**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 21510036**

**Title of practical:**

Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

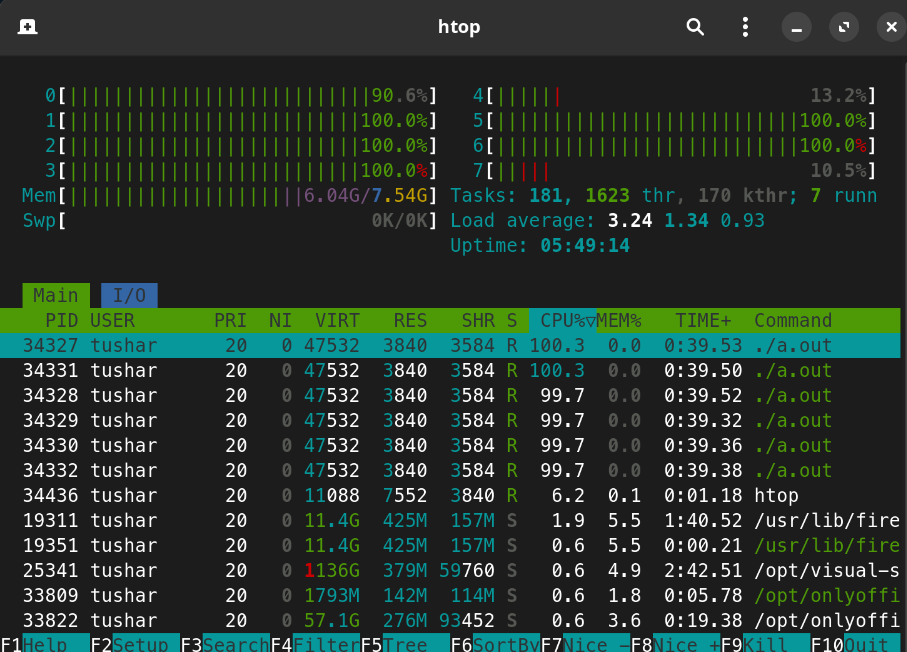
**Problem Statement 1:**

Analyse and implement a Parallel code for below program using OpenMP.

// C Program to find the minimum scalar product of two vectors (dot product)

**Screenshots:**

**Resource Consumption due to use of large values on 6 threads..**



#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <limits.h>

#define N 10 *// Adjusted size for demonstration*

void permutation(int \**array*, int *start*, int *end*, int \**min\_dot\_product*, int \**vector\_b*, int \**vector\_a*) {

if (*start* == *end*) {

int dot\_product = 0;

#pragma omp parallel for reduction(+:dot\_product)

for (int i = 0; i < N; i++) {

dot\_product += *vector\_a*[i] \* *vector\_b*[i];

}

#pragma omp critical

{

if (dot\_product < \**min\_dot\_product*) {

\**min\_dot\_product* = dot\_product;

}

}

} else {

for (int i = *start*; i <= *end*; i++) {

*// Swap*

int temp = *array*[*start*];

*array*[*start*] = *array*[i];

*array*[i] = temp;

permutation(*array*, *start* + 1, *end*, *min\_dot\_product*, *vector\_b*, *vector\_a*);

*// Swap back*

temp = *array*[*start*];

*array*[*start*] = *array*[i];

*array*[i] = temp;

}

}

}

int main() {

int vector\_a[N], vector\_b[N];

int min\_dot\_product = INT\_MAX;

for (int i = 0; i < N; i++) {

vector\_a[i] = i;

vector\_b[i] = i;

}

int perm[N];

for (int i = 0; i < N; i++) {

perm[i] = i;

}

omp\_set\_num\_threads(8);

double start = omp\_get\_wtime();

permutation(perm, 0, N - 1, &min\_dot\_product, vector\_b, vector\_a);

double end = omp\_get\_wtime();

printf("Minimum scalar product: %d\n", min\_dot\_product);

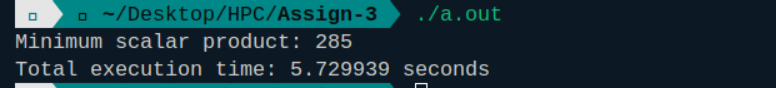
printf("Total execution time: %f seconds\n", end - start);

return 0;

