Optimal task scheduling

Oliver Sinnen

o.sinnen@auckland.ac.nz
www.ece.auckland.ac.nz/~sinnen/
Task scheduling

2 processors

+ 

task graph

Scheduling problem:
Spatial and temporal assignment of DAG onto processors
Optimal task scheduling

Goal: find schedule with shortest schedule length (makespan)

=> NP-hard problem

=> usually scheduling heuristics

  ● List scheduling, clustering,
  ● Genetic algorithms, etc.

Optimal task scheduling

**Goal:** find schedule with shortest schedule length (makespan)

=> NP-hard problem

=> usually scheduling heuristics

- List scheduling, clustering,
- Genetic algorithms, etc.


**Here:** Optimal scheduling algorithm

- based on A*
- for small graphs (still NP-hard)
Content

• Scheduling model
• A* principle
• A* for scheduling
• \(f\) function
• Pruning techniques
• Results
• Outlook
Scheduling model
Graph representation of program

Example:

**task graph (DAG)**

<table>
<thead>
<tr>
<th>Node</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a = 1</td>
</tr>
<tr>
<td>B</td>
<td>b = a+1</td>
</tr>
<tr>
<td>C</td>
<td>c = a*a</td>
</tr>
<tr>
<td>D</td>
<td>d = b+c</td>
</tr>
</tbody>
</table>

- Graph representation of program
- Input of task scheduling

Directed acyclic graph (DAG)

- Node (n): sub-task
- Edge (e): dependence (communication)
- Weight: computation $w(n)$ or communication time $c(e)$
Scheduling constraints

Schedule definitions: DAG: $G(V,E,w,c)$, node $n$, edge $e$

- start time: $t_s(n)$; finish time: $t_f(n) = t_s(n) + w(n)$
- processor assignment: $proc(n)$

Constraints:

- Processor constraint:
  
  $proc(n_i) = proc(n_j) \implies t_s(n_i) \geq t_f(n_j)$ or $t_s(n_j) \geq t_f(n_i)$

- Precedence constraint:
  
  for all edges $e_{ji}$ of $E$ (from $n_j$ to $n_i$)
  
  $t_s(n_i) \geq t_f(n_j) + c(e_{ji})$ if $proc(n_i) \neq proc(n_j)$
  
  $t_s(n_i) \geq t_f(n_j)$ if $proc(n_i) = proc(n_j)$
Classic system model

Properties:

- Dedicated system
- Dedicated processors
- Zero-cost local communication
- Communication subsystem
- Concurrent communication
- Fully connected

System model

e.g. 8 processors
A*
**A***

**Best first** state space search algorithm

**Example**
- Finding shortest route Lyon to Paris
- Map modelled as graph
  - Nodes: towns
  - Edges: links between towns; weights: km travelled
**A**

Best first state space search algorithm

**Example**

- Finding shortest route Lyon to Paris
- Map modelled as graph
  - Nodes: towns
  - Edges: links between towns; weights: km travelled
- Estimated final distance \( f \):
  - Distance already travelled + direct line to Paris
**A***

**Best first** state space search algorithm

**Example**

- Finding shortest route Lyon to Paris
- Map modelled as graph
  - Nodes: towns
  - Edges: links between towns; weights: km travelled
- Estimated final distance $f$:
  - Distance already travelled + direct line to Paris

[Diagram showing a graph with distances and $f$ values calculated for various nodes, including Paris and Lyon.]
A* -- principle

**Best first** state space search algorithm

- State $s$ represents partial solution to problem
- New states created by expanding state $s$ with all possible choices
- Only most promising state is expanded at a time – cost function $f(s)$
- Cost function $f(s)$
  - underestimate of minimum cost $f^*(s)$ of final solution – the tighter the better
- If $f^*(s) \geq f(s)$ for any $s$ (i.e. $f(s)$ is admissible)
  - then final solution is optimal
A* -- algorithm

- OPEN: priority list of unexamined states
- CLOSED: set of already examined states

Init
OPEN <= initial state
CLOSED <= Ø

Loop
Pop best state s from OPEN (with lowest $f(s)$)
Test if s is goal state
   If yes, finished
Expand s => create new states
   Calculate $f$ for states
   Insert in OPEN
Insert s in CLOSED
A* for task scheduling
A* for task scheduling

- State => partial schedule
- Cost function $f(s)$ => underestimate of schedule length
- State is expanded by scheduling one more node

State tree
A* for task scheduling

Like list scheduling:

- At each step choice between all free nodes, *free(s)*
- On all $p$ processors

$\Rightarrow$ *free(s) x p* new states created
$f(s)$ function
Cost function $f(s)$ principle

Principle

- Underestimate of length of final schedule
  - based on the current partial schedule of $s$
- Gauge minimum schedule length of any schedule based on current partial schedule
  - Use node levels, especially bottom level
Cost function $f(s)$

Previously:

- Finish time of latest node to finish: $t_f(n_{max})$
  - i.e. length of current partial schedule
- Plus maximum bottom level of any of its successors

$bl_w(n)$ bottom level: longest path from $n$ to exit

- path length is sum of its node weights
Cost function $f(s)$

Previously:

- Finish time of latest node to finish: $t_f(n_{max})$
  - i.e. length of current partial schedule
- Plus maximum bottom level of any of its successors

$$f_{KA}(s) = t_f(n_{max}) + \max_{n \in \text{succ}(n_{max})} \{ bl_w(n) \}$$

$bl_w(n)$ bottom level: longest path from $n$ to exit
- path length is sum of its node weights
Cost function $f(s)$

