Politecnico di Torino

System and Device Programming

# P-Way Graph Partitioning

*Authors:*
Canavero Serena
Colella Edoardo
Pavarino Leonardo

December 24, 2023

# Contents

## 1 Introduction

In the field of mathematics, graph partitioning involves the process of reshaping a graph into a smaller version by segregating its nodes into separate and non-overlapping groups. The resulting partitioned graph maintains connections between nodes belonging to different groups, while leaving those within the same group untouched. When the number of these inter-group connections is significantly reduced compared to the original graph, the partitioned version often proves more beneficial for various analytical and problem-solving tasks.

Discovering a partition that simplifies graph analysis is a complex task, but it holds significance in scientific computing, VLSI circuit design, and scheduling tasks on multiprocessor computers, among other domains. Recently, the importance of the graph partition problem has grown due to its application in clustering and identifying cliques in social, pathological, and biological networks. Two classic examples of graph partitioning problems are the minimum cut and maximum cut problems.

Many computer science applications involve the manipulation of large graphs, and frequently, these graphs need to be divided into non-overlapping segments. This process of dividing a graph into "*p*" parts is referred to as a "*p*-way partition."

Consider graphs represented as G = (V, E), where V represents the set of vertices and E represents the set of edges, and they are associated with two weight functions: W1, which assigns real-valued weights to vertices, and W2, which assigns real-valued weights to edges.
A p-way partition of G involves dividing G into p subgraphs in such a way that the vertices in each subset are distinct, and specific criteria are met:

- the total weight of nodes within each subgraph is evenly balanced.
- the total weight of edges connecting the subsets is minimized (i.e. *cut-size* in the following.)

This division of a graph finds applications in various fields, including web search algorithms (such as Page Rank), VLSI circuit placement and routing, social network analysis, and more.

The purpose of our project is to realize an implementation of p-way partitioning by writing a program in C++, in a Unix environment. In our program we tried to implement the algorithm in two modes: the first one executes the code exclusively in a sequential way, while the second one executes several fractions of the algorithm in a parallel way.

# 2 Input file

The file containing the graph to be partitioned is saved in binary and has a well-defined structure. The files used in testing, however, were initially not available in binary but ASCII format, so they were converted with a tool that can be found in "src/tools/ascii_bin_converter.cpp."

## 2.1 File Structure

The binary file has a very simple structure, divided into three parts:

- a header with general information about the graph, such as the number of nodes and edges
- a list of weighted nodes
- a list of weighted edges.

Each piece of data is represented by an unsigned integer on 32 bits, so the maximum number available is $2^{32} - 1 = 4,294,967,295$

### 2.1.1 Header

The first 4 bytes of the file are for the number of nodes in the graph, the next 8 bytes instead for the number of edges. The reason why the maximum size of the list of edges is represented on 8 bytes instead of 4 is because, given N nodes the maximum number of edges occurs when each node is connected to the other N-1 nodes.

In this case the number of edges would be N*(N-1)/2. $(2^{32}) * (2^{32} - 1)/2 \approx 2^{63}$ so it can be represented on 8 bytes.

Knowing from the beginning the size of the two lists will allow more efficient parallelization of reading the graph during import.

### 2.1.2 List of nodes

The list of nodes is defined by the succession of pairs of integers. The first number is an identifier of the node starting from 0, while the second number is its weight.

### 2.1.3 List of edges

The list of edges is defined by the succession of triplets of integers. The first and second numbers are the identifiers of the pair of nodes connected to each other. The third number is the weight of the edge. Since the graph is undirected, it is not possible for there to be another edge with the same pair of nodes written in reverse. At read time, when information about an edge is read, the opposite direction will also be entered.

## 2.2 File Generation

The binary files used in testing, as mentioned earlier, were not available in a ready-made version for use in this program, so it was necessary to write a program capable of converting the original file, in ASCII, into a binary file. The code for such a tool can be found in src/ascii_bin_converter.cpp

## 2.3 File Loading

Large files (on the order of hundreds of MB) can take, if read sequentially, up to several minutes with a medium-performance system. To optimize this program step, a parallel read algorithm has been implemented. The file is read through the UNIX syscall mmap, so that it can be accessed as if it were a normal array of unsigned integers.

At the beginning, the first 12 bytes are read. The first 4 bytes correspond to the number of nodes, represented with an unsigned integer. The other 8 bytes, on the other hand, are the number of edges. Then a static_cast is done to transform two unsigned integers into an unsigned long.

The variable num_thread is assigned with the number of physical threads provided by the processor of the system in use. It is even possible to define a custom number of threads to use.

Several data structures are then initialized, including the graph itself, a thread vector, a barrier initialized with num_threads, and a mutex.

The graph, file pointer, number of nodes, edges and threads are placed in a wrapper data structure of type thread_data.

