ID: 800962452 Meghana Gudaram HPC Assignment 3

Compilation: g++ -std=c++11 transfer.cpp -fopenmp -o transfer
Execution: OMP_NUM_THREADS=1 ./transfer

#pragma omp parallel – given a work, all threads work independently on work, resulting in work done by work*number of threads running

#pragma omp parallel for - given a work, all threads work together on work, resulting in work done, each thread working work/number of threads

The differences between ***static*** schedule type and ***dynamic*** schedule type is that with "static", the chunks can be pre-computed as well as decided how threads are to be scheduled during compilation itself, whereas with dynamic the same thing is done during run-time. With the usage of "dynamic", it involves some complex mechanisms like deadlock handling mechanism, load handling

Dynamic:
In the dynamic schedule, there is no predictable order in which the loop items are assigned to different threads. Each thread asks the OpenMP runtime library for an iteration number, then handles it, then asks for next, and so on. This is most useful when used in conjunction with the ordered clause, or when the different iterations in the loop may take different time to execute.

Example: #pragma omp parallel for reduction (min:minVal) schedule(dynamic,1000)

In this example, each thread asks for an iteration number, executes 1000 iterations of the loop, and then asks for another, and so on. The last chunk may be smaller than 1000, though.

Size:
All the arrays are considered 10^8, i.e. total of 4*10^8 bytes

Reduction (operation : Variable):
Throughout reduction(min:value) is used to calculate minimum of value obtained by all threads, they all share value
Example: #pragma omp parallel for reduction(min:minVal)

Firstprivate(findvalue)
Since value to be found in the array should be read all threads as initialized before calling pragma omp, default is shared

Lastprivate(Value)
Not applied in the code, but if a variable is just created without initialization, lastprivate makes sure value is copied to the main (after exiting pragma parallel for)

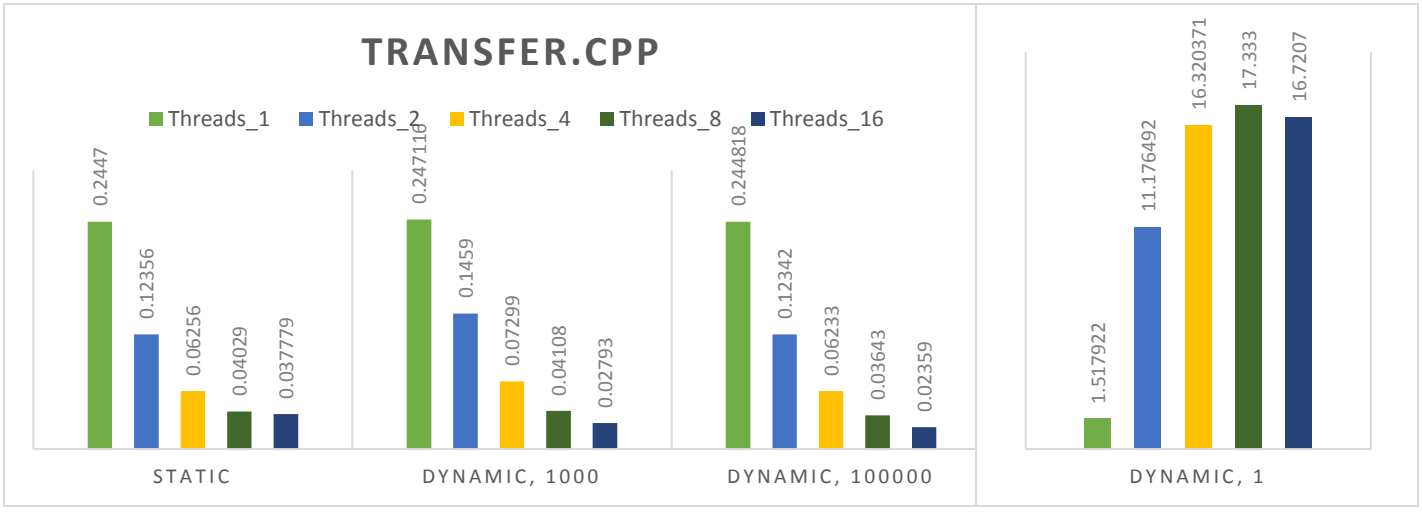
Results for this Assignment:

