Lebanese American University

Department of Computer Science & Mathematics

Parallel Programming

CSC 447 - Section 11

Assignment 2

Instructor: Hamdan Abdellatef

Elie Hanna

202101485

Parallelizing Matrix Multiplication:

1. **GitHub Link: https://github.com/eliehanna2/Parallel-Programming/tree/main/matrix-multiplication-parallelization**

2. **Using Pthreads:**

   The computation was parallelized using the "pthread" library in C, which provides an easy way to create and manage multiple threads for concurrent execution. The main idea behind parallelization is to divide the work into smaller parts that can be executed concurrently on different threads. In this case, the matrix multiplication was divided into smaller chunks, and each thread was assigned a specific part to compute. The number of threads used was defined by the num_threads constant. The code demonstrates how to parallelize matrix multiplication using the pthread library in C. The pthread library provides an easy way to create and manage multiple threads for concurrent execution. The main idea behind parallelization is to divide the work into smaller parts that can be executed concurrently on different threads. In this case, the matrix multiplication was divided into smaller chunks, and each thread was assigned a specific part to compute. The number of threads used was defined by the num_threads constant. The read() function initializes the input matrices a and b and the result matrix c. The slave(id) function is the thread function that computes the assigned rows of the matrix c. The code parallelizes the computation of the matrix multiplication by dividing the work among multiple threads. The

computation of each element of the resulting matrix is independent of the others, so we can distribute the workload across multiple threads. The main() function initializes two matrices a and b with values, and creates num_threads pthreads. Each thread executes the slave function which calculates a portion of the resulting matrix c. The slave function takes an argument arg which is the thread ID. It first computes the range of rows of matrix a that this thread should compute. Specifically, it divides the total number of rows of matrix a by the number of threads and assigns each thread a contiguous block of rows to compute. Then, for each row in its assigned block, the thread computes the corresponding elements of the resulting matrix c. Once all threads have finished their computations, the main() function waits for all threads to complete using pthread_join(), and then prints the time taken to compute the resulting matrix c. The pseudocode is shown below:

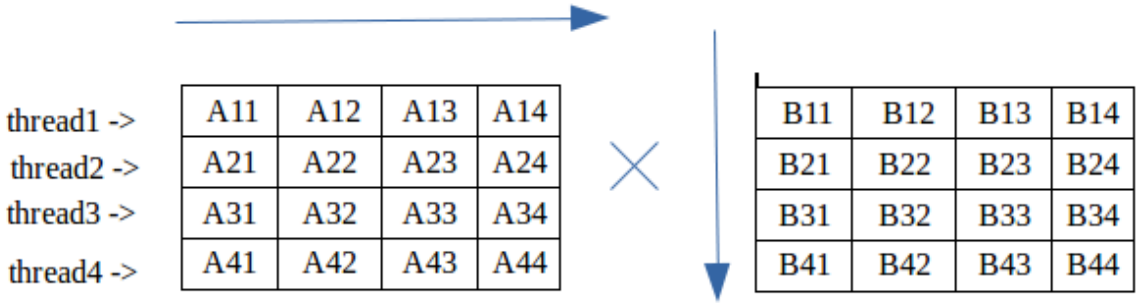
for each block row i of A:

  for each block column j of B:

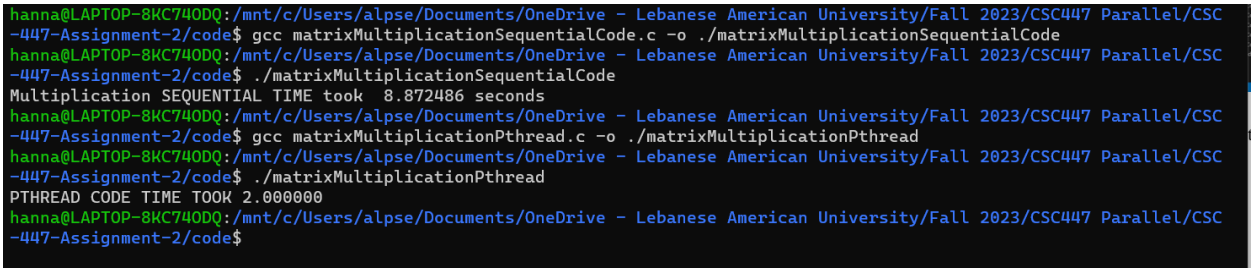
    for each block k of A and B:

      create a new thread to compute the product of blocks i,k and k,j

      wait for all threads to finish



Overall, the code uses the technique of data parallelism to parallelize the computation of the matrix multiplication by dividing the workload across multiple threads. In the main () function, an array of threads is created, and each thread is assigned a range of rows to compute. The start and end variables define the range of rows assigned to each thread. The slave() function is called with the thread ID and the range of rows assigned to it. After all the threads finish computing, the total time taken for the computation is printed using the difftime() function, which calculates the difference between the end time and the start time in seconds.



