UNIVERSITÀ DI PISA

Project Report

DAG Interpreter

A report on the DAG interpreter library implementation

Author

Giacomo Antonioli

University of Pisa
Deprartment of Computer Science

Project Report

DAG Interpreter

A report on the DAG interpreter library implementation

VERSION 1.0	DATE 12th July 2022
AUTHOR(S) Giacomo Antonioli	
PROJECT NAME DAG Interpreter	NUMBER OF PAGES AND ATTACHMENTS 12

ABSTRACT

The report sets out to analyze and implement a DAG Interpreter, first in a sequential way, to deliver a ground truth for then comparing it with a parallel implementation. This implementation is made using the c++ thread implementation, and was tested on the served provvded by the University.

Contents

Chapter 1

Problem Introduction and Analysis

Given a set of operations, with a certain structure, where some are dependent from others, these can be represented as a graph. If the set has no loops then it can be represented through a DAG (Directed Acyclic Graph), where each node represents an operation and each edge represents a dependency. The aim of this project is to implement a parallel library that allows the user to instance DAGS, specifying dependencies and defining functions for each node throughout the use of lambda functions, like in Fig. 1.1 . For each node an id, an input and output ariety is declared in order to correctly build the graph. Once the graph is defined, the process can be started by calling a blocking function that will push the input through all the graph and return a final result, obtained by the application of the functions in all the nodes of the graph, respecting the dependencies and the ordering of the operations.

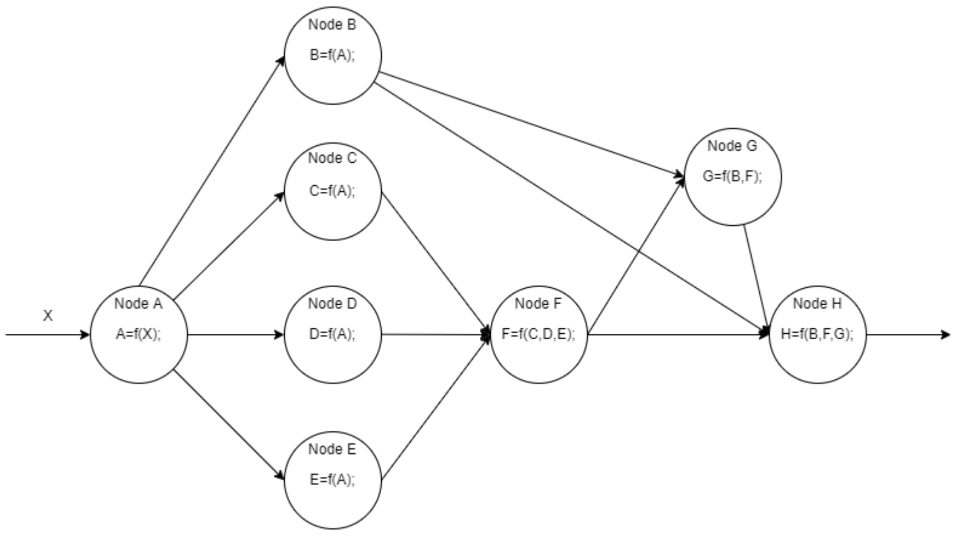

Figure 1.1: Example of a possible Dag created by the user

The aim of this library, as already stated, is to interpret all possible DAG structures, leading to very few constraints on them and leaving the developer open to many different possible patterns. This implies that for some structures, know templates are not applicable, or at least, very difficult to apply. Also each node can have different output that must be correctly rooted to different nodes as input parameters. This can increase the complexity of the communication channels and the the amount of time a function must wait in order to receive all inputs before starting the computation.

