Scheduling on Parallel Systems

- Sathish Vadhiyar

Parallel Scheduling Categories

- Job Scheduling [this class]
  - A set of jobs arriving at a parallel system
  - Choosing an order of jobs for execution to minimize total turnaround time
- Application Scheduling [next class]
  - Mapping a single application's tasks to resources to reduce the total response time
  - In general, difficult to achieve for communication-intensive applications
  - For applications with independent tasks (pleasingly parallel applications), some methods have been proposed

JOB SCHEDULING

Job Scheduling - Introduction

- A parallel job is mapped to a subset of processors
- The set of processors dedicated to a certain job is called a partition of the machine
- To increase utilization, parallel machines are typically partitioned into several non-overlapping partitions, allocated to different jobs running concurrently – space slicing or space partitioning

Introduction

- Users submit their jobs to a machine's scheduler
- Jobs are queued
- Jobs in queue considered for allocation whenever state of a machine changes (submission of a new job, exit of a running job)
- Allocation – which job in the queue?, which machine?

Introduction

- Packing jobs to the processors
- Goal – to increase processor utilization
- Lack of knowledge of future jobs and job execution times. Hence simple heuristics to perform packing at each scheduling event

Variable Partitioning

- Dilemma about future job arrivals and job terminations
- Current scheduling decisions may impact jobs that arrive in the future
- Can lead to poor utilization
- e.g.: currently running: a 64-node job. Queued: 32-node and 128-node jobs

