Projects on **Parallel and Distributed Computing**

by Bhargav Kulkarni

December 2024

Some of my projects on parallel and distributed computing have been described. Any remarks, suggestions/improvements and new project ideas can be dropped at **bhargavkul199@gmail.com**.

1 Parallel Matrix-multiplication

Aim: To parallelize matrix-multiplication.

An attempt has been made to apply and realize the effects of parallel computing. Matrix multiplication is a very widely used computational task/algorithm that is by nature computationally heavy. Image processing, Neural Networks, system of equations are only some of the areas of computation that involve matrix-multiplication. In this project, **we optimize the matrix-multiplicatio** by computing it **in a parallel manner** over multiple threads and **demonstrate improvement in runtime** almost linearly.

Tools: C, OpenMP

Algorithm: We consider the matrix-multiplication of square matrices A and B into matrix C. Let N be the dimension of the square matrices. The sequential multiplication of matrices A and B is done as follows:

```
for(int i = 0;i < N;i++)
    for(int j = 0;j < N;j++) {
        C[i][j] = 0.0;
        for(int k = 0;k < N;k++)
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
```

Let the algorithm be denoted by $seq_{M.M}$. The algorithm involves the time complexity of $T_{seq_{M.M}}(N) = \mathcal{O}(N^3)$.

Scope of parallelism : In hoping to parallelize any algorithm, we have to check if the algorithm is **parallelizable**. Any sequential program possibly has a section that is **amenable to parallelization**, meaning the task can benefit by organizing it among multiple processing units. In the case of matrix-multiplication algorithm, we can see that computation of each cell C_{ij} is independent of other cells. This means in the availability of multiple processing units, we can task each processing unit with the computation of distinct cells. The task of matrix-multiplication is hence a heavily parallelizable task.
Since computation of each C_{ij} is distinct, we can divide the space of (i, j) among p available

