Final Project : Fox's Algorithm

1 Introduction

Matrix-Matrix multiplication is central to many numerical algorithms that drive modern scientific computing. A lot of research has been done to optimize the mathematical implementation's execution time targeting different types of hardware including parallel and distributed systems. There exist multiple algorithms that achieve a great degree of improvements over the mathematical bounds of $O(n^3)$ and have managed to reduce it to $O(n^{2.375477})$ in Coppersmith–Winograd algorithm. Even then it is unclear which is the best algorithm as the fastest algorithms tend to be computationally inefficient and the choice of optimal algorithm changes with the size of the matrices to be multiplied.

In this project we analyze the performance scaling of the matrix-matrix multiplication algorithms specifically Fox's algorithm and compare it with in-depth performance modelling. We shall analyse the opportunities of acceleration Beskow's hardware and optimizations to any matrix-matrix multiplication problem.

Parameters such as Size of Matrices($N \times N$), Size of submatrices($N\_BAR \times N\_BAR$) and number of Parallel threads($PROCS$) are central to our analysis and optimization. We shall focus on strong scaling results and verify the results against test cases. Eventually draw conclusions on the speedup attained and codependency of software and hardware.

2 Assumptions and limitations

To narrow down our scope we have made certain assumptions in our design.

1. We shall limit precision to 6 decimal places
2. To correctly execute the program it is mandatory to satisfy the equation

$$\frac{N}{N\_BAR} = \sqrt{PROCS} \tag{1}$$

   (a) The equation should always be balanced for proper execution
   (b) The individual values of the variables must be whole numbers

3. Matrices are square - This simplifies the multiplication and division of the matrix into submatrices

3 Methodology

3.1 Theoretical analysis / Performance Model

Naive Matrix-Matrix multiplication :

```
for (int c = 0 ; c < N ; c++ )
    for (int d = 0 ; d < N_BAR ; d++ )
        for (int k = 0 ; k < N_BAR ; k++ )
            MatC[c][d] += MatA[c][k]*MatB[k][d];
```

There are 2 operations, 2 loads and 1 store in the line computing MatC and there are 3 nested loops, hence the algorithm theoretically runs at TCops, where $t_{ops}, t_{L\_mem}, t_{S\_mem}$ are time taken for one ALU operation, Load operation, and Store operation respectively.

$$TC_{ops} = n^3 \times (2t_{ops} + 2t_{L\_mem} + t_{S\_mem}) \tag{2}$$

In Fox's Algorithm the Matrices are divided into smaller chunks using grid decomposition[bibitem] and allocated a process to compute the value for that section of the matrix. Fox's algorithm follows the naive algorithm's time complexity in sub-matrices. The algorithm has additional communication overheads. There are $\sqrt{PROCS}$ number of stages required to compute the multiplication. Tile B takes BT_B and Tile A takes BT_A to be prepared and communicated across the processes.

$$BT_A = N\_BAR^2 t_{comm} + (\sqrt{PROCS} - 1) \times t_{startup} \tag{3}$$

$$BT_B = N\_BAR^2 t_{comm} \tag{4}$$

However, since the work is divided between PROCS the new time estimates combining Equations 1, 2, 3, and 4 we get the following.

$$Tops = \frac{N^3 \times (2t_{ops} + 2t_{L\_mem} + t_{S\_mem})}{PROCS} + \frac{2 \times N^2 t_{comm}}{\sqrt{PROCS}} + \sqrt{PROCS}(\sqrt{PROCS} - 1) \times t_{startup} \tag{5}$$

We ignore the t_{L_mem}, t_{S_mem}, and $t_{startup}$ to keep our model simple and separately treat the execution time and communication time thus giving us

$$TC_{ops} = \frac{2N^3 t_{ops}}{PROCS} \;;\; TC_{comm} = \frac{2N^2 t_{comm}}{\sqrt{PROCS}} \;;\; TC_{total} = TC_{ops} + TC_{comm} \tag{6}$$

3.1.1 Ideal Maximum Speedup

The ideal maximum speedup ignoring communication and memory access times would be

$$Speedup = \frac{OldTime}{NewTime} = \frac{2N^3 t_{ops}}{\frac{2N^3 t_{ops}}{PROCS}} = PROCS \tag{7}$$

3.2 Algorithm

3.2.1 Initialization

We first initialize the matrix A and B randomly in the function InitiateMatrix. This function under DEBUG definition initiates the matrix differently at the 2 levels of DEBUG.

