && a[i][j]!=0)

        {

            flag=1;

            break;

        }

        if(flag==1)

        break;

    }

    if(flag==1)

    printf("Not a diagonal matrix\n");

    else

    printf("Is a diagoal matrix\n");

    printf("Freeing Array...\n\n");

    freeArray(a,N);

}

void isSymmetric()

{

    //init array

    printf("Starting isSymmetric...\n");

    int flag=1;

    printf("Initialising array...\n");

    int\*\* a=initArray(N,N,100,1);

    printf("Examining...\n");

    for(int i=0;i<N;++i)

    {

        for(int  j=0;j<N;++j)

        if(a[i][j]!=a[j][i])

        {

            flag=1;

            break;

        }

        if(flag==1)

        break;

    }

    if(flag==1)

    printf("Not a symmetric matrix\n");

    else

    printf("Is a symmetric matrix\n");

    printf("Freeing Array...\n\n");

    freeArray(a,N);

}

void prefix()

{

    printf("Initialising arrays...\n");

    long \*x=(long\*)malloc(V\*sizeof(long));

    for(int i=0;i<V;++i)

    x[i]=i;

    long \*y=(long\*)malloc(V\*sizeof(long));

    y[0]=x[0];

    printf("Summing prefixes...\n");

    for(int i=1;i<V;++i)

    y[i]=y[i-1]+x[i];

    printf("Summation done. Freeing arrays...\n\n");

    free(x);

    free(y);

}

int main()

{

    MatMul();

    MatAdd();

    MatSub();

    isSymmetric();

    isDiagonal();

    prefix();

    return 0;

