Aristotle University of Thessaloniki
School of Electrical & Computer Engineering

Parallel and Distributed Computer Systems
4th Assignment

Pakas Marios, 9498, mariospakas@ece.auth.gr

Github URL: https://github.com/Mavioux/Parallel-and-Distributed-Systems-4

0. Abstract

Binary Matrix Multiplication is an extremely useful operation in science and engineering, used in various fields such as Data Mining, implementing algorithms known for a wide range of tasks, such as transitive closure, context-free parsing, and triangle detection.

1. Sparse Boolean Matrix Multiplication Sequential Implementation

The idea behind the algorithm is that the only positions with a Boolean value of 1 will be identical to the ones of the matrix filter F. Using its CSC format the [x,y] position of an existent element is calculated. That means that in order for the value in this position to be equal to 1 we need the columns of the x^{th} row of the matrix A that contain values to match at least one of the rows of the y^{th} column of the B matrix. In case at least one such pair is seen, then we can store the result of the element in the [x,y] position as 1, otherwise, it is a 0. So, by comparing every column with every row, using two sorted arrays, we get the final value. Its binary nature allows the algorithm to break out of the comparison at the first occasion of a match, hence resulting in faster code than having to compare all the remaining values. Of course, since the data structure in which the result is stored is a CSC matrix, we don't need to store anything in the positions that the filter F matrix has zero values.

To implement the multiplication $C = F \odot \mathrm{A} * \mathrm{B}$ using sparse matrices the F and A matrix are stored in CSC format so that the elements of each row are easier to gather, while the B matrix is stored in CSR format, enabling access to all the elements in its columns.

2. Sparse Boolean Matrix Multiplication Implementation

To speed the calculation up two methods of parallelization have been utilized, both independently and collaboratively. OpenCilk parallelizes the outer for-loop so that each row in the F matrix is being searched at the same time, enabling the calculation of more than one element [x,y] simultaneously. The Message Parsing Interface (MPI) aids with the parallel and distributed calculation of the block version. This was the most difficult implementation since it required the appropriate breaking of the sparse matrices, so as to assign in each processor a subset of the whole matrices. Then, by passing around the submatrices each processor had calculated a fraction of the problem, which was messaged back to processor 0 to hold the final result. As a final step, the outer for-loop in the MPI implementation was parallelized with OpenCilk to achieve the best possible result.

3. Experimental Results

Each implementation has been compiled and tested on the HPC Auth Infrastructure to check the completion times. All times presented are measured in seconds. The below table pictures all times measured and provides an elegant way to express all results.

	belgium	youtube	dble	cielskian	NACA
serial	0.087932	1.763913	0.063627	0.539022	0.320220
---	---	---	---	---	---
OpenCilk (2)	0.278805	5.074418	0.192323	1.406947	0.799982
OpenCilk (4)	0.321334	3.352160	0.195200	0.761468	0.914551
OpenCilk (8)	0.352363	3.126018	0.212793	0.422077	0.908831
OpenCilk (16)	0.393207	2.811118	0.251401	0.245278	0.883939
---	---	---	---	---	---
MPI (2)	0.098410	8.536654	0.120479	2.421233	0.345864
MPI (4)	0.096419	8.422834	0.095763	2.001625	0.205676
MPI (8)	0.054796	9.406903	0.041331	1.838767	0.160665
MPI (16)	0.018225	9.525907	0.022670	1.505502	0.081438
---	---	---	---	---	---
MPI-OpenCilk (2)	0.024839	0.535940	0.030184	0.136520	0.077520
MPI-OpenCilk (4)	0.059776	0.793653	0.029060	0.133236	0.148808
MPI-OpenCilk (8)	0.073357	0.815906	0.048248	0.144958	0.110132
MPI-OpenCilk (16)	0.157198	1.074799	0.160304	0.262570	0.445075

The above results may seem a bit peculiar at first sight, but everything seems to have been sped up in the end.

