CS 301
High-Performance Computing

Lab 5 - A1

Conventional 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.

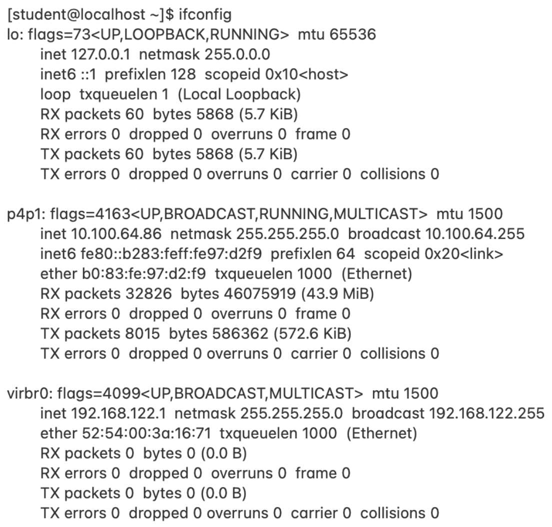
Parallelization of the above algorithms can help reduce thier computational cost and improve its performance. In this report, we present a detailed analysis of the parallelization process and evaluate thier performance on matrices of different sizes. For each problem, we provide a detailed analysis of 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 A1

3.1 Description of the problem

Conventional matrix multiplication is a mathematical operation that takes two matrices **A** and **B** as inputs and produces a third matrix **C** as output. The output matrix is obtained by multiplying each element of a row of matrix **A** by each element of a column of matrix **B** and adding them together. The output matrix **C** has the same number of rows as matrix **A** and the same number of columns as matrix **B**. Conventional matrix multiplication is also known as standard or naive matrix multiplication because it follows the definition of matrix multiplication without any optimization or algorithmic improvement.

To optimize matrix multiplication, we can utilize parallel processing techniques to perform the matrix multiplication in parallel. Parallel matrix multiplication can be implemented using various parallelization techniques, such as data parallelism, task parallelism, or pipeline parallelism. The goal is to split the matrix multiplication task into multiple sub-tasks and perform them concurrently to reduce the overall computation time. By using parallel processing, we can perform matrix multiplication faster and more efficiently, which is particularly important for large matrices.

3.2 Serial Complexity

Conventional matrix multiplication has a time complexity of $\mathcal{O}(n^3)$ where n is the size of the matrices, which means that it becomes very slow and inefficient for large matrices.