After these operations, the program generates num_threads threads, which will in parallel access different areas of the file to read from the list of nodes and the list of vertices. Since, as will be seen later, the graph is saved with adjacency lists, it is necessary that all nodes be read first and then, only later, all vertices.

To achieve efficient file loading and prevent slower threads from becoming a bottleneck for the entire execution, a strategy was devised to manage reading in the following manner:

- A counter is defined with a value of zero, indicating the start of a sequence of nodes to be read.
- Each thread, in turn, acquires a mutex on this counter, reads its value, and increments it by a constant called K.
- After releasing the mutex, the thread reads K consecutive nodes and inserts them into the graph.

Once all nodes have been read, the edges are read using a similar technique:

- A counter is defined to track the reading of edges.
- Each thread, in turn, acquires the mutex, reads the counter, and increments it.
- After releasing the mutex, the thread reads the sequence of edges and saves them in a temporary vector.
- At this point, the thread can attempt to acquire a mutex on the graph to insert the edges. If successful, it empties the edge buffer into the graph; otherwise, it restarts its cycle.
- If, at the end of file reading, the thread still has edges in the buffer, it acquires the mutex on the graph and empties the buffer.

Finally, the threads join, the file is unmapped, and closed. This approach ensures a streamlined and synchronized process for reading nodes and edges, optimizing the overall efficiency of the file loading phase.

# 3 Design

The partitioning process of a sequential graph using a specific partitioning algorithm called Multilevel-KL algorithm, which consists of three steps.

The first step involves multiple coarsening iterations, where pairs of adjacent nodes, selected through a specific algorithm, are collapsed into a single node. The new node will have a weight equal to the sum of the weights of its parent nodes. The new node will be connected to the same nodes as its parent nodes, but the weight of the edges will be equal to the sum of the weights of the edges of its parent nodes. The iterations conclude when the obtained graph has a low enough number of nodes to expedite the subsequent partitioning.

The second step is an initial partitioning, which occurs on the last graph generated in the previous step.

The initial partitioning step involves dividing the vertices of a graph into disjoint sets or partitions. The goal is to distribute the vertices in a way that minimizes certain criteria, such as balancing the size of partitions or minimizing the weight of edges between partitions (cut-size).

The quality of the initial partitioning can significantly impact the performance of subsequent refinement steps in the algorithm. A good initial partitioning should ideally satisfy the specific goals of the chosen algorithm, such as achieving a balanced distribution of vertices and minimizing the communication cost between partitions.

The third step consists, finally, of iterations (the same number as the coarsening iterations) of uncoarsening and refinement. Each parent node is assigned to the partition of its child node. Subsequently, a *Kernighan-Lin* refinement iteration is performed to improve the quality of the partitioning. At the end of this phase, the algorithm returns to the originally read graph, where each node is assigned to a partition.

## 3.1 Graph Structure

The graphs on which our testing has been conducted have all been modelled by means of a C struct containing the list of graph nodes and the list of edges.
The graph struct also contains a list of colours that have been assigned to the nodes, used to design the parallel version of the algorithm.
To perform graph manipulation we then employed several support data structures, which will be encountered in the following discussion.

Most dynamically allocated data structures make use of shared_ptr. This has simplified memory management and made the release safer. In the case of cyclic dependencies, such as between nodes and edges, weak_ptr has been employed.

To enhance code readability, the following aliases have been used for data structures with shared_ptr and vectors.

```
using NodePtr = std::shared_ptr<struct Node> ;
using EdgePtr = std::shared_ptr<struct Edge> ;
using GraphPtr =  std::shared_ptr<struct Graph>;
using EdgePtrArr = std::vector<EdgePtr>;
using NodePtrArr = std::vector<NodePtr>;
```

# 4 Implementation

In the subsequent sections, 4.1 and 4.2, we are going to provide both a sequential and a parallel solution to the graph partitioning task:

## 4.1 Sequential algorithm

### 4.1.1 Coarsening

For the coarsening step to benefit graph partitioning, the last graph to be computed must necessarily be small. As a termination condition, therefore, one of the following two eventualities was defined [4] [5]:

- 50 iterations were performed
- `coarsed_g->V() > max(30*req_parts, g->V() / (log2(req_parts)*40))`

The actual coarsening process consists of several steps.

