CS 301
High-Performance Computing

Lab 5 - C1

Block matrix multiplication

Aditya Nawal (202001402)
Divya Patel (202001420)

March 23, 2023

Contents

1 Introduction

This report investigates three different approaches to matrix multiplication. In Problem A-1, we explore the conventional matrix multiplication algorithm. In Problem B-1, we study the use of transpose matrix multiplication as a way to reduce the computational cost of matrix multiplication. Finally, in Problem C-1, we analyze the block matrix multiplication algorithm, which uses a divide and conquer strategy to compute the product of two large matrices.

Parallelization of the above algorithms can help reduce thier computational cost and improve its performance. In this report, we present a detailed analysis of the parallelization process and evaluate thier performance on matrices of different sizes. For each problem, we provide a detailed analysis of the algorithms, including their computational complexity.

2 Hardware Details

2.1 Lab 207 PC

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 4
- On-line CPU(s) list: 0-3
- Thread(s) per core: 1
- Core(s) per socket: 4
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- Stepping: 3
- CPU MHz: 3300.000
- CPU max MHz: 3700.0000
- CPU min MHz: 800.0000
- BogoMIPS: 6585.38

- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 6144K
- NUMA node0 CPU(s): 0-3
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb invpcid_single tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts

```
[student@localhost ~]$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 60  bytes 5868 (5.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 60  bytes 5868 (5.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

p4p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.100.64.86  netmask 255.255.255.0  broadcast 10.100.64.255
        inet6 fe80::b283:feff:fe97:d2f9  prefixlen 64  scopeid 0x20<link>
        ether b0:83:fe:97:d2:f9  txqueuelen 1000  (Ethernet)
        RX packets 32826  bytes 46075919 (43.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8015  bytes 586362 (572.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
        ether 52:54:00:3a:16:71  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 1: IP address of Lab PC

2.2 HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15

- Thread(s) per core: 1
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- Stepping: 2
- CPU MHz: 1976.914
- BogoMIPS: 5205.04
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 20480K
- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15

3 Problem C1

3.1 Description of the problem

Given two square matrices A and B of size $n \times n$, we want to compute their product $C = AB$. We can use a block matrix multiplication algorithm that partitions both matrices into smaller submatrices of size $block\_size \times block\_size$ and applies certain rules to obtain each submatrix in C.

Let us denote the submatrices of A, B and C as follows:

$$A = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right], B = \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right], C = \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right].$$

where each submatrix has size $block\_size \times block\_size$. Then we can apply the following rule to obtain each submatrix in C:

$$C = AB = \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right].$$

The block matrix multiplication algorithm computes each submatrix product and sum by iterating over all possible combinations of blocks in A, B and C. The algorithm assumes that n is divisible by $block\_size$. Otherwise, some elements may be left out or accessed out of bounds.

3.2 Serial Complexity

This method requires same number of operations as the standard matrix multiplication algorithm, i.e., $O(n^3)$.

3.3 Parallel Complexity

$$T_p = \mathcal{O}\left(\frac{n^3}{p}\right)$$

where T_p is the parallel time complexity, N is the size of the matrices being multiplied and p is the number of processors.

This expression assumes that the workload is evenly distributed among all processors and that there are no overheads due to parallelization. In practice, however, there may be some overheads due to thread creation and synchronization which can affect the actual parallel time complexity.

3.4 Profiling Information

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.32      1.55     1.55        1     1.55     1.55  main
  0.00      1.55     0.00        2     0.00     0.00  diff
  0.00      1.55     0.00        1     0.00     0.00  block_matmul

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012-2014 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

Figure 2: Screenshot of text file generated from profiling on Lab 207 PC using gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.51      0.75     0.75        1     0.75     0.75  main
  0.00      0.75     0.00        2     0.00     0.00  diff
  0.00      0.75     0.00        1     0.00     0.00  block_matmul

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.
`
calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012-2014 Free Software Foundation, Inc.
```

Figure 3: Screenshot of text file generated from profiling on HPC Cluster using gprof

3.5 Optimization Strategy

We chose 32 as the block size for block matrix multiplication because it is close to the upper bound given by the formula:

$$block - size \leq \sqrt{\frac{L1_{cache-size}}{3 * word - size}} \tag{1}$$

where $L1_{cache-size}$ is the size of my L1 cache in bytes and $word-size$ is the size of my data type in bytes. The formula is based on the assumption that my L1 cache can store 3 blocks (block-size * block-size) of data at a time, one for each matrix involved in the multiplication. This way, we can minimize the number of cache misses and improve the computational efficiency of the operation.