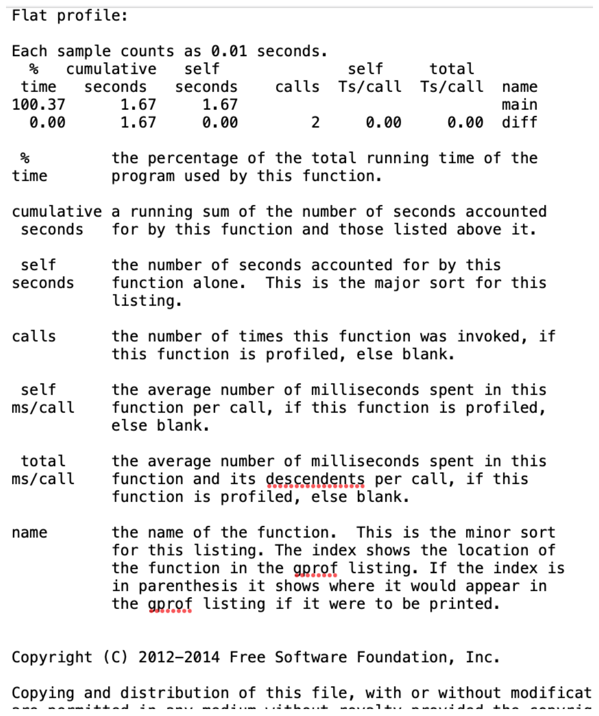
3.3 Parallel Complexity

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

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

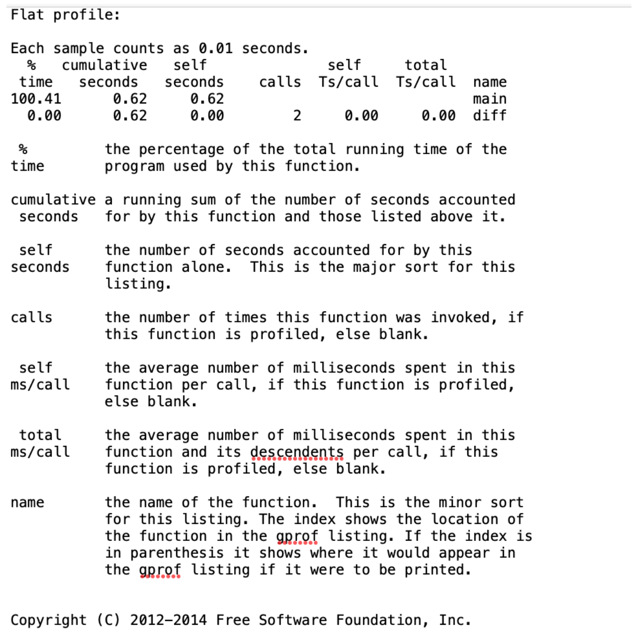
This expression assumes that the workload is evenly distributed among all processors and that there are no overheads due to parallelization. In practice, however, there may be some overheads due to thread creation and synchronization which can affect the actual parallel time complexity.

3.4 Profiling Information



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.37      1.67     1.67                             main
  0.00      1.67     0.00        2     0.00     0.00  diff

