Scheduling Task Graphs on Modern Computing Platforms

Bertrand SIMON

PhD Defense — 4 July 2018

École Normale Supérieure de Lyon, LIP laboratory

Commitee:

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claire</td>
<td>Hanen</td>
<td>Univ. Paris Nanterre</td>
</tr>
<tr>
<td>Safia</td>
<td>Kedad-Sidhoum</td>
<td>CNAM, Paris</td>
</tr>
<tr>
<td>Sascha</td>
<td>Hunold</td>
<td>TU Wien</td>
</tr>
<tr>
<td>Uwe</td>
<td>Schwiegelshohn</td>
<td>TU Dortmund</td>
</tr>
<tr>
<td>Loris</td>
<td>Marchal</td>
<td>CNRS &amp; ENS de Lyon</td>
</tr>
<tr>
<td>Frédéric</td>
<td>Vivien</td>
<td>INRIA &amp; ENS de Lyon</td>
</tr>
</tbody>
</table>
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])
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] <- DGERM(A[k][n], A[m][n], A[m][k])
A[m][k] <- DTRSM(A[k][k], A[m][k])
FOR \( k = 0 \ldots \text{TILES} - 1 \)
FOR \( n = 0 \ldots k - 1 \)
\( A[k][k] \leftarrow \text{DSYRK}(A[k][n], A[k][k]) \)
\( A[k][k] \leftarrow \text{DPOTRF}(A[k][k]) \)
FOR \( m = k + 1 \ldots \text{TILES} - 1 \)
FOR \( n = 0 \ldots k - 1 \)
\( A[m][k] \leftarrow \text{DGERM}(A[k][n], A[m][n], A[m][k]) \)
\( A[m][k] \leftarrow \text{DTRSM}(A[k][k], A[m][k]) \)
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])
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] ← DGERM(A[k][n], A[m][n], A[m][k])
      A[m][k] ← DTRSM(A[k][k], A[m][k])
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])

Characteristics:
- many processing units
- specialized units (e.g., GPUs)
- limited total memory, limited memory per GPU...
FOR $k = 0 \ldots TILES - 1$
FOR $n = 0 \ldots k - 1$
\begin{align*}
A[k][k] & \leftarrow DSYRK(A[k][n], A[k][k]) \\
A[k][k] & \leftarrow DPOTRF(A[k][k])
\end{align*}
FOR $m = k + 1 \ldots TILES - 1$
FOR $n = 0 \ldots k - 1$
\begin{align*}
A[m][k] & \leftarrow DGEMM(A[k][n], A[m][n], A[m][k]) \\
A[m][k] & \leftarrow DTRSM(A[k][k], A[m][k])
\end{align*}

Characteristics:
- many processing units
- specialized units (e.g., GPUs)
- limited total memory,
  limited memory per GPU…
FOR \( k = 0 \) to \( \text{TILES} - 1 \)
FOR \( n = 0 \) to \( k - 1 \)
\( A[k][k] \leftarrow \text{DSYRK}(A[k][n], A[k][k]) \)
\( A[k][k] \leftarrow \text{DPOTRF}(A[k][k]) \)
FOR \( m = k + 1 \) to \( \text{TILES} - 1 \)
FOR \( n = 0 \) to \( k - 1 \)
\( A[m][k] \leftarrow \text{DGE MM}(A[k][n], A[m][n], A[m][k]) \)
\( A[m][k] \leftarrow \text{DTRSM}(A[k][k], A[m][k]) \)
Characteristics:
- many processing units
- specialized units (e.g., GPUs)
- limited total memory,
  limited memory per GPU…
- …

PlaFRIM

Similar optimizations implemented in different contexts

FOR k = 0 .. TILES -1
FOR n = 0 .. k -1
A[k][k] = DSYRK(A[k][n],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])
Similar optimizations implemented in different contexts

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

“Not every astronomer builds her own telescope!”

Bruce Hendrickson, IPDPS 2018 keynote
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])
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])