}

**FUNCTIONAL PROFILING**

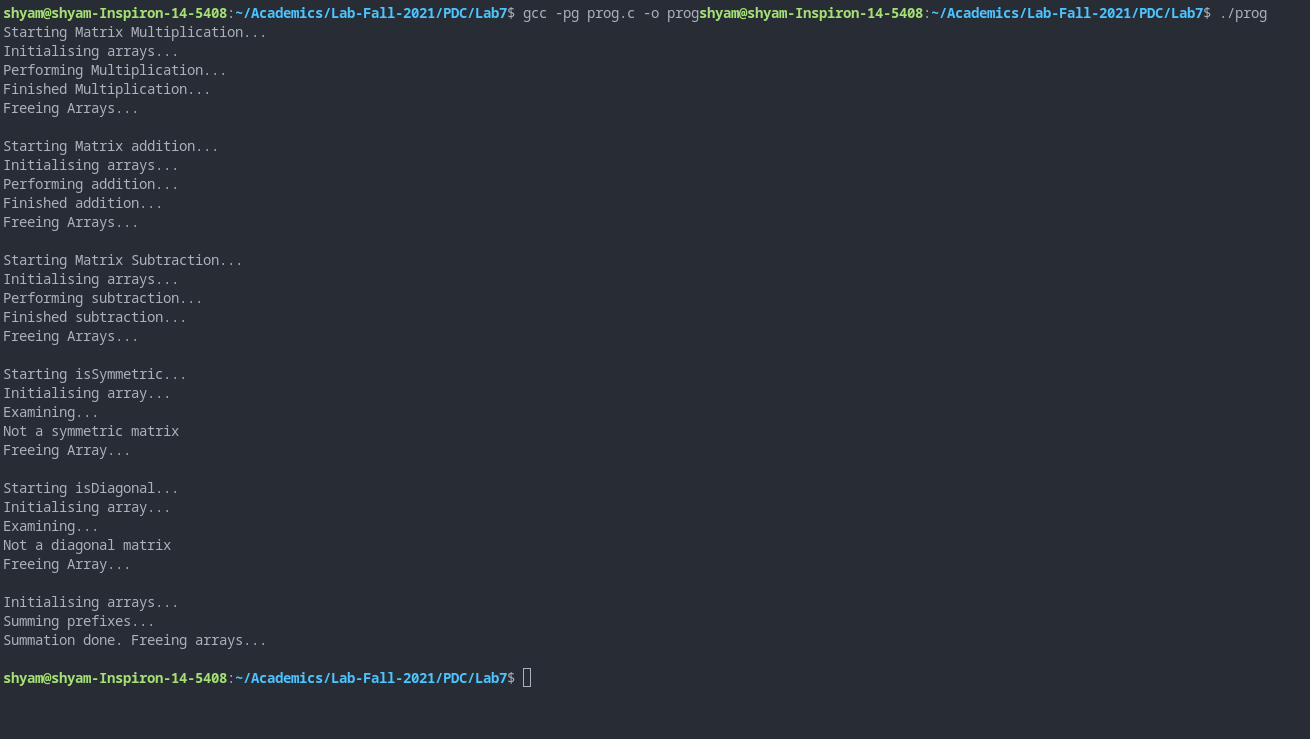
**COMPILATION AND EXECUTION**

gcc -pg prog.c -o prog

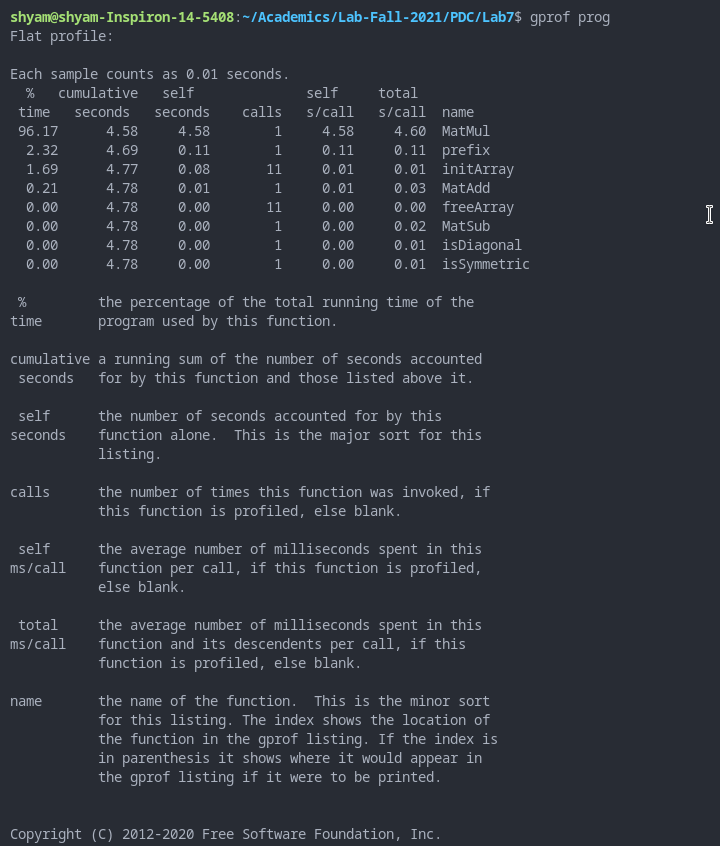
./prog

gprof prog

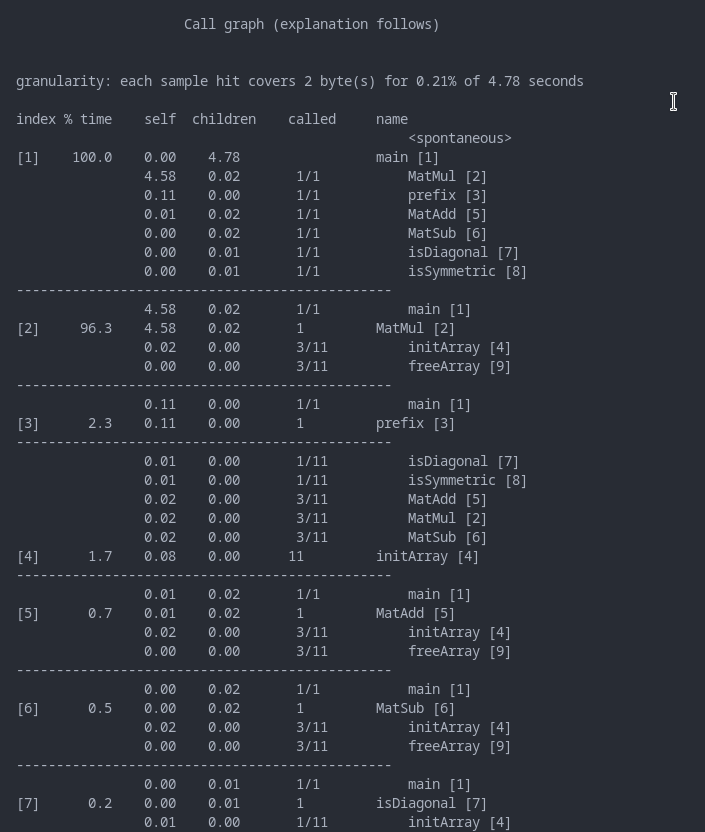
**SCREENSHOTS**

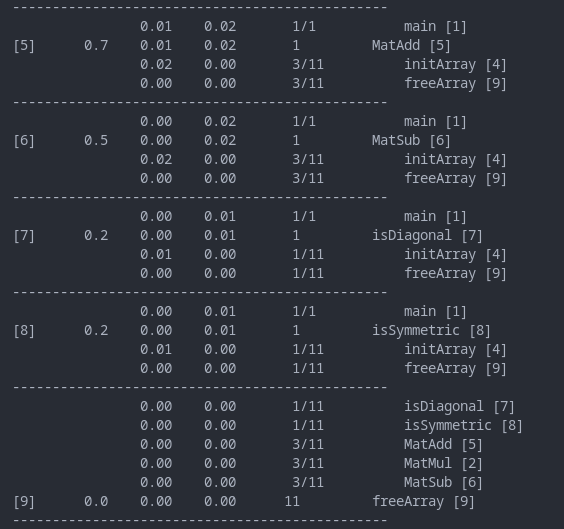
****

**Flat profile:**

****

**Call Graph:**

****

****

**INFERENCES AND OBSERVATIONS**

From the profile generated, we see that matrix multiplication takes the most amount of the total execution time, 4.58 