Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

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

Course: High Performance Computing Lab

Practical No. 3

Exam Seat No: 2020BTECS00033

Name: Prathamesh Santosh Raje

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:

Minimum Scalar Product Sequential Code:

```
// CPP Program to find the minimum scalar product of two vectors (dot product)
#include<iostream>
#include<algorithm>
#include<vector>
#include<omp.h>
using namespace std;

int main()
{
  int n=1000;
  int sum=0;
  vector<int> arr1(n);
  vector<int> arr2(n);
  int i;
  //filling the vectors
  for(i = 0; i < n ; i++)
  {
    arr1[i]=(i+1)%2;
```

```
    arr2[i]=(i+1)%2;
  }
  sort(arr1.begin(),arr1.end());
  sort(arr2.begin(),arr2.end(), greater<int>());

  double itime, ftime, exec_time;
  itime = omp_get_wtime();

  for(i = 0; i < n ; i++)
  {
    sum = sum + (arr1[i] * arr2[i]);
  }

  ftime = omp_get_wtime();
  exec_time = ftime - itime;
  cout<<"Sum= "<<sum<<endl;
  printf("Time taken = %f\n",exec_time);
  return 0;

}
```

Minimum Scalar Product Sequential Output:

```
PS D:\Final Year B.Tech\HPC\Practical No. 3> g++ -fopenmp .\min_sca_pr_seq.c
PS D:\Final Year B.Tech\HPC\Practical No. 3> .\a.exe
Sum= 0
Time taken = 0.000000
```

Minimum Scalar Product Parallel Code:

```
// CPP Program to find the minimum scalar product of two vectors (dot product)
#include<iostream>
#include<algorithm>
#include<vector>
#include<omp.h>
using namespace std;

int main()
{
  int n=100;
  int sum=0;
  vector<int> arr1(n);
  vector<int> arr2(n);
```

```
  int i;
  //filling the vectors
  for(i = 0; i < n ; i++)
  {
    arr1[i]=(i+1)%2;
    arr2[i]=(i+1)%2;
  }
  sort(arr1.begin(),arr1.end());
  sort(arr2.begin(),arr2.end(), greater<int>());

  omp_set_num_threads(4);

  double itime, ftime, exec_time;
  itime = omp_get_wtime();

  #pragma omp parallel for reduction(+:sum)
  for(i = 0; i < n ; i++)
  {
    sum = sum + (arr1[i] * arr2[i]);
  }

  ftime = omp_get_wtime();
  exec_time = ftime - itime;
  cout<<"Sum= "<<sum<<endl;
  printf("Time taken = %f\n",exec_time);
  return 0;
}
```

Minimum Scalar Product Parallel Output:

```
PS D:\Final Year B.Tech\HPC\Practical No. 3> g++ -fopenmp .\min_sca_pr_par.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 3> .\a.exe
Sum= 0
Time taken = 0.001000
```

Information and analysis:

For sequential execution:

- In this code, the thread count isn't explicitly set, so it relies on the system's default thread count.
- The code calculates the minimum scalar product (dot product) of two vectors using a loop.

- Increasing the thread count beyond the default number (usually the number of CPU cores) can potentially reduce execution time up to a point.
- However, excessive threads can lead to diminishing returns as the overhead of managing more threads can offset any performance gains.
- The execution time depends on the specific hardware and workload.

For Parallel Execution:

- The code calculates the minimum scalar product using OpenMP parallelization.
- Changing the thread count beyond 4 won't directly affect execution time, as it's fixed at 4 threads.
- Execution time can still vary based on hardware and workload characteristics.
- Specifying a thread count can provide some control over parallelization, but the optimal thread count depends on the available CPU resources and workload.