Chapter 2

Sequential Implementation

In order to have an initial implementation from where derived the parallel one, a sequential one was made. Here the main structure of the Graph was implemented, with each function encapsulated in the nodes through a lambda function capturing the node. In order to grant a correct computation, some constraints were applied while creating the graph: the created nodes must be inserted following a Breadth First Search style insertion, starting from the initial node with id 1 and assigning increasing ids following the insertion order. This was done to grant a simple and quick way to define a DAG, with it's dependencies and to have a ground truth to evaluate the parallel implementation.
The usage of this implementation is pretty straightforward: the graph and the nodes are defined. For each node the id, input ariety and output ariety are provided. The dependencies of the nodes are defined by adding to each node's dependency list the pointers of all the dependent nodes. Then to each node is assigned a lambda function and the connections between the variables in the function and the arrays in the node are let to the user to handle, linking them by hand. This link respects the same rules used for the declaration of the nodes, to allow a correct data flow in the graph. The graph is then compiled, linking all the outputs of the node with the correct input channel of the through the use of shared queues. This was done thinking directly of a parallel implementation, and accessing the queue in a sequential way, keeps the overhead negligible. To mock an input stream a loop pushing an value in the input of the first node, and starting the compute method over all the graph, was used.

Chapter 3

Parallel Implementation

3.1 First Implementation

The first implementation was derived from the sequential one by creating a thread for each node and communication between them using the shared queue. This implementation showed good performances, but on the other hand was limited by the possible bottlenecks of the graph and the computation times of each function. In fact each function in each node could be run just in single copy, in other words the system could not execute at the same time the same function of the same node because limited by having one thread per node. This lead me to rethink the approach used to develop this library and led me to the following implementation.

3.2 Final Implementation

The parallel implementation exploited the Macro Data Flow model (see Fig. 3.1), where the initial structure and the nodes are considered like the MDF Graph and the MDF Nodes (this because the library, for now, only transforms the whole graph in a MDF Graph, and does not identify and isolate known patterns in sub MDF Graphs).
For a given input x, for each node an MDFI Object is created, which is defined as a tuple <*i*,*f()*,*token*,*dest*>, and is enqueued in a list of MDFI's, not ready to be fired, contained in the MDF Repository. This repository is responsible of keeping track of the MDFIs, promoting them to firable when each presence bit in their input token list is set to true, feeding them to the thread-pool for parallel execution and routing the output to the correct MDFI to permit the data to flow correctly.

One noticeable thing that must be said is that in the first implementation the lambda function capture was directly made on the node of the graph. In this implementation a trade-off between performance and usage simplicity was made in favour of the first one: in this version the lambda function accepts a variable number of parameters, passed through a *std::vector* in input and return another vector as output. This was done in order to make the function independent from the nodes, and from the MDFIs, allowing the second to be copyable without any memory dependency, reducing controls and improving the parallelism. On the other hand this means that the MDFI inserted in the repository are copies, and if the system is used for computations on high memory data this could implies higher overheads.

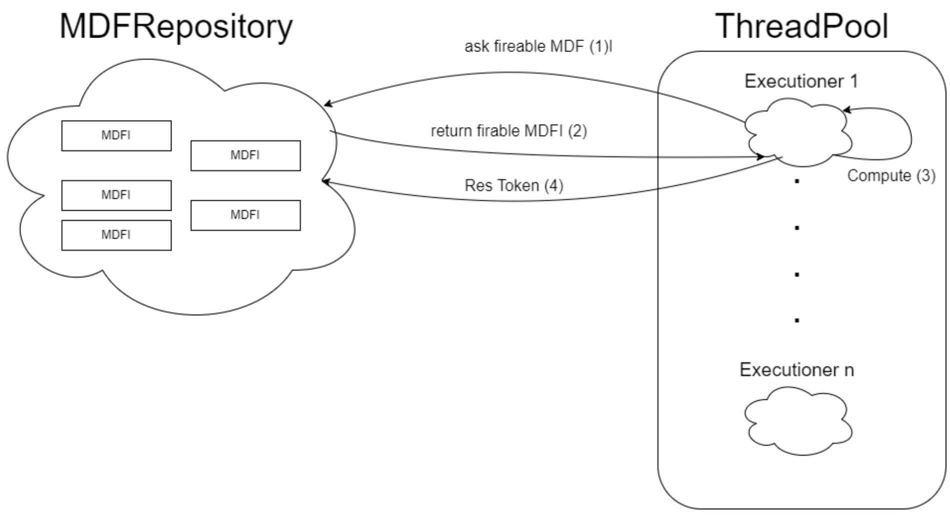

Figure 3.1: MDF Model

3.3 Implementation Details

The library is divided in different classes based on the theoretical fundamental information used to develop it. As already introduced we have a class for the Nodes and one for the Graph. Notable of explanation is class for the MDF Instructions which contains:

1. *id*: An identification value. This is useful to keep track of the destinations of the computations, in order to correctly control the data flow in the graph.
2. *f*: The function given by the user that will be computed
3. *Tokens*: A list of tokens that represent the input list. Each token is defined as follows:
   - **value**: The value of the input.
   - **presence bit**: A bit that represents if the value is valid for computation. This means that the value is not the initialization one, but is was provided by the computation of a function from which this nodes depends from.
4. *Destination Tokens*: A List of tokens that represent where and which outputs to root to which dependent nodes:
   - **i**: represent the position in the output array corresponding to the output to be sent.
   - **id**: the id of the destination MDFI to which root the output.
   - **pos**: the position in the token list of the destination MDFI in which to write the output with position **i**.

and some support arrays to perform the computation of the given function.
A class for the Repository was also created, which contains a map for storing the MDF Instructions, identified by their id as key. A map was used in order to find a target MDFI in logarithmic time, for example to update their token list. Instead the firable instructions are stored in a double ended queue. This because of how the systems stores and promotes firable functions. To explain better this point we must introduce how the inputs are routed in the first place to the first node of the DAG. With the current version it is possible to send all the inputs to the graph before starting the computation, or using a thread, with a inter-arrival time value, that slows down the throughput, simulating a stream. Lets analyze the first method. When the loop of the mock

inputs is started, and MDFI corresponding to each node. The one corresponding to the first node is directly introduce in the firable list, while the others are sent to the ready list. The double ended queue here becomes handy to reduce the completion time of the tasks: if a normal queue was used, when the all the input tokens of at least one element connected to the first node, made him eligible to promotion, he would be promoted from the waiting map to the ready queue, with a push back, adding it at the bottom of the queue. This would imply that all the inserted MDFI depending from inputs would have priority over the the newly promoted MDFI, making them wait all the inputs to be parsed before pushing them trough the graph. Instead, using a double ended queue, allows the new MDFI to be inserted in the head of the list, reducing greatly the service time of each input.
Another noticeable feature of this implementation is the use of mutexes, in fact a shared mutex is use to access the waiting list, while a standard mutex is use to control the firable de-que. The use of the shared mutex allows the application to apply an unique lock, used when the dimension of the map is modified (an MDFI i added by the emitter, or one is promoted to the firable queue), and to apply a shared lock when the threads in the threadpool access the MDFIs in the waiting list. This can be done without any risk because when the executor access the list to update the input list of the target MDFI, he is the only one that will write in that position in that token and in that MDFI. This because each MDFI is identified with an unique ID and the destination list of each MDFI point to an unique position. The use of a shared mutex implied the used of a *std::condition_variable_any*, that is less efficient with respect to *std::condition_variable*, but allows more threads to write safely in the waiting list.
The Repository is also thought for multiple graph computation. This feature is introduced in the code but not fully implemented nor tested due to time constraints for the deadline.

Chapter 4

Benchmarks

For the experiment part, to simulate an active work from the nodes in the graph, a loop that computes some default functions (such as sin(),cos() and tan()) was used for making it stay busy. Unfortunately this simulation does not take allow the thread to access different memory areas and simulate correctly cache hits and misses. The reported experiments wont take in consideration the first implementation, because already theoretically proven to be less effective. So the following results will only consider the sequential and the final implementation.
The experiments were made considering 3 different factors:

- The number of cycles in the "busy loop"
- The number of tasks, so the number of inputs, fed to the DAG
- The number of threads of the threadpool

The following graphs will report only so configurations of these parameters. In addition to the first graph introduced inf Fig. 1.1(referenced as G1), two other graphs were created to test the performances of the library. G2 is a sequential DAG made of 18 nodes, where each node is connected only with the next one. These dag was used to represent a configuration that can be considered to be like a big pipeline of operations. G3 instead represent a sort of Farm (made of 16 nodes), with an emitter and a collector, which both perform computation on the inputs. It must be pointed out that this is not completely a farm because the final node must always wait for all the 16 nodes from which it depends to complete the computation before proceeding. This can also be seen as a situation where there is a bottleneck.

