Computing the Graph Minor

Parallel and Distributed Systems

Christina Koutsou
github: @kchristin22

1 About

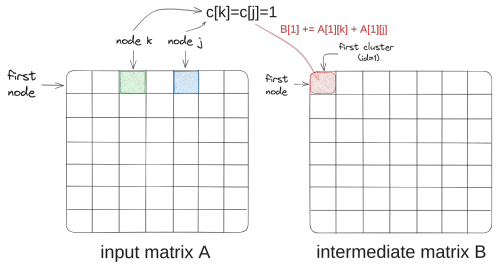
The purpose of this project is to calculate the graph minor of a given graph, based on an input vector denoting the clusters each original node belongs to. This vector contains the ids of the clusters, ranging from 1 to the amount of discrete clusters, in the equivalent positions of the corresponding nodes. No assumptions on the nature of the input graph are made. However, since most of the big networks are sparse, the primary emphasis lies on sparse matrix representations, and particularly in the Compressed Sparse Row (CSR) format. It is important to note that this report does not address any preprocessing steps required to convert the input adjacency matrix into CSR format.

2 Algorithm

In order to facilitate the explanation of the algorithm used, we firstly need to understand the intuition behind the problem. Calculating the graph minor of a graph is another way to signify the compression of a graph. The nodes belonging to the same cluster are compressed into one and the weights of their connection to each cluster are summed together. In other words, this is a problem of additions (reductions), which is also what the following formula tells us: $M = \Omega^T A \Omega$, where Ω is the n × c configuration matrix such that Ω_{ij} = 1 if node i belongs to cluster j, and Ω_{ij} = 0 otherwise. This double multiplication signifies a compression of the rows from the left and a compression of the columns from the right. Instead of having the overhead of forming the Ω matrix and the redundant multiplication of the weights by 1 before summing them, we can calculate these summations directly, using the configuration vector's indications.

The value of the i^{th} position of the configuration vector showcases the cluster the i^{th} node belongs to, which in turn corresponds to the i^{th} row and column of the adjacency matrix A. The result mimics this sorting as well: the grouping with id=1 is linked with the first row and column of the matrix M and so on. This approach takes advantage of the concept of graph isomorphism, so even though the previous node noted as first may not belong to the first grouping and, hence, to the first row and column anymore, the properties of the graph haven't changed, except from the ids of the nodes.

A dense matrix implementation would be divided in two stages: the column and the row compression. Firstly, the non-zero elements of each row are moved and added to the column index equal to their id.



This stage has no problem of cache locality as at each iteration we only parse and fill the elements of a single row of A and B. At the end of this step we have the clusters sorted in each row as mentioned above, containing the sums of the weights that this original node shares with each cluster.

Figure 2.1: *Example of column compression stage*

Afterwards, we pass to the second stage of row compression. In order to take advantage of

cache locality for the final matrix M, we search the ids of the rows of matrix B-contained in the c vector-in ascending order of grouping ids. By searching the c vector we avoid jumping rows of B frequently, which should be time-consuming due to its size. The rows of the same id are summed and stored into the row of M that the cluster id of this iteration points to.

```
    for (size_t j = 0; j < nclus; j++)
    {
         M[(id - 1) * nclus + j] += colCompressed[i * nclus + j];
    }
```

However, as mentioned previously, matrix A is sparse, and, thus, contains many zero elements. Consequently, it's time and space consuming to work with the dense matrix representation. An efficient alternative includes adjusting the above algorithm to A being in a CSR format. It is easier to see now that the step of forming the intermediate matrix B is redundant and, specifically in the case of CSR, the argument of cache locality is not that strong to justify the extra resources and time needed. Instead, we can form the final matrix M in CSR at once, with the help of an auxiliary vector. In more detail, the two stages are fused into one by focusing on a single cluster id each time in ascending order, searching the c vector for nodes that belong to it, then jumping to that index in the CSR row vector to find the corresponding ranges of the CSR column and value vectors. At this point, the elements of the CSR value vector are added to the positions of the auxiliary vector denoted by their cluster id, found by their column index, implementing the column compression of the original matrix.

```
for (size_t j = csr.row[i]; j < csr.row[i+1]; j++)
{
    auxValueVector[c[csr.col[j]] - 1] += csr.val[j]; // compress cols by summing
        the values of each cluster to the column the cluster id points to
}
```