1. When DEBUG = 2, the matrix A is generated such that the rows are populated with the incremental values and the matrix B is generated such that the columns are alternatively generated as shown in Equations 8. On multiplication it is easy to observe visually that the alternate columns should be the sum of N numbers as discussed later in Section 3.2.9.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} B = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{8}$$

2. When DEBUG = 1 or not defined, the matrices are generated randomly with a decimal number less that 1.

3.2.2 Enter MPI environment

Next we initiate the MPI environment which allows the threads to have their copy of the Matrix A and B. In bigger matrices addressing the values from the individual copies of the Matrix A and B would generate a lot of memory traffic and also lose the objective of analysing Fox's algorithm. Hence we shall restrict ourselves to using Matrix B only for initial generation of the Tile B for processing and thereafter roll the buffers to the next process. However, we have to use the Matrix A in the memory for the extraction of diagonal blocks to broadcast.

3.2.3 Creating communicators

After entering the parallel section we check for the consistency of the relationships of N, N_BAR and the number of Processes allocated as per the Equation 1. Following this we reorder the processes into Cartesian Coordinates and create a new Cartesian Communicator. We further split the Cartesian Communicator row-wise into new Row Communicators to help broadcast the sub-matrices of A.

3.2.4 Generate Tile B

We generate the tile B for the specific process based on its coordinates in the Cartesian process grid. For example rank 0 would extract the top left quarter of the matrix as shown below.

$$MatB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad SubarrayB = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \quad or \quad SubarrayB^T = \begin{bmatrix} 1 & 5 \\ 2 & 6 \end{bmatrix} \tag{9}$$

3.2.5 Generate Tile A, Broadcast(if diagonal), and multiply

There are N/N_BAR or $\sqrt{PROC}$ stages that the Fox's algorithm goes through. At every stage the diagonal matrices are identified, and verified if the process is the sender. Based on calculations (refer source code) we may generate the A tile and broadcast it to the processes belonging to the same row. Otherwise, we receive the broadcasted sub-matrix A and multiply it with the broadcasted sub-matrix B.

3.2.6 Vertical roll of B tiles

We shift the Cartesian communicator by 1 displacement units to identify the vertically adjacent processes through the variables send_to and receive_from. Knowing this we issue the concurrent send and receive command to roll our B matrices upwards. Following this we return to 3.2.5

3.2.7 Gather resulting sub-matrices

Post multiplication stages we have to gather the sub-matrices and recover the larger matrix in the Matrix C. For this we define a datatype of block2d and recvsubarray.

1. The block2d is a contiguous allocation of MPI_DOUBLE and it helps us treat one sub-matrix as a unit of transaction.

2. Recvsubarray helps us define a section of matrix as the unit of transaction and we resize this data type to restore the lower bound and extent so that we can use it in our gather process without alterations.

We also define a count and a displacement array to account for the locations where the incoming matrices should be put in the larger matrix. Finally we issue the Gatherv command to collect all sub-matrices and regenerate our Matrix C which is our answer

3.2.8 Collection of Time statistics

We utilise 2 measurements of time, independently using MPI_Wtime and mysecond(). MPI_Wtime is used to measure total execution time of the process starting from the entry into MPI parallel space till gathering of solved matrix tiles into root rank. In between mysecond() is used to time the MPI_Bcast and MPI_Sendrecv routines.

3.2.9 Verification in root rank

When DEBUG>0 we enter the Debug code section in the root process where we evaluate the Matrix C against the standard naive serial matrix multiplication in the function Debug(). In case of DEBUG=2 as discussed in Section 3.2.1, the solution was visually verified as follows

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} B = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} C = \begin{bmatrix} 6 & 0 & 6 \\ 6 & 0 & 6 \\ 6 & 0 & 6 \end{bmatrix} \tag{10}$$

3.2.10 Optimizations

The matrix matrix multiplication in this sub-matrix can be optimized further using the following schemes

1. Using OpenMP to parallelize the for-loop - to further divide the processing into available threads
2. Using SIMD reductions with OpenMP - These highly efficient hardware can compute reduction operations very fast
3. Alignment of variables to aid in SIMD operations - It helps SIMD operations perform better, also performance of memory operations are improved
4. We have extracted the B matrices in a transpose fashion so that the probability of cache thrashing during the multiplication stage is eliminated. With the B matrix transposed as shown in Equations 9. It will be a one-time cost of processing for transpose, and following multiplications would work serially.

