CS 301
High-Performance Computing

Lab 6

QR Decomposition of a Matrix

Aditya Nawal (202001402)
Divya Patel (202001420)

April 10, 2023

Contents

1 Introduction **3**

2 Hardware Details **3**
2.1 Lab 207 PC 3
2.2 HPC Cluster 4

3 Problem A1 **6**
3.1 Description of the problem 6
3.2 Serial Complexity 6
3.3 Parallel Complexity 6
3.4 Profiling Information 7
3.5 Optimization Strategy 8
3.6 Optimizing Gram-Schmidt Process 8
3.7 Optimizing Transposing of Matrix Process 9

4 Runtime and Speedup on LAB207 PCs **10**
4.0.1 Graph of Problem Size vs Algorithm Runtime for HPC Cluster 10
4.0.2 Graph of Problem Size vs End-to-End Runtime for LAB207 PCs 10
4.0.3 Speedup on LAB207 PCs 11
4.0.4 Efficiency on LAB207 PCs 11

5 Runtime and Speedup on HPC Cluster **12**
5.0.1 Graph of Problem Size vs Algorithm Runtime for HPC Cluster 12
5.0.2 Graph of Problem Size vs End-to-End Runtime for HPC Cluster 12
5.0.3 Speedup on HPC Cluster 13
5.0.4 Efficiency on HPC Cluster 13

6 Discussion **14**

1 Introduction

The problem at hand is the QR decomposition of a matrix, which involves decomposing a matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R. This is a classical linear algebra problem that has various applications such as solving systems of linear equations, calculating eigenvalues, and image processing.

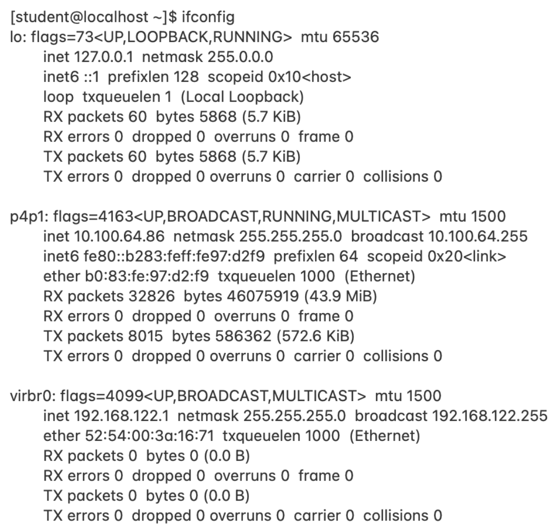
In this report, the Gram-Schmidt process is used to compute the decomposition, and the focus is on implementing efficient shared memory parallelization using OpenMP and optimizing the code to improve performance. The report also includes a brief description of the problem, its computational and memory complexity, and possible problem decomposition strategies.

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

The problem at hand is the implementation of the QR decomposition algorithm using the Gram-Schmidt process. Given an $n \times n$ matrix A, the goal is to find an orthogonal matrix Q and an upper triangular matrix R such that $A = QR$.

The Gram-Schmidt process involves the following steps:

1. Normalize the first column of A to obtain the first column of Q: $a_1 = \frac{A[:,1]}{||A[:,1]||}$, where $||.||$ denotes the Euclidean norm.

2. For $i = 2$ to n, compute the i-th column of Q and the i-th row of R:

$$a_i = A[:, i] - \sum_{j=1}^{i-1} (q_j^T A[:, i]) q_j$$

$$r_{i,j} = q_j^T A[:, i], \text{ for } j = 1 \text{ to } i - 1$$

$$r_{i,i} = ||a_i||$$

3. Normalize the i-th column of Q: $q_i = \frac{a_i}{||a_i||}$.

The above steps outline the process of obtaining the QR decomposition of a matrix using the Gram-Schmidt process. The algorithm takes as input a matrix A and outputs two matrices, Q and R, such that their product equals the input matrix.