}

**Information and analysis:**

**1 – Permution here is an highly time consuming process so took the vector size as 10 to try all possible permutes and finding the minimum scalar value. Took vector a as [0,1,2,3,4,5,6,7,8,9] and vector b as [0,1,2 ,3 4, 5, 6, 7, 8, 9] and manually we know the value for minimum scalar is 285 and program also give 285 which is correct.**



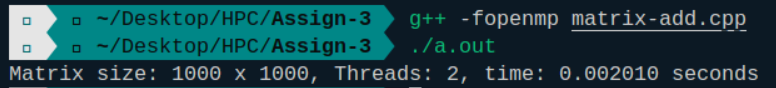
**Problem Statement 2:**

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

ii. Explain whether or not the scaling behaviour is as expected.

**Screenshots:**



**Information and analysis:**

**1 - Matrix size: 10000 x 10000, Threads: 2, time: 0.407307 seconds**

**2 - Matrix size: 10000 x 10000, Threads: 4, time: 0.285126 seconds**

**3 - Matrix size: 10000 x 10000, Threads: 8, time: 0.204092 seconds**

**4 - Matrix size: 10000 x 10000, Threads: 16, time: 0.181238 seconds**

**1 - Matrix size: 2000 x 2000, Threads: 2, time: 0.006306 seconds**

**2 - Matrix size: 2000 x 2000, Threads: 4, time: 0.003910 seconds**

**3 - Matrix size: 2000 x 2000, Threads: 8, time: 0.003646 seconds**

**4 - Matrix size: 2000 x 2000, Threads: 16, time: 0.004124 seconds**

**1 – Matrix size: 1000 x 1000, Threads: 2, time: 0.002010 seconds**

**2 - Matrix size: 1000 x 1000, Threads: 4, time: 0.001512 seconds**

**3 - Matrix size: 1000 x 1000, Threads: 8, time: 0.001022 seconds**

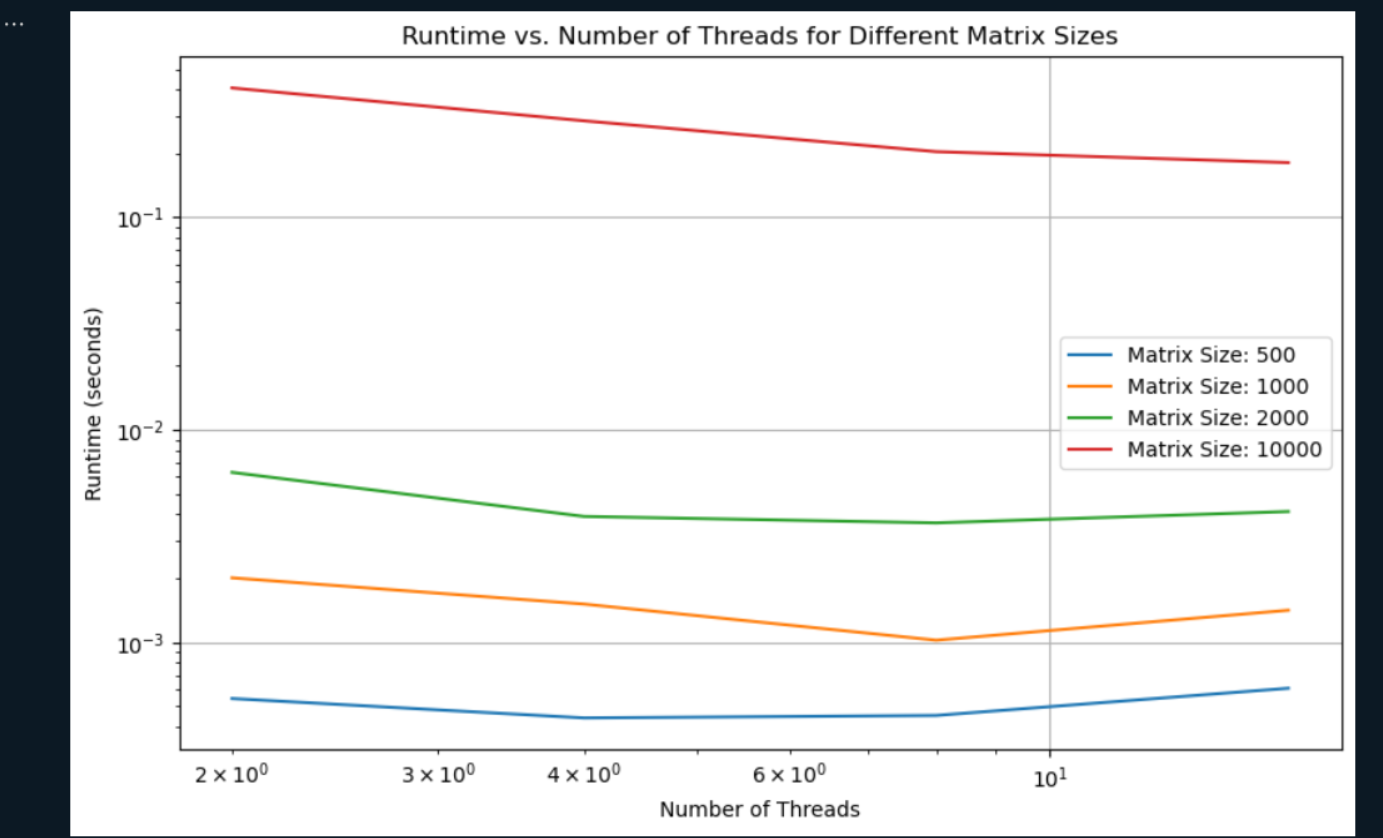
**4 - Matrix size: 1000 x 1000, Threads: 16, time: 0.001414 seconds**