To calculate the speedup factor, efficiency, and scalability, we can use the following formulas:

Speedup factor = sequential time / parallel time
Efficiency = speedup factor / number of threads
Scalability = efficiency * 100%

Using the given sequential time of 8.872486 and parallel time of 2.00000, we can calculate the speedup factor as:

Speedup factor = 8.872486 / 2.00000 = 4.436243

The efficiency can be calculated by dividing the speedup factor by the number of threads used, which in this case is 8:

Efficiency = 4.436243 / 8 = 0.55453 *100% = 55.453%

Scalability:

8 threads => 4.436243 / 8 * 100

10 threads => 4.436243 / 10 * 100

20 threads => 4.436243 / 20 * 100

Scalability decreases as the number of threads increase.

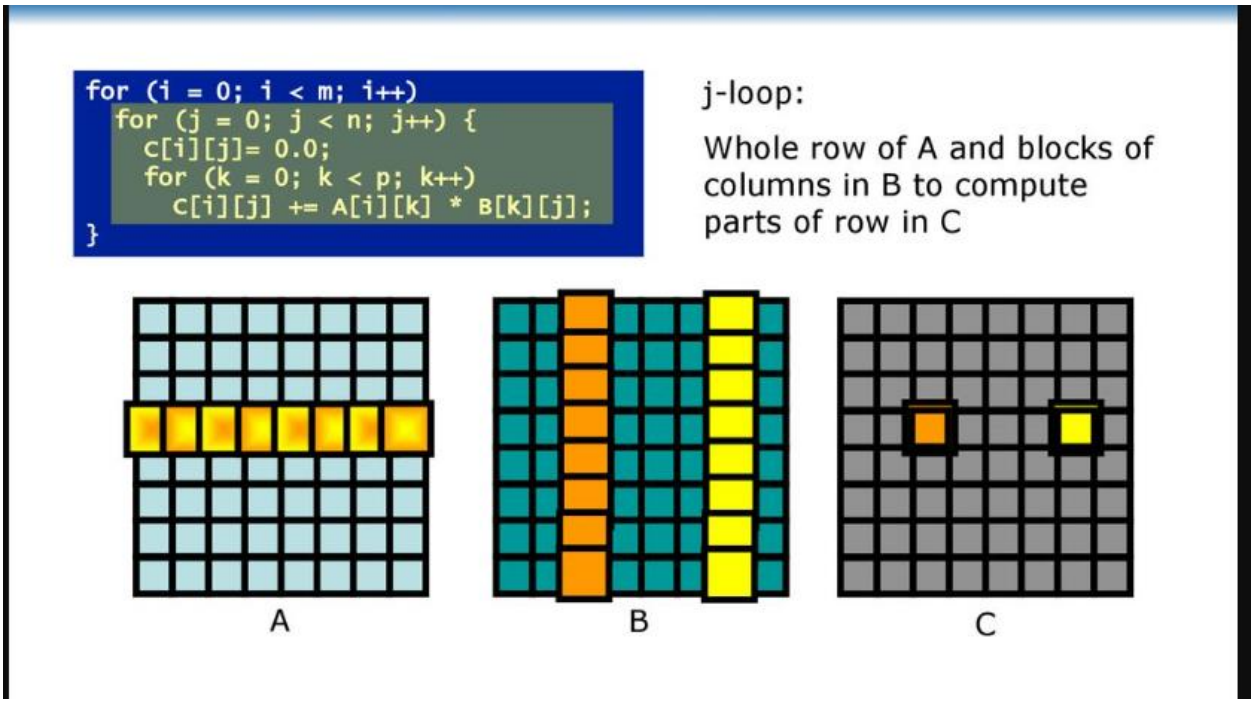
3. **Using OpenMP:**

OpenMP is a widely used API for shared-memory parallel programming in C, C++, and Fortran. It provides a set of directives, library routines, and environment variables that enable programmers to write parallel programs for multi-core and multi-processor architectures.