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:

Code:

```
#include <iostream>
#include <vector>
#include <omp.h>
#include <ctime>
#include <cstdlib>

using namespace std;

// Function to initialize a matrix with random values
void initializeMatrix(vector<vector<int>>& matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i][j] = rand() % 100;  // Filling with random values between 0
and 99
        }
    }
}

// Function to add two matrices
void addMatrices(vector<vector<int>>& A, vector<vector<int>>& B,
vector<vector<int>>& C, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

int main() {
```

```
    srand(time(0));  // Seed the random number generator

    const int sizes[] = {250, 500, 750, 1000, 2000};
    const int numThreads[] = {1, 2, 4, 8};  // Change the number of threads

    for (int sizeIndex = 0; sizeIndex < 5; sizeIndex++) {
        int size = sizes[sizeIndex];
        vector<vector<int>> A(size, vector<int>(size));
        vector<vector<int>> B(size, vector<int>(size));
        vector<vector<int>> C(size, vector<int>(size));

        initializeMatrix(A, size);
        initializeMatrix(B, size);

        cout << "Matrix Size: " << size << "x" << size << endl;

        for (int numThreadIndex = 0; numThreadIndex < 4; numThreadIndex++) {
            int numThread = numThreads[numThreadIndex];
            omp_set_num_threads(numThread);

            double start_time = omp_get_wtime();

            addMatrices(A, B, C, size);

            double end_time = omp_get_wtime();
            double execution_time = end_time - start_time;

            cout << "Threads: " << numThread << ", Time = " << execution_time <<
" seconds" << endl;
        }
        cout << endl;
    }

    return 0;
}
```

Output:

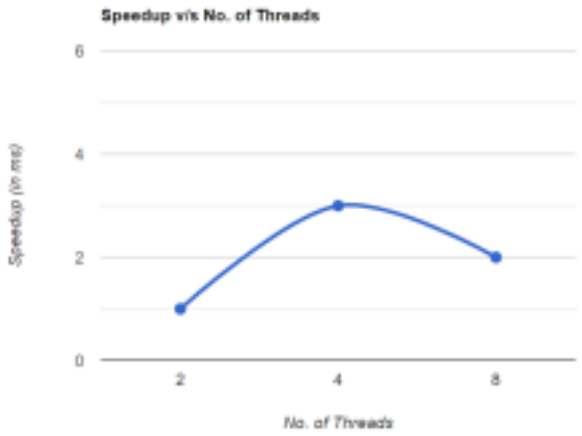
```
PS D:\Final Year B.Tech\HPC\Practical No. 3> g++ -fopenmp .\matrix_add.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 3> .\a.exe
Matrix Size: 250x250
Threads: 1, Time = 0.00100017 seconds
Threads: 2, Time = 0.000999928 seconds
Threads: 4, Time = 0.00100017 seconds
Threads: 8, Time = 0.000999928 seconds

Matrix Size: 500x500
Threads: 1, Time = 0.00400019 seconds
Threads: 2, Time = 0.00300002 seconds
Threads: 4, Time = 0.00300002 seconds
Threads: 8, Time = 0.00200009 seconds

Matrix Size: 750x750
Threads: 1, Time = 0.00800014 seconds
Threads: 2, Time = 0.00800014 seconds
Threads: 4, Time = 0.00399995 seconds
Threads: 8, Time = 0.00200009 seconds

Matrix Size: 1000x1000
Threads: 1, Time = 0.00999999 seconds
Threads: 2, Time = 0.00600004 seconds
Threads: 4, Time = 0.00699997 seconds
Threads: 8, Time = 0.00500011 seconds

Matrix Size: 2000x2000
Threads: 1, Time = 0.069 seconds
Threads: 2, Time = 0.0250001 seconds
Threads: 4, Time = 0.0239999 seconds
Threads: 8, Time = 0.016 seconds
```

Information and analysis:

