Dynamic Task Graph Adaptation with Recursive Tasks

Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Samuel Thibault, Pierre-André Wacrenier

February 2024
Task-based Programming

- Motivations:
  - Portable frameworks.
  - Exploit complex architectures.
- Applications: Directed Acyclic Graph (DAG).
- Runtime systems: scheduling, data management, communications, ...
Task-based Programming

- Motivations:
  - Portable frameworks.
  - Exploit complex architectures.
- Applications: Directed Acyclic Graph (DAG).
- Runtime systems: scheduling, data management, communications, ... 

STF: Sequential Task Flow

- Dependencies:
  - Automatically inferred.
  - Order of submission.

```
F(a)
G(a, b)
H(a, c)

F(a:RW)
G(a:R, b:RW)
H(a:R, c:RW)
wait_tasks_completion()
```
Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability: static task graphs.

⇒ How to create more dynamic task-graphs?
⇒ Recursive task graphs!

Granularity

- GPUs versus CPUs.
- Lack of parallelism versus Steady State.

⇒ Steering granularity dynamically?
Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability ? static task graphs.

⇒ How to create more dynamic task-graphs?
Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability? static task graphs.

⇒ How to create more dynamic task-graphs? ⇒ Recursive tasks graphs!
Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability? static task graphs.

⇒ How to create more dynamic task-graphs? ⇒ Recursive tasks graphs!

Granularity

- GPUs versus CPUs.
- Lack of parallelism versus Steady State.

⇒ Steering granularity dynamically?
Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability ? static task graphs.

⇒ How to create more dynamic task-graphs ? ⇒ Recursive tasks graphs !

Granularity

- GPUs versus CPUs.
- Lack of parallelism versus Steady State.

⇒ Steering granularity dynamically ?
Objectives
- Adapt task implementation at runtime.
- No spurious synchronization.

Principles
1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.
Objectives

- Adapt task implementation at runtime.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.
Objectives

- Adapt task implementation at runtime.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

Recursive task execution:
Objectives

- Adapt task implementation \textit{at runtime}.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

Recursive task execution:

\begin{itemize}
\item \textgreater\ Remain regular task.
\end{itemize}
Objectives

- Adapt task implementation at runtime.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

Recursive task execution:
- Remain regular task.
- Insert a subgraph: split.

Recursive Tasks in StarPU
Objectives

- Adapt task implementation at runtime.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

Recursive task execution:
- Remain regular task.
- Insert a subgraph: split.
Objectives
- Adapt task implementation \textit{at runtime}.
- No spurious synchronization.

Principles
1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.
   > Automatic data partition.

Recursive task execution:
- Remain regular task.
- Insert a subgraph: \textbf{split}.

Recursive Tasks in StarPU
## Recursive tasks - State of the Art

### Figure: Barrier between parent tasks

### Figure: Fine-grain dependencies

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Fine-grain Dependencies</th>
<th>Automatic data Partition</th>
<th>Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskFlow</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PaRSEC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRIS</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OmpSs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StarPU</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Recursive tasks - State of the Art

Figure: Barrier between parent tasks

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Fine-grain Dependencies</th>
<th>Automatic data Partition</th>
<th>Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskFlow</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PaRSEC</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRIS</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OmpSs</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>StarPU</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
## Recursive tasks - State of the Art

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Fine-grain Dependencies</th>
<th>Automatic data Partition</th>
<th>Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskFlow</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>PaRSEC</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>IRIS</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>OmpSs</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>StarPU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Figure:** Barrier between parent tasks

**Figure:** Fine-grain dependencies
### Recursive tasks - State of the Art

| Figure: Barrier between parent tasks |

| Figure: Fine-grain dependencies |

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Fine-grain Dependencies</th>
<th>Automatic data Partition</th>
<th>Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskFlow</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>PaRSEC</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>IRIS</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OmpSs</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StarPU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Recursive tasks - State of the Art

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Fine-grain Dependencies</th>
<th>Automatic data Partition</th>
<th>Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskFlow</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>PaRSEC</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>IRIS</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OmpSs</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>StarPU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Figure:** Barrier between parent tasks

**Figure:** Fine-grain dependencies
Dynamic task graph adaptation: splitting tasks

Which task should we split?

When do we choose to split task?
Dynamic task graph adaptation: splitting tasks

Which task should we split?

