Parallel scheduling of task graphs with minimal memory requirements

Pascal Fradet
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG
Grenoble, France
pascal.fradet@inria.fr

Alain Girault
Univ. Grenoble Alpes, INRIA CNRS, Grenoble INP, LIG
Grenoble, France
alain.girault@inria.fr

Alexandre Honorat
Univ. Grenoble Alpes, INRIA CNRS, Grenoble INP, LIG
Grenoble, France
alexandre.honorat@inria.fr

***Abstract*—Many computing systems are constrained by the amount of available shared memory they are allowed to use. Modeling an application with a task graph makes it possible to analyze and optimize its memory usage. We therefore address the problem of finding a parallel schedule of a given task graph that minimizes its memory peak (the maximum memory usage at any point), which is an NP-complete problem. We start by reusing a previous technique that is able to find the optimal sequential schedule for a large class of task graphs, optimal in the sense that the memory peak is the smallest possible one. From this optimal sequential schedule, a dynamic parallel schedule can be derived with a list scheduling algorithm, which we adapt to take into account memory requirements. Provided that the memory constraint is equal to the memory peak of the sequential schedule, our approach always succeeds in producing a parallel schedule that meets the given memory constraint, and that enjoys relatively good speedup (2.68 on average for 4 processors). When the memory constraint is less harsh, the resulting speedup is significantly more substantial (3.57 on average for 4 processors). We compare with the previous state of the art on multiple applications expressed as task graphs, scientific workflows and signal processing filters. When the given memory constraint is close to the minimum, our approach always succeeds in finding a parallel schedule meeting this constraint, whereas the other approaches mostly fail. When the given memory constraint is significantly higher, our approach is comparable to others in terms of speedup, but much faster and it can deal successfully with very large task graphs (up to $50,000$ nodes) using a naive Python implementation.**

***Index Terms*—memory minimization, task graphs, parallelism, list scheduling**

I. Introduction

Memory footprint is an important constraint to consider when developing software applications. In the domain of artificial intelligence, some neural networks (*e.g.,* large LLMs) cannot entirely fit in the memory of a single GPU. Similarly, embedded systems are subject to strict memory constraints, specially IoTs. In the first case, the memory demand is too high, while in the second case, the memory offer is too small.

The modeling of neural networks with dataflow graphs of tasks [1] enables the analysis of memory requirements and the production of schedules that optimize the memory usage [2], [3]. Similarly, (synchronous) dataflow graphs help to efficiently schedule tasks of embedded applications w.r.t. their consumed and produced amounts of data [4].

The problem we address is to find a *parallel schedule* that minimizes the *memory peak* of a task graph. We target *shared memory multi-processors*. The memory peak is the amount of memory required to execute the task graph according to a specific schedule. The *optimal* memory peak of a task graph is the *smallest* memory peak among all its possible schedules (sequential or parallel). This problem is a variation of the pebble game [5], which is NP-complete.

It is fairly easy to guarantee that each scheduling decision is safe w.r.t. the given memory constraint: one simply needs to check that the memory cost of the current task plus the transient memory amount of the partial schedule just before it does not exceed the memory constraint. Unfortunately, this check does not guarantee that the current scheduling decision makes it possible to complete the execution of the whole schedule without violating the constraint. This may become impossible, for instance, when the scheduled task produces large results leaving insufficient space for the remaining tasks whereas scheduling another ready task (*e.g.,* consuming more and producing less memory) would have permitted to complete the schedule. We solve this issue by computing first a memory-peak-optimal sequential schedule.

We have recently proposed a new technique able to find a sequential schedule minimizing the memory peak for a large class of task graphs [6], [7]. A sequential schedule is memory-efficient but slow, as only one task runs at a time. A parallel schedule is faster of course, but has higher memory demands due to concurrent task executions, which all need their own memory space. Nevertheless, we show in this article that it is possible to find parallel schedules with good speedups requiring the same minimal memory as the optimal sequential schedule.

Since task durations are hard to estimate precisely, static scheduling often exhibits poor performance, so we target *dynamic scheduling*. This in turn requires fast execution times, so we rely on list scheduling heuristics [8]. The difficulty in our case is that we must comply with a strict memory constraint provided by the user. This memory constraint may be the optimal memory peak of the sequential schedule or any value greater than that.

We take as input a task graph (that is, a dataflow graph of tasks) with memory costs attached to edges. We start by

computing an optimal sequential schedule, as described in [6], [7]. Based on this information, our dynamic scheduler is able to produce a parallel schedule that preserves the minimal memory peak of the sequential schedule and that achieves good speedups on standard benchmarks ($\times 2.5$ on average for 4 processors).

Our dynamic scheduling algorithm is a *ready list scheduling under memory constraints*. We propose three variants that differ according to the ordering of the ready list and the checks needed to satisfy memory constraints. The first and simplest variant schedules the tasks strictly in the same order as the optimal sequential schedule. Before any scheduling decision, it checks that the memory peak of the task to schedule added to the current memory occupancy satisfies the memory constraint. If the check fails the scheduler waits until the constraint is satisfied. We know that this will be eventually the case since the sequential execution respects the constraint. In the second variant, the ready list contains all the ready tasks sorted according to their bottom level [9]. It requires an additional check to guarantee that the memory constraint is met up to the end of the schedule. This variant provides better speedups. In the third variant, the tasks in the ready list are sorted according to a weighted sum balancing the bottom level (as in variant two) and their position in the sequential schedule (similarly to variant one). Like the second variant, it requires the two checks. The variant providing the better speedups depends on task graphs but the second and third variant are comparable on average.

Our main contributions consist of:

1) an algorithm that computes a dynamic parallel schedule respecting memory constraints for any graph (unlike all previous work in the field). Even when the objective is set to the same minimal memory required by the optimal sequential schedule, the algorithm produces schedules with good speedups.
2) a technique able to deal with very large task graphs. Previous work based on maximal topological cuts (max-cuts) cannot deal with thousand task graphs due to the complexity of that operation.
3) a complete implementation and experimental results that support the aforementioned claims.

The article is organized as follows. In Sec. II, we start by reviewing the related work. Sec. III defines the considered model of computation and recalls some basic notions of scheduling. Sec. IV introduces *schedule graphs*, an expressive representation of task graphs allowing us to reason on schedules and their memory usage. It also sketches the technique used to find optimal sequential schedules. Sec. V presents three schedulers derived from list scheduling to compute a parallel scheduling ensuring minimal memory peak. We provide benchmark comparisons and other experimental results in Sec. VI. Finally, Sec. VII concludes and outlines future work directions.

II. Related Work

Finding the minimum memory needed to execute a task graph is a variation of the pebble game [5], which is an NP-complete problem. Previous work presented approaches to find *sequential* schedules minimizing the memory requirements for some classes of task graphs. The algorithm presented in [10] finds an optimal sequential schedule in $\mathcal{O}(n^2)$ for tree-shaped task graphs [10]. This was extended by [11] to find an optimal sequential schedule in $\mathcal{O}(n^3)$ for Series-Parallel Directed Acyclic Graphs (SP-DAGs). Our graph transformation approach [6], [7] (see Sec. IV) solves optimally the problem for a much larger class of task graphs that includes SP-DAGs as well.

To the best of our knowledge, no existing work in the literature solves the problem of parallel scheduling a given task graph satisfying the minimal memory peak constraint (the one obtained by the sequential schedule) and minimizing the makespan, neither in the static nor in the dynamic case.