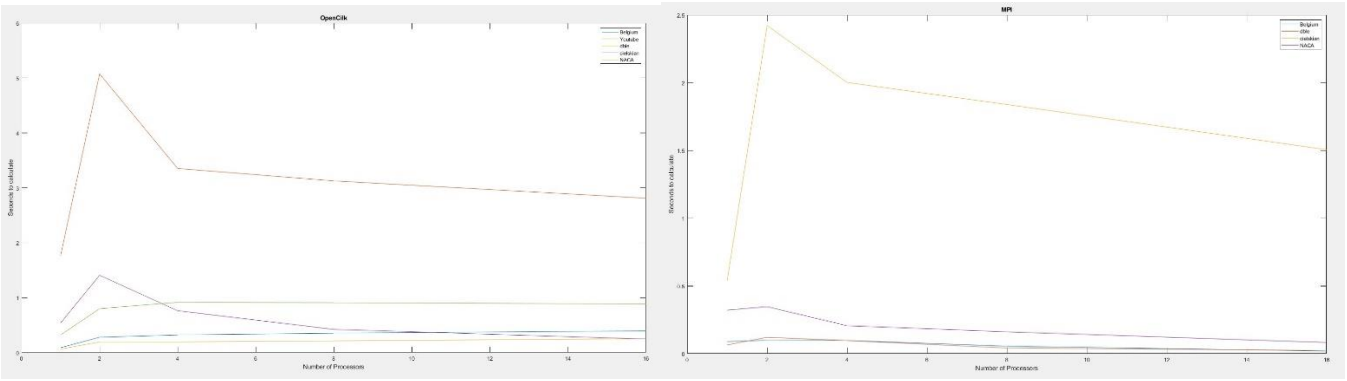
Starting with the OpenCilk implementation it is derived that it offers worse performance than its serial counterpart, and the situation seems to get even worse when adding even more cores to the equation. The reason for this is probably the overhead that it is added when OpenCilk is called, while the parallelization benefits are not enough to negate this. Another factor may as well be the already few time the serial implementation takes to finish the job, leaving little room for improvement for the parallel counterpart. The only outlier to this rule is the cielskian dataset, which actually manages to finish faster than the serial code, after adding 8 cores.

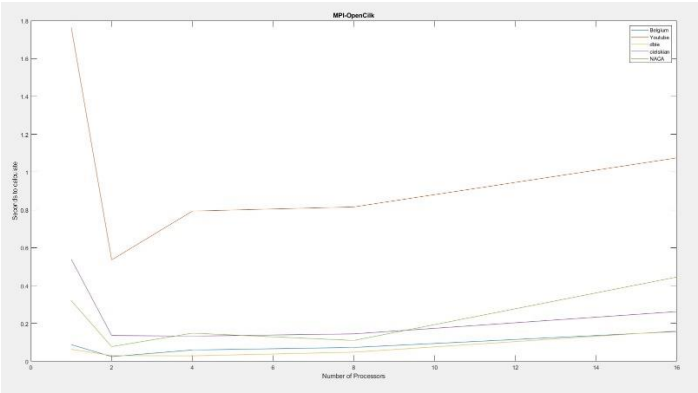
Those are some pretty disappointing results, but fortunately, the MPI implementation seems to provide a more positive perspective. This time the parallel implementation is faster than the serial one on the Belgium, dble and NACA datasets, while significantly slower on the other two. Surprisingly those other two are the ones on which the OpenCilk implementation has the less drawback. This makes up for an exciting final observation when combining MPI and OpenCilk on the same code!

Exceeding our expectations, the collaboration of these two parallel ideas manages to overcome its serial counterpart on every datasheet. Astonishingly impressive, someone might say, but still, the final results come

with an extra surprise themselves. The best possible time achieved is observed when using only two cores and gets steadily worse when going beyond that threshold. One possible explanation for this weird limit might be the nature of our datasheet which in addition to the blocking size (which is dependent on the number of cores used) makes for this result. In general, though, this collaborative implementation seems to deliver almost always a faster result than the serial one, at least up to 16 cores which was the upper limit in this testing. It seems that when adding MPI and OpenCilk together their drawbacks seem to disappear.

The above graph charts picture the above observations on paper. On the MPI implementation there is no representation of the Youtube dataset, due to its big scale relative to the others.





Weirdly enough, for the first two implementations, the choice of 2 cores provides the worst performance, while adding up cores seems to make up for it, but when used together, 2-cores becomes the best one, while adding more cores only adds needless overhead and delay.

Note: Further explanation and analysis for each implementation mentioned above are available as comments in the source code and the Github repository.

Sources:

https://arxiv.org/pdf/1909.01554.pdf