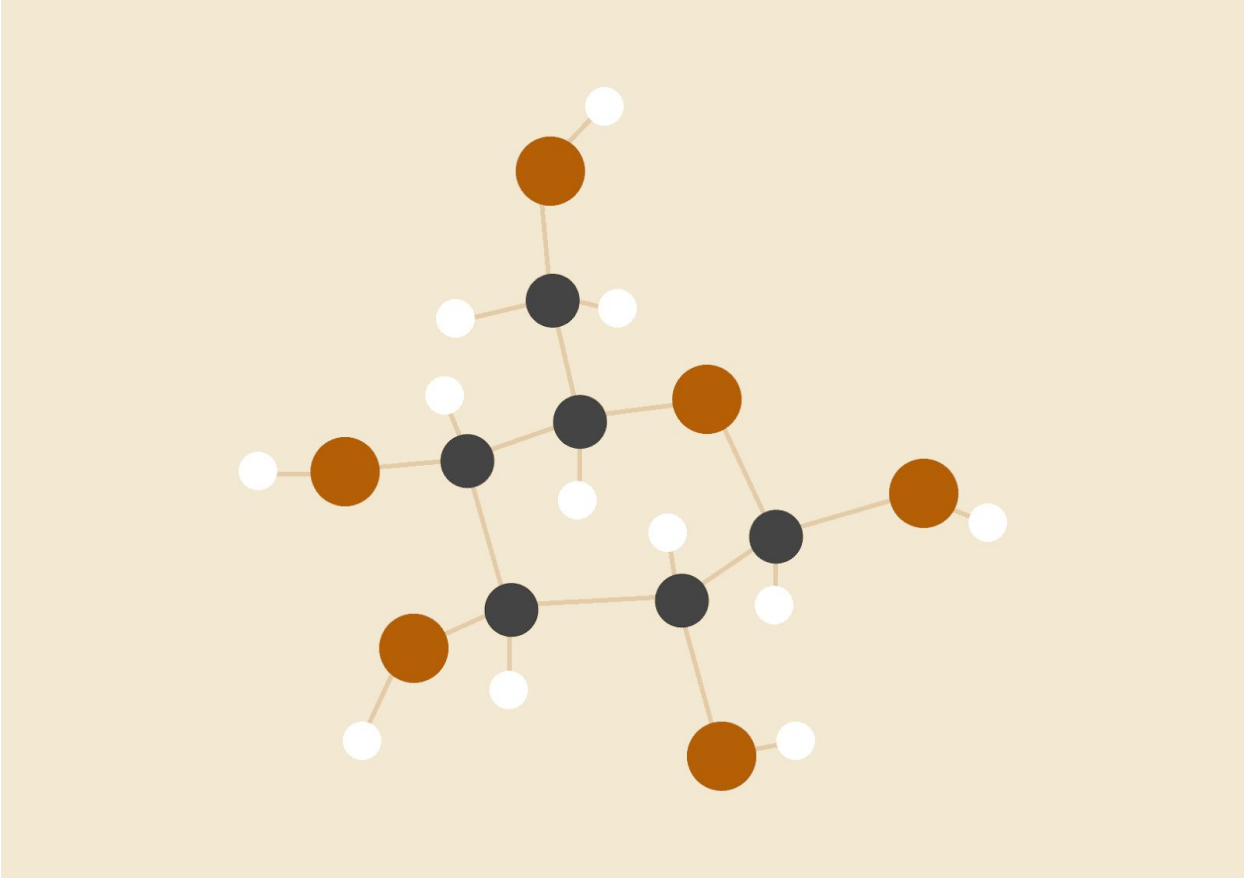
Parallel Matrix Multiplication & Cache Performance



Prabha Veerubhotla - 013785759

Nrupa Chitley - 012483276

Vinod Katta - 012420642

INTRODUCTION

Parallel Matrix Multiplication

Matrix multiplication becomes time-consuming as the size of the matrix grows. Beyond a certain size, the computer lacks sufficient resources to compute multiplication in the finite amount of time. So, to optimize the process of multiplication we have made use of multithreading by using OpenMP API. OpenMP is an API which supports shared memory multiprocessing programming in C++, C, and Fortran, across multiple platforms. In this project, we are comparing the performance of matrix multiplication with and without the use of OpenMP. We are also running this program with three different compilers: the vanilla version(non-threaded) gcc, GNU(with OpenMP) and XCODE(LLVM)(with OpenMP) gcc and comparing the results. We used language construct *#pragma* to dynamically specify number of chunks each thread can process at a time, SIMD (single instruction multiple data) to compare the results.