Some approaches try to reduce the memory footprint of parallel schedules. For instance, Ramakrishman et al. [12] consider data-intensive scientific workflows onto distributed resources. They reduce the storage requirements by removing data as soon as it is no longer needed and by using a memory aware scheduler. Sergent et al. [13] control the memory footprint of distributed applications by throttling the task submission flow rate.

Closer to our goal is work aiming at finding parallel schedules of task graphs meeting specific memory constraints. They all add fictitious dependence edges to forbid schedules that fail to respect the constraints. Sbîrlea et al. [14] consider a dataflow model whose tasks exchange items of the same size. They use a heuristic algorithm to add sufficient dependence edges to fulfill the memory constraints. When it fails, they use an exact solution based on an Integer Linear Programming (ILP) formulation initiated with the best result identified by the heuristic. Marchal et al. [15] consider general Directed Acyclic Graphs (DAGs) of tasks and use an ILP formulation to find the maximal memory peak of any parallel traversal. If it exceeds the memory constraints, a new dependency edge is added to the DAG using several heuristics. The process is iterated until the constraints are met. Different scheduling techniques can be applied to the resulting graph. Bathie et al. [16] refine the previous technique by taking into account the number of processors and propose a more efficient algorithm for SP-DAGs. However, none of these approaches can deal with large task graphs of arbitrary form. They also often fail when the memory constraint is close to the minimal memory peak requirement.

III. Background

We consider applications represented as task graphs. We first recall the main characteristics of this model and next the principles of list scheduling, especially using *bottom levels*.

A. Task Graphs

The task graph model we consider consists of a DAG where vertices represent tasks and edges represent FIFO communication buffers between tasks. The atomic execution of a task, referred to as a *firing*, consumes data from all its incoming edges (its inputs) and produces data to all its outgoing edges (its outputs). The data unit is called a *token*, and the same unit is used for all measures (production, consumption, memory peak). The number of tokens consumed (resp. produced) on a given edge at each firing is indicated on the edge and is called the consumption (resp. production) *rate*. A task can fire only when all its input edges contain enough tokens. It then reads them, execute its code, and write its result as new tokens sent to its outgoing edges according to their rates.

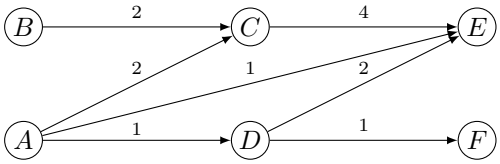

Fig. 1: A simple task graph example.

Fig. 1 presents a simple task graph with six tasks A, B, C, D, E, and F. When fired, task D consumes 1 token on its input edge and produces 2 and 1 tokens on its two output edges, which will be eventually consumed by E and F respectively. There are 14 possible sequential schedules for this task graph: $(A;D;F;B;C;E)$, $(B;A;C;D;E;F)$, *etc.*. In general, a connected graph of n tasks can have up to $(n-1)!$ schedules.

During the execution of a task graph, the occupied memory is the sum of all the tokens present on all its edges (*i.e.*, buffers). We assume that the FIFO buffers are allocated in the same *global shared memory*.

The *memory peak* of a schedule is the maximum memory occupancy reached during the execution of the schedule. It can also be defined as the minimum amount of memory necessary for the schedule to operate effectively.

In a parallel context, as a secondary objective, we also strive to minimize the *makespan* of our schedules, that is the completion time of its last task.

B. List scheduling and bottom levels

List scheduling [17] is a fast scheduling algorithm, which gathers the tasks having all their dependencies completed in a so-called *ready list*, and orders them according to some criterion. The tasks are scheduled one by one according to the order of the ready list, on the available processors. Whenever a task X completes, the ready list is updated by adding all the tasks enabled by the completion of X.

The ready list can be sorted according to different criteria: task deadlines, duration, or any objective function. In one of our strategies, we order the tasks according to their *bottom level* (or b-level) [9]. The bottom level of a task X, denoted $b\ell(X)$, is the longest path (in terms of execution time) from X to the end of the DAG, including the execution time of X itself. Bottom levels have proved to work well when the overall objective is the minimization of the makespan. We assume that each task of the DAG is annotated by an estimation of its duration.

Since task durations are hard to estimate precisely, static scheduling often exhibits poor performance, so we target *dynamic scheduling*, for which list scheduling is perfectly adequate. The dynamic scheduler is invoked each time a task completes. The selection of a processor to execute the first task in the ready list can follow a specific strategy, *e.g.*, select the processor yielding the earliest finish time on heterogeneous processors [18]. In the following, we assume homogeneous multi-processor systems and select any available processor. It would be straightforward to adapt our list-scheduling algorithms to heterogeneous platforms by taking into account a table associating to each pair (task, processor) a possibly different execution time.

In a standard dynamic list scheduler, any task of the ready list can be scheduled. However, in our work, where strict memory constraints must be met, the scheduler may have to wait or choose another task than the first one.

IV. Schedule graphs and optimal sequential schedules

We start from an optimal sequential schedule having the minimal memory peak needed for any execution (sequential or parallel). Even if, in general, a parallel execution has much larger memory requirement, we can derive from a sequential schedule a parallel one having the same memory requirement.

Finding the optimal sequential schedule is an NP-complete problem. We rely on our approach presented in [7], which is able to find optimal sequential schedules for a large class of task graphs. We present in this section the concepts that we re-use and outline the technique employed to identify these schedules. The interested reader will find further details, explanations and proofs in [7].

A. Schedule Graphs

Tasks can allocate and deallocate memory according different patterns. Two classic local memory models are (i) the consumed-before-produced (CBP) model where a task first reads and consumes (*i.e.*, frees) its input tokens, then executes its code, and finally produces its result as output tokens; (ii) the produced-before-consumed (PBC) model where a task reads and keeps its input tokens, executes its code and produces its result before consuming its input. Beyond these models, tasks may also require some additional memory to perform their internal computation. The method proposed in [7] can accommodate all these memory models.

Schedule graphs are a formalism that can represent these different behaviors and permits to reason about the memory peak of schedules. They are graphs where nodes are sequential schedules and edges represent dependencies between them. A

sequential schedule is either a single task A or a sequence of two schedules. It is formalized by the following grammar:

$$S ::= A \mid S_1; S_2$$

A schedule S has two key attributes π_S and ι_S, also written as $S^{\binom{\pi_s}{\iota_s}}$:

- its *peak* π_S, a natural number denoting the maximal memory peak reached during its execution;
- its *impact* ι_S, an integer denoting the final number of tokens added or removed after its execution.

Consider an execution where p is the current memory peak already reached and n is the current number of tokens in memory. After the execution of $S^{\binom{\pi_s}{\iota_s}}$, the new peak will be $\max(p, n+\pi_S)$ and the new number of tokens will be $n+\iota_S$.

The memory peak of a complete schedule can be computed using the following associative operation:

$$X^{\binom{\pi_x}{\iota_x}}; Y^{\binom{\pi_y}{\iota_y}} = (X;Y)^{\binom{max(\pi_x, \pi_y+\iota_x)}{\iota_x+\iota_y}} \quad (1)$$

The impact of the sequential execution of nodes X and Y is the sum of their impacts, whereas its memory peak is the maximum of the peak reached during X and the peak reached during Y taking into account the impact of X.

Assuming a PBC model, the schedule graph of the previous task graph is shown in Fig. 2.

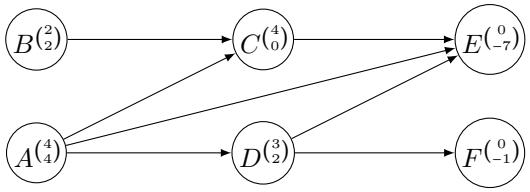

Fig. 2: The schedule graph for the task graph of Fig. 1.

