**Name: Shreeshail Mahajan**

**PRN: 2020BTECS00055**

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

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 2020BTECS00055**

**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.

**Serial Implementation of program for calculating minimum scalar product.**

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

#include <stdio.h>

#include <time.h> // for clock\_t, clock(), CLOCKS\_PER\_SEC

int sort(int arr[], int n)

{

    int i, j;

    for (i = 0; i < n - 1; i++)

    {

        for (j = 0; j < n - i - 1; j++)

        {

            if (arr[j] > arr[j + 1])

            {

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

int sort\_des(int arr[], int n)

{

    int i, j;

    for (i = 0; i < n; ++i)

    {

        for (j = i + 1; j < n; ++j)

        {

            if (arr[i] < arr[j])

            {

                int a = arr[i];

                arr[i] = arr[j];

                arr[j] = a;

            }

        }

    }

}

int main()

{

    double time\_spent = 0.0;

    clock\_t begin = clock();

    // fill the code;

    int n = 5;

    // printf("\nEnter number of elements: ");

    // scanf("%d",&n);

    int arr1[5] = {2, 1, 4, 6, 3};

    int arr2[5] = {2, 1, 5, 4, 3};

    int i;

    sort(arr1, n);

    sort\_des(arr2, n);

    int sum = 0;

    for (i = 0; i < n; i++)

    {

        sum = sum + (arr1[i] \* arr2[i]);

    }

    printf("\nMinimum scalar product is: ");

    printf("%d", sum);

    clock\_t end = clock();

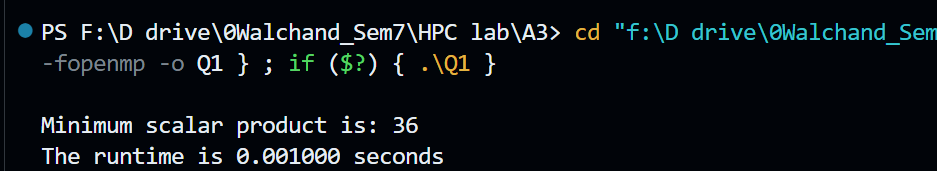
    time\_spent += (double)(end - begin) / CLOCKS\_PER\_SEC;

    printf("\nThe runtime is %f seconds", time\_spent);

    return 0;

}

**Screenshots:**

****

**Information:**

**Key Features:**

* Sequential Sorting: The code employs sequential sorting techniques (Bubble Sort) to sort two arrays (`nums1` and `nums2`) in ascending and descending order, respectively.
* Timing Measurement: The code measures the time taken for both sorting operations and calculates the total time elapsed.
* Vector Dot Product: After sorting, the code computes the dot product of the two sorted arrays to find the scalar product.

**Analysis:**

* **Sequential Sorting:** The code uses a simple bubble sort algorithm for sorting the arrays. Bubble sort is straightforward to implement but not the most efficient sorting algorithm for large datasets. More efficient sorting algorithms like quicksort or mergesort would be recommended for larger arrays.
* **Timing Measurement:** Timing measurements are included to evaluate the performance of the sorting and dot product calculations. These measurements are essential for assessing the code's efficiency and identifying potential areas for optimization.
* **Vector Dot Product:** The dot product is calculated sequentially, which means it's not parallelized in this code. If you have access to multi-core processors, parallelizing the dot product calculation could potentially speed up the overall computation.
* **Input Data:** The code initializes `nums1` and `nums2` arrays with constant values (10 and 20, respectively). In practice, you would typically read data from external sources, making the code more versatile.
* **Efficiency Consideration:** Bubble sort is not the most efficient sorting algorithm for large arrays. Its time complexity is O(n^2), making it slow for large `n` values. Consider using more efficient sorting algorithms for larger datasets.

**Overall Performance:** The code sequentially sorts two arrays and calculates their dot product to find the minimum scalar product. It serves as a baseline for performance comparison against parallelized versions of the same problem, showing the potential benefits of parallel processing.