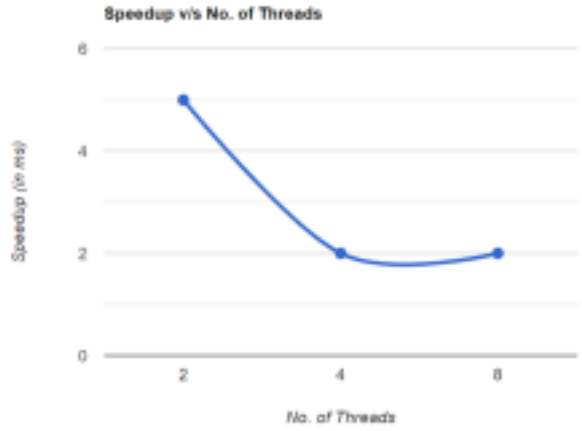
- As the number of threads increases, the execution time for matrix addition generally decreases. More threads can perform parallel work, improving overall performance.
- However, the actual impact depends on the matrix size and the available CPU resources. Smaller matrices may not benefit as much from additional threads, while larger matrices can see significant speedup.
- There is a point of diminishing returns. Increasing the thread count excessively can lead to diminishing performance gains due to overhead associated with managing a large number of threads.

Graph:

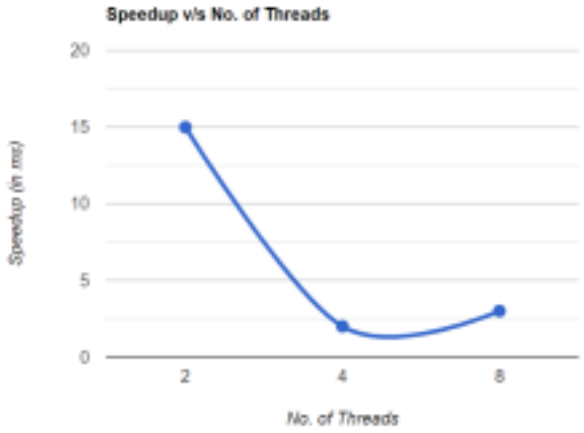
1. Matrix size = 250



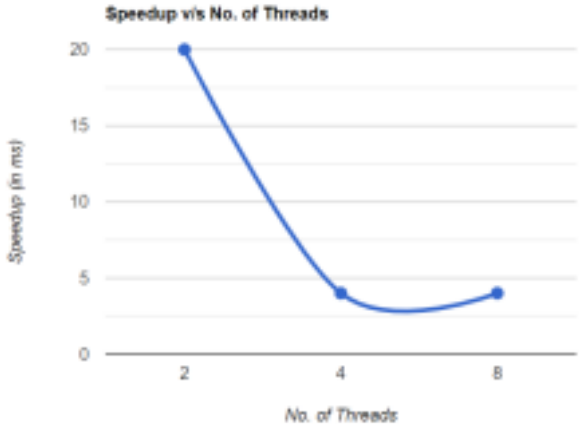
2. Matrix size = 500



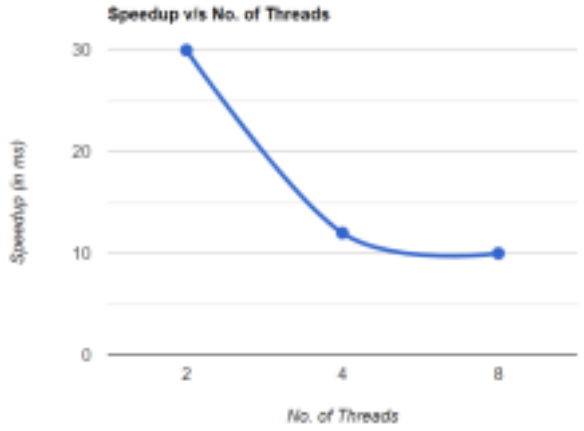
3. Matrix size = 750



4. Matrix size = 1000



5. Matrix size = 2000



From above graphs, we can see that as matrix size increases, increasing the number of threads decreases the speedup.

