Distributed Scheduling

Goal: enable transparent execution of programs on networked computing systems

Motivations: reduce response time of program execution through load balancing

An aspect of current interest in “metacomputing” systems
• globus
• legion

Opportunities for Task Distribution

![Graph showing the relationship between server utilization and the probability of task distribution. The graph peaks at a server utilization of around 0.6, indicating the optimal condition for task distribution.](graph)
In lightly loaded systems there is not much opportunity for task distribution because most servers are underutilized.

In heavily loaded systems there is not much opportunity for task distribution because no server is free to accept a task.
Task Distribution

In moderately loaded systems there are good opportunities to distribute tasks from over-utilized to under-utilized systems.

Characteristics of Approaches

Goals:
- load sharing (distribute load) vs.
- load balancing (equalize load)

Information:
- static (invariant of system state)
- dynamic (uses system state)
- adaptive (changes actions with system state)

Transfers:
- preemptive (interrupts task for transfer) vs.
- non-preemptive (transfers only new tasks)
Component Policies

• **Transfer** determines whether a node is in a state to participate in load transfers and in what role

• **Selection** determines which local task is involved in the transfer

• **Location** determines a pair of nodes to participate in task transfer

• **Information** determines what information is collected and how
  
  • demand-driven (obtained when needed)
  • periodic (at regular intervals)
  • state-change-driven (obtained when nodes change state)

Kinds of Algorithms

**sender-initiated**: an overloaded node searches for a underloaded node to take one of its tasks

location policies: random, polling-first found, polling-least loaded
stability: unstable/ineffective at high system loads

**receiver-initiated**: an underloaded node searches for a task to take from an overloaded node

location policies: random, polling
stability: stable at high system loads
drawback: uses preemptive transfers in many cases

**symmetrically-initiated**: senders and receivers search for each other
Above-Average Algorithm

This node’s estimate of the system's average load

upper threshold (UT)

lower threshold (LT)

*thresholds equidistant from average

Basic Step

if (receiver)
    { increment load;
      send accept;
    }

if (still sender)
    { send task;
    }

broadcast
Basic Step

- If (still sender), broadcast
- If (LT), too low
- If (UT), too high

Timers

- Start timer if (UT)
- If (timer expires), broadcast
- If (still sender)
  - TooHigh
- If (LT)
  - TooLow
Timers

A Stable, Symmetrically Initiated Algorithm

Transfer Policy:

Load is measured by CPU queue length

sender/overloaded

OK

receiver/underloaded
Stable, Symmetrically Initiated Algorithm

Each node maintains three lists that are searched in the following orders:

1. receiver search
2. OK search
3. sender search

Sender Protocol

```c
Sender Protocol

sender_i -> poll node at head of receiver list

if (state_j == receiver) {
    send task;
    done;
} else {
    put j on head of sender or OK list depending on state j
}

receiver_j -> poll from i

state_j

reply current state

task

Sender continues polling until receiver list empty or task is transferred.
```
Receiver Protocol

```
if (load > UT)
    send task;
else
    put j at head of receiver list;

send current state
```
A Stable Sender-Initiated Algorithm

Similar to previous algorithm except that it has a modified receiver protocol. Each node maintains a state vector, SV, indicating on which list the node is on at all other nodes.

<table>
<thead>
<tr>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>on node i SV: sender/receiver/OK</td>
</tr>
</tbody>
</table>

Note: the movement of node i to a different list on node j can only occur as a result of an interaction between nodes i and j. Thus, it is possible for node i to keep its information current.

Sender Protocol

Sender continues polling until receiver list empty or task is transferred.
Receiver Protocol

sender$_i$  

receiver$_j$

when load < LT then:

for all i:

if (SV[i] != receiver)

{ send update;
  set SV[i] = receiver;
}

put j at head of receiver list;

j is receiver

Note: receiver only informs selected nodes of its status change.

Advantages

The sender-initiated algorithm:

• avoids broadcasting of receiver state
• does not transfer preempted tasks (because it is sender-initiated)
• is stable (as for previous algorithm)
Selecting a Scheduling Algorithm

<table>
<thead>
<tr>
<th>Condition</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>no high loads</td>
<td>sender-initiated</td>
</tr>
<tr>
<td>has high loads</td>
<td>stable algorithm</td>
</tr>
<tr>
<td>wide fluctuations</td>
<td>stable symmetric</td>
</tr>
<tr>
<td>wide fluctuations and high migration cost</td>
<td>stable sender-initiated</td>
</tr>
</tbody>
</table>