P-way graph partitioning

Canavero Serena
Colella Edoardo
Pavarino Leonardo

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Introduction

In computer science, large graphs often require non-overlapping segmentation, known as "p-way partitioning."

This involves dividing a graph into "p" parts, balancing node weights and minimizing edge weights between subsets. Applications span web search algorithms, VLSI circuit placement, social network analysis, etc.

The project aims to implement p-way partitioning in C++ on a Unix platform. Two implementations are considered: a sequential execution and a parallel execution.

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Design - Graph struct

The graph struct contains three main attributes:

- List of nodes, which have as attributes:
  - Their weight
  - The list of edges they are connected to
- List of edges, which have as attributes:
  - Their weight
  - The two nodes they connect
- List of colors of the nodes.

We make use of shared_ptr to guarantee safe memory management.

P-way graph partitioning

Design

The Multilevel-KL algorithm was chosen to partition the graph, because researches showed it was effective.
It consists of three phases:

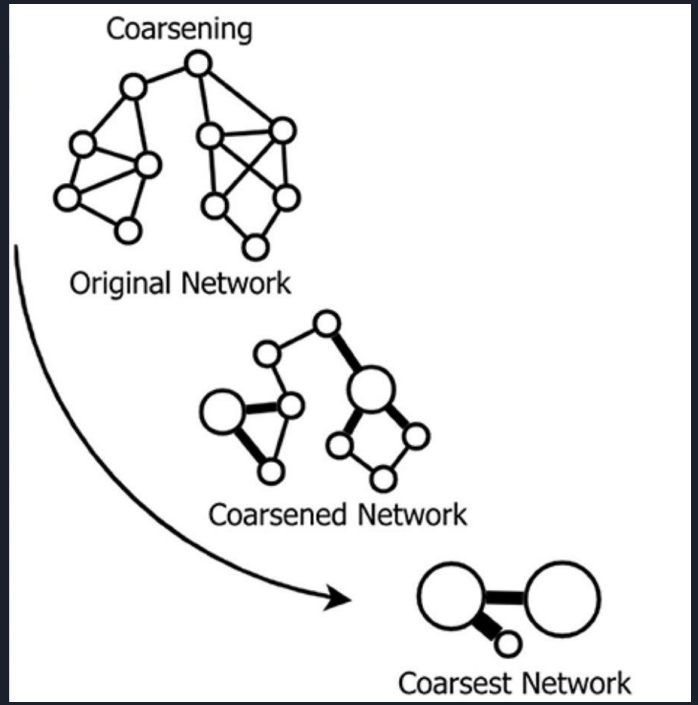
- coarsening
- initial partitioning
- uncoarsening and refinement

P-way graph partitioning

Design - coarsening

The coarsening phase consists of several stages of graph reduction.

During each iteration, pairs of nodes are collapsed, according to a given criterion, and their properties summed.



P-way graph partitioning

Design - initial partitioning

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).

P-way graph partitioning

Design - initial partitioning

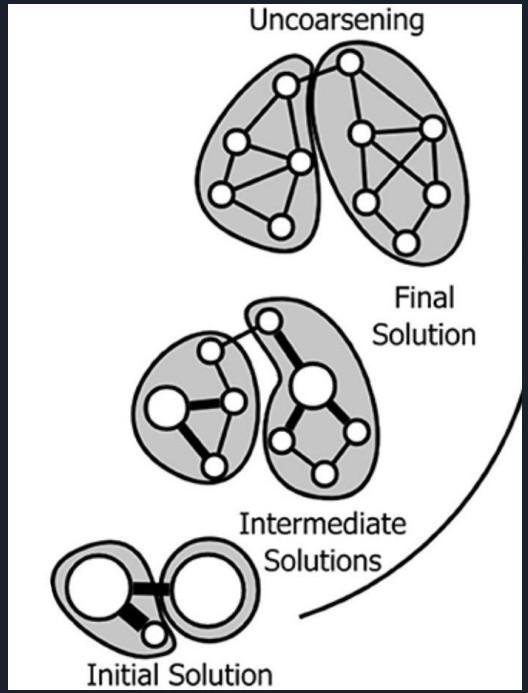
The quality of the initial partitioning can significantly impact the performance of subsequent refinement steps in the algorithm.

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.

