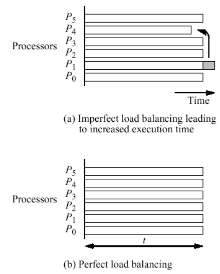
# Load Balancing, Scheduling, & Termination Detection

Reading: PP Chapter 7

## Load-Balancing Rationale




## Load-Balancing Dichotomies

- Static Load-Balancing
  - Pre-planned
- Dynamic Load-Balancing
  - Adapts as computation progresses
  - Variations:
    - Centralized
    - Decentralized
      - Centralized control
      - Distributed control
- Communication-sensitive or not
- Generic vs. Application-Specific

## Static Load Balancing Methods

- Deterministic scheduling to minimize completion time, e.g. based on fixed **priority**
- Random *a priori* distribution
- Round robin (aka cyclic mapping: "deal" the load out to the processors)
- Recursive bisection: recursively split load into two, with half going to half the processors
- Simulated annealing, genetic programming

## Deterministic Scheduling

- Problem is NP-hard for all but the most trivial classes of assumptions.
- Assumes times of tasks (and communication) are known, which they often aren't.
- Unexpected scheduling anomalies.

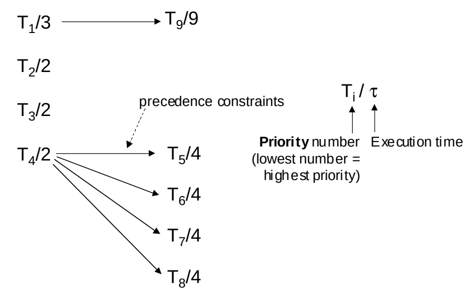
## Scheduling Anomalies (R.L. Graham, 1960's)

- The following are expected to reduce overall execution time:
  - Reducing execution times of individual tasks
  - Relaxing precedence constraints between tasks
  - Adding more processors

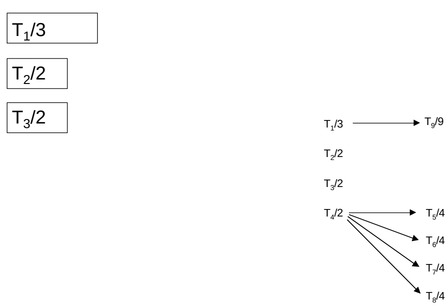
## Scheduling Anomalies
## (R.L. Graham, 1960's)

- The following are expected to reduce overall execution time:
  - Reducing execution times of individual tasks
  - Relaxing precedence constraints between tasks
  - Adding more processors
- For some algorithms, these can actually *increase* the execution time.