seconds, which forms about 96% of the total time. This is also observed in the 2nd entry of call graph. We also see that the functions called the most are initArray and freeArray, each called 11 times. From this, we see that we could speedup our program by improving upon matrix multiplication, which is parallelizable.

**LINE PROFILING**

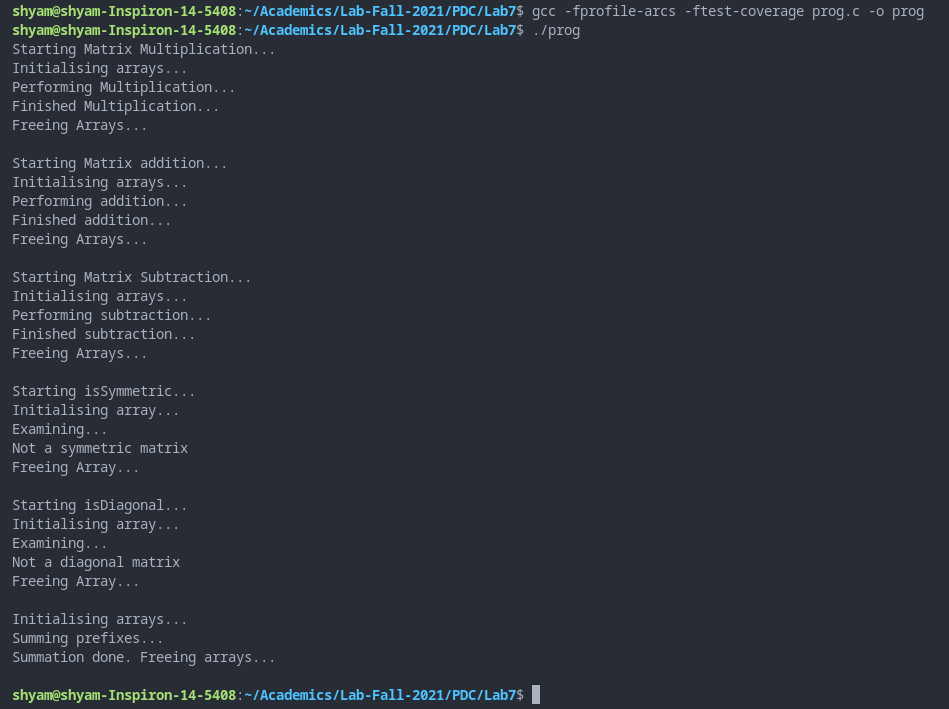
**COMPILATION AND EXECUTION**

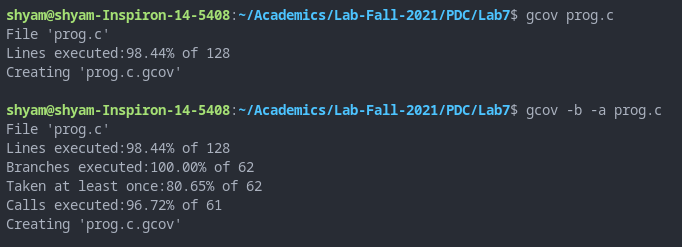
gcc -fprofile-arcs -ftest-coverage prog.c -o prog