The task D (consuming 1 token and producing 3 in Fig. 1) is represented as $D^{\binom{3}{2}}$: its local peak is 3 since it produces its results before consuming its input and its impact (after consuming its input) is 3 – 1 = 2.

The peak of the schedule $A;B;C;D;E;F$ of graph in Fig. 2 is 10 and can be computed using Eq. (1) as:

$$\begin{aligned} & A^{\binom{4}{4}}; B^{\binom{2}{2}}; C^{\binom{4}{0}}; D^{\binom{3}{2}}; E^{\binom{0}{-7}}; F^{\binom{0}{-1}} \\ = \; & (A;B)^{\binom{6}{6}}; (C;D)^{\binom{4}{2}}; (E;F)^{\binom{0}{-8}} \\ = \; & (A;B;C;D)^{\binom{10}{8}}; (E;F)^{\binom{0}{-8}} \\ = \; & (A;B;C;D;E;F)^{\binom{10}{0}} \end{aligned}$$

Fig. 3 shows the memory profile of $A;B;C;D;E;F$.

The *memory usage* at one point of the schedule is the sum of all the impacts of the preceding completed tasks. In the former schedule, the memory usage reached after the execution of D is $\iota_A + \iota_B + \iota_C + \iota_D = 4 + 2 + 0 + 2 = 8$.

The *relative peak* of a task X in a schedule is the peak reached by adding the local peak of X to the memory usage reached just before. The relative peak of D in $A;B;C;D;E;F$ is $\iota_A + \iota_B + \iota_C + \pi_D = 4 + 2 + 0 + 3 = 9$.

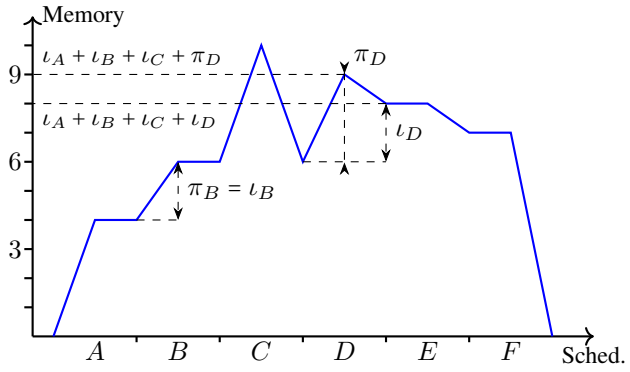

Fig. 3: The memory profile of $A;B;C;D;E;F$

The *global memory peak* of a schedule is its highest relative peak. The global memory peak of $A;B;C;D;E;F$ is 10; it is reached during the execution of C. This is the minimal global peak of all the 14 possible schedules. Some, like $B;A;C;D;F;E$ or $A;B;C;D;F;E$, enjoy also this minimal peak; others, like $A;B;D;C;E;F$, have a higher peak.

B. Schedule graph compression

Transitive reduction, node clustering and sequentialization, are three transformation rules that can be applied to compress the schedule graph while preserving its minimal memory peak. We sketch them in turn.

a) Transitive Reduction: In a schedule graph, edges do not carry other information than dependencies. Edges representing redundant dependencies can be removed. For instance, the edge $A \to E$ in Fig. 2 can be removed because it does not add any scheduling constraint. This transformation is the classic transitive reduction [19]. It returns the graph with the fewest edges having the same reachability relation. It does not directly suppress possible schedules but it makes the subsequent transformations more effective.

b) Node Clustering: Let A and B be two connected nodes such that B is the only successor of A. This dependency implies that all nodes executed between A and B can also be executed before A. We show in [7] that if $0 \leq \iota_A \wedge \pi_A \leq \pi_B + \iota_A$, then for any S the memory peak of $A;S;B$ is greater than or equal to the peak of $S;A;B$. In such case, there is no gain in interleaving S between A and B, hence and A and B can be clustered into a single node $[A;B]$. The resulting graph has one node less and much less schedules. A dual clustering rule applies when A is the only predecessor of B.

c) Node Sequentialization: Let A and B be two nodes with the same predecessor. This dependency ensures that every schedule node that can be executed after B and before A can also be executed *after* A and B. We show in [7] that if $\iota_A \leq 0 \wedge \pi_A \leq \pi_B$, then for any S the memory peak of $B;S;A$ is greater than or equal to the peak of $A;B;S$. In such case, the two nodes can be sequentialized by adding a new edge from A to B. Sequentialization eliminates useless schedules and creates new clustering opportunities. A dual sequentialization rule applies when A and B have the same successor.

d) Global algorithm: These transformation are combined within a compression algorithm that stops when no further clustering, sequentialization, or reduction are possible. Its global worst case time complexity is quartic in the number of nodes. This algorithm is quite effective and compresses a large class of graphs into a single node representing an optimal schedule. In particular, we have formally proved that it always compresses any SP-DAGs into a single node [7] .

C. Branch and Bound Algorithm

On some DAGs the previous algorithm is not able to compress the graph into a single node. In this case the result of the compression is a reduced schedule graph, whose optimal schedule and peak remain to be found.

To deal with incompressible DAGs, we have designed an optimized branch and bound (B&B) algorithm. It builds the tree of all schedules in a depth-first manner. The depth of this tree is equal to the number of tasks of the reduced schedule graph and each branch corresponds to a decision to schedule one of its node, chosen in the so-called ready list that contains all the nodes having all their predecessors already scheduled.

The B&B algorithm finds the optimal memory peak of applications up to 50 nodes in a second, but its computational time generally explodes past 100 nodes. Combined with the compression algorithm, many task graph applications can be analyzed optimally. At worst, a timeout always permits to retrieve an overestimation of the peak, corresponding to the best schedule found so far.

V. Parallel scheduling for minimal memory peak

Finding local compression rules, as in [7], which preserve the optimal memory peak is much more complex when targeting a parallel schedule. The main reason is the central role of execution time, which is not a concern for contiguous sequential schedules. In addition, there is no hope of finding the optimal schedule w.r.t. the makespan because the problem is NP-complete in the static context, and undecidable in the dynamic context because the estimates of task durations are usually very approximate. Nevertheless, we propose in this section several dynamic parallel scheduling algorithms that are optimal w.r.t. the memory peak, in the sense that no alternative schedule (sequential or parallel) has smaller memory requirements. Moreover the parallel schedules are efficient in the sense that they achieve good speedups over the sequential schedule.

The reason why we obtain good speedups is illustrated in Fig. 4, which shows the memory profile of the optimal sequential schedule of the QMF_235 benchmark (see Sec. VI). Here, the memory peak 22 is reached at 12 different points, but before and after each such point, there is room for scheduling in parallel the tasks, which is exactly what our dynamic scheduling algorithms do. This is illustrated in Fig. 5, which shows the memory profile of the parallel schedule on 4 processors of the same QMF_235 benchmark with the same memory limit. Instead of seeing a "mountain" with several memory peaks, we see a "plateau", and the average number of processors used is 2.26.

