Week-4

Applying OpenMP on Project

Roll no. - CED19I011

Name - Sandeep Ahirwar

Project Description

Problem Statement : **Parallelize Fitness calculation in Genetic algorithm**

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

There are many different complex problem which can be solved through genetic algorithm, we have taken this parallelize fitness calculation for our project.

INPUT GIVEN

- ORGS : 10000
- GENES : 50
- ALLELES : 4
- MUT 1000

Profiling Inference

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.21     14.85    14.85       81     0.18     0.18  Gen
  1.34     15.05     0.20       82     0.00     0.00  Eval
  0.00     15.05     0.00        1     0.00     0.00  Init
  0.00     15.05     0.00        1     0.00     0.00  Mem
  0.00     15.05     0.00        1     0.00    15.05  Run
```

```
Call graph:

granularity: each sample hit covers 2 byte(s) for 0.07% of 15.05 seconds

index % time    self  children    called     name
                0.00   15.05       1/1           main [2]
[1]    100.0    0.00   15.05       1         Run [1]
               14.85    0.00      81/81          Gen [3]
                0.20    0.00      82/82          Eval [4]
                0.00    0.00       1/1           Init [5]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   15.05                 main [2]
                0.00   15.05       1/1           Run [1]
                0.00    0.00       1/1           Mem [6]
-----------------------------------------------
               14.85    0.00      81/81          Run [1]
[3]     98.7   14.85    0.00      81         Gen [3]
-----------------------------------------------
                0.20    0.00      82/82          Run [1]
[4]      1.3    0.20    0.00      82         Eval [4]
-----------------------------------------------
                0.00    0.00       1/1           Run [1]
[5]      0.0    0.00    0.00       1         Init [5]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[6]      0.0    0.00    0.00       1         Mem [6]
-----------------------------------------------

Index by function name

   [4] Eval                    [5] Init                    [1] Run
   [3] Gen                     [6] Mem
```

From this profiling of our project serial code we get to know that 99% of execution time is spent on **Gen()** function

Previously there was a function call **Sel()** which was impossible to parallelize because of its depedencies in **Gen()** so I have removes the **Sel()** function and merged it in Gen() in such a way that it can be parallelized upto some extent.

So the above profiling result is after removing the **Sel()** function.

Here, I have parallelized only 1 main `for` loop in Gen(). This for loop contains 3 more for loops which I was not able to parallize because of the `break` statements.
I tried removing the break statement for the inner for loops but after removing it the execution time was getting doubled.

OpenMP Project Code

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

#define ORGS 10000
#define GENES 50
#define ALLELES 4
#define MUT 1000

char **curG, **nextG, *mod;
int *f, totF, Eval(), Run();
void Mem(), Init(), Gen();

int main(){
  float pTime = omp_get_wtime();
  Mem();
  printf("The final generation was: %d\n", Run());
  float cTime = omp_get_wtime();
  printf("%f",cTime-pTime);}

// Initialize Memory Function
void Mem(){
    int o;
    curG=(char**)malloc(sizeof(char*)*ORGS);
    nextG=(char**)malloc(sizeof(char*)*ORGS);
    mod=(char*)malloc(sizeof(char)*GENES);
    f=(int*)malloc(sizeof(int)*ORGS);

    for(o=0; o<ORGS; ++o){
        curG[o]=(char*)malloc(sizeof(char)*GENES);
        nextG[o]=(char*)malloc(sizeof(char)*GENES);
    }
}

// Run Function
int Run(){
```

```
    int gen=0;
    Init();
    while(++gen) {
        if(Eval()) {
            return gen;
        }
        Gen();
    }
}

// Initialization Function
void Init(){
    int o, g;
    for(o=0; o<ORGS; o++) {
        for(g=0; g<GENES; g++) {
            curG[o][g]=rand()%ALLELES;
        }
    }
    for(g=0; g<GENES; g++){
        mod[g]=rand()%ALLELES;
    }
}

// Evaluation Function
int Eval(){
    int o, g, curF;
    for(totF=0, o=0; o<ORGS; ++o) {
        for(curF=0, g=0; g<GENES; ++g) {
            if(curG[o][g]==mod[g]) {
                ++curF;
            }
        }
        if(curF==GENES) {
            return 1;
        }
        totF += f[o]=curF;
    }
    return 0;
}

// Generation Function
void Gen(){
    int o, g, p1, p2, cp;
```

