High Performance Programming Workshop

Adrià García Corbera
Arnau Encinas Diaz
Johan Guldberg Theil
Marcus Vinicius Hodal Xavier de Oliveira
Mikkel Norre Nielsen
Victor Amaro Vazquez
Christian Lykke Jørgensen

May 2025

1 Breadth First Search

The chosen language for this workshop part is C# .NET, this is selected for its ease of development and debugging. Furthermore, multiple group members are experienced in C#. The code for this part can be found on GitHub here click me.

In addition, extensive automated testing is used during implementation. The testing during implementation is done on multiple levels: unit, integration, and end-to-end. All benchmarks are done using BenchmarkDotNet, with a warm-up first, then 100 runs of the implementation. The graph structure chosen is adjacency lists, as it lowers the needed memory, thus allowing tests of larger graphs. The benchmarks **exclude** the time it takes to load the graph into memory before searching, but **do account** for the startup/configuration time for each implementation. All benchmarks were run on the same machine in one go with the following specifications:

- BenchmarkDotNet v0.14.0, Windows 11 (10.0.26100.3775)
- 16GB RAM, 12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
- .NET SDK 9.0.203 (9.0.425.16305), X64 RyuJIT AVX2

Figure 1 shows the code benchmarking all BFS implementations. The graph loads first, then benchmarks each implementation with the given parameters one at a time. The tooling outputs tables with data, which is used in individual sections.

```csharp
public class Program
    public static void Main(string[] args)
    {
        var summary_g1 = BenchmarkRunner.Run<BFSBenchmarks_G1>();
        var summary_g2 = BenchmarkRunner.Run<BFSBenchmarks_G2>();
    }
}

[MemoryDiagnoser(false)]
[MarkdownExporter, HtmlExporter, CsvExporter, RPlotExporter]
public class BFSBenchmarks_G1
{
    public static List<uint>[] loadedGraph;

    [GlobalSetup]
    public void Setup()
    {
        string FileName = "g1_adjacency_list.bin";
        Console.WriteLine($"Loading graph {FileName} from file...");
        loadedGraph = GraphService.LoadGraph(FileName);
    }

    [Benchmark(Baseline = true)]
    public void Sequential() => GraphSearchers.BFS_Sequential(loadedGraph, 0);

    [Benchmark]
    [Arguments(2)]
    [Arguments(4)]
    [Arguments(8)]
    [Arguments(12)]
    [Arguments(16)]
    public void BFS_ParallelSharedMemory(int maxThreads) => GraphSearchers.BFS_ParallelSharedMemory(loadedGraph, 0, maxThreads);

    [Benchmark]
    [Arguments(2)]
    [Arguments(4)]
    [Arguments(8)]
    [Arguments(12)]
    [Arguments(16)]
    public void Distributed(int numWorkers) => GraphSearchers.BFS_Distributed(loadedGraph, 0, numWorkers);
}

[MemoryDiagnoser(false)]
[MarkdownExporter, HtmlExporter, CsvExporter, RPlotExporter]
public class BFSBenchmarks_G2
{
    public static List<uint>[] loadedGraph;

    [GlobalSetup]
    public void Setup()
    {
        string FileName = "g2_adjacency_list.bin";
        Console.WriteLine($"Loading graph {FileName} from file...");
        loadedGraph = GraphService.LoadGraph(FileName);
    }

    [Benchmark(Baseline = true)]
    public void Sequential() => GraphSearchers.BFS_Sequential(loadedGraph, 0);

    [Benchmark]
    [Arguments(2)]
    [Arguments(4)]
    [Arguments(8)]
    [Arguments(12)]
    [Arguments(16)]
    public void BFS_ParallelSharedMemory(int maxThreads) => GraphSearchers.BFS_ParallelSharedMemory(loadedGraph, 0, maxThreads);

    [Benchmark]
    [Arguments(2)]
    [Arguments(4)]
    [Arguments(8)]
    [Arguments(12)]
    [Arguments(16)]
    public void Distributed(int numWorkers) => GraphSearchers.BFS_Distributed(loadedGraph, 0, numWorkers);
}
```

Figure 1: Benchmark code for all implementations across two graphs.

1A Sequential

1A.i Explain $\Theta(V + E)$

The BFS algorithm explores all vertices and edges in the graph. When a node has been explored, it is stored as explored and will not reenter the queue. Therefore, every edge/node will only be explored once, and the work done will therefore be $\Theta(V + E)$. The BFS algorithm can only be parallelized to as many edges as there are in the queue, as these are the only known nodes. The algorithm can only parallelize as many edges as there are cores. In a parallel context, the "depth" of BFS refers to the number of BFS levels, which is the number of synchronous steps needed to visit all reachable nodes from the source. This depth is typically equal to the graph's diameter or an upper bound on it. Since each BFS level can be processed in parallel, the depth dictates the minimum number of parallel steps required, making it a key factor in determining the parallel runtime of BFS. A smaller depth enables better parallel efficiency, while a larger depth limits speedup due to increased synchronization rounds.

1A.ii Implement BFS

Below is the implemented code for a single-processor BFS.

```
32 public static Bitmap BFS_Sequential(List<uint>[] graph, uint startNode)
33   {
34       var visited = new Bitmap(graph.Length);
35       var queue = new Queue<uint>();
36       visited.Set(startNode);
37       queue.Enqueue(startNode);
38
39       while (queue.Count > 0)
40       {
41           long node = queue.Dequeue();
42           foreach (var neighbor in graph[node])
43           {
44               if (visited.SetIfNot(neighbor))
45                   queue.Enqueue(neighbor);
46           }
47       }
48
49       return visited;
50   }
```

Figure 2: Code snippet for the single processor BFS.

Bitmap

The bitmap class is bit-wise tracking of visited vertices. Figure 3 shows the implementation of the primary functions. The internal tracking is done by initialising an ulong[size] array where $size = (count(V) + 63)/64$, this ensures total bits available is $\geq count(V)$. The reasoning behind using long (64-bit) and not unit (32-bit) is to utilize the 64-bit instruction set fully. The operation of setting and getting a visited vertex is done by first, locating the uint storing the vertex and create a mask matching the desired vertex, subsequent $|=$ or $\&=$ the two for either setting or determine if a vertex is visited.

```
23  [MethodImpl(MethodImplOptions.AggressiveInlining)]
24  public void Set(uint index)
25  {
26      if (index >= MaxNodeCount)
27          throw new IndexOutOfRangeException($"Index {index} is out of bounds for bitmap of size {MaxNodeCount}.");
28
29      // Below is the "easy" way to bitmap, but as 64 == 2^6 we can use bitwise operations - way faster!
30      // bits[index / 64] |= (1UL << (index % 64));
31
32      int arrayIndex = (int)(index >> 6);     // divide by 64
33      int bitPosition = (int)(index & 63);    // modulo 64
34      bits[arrayIndex] |= 1UL << bitPosition;
35  }
36
37  [MethodImpl(MethodImplOptions.AggressiveInlining)]
38  public bool Get(uint index)
39  {
40      if (index >= MaxNodeCount)
41          throw new IndexOutOfRangeException($"Index {index} is out of bounds for bitmap of size {MaxNodeCount}.");
42
43      int arrayIndex = (int)(index >> 6);
44      int bitPosition = (int)(index & 63);
45      return (bits[arrayIndex] & (1UL << bitPosition)) != 0;
46  }
47
48  /// <summary>
49  /// Set a index if it is not set alredy.
50  /// </summary>
51  [MethodImpl(MethodImplOptions.AggressiveInlining)]
52  public bool SetIfNot(uint index)
53  {
54      if (index >= MaxNodeCount)
55          throw new IndexOutOfRangeException($"Index {index} is out of bounds for bitmap of size {MaxNodeCount}.");
56
57      int arrayIndex = (int)(index >> 6);   // divide by 64
58      int bitPosition = (int)(index & 63);  // modulo 64
59      ulong mask = 1UL << bitPosition;
60
61      if ((bits[arrayIndex] & mask) != 0) return false; // Is already set
62      bits[arrayIndex] |= mask;
63      return true; // Was not set before
64  }
```

Figure 3: Code snippet for bitmap.

1A.iii Create 2 graphs

Figure 4 shows the code for graph generation. Table 1 states the input parameters used for generation and other metrics. These graphs are then saved as files and used for every benchmark test for the BFS algorithm going forward, to make sure that the only variable that changes is the searching implementation.

```
public static List<uint>[] GenerateGraph(uint nodeCount, uint maxEdgesPerNode)
{
    Console.WriteLine($"nodeCount: {nodeCount} maxEdgesPerNode: {maxEdgesPerNode}");

    var rand = Random.Shared;
    var graph = new List<uint>[nodeCount];
    var edgeSet = new HashSet<(uint, uint)>();

    for (uint i = 0; i < nodeCount; i++)
        graph[i] = new List<uint>();

    for (uint i = 0; i < nodeCount; i++)
    {
        if (i % Math.Max(1, (int)(nodeCount * 0.01)) == 0)
            Console.WriteLine($"i = {i}");

        uint localMaxEdgeCount = (uint)rand.NextInt64(maxEdgesPerNode);
        while (graph[i].Count < localMaxEdgeCount)
        {
            uint target = (uint)rand.NextInt64(nodeCount);
            if (target != i)
            {
                var edge = (Math.Min(i, target), Math.Max(i, target));
                if (!edgeSet.Contains(edge))
                {
                    graph[i].Add(target);
                    graph[target].Add(i);
                    edgeSet.Add(edge);
                }
            }
        }
    }

    return graph;
}
```

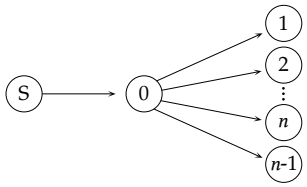
Figure 4: Code snippet for the graph generation. The parameters `nodeCount` is defined, graph-size and `maxEdgesPerNode` is a randomly generated integer.

Name	`nodeCount`	`maxEdgesPerNode`	`binarySize`
G1	100_000	671	170 MB
G2	10_000_000	67	1.68 GB

Table 1: Parameters used for generation.

1A.iv Is BFS embarrassingly parallel?

In this section, the graph shown next to this text is considered. Bearing that in mind, due to the dependency of node 0 for node 1 to n, this graph cannot use multiple processors for step 1 of this solution, and is therefore not embarrassingly parallel. In general, BFS cannot inherently be considered embarrassingly parallel, as it would pose unrealistic graph dependencies.



1A.v Benchmark

Mean (ms)	Error	StdDev	Allocated Memory (MB)	Actual Speedup	Theoretical Speedup
Using graph G1					
77.71	1.450	3.182	1.01	1.00	1.00
Using graph G2					
4216	82.3	123.2	129.19	1.00	1.00

Table 2: Sequential performance (baseline).

1B Shared-Memory Parallel Implementation

1B.i Design

The proposed system consists of a thread-pool where a defined maximum thread count is available, and each a single vertex is searched pr. thread in the pool until the frontier queue is empty. When the frontier is empty, the next, just created, is then used. This is done until the frontier queue is empty. The tracking of visited vertices is done using the same bitmap class described in the sequential case, to prevent race conditions when multiple threads run, a semaphore lock is introduced. At each iteration, the queue is copied and used as the current frontier; subsequently, the queue is cleared and used for the next frontier.