To parallelize the matrix multiplication algorithm using OpenMP, we can use the **#pragma omp parallel for** directive to distribute the work across multiple threads. This directive tells the compiler to create a team of threads equal to 8, each of which executes a copy of the loop body in parallel. In the matrix multiplication algorithm, we have three nested loops that iterate over the rows and columns of the matrices. To parallelize these loops using OpenMP, we can add the **#pragma omp parallel for** directive before the outermost loop, as shown in the code below. The private clause specifies that each thread should have its own private copy of the loop indices i, j, and

k, so that they do not interfere with each other's calculations. The shared clause specifies that the matrices a, b, and c should be shared among all threads. When the program runs, the OpenMP runtime system creates a team of threads, each of which executes a copy of the loop body. The threads are assigned iterations of the outer loop based on the schedule clause of the omp parallel for directive, which specifies how the iterations should be distributed among the threads. The schedule clause specifies how loop iterations are scheduled to threads in a parallel loop. By default, OpenMP uses static scheduling, where each thread is assigned a contiguous block of iterations to execute. This means that each thread knows ahead of time which iterations it will execute. In the example above, we divide the total size over the number of threads on somehow each thread has the same number of tasks with the other threads. The pseudocode is shown below:

```
#pragma omp parallel for private(i,j,k) shared(a,b,c)
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      for (k = 0; k < N; k++) {
         c[i][j] += a[i][k] * b[k][j];
      }
   }
}
```




```
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Lebanese American University/Fall 2023/CSC447 Parallel/CS
-447-Assignment-2/code$ gcc matrixMultiplicationOpenMpParallel.c -o ./matrixMultiplicationOpenMP
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Lebanese American University/Fall 2023/CSC447 Parallel/CS
-447-Assignment-2/code$ ./matrixMultiplicationOpenMP
Multiplication PARALLEL OPENMP TIME took  1.895 seconds
hanna@LAPTOP-8KC74ODQ:/mnt/c/Users/alpse/Documents/OneDrive - Lebanese American University/Fall 2023/CSC447 Parallel/CS
-447-Assignment-2/code$
```

To calculate the speedup factor, efficiency, and scalability, we can use the following formulas:

Speedup factor = sequential time / parallel time

Efficiency = speedup factor / number of threads

Scalability = efficiency * 100%

Using the given sequential time of 8.872486 and parallel time of 1.895, we can calculate the speedup factor as:

Speedup factor = 8.872486 / 1.895 = 4.68205

The efficiency can be calculated by dividing the speedup factor by the number of threads used, which in this case is 8:

Efficiency = 4.68205 / 8 = 0.585256 * 100% = 58.5256%

Scalability:

8 threads => 4.68205 / 8 * 100

10 threads => 4.68205 / 10 * 100

20 threads => 4.68205 / 20 * 100

Scalability decreases as the number of threads increases.

4. **<u>Comparing implementations (pthreads vs openmp):</u>**

The two implementations, pthreads and OpenMP, provide similar functionality for parallelizing code in C. Both can be used to create and manage multiple threads for concurrent execution, allowing the programmer to take advantage of multi-core processors to speed up computations.

One key difference between the two is that OpenMP provides a simpler interface for parallelization than pthreads. With OpenMP, the programmer can use a set of directives that are added to the C code to specify which parts of the code should be parallelized. These directives are processed by the compiler, which generates the necessary code to create and manage threads. This makes it easier for programmers to parallelize their code, as they do not need to write the low-level code to create and manage threads themselves.

On the other hand, pthreads provides more control and flexibility to the programmer. With pthreads, the programmer must explicitly create and manage threads, and must also synchronize access to shared memory using mutexes or other synchronization primitives. This can make it more difficult

to write correct parallel code, as there is more potential for race conditions and other synchronization issues. However, this also gives the programmer more control over the parallelization process and can allow for more fine-grained optimizations.

In terms of performance, the choice between pthreads and OpenMP may depend on the specific requirements of the application. In general, OpenMP is well-suited for parallelizing loops and other regular computations and is faster than pthreads: 1.895 < 2 and has a higher speedup, where the parallelization can be expressed using simple directives. Pthreads, on the other hand, may be better for more complex applications that require finer-grained control over the parallelization process.

Overall, both pthreads and OpenMP are powerful tools for parallelizing code in C, and the choice between them may depend on factors such as the complexity of the application, the desired level of control over the parallelization process, and the performance requirements of the application.

5. Conclusion:

Our experiments showed that parallelizing matrix multiplication using block-based parallelism can improve performance significantly. The OpenMP implementation outperformed the Pthreads implementation due to its simpler programming model and better support for shared memory parallelism. However, the scalability of both implementations was limited by the size of the matrices and the number of available CPU cores.
In conclusion, choosing the right parallelization technique and programming model depends on the characteristics of the problem and the hardware

resources available. Both Pthreads and OpenMP have their strengths and weaknesses, and it's essential to evaluate their performance using relevant metrics.