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:

1D Vector Static Code:

```
#include <iostream>
#include <vector>
#include <omp.h>

using namespace std;

int main() {
    const int size = 200;
    vector<int> a(size), b(size), c(size);

    // Initialize vectors 'a' and 'b'

    int chunk_sizes[] = {1, 2, 4, 8, 16}; // Vary the chunk sizes

    for (int chunk_size : chunk_sizes) {
        double start_time = omp_get_wtime();

        #pragma omp parallel for schedule(static, chunk_size)
        for (int i = 0; i < size; i++) {
            c[i] = a[i] + b[i];
        }

        double end_time = omp_get_wtime();
        double execution_time = end_time - start_time;

        cout << "Chunk Size " << chunk_size << ": Time = " << execution_time << "
seconds" << endl;
    }

    return 0;
}
```

1D Vector Dynamic Code:

```
#include <iostream>
#include <vector>
#include <omp.h>

using namespace std;

int main() {
    const int size = 200;
    vector<int> a(size), b(size), c(size);

    // Initialize vectors 'a' and 'b'

    int chunk_sizes[] = {1, 2, 4, 8, 16}; // Vary the chunk sizes

    for (int chunk_size : chunk_sizes) {
        double start_time = omp_get_wtime();

        #pragma omp parallel for schedule(dynamic, chunk_size)
        for (int i = 0; i < size; i++) {
            c[i] = a[i] + b[i];
        }

        double end_time = omp_get_wtime();
        double execution_time = end_time - start_time;

        cout << "Chunk Size " << chunk_size << ": Time = " << execution_time << "
seconds" << endl;
    }

    return 0;
}
```

1D Vector Nowait Code:

```
#include <stdio.h>
#include <omp.h>

#define ARRAY_SIZE 100

int main()
{
    int data[ARRAY_SIZE];
```

```
    int result[ARRAY_SIZE];

    // Initialize the data array
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        data[i] = i;
    }

#pragma omp parallel
    {
        int thread_id = omp_get_thread_num();

// Perform some computations on a portion of the data array
#pragma omp for nowait
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
            result[i] = data[i] * 2 + thread_id;
        }

        // This section will execute concurrently without waiting for other
threads
        printf("Thread %d has completed its task.\n", thread_id);

        // Additional work can be done here, independently by each thread
    }

    printf("All threads have completed their task.\n");

    return 0;
}
```

Information and analysis:

Execution time for static schedule:

```
PS D:\Final Year B.Tech\HPC\Practical No. 3> g++ -fopenmp .\1dstatic.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 3> .\a.exe
Chunk Size 1: Time = 0.000999928 seconds
Chunk Size 2: Time = 0 seconds
Chunk Size 4: Time = 0.00100017 seconds
Chunk Size 8: Time = 0 seconds
Chunk Size 16: Time = 0 seconds
PS D:\Final Year B.Tech\HPC\Practical No. 3>
```

Execution time for dynamic schedule:

```
PS D:\Final Year B.Tech\HPC\Practical No. 3> g++ -fopenmp .\1ddynamic.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 3> .\a.exe
Chunk Size 1: Time = 0.000999928 seconds
Chunk Size 2: Time = 0 seconds
Chunk Size 4: Time = 0 seconds
Chunk Size 8: Time = 0 seconds
Chunk Size 16: Time = 0 seconds
```

Execution time for nowait schedule:

```
PS D:\Final Year B.Tech\HPC\Practical No. 3> g++ -fopenmp .\1dnowait.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 3> .\a.exe
Thread 1 has completed its task.
Thread 3 has completed its task.
Thread 4 has completed its task.
Thread 2 has completed its task.
Thread 0 has completed its task.
Thread 5 has completed its task.
Thread 6 has completed its task.
Thread 7 has completed its task.
All threads have completed their task.
```

Analysis:

- Increasing the thread number may improve execution time, especially when the workload is unevenly distributed among iterations.
- Smaller chunk sizes (e.g., 1 or 2) can lead to more balanced work distribution and better utilization of threads.
- However, the impact of increasing the thread number depends on the specific workload and system characteristics.

Github Link: