Matrix Multiplication Documentation

Results

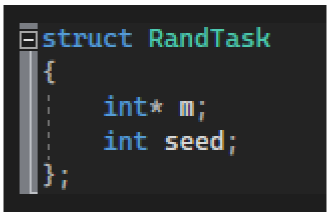
Paradigm	Number of Threads	Size of Matrix	Time
Sequential	1	10	41 microseconds
Parallel	12	10	7822 microseconds
OpenMP	12	10	62 microseconds

Parallel Program Decomposition

The matrix multiplication program consists of three stages:

1. Allocating memory
2. Calculating the matrix
3. Printing results

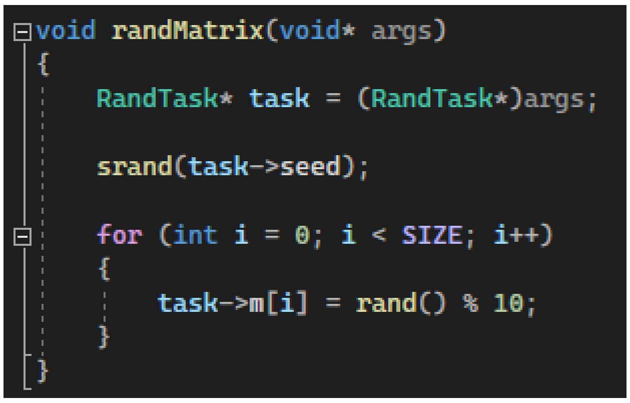
Stage 1 and 2 are parallelised. In stage 1, each thread is responsible for allocating one row of each of the three matrices. Each thread is given a task that is structured like this:



```
struct RandTask
{
    int* m;
    int seed;
};
```

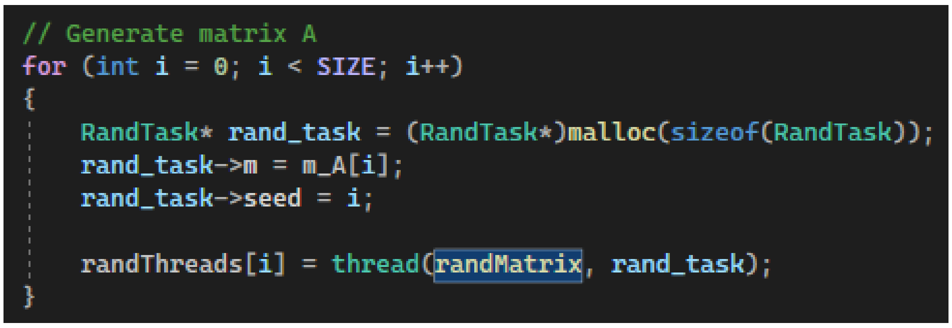
‘m’ is a pointer to the row in the matrix while the seed is given to the random number generator.

Each thread executes the following code to allocate the memory:



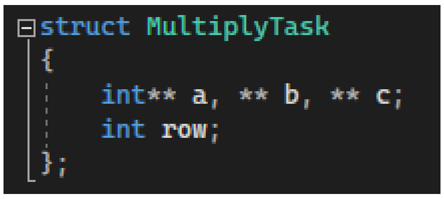
```
void randMatrix(void* args)
{
    RandTask* task = (RandTask*)args;

    srand(task->seed);

    for (int i = 0; i < SIZE; i++)
    {
        task->m[i] = rand() % 10;
    }
}
```

The threads are instantiated here:



