**Final Year B. Tech., Sem VI 2021-22**

**High Performance Computing Lab**

**Assignment submission**

**PRN No: 2019BTECS00070**

**Full name: Prathmesh Basaweshwar Killedar**

**Batch: B3**

**Assignment: 2**

**SPMD: Single Program Multiple Data**

It is a parallel programming style. In SPMD style tasks are split up and run simultaneously on multiple processors with different data. This is done to achieve results faster.

**Worksharing**

Threads are assigned, an independent subset of the total workload For example, different chunks of an iteration are distributed among the threads.

eg.

T1 -> iteration from 1 to 5

T2 -> iteration from 6 to 10 and so on…

OpenMP provides various constructs for worksharing.

**OpenMP loop worksharing construct**

OpenMP’s loop worksharing construct splits loop iterations among all active threads

*#pragma omp for*

**Types of variables**

**1. Shared Variables :**

There exist one instance of this variable which is shared among all threads

**2. Private Variables :**

Each thread in a team of threads has its own local copy of the private variable

Two ways to assign variables as private ans shared are

**Implicit and Explicit**

**Implicit :** All the variables declared outside of the pragma are by default shared and all the variables declared inside pragma are private

**Explicit:**

**Shared Clause**

eg. *#pragma omp parallel for shared(n, a) =>* n and a are declared as shared variables

**Private Clause**

eg. *#pragma omp parallel for shared(n, a) private(c) =>* here c is private variable

**Default Clause**

eg. *#pragma omp parallel for default(shared) =>*  now all variables are shared

*#pragma omp parallel for default(private) =>*  now all variables are private

**firstprivate**

firstprivate make the variable private but that variable is initialised with the value that it has before the parallel region

**lastprivate**

lastprivate make the variable private but it retain the last value of that private variable outside of the private region

**Schedule**

a specification of how iterations of associated loops are divided into contiguous

non-empty subsets.

syntax: *#pragma omp parallel for schedule([modifier [modifier]:]kind[,chunk\_size])*

Five kinds of schedules for OpenMP loop:

• static

• dynamic

• guided

• auto

• runtime

**Static Schedule**

In static scheduling openMP assingn interations to threads in a cyclic order

eg: *#pragma omp parallel for schedule(static,1)*

=> Number “1” indicates that we assign one iteration to each thread before switching to the next thread — we use chunks of size 1.

**Dynamic Schedule**

In dynamic scheduling, each thread will take one iteration, process it, and then see what is the next iteration that is currently not being processed by anyone. This way it will never happen that one thread finishes while other threads have still lots of work to do:

eg: *#pragma omp parallel for schedule(dynamic,1)*

**nowait**

When we use a parallel region, OpenMP will automatically wait for all threads to finish before execution continues. There is also a synchronization point after each omp for loop

However, if we do not need synchronization after the loop, we can disable it with nowait

eg: *#pragma omp for nowait*

**reduction**

The OpenMP reduction clause lets you specify one or more thread-private variables that are subject to a reduction operation at the end of the parallel region.

eg: if we want to calculate sum of first 100 natural numbers and want to save it into variable 'sum', we can save individual thread sums and then add all sum to get resultant sum.

Or we can use reduction clause and it will automatically handle each threads sum and then the final sum, syntax is

*#pragma omp for reduction(+:sum)*

Problems:

1. vector vector addition

#include <omp.h>

#include <stdio.h>

void main()

{

printf("Adding Two Arrays\n");

int a1[] = {1, 2, 3, 4, 5};

int a2[] = {11, 12, 13, 14, 15};

int a3[5];

int i;

#pragma omp parallel

for (i = 0; i < 5; i++)

{

a3[i] = a1[i] + a2[i];

}

for (i = 0; i < 5; i++)