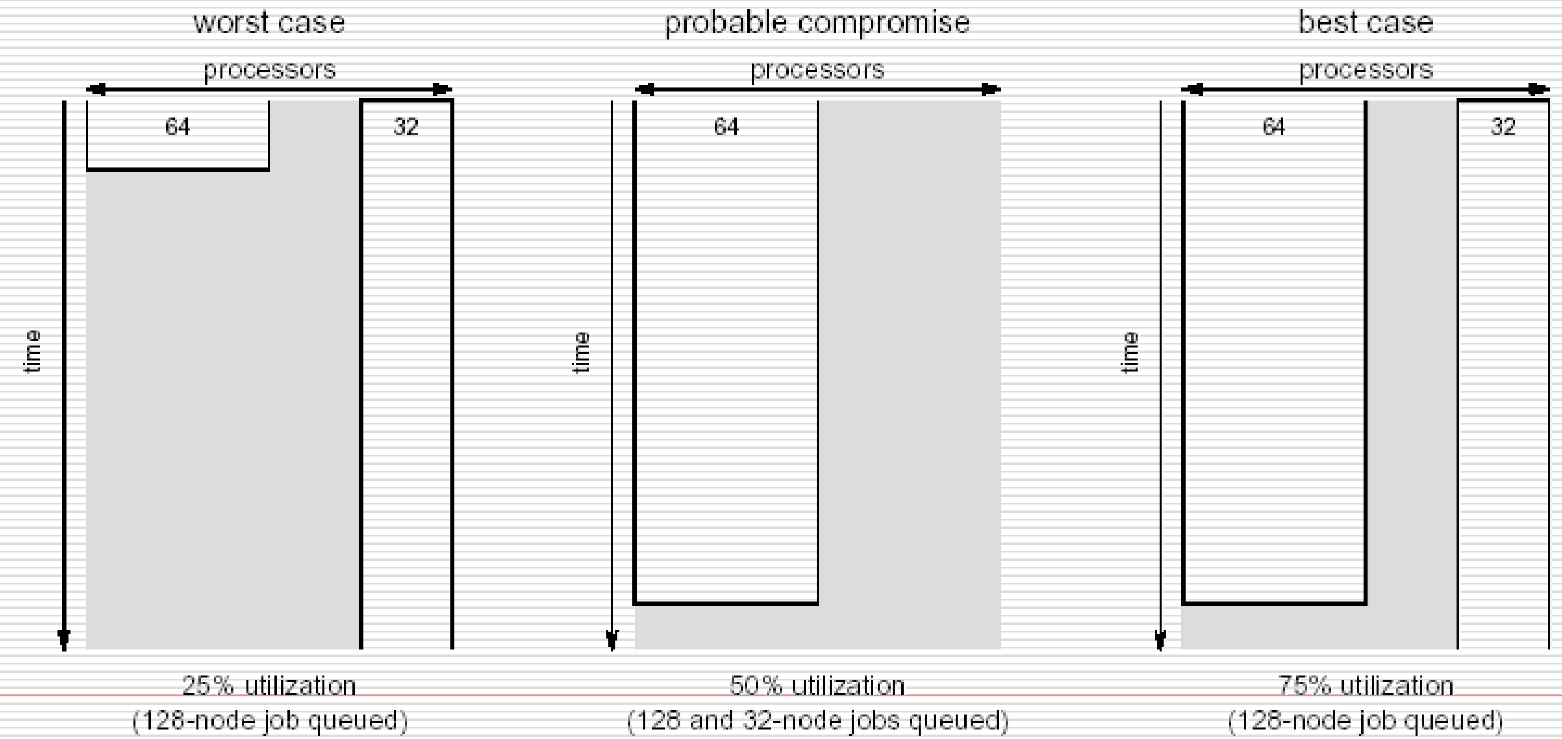


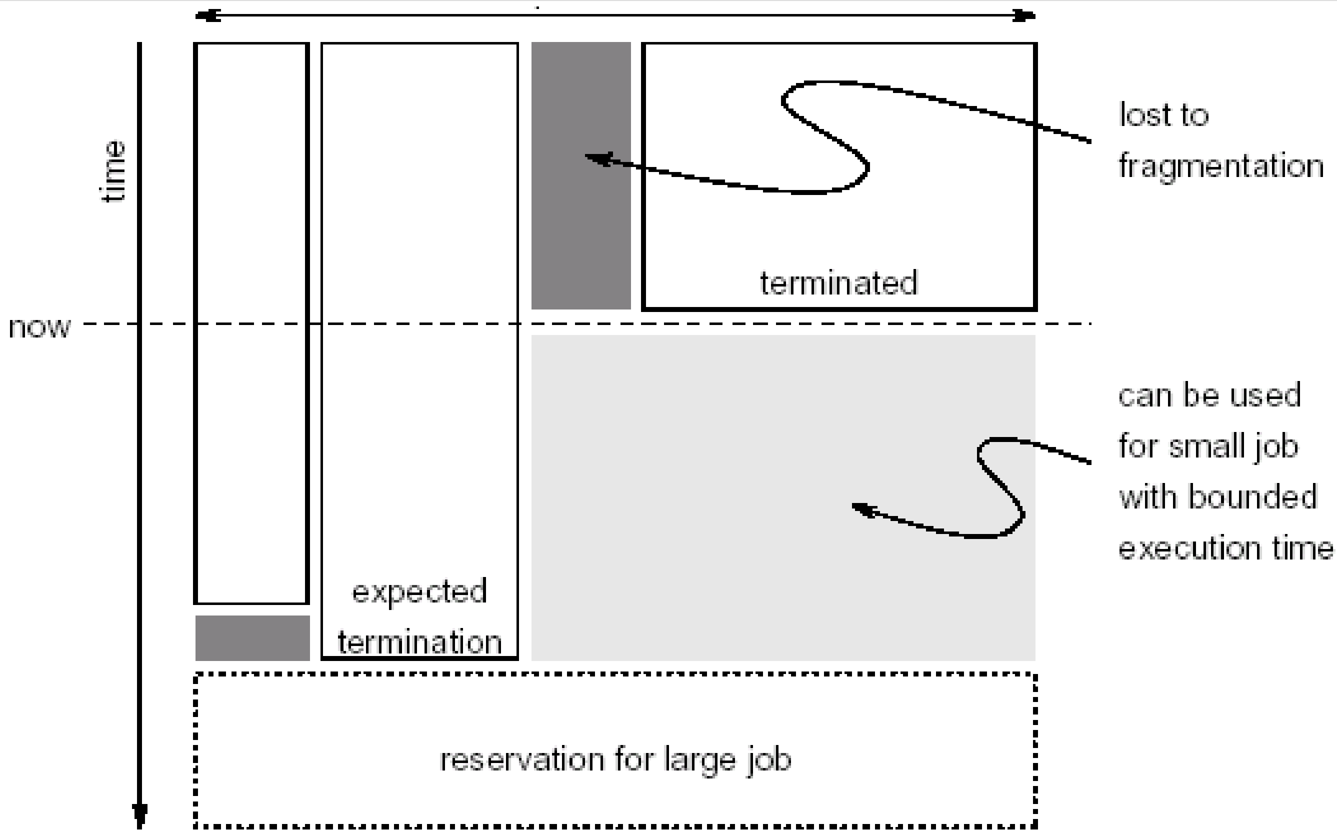
Figure 1: *Example of the problems faced by variable partitioning.*

