

Class: Final Year (Computer Science and Engineering)
Year: 2023-24
Semester: 1
Course: High Performance Computing Lab

Practical No. 5

Exam Seat No.: 2020BTECS00085

Title of practical: Study and Implementation of OpenMP program

1. Implementation of sum of two lower triangular matrices.

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

#define N 5 // Size of the matrices

void sumLowerTriangularMatrices(int A[][N], int B[][N], int C[][N]) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j <= i; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

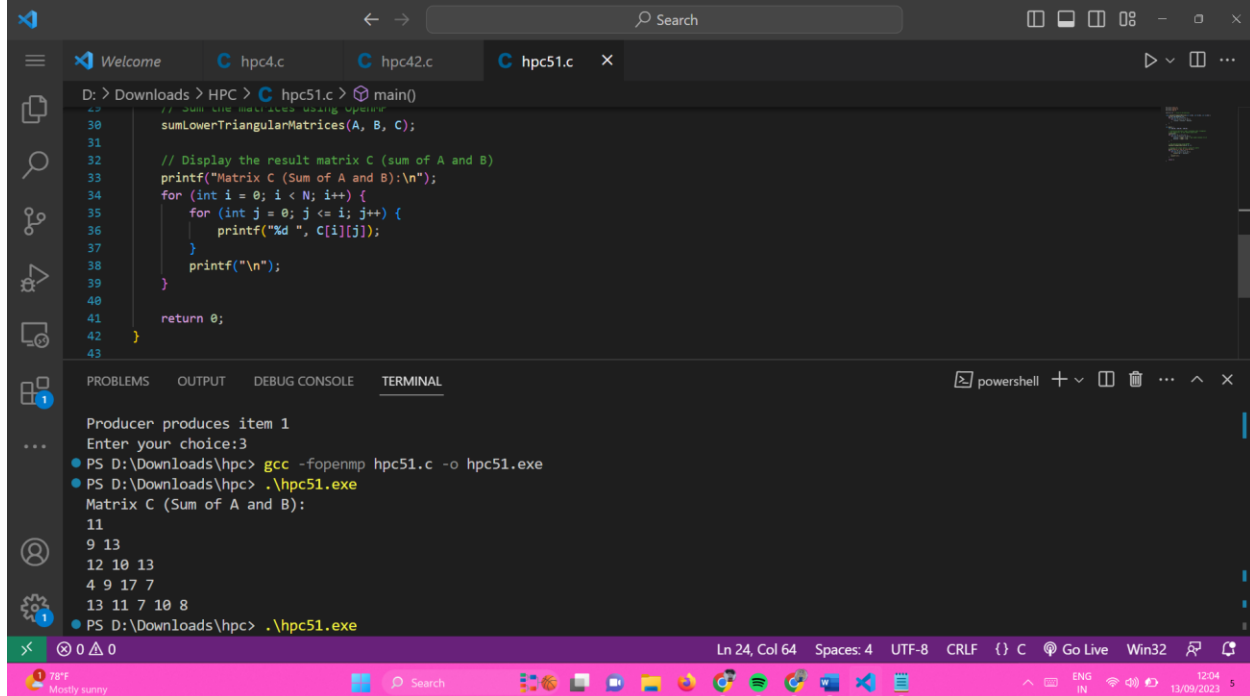
int main() {
    int A[N][N], B[N][N], C[N][N];

    // Initialize matrices A and B (assuming lower triangular)
    // For simplicity, we use random values here.
    srand(1234);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j <= i; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }

    // Sum the matrices using OpenMP
    sumLowerTriangularMatrices(A, B, C);