P-way graph partitioning

Design - uncoarsening

Graph uncoarsening is a process that reverses the coarsening of a graph.
While coarsening reduces the size of the original graph by grouping nodes together, uncoarsening attempts to reconstruct the original graph from a smaller or coarsened version, preserving the identified partitions.



P-way graph partitioning

Design - refinement

The local refinement should improve the quality of the partitioning after each uncoarsening step by moving border nodes in a different partition if this would improve the per the partitioning quality.

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

File loading

The file to be read is a binary file consisting of
- a header
- a list of nodes
- a list of edges

The file is mapped to memory with the syscall mmap, so that it can be read easily.
The program instantiates N threads that will concurrently read the file.

P-way graph partitioning

File loading

To prevent slow threads from worsening performance, each thread acquires a mutex on a counter indicating the index of the next node to be read, copies it, and increments it by a constant value.

Faster threads will perform this action several times.
Nodes are read and added to the graph.
Edges are read in a similar way.

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Sequential partitioning - Coarsening

In this version, nodes are collapsed using a modified heavy edge matching (HEM) approach.
Nodes are sorted by descending weight, then the heaviest edge is taken for each.

At this point, the two nodes connected by the found edge are collapsed.

Finally, all the nodes obtained are added to the new graph and their distances recalculated.

P-way graph partitioning

Sequential partitioning - Initial partitioning

It is implemented using a hierarchical clustering algorithm:

- each node is assigned to a cluster
- couples of adjacent clusters are sorted by decreasing cut-size
- they are merged following the list ordering until their number matches the desired partitions; after each merge the cutsizes are updated and so is the sorting in the list.

P-way graph partitioning

Sequential partitioning - Initial partitioning

```
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
```

P-way graph partitioning

Sequential partitioning - Initial partitioning

Searching for elements inside of the set in order to update their relative cut-size, would force us to perform a **linear search**.

To speed the process up we make use of an additional **hash map** implemented as an adjacency list of neighbouring clusters.

P-way graph partitioning

Sequential partitioning - Initial partitioning

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**.

P-way graph partitioning

Sequential partitioning - Uncoarsening

The uncoarsening phase 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.

P-way graph partitioning

Sequential partitioning - Refinement

This step is performed after each uncoarsening step. We use the Kernighan Lin in the Fiduccia and Mattheyes version.

The algorithm is composed of two loops.

```
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
```

P-way graph partitioning

Sequential partitioning - Refinement - Outer Loop

The outer loop iterates until no further improvements are registered 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; the list is ordered by descending gain.

The move is modelled by a *struct change* containing:

- the node
- the destination partition
- the gain the move would bring about

*gain : the reduction in the overall cut-size induced by a move

P-way graph partitioning

Sequential partitioning - Refinement - Inner Loop

- the topmost move is extracted and performed
- then the gains for all neighbouring nodes are updated

If the cut-size at the end of the iteration is lower than that of the current best partitioning, upon exiting this loop it becomes the current best.

P-way graph partitioning

Sequential partitioning - Refinement

The move with higher gain is selected even if it is negative; this is motivated by the fact that several moves that reduce the partition quality may lead to later moves that allow us to reach a lower local minimum.

This ability to climb out of local minima is a crucial feature of the KL algorithm.

P-way graph partitioning

Sequential partitioning - Refinement - Limitations

- To avoid infinite iterations of the inner loop we impose that a node may be moved only once per outer loop iteration

- A move should be allowed only if it leads to an improvement in the balance of the partitioning, i.e. if originating partition weight > average weight and destination partition weight < average weight

P-way graph partitioning

Sequential partitioning - Refinement - stop criterion

A stopping criterion such as this is needed since we want to allow for changes that may lead to subsequent improvements and at the same time limit the time requires.

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Parallel partitioning - Coarsening

At first, a graph colouring phase is executed, based on a modified version of Luby's algorithm. It will be useful even in next phases.

Then, in a parallel way, nodes are iterated over one color at a time and each node of the selected color attempts to match with the heaviest unmatched neighbor node.
There is a possibility of multiple nodes trying to match with the same target node. In such cases, the last node to set the connection get matched and the new generated node becomes part of the coarsened graph.

In the final step, the algorithm iterates over the edges to calculate the connections between nodes in the coarsed graph.