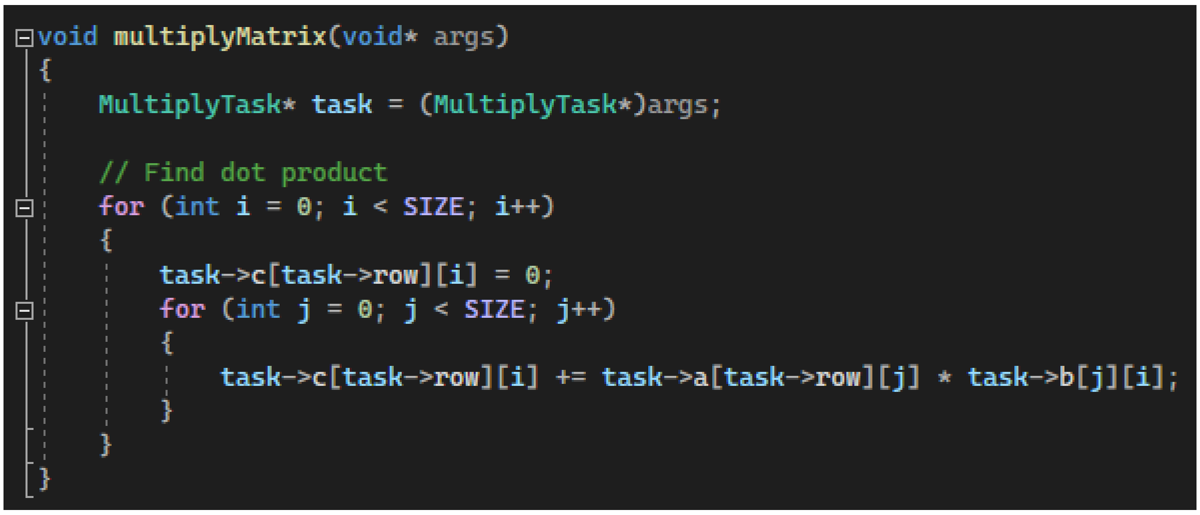
```
// Generate matrix A
for (int i = 0; i < SIZE; i++)
{
    RandTask* rand_task = (RandTask*)malloc(sizeof(RandTask));
    rand_task->m = m_A[i];
    rand_task->seed = i;

    randThreads[i] = thread(randMatrix, rand_task);
}
```

Stage 2 follows a very similar process. Each thread is given a row from each of the matrices. Here is the task structure for multiplying the rows of the matrices:



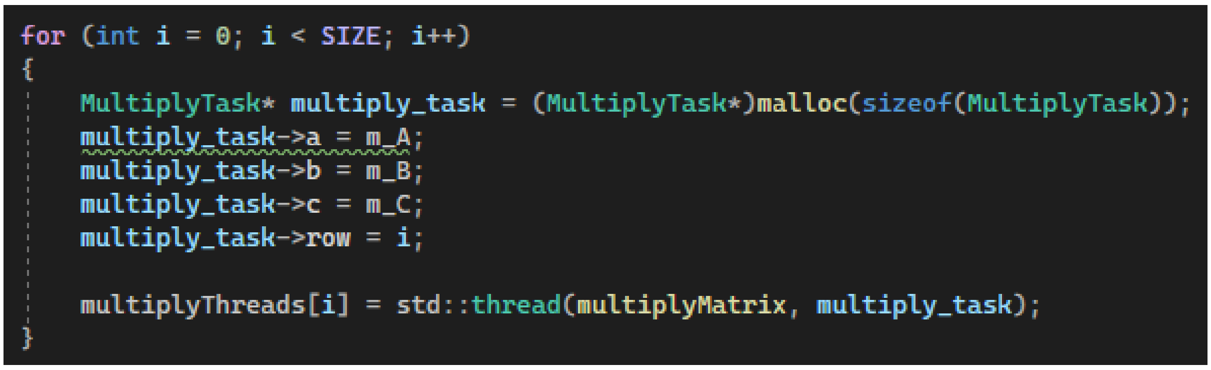
```
struct MultiplyTask
{
    int** a, ** b, ** c;
    int row;
};
```

The code below shows what each thread is doing when calculating matrix C. It is finding the dot product from a row in matrix A and a column in matrix B. The result is stored in matrix C.



```
void multiplyMatrix(void* args)
{
    MultiplyTask* task = (MultiplyTask*)args;

    // Find dot product
    for (int i = 0; i < SIZE; i++)
    {
        task->c[task->row][i] = 0;
        for (int j = 0; j < SIZE; j++)
        {
            task->c[task->row][i] += task->a[task->row][j] * task->b[j][i];
        }
    }
}
```

Here is the code that instantiates the threads:

