TASK 2.1

Pthread approach:
This thread function is where the task of matrix multiplication is decomposed into multiple subtasks that each thread can work on concurrently. It shows a solid example of the rationale of "Concurrency Without Thread Synchronization". Each thread is given a "slice" of data to work on that is passed in from its argument. From the perspective of working threads, this means that each thread can work on a subtask from start to the end without waiting for the results of other threads. In the eyes of data, it implies that each thread would only manipulate its own portion of the shared data. I divide the tasks into subtasks, for example I had matrices with 1000*1000 dimension so I divided it into 4 threads which means that each thread will deal with 250 rows and columns for multiplication.

Output with threads:

```
Duration: 2118990 microseconds for ROWS: 1000 and THREADS: 8
Duration: 1976968 microseconds for ROWS: 1000 and THREADS: 4
Duration: 1959691 microseconds for ROWS: 1000 and THREADS: 3
Duration: 1859317 microseconds for ROWS: 1000 and THREADS: 2
Duration: 123 microseconds for ROWS: 10 and THREADS: 4
Duration: 47 microseconds for ROWS: 10 and THREADS: 1
Duration: 1400 microseconds for ROWS: 100 and THREADS: 4
```

Output with sequential programming:

```
Duration: 2 microseconds for ROWS: 10
Duration: 3369 microseconds for ROWS: 100
Duration: 18927 microseconds for ROWS: 200
Duration: 69331 microseconds for ROWS: 300
Duration: 167140 microseconds for ROWS: 400
Duration: 343030 microseconds for ROWS: 500
Duration: 619412 microseconds for ROWS: 600
Duration: 1537737 microseconds for ROWS: 700
```

So if we compare the outcomes for the sequential and parallel programming using pthreads than it can easily be observed that sequential programming is more time efficient for the matrices which have less number of rows and columns but as the count of rows exceed 100 then parallel approach using pthreads is more time efficient as same 100 rows take 1400 microseconds to give you a result while in sequential programming it takes 3369 microseconds.
Similiarly, too much multithreading is just as much a problem as too little. For example, for the task of assembling a house: it's a big job and having more than one worker is helpful. But there are limits. Ten workers is great! A hundred is challenging. Ten thousand probably means the house will get built more slowly than with just twenty, this can be observe easily by output from approach using pthread. While keeping the rows constant and increasing the threads, we can observe reduction in time efficiency as it takes more time if you keep on increasing the threads.

OpenMp approach:

The OpenMP-enabled parallel code exploits coarse grain parallelism, which makes use of the cores available in a multicore machine. Basically, we have parallelized the outermost loop which drives the accesses to the result matrix *a* in the first dimension. The ideal speed-up is around 5 and we get a maximun speed up near 3.5, which exposes the memory problems of the matrix-matrix product and its impact on performance. openMp is much better than sequential programming and on par with pthreads. The performance of parallel algorithms may be increased significantly with programming techniques oriented to locality and memory issues. However, although parallel programming is hard and error-prone, it is the primary source of performance gain in modern computer systems.