P-way graph partitioning

Parallel partitioning - Partitioning

The Random Walk algorithm was employed:

- the p heaviest nodes are selected as the starting points of our partitions
- each node is assigned to a different partition ranging from 0 to p-1
- the partitions are assigned to the num_thread threads using a round robin distribution to maximize load balancing
- each node is assigned a lock

P-way graph partitioning

Parallel partitioning - Partitioning

The Random Walk algorithm was employed:

- each thread iterates on the nodes assigned to it
- it starts by adding all the neighbours of the current starting node to a set sorted by weight and then iterates over this set:
  - it will acquire the lock on the starting node and on the current node inside the set at the same
  - time to avoid deadlocks

P-way graph partitioning

Parallel partitioning - 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.

P-way graph partitioning

Parallel partitioning - Refinement

The refinement is performed by a modified version of the Kernighan Lin algorithm.

- the nodes are randomly assigned to the threads (at least one node per color is assigned to a thread)
- each thread then iterates over each color (a barrier ensures all threads are working on one same color at a time)

P-way graph partitioning

Parallel partitioning - Refinement

- each thread then calculates the weight of each partition
- a second barrier ensures all threads finish the previous step before proceeding

Differences wrt sequential version:

- only nodes eligible to be moved are the ones of the selected color
- changes with negative gain are not considered

P-way graph partitioning

Parallel partitioning - Refinement

Ending:

- continues until no change is performed i.e. when all threads terminate

Balanced partitions:

- a movement is only allowed if the source partition weight is higher the average partition weight and the destination partition weight is lower the average partition weight

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Testing

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

P-way graph partitioning

Testing

For each graph, multiple iterations of reading, sequential partitioning, and parallel partitioning were performed. The obtained data includes:

- 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

P-way graph partitioning

Testing

To allow for the testing of the memory used we used the time command to measure the memory used during its execution.

time -v <command>

P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Experimental result

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%

P-way graph partitioning

Experimental result

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%

P-way graph partitioning

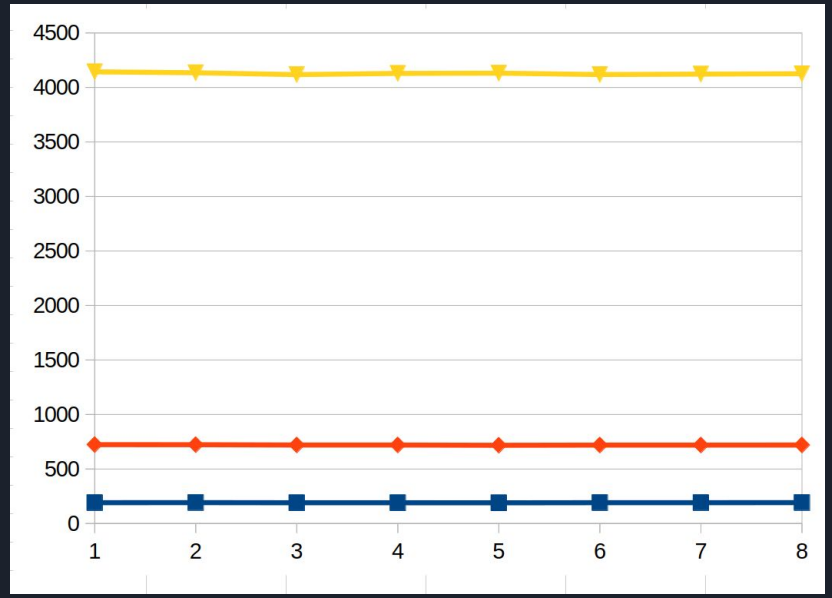
Experimental result

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 %

P-way graph partitioning

Experimental result

The parallel version requires more memory. But the number of threads used does not have much impact on the memory used.



P-way graph partitioning

- Introduction
- Design
- File Loading
- Sequential partitioning
- Parallel partitioning
- Testing
- Experimental results
- Conclusions

P-way graph partitioning

Conclusions

From the data it is visible that the optimal number of threads on the testing machine is between 4 and 5, after which the performance starts to degrade due to synchronization and management overheads.

The parallel version is faster and deliver more balanced partitions, while the sequential version greatly reduces the cut-size.
In general the parallel version requires more memory as it needs additional data structures.

Thank you for your attention