

410255:LP-V. HPC	
Experiment No: 3	Min, Max, Sum and Average operations using Parallel Reduction.

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

Theory :

array:


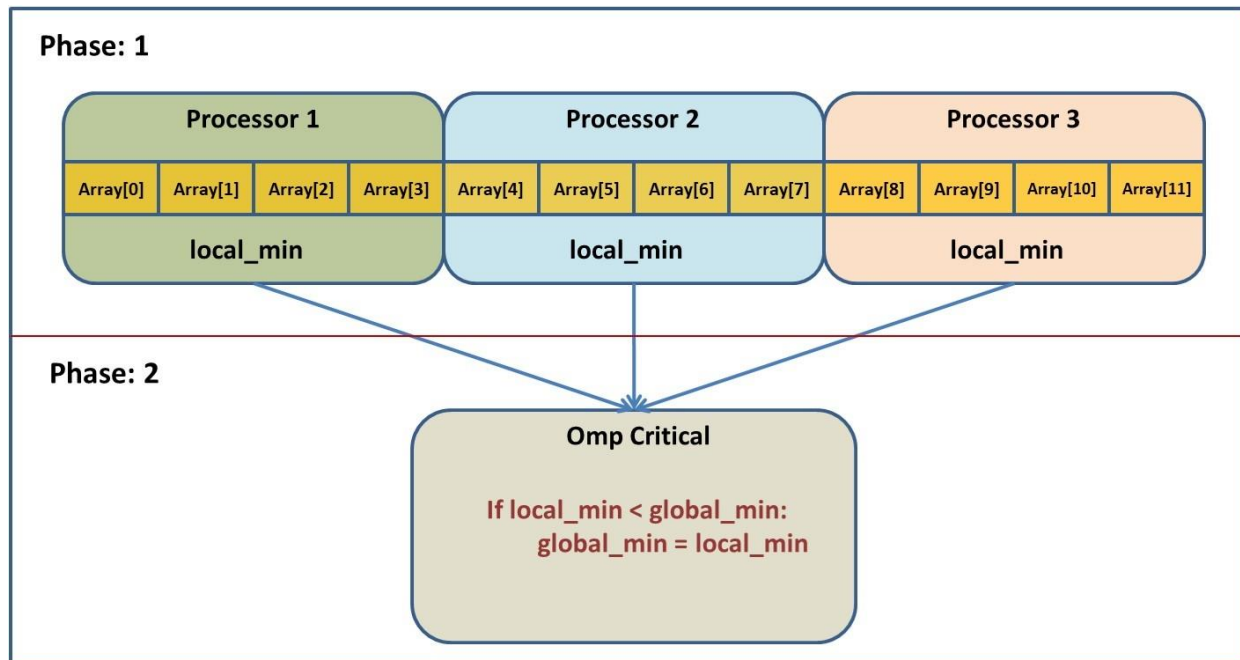
Loop step	Index	array[index]	Smallest
initialize			5.0  array[0]
first	1	10.3	5.0
second	2	8.7	5.0
third	3	5.0	5.0
fourth	4	52.9	5.0
fifth	5	18.0	5.0
sixth	6	13.0	5.0
seventh	7	2.3	2.3
eighth	8	82.7	2.3
ninth	9	68.2	2.3

Table 1: Summing integers in an array.

To implement a parallel version, you will use domain decomposition, also sometimes called data decomposition, to find the smallest (min) and largest (max) value of the array in parallel. Domain decomposition requires dividing the array into equal parts and assigning each part to a processor. Consider the simple example of 12 numbers and three processors. The computation occurs in two phases. First, the work is divided equally among the three processors. In this case, each processor finds the min and max of four numbers in the array. The first processor finds the min/max value among the elements starting at index 0 and ending at index 3, the second processor finds the min/max value among the elements starting at index 4 and ending at index 7, and the third processor finds the min/max value among the elements starting at index 8 and ending at index 11.

The second phase occurs after all of the processor are finished with the first phase. In this phase, each processor compares its min/max to the global min/max and updates the global values if needed.



The first thing you will need is the following:

```
// define gmax and gmin, the global min and max variables, here...
#pragma omp parallel
{
    // declare local min and max here...
    ...
}
```

This creates a number of threads that will each run what is in the braces.

Each thread will need a local min and max variable. If you define the variable inside of the braces then each thread gets its own copy. If it is defined before the parallel section then by default there is only one variable and all the threads can access it. We want the first option.

