ID: 800962452 **Meghana Gudaram** **HPC Assignment 3**

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

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

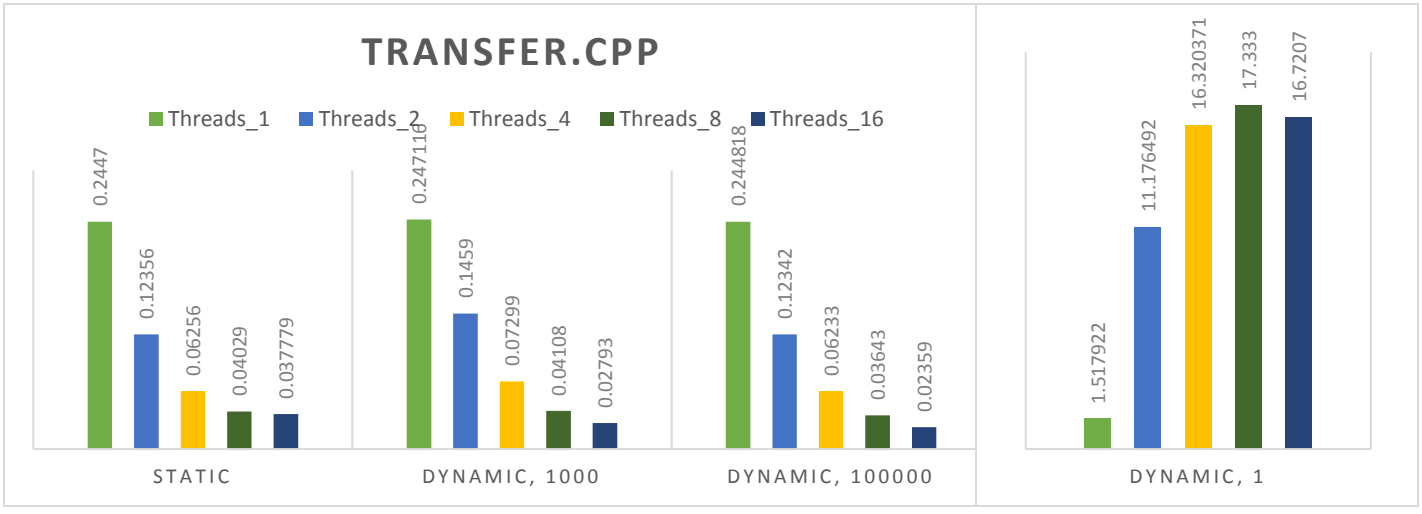
Reduction (operation : Variable):
Throughout reduction(min:value) is used to calculate minimum of value obtained by all threads, they all share value
Example: #pragma omp parallel for reduction(min:minVal)

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

1)