1B.ii Level-by-Level BFS DAG

Figure 5 shows an example scheduling of a BFS level synchronous search where the thread-pool in this case is 4. In the third level, L2, the frontier queue is not equal to the graph structure. This is representing the threads finishing at different times and thus not equals the graph.

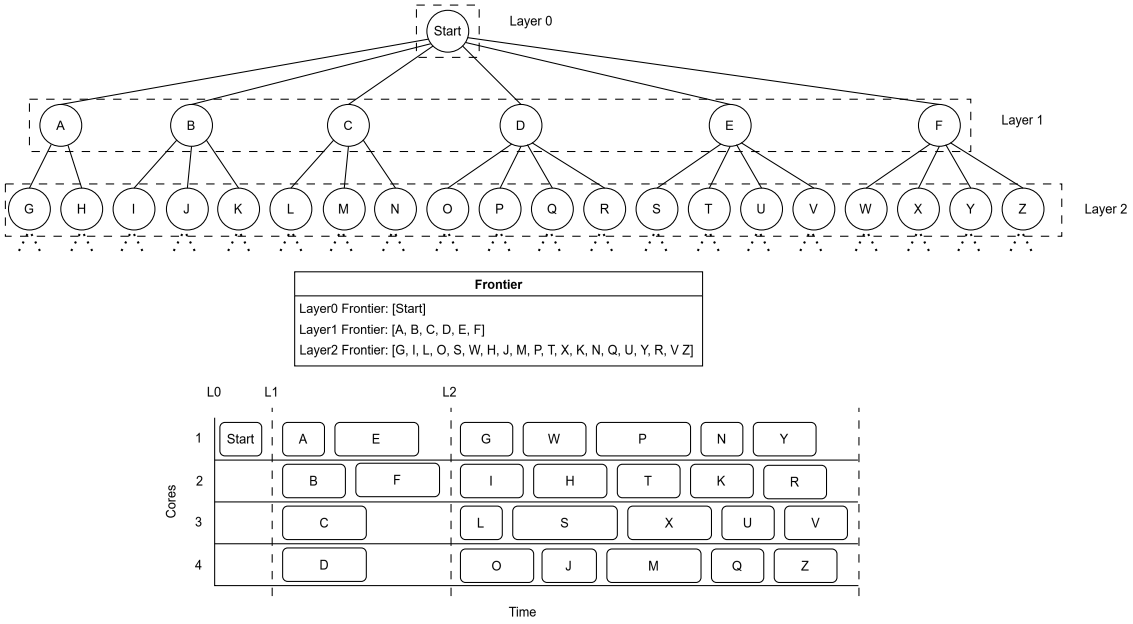


Figure 5: Figure showing task scheduling among four processors.

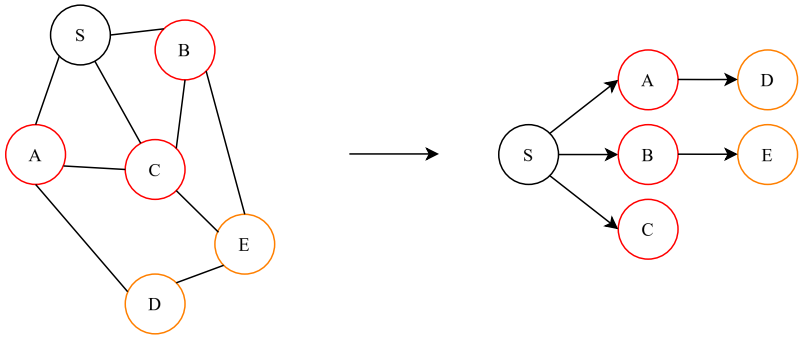


Figure 6: BFS search of a graph split among processors with frontier synchronisation.

1B.iii Total Work, Parallel Depth, Expected Speedup and Efficiency

The shared memory implementation for breadth-first search (BFS) remains asymptotically $\Theta(V + E)$, representing the cost of visiting each vertex and edge in the graph. In the proposed system,

this work is subdivided across P processes, each responsible for a partition of the graph. Ideally, the computational work per process is:

$$W_{comp} = \frac{\Theta(V+E)}{P} \tag{1}$$

However, in practice, the *total work* includes both computation and scheduling overhead. At each BFS level, processes must wait for the slowest process to finish before searching the next frontier. Therefore, the *total work* can be expressed as:

$$W_{total} = \frac{\Theta(V+E)}{P} + W_{copy} + W_{clear} \tag{2}$$

where W_{copy} Cost of copying the queue to a local array.
W_{clear} Overhead of clearing the queue after the copy at each BFS level.

The *parallel depth* of the shared-memory BFS is determined by the number of BFS levels, which corresponds to the *graph's diameter* D or an upper bound on it. Each BFS level represents a synchronization point, meaning all threads must complete processing the current frontier before proceeding to the next level. Thus, the parallel execution time is proportional to D, making it a lower bound on the number of synchronization rounds required.
The *expected speedup* can be approximated as:

$$S(P) \approx \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{\Theta(V+E)}{D \cdot \frac{V+E}{P} + \text{overhead}} \tag{3}$$

Where T_{seq} is the sequential execution time, and T_{par} is the parallel time consisting of per-level work plus overheads such as locking, copying, and load imbalance.
The *parallel efficiency* is defined as:

$$E(P) = \frac{S(P)}{P} \tag{4}$$

Due to synchronization barriers at each BFS level and contention for shared resources (e.g., visited bitmap and queue access), efficiency tends to decrease as P increases, particularly when $P \gg V_{\text{frontier}}$ for many levels. In summary, the algorithm exhibits limited parallel depth (bounded by the graph diameter), and while initial speedups can be significant, efficiency degrades at higher thread counts due to overheads and synchronization constraints.

1B.iv Sources of Overhead

The sources of overhead in the proposed system are as follows: Scheduling of parallel tasks, the semaphore for queuing vertices and marking visited, and lastly, the copying and clearing of the queue at each frontier.

1B.v Implementation of Shared-Memory BFS Algorithm

The implementation uses the built-in thread pool in .NET to handle the scheduling of a maximum number of tasks during iteration over a single frontier. Figure 7 shows the code. The potential race condition when marking a vertex visited and adding a vertex to the frontier queue is handled by a semaphore lock on line 87. Line 70 configures the system to run a single BFS level with a maximum number of threads. Line 76 copies the current queue, and line 77 clears the queue. This is done to enforce level-synchronous search among all threads. Line 79 splits the current frontier among the maximum allowed number of threads. If there are fewer vertices to search than the maximum allowed thread count, only a thread count equal to the vertices count is used. If the vertices to search is larger than the allowed thread count, only the maximum allowed count is running at any given time, then as the first complete new threads are scheduled to search the next vertex, this is done until all vertices at a given level are searched.

```
public static Bitmap BFS_ParallelSharedMemory(List<uint>[] graph, uint startNode, int maxThreads)
{
    int numNodes = graph.Length;
    var visited = new Bitmap(numNodes); // 0: not visited, 1: visited
    visited.Set(startNode);

    var currentFrontier = new Queue<uint>();
    currentFrontier.Enqueue(startNode);

    var options = new ParallelOptions { MaxDegreeOfParallelism = maxThreads }; // e.g., 4

    object _lock = new();

    while (currentFrontier.Count > 0)
    {
        var frontierArray = currentFrontier.ToArray();
        currentFrontier.Clear();

        Parallel.ForEach(frontierArray, options, node =>
        {
            foreach (var neighbor in graph[node])
            {
                // Atomically mark visited
                if (!visited.Get(neighbor))
                {
                    lock (_lock)
                    {
                        visited.Set(neighbor);
                        currentFrontier.Enqueue(neighbor);
                    }
                }
            }
        });
    }

    return visited;
}
```

Figure 7: Code of shared memory implementation.

1B.vi Benchmark

Table 3 lists the benchmarks of the shared memory implementation using different numbers of parallel threads. The baseline is the sequential implementation. The diminishing return of speedup when increasing the thread count is due to the heavy blocking when updating the visited map and adding the vertex to the queue. The G1 graph is fastest using 12 threads, while the larger graph G1 is faster using 16 threads. Of note is a version where the frontier was split and assigned to threads for searching, which did not result in any speedup compared to the current implemented version.

Threads	Mean (ms)	Error	StdDev	Allocated Memory (MB)	Actual Speedup	Theoretical Speedup
Using graph G1						
Baseline	77.71	1.450	3.182	1.01	1.00	1
2	41.63	0.815	1.116	1.40	1.87	2
4	36.69	0.726	0.643	1.40	2.12	4
8	33.42	0.665	1.359	1.40	2.33	8
12	28.15	0.559	1.623	1.41	2.76	12
16	31.74	0.633	1.292	1.41	2.45	16
Using graph G2						
Baseline	4216	82.3	123.2	129.19	1.00	1
2	2444	31.0	27.5	103.35	1.73	2
4	2327	46.4	104.6	103.36	1.81	4
8	2084	41.6	122.0	103.36	2.02	8
12	2017	39.9	80.6	103.37	2.09	12
16	1947	39.0	115.1	103.37	2.16	16

Table 3: Parallel Shared Memory performance vs sequential (baseline).

1C Distributed Design

The proposed system is a system capable of running on both a single machine and across multiple machines connected over TCP/IP. The system consists of two roles: Coordinator and Worker. The coordinator will **not** traverse the graph. It will only coordinate among workers.

1C.i Partitioning

The graph is partitioned round-robin to each worker, each partition is then transferred to the assigned worker. The reason for this method is twofold: one, it's easy, and secondly, it makes it possible to fast identify which worker can traverse a given vertex, thus only locally reachable vertices are sent to each worker.

The workers will sequentially search the frontier send by the coordinator and reply with the new (partial) frontier. The coordinator will, as workers reply, construct new frontiers for each worker, and when all replies have been processed, send the new frontiers. This is repeated until the global frontier is empty. The coordinator updates the global visited bitmap as it constructs the frontiers, then send the updated bitmap to each worker when sending the new frontier. By sending the global bitmap, each worker will only report back nodes not visited, nor being visited. Figure 8 depicts the end-to-end communication between a coordinator and two workers.