- A new graph is allocated.
- The list of nodes in the graph is sorted in ascending order based on the node weight, and iteration is performed on this list.
- If the node has not been matched already, iteration is performed on the list sorted in descending order based on the weight of outgoing edges from the respective node; otherwise, the next node is processed.
- If neither of the two nodes belonging to the edge is matched, a new node with a weight equal to the sum of the weights of the two nodes is allocated, added to the graph, and assigned as their child node. The two nodes are marked as matched.
- If, after iterating over the list of its edges, a node is still unmatched, a new node with the same weight as the parent node is created, added to the new graph, and assigned as its child node.
- After iterating over all nodes, each edge of the parent nodes is iterated over to recalculate the edges connecting the new coarsened graph. An example can be seen in Fig.(1)

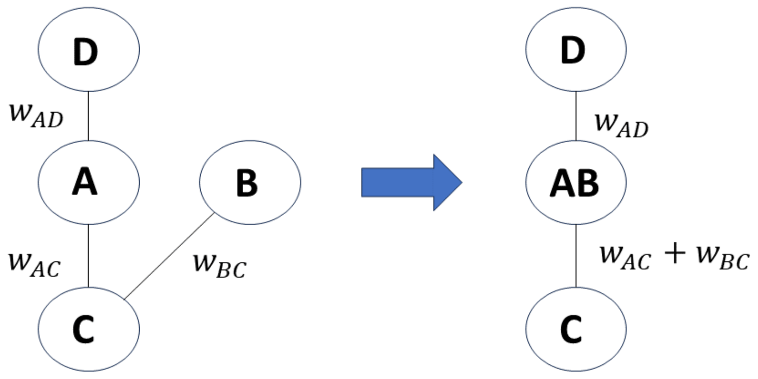


Figure 1: Edge recalculation after node merge.

- Finally, the created graph is returned.

### 4.1.2 Partitioning

Once the graph has been coarsened, we proceed to perform an initial partitioning, this step is important because it will have an influence on the final result.
We choose to implement the *hierarchical clustering algorithm*, as it provides a good starting point for the successive steps of refinement.
The algorithm works by initially assigning each node to a cluster, then couples of adjacent clusters are sorted by decreasing cut-size computed between the couple. This is achieved by inserting each possible couple inside an ordered set, which in our implementation is the std::set provided by the standard library. The couple at the top of the set is then merged into a single cluster and all of its cut sets with the neighbouring clusters are recalculated, thus resulting in a resorted set.
These steps are then repeated until the number of remaining clusters matches the desired number of partitions, at which point each node belonging to a specific cluster is assigned to the same partition.
However searching for elements inside of the set in order to update their relative cut-size, would force us to perform a linear search, so to speed the process up we decided to make use of an additional data structure in the form of an hash map implemented as an adjacency list of neighbouring clusters. In this way we can look up the cut-size among two clusters in constant time and then find the corresponding element inside the set (by providing the couple of clusters and the cut-size read from the hash map) in logarithmic time.
Here is pseudo code of the algorithm based on our specific implementation:

```
for node in graph.nodes do
    partition[node.id] = node.id
end

for edge in graph.edges do
    change c(edge.weight, edge.node1, edge.node2)
    cut_set_set.add(c)
    cut_set_hash[edge.node1][edge.node2] = edge.weight
    cut_set_hash[edge.node2][edge.node1] = edge.weight
end

number_cluster = graph.nodes.size
while (number_cluster > desired_partitions) do
    number_cluster = number_cluster - 1
    selected_fusion = cut_set_set.front()
    cut_set_set.remove(selected)
    for i from 0 to graph.nodes.size do
        if(partition[i] == partition[selected_fusion.node1])
            partition[i] = partition[selected_fusion.node2]
        end
    end
    update_hash_map_and_set(selected_fusion, cut_set_set, cut_set_hash)
end
```

### 4.1.3 Uncoarsening

The uncoarsening phase is a trivial process. It involves iterating through the list of nodes in a graph, starting from the penultimate one generated during the coarsening phase. For each node encountered, its partition is assigned based on the partition of its corresponding child node in the coarser graph. This assignment occurs has a linear complexity, making the procedure efficient and easily manageable.

```
std::vector<unsigned int> uncoarsenGraph(const GraphPtr& g,
                        std::vector<unsigned int> &partitions) {
    std::vector<unsigned int> newPartitions(g->V());

    for (const auto &n : g->nodes)
        newPartitions[n->id] = partitions[n->child->id];

    return newPartitions;
}
```

### 4.1.4 Refinement

This step is performed at each iteration of the uncoarsening phase. This is done to continuously refine the partitions to ensure we are in a local minimum of the cut-size.
The utilized algorithm is the Kernighan Lin, in particular the Fiduccia and Mattheyes version of it [2].

### 4.1.5 Outer Loop

This algorithm is composed of two loops. The external one iterates until no further improvements are performed in the inner loop. At each iteration it generates a list of possible moves of one node to each neighbouring partition along with the *gain* of such move (i.e. the reduction in the overall cut-size that this move would bring about). In fact moving a node to a non neighbouring partition is unlikely going to help reduce the overall cut-size. These moves are modelled as a struct called *change* containing the node, the destination partition and the gain that would be obtained if the move was to be made. Since these changes need to be sorted by the gain, we again make use of a std::set behaving as a priority queue to store them.