**1 - Matrix size: 500 x 500, Threads: 2, time: 0.000543 seconds**

**2 - Matrix size: 500 x 500, Threads: 4, time: 0.000440 seconds**

**3 - Matrix size: 500 x 500, Threads: 8, time: 0.000452 seconds**

**4 - Matrix size: 500 x 500, Threads: 16, time: 0.000607 seconds**



The observed results indicate that as the matrix size increases, the benefit from additional threads becomes more pronounced, improving performance and reducing runtime. For smaller matrices (500x500 and 1000x1000), the computation time is minimal, and the overhead of managing threads becomes relatively more significant, leading to less impressive speedup or even slower performance with more threads. In contrast, for larger matrices (2000x2000 and 10000x10000), the computational workload is high enough that the parallel execution effectively reduces runtime with increasing threads, demonstrating more noticeable speedup. However, there is a point of diminishing returns where adding more threads yields increasingly marginal benefits due to factors like memory bandwidth limits and thread management overhead. Thus, while parallelism scales well with matrix size, the efficiency gains can plateau or decline with excessive threads.

**Problem Statement 3:**

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following:

i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup.

ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup.

iii. Demonstrate the use of nowait clause.

**Screenshots:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define VECTOR\_SIZE 200

#define SCALAR 5

void vector\_addition\_static(int *chunk\_size*) {

int vector[VECTOR\_SIZE];

double start\_time, end\_time;

for (int i = 0; i < VECTOR\_SIZE; i++) {

vector[i] = rand() % 100;

}

omp\_set\_schedule(omp\_sched\_static, *chunk\_size*);

start\_time = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

for (int i = 0; i < VECTOR\_SIZE; i++) {

vector[i] += SCALAR;

}

end\_time = omp\_get\_wtime();

printf("STATIC Schedule, Chunk Size: %d, Runtime: %f seconds\n", *chunk\_size*, end\_time - start\_time);

}

void vector\_addition\_dynamic(int *chunk\_size*) {

int vector[VECTOR\_SIZE];

double start\_time, end\_time;

for (int i = 0; i < VECTOR\_SIZE; i++) {

vector[i] = rand() % 100;

}

omp\_set\_schedule(omp\_sched\_dynamic, *chunk\_size*);

start\_time = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

for (int i = 0; i < VECTOR\_SIZE; i++) {

vector[i] += SCALAR;