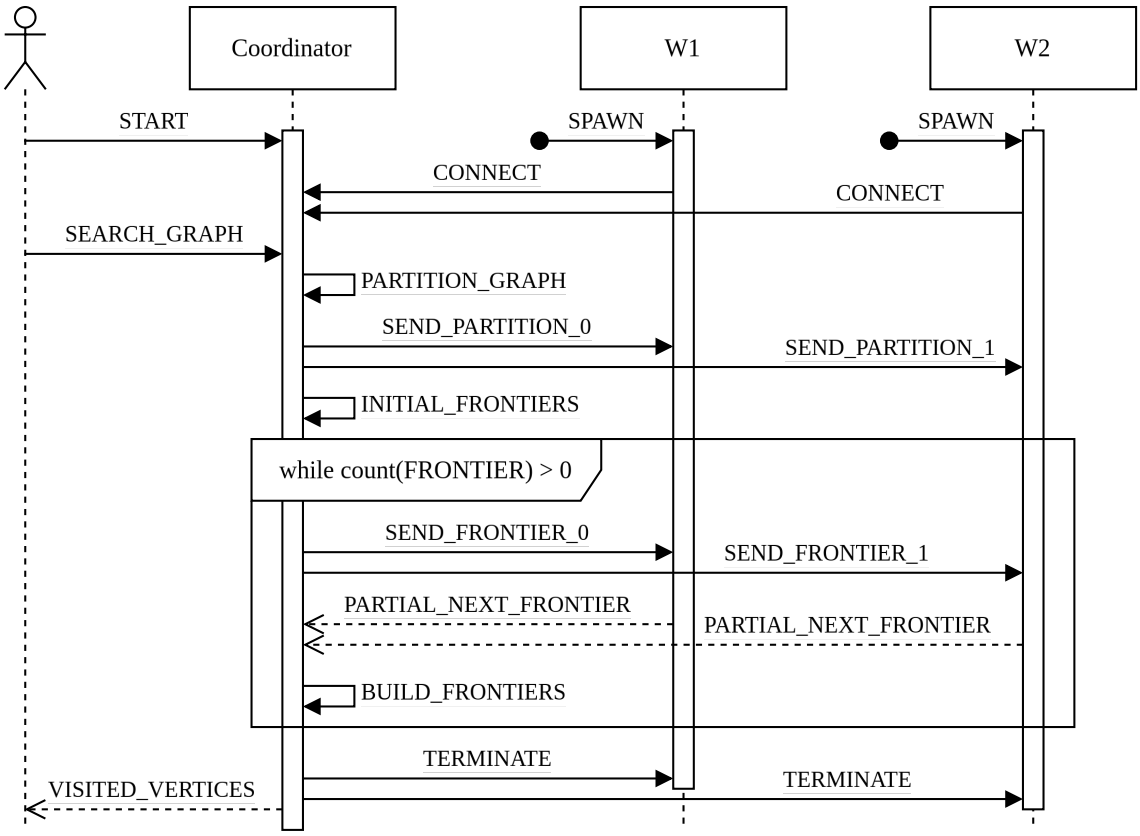


Figure 8: Sequence diagram of coordinator and two workers (W1, W2) communication.

1C.ii Frontier Exchange Cost Analysis

The added cost of frontier exchange consists of the following:

1. transferring the partial frontier to each worker
2. transferring the partial next frontier from each worker
3. update global visited and construct new frontiers

Above adds cost, but bullet 1. & 2. can be done in parallel for each worker. While the 3. bullet can be done sequentially as each worker reports back. By considering this, the total cost is significantly reduced compared to handling 1.,2.,3. sequentially and waiting for all workers between 2. and 3..

1C.iii Work Cost Analysis

In a distributed environment, the total computational work for breadth-first search (BFS) remains asymptotically $\Theta(V + E)$, representing the cost of visiting each vertex and edge in the graph. In the proposed system, this work is subdivided across P worker processes, each responsible for a partition of the graph. Ideally, the *computational work per process* is:

$$W_{comp} = \frac{\Theta(V + E)}{P} \tag{5}$$

However, in practice, the *total work* includes both computation and communication. At each BFS level, processes must exchange frontier information and synchronize global state via the coordinator. Therefore, the *total distributed work* can be expressed as:

$$W_{total} = \frac{\Theta(V + E)}{P} + W_{comm} + W_{sync} \tag{6}$$

where W_{comm} Cost of sending and receiving frontier and visited data.
W_{sync} Overhead of synchronization barriers between BFS levels.

These overheads grow with the number of processes P and the diameter of the graph. Furthermore, if the graph is irregular (e.g., has a skewed degree distribution), load imbalance can further degrade performance, as some workers may perform significantly more work than others.

As a result, while the goal of parallelization is to reduce runtime through concurrent execution, the *speedup* achieved is often sublinear:

$$Speedup(P) < P$$

This reflects the diminishing returns caused by communication overhead and workload imbalance as the number of processes increases.

1C.iv Implementation of Distributed BFS

This section gives a top-down description of the implementation. The code uses dependency injection with interfaces for enhancing the testability of the code. This enables the code to be tested individually using mocks and then, in integration tests, gradually use the real implementation until the final end-to-end test uses all real components. The implementation consists of four projects:

- **Benchmark**: The program used to run all benchmarks.
- **BFSAlgo**: The class library containing all code for all searchers.
- **DistriGraph**: A console application to run the distributed system as a coordinator or a worker. This can connect multiple systems over TCP/IP to run BFS as a real distributed system.
- **Tests**: All unit, integration and end-to-end tests.

Figure 9 depicts the code used to run the distributed implementation on a single computer. Lines 110 - 118 show the process of spawning workers in parallel, and line 121 waits for all the workers to connect before running the search on line 125. The `wait(...)` on lines 121 and 127 is a timeout, if the timeout is reached, the code exits and throws an exception to the user.

```
 99 public static Bitmap BFS_Distributed(List<uint>[] graph, uint startNode, int numWorkers, int millisecondsTimeout = -1)
100 {
101     IPAddress address = IPAddress.Loopback;
102
103     // start server
104     var coordinator = new Coordinator(address, port: 0); // use next available port
105     var runnerTask = coordinator.StartAsync();
106
107     var coordinatorEndpoint = coordinator.ListeningOn;
108
109     // spawn workers
110     var workerTasks = new Task[numWorkers];
111     for (int i = 0; i < numWorkers; i++)
112     {
113         workerTasks[i] = Task.Run(async () =>
114         {
115             var worker = new Worker(coordinatorEndpoint.Address, coordinatorEndpoint.Port);
116             await worker.Start();
117         });
118     }
119
120     // Wait for specific worker count to be connected
121     bool completed = coordinator.WaitForWorkerCountAsync(numWorkers).Wait(millisecondsTimeout);
122     if (!completed) throw new Exception("Timeout waiting for workers!");
123
124     // search graph
125     var visited = coordinator.RunAsync(graph, startNode);
126
127     completed = visited.Wait(millisecondsTimeout);
128     if (!completed) throw new Exception("Timeout searching!");
129
130     return visited.Result;
131 }
```

Figure 9: Code for running distributed implementation on a single computer.

Figure 10 shows the methods in the coordinator. Notice lines 15 and 16, where line 16 is used for testing, as the interface implementations can be mocked.

```
 1  public class Coordinator
 2 {
 3     private readonly ITcpListener _listener;
 4     private readonly INetworkStreamFactory _streamFactory;
 5     private readonly INetworkHelper _networkHelper;
 6
 7     public IPEndPoint ListeningOn => (IPEndPoint)_listener.LocalEndpoint;
 8
 9     private readonly List<INetworkStream> connectedWorkers = new();
10     private Func<Task>? _onWorkerConnectedAsync;
11
12     public int ConnectedWorkers => connectedWorkers.Count;
13
14     // Initialize
15     public Coordinator(IPAddress bindAddress, int port) { ... }
16
17     // Initalize for testing using mocks for the interfaces
18     public Coordinator(ITcpListener listener, INetworkStreamFactory streamFactory, INetworkHelper networkHelper) { ... }
19
20     // Start accepting incomming worker connections
21     public async Task StartAsync() { ... }
22
23     // Internal handling of worker connect
24     private async Task HandleWorkerConnect() { ... }
25
26     // Wait for a specific number of workers to be connected
27     public Task WaitForWorkerCountAsync(int numWorkers) { ... }
28
29     // Search a graph among currently connected workers
30     public async Task<Bitmap> RunAsync(List<uint>[] graph, uint startNode) { ... }
31
32     // Internal helper function to send frontier data in parallel to all workers
33     private async Task SendNewFrontier(INetworkStream[] streams, Bitmap visitedGlobal, List<uint>[] frontier) { ... }
34
35     // Send termination message to all workers in parallel
36     private async Task TerminateWorkers(INetworkStream[] streams) { ... }
37 }
```

Figure 10: Overview of methods in coordinator.

Figure 11 shows the main method in the coordinator. Line 79 ensures that only workers currently connected are used during the search; if additional workers connect, they are not utilized. Lines 83 and 84 partition and send the partial graph to each worker, the transmission to each worker is done in parallel. The specifics of the partitioning are explained later. Lines 91 - 96 initialize the worker-specific frontier and add the start node to the associated worker. The while loop on line 99 iterates over each frontier until all worker frontiers are empty. When it ends, the workers are signalled to terminate (line 120), and the global visited map is returned (line 121). In the

while loop, on line 102, the frontiers and the global visited map are sent to each worker in parallel. Line 116 adds the new unvisited node to the associated worker frontier, which is then sent at the beginning of the loop.

```
76 public async Task<Bitmap> RunAsync(List<uint>[] graph, uint startNode)
77 {
78     // Get current connected workers, we dont want to use workers connecting after this point
79     var streams = connectedWorkers.ToArray();
80     int workerCount = streams.Length;
81
82     // Partition the graph amung workers and transfer it to them
83     List<uint>[] partitioned = GraphPartitioner.Partition(graph, workerCount);
84     await SendPartitions(streams, partitioned, graph);
85
86     // initalize global visited map
87     var visitedGlobal = new Bitmap(graph.Length);
88     visitedGlobal.Set(startNode);
89
90     // initialize frontiers for each worker (a worker only get nodes it actually have)
91     var partitionedFrontier = new List<uint>[workerCount];
92     for (int i = 0; i < partitionedFrontier.Length; i++)
93         partitionedFrontier[i] ??= new List<uint>();
94
95     // add ninitial frontier to associated worker
96     partitionedFrontier[startNode % workerCount] = new List<uint>() { startNode };
97
98     // main search loop, continues until all frontiers are empty
99     while (partitionedFrontier.Any(x => x.Count != 0))
100    {
101        //Send frontier information to all workers
102        await SendNewFrontier(streams, visitedGlobal, partitionedFrontier);
103
104        // while waiting on worker response, clear frontiers for next round
105        for (int i = 0; i < partitionedFrontier.Length; i++)
106            partitionedFrontier[i].Clear();
107
108        // wait on workers to respond
109        var receiveTasks = streams.Select(_networkHelper.ReceiveUintArrayAsync);
110        var results = Task.WhenEach(receiveTasks); // Wait for workers partial frontiers
111
112        // sequentially handle workers frontiers as thy respond
113        await foreach (var result in results)
114            foreach (var node in await result)
115                if (visitedGlobal.SetIfNot(node))
116                    partitionedFrontier[node % workerCount].Add(node);
117    }
118
119    // Tell workers to stop
120    await TerminateWorkers(streams);
121    return visitedGlobal;
122 }
```

Figure 11: Code for running a BFS search distributed. Here, the graph is partitioned to each worker, then coordinates tailored frontiers to each worker at each level. Then wait for workers' response and construct the next tailored frontiers.