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

Separate “what” from “how”
Task example: matrix multiplication
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 $2000 \times 2000$ takes 30ms on a CPU
<table>
<thead>
<tr>
<th>Title</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploiting task parallelism</td>
<td>[Euro-Par 2015, TPDS 2018]</td>
</tr>
<tr>
<td>Chapters 1 &amp; 2 Allocate several processors per task</td>
<td></td>
</tr>
<tr>
<td>Efficiently using several types of processors</td>
<td>[Euro-Par 2018]</td>
</tr>
<tr>
<td>Chapter 3 Improved existing online algorithm minimizing makespan &amp;</td>
<td></td>
</tr>
<tr>
<td>first online lower bounds</td>
<td></td>
</tr>
<tr>
<td>Coping with a limited memory</td>
<td>[IPDPS 2018, IPDPSW 2017]</td>
</tr>
<tr>
<td>Chapter 4 Prevent schedulers from exceeding the available memory</td>
<td></td>
</tr>
<tr>
<td>Chapter 5 Minimize memory / disk transfers</td>
<td></td>
</tr>
<tr>
<td>Designing data structures minimizing memory / disk transfers</td>
<td>[PODS 2016, LATIN 2016]</td>
</tr>
<tr>
<td>Chapter 6 Work conducted during a research visit</td>
<td></td>
</tr>
</tbody>
</table>
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

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

$$time_i(10 \text{ procs.}) = \frac{w_i}{\text{speedup}_i(10 \text{ procs.})}$$

- Similar tasks $\implies$ similar speedups
Description of the problem

**Graph**

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

![Graph Diagram]

**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

\[
\text{time}_i(10\ \text{procs.}) = \frac{w_i}{\text{speedup}_i(10\ \text{procs.})}
\]

- Similar tasks $\implies$ similar speedups

![Sample task speedups Graph]
Description of the problem

Graph

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

\[ G_1 \parallel G_2 \]

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

\[
\text{time}_i(10 \text{ procs.}) = \frac{w_i}{\text{speedup}_i(10 \text{ procs.})}
\]

- Similar tasks \( \implies \) similar speedups

Sample task speedups

![Sample task speedups graph](image)
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

\[
\text{time}_i(10 \text{ procs.}) = \frac{w_i}{\text{speedup}_i(10 \text{ procs.})}
\]

- Similar tasks \( \implies \) similar speedups

Sample task 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

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

- Similar tasks $\implies$ similar speedups

Sample task speedups

![Sample task speedups graph](image-url)
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 \( 0 < \alpha < 1 \)
- \( \alpha \) 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 $0 < \alpha < 1$
- $\alpha$ 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 $p^\alpha$ 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 $p^\alpha$ 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 $p^\alpha$ 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 $p^\alpha$ 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
- $\delta_i$: tunable parameter
- ☹ insufficient accuracy ($R^2 \approx 0.9$) especially near $\delta_i$
- 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

![Graph showing speedup vs. processors](image)

![Graphs comparing speedup for different tasks](image)
Related work on explicit speedup functions

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

**Malleable tasks**
- \( p^\alpha \): optimal solution in linear time [Prasanna & Musicus 1996]

**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
Scheduling graphs of parallel tasks

**Propportional Mapping**

### Simple allocation for trees or SP-graphs

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

![Diagram](image)

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 $\delta^2$
**Principle (designed for a single threshold)**

- 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 $\delta_1^i$ processors to each task
  - if processors remain, increase the allocation to $\delta_2^i$ processors
- When the first task terminates, reset the allocations and repeat

**Illustration**

<table>
<thead>
<tr>
<th>initial profile:</th>
<th>tasks allocation:</th>
<th>next profile:</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>free tasks: {1,2,3,4}</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>p</td>
</tr>
</tbody>
</table>

Bertrand SIMON  Scheduling Task Graphs on Modern Computing Platforms 18 / 40
**Algorithm**

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

**Illustration**

**Initial profile:**

- free tasks: $\{1,2,3,4\}$

**Tasks allocation:**

**Next profile:**

- busy
**Algorithm**

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

**Illustration**

Initial profile:

<table>
<thead>
<tr>
<th>time</th>
<th>busy</th>
<th>free tasks: {1,2,3,4}</th>
</tr>
</thead>
</table>

Tasks allocation:

<table>
<thead>
<tr>
<th>time</th>
<th>busy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Next profile:

<table>
<thead>
<tr>
<th>time</th>
<th>busy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design of a greedy strategy: **GREEDY-FILLING**

**Algorithm**

- Consider free tasks by decreasing bottom-level:
  - allocate $\delta_1^i$ processors to each task
  - if processors remain, increase the allocation to $\delta_2^i$ processors
- When the first task terminates, reset the allocations and repeat

**Illustration**

- **Initial profile:**
  - Free tasks: $\{1,2,3,4\}$

- **Tasks allocation:**
  - Time
  - Busy
  - $p$

- **Next profile:**
  - Time
  - Busy
  - $p$
Design of a greedy strategy: \textbf{GREEDY-FILLING}