3.2 Serial Complexity

The serial time complexity of the QR decomposition algorithm using the Gram-Schmidt process can be analyzed as follows. For an $n \times n$ matrix A, the first step of normalizing the first column of A takes $O(n)$ time. The second step involves a loop that runs from $i = 2$ to n. For each iteration of the loop, we need to compute the dot product of two vectors of length n, which takes $O(n)$ time. Since there are $n-1$ iterations of the loop, this step takes $O(n^2)$ time. The third step of normalizing the i-th column of Q takes $O(n)$ time. Therefore, the overall time complexity of the algorithm is $O(n) + O(n^2) + O(n) = O(n^2)$.

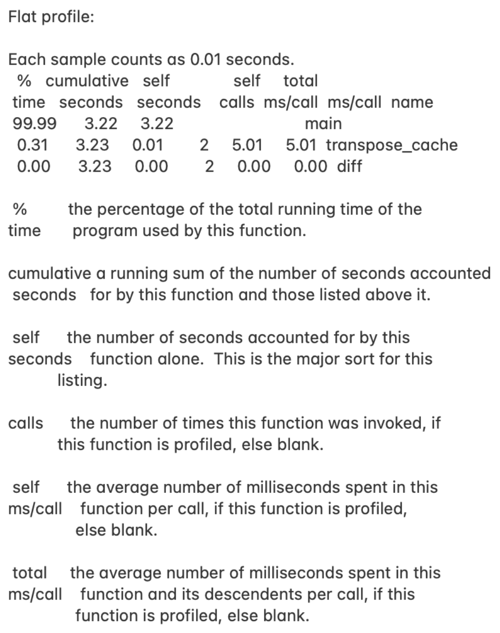
3.3 Parallel Complexity

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

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

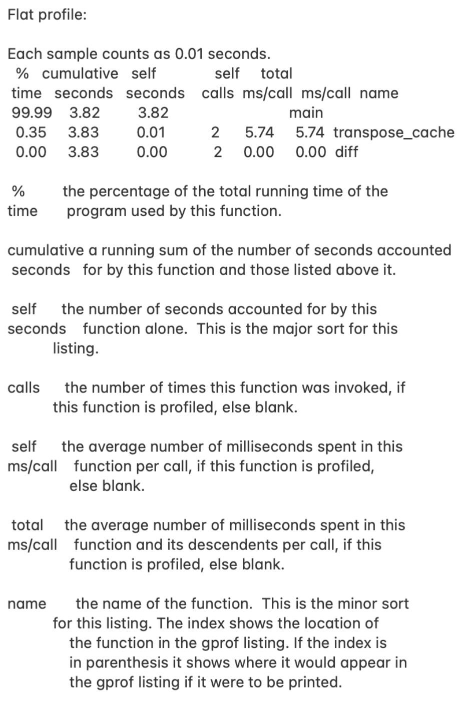
This expression assumes that the workload is evenly distributed among all processors and that there are no overheads due to parallelization. In practice, however, there may be some overheads due to thread creation and synchronization which can affect the actual parallel time complexity.

3.4 Profiling Information



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.99      3.22     3.22                             main
  0.31      3.23     0.01        2     5.01     5.01  transpose_cache
  0.00      3.23     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.
```

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  ms/call  ms/call  name
 99.99      3.82     3.82                             main
  0.35      3.83     0.01        2     5.74     5.74  transpose_cache
  0.00      3.83     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.
```

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

3.5 Optimization Strategy

3.6 Optimizing Gram-Schmidt Process

```
$A \leftarrow A^T$;
for $i \leftarrow 1$ to $n$ do
    $s \leftarrow 0$;
    for $j \leftarrow 1$ to $n$ do
        $s \leftarrow s + a_{ij}^2$;
    end
    $r_{ii} \leftarrow \sqrt{s}$;
    for $j \leftarrow 1$ to $n$ do
        $q_{ij} \leftarrow a_{ij}/r_{ii}$;
    end
    for $j \leftarrow i+1$ to $n$ do
        $s \leftarrow 0$;
        for $k \leftarrow 1$ to $n$ do
            $s \leftarrow s + a_{kj} * q_{ik}$;
        end
        $r_{ij} \leftarrow s$;
        for $k \leftarrow 1$ to $n$ do
            $a_{kj} \leftarrow a_{kj} - r_{ij} * q_{ik}$;
        end
    end
end
$Q \leftarrow Q^T$;
```

