Scheduling Task Graphs on Modern Computing Platforms

Bertrand SIMON

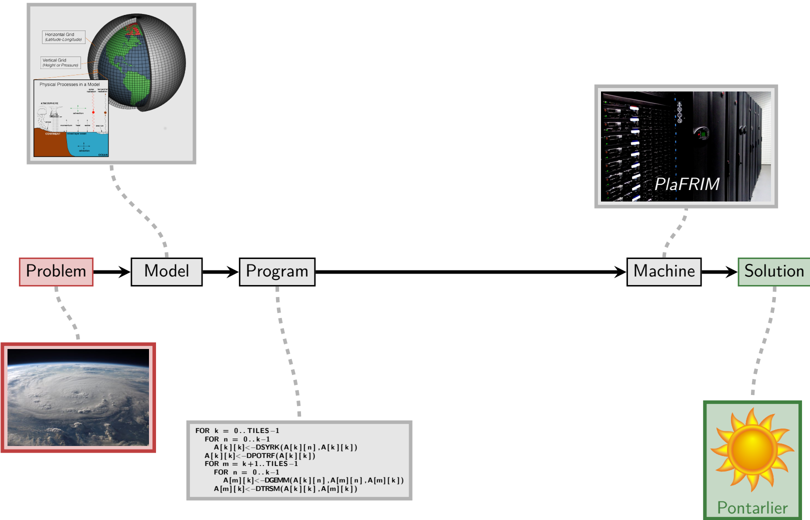
PhD Defense — 4 July 2018

École Normale Supérieure de Lyon, LIP laboratory

Commitee:

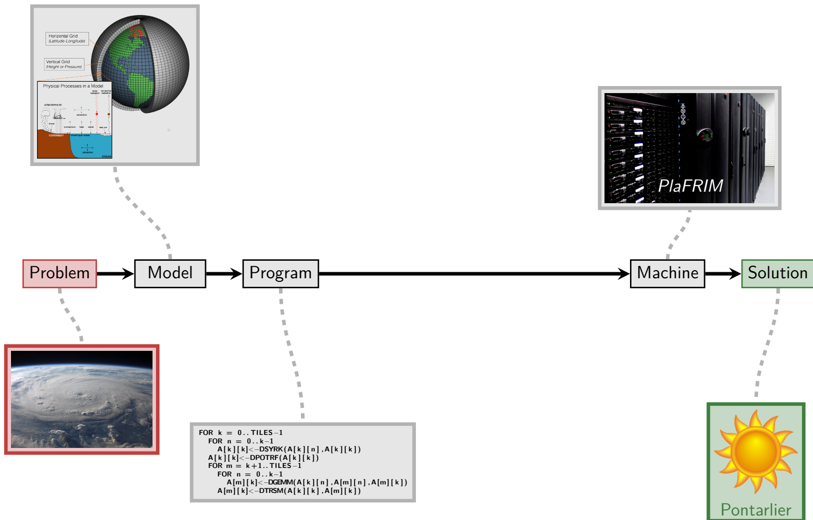
Claire	Hanen	Univ. Paris Nanterre	Reviewer
Safia	Kedad-Sidhoum	CNAM, Paris	Reviewer
Sascha	Hunold	TU Wien	Examiner
Uwe	Schwiegelshohn	TU Dortmund	Examiner
Loris	Marchal	CNRS & ENS de Lyon	Co-supervisor
Frédéric	Vivien	INRIA & ENS de Lyon	Thesis Director

Context



Context

Characteristics:
- many processing units



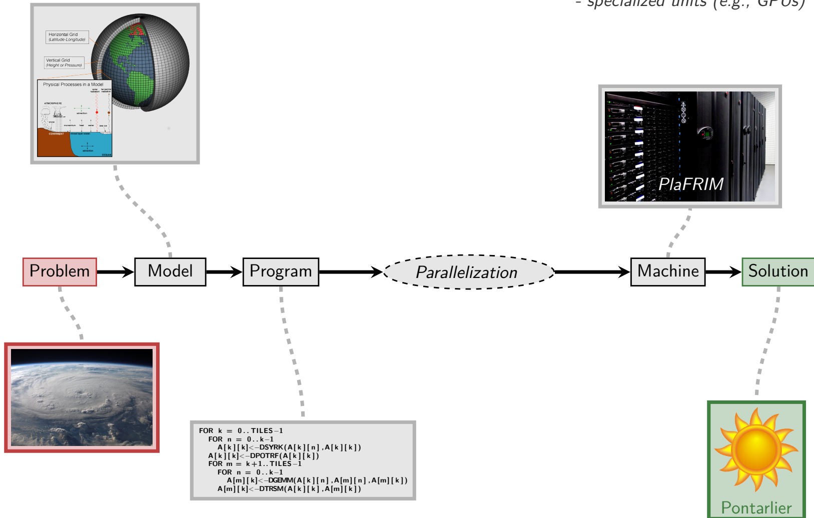
Context

Characteristics:
- many processing units

Problem → Model → Program → *Parallelization* → Machine → Solution

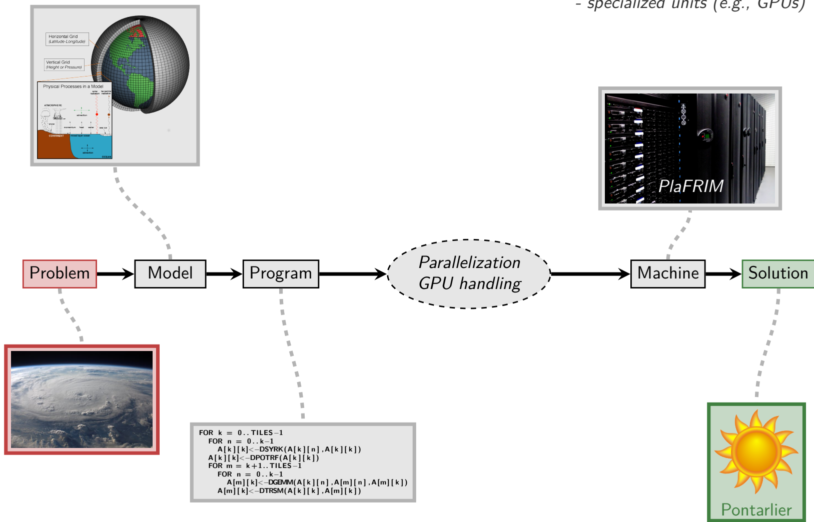
```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k]<-DSYRK(A[k][n],A[k][k])
  A[k][k]<-DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k]<-DGEMM(A[k][n],A[m][n],A[m][k])
    A[m][k]<-DTRSM(A[k][k],A[m][k])
```

Context



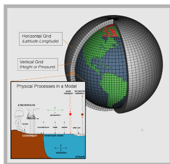
Context

Characteristics:
- *many processing units*
- *specialized units (e.g., GPUs)*



Context

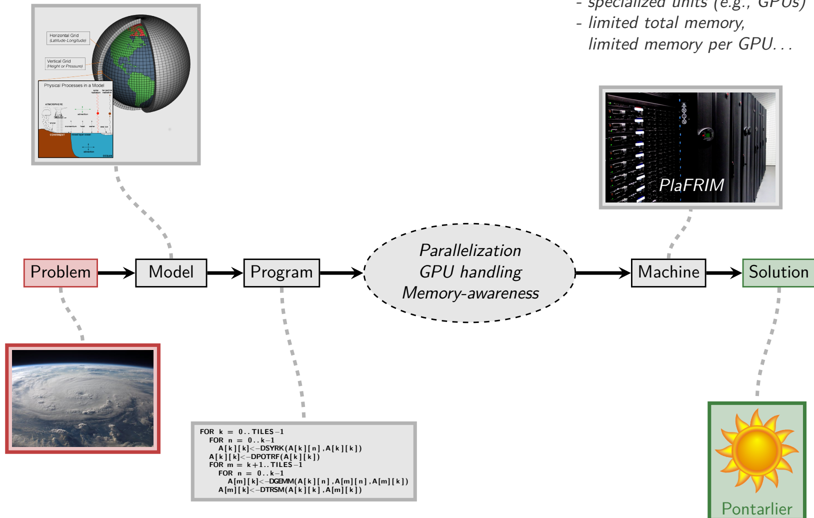
Characteristics:
- *many processing units*
- *specialized units (e.g., GPUs)*
- *limited total memory, limited memory per GPU...*