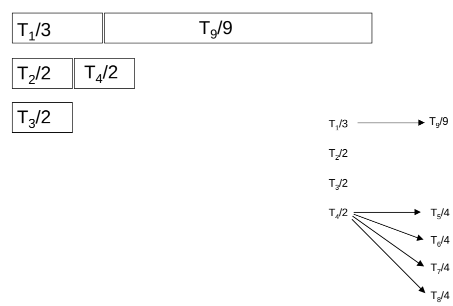
## Priority Scheduling Anomalies



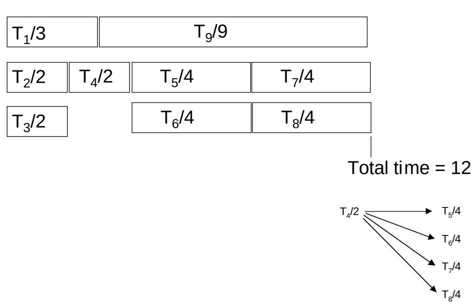

## Consider Scheduling on 3 processors



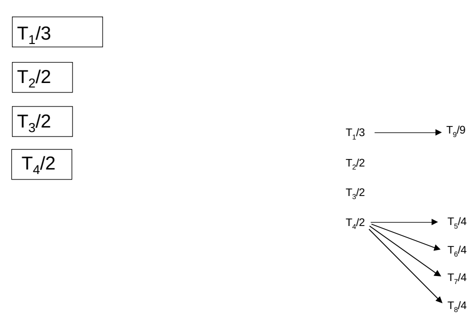

## Consider Scheduling on 3 processors



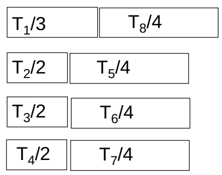

## Consider Scheduling on 3 processors

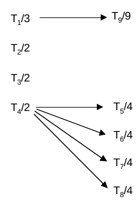



## Consider Scheduling on 4 processors

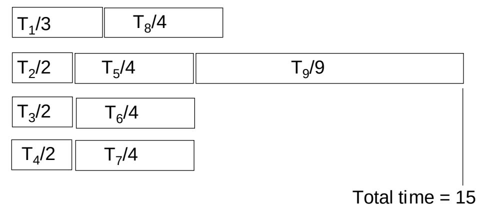


## Consider Scheduling on 4 processors

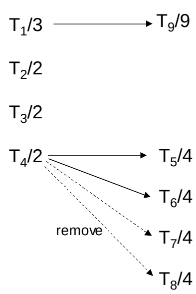


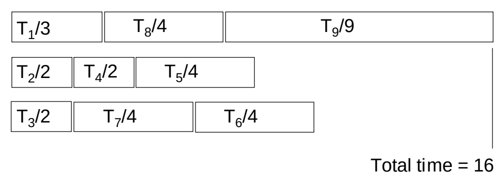

## Consider Scheduling on 4 processors




## Consider Relaxing Constraints




## Consider the relaxed constraints on 3 processors




## Cause of Anomalies

- Obviously the anomalies are caused by the use of the **priority rule** in scheduling:
  - This rule is cheap to implement (O(n)).
  - It does not take into account optimizations that would be possible by violating strict priority.
- In general, finding true optimum would entail a search, which tends to be much more expensive.

## Bounds on Anomalies
(due to R.L. Graham)

- Let ' designate times for system with relaxed constraints and shorter individual times. Then
- Time(m)/ Time(m') $\leq$ 1 + (m-1)/m',
  where m' $\geq$ m are numbers of processors.
- Example: Time(2)/Time(3) $\leq$ 4/3.
- Worst case: Time(m)/Time(m') < 2.

## More on Scheduling

- Later, when we discuss real-time.

## Dynamic Load-Balancing

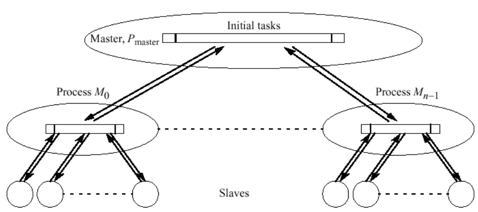
- Centralized: Processors go to a centralized work pool to get more work (e.g. in the work pool model). Also called **self-scheduling**.
- Decentralized: The work is distributed by some other method

## Pro's and Con's of Centralized

## Decentralized

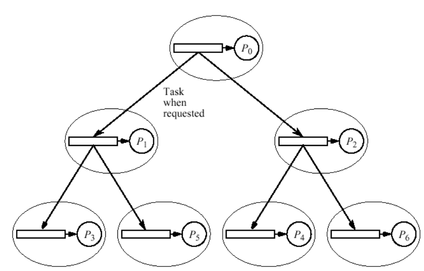
- Distributed pool
  - Processors go to pool to get work
  - Processors distribute extra work to pools
  - May necessitate rebalancing
- Push or Pull ("Fully-distributed")
  - Push: Heavily-loaded processors push their work onto other processors.
  - Pull: Lightly-loaded processors go to other processors to get work.

## Distributed Work-Pool




## Tree Method

(e.g. Keller, Lindstrom, & Patil 1979)



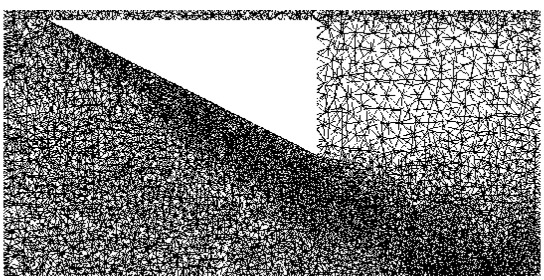
## Gradient Method
(Lin & Keller 1987)

- Tasks are like **molecules of a fluid**.
- Fluid flows from high-pressure to low-pressure areas.
- **Intelligent switches** in processing elements can multiplex load balancing along with communication traffic.
- **Distributed pressure metric** steers along shortest path to under-loaded node.
- Parallel

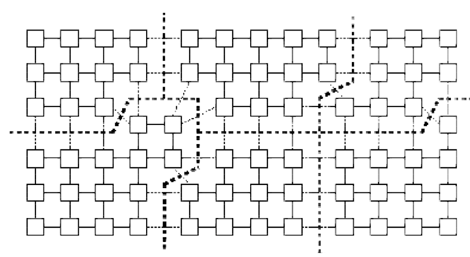
## Other Methods

- ACWN method (Kale', 1988)
- Seed-in-the-wind method
- Application-specific methods:
  - Partitioning using grids and graphs:
    - PIC computations
    - FEM computations