Algorithm 1: Modified Gram-Schmidt process

The modified Gram-Schmidt process that takes the transpose of the matrix at the beginning and end can improve cache locality compared to the classical Gram-Schmidt process. Cache locality refers to the property of memory access patterns where data that is accessed together is stored close together in memory. This can improve the performance of algorithms by reducing the number of cache misses.

In the case of the modified Gram-Schmidt process, taking the transpose of the matrix at the beginning changes the order in which the elements of the matrix are accessed. For example, if the matrix is stored in row-major order in a programming language, accessing its elements in row-major order (i.e., accessing all elements of a row before moving on to the next row) can improve cache locality. Taking the transpose of the matrix changes the access pattern from column-major to row-major, which can improve cache locality if the matrix is stored in row-major order.

Our program uses several optimization strategies to improve its performance. These strategies include:

- **SIMD vectorization:** The algorithm uses the `#pragma omp simd` directive to enable SIMD vectorization of certain loops. SIMD (Single Instruction Multiple Data) vectorization is a technique that allows the processor to perform the same operation on multiple data elements simultaneously. This can improve the performance of the algorithm by taking advantage of the parallel processing capabilities of modern CPUs.

- **Reduction:** The algorithm uses the `reduction` clause of the `#pragma omp simd` and `#pragma omp parallel for` directives to perform a reduction operation on the `sum` variable. A reduction operation combines the values of a variable across multiple threads into a single value. This can improve the performance of the algorithm by reducing the amount of synchronization required between threads.

- **Parallelization:** The algorithm uses the `#pragma omp parallel for` directive to parallelize certain loops. This allows the iterations of the loop to be executed concurrently by multiple threads. This can improve the performance of the algorithm by taking advantage of the multiple cores available on modern CPUs.

3.7 Optimizing Transposing of Matrix Process

Here we transpose matrix using blocks to

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{\top} = \begin{bmatrix} A^{\top} & C^{\top} \\ B^{\top} & D^{\top} \end{bmatrix}$$

Figure 4: Transposse of Matrix using Blocks

```
Input: src, dst, n
Output: Transposed matrix
const int $blocksize \leftarrow 32$;
for $i \leftarrow 0$ to $n$ $blocksize$ do
    for $j \leftarrow 0$ to $n$ $blocksize$ do
        for $k \leftarrow i$ to $\min(i + blocksize, n)$ do
            for $l \leftarrow j$ to $\min(j + blocksize, n)$ do
                $dst[k][l] \leftarrow src[l][k]$;
            end
        end
    end
end
```

Algorithm 2: Cache-optimized matrix transposition code

This method for transposing a matrix can improve performance by taking advantage of cache locality. It does this by dividing the matrix into blocks and transposing each block separately. This can improve cache locality because it ensures that the elements within each block are accessed together, which can reduce the number of cache misses.

The size of the blocks is chosen to match the size of the cache lines. In this case, a block size of 32 is used, which means that each block contains 32x32 elements.

According to Exercise 2.6.16 on page 116 of [1]

References

[1] Strang, Gilbert. *Introduction to Linear Algebra.* 4th ed. Wellesley-Cambridge Press, 2009.

