Topic 3
Scheduling on Parallel Processors

Minimizing Due Date

EDF algorithm

$$P\,|r_j\,, \tilde{d}_j|\,-$$

Earliest Deadline First Scheduling Policy

- means that the task that has the earliest deadline (task that has to be processed first) is to be scheduled next.
- EDF scheduler views task deadlines as more important than task priorities.
- Experiments have shown that the earliest deadline first policy is the most fair scheduling algorithm.

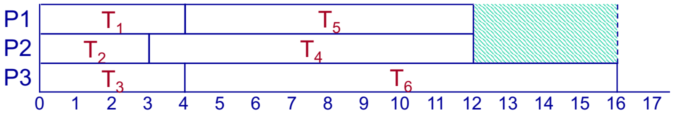
EDF algorithm

Example:

Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with (deadline, total execution time, arrival time):

$$T_1 = (5, 4, 0), T_2 = (6, 3, 0), T_3 = (7, 4, 0),$$
$$T_4 = (12, 9, 2), T_5 = (13, 8, 4), T_6 = (16, 12, 2)$$

$$T(0) = \{T_1, T_2, T_3\}$$
$$T(2) = \{T_4, T_6\}$$
$$T(3) = \{T_6\}$$
$$T(4) = \{T_5, T_6\}$$
$$T(5) = \emptyset$$



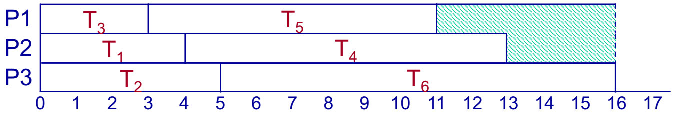
EDF algorithm

Example:

Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with (deadline, total execution time, arrival time):

$$T_1 = (5, 4, 0), T_2 = (6, 5, 0), T_3 = (4, 3, 0),$$
$$T_4 = (13, 9, 4), T_5 = (11, 8, 3), T_6 = (16, 11, 2)$$

$$T(0) = \{T_3, T_1, T_2\}$$
$$T(2) = \{T_6\}$$
$$T(3) = \{T_5, T_6\}$$
$$T(4) = \{T_4, T_6\}$$
$$T(5) = \{T_6\}$$
$$T(6) = \emptyset$$



LL algorithm

Example: Set of independent tasks $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with (deadline, total execution time, arrival time):

$$T_1 = (5, 4, 0), T_2 = (6, 3, 0), T_3 = (7, 4, 0),$$
$$T_4 = (12, 9, 2), T_5 = (13, 8, 4), T_6 = (15, 12, 2)$$

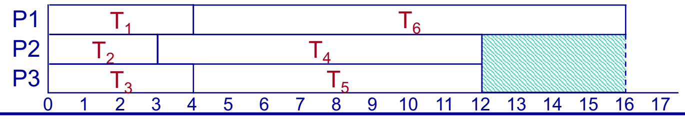
Laxity = (Deadline – (Current schedule time + Rest of Task Exec. Time)

$T(0)$: $T_1 = 5 - (0 + 4) = 5 - 4 = 1,$
$T_2 = 6 - (0 + 3) = 6 - 3 = 3,$
$T_3 = 7 - (0 + 4) = 7 - 4 = 3,$

$T(2)$: $T_4 = 12 - (2 + 9) = 12 - 11 = 1,$
$T_6 = 15 - (2 + 12) = 15 - 14 = 1,$

$T(3)$: $T_4 = 12 - (3 + 9) = 12 - 12 = 0,$
$T_6 = 15 - (3 + 12) = 15 - 15 = 0,$

$T(4)$: $T_5 = 13 - (4 + 8) = 13 - 12 = 1,$
$T_6 = 15 - (4 + 12) = 15 - 16 = -1,$



LL algorithm

Example: Set of independent tasks $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with **preemptions** and (deadline, total execution time, arrival time):

$$T_1 = (5, 4, 0), T_2 = (6, 3, 0), T_3 = (7, 4, 0),$$
$$T_4 = (12, 9, 2), T_5 = (13, 8, 4), T_6 = (15, 12, 2)$$

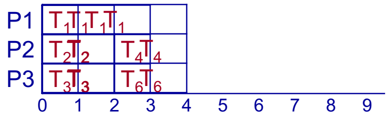
Laxity = (Deadline – (Current schedule time + Rest of Task Exec. Time)

$T(0)$: $T_1 = 5 - (0 + 4) = 5 - 4 = 1,$
$T_2 = 6 - (0 + 3) = 6 - 3 = 3,$
$T_3 = 7 - (0 + 4) = 7 - 4 = 3.$

$T(1)$: $T_1 = 5 - (1 + 3) = 5 - 4 = 1,$
$T_2 = 6 - (1 + 2) = 6 - 3 = 3,$
$T_3 = 7 - (1 + 3) = 7 - 4 = 3.$

$T(2)$: $T_1 = 5 - (2 + 2) = 5 - 4 = 1,$
$T_2 = 6 - (2 + 1) = 6 - 3 = 3,$
$T_3 = 7 - (2 + 2) = 7 - 4 = 3,$
$T_4 = 12 - (2 + 9) = 12 - 11 = 1,$
$T_6 = 15 - (2 + 12) = 15 - 14 = 1.$

$T(3)$: $T_1 = 5 - (3 + 1) = 5 - 4 = 1,$
$T_2 = 6 - (3 + 1) = 6 - 4 = 2,$
$T_3 = 7 - (3 + 2) = 7 - 5 = 2,$
$T_4 = 12 - (3 + 8) = 12 - 11 = 1,$
$T_6 = 15 - (3 + 11) = 15 - 14 = 1.$



LL algorithm

Example: Set of independent tasks $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with preemptions and (deadline, total execution time, arrival time):

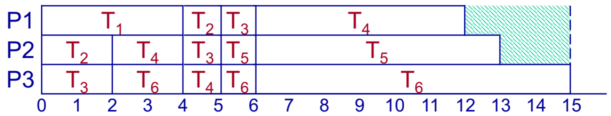
$$T_1 = (5, 4, 0), T_2 = (6, 3, 0), T_3 = (7, 4, 0),$$
$$T_4 = (12, 9, 2), T_5 = (13, 8, 4), T_6 = (15, 12, 2)$$

Laxity = (Deadline – (Current schedule time + Rest of Task Exec. Time)

$T(4)$: $T_2 = 6 - (4 + 1) = 6 - 5 = 1,$
$T_3 = 7 - (4 + 2) = 7 - 6 = 1,$
$T_4 = 12 - (4 + 7) = 12 - 11 = 1,$
$T_5 = 13 - (4 + 8) = 13 - 12 = 1,$
$T_6 = 15 - (4 + 10) = 15 - 14 = 1.$

$T(5)$: $T_3 = 7 - (5 + 1) = 7 - 6 = 1,$
$T_4 = 12 - (5 + 6) = 12 - 11 = 1,$
$T_5 = 13 - (5 + 8) = 13 - 13 = 0,$
$T_6 = 15 - (5 + 10) = 15 - 15 = 0.$

$T(6)$: $T_4 = 12 - (6 + 6) = 12 - 12 = 0,$
$T_5 = 13 - (6 + 7) = 13 - 13 = 0,$
$T_6 = 15 - (6 + 9) = 15 - 15 = 0.$

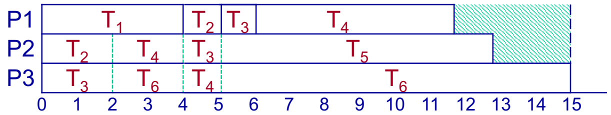


LL algorithm

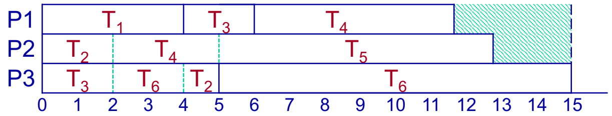
Example: Set of independent tasks $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with preemptions and (deadline, total execution time, arrival time):

$$T_1 = (5, 4, 0), T_2 = (6, 3, 0), T_3 = (7, 4, 0),$$
$$T_4 = (12, 9, 2), T_5 = (13, 8, 4), T_6 = (15, 12, 2)$$

least laxity schedule (with preemptions): ≤ 8 preemptions, total execution time is 15



Optimal schedule with 3 preemptions, total execution time = 15



$P \mid r_j, \tilde{d}_j \mid -$

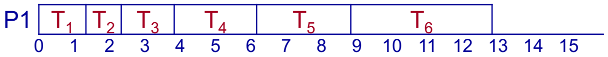
Example: Set of independent tasks $\mathrm{T} = \{T_1, T_2, \ldots, T_6\}$ and (deadline, total execution time, arrival time):

$$T_1 = (5, 4, 0), T_2 = (6, 3, 0), T_3 = (7, 4, 0),$$
$$T_4 = (12, 9, 2), T_5 = (13, 8, 4), T_6 = (15, 12, 2)$$

Execution on a single, three times faster processor:

possible with no preemptions; total execution time is 40/3

$$T(0) = \{T_1, T_2, T_3\}$$
$$T(1.33) = \{T_2, T_3\}$$
$$T(2.33) = \{T_3, T_4, T_5\}$$
$$T(3.66) = \{T_4, T_5\}$$
$$T(6.66) = \{T_5, T_6\}$$
$$T(9.33) = \{T_6\}$$



Hence: a larger number of processors is not necessarily advantageous

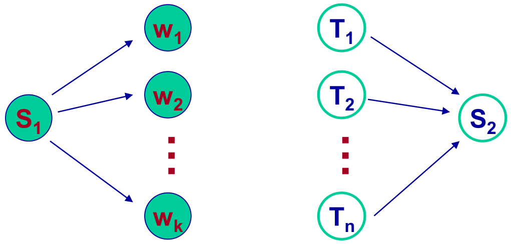
Feasibility

Given an instance of $P \mid pmtn, r_j, \tilde{d}_j \mid -$

let $e_0 < e_1 < \cdots < e_k$, $k \leq 2n - 1$ be the ordered sequence of release times and deadlines together (e_1 stands for r_i or $\tilde{d}_i$), time intervals.

1) Construct a network with source, sink and two sets of nodes: the first set (nodes w_i) corresponds to time intervals in a schedule; node w_i corresponds to interval $[e_{i-1}, e_i]$, $i = 1, 2, \ldots k$

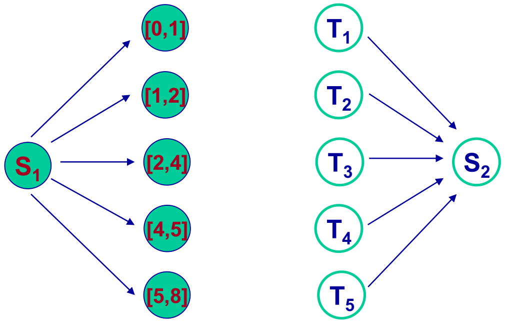
2) The second set corresponds to the tasks



Feasibility

Example: $n = 5, m = 2,$

$$p = [5, 2, 3, 3, 1], r = [2, 0, 1, 0, 2], and\ \tilde{d} = [8, 2, 4, 5, 8]$$
$$r \cup \tilde{d} = [2, 0, 1, 0, 2] \cup [8, 2, 4, 5, 8]$$
$$= [0, 1, 2, 4, 5, 8]$$
$$e_0 = [0,1], e_1 = [1,2], e_2 = [2,4], e_3 = [4,5], and\ e_4 = [5,8]$$



Feasibility

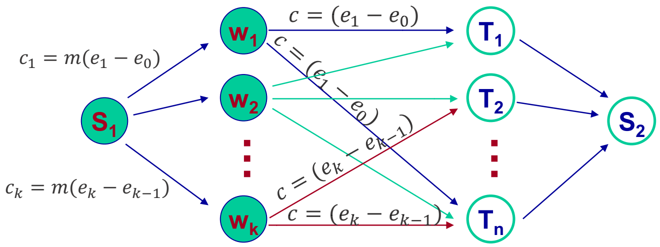
Given an instance of $P \mid pmtn, r_j, \tilde{d}_j \mid -$

Flow conditions:

1) The capacity of an arc joining the source to node w_i is $m(e_i - e_{i-1})$ this corresponds to the total processing capacity of m processors in this interval

2) If task T_j is allowed to be processed in interval $[e_{i-1}, e_j]$ then w_i is joined to T_j by an arc of capacity $e_i - e_{i-1}$

3) Node T_j is joined to the sink of the network by an arc with lower and upper capacity equal to p_j



Feasibility

Example: $n = 5, m = 2,$

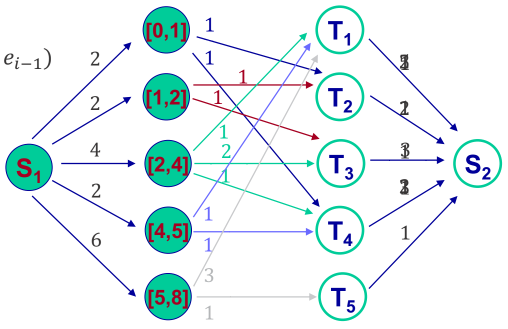
$$p = [5, 2, 3, 3, 1], r = [2, 0, 1, 0, 2], and\ \tilde{d} = [8, 2, 4, 5, 8]$$

$$r \cup \tilde{d} = [2, 0, 1, 0, 2] \cup [8, 2, 4, 5, 8]$$

$$= [0, 1, 2, 4, 5, 8]$$

$$e_0 = [0,1], e_1 = [1,2], e_2 = [2,4], e_3 = [4,5], and\ e_4 = [5,8]$$

$c = (e_i - e_{i-1})$



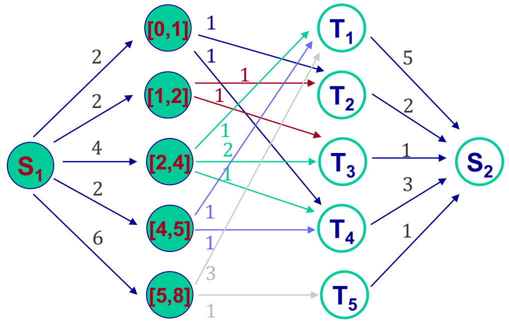
Feasibility

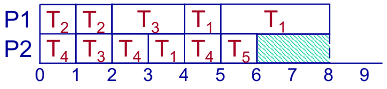
Example: $n = 5, m = 2,$

$$p = [5, 2, 3, 3, 1], r = [2, 0, 1, 0, 2], and\ \tilde{d} = [8, 2, 4, 5, 8]$$

$$r \cup \tilde{d} = [2, 0, 1, 0, 2] \cup [8, 2, 4, 5, 8]$$

$$= [0, 1, 2, 4, 5, 8]$$





Feasibility

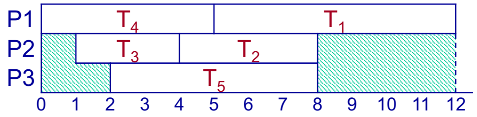
Example: $n = 5, m = 3,$

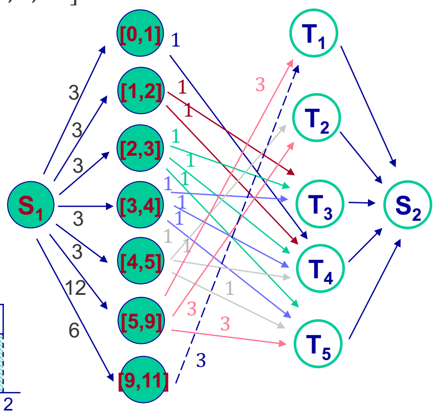
$$p = [7, 4, 3, 5, 6], r = [2, 3, 1, 0, 2], and\ \tilde{d} = [11, 9, 4, 5, 9]$$

$$r \cup \tilde{d} = [2, 3, 1, 0, 2] \cup [11, 9, 4, 5, 9]$$

$$= [0, 1, 2, 3, 4, 5, 9, 11]$$

$T(0) = \{T_4\},$	$p = [7, 4, 3, 5, 6]$
$T(1) = \{T_3\},$	$p = [7, 4, 3, 4, 6]$
$T(2) = \{T_5, T_1\},$	$p = [7, 4, 2, 3, 6]$
$T(3) = \{T_2, T_1\},$	$p = [7, 4, 1, 2, 5]$
$T(4) = \{T_2, T_1\},$	$p = [7, 4, 0, 1, 4]$
$T(5) = \{T_1\},$	$p = [7, 3, 0, 0, 3]$
$T(8) = \emptyset,$	$p = [4, 0, 0, 0, 0]$
$T(12) = \emptyset,$	$p = [0, 0, 0, 0, 0]$





Feasibility

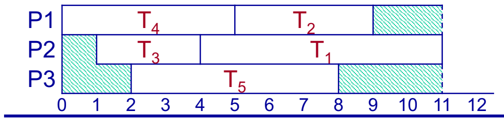
Example: $n = 5, m = 3,$

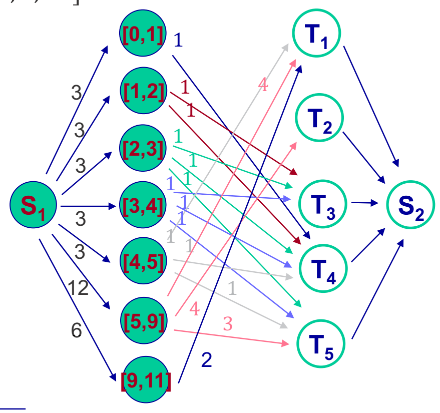
$$p = [7, 4, 3, 5, 6], r = [2, 3, 1, 0, 2], and\ \tilde{d} = [11, 9, 4, 5, 9]$$

$$r \cup \tilde{d} = [2, 3, 1, 0, 2] \cup [11, 9, 4, 5, 9]$$

$$= [0, 1, 2, 3, 4, 5, 9, 11]$$

$T(0) = \{T_4\},$	$p = [7, 4, 3, 5, 6]$
$T(1) = \{T_3\},$	$p = [7, 4, 3, 4, 6]$
$T(2) = \{T_5, T_1\},$	$p = [7, 4, 2, 3, 6]$
$T(3) = \{T_1, T_2\},$	$p = [7, 4, 1, 2, 5]$
$T(4) = \{T_1, T_2\},$	$p = [7, 4, 0, 1, 4]$
$T(5) = \{T_2\},$	$p = [6, 4, 0, 0, 3]$
$T(8) = \emptyset,$	$p = [3, 1, 0, 0, 0]$
$T(9) = \emptyset,$	$p = [2, 0, 0, 0, 0]$
$T(11) = \emptyset,$	$p = [0, 0, 0, 0, 0]$





Topic 3
Scheduling on Parallel Processors

Minimizing Maximum Lateness

Minimizing Maximum Lateness

$$P \mid \quad \mid L_{max}$$

$\boldsymbol{m > 1}$ identical processors: NP-hard $Cmax$–problems are also NP-hard under the L_{max} criterion

for example: P | | L_{max} is NP-hard

m = 1 processor: Earliest Due Date algorithm (EDD) of Jackson [Jac55]

- tasks are scheduled in order of non-decreasing due dates

The EDD rule also minimizes maximum lateness and maximum tardiness

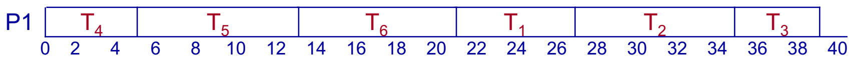
Unit processing times of tasks make the problem easy, and $P \mid pj$= 1, rj | L_{max} can be solved by an obvious application of the *EDD* rule.

Moreover, problem $P \mid p_j = p, r_j \mid L_{max}$ can be solved in polynomial time by an extension of the single processor algorithm.

EDD algorithm

Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \ldots, T_6\}$ with (due date, total execution time):

$$T_1 = (20, 6), T_2 = (25, 8), T_3 = (30, 4),$$
$$T_4 = (9, 5), T_5 = (14, 8), T_6 = (16, 8)$$

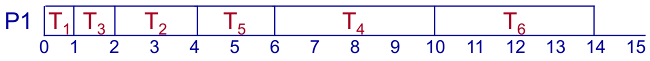


$$lateness\ L_j = c_j - d_j$$
$$L_1 = 27 - 20 = 7$$
$$L_2 = 35 - 25 = 10$$
$$L_3 = 39 - 30 = 9$$
$$L_4 = 5 - 9 = -4$$
$$L_5 = 13 - 14 = -1$$
$$L_6 = 21 - 16 = 5$$
$$L_{max} = \max\{7,10,9,-4,-1,5\} = 10$$

EDD algorithm

Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \ldots, T_6\}$ with (due date, total execution time):

$$T_1 = (2\ 1), T_2 = (5, 2), T_3 = (4, 1),$$
$$T_4 = (8, 4), T_5 = (6, 2), T_6 = (9, 4)$$



$$lateness\ L_j = c_j - d_j$$
$$L_1 = 1 - 2 = -1$$
$$L_2 = 4 - 5 = -1$$
$$L_3 = 2 - 4 = -2$$
$$L_4 = 10 - 8 = 2$$
$$L_5 = 6 - 6 = 0$$
$$L_6 = 14 - 9 = 5$$
$$L_{max} = \max\{-1,-1,-2,2,0,5\} = 5$$

Jackson's EDD rule

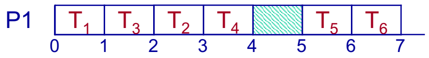
Assumption of unit execution times 1 | $r_j, p_j = 1$| L_{max} , r_j an integer
a modification of Jackson's EDD rule is optimal

Algorithm Modification of EDD rule for 1 | $r_j, p_j = 1$| L_{max} , r_j an integer.

```
begin
  r:=0;
  while $\mathrm{T} \neq \emptyset$ do
  begin
    $t := \max\left\{t, \min_{T_j \in \mathrm{T}}\{r_j\}\right\}$;
    $T' := \{T_j | T_j \in r_j \leq t\}$;
    Choose $T_i \in \{T_j | T_j \in T'\, for\ which\ d_j = \min\{d_k | T_k \in T'\}\}$;
    $\mathrm{T} = \mathrm{T} - \{T_i\}$;
    Schedule $T_i$ at time t;
    $t := t + 1$;
  end;
end;
```

Jackson's EDD rule

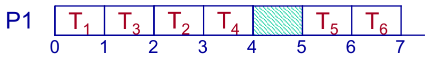
Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with
(due date, release time):
$T_1 = (2, 0), T_2 = (5, 0), T_3 = (4, 1),$
$T_4 = (8, 2), T_5 = (6, 5), T_6 = (9, 5)$



$\max\left\{t, \min_{T_j \in \mathrm{T}}\{r_j\}\right\}$	$T' := \{T_j \| T_j \in r_j \leq t\}$	Choose $\min\{d_k \| T_k \in T'\}$
$t = \max\{0,0\} = 0$	$T' = \{T_1, T_2\}$	T_1
$t = \max\{1,0\} = 1$	$T' = \{T_3, T_2\}$	T_3
$t = \max\{2,1\} = 2$	$T' = \{T_2, T_4\}$	T_2
$t = \max\{3,2\} = 3$	$T' = \{T_4\}$	T_4
$t = \max\{4,5\} = 5$	$T' = \{T_5, T_6\}$	T_5
$t = \max\{6,5\} = 6$	$T' = \{T_6\}$	T_6

Jackson's EDD rule

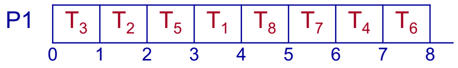
Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with
(due date, release time):
$T_1 = (2, 0), T_2 = (5, 0), T_3 = (4, 1),$
$T_4 = (8, 2), T_5 = (6, 5), T_6 = (9, 5)$



$$lateness\ L_j = c_j - d_j$$
$$L_1 = 1 - 2 = -1,$$
$$L_2 = 3 - 5 = -2,$$
$$L_3 = 2 - 4 = -2,$$
$$L_4 = 4 - 8 = -4,$$
$$L_5 = 6 - 6 = 0,$$
$$L_6 = 7 - 9 = -2,$$
$$L_{max} = \max\{-1, -2, -2, -4, 0, -2\} = 0$$

Jackson's EDD rule

Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with
(due date, release time):
$T_1 = (6, 0), T_2 = (5, 1), T_3 = (4, 0), T_4 = (8, 1)$
$T_5 = (3, 2), T_6 = (9, 4), T_7 = (7, 3), T_8 = (6, 4)$

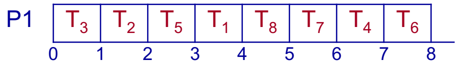


$\max\left\{t, \min_{T_j \in \mathrm{T}}\{r_j\}\right\}$	$T' := \{T_j \| T_j \in r_j \leq t\}$	Choose $\min\{d_k \| T_k \in T'\}$
$t = \max\{0, 0\} = 0$	$T' = \{T_3, T_1\}$	T_3
$t = \max\{1, 0\} = 1$	$T' = \{T_2, T_1, T_4\}$	T_2
$t = \max\{2, 0\} = 2$	$T' = \{T_5, T_1, T_4\}$	T_5
$t = \max\{3, 0\} = 3$	$T' = \{T_1, T_7, T_4\}$	T_1
$t = \max\{4, 1\} = 4$	$T' = \{T_8, T_7, T_4, T_6\}$	T_8
$t = \max\{5, 1\} = 5$	$T' = \{T_7, T_4, T_6\}$	T_7
$t = \max\{6, 1\} = 6$	$T' = \{T_4, T_6\}$	T_4
$t = \max\{7, 4\} = 7$	$T' = \{T_6\}$	T_6

Jackson's EDD rule

Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with (due date, release time):

$$T_1 = (6, 0), T_2 = (5, 1), T_3 = (4, 0), T_4 = (8, 1)$$
$$T_5 = (3, 2), T_6 = (9, 4), T_7 = (7, 3), T_8 = (6, 4)$$



$$lateness\ L_j = c_j - d_j$$
$$L_1 = 4 - 6 = -2,$$
$$L_2 = 2 - 5 = -3,$$
$$L_3 = 1 - 4 = -3,$$
$$L_4 = 7 - 8 = -1,$$
$$L_5 = 3 - 3 = 0,$$
$$L_6 = 8 - 9 = -1,$$
$$L_7 = 6 - 7 = -1,$$
$$L_7 = 5 - 6 = -1,$$
$$L_{max} = \max\{-2, -3, -3, -1, 0, -1, -1, -1\} = 0$$

Jackson's EDD rule

The preemptive mode of processing makes the problem much easier.

Single processor problem 1 | $pmtn$, r_j | L_{max}:

A modification of Jackson's rule due to Horn (1974) solves the problem optimally in polynomial time

Jackson's rule due to Horn (1974)

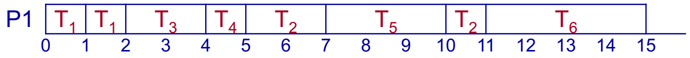
Algorithm for 1 | $pmtn$, r_j| L_{max} , (Horn, 1974).

```
Begin
 repeat
   $\rho_1 := min\{r_j | r_j \in T\}$;
   if all tasks are available at time $\rho_1$
     then $\rho_2 := \infty$
     else $\rho_2 := \min\{r_j | r_j \neq \rho_1\}$;
   $E := \{T_j \mid r_j = \rho_1\}$;
   Choose $T_k \in E$ such that $d_k = min\{d_j | T_j \in E\}$
   $l := \min\{ p_k, \rho_2 - \rho_1 \}$;
   Assign $T_k$ to the interval $[\rho_1, \rho_1 + l)$;
   if $p_k \leq l$
     then T := T – $\{T_k\}$
     else $p_k := p_k - l$;
   for all $T_j \in E$ do $r_j := \rho_1 + l$;
 until T = ∅;
end;
```

Jackson's rule due to Horn (1974)

Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with (due date, total execution time, release time):

$$T_1 = (2, 2, 0), T_2 = (12, 3, 0), T_3 = (4, 2, 1),$$
$$T_4 = (7, 1, 2), T_5 = (11, 3, 7), T_6 = (15, 4, 4)$$



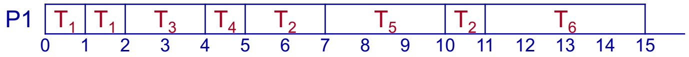
$\rho_1 = min\{r_j | r_j \in T\}$, $\rho_2 = \min\{r_j | r_j \neq \rho_1\}$, $E := \{T_j \mid r_j = \rho_1\}$, $d_k = min\{d_j | T_j \in E\}$, $l = \min\{ p_k, \rho_2 - \rho_1 \}$

ρ_1	ρ_2	E	d_k	l	$p_k = p_k - l$	$r_j = \rho_1 + l$
0	1	T_1, T_2	$\{2, 12\}$	$\{2, (1 - 0)\}$	$p_1 = 1$	$r_1 = r_2 = 1$
1	2	T_1, T_3, T_2	$\{2, 4, 12\}$	$\{1, (2 - 1)\}$	$p_1 = 0$	$r_2 = r_3 = 2$
2	4	T_3, T_4, T_2	$\{4, 7, 12\}$	$\{2, (4 - 2)\}$	$p_3 = 0$	$r_2 = r_4 = 4$
4	7	T_4, T_2, T_6	$\{7, 12, 15\}$	$\{1, (7 - 4)\}$	$p_4 = 0$	$r_2 = r_6 = 5$
5	7	T_2, T_6	$\{12, 15\}$	$\{3, (7 - 5)\}$	$p_2 = 1$	$r_2 = r_6 = 7$
7	∞	T_5, T_2, T_6	$\{11, 12, 15\}$	$\{3, (\infty - 7)\}$	$p_5 = 0$	$r_2 = r_6 = 10$
10	∞	T_2, T_6	$\{12, 15\}$	$\{1, (\infty - 10)\}$	$p_2 = 0$	$r_6 = 11$
11	∞	T_6	$\{15\}$	$\{4, (\infty - 11)\}$	$p_6 = 0$	

Jackson's rule due to Horn (1974)

Example: Set of independent tasks: $\mathrm{T} = \{T_1, T_2, \dots, T_6\}$ with (due date, total execution time, release time):

$$T_1 = (2, 2, 0), T_2 = (12, 3, 0), T_3 = (4, 2, 1),$$
$$T_4 = (7, 1, 2), T_5 = (11, 3, 7), T_6 = (15, 4, 4)$$



$$lateness\ L_j = c_j - d_j$$
$$L_1 = 2 - 2 = 0,$$
$$L_2 = 11 - 12 = -1,$$
$$L_3 = 4 - 4 = 0,$$
$$L_4 = 5 - 7 = -2,$$
$$L_5 = 10 - 11 = -1,$$
$$L_6 = 15 - 15 = 0,$$
$$L_{max} = \max\{0, -1, 0, -2, -1, 0\} = 0$$

Topic 4
Communication Delays and Multiprocessor Tasks

Scheduling Divisible Tasks

Scheduling Multiprocessor Tasks

P, cube | div, $cube_j$ | C_{max}

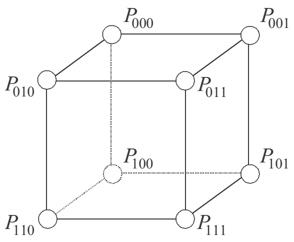
For the hypercube of dimension d there are 2^d processors in the system.

Each of the processors has direct links to d neighbors.

The label of a processor is a binary number from the interval $[0, 2^d]$.

- each of the processor's neighbors has a label differing on exactly one position.

For $d = 3$



Scheduling Multiprocessor Tasks

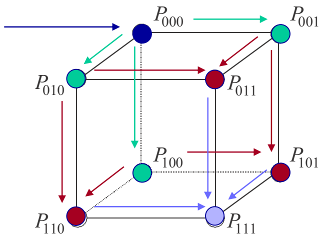
P, cube | div, $cube_j$ | C_{max}

At time 0 a task arrives to processor P_0.

Some part α_0 of the total load is processed by processor P_0 the rest of the load $(1 - \alpha_0)$ is transmitted in equal parts to its d neighbours for processing.

Immediate neighbors of processor P_0 take some part α_1 of the total load and retransmit the rest to the still idle neighbors.

This process is continued until the last idle processor in the hypercube is reached.

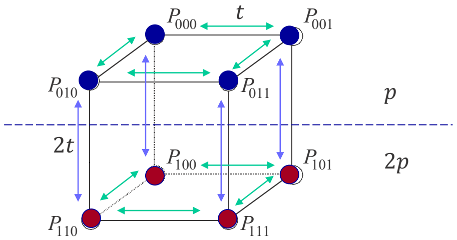


Scheduling Multiprocessor Tasks

P, $cube$ | div, $cube_j$| C_{max}

We assume that

- processing time of the task on a standard processor is p, while on processor with a different speed it is wp.
- w is proportional to the reciprocal of the processor's speed.
- transmission time of the whole task's data is t for a standard data link, while for a link with different capabilities it is zt.
- z is the reciprocal of the link bandwidth.



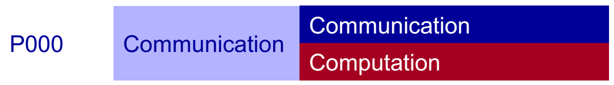
Scheduling Multiprocessor Tasks

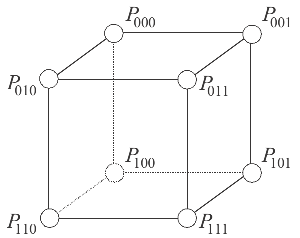
P, $cube$ | div, $cube_j$| C_{max}

We assume two things about the processing element:

it must receive all its load before transmitting the proper part to the neighbors

it is capable of simultaneous transmitting and computing.