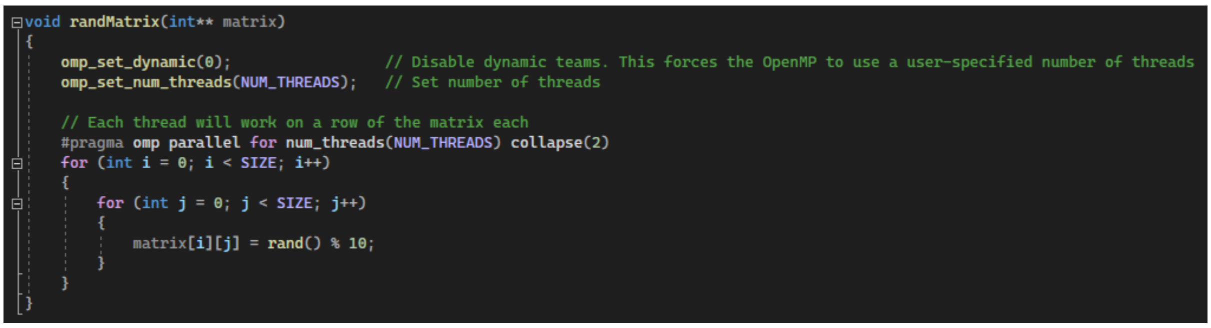


```
for (int i = 0; i < SIZE; i++)
{
    MultiplyTask* multiply_task = (MultiplyTask*)malloc(sizeof(MultiplyTask));
    multiply_task->a = m_A;
    multiply_task->b = m_B;
    multiply_task->c = m_C;
    multiply_task->row = i;

    multiplyThreads[i] = std::thread(multiplyMatrix, multiply_task);
}
```

OpenMP Program Decomposition

The OpenMP version of the program is almost identical to the sequential version of the program. The code is structured exactly the same. The only difference is that the OpenMP version includes calls to OpenMP functions that parallelise the code automatically.

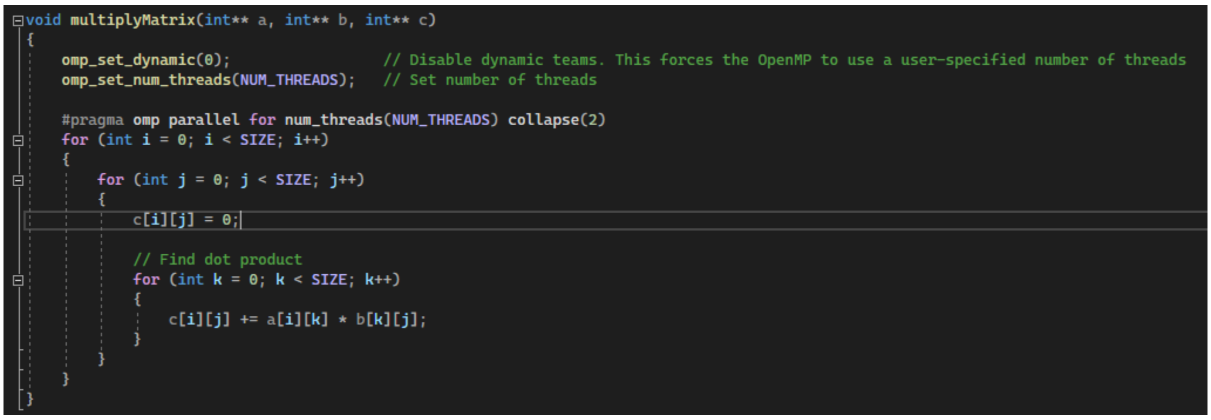
The code below shows the function that allocates the memory for each of the matrices. It is exactly the same as the sequential code.



```
void randMatrix(int** matrix)
{
    omp_set_dynamic(0);                 // Disable dynamic teams. This forces the OpenMP to use a user-specified number of threads
    omp_set_num_threads(NUM_THREADS);   // Set number of threads

    // Each thread will work on a row of the matrix each
    #pragma omp parallel for num_threads(NUM_THREADS) collapse(2)
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            matrix[i][j] = rand() % 10;
        }
    }
}
```

A feature called dynamic teams needed to be disabled to allow the number of threads to be defined by the program.

Here is the code for multiplying the matrices together:



```
void multiplyMatrix(int** a, int** b, int** c)
{
    omp_set_dynamic(0);                 // Disable dynamic teams. This forces the OpenMP to use a user-specified number of threads
    omp_set_num_threads(NUM_THREADS);   // Set number of threads

    #pragma omp parallel for num_threads(NUM_THREADS) collapse(2)
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            c[i][j] = 0;

            // Find dot product
            for (int k = 0; k < SIZE; k++)
            {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```