Cache Performance

Optimized utilization of cache is needed to achieve finite timing results of the matrix multiplication. In this program, we have analyzed the cache times for the three cache levels L1, L2, and L3 by varying the dimension of the matrix (NxN). We are only considering square matrices, in this example.

HYPOTHESIS

Parallel Matrix multiplication is efficient in terms of computational resources and time, compared to sequential(the usual way) matrix multiplication, as the matrices grows bigger and bigger..

TECHNOLOGY

1. Language - C++
2. API - OpenMP
3. Compiler - gcc

PROCEDURE

Parallel Matrix Multiplication

The program initializes two NxN matrices with random values. The matrix multiplication is performed using different compilers, mentioned above. We have conducted sequential(without threads) and parallel(with threads) tests. Parallel tests use several OpenMP compiler #pragama s. The #pragma s make sure the code yields correct behavior, without parallelism, even if the compiler does not support it. In the 3 compilers, we are using, the vanilla compiler is the non-threaded version. The code works sequentially(without parallelism), when the program is compiled with this plain compiler.

Cache Performance

In this, 2-dimensional matrix is used. We have conducted the timing analysis by varying the input matrix dimension from 32*32 (mem~32 Kb) to 2048*2048 (mem~128Mb). Simple scalar multiplication is performed on this matrix and timing analysis is done on the different cache (L1, L2 or L3) being accessed. We have repeated this experiment on all of our team members laptops (with different L1, L2, L3 cache sizes).

RESULTS

Parallel Matrix Multiplication

Matrix multiplication results based on GCC compiler:

1. Matrix Initialization Timings:

Matrix Dimension	Serial Initialization Timing (ms)	Parallel Initialization Timing (OpenMP #pragma parallel for)(ms)
3	**0.028000**	**4.054000**
400	**4.944**	**3.606**
600	**9.061**	**6.995**

2. Matrix Multiplication Timings:

SP: Simple Parallel

PT: Parallel Multiplication with 4 threads

PC: Parallel Multiplication with chunks, 3 for each thread

PN: Parallel Multiplication with Nested Loops

PS: Parallel Multiplication with SIMD Multiple Loops

All timings are in milliseconds.

Matrix Dimension (n x n)	Serial Multiplication Timing	Parallel Multiplication Timing in milliseconds				
		SP	PT	PC	PN	PS
3x3	0.000000	0.060000	0.029000	0.068000	0.084000	0.002000
400x400	552.74	269.72	1078.06	592.352	284.14	754.93
600x600	1796.28	980.13	3740.38	1905.21	953.52	2453.40

Matrix multiplication results based on XCODE compiler:

2. Matrix Initialization Timings:

All timings are in milliseconds.

Matrix Dimension	Serial Initialization Timing	Parallel Initialization Timing (OpenMP #pragma parallel for)(ms)
3x3	0.040676	4.970905
400x400	5.183	4.312
600x600	8.163554	5.240300

3. Matrix Multiplication Timings:

SP: Simple Parallel

PT: Parallel Multiplication with 4 threads

PC: Parallel Multiplication with chunks, 3 for each thread

PN: Parallel Multiplication with Nested Loops

PS: Parallel Multiplication with SIMD Multiple Loops

Matrix Dimension (n x n)	Serial Multiplication Timing	Parallel Multiplication Timing in milliseconds				
		SP	PT	PC	PN	PS
3x3	0.000491	0.090990	0.042960	0.017270	0.012837	0.002002
400x400	440.815	280.772	1033.602	515.954	1736.251	3742.258
600x600	1068.834432	321.412146	1362.600991	1052.573001	1918.711792	10146.961673

Cache Performance

Laptop 1:

L1 Cache Size: 32768 ~ 32Kb

L2 Cache Size: 262144 ~ 262 Kb

L3 Cache Size : 6291456 ~ 6.3 Mb

RAM: ~ 8 Gb

Matrix Dimension	Timing (ms)
32*32 ~ 32 Kb	0.005000
64*64~ 128 Kb	0.017000
128*128 ~ 512 Kb	0.078000
256*256 ~ 2 Mb	0.228000
512 * 512 ~ 8 Mb	0.915000