Since we're performing this search for all nodes that share the examined cluster id, each such row adds its values to the auxiliary vector, contributing to the compression of the initial rows. As a result, in the end of this id iteration, we have formed the row of the final M matrix in dense format. It is noteworthy that it is essential to check for zero elements in the auxiliary vector before assigning its values to the CSR value vector of M, as each cluster may not be connected to all the other ones or even itself. Along with the value of the auxiliary vector, its index is stored as its column. The index of the CSR column and value vectors is updated after each non-zero element is added, and appointed to the next element of the CSR row vector-as we form each grouping in ascending id order to take advantage of cache locality for the output CSR vectors.

The overall number of discrete clusters could be calculated by filling a vector with 1 in a position i$\in$ c and 0 otherwise, and performing a reduction on the number of 1s. But due to the assumption that the ids are continuous, we can simply assign the larger cluster id as the number of distinct groupings.

3 Implementation details and remarks

Sequential: CSR formation is quite a sequential algorithm, so it was easy to implement. In addition to the main function used for the tests, two more versions have been implemented: one that uses only dense matrix representation and one that has a CSR input and output, but using a dense matrix representation to compute the intermediate steps. For this specific project we assume that the matrix used for the input is sparse, hence, the function to be used is the one mentioned in Section 2 that only deals with CSR formats. It was also proven to be the most efficient of the three, as expected. In the sequential files, there is also a more general function that counts the number of discrete clusters in case the cluster ids are not continuous, which doesn't apply to the project's scope.

OpenMP: This API allows the user to have a say on the number of threads used for each parallel region , the chunk size appointed to each thread, the scheduling of the threads and the definition of variables as public or private to each thread. On top of that, there is sufficient documentation online on the pragmas and functions available in order to aid the user in taking advantage of them.

In this implementation, the chunk size consists of whole cache lines, if the number of threads is not larger than the dimension of the matrix, as a means to avoid false sharing among threads. These cache lines are attributed to each thread evenly and statically, in a way that a thread will be responsible for a single chunk before suspending. This choice was made based on testing with smaller chunks, f.i. of a single cache line, in combination with dynamic scheduling, where the overhead of the scheduling proved to be observable. In contrast, only the remaining elements that did not fit in a whole chunk are dynamically appointed to the thread that finishes its work first. Moreover, the `nowait` keyword signifies that the threads can suspend upon finishing their task without having to wait for the other ones to finish as well, contributing in performance improvement. Private copies of variables defined prior to the parallel region proved also useful as there was no need to move their definition inside the loop to keep them local. The most valuable feature is the `reduce` one, and specifically the opportunity of having an array of elements that each one of them is to be reduced.

```
#pragma omp parallel num_threads(numThreads)
#pragma omp for nowait reduction(+ : auxValueVector[ : nclus]) private(end)
    schedule(dynamic, chunk)
```

For situations other than the reduction, there's a pragma devoted to atomic operations. The `atomic capture` was used to read the current value of an index and update it atomically, without the use of locks.

Pthreads: The API of pthreads allows the user the most freedom compared to the other ones, but this can also be a curse. The overhead of spawning and joining/deleting threads can surpass the benefits of parallelism. Therefore, this implementation cannot simply mimic the OpenMP one, except for the fact that OpenMP synchronizes its threads without necessarily destroying them. Each pthread is statically appointed with two ranges to work on vectors of size N and of size equal to the number of clusters, in order to divide the work and avoid false sharing. The `auxValueVector` in this case is local and contains the compression of rows and their columns for a specific range. These local results are combined (reduced) into a common atomic vector. The decision of having the

`auxValueVector` as private and using an extra vector instead to be shared aimed to minimize the traffic in an atomic vector. At this point, the threads need synchronization, which is accomplished by the use of a barrier, waiting for a specified number of threads to arrive at it. Afterwards, the common vector includes the final reduced values of this cluster's weighted connections and can now divide it into iteration chunks for each thread to scan (this is one of the input ranges mentioned above). A shared atomic variable is used to keep track of the index of the CSR col and val vectors.

In an attempt to see if dynamic scheduling could prove beneficial, a `prod-cons` version was implemented, which divided the algorithm into tasks and push them to a queue, in order for threads to be able to get the next task as soon as they finished their work. However, even with dividing only one function into tasks, the running time was too long and overall the code was more complex. Hence, this effort was abandoned.