We do observe in the assembly instruction file generated using "mpicc -S -fverbose-asm -O3 -lm optimFox.c -march=haswell" which yields OptimData.s (available in Git) clearly showing multiple instructions have been successfully vectorized.

4 Experimental Setup

Here we introduce the hardware specifications of our experimentation as well as the software modules used during experimentation.

4.1 Introducing Beskow

We will be experimenting with the Beskow Supercomputer which is based on Intel Haswell and Broadwell processors with a theoretical peak performance of 2.43 petaflops. We will restrict ourselves to Haswell nodes only to ensure consistency in the results. Beskow specifications are as follows

1. 11 cabinets = 515 blades = 2,060 compute nodes
2. 2 x Intel CPUs per node:
   (a) 9 of the cabinets have Xeon E5-2698v3 Haswell 2.3 GHz CPUs (16 cores per CPU)

(b) 2 of the cabinets have Xeon E5-2695v4 Broadwell 2.1 GHz CPUs (18 cores per CPU)

3. 67,456 cores in total
4. High speed network Cray Aries (Dragonfly topology) with approximately 8 Gbps in each direction.
5. 156.4 TB primary memory (64 GB per node on the Haswell nodes and 128 GB per node on the Broadwell nodes)

4.2 Haswell Specifications

Number of Cores	16
Number of Threads	32
Processor Base Frequency	2.30 GHz
Max Turbo Frequency	3.60 GHz
Cache	40 MB Intel® Smart Cache
Level 1 cache size	16 x 32 KB 8-way set associative instruction caches
	16 x 32 KB 8-way set associative data caches
Level 2 cache size	16 x 256 KB 8-way set associative caches
Level 3 cache size	40 MB 20-way set associative shared cache
Bus Speed	9.6 GT/s
Max # of Memory Channels	4
Max Memory Bandwidth	68 GB/s

4.3 Source code

Git Location : `https://github.com/DevrajD/HPC/tree/master/HPC_DD2356_Final_Project`

4.4 Compilation

We have designed the program with a degree of dependence with C's preprocessor directives. The program is compiled and executed for certain combination of N and N_BAR which requires to be run using PROCS as per the Equation 1. This was done to simplify matrix addressing within the program and avoiding complicated pointer arithmetic. Also, DEBUG is enabled through the preprocessor definitions. We used Intel Program Environment inside Beskow, and a typical compilation command looks like the following.

cc -O3 optimFox.c -o OPFoxp -lm -D N=$N -D N_BAR=$N_BAR -D DEBUG=$DEBUG -fopenmp -mcmodel=medium

4.5 Execution

In our shell script we compile the program by setting N as a series of arbitrary values. If we have X threads made available by allocation, we choose a number in variable J and iterate to $\sqrt{X}$ and check if J divides N or not. If the check succeeds the we compute the value of N_BAR using the formula below, and run the executable using J^2 processes (PROCS).

$$N\_BAR = \frac{N}{J}$$

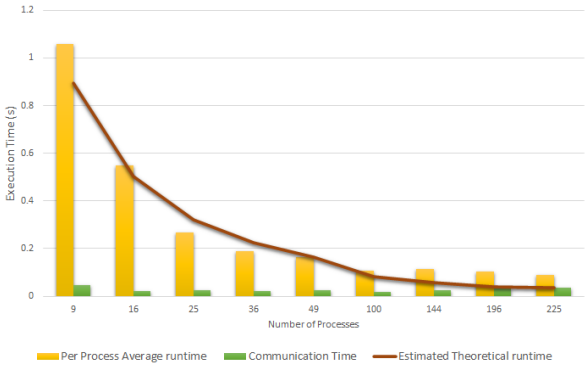
We got allocations for 8 nodes or 16 CPUs or 256 cores with 64 threads per node or 512 threads in total with Hyperthreading. We target 256 MPI processes which due to OpenMP threading would expand to 512 when thread number is set to 2, fully utilising our allocations.