Since we have a 32 KB L1 cache and a double data type (which has 8 bytes), we can calculate:

$$block - size \leq \sqrt{\frac{32 * 1024}{3 * 8}} \approx 36 \tag{2}$$

This means that the optimal block size should be less than or equal to 36 for optimal performance. We chose 32 because it is a power of two and it can fit well into the cache memory without wasting any space. Powers of two are often preferred for block sizes. We also tested different block sizes ranging from 4 to 64 and measured their execution time. We found that 32 gave us the fastest execution time among all the tested values in HPC Cluster and 16 gave the fastest results in Lab PC.

3.6 Graph of Problem Size vs Algorithm Runtime

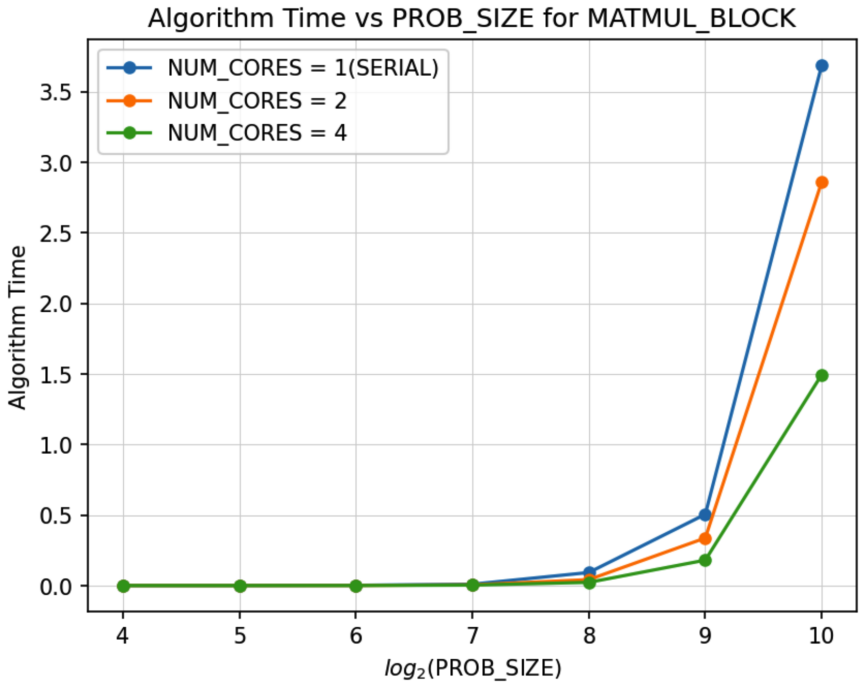

Figure 4: Graph of Problem Size vs Algorithm Runtime for Lab PC

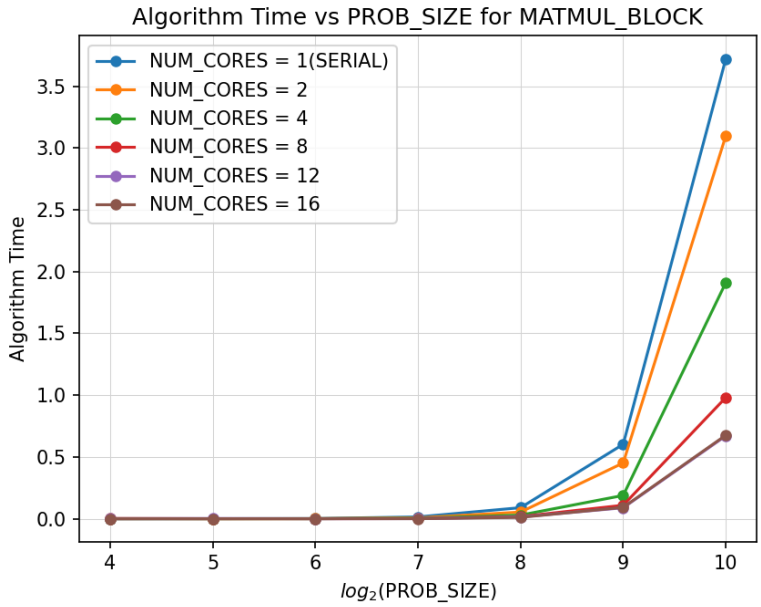

Figure 5: Graph of Problem Size vs Algorithm Runtime for HPC cluster

3.7 Graph of Problem Size vs End-to-End Runtime

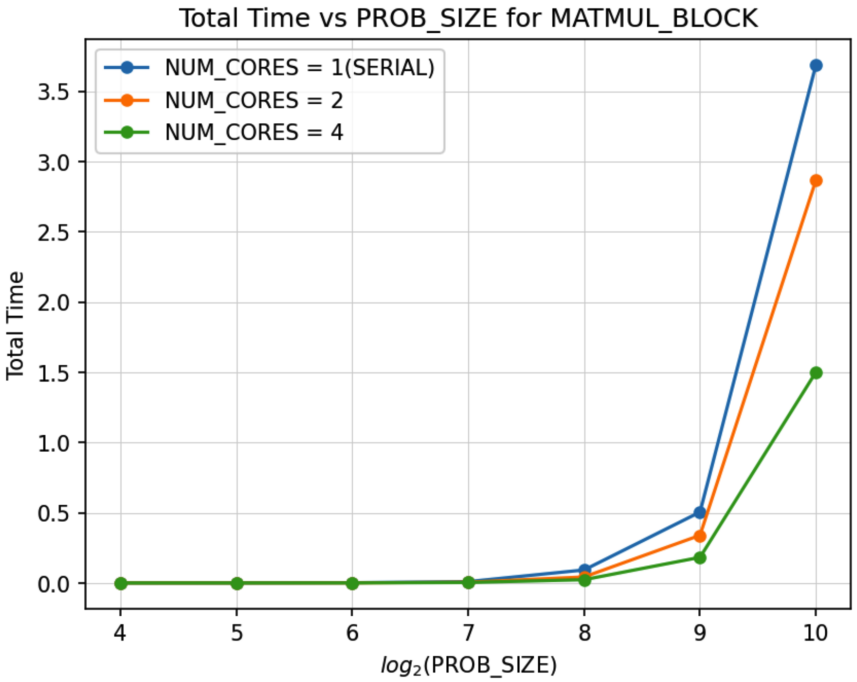

Figure 6: Graph of Problem Size vs End-to-End Runtime for Lab PC

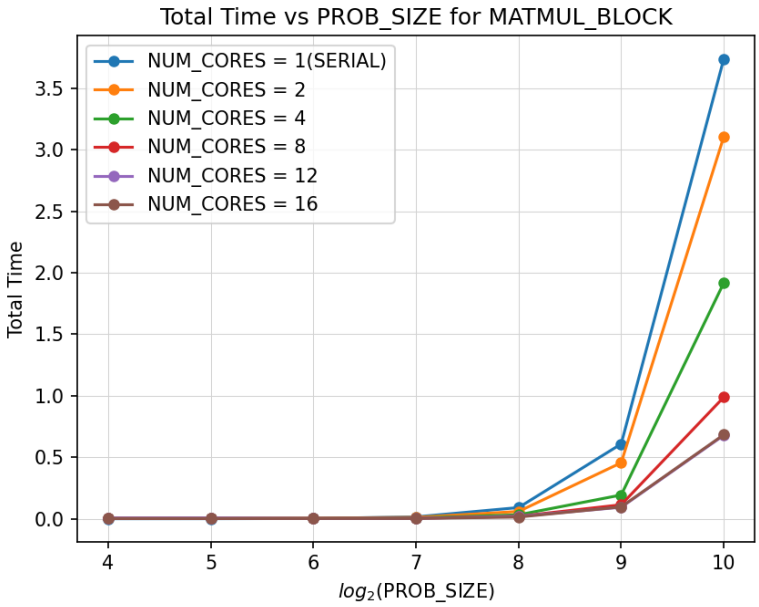

Figure 7: Graph of Problem Size vs End-to-End Runtime for HPC cluster

3.8 Discussion

Conventional matrix multiplication does not use any partitioning and multiplies each element of one matrix with each element of another matrix. It has a time complexity of $O(n^3)$ for multiplying two $n \times n$ matrices.