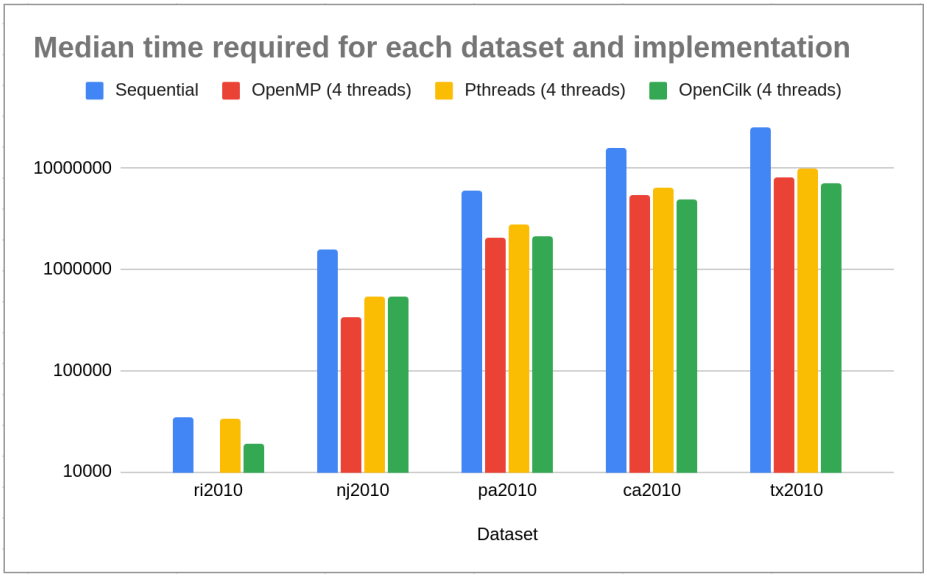
OpenCilk: Working with this API posed some challenges due to limited documentation and available features, making it less straightforward compared to the OpenMP counterpart. The level of abstraction was higher, missing some of the conveniences offered by OpenMP. For example, users couldn't define the chunk size, only set a limit for spawning threads (grainsize). Additionally, specifying reduction was possible for a single variable, but not for each element of a vector. Moreover, there is no atomic implementation tailored specifically to OpenCilk.

An alternative to the version implemented could simulate chunk division of pthreads and spawn threads to execute the parts of code in between the barriers and sync them. This could potentially introduce overhead in handling threads at each cluster id iteration, as there is no barrier offered as a synchronization technique, relying instead on spawning and joining threads. Nevertheless, similarly to OpenMP, OpenCilk may not destroy the threads it spawns, but rather suspend them for a while before re-assigning work to them. Thus, the part of handling atomic vectors could perhaps benefit from such technique, but was not tried out. Even though an atomic vector was used, this version's performance was not far off the OpenMP one and without needing any particular effort from the user.

4 Benchmarks

Dataset	Matlab	Sequential	OpenMP (4 threads)	Pthreads (4 threads)	OpenCilk (4 threads)
ri2010	4500	34289	9767	33618	19133
nj2010	56600	1584908	339045	545741	544979
pa2010	110000	6037295	2060738	2739690	2103721
ca2010	187400	15898973	5423717	6440966	4838647
tx2010	231300	25460247	8036885	9773966	7118827

Table 4.1: *Comparison of implementations using Median time in us*



OpenCilk manages to outperform OpenMP in larger data sets, despite the use of a common atomic vector. This is assumed to stem from its work-stealing properties.

Number of threads	OpenMP	Pthreads	OpenCilk
1	21633	22715	22251
4	9767	11843	19133
8	10266	11075	19243
16	38047	46053	22716
64	421239	434957	78762

Table 4.2: *ri2010: Avg time in us*

It is evident that when we increase the number of threads closer to the number of hardware threads (16) the performance decreases due to the division of the workload in very small pieces, creating an overhead of handling so many threads. In addition, OpenMP threads are tied to a single CPU, so they need cache locality, while OpenCilk dynamically allocates these threads, making it less vulnerable to a change in the number of threads.

5 Extras

Additional things to try out would concern the size of the chunks and taking advantage of attributes shared among neighbouring nodes.