4.5.1 Strategy for Strong scaling tests

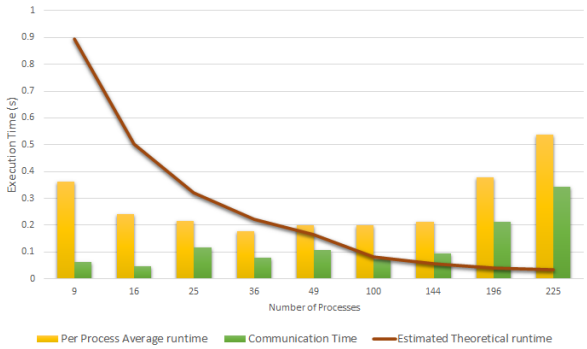
We iterate over a set of values of N which are strategically chosen to be divisible by most numbers like multiples of 6 and 10 and follow directions in Section 4.5. By this method we maximize our data points for extracting strong scaling results. N = 2100 is one of our prominent values of N which is divisible by square of multiple numbers from 2 to 16.

5 Results

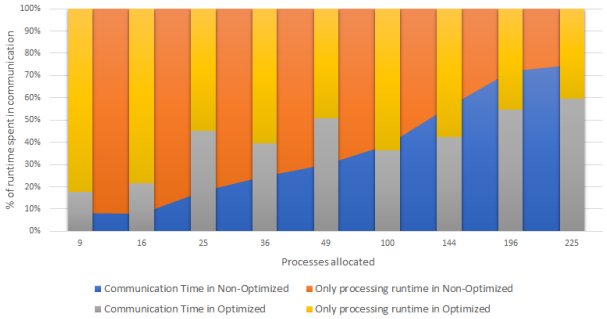
Here we present our findings on the Beskow supercomputer. The graphs are run-time graphs unless otherwise specified. The theoretical estimates are calculated as per Equation 6 using the values t_{ops} = 4.347e-10 s (from 2.3 GHz Haswell processor) and t_{comm} = 1.25e-10 s (from Cray Aries network).



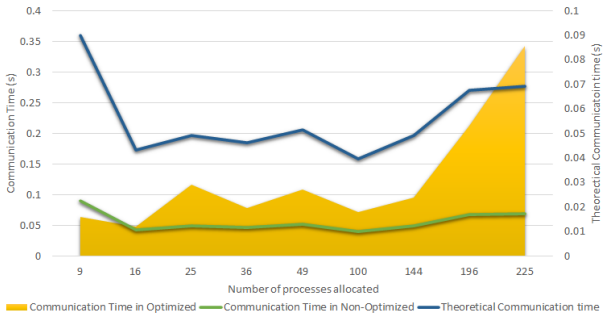
(a) Scaling results for N=2100 vs allocated processes, using non-optimized algorithm along with TC_{ops}



(b) Scaling results for N=2100 vs allocated processes, using optimized algorithm along with TC_{ops}



(c) Share of Communication time in Execution time vs allocated processes



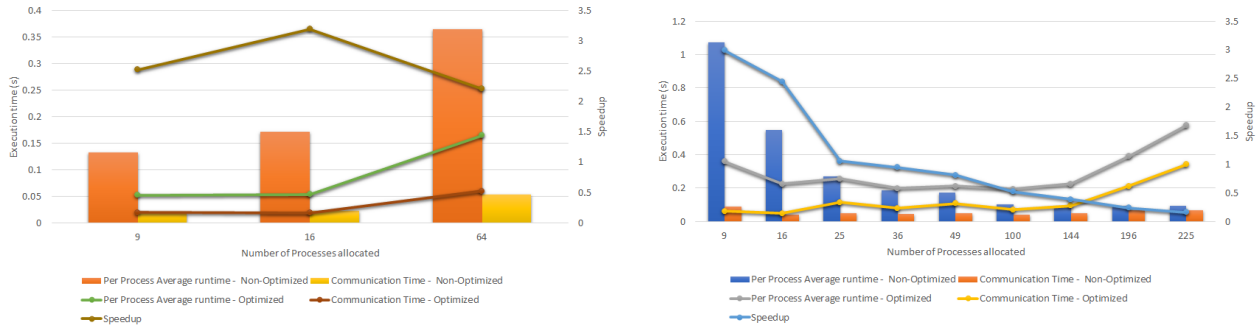
(d) Communication time vs allocated processes with TC_{comm}

Figure 5.1: Strong scaling results as observed in an allocation of 8 nodes

6 Discussion and Conclusion

6.1 Non-Optimized

From the Figure 5.1a we can see the time taken to process a NxN matrix reduces exponentially with increasing number of processes. The reduction in time stops to reduce further when the runtime is dominated by Communication time as visible in the Figure 5.1c. When the share of Communication time grows beyond 50% increasing the number of processes is futile as the benefit by increased number of process is countered by increase in Communication time.



