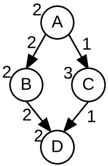
SoftEng306
Software Engineering Design 2

A/Prof Oliver Sinnen

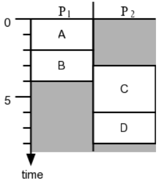
Parallel and Reconfigurable Computing (PARC) lab
Department of Electrical, Computer, and Software Engineering
University of Auckland

Task scheduling with communication delays

Scheduling task graphs with communication delays on homogeneous processors

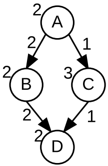


$\Rightarrow$

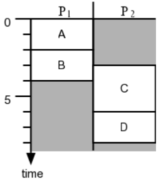


Task scheduling with communication delays

Scheduling task graphs with communication delays on homogeneous processors



$\Rightarrow$



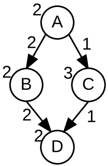
$P|prec, c_{ij}|C_{max}$

- Traditional and general problem
- Strong NP-hard

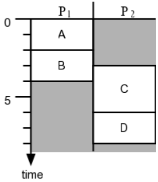
$\Rightarrow$ Heuristics, most popular is list scheduling

Task scheduling with communication delays

Scheduling task graphs with communication delays on homogeneous processors



$\Rightarrow$



$P|prec, c_{ij}|C_{max}$

- Traditional and general problem
- Strong NP-hard

$\Rightarrow$ Heuristics, most popular is list scheduling

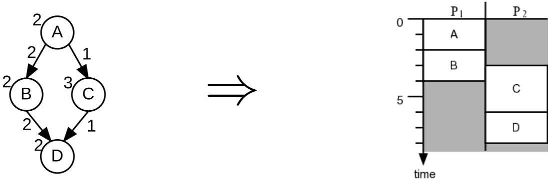
But here,

$\Rightarrow$ Optimal solver, based on state space search

Content

Scheduling problem

Finding start time and processor allocation for every task

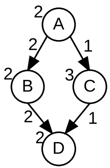


- $t_s(n)$: start time of task n
- $proc(n)$: processor of task n

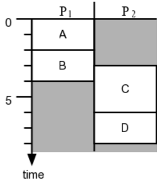
Given by task graph $G = (V, E, w, c)$

- $w(n)$: execution time of task n
  - weight of node
- $c(e_{ij})$: remote communication cost between tasks n_i and n_j
  - weight of edge

Constraints



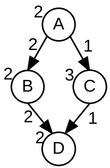
$\Rightarrow$



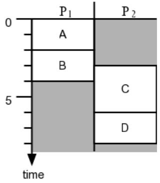
Processor constraint

$$proc(n_i) = proc(n_j) \Rightarrow \begin{cases} & t_s(n_i) + w(n_i) \leq t_s(n_j) \\ \text{or} & t_s(n_j) + w(n_j) \leq t_s(n_i) \end{cases}$$

Constraints



$\Rightarrow$



Processor constraint

$$proc(n_i) = proc(n_j) \Rightarrow \begin{cases} & t_s(n_i) + w(n_i) \leq t_s(n_j) \\ \text{or} & t_s(n_j) + w(n_j) \leq t_s(n_i) \end{cases}$$

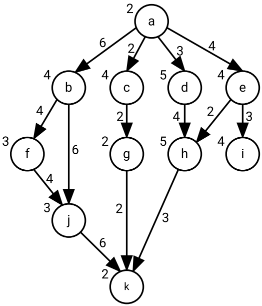
Precedence constraint

For each edge e_{ij} of E

$$t_s(n_j) \geq t_s(n_i) + w(n_i) + \begin{cases} 0 & \text{if } proc(n_i) = proc(n_j) \\ c(e_{ij}) & \text{otherwise} \end{cases}$$

Critical path and bottom level

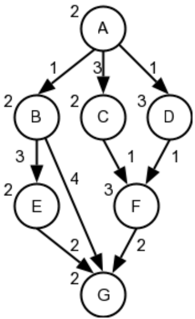
- Path length (here): sum of task weights on path
- Critical path: longest path through graph
  - Here: a, d, h, k and a, b, f, j, k, length 14
- Bottom level: longest path to exist task starting with node
  - E.g.: $bl_w(a) = 14$, $bl_w(b) = 12$, $bl_w(h) = 7$

