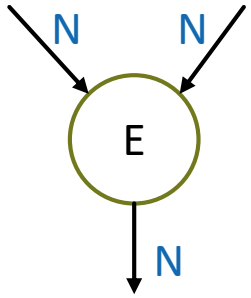
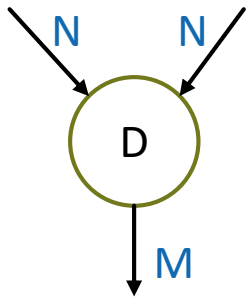
ASA: Scheduling

Type of tasks

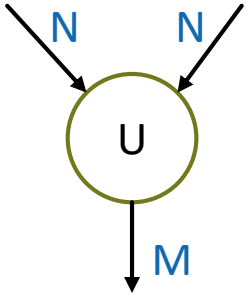
We begin by restricting ourselves to certain types of nodes



Element-Wise
(e.g., vect add)



Down-sampler
(M <N, e.g. dot product)



Up-sampler
(M>N, e.g., out.product)

Up-samplers may trigger backpressure effect

We want to study this at the *steady-state* (synchronous nodes).
Later: deal with intra-task parallelism

Streaming intervals

Steady state analysis: we want we want to keep the nodes producing elements with an interval that ensures that all following nodes can meet

We define for each edge *e* its streaming interval *s(e) a*s the average interval between elements going through the edge *e* while the edge is streaming

All streaming intervals must satisfy the condition $s(e) \geq 1$

Each type of node affects the streaming intervals. Consider a node *v* with incoming edge e_1, outgoing edge e_2 then

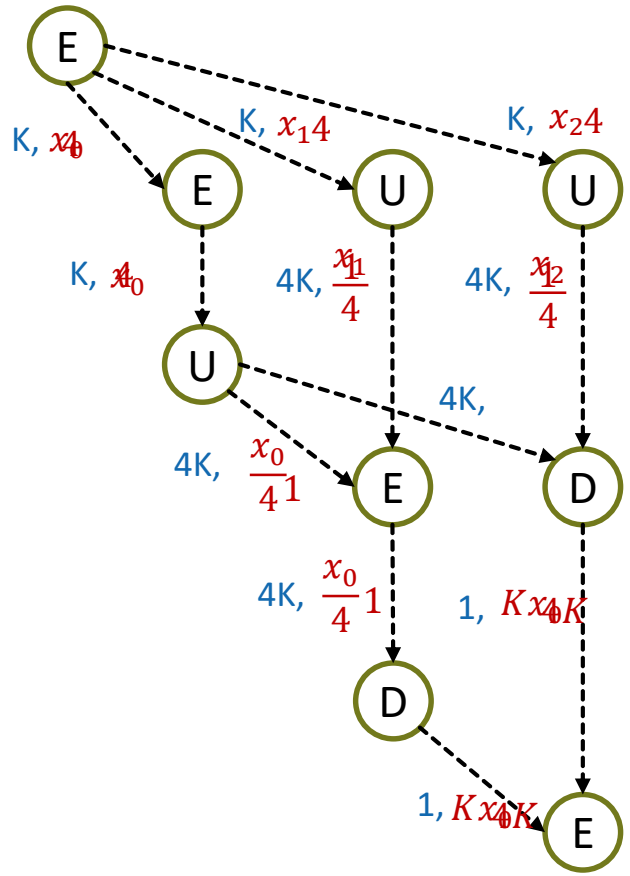
$$s(e_2) = \begin{cases} s(e_1), & if\, v\ is\ an\ element-wise \\ s(e_1)r, & if\ v\ is\ a\ down-sampler \\ \dfrac{s(e_1)}{r}, & if\ v\ is\ an\ up-sampler \end{cases}$$

Streaming intervals

How we can compute them. Let's assume that we have a single source and sink in the graph.

1. We start from the source, and we set its streaming interval to x_i for each of its output edges e_i
2. Then we traverse the graph in topological order and for each node we apply the rules seen before
3. When there are different variable at the input edges, use one of them to continue (they must be equal)
4. At the end all streaming intervals must be greater than one. Solve the system of equations that has been created to get the final result

$$\begin{cases} x_0 = x_1 \\ x_0 = x_2 \\ \frac{x_0}{4} \geq 1 \\ \frac{x_1}{4} \geq 1 \\ \dots \end{cases} \Rightarrow \begin{cases} x_0 = 4 \\ x_1 = 4 \\ x_2 = 4 \\ \\ \dots \end{cases}$$



