**PARALLEL AND DISTRIBUTED COMPUTING LAB**

**REPORT**

**NAME:** S Shyam Sundaram

**REG NO:** 19BCE1560

**PROGRAMMING ENVIRONMENT:** OpenMP

**PROBLEM:** Profiling

**DATE:** 6th October, 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 sorting algorithm, Bubble Sort, is implemented serially and in a parallelized manner using OpenMP as functions. The program is then profiled using functional, line and hardware profiling techniques.

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define SIZE 10000

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

{

    srand(time(0));

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

    for(int j=0;j<r;++j)

        arr[j]=(rand()%(u-l+1))+l;

    return arr;

}

void freeArray(int\* arr)

{

    free(arr);

}

void bubbleSortSerial(int a[])

{

    int temp;

    for(int i=0;i<SIZE;++i)

    {

        for(int j=i;j<SIZE;++j)

        {

            if(a[i]>a[j+1])

            {

                temp=a[j];

                a[j]=a[j+1];

                a[j+1]=temp;

            }

        }

    }

}

void bubbleSortParallel(int a[])

{

Int k;

    omp\_set\_num\_threads(4);

    for(int i=0;i<SIZE;++i)

    {

        k=i%2;

        #pragma omp paralel for default(none) shared(k,a)

        for(int j=k;j<SIZE-1;j+=2)

        {

            if(a[j]>a[j+1])

            {

                int temp=a[j];

                a[j]=a[j+1];

                a[j+1]=temp;

            }

        }

    }

}

void main()

{

    int \*a=initArray(SIZE,100,0);

    int \*b=initArray(SIZE,100,0);

    float start=omp\_get\_wtime();

    bubbleSortSerial(a);

    float end=omp\_get\_wtime();

    float exec=end-start;

    printf("Time taken is: %f\n",exec);

    freeArray(a);

    start=omp\_get\_wtime();

    bubbleSortParallel(b);

    end=omp\_get\_wtime();

    exec=end-start;

    printf("Time taken is: %f\n",exec);

    freeArray(b);