4 Runtime and Speedup on LAB207 PCs

4.0.1 Graph of Problem Size vs Algorithm Runtime for HPC Cluster

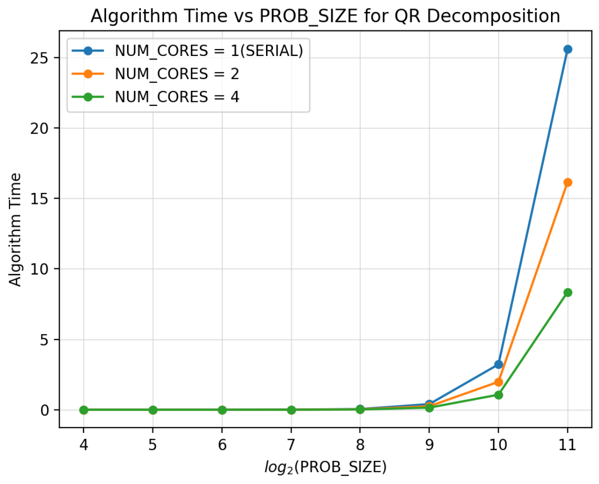


Figure 5: Graph of Problem Size vs Algorithm Runtime for Lab PC

4.0.2 Graph of Problem Size vs End-to-End Runtime for LAB207 PCs

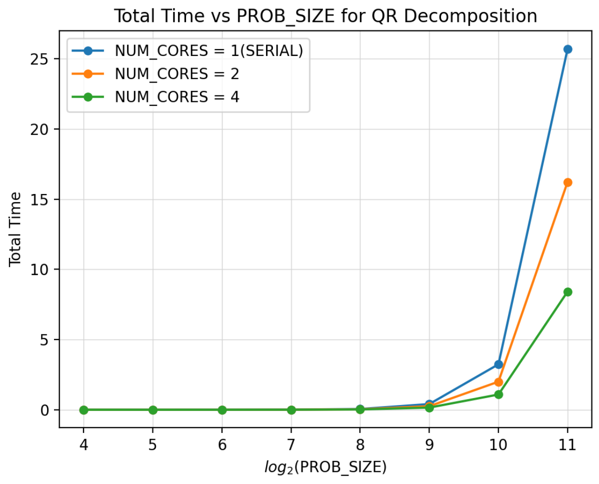


Figure 6: Graph of Problem Size vs End-to-End Runtime for Lab PC

4.0.3 Speedup on LAB207 PCs

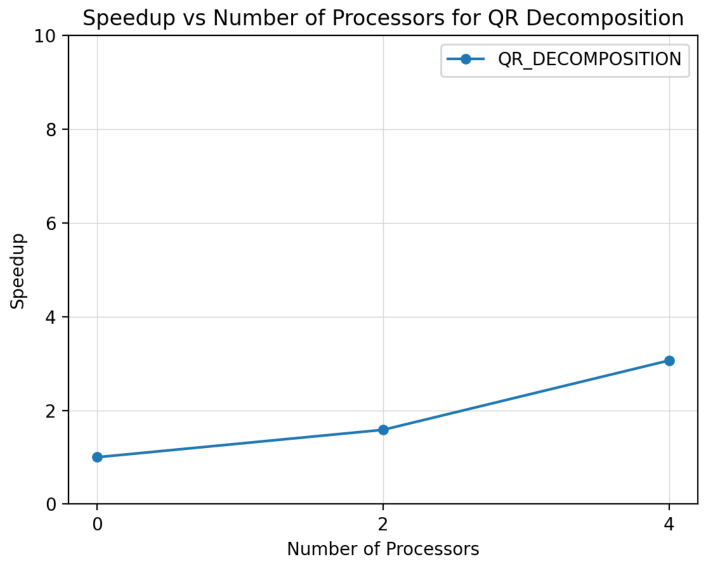


Figure 7: Graph of Speedup for Lab PC

4.0.4 Efficiency on LAB207 PCs

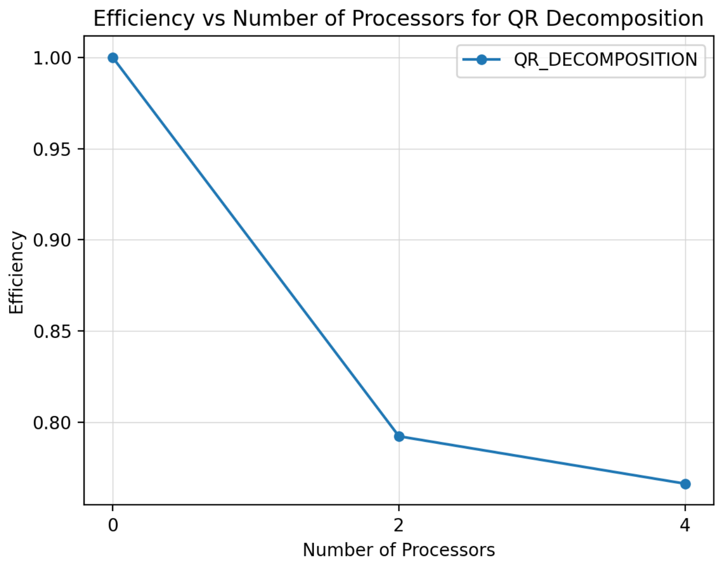


Figure 8: Graph of Problem Size vs Efficiency for Lab PC

5 Runtime and Speedup on HPC Cluster

5.0.1 Graph of Problem Size vs Algorithm Runtime for HPC Cluster

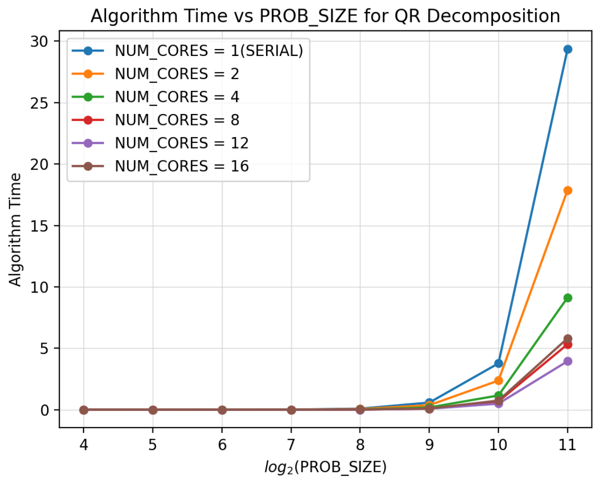


Figure 9: Graph of Problem Size vs Algorithm Runtime for HPC cluster

5.0.2 Graph of Problem Size vs End-to-End Runtime for HPC Cluster

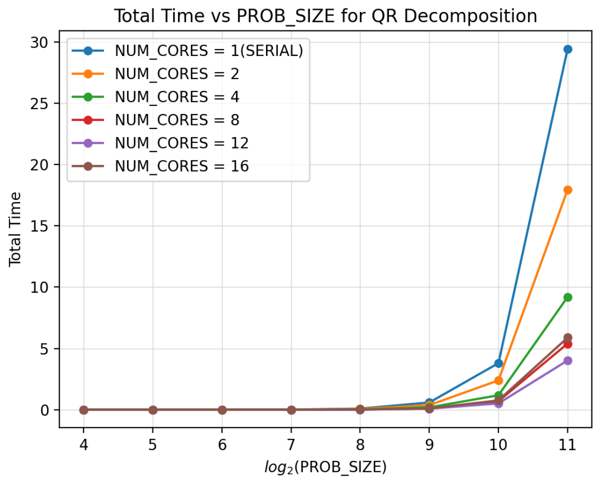


Figure 10: Graph of Problem Size vs End-to-End Runtime for HPC cluster

5.0.3 Speedup on HPC Cluster

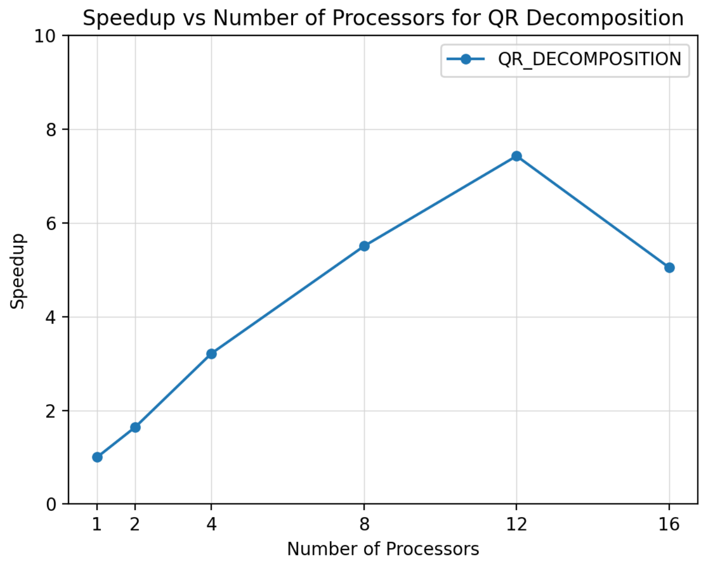


Figure 11: Processors vs Speedup for HPC cluster

5.0.4 Efficiency on HPC Cluster