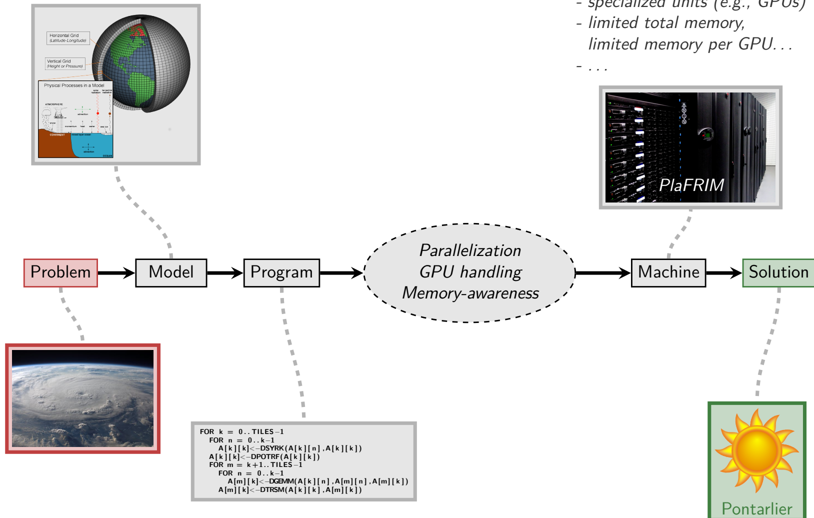
Problem → Model → Program → *Parallelization GPU handling* → Machine → Solution

```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k]<-DSYRK(A[k][n],A[k][k])
  A[k][k]<-DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k]<-DGEMM(A[k][n],A[m][n],A[m][k])
    A[m][k]<-DTRSM(A[k][k],A[m][k])
```

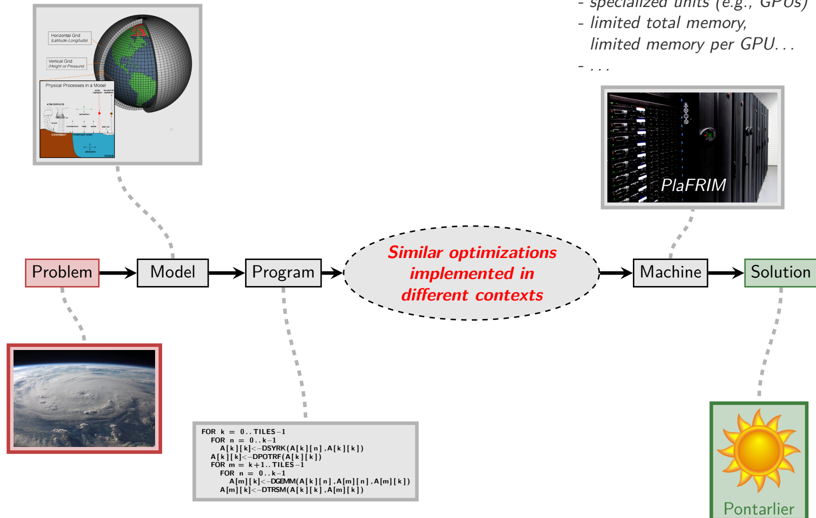
Context



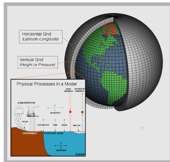
Context



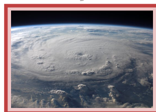
Context



Characteristics:
- *many processing units*
- *specialized units (e.g., GPUs)*
- *limited total memory, limited memory per GPU...*
- *...*



Problem → Model → Program → **Similar optimizations implemented in different contexts** → Machine → Solution

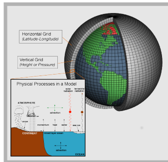


```
FOR k = 0..TILES
  FOR n = 0..k−1
    A[k][k]<−DSYR
  A[k][k]<−DPOTRF(A[k][k])
  FOR m = k+1..TILES−1
    FOR n = 0..k−1
      A[m][k]<−DGEMM(A[k][n],A[m][n],A[m][k])
    A[m][k]<−DTRSM(A[k][k],A[m][k])
```

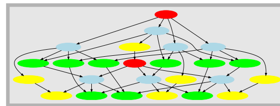
"Not every astronomer builds her own telescope!"

Bruce Hendrickson, IPDPS 2018 keynote

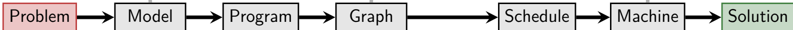
Context

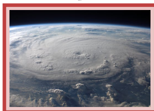


Separate "what" from "how"
Task example: matrix multiplication

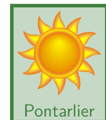


Characteristics:
- *many processing units*
- *specialized units (e.g., GPUs)*
- *limited total memory, limited memory per GPU...*
- *...*





```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k]<-DSYRK(A[k][n],A[k][k])
  A[k][k]<-DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k]<-DGEMM(A[k][n],A[m][n],A[m][k])
    A[m][k]<-DTRSM(A[k][k],A[m][k])
```

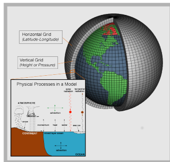


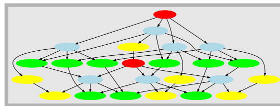
Context

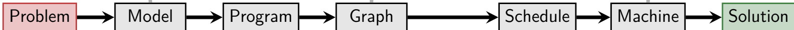
Separate "what" from "how"
Task example: matrix multiplication

Characteristics:

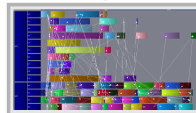
- *many processing units*
- *specialized units (e.g., GPUs)*
- *limited total memory, limited memory per GPU...*
- *...*

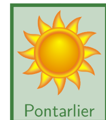




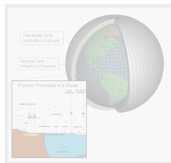


```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k]<-DSYRK(A[k][n],A[k][k])
  A[k][k]<-DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k]<-DGEMM(A[k][n],A[m][n],A[m][k])
    A[m][k]<-DTRSM(A[k][k],A[m][k])
```

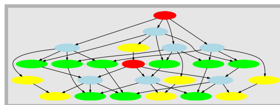




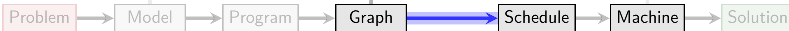
Context



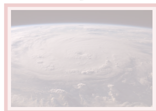
Separate "what" from "how"
Task example: matrix multiplication



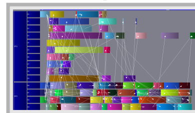
Characteristics:
- *many processing units*
- *specialized units (e.g., GPUs)*
- *limited total memory, limited memory per GPU...*
- *...*