}

**FUNCTIONAL PROFILING**

**COMPILATION AND EXECUTION**

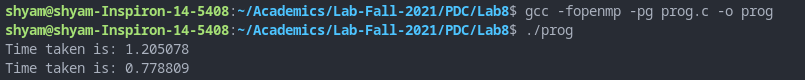
gcc -fopenmp -pg prog.c -o prog

./prog

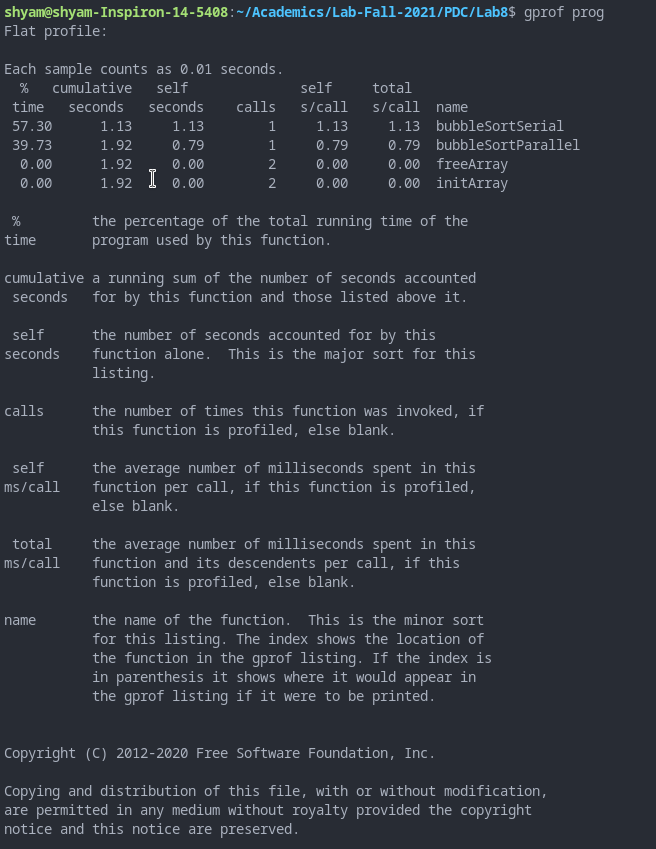
gprof prog

gprof -b prog

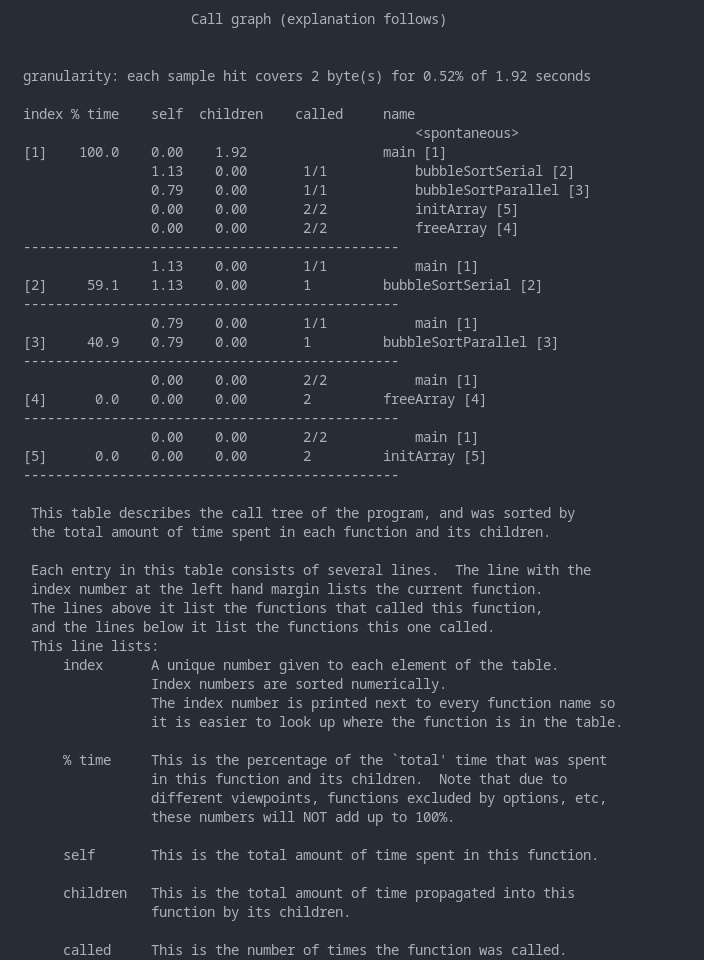
**SCREENSHOTS**

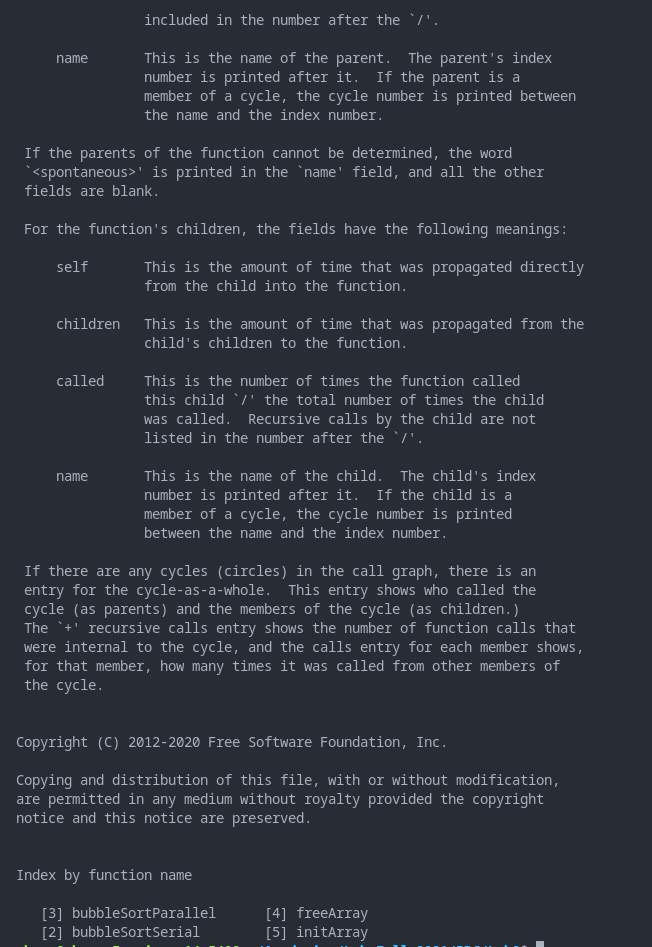
****

**Flat profile:**

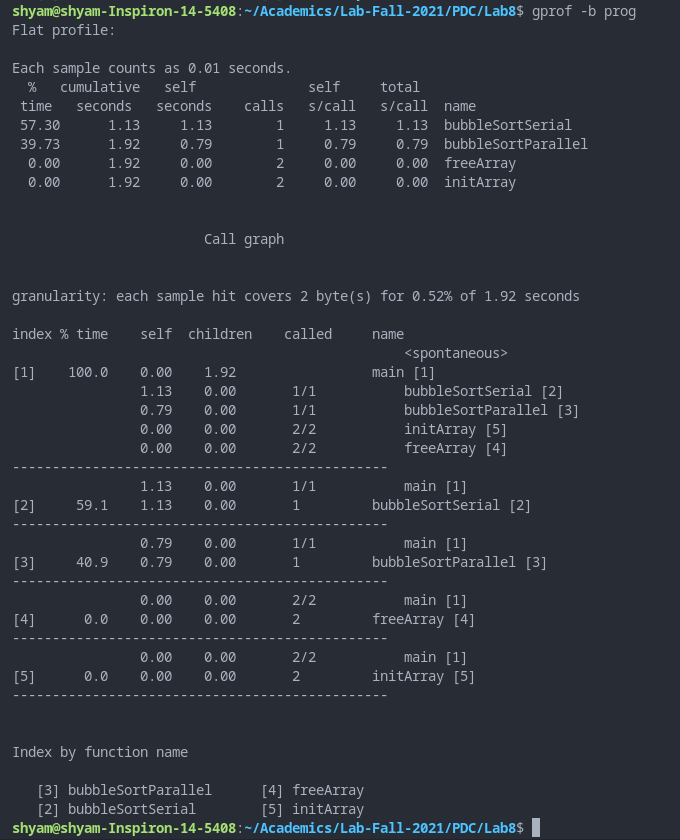