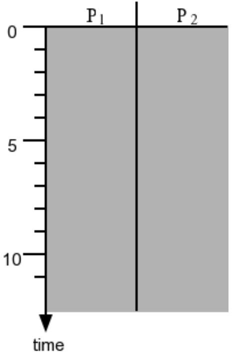


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

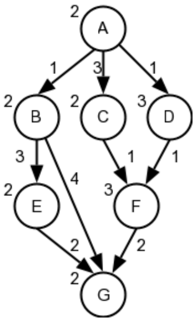


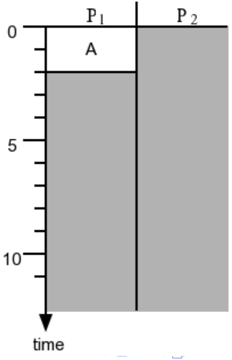


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

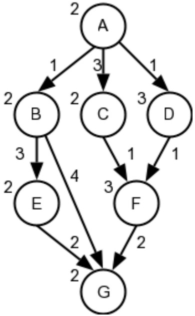


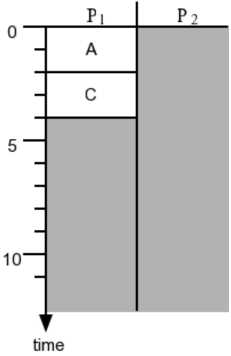


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

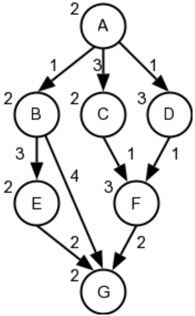


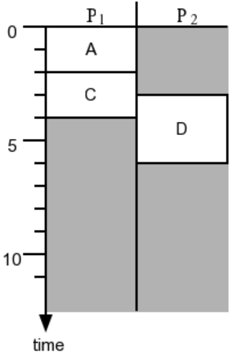


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

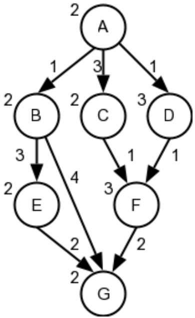


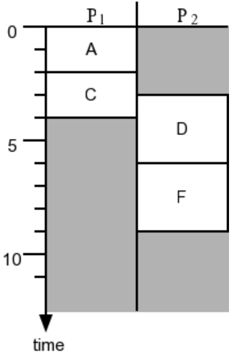


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

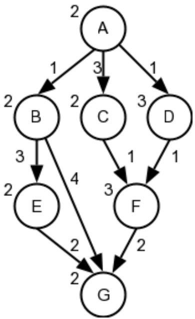


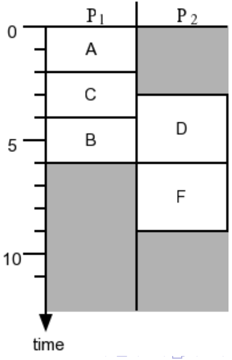


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

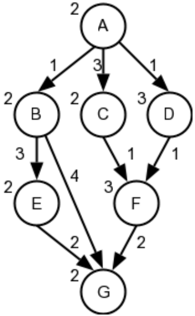


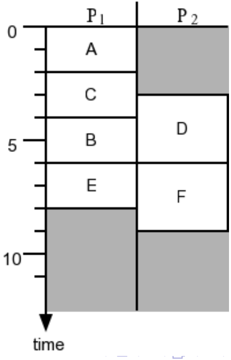


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G

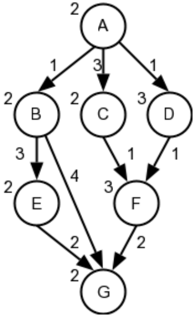


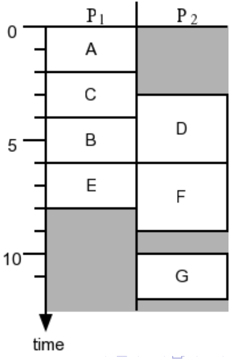


List-Scheduling

1. Order nodes of DAG according to a priority, while respecting their dependences
2. Iterate over node list from 1.) and schedule every node to the processor that allows its earliest start time.

Here: node order: A,C,D,F,B,E,G





Exhaustive solution search

- State Space Search
  - Exhaustive search through all possible solutions
  - Every state (node) s represents partial solution
  - Combinatorial problems $\Rightarrow$ search tree
  - Deeper nodes are more complete solutions

Exhaustive solution search

- State Space Search
  - Exhaustive search through all possible solutions
  - Every state (node) s represents partial solution
  - Combinatorial problems $\Rightarrow$ search tree
  - Deeper nodes are more complete solutions
- Search techniques
  - Branch and Bound – easy, limited memory search techniques
  - A* – great performance, but memory problem !