**Parallel implementation of program for calculating minimum scalar product.**

#include <omp.h>

#include <stdio.h>

#include <time.h>

#define n 100000

void sort(int nums[])

{

  int i, j;

  for (i = 0; i < n; i++)

  {

    int turn = i % 2;

#pragma omp parallel for

    for (j = turn; j < n - 1; j += 2)

      if (nums[j] > nums[j + 1])

      {

        int temp = nums[j];

        nums[j] = nums[j + 1];

        nums[j + 1] = temp;

      }

  }

}

void sort\_des(int nums[])

{

  int i, j;

  for (i = 0; i < n; ++i)

  {

    int turn = i % 2;

#pragma omp parallel for

    for (j = turn; j < n - 1; j += 2)

    {

      if (nums[j] < nums[j + 1])

      {

        int temp = nums[j];

        nums[j] = nums[j + 1];

        nums[j + 1] = temp;

      }

    }

  }

}

int main()

{

  int nums1[n], nums2[n];

  for (int i = 0; i < n; i++)

  {

    nums1[i] = 10;

  }

  for (int i = 0; i < n; i++)

  {

    nums2[i] = 20;

  }

  clock\_t t;

  t = clock();

  sort(nums1);

  sort\_des(nums2);

  t = clock() - t;

  double time\_taken = ((double)t) / CLOCKS\_PER\_SEC;

  printf("Time taken (seq): %f\n", time\_taken);

  int sum = 0;

  for (int i = 0; i < n; i++)

  {

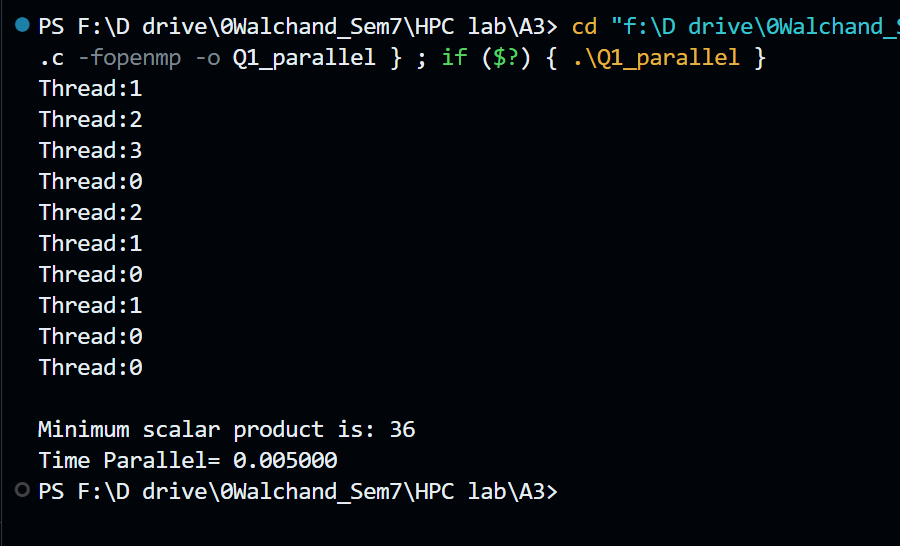
    sum = sum + (nums1[i] \* nums2[i]);

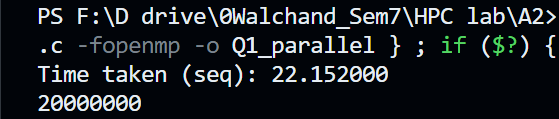
  }

  printf("%d\n", sum);

  return 0;

}

****

****

**Information:**

**Key Features:**

**1. Parallel Sorting:** The code employs parallel sorting techniques using OpenMP to sort two arrays (`nums1` and `nums2`) in both ascending and descending order. Parallel sorting can be beneficial for large arrays on multi-core processors.

**2. Timing Measurement:** The code measures the time taken for both sorting operations and calculates the total time elapsed.

**3. Vector Dot Product:** It computes the dot product of the two sorted arrays after sorting and calculates the scalar product.