Focus of this thesis



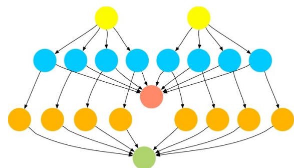
```
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k]<-DSYRK(A[k][n],A[k][k])
  A[k][k]<-DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k]<-DGEMM(A[k][n],A[m][n],A[m][k])
    A[m][k]<-DTRSM(A[k][k],A[m][k])
```



Task graph paradigm

Widely used in runtime systems

- Goal: relieve software engineers of low-level architecture-specific decisions
- Vertex = task,
  edge = data dependence
- Runtime scheduler decides the allocation



Schedulers face multiple challenges

Need for theoretical insights in order to implement efficient solutions

Assumptions in this presentation

- The platform is a shared-memory system
- Whole graph is known beforehand
- Estimated execution times are available
  - Ex: matrix multiplication 2000×2000 takes 30ms on a CPU

Outline of the thesis

Exploiting task parallelism **[Euro-Par 2015, TPDS 2018]**

Chapters 1 & 2 Allocate several processors per task

Efficiently using several types of processors **[Euro-Par 2018]**

Chapter 3 Improved existing online algorithm minimizing makespan & first online lower bounds

Coping with a limited memory **[IPDPS 2018, IPDPSW 2017]**

Chapter 4 Prevent schedulers from exceeding the available memory

Chapter 5 Minimize memory / disk transfers

Designing data structures minimizing memory / disk transfers **[PODS 2016, LATIN 2016]**

Chapter 6 *Work conducted during a research visit*

Outline of the thesis

Exploiting task parallelism [Euro-Par 2015, **TPDS 2018**]

Chapters 1 & 2 Allocate several processors per task

Efficiently using several types of processors [Euro-Par 2018]

Chapter 3 Improved existing online algorithm minimizing makespan & first online lower bounds

Coping with a limited memory [**IPDPS 2018**, IPDPSW 2017]

Chapter 4 Prevent schedulers from exceeding the available memory

Chapter 5 Minimize memory / disk transfers

Designing data structures minimizing memory / disk transfers [PODS 2016, LATIN 2016]

Chapter 6 *Work conducted during a research visit*

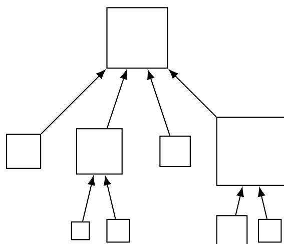
Outline

1. Scheduling graphs of parallel tasks
   - Evaluation of existing speedup models and our proposition
   - Analysis of scheduling algorithms to minimize the makespan
   - Experimental comparison
2. Coping with a limited available memory
   - Model and maximum memory peak
   - Efficient scheduling with bounded memory & simulation results
3. Conclusion

Context of the project

Target application

- Workflow occurring in linear algebra:
  QR factorization of a sparse matrix in the qr_mumps software
- Assembly tree: each node has exactly one successor



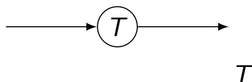
Computations inside each task

- QR decomposition of a dense matrix of a given size
- Each task can be in turn parallelized
- Need to decide how many processors are allocated to each task

Description of the problem

Graph

- Tree generalized to a Series-Parallel graph
- Purpose: find a schedule achieving the shortest makespan



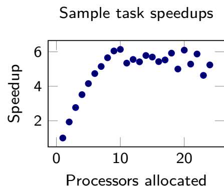
T

Parallel and malleable tasks

- Processors can be added to a task or removed during its execution
- Each task: sequential processing time w_i and speedup function
- Speedup function

$$time_i(10\ procs.) = \frac{w_i}{speedup_i(10\ procs.)}$$

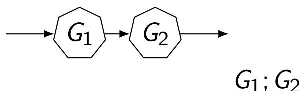
- Similar tasks $\Longrightarrow$ similar speedups



Description of the problem

Graph

- Tree generalized to a Series-Parallel graph
- Purpose: find a schedule achieving the shortest makespan

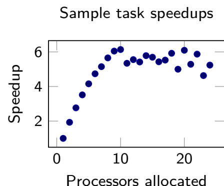


Parallel and malleable tasks

- Processors can be added to a task or removed during its execution
- Each task: sequential processing time w_i and speedup function
- Speedup function

$$time_i(10\ procs.) = \frac{w_i}{speedup_i(10\ procs.)}$$

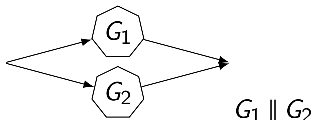
- Similar tasks $\Longrightarrow$ similar speedups



Description of the problem

Graph

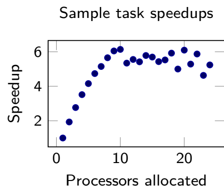
- Tree generalized to a Series-Parallel graph
- Purpose: find a schedule achieving the shortest makespan



Parallel and malleable tasks

- Processors can be added to a task or removed during its execution
- Each task: sequential processing time w_i and speedup function
- Speedup function

$$time_i(10\ procs.) = \frac{w_i}{speedup_i(10\ procs.)}$$

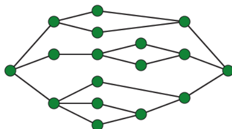


- Similar tasks $\Longrightarrow$ similar speedups

Description of the problem

Graph

- Tree generalized to a Series-Parallel graph
- Purpose: find a schedule achieving the shortest makespan

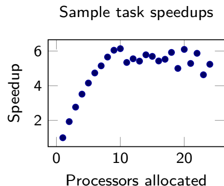


Parallel and malleable tasks

- Processors can be added to a task or removed during its execution
- Each task: sequential processing time w_i and speedup function
- Speedup function

$$time_i(10\ procs.) = \frac{w_i}{speedup_i(10\ procs.)}$$

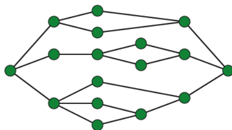
- Similar tasks $\Longrightarrow$ similar speedups

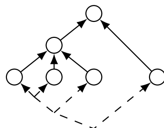


Description of the problem

Graph

- Tree generalized to a Series-Parallel graph
- Purpose: find a schedule achieving the shortest makespan



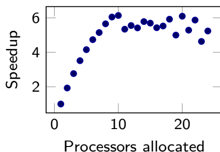


Parallel and malleable tasks

- Processors can be added to a task or removed during its execution
- Each task: sequential processing time w_i and speedup function
- Speedup function

$$time_i(10\ procs.) = \frac{w_i}{speedup_i(10\ procs.)}$$

- Similar tasks $\Longrightarrow$ similar speedups



Need for a speedup model

Moldable tasks (constant allocation), any speedup

- High-complexity FPTAS [Günther et al. 2014]
- Low-complexity heuristic [Hunold 2014]

Malleable tasks, concave & non-decreasing speedup

- $(2+\varepsilon)$-approximation of huge complexity [Makarychev et al. 2014]

Objectives:

- Design an accurate speedup model for assembly tree tasks
- Prove and propose low-complexity guaranteed algorithms

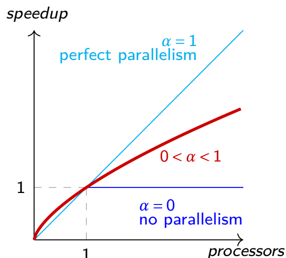
Outline

1. Scheduling graphs of parallel tasks
   - Evaluation of existing speedup models and our proposition
   - Analysis of scheduling algorithms to minimize the makespan
   - Experimental comparison
2. Coping with a limited available memory
   - Model and maximum memory peak
   - Efficient scheduling with bounded memory & simulation results
3. Conclusion

The speedup model of Prasanna and Musicus [1996]

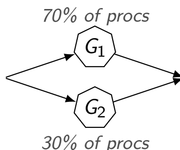
Description of the model

- Advocated for matrix operations
- $Speedup(p) = p^{\alpha}$, with $0 < \alpha < 1$
- ☹ same α for all tasks,
  non-integral allocation,
  infinite speedup



Theorem (Prasanna & Musicus, proof simplified in this thesis)

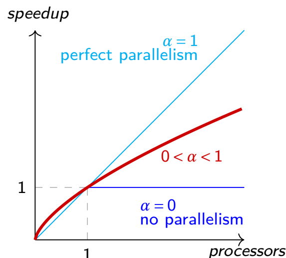
In the unique optimal schedule, at any parallel node $G_1 \parallel G_2$, the share of processors given to G_1 is constant and easily computed.