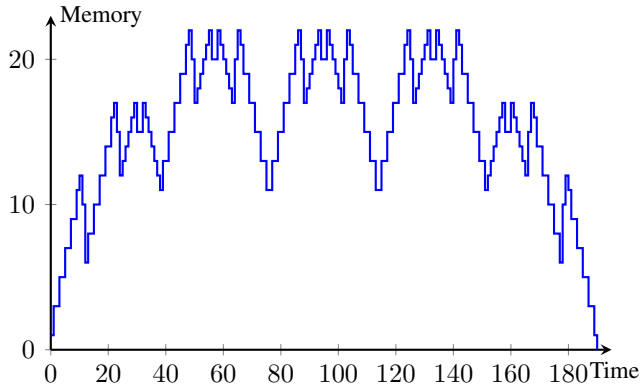

Fig. 4: Memory profile of the optimal sequential schedule of QMF_235.

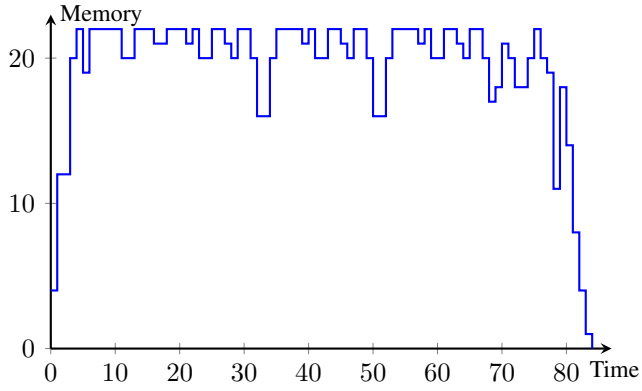

Fig. 5: Memory profile of a parallel schedule of QMF_235.

We start by evaluating the optimal sequential schedule using the approach sketched in Sec. IV. In most cases, and in particular in all the benchmarks of Sec. VI, we find a schedule that is optimal for the memory peak. Otherwise, we start with the best approximation returned by the B&B algorithm. We denote this sequential schedule as S_o, and its corresponding memory peak as π_{S_o}.

A key feature of our dynamic parallel algorithms is that we rely on S_o to schedule dynamically the tasks while guaranteeing that the memory usage does not exceed the memory constraint given by the user, supposed to be greater than or equal to π_{S_o}. In this sense, our algorithm is a scheduling algorithm under (memory peak) constraint.

We assume that the user provides a memory constraint, denoted Π, which is the memory limit that the parallel schedule should never exceed. This limit is assumed to belong to the interval $[\pi_{S_o}, \pi_{CP}]$, where π_{S_o} and π_{CP} are respectively the memory peak reached by the optimal sequential schedule, and the memory peak reached by a classical Critical Path (CP) scheduling algorithm using only bottom levels:

$$\pi_{S_o} \leq \Pi \leq \pi_{CP} \tag{2}$$

We propose three variants to compute dynamically a parallel schedule ensuring strict memory constraints. All of them rely

on the principles of ready list scheduling. The pseudo-code of Alg. 1 is common to the three of them.

Algorithm 1: Generic dynamic memory-aware scheduler

```
1  if (a task completes) then
2      p ← nbIdleProcessors();
3      ℓ ← getReadyList();
4      while (p > 0 and size(ℓ) > 0) do
5          X ← pop(ℓ);
6          if canSched(X) then        ▷ check new memory
               peak if running X immediately
7              p ← p − 1;
8              launch(X); ▷ X is launched immediately
                   on any idle processor
9          else
10             break or continue ; ▷ depends on variant
```

The dynamic scheduler is triggered as soon as any task completes its execution. We assume that, upon this event, the number of idle processors is increased, the ready list is updated with the new ready tasks and is reordered. We do not describe these operations here. The updated ready list is copied in ℓ and number of the idle processors in p. If ℓ is not empty and at least one processor is available, the first task X is extracted from ℓ. Before scheduling X, we check that the memory constraint Π will not be exceeded, neither when running X nor when scheduling the rest of the task graph. This is the purpose of the canSched() function, which uses the current memory occupancy, the memory peak of X, and the rest of optimal sequential schedule (its suffix). If the condition is true, the task is scheduled and the process iterated. Otherwise, the iteration either stops or continues depending on the variant.

The three variants differ by the ordering of the reading list, the memory check (the canSched() function), and the content of the **else** branch (**break** or **continue**). We explain them using the same following example. We consider the optimal sequential schedule

$$S_o = A_1; \ldots; A_m; R_1; R_2; N_3; R_4; N_5, \ldots; N_n \qquad (3)$$

and we assume that, when the dynamic scheduler is called, all the tasks A_i have already been scheduled. The suffix of S_o that remains to be scheduled, denoted as S^r, is made of the n tasks R_1 to N_n, where R_1, R_2, and R_4 are ready whereas none of the N_i are.

A. Ready list sorted by the sequential schedule

In the first variant, the tasks are scheduled in exactly the same order as in S_o. Recall that S^r is the suffix of S_o that remains to be scheduled: then the ready list is the largest prefix of S^r only made of ready tasks. In our example, the ready list ℓ is $[R_1; R_2]$. Since N_3 is not ready, the task R_4 does not belong to ℓ. If S^r started with a non ready task N, then ℓ would be empty, even if there were several ready tasks occurring after N in S^r. In that case, the scheduler would wait until all the predecessors of N have completed and N becomes ready.

The condition canSched(X) checks if the memory peak of X, added to the current memory used at time t, denoted transientMem(t), does not exceed the memory constraint Π:

$$\text{canSched}(X) \stackrel{\text{def}}{=} \text{transientMem}(t) + \pi_X \leq \Pi \qquad (4)$$

The memory used at time t is denoted transientMem(t) because it is the memory used while some tasks are still running. It is the sum of the results produced so far (*i.e.*, the sum of impacts of all completed tasks) plus the memory peaks of the running tasks. Formally,

$$\text{transientMem}(t) \stackrel{\text{def}}{=} \sum_{Y \in \text{completed}(t)} \iota_Y + \sum_{Z \in \text{running}(t)} \pi_Z \qquad (5)$$

For instance, in Fig. 6 we have transientMem$(6) = \iota_A + \iota_B + \iota_C + \pi_D = 9$ because D is still running at $t = 6$.

Fig. 6 depicts the parallel schedule on two processors (top part) for the task graph of Fig. 1, with its aligned memory profile (bottom part) with a memory constraint $\Pi = 10$. At time $t = 3$, task B finishes so the dynamic scheduler is invoked. The ready list is $\ell = \{C, D\}$, which complies with the total ordering of the suffix $C; D; E; F$ of the sequential schedule $A; B; C; D; E; F$. The candidate task is therefore C, and we have transientMem$(t) = 6$ and $\pi_C = 4$; hence Eq. (4) is satisfied, so C is launched, here on processor 1. Task D is the new candidate, but it does not pass the test (transientMem$(t) = 10$ and $\pi_D = 3$). In this first variant, the **else** branch of Alg. 1 is a **break**. The launching of D is postponed until some future task completes and transientMem(t) decreases enough.

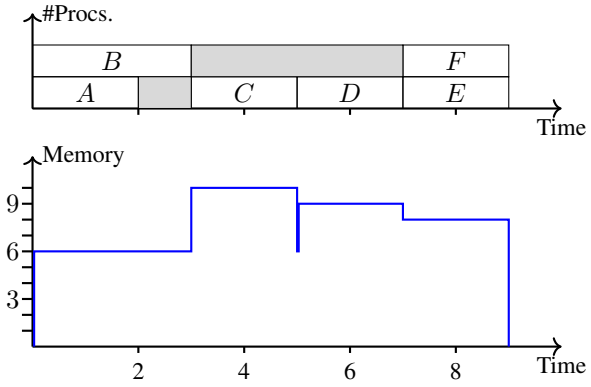

Fig. 6: Parallel schedule for the schedule graph of Fig. 1.