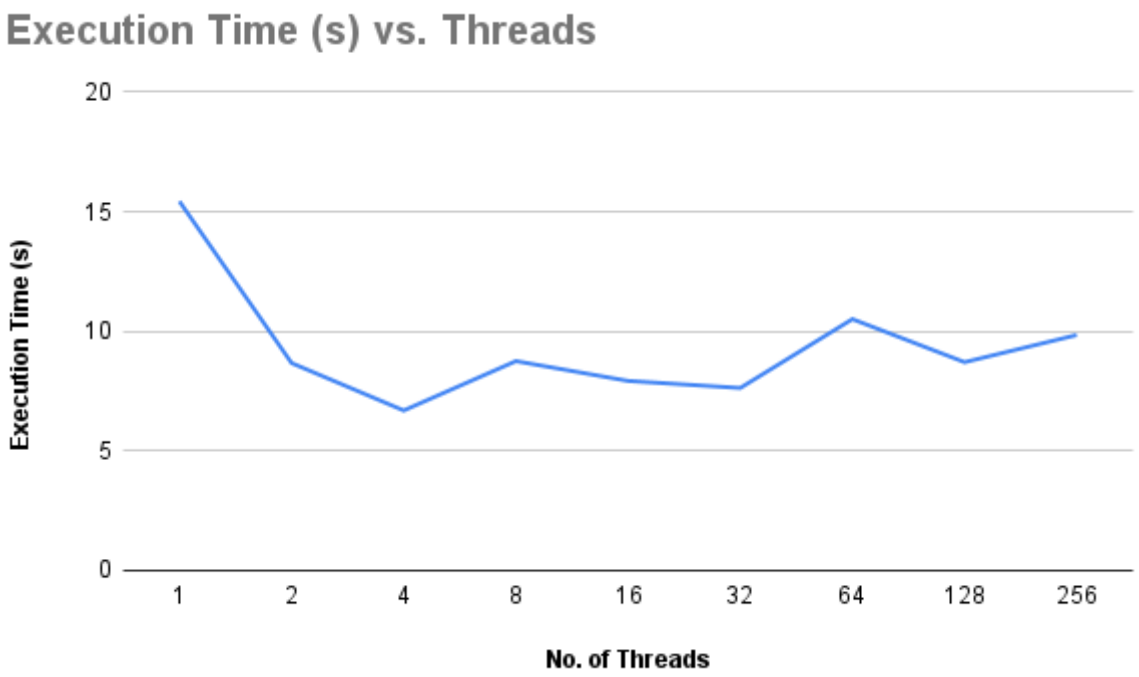
```
    #pragma omp parallel for
    for(o=0;o<ORGS;++o) {
        int tot = 0, pt = rand()%(totF+1);

        for(int o1 = 0; o1 < ORGS; ++o1){
            if((tot += f[o1]) >= pt) {
                p1 = o1;
                break;
            }
        }
        tot = 0; pt = rand() % (1+totF);
        for(int o2 = 0; o2 < ORGS; ++o2){
            if((tot += f[o2]) >= pt) {
                p2 = o2;
                break;
            }
        }

        for(p1, p2, cp = rand() % GENES, g = 0; g < GENES; ++g) {
            if(rand()%MUT) {
                if(g<cp) {
                    nextG[o][g] = curG[p1][g];
                } else {
                    nextG[o][g] = curG[p2][g];
                }
            } else {
                nextG[o][g] = rand()%ALLELES;
            }
        }
    }

    for(o=0; o<ORGS; ++o) {
        for(g=0; g<GENES; ++g) {
            curG[o][g]=nextG[o][g];
        }
    }
}
```