**Analysis:**

* **Parallel Sorting:** Parallel sorting is a suitable approach for optimizing performance when dealing with large arrays. In this code, sorting is parallelized using OpenMP directives, which can lead to substantial speedup on multi-core processors. However, the effectiveness of parallel sorting depends on the size of the input data and the number of available CPU cores.
* **Timing Measurement:** The code includes timing measurements to assess the performance of the sorting operations. Timing measurements are essential for evaluating the impact of parallelism and can help identify performance bottlenecks.
* **Vector Dot Product:** After sorting, the code computes the dot product of the two sorted vectors. The dot product operation itself is not parallelized in this code, but it can potentially benefit from parallelization if needed.
* **Input Data:** The code initializes `nums1` and `nums2` arrays with constant values (10 and 20, respectively). In a real-world scenario, you would typically read data from external sources.
* **Efficiency Consideration:** The efficiency of parallel sorting depends on various factors, including the size of the arrays and the number of available CPU cores. For small arrays or single-core systems, parallel sorting might not provide a significant performance improvement.

**Sequential vs. Parallel:** The code measures and prints the time taken for sorting and dot product calculations sequentially. It allows you to compare the performance of the parallel sorting approach against the sequential one.

**Information 1:** The default clause explicitly determines the data-sharing attributes of variables that are referenced in a parallel, teams, or task generating construct and would otherwise be implicitly determined. The default (shared) clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared. The default (none) clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its datasharing attribute explicitly determined by being listed in a data-sharing attribute clause.

**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.

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 500

void add(int \*\*a, int \*\*b, int \*\*c)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            c[i][j] = a[i][j] + b[i][j];

        }

    }

}

void input(int \*\*a, int num)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            a[i][j] = num;

        }

    }

}

void display(int \*\*a)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            printf("%d ", a[i][j]);

        }

        printf("\n");

    }

}

int main()

{

    int \*\*a = (int \*\*)malloc(sizeof(int \*) \* N);

    int \*\*b = (int \*\*)malloc(sizeof(int \*) \* N);

    int \*\*c = (int \*\*)malloc(sizeof(int \*) \* N);

    for (int i = 0; i < N; i++)

    {

        a[i] = (int \*)malloc(sizeof(int) \* N);

        b[i] = (int \*)malloc(sizeof(int) \* N);

        c[i] = (int \*)malloc(sizeof(int) \* N);

    }

    input(a, 1);

    input(b, 1);

    double start = omp\_get\_wtime();

    add(a, b, c);

    double end = omp\_get\_wtime();

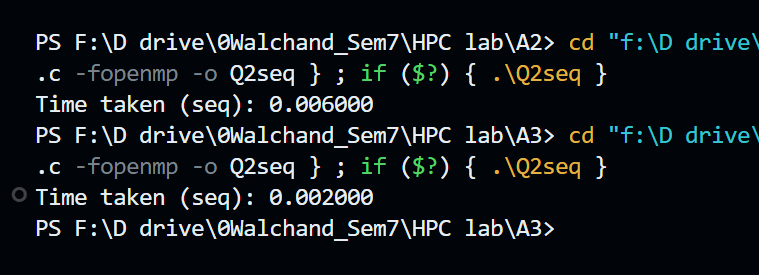
    // display(c);

    printf("Time taken (seq): %f\n", end - start);

}

**Screenshots:**

**For 1000 and 500 resp.**

****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 250 | 500 | 750 | 1000 | 2000 |
| Time | 0.000000 | 0.001000 | 0.003000 | 0.005000 | 0.027000 |

**Parallel Code:**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 1000

void add(int \*\*a, int \*\*b, int \*\*c)

{

#pragma omp parallel for

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            c[i][j] = a[i][j] + b[i][j];

        }

    }

}

void input(int \*\*a, int num)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            a[i][j] = num;

        }

    }

}

void displayMatrix(int \*\*a)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            printf("%d ", a[i][j]);

        }

        printf("\n");

    }

}

int main()

{

    int \*\*a = (int \*\*)malloc(sizeof(int \*) \* N);

    int \*\*b = (int \*\*)malloc(sizeof(int \*) \* N);

    int \*\*c = (int \*\*)malloc(sizeof(int \*) \* N);

    for (int i = 0; i < N; i++)

    {

        a[i] = (int \*)malloc(sizeof(int) \* N);

        b[i] = (int \*)malloc(sizeof(int) \* N);

        c[i] = (int \*)malloc(sizeof(int) \* N);

    }

    input(a, 1);

    input(b, 1);

    double start = omp\_get\_wtime();

    add(a, b, c);