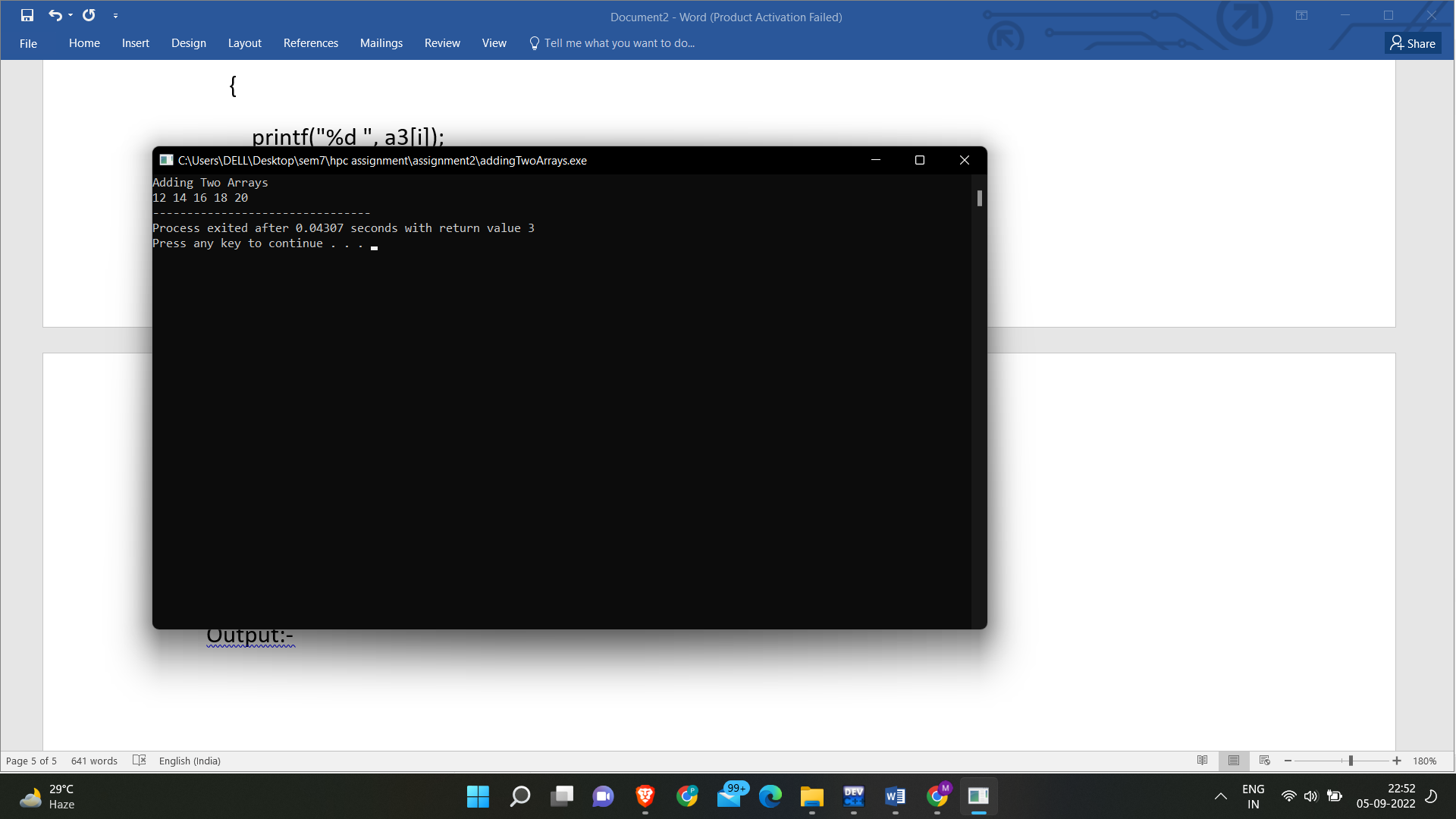
{

printf("%d ", a3[i]);

}

}

Output:-



Vector Scaler Addition:-

Code:-

#include <omp.h>

#include <stdio.h>

void main()

{

printf("Adding Vector into Scalar\n");

int a1[] = {1, 2, 3, 4, 5};

int answer = 0;

int i;

#pragma omp parallel

for (i = 0; i < 5; i++)

{

answer += a1[i];

}

printf("Answer %d ", answer);

}

Output:-