We know that transientMem(t) eventually decreases to the same amount of memory (represented by the sum of impacts of all completed tasks) as in S_o before executing a task (*e.g.*, D above). This ensures that it will **always** be possible to execute a task (like D) provided that the memory constraint Π is greater than or equal to π_{S_o} (which is assumed to be the case, see Eq. (2)).

This scheduling method is effective because it always satisfies the given memory constraint, even the minimal possible one. It does not rely on execution time estimations and

nevertheless it yields reasonable speedups, as we will see in Sec. VI. However, we can get much better makespans and speedups by taking advantage of the notion of bottom level in the variant presented next.

B. Ready list sorted by bottom levels

In our second variant, the ready list contains *all* the ready tasks, ordered according to their bottom level (see Sec. III). As before, we must check that the scheduling of the task does not exceed the memory limit, *i.e.*, satisfies Eq. (4). However, a second dynamic check is necessary. In Sec. V-A, we knew that the rest of the tasks to schedule was a suffix of S_o, so, at worst, the memory constraint would be satisfied by waiting for all tasks to complete and then executing the suffix sequentially. Here, the tasks remaining to schedule may not be a suffix of S_o and the task launched may have a larger impact than the first one of S_o's suffix.

In the example of Eq. (3), if we assume that the bottom levels are such that $b\ell(R_4) > b\ell(R_1) > b\ell(R_2)$, then the ready list is $[R_4; R_1; R_2]$. After scheduling R_4, the tasks remaining to schedule will be $R_1; R_2; N_3; N_5, \ldots; N_n$, which is equal to $S^r \setminus R_4$. We must check that the memory used reached after all the past tasks complete (*i.e.*, $A_1; \ldots; A_m; R_4$) allows the execution, as least sequentially, of $S^r \setminus R_4$ while satisfying the memory constraint. For this reason, the condition $\text{canSched}(X)$ becomes:

$$\begin{aligned}\text{canSched}(X) \stackrel{\text{def}}{=} \quad & \text{transientMem}(t) + \pi_X \leq \Pi \quad \wedge \\ & \text{projectedMem}(t) + \iota_X + \pi_{S^r \setminus X} \leq \Pi\end{aligned} \tag{6}$$

The first clause is the same as in Eq. (4). The second clause guarantees that, if we wait for all the tasks currently running to complete, then the memory usage reached at that point (denoted as $\text{projectedMem}(t)$) allows all the remaining tasks to be scheduled sequentially. In other words, the memory usage reached after all the currently running tasks complete (including X), plus the memory peak of the tasks remaining to schedule (*i.e.*, $S^r \setminus X$), also satisfy the memory constraint.

We denote the memory usage reached when all the running tasks complete as $\text{projectedMem}(t)$ which is formally defined as:

$$\text{projectedMem}(t) \stackrel{\text{def}}{=} \sum_{Y \in \text{completed}(t) \,\cup\, \text{running}(t)} \iota_Y \tag{7}$$

For instance, in Fig. 6 we have $\text{projectedMem}(6) = \iota_A + \iota_B + \iota_C + \iota_D = 8$ which is the memory usage after D completes.

Finally, if the first task of ℓ is not schedulable, then the next one is considered (the **else**-branch of Alg. 1 is a **continue**).

Two optional refinements can be brought to this algorithm. Let X be the task to schedule (*i.e.*, X is the head of ℓ) and let the suffix be $S^r = S_1; X; S_2$, with S_1 and S_2 two sub-schedules.

The first refinement is to notice that to check the second clause of $\text{canSched}(X)$, we do not need to compute the peak of all the tasks to schedule ($S^r \setminus X$, hence here $S_1; S_2$). It is sufficient to check that the memory peak reached after executing S_1 remains below Π. Indeed, at this point, we reach the same memory usage as S_o and the tasks remaining to schedule, S_2, is a suffix of S_o.

The second refinement is to notice that the schedule S_1 may not be optimal now that X has been executed out of order w.r.t. S_o can thus recompute a memory-peak-optimal sequential schedule of the tasks in S_1 before performing check the second clause of $\text{canSched}(X)$. This can be done using the optimal technique of Sec. IV or approximated by an efficient linear heuristics.

The first refinement improves slightly the execution time of our dynamic scheduling algorithm because the memory peak is (linearly) computed on a shorter sequence. The second one improves the makespan of the obtained schedules (and significantly on some benchmarks), but it incurs a significant execution overhead. Its usage therefore depends on size of the task graph and on the granularity of the tasks. None of them affect the correctness of our scheduling algorithm.

The dynamic scheduling algorithm presented in this section and relying on bottom levels always succeeds in producing a schedule satisfying the given memory constraint, even when Π is the optimal memory peak of the sequential schedule. As will be shown in Sec. VI, this algorithm yields much better speedups than the algorithm of Sec. V-A.

C. Ready list mixing bottom levels and sequential schedule

In Sec. V-B we rely on an additional test in order to schedule tasks in the decreasing order of their bottom level. Actually this test allows us to consider *any other ordering* of the ready list.

A natural idea is to combine both variants presented in Sec. V-A and Sec. V-B by crafting an objective function that combines the two orderings of tasks, w.r.t. S_o and w.r.t. the bottom levels.

Regarding the ordering w.r.t. S_o, we assign to each task A in S^r the value

$$O_{S_o}(A) = \frac{1}{i} \quad \text{where } i \text{ is } A\text{'s index in } S^r. \tag{8}$$

As a result, the first task in S^r is assigned the value 1 while the last one is assigned a value closer to 0. This fraction belongs to $[\frac{1}{|S^r|}, 1]$. In our example, the scores of R_1, R_2 and R_4 are 1, $\frac{1}{2}$, and $\frac{1}{4}$ respectively.

Regarding the ordering w.r.t. bottom levels, we normalize the bottom level of any task A according to the formula

$$O_{b\ell}(A) = \frac{b\ell(A)}{\max_{X \in \ell} b\ell(X)} \tag{9}$$

which assigns the highest score of 1 to the task with the largest bottom level. The scores of other tasks are close to 1 if their bottom level is comparable to the largest one and close to 0 if it is much smaller.

These two metrics O_{S_o} and $O_{b\ell}$ are combined in a weighted sum:

$$Score(A) = r\, O_{S_o}(A) + (1 - r)\, O_{b\ell}(A) \tag{10}$$

where the multiplier r is defined as

$$r = \frac{\pi_{CP} - \Pi}{\pi_{CP} - \pi_{S_o}} \tag{11}$$

which is guaranteed to belong to $[0, 1]$ because Π is assumed belong to the interval $[\pi_{S_o}, \pi_{CP}]$.

The value of r depends on where the memory constraint Π lies within the interval $[\pi_{S_o}, \pi_{CP}]$: the closer Π is to π_{S_o}, the closer r is to 0, and conversely, the closer Π is to π_{CP}, the closer r is to 1. A midway memory constraint results in $r = 1 - r = 0.5$.

If the given memory constraint Π is equal to π_{CP}, then $r = 0$ and $Score(A) = O_{b\ell}(A)$, meaning that the ready tasks are ordered according to their bottom level. In this case our algorithm is exactly that of CP scheduling, presented in Sec. V-B.

If Π is equal to π_{S_o}, then $r = 1$ and $Score(A) = O_{S_o}(A)$, meaning that the ready tasks are ordered according to their order appearance in S_o. In this case our algorithm differs from that of Sec. V-A because the ready list ℓ contains all the ready tasks (instead of the largest prefix of S^r containing only ready tasks), and when a task is not schedulable the next one is considered (the **else**-branch of Alg. 1 is a **continue**). Compared to the previous variant, this hybrid variant may yield better or worse speedups depending on graphs. Both variants are comparable on average.