Figure 12 shows the code partitioning the graph. The returned data is a list of vertices that each worker should have. The list contains the *index* each vertex has in the adjacency lists.

```
1  public static class GraphPartitioner
2  {
3      // Roundrobin partition
4      public static List<uint>[] Partition(List<uint>[] graph, int partitions)
5      {
6          if (partitions <= 0)
7              throw new ArgumentOutOfRangeException(nameof(partitions), "Partition count must be greater than zero.");
8
9          // Initialize partition map
10         var partitionedGraphs = new List<uint>[partitions];
11         for (int i = 0; i < partitions; i++)
12             partitionedGraphs[i] = new List<uint>();
13
14         for (uint i = 0; i < graph.Length; i++)
15         {
16             int partitionId = (int)(i % (uint)partitions);
17             partitionedGraphs[partitionId].Add(i);
18         }
19
20         return partitionedGraphs;
21     }
22 }
```

Figure 12: Code round-robin partitioning a graph mung a set of workers.

Figure 13 shows the code for a worker. Notice, again, that the constructor accepting implementa-

tions of each interface on line 18 is used for automated tests. The main constructor is on line 15, where it takes an IP-address and port number of a coordinator to connect to. Line 22 starts the worker, connects to the coordinator, and then waits until it receives the assigned partial graph. The `runMainLoop` runs for each frontier it receives until a termination signal is received. Line 43 waits for a frontier and the global visited map from the coordinator, then updates the local visited map with the new on line 43. The newly received frontier is searched on line 51, the specific searching implementation is described later. The searcher returns the new local frontier, which is responded back to the coordinator on line 55.

```
1  public class Worker
2  {
3      private readonly IPAddress coordinatorAddress;
4      private readonly int coordinatorPort;
5      private INetworkStream _stream;
6      private readonly INetworkHelper _networkHelper;
7      private Bitmap visited;
8
9      // Will point to a array of pointers, with a length equal the global node count
10     // we only save a pointer to the neighbors for each node and thus not too large - but fast for lookup
11     // if a worker, worskcase, try to access a node it do not have, an emppty array is returned
12     private ArraySegment<uint>[] partialGraph;
13
14
15     public Worker(IPAddress coordinatorAddress, int coordinatorPort) { ... }
16
17     // Constructor for DI/testing
18     public Worker(INetworkStream stream, INetworkHelper networkHelper) { ... }
19
20
21     public async Task Start()
22     {
23         // connect to coordinator
24         _stream ??= await NetworkStreamWrapper.GetWorkerInstanceAsync(coordinatorAddress, coordinatorPort);
25
26         // get partial graph and the global vertex count
27         (this.partialGraph, var totalNodeCount) = await _networkHelper.ReceiveGraphPartitionAsync(_stream);
28
29         // initialize local visited map
30         visited = new Bitmap(totalNodeCount);
31
32         // run until termination from coordinator
33         await RunMainLoop();
34
35         _stream.Close();
36     }
37
38     private async Task RunMainLoop()
39     {
40         while (true)
41         {
42             // get frontier information
43             var frontier = await _networkHelper.ReceiveUintArrayAsync(_stream);
44             if (frontier == null) break; // detect termination
45
46             // get the updated global visited, and update local map
47             var globalVisited = await _networkHelper.ReceiveByteArrayAsync(_stream);
48             visited.OverwriteFromByteArray(globalVisited);
49
50             // search single frontier in the partial graph recieved
51             var nextFrontier = FrontierSearchers.SearchFrontier(frontier, partialGraph, visited);
52
53             // respond with next parial frontier
54             var data = nextFrontier.ToReadOnlyMemory();
55             await _networkHelper.SendByteArrayAsync(_stream, data);
56             await _networkHelper.FlushStreamAsync(_stream);
57         }
58     }
59 }
```

Figure 13: Worker code, where line 27 receives the partial graph and line 43 receives each frontier update and line 51 searches the partial frontier and responds on line 55.

Figure 14 shows the code searching a single frontier locally on a worker. The single-level search is done sequentially and updates the local visited map on line 27. By receiving the global visited map, the worker only adds vertices that have not been visited nor are currently being visited by other workers.

```
/// <summary>
/// Searchers for a single frontier, used by workers
/// </summary>
public class FrontierSearchers
{
    /// <summary>
    /// Seach a single frontier and return the next
    /// </summary>
    /// <param name="frontier">
    /// Set of nodes to search, each value should correspond to an index in <param name="partialGraph"/>
    /// </param>
    /// <param name="partialGraph">A partial graph to serch, the length shuld equal the global graph node count</param>
    /// <param name="visited">The currently visited nodes, this will be updated as <paramref name="frontier"/> is searched</param>
    /// <returns>A list of nodes for the next frontier</returns>
    public static List<uint> SearchFrontier(Span<uint> frontier, ArraySegment<uint>[] partialGraph, Bitmap visited)
    {
        List<uint> nextFrontier = new();
        if (frontier.IsEmpty) return nextFrontier;

        for (int i = 0, fl = frontier.Length; i < fl; i++)
        {
            var neighbors = partialGraph[frontier[i]];

            for (int j = 0, length = neighbors.Count; j < length; j++)
            {
                uint neighbor = neighbors[j];
                if (visited.SetIfNot(neighbor))
                    nextFrontier.Add(neighbor);
            }
        }

        return nextFrontier;
    }
}
```

Figure 14: Code for searching a single frontier on a worker.

To ensure fast network transfer of the graph partitions, frontier data and visited map, quite some time have been spent on optimizing the send/receive functions. The primary optimization comes from the reduction of time spent converting and copying data by accessing the data using pointers and reading back the raw byte data as another type and likewise in reverse, look at received byte data as a different type. The partial graph data is transferred in binary using the format depicted in figure 15. The upper part of the figure shows the format where **B** is bytes. The lower part of the figure shows an example encoding of a graph split between two workers, where worker zero has the vertices highlighted in bold and worker one have the vertices highlighted in italic. The reason for the "x4" in the header of each workers data is to represent the value is in bytes and each of the fields after all are unsigned int32 i.e. 4 bytes each.

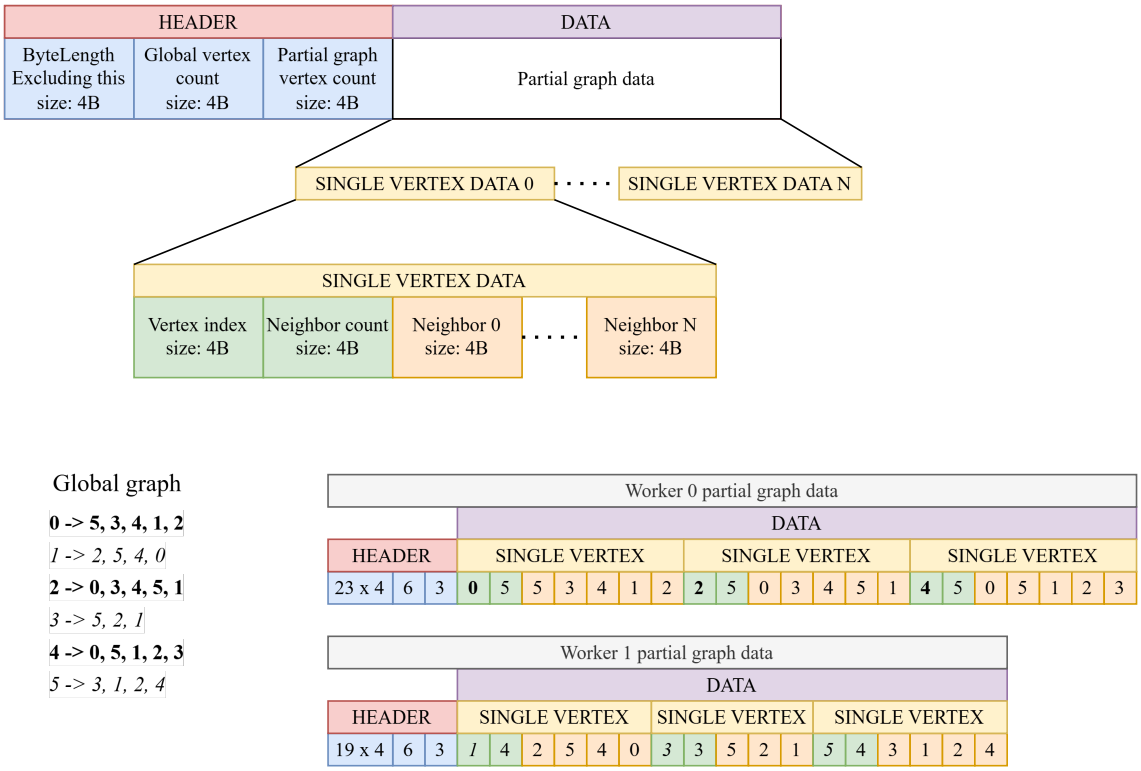


Figure 15: Upper part shows the data format of transmitting a partial graph, here **B** denote bytes. The lower part shows an example of a graph being split, encoded and sent to two workers.

During development, numerous unit, integration, and end-to-end tests have been implemented as seen in figure 16. With a total of 73 tests, where 60 are unit-tests resulting in 99% code coverage - **that is, automated tests cover 99% of the entire code base**. The tests have helped tremendously when rapid prototyping was conducted during optimization phases. The test shown here also covers the sequential and shared-memory implementations documented earlier. The integration tests gradually introduce the real interface implementations instead of mocked versions, by doing this, possible integration errors in individual modules can easily be discovered.

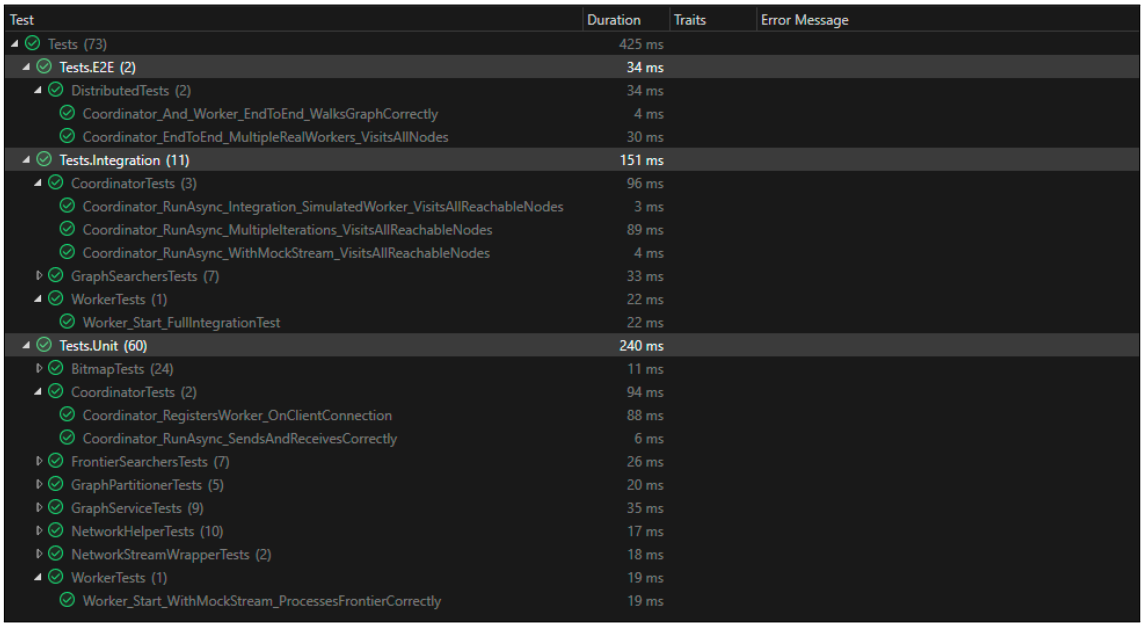


Figure 16: Overview of automated tests; Unit, Integration and End-to-End.