Scheduling Multiprocessor Tasks

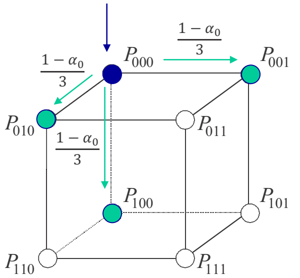
P, $cube$ | div, $cube_j$| C_{max}

When processor P_0 receives data to process, it takes

- α_0 of it for local processing;
- $(1-\alpha_0)$ of the load is transmitted to *d* neighbors.

Since processor P_0 has no 1 in its address, its neighbors have exactly one 1 in their addresses.

The part $(1-\alpha_0)$ transmitted from processor P_0 is fairly divided among all *d* neighbors.



Scheduling Multiprocessor Tasks

P, $cube$ | div, $cube_j$| C_{max}

Then, each of the processors with only one 1 in the address takes α_1 of the whole load for local processing from the part it receives from processor P_0.

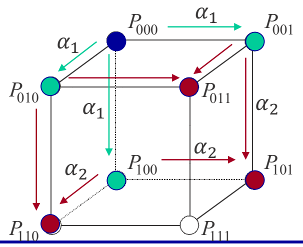
The rest is transmitted, in equal shares, to its $d-1$ idle neighbors.

Processors with one 1 in the address have $d-1$ idle neighbors with exactly two 1's in the address.

Note, that processors with one 1 in the address can be reached from the originator of the load via only one link, while processors with two 1's can be accessed via two links.

$$\alpha_1 = \frac{1-\alpha_0}{3}$$

$$\alpha_2 = \frac{1-\alpha_1}{2}$$

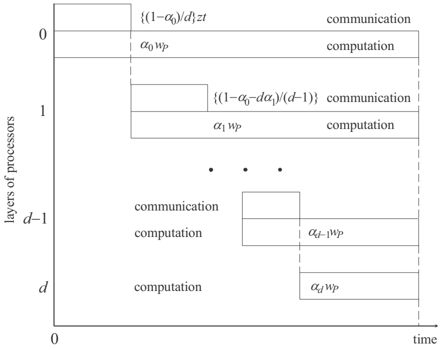


Scheduling Multiprocessor Tasks

P, *cube* | *div*, $cube_j$| C_{max}

layer i a set of all processors reached in the same number *i* of processors,

- starting from layer 0 (processor P_0).
- The last layer *d* consists of a single processor



Scheduling Multiprocessor Tasks

P, *cube* | *div*, $cube_j$| C_{max}

The *speedup S,* measured as a ratio of the sequential computation time, i.e. on the sole originator, to the working time of the originator embedded in the hypercube,

$$S = \frac{dwp}{w_1 p + zt}$$

The *average processor utilization* of *(U)* can be found:

$$U = \frac{1}{2^d \alpha_0}$$

Where w_1 is calculated according to a recursive procedure

Using the above formulae one can analyze a performance of the hypercube depending on such parameters

dimension of the hypercube (d), reciprocal of the communication speed (z), reciprocal of the processing speed (w) and size of the computing task (p)

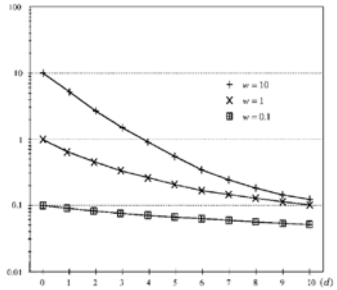
Scheduling Multiprocessor Tasks

P, *cube* | *div*, $cube_j$| C_{max}

The execution time of the task decreases with the dimension of the hypercube.

- for faster processors (w = 0.1) the reduction is relatively smaller than for slow processors (w = 10)

The gain from parallel processing on slow processors is higher than on fast processors.



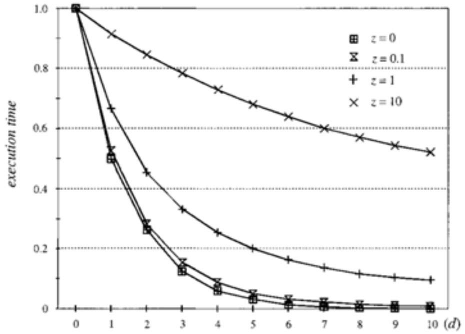
Execution time vs. processor speed and dimension

Scheduling Multiprocessor Tasks

P, *cube* | *div*, $cube_j$| C_{max}

Fast communication network (z = 0.1) is more reasonable than the slow one (z = 10).

- z = 0 is a case of the ideal network (without transportation delays).



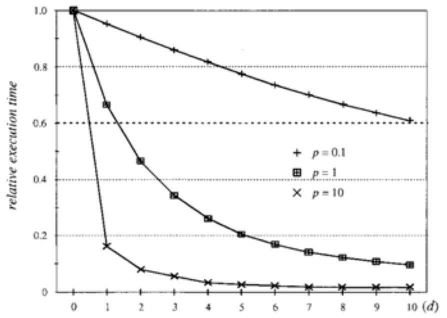
Execution time vs. communication speed and dimension

Scheduling Multiprocessor Tasks

$P, cube \mid div, cube_j \mid C_{max}$

Relative processing time as a function of p and d .The relative execution time is equal to the quotient of the actual processing time and processing time on the processor of the same speed

- The gain from parallel processing is bigger for long tasks (p = 10).



Execution time vs. size of the computing taskp and dimension

CICESE

Algoritmos y métodos de Calendarización

Optimization in Cluster, Grid and Cloud computing

4.4 Bin Packing

Dr. Andrei Tchernykh

Introduction

Metaphorically, there never seem to be enough bins for all one needs to store

Mathematics comes to the rescue with the *bin packing problem* and its relatives

The bin packing problem raises the following question:

Given a finite collection of n *weights* $w_1, w_2, \ldots w_n$ and a collection of *identical bins* with capacity C (which exceeds the largest of the weights)

What is the *minimum number* k of bins into which the weights can be placed without exceeding the bin capacity C?

Introduction

We want to know how few bins are needed to store a collection of items

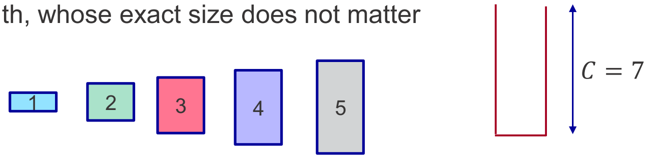
This problem, known as the *1-dimensional bin packing problem*, is one of many mathematical packing problems which are of both theoretical and applied interest

Keep in mind that *weights* are indivisible objects rather than something like oil or water

- To have part of a weight in one bin and part in another is not allowed

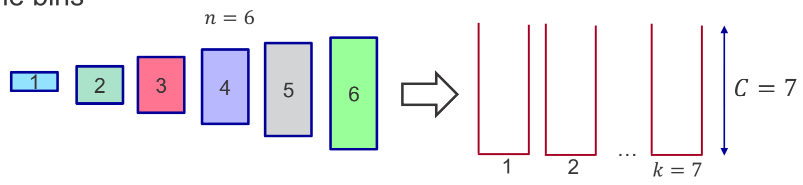
One way to visualize the situation is as a collection of rectangles which have

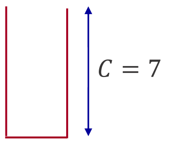
- Height minor or equal to the capacity C
- Fixed width, whose exact size does not matter



Introduction

When an item is put into the bin, it either falls to the bottom or is stopped at a height determined by the weights that are already in the bins





The diagram shows a bin of capacity 7 where three weights have been placed in the bin, leaving 1 unit of empty space

Basic ideas

The bin packing problem asks for the minimum number k of identical bins of capacity C needed to store a finite collection of weights $w_1, w_2, \dots w_n$ so that no bin has weights stored in it whose sum exceeds the bin's capacity

Traditionally,

- Bin capacity C is chosen to be 1
- Weights are real numbers which lie between 0 and 1

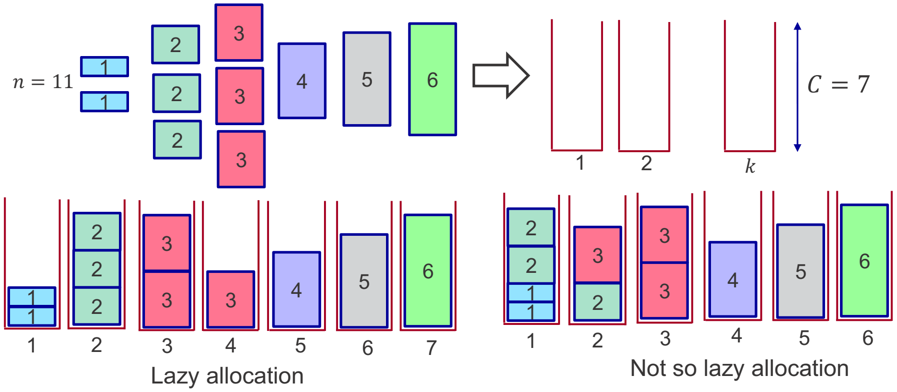
For convenience of exposition,

- C is a positive integer
- $w_1, w_2, \dots, w_n$ are positive integers which are less than the capacity

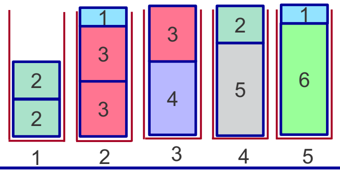
Example 1: Suppose we have bins of size 7

How few of them are required to store weights of size 1, 1, 2, 2, 2, 3, 3, 3, 4, 5, 6?

Basic ideas



My best allocation



Worst case?

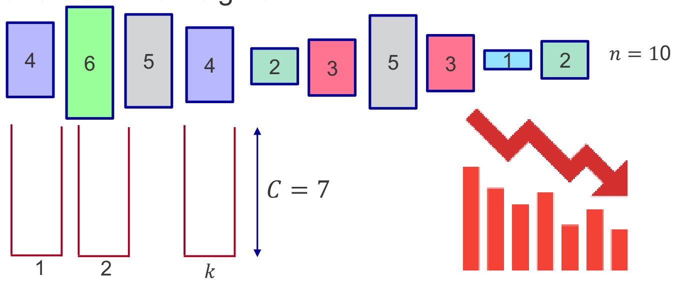
Best case?

Basic ideas

The weights to be packed above have been presented in the form of a *list* L ordered from left to right

The *seek procedures* (algorithms) for packing the bins are "driven" by a *list* L and a *capacity* C for the bins

The goal of the procedures is to *minimize the number of bins* needed to store the weights



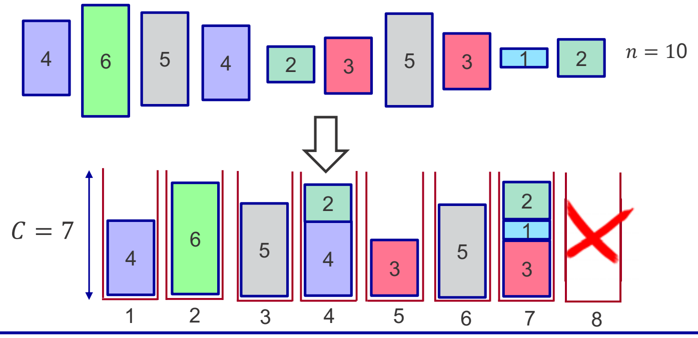
A variety of simple ideas as to how to pack the bins suggest themselves

Next Fit

Next Fit (***NF***) is one of the simplest approaches

The idea is to open a bin and place the items into it in the order that they appear in the list

If the current item on the list does not fit into the open bin, this bin is closed permanently, a new one is open, and the packing process continues with the remaining items in the list



Next Fit

Advantages of ***NF***:

- Very simple
- Bins are shipped off quickly, ***NF*** does not hope that an item will come along later in the list which will fill this empty space

One can imagine a *fleet of trucks* with a weight restriction (the capacity C) and one packs weights into the trucks

- If the next weight can not be packed into the truck at the loading dock, this truck leaves and a new truck pulls into the dock
- We keep track of how much room remains in the bin open at that moment

Next Fit

NF requires a linear amount of time to find the number of bins for n weights $O(n)$

NF does not always produce an optimal packing for a given set of weights

- We will find a way to pack the weights of Example 1 into 5 bins

Procedures such as ***NF*** are referred to as *heuristic algorithms* because although they were conceived as ways to solve a problem optimally, they do not always deliver an optimal solution

Can we find a way to improve *NF*?

A strategy will typically use fewer bins if it keeps bins open for filling their empty space with items later in list L

First Fit

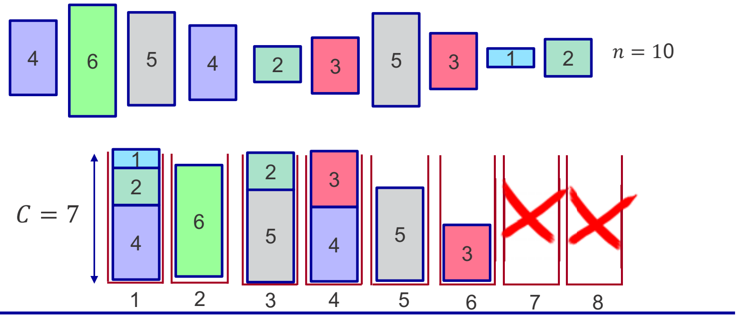
First Fit (***FF***) is a simple way to carry out this idea

FF places the current item in the list into the first bin which has not been completely filled into which it will fit

- The list is numbered from left to right

Bins are closed when they are completely filled

A new bin is opened when an item does not fit into any currently open bin



First Fit

Both methods yield 6 and 7 bins, but

is this the best we can hope for ?

One simple insight is computing the total *sum of the weights* and dividing this number by the *capacity of the bins*

The number of bins must be at least $\lceil \Omega/C \rceil$ where

- $\Omega = \sum_{i=1}^{n} w_i$
- $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to x

The number of bins must always be an integer

Since Ω=35 and C=7 in example 1, there is a hope of using 5 bins

However, neither ***NF*** nor ***FF*** achieves this value with the list given in Example 1

Perhaps we need a better procedure

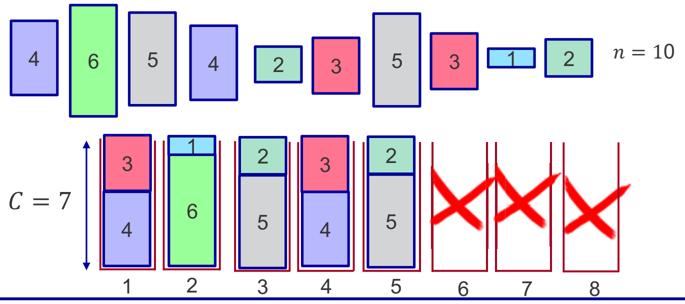
Two other simple methods in the spirit of ***NF*** and ***FF*** have also been looked at

Best Fit

Best Fit (***BF***) keeps bins open in the hope that a later smaller item will fit

The criterion for placement is that ***BF*** puts the next item into the currently open bin (e.g. not yet full) which leaves the least room left over

- In the case of a tie, ***BF*** puts the item in the lowest numbered bin as labeled from left to right

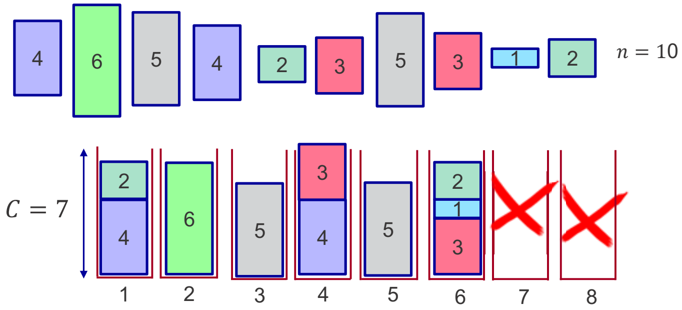


Worst Fit

Worst Fit (***WF***) also keeps bins open in the hope that a later smaller item will fit

The criterion for placement is that ***WF*** places the item into that currently open bin into which it will fit with the most room left over

- In the case of a tie, ***WF*** puts the item in the lowest numbered bin as labeled from left to right



Best fit and Worst Fit

The amount of time necessary to find the minimum number of bins using either ***FF***, ***WF***, or ***BF*** is higher than for ***NF***.

- $O(\boldsymbol{nlogn})$ in terms of the number n of weights.

The distinction between ***FF***, ***WF***, and ***WF***:

- Suppose that there are only 3 bins open with capacity 10
- The remaining space is: Bin 1 (4), bin 2 (7), and bin 3 (3)
- Suppose the next item in the list has size 2

Allocation of the strategies:

- FF puts this item in bin 1
- BF puts it in bin 3
- WF puts it in Bin 2

One difficulty is that we are applying "good procedures" but on a "lousy" list

If we know all the weights to be packed in advance, is there a way of constructing a good list?

More approaches to bin packing

Previous algorithms have the property that there may be more items in the list to pack but they do not affect what is done to pack the current item

- farther to the right than one being currently worked on

Such algorithms are known as on-line, the idea for on-line algorithms is that not all the components of the problem are known in advance

- they are being used to solve bin packing problems or other combinatorial optimization problems

In the bin packing case, one can imagine an industrial situation where items with different weights are being produced and then are being placed in bins which are at some stage to be shipped to customers

More approaches to bin packing

In contrast to an on-line point of view, an off-line approach is a possibility

- one thinks of having all of the items to be packed in advance
- one can ask for the given weights to be packed if there is some rearrangement of the weights into a list different from the original which might be used to give a better result for the number of bins required

The number of potential lists for n items to pack is $n!$

- Since the first item can be chosen in n ways, the second in $n -$ 1, etc., giving $n!$ as the number of different possible lists.

Choosing a list for an optimal packing has the flavor of looking for a needle in a haystack

- Even for 20 items to pack, say, 20! is a very large number

More approaches to bin packing

For example, many bins can be open at a specific stage when ***FF*** is used in an on-line environment

- To have so many empty bins is economically unrealistic
- Monitoring that later items will fit efficiently

This situation suggests a version of bin packing where a packing heuristic is limited to having at most K bins open at a given time

- These heuristics are known as bounded-space on-line heuristics

For the heuristics discussed above one can try to develop a K open bin bounded space version of the heuristic

The situation for K > 1 open bins raises some tantalizing issues:

- The way the bins are packed
- When a bin will be shut down (closed permanently) other than when it is completely full!

More approaches to bin packing

Suppose K open bins and a weight which does not fit into any of them

- Open a new bin implies closing an old one

This can be done either by

- picking the lowest numbered bin to shut down (this is in the spirit of ***FF***)
- shutting down the bin which is closest to being completely full (this is in the spirit of ***BF***).

Four new heuristics in the K open bin environment use

- ***FF*** or ***BF*** as packing rule and
- ***FF*** or ***BF*** as closing rule

LIMITED

More approaches to bin packing

Very large item to pack at the end of the list made it necessary to have an extra bin opened at the end

- Packing large items first seems like a good idea

The same strategy you would use to pack a suitcase for a vacation

- Not to leave a large-volumed item to the end!

This suggests an approach to off-line bin packing

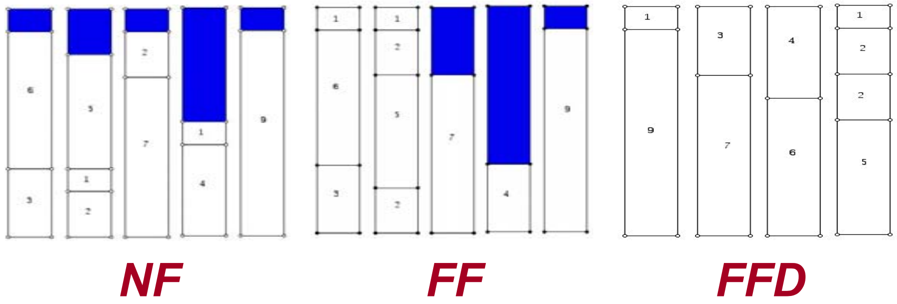
- Choose the list where the weights appeared in sorted order, from largest to smallest
- Use any algorithms previously discussed to carry out the packing

NFD, ***FFD***, ***BFD***, and ***WFD*** are the new approaches where "D" refers to “decreasing” for algorithms ***NF***, ***FF***, ***BF***, and ***WF***

- ***FFD*** means First Fit Decreasing

More approaches to bin packing

Example 2: bin size = 10 and weights = {3, 6, 2, 1, 5, 7, 2, 4, 1, 9}



Note that ***FFD*** packing is optimal

Not only can one not pack these weights into fewer bins but also there is no wasted space

Of course, there are many situations where optimal packings still have unused space

More approaches to bin packing

Among all packings of this kind, one might look for that packing with various characteristics

- The fewest bins have extra space
- The number of bins used is minimal but as many bins as possible have some extra room
  - A little extra space might allow putting packing material into the bin to prevent breakage during shipment

Rearrange the items in one of the bins or between bins in an optimal packing that achieves some secondary goal

- Beyond minimizing the number of bins

FFD found an optimal solution in Example 2, but it does not mean that this will be the case for other problems

Perhaps you are not surprised to learn that ***FFD*** does not always yield optimal solutions

More approaches to bin packing

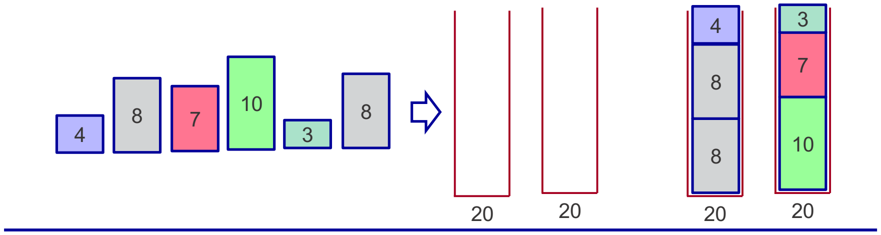
Example 3: bin size = 20 and weights = {4, 8, 7, 10, 3, 8}

2 bins might be achieved since the sum of the weights is 40 and bin capacity is 20

A packing with only two bins is: Bin_1 (8, 8, 4), and Bin_2 (10, 7, 3)

In this notation, the items in Bin_1 are:

- 8 is at the bottom, next comes the weight 8, and the item of size 4 stays at the top of the bin
- this solution is not unique, items can be permuted



More approaches to bin packing

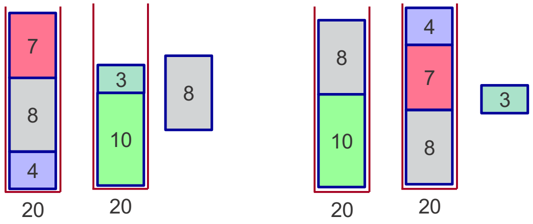
Example 3: bin size = 20 and weights = {4, 8, 7, 10, 3, 8}

None of the procedures (***NF***, ***FF***, ***BF***, ***WF***, ***NFD***, ***FFD***, ***BFD***, or ***WFD***) yields an optimal number of bins with this particular list

- 3 bins are required in each case (you can check)

NF, ***FF***, ***BF***, and ***WF*** provide an optimal packing with the list 8, 4, 8, 10, 7, 3

- The weights are packed slightly differently from the solution given above



More approaches to bin packing

After finding eight heuristics or approximation algorithms in the search to find an optimal solution to the bin packing problem

finding an optimal solution to bin packing is very hard?

Mathematicians have attempted to show that some problems are indeed very hard using a variety of approaches

- One approach involves showing that a problem is NP-complete

Intuitively, a problem Z is NP-complete when it has been shown that

- if Z can be solved in polynomial time, then so can a very large number of other problems
- if Z requires an exponential amount of time to solve, then so will all of the other problems

NP-complete problems are thought to be hard to solve but either all of the problems are

- Not really that hard (in the sense of requiring polynomial work)

More approaches to bin packing

- All of the problems require an exponential amount of work to solve

Since many NP-complete problems are of great importance for applications in operations research (management science), the fact that mathematicians and computer scientists are unsure of the status of the NP-complete problems is frustrating!

You guessed it--the "decision version" of bin packing is known to be NP-complete

That is, given a capacity C and a list L of weights and an integer D the problem of determining if the weights in L can be packed into D or fewer bins of capacity C is NP-complete

Thus, finding approximately optimal solutions for bin packing in polynomial time is probably the best we can hope for

Applications of bin packing

Packing trucks with weight capacity C is one of many applications of bin packing

- Realistic versions of truck packing problems typically go beyond issues of one-dimensional bin packing

Bin packing has an important connection with another important collection of operations research problems

- Often referred to as machine scheduling problems

Consider the problem of scheduling identical machines with tasks that are independent of each other

- The tasks can be done by the machines in any order
- Each of the tasks has a time that is necessary to complete it on one of the machines

More complex machine scheduling problems have to deal with tasks that cannot be worked on before other tasks are completed

Applications of bin packing

What is the minimum number of machines which are needed to finish a collection of independent tasks by time T with times to complete the tasks of $t_1, t_2, \ldots, t_n$?

This question is bin packing with a very minimal disguise (change *t* to *w*; T to C)!

For example, suppose one has photocopying jobs of varying numbers of pages that have been brought into a photocopy shop

If the shopkeeper wants the automatic work of the machines to enable her to go home within 3 hours of the start of the photocopy tasks

- how many machines would have to be available to do the work?
- how should the jobs be assigned to the machines?

Bin Packing and Machine Scheduling

Packing ads into breaks is an example of a bin packing problem packing a collection of ads of different lengths into the minimum number of bins of fixed size (the length of the ad break)

Scheduling problems are present in the operation of all large systems:

- scheduling classes in schools;
- scheduling planes, trains, and buses in the transportation sector of the economy;
- scheduling machines in the manufacture of the products

Bin Packing and Machine Scheduling

The one-dimensional bin packing problem belongs to the complexity class of problems for which no known algorithm solves the problems in a polynomial time

- Polynomial in the number of weights

How badly are various heuristic algorithms which might be used to solve bin packing problem

A heuristic algorithm not always guarantees to find the optimal solution to a problem

- It finds an "approximate" solution to a problem

We can now ask the question: Given a collection of weights and bins of capacity C

how many bins--compared with the optimum number actually required--do these approximation algorithms require?

Bin Packing and Machine Scheduling

Suppose we are given weights, a capacity, a list L, and an algorithm A

Denote by

- $OPT(L)$ the optimum number of bins,
- $A(L)$ the number of bins which are required by algorithm A applied to list L

How do $OPT(L)$ and $A(L)$ compare in size?

To answer questions such as this, one can apply mathematical arguments to show that:

$$A(L) \leq r\,OPT(L)$$

where r is some positive constant

One can then try to show by way of examples that the r in the equation above can actually occur

Bin Packing and Machine Scheduling

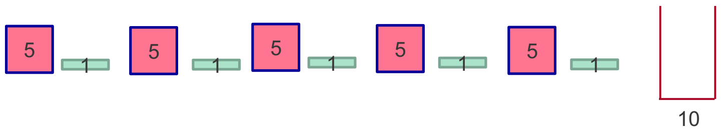
Many researchers in mathematics and computer science have contributed to insights in this vein for the literally dozens of different algorithms that have been investigated to understand one-dimensional bin packing

Suppose we are dealing with the heuristic ***NF***. It is not difficult to see that ***NF*** obeys:

$$NF(L) \leq 2OPT(L)$$

With a bin capacity of 1, two consecutive bins that are packed by ***NF*** can not both be filled to less than the halfway mark

The reason is that the contents of two such bins would then fit in only one of the bins, because of the way ***NF*** fills the bins



Bin Packing and Machine Scheduling

The proof of the inequality above uses this fact

For a further insight, consider the following list of weights for the case where the bins have capacity 1:

$$L = (\frac{1}{2}, \frac{1}{2N}, \frac{1}{2}, \frac{1}{2N}, \dots, \frac{1}{2})$$

where N is a positive integer which is 2 or bigger

NF applied to this list will produce a packing which has $2N$ bins,

- Each bin packed with an item of weight $1/2$ at the bottom and an item of weight $1/2N$ on top

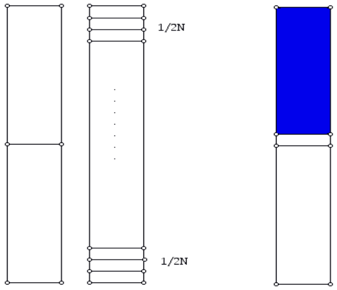
The optimal packing for this list uses $N + 1$ bins:

- N bins, each with two weights of size $1/2$, and
- One additional bin to contain the N items of size $1/2N$

Bin Packing and Machine Scheduling

Below are some diagrams to help visualize the packing which is optimal versus what happens when ***NF*** is used

- The optimal packing has N copies of the one bin pictured on the far left and one copy of the next bin



- The ***NF*** packing has $2N$ copies of the bin on the far right, the blue area denotes unused space

Bin Packing and Machine Scheduling

This analysis shows that we have a family of lists for which

$$NF(L) \geq 2OPT(L) - 1$$

Over a period of about 30 years, researchers have

- examined a wide variety of bin packing algorithms and
- obtained increasingly better worst case analyses of the different algorithms.

For example, ***FF*** does significantly better than ***NF*** but not spectacularly so:

$$FF(L) \leq \left\lceil \left(\frac{17}{10}\right) OPT(L) \right\rceil$$

where $\lceil y \rceil$, the ceiling function, denotes the smallest integer greater than or equal to *y*.

Intuitively, one would expect the "decreasing" heuristics to work better and this turns out to be the case

Bin Packing and Machine Scheduling

For ***FFD*** we have the result:

$$FFD(L) \leq \left(\frac{11}{9}\right) OPT(L) + 4$$

In addition to analysing the worst-case performance of bin packing algorithms, one can also see how different heuristics perform on the average

Initially, doing simulations sometimes only hints at results. However, insights gleaned from these simulations often lead to probability models in which precise results can be obtained

Applications of bin packing

Many applications of bin packing, for example,

- Placing computer files with specified sizes into memory blocks of fixed size
- Recording music, where the length of the pieces to be recorded are the weights and the bin capacity is the amount of time that can be stored on an audio CD, about 80 minutes

The abstraction of packing bins with weights allows to apply this mathematical techniques in many situations

- ***FF*** can be used in these different situations

We can find problems where bin packing and the algorithms which have been developed to get insights into this problem can be put to use

Applying mathematics for one context to other contexts which have "***additional aspects***" encourages one to improve on the mathematics that has been done already

Applications of bin packing

The problem of preparing a collection of musical pieces to store on audio compact discs

- An audio CD has a maximal capacity
- The pieces on the compact discs play the roles of the weights
- A piece cannot start on one CD and finish on another

There is a subtlety here, classical music pieces often come in sections called movements

- Ideally, whole pieces should be placed on the same CD,
- Having the first three movements of a symphony on one CD and the last movement at the beginning of the next CD might be "acceptable“, it keeps costs down

The heuristics can not guarantee fitting the music onto the CD would respect the order of the movements if weights are treated as movements

This constraint inspires an algorithms where some of the weights need to be packed in a particular order

Applications of bin packing

Inspired by various "packing" situations, a variety of "*extensions*" for the original bin packing model are:

a) Packings in which the number of items that can be placed in a bin is restricted in advance not to exceed a certain number

- This restriction might be required in the situation where one is packing trucks

b) Restrictions on items which can be placed into the same bin

- This restriction might come about if items are being shipped from A to B in bins
  - ○ In order to guarantee that if one bin gets lost or destroyed in transit to B, one could send redundant items and require that they not be packed in the same bin
- An alternate scenario is when items may be generating heat, perhaps because they have some level of radioactivity
  - ○ The items in one bin not only fit in the bin but also the items not generate too much heat during the period they are in the bins

Applications of bin packing

Inspired by various "packing" situations, a variety of "extensions" for the original bin packing model are:

c) Packings in which there is an ordering attached to some of the weights which limits the way those items can be packed

- This restriction is in the sprit of the example discussed in a little detail above where music is being packed into compact discs of the same size

d) Packings in which the items being packed may be allowed to disappear during the packing process

- An item placed in a bin which has not yet been closed may be
  - ○ Allowed to be removed from the bin either because fewer bins will be needed once this item is transferred to another bin
  - ○ Leave the system, sometimes items not only enter the system

Applications of bin packing

A scenario with potential applications for this variant of bin packing

Imagine that when items are packed, a test is started to determine if the item which has been packed has spoiled. This test takes a certain amount of time to complete

- If the packed item is put into a bin which is closed before the testing period is done, the packed bin is just shipped out
- If the bin is still open during the period of packing and the finished test shows the item to have spoiled, then this item can be removed and the space used for items that haven't spoiled

With cleverness one can perhaps find additional unexpected uses here

One interesting variant of the bin packing problem involves a loosening of the requirement that all the ***bins have the same size***

- A standard capacity or size, say 1, with a cost of 1

Applications of bin packing

Imagine that we can ***stretch or enlarge bins*** but we must pay a penalty for doing this

- The penalty or cost takes the form that a bigger bin is used
- The goal is to pack the weights into stretched bins if necessary so that the total cost is a minimum

In the standard one-dimensional bin packing problem, the bins are assumed to have the same size

You may enjoy formulating a variety of problems with

- a limited number of sizes for bins where one wants to pack the weights so that the total size of the bins used is as small as possible
- situations where problems such as these might arise

Applications of bin packing

Another variant of the classical bin packing problem has to do with the common phenomenon of ***recycling bins***

For simplicity, we assume that a collection of bins contain three types of glass: clear, brown, and frosted

- The goal is to sort the different kinds of glass so that after the sorting process one bin has only clear glass, one only brown glass, and one only frosted glass

Another variant specifies the type of glass for the bins beforehand, we did not specify which of the bins is to have which kind of glass

- The goal is to conduct the sorting with as few moves as possible

Bin packing and machine scheduling

The essence of mathematical modeling is controlled simplification

- One takes the situation outside of mathematics, holds onto essential aspects of the problem and "*disregards details*" that are secondary to the situation
- If one is fully successful doing this process, then after the mathematical problem associated with the model is solved, the mathematics can be used to get important insights into the original problem even though significant detail of the original problem was disregarded
- To some extent the creation of the mathematics problem known as bin packing grew out of attempts to get insight into problems outside of mathematics

Bin packing and machine scheduling

The range of scheduling problems makes it virtually impossible for one mathematical model to be useful in all situations

How is bin packing connected with machine scheduling?