The costs of the dynamic tests are small. The only costly option is to recompute an optimal sequential schedule when the task chosen to be launched does not occur first in the S^r (second refinement in Sec. V-B). In the experiments, we do reschedule but using a linear time heuristic.

VI. Experiments

We have conducted a series of experiments to evaluate the three schedulers proposed in Sec. V and to compare them with the previous state-of-the-art methods [15], [16]. We first describe our experimental setup (benchmarks, parameters and implementation) and then the results themselves.

A. Experimental setup

a) Benchmarks: We use two benchmarks: Pegasus [20] that generates scientific workflows and Quadrature Mirror Filterbanks (QMFs) that are signal processing applications.

As in [15], three Pegasus applications are considered, with the same parameters: LIGO, MONTAGE, and GENOME. For each application, we generate 20 task graphs of 50 nodes and 20 task graphs of 100 nodes, resulting in a set of 120 graphs in total. Pegasus provides random plausible memory sizes of data transfers and execution times of tasks.

These applications load files whose data is shared by several tasks. The standard task model of Sec. III-A does not directly support data sharing. Thus, we use the encoding of tasks with shared output data as described in [15]. This encoding introduces two dummy tasks, one which plays the role of a memory loader and the other one of a deallocator. As proved in [15], this represents faithfully the memory peak of applications with shared data.

From this set, we differentiate the *trivial* graphs for which the minimal sequential memory peak is satisfied by the standard CP scheduling algorithm. On the 120 samples to evaluate, this leaves us with 89 (4 processors) and 107 (8 processors) *non-trivial* graphs.

In addition, as in [21] and [7], we take as benchmark the QMFs signal processing application to evaluate the scalability of the different algorithms. It is a parameterized set of SDF graphs [22] that differ from one another by their *rates* (amount of data transfers) and *depth* (imbrication levels). Depending on these parameters, their translations into task graphs may produce huge graphs. The sizes of the four graphs we consider vary from 190 to 50.000 tasks. The execution times of their tasks are not provided, so we assigned a unit time duration to each task.

b) Parameters: We consider two shared-memory architectures, respectively with 4 and 8 processors, which are common numbers in current computers. We consider two values for the memory constraint Π. Firstly, we take the memory peak of an optimal sequential schedule. For all the task graphs of the two benchmarks, the method outlined in Sec. IV always succeeds in finding a memory-optimal schedule, hence providing the minimal memory peak. Secondly, we take a relaxed "*midway*" memory constraint, consisting of the average between the minimal memory peak and the memory peak obtained by the CP scheduling algorithm without any memory constraint.

Finally, since the algorithms from the literature are computationally expensive, we set a timeout to 600 seconds.

c) Implementation: We implemented the previous state-of-the-art method [15][1], refined in [16][2] using the HiGHS linear solver [23]. This method first computes a max-cut of the graph in order to evaluate its maximal parallel memory consumption. The max-cut is defined by an ILP formulation that is usually too costly to solve. Instead, a linear relaxation heuristic is used, which yields a cut that is not guaranteed to be maximal. If the cut exceeds the memory constraint, dependencies are added as fictitious edges to decrease the memory consumption. This is done according to two heuristics: MINLEVELS (ML), which favors bottom levels, and RESPECTORDER (RO), which respects a given sequential order. For RO, [15] considers 21 combinations of DFS and BFS schedules of the graph, none of them being obtained with memory costs taken into account. We also evaluate a new hybrid method (denoted as RO+S_o) where our optimal sequential order (S_o) is used for the RO heuristic of [15]. For both methods, once enough dependencies have been added to the graph, it can be fed to any scheduler. In our experiments, we use a classical CP scheduler without any memory checks.

We have implemented the three scheduling variants of Sec. V. For the variants presented in Secs. V-B and V-C and the Pegasus benchmark, the second optional refinement (recomputation of remaining memory peak with a linear heuristics, see Sec. V-B) is activated.

All these schedulers process the dummy tasks implementing data sharing in a specific way. A task loading shared memory

[1] [15] https://gitlab.inria.fr/lmarchal/memdag

[2] [16] https://github.com/GBathie/PMaxcut

TABLE I: *Percentages of successes to meet the minimal and midway memory constraints with* 4 *and* 8 *processors on respectively* 89 *and* 107 *non-trivial Pegasus graphs.*

objective	4 processors				8 processors			
	ML; CP	RO; CP	RO+S_o; CP	Sec. V	ML; CP	RO; CP	RO+S_o; CP	Sec. V
min. peak	0%	7%	49%	**100%**	6%	21%	66%	**100%**
midway	4%	33%	75%	**100%**	8%	44%	92%	**100%**

is scheduled right before the first task needing that data is executed. A task deallocating shared memory is executed as soon as the last task using that shared memory has completed.

All the above methods have been implemented using Python 3.10 with the graph library Networkx 2.8.8, and executed on a laptop equipped with an Intel® Core™ i5-8265U @1.60GHz processor, with 16 GB of RAM, running Linux Ubuntu 22.04. Our implementation is open-source and available online[3].

B. Experimental results

We evaluate the proposed methods on several criteria: success rate for a given memory constraint, speedup w.r.t. the sequential schedule, overhead w.r.t. the CP schedule, and the runtimes of preprocessing and scheduling.

a) Satifisfaction of the memory constraint: Table I shows the success rate of the different methods on non-trivial graphs. The success rate of our three variants (V-A, V-B, and V-C) is always 100%. This is expected since they have been designed to meet even the strictest memory constraint, provided they are greater then or equal to the memory peak of the sequential schedule.

The success rates of the two heuristics ML and RO from [15] are much lower: ML's success rate varies between 0% and 8%, whereas that of RO varies between 7% and 44%. Interestingly, the success rate of RO+S_o varies between 49% and 92%, which is much better than the original RO method.

The success rates of ML, RO, and RO+S_o increase when the memory constraint is relaxed (midway line of Table I). This is expected because the problem is less difficult to solve and needs less extra dependencies to be added to the graph. Their success rates also increase when the number of processors increases from 4 to 8. This is probably due to the bound on the number of cut edges that is higher when the number of processors increases. This provides more candidates to add the extra dependencies.

The use of linear relaxation heuristics in [15] creates a drawback not shown in Table I: some of the restricted graphs returned by the algorithm do not respect the memory constraint when they are scheduled. Although the constraint is usually exceeded by only a few percents, this makes the success rate drop. For example, Table I shows a 96% failure rate for ML with the midway memory constraint and 4 processors; among those 96% of failed graphs, 70% are failures of the ML algorithm and 26% are restricted graphs that are returned but fail to respect the constraints.

[3]https://gitlab.inria.fr/spades-pub/mastag

b) Speedup and overhead: In order to evaluate the parallel schedules obtained under memory constraints, we compare them according to their *speedup*, defined as the ratio of the makespan $C_{\max}$ of the sequential schedule to the makespan of the parallel one:

$$\text{Speedup} = \frac{C_{\max}(S_o)}{C_{\max}([\text{method}])} \tag{12}$$

Keeping the memory peak under a given constraint may reduce the potential degree of parallelism. In order to evaluate this aspect, we consider another evaluation metric: the *overhead* w.r.t. the CP schedule obtained without any memory constraint, formally defined as:

$$\text{Overhead} = \frac{C_{\max}([\text{method}])}{C_{\max}(\text{CP})} \tag{13}$$

Table II shows the speedup (and the overhead in parentheses) for the tenth samples of each graph from the Pegasus benchmark, for 4 processors, and under the minimal memory peak constraint. Note that MONTAGE_50_10 is the only trivial graph among the tenth samples.

