DAG Interpreter: a parallel implementation

Luigi di Micco

February 2023

Abstract

In this report we analyze the implementation of a library supporting the parallel execution of interdependent tasks set, modeled as a Directed Acyclic Graph. We start discussing a simple solution and then we present a more performing solution, analyzing the C++ implementation and the perfomances achieved on different structured DAGs.

1 Introduction

A DAG(Directed Acyclic Graph) is a directed graph without cycles, it is really suitable to represent activities and relations between them. Here we can contextualize a DAG as structure such that each node is a task and the edges are the dependences between the tasks. This is a reasonable view, there exist a lot of scenarios where this can be applied. A near context are Machine Learning pipelines computations, where any kind of speedup would be appreciated. Here we want to analyze how to parallelize the execution of this tasks, respecting the dependences, but achieving performances such that justify (the development) the usage of this library for future users.

2 Problem analysis

As already introduced, DAGs can represent tasks with dependence relations, so that we can argue that in general the dependence comes from data. Then, our aim is to realize data parallel computation, but there exists crucial factors to take into account, coming from DAG's structure:

- **dependences** must be respected: they have an impact on the overall computation results;
- **maximum parallelism** achievable: this is influenced by the task interconnections i.e. graph edges (and system limitations, but we cannot play with).

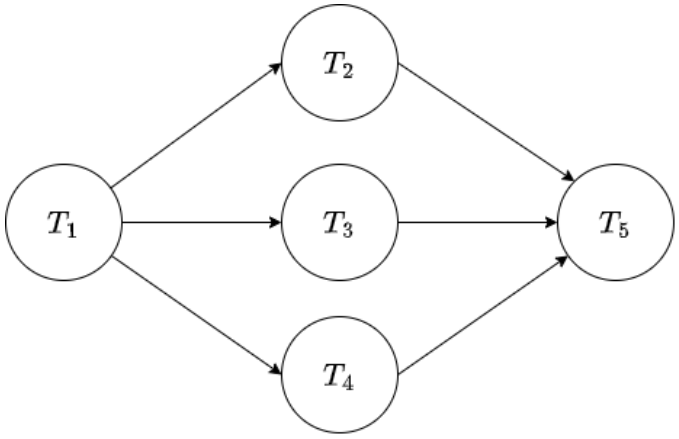


Figure 1: A simple DAG of tasks.

Let us use an example to better explain the issues. Assuming the very simple DAG in figure 1 represents a fully meaningful computation, where there are overall 5 tasks to be executed:

- T_1 starts the computation;
- T_2, T_3, T_4 depend from T_1 and will start the computations as soon as T_1 ends;
- T_5 will end the entire computation as soon as its dependences are satisfied.

In this graph, the achievable parallelism degree is 3, that is the number of indipendent tasks, after the common dependence T_1 is satisfied.

Thus, the achievable performances strictly depend on:

1. how fast the dependencies are solved;
2. the number of indipendent tasks per level.

Let us better formalize the scenario, starting with the execution times, both sequential and ideal parallel, on the example graph:

$$T_{seq} = \sum_{i=1}^{5} T_i \tag{1}$$

where we are assuming T_i be the execution time of the single task T_i. Now, the ideal parallel execution time:

$$T_{par} = T_1 + \frac{T_2 + T_3 + T_4}{3} + T_5, \tag{2}$$

assuming the system makes available at least 3 workers. We observe that we are reducing the time of a factor 3 only with respect to the execution times of three tasks: the ones that can execute in parallel at a certain point. This shows that in general we cannot expect to decrease the execution time strictly proportionally to the number of worker assigned. An example of DAG on which we will never able to improve the execution time exploiting parallel execution is the one shown in figure 2.

Figure 2: A DAG with all interpendant tasks.

All the tasks have a dependence, thus it is not possible to have multiple workers running two or more tasks at the same time: each task is dependant from the previous one. Assuming to have n tasks, we get:

$$T_{seq} = \sum_{i=1}^{n} T_i \tag{3}$$

and

$$T_{par}(nw) = \sum_{i=1}^{n} T_i + \delta(nw) = T_{seq} + \delta(nw). \tag{4}$$

Thus, the execution such kind of DAG in parallel does not make any sense: we will spend a bit more than computing in a sequential way. This *"a bit more"* is quantified using a function δ growing with the number of worker nw, and indicates the overheads due to creation and destruction of the workers.