Scheduling Policies

- FCFS
- If the machine's free capacity cannot accommodate the first job, it will not attempt to start any subsequent job
- No starvation; But poor utilization
- Processing power is wasted if the first job cannot run

Backfilling

- Allows small jobs from the back of the queue to execute before larger jobs that arrived earlier
- Requires job runtimes to be known in advance – often specified as runtime upper-bound



Backfilling

- Identifies holes in the 2D chart and moves smaller jobs to fill those holes
- 2 types – conservative and aggressive (EASY)

EASY Backfilling

- Aggressive version of backfilling
- Any job can be backfilled provided it does not delay the first job in the queue
- Starvation cannot occur for the first job since queuing delay for the first job depends only on the running jobs
- But jobs other than the first may be repeatedly delayed by newly arriving jobs

Conservative Backfilling

- Makes reservations for all queued jobs
- Backfilling is done subject to checking that it does not delay any previous job in the queue
- Starvation cannot occur at all

Backfilling Variants

1. Depending on the order in which the queue is scanned to find backfilling jobs
   1. By estimated runtime or estimated slowdown
      1. Slowdown – (wait_time + running time)/running_time
2. Dynamic backfilling/slack-based backfilling – overruling previous reservation if introducing a slight delay will improve utilization considerably\
   1. Each job in the queue is associated with a slack – maximum delay after reservation.
   2. Important jobs will have little slack
   3. Backfilling is allowed only if the backfilled job does not delay any other job by more than that job's slack
   4. e.g. reservations to only those jobs whose expected slowdowns > threshold

Backfilling Variants

3. Multiple-queue backfilling
   1. Each job is assigned to a queue according to its expected execution time
   2. Each queue is assigned to a disjoint partition of the parallel system on which only jobs from this queue can be executed
   3. Reduces the likelihood that short jobs get delayed in the queue behind long jobs

LOS (Lookahead Optimizing Scheduler)

- Examines all jobs in the queue to maximize utilization
- Instead of scanning the queue in any order and starting any job that is small enough not to violate prior reservations
- LOS tries to find combination of jobs
- Using dynamic programming
- Results in local optimum; not global optimum
- Global optimum may leave processors idle in anticipation of future arrivals

Notations

Summary of notation

Symbol	Meaning
N	Machine size
n	Free capacity
rj_i	Running job number i
R	The set of all running jobs
wj_i	Waiting job number i
WQ	The set of all waiting jobs
S	The set of jobs selected for scheduling

- Scheduler is invoked at t
- Machine runs jobs R = $\{rj_1, rj_2, \ldots, rj_r\}$ each with 2 attributes:
  - Size
  - Estimated remaining time, rem
- Machine's free capacity, n = N – sum(rj_i.size)
- Waiting jobs in the queue, WQ = $\{wj_1, wj_2, \ldots, wj_q\}$, each with 2 attributes
  - Size requirements
  - User's runtime estimate, time