Suppose that we have tasks which take differing times to complete which can be performed in any order on identical machines

- Our goal is to determine the minimum number of machines needed to finish the tasks by a fixed time

The fixed desired completion time corresponds to the capacity of the bins, and weights are the times to do the tasks

Finding the minimum number of bins needed to pack the weights provides the minimum number of machines needed to finish by the desired time !

Bin packing and machine scheduling

A sample of the many complexities that machine scheduling problems have

1) A job that must be completed by scheduling the tasks that make up the job often has to be carried out on very different kinds of processors

- The construction of a home requires the completion of a gigantic number of steps
  - One needs a building permit, someone to lay the foundation, and someone to put on the roof
  - The "processors" that must be scheduled for doing this work are not interchangeable. Plumbers rarely do electrical work and electricians rarely do roofing

2) Schedules for the manufacturing of many identical products often require that each product get some time on a fixed number of different machines

Bin packing and machine scheduling

- Several identical machines of one kind
- Manufacturing a computer chip

3) When schedules are constructed, there are different approaches to how the processors work on the tasks assigned

- When a processor begins work on a task then it continues until the work on that task is complete
- A condition in the processor will permit to interrupt the current work on one task to begin another task (an even)

Whereas in manufacturing, once a machine starts work on a task, it usually continues to work until the task is complete

Bin packing and machine scheduling

Things are somewhat different in medicine, when doctors schedule patients, they typically stay with a patient until the task is done

Ophthalmologist can be an exception, he

- Put drops in a patient's eyes,
- See another patient while the first patient's pupils are increasing in size
- Return to complete the first patient's examination

Similarly, a doctor in regular practice who is seeing a patient who has a head cold would probably interrupt her work to see a patient who arrived in the office complaining of chest pains

Bin packing and machine scheduling

These scheduling problems are completely deterministic

- Times for completing every task making up a complex job are known in advance

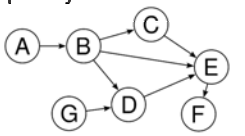
A directed graph called *task-analysis digraph* indicates which tasks come immediately before other tasks

This digraph will have no directed circuit because that would lead to a contradiction in the precedence relations between the tasks

- A way to follow edges and start at a vertex and return there

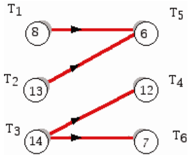
An arrow from task T_i to T_j means that task T_i must be completed before task T_j can begin

Such restrictions are very common in carrying out the individual tasks that make up a complex job



Bin packing and machine scheduling

In the task-analysis digraph below, each vertex (task) has the processing time inside the circle



The objective is to schedule six tasks on some fixed number of identical machines in such a manner that the tasks are completed by the earliest possible time

- Sometimes referred to as finding the makespan for the tasks

Also, defining which tasks should be scheduled on which machines during a given time period

The scheduling is carried out without (voluntarily) idle machines

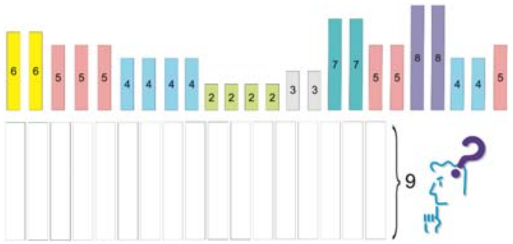
When a machine begins working on a task, it will continue to work on it without interruption until the task is completed

One-Dimensional Bin Packing Problem

One-dimensional bin packing problem can be described as follows:

Given a bin capacity $C > 0$ and a list of objects $\{p_1, p_2, \dots, p_n\}$, what is the smallest number of bins needed to accommodate all of the objects?

Each p_i has size s_i, such that $0 \le s_i \le C$. i.e. none of the objects too big to fit in a bin



One-Dimensional Bin Packing Problem

Four easily recognizable one-dimensional bin packing problems:

- A material such as piping or cable is supplied in quantities of a standard length C
  - ○ The demands for pieces of the material are for varying lengths that do not exceed C
  - ○ The idea is to use the least number of standard lengths of the material in producing a given set of required pieces. Hence minimizing the wastage
- Advertisements of arbitrary lengths must be assigned to commercial break time slots on television
  - ○ Each break must last no longer than three minutes
- Removal lorries with set weight limits are to be packed with furniture
  - ○ The aim is to use as few lorries as possible to pack all the items, without exceeding the maximum weight in any lorry

One-Dimensional Bin Packing Problem

- The execution of a set of tasks with known, arbitrary execution times, on some identical processors by a given deadline
  - ○ Schedule all of the tasks onto the least number of machines so that the deadline is met

In the last example, time is the critical resource. It is a scheduling problem;

- the processors are the bins,
- the deadline is the common bin capacity, and
- the individual execution times of the tasks are the objects (p_1) in the list

Two-Dimensional Bin Packing Problem

This problem is concerned with packing different sized objects (most commonly rectangles) into fixed sized, two-dimensional bins, using as few of the bins as possible

Stock cutting example

- quantities of material such as glass or metal, are produced in standard sized, rectangular sheets
- Demands for pieces of the material are for rectangles of arbitrary sizes
- no bigger than the sheet itself

The problem is to use the minimum number of standard sized sheets in accommodating a given list of required pieces

A variation on the two-dimensional bin packing problem is the strip packing problem

Strip Packing Problem

The problem

- packing a set of n rectangles into an open-ended bin of
  - ○ fixed width C and
  - ○ infinite height
  - ○ the rectangles must not overlap each other

The idea is to pack the rectangles in a way that minimizes the overall height of the bin

The clear difference between this problem and the two-dimensional bin packing problem, is that there is only one bin

- The aim is to minimize the height of the single bin instead of trying to minimize the number of bins used

All the rectangles must be packed orthogonally.

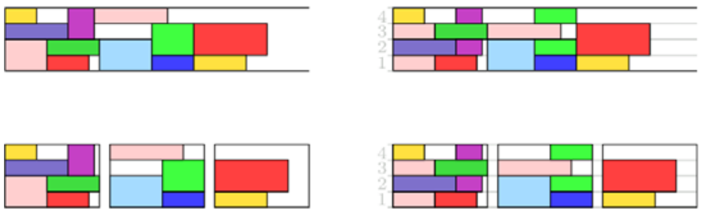
- They cannot be rotated and must be packed with their width parallel to the bottom of the bin

Strip Packing Problem

Strip Packing and machine scheduling

The rectangles correspond to a set of tasks,

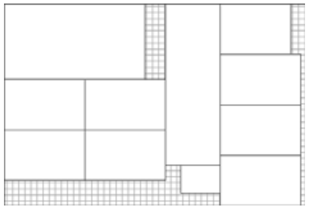
- heights being the amount of processing time they require
- widths the amount of contiguous memory (processors) they need
- The width of the bin corresponds to the amount of memory (procs) available
- The aim is to schedule all of the tasks so that they are completed in the least possible time



Strip Packing Problem

Another common application of the strip packing problem is stock cutting

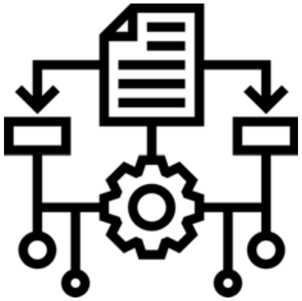
- Material such as cloth, paper, or sheet metal, comes in rolls of a set width
- These rolls may need to be cut into rectangles of arbitrary widths and heights
- The goal is then to cut out all the required rectangles from the shortest length of roll possible, so minimizing the wastage
- the roll of material corresponds to the bin



Strip Packing Algorithms

The problem of finding an optimal solution for the strip packing problem is NP-complete,

- Approximation algorithms find near optimal solutions but do not guarantee to find the optimal packing for every set of data.
- Most algorithms pack the rectangles into the bin using one of five approaches:
- bottom left,
- level-orientated,
- split,
- shelf or
- hybrid



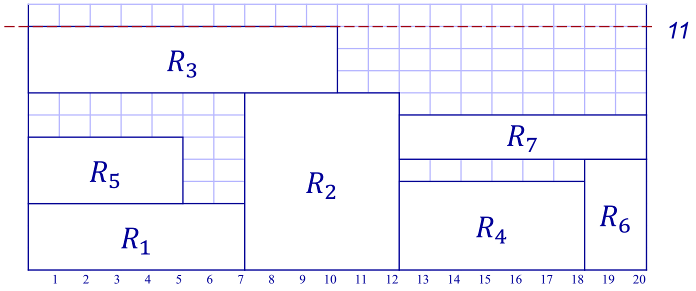
Bottom-left (BL) algorithm

Rectangle to be placed as near to the bottom of the bin as it will fit

Then as far to the left as it can go at that level without overlapping any other packed rectangles

List of rectangles require no pre-sorting

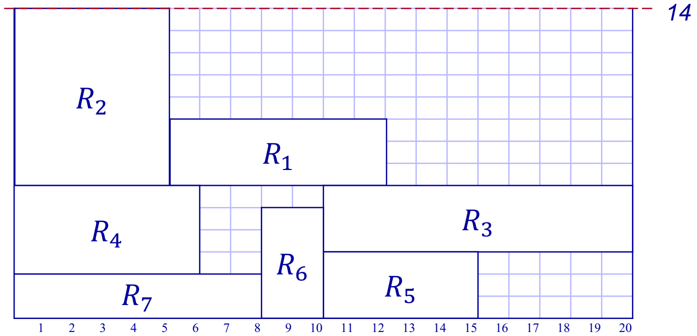
i	1	2	3	4	5	6	7
$w(i)$	7	5	10	6	5	2	8
$h(i)$	3	8	3	4	3	5	2



Bottom-left (BL) algorithm

Example: BL for $L = \{7,6,5,4,3,2,1\}$

i	1	2	3	4	5	6	7
$w(i)$	7	5	10	6	5	2	8
$h(i)$	3	8	3	4	3	5	2



Strip Packing Algorithms

Level-oriented algorithms

- The list of rectangles are pre-sorted into order of decreasing height
- The packing is done on a series of levels, that the bottom of each rectangle rests on
- The first level is the bottom of the bin and subsequent levels are defined by the height of the tallest rectangle on the previous level

Split algorithms

- The level oriented algorithms split the bin horizontally into blocks of a set width
- split the open ended bin vertically into smaller open ended bins depending on the widths of the rectangles
- The rectangles are first sorted by width

Strip Packing Algorithms

Shelf algorithms

- modifications of the level algorithms, that avoid pre-sorting the list of rectangles
- Rather than being determined by the tallest rectangle, the levels are fixed height shelves
- The shelf heights are set by a parameter r. $(0 < r < 1)$

Hybrid algorithms

- These use the characteristics of two or more of the types of algorithms described above. They may or may not involve pre-sorting

Strip Packing Algorithms

The Slave Algorithm

A slave algorithm is the approach that is used to decide which shelf a rectangle should be put on once all appropriate shelves have been determined

For the level-by-level algorithms, each level, or shelf, in the bin has exactly the same width, C. (the width of the bin itself)

This means that, for each rectangle, once the algorithm has calculated the levels it may be placed on, the decision of which of these levels to use is exactly a one-dimensional bin packing problem

That is, the number of levels being used, corresponds to the number of one-dimensional bins, of capacity C, that are required

This one-dimensional problem is solved by the slave algorithm

There are many one-dimensional algorithms that could be used as the slave algorithm, for instance: ***NF*** and ***FF*** algorithms

Strip Packing Algorithms

On-line vs Off-line

Whether or not the rectangles required sorting before being placed into the bin

This is an important factor to take into consideration when deciding on an appropriate algorithm for a bin packing problem

Consider the situation where the list of rectangles arrives one at a time

When each rectangle arrives it must immediately be assigned its place in the bin

Only once this has happened, does the identity of the next rectangle become known

This type of environment is called on-line

On-line algorithms must assume no prior knowledge of the rectangles in the list, so pre-sorting the list of rectangles is not an option

Strip Packing Algorithms

Off-line environment

it is possible to view the whole list of rectangles first and so sort them into any order before they are placed on the bin

- Off-line algorithms assume prior knowledge of the whole problem before any packing has to be done
- An on-line algorithm would be used in a situation where the jobs had to be done in order. For example, the items may be subject to different priorities or deadlines

An off-line algorithm would be used when the order did not matter

- For example, if pieces of material were being cut out to make a jacket, it would not matter if a sleeve was cut out before the collar, just as long as all the required pieces were produced in the end.

Strip Packing Algorithms

Next-fit decreasing-height (NFDH)

The algorithm:

- sorts the items by order of non-increasing height
- places the first item in the position (0,0)
- places the items next to each other in the strip until the next item will overlap the right border of the strip

New level at the top of the tallest item in the current level and places the items next to each other in this new level

First-fit decreasing-height (FFDH)

The algorithm works similar to the NFDH algorithm,

However, when placing the next item

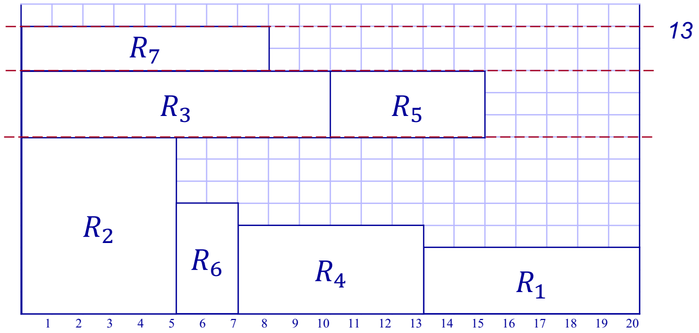
- Scans the levels from bottom to top
- Places the item in the first level on which it will fit

A new level is only opened if the item does not fit in any previous ones

Next-fit decreasing-height (NFDH)

Example:

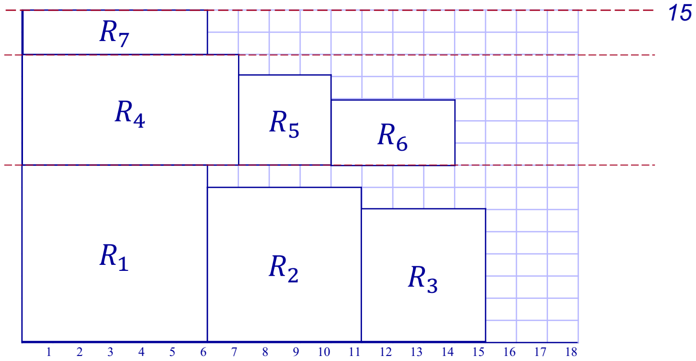
i	1	2	3	4	5	6	7
$w(i)$	7	5	10	6	5	2	8
$h(i)$	3	8	3	4	3	5	2



Next-fit decreasing-height (NFDH)

Example:

i	1	2	3	4	5	6	7
$w(i)$	6	5	4	7	3	4	6
$h(i)$	8	7	6	5	4	3	2



Strip Packing Algorithms

Next-fit decreasing-height (NFDH)

The algorithm:

- sorts the items by order of non-increasing height
- places the first item in the position (0,0)
- places the items next to each other in the strip until the next item will overlap the right border of the strip

New level at the top of the tallest item in the current level and places the items next to each other in this new level

First-fit decreasing-height (FFDH)

The algorithm works similar to the NFDH algorithm,

However, when placing the next item

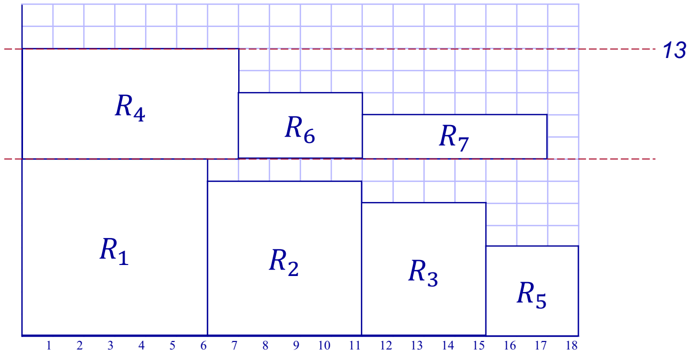
- Scans the levels from bottom to top
- Places the item in the first level on which it will fit

A new level is only opened if the item does not fit in any previous ones

First-fit decreasing-height (FFDH)

Example:

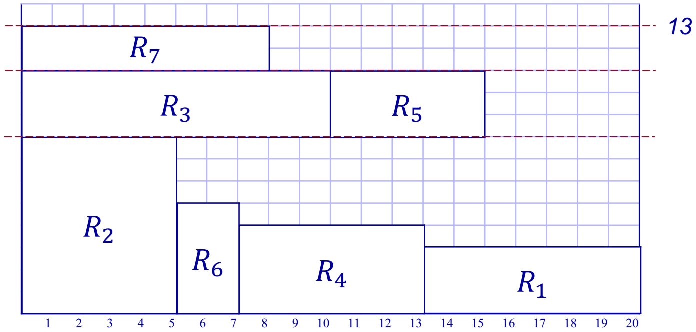
i	1	2	3	4	5	6	7
$w(i)$	6	5	4	7	3	4	6
$h(i)$	8	7	6	5	4	3	2



First-fit decreasing-height (FFDH)

Example:

i	1	2	3	4	5	6	7
$w(i)$	7	5	10	6	5	2	8
$h(i)$	3	8	3	4	3	5	2



Strip Packing Algorithms

Best-fit decreasing-height (BFDH)

The algorithm works similar to the NFDH algorithm,
However, when placing the next item

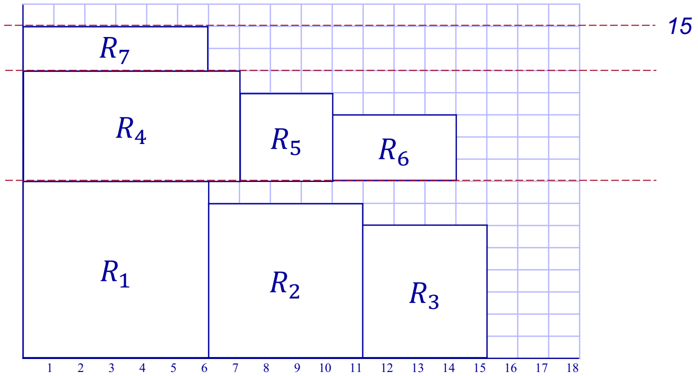
- Scans the levels from bottom to top
- Places the item in the first level on which the horizontal residual space is the minimum

A new level is only opened if the item does not fit in any previous ones

Best-fit decreasing-height (BFDH)

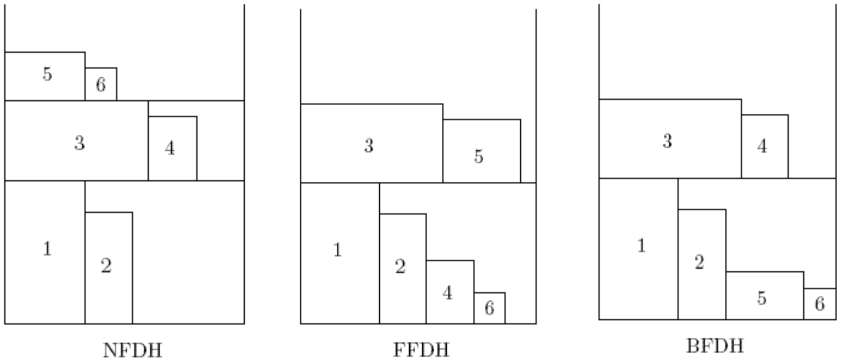
Example:

i	1	2	3	4	5	6	7
$w(i)$	6	5	4	7	3	4	6
$h(i)$	8	7	6	5	4	3	2



Strip Packing Algorithms

Example:



NFDH FFDH BFDH

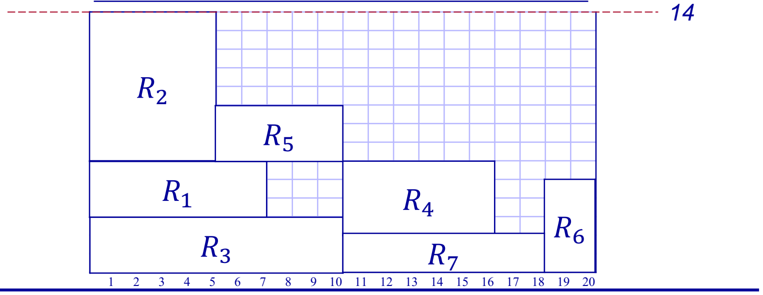
https://cgi.csc.liv.ac.uk/~epa/surveyhtml.html

Bottom-Left (BL) Algorithm

- Orders items by non-increasing width.
- Packs the next item as near to the bottom as it will fit and then as close to the left as it can go without overlapping with any packed item

Note that BL is not a level-oriented packing algorithm

i	1	2	3	4	5	6	7
$w(i)$	7	5	10	6	5	2	8
$h(i)$	3	8	3	4	3	5	2



Strip Packing Algorithms

Baker's Up-Down (UD) algorithm

- Combines BL and a generalization of NFDH
- Normalizes the width of strip and items, strip is of unit width
- Divides the items into $(1/2,1], (1/3,1/2], (1/4,1/3], (1/5,1/4], (0,1/5]$, items should order in non-increasing
- The strip is also divided into five regions $R_1, \dots, R_5$
- Some items of width in the range are packed to region Ri by BL
- Packs the item to R_j for $j = 1, \dots, 4$ (in order) from top to bottom
  - Since BL leaves a space of increasing width from top to bottom at the right side of the strip
- The item is packed to R_i by BL if there is no such space
- Finally, items of size at most $1/5$ are packed to the spaces in $R_1, \dots, R_4$ by the (generalized) NFDH algorithm
  - if there is no space in these regions, the item is packed to R_5 using NFDH

Strip Packing Algorithms

Baker's Up-Down (UD) algorithm

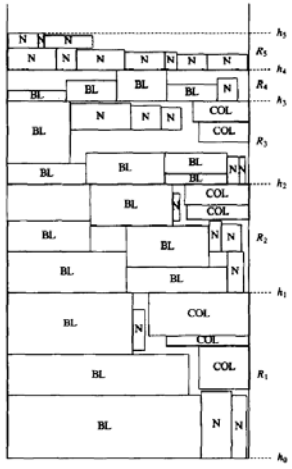


FIG. 2. A UD packing.

Strip Packing Algorithms

Reverse-fit (RF) algorithm

- Normalizes the width of the strip and the items
- Stacks all items of width greater than 1/2
- Remaining items are sorted in non-increasing height and will be packed above the height H_0 reached by those greater than 1/2
- Packs items from left to right with their bottom along the line of height H_0 until there is no more room
- Packs items from right to left and from top to bottom (called reverse-level) until the total width is at least 1/2
- The reverse-level is dropped down until (at least) one of them touches some item below.
- The drop down is somehow repeated

Strip Packing Algorithms

Reverse-fit (RF) algorithm

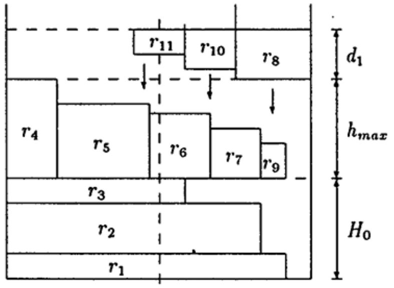


Figure 1

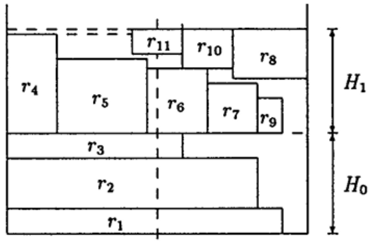


Figure 2

Strip Packing Algorithms

Steinberg's algorithm

- Estimates an upper bound of the height H required to pack all the items such
  - ○ it is proved that the input items can be packed into a rectangle of width W and height H
- Define seven procedures (with seven conditions), each to divide a problem into two smaller ones and solve them recursively

It has been showed that any tractable problem satisfies one of the seven conditions

Strip Packing Algorithms

Split-Fit algorithm (SF)

- Divides items into two groups, L1 with width greater than 1/2 and L2 at most 1/2
- All items of L1 are first packed by FFDH
- Then they are arranged so that all items with width more than 2/3 are below those with width at most 2/3.
  - ○ This creates a rectangle R of space with width 1/3
- Remaining items in L2 are then packed to R and the space above those packed with L1 using FFDH
  - ○ The levels created in R are considered to be below those created above the packing of L1

Strip Packing Algorithms

Split-Fit algorithm (SF)

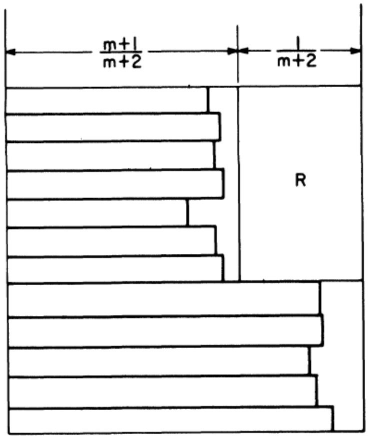


FIG. 6. *The rectangle R created in the space left by the rearranged* FFDH *packing of* L_1.

Bin Packing Problem

Outline

1. Introduction

Metaphorically, there never seem to be enough bins for all one needs to store. Mathematics comes to the rescue with the *bin packing problem* and its relatives.

The bin packing problem raises the following question:

- given a finite collection of *n* weights $w_1, w_2, w_3, \ldots, w_n$, and
- a collection of identical bins with capacity C (which exceeds the largest of the weights),
- what is the minimum number *k* of bins into which the weights can be placed without exceeding the bin capacity C?

Outline

We want to know how few bins are needed to store a collection of items.

This problem, known as the 1-dimensional bin packing problem, is one of many mathematical packing problems which are of both theoretical and applied interest.

It is important to keep in mind that "weights" are to be thought of as indivisible objects rather than something like oil or water.

For oil one can imagine part of a weight being put into one container and any left over being put into another container.

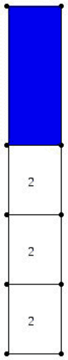
However, in the problem being considered here we are not allowed to have part of a weight in one container and part in another.

One way to visualize the situation is as a collection of rectangles which have height equal to the capacity C and a fixed width, whose exact size does not matter.

When an item is put into the bin it either falls to the bottom or is stopped at a height determined by the weights that are already in the bins.

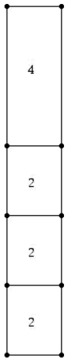
Outline

The diagram below shows a bin of capacity 10 where three identical weights of size 2 have been placed in the bin, leaving 4 units of empty space, which are shown in blue.



Outline

By contrast with the situation above, the bin below has been packed with weights of size 2, 2, 2 and 4 in a way that no room is left over.



Basic ideas

The bin packing problem asks for the minimum number *k* of identical bins of capacity C needed to store a finite collection of weights $w_1, w_2, w_3, \ldots, w_n$ so that no bin has weights stored in it whose sum exceeds the bin's capacity.

Traditionally

- capacity C is chosen to be 1 and
- weights are real numbers which lie between 0 and 1,
- for convenience of exposition, C is a positive integer and the weights are positive integers which are less than the capacity.

Example 1:

- Suppose we have bins of size 10. How few of them are required to store weights of size 3, 6, 2, 1, 5, 7, 2, 4, 1, 9?

Basic ideas

The weights to be packed above have been presented in the form of a *list* L ordered from left to right.

For the moment we will seek procedures (algorithms) for packing the bins that are "driven" by a given ***list*** **L** and a **capacity size C** for the bins.

The goal of the procedures is to **minimize the number of bins** needed to store the weights.

A variety of simple ideas as to how to pack the bins suggest themselves.

One of the simplest approaches is called *Next Fit* (NF).

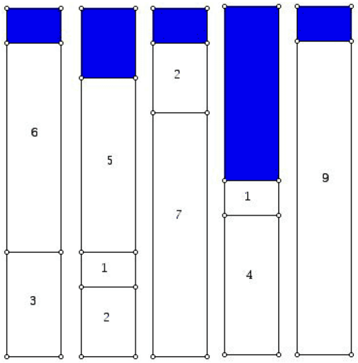
The idea behind this procedure is to open a bin and place the items into it in the order they appear in the list.

If an item on the list will not fit into the open bin, we close this bin permanently and open a new one and continue packing the remaining items in the list.

Basic ideas *Next Fit* (NF)

If some of the consecutive weights on the list exactly fill a bin, the bin is then closed and a new bin opened.

When this procedure is applied to the list above we get the packing shown below.



Basic ideas *Next Fit* (NF)

Next Fit is

- very simple,
- allows for bins to be shipped off quickly, because even if there is some extra room in a bin, we do not wait around in the hope that an item will come along later in the list which will fill this empty space.

One can imagine having a fleet of trucks with a weight restriction (the capacity C) and one packs weights into the trucks.

If the next weight cannot be packed into the truck at the loading dock, this truck leaves and a new truck pulls into the dock.

We keep track of how much room remains in the bin open at that moment.

In terms of how much time is required to find the number of bins for *n* weights, one can answer the question using a procedure that takes a linear amount of time in the number of weights (*n*).

Clearly, NF does not always produce an optimal packing for a given set of weights. You can verify this by finding a way to pack the weights in Example 1 into 4 bins.

Basic ideas *Next Fit* (NF)

Procedures such as NF are sometimes referred to as *heuristics* or *heuristic algorithms* because although they were conceived as ways to solve a problem optimally, they do not always deliver an optimal solution.

Can we find a way to improve on NF so as to design an algorithm which will always produce an optimal packing?

A natural thought would be that if we are willing to keep bins open in the hope that we will be able to fill empty space with items later in list L, we will typically use fewer bins.

Basic ideas *First Fit* (FF)

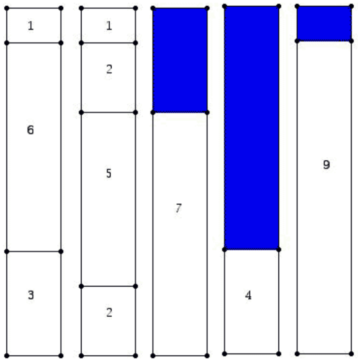
The simplest way to carry out this idea is known as *First Fit*.

We place the next item in the list into the first bin which has not been completely filled (thought of as numbered from left to right) into which it will fit.

- When bins are filled completely they are closed,
- If an item will not fit into any currently open bin, a new bin is opened.

Basic ideas *First Fit* (FF)

The result of carrying out First Fit for the list in Example 1 and with bins of capacity 10 is shown below:



Basic ideas *First Fit* (FF)

Both methods we have tried have yielded 5 bins.

We know that this is not the best we can hope for.

One simple insight is obtained by computing the total sum of the weights and dividing this number by the capacity of the bins.

Since we are dealing with integers, the number of bins we need must be at least $\lceil \Omega / C \rceil$ where $\Omega = \sum_{i=1}^{n} w_i$.

(Note that $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to *x*).

Clearly, the number of bins must always be an integer. In Example 1, since Ω is 40 and C is 10, we can conclude that there is hope of using only 4 bins.

However, neither Next Fit nor First Fit achieves this value with the list given in Example 1. Perhaps we need a better procedure.

Basic ideas *Best Fit* (BF) and *Worst Fit* (WF)

Two other simple methods in the spirit of Next Fit and First Fit have also been looked at.

These are known as *Best Fit* (BF) and *Worst Fit* (WF).

For **Best Fit**, one again keeps bins open even when the next item in the list will not fit in previously opened bins, in the hope that a later smaller item will fit.

The criterion for placement is that we put the next item into the currently open bin (e.g. not yet full) which leaves the least room left over. (In the case of a tie we put the item in the lowest numbered bin as labeled from left to right.)

For **Worst Fit**, one places the item into that currently open bin into which it will fit with the most room left over.

Basic ideas *Best Fit* (BF) and *Worst Fit* (WF)

The amount of time necessary to find the minimum number of bins using either FF, WF or BF is higher than for NF. What is involved here is $n \log n$ implementation time in terms of the number n of weights.

The distinction between First Fit, Best Fit and Worst Fit:

- ○ suppose that we currently have only 3 bins open with capacity 10
- ○ *remaining space* as follows:
  - Bin 4, 4 units,
  - Bin 6, 7 units, and
  - Bin 9 with 3 units.

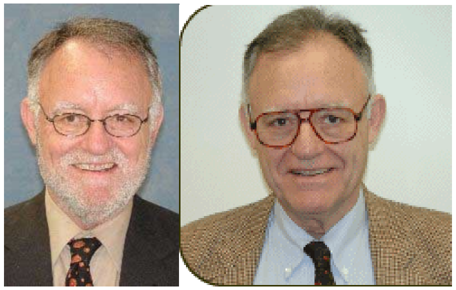
Suppose the next item in the list has size 2.

First Fit puts this item in Bin 4, Best Fit puts it in Bin 9, and Worst Fit puts it in Bin 6!

One difficulty is that we are applying "good procedures" but on a "lousy" list. If we know all the weights to be packed in advance, is there a way of constructing a good list?

Basic ideas

Bin packing is a very appealing mathematical model, and yet work on this problem is surprisingly recent. As an organized subject this topic is only about 35 years old. Major pioneers and contributors in working on this problem are Edward Coffman, Jr., Michael Garey, Ronald Graham, and David Johnson.