The speedup model of Prasanna and Musicus [1996]

Description of the model

- Advocated for matrix operations
- $Speedup(p) = p^{\alpha}$, with $0 < \alpha < 1$
- ☹ same α for all tasks, non-integral allocation, infinite speedup



Results on two nodes of p and q cores

- Scheduling independent tasks is NP-hard even if $p = q$
- Design of a $\left(\frac{4}{3}\right)^{\alpha}$-approximation for $p = q$
- Design of an FPTAS for independent task scheduling and $p \neq q$

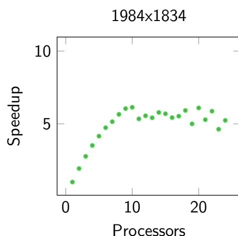
Experimental evaluation of the Prasanna & Musicus model

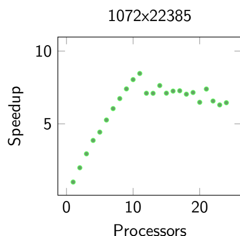
Instances

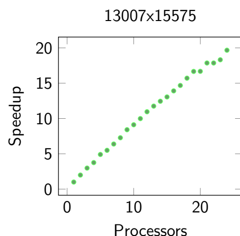
- Graphs: assembly trees of sparse matrices (SuiteSparse collection)
  tasks: QR decompositions of a dense matrix

Results

- Benchmark > 10000 tasks with 1 to 24 cores (PlaFRIM platform)
  - Each task: plot speedup, correct decrease
  - Fit the p^α model with $\alpha = 0.9$







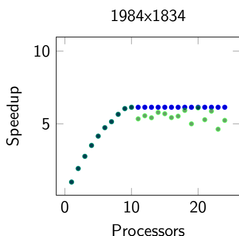
Experimental evaluation of the Prasanna & Musicus model

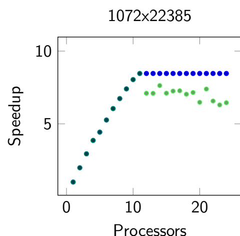
Instances

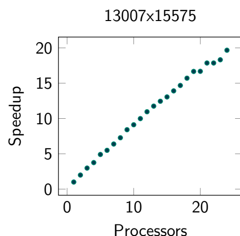
- Graphs: assembly trees of sparse matrices (SuiteSparse collection)
  tasks: QR decompositions of a dense matrix

Results

- Benchmark > 10000 tasks with 1 to 24 cores (PlaFRIM platform)
  - Each task: plot speedup, correct decrease
  - Fit the p^α model with $\alpha = 0.9$







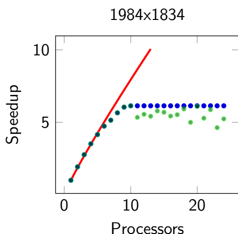
Experimental evaluation of the Prasanna & Musicus model

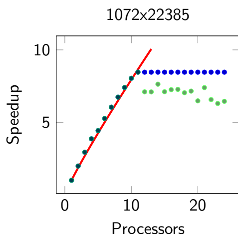
Instances

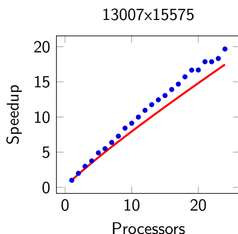
- Graphs: assembly trees of sparse matrices (SuiteSparse collection)
  tasks: QR decompositions of a dense matrix

Results

- Benchmark >10000 tasks with 1 to 24 cores (PlaFRIM platform)
  - Each task: plot speedup, correct decrease
  - Fit the p^α model with $\alpha = 0.9$







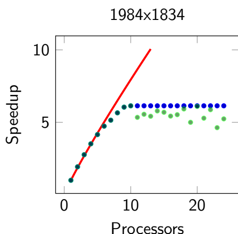
Experimental evaluation of the Prasanna & Musicus model

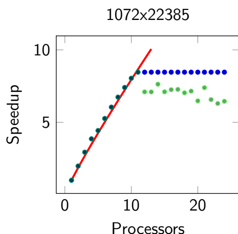
Instances

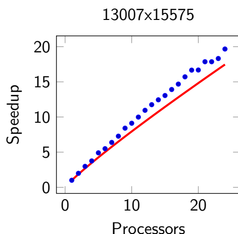
- Graphs: assembly trees of sparse matrices (SuiteSparse collection)
  tasks: QR decompositions of a dense matrix

Results

- Benchmark > 10000 tasks with 1 to 24 cores (PlaFRIM platform)
  - Each task: plot speedup, correct decrease
  - Fit the p^α model with $\alpha = 0.9$
- ☹ Insufficient accuracy: same speedup for all tasks, unknown limit



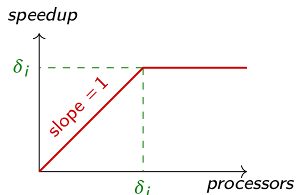


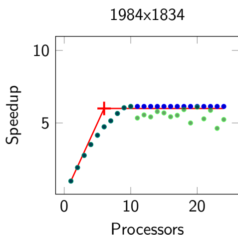


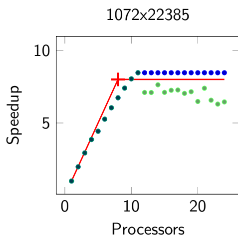
The well-known roofline model

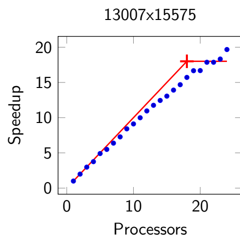
Description of this model

- First processors are fully used; a plateau is ultimately reached
- δ_i: tunable parameter
- ☹ insufficient accuracy ($R^2 \approx 0.9$) especially near δ_i
- Optimal schedule NP-hard (new proof in this thesis)





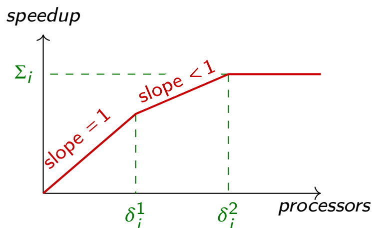


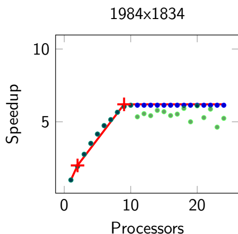


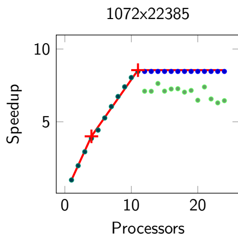
Our speedup model proposition

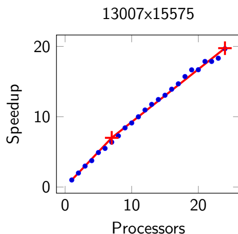
Simple and accurate model

- Perfect then linear then plateau
- Three parameters per task
- ☺ Good accuracy ($R^2 \approx 0.98$)
- ☹ Optimal schedule NP-hard