****

**Call Graph:**

****

****

**GPROF with brief report of flat profile and call graph: ( gprof -b prog )**

****

**INFERENCES AND OBSERVATIONS**

From the profile generated, we see that serial bubble sort function takes the longest time. It is followed by parallel bubble sort and free and array initialising functions take very little time.

**LINE PROFILING**

**COMPILATION AND EXECUTION**

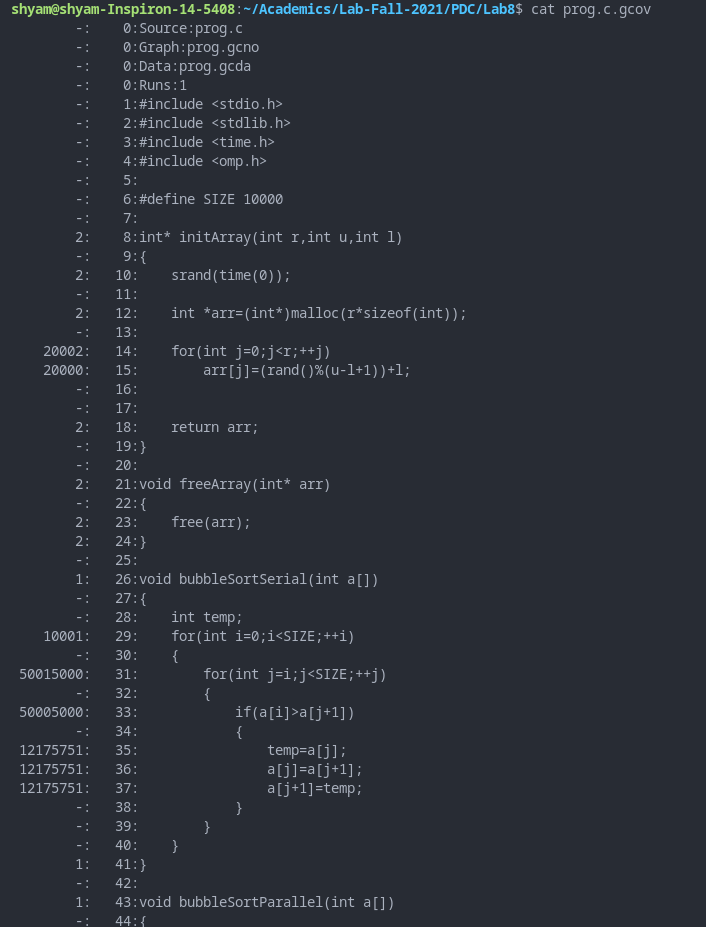
gcc -fopenmp -fprofile-arcs -ftest-coverage prog.c -o prog

