Graph partitioning project

September 4, 2023

Gerardo Maruotti s317642

Simone Varriale s315962

Table of contents

1. Abstract

This project is part of the System And Device Programming Exam (year 2023, Politecnico di Torino). The goal was to implement at least one sequential and parallel versions of a graph partitioning algorithm. It was requested us firstly to analyse the behaviour of both versions of different algorithms and then compare the results to in terms of execution time, cpu usage and memory usage. The results we have obtained were compared on different benchmark od increasing size and complexity in order to assess the quality of the algorithms.

2. Introduction

The structure of the graph used in out project was based on $G = (V, E)$ where V is the number of nodes of the graph and E is the number of edges of the graph. These graphs have also two weight functions $W_1: V \rightarrow R$ which maps vertices to real-valued weights and $W_2: E \rightarrow R$ which maps edges to real-valued weights. Our goal is to implement algorithms that performs p-way partitioning which is the division of graph G into p sub-graphs in which their vertices do not overlap and two specific properties are satisfied:

- The sum of the weights of the nodes in each subgraph is balanced.
- The sum of the weights of the edges crossing between subsets is minimized.

The implementation of the graph is without coordinates so it is not embedded in space and it is also undirected.

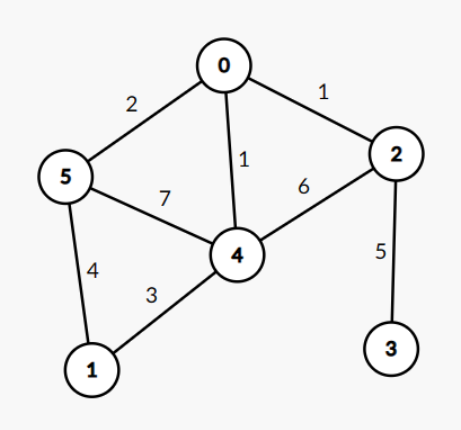

Figure 1: Example of Graph

3. Computing facilities

We have tested all our work on a Macbook Pro 14” with chip Apple M1 Pro which is based on ARM architecture and produced by TSMC with 5nm process production.[1]

- CPU: Apple chip M1 Pro utilizes the "ARM big.LITTLE" design with 8 "Firestorm" core used for high performance (con un clock a 3228 MHz) and 2 "Icestorm" core for energy efficiency (with clock frequency 2064 MHz), so it is a total of 10 core. The 8 high performance core are divided in two cluster. Each cluster shares 12MB of L2 cache, while the two cores for high efficiency share 4MB of L2 cache. At system level the CPU is equipped with 24MB of cache.
- GPU: The M1 Pro integrates a GPU that has up to 16 core designed by Apple. Each GPU core is divided in 16 Execution Units, each of them has 8 ALU.

- NPU: Il SoC is also equipped with a neural process unite (NPU) named "Neural Engine" with 16 core.
- SOC: The M1 Pro has three controls Thunderbolt 4 and media engine which supports encoding and decoding ProRes.
- MEMORY: The chip M1 Pro owns a unified memory architecture of 16 GB, so all the components like CPU and GPU share the same RAM memory.

4. Graph data structure

Our implementation of the graph is characterized by three basic struct:

- Node: int weight, int degree, Coarse* coarse
- Edge: int n1, int n2, int weight
- Coarse: int n1, int n2, int weight1, int weight2, std::vector<std::vector<std::vector<int>>> adj;

Struct coarse contains all the important information for the coarsening phase of multilevel algorithm and it will be explained in details in the following paragraph of the report.

The class Graph is then organized with the following private data structure:

- Nodes: map<int,Node>
- Edges: vector<Edge>
- vector<vector<vector<int>>> MatAdj
- vector<vector<int>>> MatDegree
- int sizeN, int sizeE

We wanted to keep our implementation really simple and straightforward in order to be understood easily by people who did not worked on the project. Then, we added all the utility functions used to populate the graph and to compute all the necessary data structure used by the algorithms.