 %         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 modificat
```

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  Ts/call  Ts/call  name
100.41      0.62     0.62                             main
  0.00      0.62     0.00        2     0.00     0.00  diff

 %         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

Loop Order IKJ is used.
The loop parallelization is achieved using OpenMP directive “#pragma omp parallel for”, which allows the parallel execution of the outermost for-loop across multiple threads. The loop collapsing is done using the “collapse(2)” clause, which combines the two nested loops into a single loop that is then parallelized across threads.

The data sharing is done using the “shared” clause, which makes the matrices A, B and C shared across threads, so that each thread can access and update them. However, to avoid data races, the loop variables i, j and k are made private using the “private” clause, which ensures that each thread has its own copy of these variables.

Finally, to ensure the correctness of the calculation, the "reduction(+:sum)" clause is used for the innermost loop, which sums the product of A and B matrices and stores the result in the variable "sum". The reduction clause ensures that the final result is the sum of the individual results obtained by each thread, thus avoiding race conditions and ensuring the correctness of the calculation.

3.6 Graph of Problem Size vs Algorithm Runtime

3.6.1 Graph of Problem Size vs Algorithm Runtime for LAB207 PCs

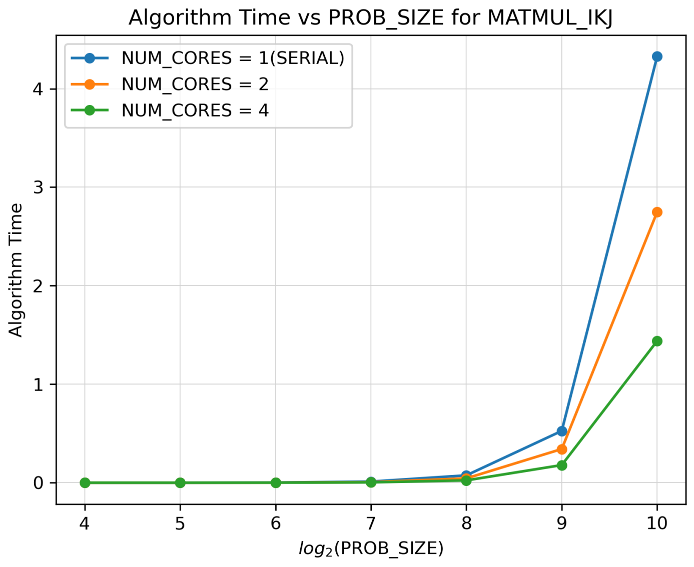


Figure 4: Graph of Problem Size vs Algorithm Runtime for Lab PC

3.6.2 Graph of Problem Size vs Algorithm Runtime for HPC Cluster

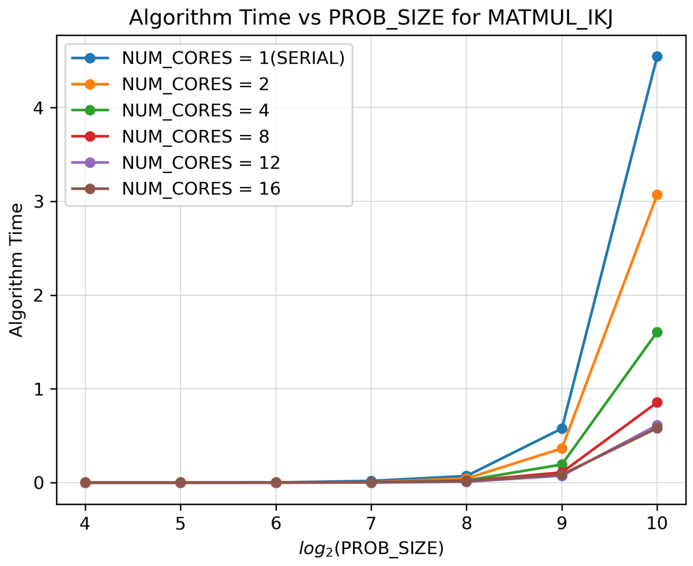


Figure 5: Graph of Problem Size vs Algorithm Runtime for HPC cluster

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

3.7.1 Graph of Problem Size vs End to End Runtime for LAB207 PCs

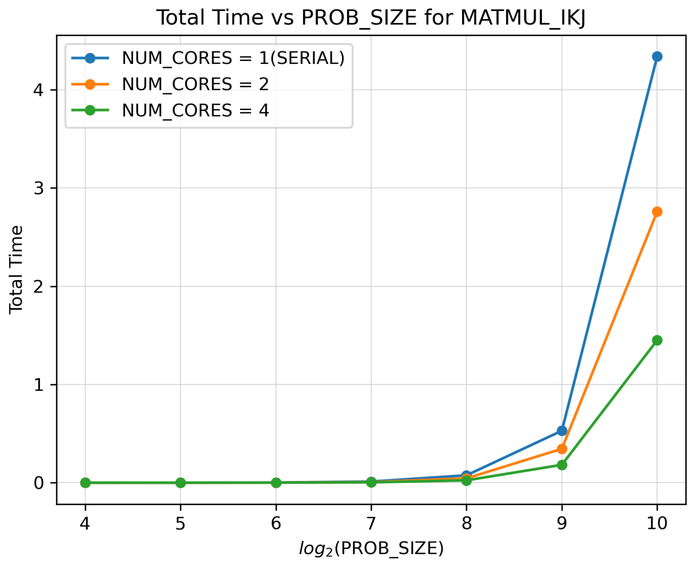


Figure 6: Graph of Problem Size vs End-to-End Runtime for Lab PC

3.7.2 Graph of Problem Size vs End to End Runtime for HPC Cluster

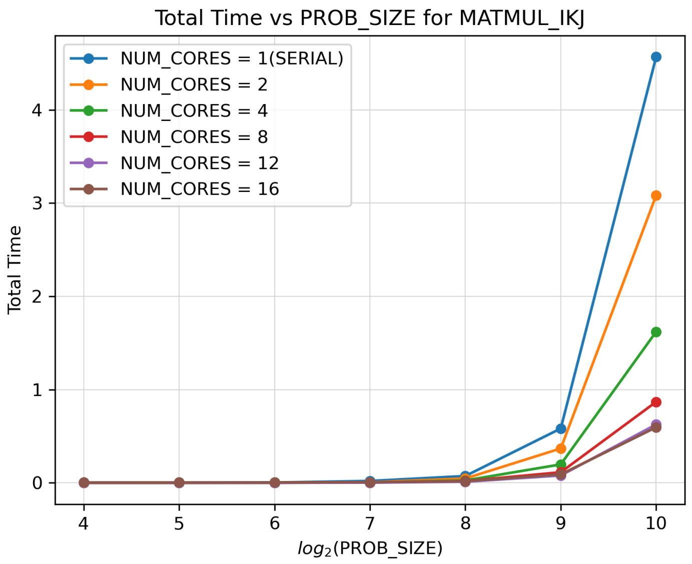


Figure 7: Graph of Problem Size vs End-to-End Runtime for HPC cluster

3.8 Discussion

The code in innermost loop is simply *C[i][j] += A[i][k] * B[k][j];* The order of loops in matrix multiplication affects spatial locality because it determines how efficiently the elements of the matrices are accessed in cache. The matrices are stored in row-major order, which means that consecutive elements in a row are stored next to each other in cache. This means that accessing elements in a row is faster than accessing elements in a column, because it reduces cache misses and increases cache hits.

Loop orders KIJ and IKJ have lower runtime because they access matrix A by rows and matrix B by columns. This minimizes cache misses and maximizes spatial locality. This loop order accesses matrix A by rows and matrix B by columns. It has good spatial locality for both matrices, and also for matrix C, which is updated by elements. It has high performance.

On parallelizing this code upto two levels by using collapse and parallel for, we obbserve that runtime decreases significantly. This is because the code is now being executed in parallel by multiple threads. The number of threads is equal to the number of processors in this case. The number of processors is 4 for Lab PC and 16 for HPC cluster. This means that the code is being executed in parallel by 4 and 16 threads respectively. This results in a significant decrease in runtime.