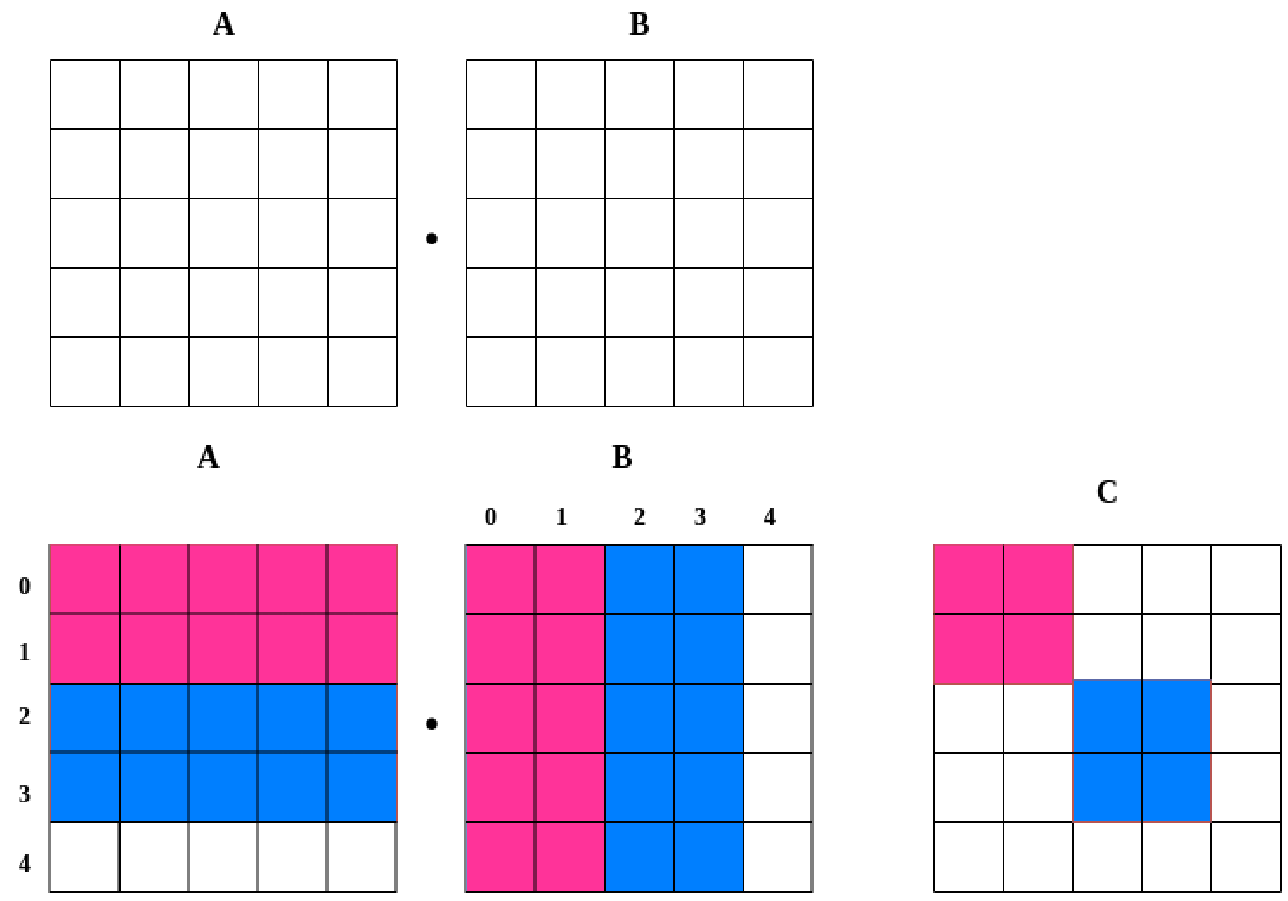

Figure 1: Parallel computation of matrix-multiplication

processors. Each processor can then independently compute C_{ij} for the set of (i, j) it is assigned and store the result in the memory of `C`.

Fig 1. demonstrates the parallelization. If Thread 1 and Thread 2 are involved in parallely computing the result, they must be assigned set of cells for which they have to compute the dot product. When Thread 1 is assigned (i, j) values of $(0, 0), (0, 1), (1, 0), (1, 1)$, it computes (sequentially) $C_{00}, C_{01}, C_{10}, C_{11}$. Similarly when Thread 2 is assigned $(2, 2), (2, 3), (3, 2), (3, 3)$ it computes the corresponding values of C matrix. This is exactly what we will achieve using OpenMP.

OpenMP is a framework in C that offers multi-core programming functionality. With the addition of just a few lines of code to the existing code, a sequential program can be converted into a parallel one with the help of OpenMP. We will divide and assign the space of (i, j) to different threads by parallelization of `for` loops. The appropriate code that achieves this is:

```
#pragma omp parallel for collapse(2)
for(int i = 0;i < N;i++)
    for(int j = 0;j < N;j++) {
        C[i][j] = 0.0;
        for(int k = 0;k < N;k++)
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
```

OpenMP internally collapses 2 outer-`for` loops into one and partitions the combined iteration space of (i, j) among the set number of threads. Let us denote the parallel algorithm as $par_{M.M}$.

Let p be the number of threads (1 per processor-core) on which the parallel algorithm runs on. As all the computation is divided among threads and each thread is loaded equally, computing totally independently of other threads, the theoretical run-time complexity of $par_{M.M}$ is

$$T_{par_{M.M}}(N,p) = \mathcal{O}(N^3/p) \tag{1}$$

The expected speedup of this algorithm would be

$$S_{par_{M,M}}(p) = \frac{T_{seq}}{T_{par}} = p$$

which is the **maximum speedup achievable**.

Results: Both the algorithms were benchmarked on square matrices of size 1000[1]. The host machine has Intel 13th Gen i7 1360P processor containing **16 physical cores**. The execution times of the parallel algorithm was measured over different number of threads (number of threads controlled during runtime using OpenMP shell variables).

Sequential algo.	Parallel algo.	
	4 threads	16 threads
6.800	1.656	0.861
6.445	1.653	0.839
6.129	1.631	0.815

Table 1: Execution times of the algorithms (in sec) for matrix-multiplication of size 1000

A significant improvement in compute times can be observed, indicating that the **task of matrix multiplication is highly parallelizable**.
To measure the speedup achieved, the parallel algorithm was run with 4, 8, 12 and 16 threads. Speedups corresponding to the number of threads were measured and plotted. Speedup is

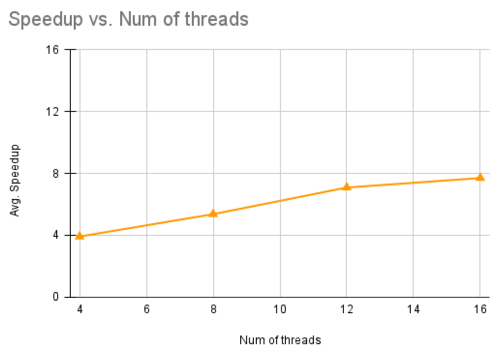

Figure 2: Plot of speedup obtained against number of threads

found to flatten as the number of threads increase. This could be due to overhead of creation of thread-pool and synchronization at the end of loop.