Related work on explicit speedup functions

Moldable tasks

- Single-threshold: $(3-\frac{2}{p})$ - approximation [Wang & Cheng 1992]
- $time(p) = \frac{w_i}{p} + (p-1)c$ [Kell & Havill 2015]
- $time(p) = w_i^{(s)} + \frac{w_i^{(p)}}{p}$: Amdahl's law

Malleable tasks

- p^{α}: optimal solution in linear time [Prasanna & Musicus 1996]
- Single-threshold: 2-approximation FlowFlex [Balmin et al. 2013]

Transform a non-integer allocation into an integer allocation

- Valid for malleable tasks under piecewise linear speedups [McNaughton 1959]

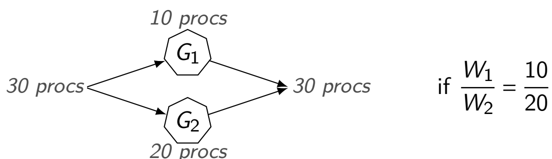
Outline

1. Scheduling graphs of parallel tasks
   - Evaluation of existing speedup models and our proposition
   - Analysis of scheduling algorithms to minimize the makespan
   - Experimental comparison
2. Coping with a limited available memory
   - Model and maximum memory peak
   - Efficient scheduling with bounded memory & simulation results
3. Conclusion

PROPORTIONALMAPPING

Simple allocation for trees or SP-graphs [Pothen et al. 1993]

- On $G_1 \parallel G_2$: constant share to G_i, proportional to its weight W_i
- Then schedule each task ASAP



$$\text{if } \frac{W_1}{W_2} = \frac{10}{20}$$

Imperfect speedup: tasks do not finish simultaneously so processors idle

Proposed extensions for our model

- PROPMAPEXT: when a task terminates, reallocate its processors
- PROPMAPEXTTHRESH: idem but never exceeds δ^2

FlowFlex

Principle (designed for a single threshold) [Balmin et al. 13]

- Schedule the graph on an infinite number of processors
- Downscale the allocation on each constant-allocation interval

Adaptation to our model

- Similar to PropMapExtThresh: rebalance idling processors

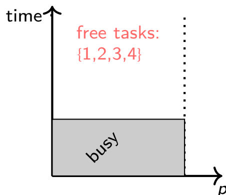
Design of a greedy strategy: GREEDY-FILLING

Algorithm

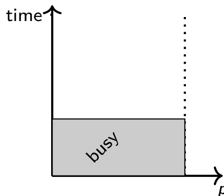
- Consider free tasks by decreasing bottom-level:
  - allocate δ_i^1 processors to each task
  - if processors remain, increase the allocation to δ_i^2 processors
- When the first task terminates, reset the allocations and repeat

Illustration

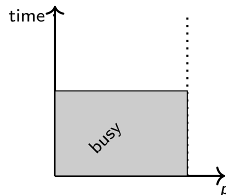
initial profile:



tasks allocation:



next profile:



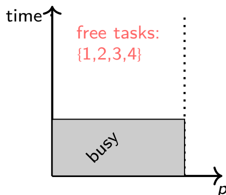
Design of a greedy strategy: GREEDY-FILLING

Algorithm

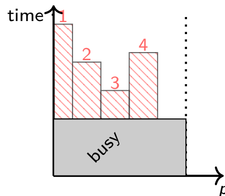
- Consider free tasks by decreasing bottom-level:
  - allocate δ_i^1 processors to each task
  - if processors remain, increase the allocation to δ_i^2 processors
- When the first task terminates, reset the allocations and repeat

Illustration

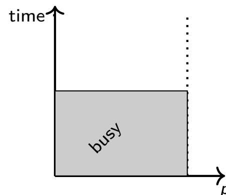
initial profile:



tasks allocation:



next profile:



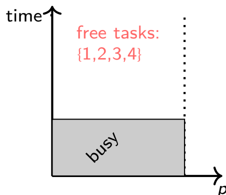
Design of a greedy strategy: GREEDY-FILLING

Algorithm

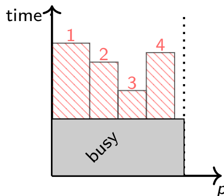
- Consider free tasks by decreasing bottom-level:
  - allocate δ_i^1 processors to each task
  - if processors remain, increase the allocation to δ_i^2 processors
- When the first task terminates, reset the allocations and repeat

Illustration

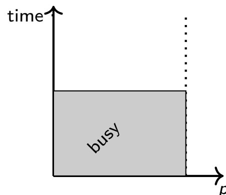
initial profile:



tasks allocation:



next profile:



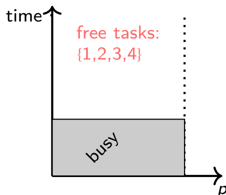
Design of a greedy strategy: GREEDY-FILLING

Algorithm

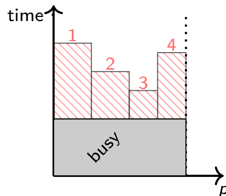
- Consider free tasks by decreasing bottom-level:
  - allocate δ_i^1 processors to each task
  - if processors remain, increase the allocation to δ_i^2 processors
- When the first task terminates, reset the allocations and repeat

Illustration

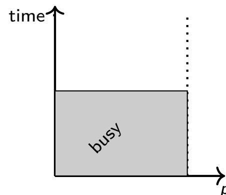
initial profile:



tasks allocation:



next profile:



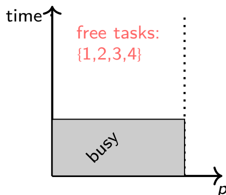
Design of a greedy strategy: Greedy-Filling

Algorithm

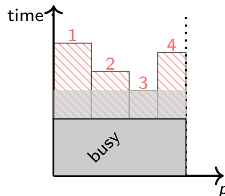
- Consider free tasks by decreasing bottom-level:
  - allocate δ_i^1 processors to each task
  - if processors remain, increase the allocation to δ_i^2 processors
- When the first task terminates, reset the allocations and repeat

Illustration

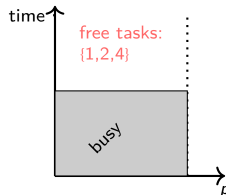
initial profile:



tasks allocation:



next profile:

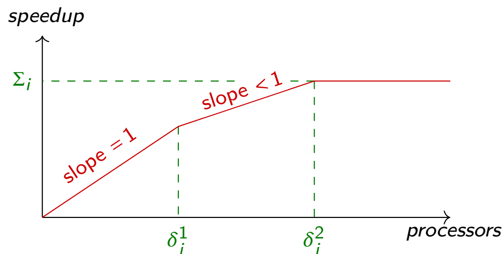


Theoretical guarantees

Theorem

ProportionalMapping, Greedy-Filling and FlowFlex are $(1+r)$-approximation of the optimal makespan, with $r = \max_i(\delta_i^2/\Sigma_i) \geq 1$.

Corollary: they are 2-approximation for the single-threshold model.



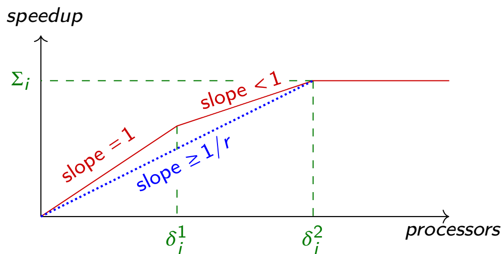
Note: same factor, but two different arguments

Theoretical guarantees

Theorem

ProportionalMapping, Greedy-Filling *and* FlowFlex *are* $(1+r)$*-approximation of the optimal makespan, with* $r = \max_i (\delta_i^2 / \Sigma_i) \geq 1$.

Corollary: they are 2-approximation for the single-threshold model.