Figure 4.1 highlights how the system is weak to overheads when many threads are deployed to perform the parallelization having few quick tasks. Raising the number of tasks while keeping the same number of cycles wont change the graph because the communication time remains comparable to the computation time.
If instead the cycle parameter is raised the then the communication time becomes negligible and the system performs better.
Looking at 4.2 we can see that the speed up of the library applied to the graph g1 behaves as expected. Instead has an anomalous behavior of the other 2 graphs. My explanation to this could be that, having both 18 nodes in total and the system on which the library was tested has 32 cores, two things happen: For what concerns Amdahl's law there is a super linear speed up for a number of cores that is approximately the same as the number of nodes. Then being the two graphs a particular form of a DAG, and being the nodes less than the total number of cores, the processed MDFI's could be computing more than one whole sequence at a time. This is to be intended like what happens in a pipeline, that when an input is fed to the pipe, an output of another input is returned. The same could be happening with these two particular graphs, but having more threads than nodes this could be boosting even more the performances of the library.

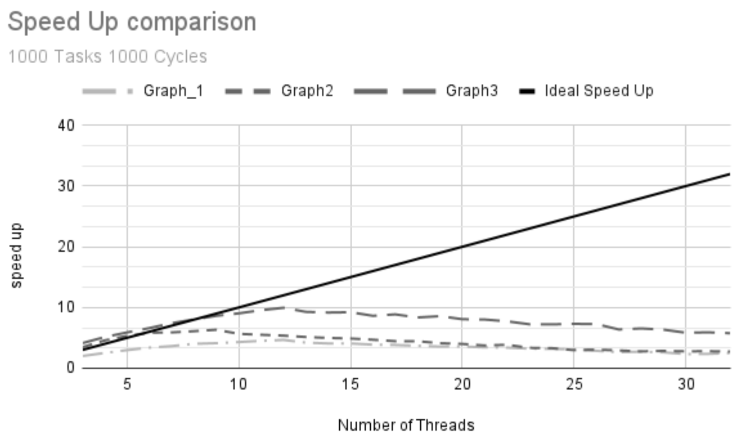

Figure 4.1: Speed Up Test 1

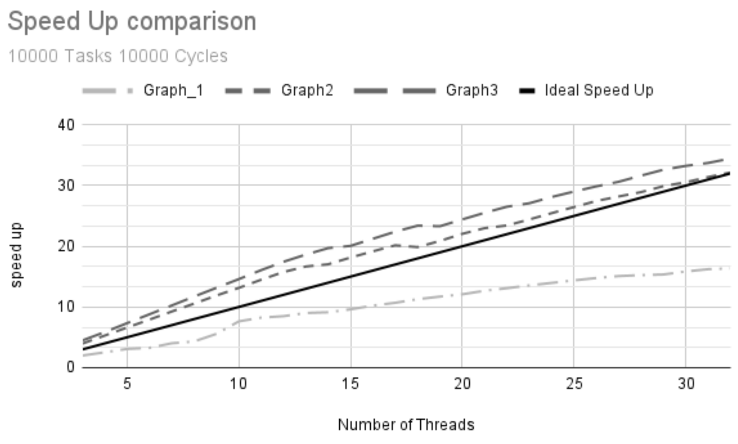

Figure 4.2: Speed Up Test 2

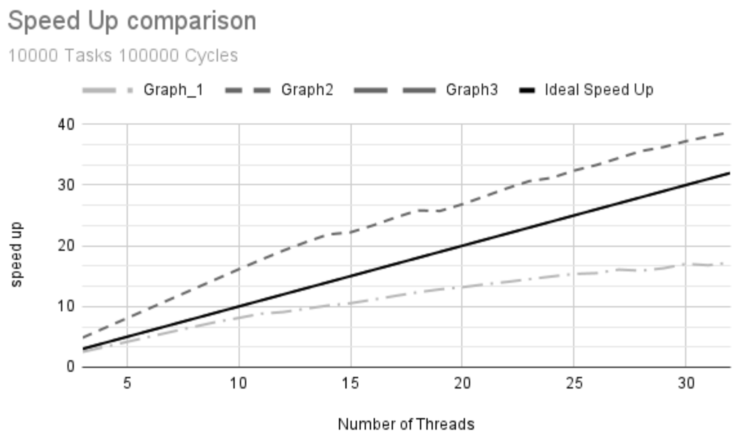

Figure 4.3: Speed Up Test 3

Chapter 5

Deployment and usage

To import he library simply import he MDFRepo.h header file. This will allow the user to access all the needed classes and to create a DAG, compile it and perform the parallel computation.

To define a Node of the dag, create an object Node with the wanted type (for now only float and int)

Listing 5.1: Node instancing

```
Node<typename> A ( node_id , input_ariety ,output_ariety );
//the first node id must be 1 and it's input ariety also 1
//for the last node the output ariety must be 1
```

After the declaration of all the nodes, to link them together, add the dependent nodes to each node, using a pointer:

Listing 5.2: Node instancing

```
A.addDep (&B);
//the last now wont require this step
```

For each node add a lambda function throughout the following method.

Listing 5.3: Function adding

```
 B.addFunction([](std::vector<typename> values) {
        float in1, in2 ,out1, out2;

        in1=values[0];
        in2=values[1];
       out1=(in1+1)*(in2 -1);
