Practical Steady-State Scheduling for Tree-Shaped Task Graphs

Sékou DIAKITÉ\textsuperscript{1}, Loris MARCHAL\textsuperscript{2}, Jean-Marc NICOD\textsuperscript{1}, Laurent PHILIPPE\textsuperscript{1} - 19/11/2009
\textsuperscript{1}: Laboratoire d’Informatique de Franche-Comté
Université de France Comté, France

\textsuperscript{2}: Laboratoire de l’Informatique du Parallélisme
CNRS - INRIA - Université de Lyon, France
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions
Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms
Synthesis
Scheduling problem

Definitions

Execution platform

- undirected graph, $G_p = (V_p, E_p)$
  - $V_p = \{P_1, \ldots, P_n\}$: $n$ processors
  - $E_p$: communication links between the processors
    - bidirectional one-port model
    - $c_{i,j}$ is the time needed to send a unit of data from $P_i$ to $P_j$

Example

Diagram:

- $P_1 \rightarrow P_2$
- $P_1 \leftrightarrow P_3$
- $P_2 \rightarrow P_4$
- $P_3 \leftrightarrow P_4$
Scheduling problem

Definitions

Application

DAG with no forks (in-trees), \( G_a = (V_a, E_a) \)
- \( V_a = \{T_1, \ldots, T_k\} : k \) tasks
  - unrelated computation model, \( w_{i,k} \) : time needed by \( P_i \) to execute \( T_k \)
- \( E_a \) dependencies between tasks
  - \( F_{k,l} \) is the amount of data (File) produced by \( T_k \) and consumed by \( T_l \)

Example

\( T_1 \)
\( T_2 \)
\( T_3 \)
\( T_4 \)

10 to 1000 times
Scheduling problem

How to?

Problem
Executing a batch of graphs (from 10 to 1000)

Objective
Minimizing the makespan $C_{max}$

Chosen method
Steady-state technique which is asymptotically optimal (throughput)
Outline

Scheduling problem

Principle of steady-state scheduling
  Overview
  Shortcomings

Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach

Using non-conservative steady-state solutions

Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms

Synthesis
This study is based on

Principle of steady-state scheduling

Overview

Converting the scheduling problem to a linear program

- the steady-state is characterized by activities variables
  - the average number of $T_k$ processed by $P_i$ in one time unit
  - the average number of $F_{k,l}$ sent by $P_i$ to $P_j$ in one time unit

- these activities variables allow us to write constraints
  - on processor speeds and link bandwidths
  - "conservation laws" to state that $F_{k,l}$ has to be produced by $T_k$ and consumed by $T_l$

- these constraints describe a valid steady-state schedule

- by adding the objective of maximizing the steady-state throughput, we obtain a linear program
Principle of steady-state scheduling

Overview

From the linear program to a periodic schedule (period)

- The optimal solution of the linear program gives rational activities.
- We cannot split tasks and files.
  - The period length $L$ is equal to the LCM of activities denominators.
  - We multiply every activity by $L$, activities are now integers.
- $L$ is large but bounded.
- The period allows us to schedule any number of graphs, the final schedule consists in 3 phases:
  - Initialization
  - Steady-state: $n \times$ periods
  - Clean-up
Principle of steady-state scheduling

Overview

Example

Platform graph

Task graph

Allocations

Steady-state period

processor $P_1$

processor $P_2$

processor $P_3$

link $P_1 \rightarrow P_3$

link $P_2 \rightarrow P_3$

$L$

$F_{1,2}$
Outline

Scheduling problem

Principle of steady-state scheduling
  Overview
  Shortcomings

Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach

Using non-conservative steady-state solutions

Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms

Synthesis
**Principle of steady-state scheduling**

**Shortcomings**

**Long latency**
- Several periods are necessary to process an instance
  - Drawback for interactive applications
  - Lead to large buffers: at every time step, a large number of ongoing job has to be stored

**Long initialization and clean-up phases**
- The period contains a large number of ongoing job
  - Long initialization phase to enter steady-state
  - Long clean-up phase to leave steady-state
- Initialization and clean-up are done with heuristic scheduling
  - We lose the benefit of the optimal steady-state phase
Principle of steady-state scheduling

Shortcomings

Addressing the shortcomings

- the original steady-state algorithm reaches good $C_{\text{max}}$ as soon as the number of instances is large enough
- in this study, we aim at reducing this threshold
Principle of steady-state scheduling

Addressing the shortcomings

Means of actions

- decrease the length of the period
  - hard to do when we want to keep an optimal period

- reduce the latency (inter/intra dependencies)
  - side benefit: less work to do in initialization and clean-up (gain on $C_{max}$)

- reduce the period length by allowing a small reduction of the throughput
  - side benefit: reducing the latency
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
    Mixed Integer Program
    Heuristic approach
Using non-conservative steady-state solutions
Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms
Synthesis
Reducing the latency

How to reduce the latency?