*Efficiency VS Completion Time*

When do we choose to split task?
Which task should we split?

*Efficiency VS Completion Time*

When do we choose to split task?

*Submission, execution, ...*
Exploit informations
### Exploit informations

1. Split efficiency.

<table>
<thead>
<tr>
<th>Time</th>
<th>Worker 0</th>
<th>Worker 1</th>
<th>Worker 2</th>
<th>Worker 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Completion time: 8 units.
Cumulated time: 8 units.
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.
**Exploit informations**

1. **Split efficiency.**

2. Current parallelism on Runtime System.
Exploit informations

1. **Split efficiency.**

2. **Current parallelism on Runtime System.**

Split the task - Gantt chart

- Worker 3
- Worker 2
- Worker 1
- Worker 0
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Split the task - Gantt chart

- worker 3
- worker 2
- worker 1
- worker 0

Completion time: 4 units.
Cumulated time: 11 units.

Not split the task - Gantt chart

- worker 3
- worker 2
- worker 1
- worker 0

Completion time: 8 units.
Cumulated time: 8 units.
Exploit informations

1. **Split efficiency.**

2. Current parallelism on Runtime System.

---

**Split the task - Gantt chart**

- Completion time: 4 units.

**Not split the task - Gantt chart**

- Completion time: 8 units.

---

Completion Time versus Efficiency

Which task do we split
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Split the task - Gantt chart
- Completion time: 4 units.
- Cumulated time: 11 units.

Not split the task - Gantt chart
- Completion time: 8 units.
- Cumulated time: 8 units.
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Split the task - Gantt chart
- Completion time: 4 units.
- Cumulated time: 11 units.

Not split the task - Gantt chart
- Completion time: 8 units.
- Cumulated time: 8 units.

Completion Time versus Efficiency
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.
Exploit informations

1. Split efficiency.

2. Current parallelism on Runtime System.
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Situation 1: Steady State

- worker 0
- worker 1
- worker 2
- worker 3

+ 24 ready tasks
Exploit informations

1. Split efficiency.

2. Current parallelism on Runtime System.

Situation 1: Steady State

- worker 3
- worker 2
- worker 1
- worker 0

Which task do we split?
Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Situation 2

worker 0
worker 1
worker 2
worker 3

+ 0 ready tasks

Which task do we split
Exploit informations

1. Split efficiency.

2. Current parallelism on Runtime System.

Situation 2: Starvation

- Worker 0
- Worker 1
- Worker 2
- Worker 3

Which task do we split?
Exploit informations

1. Split efficiency.

2. Current parallelism on Runtime System.

Situation 2: Starvation

- worker 0
- worker 1
- worker 2
- worker 3
1. Split efficiency.

2. Current parallelism on Runtime System.

Situation 2: Starvation

- worker 3
- worker 2
- worker 1
- worker 0

Which task do we split
When do we choose to split tasks

Task life path

app \hspace{1cm} submission \hspace{1cm} wait dependencies \hspace{1cm} release \hspace{1cm} scheduler \hspace{1cm} queue \hspace{1cm} data transfer \hspace{1cm} data fetching \hspace{1cm} execute \hspace{1cm} worker
When do we choose to split tasks

Adding the splitter

- app
- wait dependencies
- scheduler
- data transfer
- worker

- release
- queue
- execute

- rec. task
- reg. task
- splitter
When do we choose to split tasks

Position of the splitter - at submission

- app
- wait dependencies
- scheduler
- data transfer
- data fetching
- execute
- worker
- splitter
- reg. task
- release
- queue
- execute
- rec. task
- reg. task
- rec. task

Notes:
- Easy.
- Lack of information.
When do we choose to split tasks

Position of the splitter - at submission

- Easy.
When do we choose to split tasks

- Easy.
- Lack of information.
When do we choose to split tasks

Position of the splitter - Execution

app ➔ submission ➔ wait dependencies ➔ release ➔ scheduler ➔ queue ➔ data transfer data fetching ➔ execute ➔ worker

rec. task  ➔ rec. task  ➔ reg. task

splitter

Runtime information.

