Parallel Matrix Multiplication

cs15btech11024, cs16btech11016

Parallel Matrix Multiplication:

- To multiply two matrices, one can naively assign threads to compute each element of the product matrix. While this may seem parallel, this involves serious thread creation overhead.
- A more effective way is to pool threads and assign each of them a short-lived computation. Each thread performs the task and picks the next task from the available tasks list.
- Consider matrices whose dimension n is a power of 2. Any such matrix can be decomposed into four submatrices. As in the following figure, the eight product terms can be computed in parallel, and when those computations are done, the four sums can then be computed in parallel.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{pmatrix}$$

Work Balance through Work Stealing:

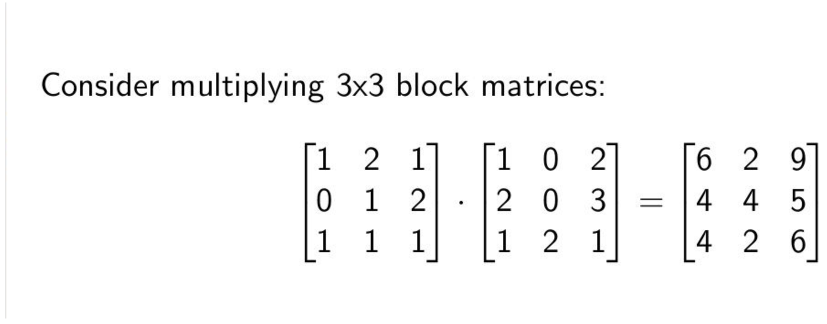
- A work distribution algorithm is needed to assign the tasks to idle threads. Cause, in most of the cases, multithreaded computations create and destroy tasks dynamically in unpredictable ways.
- Work Stealing is a simple work distribution approach in which a thread that runs out of work tries to “steal” work from the heavily loaded threads.
- Every thread keeps a pool of tasks writing to be executed in the form of a double-ended queue (DE Queue) which has the following methods
    a. **pushBottom()**
    b. **popBottom()**
    c. **popTop()**
- When a thread creates a new task it calls pushBottom() (it adds the task to the DEQueue of the particular thread). Similarly to remove and execute a task from the DEQueue it calls popBottom().
- All threads share an array of DEQueues and we can implement an efficient linearizable DEQueue.

- If a thread discovers its DEQueue to be empty then it randomly chooses some other thread (victim thread) and calls that thread's DEQueue 's popTop(), which is nothing but stealing of work from the other thread.

<u>SUMMA Algorithm</u>

- .SUMMA:Scalable Universal Matrix Multiplication Algorithm
- Slightly less efficient, but simpler and easier to generalize.
- The SUMMA algorithm computes n partial outer products:
```
for k := 0 to n − 1
C [:, :] + = A[:, k] · B[k, :]
```
- Each row k of B contributes to the n partial outer products
- Compute the sum of n outer products
- Each row and column (k) of A and B generates a single outer product
A[:, k + 1] · B[k + 1, :]
```
 for k := 0 to n − 1
          C [:, :] + = A[:, k] · B[k, :]
```
- Compute the sum of n outer products
- Each row and column (k) of A and B generates a single outer product
A[:, n − 1] · B[n − 1, :]
```
for k := 0 to n − 1
          C [:, :] + = A[:, k] · B[k, :]
```

For example consider a 3x3 matrix



Consider multiplying 3x3 block matrices:

$$\begin{bmatrix}1 & 2 & 1\\0 & 1 & 2\\1 & 1 & 1\end{bmatrix} \cdot \begin{bmatrix}1 & 0 & 2\\2 & 0 & 3\\1 & 2 & 1\end{bmatrix} = \begin{bmatrix}6 & 2 & 9\\4 & 4 & 5\\4 & 2 & 6\end{bmatrix}$$

	1	0	2
1	1	0	2
0	0	0	0
1	1	0	2

	2	0	3
2	4	0	6
1	2	0	3
1	2	0	3

	1	2	1
1	1	2	1
2	2	4	2
1	1	2	1

- When we sum all the entries we get the following matrix:

$$\begin{bmatrix} 6 & 2 & 9 \\ 4 & 4 & 5 \\ 4 & 2 & 6 \end{bmatrix}$$

Parallel Tilling

- Instead of calculating the whole matrix at once, we are dividing the matrix in to the blocks and then we are doing the matrix multiplication. By doing this we can make use of our cache efficiently i.e cache hit rate increases.