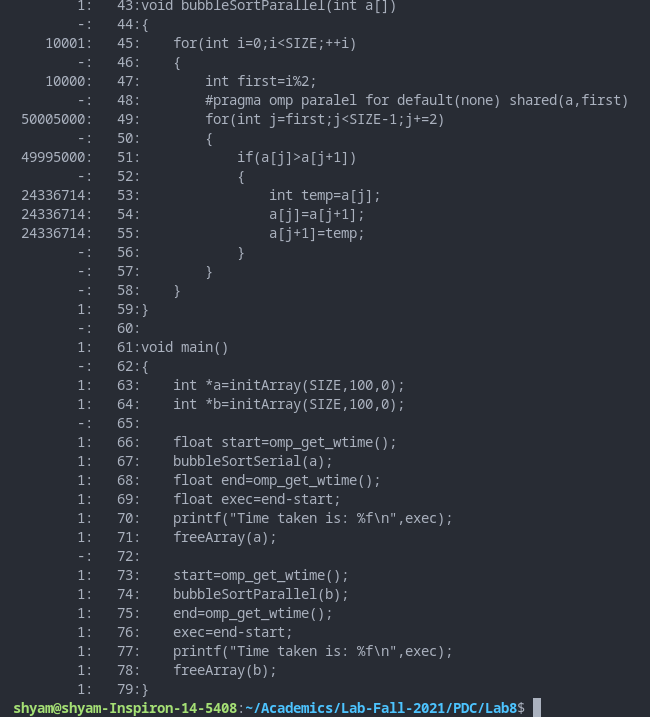
./prog

gcov prog.c

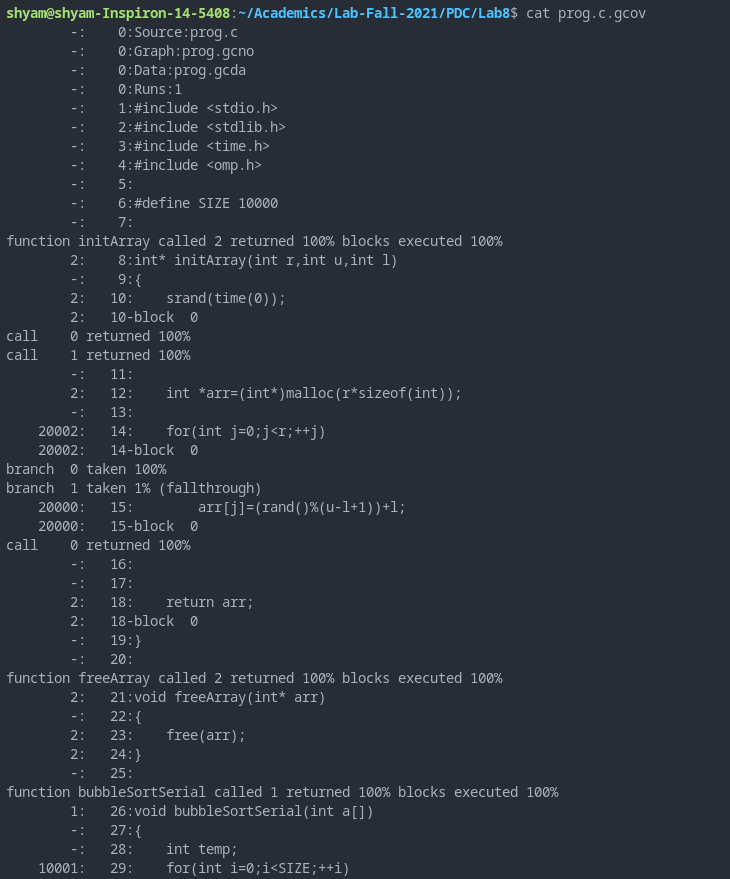
cat prog.c.gcov

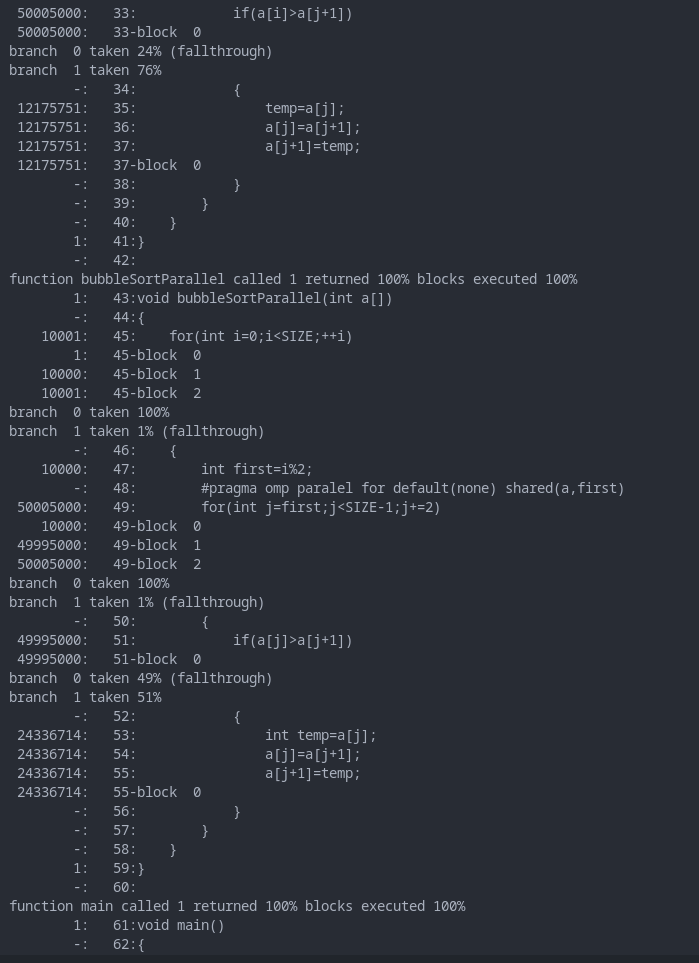
**SCREENSHOTS**

****

