Dear colleagues, I am presenting the paper Memory-aware Adaptive Scheduling of Scientific Workflows On Heterogeneous Architectures, written by me and my colleagues Profs anne benoit and henning meyerhenke.

Fields of science such as genomics or biomedical imaging require the analysis of massive datasets. This analysis often takes the form of workflows, i.e., separate software components chained together in some kind of complex pipeline.

These workflows are often represented as a directed acyclic graph (DAG) like this methylseq workflow used to analyse DNA methylation patterns.

These scientific workflows needs to be scheduled on a parallel or distributed system, consisting of multiple computers.

This execution environment is usually heterogeneous with different machines, and often with limited memories.

Exceeding memory limits means we get this muffin situation, where there the memory requirement does not fit into this limited memory available. It is in the best case extremely costly to offload parts of memory to the disk. In worst case, the workflow is written in a way that fails the execution if there is not enough memory. In particular, nextflow workflows often have these hard requirements and fail if not provided with adequate resources.

Now, we can be overly conservative and always require too much memory for each task, but this wastes resources and excludes computers with smaller memories from processing tasks they might be able to process. In the interest of managing the heterogeneous execution environment well, the schedule should be aware of memory sizes.

We propose a variant of HEFT algorithm HEFT stands for Heterogeneous Earliest Finish Time.

HEFT is a highly influential scheduling algorithm, de-facto standard. Variants of HEFT exist for nearly every possible scheduling model, except for limited memories. Unlike the original, our algorithm respects memory sizes and includes eviction strategies for cases when it might be beneficial to remove some data from memory.

In our experiments, our heuristics produce valid makespans, while HEFT does not. We still compare these valid results with invalid HEFT ones, thus giving HEFT a bonus, so to speak. Still, our makespans are only 10-30% worse. HeftMM even shows an unique resilience in being able to schedule workflows in extremely memory-constrained conditions.

Furthermore, while HEFT assumes perfect knowledge of the execution time of each task, the actual values might differ upon execution. We propose an adaptive scheduling strategy, where a schedule is recomputed when there has been a significant variation in terms of execution time or memory. This recomputation yields over 20% makespan improvement even for the smallest workflows.

Let us first properly define our model.

The workflow has a shape of a directed acyclic graph.

The vertices of this DAG represent the tasks, the edges the dependencies between the

tasks (usually data to be communicated to successor tasks).

When executed, each task requires a certain amount of memory and if that exceeds the available memory, the execution fails.

It also takes a certain number of operations to be executed, hence its makespan weight wu.

A cost cu,v is associated with each edge, representing the size of the output of task u, to be used by task v.

Mu is the total memory usage of the task during its execution, including input and output files currently being read and written.

The total memory requirement for executing task u, ru , is the maximum of the sum of weights of all input files, sum of all output files, and the total memory size mu (which often reaches the

maximum).

The execution environment consists of heterogeneous machines. Here hte size of the box represents the memory size of the processor, with p1 being the processor with the most memory. The processors also have heterogeneous speeds, as the arrows show. Each processor also has a communication buffer, not depicted here. The communication buffer size is much larger than the memory size.

The processors communicate between each pair. Only one file can be transferred at a time between each pair. Some communication channels are faster than the others, as you can see between üprocessors p1 and p3. We keep track of ready times on each individual ruffer, and on each communications channel between a pair of processors, called rt j j’.

The typical goal is to execute the workflow without failures (i.e., satisfying the memory constraints) and with the shortest possible execution time (i.e., to minimize its makespan).

HEFT is a fundamental scheduling algorithm for heterogeneous execution environments, introduced in 2002. It first assigns each task a rank. Usually, this rank is the bottom level of this task, so that upper tasks receive higher priorities. Then it tries to tentatively assign each task in this order. It tries to put each task on each processor and chooses the one that minimizes the finish time of this task. Suppose tasks 1 and 2 were assigned to processor p1, where do we put task3= we can either assign it to the same processor, or we can assign it to another one. However, then we need to communicate the file on the edge 1-3. We need to first write it to disk, then read it. These communications may take time.

Because processors are heterogeneous, these communications may take a different time if we would read the file on another processor p3.

This is all nice, but what about the memory sizes? Those are limited, right? HEFT doesn’t check for a fitting memory size. OUr heuristics do. However, there’s a catch. The peak memory requirement of a workflow is not a fixed value. It depends on the order in which we traverse this workflow. Let me illustrate with an example

We have three tasks, one edge has a very high weight, the other task has a very high weight. We can execute it as task 1 then 2 then 3. We can execute this workflow in two orders 1 then 3 the 2, and one then 2 then 3.

Load 1000 MEMORY WORDS

In each case, after task 1 finishes, we need to hold in memory all its outgoing files, that is 1001.

If we first execute task 2, then we need to load into memory its weight, 1000. Be we also cannot forget about the edge 1-3. So we have to hold in memory both the ths. Our overall memory requirement is 2001.

But if we first execute task 3, then we don’t need to hold this edge weight in emory longer than needed. By the time we come to task 2, task 3 is finished and its files are deleted from memory.