1024*1024 ~ 32 Mb	**3.324000**
2048*2048 ~128 Mb	**9.646000**

Laptop 2:

L1 Cache Size: 32768 ~ 32Kb

L2 Cache Size: 262144 ~ 262 Kb

L3 Cache Size : 3145728 ~ 3.1 Mb

RAM: ~ 8 Gb

Matrix Dimension	Timing (ms)
32*32 ~ 32 Kb	**0.006000**
64*64~ 128 Kb	**0.025000**
128*128 ~ 512 Kb	**0.105000**
256*256 ~ 2 Mb	**0.367000**
512 * 512 ~ 8 Mb	**1.165000**
1024*1024 ~ 32 Mb	**4.08000**
2048*2048 ~128 Mb	**14.783000**

Laptop 3:

L1 Cache Size: 32Kb

L2 Cache Size: 256Kb

L3 Cache Size : 3.072 Mb

RAM: ~ 12 Gb

Matrix Dimension	Timing (ms)
32*32 ~ 32 Kb	0.009612
64*64~ 128 Kb	0.054631
128*128 ~ 512 Kb	0.136106
256*256 ~ 2 Mb	0.52264512
512 * 512 ~ 8 Mb	2.056201
1024*1024 ~ 32 Mb	3.210601
2048*2048 ~128 Mb	12.530276

ANALYSIS OF RESULTS

Parallel Matrix Multiplication

We noticed following differences with the different OpenMP constructs:

1. Small size matrix have better time efficiency with serial computation over parallel computation. As with parallel execution, there is time loss with context switching.
2. For large size matrix, parallel execution is efficient. OpenMp '`#pragma omp parallel for`' construct initializes N threads at runtime based on number of CPU cores. Thus this construct gives the best efficiency.
3. When we tried to limit the number of threads to 4, which is almost half of the resources we have, we are not utilizing the system , to its full potential, hence we observed a dip in performance.(`#pragma omp parallel num_threads(4) )`
4. In '`#pragma omp for schedule(dynamic, 3)`' construct, each thread asks the

OpenMP runtime library for iteration number and then executes it and again communicates with the runtime library for the next number. This increased communication affects the execution time a bit, thus giving higher timings than simple parallel for construct.

5. The constructs '`#pragma omp parallel for collapse(1)`' and '`#pragma omp simd collapse(3)`' are most effective when the nested loops are independent from each other. However, in this case the all the loops in the nested loops are dependent on each other. So the time efficiency is affected a lot as time is wasted in thread creation, context switching and thread wait.

Cache Performance

To test the cache performance we used matrices of different sizes. Typical laptops have 32 Kb L1 cache, 256Kb L2 cache and around 3MB L3 cache size. So we tested with matrix sizes 32Kb, 128Kb, 512Kb, 2MB, 8MB, 32MB, 128MB.

Our basic idea is to use different sizes of matrix so that system uses different levels of cache and analyze the timings.

We observed few interesting cases:

I. Process execution is very fast when we used sizes less the L1 cache. And as the memory range falls in higher level the efficiency is getting reduced.

II. Also, when the size is larger the L3 cache the data is moved to RAM and this takes significant time. The time also depends on the system RAM utilization.

CONCLUSION

Parallel Matrix Multiplication

We noticed that for the smaller sizes of the matrices, using OpenMP #pragma s did not help much with the timing, for matrix multiplication, but as the size of the matrices increased to a significant number (say 3 to 200) OpenMP #pragma s proved to be more efficient. If the parallel computation is performed on small size matrix, there is a loss of efficiency due to context switching between threads. Thus sequential computation is more efficient for small size matrix. While large size matrix multiplication are greatly benefitted by parallel computation.

Cache Performance

Cache timing depends on L1, L2, L3 cache sizes of that particular laptop. We noticed that until the matrix size is within the L1 cache size, the time taken is very less to access the cache, as it is the 1st cache in the pipeline. As, the matrix size goes from one level to

another, the performance decreases, as we go close to physical disk, making it more and more expensive to write or retrieve data. So, we can say that, if the matrix dimension is really large, we need to provide ample resources with larger L1, L2, L3 caches, to exploit the parallelism from OpenMP.

REFERENCES

1. https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27
2. https://bisqwit.iki.fi/story/howto/openmp/
3. https://austingwalters.com/the-cache-and-multithreading/
4. https://austingwalters.com/cache-optimizing/