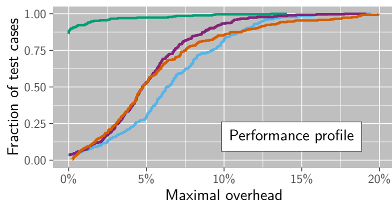


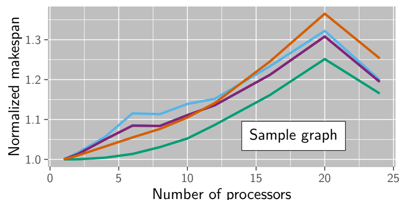
Note: same factor, but two different arguments

Outline

1. Scheduling graphs of parallel tasks
   - Evaluation of existing speedup models and our proposition
   - Analysis of scheduling algorithms to minimize the makespan
   - Experimental comparison
2. Coping with a limited available memory
   - Model and maximum memory peak
   - Efficient scheduling with bounded memory & simulation results
3. Conclusion

Synthetic graphs (200 nodes)



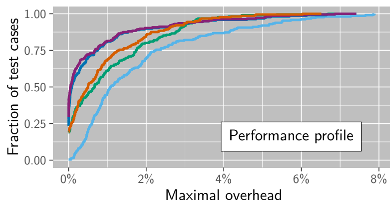


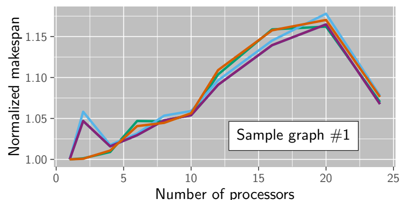
Algorithm — GREEDY-FILLING — PROPMAPNAIVE — PROPMAPEXT — PROPMAPEXTTHRESH — FLOWFLEX

Speedup: $\delta_i^1 \propto$ *time(1 proc.) and* δ_i^2 *uniform in* $[\delta_i^1, 2\delta_i^1]$

- Right: makespan normalized by a lower bound *(best is 1.0, bottom)*
  - Sample representative random graph
- Left: performance profile *(best is top-left)*
  - GREEDY-FILLING is almost always the best
  - Gains $> 5\%$ in 50% of the cases against any other heuristic

Assembly trees [SuiteSparse collection] (30 to 6000 nodes)



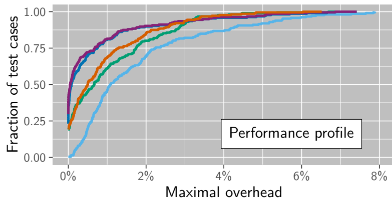


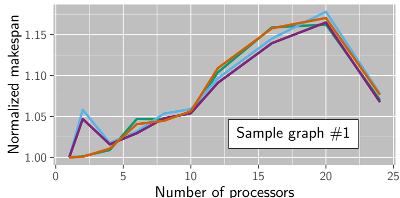
Algorithm — GREEDY-FILLING — PROPMAPNAIVE — PROPMAPEXT — PROPMAPEXTTHRESH — FLOWFLEX

Speedup = actual timings

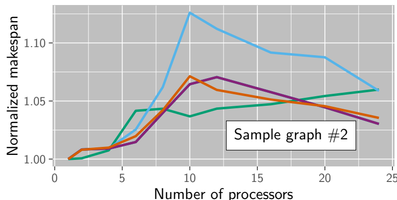
- Left: performance profile *(best is top-left)*
  - PROPORTIONALMAPPING performs the worst, its extensions are the best
- Right: makespan normalized by a lower bound *(best is 1.0, bottom)*
  - Sample tree
  - Results heavily depend on the tree & number of processors

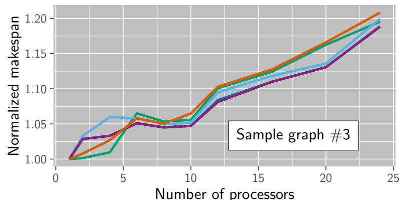
Assembly trees [SuiteSparse collection] (30 to 6000 nodes)





Algorithm — Greedy-Filling — PropMapNaive — PropMapExt — PropMapExtThresh — FlowFlex





Algorithm — Greedy-Filling — PropMapNaive — PropMapExt — PropMapExtThresh — FlowFlex

Summary of this part

On the two-threshold model

- Far more accurate than existing ones for QR decompositions
- NP-complete, as the single-threshold one
- Theoretically guaranteed low-complexity heuristics

On the heuristics

- Greedy-Filling (also on DAGs)
  - best on well-balanced instances (low idle times)
- ProportionalMapping extensions
  - best when several paths should be prioritized
  - globally the best on our assembly trees dataset

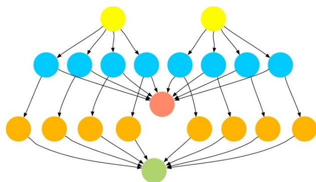
Outline

1. Scheduling graphs of parallel tasks
   - Evaluation of existing speedup models and our proposition
   - Analysis of scheduling algorithms to minimize the makespan
   - Experimental comparison
2. Coping with a limited available memory
   - Model and maximum memory peak
   - Efficient scheduling with bounded memory & simulation results
3. Conclusion

Coping with a limited available memory

Focus on massively parallel graphs

- Many tasks executed concurrently



Limited available memory

- Some traversals may go out-of-memory
- Assume we know one traversal that fits

Objective

- Prevent **dynamic** schedulers from exceeding memory ($\neq$ provide one static schedule)

Maximum memory peak of a graph: maximum memory that any schedule may use

Outline

1 Scheduling graphs of parallel tasks
- Evaluation of existing speedup models and our proposition
- Analysis of scheduling algorithms to minimize the makespan
- Experimental comparison

2 Coping with a limited available memory
- Model and maximum memory peak
- Efficient scheduling with bounded memory & simulation results

3 Conclusion

An elementary memory model

Task graph weights

- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

An elementary memory model

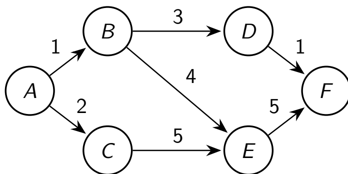
Task graph weights

- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

Memory behavior

- Task starts: free inputs (instantaneously)
  allocate outputs
- Task ends: outputs stay in memory

$M_{used} = 0$



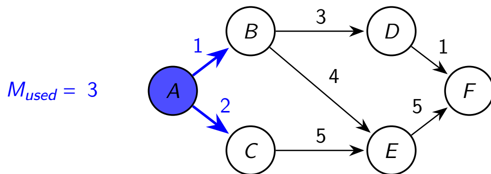
An elementary memory model

Task graph weights

- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

Memory behavior

- Task starts: free inputs (instantaneously)
  allocate outputs
- Task ends: outputs stay in memory



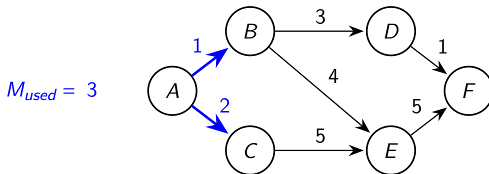
An elementary memory model

Task graph weights

- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

Memory behavior

- Task starts: free inputs (instantaneously)
  allocate outputs
- Task ends: outputs stay in memory



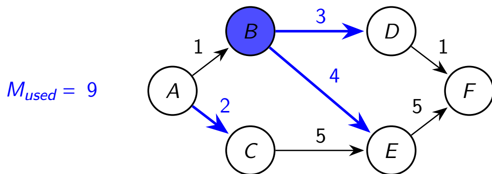
An elementary memory model

Task graph weights

- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

Memory behavior

- Task starts: free inputs (instantaneously)
  allocate outputs
- Task ends: outputs stay in memory