Code for tiling: this is the function to which we create threads. This shows that each thread initially will work on a row of resultant matrix after that it chooses another row by adding (num_threads) to its present row number.

void multiply(int id, int num)

```
{
   int l=id;
   while(l<m)
   {
         for(int jj=0;jj<z;jj= jj+2)
         {
                  for(int kk=0;kk<n;kk=kk+2)
                  {

                           for(int i=jj;i< min(jj+2,z);i=i+1)
                           {
```

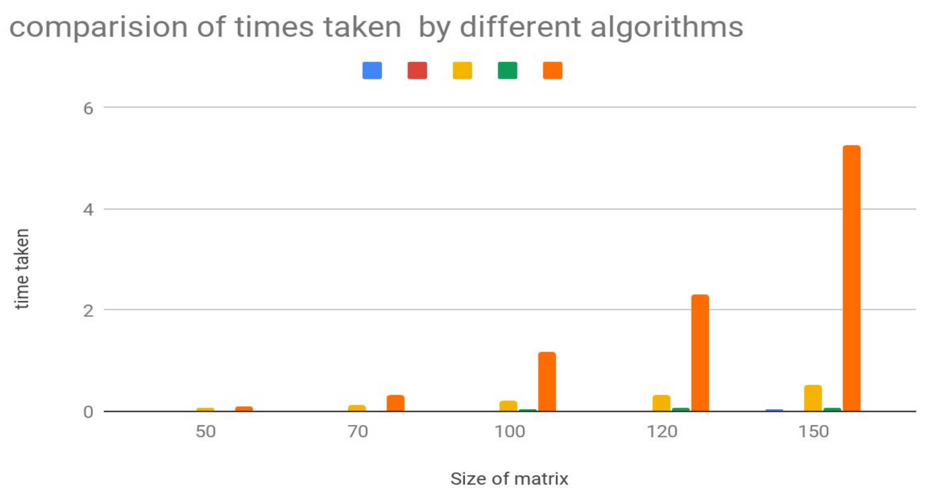
```
                                        int r=0;
                                        for(int k=kk;k<min(n,kk+2);k=k+1)
                                        {
                                                  r=r+a[l][k]*b[k][i];
                                        }
                                        c[l][i] = c[l][i] + r;
                              }

                    }
          }
   l=l+num;
   }

}
```

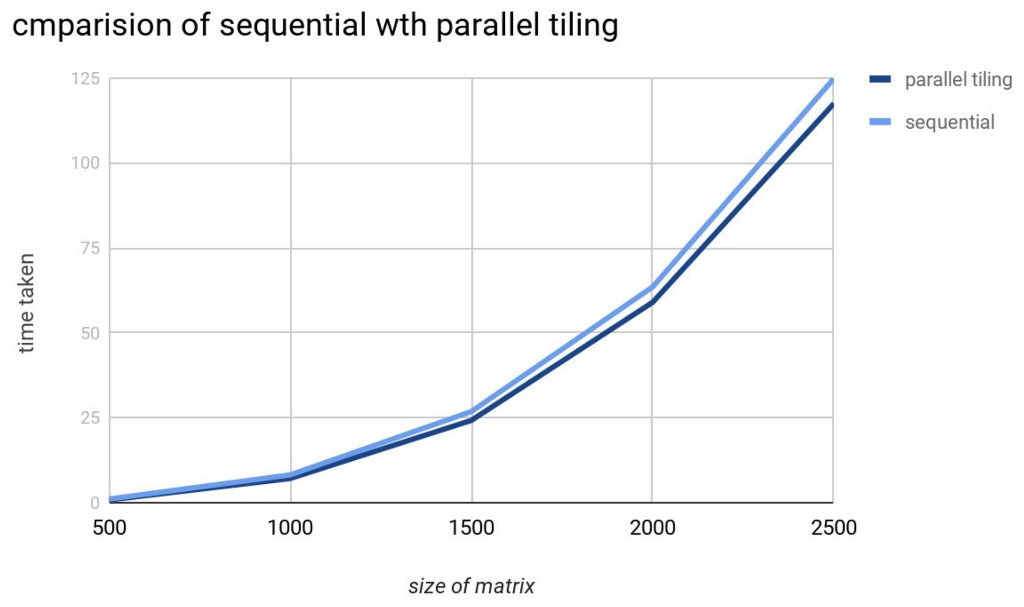
Comparison

Following is the plot showing the performance of different matrix multiplication algorithms on varying the size of the matrix.