5. Recursive spectral bisection

Recursive Spectral Bisection was proposed by Pothen et al. for the first time in 1989 [2] in order to implement an efficient way to compute the ordering of a matrix in a parallel way. The idea was based on the fact that the matrix can exceed the storage of a single processor so a good approach to achieve the goal would be the divid and conquer. It uses an adjacency matrix and the Laplacian matrix from which it compute the eigenvectors used to perform partitioning. RSB is not a local algorithm like Kernighan-Lin, but it is called spectral algorithm for three different reasons:

- The spectral methods employs global information to compute separators and it does not uses neighbours like KL algorithm.
- It makes a continuous choice in order to belong to one partition instead of having a discrete choice like KL.
- The dominant computation is an eigenvector computation through Lanczos or similar algorithm.

The Laplacian matrix is defined as follows $L = D - A$ where D is the degree matrix that contains the number of edges of each vertex while A is the adjacency matrix of graph G. Since

the graph is weighted, we actually used the weight of the edge. The spectral properties of L has been studied and it is proven that is positive semidefinite. L is crucial for this algorithm since it captures the structure of the graph and its connectivity. Then, after the computation of L we need to focus our attention on the computation of the eigenvectors of the matrix since the second eigenvector is the most interesting one for us and it is called Fiedler Vector.

In order to obtain the Fiedler vector we need to compute the eigenvalue decomposition of the Laplacian Matrix. Eigenvectors corresponding to smaller eigenvalues capture the connectivity patterns of the graph, with the second eigenvector corresponding to the "Fiedler vector". After that, since the graph is weighted, it is important to compute the median value of the Fiedler vector and then based on the index of the fiedler vector producing the partition. After this operation we obtain two partitions and there is the possibility to perform a refinement on the partition using other algorithm or keep the partition in the same way if some criterion are met.

Advantages

The RSB method leverages spectral properties to capture global structural information, which can lead to effective partitioning. It often produces well-balanced partitions with a relatively small number of cut edges. The algorithm is parallelizable, making it suitable for modern parallel computing architectures.

Limitations

The RSB method's performance can be sensitive to the choice of eigenvector and the stopping criterion. In some cases, the algorithm might struggle with graphs that have irregular structures or complex connectivity patterns. RSB might not be the most suitable option for graphs with specific properties, such as community structures.

Pseudocode

Data: Graph G(V,E), number of partition

Return value: matrix of bool where each row represents a partition

$MatDegree \rightarrow G.getMatDeg()$

$MatAdj \rightarrow G.getMatAdj()$

$L \rightarrow MatDegree - MatAdj$

$Fiedler\ Vec \rightarrow eigenSolver(L).col(1)$

$SortedIdxs \rightarrow sortIndices(FiedlerVec)$

$Partition\ size \rightarrow G.numofnodes()/p$

$for\ each\ partition$

$Partitions[i][sortedIDxs[j] = true;$

$Compute\ partitions\ weight$

$Cut\ size \rightarrow calculateCutSize(partition)$

6. Multilevel RSB

Multilevel [3] has been introduced to address the problem of big graph due to high computatinal time. It takes advantage of a procedure called coarsening which is applied to the graph in order to create a hierarchy of smaller graphs, it involves merging nodes to form super-nodes, reducing the graph's size while preserving its essential characteristics. Common coarsening techniques include heavy-edge matching, contraction, and aggregation. In our project, HEM was used to choose which nodes would be merged. We stopped the coarsening phase when a relatively small number of nodes was reached in order to speed up the computation.
Coarsening pseudocode:

$Matching \rightarrow HEM(G)$

$for\ each\ match$:

$Coarse \rightarrow matching\ nodes, weights, adjacency$

$G1.setNode(G1.lastNodeID,\ weight, coarse)$

$Process\ unmatched\ nodes \rightarrow Coarse\ with\ same\ parent\ node$

$end\ for$

$for\ each\ match$

For each node in G1:

Set G1 edges based on the adjacency of G node

Return new graph

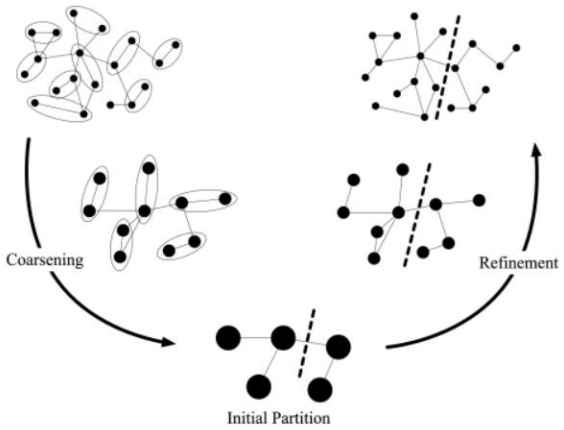

Figure 2: Example of grapgh coarsening/uncoarsening

When coarsening is finished, on the coarsest level of the hierarchy, an initial computation of the Fiedler vector with RSB is performed. Basically, the RSB is applied to the smaller graph and then to obtain the partitioning of the original graph there are a series of approximation of the Fiedler Vector.

Fiedler:

$MatDegree \rightarrow G.getMatDeg()$

$MatAdj \rightarrow G.getMatAdj()$

$L \rightarrow MatDegree - MatAdj$

$if(!stoppingCondition)$

G1 = coarsening(G)

$fv1 = Fiedler(G1)$

$fv = interpolate(fv1, L, sizeNodes)$

$fv = RQI(fv, L, sizeNodes)$

$else$

$fv = eigenSolver(L).eigenvectors.real.col(1)$

Return fv

This procedure of uncoarsening is slightly different from the original procedure in which the original graph is reconstructed, in this case, we are just considering the Fiedler Vector. This kind of uncoarsening uis characterized by two phases of interpolation and refinement for which Rayleigh Quotient Iteration (RQI) has been used.

A great part of the algorithm has been done by fiedler function which is a recursive function that iteratively apply coarsening to the graph until some criterion are met and then it compute the fiedler vector. Since it is a recursive function, when the fiedler vector is compute it return it and start the process of interpolation followed by RQI. The interpolation function just expand the fiedler Vector in order to make it bigger and come back to the original form. Precisely, given a Fiedler Vector $f' = < f_i' >, i = 1 \dots n'$ from a contracted Graph G' with $n' = |V'|$ the interpolation construct an expansion of this vector $f^0 = < f_i^0 >, i = 1 \dots n$ that can be used as an approximation of the original graph G. There is a mapping of $V' to\ V$, from that we obtain $f^0 = f'_i, i = 1 \dots n'$ and the remaining nodes are set by averaging the elements of their neighbours set during the phase of injection.

RQI:

$\theta \rightarrow v^T Lv$

do

$solve\ (L - \theta I)x = v$

$v \rightarrow x/||x||$

$\theta \rightarrow v^T Lv$

$p \rightarrow \sqrt{(Lv)^T(LV) - \theta^2}$

$until\ p < \varepsilon$

Return v

For the refinement phase, RQI is used to help Lanczos algorithm to produce a good approximation of the Fiedler vector. RQI solves a linear system in each iteration, then through the solution a new fiedler vector is computed and the iteration are repeated untile the convergence is reached. Since, an already good approximation of the vector exists the number of iteration is low. The convergence is decided by epsilon value which, in our case, is 10^{-7}

MLRSB:

$fv = fiedler(G)$

$Median \rightarrow computeMedian(fv)$

$for\ each\ node\ of\ G$

$if(fv[i] \leq median)$

$\quad partition[i] = false$

$else$

$\quad partition[i] = true$

Return partition

7. Parallel implementation of RSB

For the parallel implementation of the algorithm, thread and synchronization with mutexes have been used. Precisely, since we used the Eigen library to perform computation we tried to parallelize all the possible operation with different number of threads. Basically the logic of the implementation is the same.
For the normal RSB the main idea was to compute fiedler vector based in the same way as the sequential version of the algorithm, what changes are the following operation:

- Computation of the Laplacian matrix
- Computation of the median of the Fiedler Vector
- Computation of the partitions

Given the number of threads the size of the matrix has been divided by that number and each thread compute a chunk of it. In the same way the partitioning is performed based on the different chunk by different thread.
For the multilevel RSB [4] our idea was to parallelize each part of the function, it was impossible for RQI iteration since it needs to solve a linear system through Eigen library, but it was possible to parallelize the phase of the interpolation and as before also the computation of the Laplacian has been parallelized.

Parallel Fiedler:

$MatDegree \rightarrow G.getMatDeg()$

$MatAdj \rightarrow G.getMatAdj()$

$Compute\ Chunk\ size = sizeNodes/numThreads$

$Create\ Thread\ for\ each\ row\ of\ L$

$L[row] \rightarrow MatDegree[row] - MatAdj[row]$

$Join\ Threads$

$if\ (!\ stoppingCondition)$

G1 = coarsening(G)

$fv1 = Parallel\ Fiedler(G1)$

$fv = \ Parallel\ interpolate(fv1, L, sizeNodes)$

$fv = RQI(fv, L\ , sizeNodes)$

$else$

$fv = eigenSolver(L).eigenvectors.real.col(1)$

Return fv

Parallel Interpolation:

$Compute\ first\ part\ of\ FV$

$Compute\ Chunk\ size = sizeNodes/numThreads$

$Create\ Thread\ for\ each\ missing\ row\ of\ FV$

$Compute\ sum\ of\ the\ neighbors$

$Assign\ average$

Return fv

8. Benchmark file

It was difficult for us to find benchmark graph that suited best our application, since we needed graph weighted both on nodes and edges we decided to implement a function called graph generator that given the number of nodes and edges provides us a file containing a graph.
File structure:

- First row: number of nodes
- Second row: number of edges
- Following rows: (id,weight) of the nodes
- Following rows: (id, id, weight) of the edges

The used files have the following name and format:

	Nodes	Total weight of nodes	Edges	Total weight of edges
graph_50_128.txt	50	300	128	671
graph_100_256.txt	100	587	256	1320
graph_250_640.txt	250	1419	640	3400

graph_500_1024.txt	500	2740	1024	5578

Since our computation facility was limited, we decided to work on graph with an increasing number of nodes starting from 50 and arriving to 500 nodes with a varying number of edges. We limited the weight of nodes and edges to 10 and it is assigned through a random function.

9. Results

Firstly, we will take a look at the time used to read the input file and populate the graph and then we will see the performance of the algorithm.

	Time (ms)
graph_50_128.txt	0.4
graph_100_256.txt	1.1
graph_250_640.txt	6.3
graph_500_1024.txt	24

In the following paragraph there are the results of the performance of the algorithms in terms of time, balance of the partitions, cpu and memory allocated by the program.

The cut size reported for more partitions is the average cut size computed for each of them and this applies to balance factor as well for more than two partitions.

File: graph_50_128.txt with 2 partitions

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	0.06934	0.0660	0.0650	0.0659	0.0637	0.0657	0.0650	0.0655
Balance	0.7857	0.7857	0.7857	0.7857	0.7647	0.7647	0.7647	0.7647
Cut Size	529	529	529	529	540	540	540	540
CPU (%)	96.88	98.82	98.71	98.39	99.04	98.80	98.60	98.97
Memory (GB)	4.20	4.20	4.18	4.17	4.14	4.90	5.03	5

File: graph_50_128.txt with 4 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	0.064	0.0649	0.0661	0.0653	0.0655	0.0653	0.0656	0.0652
Balance	0.651	0.666	0.666	0.666	0.597	0.597	0.597	0.597
Cut Size	295.5	281.5	281.5	281.5	292	292	292	292
CPU	98.57	98.69	98.94	98.72	98.61	98.90	98.90	98.66
Memory	4.09	4.29	4.25	4.21	4.12	4.93	5.04	5