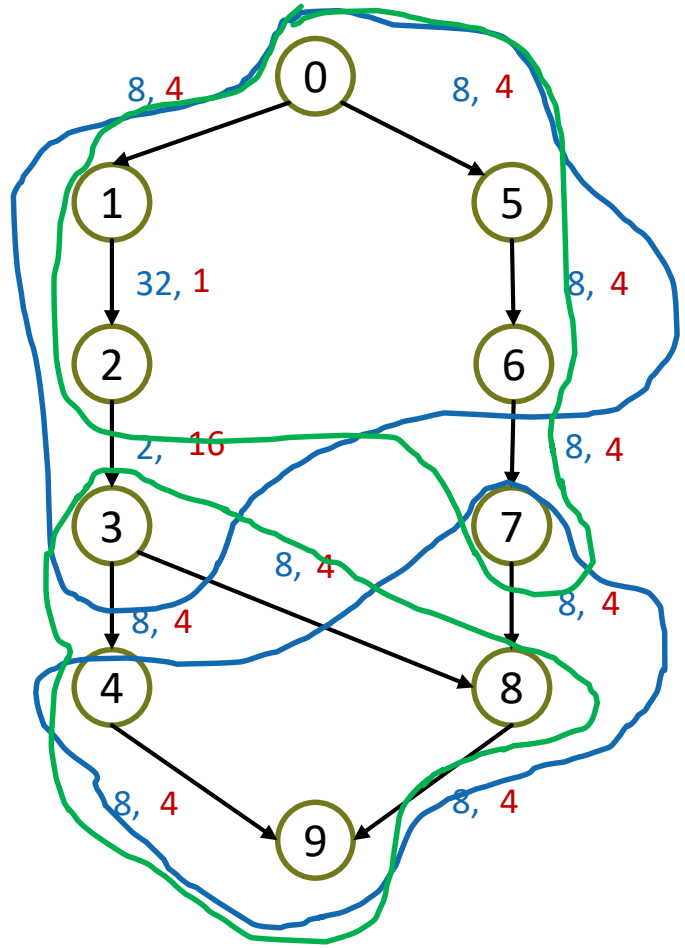
Can be extended to multiple sources/sinks

Streaming blocks

Now that we have info about the streaming interval, we can use this insights to partition the graph in streaming blocks.

Here we can use several heuristics. Some examples:

- **Max work**: we build the streaming blocks by picking the next ready node that has the maximum work (no backpressure)
- **Max running time**: we build the streaming blocks by picking the next ready node that has the maximum running time. This is computed by using the streaming intervals, therefore considering backpressure
- **Min Streaming interval**: you may add a node to a streaming block that will (or can be) slow down the execution of others. In those cases, better to put that node in a new streaming block.