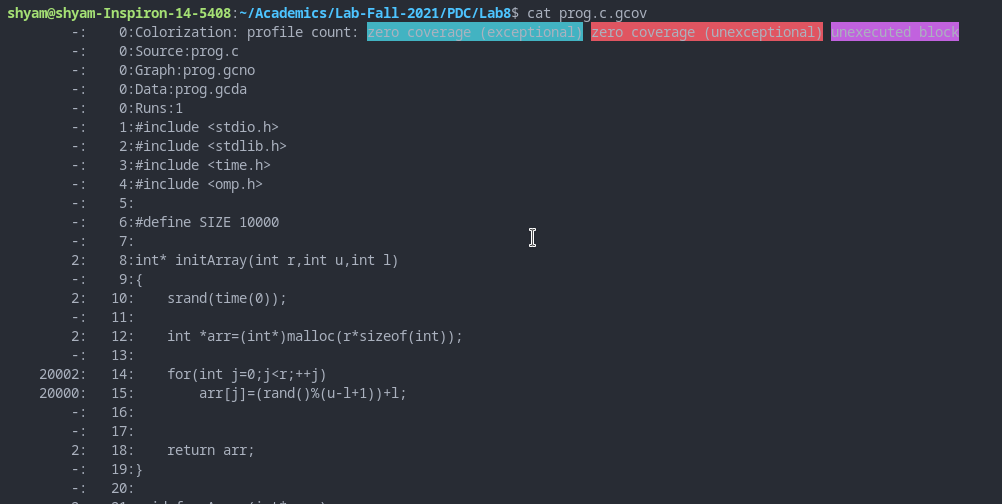
****

**Write individual execution counts for every basic block and branch frequencies to the output file along with branch summary info: ( gcov -b -a prog.c )**