The intuition says that we can find a traversal that minimizes memory requirement. This is true and such an algorithm was found by Kayaaslan and colleagues.

Another option of our model are evictions of files from emory to the communication buffer. Let us consider this example. We have several files in the memory of this processor. Therefore, its available memory is smaller than its actual full memory. Now comes a task. IT requires more memory than is available. We can still execute this task, if we evict some of the files into the communication buffer. However, this takes time, and we need to weight the time to evict versus the time to execute elsewhere. Files that were evicted into the buffer, cannot be retrieved back, they can only be communicated to other processors.

Our heuristics work in the same two steps and HEFT. FIrst, the rank the tasks. We propose three variants. One ranks with simple bottom leves, the second one uses bottom levels with communication. IT prioritizes those tasks who have larger incoming communications. OUr intuition is that we will achieve a better memory management this way. Finally our third variant, treaverses the workflow in the same order produced by Kayaslaans memory optimal traversal algorithm.

In the next step, we tentatively assign each task to each processor and take the one that is valid and minimizes finishing time. The files of all predecessors that finished on the same processor, cannot have been evicted, otherwise an execution here is impossible.

Then we assess, how much memory the task needs versus the available memory on the processor.With this, we want to understand, how many files we need to evict. If an eviction is required, we tentatively assess is such an eviction is possible due to buffer size, and calculate the finishing time taking into account the time needed to evict.

For the finishing time, we first need to orchestrate all communications from all predecessors that were scheduled to other processors. These communications can take place if the communication channels are free (or after they have been freed). Then we add the execution time on this processor.

After the best processor has been chosen, we finalize the assignment. We carefully manage the content of the memory - for evicted files, we move them to the buffer, reduce its available size accordingly, remove the incoming and add the outgoing files, and update the ready times on all communication channels involved.

Finally, our adaptive scenario.

We cannot assume to know exactly how long each task will take to execute and how much memory it will consume.

In a workflow execution environment, the scheduling algorithm interacts with the runtime environment. A monitoring system observes the workflow execution and collects metrics for tasks. Actual execution times may vary due to different factors. Also the underlying infrastructure can change during the workflow execution.

Our scheduler has been closely integrated with a runtime system, that warns the scheduler when the task parameters have changed, and a schedule can be recomputed on the fly. The changes we model are workflow-related. A task can take longer or shorter to execute and take more or less memory than predicted.

These changes can do three things - they can invalidate the schedule, if for example the task needs more memory than the processor has available. It can lead to a later finishing time, for example if the previously scheduled task on this processor takes longer to execute.

Our algorithms receive the information about these changes from the runtime system and react to them by retracing them in the original schedule and changing this schedule.

We ran our experiments on real.world workflows and ones generated from using the workflow generator. We divided them by size. We test two execution environments with 36 processors of 6 different kinds. In a memory-constrained cluster we gave all machines 10 times less memory than in the default one.

These are success rates in the default cluster. This is the ration of valid schedules per each algorithm. You can see that HEFT produces largely invalid schedules the ratio of valid schedules falls to zero very fast. All our algorithms are able to produce valid schedules on all workflows, including the largest ones with 30k tasks.

Here are relative makespans of the schedule produced by our algorithms in comparison to the makespans produced by largely invalid heft schedules. You can see that our valid schedules are comparable with heft schedule for heftm bl and heftm blc. For the middle sized workflows, the different is less than 20%. Heftmm produces worse makespans that are on occasion even twice worse than heft. However, they are all valid.

Now let us lok at the memory constrained cluster. Here, heft cannot produce any valid schedules over the size of 200 tasks. Also our heuristics bl and bls fail to produce valid schedule for workflow over 10k tasks. You can see here the decline of rate of valid schedule with size. But look here - heftmm was able to schedule all workflows, even the largest ones, even in these extremely constrained environments. Heft mms success rates are unparalleled.

Here are comparative makespans. You can see again that heft mm makespans are larger, but have in mind that these are actual valid, achievable makespans. Where bl and blc can produce a makespan, their results are also comparable with heft.

In our adaptive scenario, sadly most schedules were rendered invalid. Because even one task exceeding the memory size invalidates the static schedule, we have no data forn workflow sizes over 2000 tasks. But under this size, you can see that recomputing the schedule is beneficial. This recomputation improves the makespan by over 20% even for these small workflows.

Finally, the runtime of our scheduler grows with workflow size. YOu can see the heft mm takes the longest to execute. This is because an optimal memory traversal has to be found for these large workflows.

I presented you our memory-aware scheduling heuristics for heterogeneous execution environments.

In our model, the communication happens over a communication buffer.

Our algorithms are heft-based, relying on bottom levels and optimal memory traversal for ranking.

UNlike heft, they produce valid schedules successfully. The variant heftmmm is even able to schedule all workflows in an extremely memory-constrained cluster. However, it produces worse makespans than the variants bl and blc and also takes longer to execute. We also adapted our algorithms to a dynamic setting.

Thank you