- In a summary, time of execution decrease with increase in number of threads
- Time of execution decreased as follows (dynamic,100000) < (dynamic,1000) < (static) < (dynamic,1)
- This is achieved as the array is of size 10^8, for array of size in a couple of hundreds, switching between threads is expensive and sequential execution is preferred

1) Transfer.cpp to make array of random values array of random squared values

```
#include<stdio.h>
#include <omp.h>
#include <immintrin.h>
#include <iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

main()
{
        int size=100000000; // size of each array 10^8
        int *randa = (int*)malloc(sizeof(int) * size);
        int *randb = (int*)malloc(sizeof(int) * size);

        int i=0;
        for(int j=0;j<size;j++)
        {
                randa[j]=rand()%size;
                randb[j]=randa[j];
        }

        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        //omp_set_num_threads(4); //number of threads sent as parameter

        #pragma omp parallel for schedule(dynamic,100000)
        for(int ii=0;ii<size;ii++)
        {
                randb[ii]=randb[ii]*randb[ii];
        }

        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        duration<double> time_span = duration_cast<duration<double>> (t2 - t1);

        printf("time taken by %d threads is %lf\n",omp_get_max_threads(),time_span.count());
        return 0;
}
```

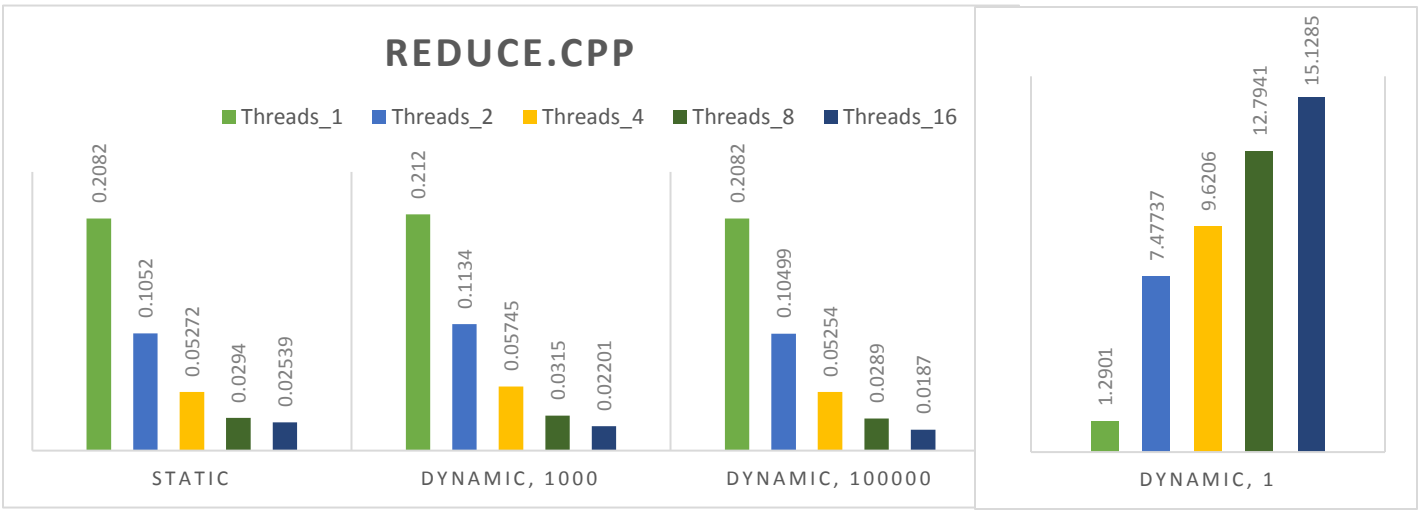
- In (dynamic, 1) each thread asks for an iteration number, executes 1 iteration of the loop, and then asks for another, and so on for 10^8 iterations. Hence context switching takes up a lot of time and resulted in higher time consumption, increasing threads makes it worse because at runtime, scheduler has to decide which iteration goes to which thread
- For rest of the cases, static, dynamic, 1000 and dynamic, 100000 time consumption decreased with increase in number of threads as expected