    // Display the result matrix C (sum of A and B)
    printf("Matrix C (Sum of A and B):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j <= i; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:



```
D: > Downloads > HPC > C hpc51.c > main()
42 // Sum the matrices using OpenMP
30 sumLowerTriangularMatrices(A, B, C);
31
32 // Display the result matrix C (sum of A and B)
33 printf("Matrix C (Sum of A and B):\n");
34 for (int i = 0; i < N; i++) {
35     for (int j = 0; j <= i; j++) {
36         printf("%d ", C[i][j]);
37     }
38     printf("\n");
39 }
40
41 return 0;
42 }
43
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Producer produces item 1
Enter your choice:3

PS D:\Downloads\hpc> gcc -fopenmp hpc51.c -o hpc51.exe
PS D:\Downloads\hpc> .\hpc51.exe

Matrix C (Sum of A and B):
11
9 13
12 10 13
4 9 17 7
13 11 7 10 8

PS D:\Downloads\hpc> .\hpc51.exe

Information:

- **Matrix Size:** The code works with square matrices of size $N \times N$, where N is defined as 5. You can modify N to suit your specific matrix size.
- **Matrix Initialization:** The code initializes matrices A and B with random integer values between 0 and 9. This initialization is done for demonstration purposes. In practice, you can use your own matrices or input method.
- **Parallelization with OpenMP:** OpenMP is used to parallelize the matrix addition operation. The `#pragma omp parallel for` directive distributes the work across multiple threads for efficient parallel execution.
- **Result Matrix:** The result of the matrix addition is stored in matrix C , which is then displayed to the console.

Analysis:

- **Parallelism:** The primary advantage of this code is the use of OpenMP to parallelize the addition of lower triangular matrices. This can significantly speed up the operation, especially for larger matrices and on multi-core processors.
- **Matrix Initialization:** The code initializes matrices A and B with random values. However, this is a simple example, and in real-world applications, you may need to load data from files or other sources.
- **Performance:** The performance gain from parallelism depends on the matrix size and the number of available CPU cores. For larger matrices, the speedup achieved by parallel execution can be more significant.

- **Load Balancing:** This code does not explicitly handle load balancing among threads. Depending on the matrix size and the number of threads, some threads may complete their work earlier than others. In more complex applications, load balancing strategies may be necessary for optimal performance.
- **Data Dependency:** In this code, there are no data dependencies, so threads can work independently on different parts of the matrices. However, in other matrix operations, you may need to consider data dependencies and synchronization mechanisms.
- Overall, this code serves as a basic example of parallelizing matrix addition using OpenMP, demonstrating how parallelism can be utilized to improve the efficiency of matrix operations.

2. Implementation of Matrix-Matrix Multiplication.

Source Code:

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

#define N 3 // Number of rows in matrices A and C
#define M 4 // Number of columns in matrix A and rows in matrix B
#define P 2 // Number of columns in matrices B and C

void matrixMultiply(int A[N][M], int B[M][P], int C[N][P]) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < P; j++) {
            C[i][j] = 0; // Initialize the result element to 0
            for (int k = 0; k < M; k++) {
                C[i][j] += A[i][k] * B[k][j]; // Matrix multiplication
            }
        }
    }
}

int main() {
    int A[N][M], B[M][P], C[N][P];

    // Initialize matrices A and B with sample values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            A[i][j] = i * M + j + 1;
        }
    }

    for (int i = 0; i < M; i++) {
        for (int j = 0; j < P; j++) {
            B[i][j] = i * P + j + 1;
        }
    }

    // Perform matrix multiplication using OpenMP
```

```

matrixMultiply(A, B, C);

// Display the result matrix C
printf("Matrix C (Result of A x B):\n");
for (int i = 0; i < N; i++) {
    for (int j = 0; j < P; j++) {
        printf("%d ", C[i][j]);
    }
    printf("\n");
}

return 0;
}

```

Output:

The screenshot shows the Visual Studio Code interface with a C program being executed. The code defines three matrices: A (N x M), B (M x P), and C (N x P). Matrices A and B are initialized with sample values. The program then performs matrix multiplication to calculate the result matrix C. The output shows the resulting matrix C.

```

D: > Downloads > HPC > hpc52.c > main()
17 }
18 }
19 }
20
21 int main() {
22     int A[N][M], B[M][P], C[N][P];
23
24     // Initialize matrices A and B with sample values
25     for (int i = 0; i < N; i++) {
26         for (int j = 0; j < M; j++) {
27             A[i][j] = i * M + j + 1;
28         }
29     }
30
31     for (int i = 0; i < M; i++) {
32         for (int j = 0; j < P; j++) {
33             B[i][j] = i * P + j + 1;
34         }
35     }
36
37     matrixMultiply(A, B, C);
38
39     // Display the result matrix C
40     printf("Matrix C (Result of A x B):\n");
41     for (int i = 0; i < N; i++) {
42         for (int j = 0; j < P; j++) {
43             printf("%d ", C[i][j]);
44         }
45         printf("\n");
46     }
47
48     return 0;
49 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

PS D:\Downloads\hpc> gcc -fopenmp hpc52.c -o hpc52.exe
PS D:\Downloads\hpc> .\hpc52.exe
Matrix C (Result of A x B):
50 60
114 140
178 220
PS D:\Downloads\hpc>

```

Information:

- **Matrix Sizes:** The code defines the sizes of the matrices `A`, `B`, and `C` using constants `N`, `M`, and `P`. These constants determine the number of rows and columns in each matrix. You can adjust these values according to your specific matrix dimensions.
- **Matrix Initialization:** The code initializes matrices `A` and `B` with sample values. It uses nested loops to assign values to each element of the matrices. You can replace these initializations with your own matrices or input method.
- **Matrix Multiplication Function:** The `matrixMultiply` function is responsible for performing matrix multiplication. It uses nested loops to iterate through the matrices and calculate the product matrix `C`. OpenMP parallelism is applied to the outer loops, enabling concurrent execution of matrix multiplication for improved performance.

- **Displaying the Result:** After matrix multiplication, the code prints the result matrix `C` to the console.

Analysis:

- **Parallelism with OpenMP:** The primary benefit of this code is the efficient use of OpenMP to parallelize the matrix multiplication process. By distributing the workload across multiple threads, the code can take advantage of multi-core processors, significantly speeding up matrix operations for larger matrices.
- **Matrix Initialization:** The code initializes matrices `A` and `B` with simple sequential loops. For larger matrices, this process can be time-consuming. In practice, matrix initialization might be done in parallel as well, especially when dealing with very large datasets.
- **Performance Scalability:** The code is suitable for moderately sized matrices due to the efficient parallelism. However, the performance gain from parallelism depends on the available hardware cores and memory bandwidth. For very large matrices, memory access patterns can become a bottleneck, and more advanced optimization techniques might be required.
- **Accuracy and Precision:** This code assumes integer matrices, and there's no consideration for numerical precision or potential overflow/underflow issues. In real-world applications, you might need to use floating-point types and handle numerical stability.
- **Code Flexibility:** The code is a simplified example. In practical applications, error handling, input validation, and memory management should be considered. Additionally, more advanced matrix multiplication algorithms like Strassen's algorithm or library functions like BLAS might be used for improved performance.

Github Link:

<https://github.com/manjiri-chandure/HPC>