### 4.1.6 Inner Loop

In the inner loop, the topmost possible change is extracted (even if it is negative) and it is performed, then the gains for all neighbouring nodes are updated in the set. If the cut-size at the end of the iteration is lower than the cut-size of the current best partitioning we are tracking, the current partitioning is saved as the current best to be restored at the exit from the inner loop.

Saving the current best partitioning is necessary because this algorithm tests whether it is possible to exit from the current minimum or not, also taking into account changes that have a negative gain in hopes that they will make possible subsequent changes leading to a lower minimum.

In fact such negatively weighted moves that reduce the partition quality may lead to later moves that more than compensate for the initial regression. This ability to climb out of local minima is a crucial feature of the KL algorithm [3].

As it stands there is the possibility for the inner loop to continuously generate partitions that were already met in the past, whenever performing a change with negative gain results in a configuration for which the change with the highest possible gain is the opposite of the change that was just performed. To solve this problem we impose that a node can be moved only once per iteration of the outer loop.
Another limitation that we imposed is that a move should be allowed only if it leads to an improvement in the balance of the partitioning, that is if the originating partition weight is greater than the average

weight and the destination partition weight is lower than the average.

The stopping condition for the internal loop has to be considered with care since we want the algorithm to be able to try changes that may lead to an immediate increase of cut-size, but allow for further improvements at a later point, but we don't want it to spend too much time performing such an action. The condition that we settled for is the one suggested by the paper by Gilbert and Zmijewski [2] according to which the total number of successive negative changes performed so far should not exceed *d*; or the total gain from the beginning of this execution of the inner loop should not be lower than -*d*. *d* is defined as the maximum node degree of the graph, that is the maximum number of edges connected to a single node.

```
Until No better partition is discovered
    Best Partition := Current Partition
    Compute all initial gains
    Until Termination criteria reached
        Select vertex to move
        Perform move
        Update gains of all neighbors of moved vertex
        If Current Partition balanced and better than Best Partition Then
            Best Partition := Current Partition
    End Until
    Current Partition := Best Partition
End Until
```

Figure 2: Pseudo Code for Kernighan Lin

## 4.2 Parallel algorithm

### 4.2.1 Coarsening phase

The parallel coarsening procedure is decidedly different from its sequential counterpart and is also much more complex. The termination condition for the coarsening loops remain the same. Initially, the graph is parallel-colored using a modified version of the *Luby algorithm.*[1]

### 4.2.2 Luby algorithm

Each thread takes charge of $V$/num_thread nodes and assigns a random number to each.
Then, it pauses at a barrier and waits for all the other threads.
At this point, for each node, we check whether it has the smallest assigned random number among its neighbors. The situation of equal numbers is also considered; in that case, the node with the higher ID is considered. If positive, its ID is placed in a buffer.
At the end of the checks, each thread will acquire a lock and iterate over its buffer, assigning the nodes the value INT_MAX so that they are no longer considered in subsequent iterations, assign them color, and increment the counter of colored nodes. The last thread to acquire the lock also increments the color.
In the end, each node will have an assigned color.

### 4.2.3 Actual coarsening

Next is the actual coarsening phase.
Iterating over one color at a time, each node of that color tries to match an unmatched neighboring node with the heaviest edge.
It might happen that multiple nodes try to match the same node. The process is as follows: each node tries to force its matching on the chosen node. When the iterations complete, the nodes check if the matches are still valid. If positive, the match occurs, and the new node is added to the coarsened graph. Otherwise, the node remains unmatched, and its child node will have the same weight.
Finally, in the same way as the sequential version 1, iterating over the edges it calculates the edges connecting the nodes in the generated graph.

### 4.2.4 Partitioning

To partition the graph we are making use of a *random walk algorithm* which selects the $p$ heaviest nodes, as these will be the starting points of our partitions. We assign each of these nodes to a different partition ranging from 0 to $p - 1$. We then assign these nodes to the num_thread threads using a *round robin* assignment to try and maximize load balancing between the threads.
To allow for synchronization among threads we use a list of locks, one for each node.
Each thread will iterate on all the nodes assigned to it. The thread will begin by adding all the neighbours of the current starting node to a std::set sorted by weight. The thread will then iterate over this set:

- first it will acquire the lock on the starting node and on the current node inside the set at the same time to avoid deadlocks
- it will check if the current node has not been assigned yet to a partition and that it is not too heavy to create an imbalanced partition. If it does not satisfy these conditions the locks will be released and the thread will try the next node in the set.
- if it satisfies the conditions it will then remove the current node from the set and assign to it the same partition of the starting node.
- Finally it adds all of its neighbours to the set.

We check if it is possible to add a node by enforcing that the resulting partition weight must not exceed 1.4 times the average partition weight. This particular value was chosen because it fits inside a range of reasonable values.