(a) Weak Scaling Speedup for N_BAR=256

(b) Strong Scaling speedup for N=2100

Figure 5.2: Speedup graphs as observed in an allocation of 8 nodes

6.2 Optimized

From the Figure 5.1b we see an improvement in the run-time of the program. However, the improvements are counterproductive at higher number of threads due to the limitations on the thread allocations available. This worsens the effects of Communication, Resource contention and Cache coherency.

As per the Figure5.1d the optimized algorithm's communication times are greater compared to non-optimized. The local maximas of optimized algorithm in the graph follow closely with the theoretical expected times but are significant compared to the local maximas exhibited by non-optimized algorithm which has a relatively flat profile. From the Figure 5.1c we can also see that on average the Optimized algorithm has a greater share of the total run-time dedicated to the communication routines.

The utilisation of optimizations successfully improves the execution time of the program at lower number or PROCS. It is also observed that the average time spent in communication is greater in the optimized algorithm, this is because the shorter execution time increases the frequency of messages being injected into the network. Higher frequency of messages gives rise to increased contention and therefore delay in communication.

6.3 Performance comparison

Optimized algorithm is slower at higher PROCS but faster and more efficient at lower PROCS. From the Figure 5.2 we can see our optimization gives an average of 2.6x improvement when workload per MPI process size is kept constant (at 256x256 elements). Whereas from Figure 5.2 we can see the speedup deteriorates at higher number of processes. This is because we have stuffed multiple MPI Processes into one node and we have OpenMP threading enabled which spawns more threads i.e. 36 PROCS would be run using 72 threads. If we decongest the nodes and allocate PROCS number of nodes for our program we should see a greater improvement in the runtime.

6.3.1 Comparison with the performance model

We see close correlation with the theoretical model developed in the Section 3. Both $TC_{ops} and TC_{comm}$ correlates well with our experiments in Figure 5.1a and 5.1d.

7 Summary

7.1 Fox's algorithm

Dividing the matrix into smaller chunks Fox's algorithm reduces the problem size and reduces the worst case execution time as per $O(n^3)$ The smaller chunks can be subjected to other types of optimization as we see fit. The algorithm generates a substantial communication and memory overheads due to the roll of B tiles, and accessing the diagonals and broadcasting at every iteration. Due to the dependency of the algorithm on communication, it has an added bottleneck of communication systems which is usually slower than memory addressing. The algorithm itself does not accelerate how we process a matrix-matrix multiplication but only to divide the matrix into smaller manageable chunks.

7.2 Optimizations

The Transpose of the matrix from the beginning helps to avoid future transpose and is a one time startup cost. Vectorization is more efficient compared to threading, hence utilising this along with threading improves efficiency of the code. Vectorization usually are most efficient when we use the full SIMD lane, in the current implementation the threading divides the stream of data for the vector processing into sub-optimal lengths. Spawning multiple threads can lead to cache thrashing under resource constraints hence reducing the benefits of threading.

7.3 Challenges

To gather the solutions from different processes was a major challenge, after a lot of study and analysing samples, we implemented the MPI_Gather with a special datatype to collect data directly into the final matrix.

7.3.1 Further optimizations

We can utilise blocked matrix operations, and GPU calculations to speedup the Multiplication process. To reduce the communication time if we schedule the MPI threads to run on nodes lying closer to each other in the network can potentially reduce communication time.

References

Fox, G.C, et al. "Matrix Algorithms on a Hypercube I: Matrix Multiplication." Parallel Computing, vol. 4, no. 1, 1987, pp. 17–31., doi:10.1016/0167-8191(87)90060-3.

Gall, François Le. "Powers of Tensors and Fast Matrix Multiplication." 2014.

Intel® 64 and IA-32 Architectures, Optimization Reference Manual