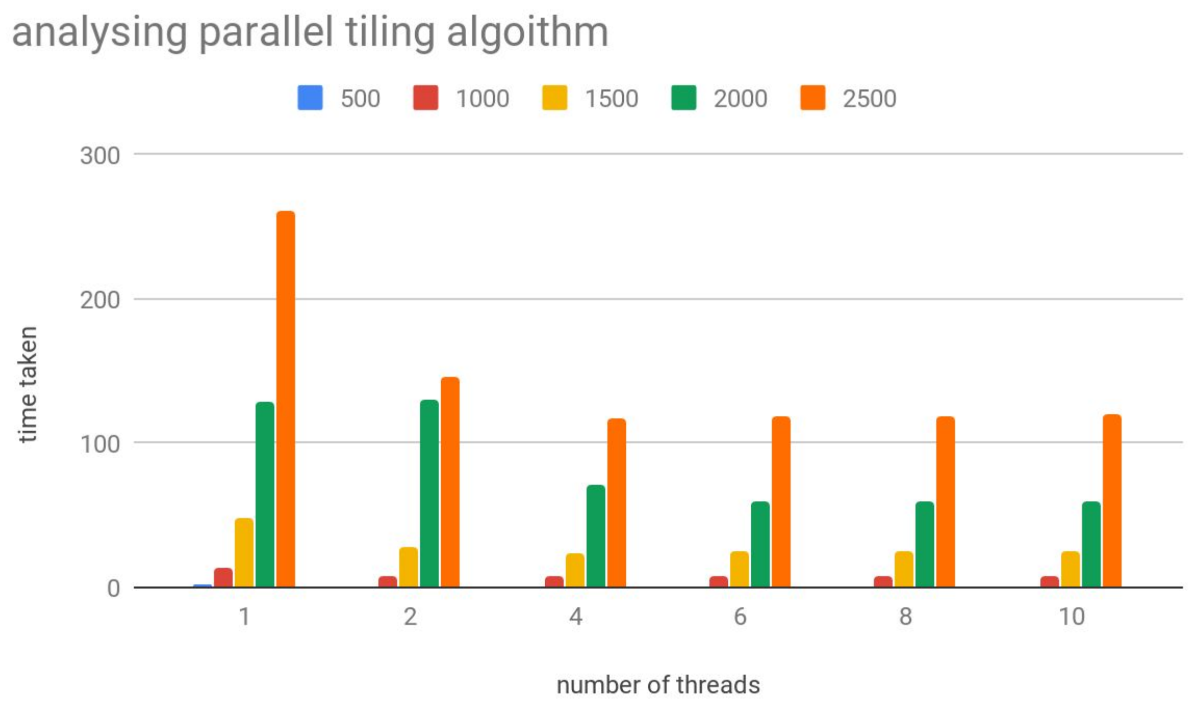


We see from the following graph the algorithm with orange colour is taking long time to perform matrix multiplication,it is even greater than sequential time.

This is method is work stealing and it takes too much time because of memory over head. Because it has to store the all 3 matrices and it has separate queue for each of the thread (array of queues which is global) and for every task it has to come to access queue.
So there are too many memory accesses. This slows down the algorithm greatly.



From the graph it is evident that parallel tiling is better than the sequential because of thread parallelism, and efficient use of hardware cache. So there is rise in performance.
As the thread number increases thread overhead which may reduce its performance



From the all the above graphs and observations, it is evident that our cache tiling algorithm is better than all other matrix multiplication algorithms implemented.
So detailed analysis of algorithm is provided in graph.
Time variance with threads and size of matrix.