When the end of the set is reached, the thread moves on to the next node in its initially assigned list. Once it has finished this for all its nodes it returns.
The main thread will wait for all threads to terminate and then will check that all nodes have been assigned to a partition, and if this is not the case it will assign them to the nearest partition that has the lowest partition weight.

### 4.2.5 Uncoarsening

The parallel uncoarsening version is analogous to its sequential version. To improve performance, n threads are instantiated that will be responsible for assigning the parent nodes the same partition value as the child node.

```
void uncoarsen_graph_step(const GraphPtr& g, std::vector<unsigned int> &partitions,
                          std::vector<unsigned int> &newPartitions,
                          int num_nodes, int start, int step) {
    int i = start;
    while (i < num_nodes) {
        newPartitions[g->nodes[i]->id] = partitions[g->nodes[i]->child->id];
        i += step;
    }
}

std::vector<unsigned int> uncoarsen_graph_p(const GraphPtr& g,
    std::vector<unsigned int> &partitions, int num_thread) {
    unsigned int num_nodes = g->V();
    std::vector<unsigned int> newPartitions(num_nodes);
    std::vector<std::thread> threads;

    for (int i = 0; i < num_thread; i++)
        threads.emplace_back(uncoarsen_graph_step, ref(g),
            ref(partitions), ref(newPartitions), num_nodes, i, num_thread);

    for (auto &t : threads)
        t.join();

    return newPartitions;
}
```

Although the operation has linear complexity, in situations of large graphs (on the order of millions of nodes), the operation leads to a lower execution time anyway.

### 4.2.6 Refinement

The refinement is performed by a modified version of the *Kernighan Lin* algorithm. The nodes get randomly assigned to the threads, the randomness guaranties that at least a node per color is assigned to each thread.
The thread then iterate over each possible color, the use of a barrier ensures that all threads are working on the same color at the same time.
The thread will then calculate the weight of each partition and store it in a local variable, and a second barrier ensures that all threads have completed before proceeding to the next step.
The threads will try to perform a very similar algorithm to the one used for the sequential part however they will only consider eligible to be moved only the nodes of the selected color. Another difference with the sequential algorithm is that this parallel version does not consider changes that have a negative gain,

thus making it impossible for this version to leave a local minima.
This continues until changes are performed, once a thread no longer performs a change, it terminates.
The condition to enforce balanced partition is the same that was used for the sequential version of this algorithm, that is to say that a movement is only allowed if the source partition weight is greater than the average partition weight and the destination partition weight is less than the average partition weight.

# 5 Experimental Results

To assess the quality and efficiency of the written code, we designed a test suite capable of providing us with all the relevant statistics.

## 5.1 Setting the Environment

The testing environment used was an Apple MacBook Air with an M1 processor and 8GB of RAM. Three graphs of variable sizes were used as input files:

- Graph-A with 264,346 nodes and 366,923 edges
- Graph-B with 1,070,376 nodes and 1,356,399 edges
- Graph-C with 6,262,104 nodes and 7,624,073 edges

For each graph, multiple iterations of reading, sequential partitioning, and parallel partitioning were performed. A consistent requirement was set for 100 partitions in all cases. In the case of reading and parallel partitioning, 100 iterations were conducted, varying the number of threads used from 1 to 8. For sequential partitioning, only 100 iterations with a single thread were performed.

For reading, the obtained data represents the average execution time as the number of threads used varied. Regarding partitioning, the obtained data includes the average of:

- Execution time
- Average weight of obtained partitions
- Weight of the smallest obtained partition
- Weight of the largest obtained partition
- Weight of the median partition
- Standard deviation
- Cutsize
- RAM used during the execution

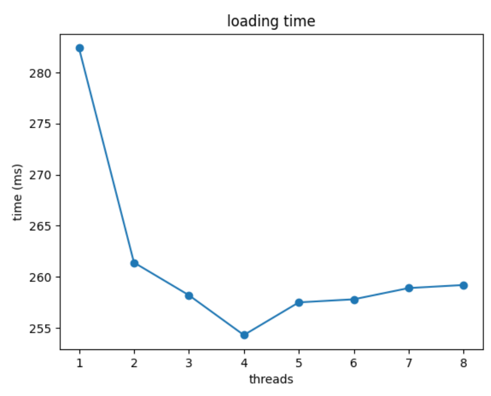
For parallel partitioning, the obtained data is the same as in sequential partitioning, but expressed as the number of threads used varies.