Department of Industrial Engineering and Operations Research, Computer Science Department, President, Armstrong Memorial Research Foundation
Columbia University

Edward Coffman, Jr.

Michael Garey

Ronald Graham

David Johnson

More approaches to bin packing

All of the algorithms we have discussed so far have the property that the fact that there may be more items in the list to pack (farther to the right than one being currently worked on) does not affect what is done to pack the current item.

Such algorithms are known as *on-line*.

The idea for on-line algorithms, whether they are being used to solve bin packing problems or other combinatorial optimization problems, is that

- not all the components of the problem are known in advance

In the bin packing case one can imagine an industrial situation where items with different weights are being produced and then are being placed in bins which are at some stage to be shipped to customers.

More approaches to bin packing

In contrast to an on-line point of view is the possibility of using an *off-line* approach.

- one thinks of having all of the items to be packed in advance.
- one can ask for the given weights to be packed if there is some rearrangement of the weights into a list different from the original which might be used to give a better result for the number of bins required.
- If there are *n* items to pack, the number of potential lists for these *n* items is *n*!, since the first item can be chosen in *n* ways, the second in *n*-1, etc., giving *n*! as the number of different possible lists.
- choosing a list for an optimal packing has the flavor of looking for a needle in a haystack. (Even for 20 items to pack, say, 20! is a very large number.)

More approaches to bin packing

In the on-line environment using FF, for example, at a given stage one may have so many open bins that it becomes economically unrealistic to have so many empty bins being monitored in the hope that later items will fit efficiently into them.

This suggests a version of bin packing where one seeks a packing heuristic but is limited to having at most K bins open at a given time.

These heuristics are known as ***bounded-space on-line heuristics***.

For the heuristics discussed above one can try to develop a K open bin bounded space version of the heuristic.

However, the situation for $K > 1$ open bins raises some tantalizing issues. Not only do we have the option of specifying the way the bins are packed, we have an option of specifying when a bin will be shut down (closed permanently) other than when it is completely full!

More approaches to bin packing

Suppose that we have K open bins and we now must pack a weight which does not fit into any of the open bins.

Since we must open a new bin, we have to choose an old bin to shut down.

This can be done either by

- picking the lowest numbered bin to shut down (this is in the spirit of FF) or by
- shutting down the bin which is closest to being completely full (this is in the spirit of BF).

Using either the FF or BF packing rule with the FF or BF closing rule, we get four(!) new heuristics in the K open bin environment.

More approaches to bin packing

In Example 1 we noticed that the very large item to pack at the end of the list made it necessary to have an extra bin opened at the end.

Intuitively, it seems like a good idea to try to pack large items first.

This is the same strategy you would use to pack a suitcase for a vacation; you would not leave a large-volumed item to the end!

This suggests the following approach to off-line bin packing.

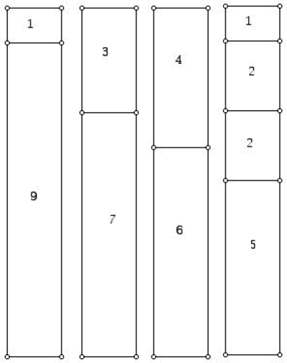
Choose the list where the weights appeared in sorted order, from largest to smallest. Now use any of the algorithms that we have discussed to carry out the packing.

We have already discussed four packing algorithms: NF, FF, BF, and WF and

We now have four new approaches which we will call NFD, FFD, BFD, and WFD where in each case the "D" refers to decreasing.

For example, FFD means First Fit Decreasing, and would give rise to the packing for the weights in Example 1 shown below. Note that this packing is optimal.

More approaches to bin packing



Not only can one not pack these weights into fewer bins but also there is no wasted space.

Of course, there are many situations where optimal packings still have unused space.

More approaches to bin packing

Among all packings of this kind one might look for that packing with various characteristics, say, that the fewest bins have extra space.

Alternatively, one might want a packing where the number of bins used is minimal but as many bins as possible have some extra room. (A little extra space might allow putting packing material into the bin to prevent breakage during shipment.)

Once one has an optimal packing, one can often either rearrange the items in one of the bins or between bins in a way that achieves some secondary goal beyond minimizing the number of bins.

The fact that FFD yielded an optimal solution in Example 1 does not mean that this will be the case for other problems the method is applied to.

Perhaps you are not surprised to learn that First Fit Decreasing does not always yield optimal solutions. Here is an example:

More approaches to bin packing

Example 2:

Suppose that one has bins of capacity 20.

What would be the fewest bins needed to pack weights of size 4, 8, 7, 10, 3, 8?

More approaches to bin packing

Since the sum of the weights is 40, with bins of capacity 20 a packing with only 2 bins might be achieved.

In fact there is indeed a packing needing only two bins:

- Bin 1: 8, 8, 4 and
- Bin 2: 10, 7, 3.

In this notation the item of weight 8 is at the bottom of Bin 1, next comes the item of weight 8, and the item of size 4 occupies the space at the top of the bin. Note that this solution is not unique because the items within the bin can be permuted.

However, with this particular list none of the procedures, NF, FF, BF, WF, NFD, FFD, BFD, or WFD yields an optimal number of bins.

In each case 3 bins are required, as you can check for yourself.

If one had been given, say, the list 8, 4, 8, 10, 7, 3 then NF, FF, BF, and WF would all give rise to an optimal packing (with the weights packed slightly differently from the solution given above).

More approaches to bin packing

After finding eight appealing heuristics or approximation algorithms in the search to find an optimal solution to the bin packing problem,

One might be tempted to wonder if finding an optimal solution to bin packing is very hard!

Mathematicians have attempted to show that some problems are indeed very hard using a variety of approaches.

One of these approaches involves showing that a problem is *NP-complete*.

Intuitively, a problem Z is NP-complete when it has been shown that if this problem can be solved in polynomial time, then so can a very large number of other problems;

while if Z can be proved to require an exponential amount of time to solve, then so will all of the other problems.

Thus, the NP-complete problems are thought to be hard to solve but either all of the problems are, in fact, not really that hard (in the sense of only requiring polynomial work) or all of the problems actually require an exponential amount of work to solve.

More approaches to bin packing

Since many NP-complete problems are of great importance for applications in operations research (management science), the fact that mathematicians and computer scientists are unsure of the status of the NP-complete problems is frustrating!

You guessed it--the "decision version" of bin packing is known to be NP-complete. That is, given a capacity C and a list L of weights and an integer D the problem of determining if the weights in L can be packed into D or fewer bins of capacity C is NP-complete.

Thus, finding approximately optimal solutions for bin packing in polynomial time is probably the best we can hope for.

Applications of bin packing

So far we have alluded to only a few applications of bin packing, such as packing trucks with weight capacity C. (Realistic versions of truck packing problems typically go beyond issues of one-dimensional bin packing.)

There is an important connection between bin packing and another very important collection of operations research questions, often referred to as ***machine scheduling* problems**.

Consider the problem of scheduling identical machines with tasks that are independent of each other, that is, the tasks can be done by the machines in any order.

Each of the tasks has a time that is necessary to complete it on one of the machines.

More complex machine scheduling problems have to deal with the problem that often some tasks cannot be worked on before other tasks are completed.

Applications of bin packing

Suppose one wants to know what is the minimum number of machines which are needed to finish a collection of independent tasks by time T with times to complete the tasks of $t_1, t_2, \ldots, t_n$?

As you probably realize this question is bin packing with a very minimal disguise (change *t* to *w*; T to C)!

For example, suppose one has photocopying jobs of varying numbers of pages that have been brought into a photocopy shop.

If the shopkeeper wants the automatic work of the machines to enable her to go home within 3 hours of the start of the photocopy tasks, how many machines would have to be available to do the work, and how should the jobs be assigned to the machines?

http://www.ams.org/featurecolumn/archive/bins5.html

Bin Packing and Machine Scheduling

Feature Column Archive
http://www.ams.org/featurecolumn/archive/packings1.html

Bin Packing and Machine Scheduling. Introduction

Packing ads into breaks is an example of a bin packing problem.

Packing a collection of ads of different lengths into the minimum number of bins of fixed size (the length of the ad break).

Scheduling problems are present in the operation of all large systems:

- scheduling classes in schools;
- scheduling planes, trains, and buses in the transportation sector of the economy;
- scheduling machines in the manufacture of the products.

Bin Packing Insights into solving hard problems

The one-dimensional bin packing problem belongs to the complexity class of a group of problems for which no known algorithm solves the problems in a polynomial time function (polynomial in the number of weights) of effort.

The next step was to see how badly the various appealing "heuristic algorithms" which might be used to solve them might behave.

We can think of a heuristic algorithm (one not always guaranteed to find the optimal solution to a problem) as finding an "approximate" solution to a problem.

We can now ask the question:

Given a collection of weights and bins of capacity C,

how many bins--compared with the optimum number actually required--do these approximation algorithms require?

Bin Packing Insights into solving hard problems

Suppose we are given weights, a capacity, a list L, and an algorithm A.

Denote by

- $OPT(L)$ the optimum number of bins,
- $A(L)$ the number of bins which are required by algorithm A applied to list L.

How do OPT(L) and A(L) compare in size? To answer questions such as this, one can apply mathematical arguments to show that:

$$A(L) \leq rOPT(L)$$

where *r* is some positive constant.

One can then try to show by way of examples that the r in the equation above can actually occur.

Bin Packing Insights into solving hard problems

Many researchers in mathematics and computer science have contributed to insights in this vein for the literally dozens of different algorithms that have been investigated to understand one-dimensional bin packing.

Suppose we are dealing with the heuristic NF, Next Fit. It is not difficult to see that Next Fit obeys:

$$NF(L) \leq 2OPT(L)$$

In the case where the capacity of the bins is 1, two consecutive bins that are packed by Next Fit can not both be filled to less than the halfway mark.

The reason is that the contents of two such bins would then fit in only one of the bins, because of the way Next Fit fills the bins.

Bin Packing Insights into solving hard problems

The proof of the inequality above uses this fact.

For a further insight, consider the following list of weights for the case where the bins have capacity 1:

$$L = (\frac{1}{2}, \frac{1}{2N}, \frac{1}{2}, \frac{1}{2N}, \ldots, \frac{1}{2})$$

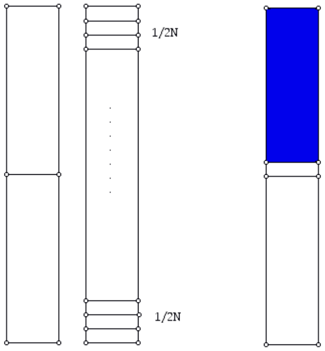
where N is a positive integer which is 2 or bigger.

NF applied to this list will produce a packing which has 2N bins, each bin packed with an item of weight 1/2 at the bottom and an item of weight $\frac{1}{2N}$ on top.

However, the optimal packing for this list uses N + 1 bins: N bins, each with two weights of size 1/2, and one additional bin to contain the N items of size $\frac{1}{2N}$.

Bin Packing Insights into solving hard problems

Below are some diagrams to help visualize the packing which is optimal versus what happens when NF is used.



The optimal packing has N copies of the one bin pictured on the far left and one copy of the next bin. The NF packing has 2N copies of the bin on the far right. The blue denotes unused space.

Bin Packing Insights into solving hard problems

This analysis shows that we have a family of lists for which

$$NF(L) \geq 2OPT(L) - 1$$

Over a period of about 30 years, researchers have examined a wide variety of variants of bin packing algorithms and obtained increasingly better worst case analyses of the different algorithms.

For example, First Fit does significantly better than NF but not spectacularly so:

$$FF(L) \leq \lceil (17/10)OPT(L) \rceil$$

where $\lceil y \rceil$, the ceiling function, denotes the smallest integer greater than or equal to y. Intuitively, one would expect the "decreasing" heuristics to work better and this turns out to be the case.

Bin Packing Insights into solving hard problems

For First Fit Decreasing we have the result:

$$FFD(L) \leq \left(\frac{11}{9}\right) OPT(L) + 4$$

In addition to analyzing the worst-case performance of bin packing algorithms, one can also see how different heuristics perform on the average.

Initially, doing simulations sometimes only hints at results. However, insights gleaned from these simulations often lead to probability models in which precise results can be obtained.

Applications of bin packing

Many applications of bin packing come to mind. For example,

- placing computer files with specified sizes into memory blocks of fixed size, or
- the recording of all of a composer's music, where the length of the pieces to be recorded are the weights and the bin capacity is the amount of time that can be stored on an audio CD (about 80 minutes)

One power of mathematics is that, having abstracted the problem of packing bins with weights, there are many situations to which the mathematical techniques apply

- First Fit heuristic can be used in these different situations

We can now go out and look for problems where bin packing and the algorithms which have been developed to get insights into this problem can be put to use

However, applying mathematics originally developed in one context to other applications contexts which have "additional aspects" to them, encourages one to improve on the mathematics that has been done already

Applications of bin packing

The problem of preparing a collection of musical pieces to store on audio compact discs.

- An audio CD has a maximal capacity; in that regard it follows the bin packing model
- The pieces one wants to put on the compact discs play the roles of the weights, and
- one would not want to have a piece start on one compact disc and finish on another

But there is a subtlety here. Classical music pieces often come in sections called movements.

- Although ideally one would prefer to have whole pieces on each CD,
- it might be "acceptable" to keeping costs down and give consumers more value for their money
    - ○ the first three movements of a symphony on one CD and the last movement at the beginning of the next CD.

If we just treat the weights as movements, we cannot guarantee that all of the heuristics which might be applied to fitting the music onto the compact discs would respect the fact that the weights which correspond to the movements of a piece must be packed in order of the movements.

This additional constraint on the problem might inspire one to find specialized algorithms for solving bin packing problems where some of the weights need to be packed in a particular order.

Applications of bin packing

Inspired by various "packing" situations here are a variety of "extensions" of the original bin packing model that might surface:

a. *Packings in which the number of items that can be placed in a bin is restricted in advance not to exceed a certain number.*

- This restriction might be required in the situation where one is packing trucks.

b. *Restrictions on items which can be placed into the same bin.*

- This restriction might come about if items are being shipped from A to B in bins. In order to guarantee that if one bin gets lost or destroyed in transit to B, one could send redundant items and require that they not be packed in the same bin.
- An alternate scenario is that the items being packed may be generating heat, perhaps because they have some level of radioactivity. Now one desires that the items in one bin not only fit in the bin but that the items not generate too much heat during the period they are in the bins.

Applications of bin packing

c. *Packings in which there is an ordering attached to some of the weights which limits the way those items can be packed.*

- This restriction is in the sprit of the example discussed in a little detail above where music is being packed into compact discs of the same size.

d. *Packings in which the items being packed may be allowed to disappear during the packing process.*

- Thus, an item placed in a bin which has not yet been closed may be allowed to be removed from the bin either because fewer bins will be needed once this item is transferred to another bin, or because sometimes items not only enter the system but also leave the system.

Applications of bin packing

Here is a scenario which offers a potential applications environment for this variant of bin packing.

Imagine that when items are packed, a test is started to determine if the item which has been packed has spoiled. This test takes a certain amount of time to complete.

- If the packed item is put into a bin which is closed before the testing period is done, the packed bin is just shipped out.
- However, if the bin is still open during the period of packing and the finished test shows the item to have spoiled, then this item can be removed and the space used for items that haven't spoiled.

With cleverness one can perhaps find additional unexpected uses here.

One interesting variant of the bin packing problem involves a loosening of the requirement that all the bins have the same size.

- Suppose that we have bins of a standard capacity or size, say 1, and we think of this capacity also as having a cost of 1.

Applications of bin packing

Suppose that we can occasionally imagine that bins are stretched or enlarged but we must pay a penalty for doing this.

- The penalty or cost takes the form that a bigger bin is used.
- The goal is to pack the weights into stretched bins if necessary so that the total cost is a minimum.

In the standard one-dimensional bin packing problem, the bins are assumed to have the same size,

but you may enjoy formulating a variety of problems with a limited number of sizes for bins where one wants to pack the weights so that the total size of the bins used is as small as possible.

You may also want to think up applied situations where problems such as these might arise

Applications of bin packing

Another variant which uses the word bin but is in many ways quite far from the classical bin packing problem has to do with the common phenomenon of recycling bins.

For simplicity we will assume that we have a collection of bins which contain three types of glass: clear, brown, and frosted.

- The goal is to sort the different kinds of glass so that after the sorting process one bin has only clear glass, one only brown glass, and one only frosted glass.

We do not specify which of the bins is to have which kind of glass, but one can imagine another variant where the bins are specified beforehand.

- The goal of the problem is to conduct the sorting with as few moves as possible.

Bin packing and machine scheduling

The essence of mathematical modeling is controlled simplification.

- One takes the situation outside of mathematics, holds onto essential aspects of the problem and disregards "details" that are secondary to the situation.
- If one is fully successful doing this process, then after the mathematical problem associated with the model is solved, the mathematics can be used to get important insights into the original problem even though significant detail of the original problem was disregarded.
- To some extent the creation of the mathematics problem known as bin packing grew out of attempts to get insight into problems outside of mathematics.

Bin packing and machine scheduling

The range of scheduling problems makes it virtually impossible for one mathematical model to be useful in all situations.

How is bin packing connected with machine scheduling?

Suppose that we have tasks which take differing times to complete which can be performed in any order on identical machines.

Our goal is to determine the minimum number of machines needed to finish the tasks by a fixed time.

The fixed desired completion time corresponds to the capacity of the bins, and weights can be thought of as the times to do the tasks.

When we find the minimum number of bins needed to pack the weights we are finding the minimum number of machines needed to finish by the desired time!

Bin packing and machine scheduling

Here is a sample of the many complexities that machine scheduling problems have.

1. A job that must be completed by scheduling the tasks that make up the job often has to be carried out on very different kinds of processors.

Example: To schedule the construction of your dream home requires the completion of a gigantic number of steps. One needs a building permit, someone to lay the foundation, and someone to put on the roof. The "processors" that must be scheduled for doing this work are not interchangeable. Plumbers rarely do electrical work and electricians rarely do roofing.

2 Schedules for the manufacturing of many identical products often require that each product get some time on a fixed number of different machines, where there may be several identical machines of one kind.

Example: Manufacturing a computer chip.

Bin packing and machine scheduling

3. When schedules are constructed there are different approaches to how the processors work on the tasks assigned.

- One approach is that once a processor begins work on a task, that processor continues until the work on that task is complete.
- Alternatively, there may be conditions under which a processor will be permitted, perhaps even required, to interrupt work on one task to begin another task.

Examples: Whereas in manufacturing once a machine starts work on a task it usually continues to work on it until the task is complete

In medicine things are somewhat different.

- When doctors schedule patients, they typically stay with a patient until the task is done.
- An exception would be an ophthalmologist who might put drops in a patient's eyes, see another patient while the first patient's pupils are increasing in size, and then return to complete the first patient's examination.
- Similarly, a doctor in regular practice who is seeing a patient who has a head cold would probably interrupt her work to see a patient who arrived in the office complaining of chest pains.

Algorithm for machine scheduling

The scheduling problems we consider are completely deterministic.

- times for completing every task making up a complex job are known in advance.
- we are given a directed graph which is called a *task-analysis digraph*, which indicates which tasks come immediately before other tasks. (Note that this digraph will have no directed circuit (e.g a way to follow edges and start at a vertex and return there) because that would lead to a contradiction in the precedence relations between the tasks.)
  - An arrow from task T_i to T_j means that task T_i must be completed before task T_j can begin.

Such restrictions are very common in carrying out the individual tasks that make up a complex job.

Algorithm for machine scheduling

Also, in the task-analysis digraph below, each vertex, which represents a task, has the time for the task inside the circle corresponding to that vertex.

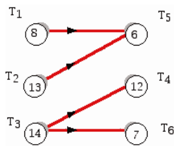


Figure 1

We want to schedule these six tasks on some fixed number of identical machines in such a manner that the tasks are completed by the earliest possible time.

- This is sometimes referred to as finding the *makespan* for the tasks that make up the job.

We also want to be specific about which tasks should be scheduled on which machines during a given time period.

We will assume that the scheduling is carried out so that no machine will remain voluntarily idle, and that once a machine begins working on a task it will continue to work on it without interruption until the task is completed.

Algorithm for machine scheduling

There is an appealing heuristic with which to approach this problem.

The heuristic has the advantage of being relatively simple to program on a computer, and when carried out by hand by different people, leads to the same schedule for the tasks (because the ways ties can be broken is specified).

Typically, the heuristic gives a good approximation to an optimal schedule. This algorithm (heuristic) is known as the *list processing algorithm*.

The algorithm works by coordinating the scheduling of the tasks on the machines and taking account of a "priority list" and then coordinating this list with the demands imposed by the task analysis digraph.

You can think of the given list as a kind of priority list which is independent of the scheduling requirements imposed by the task analysis digraph. The tasks are given in the list so that when read from left to right, tasks of higher priority are listed first. For example, the list may reflect an ordering of the tasks based on the size of cash payments that will be made when the tasks are completed.

Alternatively, the list may have been chosen with the specific goal of trying to minimize the completion time for the tasks.

Algorithm for machine scheduling

With respect to constructing a schedule,

- a task is called *ready* at a time *t* if it has not been already assigned to a processor and
- all the tasks which precede it in the task analysis digraph have been completed by time *t*.

For the task analysis digraph in Figure 1 the tasks ready at time 0 are T_1, T_2, and T_3.

Remember that we are assuming that machines do not stay voluntarily idle. This means that as soon as one processor's task is completed, it will look for a new task on which to work.

In determining what this next task should be, one takes into account where on the priority list the task appears, as well as any constraints imposed by the task analysis digraph.

A machine will stay idle for a period of time only if there are no ready tasks (unassigned to other machines) that are ready at the given time.

The list processing heuristic assigns at a time *t* a ready task (reading from left to right) that has not already been assigned to the lowest-numbered processor which is not currently working on a task.

Algorithm for machine scheduling

As an example, consider trying to schedule the tasks in Figure 1 on two machines.

- I will refer to the two machines which must be scheduled as Machines M1 and M2.

Suppose we are given the list $L = T_1, T_2, T_3, T_4, T_5, T_6$. At time 0,

- M1 being idle, and T_1 being ready and first in the list, we can schedule T_1 on M1, which will keep that machine busy until time 8.
- M2 is free at time 0 so it also seeks a task to work on at time 0. The next task in the list, T_2, is ready at 0 so M2 can start at time 0 on task T_2.
  - ○ Both machines are now "happily" working until time 8 when M1 becomes free.
- Since T_3 is next in the priority list, and its predecessors (there are none) are done at time 8, M1 can work on this task because it is ready at time 8.
- However when time 13 comes, M2, just finishing T_2, would like to begin the next task in the priority list which has not yet been assigned to a machine.
  - ○ This would be T_4 but this task is not ready at time 13 because T_3 has not been completed.
- So M2 tries the next task in the priority list, T_5, and this task being ready at 13 (both T_1 and T_2 are done) can be assigned to M2.

Algorithm for machine scheduling

- You can keep track of what is going on in Figure 2 below where the tasks assigned to M1 are represented in the top row and the tasks assigned to M2 are shown in the bottom row.
  - ○ Idle time on a machine is represented in blue.

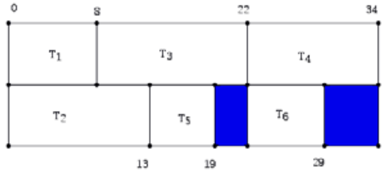


Figure 2

Continuing our analysis of how Figure 2 is generated, what happens when time 19 arrives, and M2 tries to find an unassigned task in the priority list?

- Since both task T_4 and T_6 are not ready at time 19 (because they can only begin when T_3 is done), M2 will stay idle until time 22
- At time 22, both machines are free, and in accordance with our tie-breaking rule, T_4 gets assigned to Machine 1 while T_6 gets assigned to M2.
- The completion time for this list is 34.

Algorithm for machine scheduling

Is this the best we can do?

One way to check, when task times are integers, is that we know that a lower bound for completing the tasks is the ceiling function applied to the following quotient:

- the sum of all task times divided by the number of machines.

In this case, we get a lower bound of 30, which means that there is perhaps some hope of finding a better schedule, one that finishes by time 30.

In addition to 30 there is potentially another independent lower bound that we can take into account. Suppose we find the length of all the paths in the task analysis digraph. In this example, two typical such paths are T_2, T_5 and T_3, T_4. The lengths of these two paths, computed by summing the numbers inside the vertices making up the paths, are respectively, 19 and 26. How are these numbers related to the completion time for all of the tasks? Clearly, since we are working with directed paths, the early tasks in the paths must be completed before the later ones. Thus, the completion time for all the tasks is at least as long as the time necessary to complete all of the tasks in any of the paths in the task analysis digraph. In particular, the earliest completion time is at least as long as the length of the longest path in this digraph. In this example, the length of this longest path is 26 and the path involved is T_3, T_4. The path in the task analysis digraph which has the longest length is known as a *critical path*. A given task analysis digraph has one or more critical paths. (This is true even if the digraph has no directed edges. In this case the length of the critical path is the amount of time it takes to complete the longest task.) Speeding up tasks that are not on the critical path does not affect this lower bound, regardless of the number of machines available. The earliest completion time must still be at least as long as the critical path, despite having a lot of processors to do the tasks not on the critical path(s).

Is it possible to improve the schedule that is displayed in Figure 2? One idea is to use a list to prevent the difficulties when lengthy tasks appear late in a list or when tasks on the critical path(s) are given "high priority." Thus, one can construct the list which orders the tasks by decreasing time (ties broken in favor of tasks with a lower number). The decreasing time list in this case would be: $T_3, T_2, T_4, T_1, T_6, T_5$. This list leads to a schedule with completion time 32. You can practice trying to use the list processing algorithm on this list with 2 processors. The result is a schedule that finishes at time 32 with only 4 time units of idle time on the second processor. Here is the way the tasks are scheduled: Machine 1: T_3, T_4, T_5; Machine 2: T_2, T_1, T_6, idle from 28-32. Although this schedule is better than the one in Figure 2, it may not be the optimal one, because there is still hope that a schedule which uses 30 time units on each machine with no idle time is possible.

Is there another list we could try that might achieve a better completion time? We have mentioned that tasks on the critical path are "bottlenecks" in the sense that when they are delayed, the time to complete all the tasks grows. This suggests the idea of a *critical path list*. Begin by putting the first task on a longest path (breaking ties with the task of lowest number) at the start of the list. Now remove this task and edges that come out of

it from the task analysis digraph and repeat the process by finding a new task to add to the list that is at the start of a longest path. Doing this gives rise to the list $T_3, T_2, T_1, T_4, T_6, T_5$. This list, though different from the decreasing time list actually, gives rise to exactly the same schedule we had before that finished at time 32. There are $6! =$ 720 different lists that can arise with six tasks, but the schedules that these lists give rise to need not be different, as we see in this case.

We have tried three lists and they each finish later than the theoretical, but *a priori* possible, optimum time of 30. There is also the possibility that there is a schedule that completes at time 31, with 2 units of idle time. You can check that no sum of two sets of task times yields a value of 30. You can also check that although there are two sets of task times (e.g. 13, 12, and 6; 14, 8 7) that sum to 31 and 29, no schedule based on the assignment of the associated tasks in order to schedule two machines obeys the restrictions imposed by the task analysis digraph. Thus, with a bit of effort one sees that the optimal schedule in this case completes at time 32.

The analysis of this small example mirrors the fact that for large versions of the machine scheduling problem, there is no known polynomial time procedure that will locate what the optimal schedule might be. This point brings us full circle to why the list processing heuristic was explored as a way of trying to find good approximate schedules. More is known for the case where the tasks making up the job can be done in any order, so-called *independent* tasks. This is the case where the task analysis digraph has no edges.

Ronald Graham has shown that for independent tasks the list algorithm finds a completion time using the decreasing time list which is never more than

$$((4/3) - 1/(3m))T$$

where T is the optimal time to schedule the tasks and m (at least 2) is the number of machines the tasks are scheduled on. This result is a classic example of the interaction of theoretical and applied mathematics.

Bin packing, an applied problem, led to many application insights as well as tools for solving a variety of theoretical problems. Bin packing relates to some machine scheduling problems, which in turn have rich connections with both pure and applied problems. Next month, I will explore some of these connections.

6. References

1. Assmann, S. and D. Johnson, D. Kleitman, J. Leung, On a dual version of the one-dimensional bin packing problem, J. Algorithms 5 (1984) 502-525.
2. Baker, B., A new proof for the first-fit decreasing bin-packing algorithm, J. Algorithms 6 (1985) 49-70.
3. Baker, B. and E. Coffman, Jr., A tight asymptotic bound for next-fit-decreasing bin packing, SIAM J. Alg. Disc. Math., 2 (1981) 147-152.
4. Baker, K., Introduction to Sequencing and Scheduling, Wiley, New York, 1974.
5. Bartal, Y. and A. Fiat, H. Karloff, R. Vohra, New algorithms for an ancient scheduling problem, In Proc. 24th ACM Symposium on the Theory of Computing, 1992, p. 51-58.
6. Bartal, Y. and H. Karloff, Y. Rabani, A better lower bound for on-line scheduling, Information Processing Letters, 50 (1994) 113-116.
7. Bentley, J. and D. Johnson, F. Leighton, C. McGeoch, L. McGeoch, Some unexpected expected behavior results for bin packing., in Proceedings of the 16th Annual ACM Sym. on Theory of Computing, 1984, p. 279-288.
8. Brucker, P., Scheduling Algorithms, Springer-Verlag, New York, 1995.
9. Coffman, Jr., E., (Ed.), Computer & Job/Shop Scheduling Theory, Wiley, New York, 1976.
10. Coffman, Jr., E. An introduction to proof techniques for packing and sequencing algorithms, in Deterministic and Stochastic Scheduling, M. Dempster, et al., (eds.), Reidel, Amsterdam, 1982, p. 245-270.
11. Coffman, Jr., E. and C. Courcoubetis, M. Garey, D. Johnson, L. McGeoch, P. Shor, R. Weber, M. Yannakakis, Fundamental discrepancies between average-case analyses under discrete and continuous distributions: A bin packing case study, STOC, 19991, p. 230-240.
12. Coffman, Jr., E. and G. Galambos, S. Martello, and D. Vigo, Bin Packing Approximation Algorithms: Combinatorial Analysis, in Handbook of Combinatorial Optimization, D. Du and P. Pardalos, (eds.), Kluwer, Amsterdam, 1998.
13. Coffman, Jr., E. and M. Garey, D. Johnson, Dynamic bin packing, SIAM J. Comput., 12 (1983) 227-258.
14. Coffman, Jr., E. and M. Garey, D. Johnson, Approximation Algorithms for Bin-Packing,: An updated survey, in Algorithm Design for Computer Systems Design, G. Ausiello, M. Lucertini, and P. Serafini, (eds.), Springer-Verlag, New York, 1984, 49-106.
15. Coffman, Jr., E. and M. Garey, D. Johnson, An application of bin-packing to multiprocessor scheduling, SIAM J. Comput., 7 (1987) 1-17.
16. Coffman, Jr., E. and M. Garey, D. Johnson, Approximation Algorithms for NP-Hard Problems, in D. Hochbaum, (ed.), Prindle Weber and Schmidt, Boston, 1996, p. 46-93.
17. Coffman, Jr., E. and M. Garey, D. Johnson, Bin Packing with divisible item sizes, J. Complexity, 3 (1987) 405-428.
18. Coffman, Jr., E. and G. Lueker, Probabilistic Analysis of Packing and Partition Algorithms, Wiley, New York, 1991.
19. Coffman, Jr., E. and G. Lueker, Approximation Algorithms for extensible bin packing, Proceedings, 12th Annual ACM-SIAM Symposium on Discrete Algorithms, 2001.
20. Coffman, Jr., E. and K. So, M. Hofri, A. Yao, A stochastic model of bin packing, Information and Control 44 (1980) 105-115.
21. Conway, R. and W. Maxwell, L. Miller, Theory of Scheduling, Addison-Wesley, Reading, 1967.
22. Courcoubetis, C. and R. Weber, Necessary and sufficient conditions for the stability of a bin packing system, J. Appl. Prob., 23 (1986) 989-999.
23. Csirik, J., The parametric behavior of the first-fit decreasing bin packing algorithm, J. Algorithms 15 (1993) 1-28.
24. Csirik, J. and J. Frenk, G. Galambos, A. Rinnooy Kan, Probabilistic analysis of algorithms for dual bin packing problems, J. Algorithms 12 (1991) 189-203.
25. Csirik, J. and D. Johnson, Bounded space on-line bin packing; best is better than first, In Proceedings, Second Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1991, p. 309-319.
26. Fernandez del la Vega, W. and G. Lueker, Bin packing can be solved in 1 + e in linear time, Combinatorica 1 (1981) 34-355.
27. Flexzar, K. and K. Hindi, New heuristics for one-dimensional bin packing, Computers and Operations Research 29 (1902) 821-839.
   Floyd, S. and R. Karp, FFD bin packing for items sizes with distribution on [0, 1/2], Algorithmica, 6 (1991) 222-240.
28. French, S., Sequencing and Scheduling, Wiley, New York, 1982.
29. Galambos, G. and G. Woeginger, An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling, SIAM J. Computing, 22 (1993) 349-355.
30. Garey, M. and R. Graham, D. Johnson, A. Yao, Resource constrained scheduling as generalized bin packing, J. Combinatorial Theory Ser. A, 21 (1976) 257-298.