File: graph_50_128.txt with 8 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	0.065	0.0665	0.0654	0.0651	0.0638	0.0649	0.0649	0.0660
Balance	0.5	0.574	0.574	0.574	0.46	0.46	0.46	0.46
Cut Size	155.75	148.75	148.75	148.75	156.75	156.75	156.75	156.75
CPU	98.92	98.63	98.60	98.85	99.17	98.94	98.75	98.68
Memory	4.10	4.29	4.34	4.31	4.12	5.03	4.92	4.90

File: graph_100_256.txt with 2 partitions

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	0.399	0.400	0.400	0.402	0.038	0.038	0.0382	0.041
Balance	0.983	0.983	0.983	0.983	0.930	0.893	0.893	0.893
Cut Size	1073	1073	1073	1073	657	657	657	657
CPU	98.96	98.81	98.90	98.57	98.30	98.93	98.72	94.23
Memory	5.87	5.92	5.92	5.93	7.03	7.15	6.75	7.57

File: graph_100_256.txt with 4 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	0.398	0.396	0.397	0.397	0.038	0.039	0.041	0.039
Balance	0.853	0.853	0.853	0.853	0.838	0.838	0.838	0.838
Cut Size	583.5	583.5	583.5	583.5	473	473	473	473
CPU	99.15	98.93	98.88	98.95	98.06	95.47	96.35	96.60
Memory	5.79	6.03	5.93	5.85	6.79	8.87	8.75	8.37

File: graph_100_256.txt with 8 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	0.395	0.395	0.399	0.397	0.038	0.039	0.048	0.0393
Balance	0.602	0.746	0.746	0.746	0.7	0.7	0.7	0.7
Cut Size	306	279.125	279.125	279.125	286.5	286.5	286.5	286.5
CPU	99.33	99.07	98.80	99.02	98.03	95.33	97.55	97.13
Memory	5.82	6.04	6.07	5.96	6.57	8.5	8.75	8.68

File: graph_250_640.txt with 2 partitions

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	5.35	5.36	5.36	5.34	0.155	0.150	0.154	0.153
Balance	0.91	0.912	0.912	0.912	0.957	0.957	0.957	0.957
Cut Size	996	996	996	996	1603	1603	1603	1603
CPU	99.09	99.13	99.03	99.20	95.83	97.82	96.71	98.32
Memory	16.53	16.76	16.87	16.78	25.5	29.59	28.92	30.84

File: graph_250_640.txt with 4 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	5.31	5.38	5.34	5.327	0.148	0.152	0.151	0.152
Balance	0.802	0.829	0.829	0.829	0.840	0.840	0.840	0.840
Cut Size	960	977	977	977	1260.5	1260.5	1260.5	1260.5
CPU	99.20	99.03	99.15	99.16	98.15	97.49	98.64	99.31
Memory	16.75	16.78	16.93	16.76	23.87	31.18	29.62	29.76

File: graph_250_640.txt with 8 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	5.33	5.34	5.35	5.34	0.150	0.153	0.151	0.153
Balance	0.72	0.772	0.772	0.772	0.744	0.744	0.744	0.744
Cut Size	651.75	659	659	659	746	746	746	754.5
CPU	99.17	99.17	99.19	99.16	97.62	97.01	98.17	98.74
Memory	16.68	16.82	16.82	16.85	23.34	28.39	29.23	28.96

File: graph_500_1024.txt with 2 partitions

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	37.96	38.49	38.41	38.09	0.659	0.662	0.652	0.656
Balance	0.988	0.988	0.988	0.988	0.921	0.921	0.921	0.921
Cut Size	4472	4472	4472	4472	2719	2719	2719	2719
CPU	99.21	98.99	99.11	99.16	97.48	98.02	99.06	99.57
Memory	55.23	55.39	55.42	54.6	90.04	98	96.84	97.18