Objective

- Task is to select a subset, S in WQ, selected jobset that maximizes machine utilization; these jobs removed from the queue and started immediately
- Selected jobset is **safe** if it does not impose a risk of starvation

Matrix M

- Size of *M* = (|WQ+1|) x (n+1)
- $m_{i,j}$ contains an integer value *util*, boolean flag *selected*
- util – (i,j) holds the maximum achievable utilization at this time, if machine's free capacity is j and only waiting jobs [1…i] are considered for scheduling
- Maximum achievable utilization – maximal number of processors that can be utilized by the considered waiting jobs

Matrix M

- selected – if set, indicates that wj_i was chosen for execution; when the algorithm finished calculating M, it will be used to trace the jobs which construct S
- i=0 row and j=0 column are filled with zeros

Filling M

- M is filled from left-right and top-bottom
- If adding another processor (bringing the total to j) allows the currently considered job wj_i to be started:
  - then check if including wj_i will increase utilization
- The utilization that would be achieved assuming this job is included is calculated as util'
- If util' higher than utilization without this job, the selected flag is set to true for this job
- If not, or if the job size is larger than j, the utilization is what it was without this job, that is $m_{i-1,j}$.util
- The last cell shows the maximal utilization

Constructing M

Algorithm 1. Constructing M

```
for j = 0 to n
    m_{0,j}.util ← 0                              // init top row
for i = 1 to |WQ|                                 // outer loop on rows (jobs)
    m_{i,0}.util ← 0                              // init first column
    for j = 1 to n                                // inner loop on columns (free processors)
        m_{i,j}.util ← m_{i-1,j}.util             // default: don't use this job
        m_{i,j}.selected ← False
        if wj_i.size ⩽ j                          // job is a potential candidate
            util' ← m_{i-1,j-wj_i.size}.util + wj_i.size   // find achievable utilization with it
            if util' > m_{i-1,j}.util             // improves utilization
                m_{i,j}.util ← util'              // so use it
                m_{i,j}.selected ← True
```

Example

- A machine of size, N = 10
- At t=25, the machine runs rj1 with size=5, and rem=3
- The machine's free capacity, n=5
- Set of waiting jobs and resulting M is shown
- Selected flag is denoted by ↖ if set and by ↑ if cleared

Table M for Example

Table 2

Resulting M for the example

i (*size*)	j					
	0	1	2	3	4	5
0 (ϕ)	0	0	0	0	0	0
1 (7)	0	0 ↑	0 ↑	0 ↑	0 ↑	0 ↑
2 (3)	0	0 ↑	0 ↑	3↖	3↖	3↖
3 (1)	0	1↖	1↖	3 ↑	4↖	4↖
4 (2)	0	1 ↑	2↖	3 ↑	4 ↑	5↖
5 (2)	0	1 ↑	2 ↑	3 ↑	4 ↑	5 ↑

Example Explanations

- Job 1 requires 7, hence does not fit in any of the 5; hence util is 0 and selected false for the entire row
- For job 2, when 3 or more processors are used, it is selected and util is 3
- Job 3
  - When only 1 or 2 processors are used, it is selected and util 1
  - When 3 processors are considered, it is better to select the second one; not the third
  - With 4 or more, job 2 and job 3 can be selected; util is 4
- Job 4 is selected
  - When 2 processors are considered (better than utilizing job 3 with util 1)
  - When 5 are considered (together with job 2 with util 5)
- Job 5 does not add anything, never selected
- Thus max util is 5
- Conventional backfilling would have selected jobs 2 and 3 leading to utilization of 4.