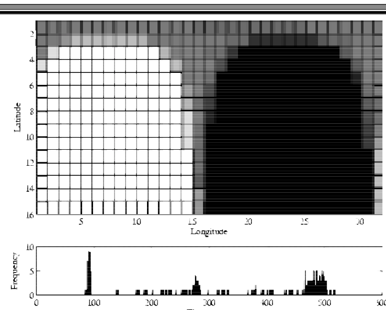
## FEM mesh example



## Performance Comparisons: Decentralized Load-Balancing for a Grid Problem
(from Foster's DBPP)



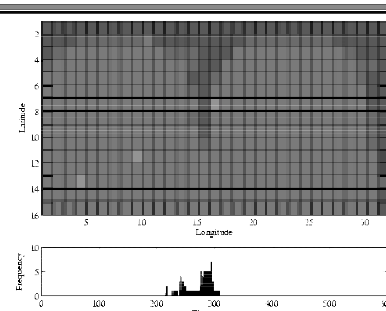
## Load Distribution without Load Balancing (Physical grid, 16x32 procs)




Processor time-distribution histogram

diverse

## Load Distrubution *with* Load Balancing (cyclic-mapping algorithm)




Processor time-distribution histogram

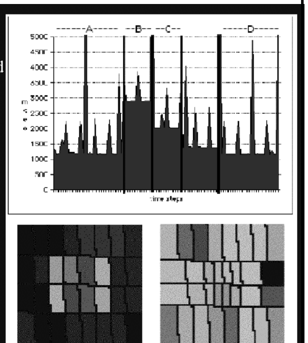
more uniform

Example of dynamic load balancing in an atmospheric model: rebalancing a “hot spot”: http://www-unix.mcs.anl.gov/~michalak/daoslides/

- Time steps from 8-hour run on 32 processors with an induced load imbalance between 2.5 and 4.5 hours (B, C).
- Efficiency ($T_{mean} / T_{max}$):

| | | |
|---|---|---|
| A: .95 | B: .50 | C: .96 |
| C/D: .78 | D: .95 | |

- Calculate remap: 2.5 seconds
- Full rebalancing: 12 seconds

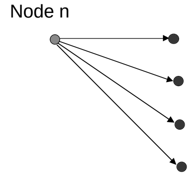


B. C.

## Example from PP sec. 7.4

- Single-source shortest path on a directed graph
- Find the distances from a designated node to all other nodes
- Possibilities:
  - Moore’s algorithm
  - Dijkstra’s algorithm
  - Floyd’s algorithm (for all sources)

## Nomenclature




## Moore’s Algorithm

- Upper bounds on distances are maintained.
- Nodes are visited by selecting from a queue, initially containing the single source.
- As a node is visited, the upper bounds on its targets are updated. If an update results in an improvement, the node is re-enqueued.

## Sequential timing for Moore

- Each enqueuing may entail updating of up to n nodes.
- Claim: A node won’t be enqueued more than $O(n^2)$ times.
- Therefore this algorithm is $O(n^3)$.

## PP 7.4 uses the Moore Algorithm

- A work pool model is used.
- A work unit is a node to be visited.
- The work pool is therefore the same as the queue.
- The work pool can be centralized or distributed.

## Distributed Moore Algorithm

- The authors suggest one process per node, which maintains the upper bound distance on the node.
- When the node is updated, it will notify the processes of its target nodes, so that they can similarly update.
- So there is no actual queue, just processes waiting for updates.

## Dijkstra's Algorithm

- Upper bounds on distance-from-source are maintained for all nodes.
- Nodes are "retired" in succession, starting with the source. When a node is retired, its upper bound is provably the shortest path from the source.
- Each time a node is retired, the distance-from-source upper bounds of its unretired targets are updated.
- The unretired node with the smallest distance is the next chosen for retirement.

## Sequential Timing for Dijkstra

- n (number of nodes) iterations, each iteration has to find min of up to n unretired nodes and update up to n targets: $O(n^2)$
- If the graph is "sparse", meaning a constant upper bound on fan-out for all graphs, then the update step is O(1). Finding min can be done in O(log n), so for the **sparse** case: O(n log n)

## Parallel Dijkstra?

## All-Sources Shortest Paths

- Repeated parallel matrix multiplication
  - starting with connection matrix + I (using + min rather than + *):
  - log n matrix multiples, each $O(n^3)$ $\rightarrow O((n^3 \log n)/p)$.

## All-Sources Shortest Paths

- Parallel Floyd's method
  - Triply-nested loops, O(n) iterations each
  - **Middle** nest can be done in parallel
    - After each outer iteration, **broadcast** to all processors (counting O(1) for broadcast)
    - $t_{comp}*n^3/p + t_{comm}*n^2$
  - Inner **two** nests can be done in parallel
    - $t_{comp}*n^3/p + t_{comm}*n^2/sqrt(p)$

## All-Sources Shortest Paths

- Multiple Dijkstra's algorithms can be run in parallel for different sources.
- There is no communication cost.
- However, multiple sequential Dijkstra for dense graphs is slower than sequential Floyd.
- If communication is expensive, or the graph is sparse, this could be a win.