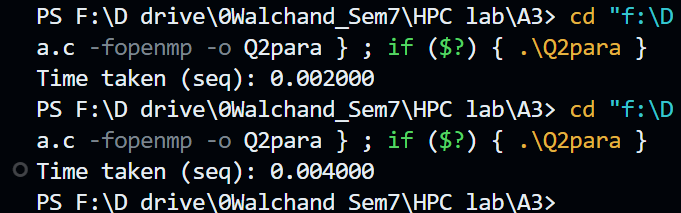
    double end = omp\_get\_wtime();

    // display(c);

    printf("Time taken (seq): %f\n", end - start);

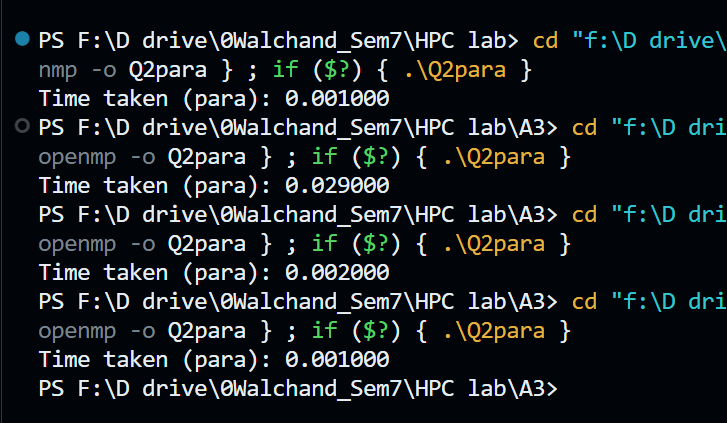
}

**For 500 and 1000 resp.**

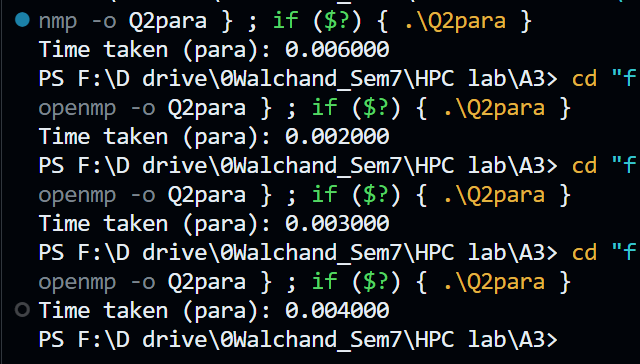
****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 250 | 500 | 750 | 1000 | 2000 |
| Time | 0.000000 | 0.001000 | 0.002000 | 0.001000 | 0.006000 |

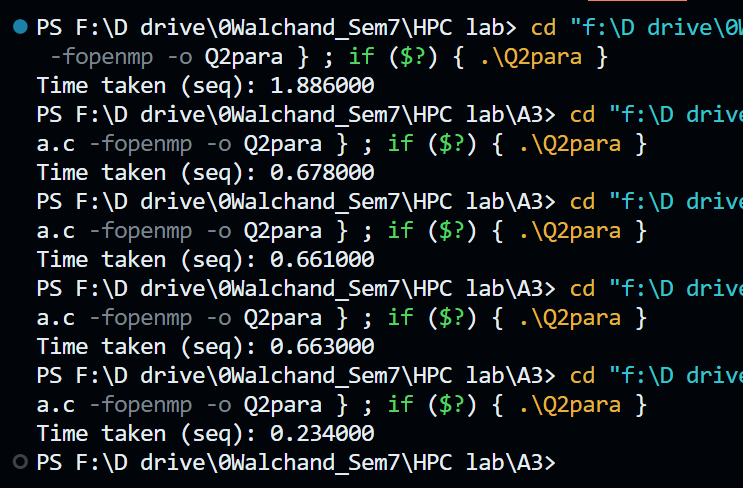
**For 2,4,8,16 threads respectively with N=250:**

****

**For 2,4,6,8,16 threads respectively with N=1000:**

****

**For 2,4,6,8,16 threads respectively with N=10000:**

****

**Information and analysis:**

The reduction clauses are data-sharing attribute clauses that can be used to perform some forms of recurrence calculations in parallel. Reduction clauses include reduction scoping clauses and reduction participating clauses. Reduction scoping clauses define the region in which a reduction is computed. Reduction participating clauses define the participants in the reduction. Reduction clauses specify a reduction-identifier and one or more list items. A reduction-identifier is either an identifier or one of the following operators: +, -, \*, &, |, ^, && and ||.