2) Reduce.cpp to find minimum value in array of random values

```
#include<stdio.h>
#include <omp.h>
#include <immintrin.h>
#include <iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

main()
{
        int size=100000000; // size of Arrays is 10^8
        int *randa = (int*)malloc(sizeof(int) * size);
        int *randb = (int*)malloc(sizeof(int) * size);

        int i=0,minVal=size;
        for(int j=0;j<size;j++)
        {
                randa[j]=rand()%size; // random numbers according to the size of array
                randb[j]=randa[j];
        }

        high_resolution_clock::time_point t1 = high_resolution_clock::now();

        //omp_set_num_threads(4); //number of threads is passed while executing output(a.out) file
        #pragma omp parallel for reduction(min:minVal) schedule(dynamic,100000) // scheduling policy is manually selected,
miniVal is minimum of minVal's by all the threads
        for(int ii=0;ii<size;ii++)
        {       if(minVal>randb[ii])
                {
                        minVal=randb[ii]; // minVal is a shared resource, whose values are selected min through reduction
                }
        }

        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        duration<double> time_span = duration_cast<duration<double>> (t2 - t1);

        printf("time taken by %d threads is %lf\n",omp_get_max_threads(),time_span.count()/100);
        printf("minimum value is %d\n",minVal); // printing minimum value, since it is a shared variable, after omp for
value is still restored
        return 0;
}
```

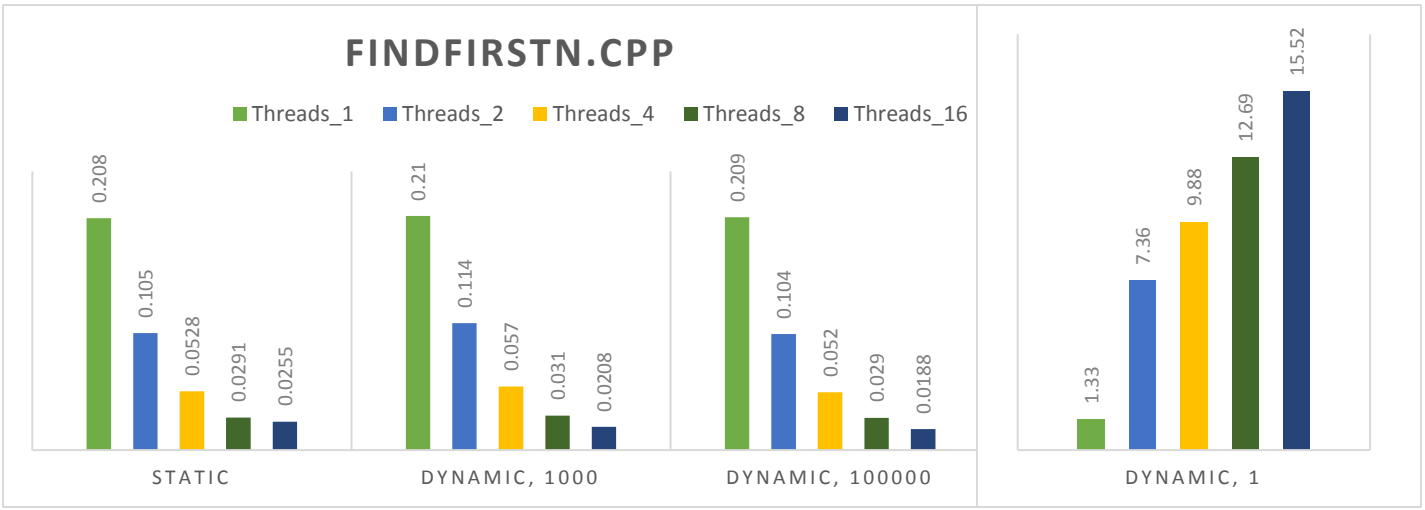
- Same explanation as above for (dynamic,1) consuming more time than expected
- Threads find minVal, and reduction(min:minVal) makes sure minimum of all values by threads is selected
- Again static, dynamic, 1000 and dynamic, 100000 time consumption decreased with increase in number of threads as expected