Previously:

- Finish time of latest node to finish: $t_f(n_{\text{max}})$
  - i.e. length of current partial schedule
- Plus maximum bottom level of any of its successors

\[
f_{KA}(s) = t_f(n_{\text{max}}) + \max_{n \in \text{succ}(n_{\text{max}})} \{bl_w(n)\}
\]

\[
= t_s(n_{\text{max}}) + bl_w(n_{\text{max}})
\]

$bl_w(n)$ *bottom level*: longest path from $n$ to exit

- path length is sum of its node weights
Proposed cost function $f(s)$

Part 1:
- Maximum of start time and bottom level of any node
  - Better, since last node to finish can have small bottom level
  - Can be calculated incrementally, i.e. $O(1)$ for each state

$$f_1(s) = \max_{n \in \text{partial sched. of } s} \left\{ t_s(n) + bl_w(n) \right\}$$
Proposed cost function $f(s)$

Part 2:

Schedule length is bounded by total computation load plus total idle time over number of processors $p$

- $idle(s)$: total idle time of partial schedule of $s$

$$f_2(s) = (idle(s) + \sum_{n \in V} w(n))/p$$
Proposed cost function $f(s)$

Part 3:

- Part 1 only considers already scheduled nodes
- Now, additionally consider all *free* nodes
- Minimal Data Ready Time
  - Earliest time a node can start on any processor

\[
t_{dr}(n) = \min_{P \in P} \left\{ t_{dr}(n, P) \right\}
\]
Proposed cost function $f(s)$

Part 3:
- Part 1 only considers already scheduled nodes
- Now, additionally consider all *free* nodes
- Minimal Data Ready Time
  - Earliest time a node can start on any processor
    
    $$t_{dr}(n) = \min_{P \in P} \{ t_{dr}(n, P) \}$$

$$f_3(s) = \max_{n \in \text{free}(s)} \{ t_{dr}(n) + bl_w(n) \}$$
Proposed cost function $f(s)$

Proposed cost function:

- Maximum of all three parts

\[
f(s) = \max \{ f_1(s), f_2(s), f_3(s) \}
\]

\[
f_1(s) = \max_{n \in \text{partial sched. of } s} \{ t_s(n) + bl_w(n) \}
\]

\[
f_2(s) = (\text{idle}(s) + \sum_{n \in V} w(n)) / p
\]

\[
f_3(s) = \max_{n \in \text{free}(s)} \{ t_{dr}(n) + bl_w(n) \}
\]
Pruning techniques
Detecting duplicates

Pruning of unpromising subtrees

• Duplicate states
  – Identical partial schedule
  – Only one needs to be expanded
• Automatically detected when inserting into OPEN and comparing with CLOSED

=> Importance of CLOSED set
Processor normalisation

Normalising schedules

- Condition: homogeneous processors

Principle

- Order nodes, e.g., $a,b,c,d \ldots$
- Name processor on which first node, $a$, is schedule $P_1$, name processor of $b$ $P_2$ (unless on same processor as $a$) and so on

$\Rightarrow$ leads to more duplicates
Node equivalence

If nodes are equivalent, it is enough to only consider (any) one scheduling order

- Implemented with virtual edges
- Reduces number of free nodes

$\Rightarrow$ branching factor

Node equivalence:
- same node weights
- same predecessors and successors
- same in/out edge weights
Other pruning/optimisation

- Use heuristic (list scheduling) to find upper bound on schedule length
  - All states with higher $f(s)$ value can be discarded
    - Would never be expanded
  => saves memory and makes OPEN list faster
- Partial expansion
  - Only expand a state until a new state with same $f(s)$ is found
  - One node at a time, not all free nodes
  => go deep as fast as possible
  - Good if $f(s) = f_{opt}(s)$
Experiments
Experiments

- A* scheduling implemented in Java
- Workload
  - Large set of task graphs
    - 5-40 nodes
    - Different structures
    - Different weights
- Scheduled on 2, 4 and 8 processors
- Two minute time limit => unfinished not included in results

Remember, all produced schedules are optimal
- observing runtime of scheduling algorithm, not schedule length
  - normalised as number of states
  - actual runtime ranged from seconds to minutes
Results

- Proposed $f(s)$ function dramatically more efficient
Results

- New pruning technique -- processor normalisation -- significantly reduces number of states.
Results

- Pruning technique – node equivalence -- significantly reduces number of states
Results

- Runtime graph structure dependent!
Results

- Importance of CLOSED list
Results

- Improvements can occasionally lead to more states
Outlook

• With contention
  – One port model
• Round costs (accepting cost errors) to make more nodes identical
  – Approximation ratio
• Only clustering
  – Using properties of clustering
  – e.g. optimal linear clustering exists for coarse grain graphs
• Node duplication
Conclusions

A* based optimal task scheduling algorithm

• Improved previous approach
  – Dramatically improved $f(s)$ cost function
  – Introduced new efficient pruning techniques

• New insights from experimental results
  – Importance of task graph structure
    • Branching factor

Suitable for small task graphs

• For time crucial applications
• To evaluate scheduling algorithms
• For optimally scheduling kernels of iterative computations
Results

- Evaluation of heuristics possible – here List Scheduling