File: graph_500_1024.txt with 4 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	37.86	38.25	38.20	38.19	0.654	0.659	0.653	0.653
Balance	0.975	0.975	0.975	0.975	0.795	0.795	0.795	0.795
Cut Size	2450.5	2450.5	2450.5	2450.5	2006.5	2006.5	2006.5	2006.5
CPU	99.21	99.20	99.12	99.15	97.784	98.30	98.92	99.59
Memory	54.96	55.82	55.26	55.64	82.15	91.65	90.81	92.07

File: graph_500_1024.txt with 8 partitions:

	pRSB	Parallel pRSB			pMLRSB	Parallel pMLRSB		
Threads	1	2	4	8	1	2	4	8
Time (s)	38.13	38.69	38.507	38.36	0.650	0.656	0.649	0.648
Balance	0.899	0.907	0.907	0.907	0.746	0.746	0.746	0.746
Cut Size	1295.75	1269.62	1269.62	1269.62	1210.75	1210.75	1210.75	1210.75
CPU	99.12	99.06	99.05	99.13	97.93	98.51	99.40	99.9
Memory	55.82	55.62	55.37	55.54	86.75	95.95	94.64	92.04

10. Conclusion

From the previous tables is possible to notice:

- **Graph_50_128.txt**
  The performance of the algorithms in this graph are not so good, the normal version of the algorithm perform a little bit better than the multilevel versions in terms of balance factor and cut size, probably this is due to the approximation made during the coarsening procedure. It is noticeable that memory allocated by the multilevel version is bigger due to the saved coarsened graph in memory. This is a pattern that would be present in all the analysis made on the different graph. The results on the balance are good enough, but not excellent, also this results worsen a little when going higher with the number of partition. This is probably due to the small number of node present in each partition that in case of multilevel lead to non acceptable results in terms of balance.
- **Graph_100_256.txt**
  In this case, the performance of the algorithm starts to be a little bit better, with an obvious increase in execution time due to the doubled number of nodes. Balance factor for 2 partitions is really good for RSB algorithm that in its sequential version performs really well, but in terms of cut size is the multilevel one which minimizes it the most. In terms of memory again is the normal RSB that allocates less memory. In case of 4 and 8 partition the results worsen a little, with still a better results in case of RSB, but again the MLRSB wins on the cut size.
- **Graph_250_640.txt**
  From this file the performance are really good in terms of balance factor and is its also where the MLRSB shows its strength since it is 50 times faster than normal RSB with also better results in terms of balance factor. Cut size instead is worse here and as before the memory allocated is a lot more. This pattern is kept also with 4 partitions, instead with 8 partitions the results in terms of balance are really similar and also the cut size tends be aligned even if it is still worse for multilevel RSB.
- **Graph_500_1024.txt**
  The important part of the analysis starts here, because with a bigger number of nodes we can see that the execution time of MLRSB is still 16 times faster than the normal RSB. In

this case the partitions produced are a little bit more unbalanced, but still with an optimal results. Cut size again is still less for MLRSB and also the memory allocated is a lot more than the normal version. The results for MLRSB are a little bit worse with 4 and 8 partition while RSB remains basically consistent with the results.

In conclusion, the choice between RSB and MLRSB depends on the specific requirements and priorities of the partitioning task. RSB tends to excel in balance, particularly for smaller graphs and fewer partitions, while MLRSB offers superior speed, making it highly efficient for larger graphs. Cut size optimization often tilts in favour of MLRSB, despite higher memory consumption. So, it is crucial to select the most suitable algorithm for a given partitioning scenario.

11. References

[1] https://it.wikipedia.org/wiki/Apple_M1_Pro

[2] Alex Pothen, Horst D. Simon, and Kang-Pu Liou, Partitioning Sparse Matrices with Eigenvectors of Graphs

[3] Barnard, Stephen & Simon, Horst. (1993). A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems.. 711-718.

[4] S. T. Barnard, "A parallel implementation of MRSB," Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences, Wailea, HI, USA, 1996, pp. 594-603 vol.1, doi: 10.1109/HICSS.1996.495510.