Sinhgad Institutes

Name of the Student: ___________________________________ **Roll No:** ___

CLASS: - B. E. [COMP] **Division:** **Course: LP-V**

High Performance Computing

Assignment No. 03

- **PARALLEL REDUCTION USING MIN, MAX, SUM AND AVERAGE**

Marks:

Date of Performance: **Sign with Date:**

Title: Implement Min, Max, Sum and Average operations using Parallel Reduction.

Objectives:

- To study and implementation of directive based parallel programming model.

Outcomes:

- Students will understand implementation of sequential program augmented with compiler directives to specify parallelism.

Pre-requisites:

64-bit Open source Linux or its derivative
Programming Languages: C/C++

PEOs, POs, PSOs and COs satisfied

PEOs: I, III POs: 1, 2, 3, 5 PSOs: 1 Cos: 1

Problem Statement:

Implement Min, Max, Sum and Average operations using Parallel Reduction.

Theory:

OpenMP:

- **OpenMP (Open Multi-Processing):**

It is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems. OpenMP provides a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications.

- **OpenMP consists of three primary API components:**

1.Compiler Directives
2.Runtime Library Routines
3.Environment Variables

I. Compiler Directives :

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag. OpenMP compiler directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

Compiler directives have the following syntax:

sentinel directive-name [clause, ...]

Example: #pragma omp parallel default(shared) private (beta,pi)

II. Run-time Library Routines:

The OpenMP API includes an ever-growing number of run-time library routines. These routines are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, t the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

Example : #include < omp.h >

int omp_get_num_threads(void)

○**OMP SET NUM THREADS :** Sets the number of threads that will be used in the next parallel region

○**OMP GET NUM THREADS :**Returns the number of threads that are currently in the team executing the parallel region from which it is called.

○**OMP GET THREAD NUM :**Returns the thread number of the thread, within the team, making this call.

III. Environment Variables :

OpenMP provides several environment variables for controlling the execution of parallel code at run-time. These environment variables can be used to control such things as:

- Setting the number of threads
- Specifying how loop iterations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

Example: export OMP_NUM_THREADS=8

1.**OMP NUM THREADS :** Sets the maximum number of threads to use during execution $setenvOMP_{N}UM_{T}HREADS8$

2.**OMP DYNAMIC :**Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE. $setenvOMP_{D}Y\ NAMICTRUE$

3.**OMP PROC BIND :**Enables or disables threads binding to processors. Valid values are TRUE or FALSE. $setenvOMP_{P}ROC_{B}INDTRUE$

4.**OMP NESTED:** Enables or disables nested parallelism. Valid values are TRUE or FALSE. $setenvOMP_{N}ESTEDTRUE$

All OpenMP directives in C and C++ are indicated with a # pragma omp followed by parameters, ending in a newline. The pragma usually applies only into the statement immediately following it.

The parallel pragma :

The parallel pragma starts a parallel block. It creates a team of N threads (where N is determined at runtime, usually from the number of CPU cores,), all of which execute the next statement (or the next block, if the statement is a -enclosure). After the statement, the threads join back into one.

```
#pragma omp parallel
{

// Code inside this region runs in parallel.

}
```

Reduction clause:

```
#pragma omp parallel for(op:var)
op à operator, varàvariable name (private)
```

- Performs a collective operation on variable according as per the given operators
- After executing the parallel instructions, var from each thread are combined to a single var, as per the operator

Following is the sample code which illustrates sum operation usage in OpenMP :

```
int sum=0;
#pragma omp parallel for reduction (+:sum)
    for (int i = 0; i < 4; i++)
            sum += i;
printf("sum= %d", sum);
```

Following is the sample code which illustrates mul operation usage in OpenMP :

```
int sum=10;
#pragma omp parallel for reduction (*:sum)
        for (int i = 0; i <4 ; i++)
                sum += i;
printf("sum= %d", sum);
```

Following is the sample code which illustrates max operation usage in OpenMP :

```
double arr[10];
double max_val=0.0;
int i;
for( i=0; i<10; i++)
        arr[i] = 2.0 + i;
omp_set_num_threads(4);
#pragma omp parallel for reduction(max : max_val)
for( i=0;i<10; i++)
{
        printf("thread id = %d and i = %d", omp_get_thread_num(), i);
        if(arr[i] > max_val)
                max_val = arr[i];
}
printf("\nmax_val = %f", max_val);
```

Following is the sample code which illustrates min operation usage in OpenMP :

```
double arr[10];
double min_val=0.0;
int i;
for( i=0; i<10; i++)
        arr[i] = 2.0 + i;
omp_set_num_threads(4);
#pragma omp parallel for reduction(min : min_val)
for( i=0;i<10; i++)
{
        printf("thread id = %d and i = %d", omp_get_thread_num(), i);
        if(arr[i] < min_val)
                min_val = arr[i];
}
printf("\nmin_val = %f", min_val);
```

Conclusion:

We have implemented parallel reduction using Sum ,Min,Max and Average operations.

Write short answer of following questions:

1. Write difference between concurrent and parallel programming.
2. What is OPENMP?
3. Explain the #pragma OMP parallel construct of OPENMP?

Code:-

```
#include <iostream>
#include<limits.h>
#include <vector>
#include <omp.h>
using namespace std;
void min_reduction(vector<int>& arr)
{
int min_value = INT_MAX;
#pragma omp parallel for reduction(min: min_value)
for (int i = 0; i < arr.size(); i++)
{
if (arr[i] < min_value)
{
min_value = arr[i];
}
}
cout << "Minimum value: " << min_value << endl;
}
void max_reduction(vector<int>& arr)
{
int max_value = INT_MIN;
#pragma omp parallel for reduction(max: max_value)
for (int i = 0; i < arr.size(); i++)
{
if (arr[i] > max_value)
{
max_value = arr[i];
}
}
cout << "Maximum value: " << max_value << endl;
}
void sum_reduction(vector<int>& arr) {
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < arr.size(); i++) {
sum += arr[i];
}
cout << "Sum: " << sum << endl;
}
void average_reduction(vector<int>& arr) {
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < arr.size(); i++) {
```

```
sum += arr[i];
}
cout << "Average: " << (double)sum / arr.size() << endl;
}
int main() {
vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
min_reduction(arr);
max_reduction(arr);
sum_reduction(arr);
average_reduction(arr);
}
```

Steps to compile and run

```
C:/MinGw/Bin>g++ -fopenmp reduction.cpp
C:/MinGw/Bin>a
```