Now we want to split up the array so that each thread gets a portion to search through. This can be done using:

```
#pragma omp for
for(int i = 0; i < ...) {
    // put code to update local min and max here...
}
```

Putting this pragma before a loop will assign each thread an equal portion of the indices looped over.

Now, inside the FOR loop, we can read the values and update the local min and max as needed.

The final thing to do is to combine the local mins and maxes into a single total min and max. The potential problem here is that there is only one global min (and max) variable. If multiple threads try to change the value then they will create a *race condition*. In other words, the final value depends on the order of the writes. In fact, if you run the program multiple times, you can get different answers each time! The solution to this is to use a critical section:

```
#pragma omp critical
{
    // put code here to update gmax and gmin
}
```

The code inside the braces is executed by all the threads, but only by one thread at a time. We can compare the local min/max to the global min/max and update the global values if needed inside this section to avoid the issues mentioned.

To see if your parallel version of this program is any faster, add some timing code. OpenMP includes a function to get the current time. Its prototype is:

```
double omp_get_wtime( ).
```

Put a call to this function before and after your parallel code, then print the difference, i.e.

```
double start = omp_get_wtime();
// Do all the parallel stuff above right here...
double end = omp_get_wtime();
std::cout << "The time was " << end-start << std::endl;
```

Min, Max, Sum & Average operations using Parallel Reduction

```
#include<stdio.h>

#include<omp.h>

int main()
{
    omp_set_num_threads(4);

    double arr[10]={1,2,3,4,5,6,7,8,9,10};

    double max_val=0.0;
    double min_val=100.0;

    float avg=0.0,sum=0.0,sum_val=0.0;

    int i;

    #pragma omp parallel for reduction(min:min_val)
    for(i=0;i<10;i++)
    {
        printf("thread id = %d and i = %d \n", omp_get_thread_num(),i);
        if(arr[i] <min_val)
        {
            min_val = arr[i];
        }
    }

    printf("min_val = %f", min_val);
    printf("\n");
    printf("\n");

    #pragma omp parallel for reduction(max:max_val)
```

```

for(i=0;i<10;i++)
{
printf("thread id = %d and i = %d \n", omp_get_thread_num(),i);
    if(arr[i]>max_val)
    {
max_val = arr[i];
    }
}
printf("max_val = %f", max_val);
printf("\n");
printf("\n");

```

```

#pragma omp parallel for reduction(+:sum_val)
for(i=0;i<10;i++)
{
printf("thread id = %d and i = %d \n", omp_get_thread_num(),i);
sum_val=sum_val+arr[i];
}
printf("sum_val = %f", sum_val);
printf("\n");
printf("\n");

```

```

#pragma omp parallel for reduction(+:sum)
for(i=0;i<10;i++)
{
printf("thread id = %d and i = %d \n", omp_get_thread_num(),i);
    sum=sum+arr[i];
}

```

```
}  
avg=sum/10;  
printf("avg_val = %f", avg);  
printf("\n");  
printf("\n");  
}
```

Output –

```
thread id = 3 and i = 8  
thread id = 3 and i = 9  
thread id = 1 and i = 3  
thread id = 1 and i = 4  
thread id = 1 and i = 5  
thread id = 2 and i = 6  
thread id = 2 and i = 7  
thread id = 0 and i = 0  
thread id = 0 and i = 1  
thread id = 0 and i = 2  
min_val = 1.000000
```

```
thread id = 1 and i = 3  
thread id = 1 and i = 4  
thread id = 1 and i = 5  
thread id = 3 and i = 8  
thread id = 3 and i = 9  
thread id = 0 and i = 0  
thread id = 0 and i = 1  
thread id = 0 and i = 2
```

thread id = 2 and i = 6

thread id = 2 and i = 7

max_val = 10.000000

thread id = 3 and i = 8

thread id = 3 and i = 9

thread id = 2 and i = 6

thread id = 2 and i = 7

thread id = 1 and i = 3

thread id = 1 and i = 4

thread id = 1 and i = 5

thread id = 0 and i = 0

thread id = 0 and i = 1

thread id = 0 and i = 2

sum_val = 55.000000

thread id = 1 and i = 3

thread id = 1 and i = 4

thread id = 1 and i = 5

thread id = 0 and i = 0

thread id = 0 and i = 1

thread id = 0 and i = 2

thread id = 3 and i = 8

thread id = 3 and i = 9

thread id = 2 and i = 6

thread id = 2 and i = 7

avg_val = 5.500000