Intra-period dependencies.

The original steady-state (only inter-period dependencies)
Reducing the latency

Dependencies

The steady-state with intra-period dependencies

Platform graph  Task graph

- ➤ inter-period dependency
- ➤ intra-period dependency
Outline

Scheduling problem
Principle of steady-state scheduling
   Overview
   Shortcomings
Reducing the latency
   Dependencies
      Mixed Integer Program
      Heuristic approach
Using non-conservative steady-state solutions
Experimental results
   Simulation settings
   Inter-period dependencies
   Scheduling efficiency
   Number of running instances
   Running time of the algorithms
Synthesis
Reducing the latency
Mixed Integer Program

Ordering

Tasks \((T_j, T_k)\) on the same processor \(P_i\)
- binary variable \(y_{j,k} = 1\) if and only if \(T_j\) is processed before \(T_k\)
- \(t_j\) is the starting time of task \(T_j\), \(L\) is the length of the period

\[
t_j - t_k \geq -y_{j,k} \times L \quad (1)
\]
\[
y_{j,k} + y_{k,j} = 1 \quad (2)
\]
\[
t_k - (t_j + w_{i,j}) \geq (y_{j,k} - 1) \times L \quad (3)
\]
\[
t_j + w_{i,j} \leq L \quad (4)
\]
Reducing the latency
Mixed Integer Program

Dependencies

- For each dependency $T_j \rightarrow T_k$
  - binary variable $e_{j,k} = 1$ intra-period dependency ($e_{j,k} = 0$ inter-period)

$$t_k - (t_j + w_{i,j}) \geq (e_{j,k} - 1) \times L \quad (5)$$

Objective

\[
\begin{align*}
\{ & \text{Maximize } \sum e_{j,k} \\
& \text{under the constraints (1), (2), (3), (4) and (5)} \}
\end{align*}
\]
Outline

Scheduling problem
Principle of steady-state scheduling
    Overview
    Shortcomings
Reducing the latency
    Dependencies
    Mixed Integer Program
    Heuristic approach
Using non-conservative steady-state solutions
Experimental results
    Simulation settings
    Inter-period dependencies
    Scheduling efficiency
    Number of running instances
    Running time of the algorithms
Synthesis
Reducing the latency

Heuristic approach

Limitations

The heuristic algorithm is not allowed to move tasks inside the period

Algorithm 1: Heuristic algorithm

\[\text{IntraDep} \leftarrow \emptyset;\]
\[\text{Prod} \leftarrow \text{set of all sources of the dependencies (sorted by completion time);}\]
\[\text{Cons} \leftarrow \text{set of all destinations of the dependencies (sorted by starting time);}\]
\[\text{forall } T_{\text{src}} \in \text{Prod do}\]
\[\text{forall } T_{\text{dst}} \in \text{Cons do}\]
\[\text{if There is a dependency } T_{\text{src}} \rightarrow T_{\text{dst}} \text{ then}\]
\[\text{if } \text{end}(T_{\text{src}}) \leq \text{start}(T_{\text{dst}}) \text{ then}\]
\[\text{remove } T_{\text{dst}} \text{ from Cons;}\]
\[\text{IntraDep} \leftarrow \text{IntraDep} \cup \{(T_{\text{src}} \rightarrow T_{\text{dst}})\};\]
\[\text{continue}\]
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach

Using non-conservative steady-state solutions

Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms

Synthesis
Using non-conservative solutions

Motivation

How to reduce the period length?