**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.

**Static Schedule (Source Code):**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#define N 8

int main()

{

  int \*a = (int \*)malloc(sizeof(int) \* N);

  int \*c = (int \*)malloc(sizeof(int) \* N);

  int b = 10;

  omp\_set\_num\_threads(6);

  for (int i = 0; i < N; i++)

  {

    a[i] = 0;

  }

  double itime, ftime, exec\_time;

  itime = omp\_get\_wtime();

#pragma omp parallel for schedule(static, 2)

  for (int i = 0; i < N; i++)

  {

    c[i] = a[i] + b;

  }

  ftime = omp\_get\_wtime();

  exec\_time = ftime - itime;

  printf("\n\nTime taken is %f\n", exec\_time);

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chunk Size | 2 | 4 | 6 | 8 |
| Time | 0.001000 | 0.001900 | 0.002000 | 0.002000 |

**Dynamic Schedule (Source Code):**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#define N 8

int main()

{

  int \*a = (int \*)malloc(sizeof(int) \* N);

  int \*c = (int \*)malloc(sizeof(int) \* N);

  int b = 10;

  omp\_set\_num\_threads(6);

  for (int i = 0; i < N; i++)

  {

    a[i] = 0;

  }

  double itime, ftime, exec\_time;

  itime = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic, 2)

  for (int i = 0; i < N; i++)

  {

    c[i] = a[i] + b;

  }

  ftime = omp\_get\_wtime();

  exec\_time = ftime - itime;

  printf("\n\nTime taken is %f\n", exec\_time);

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chunk Size | 2 | 4 | 6 | 8 |
| Time | 0.000000 | 0.0018 | 0.0017 | 0.0015 |

**Nowait Clause:**

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#define N 10

void hello\_world()

{

  printf("Hello world\n");

}

void print(int i)

{

  printf("Value %d\n", i);

}

int main()

{

#pragma omp parallel

  {

#pragma omp for nowait

    for (int i = 0; i < N; i++)

    {

      print(i);

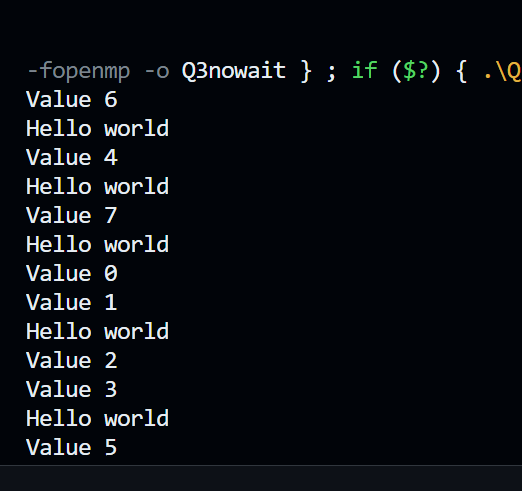
    }

    hello\_world();

  }

}

Output:



**Information and Analysis:**

This C code uses OpenMP to parallelize a loop that prints values and a "Hello world" message. Here's a brief analysis and information:

**Parallelization (OpenMP):** The code utilizes OpenMP directives to parallelize the loop inside the `main` function. It splits the loop into multiple threads, each printing a different value of `i`.

**Functions:** Two functions are defined - `hello\_world` simply prints "Hello world," and `print(int i)` prints the value of `i`.

**Loop Execution:** The parallel loop runs from 0 to N (10) and each thread calls the `print` function to display its value. The `nowait` clause indicates that threads can proceed without waiting for all iterations to complete.

**Thread Interaction:** After the loop, all threads execute the `hello\_world` function concurrently, potentially leading to interleaved "Hello world" messages.

**Output:** The code produces output with values of `i` printed by multiple threads and "Hello world" messages. The order of output may vary due to parallel execution.

**Potential Improvements:** This code serves as a simple example of parallelization with OpenMP. For more complex tasks, you can utilize thread synchronization mechanisms to control the order of execution and prevent race conditions,

Overall, this code demonstrates how to use OpenMP to create parallel regions and parallelize a loop with multiple threads, allowing for concurrent execution of tasks.