31. Garey, M., and D. Johnson, Approximation algorithms for bin packing problems-A survey, in Analysis and Design of Algorithms in Combinatorial Optimization, G. Ausiello and M. Lucertini, (eds.)., Springer-Verlag, New York, 1981, p. 147-172.
32. Garey, M. and D. Johnson, A 71/60 theorem for bin packing, J. of Complexity, 1 (1985) 65-106.
33. Graham, R., Bounds for certain multiprocessing anomalies, Bell System Tech. J., 45 (1966) 1563-1581.
34. Graham, R., Bounds on multiprocessing anomalies, SIAM J. Applied Math., 17 (1969) 263-269.
35. Graham, R., Combinatorial Scheduling, in Mathematics Today, L. Steen, (Ed.), Springer-Verlag, New York, 1978, p. 183-211.
36. Graham, R., and E. Lawler, E. Lenstra, A. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Annals of Discrete Mathematics, 5 (1979) 287-326.
37. Gusfield, D., Bounds for naive multiple machine scheduling with release times and deadlines, J. of Algorithms, 5 (1984) 1-6.
38. Hofri, M., Probabilistic Analysis of Algorithms, Springer-Verlag, New York, 1987.
39. Jackson, J., Scheduling a production line to minimize maximum tardiness, Research Report 43, Management Science Research Project, UCLA, 1955.
40. Johnson, D., Near-Optimal Bin Packing Algorithms, Doctoral Thesis, MIT, Cambridge, 1973.
41. Johnson, D., Fast algorithms for bin packing, J. Comput. System Sci., 8 (1974) 272-314.
42. Johnson, D. and A Demers, J. Ullman, M. Garey, R. Graham, Worst-case performance bounds for simple one-dimensional packing algorithms, SIAM J. Comput., 3 (1974) 299-325.
43. Karger, D. and S. Phillips, E. Torng, A better algorithm for an ancient scheduling problem, In Proc. 5th ACM_SAIM Symposium on Discrete Algorithms, 1994, 132-140.
44. Karger, D. and C. Stein, J. Wein, Scheduling Algorithms, In Algorithms and Theory of Computation Handbook, M. Atallah, (ed.), CRC, Boca Raton, 1999, 35-1 - 35-33.
45. Karmarkar, N. and R. Karp, An efficient approximation scheme for the one-dimensional bin packing problem, Proc. 23rd Annual Symposium Foundation Computer Science (FOCS 1982), p. 312-320.
46. Katona, G., Edge disjoint polyp packing, Discrete Applied Mathematics 78 (1997) 133-152.
47. Kucera, L., Combinatorial Algorithms, Adam Hilger, Bristol, 1990.
48. Lawler, E., Optimal sequencing of a single machine subject to precedence constraints, Management Science, 19 (1973) 544-546.
49. Malkevitch, J. et al., For All Practical Purposes, (6th, and earlier editions) W. H. Freeman, New York, 2003.
50. Moore, J., An n-job, one machine sequencing algorithm for minimizing the number of late jobs, Management Science, 15 (1968) 102-109.
51. Parker, R., Deterministic Scheduling, Chapman-Hall, 1995.
52. Pinedo, M., Scheduling: Theory, Algorithms, Systems, Prentice-Hall, Upper Saddle River, 1995.
53. Shor, P., The average case analysis of some on-line algorithms for bin packing, Combinatorica 6 (1986) 179-200.
54. Shor, P., How to pack better than best-fit: Tight bounds for average-case on-line bin packing. In Proceedings, 32 Annual Symp. on Foundations of Computer Science, New York, 1991, 752-759.
55. Simchi-Levi, D., New worst-case results for the bin packing problem, Naval Res. Log., 4 (1994(579-585.
56. Xu, K., A Bin-Packing Problem with Item Sizes in the Interval (o, a] for a ≤1/2, Doctoral Thesis, Chinese Academy of Sciences, Beijing, China, 1993.
57. Yao, A., New algorithms for bin packing. J. Assoc. Comput. Mach., 22 (1980) 207-227.
58. Yue, M. A simple proof of the inequality FFD(L) ≤(11/9)OPT(L) + 1, for all L, for the FFD bin-packing algorithm, Acta. Math. Appl. Sinica 7 (1991) 321-331.
59. Yue, M., On the exact upper bound for the multifit processor scheduling algorithm, Ann. Oper. Res., 24 (1990) 233-260.

One-Dimensional Bin Packing Problem

One-dimensional bin packing problem can be described as follows:

Given a bin capacity $C > 0$ and a list of objects $\{p_1, p_2, \ldots, p_n\}$, what is the smallest number of bins needed to accommodate all of the objects?

(Each $p_i$has size s_i, such that $0 \leq s_i \leq C$. i.e. none of the objects too big to fit in a bin)

One-Dimensional Bin Packing Problem

Four easily recognisable one-dimensional bin packing problems:

- A material such as piping or cable is supplied in quantities of a standard length *C*. The demands for pieces of the material are for varying lengths that do not exceed *C*. The idea is to use the least number of standard lengths of the material in producing a given set of required pieces. Hence minimising the wastage.
- Advertisements of arbitrary lengths, must be assigned to commercial break time slots on television. Each break must last no longer than three minutes.
- Removal lorries with set weight limits are to be packed with furniture. The aim is to use as few lorries as possible to pack all the items, without exceeding the maximum weight in any lorry.
- A set of tasks with known, arbitrary execution times, need to be executed on some identical processors by a given deadline. The problem is to schedule all of the tasks onto the least number of machines so that the deadline is met.

In the last example, time is the critical resource. It is a scheduling problem; the processors are the bins, the deadline is the common bin capacity, and the individual execution times of the tasks are the objects (p_i) in the list.

Two-Dimensional Bin Packing Problem

This problem is concerned with packing different sized objects (most commonly rectangles) into fixed sized, two-dimensional bins, using as few of the bins as possible.

Stock cutting example

- quantities of material such as glass or metal, are produced in standard sized, rectangular sheets.
- Demands for pieces of the material are for rectangles of arbitrary sizes,
- no bigger than the sheet itself.
- The problem is to use the minimum number of standard sized sheets in accommodating a given list of required pieces.

A variation on the two-dimensional bin packing problem is the strip packing problem.

Strip Packing Problem

The problem

- packing a set of *n* rectangles into an open-ended bin of
  - fixed width *C* and
  - infinite height.
  - the rectangles must not overlap each other
  - The idea is to pack the rectangles in a way that minimises the overall height of the bin.

The clear difference between this problem and the two-dimensional bin packing problem, is that there is only one bin, so instead of trying to minimise the number of bins used, the aim is to minimise the height of the single bin.

All the rectangles must be packed orthogonally. They cannot be rotated and they must be packed with their width parallel to the bottom of the bin.

The rectangles correspond to a set of tasks,

- heights being the amount of processing time they require
- widths the amount of contiguous memory (processors) they need.
- The width of the bin corresponds to the amount of memory (procs) available
- The aim is to schedule all of the tasks so that they are completed in the least possible time

Strip Packing Problem

Another common application of the strip packing problem is stock cutting.

- Material such as cloth, paper, or sheet metal, comes in rolls of a set width.
- These rolls may need to be cut into rectangles of arbitrary widths and heights.
- The goal is then to cut out all the required rectangles from the shortest length of roll possible, so minimising the wastage.
- the roll of material corresponds to the bin.

Strip Packing Algorithms

The problem of finding an optimal solution for the strip packing problem is NP-complete,

- Approximation algorithms find near optimal solutions but do not guarantee to find the optimal packing for every set of data.
- Most algorithms pack the rectangles into the bin using one of five approaches:
  - bottom left,
  - level-orientated,
  - split,
  - shelf or
  - hybrid.

Strip Packing Algorithms

Bottom-left algorithm

- rectangle to be placed as near to the bottom of the bin as it will fit
- then as far to the left as it can go at that level without overlapping any other packed rectangles.
- list of rectangles require no pre-sorting

Level-oriented algorithms

- The list of rectangles are pre-sorted into order of decreasing height.
- the packing is done on a series of levels, that the bottom of each rectangle rests on.
- The first level is the bottom of the bin and subsequent levels are defined by the height of the tallest rectangle on the previous level.

Strip Packing Algorithms

Split algorithms

The level oriented algorithms split the bin horizontally into blocks of a set width.

Split algorithms split the open ended bin vertically into smaller open ended bins depending on the widths of the rectangles.

The rectangles are first sorted by width.

Shelf algorithms

modifications of the level algorithms, that avoid pre-sorting the list of rectangles.

Rather than being determined by the tallest rectangle, the levels are fixed height shelves.

The shelf heights are set by a parameter *r*. $(0 < r < 1)$

Strip Packing Algorithms

Hybrid algorithms

These use the characteristics of two or more of the types of algorithms described above. They may or may not involve pre-sorting.

Strip Packing Algorithms

The Slave Algorithm

A slave algorithm is the approach that is used to decide which shelf a rectangle should be put on once all appropriate shelves have been determined.

For the level-by-level algorithms, each level, or shelf, in the bin has exactly the same width, *C*. (the width of the bin itself)

This means that, for each rectangle, once the algorithm has calculated the levels it may be placed on, the decision of which of these levels to use is exactly a one-dimensional bin packing problem.

That is, the number of levels being used, corresponds to the number of one-dimensional bins, of capacity *C*, that are required.

This one-dimensional problem is solved by the slave algorithm.

Strip Packing Algorithms

There are many one-dimensional algorithms that could be used as the slave algorithm, for instance: Next-fit (NF) and First-fit (FF) algorithms.

Next-fit v's First-fit

Strip Packing Algorithms

Next-fit algorithm

- each rectangle is put onto the highest level on which it will fit, i.e. the current level of the required specifications.
- no backtracking is allowed.
- For example, if the height of the bin corresponded to time, and the time was continuously ticking by, you would not be able to go back in time and schedule any jobs for earlier points on the bin since that time has already passed!
- if the bin is a strip of material, going along a conveyor belt. The rectangles get put into position on the material as it precedes, and at the end of the conveyor they get cut out.
- In these circumstances the levels lower down the bin (closer to the cutters), are continuously getting cut up so no more rectangles could be put on these levels as they would have missed the cutting process.

Strip Packing Algorithms

First-fit algorithm

- each rectangle is put onto the lowest (first) level that it will fit on.
- most economical choice since it allows to place the rectangles on all the levels lower down the bin
- suitable to use so long as all the rectangles may be arranged in the bin before anything happens to it, e.g. before the jobs are executed, or the strip starts to be cut up. Then it is fine to place the rectangles anywhere on the bin.

Strip Packing Algorithms

On-line v's Off-line

- whether or not the rectangles required sorting before being placed into the bin.
- This is an important factor to take into consideration when deciding on an appropriate algorithm for a bin packing problem.
- Consider the situation where the list of rectangles arrives one at a time.
- When each rectangle arrives it must immediately be assigned its place in the bin.
- Only once this has happened, does the identity of the next rectangle become known.
- This type of environment is called on-line.
- On-line algorithms must assume no prior knowledge of the rectangles in the list, so pre-sorting the list of rectangles is not an option.

Strip Packing Algorithms

off-line environment

- it is possible to view the whole list of rectangles first and so sort them into any order before they are placed on the bin.
- Off-line algorithms assume prior knowledge of the whole problem before any packing has to be done.

An on-line algorithm would be used in a situation where the jobs had to be done in order. For example, the items may be subject to different priorities or deadlines.

An off-line algorithm would be used when the order did not matter. For example, if pieces of material were being cut out to make a jacket, it would not matter if a sleeve was cut out before the collar, just as long as all the required pieces were produced in the end.

Strip Packing Algorithms

Guillotine Cuts

Level algorithms and shelf algorithms are both level-by-level packing approaches.

- use in relation to guillotine cuts. (Level-by-level packings involve every rectangle being completely inside or completely outside of each level.
- This means that every vertical line through a level, intersects at most one rectangle)
- Guillotine cuts are cuts from one edge, across to the other edge of a rectangular section of bin, either parallel to its height or its width. I.e. they cut the area of rectangle into into two smaller rectangles, since the cuts must go all the way from one edge to the other.
- Level-by-level packings allow for 3-stage guillotine cuts. These involve first some horizontal guillotine cuts, then some vertical guillotine cuts and lastly, some more horizontal guillotine cuts to trim the rectangles to size.

The most common need for the ability to perform the 3-stage guillotine cuts is for its use in stock-cutting. In a factory situation, the cutting machines are generally set up either perpendicular or parallel to the strip of material (glass, sheet metal, etc) and must cut from one edge to the other.

Strip Packing Algorithms

Level Algorithms and Shelf Algorithms

You can view demonstrations of all the algorithms explained below by clicking on the one you are interested in

* Next-Fit Decreasing Height * First-Fit Decreasing Height * Next-Fit Shelf * First-Fit Shelf *

The performance of all the demonstrations is assessed, and displayed on the screen after all of the rectangles have been placed in the bin. This is shown as the percentage of wasted space (space not filled by a rectangle) in the bin. It is calculated by finding the ratio of the total area of all the rectangles in the list, and the area of bin used to pack them, (the area below the top of the highest rectangle).

Level Algorithms

Level algorithms are off-line algorithms. Their first step involves pre-sorting the list of rectangles into order of decreasing heights, i.e. the first rectangle to be packed will be the tallest and the last, the shortest. The packing is then made

up of a series of levels. Each rectangle is successively packed into the bin by placing its bottom edge so as it rests on one of the levels. The first level is the bottom of the bin and each new level is defined by drawing a horizontal line across the bin through the top of the tallest (i.e. first) rectangle on the previous level.

Next-fit decreasing height (NFDH) algorithm

The NFDH algorithm is a level algorithm which uses the next-fit approach to pack the sorted list of rectangles. The rectangles are packed, left-justified on a level until the next rectangle will not fit. This rectangle is used to define a new level and the packing continues on this level. The earlier levels are not revisited.

To view a demonstration of the NFDH algorithm click here

First-fit decreasing height (FFDH) algorithm

This is another level algorithm which this time uses the first-fit approach. Each rectangle is placed on the first (i.e. lowest) level on which it will fit. If none of the current levels have room, a new level is started.

To view a demonstration of the FFDH algorithm click here

Shelf Algorithms

Shelf algorithms are variants of the level algorithms which avoid pre-sorting the list of rectangles. Therefore, unlike the level algorithms, shelf algorithms are on-line. To achieve this, the levels, rather than being determined by their tallest rectangle, come in fixed sized shelves, whose heights are determined by a parameter *r*, (0 < *r* < 1). Each arriving rectangle is classified according to its height. Then, the packing is constructed as a series of shelves, with rectangles of similar heights being packed onto the same shelves.

Shelf sizes come in the form r^k. Each rectangle of height *h*, is packed onto a shelf having height r^k, which satisfies the equation,

$$r^{k+1} \leq h \leq r^k$$

for some integer value *k*. The parameter *r* is pre-specified, and its aim is to limit the amount of wasted space allowed on each shelf. You are able to vary *r* in the demonstration applets to see how it affects the packing.

For small *r*, (*r* approximately equal to zero), the range of heights allowed on each shelf is large. This would often be required if there was only a small number of rectangles to pack, as you would not want every rectangle on its own individual shelf with the shelves not getting full up. It would also be useful if the list of rectangles had a wide selection of heights. In general, for small *r*, there would be a fewer amount of shelf heights to choose from, with each shelf containing rectangles with a larger range of heights.

For large *r*, (*r* approximately equal to one), each shelf would contain only rectangles that were very similar in height. This may be used if the list of rectangles was very long, because it would mean the likelihood of each different sized shelf becoming full up, would be increased. Another reason for having a large *r*, would be if the rectangles in the list, all had similar heights, so the algorithm would distinguish between the smaller differences in height. Generally, a large *r* would produce a wider selection of shelf heights, with the rectangles on each shelf having very similar heights.

Next-fit shelf (NFS) algorithm

Using the next-fit approach, this algorithm packs each rectangle as far to the left as possible on the highest shelf that has the required height. (The currently active shelf of this height) If there is no room on this shelf, or there is not yet a shelf of this height in the packing, a new such shelf is created and becomes the currently active shelf for that height. Shelves of the same height, below the active shelf are not revisited.

To view a demonstration of the NFS algorithm click here

First-fit shelf (FFS) algorithm

With the first-fit approach, the lowermost shelf of the correct height, onto which the rectangle will fit, is chosen. If there is no shelf of the required height, or none of the appropriate shelves have sufficient room, then a new shelf of that height is created.

To view a demonstration of the FFS algorithm click here

http://users.cs.cf.ac.uk/C.L.Mumford/heidi/BinPacking.html

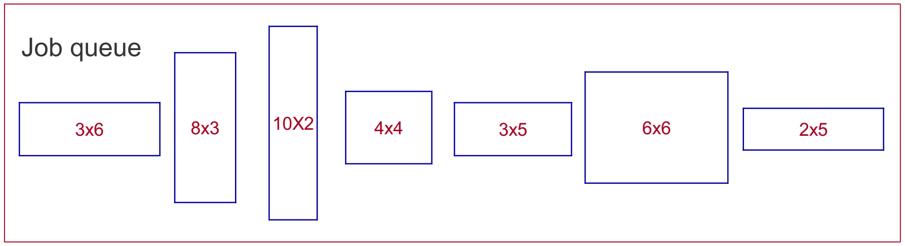
http://www.um.es/estructura/equipo/vic-estudiantes/arquimedes2003/pdf/015-JesusBeltran-EduardoCalderon.pdf

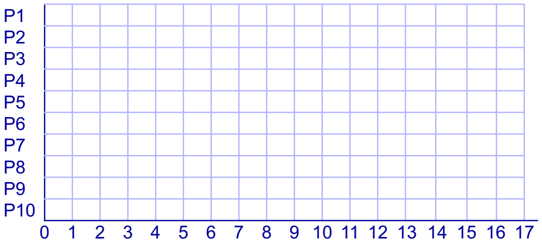
Topic 5
Backfill

© 2022 A. Tchernykh. Scheduling

FIFO

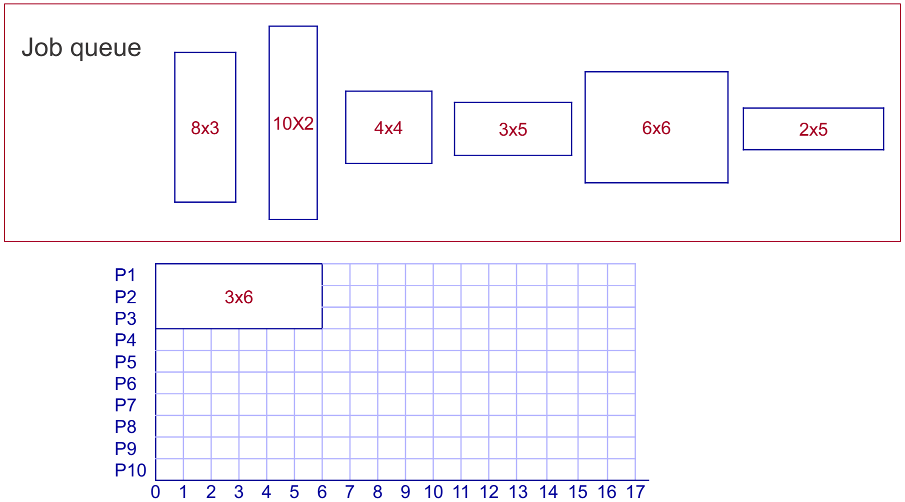
Example 1:





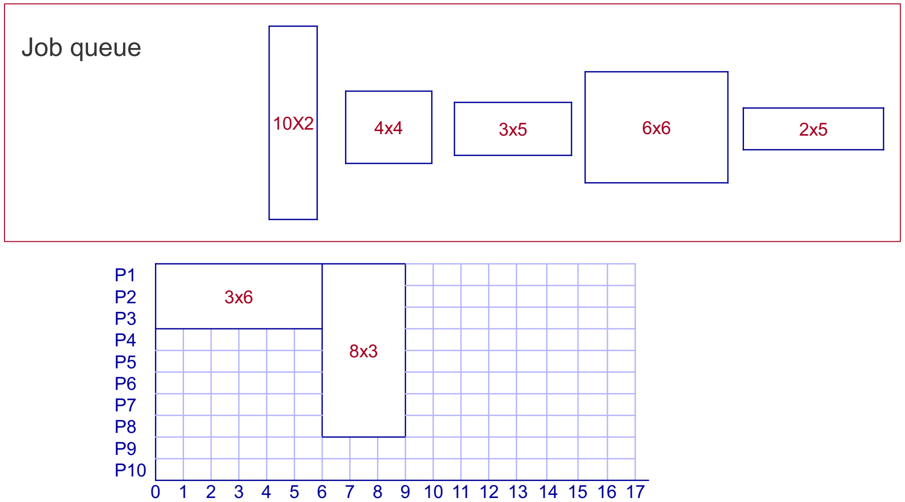
FIFO

Example 1:



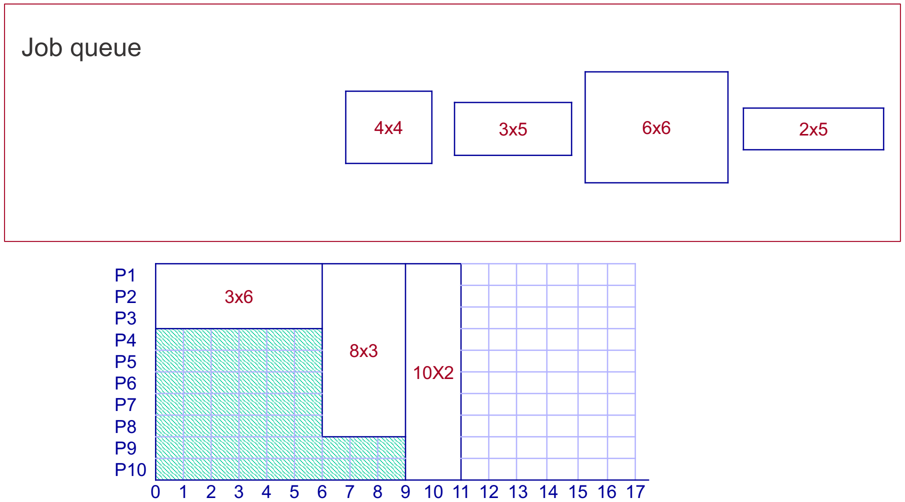
FIFO

Example 1:



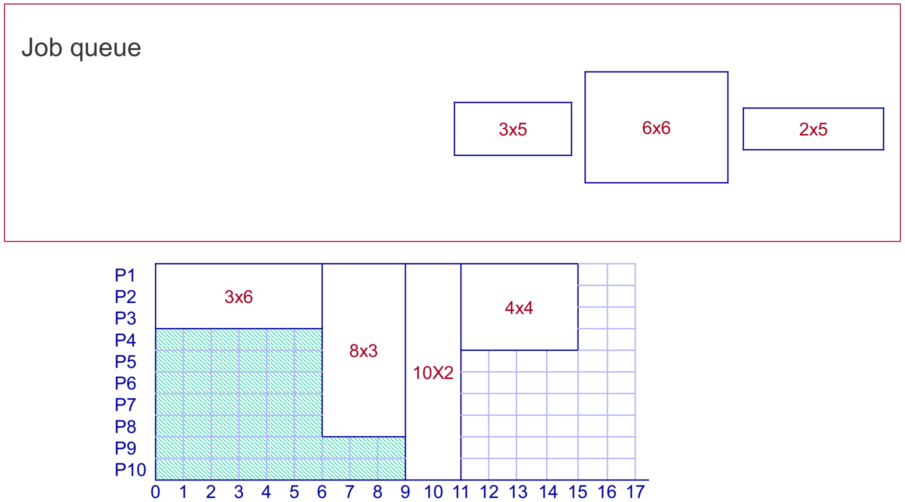
FIFO

Example 1:



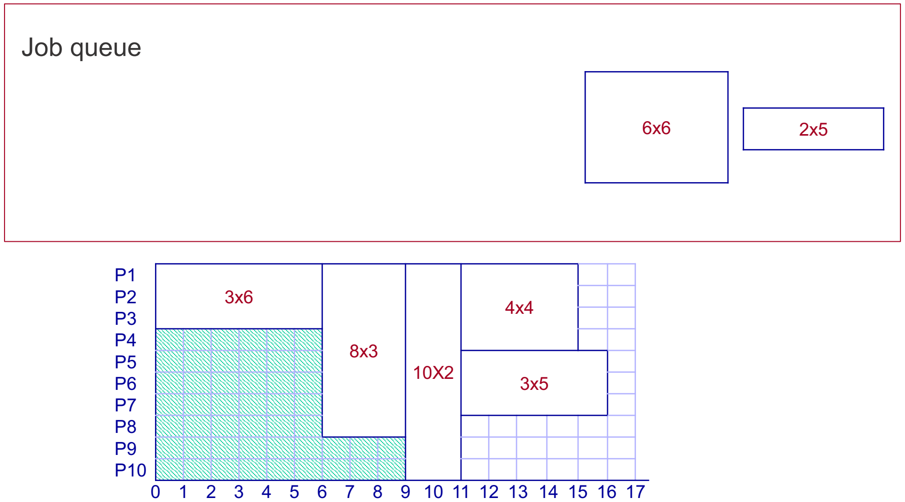
FIFO

Example 1:



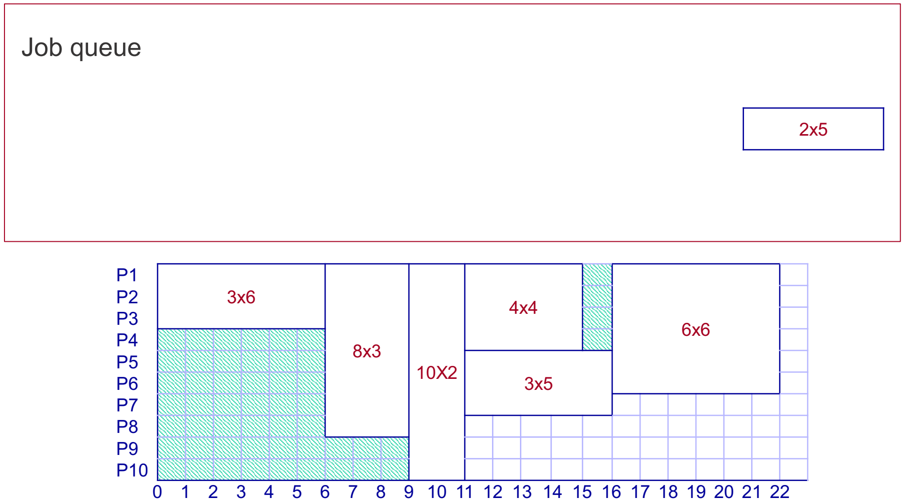
FIFO

Example 1:



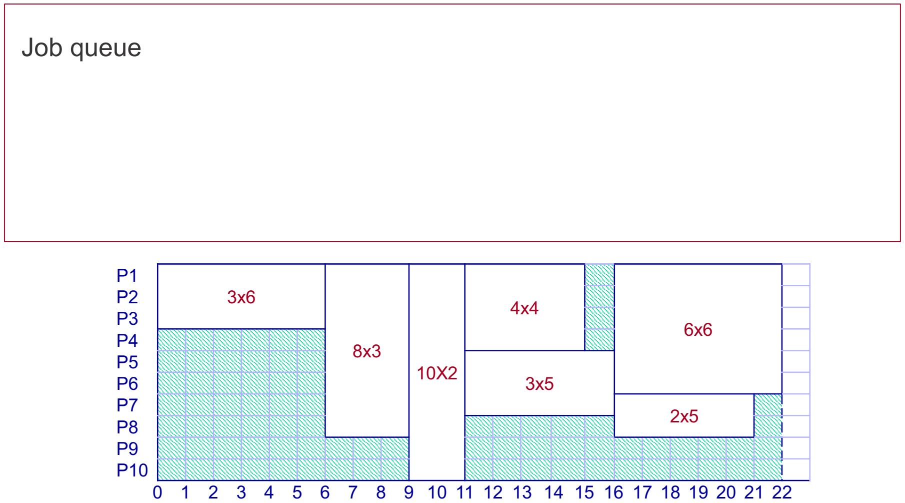
FIFO

Example 1:



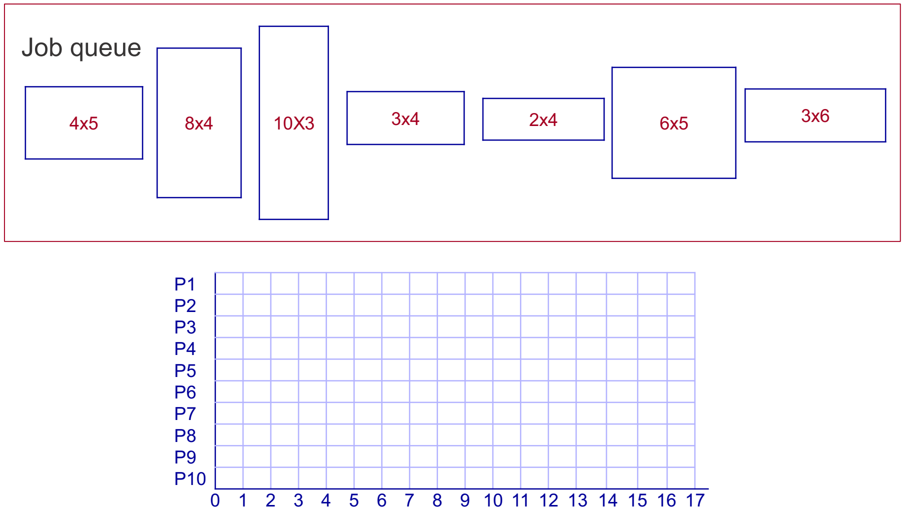
FIFO

Example 1:



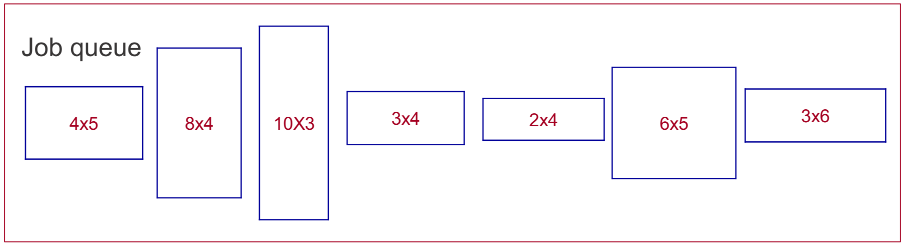
FIFO

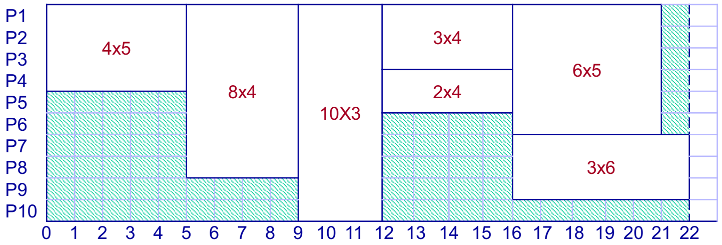
Exercise 1:



FIFO

Example 1:

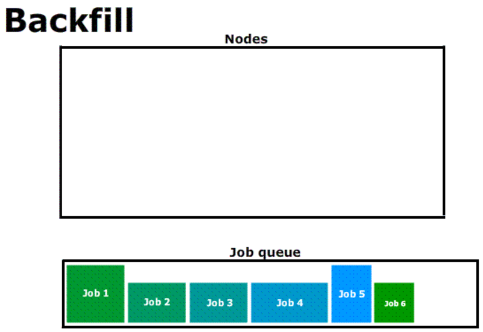




FIRSTFIT

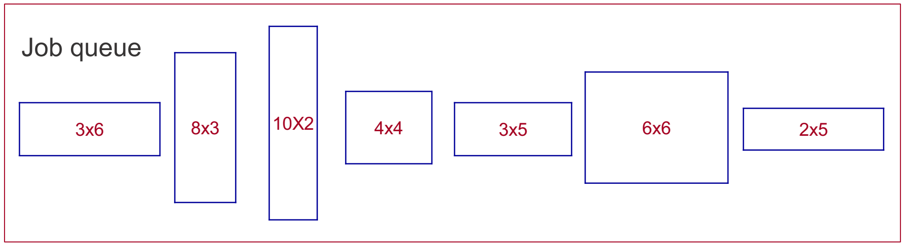
Assuming the FIRSTFIT algorithm is applied, the following steps are taken:

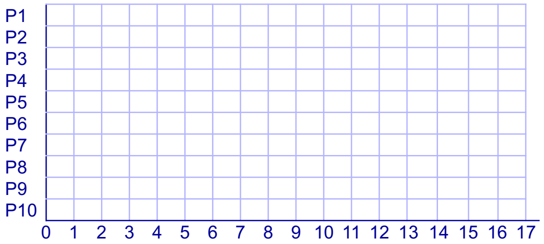
1. The list of feasible backfill jobs is filtered, selecting only those which will actually fit in the current backfill window.
2. The first job is started.
3. While backfill jobs and idle resources remain, repeat step 1.