Solution space for scheduling problem

One possibility: like list scheduling, trying out all task orders and all processor allocations

- State: partial schedule
- Initial state: empty schedule
- Cost function $f(s)$: underestimate of makespan for complete schedule based on s

Solution space for scheduling problem

One possibility: like list scheduling, trying out all task orders and all processor allocations

- State: partial schedule
- Initial state: empty schedule
- Cost function $f(s)$: underestimate of makespan for complete schedule based on s

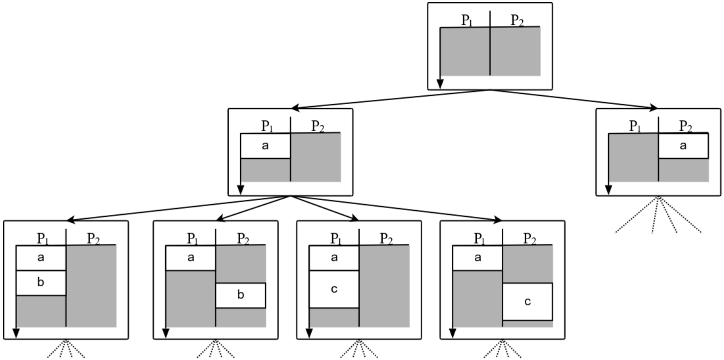
Expansion

- Given state s, let $\mathbf{free}(s)$ be free tasks

for all $i \in \mathbf{free}(s)$ **do**
    for all $P \in \mathbf{P}$ **do**
        Create new state: i scheduled on P as early as possible

Solution tree

- Task graph on two processors



Lower bounds on (partial) schedules

- Perfect load balance plus current idle time

$$\frac{\sum_{i \in \mathbf{V}} w(n_i) + idle(s)}{|\mathbf{P}|}$$

Lower bounds on (partial) schedules

- Perfect load balance plus current idle time

$$\frac{\sum_{i \in \mathbf{V}} w(n_i) + idle(s)}{|\mathbf{P}|}$$

- Max (start time of scheduled tasks plus their bottom level)

$$\max_{n_i \in s}\{t_s(n_i) + bl_w(n_i)\}$$

DFS Branch and Bound

- Depth First Search (DFS) branch and bound
- Usual meaning of "Branch and Bound"

DFS Branch and Bound

- Depth First Search (DFS) branch and bound
- Usual meaning of "Branch and Bound"

B & B

$B \leftarrow upperBound$
DFS on state space (depth until $f(s) \geq B$):
if complete solution s_c found & $f(s_c) < B$ **then**
  $B \leftarrow f(s_c)$

DFS Branch and Bound

- Depth First Search (DFS) branch and bound
- Usual meaning of "Branch and Bound"

B & B

$B \leftarrow upperBound$
DFS on state space (depth until $f(s) \geq B$):
if complete solution s_c found & $f(s_c) < B$ **then**
  $B \leftarrow f(s_c)$

- Memory required is $O(|V|P)$
- Benefits from tight upper bounds for initial B

A*

- Best first search
  - Expand most promising state first (best $f(s)$) $\Rightarrow$ Head of *OPEN*
  - Cost $f(s)$ must be underestimate to find optimal solution

A*

- Best first search
  - Expand most promising state first (best $f(s)$) $\Rightarrow$ Head of $OPEN$
  - Cost $f(s)$ must be underestimate to find optimal solution

A*

$OPEN \leftarrow emptyState$
while $OPEN \neq \emptyset$ **do**
  $s \leftarrow PopHead(OPEN)$
  if s is complete solution **then**
    return s as optimal solution
  Expand state s into children and compute $f(s_{child})$ for each
  $OPEN \leftarrow$new states

A*

- Best first search
  - Expand most promising state first (best $f(s)$) $\Rightarrow$ Head of *OPEN*
  - Cost $f(s)$ must be underestimate to find optimal solution

A*

$OPEN \leftarrow emptyState$
while $OPEN \neq \emptyset$ **do**
  $s \leftarrow PopHead(OPEN)$
  if s is complete solution **then**
    return s as optimal solution
  Expand state s into children and compute $f(s_{child})$ for each
  $OPEN \leftarrow$new states

- Very, very memory hungry (Breadth First Search)
- With given $f(s)$ function, A* explores least number of states!