3) Findfirstn.cpp to find given value with least index in array of random values (0 to N (i.e. size))

```
#include<stdio.h>
#include<iostream>
#include <omp.h>
#include <immintrin.h>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

main()
{
        int size=100000000;
        int *randa = (int*)malloc(sizeof(int) * size);
        int *randb = (int*)malloc(sizeof(int) * size);
        int *next = (int*)malloc(sizeof(int) * size);
        int findVal=46930886;
        int i=0,index=size;
        for(int j=0;j<size;j++)
        {
                randa[j]=rand()%size;
                randb[j]=randa[j];
        }

        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        #pragma omp parallel for schedule(dynamic, 100000) shared(findVal) reduction(min:index)
        for(int ii=0;ii<size;ii++)
        {
                if((findVal == randb[ii]) && (ii<index))
                {//      #pragma omp atomic
                        index=ii;
                }
        }

        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        duration<double> time_span = duration_cast<duration<double>> (t2 - t1);

        printf("time taken for %d threads is %lf\n",omp_get_max_threads(),time_span.count());
        printf("first index is %d\n",index);
        return 0;
}
```

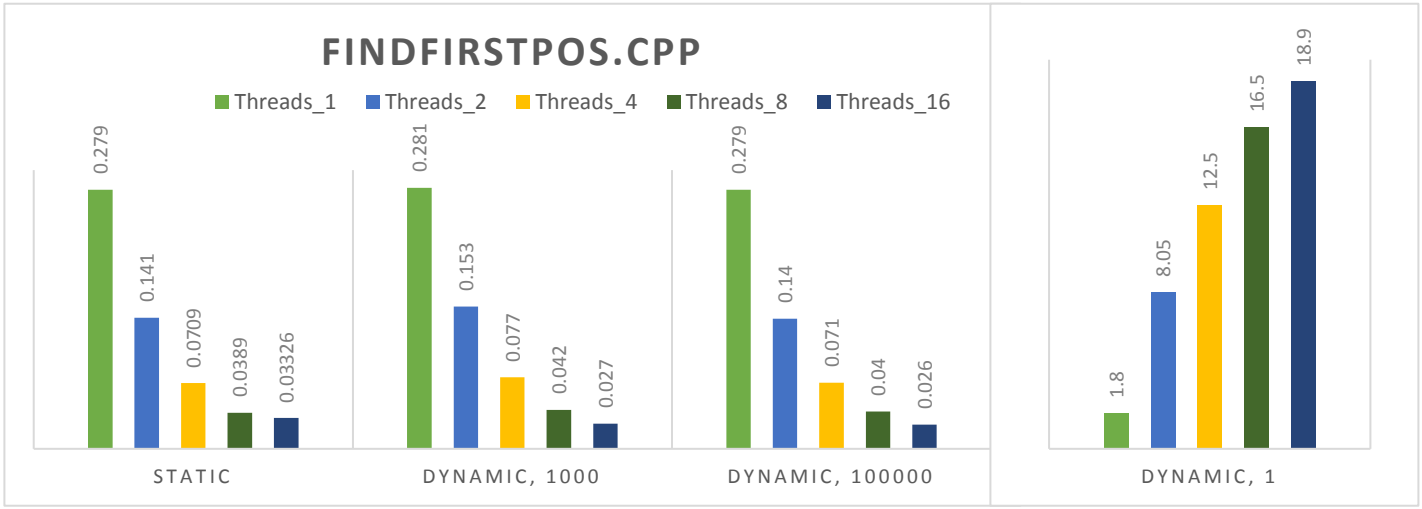
- Same explanation as above for (dynamic,1) consuming more time than expected
- Threads find findVal, and reduction(min:index) makes sure minimum of all indexes obtained by threads is selected. This way, given value with minimum index or which occurred for the first time is obtained
- Again static, dynamic, 1000 and dynamic, 100000 time consumption decreased with increase in number of threads as expected



