Advances in Optimal Task Scheduling

Oliver Sinnen

PARALLEL AND RECONFIGURABLE COMPUTING GROUP

Department of Electrical and Computer Engineering
University of Auckland
New Zealand

www.ece.auckland.ac.nz/~parallel/
Introduction

Scheduling problem

task graph (DAG) with computation and communication costs on $p$ processors, minimizing makespan $\rightarrow$ NP-hard

Optimal solution with A*

Extensive search through all possible schedules $\rightarrow$ guided by cost function (best-first-search)
Introduction

Optimal solution with A*

- Good for small graphs (<30 tasks), few processors (<8)

Paradox

- Difficult for “simple” scheduling problems! (i.e. task number limit is low)
  - Independent tasks (mostly)
  - Forks or joins
  - Fork-joins

Contribution

- Novel concepts and techniques to address this and more: reduce search space, accelerate search
Outline

- Define scheduling problem
- State space formulation and A*
- Pruning techniques
- Some observations
- Equivalent schedule
- Fix order ready list
- First evaluation results
- Conclusions
Scheduling problem: $P|\text{prec},c_{ij}|C_{\text{max}}$

Task graph and $p$ (identical) processors

DAG: tasks ($n$) and edges ($e$) with weights (computation cost $w(n)$ and communication cost $c(e)$)

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

Objective: minimise makespan

Constraints:

- Processor constraint:
  
  $\text{proc}(n_i)=\text{proc}(n_j) \Rightarrow t_s(n_i) \geq t_f(n_j) \text{ or } t_s(n_j) \geq t_f(n_i)$

- 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 $\text{proc}(n_i) \neq \text{proc}(n_j)$
  
  $t_s(n_i) \geq t_f(n_i)$ if $\text{proc}(n_i) = \text{proc}(n_j)$
Solution space

Solution

processor allocation and task order

- Allocation problem *plus*
- Permutation problem
- Each problem is NP-hard

Finding optimal solution

- Trying all possible processor allocations with all possible permutations (*naïve*)

\[ p^V \times V! \]

- Example: 10 tasks, 3 processors: 219 billion possibilities!
A* for task scheduling

A*: **best first** search, guided by cost function

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

At each step, expand most promising state with best cost $f(s)$

- All free nodes, $\text{free}(s)$, scheduled on all $p$ processors

$\Rightarrow \text{free}(s) \times p$ new states created, costs calculated

State tree
A* algorithm

- Priority queue OPEN for states to be expanded (ordered by $f(s)$)
- Recording already expanded states: duplication detection

=> very, very memory hungry

- Cost function $f(s)$: underestimate of minimum cost $f^*(s)$ of final solution – the tighter the better → less states
Pruning

- State space is tree
- Try to prune entire branches early to limit search space

So far

- Duplication detection
- Processor normalisation
- Identical tasks
Pruning – duplicated states

- Same schedule created in different ways
  => duplicates

- Duplicates are detected and discarded

Global ordering
→ local ordering
- Permutations per processor with less tasks
Pruning – processor normalisation

- Equivalent schedules for homogeneous processors
- Normalise schedule/state
  - e.g. processor of task a is always $P_1$
- Pruned as duplicates
- Processor allocation problem $\rightarrow$ subset problem
New pruning

- Address task ordering

Observations in the next slides
- Independent tasks
- Fork, Joins
- Mixed graphs
Independent tasks

Two schedules:
- Same length
- Same processor allocation
- Different task order

=> Order does not matter
=> Only allocation problem!

(with processor normalisation: only subset problem)
- Number of states reduced by $V!$
Fork and joins

Fork/Join graphs
- Task order does matter

BUT
- Optimal processor allocation enough
- Ordering by non-decreasing in-edge weight (join: by non-increasing out-edge weight)

=> still only allocation problem (strong NP-hard)
Using observations in A*

- OK, for some graphs order does not matter/can be computed
  - Independent, fork, join
- But how does that help us with general graphs?

=> First, look at mixed graphs
Independent tasks

- Independent tasks F, G, H can be in any order on $P_2$ (between C and E)

Tasks must be
- Independent
- Consecutive
- On same processor
Forks (or sink tasks)

- Tasks F, G, H might be reordered on P\(_2\) (between C and E)
  - Depends on data arrival time

Rules
- Last task finishes at the same time/earlier as originally
- Consecutive tasks
- No out edges → no consequence for rest of schedule
Using observations in A*

OK, also meaningful for mixed graphs
- But how to use these observations in A*?

=> Scheduling horizon and equivalent schedules concept
Equivalent schedule pruning
Schedule horizon

Partial schedules of same tasks

Only relevant:
- Finish time of processors
- Potential start time of successors

=> Schedule Horizon

- Some partial schedules dominate others (better in every aspect) → discard
Using schedule horizon

• How to use this in A*?