Figure 17 shows one of the 11 integration tests, specifically the full integration test of a worker. Here, the controller is simulated, and the test utilizes the real implementations of a worker, TCP streams and lower-level network helpers. The result from the worker is asserted to be correct based on the graph sent in the beginning.

```
public class WorkerTests
{
    [Fact]
    public async Task Worker_Start_FullIntegrationTest()
    {
        // Arrange
        var assignedNodes = new List<uint> { 0, 1 };
        var fullGraph = new List<uint>[]
        {
            [1],
            [0],
            []
        };

        INetworkHelper networkHelper = new NetworkHelper();
        // setup tcp server at any available port
        var listener = new TcpListener(IPAddress.Loopback, port: 0);
        listener.Start();
        var serverEndpoint = (IPEndPoint)listener.LocalEndpoint;

        // initiate worker
        var worker = new Worker(serverEndpoint.Address, serverEndpoint.Port);
        var workerTask = Task.Run(worker.Start);

        // wait for worker connection
        var tcpClient = await listener.AcceptTcpClientAsync();
        var networkStream = new NetworkStreamWrapper(tcpClient);

        // Act
        // 0. Send partial graph
        await networkHelper.SendGraphPartitionAsync(networkStream, assignedNodes, fullGraph);
        // 1. Send a frontier containing node 0
        await networkHelper.SendByteArrayAsync(networkStream, new List<uint> { 0 }.ToReadOnlyMemory());
        // 2. Send empty visited bitmap
        var visitedBitMap = new Bitmap(fullGraph.Length);
        await networkHelper.SendByteArrayAsync(networkStream, visitedBitMap.AsReadOnlyMemory);
        await networkStream.FlushAsync();

        // 3. Receive the new frontier
        var receivedFrontier = await networkHelper.ReceiveUintArrayAsync(networkStream);

       // 4. Send termination signal
        await networkHelper.SendByteArrayAsync(networkStream, null);
        await networkStream.FlushAsync();

        // Wait for the worker to terminate
        await workerTask;

        // Assert
        Assert.Single(receivedFrontier);
        Assert.Equal(1U, receivedFrontier[0]);
    }
}
```

Figure 17: Integration test of a Worker using real TCP streams and mocking behaviour of a coordinator. The test aims to validate that the worker correctly interacts with the underlying modules, such as the send/receive of partial graph data and frontier information exchange.

1C.v Benchmark

Table 4 shows the benchmarks of the distributed system running on a single machine. Due to the heavy overhead, primarily at startup, the distributed system is slower than the sequential for G1 in all cases. For the larger graph G2, the distributed system shows a marginal speedup up to 8 workers, where the fastest is using 4. For 12 and 16 workers, the overhead is larger than the performance gain and thus results in a speedup < 1.

Workers	**Mean (ms)**	**Error**	**StdDev**	**Allocated Memory (MB)**	**Actual Speedup**	**Theoretical Speedup**
Using graph G1						
Baseline	77.71	1.450	3.182	1.01	1.00	1
2	177.59	3.546	7.480	182.74	0.44	2
4	144.64	2.848	5.418	187.33	0.54	4
8	125.69	2.492	5.146	197.08	0.62	8
12	125.37	2.470	5.100	203.51	0.62	12
16	134.65	2.666	3.908	211.22	0.58	16
Using graph G2						
Baseline	4216	82.3	123.2	129.19	1.00	1
2	3750	74.9	114.4	2860.31	1.12	2
4	3245	64.5	170.0	3498.4	1.30	4
8	3670	88.2	259.9	8041.17	1.15	8
12	4368	86.8	222.6	10647.99	0.97	12
16	4674	97.5	285.9	9261.84	0.90	16

Table 4: Distributed performance vs sequential (baseline).

2 JPEG Image Compression

The chosen language for this workshop part is Python. The code for this part can be found on GitHub here click me.

2.i Detailed Explanation of Each Step Along with Work and Complexity

This section analyzes the computational work and complexity of the first five steps typically involved in JPEG image compression. Let $n \times n$ be the size of the input image, so the total number of pixels is n^2. The image is assumed to be square for simplicity, but the analysis applies similarly to rectangular images.

1. **Conversion**
   The image is converted to grayscale or only the luma component is kept. The pixel values are centered around 0.
   - **Work:** This involves converting each pixel's RGB values to a single grayscale value or extracting the luma component.
   - **Complexity:** $O(n^2)$, as each of the n^2 pixels must be processed.
2. **Block Partitioning**
   The image is divided into blocks of size 8×8 pixels.
   - **Work:** Partitioning the image into smaller blocks so that the next steps can be carried out correctly. If the image dimensions are not divisible by 8, padding is required.
   - **Complexity:** $O(n^2)$, as each pixel is assigned to a block.
3. **Performing the 2D DCT**
   The 2D Discrete Cosine Transform (DCT) is applied to each block.
   - **Work:** Compute the DCT for each 8×8 block using a fast DCT algorithm.
   - **Complexity:** $O(n^2 \log K)$, where $K = 64$, since the DCT is applied independently to $\frac{n^2}{K}$ blocks, each of size K, and computing the DCT of one block takes $O(K \log K)$.
4. **Quantization**
   The DCT coefficients are divided by a quantization matrix and rounded to the nearest integer.
   - **Work:** Perform element-wise division and rounding for each coefficient in each block.
   - **Complexity:** $O(n^2)$, as each of the n^2 DCT coefficients is processed.
5. **Reconstruction**
   The inverse DCT is applied to the quantized coefficients to transform them back into the spatial domain.
   - **Work:** Compute the inverse DCT for each 8×8 block.
   - **Complexity:** $O(n^2 \log K)$, similar to the forward DCT, as it involves the same number of operations per block.

2.ii Directed Acyclic Graphs (DAGs) for the parallel implementation of steps 1 to 5

In order to provide a simple and illustrative explanation of the parallel behavior of the JPEG compression algorithm, the required steps are represented with a reduced number of processing units, which facilitates the construction and interpretation of a Directed Acyclic Graph, DAG.

To illustrate the parallel distribution, a configuration of **5 processors** is assumed. This number reflects the Message Passing Interface, MPI, implementation model, since a small number of processors are used, contrary to GPU-based implementations, which involve many threads. Then, 5 becomes the optimal number of processors for this algorithm, as justified later in Section X.

- **Image Partitioning and Block Division for the example**

  For this purpose, a square image of size 400×400 **pixels** is considered, providing a concrete example while still preserving the structure of a parallelizable implementation. Since the image is processed in non-overlapping blocks of 8×8 pixels, it follows that the image contains 50 blocks per row and 50 blocks per column, resulting in a total of 2500 blocks of size 8×8 pixels.

  The MPI implementation divides the image among the processes in a row-wise manner. Each process is assigned a horizontal strip consisting of 80 pixel rows, since $400/5 = 80$. Given that each block spans 8 rows, each process manages 10 block rows, with 50 blocks per row, resulting in a total of 500 blocks per process.

Process	Image Rows (pixels)	Block Rows (8 px)	Block Range
P0	0–79 (80 px)	0–9	B0 – B499
P1	80–159 (80 px)	10–19	B500 – B999
P2	160–239 (80 px)	20–29	B1000 – B1499
P3	240–319 (80 px)	30–39	B1500 – B1999
P4	320–399 (80 px)	40–49	B2000 – B2499

Table 5: Block-level partitioning of a 400×400 image using 5 MPI processes.

- **DAG Structure and Parallel Execution**

  Each MPI processor receives a distinct and non-overlapping segment of the image and processes its assigned 8×8 blocks independently. As a result, the DAG associated with this implementation consists of five parallel branches (one per processor) as illustrated in figure 18.

  As later developed in Section X, the implementation codes execute steps 3–5 (DCTs, Quantization and Reconstruction) in a loop over the assigned blocks to each process. The reconstructed image blocks of all branches are gathered into the final compressed image.

  The following DAG is not intended as a performance-optimized design, but rather as a pedagogical representation of parallelism in JPEG compression, which will serve as a conceptual foundation for the more advanced discussion in later sections.

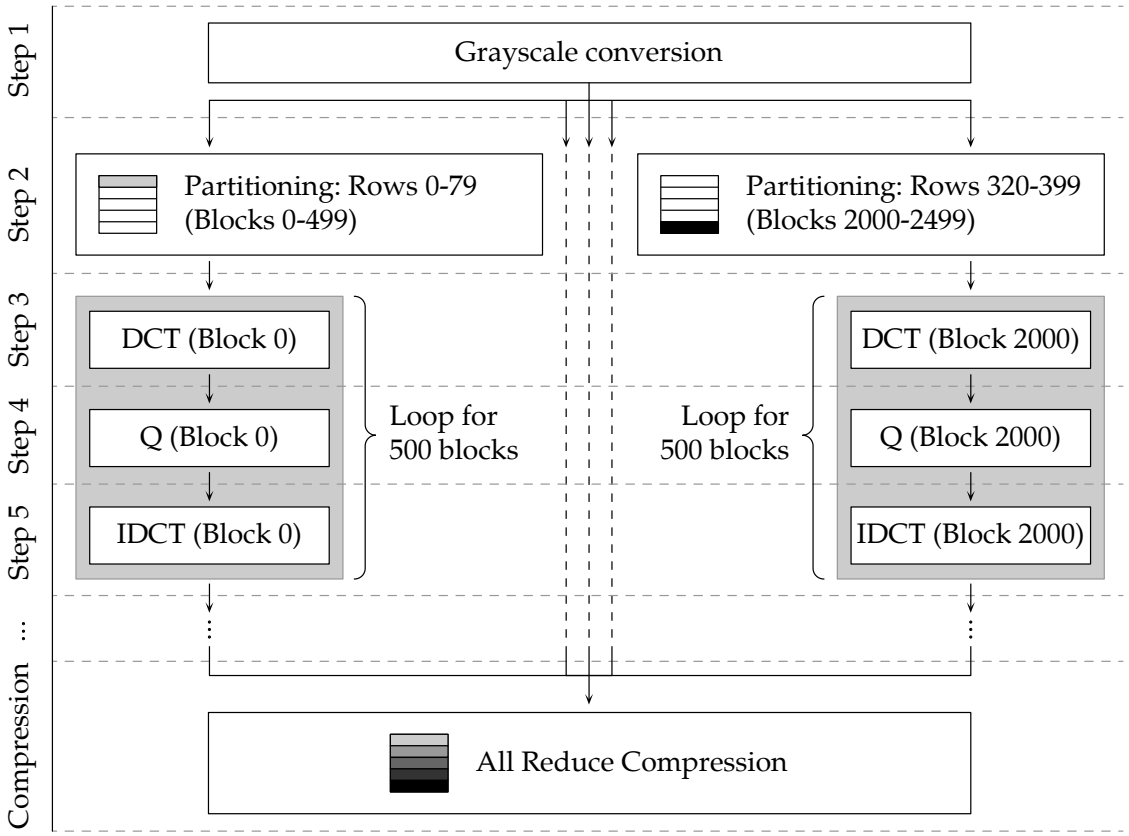


Figure 18: Directed Acyclic Graph, DAG, representing the parallel execution of JPEG compression steps using 5 MPI processors over a 400×400 pixel image.

