Load Balancing, Scheduling, & Termination Detection

Reading: PP Chapter 7

Load-Balancing Rationale

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

```
T_{1/3}  T_{3/2}  T_{2/2}
T_{1/3}  T_{4/2}  T_{4/2}  T_{4/2}
T_{1/3}  T_{9/9}
```

Total time = 12

Consider Scheduling on 4 processors

```
T_{1/3}  T_{3/2}  T_{2/2}  T_{4/2}
T_{1/3}  T_{9/9}  T_{4/2}  T_{4/2}  T_{4/2}  T_{4/2}
T_{1/3}  T_{9/9}  T_{4/2}  T_{4/2}  T_{4/2}  T_{4/2}
```

Total time = 12
Consider Scheduling on 4 processors

\[ \begin{align*}
T_1/3 & \quad T_9/9 \\
T_2/2 & \quad T_9/4 \\
T_3/2 & \quad T_8/4 \\
T_4/2 & \quad T_7/4 \\
T_5/4 & \quad T_6/4 \\
T_6/4 & \quad T_5/4 \\
T_7/4 & \quad T_4/2 \\
T_8/4 & \quad T_3/2 \\
T_9/9 & \quad T_2/2 \\
\end{align*} \]

Consider Scheduling on 4 processors

\[ \begin{align*}
T_1/3 & \quad T_9/9 \\
T_2/2 & \quad T_9/4 \\
T_3/2 & \quad T_8/4 \\
T_4/2 & \quad T_7/4 \\
T_5/4 & \quad T_6/4 \\
T_6/4 & \quad T_5/4 \\
T_7/4 & \quad T_4/2 \\
T_8/4 & \quad T_3/2 \\
T_9/9 & \quad T_2/2 \\
\end{align*} \]

Total time = 15

Consider Relaxing Constraints

\[ \begin{align*}
T_1/3 & \quad T_9/9 \\
T_2/2 & \quad T_9/4 \\
T_3/2 & \quad T_8/4 \\
T_4/2 & \quad T_7/4 \\
T_5/4 & \quad T_6/4 \\
T_6/4 & \quad T_5/4 \\
T_7/4 & \quad T_4/2 \\
T_8/4 & \quad T_3/2 \\
T_9/9 & \quad T_2/2 \\
\end{align*} \]

Consider the relaxed constraints on 3 processors

\[ \begin{align*}
T_1/3 & \quad T_9/4 \\
T_2/2 & \quad T_9/4 \\
T_3/2 & \quad T_8/4 \\
T_4/2 & \quad T_7/4 \\
T_5/4 & \quad T_6/4 \\
T_6/4 & \quad T_5/4 \\
T_7/4 & \quad T_4/2 \\
T_8/4 & \quad T_3/2 \\
T_9/9 & \quad T_2/2 \\
\end{align*} \]

Total time = 16

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

\[ \begin{align*}
\text{Let } T' & \text{ designate times for system with relaxed constraints and shorter individual times. Then } \\
\frac{T(m)}{T(m')} & \leq 1 + \frac{m-1}{m'}, \\
\text{where } m' & \geq m \text{ are numbers of processors.} \\
\text{Example: } & \frac{T(2)}{T(3)} \leq 4/3. \\
\text{Worst case: } & T(m)/T(m') < 2. \\
\end{align*} \]

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.
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

- 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)

Load Distribution with Load Balancing (cyclic-mapping algorithm)
Example of dynamic load balancing in an atmospheric model: rebalancing a "hot spot": http://www-unix.mcs.anl.gov/~michalak/daoslides/

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?

- 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)
    - \( t_{\text{comp}} \cdot n^2/p + t_{\text{comm}} \cdot n^2 \)
  - Inner two nests can be done in parallel
    - \( t_{\text{comp}} \cdot n^2/p + t_{\text{comm}} \cdot n^2/\sqrt{p} \)

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)
    - \( t_{\text{comp}} \cdot n^2/p + t_{\text{comm}} \cdot n^2 \)
  - Inner **two** nests can be done in parallel
    - \( t_{\text{comp}} \cdot n^2/p + t_{\text{comm}} \cdot n^2/\sqrt{p} \)
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 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?
Ring Termination with Reactivation

- 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 $P_i$.

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.

**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.

**Simulation, 2 of 3**

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

**Simulation, 3 of 3**

A white token arriving at the first node indicates termination.
<table>
<thead>
<tr>
<th>Petri Net Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Board lecture</td>
</tr>
</tbody>
</table>