Constructing S

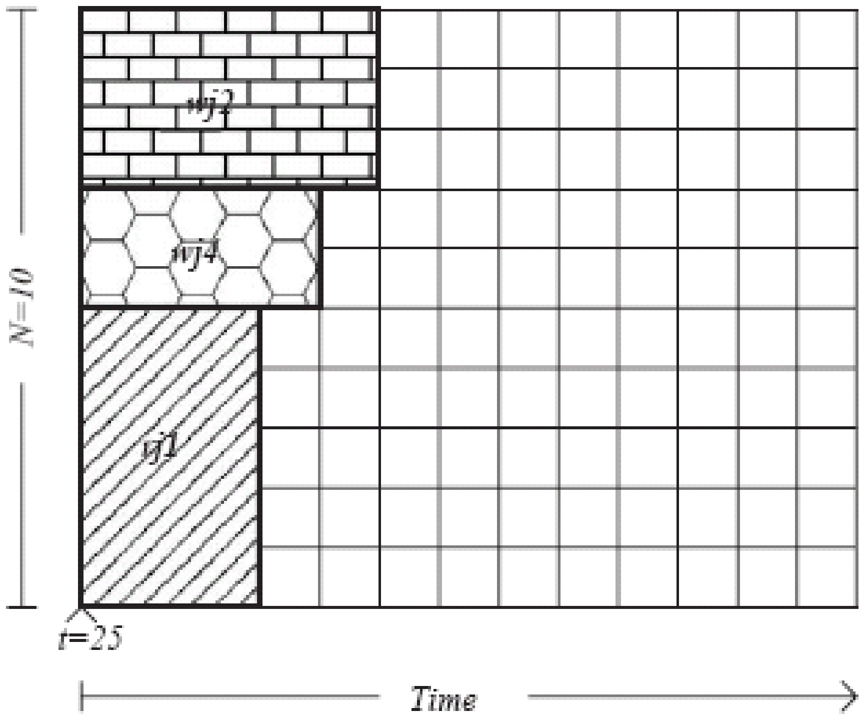
- Starting at the last computed cell, S is constructed by following the boolean flags backwards
- Jobs that are marked as selected are added to S

Algorithm

Algorithm 2. Constructing S

$S \leftarrow \{\}$	// initially empty
$i \leftarrow \|WQ\|$	// start from end
$j \leftarrow n$	
while $i > 0$ and $j > 0$	// continue until reach edge
if $m_{i,j}.selected = True$	
$S \leftarrow S \cup \{wj_i\}$	// add this job
$j \leftarrow j - wj_i.size$	// skip appropriate columns
$i \leftarrow i - 1$	

Scheduling wj2 and wj4



wj	size	time
1	7	4
2	3	5
3	1	6
4	2	4
5	2	2

Fig. 1. Scheduling wj_2 and wj_4 at $t = 25$.

