Assign_1Report_ES22BTECH11026

28 January 2024 18:07

Implementation of Mixed-Technique

In this technique, work is uniformly distributed between the K threads. Since the **Thread1** will be responsible for the rows:1, k+1, 2*k+1 .... Similarly,
Thread2 will be responsible for the following rows of the C matrix: 2, k+2, 2*k+2 ....This ensures that no scenario of a load imbalance arises among the K threads.
Take the case of N=8, K=3.
Here the THREAD1 computes the rows 1,4,7 of the matrix C.
THREAD2 computes the rows 2,5,8.
THREAD3 computes rows 3,6.

Since you can clearly see, that work is almost uniformly distributed among the K threads.

Implementation of Chunk-Technique

In this technique, the rows of the matrix A(whose square is to be computed) are divided into K chunks each of size p= N/K. Then thread1 will compute the values of the **rows 1 to p of C**; thread2 will compute the values of the rows **p + 1 to 2* p**; thread3 will compute the values of the rows 2*p + 1 to 3*p and so on.

But the Mixed-technique proves to be the **better technique** of distribution of workload among the K threads.
Since, in the Chunk-Technique, there's a **high probability that the last thread might get a chunk size of more than p**..
i.e from (K-1)*p + 1, N.

Take the case similar to that taken in the previous example, N=8, K=3
Chunk size (p) = 2
Here the THREAD1 computes the rows 1,2 of the matrix C.
THREAD2 computes the rows 3,4.
THREAD3 computes rows 5,6,7,8.

Clearly there's a load imbalance among the threads since the last threads has to compute 2 more rows. This imbalance would not have been created had N%K = 0.
But, since the cases we have worked upon doesn't have N%K!=0..such a momentous runtime difference can't be observed between the runtimes for the two techniques.

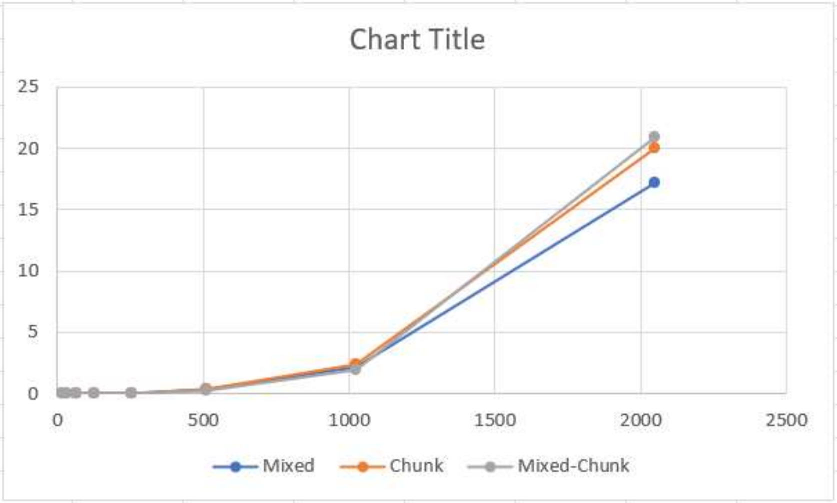
Implementation of Mixed-Chunk Technique
Basically, this technique is the combination of the previous two techniques.
I take here instead chunk sizes of **p = N/2K**;

Hence, the whole **N rows are distributed into 2K chunks**. I intend to give each thread 2 chunks, i.e each thread works upon N/K rows but with a slight modification.

Take the first thread for example. **It would work upon the 1st chunk(of N/2K rows), and the N/2th chunk(of similar size).**
Similarly, the 2nd thread works upon the 2nd chunk and the N/2th +1 chunk(of same sizes).
Hence, the advantage of this is that each thread gets to **work upon equal number of rows**, without any imbalance of workload.

Graphs given below:

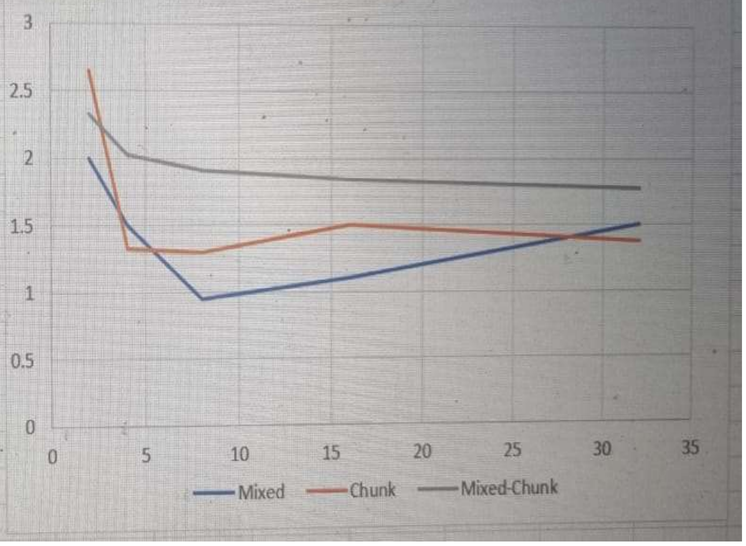
	TIME(in seconds)		
N	Mixed	Chunk	Mixed-Chunk
16	0.003	0.001	0.003
32	0.008	0.007	0.007
64	0.011	0.005	0.009
128	0.016	0.012	0.01
256	0.059	0.065	0.042
512	0.404	0.35	0.305
1024	2.124	2.37	1.944
2048	17.23	20	20.96



TIME v/s SIZE,N

The no. of threads K=8 (fixed). X-AXIS denotes the value of N(size of input matrix). Meanwhile Y-Axis denotes the time(in seconds).

	N = 1024		
K	Mixed	Chunk	Mixed-Chunk
2	2	2.65	2.33
4	1.5	1.33	2.03
8	0.95	1.3	1.91
16	1.1	1.5	1.84
32	1.5	1.37	1.77



TIME v/s Number Of Threads, K

Here, the input matrix size, N is fixed at 1024, The X-Axis denotes the values of the no. of threads varying from 2-32. The Y-Axis denotes the values for the time(in seconds).

INTERPRETATION OF THE OBTAINED GRAPHS

As I had described earlier, one would expect the Chunk-technique to create a load imbalance among the threads and hence we would expect it's runtime to be slightly more than that for mixed-technique. That's has been the case as one can see from the graphs above. For smaller input sizes, the difference is almost negligible, but it becomes more significant as the value of N increases.

Also note the **performance of my 3rd technique**, i.e the mixed-chunk approach. It performs slightly better than the other two techniques as the size of N becomes very high.

Meanwhile, for the **2nd graph, as one can see that the Mixed-technique performs better than both chunk & mixed-chunk technique**. It can be owed to that since a thread is **contiguously working upon rows with step size of k**, while in mixed-chunk technique, the thread has to work upon two **different** chunks.

BUT AS SUCH, not a very astronomical difference in run times.

I have already described the role of each function in the code itself(using comments).

Difficulties faced:

One of the difficulties I faced was to pass multiple arguments to the thread function "Compute_Cij". Since the function only accepts one argument, I had to make a structure containing all the values that I would need to use in the thread function. I also had to make an array of structures, since initially I was facing "Segmentation Fault-core dumped" error, since multiple threads were trying to access the same structure. I had to allocate memory to each of the structures in the array.