FIRSTFIT

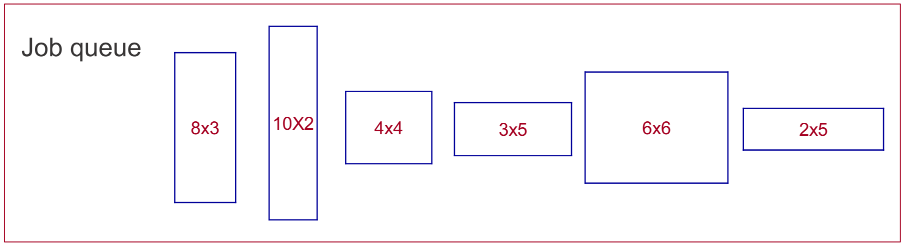
Example 2:

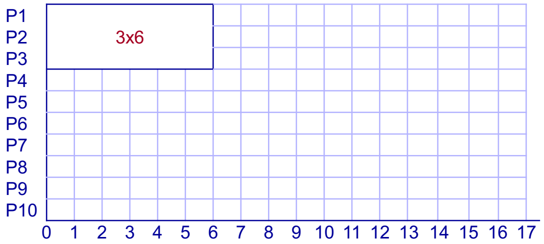




FIRSTFIT

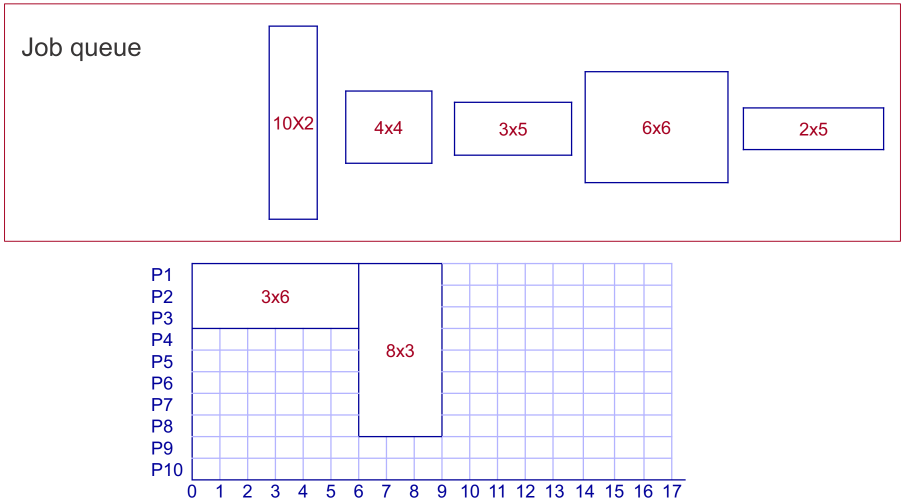
Example 2:





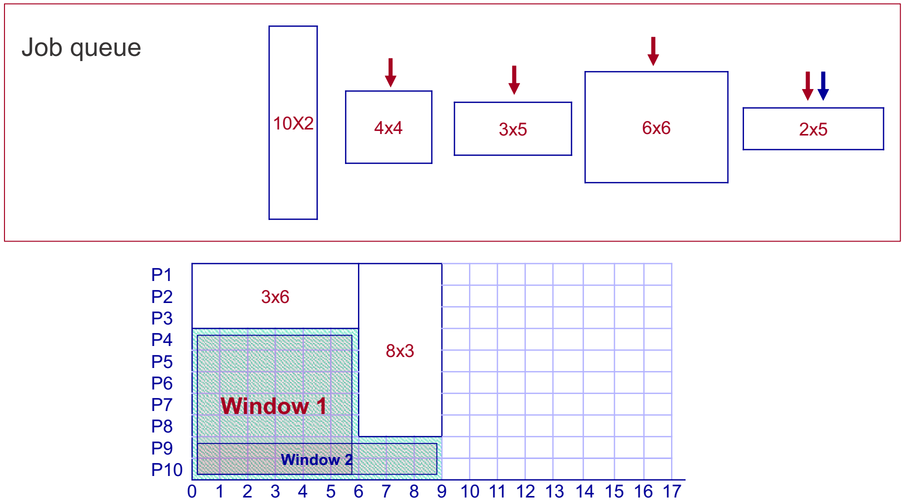
FIRSTFIT

Example 2:



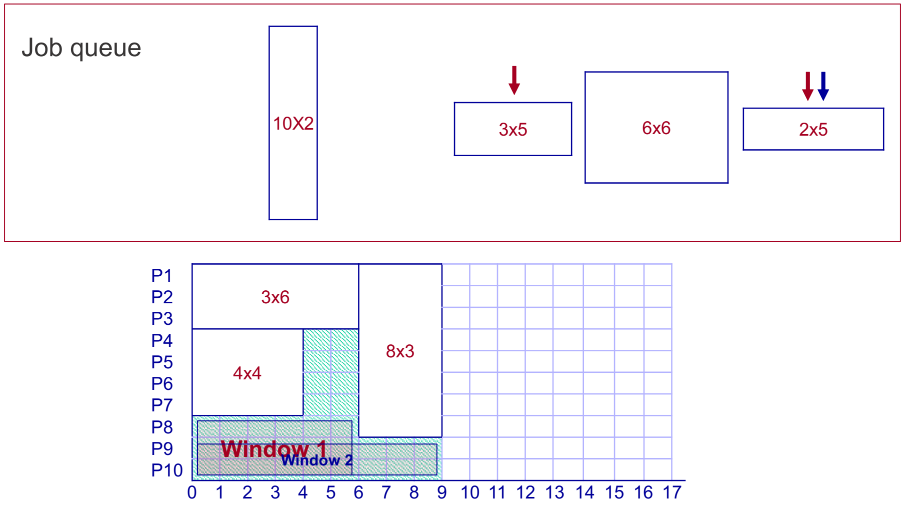
FIRSTFIT

Example 2:



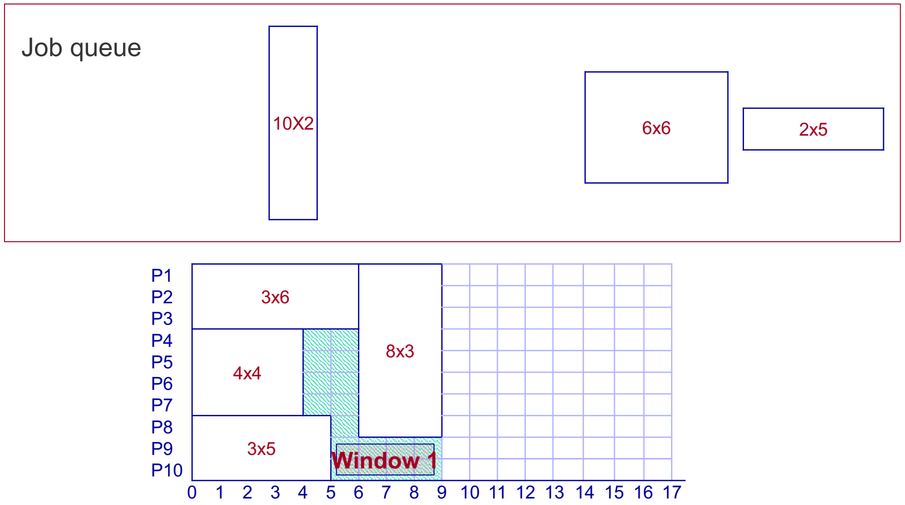
FIRSTFIT

Example 2:



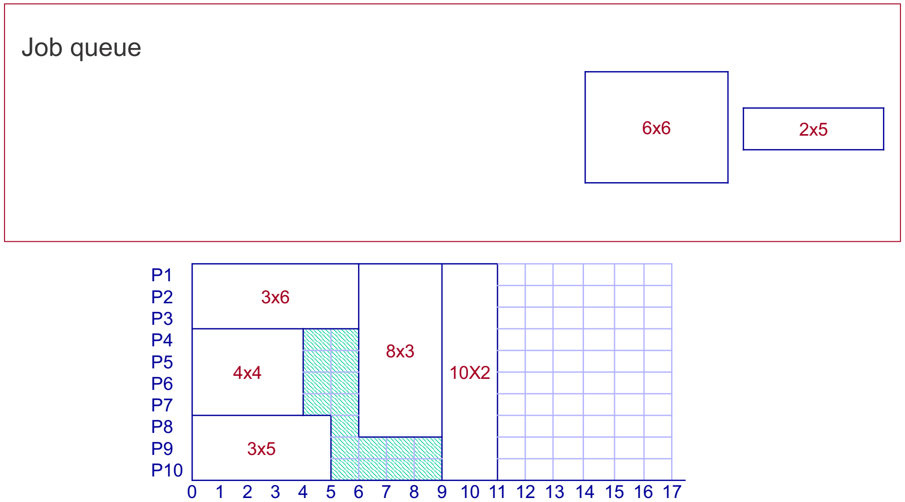
FIRSTFIT

Example 2:



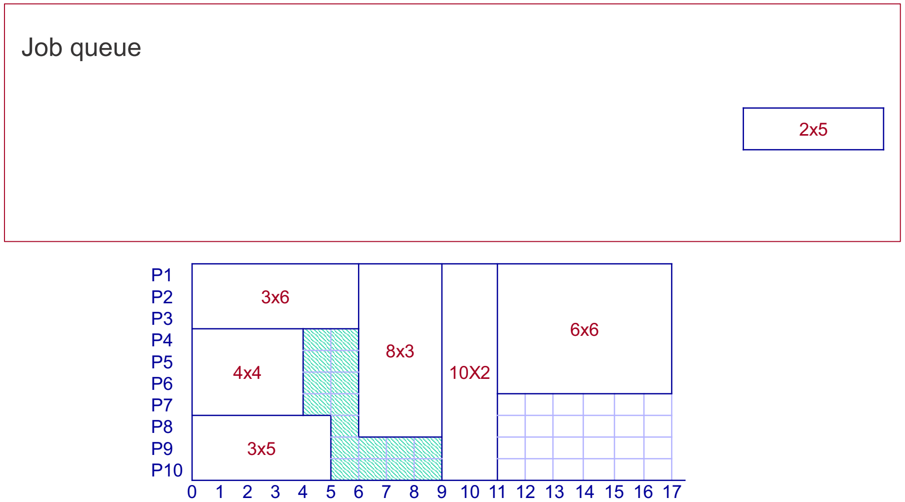
FIRSTFIT

Example 2:



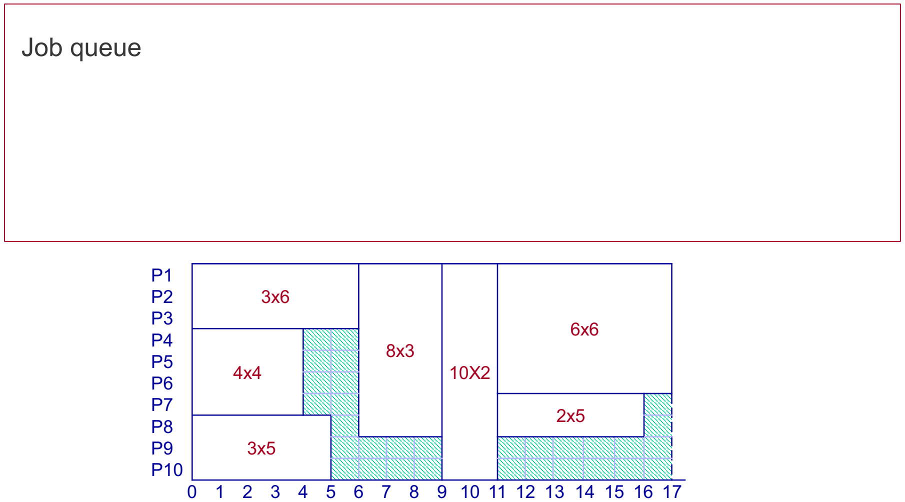
FIRSTFIT

Example 2:



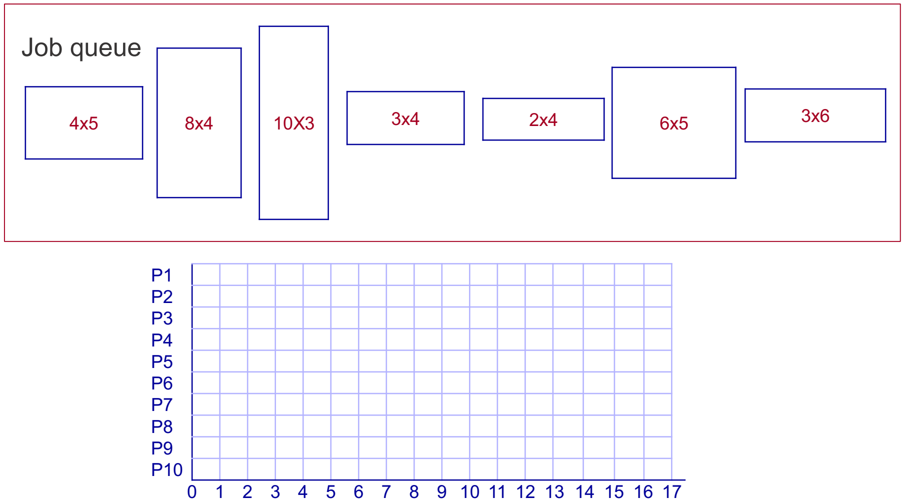
FIRSTFIT

Example 2:



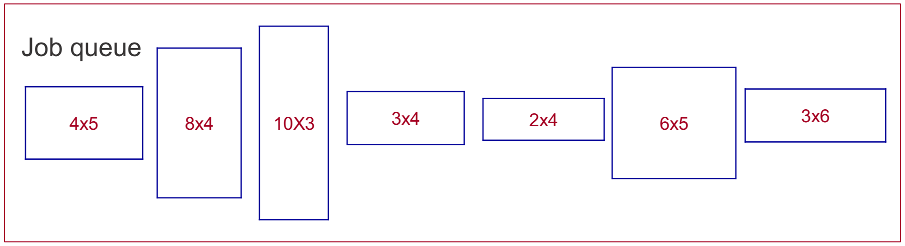
FIRSTFIT

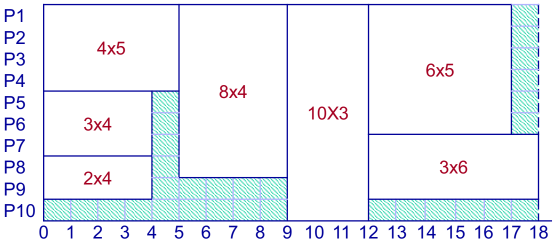
Exercise 2:



FIRSTFIT

Exercise 2:



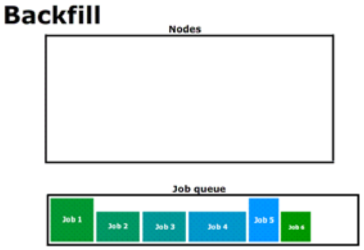


BESTFIT

Assuming the BESTFIT algorithm is applied, the following steps are taken:

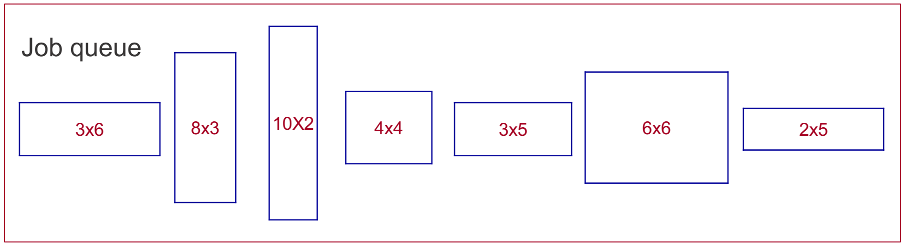
1. The list of feasible backfill jobs is filtered, selecting only those which will actually fit in the current backfill window.
2. The *degree of fit* of each job is determined based on the ***SCHEDULINGCRITERIA*** parameter (i.e., processors, seconds, processor-seconds, etc)
3. The job with the best fit is started.

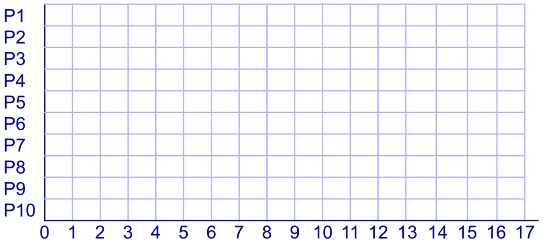
While backfill jobs and idle resources remain, repeat step 1.



BESTFIT

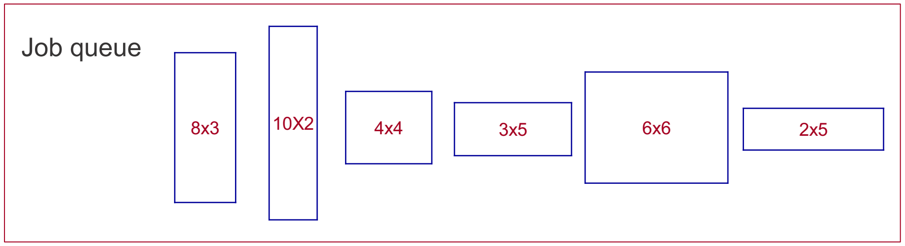
Example 3:

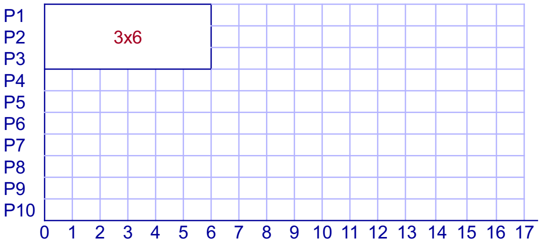




BESTFIT

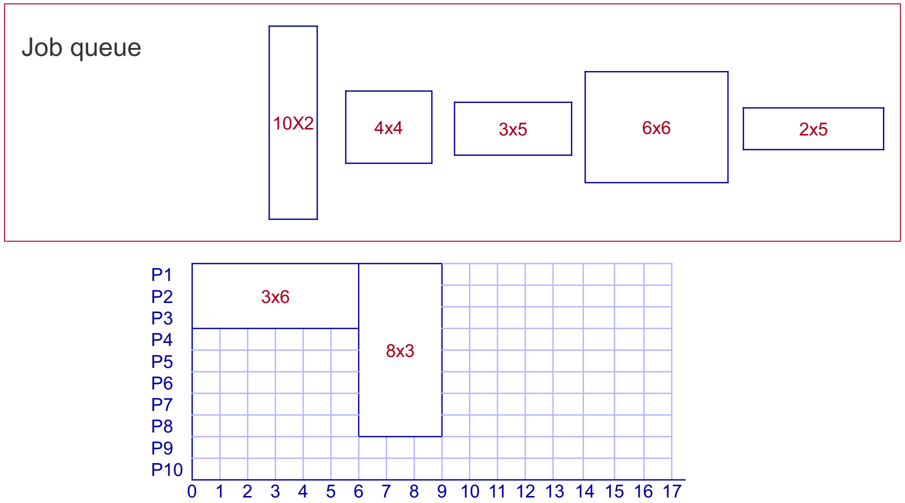
Example 3:





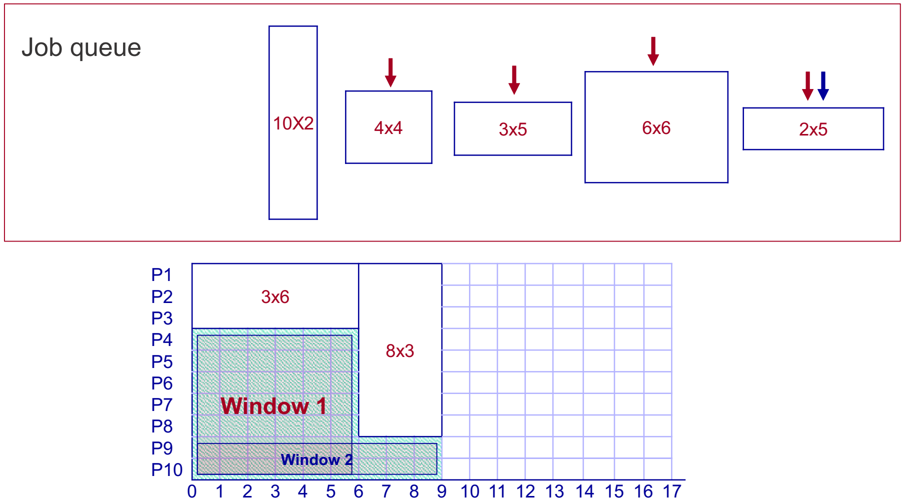
BESTFIT

Example 3:



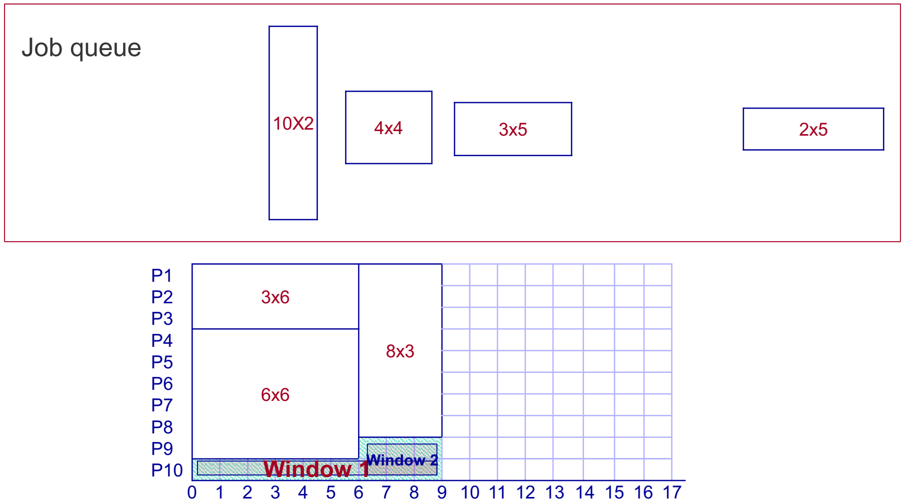
BESTFIT

Example 3:



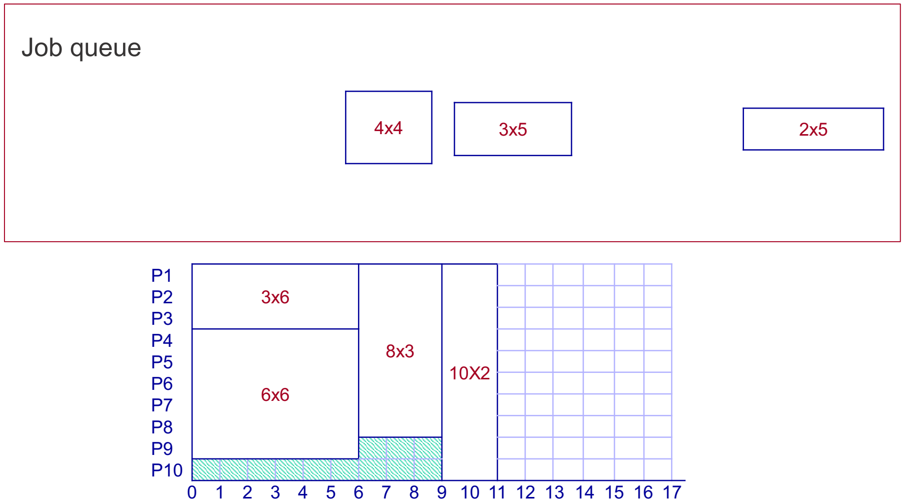
BESTFIT

Example 3:



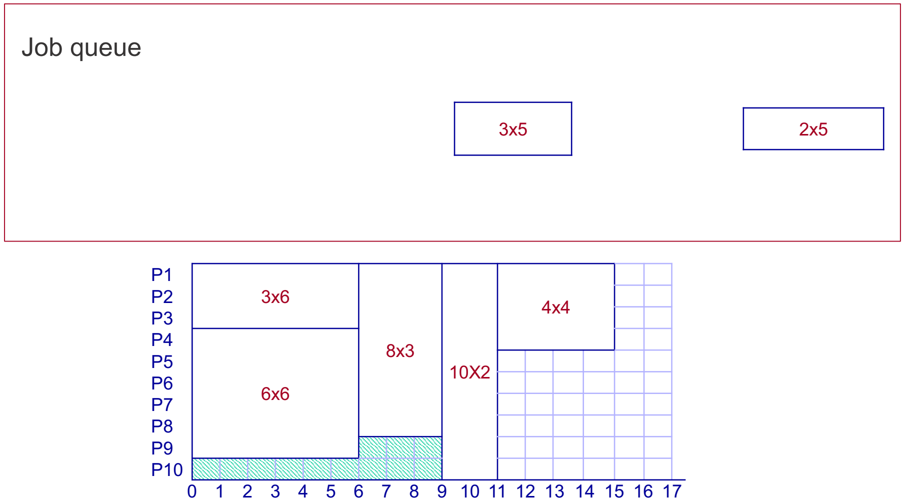
BESTFIT

Example 3:



BESTFIT

Example 3:



Job queue

3x5

2x5

3x6

6x6

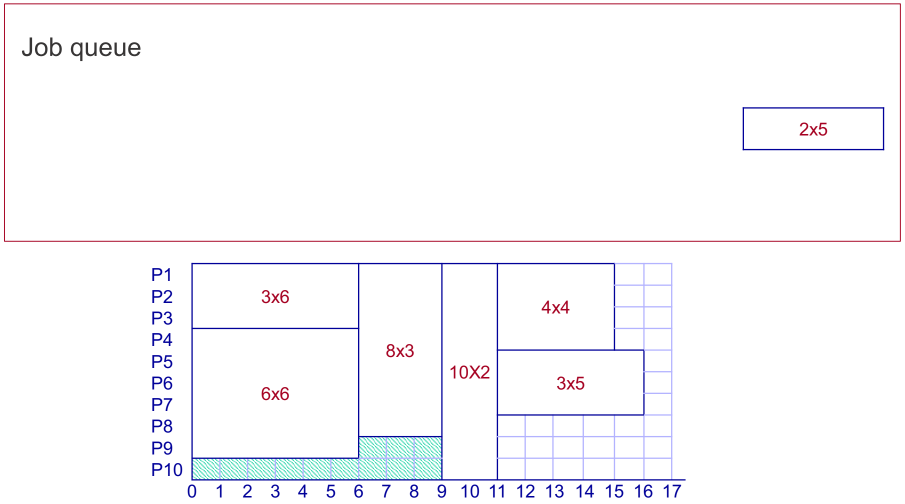
8x3

10X2

4x4

BESTFIT

Example 3:



Job queue

2x5

3x6

6x6

8x3

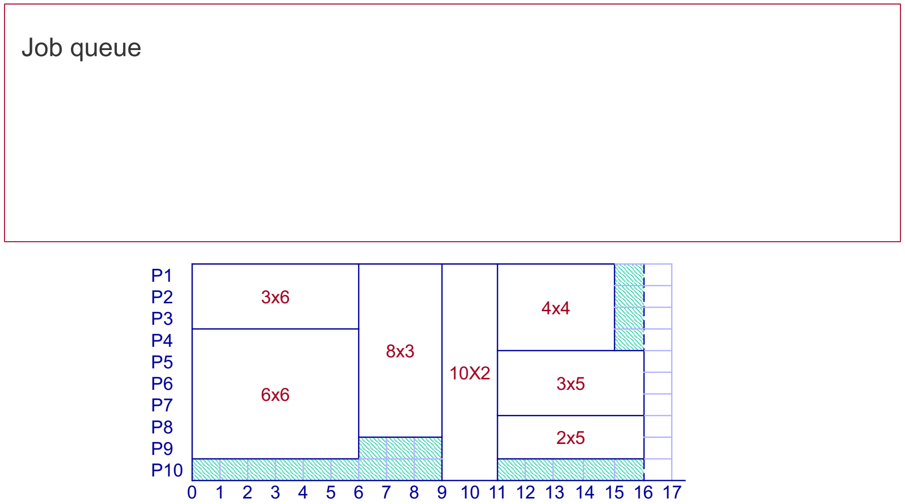
10X2

4x4

3x5

BESTFIT

Example 3:



Job queue

3x6

6x6

8x3

10X2

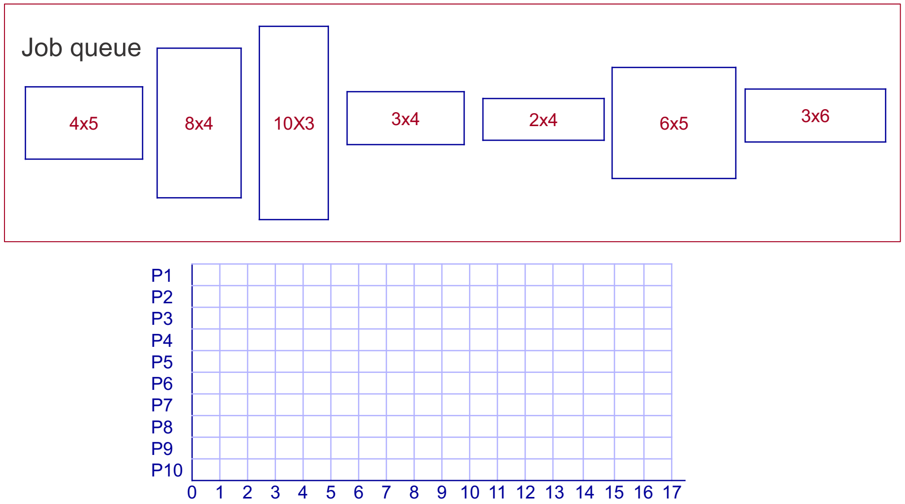
4x4

3x5

2x5

BESTFIT

Exercise 3:



Job queue

4x5

8x4

10X3

3x4

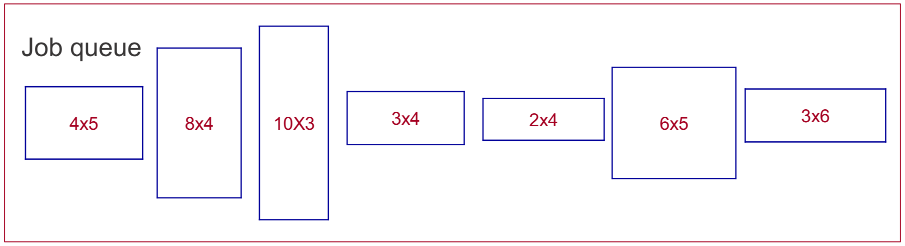
2x4

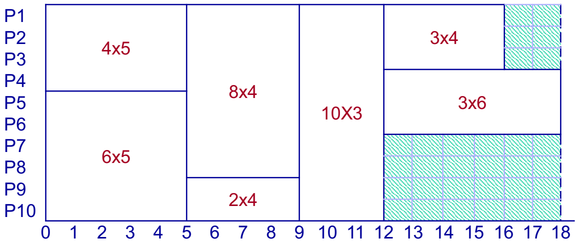
6x5

3x6

BESTFIT

Exercise 3:

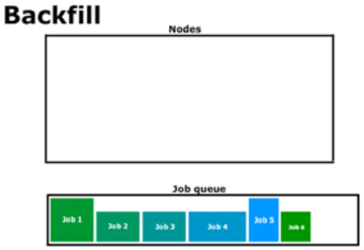




GREEDY

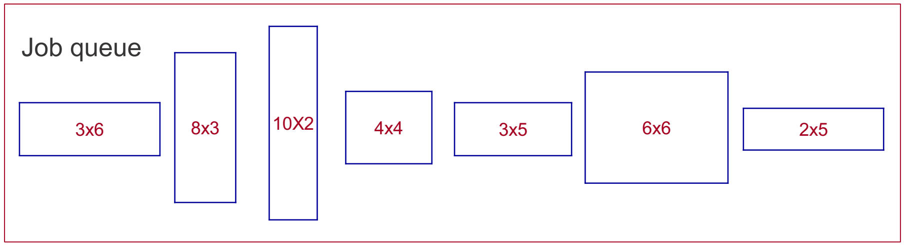
Assuming the GREEDY algorithm is applied, the following steps are taken:

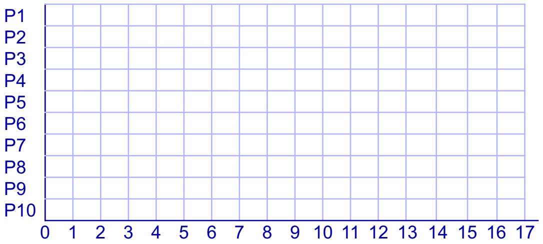
1. The list of feasible backfill jobs is filtered, selecting only those which will actually fit in the current backfill window.
2. All possible combinations of jobs are evaluated, and the degree of fit of each combination is determined based on the ***SCHEDULINGCRITERIA*** parameter (i.e., processors, seconds, processor-seconds, etc)
3. Each job in the combination with the best fit is started.
4. While backfill jobs and idle resources remain, repeat step 1.