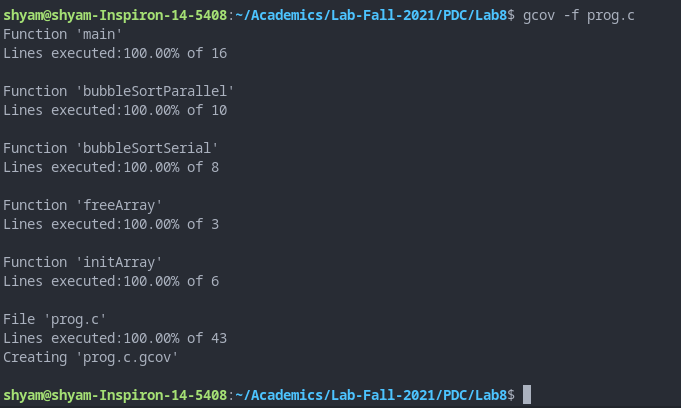
****

****

**Use colors for lines of code that have zero coverage. (gcov -k prog.c)**



**Output summaries for each function in addition to the file level summary (gcov -f prog.c )**

****

**INFERENCES AND OBSERVATIONS**

Gcov helps us profile our code line by line. Various options of gcov are also explored such as coloring lines with no coverage, display summaries of each function etc.

**HARDWARE PROFILING**

**COMPILATION AND EXECUTION**

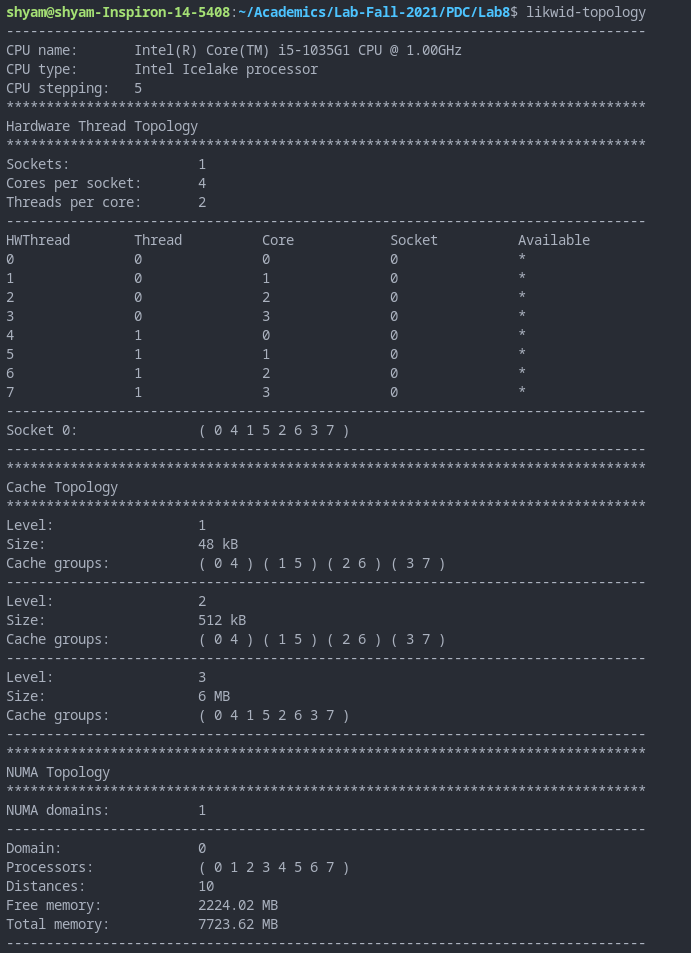
gcc -fopenmp prog.c -o prog

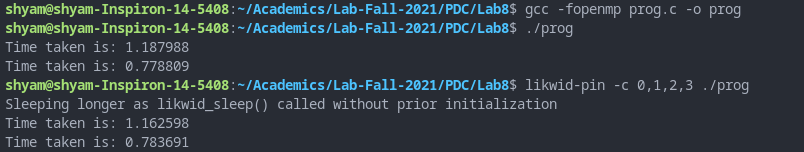
./prog

likwid-topology

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

**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 serial and parallel bubble sort code written in C. Various options of gprof and gcov were also explored.