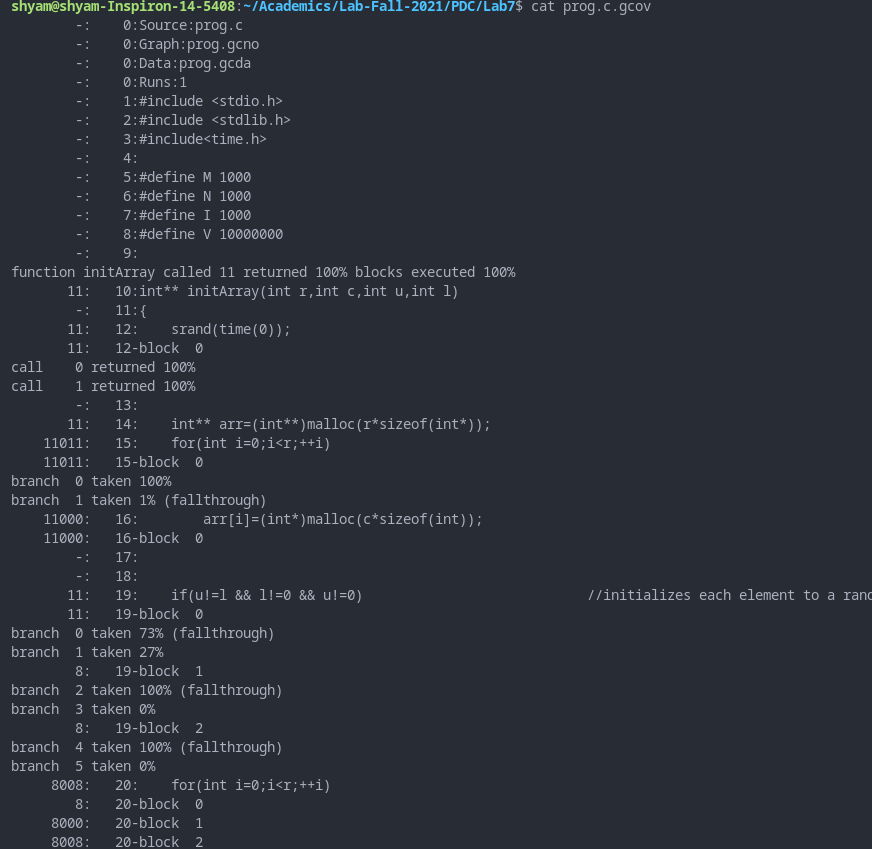
./prog

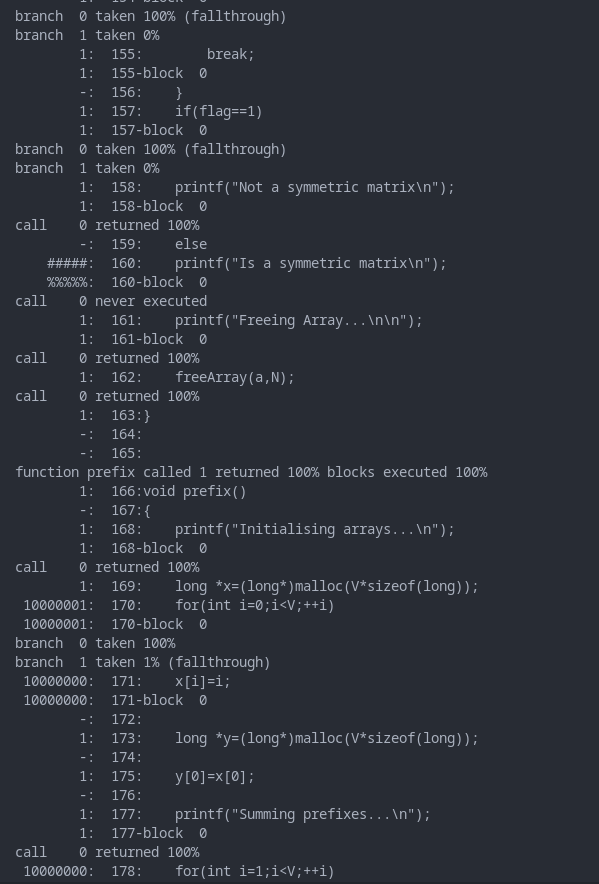
gcov prog.c

**SCREENSHOTS**

****

****

****

****

**INFERENCES AND OBSERVATIONS**

Gcov helps us profile our code line by line. We see that 98.44% of 128 lines are executed. This is attributed to the fact the certain conditions were not met and those statements are hence not executed. We also see how many times each line was executed in the last two screenshots (the number prefixed at each line is the count).