## Termination Issue

- With the distributed Moore algorithm, there is the issue of determining when the computation is done.
- Sufficient conditions are:
  - Every process must be idle.
  - There can be no messages in transit.
- Why only sufficient?

## Termination Issue

- In general, but not for Moore's algorithm, the system might have the answer but there are still non-idle processes or live messages.
- One way to address is to have a **master process** notify the others to shut down, e.g. by sending a high-priority message.
- There would need to be a way of ignoring messages arriving after the shutdown, and shutdown would have to acknowledge to the master.

## PP 7.3 Distributed Termination

- **Acknowledgment messages method**
  - Messages originally emanate from a single node.
  - Sending a message **that makes a node active** imposes an implicit **tree structure** on the processes.
  - Normally all messages are acknowledged, but a parent is acknowledged only **when the child goes from active to idle**.
  - When all of a node's children have acknowledged and there is no more processing, the node becomes idle itself.
  - The computation is done when the original node becomes idle.

## PP 7.3 Distributed Termination

- **Ring Termination Method**
  - Processes are arranged in a **virtual ring structure**, $P_0$, …., $P_{n-1}$
  - When $P_0$ terminates, it sends a ready-to-shutdown token to $P_1$.
  - When $P_i$ receives a token, it holds it until it terminates, then passes it to $P_{i+1(\text{mod } n)}$.
  - When $P_0$ receives a token, all $P_i$ have terminated.
  - $P_0$ then sends a shut-down to all processes.

## PP 7.3 Distributed Termination

- Ring Termination Method previously described ***assumes*** a process **cannot be reactivated once it has terminated**.
- In a work pool model, termination is analogous to the local pool being empty.
- But we could have a situation where a "terminated" process gets reactivated.
- How can we detect global termination in such a situation?

# PP 7.3 Distributed Termination

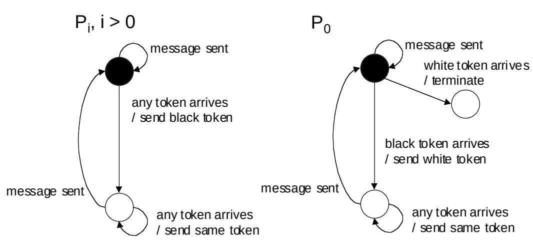
- Ring Termination with Reactivation
  - If terminated $P_i$ receives a reactivation message from $P_j$, $j > i$, and has not yet received a **ready-to-shutdown** token, then there is no problem.
  - However, if $P_i$ has ***already*** received such a **ready-to-shutdown** token and passed it on, the pending shutdown has to be ignored by $P_i$ .

# Ring Termination with Reactivation

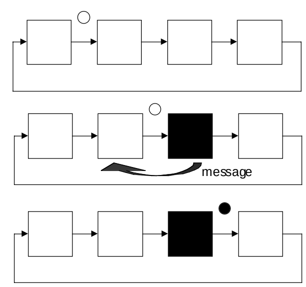
- Processes **and** tokens are colored black or white.
- **Symbolism:**
  - **black** signifies that a message has been sent since the last token was forwarded.
  - **white** signifies that no message was sent.
- To start, $P_0$ sends a white token to $P_1$.
- For general $i > 0$:
  - If $P_i$ is white, it passes the token unchanged.
  - If $P_i$ is black, it changes the token to black and passes it.
- If a white process subsequently sends a message (other than a token) to another process, it changes to black.
- Any token arriving at a node changes the node to white.
- Eventually a token returns to $P_0$:
  - If $P_0$ receives a white token, then final shutdown takes place.
  - If $P_0$ receives a black one, it again sends a white token.

# Ring Termination with Reactivation

- The following state-machine tries captures the essence of the node's behavior:




# Simulation, 1 of 3

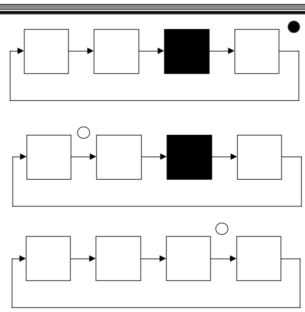



A white token circulates from the initial node. White nodes leave the token color unchanged, but change the node to white.

When a node sends a message, it changes to black.

A black node changes any token to black.

# Simulation, 2 of 3

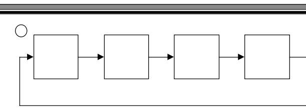


A black token stays black until it gets to the original node.

The token arriving at the first processor as black is recirculated as white.

A token arriving at a black node changes it to white.

# Simulation, 3 of 3



A white token arriving at the first node indicates termination.

# Petri Net Model

- Board lecture