Useless data transfer: cancel decision.
Position of the splitter - Execution

- Runtime information.
• **Runtime information.**
• **Useless data transfer:** cancel decision.
When do we choose to split tasks

Position of the splitter - trade-off

- **app** submission
- **wait dependencies**
- **reg. task**
- **scheduler**
- **queue**
- **data transfer data fetching**
- **execute**
- **worker**

- **splitter**
- **rec. task**
- **subDAG submission**
When do we choose to split tasks

Recursive Task Path - Release dependency

1. $R_1$ → 2. $R_2$ → 3. $R_3$ → 4. $R_4$
Recursive Task Path - Release dependency

When do we choose to split tasks
When do we choose to split tasks

Recursive Task Path - Release dependency

When do we choose to split tasks
When do we choose to split tasks

Recursive Task Path - Release dependency

When do we choose to split tasks
When do we choose to split tasks

Recursive Task Path - Release dependency

When do we choose to split tasks
When do we choose to split tasks

Recursive Task Path - Release dependency

When do we choose to split tasks
When do we choose to split tasks

Over-synchronization solution

\[ R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 \]
Over-synchronization solution

When do we choose to split tasks

10 — WAMTA, Knoxville — Thomas Morin — February 2024
When do we choose to split tasks

Over-synchronization solution
When do we choose to split tasks
When do we choose to split tasks

Over-synchronization solution

Diagram showing tasks and their connections.
The tests were run on PlaFRIM’s bora nodes:

- 2x 18-core Cascade Lake Intel Xeon Skylake Gold 6240 @ 2.6 GHz
- 192 GB (5.3 GB/core) (@2933 MHz)
- Scheduler: Locality-aware Work-Stealing (LWS)
The tests were run on PlaFRIM’s bora nodes:

- 2x 18-core Cascade Lake Intel Xeon Skylake Gold 6240 @ 2.6 GHz
- 192 GB (5.3 GB/core) (@2933 MHz)
- Scheduler : Locality-aware Work-Stealing (LWS)

Tile sizes choosen :

- 1120 : "big" : the most efficient.
- 280 : "small": no lack of parallelism.
- 560 : "mid": trade-off.
The tests were run on PlaFRIM’s bora nodes:

- 2x 18-core Cascade Lake Intel Xeon Skylake Gold 6240 @ 2.6 GHz
- 192 GB (5.3 GB/core) (@2933 MHz)
- Scheduler : Locality-aware Work-Stealing (LWS)

Tile sizes chosen:

- 1120 : "big" : the most efficient.
- 280 : "small" : no lack of parallelism.
- 560 : "mid" : trade-off.

We split a task if:

- \( N_{\text{ready}} \leq 4N_{\text{cores}} \)
- Split efficiency \( \geq 50\% \).
Figure: Performance comparison between different Cholesky Factorization versions.
Figure: Performance comparison between different Cholesky Factorization versions.
Figure: Performance comparison between different Cholesky Factorization versions.
Figure: Performance comparison between different Cholesky Factorization versions.
Figure: Performance comparison between different Cholesky Factorization versions.
Figure: Flops evolution according to execution time during recursive-splitter Cholesky Factorization execution, with matrix of size 26880.
• Recursive tasks:
  > Insert subgraph at runtime.
  > More dynamic DAG.

• Splitting task dynamically brings different questions:
  > Which task should we split.
  > When do we choose to split.

Future Work

• Scheduling questions:
  > How should we split tasks?

• Extend current work:
  > Heterogeneous platforms.
  > Distributed recursive tasks.
Heterogeneous

Task criticality
Completion Time
Load balance
Efficiency first

Parallelism Available
Heterogeneous

Matrix order (N)

TFlop/s

Version: Tile sizes
- Non-Recursive: 1920
- Non-Recursive: 2880
- Recursive: 5760 / 960 dynamic
- Recursive: 5760 / 640 dynamic
- Recursive: 2880 / 960 dynamic
- Recursive: 2880 / 640 dynamic

dpotrf
62 AMD + 2 A100
30000 60000 90000 120000
0
10
20

16 — WAMTA, Knoxville — Thomas Morin — February 2024
Shared data

A: shared 0 & 1

A₀: 0  A₁: 1

Auto-pruning

R : A
Conclusion

Shared data

A: shared 0 & 1

A₀: 0

A₁: 1

Auto-pruning

A₀

A₁
Conclusion

Shared data

A: shared 0 & 1

A0: 0

A1: 1

Auto-pruning

Node 0:

A0
Shared data

A: shared 0 & 1

- $A_0: 0$
- $A_1: 1$

Auto-pruning

Node 1:

- $A_1$