To test for the memory used at runtime we make use of the time tool that is exists for unix systems; utilizing the verbose option this tool is able to give us many information about the execution of the the program, such as the maximum resident set that was used by the program during its execution. The command line that was actually used is the following:

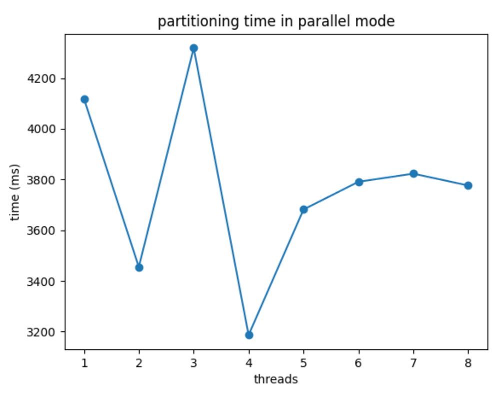
```
time -v <command>
```

This command gives as output a lot of different statistics, the output was parsed to extract only information on the maximum memory used

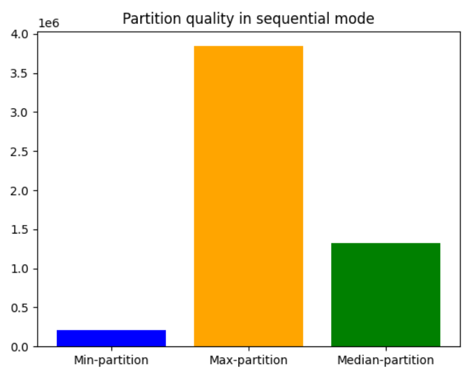
## 5.2 Graph-A



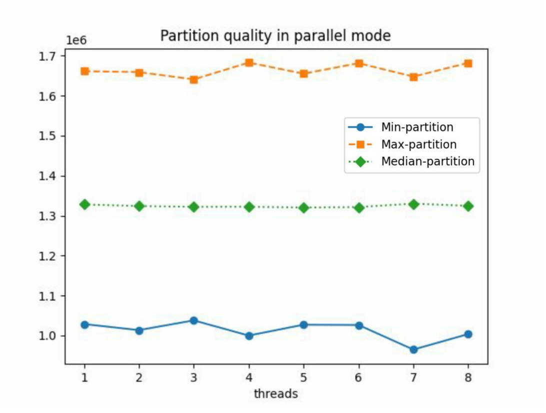
(a) Loading time



(b) Partitioning time in parallel mode



(c) Partition quality in sequential mode



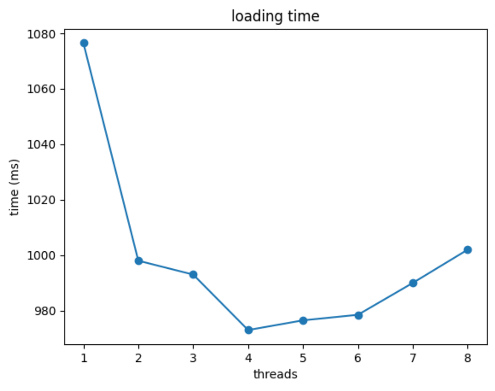
(d) Partition quality in parallel mode

Figure 3: Test results in Graph-A

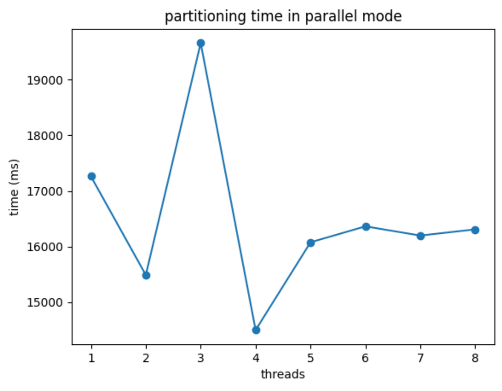
| **Property** | **Sequential** | **Parallel (best)** | **diff** |
|---|---|---|---|
| Execution Time [ms] | 5684.3 | 4318.6 | −24.02% |
| Min-Partition | 210 717.0 | 4318.6 | +392.48% |
| Max-Partition | 3 843 322.0 | 1 640 876.6 | −57.30% |
| Median-Partition | 1 323 592.0 | 1 321 916.1 | −0.12% |
| Std-Deviation | 454 212.0 | 123 501.3 | −72.81% |
| Cutsize | 3 941 520.0 | 13 492 930 | +242.33% |
| Memory [MB] | 185.46 | 189.38 | +2.11% |

