MatrixOps

Michael Edelman

December 2021

1 Design

1.1 Addition Algorithm

The addition parallel algorithm decomposed the problem of matrix addition into subtasks of matrix addition by partitioning both nXn matrices A and B into disjoint layers. Each of those layers would be partitioned further and so on and so on until the number of rows n each matrix was $\leq$ the specified threshold. The algorithm would then add the corresponding rows in A and B to each other. This implementation relies on the observation that a nxn matrix can be divided into sub-matrices provided $n > 1$ and in our case $2^x = n$.

1.2 Multiplication Algorithm

The multiplication parallel algorithm first recognized that multiplying two nxn matrices is actually the sum of n matrices. Each column of A is multiplied with Matrix B, where the i^{th} element of a column of A scales/multiplies each corresponding i^{th} row vector of B resulting in a new matrix. If done for every column in A, n matrices will be obtained. The sum of these n matrices is the product of AxB. Since matrix addition is commutative the parallel algorithm decomposes the n matrices to be summed together into two disjoint sub problems of 1 ... n/2 matrices and ((n/2) + 1) ... n matrices. Each of these sub problems would be further divided until their respective sizes were $\leq$ the specified threshold (That was the fork). Afterwards, the multiplication algorithm would use the add() algorithm to combine the resulting matrices (this is the join).

2 Experiments

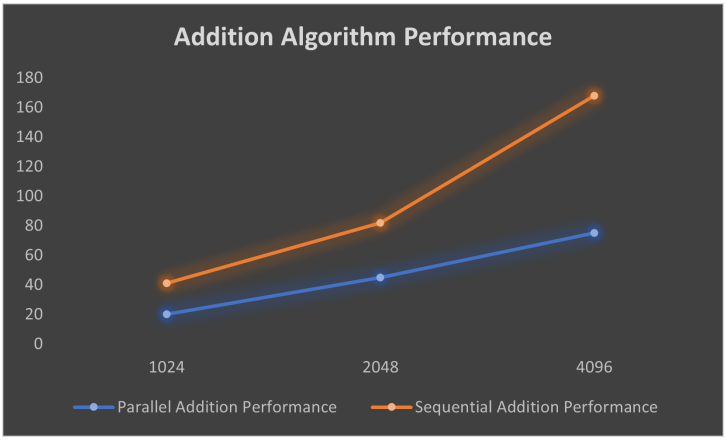

Figure 1: Addition Algorithm Performance, Y axis in milliseconds

Thresholds that produced best results are 512 for n = 1024, 512 for n = 2048, and 4096 for n = 4096.

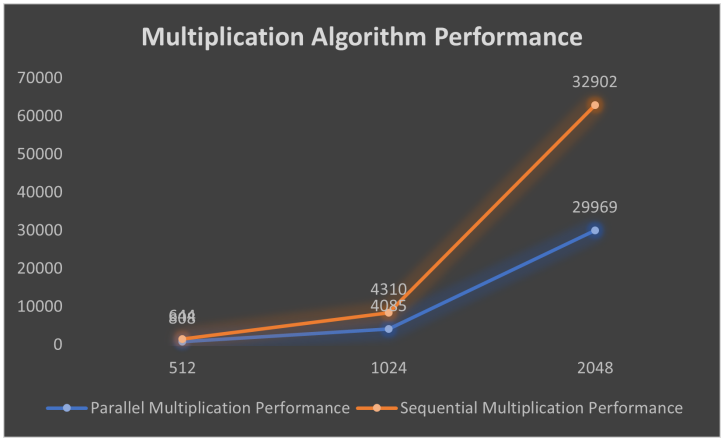

Figure 2: Multiplication Algorithm Performance, Y axis in milliseconds Thresholds that produced best results are 512 for n = 512, 512 for n = 1024, and 1024 for n = 2048.

Table 1: Sequntial Performances

Dimensions n	**Addition Performance** (ms)	**Multiplication Performance** (ms)
512	10	656
1024	20	4310
2048	32	32902
4096	98	
8192	375	

3 Analysis

3.1 Which Algorithm Performed Best?

The JDKFJMatrixOps Addition Algorithm performed best for the larger input sizes compared to its sequential counterpart, particularly when n = 4096. Notably its performance was not as good for when n = 8192. The JDKFJMatrixOps Multiplication Algorithm performed best for when n = 2048 - the largest data size tested.

3.2 Why did an Algorithm perform well for some n values and not others?

In general as n gets larger we would expect to see larger performance gains relative to the sequential algorithm for the parallel ones. this is because thread creation and synchronization are costly tasks and each needs to perform a sizeable amount of work to offset that expense. This general trend was observed, but it was not perfect as noted earlier. the parallel algorithm performed better for n = 8192 than when n = 4096 in the addition algorithm.

3.3 What role did the threshold parameter play?

The threshold parameter is used to determine at what point a sub task should be processed sequentially or further "forked" into subtasks. In terms of performance it would be expected that a threshold that allowed the task to be subdivided into some constant of the number of cores/hardware threads on ones machine would result in large performance gains for the parallel algorithm. However, in this assignments experimentation that concept was only partially observed. The performance did improve, but usually only for thresholds half the size of n. My machine has 8 logical processors so smaller thresholds than that would be expected to improve performance, however performance usually stagnated or deteriorated after more than one division of n.