2.1 A straightforward (inefficient) solution

A first really basic solution consists to topologically sort the DAG and then execute it. We can logically think to "split" the sorted DAG in levels, where with "level" we are referring to the graph terminology sense: the partition of nodes with a certain distance from the root node. So, we supply the allocated number of workers with the tasks of each level, until we reach the sink(s). We manage the dependences exploiting such level concept: we synchronize the data states, putting a synchronization point at the end of each level. A visual look of the explained idea applied on the graph in picture 1 is shown in picture 3: there are three levels and two synchronization point, that we can imagine as barriers.

2.2 Inefficiencies

Despite the solution's simplicity, it introduces clear inefficiencies. First of all, we notice that the barriers at the end of each level carry a pretty evident

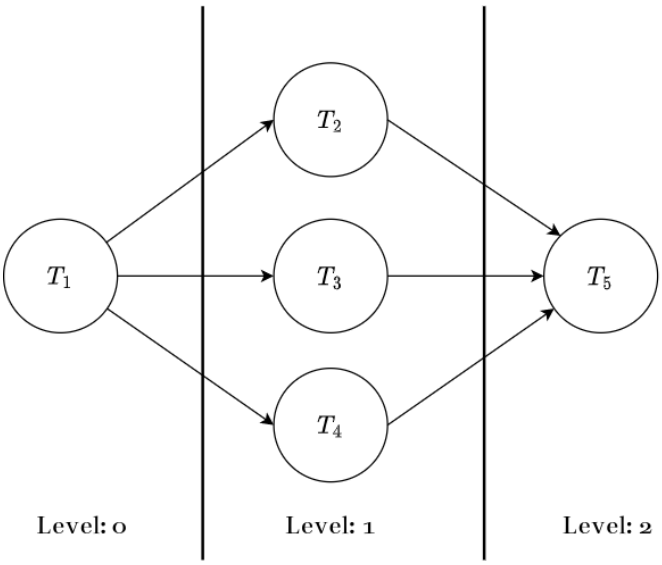


Figure 3: Previous tasks DAG split in levels.

overhead, that becomes consistent with an high number of workers. Moreover, using this approach we constrain the advance on level on the biggest execution time for a level: in other words if all the workers finished computing their tasks but a single one not, and the dependences for some tasks in a successive level are satisfied, the computation on these cannot start until this long last task concludes, or more general all the tasks of the level are completed. This can be seen in picture 4, assuming $T_2 << T_3, T_4$ and $T_5 >> T_6$ we are slowing down a lot the overall computation and cutting away the potential application of the parallel approach on this case study.

2.3 A straightforward effective solution

A really effective approach to this kind of problem follows these ideas:

- run a task as soon as possible;
- exploiting the available workers as much as possible.

And this in a more detailed way can be expressed in a very high level pseudocode as below.

---

start source nodes execution
for all node n whose dependences are satisfied **do**
    start n execution
end for

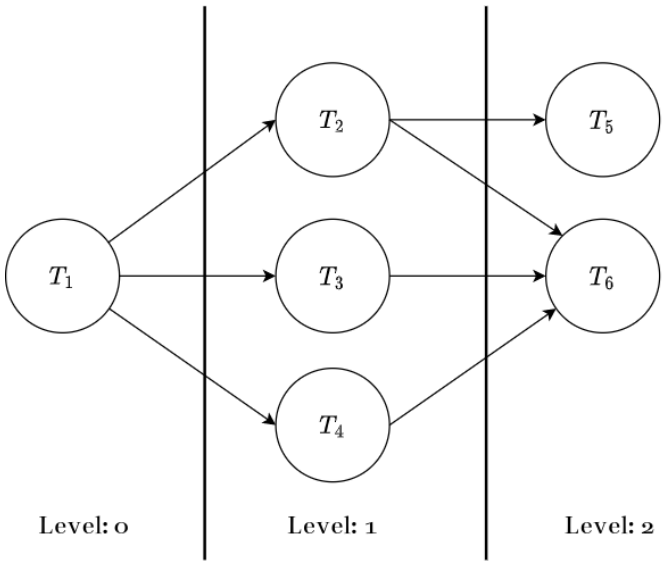


Figure 4: A case study DAG showing a limitation of the approach.

Here we are hiding the workers management and the tasks assignment, but we give a rough idea of the solution that is enough for seeing the advantages and the effectiveness. And what we expect is a real speedup pretty near to the number of instantiated worker because our aim is to exploit the workers as much as possible, thus reduce them idle time during execution. A rough formula describing the ideal parallel execution time with this approach can be:

$$T_{par}(nw) \approx \sum_{i=1}^{l} \frac{\sum_{j=1}^{\#nodes(l)} T_j}{\min\{\#nodes(l), nw\}} \text{ where } l = \#\text{DAG's levels}, \quad (5)$$

and with $\#nodes(l)$ is intended the number of nodes at level l. As already stated it is a very rough estimation because we are assuming to make parallel execution per level and that for each level we will have at least $\min\{\#nodes(l), nw\}$ number of worker available.