       out2=(in1 -1)*(in2+1);

        return (std::vector<typename>{out1,out2});
    });
```

Here the user must be careful assigning the right user defined variables to the right inputs and outputs. Remembering the order in which the users added the dependencies, the indexes of the input array will be in the same order of the incoming arrows. The user must also apply the same logic while writing the output variables in the vector in the return statement.

The user can the define the graph:

Listing 5.4: Graph instancing

```
Graph<typename> g;
```

and then proceed adding the nodes to the graph:

Listing 5.5: Graph Population

```
  g.addNode(&A);
```

Finally the user can define the MDFRepository:

Listing 5.6: Repository instancing

```
MDFRepo<typename> Repo;
```

and add the graphs. (remember that for now only one graph per time was tested, so please try it with just one graph per time)

Listing 5.7: Repository Population

```
  Repo.addGraph(&g);
```

Now the user can start the emitter and for demonstrative and testing purposes the user can only specify how many inputs he wants to pass to the graph. The user can also decide whether to make the emitter deliver all the tasks before starting the parallel computation or making in asynchronous, with a default arrival time of $100ms$.

Listing 5.8: Emitter start

```
  Repo.startEmitter(nTasks, bool async_emitter);
```

Lastly the user can start the parallel computation specifying the number of threads to be used.

Listing 5.9: Flow start

```
Repo.startFlow(nWorkers);
```

An example file is provided, to run it type the following in the console:

```
MDF_2> ./main #cycle_loop_iterations #numberOfTasks #workers
```

so for example to run the graph with 1000 loop iterations in the mock functions, 1000 different inputs and 16 threads:

```
MDF_2> ./main 1000 1000 16
```

Chapter 6

Conclusions

This project allowed me to apply what i learnt during the course and to analyze more in depth the topic of Macro Data Flow. It also gave me the possibility to find different ways to tackle the problem forcing me to evolve the implementation taking in consideration the performance aspect, using the theoretical background gained by studying the course material and finding the empiric evidence by applying what i learnt. It was challenging but also fun to produce a library that could parallelize a given DAG, making it flexible while also efficient. This project could still be improved, maybe optimizing the MDFI and also by applying derivation rules to optimize the graph before starting the computation, in the compilation phase.