**HARDWARE PROFILING**

**CODE (Parallelized matrix multiplication)**

#include<stdio.h>

#include<stdlib.h>

#include<omp.h>

#define M 2500

#define N 250

#define L 300

int main()

{

    int chunk = 10;

    printf("Name: Shyam Sundaram\nReg num: 19BCE1560\nPDC Lab:\n\n");

    float a[M\*L],b[L\*N],c[M\*N];

    for(int i=0;i<M;++i)

    for(int j=0;j<L;++j)

    a[j+i\*L]=10\*j+i;

    for(int i=0;i<L;++i)

    for(int j=0;j<N;++j)

    b[j+i\*N]=10\*j+i;

    for(int i=0;i<M;++i)

    for(int j=0;j<N;++j)

    c[j+i\*N]=0;

    omp\_set\_num\_threads(32);

    float start=omp\_get\_wtime();

    int i,j,k;

    #pragma omp parallel private(i,j,k) shared(a,b) reduction(+:c)

    {

        #pragma omp for schedule(static,chunk) collapse(3)

        for(i=0;i<M;++i)

        {

            for(j=0;j<N;++j)

            {

                for(k=0;k<L;++k)

                {

                    c[j+i\*N]+=a[k+i\*L]\*b[j+k\*N];

                }

            }

        }

    }

    float end=omp\_get\_wtime();

    float exec=end-start;

    printf("Thread count: %d Time taken is: %f\n",32,exec);

    return 0;