4) Findfirstp.cpp to find given value (first found by any of the threads) array of random values (0 to N (i.e. size))

```
#include<stdio.h>
#include<iostream>
#include<omp.h>
#include<immintrin.h>
#include<iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

main()
{
        int size=100000000;
        int *randa = (int*)malloc(sizeof(int) * size); // size of each array 10^8
        int *randb = (int*)malloc(sizeof(int) * size);
        int findVal=64095060;
        int ii=0, index=size, flag=0;
        for(int j=0;j<size;j++)
        {
                randa[j]=rand()%size;
                randb[j]=randa[j];
         }

        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        for(int part=1;j<size;part=part*2)
        {
                #pragma omp parallel for schedule(static) firstprivate(part, flag) shared(findVal)
                for(ii=0;ii<part;ii++)
                {
                        if(findVal==randb[ii] && flag==0)
                        {//      #pragma omp atomic
                                index=ii;
                                flag=1;
                        }
                }
        }
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        duration<double> time_span = duration_cast<duration<double>> (t2 - t1);
        printf("time taken for %d threads is %lf\n",omp_get_max_threads(),time_span.count());
        printf("first index is %d\n",index);
        return 0;
}
```

- Logic goes, when thread finds findVal, it exits the for loop and the index of the found is printed
- Results are almost the same as above experiments



```
Randb[0] = 83
Randb[1] = 86
Randb[2] = 77
Randb[3] = 15
Randb[4] = 93
Randb[5] = 35
Randb[6] = 86
Randb[7] = 92
Randb[8] = 49
Randb[9] = 21
Randb[10] = 62
Randb[11] = 27
Randb[12] = 90
Randb[13] = 59
Randb[14] = 63
Randb[15] = 26
Randb[16] = 40
Randb[17] = 26
Randb[18] = 72
Randb[19] = 36
Randb[20] = 11
Randb[21] = 68
Randb[22] = 67
Randb[23] = 29
Randb[24] = 82
Randb[25] = 30
Randb[26] = 62
Randb[27] = 23
Randb[28] = 67
Randb[29] = 35
Randb[30] = 29
Randb[31] = 2
Randb[32] = 22
Randb[33] = 58
Randb[34] = 69
Randb[35] = 67
Randb[36] = 93
Randb[37] = 56
Randb[38] = 11
Randb[39] = 42
Randb[40] = 29
Randb[41] = 73
Randb[42] = 21
Randb[43] = 19
Randb[44] = 84
Randb[45] = 37
Randb[46] = 98
Randb[47] = 24
Randb[48] = 15
Randb[49] = 70
Randb[50] = 13
Randb[51] = 26
Randb[52] = 91
Randb[53] = 80
Randb[54] = 56
Randb[55] = 73
Randb[56] = 62
Randb[57] = 70
Randb[58] = 96
Randb[59] = 81
Randb[60] = 5
Randb[61] = 25
Randb[62] = 84
Randb[63] = 27
Randb[64] = 36
Randb[65] = 5
Randb[66] = 46
Randb[67] = 29
Randb[68] = 13
Randb[69] = 57
Randb[70] = 24
Randb[71] = 95
Randb[72] = 82
Randb[73] = 45
Randb[74] = 14
Randb[75] = 67
Randb[76] = 34
Randb[77] = 64
Randb[78] = 43
Randb[79] = 50
Randb[80] = 87
Randb[81] = 8
Randb[82] = 76
Randb[83] = 78
Randb[84] = 88
Randb[85] = 84
Randb[86] = 3
Randb[87] = 51
Randb[88] = 54
Randb[89] = 99
Randb[90] = 32
Randb[91] = 60
Randb[92] = 76
Randb[93] = 68
Randb[94] = 39
Randb[95] = 12
Randb[96] = 26
Randb[97] = 86
Randb[98] = 94
Randb[99] = 39
```

[mgudaram@mba-i1 Assignment3]$ OMP_NUM_THREADS=4 ./f2s
time taken for 4 threads is 0.167004
first index is 44

[mgudaram@mba-i1 Assignment3]$ OMP_NUM_THREADS=4 ./f2s
time taken for 4 threads is 0.169200
first index is 62

Out of 100, when 84 was given as value to be found, parallel threads exited when encountered 84 at index 44, second time at 62..hence concluded value found is of θ(pos)