[1]Matrix of such large dimensions have been allocated in the heap memory in a dynamic fashion. This is because the stack memory is small and does not fit very large arrays

1.1 Memory optimizations

While the parallel algorithm significantly improves on the original, sequential one, there is still scope for improvement. The implementation can be fine tuned for ease of memory access, which will further reduce the runtime of the algorithm. While accessing `B[k][j]` in the inner loop where `k` varies, each time the program fetches (a part of) the row `B[k]`. This breaks memory access each time. To improve over this, a (global variable) `flat_B` is created to store elements of `B` in a column major order. Accesses to `flat_B` would be sequential.
The fine-tuned implementation of the algorithm is as follows:

```
for(int i = 0;i < size;i++)
    for(int j = 0;j < size;j++)
        flat_B[j*size + i] = B[i][j];

#pragma omp parallel for collapse(2)
for(int i = 0;i < size;i++) {
    for(int j = 0;j < size;j++) {
        int j_off = j*size;
        double tot = 0.0;
        for(int k = 0;k < size;k++) {
            // Elements of B are accessed sequentially
            tot += A[i][k]*flat_B[j_off + k];
        }
        C[i][j] = tot;

    }
}
```

Results: The runtimes of the memory optimized algorithm was compared against the unoptimized parallel algorithm for different sizes of matrix over **8 threads**.

Size of matrices	Execution times (in s)	
	Parallel algo.	Memory optimized parallel algo.
250	0.024	0.017
500	0.148	0.141
1000	1.193	0.723
1500	5.564	2.445
2000	16.582	5.737

Table 2: Execution times of the algorithms (in sec) for matrix-multiplication of size 1000

Improvement in runtime is significant, indicating that memory accesses constitute a notable portion of total time and its **optimization is hugely beneficial for large matrices**.

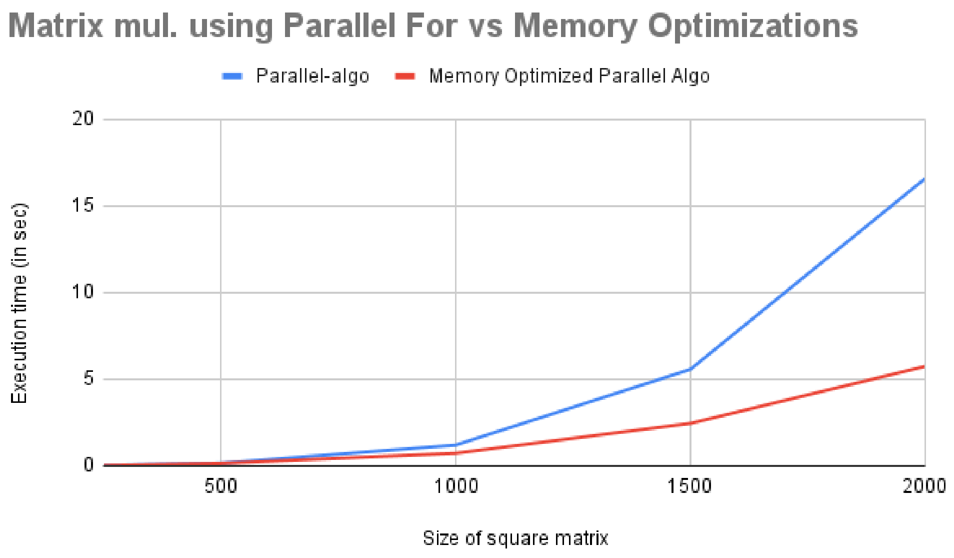

Figure 3: Comparing runtimes of memory-optmized algorithm against the naive parallel algorithm