}

end\_time = omp\_get\_wtime();

printf("DYNAMIC Schedule, Chunk Size: %d, Runtime: %f seconds\n", *chunk\_size*, end\_time - start\_time);

}

void vector\_addition\_nowait() {

int vector[VECTOR\_SIZE];

double start\_time, end\_time;

for (int i = 0; i < VECTOR\_SIZE; i++) {

vector[i] = rand() % 100;

}

start\_time = omp\_get\_wtime();

#pragma omp parallel

{

#pragma omp for nowait

for (int i = 0; i < VECTOR\_SIZE; i++) {

vector[i] += SCALAR;

}

#pragma omp single

{

printf("Vector addition with nowait completed.\n");

}

}

end\_time = omp\_get\_wtime();

printf("NOWAIT Clause, Runtime: %f seconds\n", end\_time - start\_time);

}

int main() {

int chunk\_sizes[] = {1, 10, 50, 100}; *// Different chunk sizes to test*

int num\_chunks = sizeof(chunk\_sizes) / sizeof(chunk\_sizes[0]);

printf("Testing STATIC Scheduling:\n");

for (int i = 0; i < num\_chunks; i++) {

vector\_addition\_static(chunk\_sizes[i]);

}

printf("Testing DYNAMIC Scheduling:\n");

for (int i = 0; i < num\_chunks; i++) {

vector\_addition\_dynamic(chunk\_sizes[i]);

}

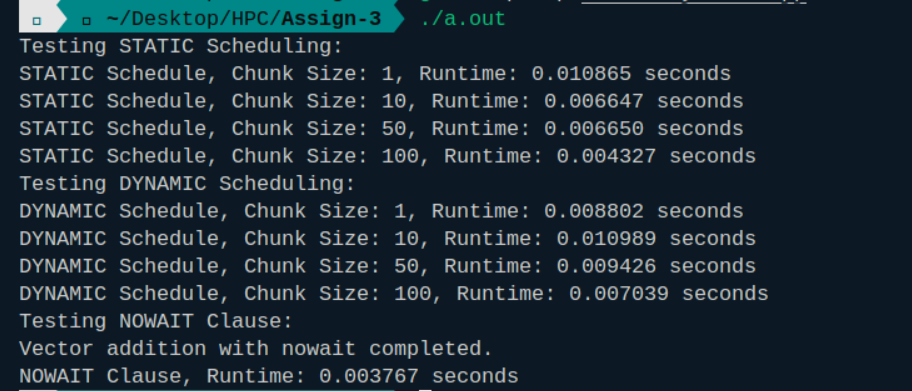
printf("Testing NOWAIT Clause:\n");

vector\_addition\_nowait();

return 0;

}

**Output -**



**Information and analysis:**

### **Static Scheduling**

* **Description:** In static scheduling, the iterations of a loop are divided into chunks of equal size and assigned to threads before the loop starts. Each thread gets a predefined portion of iterations based on the chunk size.
* **Use Case:** It is suitable for loops with uniform iteration time where the load is evenly distributed among threads.
* **Advantage:** Minimizes overhead due to the lack of runtime decisions.
* **Disadvantage:** May lead to load imbalance if iterations have varying execution times.

### **Dynamic Scheduling**

* **Description:** In dynamic scheduling, iterations are assigned to threads at runtime. Each thread requests iterations from a queue as it finishes its current work. The size of these chunks is determined by the chunk size.
* **Use Case:** Useful for loops with varying iteration times or unpredictable workloads, allowing threads to dynamically balance the load.
* **Advantage:** Better load balancing for irregular workloads.
* **Disadvantage:** Can introduce overhead due to runtime management of iterations.

### **Nowait Clause**

* **Description:** The nowait clause in OpenMP allows threads to proceed with the next section of code without waiting for a preceding for or sections construct to complete.
* **Use Case:** Useful when subsequent code sections are independent of the for or sections construct and can be executed in parallel without synchronization.
* **Advantage:** Reduces synchronization overhead and improves parallel efficiency by allowing concurrent execution of independent tasks.

**Github Link:**