GREEDY

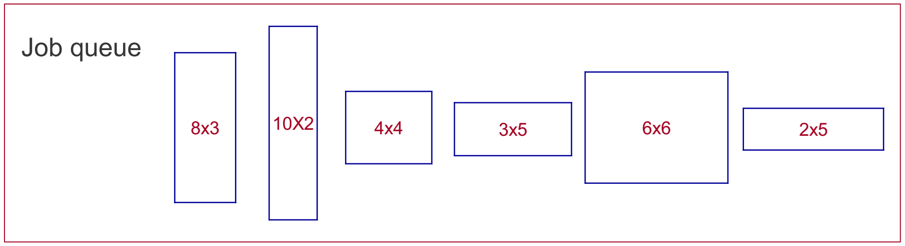
Example 4:

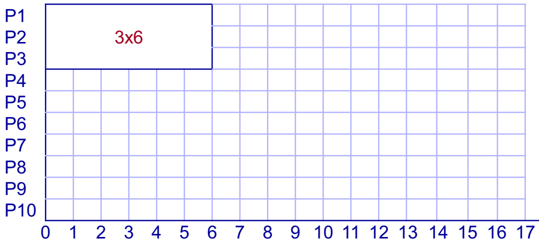




GREEDY

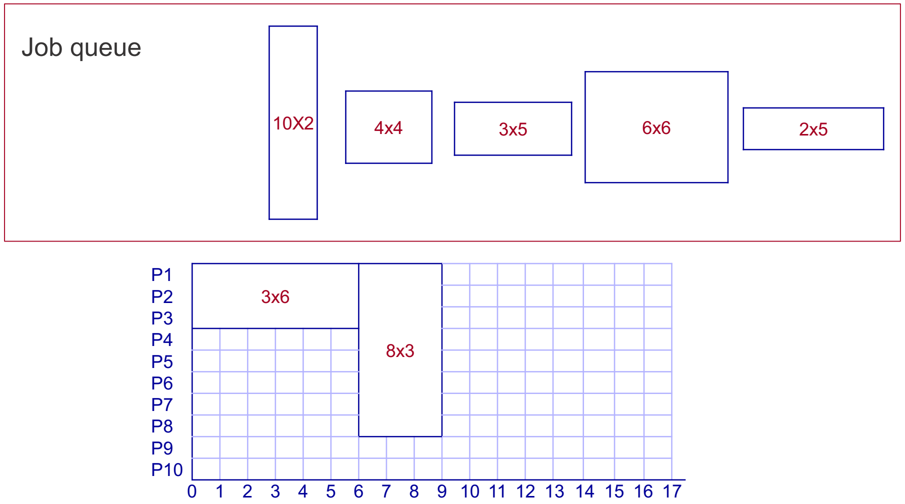
Example 4:





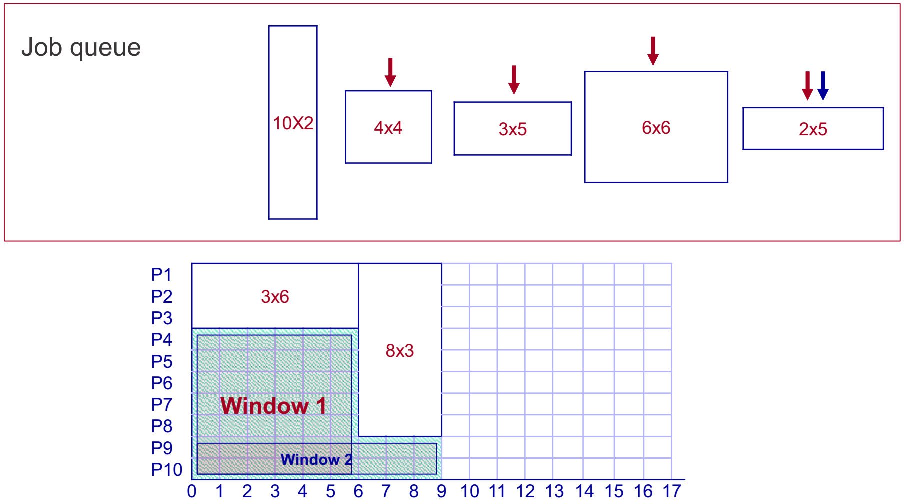
GREEDY

Example 4:



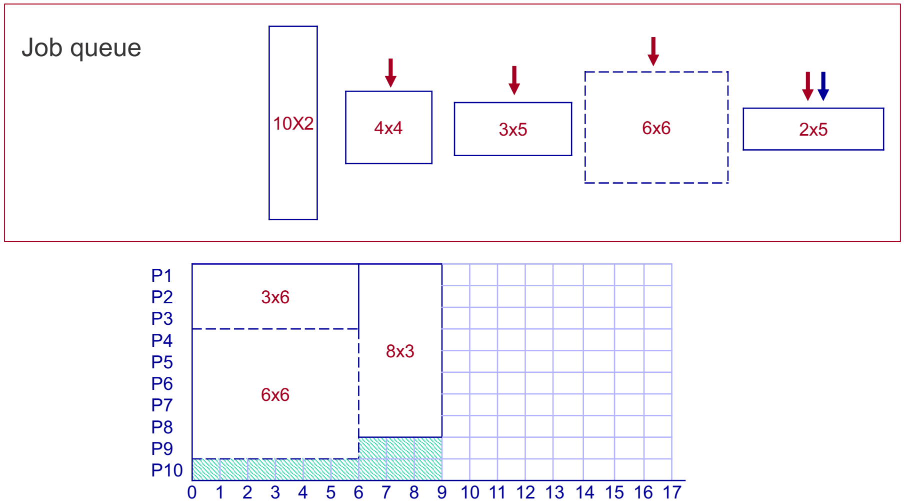
GREEDY

Example 4:



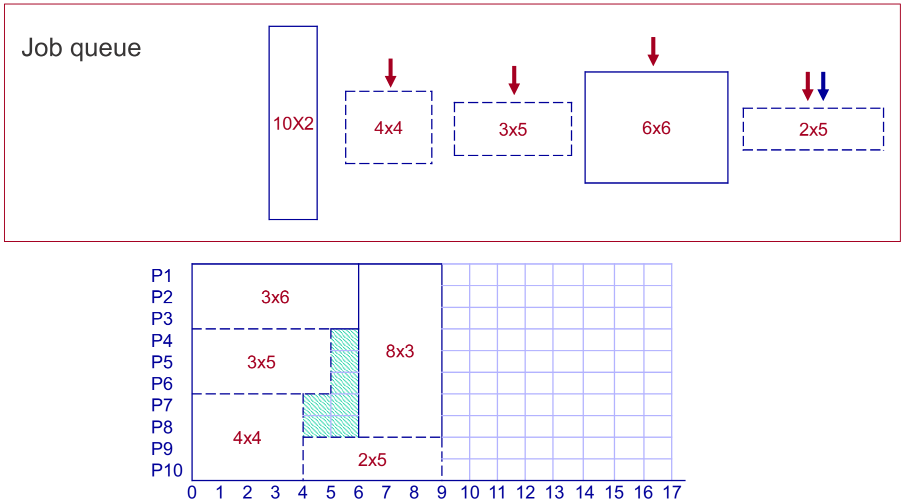
GREEDY

Example 4:



GREEDY

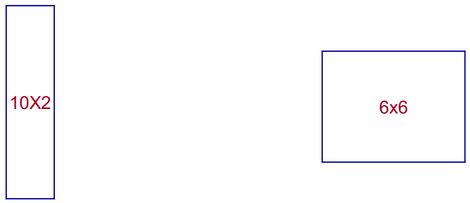
Example 4:

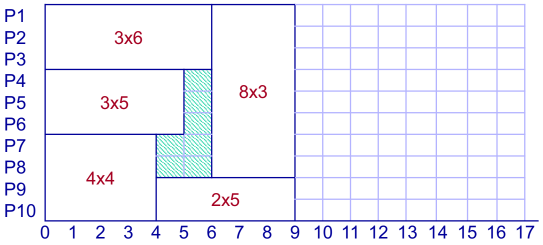


GREEDY

Example 4:

Job queue

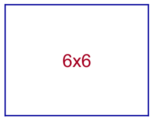


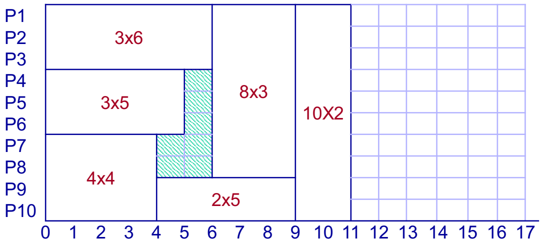


GREEDY

Example 4:

Job queue

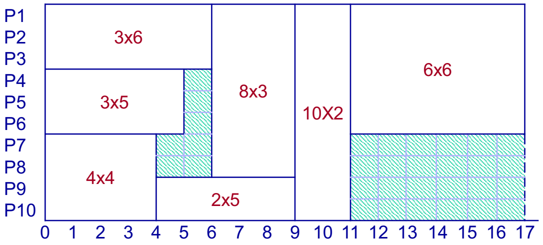




GREEDY

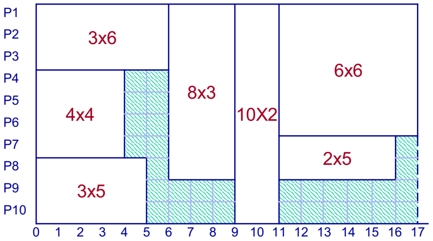
Example 4:

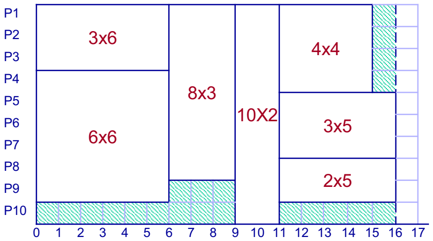
Job queue

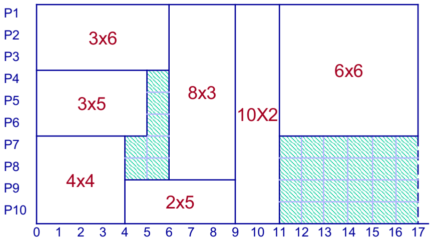


FIRSTFIT, BESTFIT and GREEDY

Examples 2, 3, and 4







3. Foundation of Parallel Computing

Von Neumann Architecture

1 For over 70 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.
2 A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

3 Basic design:
- Memory is used to store both program and data instructions
- Program instructions are coded data which tell the computer to do something
- Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then **sequentially** performs them.

Taxonomies of parallel machines

There are different ways to classify parallel computers.

Most popular classifications are based on
1 **Control** and **data** streams (Flynn taxonomy);
2 **Memory allocation** (shared memory, distributed memory, distributed-shared memory);
3 Interconnection
  ○ static (mesh, hypercube, tree,...);
  ○ dynamic (bus, crossbar, multistage network, ...)
4 **Distribution in space** (centralized and distributed)
5 **Memory access**
  ○ symmetrical multiprocessor, UMA
  ○ asymmetric multiprocessor, NUMA.
6 **Communication**
  o Loosly coupled multiprocessors
  o Tightly coupled multiprocessors

Flynn's Classical Taxonomy

1 One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
2 Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.
3 The matrix below defines the 4 possible classifications according to Flynn.

			CONTROL
		Single	Multiple
DATA	Single	**SISD** Single Instruction, Single Data	**MISD** Multiple Instruction, Single Data
	Multiple	**SIMD** Single Instruction, Multiple Data	**MIMD** Multiple Instruction, Multiple Data

Single Instruction, Single Data (SISD)

1 A serial (non-parallel) computer
2 **Single instruction**: only one instruction stream is being acted on by the CPU during any one clock cycle
3 **Single data**: only one data stream is being used as input during any one clock cycle
4 Deterministic execution
5 This is the oldest and until recently, the most prevalent form of computer
6 Examples: most PCs, single CPU workstations and mainframes

Single Instruction, Multiple Data (SIMD)

1 **Single instruction**: All processing units execute the same instruction at any given clock cycle
2 **Multiple data**: Each processing unit can operate on a different data element
3 One clock

4 Best suited for specialized problems characterized by a high degree of regularity such as image processing.
5 Synchronous and deterministic execution
6 Two varieties: Processor Arrays and Vector Pipelines
7 Examples:
  o Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  o Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

Multiple Instruction, Single Data (MISD)

Several instructions are operating on a **single piece** of data.

1 **Few actual examples** of this class of parallel computer have ever existed
2 Some possible examples might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

Pipelined vector processor
Systolic array
1 could be classified as MISD
2 if elements of a vector may be considered to belong to the same piece of data and all pipeline stages represent multiple instructions that are being applied to that vector.

Multiple Instruction, Multiple Data (MIMD)

1 most common type of parallel computer
2 **Multiple Instruction**: every processor may be executing a different instruction stream
3 **Multiple Data**: every processor may be working with a different data stream
4 Execution can be synchronous or asynchronous, deterministic or non-deterministic
5 Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

Pipeline

The principle of pipelining can he exploited in computer architecture at **various levels**:

Pipelining at the **gate level**
Instruction
1 passes through the pipeline
2 entered and emitted in every cycle.

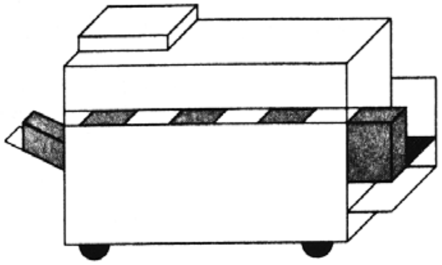
Instr. Fetch	Instr. Decode	Addr. Gener.	Oper. Fetch	…	Exec.

Pipelining at the **level of subsystems.**
1 Pipelined arithmetic units are typical examples.
2 The pipelined operations ADD, MUL, DIV are found in many contemporary, even though not typically pipelined, computer structures.

The example of computer with pipeline architecture is well-known CRAY-1

Pipeline

A pipeline is analogous to an **assembly line.**
1 If each segment of the pipe requires the **same amount of time**,
2 increase in the **throughput** is **equal to**
o **the number of segments** in the pipeline.



Pipelining in the context of a high-performance photocopier.

Data parallelism

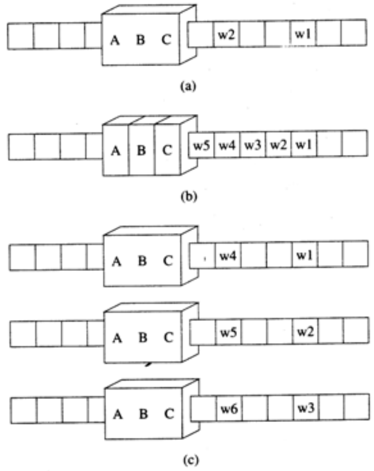
Data parallelism is the use of multiple functional units to apply the same operation simultaneously to elements of a data set.

1 A ***k***-fold increase in the number of functional units leads to a ***k***-fold increase in the throughput of the system
2 if there is no overhead associated with the increase in parallelism.

Contrasting Pipelining and Data Parallelism

Assume that

- it takes three units of time to assemble.
- assembly consists of three steps-A, B, and C,
- each step requires exactly one unit of time.



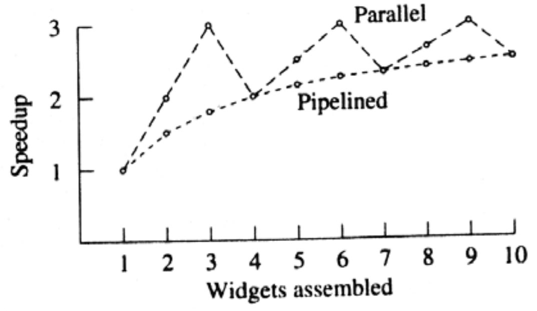
Three-segment pipeline

Each of the subassembly tasks has been assigned to a separate machine
1 first machine performs subassembly task every time unit
2 passes the partially result to the second machine.
3 second machine performs subassembly task B
4 third machine performs subassembly task C.

The pipelined assembly machine produces one result in three time units as does the sequential machine
1 After the initial time to fill the pipe, one result appears every time unit.
2 The second result appears at time unit four
3 The third one at time unit five, and so on.

Three data-parallel assembly machines

1 Each machine performs every subassembly task, as the sequential assembler.
2 Throughput is increased by replicating machines.
3 Another three results appear every three time units.
4 Time needed to produce four results is the same as the time needed to produce five or six results.



Speedup achieved by the pipelined and data-parallel machines.

Pipeline speedup

Consider ***n*** instructions processed in ***s*** stages, with ***t*** stage delay (cycle time).

Unpipelined processor:

1 ***s*** stages per instruction

2 for a total time of ***nst*** .

Pipelined processor:

1 first instruction finishes in ***s*** cycles,

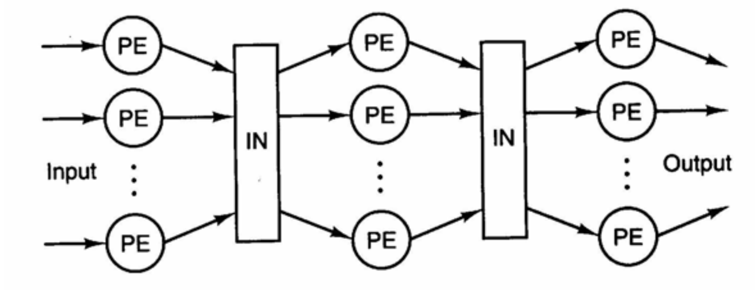
2 ***1*** cycle for following ***n-1*** instructions,

3 for a total time of **(*s*+(*n*-1))*t***.

$$S = \frac{ns}{s+n-1}$$

If ***n*** is large, this approaches ***s***.

Systolic Array

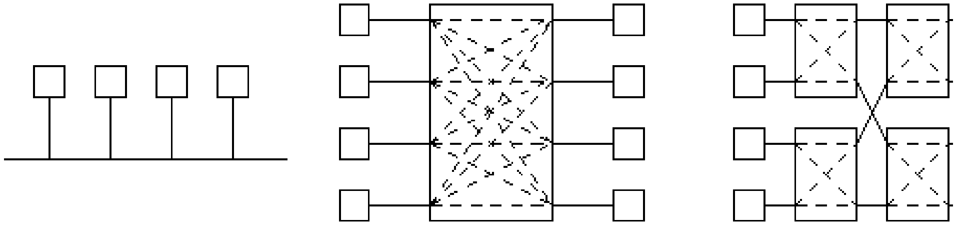


1 Large number of identical processing elements (PEs).
2 Each PE has limited local storage
3 Each PE is only allowed to be connected to neighboring PEs through interconnection networks.
4 PEs are arranged in a well-organized pipeline structure
o linear
o two-dimentional array.

Interconnects

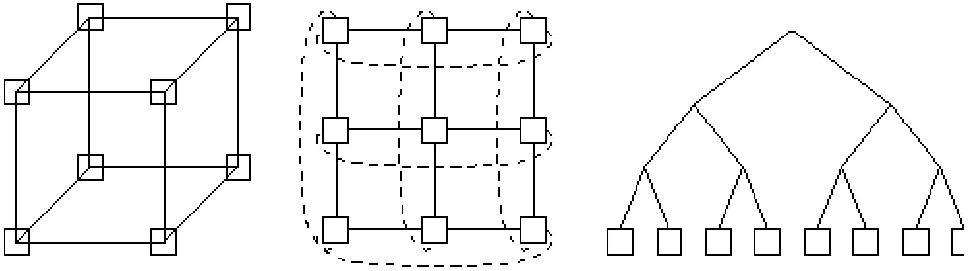
Dynamic Interconnects

1 Paths are established as needed between processors
2 System expansion difficult
3 Processors are usually equidistant
4 Examples : Bus based, Crossbar, Multistage networks



Static Interconnects

1 Consist of point-to-point links between processors
2 Can make parallel system expansion easy
3 Some processors may be "closer" than others
4 Examples : Hypercube, Mesh/Torus, Tree



Interconnect representation

A processor organization can be ***represented by a graph***
- **nodes** (vertices) represent processors
- **edges** represent communication paths between pairs of processors

Processor organizations are evaluated according to **criteria** that help us understand their effectiveness in implementing efficient parallel algorithms on real hardware.

These criteria are:
- **Diameter**.
- **Bisection width.**
- **Number of edges per node**
- **Maximum edge length**

Criteria

- The diameter of a network is the **largest distance** between two nodes.
- Low diameter is better,
  because the diameter puts **a lower bound** on the **complexity** of parallel algorithms requiring communication between arbitrary pairs of nodes.

Bisection width of the network.

- The bisection width of a network is the **minimum number of edges** that must be removed in order to **divide** the network into **two halves** (within one).
- **High bisection width is better**,
  because in algorithms requiring large amounts of data movement, the size of the data set divided by the bisection width puts **a lower bound** on the **complexity** of the parallel algorithm.

Criteria

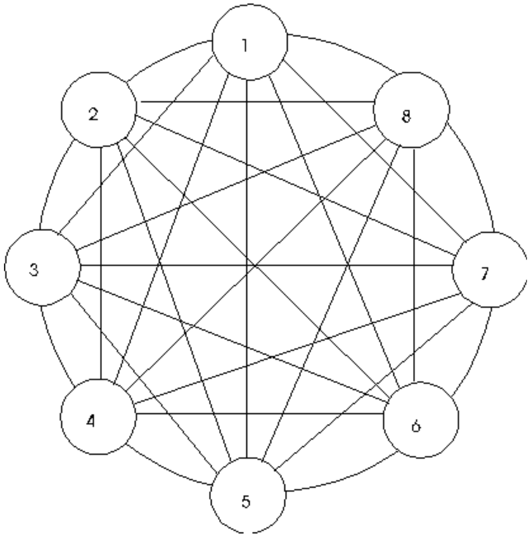
Number of edges per node

- It is best if the **number of edges** per node is a **constant** independent of the network size
- The processor organization **scales more easily** to systems with large numbers of nodes.

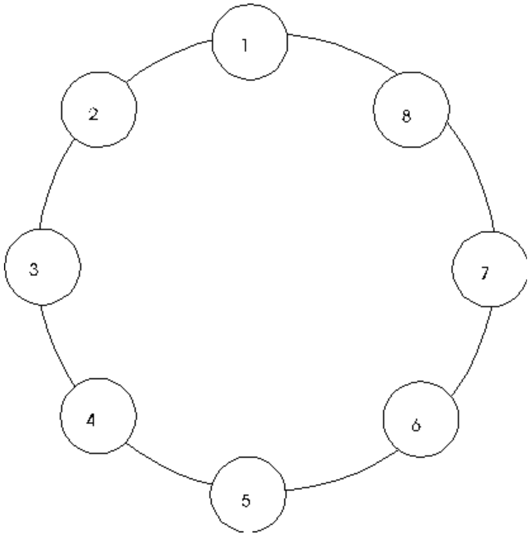
Maximum edge length

For scalability reasons it is best if the nodes and edges of the network can be ***laid out in three-dimensional space*** so that the maximum edge length is a ***constant independent of the network size***.

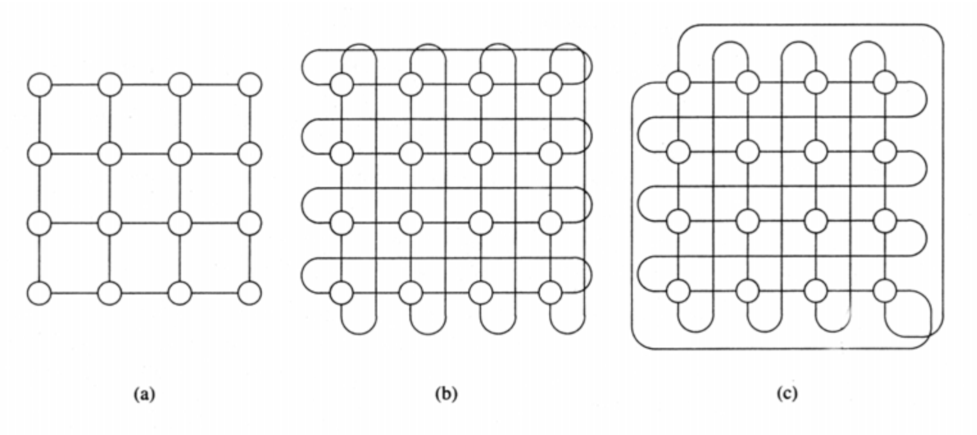
Full connected



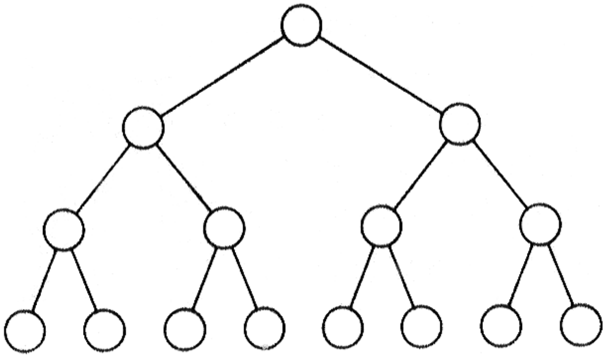
Ring



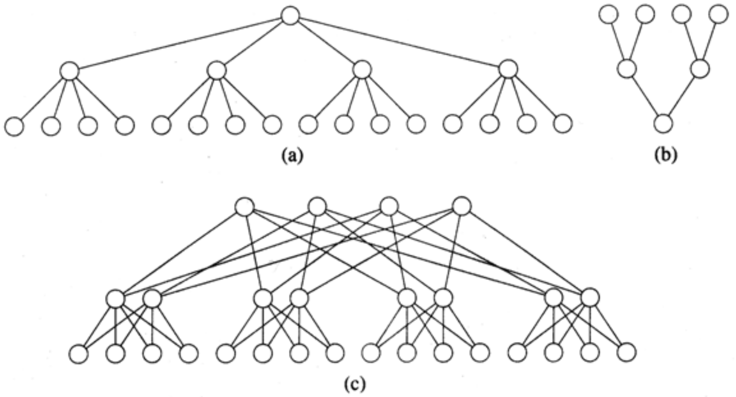
Mesh



Binary Tree



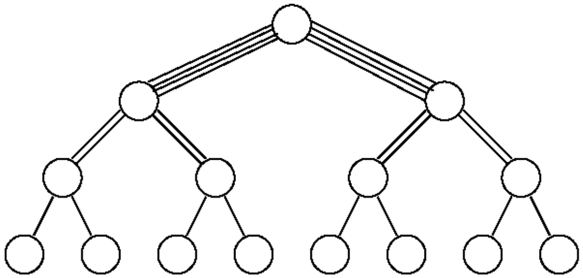
Hypertree



Hypertree network of degree 4 and depth 2

(a) Front view
(b) Side view
(c) Complete network.

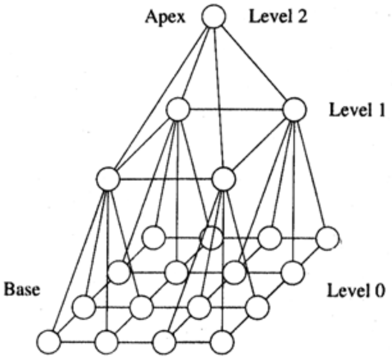
Binary Fat Tree



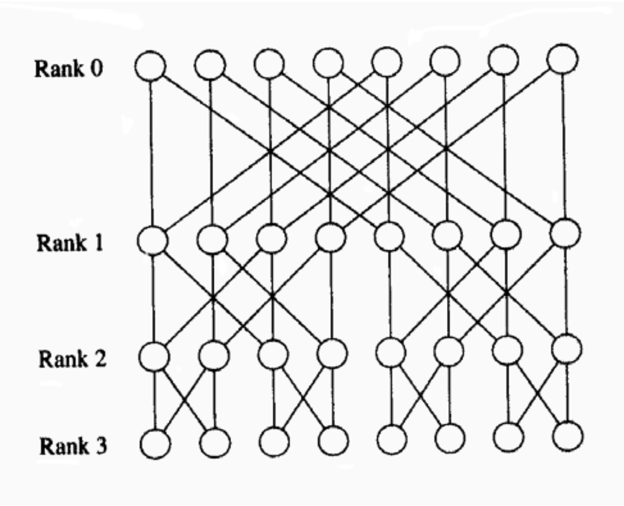
Pyramid

Every interior **processor** is connected to **nine** other processors:

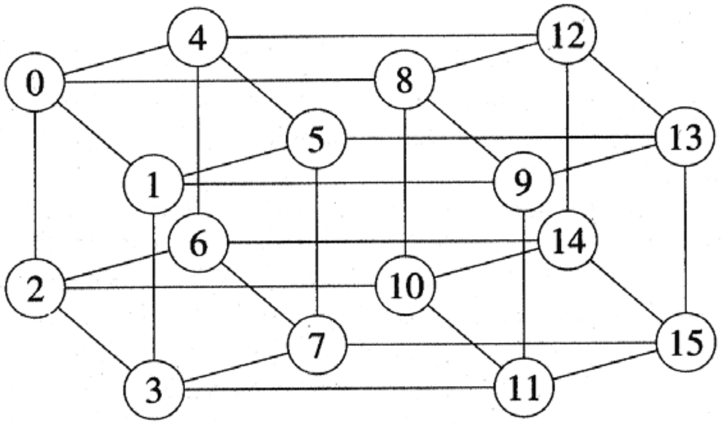
1 one parent,
2 four mesh neighbors,
3 four children.



Butterfly Network



Four-dimensional hypercube

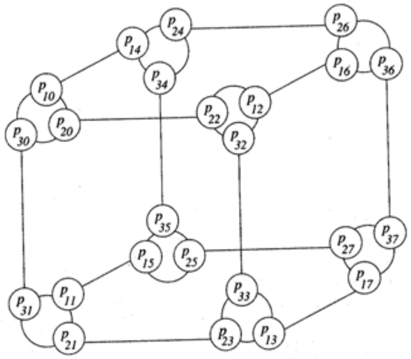


Cube-Connected Cycles Networks

The cube-connected cycles network is a ***k***-dimensional hypercube

1 ***2^k*** "vertices" are actually cycles of ***k*** nodes.
2 For each dimension, every cycle has a node connected to a node in the neighboring cycle in that dimension.

24-node cube-connected cycles network.



Characteristics of networks

These criteria are:

1 Diameter.
2 Bisection width.
3 Number of edges per node
4 Maximum edge length

Network	Nodes	Diameter	Bisection Width	Constant Number of Edges	Constant Edge Length
1-D mesh	*K*	K-1	1	Yes	Yes
2-D mesh	k^2	2(k-1)	K	Yes	Yes
3-D mesh	k^3	3(k-1)	k^2	Yes	Yes
Binary tree	2^k-1	2(k-1)	1	Yes	No
4-ary hypertree	$2^k(2^{k+l}-1)$	2k	2^{k+l}	Yes	No
Pyramid	$(4k^2-1)/3$	2logk	2k	Yes	No
Butterfly	$(k+1)2^k$	2k	2k	Yes	No
Hypercube	2^k	K	2^{k-1}	No	No
Cube-connected cycles	$k2^k$	2k	2^{k-1}	Yes	No

Performance Characteristics of Networks

1 **Bandwidth**: Maximum rate at which network can propagate information (bits/sec)
2 **Time of flight**: Time it takes for the first bit to reach the receiver (depends among others on physical distance).
3 **Transmission time**: Time it takes for a message to pass through the network. transmission time = (message size)/(bandwidth)
4 **Transport latency**: Time it takes for a message to pass through the network. Transport latency = time of flight + transmission time
5 **Sender overhead**: Time it takes for a sender to inject message into network (start up overhead). Mostly independent of message size.
6 **Receiver overhead**: Time it takes for a receiver to extract message from network. Mostly independent of message size.
7 **Total latency**: total latency = sender overhead + time of flight + transmission time + receiver overhead
total latency = overhead + (message size)/bandwidth Time it really takes a message to pass from sender to receiver.
8 **Effective bandwidth**: effective bandwidth = message size/(total latency) More realistic performance indicator than pure bandwidth. Depends on message size.

4. Parallel Terminology

Parallel Terminology

Some of the more commonly used terms are listed below.

Task

A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Parallel Task

A task that can be executed by multiple processors safely (yields correct results)

Serial Execution

Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

Parallel Execution

Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

Parallel Terminology

Shared Memory

From a hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory.
In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists. (**Single address space**)

Advantages:

1. Same memory layout as on uniprocessor.
2. Data sharing between tasks simple.
3. Data sharing between tasks fast.

Disadvantages:

1. Synchronization requirements.
2. Limited scalability of performance.
3. Progressive increase in cost due to special hardware.

Parallel Terminology

Distributed Memory

In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Advantages:

1. Processor scalability.
2. Memory scalability.
3. Each processor can quickly access its own memory.
4. Cost effectiveness.

Disadvantages:

1. Requires decomposition of data structures.
2. Difficult to program.

Parallel Terminology

Communications

Parallel tasks typically need to exchange data.

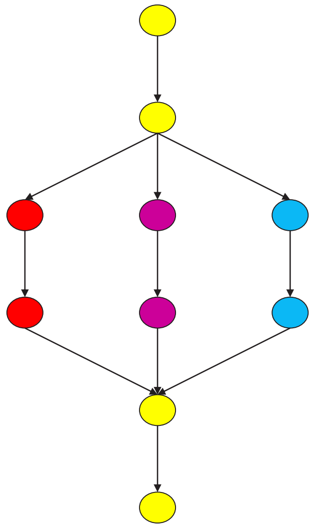
There are several ways this can be accomplished, such as
1 through a shared memory
2 over a network

However the actual event of data exchange is commonly referred to as communications regardless of the method employed.

- Embarrassingly parallel
- Regular and synchronous
- Irregular and/or asynchronous

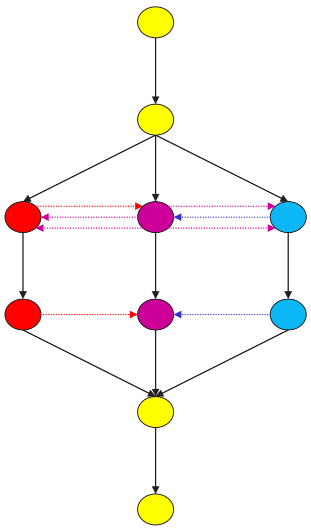
Embarrassingly parallel problems

- No communication is required
- Easily load balanced
- Expect perfect speedup
- Dynamic load balancing can be done using a task farm approach



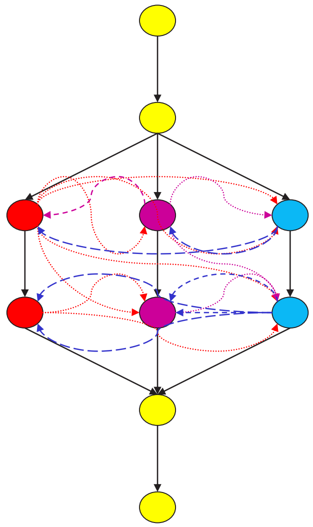
Regular and Synchronous problems

- Synchronous (or loosely synchronous) communications
- Reasonable speedup
- Computation time is grater than communication time
- Size of array is much larger than number of processors



Irregular and/or asynchronous

- Cannot be implemented efficiently
- Usually high communication overhead
- Careful coding of complex asynchronous communication
- Careful load balancing
- Difficult to get good speedup



Parallel Terminology

Synchronization

The coordination of parallel tasks in time, very often associated with communications.

