Matrix Multiplication with OpenMP

Rahul Ganesh Prabhu

2018A7PS0193P

The Problem

Matrix multiplication is a binary operation producing a matrix C from two other matrices A and B. A serial implementation for matrix multiplication in C can be seen below:

```
// Assume multidimensional arrays A,B and C have already been
// initialized.

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < p; j++)
    {
        for (int k = 0; k < q; k++)
        {
            C[i][j] += A[i][k] * B[k][j]
        }
    }
}
```

The assignment was to parallelize matrix multiplication with OpenMP in the following different ways:

(a) Parallelizing only the outermost loop.

(b) Parallelizing the outermost loop as well as the middle loop.

(c) Parallelizing all three loops.

The Implementation

Partitioning

Matrix multiplication can be decomposed into tasks of calculating each element in the matrix by the formula:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

These tasks can *further* be decomposed into simpler tasks of multiplying $A_{i,k}$ and $B_{k,j}$ and adding it to $C_{i,j}$. The resulting number of tasks depends on the dimensions of the matrices used. This is a form of output data decomposition, where we partition the problem into tasks based on the output space.

Communication

OpenMP is a shared memory platform. This means that we do not need any explicit communication structure since the threads write to and read from the same shared memory space.

The results matrix C must be shared among all the threads, since every thread adds the product of $A_{i,k}$ and $B_{k,j}$ to $C_{i,j}$, while A and B can be private or shared among the threads since they are read only.

Agglomeration

The agglomeration of partitioned tasks differs between the three cases.

In the first case, we parallelize only one loop. This amounts to agglomerating the tasks into a single "supertask" of calculating a single row of C. It is beneficial to agglomerate it this way as it ensures we are able to take advantage of cache locality.

In the second case, we parallelize two loops. Here, there is agglomeration into smaller tasks, where every thread only calculates *part* of a row of C.

In the third case, we parallelize every loop. This further fragments and the agglomerated tasks are even smaller. Once again, threads only calculate a part of a row of C, even smaller than before. However, this further fragmentation leads to slowdown, since we lose the benefits of cache locality.

Mapping

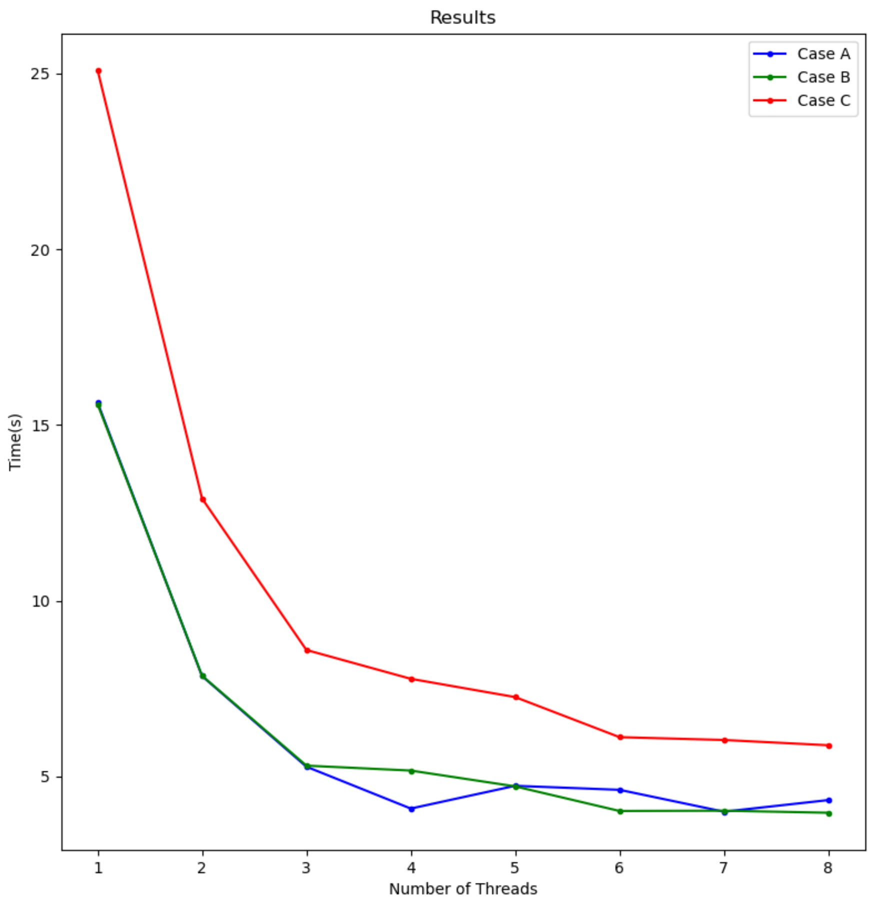
Tasks are mapped to threads by the means of OpenMP's `schedule` clause. Here, we use the static scheduling system. This means tasks are mapped statically to the threads in a round robin fashion, each getting a block of tasks. For instance, in case (a), this means each row receives a contiguous block of rows. This type of mapping is helpful for two reasons - firstly, we are able to take advantage of cache locality, and secondly, we do not need to worry about several ”blocks” overlapping, and don't need to enforce operation atomicity, which can be computationally expensive.

Other Details

The matrices are internally implemented as arrays allocated on the heap, so that they are contiguous in memory improving cache access. Also, B is internally stored as its transpose. This is also to take advantage of cache locality during multiplication, as now it will be accessed along the row instead of along the column. This can also empirically be seen to be faster.

Results

The implementation was tested by multiplying two 1000x1000 arrays, seen in `tests/input.txt`. The results can be seen in the plot below:



While case (a) and (b) are mostly equivalent, we see a slowdown when it comes to case (c). This is probably because we lose efficient use of the cache in (c).