- one of the main drawbacks of the method
- solution of a linear program
  - hard to find an other optimal solution with a smaller period

→ modify the solution

A sub-optimal solution

- decrease the system throughput
- gain flexibility on the period length

→ our claim : much shorten the period at the cost of a slight reduction of the throughput
  - side benefits : shorter latencies and smaller buffers
Using non-conservative solutions

Principle

Steady state scheduling and allocations

- allocations: \( A_1, \ldots, A_m \)
- throughput of \( A_k \) \( \rho_k = \alpha_k / \beta_k \) and total throughput \( \rho = \sum_k \rho_k \)
- period length \( T = \text{lcm}_k \beta_k \)
- in one period \( A_k \) processed \( T \times \alpha_k / \beta_k \in \mathbb{N} \)

Influence of large value of \( \beta_k \)

- contribution to a small amount the total throughput
- responsible of large period size
- suppress \( \beta_k \) from the computation of \( T \) (scale \( \lfloor (\alpha_j \times T) / \beta_j \rfloor \))
- hope: loss in \( \rho \) compensated by a shorter value of \( T \)
Using non-conservative solutions

Algorithm 2 : Shorten the period of the steady state schedule

Data : $N_{total}$ instances, $m$ allocations $A_i$ with the throughput $\alpha_i/\beta_i$.

Parameters : $K = 0.25$ (ratio initialization/total) and $L = 0.85$ (maximum degradation).

Sort allocation by non-increasing $\beta_k$, so that $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_m$

$N_{init} \leftarrow \text{estimateInitTermJobCount} (\rho_1, \ldots, \rho_m)$

$i \leftarrow 1$ ; $\rho_{\text{orig}} = \sum_{k=1}^{m} \alpha_k / \beta_k$

while $i < m - 1$ and $(N_{init}/N_{total} > K)$ and $(\rho > L \times \rho_{\text{orig}})$ do

$T \leftarrow \text{lcm}\{\beta_1, \ldots, \beta_m\}$

foreach allocation $A_k$ in $\{A_1, \ldots A_m\}$ do

\[
\rho_k^{\text{rollback}} \leftarrow \rho_k \\
\rho_k \leftarrow \left\lfloor (\alpha_k \times T) / \beta_k \right\rfloor
\]

$\rho \leftarrow \sum_{k=1}^{m} \rho_k$

$N_{init} \leftarrow \text{estimateInitTermJobCount} (\rho_1, \ldots, \rho_m)$

$i \leftarrow i + 1$

if $(N_{init}/N_{total} \leq K)$ or $(\rho \leq L \times \rho_{\text{orig}})$ then

foreach allocation $A_k$ in $\{A_1, \ldots A_m\}$ do

$\rho_k \leftarrow \rho_k^{\text{rollback}}$

return $(\rho_1, \ldots, \rho_m)$
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions
Experimental results
  Simulation settings
    Inter-period dependencies
    Scheduling efficiency
    Number of running instances
    Running time of the algorithms
Synthesis
Experimental results

Comparison of 6 algorithms

- the original steady-state implementation;
- the steady-state with the reduction of inter-period using MIP (steady-state+MIP);
- the steady-state with the reduction of inter-period using the greedy heuristic (steady-state+greedy);
- the steady-state with the non-conservative period reduction (steady-state+suboptimal);
- the steady-state with both the greedy heuristic and the non-conservative period reduction (steady-state+heuristic+suboptimal);
- a classical list scheduling algorithm based on HEFT.
Experimental results

Simulation settings

Simulator

- results are obtained with a simulator on top of SimGrid
- simulations of 200 random platform/application scenarios
- batches from 1 to 1000 task graphs
- MIP solving using CPLEX

Limitations

- the MIP solver was able to find a solution within 15 minutes for 142 SIMPLE scenarios
- in the GENERAL case we do not give the results for the MIP approach
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions

Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms

Synthesis
Experimental results

Inter-period dependencies

Results

- SIMPLE case: the MIP solves 32% of the intra-period dependencies
- SIMPLE case: the heuristic solves 26% of the intra-period dependencies
- GENERAL case: the heuristic solves 25% of the intra-period dependencies

Notes

- both the MIP and the heuristic achieve good performances for the resolution of intra-period dependencies
- how does they perform on other metrics?
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions

Experimental results
  Simulation settings
  Inter-period dependencies
    Scheduling efficiency
  Number of running instances
  Running time of the algorithms
Synthesis
Experimental results