Table 1: Comparison of results of the two algorithms on Graph-A

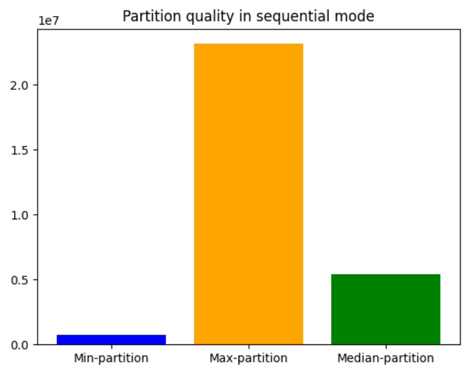
## 5.3 Graph-B



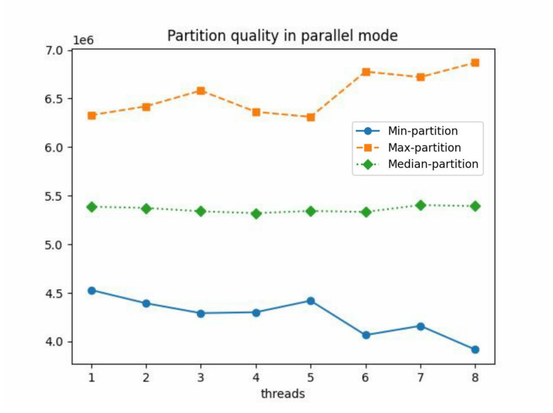
(a) Loading time



(b) Partitioning time in parallel mode



(c) Partition quality in sequential mode



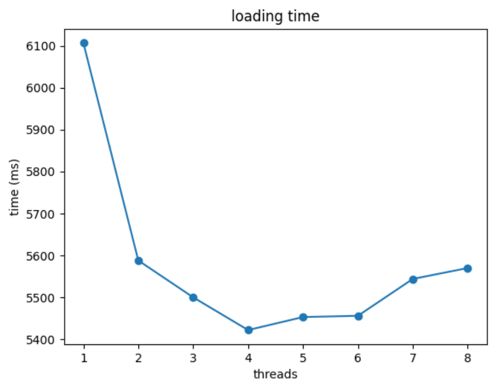
(d) Partition quality in parallel mode

Figure 4: Test results in Graph-B

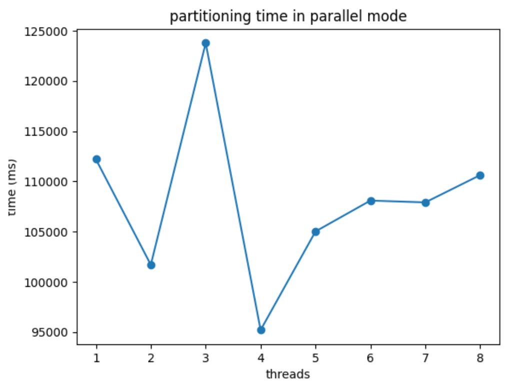
| **Property** | **Sequential** | **Parallel (best)** | **diff** |
|---|---|---|---|
| Execution Time [ms] | 18 582.5 | 16 078.5 | −13.47% |
| Min-Partition | 712 166 | 4 420 187.5 | +520.67% |
| Max-Partition | 23 174 031 | 6 310 612.5 | −72.77% |
| Median-Partition | 5 379 490.5 | 5 343 682.5 | −0.66% |
| Std-Deviation | 2 971 352.5 | 386 463 | −86.99% |
| Cutsize | 6 300 402.0 | 25 702 963 | +307.96% |
| Memory [MB] | 692.15 | 717.11 | +3.61% |

Table 2: Comparison of results of the two algorithms on Graph-B

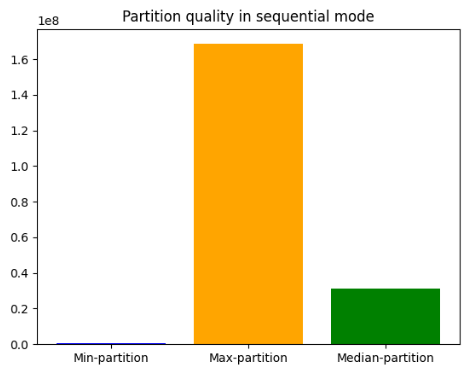
## 5.4 Graph-C



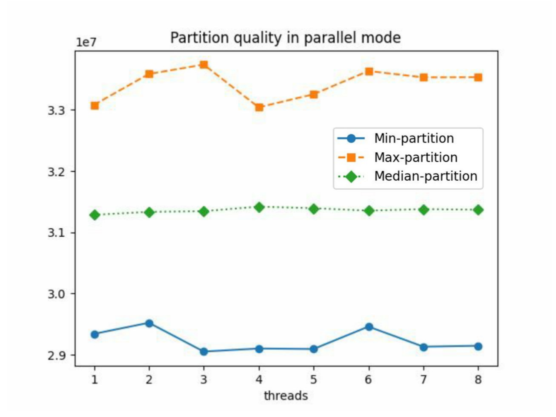
(a) Loading time



(b) Partitioning time in parallel mode



(c) Partition quality in sequential mode



(d) Partition quality in parallel mode

Figure 5: Test results in Graph-C

| **Property** | **Sequential** | **Parallel (best)** | **diff** |
|---|---|---|---|
| Execution Time [ms] | 192 554 | 95 204 | −50.56% |
| Min-Partition | 538 372 | 29 102 300 | +5305.61% |
| Max-Partition | 168 603 071 | 33 040 493 | −80.40% |
| Median-Partition | 31 351 920 | 31 419 306 | 0.21% |
| Std-Deviation | 24 222 666 | 933 865 | −96.14% |
| Cutsize | 50 962 668 | 242 328 854 | +375.50% |
| Memory [MB] | 3971.56 | 4117.30 | +3.67 % |