}

**COMPILATION AND EXECUTION**

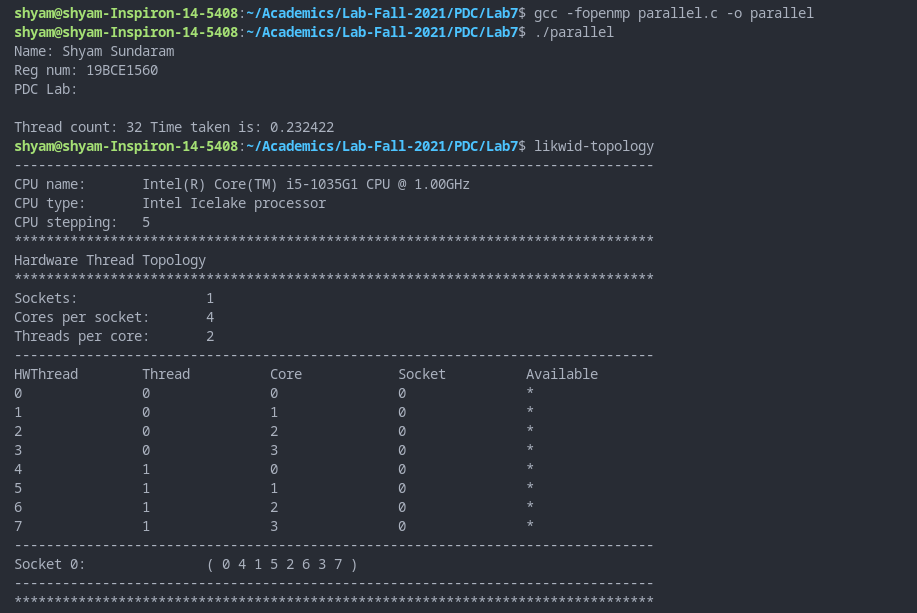
gcc -fopenmp parallel.c -o parallel

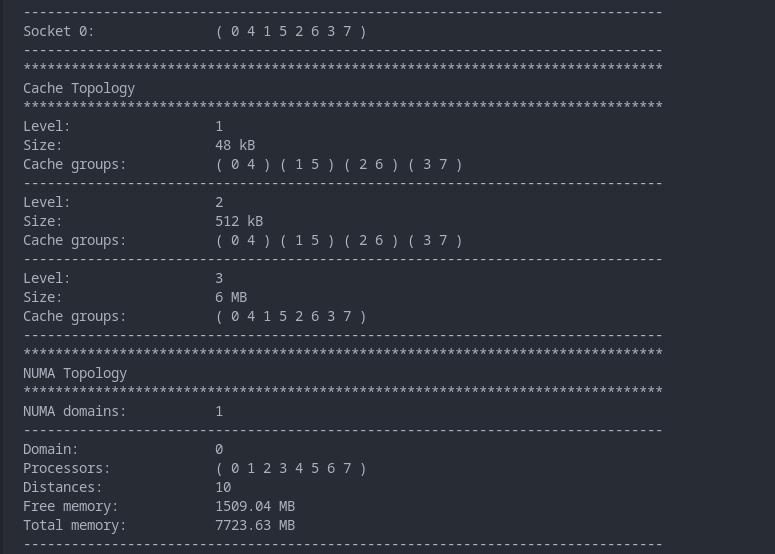
./parallel

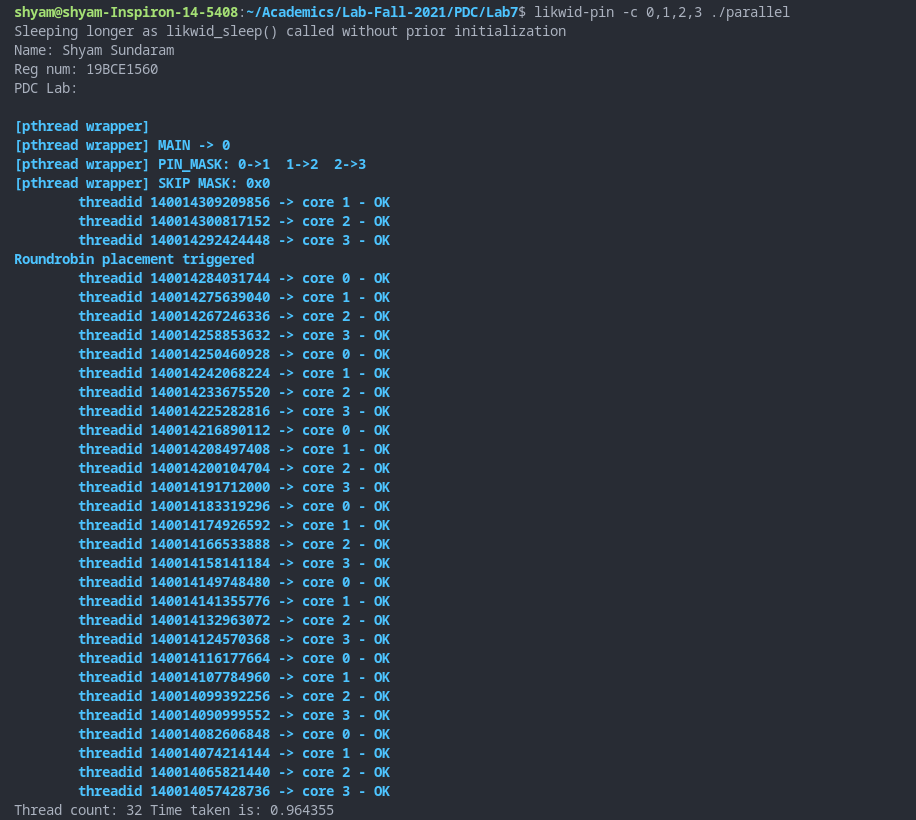
likwid-topology

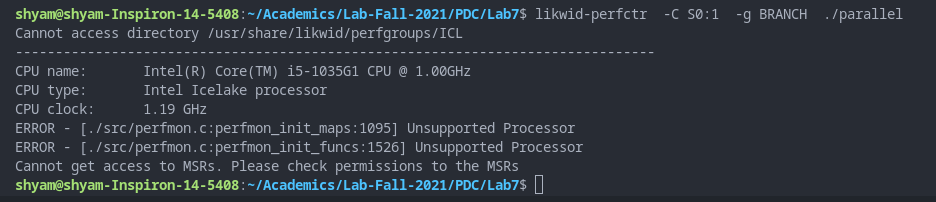
likwid-pin -c 0,1,2,3 ./parallel

**SCREENSHOTS**









**INFERENCES AND OBSERVATIONS**

Likwid is a suite of command line applications used to estimate a program’s performance. With ‘likwid-topology’, we are shown the thread, cache, NUMA and GPU topologies, properties and information. With likwid-pin, we pin each software thread to hardware and evaluate the code. This means we specify which threads to use (as seen in third screenshot command and output).

**CONCLUSION**

We have used three different profiling techniques and tools to evaluate the performance of code written in C. Doing so has given us better understanding and identification of the parts of program which have room for improvement and reduce overall time taken to execute the program.