Starvation

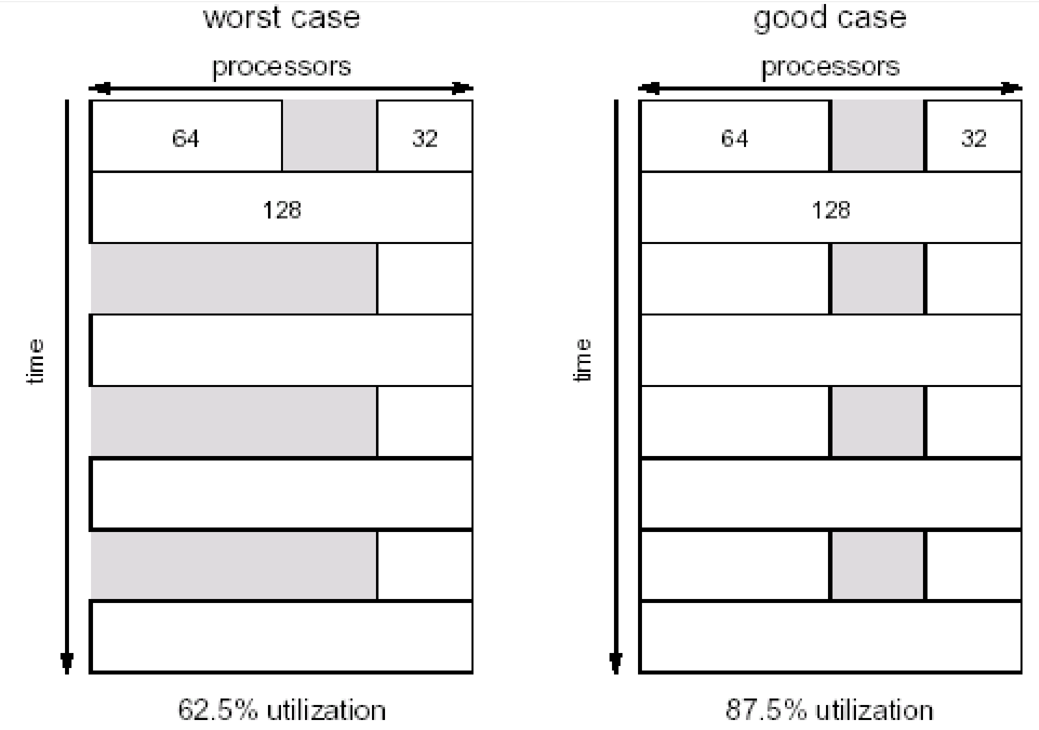
- Algorithm 1 has the drawback that it might starve large jobs
- In our example, the first queued job has size requirements 7
- Since it cannot start at t, wj2 and wj4 are started.
- But after 3 time units, rj1 releases it processors; however, processors are not available for wj1 since wj2 and wj4 are occupying processors;
- This can continue….

Freedom from Starvation

- Bound the waiting time of the first queued job
- The algorithm tries to start wj1
- If wj1.size < n, it removes the job from the queue and starts it
- If not, the algorithm computes the shadow time at which wj1 can begin execution
- Does this by traversing the running job list until reaching a job rjs, such that wj1.size < $n+sum_{i=1 to s}$(rji.size)
- shadow = t+rjs.rem
- Reservation is made for wj1 at shadow
- In the example, shadow = 28

Gang Scheduling

- Executing related threads/processes together on a machine
- Time sharing. Time slices are created and within a time slice processors are allocated to jobs.
- Jobs are context switched between time slices.
- Leads to increased utilization



Gang Scheduling

- Multi Programming Level: scheduling cycle in gang scheduling
- Scheduling matrix recomputed at every scheduling event – job arrival or departure
- 4 steps – cleanmatrix, compactmatrix, schedule, fillmatrix

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
time-slice 0	J_1^0	J_1^1	J_1^2	J_1^3	J_1^4	J_1^5	J_1^6	J_1^7
time-slice 1	J_2^0	J_2^1	J_2^2	J_2^3	J_2^4	J_2^5	J_2^6	J_2^7
time-slice 2	J_3^0	J_3^1	J_3^2	J_3^3	J_4^0	J_4^1	J_5^0	J_5^1
time-slice 3	J_6^0	J_6^1	J_6^2	J_6^3	J_4^0	J_4^1	J_5^0	J_5^1

Gang Scheduling Steps

- CleanMatrix

```
for i = first row to last row
    for all jobs in row i
        if row i is not home of job, remove it
```

- CompactMatrix

```
do{
    for i = least populated row to most populated row
        for j = most populated row to least populated row
            for all jobs in row i
                if they can be moved to row j,
                    then move and break
}while matrix changes
```

- Schedule other jobs – FCFS

- FillMatrix

```
do {
    for each job in starting time order
        for all rows in matrix,
            if job can be replicated in same columns
                do it and break
} while matrix changes
```

APPLICATION SCHEDULING

Background

- Tasks of a job do not have dependencies
- A machine executes a single task at a time
- Collection of tasks and machines are known apriori
- Matching of tasks to machines done offline
- Estimates of execution time for each task on each machine is known