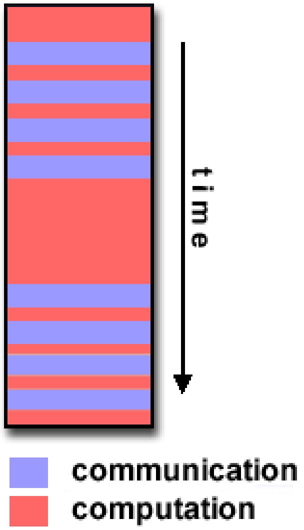
Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.

Synchronization usually involves waiting by at least one task, can therefore cause a parallel application's wall clock execution time to increase.

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

1. ***Coarse:*** relatively large amounts of computational work are done between communication events
2. ***Fine:*** relatively small amounts of computational work are done between communication events



Parallel Terminology

Observed Speedup

Observed speedup of a code which has been parallelized, defined as:
(time of serial execution)/(time of parallel execution)

One of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

1. Task start-up time
2. Synchronizations
3. Data communications
4. Operating System overhead
5. Software overhead imposed by parallel compilers, libraries, tools, etc.
6. Task termination time

Parallel Terminology

Massively Parallel

Refers to the hardware that includes a given parallel system - having many processors. The meaning of many keeps increasing, but currently means more than 1000.

Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors.

Factors that contribute to scalability include:

1. Hardware - particularly memory-CPU bandwidths and network communications
2. Application algorithm
3. Characteristics of your specific application and coding

Parallel Terminology

Peak Performance

Ppeak = clock frequency * number of processors * number of functional units = FLOP (Floating Point Operations per Second)

1. upper limit to performance
2. ignores
   - synchronization
   - communication
   - interconnection network capacity

Parallel Terminology

Benchmarking

Many applications do not achieve anywhere near the peak performance, particularly on high-performance computers.

Many standard benchmarks have been developed to determine actual performance of a computer over a range of applications.

Benchmarks aimed at parallel and vector HPC machines include:

1. LINPACK matrix solver (from SCALAPACK parallel linear algebra library), used to rank Top 500 list
2. NAS Benchmarks (kernels for some NASA fluid dynamics applications)
3. Others at BenchWeb (www.netlib.org/benchweb/)

Best benchmark is of course to run your applications (or the compute-intensive application kernel) on the machine.

5. Parallel Computer Memory Architectures

Shared Memory

General Characteristics:

1 Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
2 Multiple processors can operate independently but share the same memory resources.
3 Changes in a memory location effected by one processor are visible to all other processors.
4 Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Uniform Memory Access (UMA):

1 Most commonly represented today by Symmetric Multiprocessor (SMP) machines
2 Identical processors
3 Equal access and access times to memory
4 Sometimes called CC-UMA (Cache Coherent UMA). Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Advantages:

1 Global address space provides a user-friendly programming perspective to memory

Disadvantages:

1 lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path
2 Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
3 Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Non-Uniform Memory Access (NUMA):

1 Often made by physically linking two or more SMPs
2 One SMP can directly access memory of another SMP
3 Not all processors have equal access time to all memories
4 Memory access across link is slower
5 If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

General Characteristics:

1 Require a communication network to connect inter-processor memory.
2 Processors have their own local memory.

Parallel Computer Distributed Memory Architectures

Taxonomy

1. Distributed Memory Supercomputer
2. Cluster
3. Local Area Network
4. Computational Grid

Advantages:

1. Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
2. Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
3. Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

1. The programmer is responsible for many of the details associated with data communication between processors.
2. It may be difficult to map existing data structures, based on global memory, to this memory organization.

Comparison of Shared and Distributed Memory Architectures

Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	SGI Origin Sequent HP Exemplar IBM POWER4	Cray T3E IBM SP2
Communications	MPI Threads OpenMP	MPI Threads OpenMP	MPI
Scalability	to 10s of procs	to 1000s of procs	to 10000s of procs
Draw Backs	Limited memory bandwidth	New architecture Point-to-point communication	System administration, Programming is hard to develop and maintain

1. The shared memory component is usually a cache coherent SMP machine.
2. The distributed memory component is the networking of multiple SMPs.
3. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.

6. Parallel Programming Models

Parallel Programming Models

Overview

1. Parallel programming models exist as an abstraction above hardware and memory architectures.

2. There are several parallel programming models in common use:

○ Shared Memory	(OpenMP)
○ Threads	(Posix Threads)
○ Message Passing	(MPI)
○ Data Parallel	(HPF)
○ Hybrid	

Parallel Programming Models

1. These models are **NOT** specific to a particular type of machine or memory architecture.
2. Any of these models can (theoretically) be implemented on any underlying hardware. Two examples:

Shared memory model on a distributed memory machine:
Machine memory was physically distributed, but appeared to the user as a single shared memory (global address space). Has been implemented both

1. in software (e.g., to provide the shared memory programming model on networks of workstations)
2. in hardware (e.g., using cache consistency protocols to support shared memory across physically distributed main memories. ANSI/IEEE Scalable Coherent Interface (**SCI**) standard, IEEE Std. 1596-1992)

Generically, this approach is referred to as "virtual shared memory".
Message passing model on a shared memory machine: MPI on SGI Origin.
The SGI Origin employs the CC-NUMA type of shared memory architecture, where every task has direct access to global memory. However, the ability to send and receive messages with MPI, as is commonly done over a network of distributed memory machines, is not only implemented but is very commonly used.

Shared Memory Model

1. In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
2. Various mechanisms such as locks and semaphores may be used to control access to the shared memory.
3. An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
4. An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

Threads Model

1. In the threads model of parallel programming (lightweight processes), a single process can have multiple, concurrent execution paths.
2. Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
   - The main program performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
   - Each thread has local data, but also, shares the entire resources of the main program. This saves the overhead associated with replicating a program's resources for each thread.
   - Any thread can execute any subroutine at the same time as other threads.
   - Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.
   - Threads can come and go, but the main program remains present to provide the necessary shared resources until the application has completed.
3. Threads are commonly associated with shared memory architectures and

operating systems.

Threads Model

The scheduling of threads will be performed in a similar way as for processes.

Scheduling will be performed on a per-thread basis. In other words, the process-thread model is a *finer grain scheduling model* than the process model.

It has numerous *advantages*

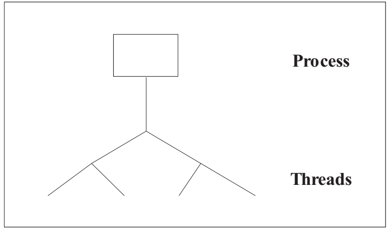
1 with finer grained entities more parallelism can be exposed
2 creation of threads or the communication, synchronization or switch among threads are far less expensive operations then those for processes, since all threads belonging to the same process are sharing the same resources.
3 most operating systems are based on the process-thread model.

Threads Model

4 Standard is prepared to standardize the thread interface (IEEE POSIX 1003.40).
5 Many operating systems are taking into consideration this standard.

Threads have a similar *lifecycle* as the processes and will be managed mainly in the same way as processes are.

1 Initially each process will be created with one single thread.
2 Threads are usually allowed to create new ones using particular system calls
3 Typically for each process a thread tree will be created.



Threads Model Implementations

From a programming perspective, threads implementations commonly include:

1 **library of subroutines** that are called from within parallel source code
2 set of **compiler directives** imbedded in either serial or parallel source code

In both cases, the programmer is responsible for determining all parallelism.

1 Thread level concurrent execution is termed as **multithreading**.
2 Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

3 Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.

POSIX Threads

1 **Library based**; requires parallel coding
2 Specified by the IEEE POSIX 1003.1c standard (1995).
3 Commonly referred to as Pthreads.
4 Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
5 Very explicit parallelism; requires significant programmer attention to details.

OpenMP

1 **Compiler directive based**; can use serial code
2 Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
3 Portable / multi-platform, including Unix and Windows NT platforms
4 Available in C/C++ and Fortran implementations
5 Can be very easy and simple to use - provides for "incremental parallelism"

Message Passing Model

1 A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.

2 Tasks exchange data through communications by sending and receiving messages.

3 Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

4 From a programming perspective, message passing implementations commonly includes **a library of subroutines** that are imbedded in source code. The programmer is responsible for determining all parallelism.

Message Passing Model Implementations

1 Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
2 In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.

PVM had been de-facto standard focused in distributed computing

3 Part 1 of the **Message Passing Interface (MPI)** was released in 1994.
4 Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web: http://www-unix.mcs.anl.gov/mpi/
5 MPI is now the **"de facto" industry standard** for message passing, replacing virtually all other message passing implementations. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. Very few have a full implementation of MPI-2.

6 For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory

(memory copies) for performance reasons.

Other Models

Other parallel programming models besides those previously mentioned certainly exist.
Only three of the more common ones are mentioned here.

a. Hybrid
b. SPMD
c. MPMD

Hybrid:

1 In this model, any two or more parallel programming models are combined.

2 A common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP).

3 Another common example of a hybrid model is combining data parallel with message passing. As mentioned in the data parallel model section previously, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks,

transparently to the programmer.

Single Program Multiple Data (SPMD)

1 SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

2 A single program is executed by all tasks simultaneously.

3 At any moment in time, tasks can be executing the same or different instructions within the same program.

4 SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.

5 All tasks may use different data

Multiple Program Multiple Data (MPMD)

1 Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

2 MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.

3 All tasks may use different data

7. Designing Parallel Programs

Automatic vs. Manual Parallelization

1 Designing and developing parallel programs is a very manual process.

2 The programmer is typically responsible for both identifying and actually implementing parallelism.

3 Very often, manually developing parallel codes is a time consuming, complex and error-prone and **iterative** process.

4 For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs.

5 The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.

Automatic vs. Manual Parallelization

A parallelizing compiler generally works in two different ways:

1 Fully Automatic

- o The compiler analyzes the source code and identifies opportunities for parallelism.
- o The analysis includes identifying bottlenecks of parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
- o Loops (do, for) loops are the most frequent target for automatic parallelization.

2 Programmer Directed

- o Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
- o May be able to be used in conjunction with some degree of automatic parallelization also.

Automatic vs. Manual Parallelization

1 If you are beginning with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer.

2 There are several important problems that apply to automatic parallelization:

- o Wrong results may be produced
- o Performance may actually degrade
- o Much less flexible than manual parallelization
- o Limited to a subset (mostly loops) of code
- o May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex
- o Most automatic parallelization tools are for Fortran

Designing Parallel Programs

Problem ->

Algorithm ->

Language ->

Program ->

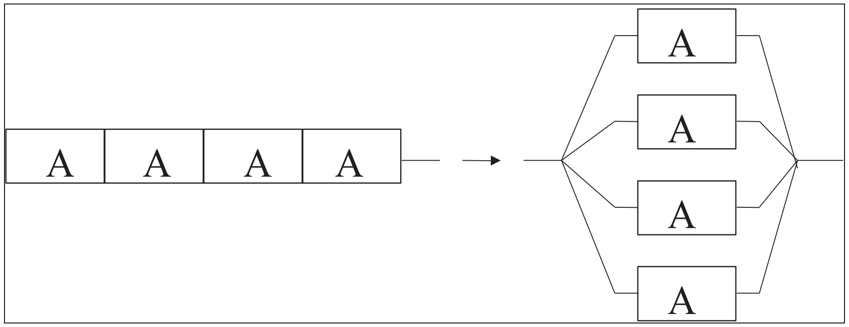
Object Code ->

Execution

Homogeneous Parallelization

The work to be done can be broken into identical (homogeneous) subtask, each working on a portion of the total task.

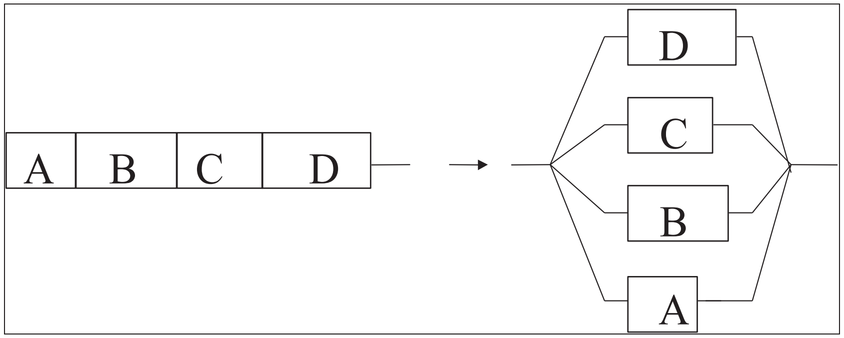
The obvious candidates for homogeneous parallelization are loops composed of a finite number of iterations.



Heterogeneous parallelization

The work to be done is spread over a large number of different subtasks, each of which works a discrete portion of the total algorithm.

An algorithm that has multiple independent components, in which each can be executed separately, would be a candidate for heterogeneous parallelization.



Understand the Problem and the Program

1 The first step in developing parallel software is to first understand the problem that you wish to solve in parallel.
2 If you are starting with a serial program, this necessitates understanding the existing code also.
3 Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

- o Example of **Parallelizable Problem**: Matrix Multiplication
- o Example of a **Non-parallelizable Problem**:

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: F(k + 2) = F(k + 1) + F(k)

This is a non-parallelizable problem because the calculation of the Fibonacci sequence entail dependent calculations rather than independent ones.
The calculation of the k + 2 value uses those of both k + 1 and k.
These three terms cannot be calculated independently and therefore, not in parallel.

Understand the Problem and the Program. Scalar product

Problem: Compute the scalar product ***c*** of two vectors ***a*** and ***b*** of length ***n***.

Specification: $$c = \sum_{i=1}^{n} a_i * b_i$$

Sequential code:

```
c = 0;
for (i=0; i<n; i++)
{
c = c + a[i] * b[i];
}
```

Parallel code:

NO concurrency in sequential code

Problem -> Algorithm -> Sequential code --> NO Parallel code

Understand the Problem and the Program. Scalar product

Sequential code -> Modified sequential code -> Parallel code

```
c = 0;
for (i=0; i<n; i++)
{
tmp[i] = a[i] * b[i];
}
for (i=0; i<n; i++)
{
c = c + tmp[i];
}
```

Parallel code:

First iteration can be parallelized

Understand the Problem and the Program. Loop Distribution

1 Code fragment:

```
for(i=0;i<n;i++) {
   a[i]= b[i]+ c[i]*d;
   c[i]= a[i-1];
 }
```

2 Problem: Dependence carried on **a**
3 Solution: Distribute loop into two loops, both are now i parallelizable

Understand the Problem and the Program. Fibonacci number

Problem: Compute n-th Fibonacci number

The Fibonacci series is formed by adding the latest two numbers to get the next one, starting from 0 and 1:

0 1 --the series starts like this.
0+1=1 so the series is now 0 1 1
1+1=2 so the series continues... 0 1 1 2 and the next term is
1+2=3 so we now have 0 1 1 2 3 and it continues as follows ...

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

Algorithm 1:

```
fib( n) = if n <= 1 then 1 else fibx( n, 2, 1, 1)

fibx( n, i, val, prev) = if i = n then val + prev else fib( n, i+1, val+prev, val)
```

Understand the Problem and the Program. Fibonacci number

The (recurrence) formula for these Fibonacci numbers is:
F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2) for n>1.

Algorithm 2:

```
fib( n) = if n <= 1
then 1
else fib( n-1) + fib( n-2)
```

An explicit formula for F(n) just in terms of ***n*** (not previous terms)

$$Fib(n) = \frac{1.6180339...^{n} - (-0.6180339...)^{n}}{2.236067977...}$$

Problem -> Alternative Algorithm -> Alternative Sequential code -> Parallel code

Understand the Problem and the Program. Factorial calculation

```
fact1 0    = 1
fact1 n    = n * fact1 (n - 1)
```

(a)

```
fact2 0    = 1
fact2 n    =  prod 1 n
prod m n = if m = n then  m
    else (prod  m halfway) * ( prod  halfway+1  n )
       where halfway  =  m +  (( n  - m) div 2 )
```

(b)

Understand the Problem and the Program. Matrix Multiplication

Problem: Compute the multiplication c of the two matrices *a* and *b* of size *n*n*

Specification: $C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$

Sequential Code:

```
for(i=0;i<n; i++) {
    for(j=0;j<n; j++) {
      for(k=0;k<n; k++) {
        c[i][j]+= a[i][k] * b[k][j];
      }
    }
  }
```

Parallel code:

different ways

Understand the Problem and the Program. Matrix Multiplication

Loop Reordering

```
for(k=0;k<n;k++) {
    for(i=0;i<n;i++) {
      for(j=0;j<n;j++) {
        c[i][j]+= a[i][k] * b[k][j];
      }
    }
  }
```

1 Outer loop cannot be parallelized due to dependence carried on **a.**

2 Can parallelize **i** loop, but very little work inside the parallel loop

3 Interchanging **k** and ***j*** loops does not alter program semantics; however, parallelizing the outer **i** loop creates more work inside the parallel loop

Understand the Problem and the Program. Matrix Multiplication

Loop Reordering

```
void parallel_mm() {
  int I, id, nprocs;

  id=m_get_myid();
  nprocs=m_get_numprocs();

  for(i=id;i<n;i+=nprocs) {
      for(k=0;k<n;k++) {
         for(j=0;j<n;j++) {
            c[i][j]+= a[i][k] * b[k][j];
         }
      }
   }
}
```

Understand the Problem and the Program. Example

Problem: Find a minimum element of ***n*** numbers

Data and Functional Parallelism

Data parallel:

loops where each iteration of a loop is independent and represents a simple statement and is executed on a different processor

```
for (i=0; i < 1000; i++)
  a[i] = b[i] + c[i];
```

Functional parallel:

loops which cannot be parallelized individually, but the different code blocks are independent and are executed on different processors

```
for (i=0; i < 10; i++)  /* block 1 */
  b[i-1] = b[i] + c[i];
 ...
for (j=0; j < 5; i++)   /* block n */
a[j-1] = a[j] + d[j];
```

Data Allocation

A two-dimensional array A[0:3, 0:3] is located in 4 memory units.

- **data retrieval** within a memory units is **sequential**
- retrieval of columns is ***n*** times slower than that of rows.

The data could be allocated so that access time for rows and columns will be balanced.

0	1	2	3
a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

0	1	3	3
a_{00}	a_{01}	a_{02}	a_{03}
a_{13}	a_{10}	a_{11}	a_{12}
a_{22}	a_{23}	a_{20}	a_{21}
a_{31}	a_{32}	a_{33}	a_{30}

Versions of matrix allocation in parallel access memory

Communications

Most problems do need communication !!

1 **(almost) NOT need communications**

- Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example,
  - image processing operation where every pixel in a black and white image needs to have its color reversed.
  - image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
- These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. **Very little inter-task communication is required**.

2 **need communications**

- Most parallel applications are not so simple, and do require tasks to share data with each other. For example,
  - 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data.
  - Changes to neighboring data has a direct effect on that task's data.

Synchronous vs. asynchronous communications

- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.
- Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
  - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
- Overlapping computation with communication is the greatest benefit for using

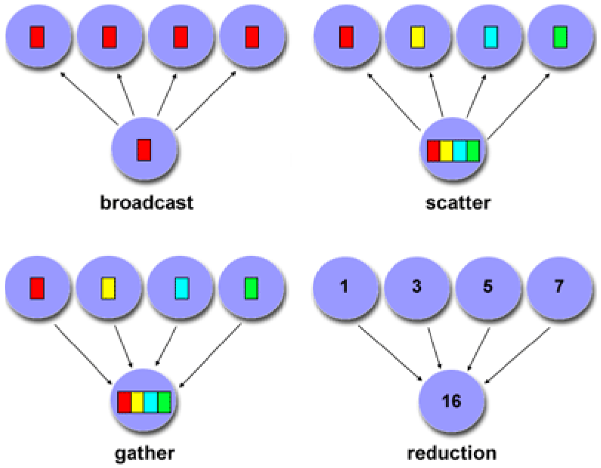
asynchronous communications.

Scope of communications

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
- ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ONE Sender
  - ONE Receiver
  - ONE Message
- ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

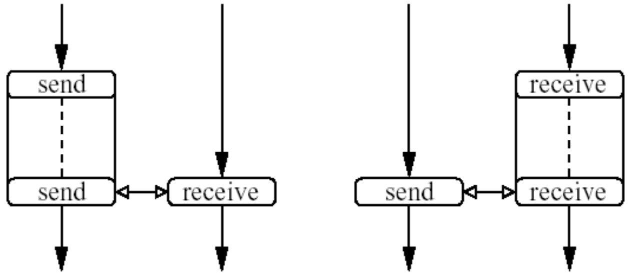
Scope of communications

- ***Collective*** - Some common variations:



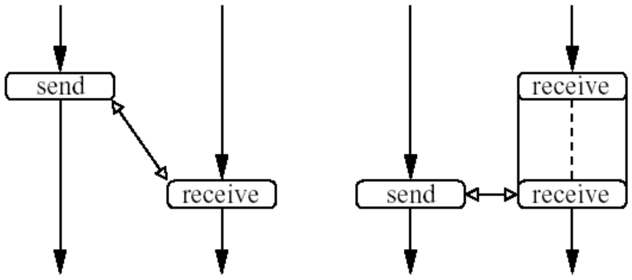
Synchronous vs. Asynchronous Communication

Blocking Send - Blocking Receive:



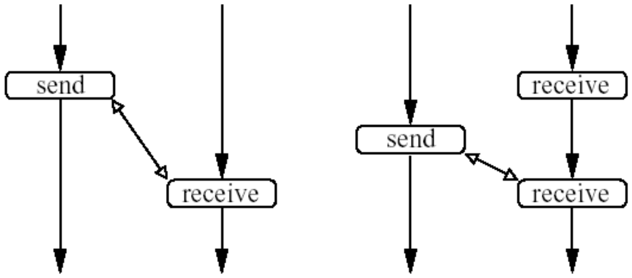
Synchronous vs. Asynchronous Communication

Non-Blocking Send - Blocking Receive:



Synchronous vs. Asynchronous Communication

Non-Blocking Send – Non-Blocking Receive:



8. Synchronization

Synchronization

Types of Synchronization:

1 **Barrier**

- Usually implies that all tasks are involved
- Each task performs its work until it reaches the barrier. It then stops, or "blocks".
- When the last task reaches the barrier, all tasks are synchronized.
- After that the tasks are automatically released to continue their work.

2 **Lock / semaphore**

- Can involve any number of tasks
- Typically used to serialize (protect) access to global data or a section of code.
- Only one task at a time may use (own) the lock / semaphore.
- The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
- Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.

Semaphore

Edsger Dijkstra (1965) proposed a semaphore as a **synchronization mechanism**

Semaphores are **special variables**

1 **non-negative integer values** (binary semaphore assumes 0 and 1)

2 only **two operations** *P* and *V* are defined.

For semaphore *S*,

1 operator *V* is equivalent to *S:=S+1*

2 operation *P* is equivalent to l: if S>0 **then** S:=S-1 **else goto** l;

Operation *P* and *V* are regard as **indivisible**, i.e. at any instant only one such operation can be executed.

Counting vs. Binary Semaphore

Counting Semaphore

- can take any value
- V operation never blocks
- P and V operations do not have to alternate
- V could always be the first operation

Binary Semaphore

- Can only take 0 or 1
- Both P and V operation may block
- P and V operations must alternate
- If the initial value is 0, the first operation must be V;
- if the initial value is 1, the first operation must be P.

Binary vs. Lock Semaphore

Binary Semaphore

- Has no concept of ownership
- Any thread can invoke P or V operations
- Consecutive P or V operations will be blocked
- Need to specify an initial value

Lock

- A lock can be owned by at most one thread at any given time
- Only the owner can invoke unlock operations
- The owner can invoke lock/unlock operations
- Does not have to be initialized

Dining Philosophers Problem (DPP)

A problem introduced by Dijkstra concerning resource allocation between processes.

The DPP is a model and universal method for testing and comparing theories on resource allocation.

The problem consists of a finite set of processes which

- share a finite set of resources,
- each of which can be used by only one process at a time,

thus leading to potential deadlock.

Dining philosophers problem (DPP)

Five philosophers sit around a table and think until hungry.

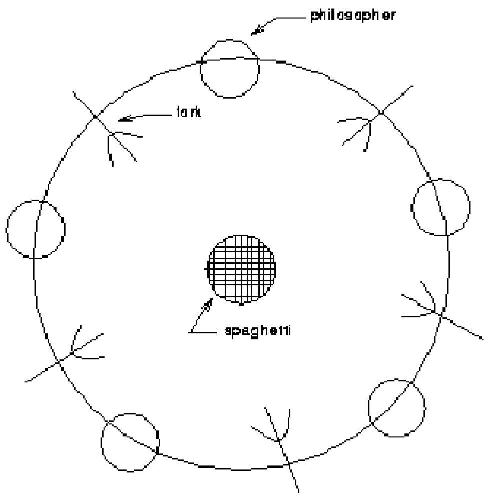
Interspersed between the philosophers are five forks.

A hungry philosopher must have exclusive access to both its left and right forks in order to eat. If they are not both free the philosopher waits.

The following algorithm

1. does not deadlock
   - it never happens that all philosophers are hungry each holding one fork and waiting for the other
2. allows maximal parallelism
   - philosopher never picks up and holds a fork while waiting for the other fork to become available when the fork it is holding could be used for eating by its neighbor
3. **allows starvation**
   - philosopher's two neighbors can collaborate and alternate their eating so the one in the middle never can use the forks.

Dining philosophers problem (DPP)



Dining philosophers problem (DPP)

Each fork is represented by a semaphore and each hungry philosopher does a ``P'' on its left fork and then its right fork.

We can fix the deadlock problem and retain no starvation but we still do not have maximal parallelism.

All philosophers pick up left then right except one designated philosopher who picks up right then left.

9. Granularity

Fine-grain Parallelism

1 Relatively small amounts of computational work are done between communication events

Fine grain: implies tens of instructions, e.g. statements in programs

Each loop iteration of C program is executed on a different processor

```
for (i=0; i < 1000; i++) a[i] = b[i] + c[i];
```

2 Low computation to communication ratio
3 Facilitates load balancing
4 Implies high communication overhead and less opportunity for performance enhancement
5 If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

Coarse-grain Parallelism

1 Relatively large amounts of computational work are done between communication/synchronization events

Coarse grain:
implies thousands of instructions, e.g. functions or procedure calls in programs --
Each group of loop iterations of C program representing complex sets of statements containing function calls executed on different processor

```
for (i=0; i < 1000; i++) a[i] = b[i] + c[i] * work(d[i]);
```

2 High computation to communication ratio
3 Implies more opportunity for performance increase
4 Harder to load balance efficiently

Which is Best?

1 The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
2 In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
3 Fine-grain parallelism can help reduce overheads due to load imbalance.

10. Limits and Costs of Parallel Programming

Performance

The three most commonly used terms used to describe the performance of parallel computers and computations are **speed-up (absolute, relative), efficiency, cost.**

$$S_p = \frac{T_1}{T_p} \qquad E_p = \frac{S_p}{p}$$

$$Speedup = \frac{Time\ of\ the\ best\ sequential\ program}{Time\ of\ the\ parallel\ program\ on\ p\ processors}$$

$$Efficiency = \frac{Speedup}{Number\ of\ processors}\ (speed-up\ per\ processor)$$

Cost **of parallel program** = time*number of processors *Cp=T*p*

A suitable measure for comparing algorithms is a ratio *Sp/Cp* = *Ep*Sp/T*$_1$, so that in designing algorithms we try to maximize the product of efficiency and speedup.

Performance

$$relative\ Speedup = \frac{Time\ of\ the\ parallel\ program\ on\ one\ node}{Time\ of\ the\ parallel\ program\ on\ p\ nodes}$$

1 Parallelizability

2 Used to estimate the relative program performance as the number processors increases.

3 Speed-up provides an indication of the effective number of processors utilized

Parallel Speedup and Amdahl's Law

When your program runs on more than one CPU, its total run time should be less.

- But how much less?
- And what are the limits on the speedup?

In 1967, an IBM designer, Gene Amdahl made the statement that the bottlenecks present in sequential computers would be difficult to overcome because even parallel solutions involved an overhead that is sequential and therefore unlikely to be overcome by parallel techniques.

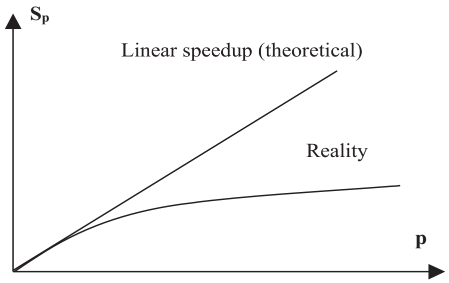


Figure 2. Speedup

Amdahl's Law

There are two basic limits to the speedup you can achieve by parallel execution:

1 The fraction of the program that can be run in parallel, *p_f*, is never 100%.

2 Because of hardware constraints, after a certain point, there is less and less benefit from each added CPU.

We can estimate of how much performance is lost using *Amdahl's Law*.

1 This rule of parallel computation states that the performance gain one may achieve from parallelizing a program **is limited by the amount of the program that runs sequentially.**

Amdahl's Law

2 If ***s_f***, is the portion of a program that runs sequentially, and

3 ***p_f***, is the part that runs in parallel
(s_f,+p_f, = 1),

4 parallel speedup is

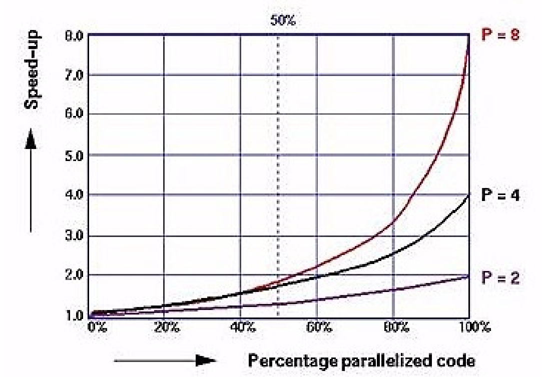
$$Speedup = \frac{s_f + p_f}{s_f + p_f / p} = \frac{1}{s_f + (1 - s_f)/p} \leq \frac{1}{s_f}$$

Suppose p_f, = 0.8;
then Speedup(2) = 1/(0.4+0.2) = 1.67,
and Speedup(4) = 1/(0.2+0.2) = 2.5.

1 The maximum possible speedup --- if you could apply an infinite number of CPUs --- would be 1/(1-p_f,)=1/s $_f$,

Amdahl's Law

The fraction ***p_f***, has a strong effect on the possible speedup, as shown in this graph:

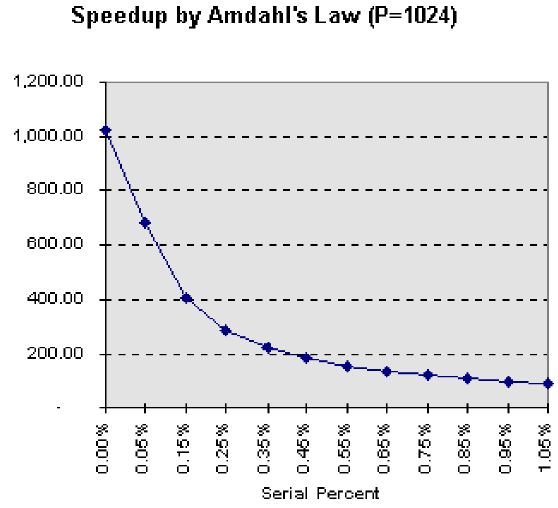


5 more CPUs you have, the more benefit you get from increasing *p_f*,.

6 Using only 4 CPUs, you need only *p_f*, = *0.6* to get half the ideal speedup.

7 With 8 CPUs, you need p_f, = 0.85 to get half the ideal speedup.

The expected speedup



Amdahl's Law

Expected Speedup

$$T_1 = T_{sf} + T_{pf}, T_p = T_{sf} + \frac{T_{pf}}{p}.$$

$$T_{sf} = ?\, T_{pf} = ?$$

s_f=? p_f=?

Predicting Execution Time with n CPUs

You can use the calculated value of p_f to extrapolate the potential speedup with higher numbers of CPUs. For example, if *p*=0.895 and T(1)=188 seconds, what is the expected time with four CPUs?

Speedup(4)= 1/((0.895/4)+(1-0.895)) = 3.04
T(4)= T(1)/Speedup(4) = 188/3.04 = 61.8

Speed up classes

There are three possible relationship between a speedup and the number of processors:

1 ***Speedup < P***, or sublinear speedup;
2 ***Speedup = P***, or linear speedup;
3 ***Speedup > P***, or superlinear speedup.

Practical parallel program consolidates the final answer in one program

1 serial percentage in Amdahl's Law is never zero in practice.
2 Thus, theoretically linear and superlinear speedups are not possible.

In reality, however, there are two factors that can be used to produce linear or superlinear speedups:

1 Use of a resource constrained serial execution as the base for speedup calculation;

2 Use a parallel implementation that can bypass large amount of calculation steps while yield the same output of the corresponding serial algorithm.

Costs of Parallel Programming. Complexity

1 In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.

2 The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:

- o Design
- o Coding
- o Debugging
- o Tuning
- o Maintenance