Block matrix multiplication uses a partitioning scheme that divides the matrices into smaller submatrices or blocks, and then multiplies them using standard matrix multiplication. It has a time complexity of $O(n^3)$ for multiplying two $n \times n$ matrices, but it can be faster than conventional matrix multiplication due to better cache utilization.

The advantage of block matrix multiplication is that it reduces the number of cache misses by reusing the data in each block as much as possible. The optimal block size depends on the size of the matrices and the cache parameters. The disadvantage of block matrix multiplication is that it requires additional memory allocation and copying for storing and accessing the blocks.

In conclusion, block matrix multiplication and conventional matrix multiplication are two equivalent methods for computing the product of two matrices, but they have different trade-offs in terms of performance and memory usage. Block matrix multiplication can be faster than conventional matrix multiplication by exploiting spatial and temporal locality of data, but it requires more memory management and careful partitioning.

4 Alternative Approach(Experiment)

Consider two matrices A and B with dimensions $m \times n$ and $n \times p$, respectively. The product of these matrices is a matrix C with dimensions $m \times p$. To compute the element c_{ij} of matrix C, we need to compute the dot product of the i^{th} row of matrix A and the j^{th} column of matrix B.

If we store matrix A in a row-major 1-D array and matrix B in a column-major 1-D array, then accessing the elements required for computing the dot product becomes more efficient. This is because consecutive elements in a row or column are stored in consecutive memory locations, improving cache locality.

4.1 Illustration of this method

Consider two matrices A and B with dimensions 2×3 and 3×2, respectively:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

We can convert matrix A to a row-major 1-D array and matrix B to a column-major 1-D array as follows:

$$A_{row\_major} = [1, 2, 3, 4, 5, 6] \quad B_{col\_major} = [7, 9, 11, 8, 10, 12]$$

Now let's compute the element $c_{1,1}$ of the resulting matrix C. This element is obtained by computing the dot product of the first row of matrix A and the first column of matrix B. In our row-major and column-major representations this corresponds to:

$$\begin{aligned} c_{1,1} &= A_{row\_major}[0] \cdot B_{col\_major}[0] + A_{row\_major}[1] \cdot B_{col\_major}[1] + A_{row\_major}[2] \cdot B_{col\_major}[2] \\ &= (1 \cdot 7) + (2 \cdot 9) + (3 \cdot 11) \\ &= 58 \end{aligned}$$

4.1.1 Graph of Problem Size vs Algorithm Runtime

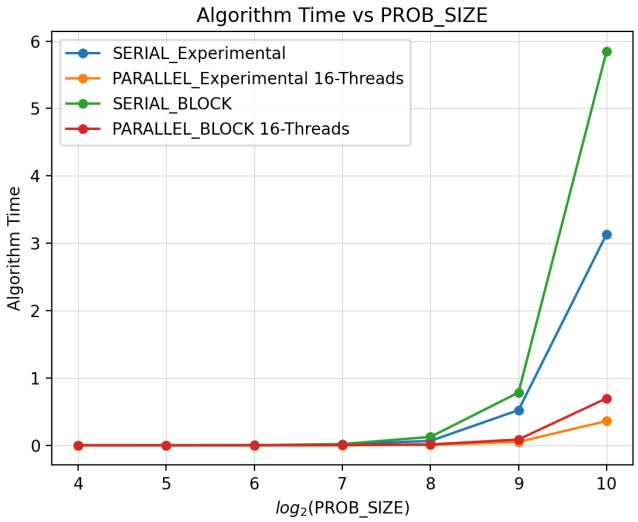

Figure 8: Graph of Problem Size vs Algorithm Runtime for HPC Cluster

From the graph we can see that the performance of this method is even better than the block matrix multiplication method for this problem size range.