As already noticed, ML never succeeds. RO also fails most of the time but achieves fair speedups when it succeeds. Specifically, both of these algorithms fail when they cannot add any extra dependency to limit the cut. RO+S_o yields good speedups in most cases. However, in two instances, it reaches the timeout (600 sec.). Its success rate is higher but it takes more time.

The last three columns of Table II show the results of our three scheduling variants from Sec. V. They always succeed and reach fair to excellent speedups (between 2.78 and 3.57 for the variants V-B and V-C), except for the two GENOME samples. The overheads presented inside parentheses are satisfactory (between 1.00 and 1.34 for the variants V-B and V-C), again except for the two GENOME samples, the parallelization of which definitely requires less strict memory constraints. The reason is that the sequential schedule of GENOME remains close to its minimal memory peak most of the time. As a result, it does not offer much opportunities of parallelization with such a harsh memory constraint.

The next two lines of Table II show the average and maximum speedups (and the overheads in parentheses) for all 120 samples, computed only for the algorithms that always succeed in meeting the given memory peak constraint. Variant V-A is the less effective one. Its strong point is that it is simpler and does not rely on any estimate of the execution times of the tasks. The other two variants V-B and V-C are competitive and comparable, with a slight edge to V-B. In this experiment,

TABLE II: *Speedup and overhead (in parentheses) of each method to meet the minimal memory peak with* 4 *processors on the Pegasus benchmark (the tenth sample of each application and the average/maximum on all and only successful samples*).

sample	RO; CP	RO+S_o; CP	Sec. V-A	Sec. V-B	Sec. V-C
LIGO_50_10	Fail	**3.09 (1.21)**	1.49 (2.51)	3.04 (1.23)	2.78 (1.34)
LIGO_100_10	Fail	T/O	1.45 (2.71)	**3.57 (1.10)**	3.53 (1.12)
MONTAGE_50_10	**3.57 (1.00)**	3.56 (1.00)	2.43 (1.47)	**3.57 (1.00)**	**3.57 (1.00)**
MONTAGE_100_10	Fail	3.12 (1.14)	2.44 (1.46)	**3.13 (1.14)**	3.03 (1.18)
GENOME_50_10	Fail	1.08 (2.93)	1.08 (2.93)	**1.26 (2.52)**	1.08 (2.93)
GENOME_100_10	Fail	T/O	1.08 (3.52)	1.26 (3.02)	**1.28 (2.97)**
average (all samples, 120)	N/A	N/A	1.64 (2.51)	**2.68 (1.75)**	2.62 (1.80)
maximum (all samples, 120)	N/A	N/A	2.52 (3.84)	**3.74 (3.49)**	3.64 (3.49)
average (∩ non-trivial successful, 6)	3.10 (1.17)	**3.26 (1.11)**	2.31 (1.65)	**3.26 (1.11)**	3.22 (1.13)
maximum (∩ non-trivial successful, 6)	3.36 (1.21)	**3.37 (1.17)**	2.50 (2.69)	3.36 (1.17)	**3.37 (1.27)**

TABLE III: *Speedup of selected methods to meet the minimal and midway memory peak with* 4 *and* 8 *processors on the Pegasus benchmark (the tenth sample of each application).*

sample	objective	4 processors				8 processors			
		RO; CP	RO+S_o; CP	Sec. V-B	Sec. V-C	RO; CP	RO+S_o; CP	Sec. V-B	Sec. V-C
LIGO_50_10	min. peak	Fail	**3.09**	3.04	2.78	Fail	4.11	**4.39**	4.22
LIGO_100_10		Fail	T/O	**3.57**	3.53	Fail	T/O	**6.04**	5.85
MONTAGE_50_10		**3.57**	3.56	**3.57**	**3.57**	4.71	5.21	**5.24**	**5.24**
MONTAGE_100_10		Fail	3.12	**3.13**	3.03	Fail	4.61	**4.62**	4.40
GENOME_50_10		Fail	1.08	**1.26**	1.08	Fail	1.08	**1.26**	1.08
GENOME_100_10		Fail	T/O	1.26	**1.28**	Fail	T/O	1.26	**1.28**
LIGO_50_10	midway	3.13	**3.56**	3.39	3.15	4.71	5.08	**5.35**	**5.35**
LIGO_100_10		Fail	**3.95**	3.55	3.86	Fail	T/O	5.89	**7.18**
MONTAGE_50_10		**3.57**	3.56	**3.57**	**3.57**	5.23	5.21	**5.25**	5.24
MONTAGE_100_10		**3.34**	3.33	**3.34**	3.23	5.65	**5.66**	**5.66**	5.35
GENOME_50_10		Fail	3.21	3.21	**3.22**	Fail	**5.48**	**5.48**	5.12
GENOME_100_10		Fail	Fail	**3.79**	3.40	Fail	**6.93**	**6.93**	5.76

TABLE IV: *Preprocessing and scheduling runtimes (in sec.) of the main methods to meet the midway memory peak with* 8 *processors on the QMF benchmark. The right-most column is a faster variant of Sec. V-C without the reordering refinement.*

sample	#tasks	CP	RO; CP			Sec. V-B		Sec. V-C	
		sched.	preproc.	sched.	preproc.	scheduling		scheduling	
						W/ refin.	W/O refin.	W/ refin.	W/O refin.
QMF_235_d2	190	0.01	358.29	0.01	0.04	0.21	0.01	0.11	**0.01**
QMF_235_d3	1,300	0.08	T/O	N/A	0.76	190.53	0.28	20.45	**0.19**
QMF_235_d4	8,250	1.56	OoM	N/A	25.17	T/O	12.00	789.69	**6.51**
QMF_235_d5	50,000	46.48	OoM	N/A	828.27	T/O	854.20	T/O	**396.28**

the use of the second refinement presented in Sec. V-C yields no significant difference on the speedup. Nevertheless, this refinement is quite effective for some of the QMF applications (not shown here).

The last two lines of Table II show the average and maximum speedups (and the overheads in parentheses) of all the methods on the subset of non-trivial samples for which both RO and RO+S_o succeed; this subset contains 6 samples among the 120. On this subset, all the evaluated methods (RO, RO+S_o, V-B, and V-C) achieve good and comparable speedups.

Table III shows the speedups of RO, RO+S_o, V-B, and V-C for both the minimal and midway memory constraints, and for 4 and 8 processors. With the midway memory constraint, good speedups are now obtained for the GENOME samples (between 3.22 and 3.79 on 4 processors). For the other samples, the speedups remain very good. The average speedup on all 120 samples of our variant V-B (not shown in the table) even becomes 3.57 for the midway memory constraint and for 4 processors (the average is 2.68 in Table II for the minimum memory constraint). These results confirm two expected behaviors: speedups always increase when adding more processors or when relaxing the memory constraint. The overheads (not shown in this table) improve accordingly.

c) Preprocessing and Scheduling runtimes: Finally, we evaluate the scalability of the state-of-the-art algorithm RO from [15] and of our variants V-C and V-C on four different QMF_235 graphs made of from 190 tasks to 50,000 tasks. Table IV gathers the runtimes of (1) the preprocessing for either restricting the graph (required by RO) or computing an optimal sequential schedule [7] (required by variants V-B and V-C), and (2) the scheduling runtimes for these three techniques. The scheduling runtime of the standard CP scheduling is also given as a reference. Given the large size of these task

graphs compared to the Pegasus benchmark, the timeout has been increased to $3,600$ sec. Recall that these runtimes are obtained using a non-optimized Python implementation on an average laptop.

