1.2.

```
[Running] cd "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\" && g++ -std=c++17 sequential_matrix_multiplication.cpp -o
sequential_matrix_multiplication && "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\"sequential_matrix_multiplication
Time taken for matrix multiplication (sequential): 12 seconds.

[Done] exited with code=0 in 14.724 seconds
```

3.

Data partitioning:

The computational tasks of matrix C are divided into rows, and each thread is responsible for calculating the results of several rows. In this way, multiple rows can be processed in parallel, thereby speeding up the calculation of the entire matrix multiplication.

Task parallelization:

Each thread can independently calculate the rows assigned to it to avoid data dependency between threads.

Sequential and parallel tasks:

Parallel tasks: Calculate the rows of matrix C. Tasks can be assigned to different threads through multithreading.

Sequential tasks: Matrix initialization and result output need to be executed sequentially before and after parallel calculation.

4.

```
[Running] cd "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\" && g++ -std=c++17 parallel_matrix_multiplication_pthread.cpp -o
parallel_matrix_multiplication_pthread && "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\"parallel_matrix_multiplication_pthread
Time taken for matrix multiplication (pthreads): 4 seconds.

[Done] exited with code=0 in 4.858 seconds
```

5.

```
[Running] cd "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\" && g++ -std=c++17 parallel_matrix_multiplication_pthread.cpp -o
parallel_matrix_multiplication_pthread && "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\"parallel_matrix_multiplication_pthread
Time taken for matrix multiplication (pthreads): 6 seconds.

[Done] exited with code=0 in 7.954 seconds

[Running] cd "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\" && g++ -std=c++17 parallel_matrix_multiplication_pthread.cpp -o
parallel_matrix_multiplication_pthread && "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\"parallel_matrix_multiplication_pthread
Time taken for matrix multiplication (pthreads): 36 seconds.

[Done] exited with code=0 in 37.831 seconds

[Running] cd "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\" && g++ -std=c++17 parallel_matrix_multiplication_pthread.cpp -o
parallel_matrix_multiplication_pthread && "f:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P\"parallel_matrix_multiplication_pthread
Time taken for matrix multiplication (pthreads): 59 seconds.

[Done] exited with code=0 in 61.051 seconds
```

Matrix Size	Thread Count	Time	Speedup
1000x1000	2	4	2.0
1000x1000	4	6	3.0
2000x2000	2	36	1.63
2000x2000	4	59	2.67

6.

```
PS F:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P> ./parallel_matrix_multiplication_openmp
Time taken for matrix multiplication (OpenMP): 1.721 seconds.
PS F:\Deakin Year3 Tasks and Assigments\SIT315\M2.T1P>
```

7.

Matrix Size	Implementation Type	Time	Speedup
1000x1000	Sequential	12	1.0
1000x1000	pthreads (4 threads)	4	3.0
1000x2000	OpenMP	1.7	7.06

The performance comparison between the OpenMP, pthreads, and sequential matrix multiplication implementations highlights the clear advantages of parallelization. The sequential program serves as the baseline, taking 12 seconds for a 1000x1000 matrix, while the pthreads implementation reduces this to 4 seconds, achieving a speedup of 3.0 by manually managing threads. However, the OpenMP implementation significantly outperforms both, completing the task in just 1.7 seconds with a speedup of 7.06. This demonstrates OpenMP's superior efficiency in thread management and load balancing, allowing it to better utilize multi-core processors with minimal overhead, making it the most effective approach for parallelizing matrix multiplication and achieving maximum performance gains.