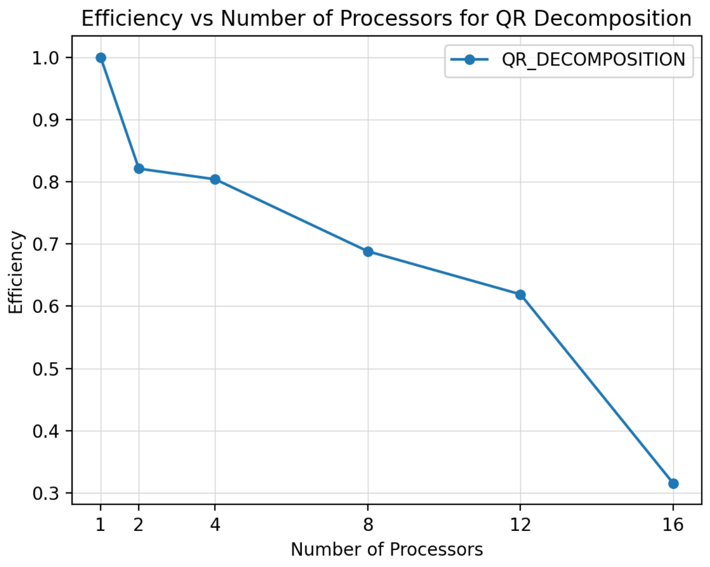


Figure 12: Number of Threads vs Efficiency for HPC cluster

6 Discussion

From our experimental results, we can observe that the speedup of the algorithm decreases when the number of threads is increased from 12 to 16. This behavior suggests that the algorithm is not able to fully utilize the additional computational resources provided by the extra threads.

There could be several factors contributing to this decrease in speedup. One possibility is that the overhead of managing the threads becomes too large as the number of threads increases. This overhead can include the time required to create and destroy threads, as well as the time required to schedule threads on CPU cores. As the number of threads increases, this overhead can reduce the performance gains achieved by parallelization.

Another possibility is that the algorithm is limited by the memory bandwidth of the hardware on which it is being run. As more threads are used, more data must be transferred between the CPU and memory. If the memory bandwidth is not sufficient to support this increased data transfer rate, then contention for memory bandwidth can reduce performance.

Further analysis and profiling would be necessary to determine the exact cause of this decrease in speedup. However, our results suggest that there may be a limit to the scalability of this algorithm with respect to the number of threads.

Amdahl's law is a formula used to calculate the theoretical maximum speedup of a program when parallelized. The formula is given by:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

where S is the speedup, P is the fraction of the program that can be parallelized, and N is the number of processors.

In this case, we are given that the maximum **speedup at 12 threads is 7.43**. We can use this information and Amdahl's law to calculate the fraction of the program that can be parallelized. Substituting the given values into Amdahl's law, we get:

$$7.43 = \frac{1}{(1-P) + \frac{P}{12}} \tag{2}$$

Solving for P, we find that $P \approx 0.944$. This means that approximately 94.4% of the program can be parallelized.

The fraction of the program that is serial (i.e., cannot be parallelized) is given by $1 - P$. In this case, the serial fraction is approximately $1 - 0.944 = 0.056$. This means that approximately 5.6% of the program is serial.