2.iii Analysis of Implementation Methods

In this section, different implementation strategies for the five main steps of JPEG compression (from color conversion to reconstruction) are analyzed. Each method is evaluated in terms of expected performance depending on the computational characteristics of the steps and the size of the blocks being processed.

1. **Explicit Vectorization with Numpy**
   Numpy excels at operations that can be expressed in terms of whole-array calculations. It is particularly efficient for simple and repetitive operations such as converting RGB to grayscale, splitting the image into blocks, or applying quantization. It benefits steps like: **Step 1 (Conversion)**, **Step 2 (Block Partitioning)**, and **Step 4 (Quantization)**. However, Numpy does not offer significant benefits for complex transformations like DCT.

2. **Just-In-Time Compilation with Numba (JIT)**
   Numba JIT allows Python functions to be compiled into machine code, leading to considerable speedups for numerical operations that cannot be easily vectorized. It is particularly effective in **Step 3 (DCT)** and **Step 5 (Reconstruction)**, which involve nested loops and floating-point calculations.

3. **Numba Vectorization**
   Numba lets you define functions that operate element-wise on arrays. It can outperform standard Numpy in specific scalar operations, such as those in **Step 4 (Quantization)**. However, it is not suitable for operations with internal loops or multiple dimensions like DCT, and offers limited improvement for large or complex data structures.

4. **Numba JIT with parallel=True**
   Enabling parallelism in Numba JIT allows the compiler to parallelize loops automatically. This can significantly speed up block-wise operations, especially in **Step 3 (DCT)** and **Step 5 (Reconstruction)**, where each 8×8 block can be processed independently. This method is particularly efficient for large images where block-level parallelism can be fully exploited on multi-core CPUs.

5. **Multiprocessing**
   Using the `multiprocessing` module, it is possible to split work among several processes. This is well-suited for independent tasks like computing the DCT or the inverse DCT on separate blocks. It performs best on large images with sufficient computation per block to offset the overhead of inter-process communication. Best used in **Step 3 (DCT)** and **Step 5 (Reconstruction)**.

6. **Threading**
   Threading is not ideal for any of the steps in image compression, as it mostly benefits long I/O operations. The only place this operations would take place during image compression would be if the image(s) were located on a network share and not locally on the host.

7. **Pool Executor (concurrent.futures)**
   This high-level interface allows parallel execution of functions using either threads or processes. It is convenient for distributing DCT calculations or other block-level work across processes. It provides similar performance to multiprocessing, with easier syntax. Applicable to **Step 3 (DCT)** and **Step 5 (Reconstruction)**

8. **Message Passing Interface (MPI)**
   MPI is designed for distributed systems and is powerful in high-performance computing (HPC) environments. It enables running computations across multiple machines. While overkill for small-scale problems, it is ideal for processing very large images or batch-processing many images simultaneously in **Step 3 (DCT)** and **Step 5 (Reconstruction)**. However, it introduces significant development and infrastructure overhead.

9. **GPU Programming (CUDA or OpenCL)**
   Using the GPU enables massive parallelism and is extremely effective for compute-intensive operations with large datasets, like **Step 3 (DCT)** and **Step 5 (Reconstruction)**. However, the benefits depend on efficient memory transfer between CPU and GPU.

Method	Best suited for	Most benefited steps	Optimal block sizes
Numpy (Vectorized)	Simple, vectorizable operations	1, 2, 4	Small to medium
Numba JIT	Iterative code not easily vectorized	3, 5	Small to medium
Numba Vectorize	Element-wise simple operations	1, 4	Small
Numba JIT (Parallel=True)	Parallel computation over blocks	3, 5	Medium to large
Multiprocessing	Independent processes per block	3, 5	Large
Threading	I/O-bound or very light tasks	None relevant	Not applicable
Pool Executor (Futures)	Simpler multiprocessing alternative	3, 5	Medium to large
MPI (Message Passing)	Distributed computation across nodes	3, 5 (at scale)	Very large
GPU (CUDA/OpenCL)	Massive parallelism	3, 5 (Compute-heavy)	Very large

Table 6: Overview of the strengths, suited steps, and optimal problem sizes for each implementation method.

While Numpy and Numba-based methods provide benefits in a greater number of steps, the major computational load of JPEG compression lies in steps 3 (2D DCT) and 5 (Inverse DCT). These steps involve complex matrix operations and are repeated over many blocks, making them the bottleneck in terms of performance.

Therefore, despite being useful in smaller or simpler steps, the methods that offer the best overall acceleration are GPU (CUDA/OpenCL) and MPI, which excel at handling large-scale, parallelizable, and computation-heavy tasks. When optimizing for real performance gains, focusing on these two steps with high-performance parallel methods yields the most significant improvements. This our case, if it is compared the performance for small and large block sizes then the results are different, since MPI and GPU are not that good for that kind of situations.

The following two pages show tables 7 and 8, which enumerate the performance of each model at each stage.

Step	Method Ranking (Best to Worst)
Step 1: Conversion	1. Numpy (vectorized) 2. Numba Vectorize 3. Numba JIT 4. Numba JIT (parallel=True) 5. Pool Executor 6. Multiprocessing 7. Threading 8. MPI 9. GPU (CUDA/OpenCL)
Step 2: Block Partitioning	1. Numpy (vectorized) 2. Numba JIT 3. Numba JIT (parallel=True) 4. Pool Executor 5. Multiprocessing 6. Threading 7. MPI 8. GPU (CUDA/OpenCL) 9. Numba Vectorize
Step 3: 2D DCT	1. Numba JIT 2. Numba JIT (parallel=True) 3. Pool Executor 4. Multiprocessing 5. Numpy (vectorized) 6. Numba Vectorize 7. Threading 8. GPU (CUDA/OpenCL) 9. MPI
Step 4: Quantization	1. Numpy (vectorized) 2. Numba Vectorize 3. Numba JIT 4. Numba JIT (parallel=True) 5. Pool Executor 6. Multiprocessing 7. Threading 8. MPI 9. GPU (CUDA/OpenCL)
Step 5: Inverse DCT	1. Numba JIT 2. Numba JIT (parallel=True) 3. Pool Executor 4. Multiprocessing 5. Numpy (vectorized) 6. Numba Vectorize 7. Threading 8. GPU (CUDA/OpenCL) 9. MPI

Table 7: Best to worst method ranking per step for small block sizes.

Step	Method Ranking (Best to Worst)
Step 1: Conversion	1. Numpy (vectorized) 2. Numba JIT 3. Numba JIT (parallel=True) 4. Pool Executor 5. Multiprocessing 6. Numba Vectorize 7. MPI 8. GPU (CUDA/OpenCL) 9. Threading
Step 2: Block Partitioning	1. Numba JIT 2. Numba JIT (parallel=True) 3. Pool Executor 4. Multiprocessing 5. Numpy (vectorized) 6. MPI 7. GPU (CUDA/OpenCL) 8. Threading 9. Numba Vectorize
Step 3: 2D DCT	1. GPU (CUDA/OpenCL) 2. MPI 3. Multiprocessing 4. Pool Executor 5. Numba JIT (parallel=True) 6. Numba JIT 7. Numpy (vectorized) 8. Numba Vectorize 9. Threading
Step 4: Quantization	1. Numba JIT (parallel=True) 2. Numpy (vectorized) 3. Numba JIT 4. Pool Executor 5. Multiprocessing 6. GPU (CUDA/OpenCL) 7. MPI 8. Numba Vectorize 9. Threading
Step 5: Inverse DCT	1. GPU (CUDA/OpenCL) 2. MPI 3. Multiprocessing 4. Pool Executor 5. Numba JIT (parallel=True) 6. Numba JIT 7. Numpy (vectorized) 8. Numba Vectorize 9. Threading

Table 8: Best to worst method ranking per step for large block sizes.

2.iv DCT algorithm sequentially and in parallel with four different methods

For the implementation of the code, the following methods are the ones selected:

- **For vectorization**: Numpy explicit vectorization and Numba JIT
- **For parallelization**: MPI multiprocessing and openCL.

The implemented code reproduces the core structure of JPEG compression but omits several advanced steps to maintain clarity and focus on the most essential transformations. It performs a conversion from RGB to YCbCr to separate luminance from chrominance, applies padding to make image dimensions compatible with block-based processing, and computes a 2D DCT on each 8×8 block. Quantization is performed using standard JPEG matrices, and inverse steps reconstruct the image back to RGB.

The Discrete Cosine Transform (DCT) is a key component in image compression techniques like JPEG. It transforms spatial data into frequency components, allowing high-frequency components (which are often less perceptible to the human eye) to be discarded or reduced. First, as demonstrated in figure 19, the decomposition of the RGB data into YCbCr is implemented:

```python
# Step 1: RGB to YCbCr conversion (keep Y/luma channel)
def rgb_to_ycbcr(image):
    R = image[:, :, 0]
    G = image[:, :, 1]
    B = image[:, :, 2]
    Y  =  0.299 * R + 0.587 * G + 0.114 * B
    Cb = -0.168736 * R - 0.331264 * G + 0.5 * B + 128
    Cr =  0.5 * R - 0.418688 * G - 0.081312 * B + 128
    return Y, Cb, Cr

# Reverse conversion from YCbCr to RGB
def ycbcr_to_rgb(Y, Cb, Cr):
    R = Y + 1.402 * (Cr - 128)
    G = Y - 0.344136 * (Cb - 128) - 0.714136 * (Cr - 128)
    B = Y + 1.772 * (Cb - 128)
    rgb = np.stack((R, G, B), axis=-1)
    return np.clip(rgb, 0, 255).astype(np.uint8)
```

Figure 19: Decomposition of the RGB data into YCbCr and it's inverse

Once this step is completed, the block partitioning starts. This step was implemented alongside the 3 to 5 step, so the following variations of the code will show the actual implementation.