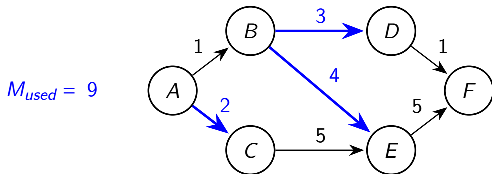
An elementary memory model

Task graph weights

- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

Memory behavior

- Task starts: free inputs (instantaneously)
  allocate outputs
- Task ends: outputs stay in memory



An elementary memory model

Task graph weights

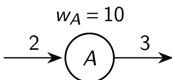
- Vertex w_i : estimated task duration
- Edge $m_{i,j}$: data size

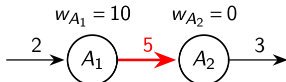
Memory behavior

- Task starts: free inputs (instantaneously)
  allocate outputs
- Task ends: outputs stay in memory

Emulation of other memory behaviors

- Inputs not freed, additional execution memory: duplicate nodes

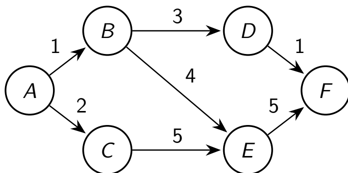




Maximum memory peak equivalent

Topological cut = partition of the vertices (S, T) with

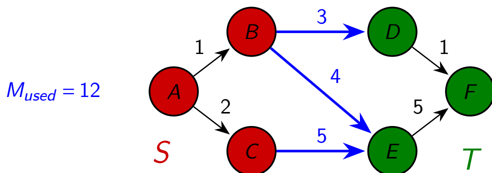
- Source $s \in S$ and sink $t \in T$
- No edge from T to S
- Weight of the cut = sum of all edge weights from S to T



Maximum memory peak equivalent

Topological cut = partition of the vertices (S, T) with

- Source $s \in S$ and sink $t \in T$
- No edge from T to S
- Weight of the cut = sum of all edge weights from S to T

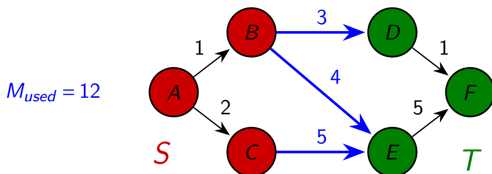


Topological cut $\longleftrightarrow$ execution state where T nodes are not started yet

Maximum memory peak equivalent

Topological cut = partition of the vertices (S, T) with

- Source $s \in S$ and sink $t \in T$
- No edge from T to S
- Weight of the cut = sum of all edge weights from S to T



Topological cut $\longleftrightarrow$ execution state where T nodes are not started yet

Equivalence in our model between:

- Maximum memory peak (any parallel execution)
- Maximum weight of a topological cut

Computing the maximum topological cut

Literature: Min-Cut polynomial, Max-Cut NP-hard even on DAGs

Theorem

Computing the maximum topological cut on a DAG is polynomial.

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow *maxdiff* in G^{diff}
4. $F - maxdiff$ is a minimum flow in G
5. Residual graph → maximum topological cut

Computing the maximum topological cut

Literature: Min-Cut polynomial, Max-Cut NP-hard even on DAGs

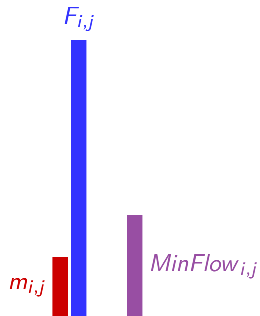
Theorem

Computing the maximum topological cut on a DAG is polynomial.

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow *maxdiff* in G^{diff}
4. $F - maxdiff$ is a minimum flow in G
5. Residual graph → maximum topological cut



Computing the maximum topological cut

Literature: Min-Cut polynomial, Max-Cut NP-hard even on DAGs

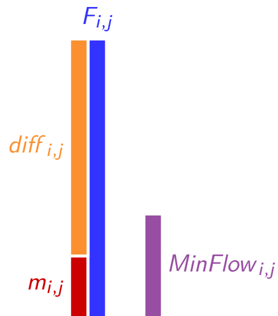
Theorem

Computing the maximum topological cut on a DAG is polynomial.

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $maxdiff$ in G^{diff}
4. $F - maxdiff$ is a minimum flow in G
5. Residual graph → maximum topological cut



Computing the maximum topological cut

Literature: Min-Cut polynomial, Max-Cut NP-hard even on DAGs

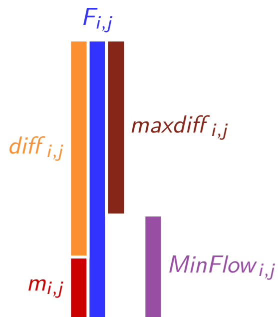
Theorem

Computing the maximum topological cut on a DAG is polynomial.

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $maxdiff$ in G^{diff}
4. $F - maxdiff$ is a minimum flow in G
5. Residual graph → maximum topological cut



Computing the maximum topological cut

Literature: Min-Cut polynomial, Max-Cut NP-hard even on DAGs

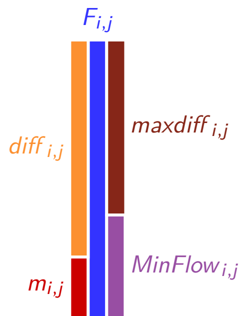
Theorem

Computing the maximum topological cut on a DAG is polynomial.

- Dual problem: Min-Flow *(larger than all edge weights)*
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

1. Build a large flow F on the graph G
2. Consider G^{diff} with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow *maxdiff* in G^{diff}
4. $F - maxdiff$ is a minimum flow in G
5. Residual graph → maximum topological cut



Outline

1 Scheduling graphs of parallel tasks
- Evaluation of existing speedup models and our proposition
- Analysis of scheduling algorithms to minimize the makespan
- Experimental comparison

2 Coping with a limited available memory
- Model and maximum memory peak
- Efficient scheduling with bounded memory & simulation results

3 Conclusion

Coping with limited memory

Problem

- Allow use of dynamic schedulers
- Limited available memory M
- Keep high level of parallelism

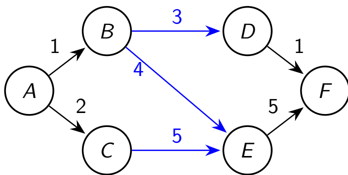
Coping with limited memory

Problem

- Allow use of dynamic schedulers
- Limited available memory M
- Keep high level of parallelism

Our solution

- Add edges to guarantee that any parallel execution stays below M
- Minimize the obtained *critical path*



$M_{\text{available}} = 10$

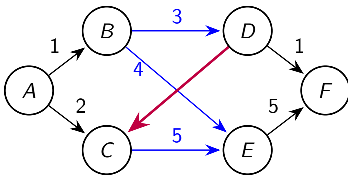
Coping with limited memory

Problem

- Allow use of dynamic schedulers
- Limited available memory M
- Keep high level of parallelism

Our solution

- Add edges to guarantee that any parallel execution stays below M
- Minimize the obtained *critical path*



$M_{\text{available}} = 10$

Problem definition and complexity

Definition (PARTIALSERIALIZATION of a DAG G under a memory M)

Compute a set of new edges E' such that:

- $G' = (V, E \cup E')$ is a DAG
- $MaxTopologicalCut(G') \leq M$
- $CritPath(G')$ is minimized

Theorem (Sethi 1975)

Computing a schedule that minimizes the memory usage is NP-hard.