Remember: To create a new state, one more task is scheduled at the end on one processor

Approach
• Try to “bubble up” the task to bring into certain order (index order/alphabetical order)
  - Accept if horizon does not get worse

=> normalisation
=> duplicate detection
Example using schedule horizon

**Top:**
partial schedule

**Bottom:**
- Schedule I on $P_3$ at the end (after L)
  
  => new horizon
  
  (here new finish time for $P_3$)
Example using schedule horizon

Try to bubble up I

- Swap of I and L can be made → alphabetical order
- Finish time of P_3 the same
- Communication from L now later, but communication from E dominates

=> Horizon is the same

- Task I could go higher, but already in alphabetical order (Also communication from D might arrive too late on P_4)
Equivalent scheduling pruning

Procedure

- Create new state by scheduling free task $n$ on processor $P$
- Try to “bubble up” task $n$ until
  - In index order of tasks or
  - Schedule horizon gets worse

$\Rightarrow$ schedules are normalised $\Rightarrow$ duplicates are detected

Index order necessary, otherwise different schedules still possible
Equivalent scheduling pruning

Testing for outgoing edges of moved tasks not trivial:

- For each moved task, check that child tasks have the same earliest start time on every processor as before
- Some parents of these tasks might not have been scheduled yet (use best case for unknowns)
Discard instead of normalise

- Normalisation and duplication detection problematic for f-value
  - Idle time might be reduced → better estimate
  - Estimate change not allowed (at least problematic) in A*
  - Better discard state immediately
Discard instead of normalise

- Normalisation and duplication detection problematic for f-value
  - Idle time might be reduced → better estimate
  - Estimate change not allowed (at least problematic) in A*
  - Better discard state immediately

- Remember, all processor allocations and all task orders are created

- Can discard state as soon as we can “bubble up” by at least one position (because this state will be created anyway)

=> much better than normalisation (less computing)
Fixed order free list
Fixed order free list

- Discarding equivalent schedules is great
- Even better would be not to create them!

We know:
- Certain tasks can be scheduled in certain orders
  - Independent: any order
  - Fork: by non-decreasing in-edge weight

Using this in A*
- When free task list only contains such tasks, fix order
  → effectively reduce list to one element in each step
  → reduces branching factor in tree
  - Safe: without out-edges there will be no new task coming to free list
Fixed order free list

When free task list only contains such tasks, fix order

Can be extended to

- **Join**: order tasks by non-increasing out-edge weight
  - Safe, for a single join, as out-edge of all tasks goes to the same task
- **Fork-join**: order by non-decreasing in-edge weight
  - If this can also be an order by non-increasing out-edge weight → fix order
  - Safe, all out-edges go to same task
- **Mixtures of independent, fork, join, fork-join** also work
  - Generalised by treating missing edges as zero weight
Combining pruning

• Equivalent schedule pruning
• Fixed free list order

Problem
• Using combination of techniques can destroy assumptions
  – e.g. that all permutations are created

Solution
• When fixing the list order, equivalent schedule pruning is disable (not necessary anyway)
First experimental results

<table>
<thead>
<tr>
<th>Graph type</th>
<th>#tasks</th>
<th>processors</th>
<th>States before</th>
<th>States with new pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>30</td>
<td>8</td>
<td>&gt;370m</td>
<td>37m</td>
</tr>
<tr>
<td>Fork</td>
<td>21</td>
<td>2</td>
<td>&gt;19m</td>
<td>74</td>
</tr>
<tr>
<td>Fork</td>
<td>12</td>
<td>2</td>
<td>3.5m</td>
<td>36</td>
</tr>
<tr>
<td>Join</td>
<td>16</td>
<td>2</td>
<td>&gt;200m</td>
<td>18358</td>
</tr>
<tr>
<td>Join</td>
<td>16</td>
<td>4</td>
<td>&gt;&gt;200m</td>
<td>9.3m</td>
</tr>
<tr>
<td>Out-tree</td>
<td>18</td>
<td>2</td>
<td>&gt;17m</td>
<td>488224</td>
</tr>
<tr>
<td>In-tree</td>
<td>21</td>
<td>2</td>
<td>-</td>
<td>&gt;180m</td>
</tr>
<tr>
<td>In-tree</td>
<td>14</td>
<td>2</td>
<td>&gt;200m</td>
<td>115m</td>
</tr>
<tr>
<td>Random (density 2)</td>
<td>30</td>
<td>2</td>
<td>106</td>
<td>28m</td>
</tr>
<tr>
<td>Stencil</td>
<td>24</td>
<td>2</td>
<td>15.9m</td>
<td>2.3m</td>
</tr>
<tr>
<td>Stencil</td>
<td>24</td>
<td>3</td>
<td>163m</td>
<td>29m</td>
</tr>
</tbody>
</table>

- All random weights, CCR 1.0

=> no node equivalence pruning
Conclusions

New pruning techniques, motivated by A*'s promise, but bad results with simple graphs

- Equivalent schedule pruning
- Fixed order free list

Dramatically improves results for
  - Independent, fork, join
  - Also in/out trees, others

- Goal: do not generate duplicates → get rid of duplication detection
- At the moment: implement memory bounded A*, e.g. SAM*