DCT Implementation with Numpy:
For the Numpy implementation, the Numpy cosines function included in the Numpy module were the first option for doing the twin loop for the calculation for the 2-dimensional DCT. This implementation took hours to complete a mere 4 MB .png image, so it was decided to use the matrix functions of Numpy, achieving amazing improvements in speed. As numpy benefits from array calculations rather than applying loops with it. Figure 20 shows how the code has been implemented.

```python
# Step 3: Generate DCT transformation matrix
def dct_matrix(N=8):
    C = np.zeros((N, N))
    for k in range(N):
        for n in range(N):
            alpha = np.sqrt(1/N) if k == 0 else np.sqrt(2/N)
            C[k, n] = alpha * np.cos(np.pi * (2*n + 1) * k / (2 * N))
    return C

# Step 3: 2D DCT and Inverse DCT
def dct_2d(block, C):
    return C @ block @ C.T

def idct_2d(block, C):
    return C.T @ block @ C
```

Figure 20: DCT implemented using NumPy

DCT Implementation with Numba:
For the Numba using JIT (Just-In-Time), the twin cosinus loops for calculating the 2 dimension DCT were also used as can be seen in figure 21. Although, the loop was for every 8×8 block, making the loop quite small, so it did not benefit much of the JIT implementation, although it showed a big time improvement for the calculation of the loop. Probably, this implementation will benefit of bigger 2D DCT blocks. The rest of the code is exactly the same as the Numpy one, except the section described above.

```python
# Step 3: Define DCT using Numba JIT
@jit(nopython=True)
def dct_2d_numba(block):
    N = block.shape[0]
    result = np.zeros((N, N))
    for u in range(N):
        for v in range(N):
            sum_val = 0.0
            for x in range(N):
                for y in range(N):
                    sum_val += block[x, y] * np.cos(np.pi * (2*x + 1) * u / (2 * N)) * np.cos(np.pi * (2*y + 1) * v / (2 * N))
            alpha_u = np.sqrt(1/N) if u == 0 else np.sqrt(2/N)
            alpha_v = np.sqrt(1/N) if v == 0 else np.sqrt(2/N)
            result[u, v] = alpha_u * alpha_v * sum_val
    return result
```

Figure 21: DCT implemented using Numba

DCT Implementation with MPI:
As the MPI implementation uses a different number of processors, most of the code needs to be adjusted. Firstly, it uses the core 0 of the CPU for uploading the image, in order to extract the RGB information with `rgb = comm.bcast(rgb, root=0)`. Once the RGB information is extracted, the core 0 broadcasts the RGB information to all the assigned cores.

Once the image is loaded, each core proceeds with the first step: RGB decomposition. After that, Core 0 calls the `process_channel_parallel` function to create the list of all 8×8 block positions. These blocks are then distributed among the cores using the following expression: `local_blocks = [block_positions[i] for i in range(len(block_positions)) if i % size == rank]`.

Each core then performs steps 2 to 5 sequentially on its assigned blocks and stores the output in a local matrix. Once all cores have completed their processing, Core 0 gathers the compressed matrices from each core, with the function `comm.Allreduce` and reconstructs the full compressed channel (Y, Cb, or Cr). Then the CPU 0 does the rest of steps to restore the jpeg image.
The `dct_2d_vector` uses the same dct calculation using numpy matrices, as it's faster than just doing the dct calculation directly as can be seen in figure 22.

```python
# === Step 5: Parallel DCT Compression (block-wise distribution) ===
def process_channel_parallel(channel, Q):
    block_size = 8
    h, w = channel.shape
    compressed = np.zeros_like(channel)

    # Create list of all 8x8 block positions
    block_positions = [(i, j) for i in range(0, h, block_size)
                              for j in range(0, w, block_size)]

    # Distribute blocks among processes
    local_blocks = [block_positions[i] for i in range(len(block_positions)) if i % size == rank]

    for i, j in local_blocks:
        block = channel[i:i+block_size, j:j+block_size]
        if block.shape != (block_size, block_size):
            continue  # Skip incomplete blocks
        block -= 128
        dct_block = dct_2d_vec(block)
        quantized = np.round(dct_block / Q)
        dequantized = quantized * Q
        idct_block = idct_2d_vec(dequantized) + 128
        compressed[i:i+block_size, j:j+block_size] = idct_block

    # Reduce all partial results into full image
    full_compressed = np.zeros_like(channel)
    comm.Allreduce(compressed, full_compressed, op=MPI.SUM)
    return full_compressed
```

Figure 22: DCT implemented using MPI (continued)

DCT Implementation with GPU Parallelization:
In our OpenCL implementation, the host (CPU) and the device (GPU) interact through a series of well-defined steps that include memory allocation, data transfer, kernel execution, and result retrieval. These steps are critical to offloading computation to the GPU and ensuring that the processed data is correctly returned to the host.

1. **Open CL setup**

   The first thing the code does is to extract the RGB information form the image and then it proceeds to do the first step of transforming the RGB information to YCrCb. After it does this, it start the set up of the openCL:

   ```
   platform = cl.get_platforms()[0]
   device = platform.get_devices(cl.device_type.GPU)[0]
   ctx = cl.Context([device])
   queue = cl.CommandQueue(ctx)
   mf = cl.mem_flags
   ```

   This code initializes the OpenCL environment by selecting the first available platform and GPU device. It then creates a context for that device and a command queue to send instructions to it. The `mf` variable is used to define memory flags for creating buffers.

2. **Buffer Allocation (GPU Memory)** Before any data can be processed on GPU, memory buffers is allocated on the device. This is performed using `cl.Buffer` function, where the size of the type of access(just writing, writing, or reading and writing) and the size of the buffer are specified. To ensure the buffer with data from the host, the `COPY_HOST_PTR` flag is used along with the actual data array:

   ```
   mf = cl.mem_flags
   input_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=input_data)
   output_buf = cl.Buffer(ctx, mf.WRITE_ONLY, input_data.nbytes)
   ```

This operation both allocates memory on the GPU and, optionally, uploads initial data.

3. **Data Transfer: Host to Device**

   Data is typically sent from the CPU to the GPU using the `COPY_HOST_PTR` flag during buffer creation, as shown above. Alternatively, the transfer can be explicitly performed with:

   ```
   cl.enqueue_copy(queue, device_buffer, host_array)
   ```

   This explicit method gives more control and can be useful when reusing buffers or updating data between kernel calls.

4. **Kernel Execution on the GPU**

   Once the input data is on the GPU, Is it executed the kernel(a parallel function written in OpenCL C) by calling it from the host with the following syntax: `program.kernel_name(queue, global_size, local_size, args...)`

   For example:
   `program.dct_quant(queue, (num_blocks,), None, input_buf, output_buf, quant_buf)`

   This command schedules the kernel for execution on the GPU using the specified work sizes and arguments (buffers and scalars).

5. **Data Transfer: Device to Host**

   After the kernel completes execution, the results are transferred back to the CPU memory. This is done using the `cl.enqueue_copy` function, where it copies the data from a device buffer to a host array (`cl.enqueue_copy(queue, host_array, device_buffer)`):

   For example:
   `dct_blocks = np.empty_like(blocks)`
   `cl.enqueue_copy(queue, dct_blocks, output_buf)`

   This step is essential to retrieve and use the GPU-processed data on the host side, such as for visualization or further CPU-based processing.

In summary, the data path involves copying input data to GPU buffers, launching the kernel to process the data in parallel, and then copying the results back to the CPU. Efficient use of these operations is critical to achieving high performance, as excessive or unnecessary transfers between the host and device can become a bottleneck. In figure 23, the DCT algorithm can be seen.

```
kernel_code = """
__kernel void dct_quant(__global float* input, __global float* output, __global float* Q, int N) {
    int block_id = get_global_id(0);
    int u = get_global_id(1) / N;
    int v = get_global_id(1) % N;

    float sum_val = 0.0f;
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++) {
            float pixel = input[block_id * N*N + x*N + y];
            sum_val += pixel *
                       cos((float)M_PI*(2*x+1)*u/(2*N)) *
                       cos((float)M_PI*(2*y+1)*v/(2*N));
        }
    }
    float alpha_u = (u == 0) ? sqrt(1.0f/N) : sqrt(2.0f/N);
    float alpha_v = (v == 0) ? sqrt(1.0f/N) : sqrt(2.0f/N);
    float dct_coeff = alpha_u * alpha_v * sum_val;
    output[block_id * N*N + u*N + v] = round(dct_coeff / Q[u*N + v]);
}
```

Figure 23: DCT implemented on the kernel code for the GPU

Once the DCT is implemented, as previously commented, the **quantization** starts. This process consists on reducing the precision of transformed coefficients to allow efficient compression. It typically uses a standard 8×8 matrix but can be generalized to other sizes as demonstrated down below:

- **Quantization with 8×8 Blocks:** Using the standard 8×8 quantization matrix, for example on a 64×64 block 2D DCT by simply repeating it is not ideal. The original matrix was specifically designed for small blocks, where it preserves low-frequency components with high precision and aggressively compresses higher frequencies. However, when applying a 64×64 DCT, the number of frequency components increases drastically, from 64 to 4096. If the 8×8 matrix is simply tiled across the larger block, the resulting quantization does not reflect the true frequency distribution of the larger transform.

  A more effective approach is to first scale the original 8×8 matrix to better reflect the wider range of frequencies present in a 64×64 DCT. Only after this scaling should the matrix be extended to match the block size. This ensures that quantization adjusts appropriately to the spatial frequency distribution of the larger block, maintaining a good balance between compression and image quality. The code can be seen in the following figure 24

```
Q_Y = np.array([
    [16,11,10,16,24,40,51,61],
    [12,12,14,19,26,58,60,55],
    [14,13,16,24,40,57,69,56],
    [14,17,22,29,51,87,80,62],
    [18,22,37,56,68,109,103,77],
    [24,35,55,64,81,104,113,92],
    [49,64,78,87,103,121,120,101],
    [72,92,95,98,112,100,103,99]
])

Q_C = np.array([
    [17,18,24,47,99,99,99,99],
    [18,21,26,66,99,99,99,99],
    [24,26,56,99,99,99,99,99],
    [47,66,99,99,99,99,99,99],
    [99,99,99,99,99,99,99,99],
    [99,99,99,99,99,99,99,99],
    [99,99,99,99,99,99,99,99],
    [99,99,99,99,99,99,99,99]
])

# Step 3-5: Compress each channel (DCT + Quantization + Reconstruction)
Y_compressed  = process_channel(Y_padded,  Q_Y, C)
Cb_compressed = process_channel(Cb_padded, Q_C, C)
Cr_compressed = process_channel(Cr_padded, Q_C, C)
```

Figure 24: Quantization with 8×8 blocks.

- **Quantization with $N \times M$ Blocks:** The extension code for the 8×8 quantization matries takes de standard JPEG 8×8 matrices ficrst as the variable Q. This extension is based on two functions:

  - `scale_quant_matrix`**:** This function adjusts the quantization matrix based on a user-defined quality level, which ranges from 1 (low quality, high compression) to 100 (high quality, low compression). It calculates a scale factor and modifies each value in the matrix accordingly, ensuring the result stays within the 8-bit range of 1 to 255 using the clip function. When the quality is set to 50, the matrix remains largely unchanged. Lower quality values increase the matrix values, leading to more aggressive compression and a greater loss of detail. Higher quality values reduce the matrix entries, preserving more detail but resulting in larger file sizes.
  - `upscale_quant_matrix`**:** This function extends the 8×8 quantization matrix to match the size of the entire image by repeating the pattern horizontally and vertically. This makes it possible to apply quantization across the full image using efficient, vectorized

operations. The function first calculates how many repetitions are needed along each axis, then tiles the matrix accordingly, and finally crops the result to fit the image dimensions exactly. This approach keeps the JPEG standard's block-wise structure intact while simplifying the implementation.