Output

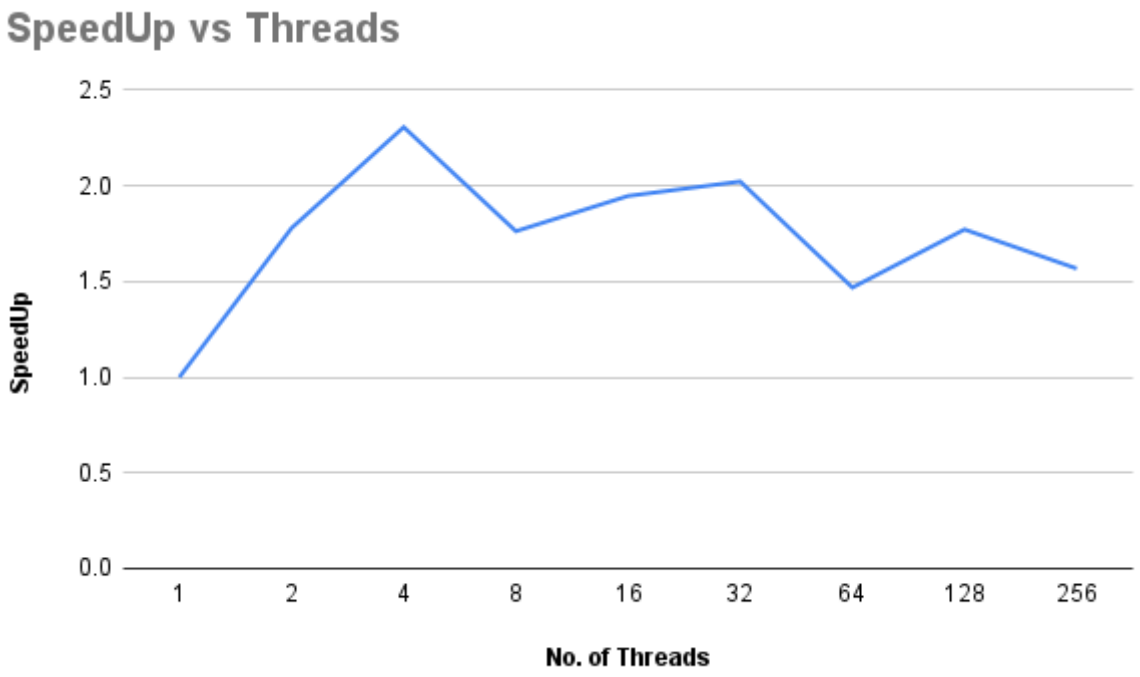
```
➜ PROJECT PROFILING export OMP_NUM_THREADS=1
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 82
15.433105%
➜ PROJECT PROFILING export OMP_NUM_THREADS=2
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 76
8.673340%
➜ PROJECT PROFILING export OMP_NUM_THREADS=4
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 89
6.690430%
➜ PROJECT PROFILING export OMP_NUM_THREADS=8
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 134
8.754883%
➜ PROJECT PROFILING export OMP_NUM_THREADS=16
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 121
7.926758%
➜ PROJECT PROFILING export OMP_NUM_THREADS=32
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 107
7.630859%
➜ PROJECT PROFILING export OMP_NUM_THREADS=64
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 131
10.510254%
➜ PROJECT PROFILING export OMP_NUM_THREADS=128
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 120
8.709961%
➜ PROJECT PROFILING export OMP_NUM_THREADS=256
➜ PROJECT PROFILING gcc -fopenmp -o parallel project_omp.c && ./parallel
The final generation was: 118
9.845215%
```

Result

Thread vs Time Graph



SpeedUp vs Processors



Parallelization Fraction

Threads	Execution Time(s)	Speedup	Speedup(%)	Parallelization Factor
1	15.433105	1	0	--
2	8.67334	1.779372768	77.93727676	0.8760084247
4	6.69043	2.306743363	130.6743363	0.7553178703
8	8.754833	1.762809753	76.28097532	0.4945414975
16	7.926758	1.946963059	94.69630585	0.5188048765
32	7.630859	2.022459726	102.2459726	0.5218607244
64	10.510254	1.46838554	46.83855404	0.3240431198
128	8.709961	1.771891401	77.18914011	0.4390614942
256	9.845215	1.567574197	56.75741972	0.3634915524
			Average Parallelization	**0.536641195**

Average Parallelization Factor = **0.536641195**

Inference

From the thread vs time and speedup vs processors plot, we can observe that the performance of the program has increased with the increase in number of threads but as we go more higher in no. of

threads the context switches is more than the actual function parallelized that is why after few threads the execution time again increases.

Also the best performance, i.e. the best degree of parallelization was observed when the program was run using 4 threads. This was the optimal scenario where the context switches did not increase the runtime of the program by much ans the effect of parallelism was able to take place and reduce the time needed to execute the program.

We were able to get the maximum of 130.6% improvement in performance while using 4 threads.

Thank You