The preprocessing for RO times out at $1,300$ tasks and then triggers an "Out of Memory" (OoM) exception when the task graph becomes too large for the linear solver. In contrast, finding the optimal sequential schedule succeeds for all samples, even if it takes time (828 sec.) for the 50.000 task graph.

Regarding the scheduling time, both variants V-B and V-C entail additional costs: if the task to schedule does not fulfill the memory constraint, the next one must be tried, and so on possibly until the end of the ready list. When the task graph and its ready lists are large, these failures cause important overheads. Similarly, the second refinement described in Sec. V-B is not practical for graphs with large ready lists. Without it, the runtimes are lower by two orders of magnitude. In Table IV, without the refinement, the variant V-B is about twice slower than variant V-C. Since those two variants only differ by the ordering of the ready list, this behavior can be explained by variant V-B being more prone to failures when trying to schedule the next task. Indeed, sorting by bottom levels does not take into account the impact of tasks on the memory while the sorting of variant V-C favors the optimal sequential schedule S_o when the memory constraint is tight. Thus, the variant V-C encounters less failure and runs faster than the variant V-B, but at the cost of slightly lower speedups as can been seen in Table III.

A potential solution to limit the scheduling overheads for huge task graphs would be to bound the size of ready lists[4]. This bound on the ready list could improve scheduling times, but it might also decrease the speedups. Finding the right bounds of the ready list to ensure good trade-offs between scheduling times and speedups is a topic for future research.

VII. Conclusion

In this article, we have proposed an approach to build dynamic parallel schedulers for task graphs that can guarantee the lowest possible memory peak. They rely on a sequential schedule minimizing the memory peak to postpone the parallel execution of tasks that would otherwise make it impossible to fulfill memory constraints. This approach is simple yet effective. It considered homogeneous multi-processor systems with shared memory but could easily be adapted to the heterogeneous case.

The experiments on scientific workflow applications show that our approach outperforms the state-of-the-art [15], [16]. Our algorithm always computes a parallel schedule that respects the given memory constraint. In contrast, the approach proposed in [15] fails nearly all the time when the constraint is close to the minimal peak. When the algorithm of [15] succeeds, it produces results that are on par with ours in terms of makespan and speedup. However, our preprocessing is shorter in time and better scales. We can handle graphs with up to $50,000$ tasks whereas the max-cut approach of [15] usually blows up past a few hundreds tasks.

We have implemented just a few simple heuristics and a single cost function. A short term future work is to evaluate other heuristics and combinations. As mentioned in Sec. VI, limiting the sizes of ready lists should be considered when dealing with huge task graphs. This point needs further study and experiments. From a practical standpoint, the next logical step is to embed our approach in a runtime scheduler such as StarPU [24]. A long term future work is to consider more complicated memory architectures like Non-Uniform Memory Access (NUMA) which makes the execution time of the tasks depend on their mapping.

In some contexts (*e.g.,* hard real-time or some scientific algorithms) the behavior of tasks in terms of execution time or memory consumption is known precisely. Knowing the memory consumption of a task as a function of time instead of as a constant peak could be used by our approach to implement more precise checks and to produce more efficient schedules. Knowing the precise execution time of tasks permits also to consider static scheduling. The first and most obvious advantage is efficiency: no tests have to be done dynamically. The optimal reordering (a refinement hinted at in Sec. V-B), too costly to be done dynamically, could be considered. It also makes it possible to statically produce multiple schedules based on different heuristics and to select the best one.

[4]To implement this option while guaranteeing the memory constraint, the ready list must always include the first task in S^{τ} if it is ready.

References

[1] S. Minakova, E. Tang, and T. Stefanov, "Combining task- and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Cham: Springer International Publishing, 2020, pp. 18–35.

[2] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Pipelined model parallelism: complexity results and memory considerations," in *27th International European Conference on Parallel and Distributed Computing*, Lisbon, Portugal, Aug. 2021. [Online]. Available: https://hal.inria.fr/hal-02968802

[3] ——, "Efficient combination of rematerialization and offloading for training dnns," in *Advances in Neural Information Processing Systems*, vol. 34. Curran Associates, Inc., 2021, pp. 23 844–23 857. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/c8461bf13fca8a2b9912ab2eb1668e4b-Paper.pdf

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software synthesis from dataflow graphs*. Kluwer Academic Pub., Hingham, MA, 1996.

[5] R. Sethi, "Complete register allocation problems," in *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 182–195. [Online]. Available: https://doi.org/10.1145/800125.804049

[6] P. Fradet, A. Girault, and A. Honorat, "Sequential scheduling of dataflow graphs for memory peak minimization," in *LCTES 2023 - 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. Orlando (FL), United States: ACM, Jun. 2023, pp. 76–86. [Online]. Available: https://hal.science/hal-04163123

[7] ——, "Graph transformations for memory peak minimization by scheduling," *ACM Trans. Embed. Comput. Syst.*, jan 2025. [Online]. Available: https://doi.org/10.1145/3707206

[8] T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321–1344, 1993.

[9] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961. [Online]. Available: https://doi.org/10.1287/opre.9.6.841

[10] J. W. H. Liu, "An application of generalized tree pebbling to sparse matrix factorization," *SIAM Journal on Algebraic Discrete Methods*, vol. 8, no. 3, pp. 375–395, 1987. [Online]. Available: https://doi.org/10.1137/0608031

[11] E. Kayaaslan, T. Lambert, L. Marchal, and B. Uçar, "Scheduling series-parallel task graphs to minimize peak memory," *Theoretical Computer Science*, vol. 707, pp. 1–23, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397517307053

[12] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensive workflows onto storage-constrained distributed resources," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 2007, pp. 401–409.

[13] M. Sergent, D. Goudin, S. Thibault, and O. Aumage, "Controlling the memory subscription of distributed applications with a task-based runtime system," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 318–327. [Online]. Available: https://doi.org/10.1109/IPDPSW.2016.105

[14] D. Sbîrlea, Z. Budimlić, and V. Sarkar, "Bounded memory scheduling of dynamic task graphs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 343–356. [Online]. Available: https://doi.org/10.1145/2628071.2628090

[15] L. Marchal, H. Nagy, B. Simon, and F. Vivien, "Parallel scheduling of dags under memory constraints," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 204–213.

[16] G. Bathie, L. Marchal, Y. Robert, and S. Thibault, "Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms," in *IPDPS - 2020 - IEEE International Parallel and Distributed Processing Symposium Workshops*, New Orleans / Virtual, United States, May 2020, pp. 1–10. [Online]. Available: https://hal.inria.fr/hal-03024626

[17] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal of Applied Mathematics*, vol. 17, pp. 416–429, 1969.

[18] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[19] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM J. Comput.*, vol. 1, no. 2, pp. 131–137, 1972.

[20] R. F. d. Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling research in distributed scientific workflows," in *2014 IEEE 10th International Conference on e-Science*, vol. 1, 2014, pp. 177–184.

[21] P. Murthy and S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 177–198, 2001.

[22] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[23] Q. Huangfu and J. A. J. Hall, "Parallelizing the dual revised simplex method," *Math. Program. Comput.*, vol. 10, no. 1, pp. 119–142, 2018. [Online]. Available: https://doi.org/10.1007/s12532-017-0130-5

[24] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877