Table 3: Comparison of results of the two algorithms on Graph-C

## 5.5 Memory tests

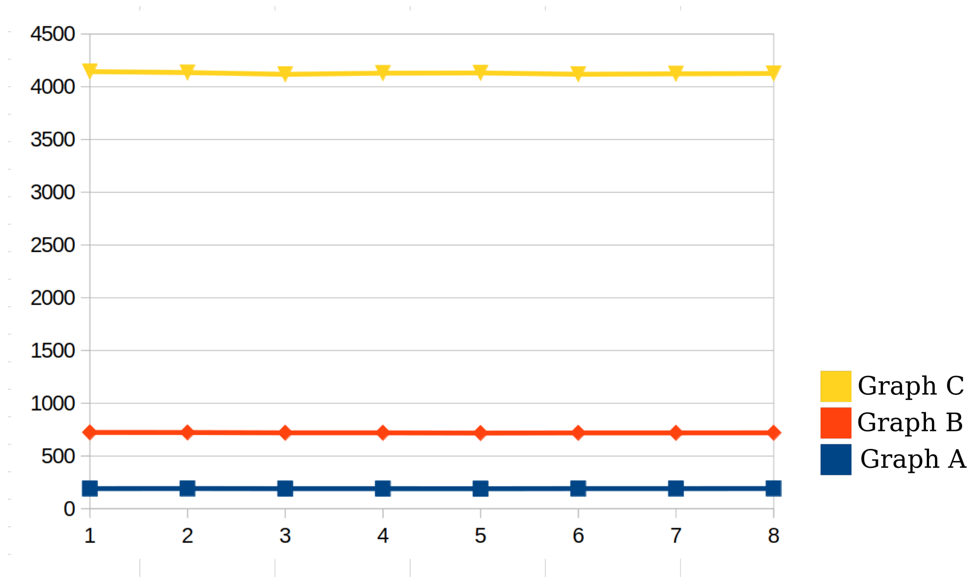


Figure 6: Memory used in MB by the different graphs depending on threads.

## 5.6 Conclusions

The experimental data highlights that the optimal number of threads to achieve minimal execution time, both for reading (3a, 4a, 5a) and parallel partitioning (3b, 4b, 5b), is 4.

As the number of threads increases, performance tends to degrade due to the overhead of creating and managing a larger quantity of threads. This, combined with increased wait times due to synchronization, fails to compensate for the enhanced parallelization.

Comparing the two algorithms (1, 2, 3), parallel partitioning generally proves to be faster in execution and yields more balanced partitions. Simultaneously, the sequential algorithm significantly reduces the graph's cutsize.

The memory necessary to use the parallel version version is slightly higher in the parallel version, as additional data structures are needed to allow for parallelization, such as the one to keep track of the color of each node, and to allow synchronization among the threads. However the memory used in the parallel version does not vary much for different number of threads (6). This indicates that the data structures that allow for parallelization are responsible for the increased memory need, while the impact from those that allow for synchronization is almost negligible.

These insights provide a nuanced understanding of the trade-offs between parallel and sequential approaches, offering valuable guidance for optimizing code performance and achieving well-balanced graph partitions. Further research and optimization strategies may be explored to enhance the overall efficiency and scalability of the implemented algorithms.

## References

[1] Michael Luby. "A simple parallel algorithm for the maximal independent set problem". In: *SIAM Journal* (1986).

[2] John R. Gilbert and Earl Zmijewski. "A parallel graph partitioning algorithm for a message-passing multiprocessor". In: *International Journal of Parallel Programming* 16.6 (Dec. 1987), pp. 427–449. ISSN: 1573-7640. DOI: `10.1007/BF01388998`. URL: `https://doi.org/10.1007/BF01388998`.

[3] Bruce Hendrickson and Robert Leland. "A Multi-Level Algorithm For Partitioning Graphs". In: Feb. 1995, pp. 28–28. ISBN: 0-89791-816-9. DOI: `10.1109/SUPERC.1995.242799`.

[4] G. Karypis and V. Kumar. "Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs". In: *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 1996, pp. 35–35.

[5] Karypis. *METIS - Serial Graph Partitioning and Fill-reducing Matrix Orderingn*. `https://github.com/KarypisLab/METIS`. Accessed: 2023-11-28.