\textbf{Algorithm}

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

\textbf{Illustration}

\begin{itemize}
  \item \textbf{initial profile:}
  \begin{itemize}
    \item free tasks: \{1,2,3,4\}
    \item busy
  \end{itemize}

  \begin{itemize}
    \item time
    \item \textbf{tasks allocation:}
    \begin{itemize}
      \item 1
      \item 2
      \item 3
      \item 4
    \end{itemize}
    \item busy
  \end{itemize}

  \begin{itemize}
    \item next profile:
    \begin{itemize}
      \item free tasks: \{1,2,4\}
      \item busy
    \end{itemize}
  \end{itemize}
\end{itemize}
Theoretical guarantees

**Theorem**

**PROPORTIONAL MAPPING, 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.

![Graph](image)

Note: same factor, but two different arguments
Theoretical guarantees

**Theorem**

**PROPORTIONAL MAPPING, GREEDY-FILLING and FLOWFLEX** are $(1 + r)$-approximation of the optimal makespan, with $r = \max_i \left( \frac{\delta_i^2}{\Sigma_i} \right) \geq 1$.

**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)

Performance profile

Maximal overhead

Fraction of test cases

Number of processors

Normalized makespan

Algorithm

Greedy-Filling

PropMapNaive

PropMapExt

PropMapExtThresh

FlowFlex

Gains > 5% in 50% of the cases against any other heuristic

Speedup: $\delta_i^1 \propto \text{time(1 proc.)}$ and $\delta_i^2$ 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
Assembly trees [SuiteSparse collection] (30 to 6000 nodes)

- Left: performance profile  
  - PROPORTIONAL MAPPING performs the worst, its extensions are the best

- Right: makespan normalized by a lower bound  
  - Sample tree  
  - Results heavily depend on the tree & number of processors

**Speedup = actual timings**

(best is top-left)
Assembly trees [SuiteSparse collection] (30 to 6000 nodes)

Performance profile

Sample graph #1

Sample graph #2

Sample graph #3

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)
- **Proportional Mapping** 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
  \((\neq \text{provide one static schedule})\)

**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

![Task Graph Diagram]

$M_{used} = 0$
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

![Task graph example](attachment:image)

$M_{used} = 3$
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

![Diagram of a task graph]

$M_{used} = 3$
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

$M_{used} = 9$
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

$$M_{used} = 9$$
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

![Graph示意图](attachment:graph.png)
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 = 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 \(T\)

\(M_{used} = 12\)

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 \(T\)

\[ M_{used} = 12 \]

**Topological cut \rightleftharpoons \text{execution state where } T \text{ 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 \((\text{larger than all edge weights})\)
- Idea: use an optimal algorithm for Max-Flow

Algorithm sketch

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

\(m_{i,j}\)
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 $G^{\text{diff}}$ with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $\text{maxdiff}$ in $G^{\text{diff}}$
4. $F - \text{maxdiff}$ is a minimum flow in $G$
5. Residual graph $\rightarrow$ 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 $G^{\text{diff}}$ with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $\text{maxdiff}$ in $G^{\text{diff}}$
4. $F - \text{maxdiff}$ is a minimum flow in $G$
5. Residual graph $\rightarrow$ 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 $G_{\text{diff}}$ with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $\text{maxdiff}$ in $G_{\text{diff}}$
4. $F - \text{maxdiff}$ is a minimum flow in $G$
5. Residual graph $\rightarrow$ 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 $G^{\text{diff}}$ with edge weights $F_{i,j} - m_{i,j}$
3. Compute a maximum flow $\text{maxdiff}$ in $G^{\text{diff}}$
4. $F - \text{maxdiff}$ is a minimum flow in $G$
5. Residual graph $\rightarrow$ 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

![Graph](image_url)
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*

![Graph diagram](Image)

$M_{available} = 10$
Definition (Partial Serialization 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
- $\text{MaxTopologicalCut}(G') \leq M$
- $\text{CritPath}(G')$ is minimized

Theorem (Sethi 1975)

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

Theorem

Partial Serialization 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)

- DFS memory $\equiv 0$
- $1 \equiv \text{MaxTopCut}$
- Median ratio $\text{MaxTopCut} / \text{DFS} \approx 20$
- $\text{MinLevels}$ performs best, $\text{RespectOrder}$ always succeeds
- Memory divided by 5 for CP multiplied by 3
Simulations – Pegasus workflows (LIGO 100 nodes)

DFS memory $\equiv 0$

lower is better

$1 \equiv \text{MaxTopCut}$

Heuristic: 
- MinLevels
- RespectOrder
- MaxMinSize
- MaxSize

Failure rate on the DFS memory: 
- 100%
- 0%
- 15%
- 65%

- $\text{Median ratio MaxTopCut} / \text{DFS} \approx 20$
- 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