Efficiency

ratio to the optimal throughput obtained by an algorithm

One complex example

<table>
<thead>
<tr>
<th>Efficiency (ratio to optimal throughput)</th>
<th>Number of jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>steady-state</td>
<td>10000</td>
</tr>
<tr>
<td>steady-state+heuristic</td>
<td></td>
</tr>
<tr>
<td>steady-state+suboptimal</td>
<td></td>
</tr>
<tr>
<td>steady-state+heuristic+suboptimal</td>
<td></td>
</tr>
<tr>
<td>list-scheduling heuristic</td>
<td></td>
</tr>
</tbody>
</table>

DIAKITÉ, MARCHAL, NICOD, PHILIPPE
ROMA/GRAAL working Group - 19/11/2009
Experimental results

Scheduling efficiency

Simple

General

Notes
Proportion of scenarios where we reach 90% of the optimal throughput
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions
Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
    Number of running instances
  Running time of the algorithms
Synthesis
Experimental results

Number of running instances

Example

Results

- SIMPLE case: MIP induces 30% less running instances
- SIMPLE case: heuristic induces 24% less running instances
- GENERAL case: heuristic + non-conservative induces 37% less running instances (548 down to 126)

→ shorten the buffer size
### Experimental results

#### Latency and buffer size

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SIMPLE scenarios</th>
<th>GENERAL scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average latency</td>
<td>maximum latency</td>
</tr>
<tr>
<td>MIP</td>
<td>94%</td>
<td>67%</td>
</tr>
<tr>
<td>heuristic</td>
<td>95%</td>
<td>74%</td>
</tr>
<tr>
<td>suboptimal</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>heuristic+</td>
<td>95%</td>
<td>74%</td>
</tr>
</tbody>
</table>

**Tab.:** Performance of the algorithms in latency and buffer size relatively to original steady-state implementation. (Smaller latency and number of running jobs are better.)

**NB:** GENERAL cases include SIMPLE cases too, so the decrease is important for complex cases.

---

[Image of a keyboard and other elements]
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions
Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms
Synthesis
Experimental results

Running time of the algorithms

**SIMPLE**

<table>
<thead>
<tr>
<th>Number of jobs</th>
<th>CPU time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Steady-state</td>
</tr>
<tr>
<td>10</td>
<td>Steady-state + Heuristic</td>
</tr>
<tr>
<td>100</td>
<td>Steady-state + MIP</td>
</tr>
<tr>
<td>1000</td>
<td>Steady-state suboptimal</td>
</tr>
<tr>
<td>10000</td>
<td>Steady-state suboptimal + Heuristic</td>
</tr>
<tr>
<td></td>
<td>List-scheduling heuristic</td>
</tr>
</tbody>
</table>

**GENERAL**

<table>
<thead>
<tr>
<th>Number of jobs</th>
<th>CPU time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Steady-state</td>
</tr>
<tr>
<td>10</td>
<td>Steady-state + Heuristic</td>
</tr>
<tr>
<td>100</td>
<td>Steady-state + MIP</td>
</tr>
<tr>
<td>1000</td>
<td>Steady-state suboptimal</td>
</tr>
<tr>
<td>10000</td>
<td>Steady-state suboptimal + Heuristic</td>
</tr>
<tr>
<td></td>
<td>List-scheduling heuristic</td>
</tr>
</tbody>
</table>

**Notes**

Average CPU-time in seconds
Outline

Scheduling problem
Principle of steady-state scheduling
  Overview
  Shortcomings
Reducing the latency
  Dependencies
  Mixed Integer Program
  Heuristic approach
Using non-conservative steady-state solutions
Experimental results
  Simulation settings
  Inter-period dependencies
  Scheduling efficiency
  Number of running instances
  Running time of the algorithms

Synthesis
Summary

- study of an adaptation of the steady state techniques in practical conditions:
  - medium size batches
  - performance metrics (throughput) and practical interest (latency and buffer)
- two optimizations:
  - dependency reorganization (NP-Complet: MIP + heuristic)
  - shorten the period by decreasing the throughput (< 15%)
- measure of the impact of our optimizations (efficiency, buffer size, latency)

Conclusion

- steady-state scheduling is an efficient tool for dealing collections of task graphs
Future work

- steady state techniques and tolerance to the variation the platform capabilities
- evaluation onto a GRID context (cf MAO project)
Thank you for your attention

Questions?