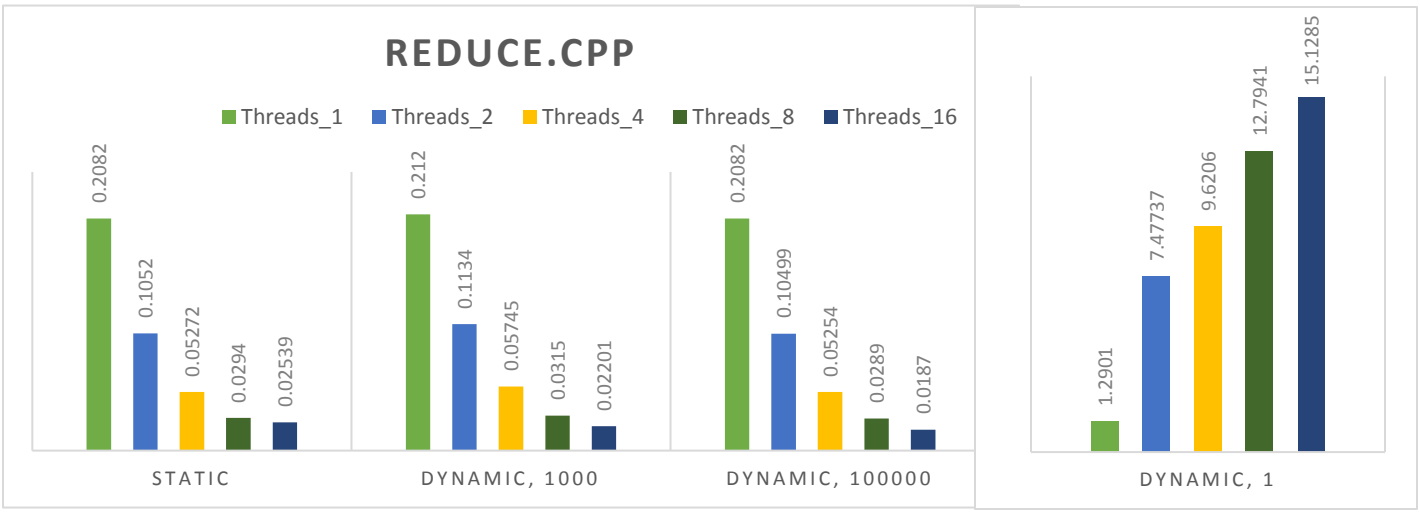
```
#include<stdio.h>
#include <omp.h>
#include <immintrin.h>
#include <iostream>
#include<chrono>
#include<ctime>

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,minVal=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();
        //omp_set_num_threads(4); //number of threads sent as parameter

        #pragma omp parallel for reduction(min:minVal) schedule(dynamic,100000)
        for(int ii=0;ii<size;ii++)
        {                if(minVal>randb[ii])
                {
                        minVal=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());
        printf("minimum value is %d\n",minVal);
        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



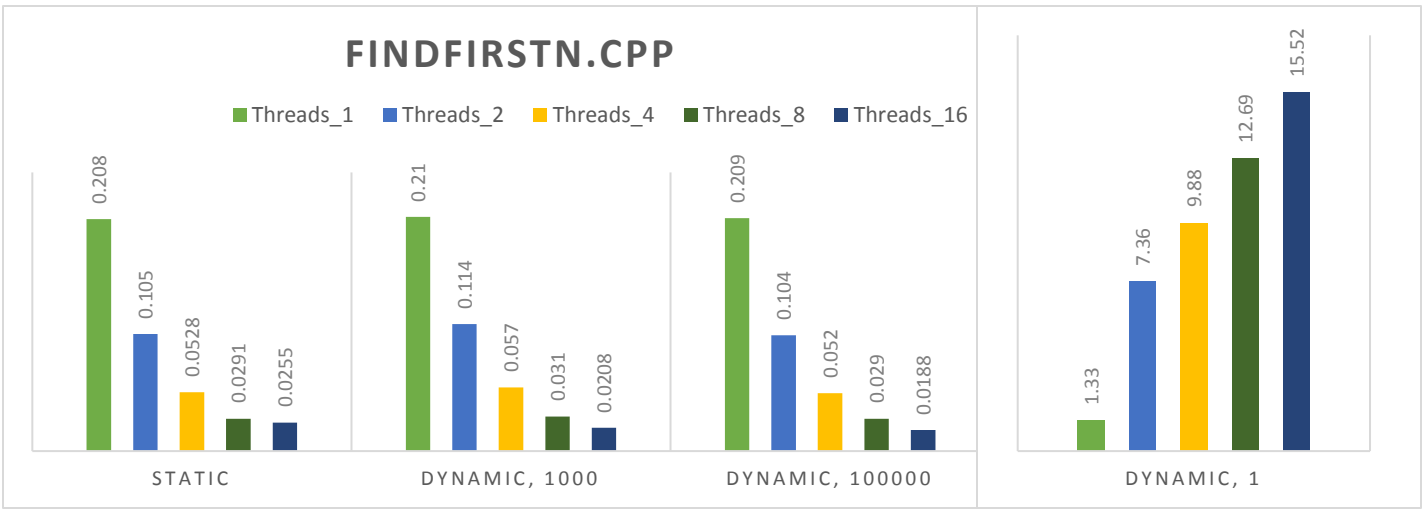
2)

```
#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;
}
```



3)

```
#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();

        //omp_set_num_threads(4);
        #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;
}
```



4)

```
#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=8905753;
        int ii=0,index=size;
        for(int j=0;j<size;j++)
        {
                randa[j]=rand()%size;
                randb[j]=randa[j];
         }

        #pragma omp parallel for schedule(static) shared(findVal) reduction(min:index)
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        for(ii=0;ii>0;ii++)
        {
                if(findVal==randb[ii])
                {//      #pragma omp atomic
                        index=ii;
                        ii=-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;
}
```