Scheduling Problem

- ETC – Expected time to compute matrix
- ETC(i,j) – estimated execution time of task i on machine j
- Notations:
  - mat(j) – machine availability time for machine j, i.e., earliest time at which j has completed all tasks that were previously assigned to it
  - Completion time, ct(i,j) = mat(j)+ETC(i,j)
- Objective – find a schedule with minimum **makespan**
- Makespan – max (ct(i,j))

Scheduling Heuristics

- Opportunistic Load Balancing (OLB)
  - Assign next task (arbitrary order) to the next available machine
  - Regardless of task's ETC on that machine
- User Directed Allocation (UDA)
  - Assign next task (arbitrary order) to the machine with lowest ETC
  - Regardless of machine availability

Scheduling Heuristics

- Min-Min
  - Start with a list of Unmapped tasks, U.
  - Determine the set of minimum completion times for U.
  - Choose the next task that has min of min completion times and assign to the machine that provides the min. completion time.
  - The new mapped task is removed from U and the process is repeated.
  - Theme - Map as many tasks as possible to their first choice of machine
  - Since short jobs are mapped first, the percentage of tasks that are allocated to their first choice is high

Scheduling Heuristics

- Max-Min
  - Start with a list of Unmapped tasks, U.
  - Determine the set of minimum completion times for U.
  - Choose the next task that has max of min completion times and assign to the machine that provides the min. completion time.
  - The new mapped task is removed from U and the process is repeated.
  - Avoids starvation of long tasks
  - Long tasks executed concurrently with short tasks
  - Better machine-utilization

Scheduling Heuristics

- Genetic Algorithm
- General steps of GA

```
initial population generation;
evaluation;
while (stopping criteria not met) {
        selection;
        crossover;
        mutation;
        evaluation;
}
```

GA

- Operates 200 chromosomes. A chromosome represents a mapping of task to machines, a vector of size t.
- Initial population – 200 chromosomes randomly generated with 1 Min-Min seed
- Evaluation – initial population evaluated based on fitness value (makespan)
- Selection –
  - Roulette wheel – probabilistically generate new population, with better mappings, from previous population
  - Elitism – guaranteeing that the best solution (fittest) is carried forward

GA - Roulette wheel scheme

Chromosomes	1	2	3	4
Score	4	10	14	2
Probability of selection	0.13	0.33	0.47	0.07

Select a random number, r, between 0 and 1.

Progressively add the probabilities until the sum is greater than r

GA

- Crossover
  - Choose pairs of chromosomes.
  - For every pair
    - Choose a random point
    - exchange machine assignments from that point till the end of the chromosome
- Mutation. For every chromosome:
  - Randomly select a task
  - Randomly reassign it to new machine
- Evaluation
- Stopping criterion:
  - Either 1000 iterations or
  - No change in elite chromosome for 150 iterations

Simulated Annealing

- The procedure is similar to metal annealing/formation process
- Poorer solutions accepted with a probability that depends on temperature value
- Initial mapping; Initial temperature – initial makespan
- Each iteration:
  - Generate new mapping based on mutation of prev. mapping. Obtain new makespan
  - If new makespan better, accept
  - If new makespan worse, accept if a random number z in [0,1] > y where

$$y = \frac{1}{1 + e^{\left(\frac{\text{old makespan} - \text{new makespan}}{\text{temperature}}\right)}}.$$

  - Reduce temperature by 10%

Tabu search

- Keeps track of regions of solution space that have already been searched
- Starts with a random mapping
- Generate all possible pairs of tasks, (i,j), i in (0, t-1) and j in (i+1, t)
- i and j's machine assignments are exchanged (short hop) and makespan evaluated
- If makespan better (successful short hop), search begins from i=0, else search continues from previous (i,j)

Tabu search

- Continue until 1200 successful short hops or all pairs have been evaluated
- Add final mapping to tabu list. The list keeps track of solution space searched
- A new random mapping generated that differs from solution space by atleast half the machine assignments (long hop)
- Search continued until fixed number of short and long hops