3 Implementation details

The implementation is surely based on the previous exposed pillars but more in practice exploits the Macro Data Flow framework implementation concepts. Let us start from top, looking at files tree:

```
ParallelDag
├── utimer.cpp
├── src
│   └── main.cpp
└── include
    ├── graph.h
    ├── node.h
    ├── Mdfg.h
    └── Mdfi.h
```

the library core is located in `include` directory.
All the classes in this library are templatized just using the C++ template support, for supporting computations on different types in a safer way. Each one of these files contain a class declaration:

- `graph.h`: declares the Graph class. This is the core class for a library user: he creates a graph object using this class. After he adds the nodes to the graph, through the graph object sets up the computation type and starts it, supplying the input data. This is the class in charge to create the MDF Graph upon our graph (the one created by a user) and the threads orcherstrator: here we define the thread's work, start the multi-thread execution on the MDFG and we collect the threads.

- `node.h`: declares the Node class. Here we have the basic graph units, all the graph logic is embedded here: input/output arity, nodes' relation that are dependence ones. Let us remark that a node represents a task, indeed we store it as a function taking a vector as parameter and returning a vector. This design choice is for managing the possibility to get data from multiple incoming nodes and for supplying the dependant nodes with the produced data.

- `Mdfg.h`: declares the Mdfg class. Here we find the ad hoc implementation of the MDFG for our graphs problem. All DAG nodes are compiled into a MDF Instruction: we transform the tasks in a suitable form for our interpreter. We put all the instructions in the MDFG repository, except the ones already runnable (the sources of the DAG) that will go in the firable instructions queue. This queue will be used during the entire execution: as the instructions dependences are satisfied, they become firable and then are pushed in the queue. All the threads will pop the instructions from the firable queue and keep the graph execution going on until the MDFI are finished. Notice multiple threads can access the firable queue, thus here are also encoded synchronization mechanisms in a way that all the threads have a coherent view. We will focus later on this kind of problems. Even though the threads' work is defined inside the Graph class, what it is needed to

let the MDFG flow going on is declared here:

  - `getSources`: provides us the entry point for the MDFG interpretation;
  - `getFirable`: this methods returns a firable instruction but it checks if the MDFG computation is completed too;
  - `sendToken`: this is a really crucial method. It propagates the results of ran instructions to the dependants, or in other words it sends the tokens. The instructions that no more miss other tokens are promoted to firable and thus pushed in the respective queue.

- `Mdfi.h`: declares the Mdfi class. This is the class used to represent the Macro Data Flow instruction, as previosly revealed. Here are defined the names useful to transform a simple DAG task into an instruction runnable in a MDFG environment. For example: we store the references to the dependant instructions, as previously stated, will be used for sending tokens; the number of missing tokens, it is useful for arguing when an instruction is going to become firable and so on.

3.1 Synchronization mechanisms

In this section we are going to analyze the adopted solution to the contentions in this project. The program logic vulnerable to multi-thread enviroment problem is located in the MDFG class, the objects that could be subject to are the firable queue and the repository vector. The queue is subject to concurrent write and read, let us summarize the possible situations in two macro scenarios:

- Unpredictable: a thread can pop while another is pushing, a thread can pop while antoher is just reading the size and so on;
- Empty queue: some threads are working maybe on long instructions, other would like to get a task but the queue is empty.

These issues are solved with the use of mutex and condition variables: the mutexes for the firable queue and the repository are just locked when accessing them and released as soon as the use is finished. Meanwhile, the condition variable is used for the queue management: the threads stay on the wait if the queue is empty and there is still to compute. They are woke up (`notify_all`) if something new becomes firable or the entire graph computation is done.

4 Performance analysis

As largely previous analyzed, the performances achieved with the developed library are subject to the graph structure. For this reason we are going to analyze the behaviour on differently structured DAGs, in this way we will show that the previous considerations verify in practical. For the sake of shortness, three different structured DAGs will be analyzed:

1. the most parallelizable and simple: one root, plenty of multiple dependant chains of nodes;
2. a sequential chain (the second needs the first, the third needs the second, and so on...);
3. one with multiple dependendance relations, not exactly the "naive" scenario as the previous.