Theorem

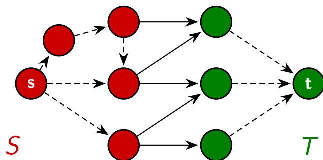
PARTIALSERIALIZATION *is NP-hard given a memory-efficient schedule.*

Optimal solution computable by an ILP (builds transitive closure)

Heuristic solutions for PARTIALSERIALIZATION

Framework – *inspired by [Sbîrlea et al. 2014]*

1. Compute a max. top. cut (S, T)
2. If weight $\leq M$: succeeds
3. Add edge (u, v) with $u \in T, v \in S$ without creating cycles; or fail
4. Goto Step 1



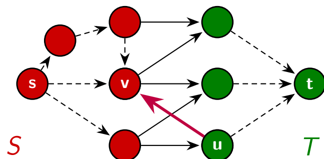
Several heuristic choices for Step 3

MinLevels	does not create a large critical path
RespectOrder	follows a precomputed memory-efficient schedule, always succeeds
MaxSize	targets nodes dealing with large data
MaxMinSize	variant of MaxSize

Heuristic solutions for PartialSerialization

Framework – *inspired by [Sbîrlea et al. 2014]*

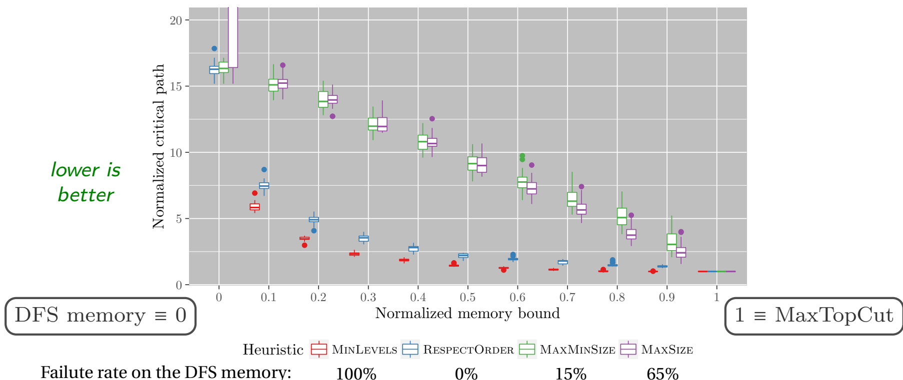
1. `Compute a max. top. cut` (S,T)
2. `If weight` $\leq M$ `: succeeds`
3. `Add edge` (u,v) `with` $u \in T, v \in S$ `without creating cycles; or fail`
4. `Goto Step 1`



Several heuristic choices for Step 3

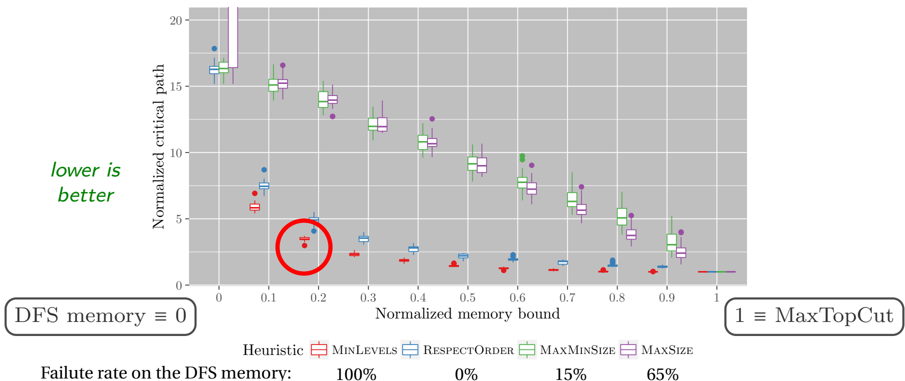
MinLevels	does not create a large critical path
RespectOrder	follows a precomputed memory-efficient schedule, always succeeds
MaxSize	targets nodes dealing with large data
MaxMinSize	variant of MaxSize

Simulations – Pegasus workflows (LIGO 100 nodes)



- *Median ratio MaxTopCut / DFS ≈ 20*
- MinLevels performs best, RespectOrder always succeeds
- Memory divided by 5 for CP multiplied by 3

Simulations – Pegasus workflows (LIGO 100 nodes)



- *Median ratio MaxTopCut / DFS ≈ 20*
- MinLevels performs best, RespectOrder always succeeds
- Memory divided by 5 for CP multiplied by 3

Summary of this part

Memory model proposed

- Elementary but equivalent to more complex models
- Explicit algorithm to compute the maximum memory peak

Prevent dynamic schedulers from exceeding memory

- Add edges, aiming at low critical path length
- NP-hard to get the lowest CP length
- Several heuristics with good performance on actual graphs (+ ILP)

Outline

1. Scheduling graphs of parallel tasks
   - Evaluation of existing speedup models and our proposition
   - Analysis of scheduling algorithms to minimize the makespan
   - Experimental comparison
2. Coping with a limited available memory
   - Model and maximum memory peak
   - Efficient scheduling with bounded memory & simulation results
3. Conclusion

Conclusion

Common approach for each problem

- Design of an ideal but realistic model
- Complexity study and algorithms
- Evaluation via simulations on mostly actual datasets
- Goal: identify the challenges & influence future implementations

Part 1: Scheduling malleable task graphs

- Accurate speedup model for linear algebra workflows
- Design & evaluation of guaranteed algorithms

Part 2: Coping with a limited available memory

- Elementary but expressive memory behavior
- Design & validation of heuristics relying on graph theory tools

Short-term perspectives on the parts covered

Part 1: Handle data movements

- Difficult to study with a general model
- Observation: Proportional Mapping has good locality properties & quite good makespan
- *Improve its makespan by heuristic modifications, preserving locality properties*

Part 2: Reduce heuristics complexity

- Current algorithm: too many iterations for each heuristic
- *Add many edges per iteration, use synchronization vertices, choose endpoints further from the cut...*
- Second direction: adapt the solution to the platform, i.e., change the goal (critical path length)

Long-term perspective: going distributed

Shared-memory platforms: at most tens of processors

Makespan minimization

- Problem: allocation of tasks to nodes avoiding communications
  Direction: graph clustering algorithms on hierarchical tasks (new paradigm under development in StarPU) + dynamic corrections
- Scheduler must be distributed

Memory handling

- Memory distributed among nodes
- Need to model memory operations
- *Shared-memory solutions: "Don't start too many tasks!"*
  Distributed memory: need for a new approach, depends on the allocation to the nodes

List of publications in this thesis

- A. Guermouche, L. Marchal, B. Simon and F. Vivien. Scheduling Trees of Malleable Tasks for Sparse Linear Algebra. *Euro-Par Conference*, 2015.
- M. Bender, J. Berry, R. Johnson, T. Kroeger, S. McCauley, C. Phillips, B. Simon, S. Singh and D. Zage. Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries. *PODS Conference*, 2016.
- M. Bender, R. Chowdhury, A. Conway, M. Farach-Colton, P. Ganapathi, R. Johnson, S. McCauley, B. Simon and S. Singh. The I/O Complexity of Computing Prime Tables. *LATIN Conference*, 2016.
- L. Marchal, S. McCauley, B. Simon and F. Vivien. Minimizing I/Os in Out-of-Core Task Tree Scheduling. *APDCM Workshop*, 2017.
- L. Marchal, B. Simon, O. Sinnen and F. Vivien. Malleable Task-Graph Scheduling with a Practical Speed-up Model. *TPDS Journal*, 2018.
- L. Marchal, H. Nagy, B. Simon and F. Vivien. Parallel Scheduling of DAGs Under Memory Constraints. *IPDPS Conference*, 2018.
- L.-C. Canon, L. Marchal, B. Simon and F. Vivien. Online Scheduling of Sequential Task Graphs on Hybrid Platforms. *Euro-Par Conference*, 2018.