The code can be seen in the following figure 25:

```python
def scale_quant_matrix(Q, quality):
    quality = max(1, min(100, quality))
    if quality < 50:
        scale = 5000 / quality
    else:
        scale = 200 - quality * 2
    return np.clip((Q * scale + 50) / 100, 1, 255)

def upscale_quant_matrix(Q, shape):
    reps = (shape[0] // Q.shape[0] + 1, shape[1] // Q.shape[1] + 1)
    Q_big = np.tile(Q, reps)
    return Q_big[:shape[0], :shape[1]]
```

Figure 25: Quantization with arbitrary block sizes ($N \times M$)

2.v Execution time of the methods for two image sizes

Before analyzing the performance of each method,a benchmarking script was developed to automate the testing process across all implementations. The script executes each method multiple times, collects the execution time, and computes statistical metrics such as the mean, variance, and standard deviation. The first part of the benchmark script, shown in figure 26, defines a list of commands corresponding to various compression implementations. These commands span multiple backends—Numba (JIT), NumPy (vectorized), OpenCL (GPU), and MPI (CPU parallel)—and apply them to two image formats: NEF (RAW) and PNG. For MPI-based methods, the list includes multiple entries to evaluate performance scaling from 1 to 8 processes.

```
# List of implementations
implementaciones = [
    ("JIT Numba (NEF)", "python scripts/jit_raw.py"),
    ("JIT Numba (PNG)", "python scripts/jit_png.py"),
    ("Numpy Vectorized (NEF)", "python scripts/numpy_vectorized_raw.py"),
    ("Numpy Vectorized (PNG)", "python scripts/numpy_vectorized_png.py"),
    ("OpenCL (NEF)", "python scripts/opencl_raw.py"),
    ("OpenCL (PNG)", "python scripts/opencl_png.py"),
    ("MPI (NEF)", "mpiexec -n 1 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 1 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 2 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 2 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 3 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 3 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 4 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 4 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 5 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 5 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 6 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 6 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 7 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 7 python scripts/mpi_png.py"),
    ("MPI (NEF)", "mpiexec -n 8 python scripts/mpi_raw.py"),
    ("MPI (PNG)", "mpiexec -n 8 python scripts/mpi_png.py"),
]
```

Figure 26: Python benchmark script: list of implementations and corresponding commands to execute.

Figure 27 illustrates the second part of the script, where each entry from the list is executed sequentially. The execution time is measured and stored, enabling a systematic comparison of the different approaches.

```python
def run_command(cmd):
    start = time.perf_counter()
    try:
        subprocess.run(cmd, shell=True, check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    except subprocess.CalledProcessError as e:
        print(f"❌ Error in: {cmd}\n{e.stderr.decode()}")
        return None
    end = time.perf_counter()
    return end - start

def benchmark_command(name, cmd, reps=5):
    times = []
    print(f"\n🏁 Benchmarking: {name}")
    for i in range(reps):
        print(f"  → Execution {i+1}/{reps}...", end="")
        t = run_command(cmd)
        if t is not None:
            times.append(t)
            print(f" {t:.4f} s")
        else:
            print(" Error")
    times = np.array(times)
    return {
        'mean': np.mean(times),
        'standard deviation': np.std(times),
        'variance': np.var(times),
        'repetitions': reps,
        'all repetitions': times.tolist()
    }

def main():
    reps = 20  # number of repetitions per implementation
    results = {}

    for name, command in implementaciones:
        results[name] = benchmark_command(name, command, reps)
    print("\n📊 FINAL RESULTS:")
    for name, stats in results.items():
        print(f"\n[{name}]")
        for key, value in stats.items():
            if isinstance(value, float):
                print(f"  {key}: {value:.6f} s")
            elif isinstance(value, list):
                print(f"  {key}: {[round(v, 4) for v in value]}")
            else:
                print(f"  {key}: {value}")

if __name__ == "__main__":
    main()
```

Figure 27: Python benchmark script: execution loop that measures and records the performance of each implementation.

Each benchmark was executed 20 times per method for the two different images. The execution time is reported down below in tables 9 and 10, representing the average value computed from those 20 runs for both PNG and NEF images.

- **Image (PNG) [2252 × 1600 pixels]:**
  The first test was conducted on a large image in PNG format with dimensions of 2252 × 1600 pixels. The average execution times (in seconds) for the selected methods are:

Method	Execution Time (s)
JIT Numba	26.49
Numpy Vectorized	3.72
OpenCL (GPU)	1.19
MPI	1.73

Table 9: Execution times for different methods on a large PNG image (2252×1600).

- **Very Large Image (NEF) [6034 × 4028 pixels]:**
  A second test was carried out using a very large image in NEF format with dimensions of 6034 × 4028 pixels. This image stresses computational resources significantly more than the PNG case.

Method	**Execution Time (s)**
JIT Numba	231.07
Numpy Vectorized	21.30
OpenCL (GPU)	6.47
MPI	22.65

Table 10: Execution times for different methods on a very large NEF image (6034×4028).

To further analyze the performance of MPI-based parallel implementation it was done the same test for different number of processors, going from 1 to 8, which was summarized in table 11.

Cores	**PNG Image Time (s) [2252×1600]**	**NEF Image Time (s) [6034×4028]**
1	4.908	27.15
2	3.189	16.39
3	2.753	13.906
4	2.636	12.432
5	2.641	12.037
6	2.75	12.134
7	2.708	12.5
8	2.872	13.084

Table 11: Mean execution time of MPI method using 1 to 8 cores, repeated 20 times per case.

Results show that the increase in the number of core in MPI implementation reduces the execution time significantly, especially between 1 and 4 core. However, beyond 4 core, performance improves margins and in some cases slightly low. For example, on 8 core, execution time increases slightly compared to using 5-7 core. This trend indicates that parallelization begins to beat the benefits of adding overhead and inter-process communication more core. In particular, for both image sizes, the most optimal number of processes appears to be 5 core, which attains the lowest performance time for NEF image (12.037S), and one of the lowest for PNG image (2.641s). These findings suggest that, under the current charge and hardware setup, using more than 5 processes reduces the return. Additionally, large NEF image is more beneficial than equality, which confirms the scalability benefit of MPI with increasing data size - up to an optimal point.

2.vi Quality of the reconstructed image

In this section, it is evaluated the visual quality of the reconstructed images after applying the 4 different processes, comparing the raw image shown in figure 28 and its compression output in figure 29. The rank is assigned by visual inspection and goes from 0 to 5.

Figure 28: Original image (.NEF converted).

Figure 29: Compressed JPEG image.

To better appreciate the quality loss due to compression, especially in areas with more detail and sharp edges, a zoomed-in view of both images is shown in figures 30 and 31.

Figure 30: Zoomed-in original image.

Figure 31: Zoomed-in compressed JPEG image.

Upon closer inspection of the zoomed-in images, several compression factors introduced by the DCT algorithm become apparent. In particular, the fine details of the wheel and the characters present in the original image30 are noticeably lost in the compressed image31, and a slight increase in visual noise is evident with darkening of the image. Moreover, the JPEG version shows a reduction in both brightness and color compared to the original, resulting in an overall worst appearance. These degradations are typical of lossy compression and highlight the trade-off between file size and image quality. It probably has a 4 in the scale.

2.vii Theoretical Speedup with MPI on a Distributed Cluster

Assume a hypothetical distributed memory system where nodes communicate over wired links with a data rate of $R = 200\,\text{Mbps}$. For this example it is assumed that there will be a total of 16 computers working as processors, with the master processor also been able to compute calculations and not just gathering and organizing the data. The data will be assumed to be computed sequentially for being able to make easier assumptions in the calculus section.

Transmitting a variable of size $w \in \{8, 32\}$ bits takes

$$t_w = \frac{w}{R} \quad \text{seconds.}$$

Let T_1 be the execution time on a single processor, and T_p the execution time on p processors. In the ideal case (no communication overhead), the maximum speedup is:

$$S_{\text{ideal}}(p) = \frac{T_1}{T_p} = p.$$

However, in practice, communication overhead introduces a performance penalty. In an MPI implementation, each process typically handles a subset of the 8×8 blocks. For operations like the DCT (Step 3) or inverse DCT (Step 5), minimal communication is needed. However, if the image needs to be gathered or scattered across nodes, communication becomes relevant.

Assume a distributed-memory system where communication occurs over wired links at a data rate of $R = 200\,\text{Mbps}$. In such a scenario, the maximum speedup achievable using MPI is analyzed considering both computation and communication. An image of 29.2 MB with size 6034×4028 pixels is considered, with a compression algorithm that works on 8×8 pixel blocks.

- Total uncompressed size: $29.2\,\text{MB} = 29.2 \times 10^6 \times 8 = 233.6 \times 10^6\,\text{bits}$.
- Each block contains 64 floats (32 bits per float): $64 \times 32 = 2048\,\text{bits}$.
- Number of blocks: $N = 233.6 \times 10^6 / 2048 \approx 114063$.
- Communication time per block: $t_{\text{comm}} = 2048/(200 \cdot 10^6) = 10.24\,\mu\text{s}$.

Assuming each processor computes its assigned blocks and sends the results to a master node at the same time, although computing the block in a sequential way, and the master node is also compressing, the total transmitter communication time becomes:

$$T_{\text{comm}} = \frac{N}{p-1} \cdot \left(1 - \frac{1}{p}\right) \cdot t_{\text{comm}}$$

For example, including the master node, with $p = 16$:

$$T_{\text{comm\_tx}} = \frac{114063}{16-1} \cdot \left(1 - \frac{1}{16}\right) \cdot 10.24 \times 10^{-6} \approx 73.1 \times 10^{-3}\,\text{s}$$

For knowing the receiver communication time, the same equation can be used if it is taken into account that the image has been modified to have less bits. In the cases tested, the image was compressed by a factor of 15.8, so the receiving time would be equal to:

$$T_{\text{comm\_rx}} = \frac{T_{\text{comm\_tx}}}{15.8} = \frac{73.1 \times 10^{-3}}{15.8} \approx 4.63 \times 10^{-3}\,\text{s}$$

So the total time that would be spent transporting the data is:

$$T_{\text{total\_comm}} = T_{\text{comm\_tx}} + T_{\text{comm\_rx}} = 73.1 \times 10^{-3} + 4.63 \times 10^{-3} = 77.7 \times 10^{-3}\,\text{s}$$

Although MPI can offer near-linear speedup for highly parallelizable tasks such as the DCT or IDCT on large images, the communication overhead between nodes can, in some scenarios, become a limiting factor. However, in the example analyzed—an image of 29.2 MB divided into 64-bit blocks and processed using 16 nodes—the total communication time remains relatively low compared to the total execution time. It is worth noting that only the transmission phase contributes significantly, since the reception involves already-compressed data. Therefore, in this specific case, communication overhead does not substantially limit performance, although it could become critical in systems with slower interconnects or smaller computation loads.