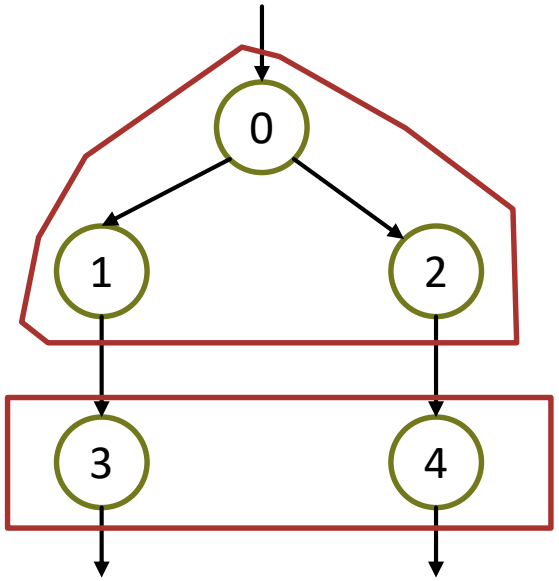


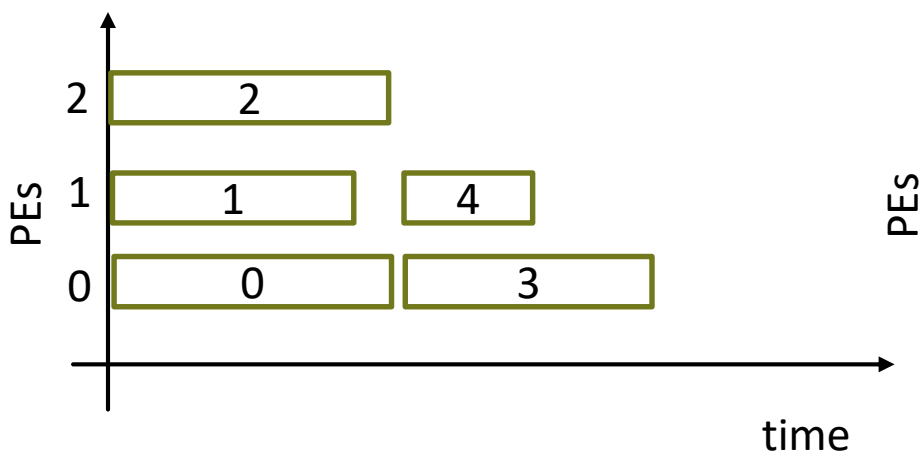
6 PEs

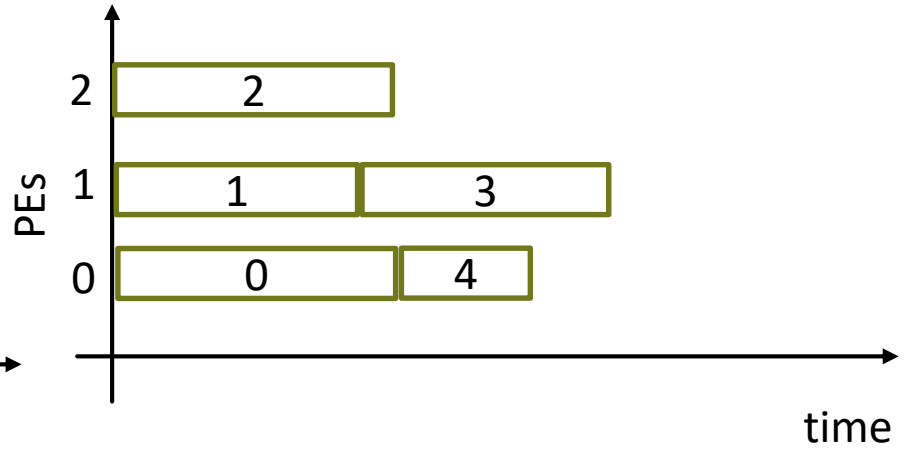
In this particular case, the latter produce better result (smaller makespan)

Scheduling Heuristics

Gang-scheduling heuristics: all nodes in a streaming block are scheduled together. When they finish, we move to the next streaming block (e.g., by reconfiguring the architecture)







List-based schedule

Algorithm 1. List-Scheduling

while *there are tasks to be scheduled* **do**
  Identify a highest priority task n (e.g., from a list);
  Choose a processor p for n;
  Schedule n on p at $est(n, p)$;
end

- Tasks are ordered according to streaming blocks
- Uses insertion slot to create more compact schedule
- (may allocate on the same PE two streaming tasks)

Implementation

Proof-of-concept of the analysis, heuristics (streaming blocks and scheduling) to assess our findings

- Written in Python
- DAG is represented by using a direct Graph (networkx) with weights over the edge
- The framework takes in input the DAG, the number of PEs and:
  - Partition the DAG in streaming blocks
  - Compute buffer space (TODO)
  - Schedule the DAG

Current evaluation: done considering random graph generated from different use cases (linear chains, FFT, Gaussian Elimination, …. More to come)

To schedule a DAG we go through an analysis that estimates backpressure (the computation of the streaming intervals, buffer space), and heuristics to derive streaming blocks and schedule them. We would like to:

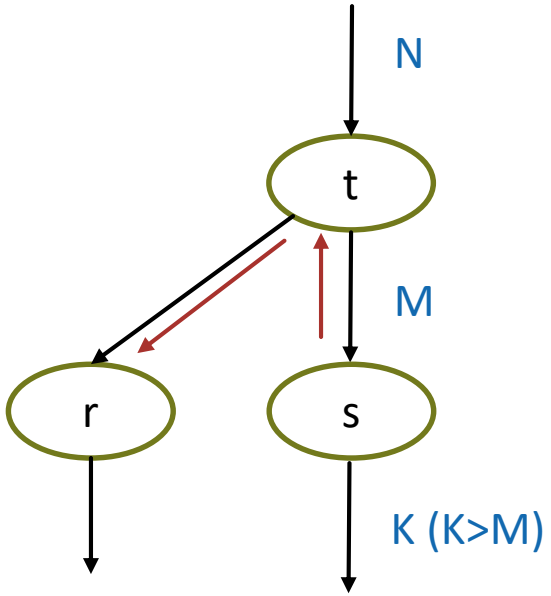
- Assess the Quality of Result (is our schedule good?)
- Check for Correctness: we will need to compute buffer space (TODO). We want to be sure that it is computed correctly so that it prevent deadlocks

We would like to use simulation (maybe Discrete Event Simulation) to do this

TODO

- Continue with implementation/validation
- Simulation
- Buffer space
- [Study how to represent more generic computations]

Upsampler



The running time is given by the volume of data being ingested/consumed

$$\overline{T_t} = \max\{N, M\}$$

Given an up-sampler with ratio *r* (K/M), it will ingest an input element every *r* unit of time, producing a new output element every unit of time

An up-sampler node may slow-down its parents. This affects its running time and the running time of other parent's descendants

$$T_t \geq \overline{T_t}$$

The backpressure effect must be taken into account while scheduling the tasks

We want to study this at the *steady-state* (synchronous nodes). No queuing theory or other, since it would never work…rather let's reason about at which time we should generate the data so that it is not backpressured?

Alternatively: assume that we can always scale a task. This must be taken into account in the analysis

Scheduling

Streaming ⟷ **Scheduling**

Once we build our streaming blocks, we can perform the actual schedule. By means of analysis and heuristics for building the streaming blocks we can break this cyclic dependency

First approach: use a gang-scheduling heuristics: all nodes in a streaming block are scheduled together. When they finish, we move to the next streaming block (e.g., by reconfiguring the architecture)