4.1 The most parallelizable

We show the most parallelizable previously introduced DAG in figure 5: it is enough to compute the task A to let the other workers proceed to compute the outgoing chains. The library is tested on this graph containing in each nodes some not trivial computations and then speedup, efficiency and scalability (varying the number of workers) were extracted and summarized in the charts in figure 6.

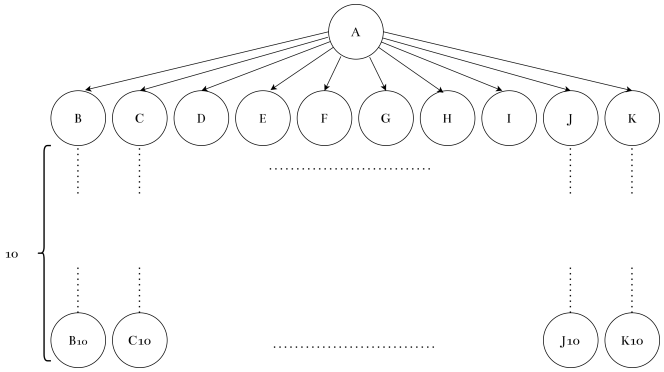
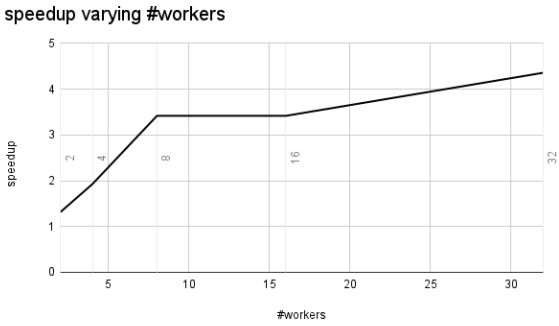


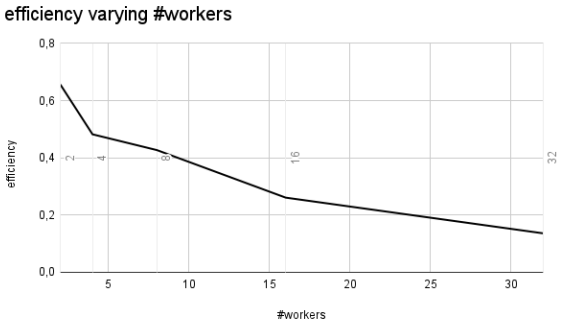
Figure 5: The analyzed DAG having a parallelizable shape.

4.2 The sequential chain

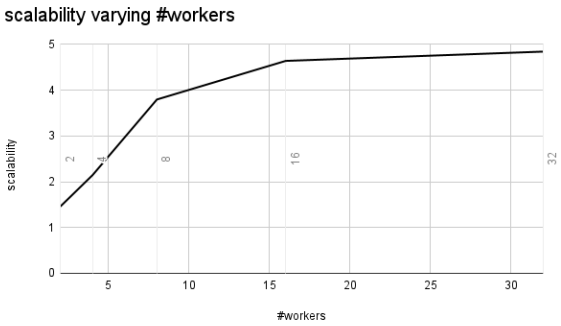
In this case we supply to the executor a DAG that basically consists in a list of interdependant tasks. Thus, there is no advantage in introducing multiple workers, considering that each task should be executed after the predecessor has been completed. This is what happens in practice and we



(a) Speedup measurements varying number of workers.



(b) Efficiency measurements varying number of workers.



(c) Scalability measurements varying number of workers.

Figure 6: Most promising structured DAG measurements.

summarize it in the table 1. It is possible to notice that the best execution time is for the sequential execution scenario, while for all the parallel cases **slightly** increases with the number of threads. This slight increase in the execution times should be due to the fact that nonetheless the overheads increase with the number of workers, there is some advantage during the execution in a way that the overheads are compensated by this a bit faster execution. In particular, looking to which threads take which task is not like in the sequential scenario: all the threads are involved into use up the tasks. The operating system is charge of assigning the tasks to the threads, thus its scheduling strategy takes into accounts of the threads waiting for a task, allowing to advance with execution exploiting all the threads.

Execution mode	**Execution time(usec)**
Sequential	8030559
Parallel, 2 workers	8653012
Parallel, 4 workers	8660107
Parallel, 8 workers	8658713
Parallel, 16 workers	8639635
Parallel, 32 workers	8799159

Table 1: Execution times depending execution mode.

