**Matrix Addition**

**OBJECTIVE:**

1. Perform matrix addition on ‘n2’ double precision floating point numbers of two n X n matrices

**Serial Code:**

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include "omp.h"

int main(){

int n;

scanf("%d", &n);

double v1[n][n], v2[n][n], ans[n][n];

for(int i = 0; i < n; i++){

for(int j = 0; j < n; j++){

v1[i][j] = (float)rand()/(float)(RAND\_MAX/n);

v2[i][j] = (float)rand()/(float)(RAND\_MAX/n);

ans[i][j] = 0;

}

}

for(int i = 0; i < n; i++){

for(int j = 0; j < n; j++){

ans[i][j] = v1[i][j] + v2[i][j];

}

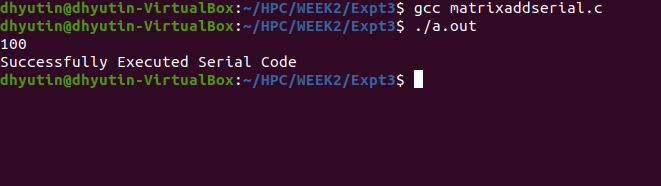
}

printf("Successfully Executed Serial Code\n");

return 0;

}

**Output:**

****

**Parallelized Code:**

#include <stdio.h>

#include <stdlib.h>

#include "omp.h"

int main(){

int n;

scanf("%d", &n);

double v1[n][n], v2[n][n], ans[n][n];

for(int i = 0; i < n; i++){

for(int j = 0; j < n; j++){

v1[i][j] = (float)rand()/(float)(RAND\_MAX/n);

v2[i][j] = (float)rand()/(float)(RAND\_MAX/n);

ans[i][j] = 0;

}

}

double wallclock\_initial = omp\_get\_wtime();

#pragma omp parallel

{

int id = omp\_get\_thread\_num();

printf("Thread No - %d\n", id);

#pragma omp for collapse(2)

for(int i = 0; i < n; i++){

// #pragma omp for

for(int j = 0; j < n; j++){

ans[i][j] = v1[i][j] + v2[i][j];

}

}

}

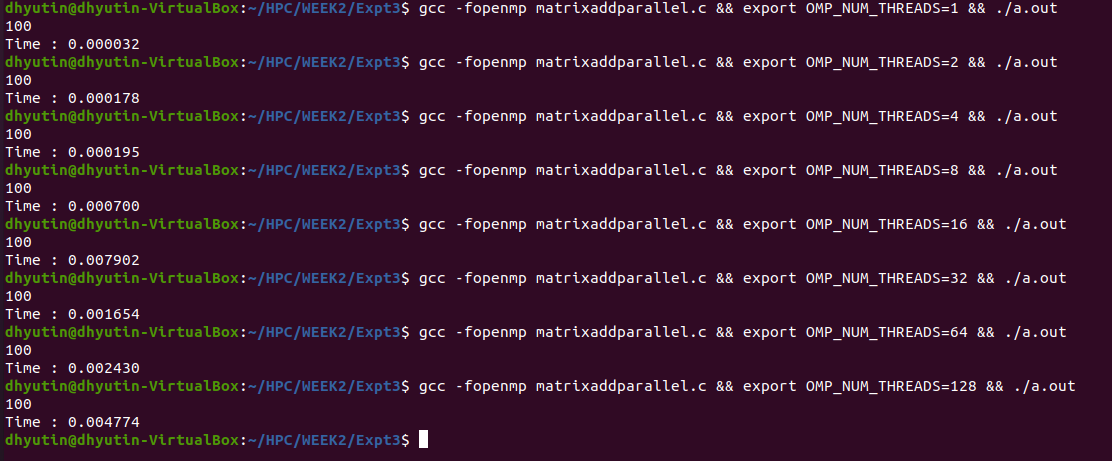
double wallclock\_final = omp\_get\_wtime();

printf("Time : %lf\n", wallclock\_final - wallclock\_initial);

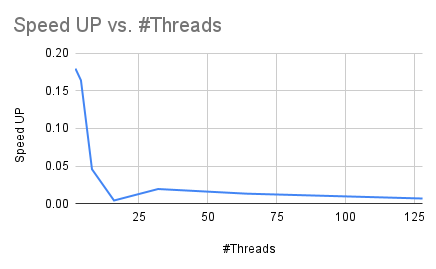
return 0;

}

**Output:**

****

**Speedup V/S Number of Processors:**

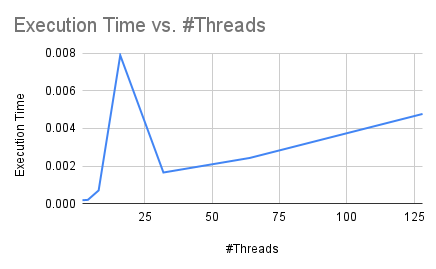
****

**Inference:**

It can be inferred from the above graph that till a threshold of 16 threads(16 processors), speedup value is decreasing and it increases till 64 threads and decreases.

Similar to the previous experiments, it is evident that this isn’t a computationally expensive task and thus single thread execution is the most optimum one to choose to attain results soon.

**Execution Time V/S Number of Threads:**

****

**Inference:**

It can be inferred from the above graph that till a threshold of 16 threads(16 processors), Execution time is increasing and it decreases till 64 threads and increases.

Similar to the previous experiments, it is evident that this isn’t a computationally expensive task and thus single thread execution is the most optimum one to choose to attain results soon.

**Parallelization Factor (f):**

| **#Threads** | **Execution Time** | **Speed UP** | **Efficiency (in %)** | **f** |
| --- | --- | --- | --- | --- |
| 1 | 0.000032 | 1 | 100 | n/a |
| 2 | 0.000178 | 0.1797752809 | 8.988764045 | -9.125 |
| 4 | 0.000195 | 0.1641025641 | 4.102564103 | -6.791666667 |
| 8 | 0.0007 | 0.04571428571 | 0.5714285714 | -23.85714286 |
| 16 | 0.007902 | 0.004049607694 | 0.02531004809 | -262.3333333 |
| 32 | 0.001654 | 0.01934703748 | 0.06045949214 | -52.32258065 |
| 64 | 0.00243 | 0.01316872428 | 0.02057613169 | -76.12698413 |
| 128 | 0.004774 | 0.006702974445 | 0.005236698785 | -149.3543307 |

**Inference:**

Considering that single thread execution is the most optimum for this program, it is clear that ‘f’ value will be in negative for any number of threads apart from 1.

Same trend of decreasing till 16 threads and increasing till 64 threads and further decreasing is noticed for values of ‘f’ also.

**THE END**