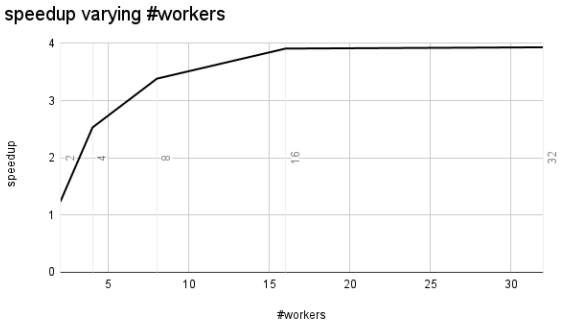
4.3 An "average" scenario

As anticipated we are going to show the library's performances on a DAG artificially constructed nearer to a possible real one e.g. with some syncronization points: at a certain point are needed all the previous computations to merge in a value. The analyzed example has two collectors (or synchronization points). The performances are in the charts in figure 7. We can notice that are not so promising as the overoptimistic first case and it is possible to see that with large number of threads i.e. 16 and 32, the performances do not improve. A larger DAG example could be needed.

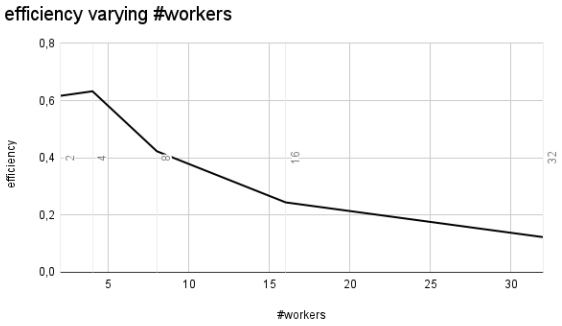
5 How to use the library

The library usage is pretty immediate, the only components a user needs to interact with are Graph and Node classes. A simple example miming the one from the project specification is here:

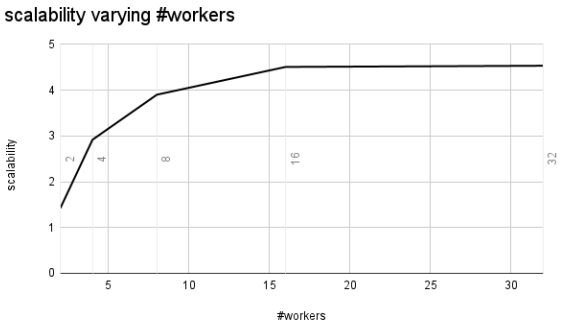
```
Graph<int> g;
auto A = new Node<int>(1,0,2,1);
auto B = new Node<int>(2,1,1);
auto C = new Node<int>(3,1,1);
```



(a) Speedup measurements varying number of workers.



(b) Efficiency measurements varying number of workers.



(c) Scalability measurements varying number of workers.

Figure 7: A possible average case performing DAG.

```
auto D = new Node<int>(4,2,0);
A->addDependant(B);
A->addDependant(C);
B->addDependant(D);
C->addDependant(D);
A->addCompute([](std::vector<int> input){ int yb, yc, x = input[0];
                  yb = x-1; yc = x+1;
                  return std::vector<int> {yb,yc};} );
B->addCompute([](std::vector<int> input){ int zb, yb = input[0];
                  zb = yb* 2 ; return std::vector<int> {zb};});
C->addCompute([](std::vector<int> input){ int zc, yc = input[0];
                  zc = yc* 3 ; return std::vector<int> {zc};});
D->addCompute([](std::vector<int> input){ int zb = input[0],
                  zc = input[1], res; res = zb+zc;
                  return std::vector<int> {res};});

g.addNode(A);
g.addNode(B);
g.addNode(C);
g.addNode(D);

g.setUpComp(nw);
g.compute(std::vector<float>{9});
```

Listing 1: Snippet for example dag in the project definition. (Only parallel execution here defined.)

Important things to notice:

- only source/root nodes should be created with the fourth parameter. It indicates the number of inputs the embedded tasks will wait for;
- all the "ending"/sink nodes should have the third parameter i.e. output arity set to zero.

The CMake file as it is will compile and generate two files: one for the parallel version and another for the sequential one. Multiple main files are left, in order to avoid the stuffing graph definition and to make possible the testing of the multiple DAGs previously introduced.

6 Conclusion

The project allowed to dirty the hands directly with C++ threads and let me understand how much effort could be needed to craft a library. What here is obtained is a really alpha ($alpha^{1000}$) library version, nonetheless achieving pretty satisfying results.