Depth First Search and Dynamic Load Balancing

Sathish Vadhiyar
Parallel Depth First Search

- Easy to parallelize
- Left subtree can be searched in parallel with the right subtree
- Begin as BFS; Statically assign a node to a processor - the whole subtree rooted at that node can be searched independently.
- Can lead to load imbalance; Load imbalance increases with the number of processors (more later)
Dynamic Load Balancing (DLB)

- Difficult to estimate the size of the search space beforehand
- Need to balance the search space among processors dynamically
- In DLB, when a processor runs out of work, it gets work from another processor
Maintaining Search Space

- Each processor searches the space depth-first
- Unexplored states saved as stack; each processor maintains its own local stack
- Initially, the entire search space assigned to one processor
- When a processor’s local stack is empty, it requests untried alternative from another processor’s stack
Work Splitting

- When a processor receives work request, it splits its search space.
- **Half-split**: Stack space divided into two equal pieces - may result in load imbalance.
- Giving stack space near the bottom of the stack can lead to giving bigger trees.
- Stack space near the top of the stack tend to have small trees.
- To avoid sending very small amounts of work - nodes beyond a specified stack depth are not given away - **cutoff depth**.
Strategies

1. Send nodes near the bottom of the stack
2. Send nodes near the cutoff depth
3. Send half the nodes between the bottom of the stack and the cutoff depth

Example: Figures 11.5(a) and 11.9
Load Balancing Strategies

- **Asynchronous round-robin**: Each processor has a target processor to get work from; the value of the target is incremented with modulo.

- **Global round-robin**: One single target processor variable is maintained for all processors.

- **Random polling**: Randomly select a donor.
Termination Detection

- As processors search independently, how will they know when to terminate the program?
- Two strategies
  - Dijkstra’s token based
  - Tree-based
Termination Detection

- Dijkstra’s Token Termination Detection Algorithm
  - Based on passing of a token in a logical ring; PO initiates a token when idle; A processor holds a token until it has completed its work, and then passes to the next processor; when PO receives again, then all processors have completed.
  - However, a processor may get more work after becoming idle.
Algorithm Continued....

- Taken care of by using white and black tokens
- A processor can be in one of two states: black and white
- Initially, the token is white; all processors are in white state
Algorithm Continued....

- If a processor P_j sends work to P_i (i<j), the token must traverse the ring again.
- A processor j becomes black if it sends work to i<j.
- If j completes work, it changes token to black and sends it to next processor; after sending, changes to white.
- When P_0 receives a black token, reinitiates the ring.
Tree Based Termination Detection

- Uses weights
- Initially processor 0 has weight 1
- When a processor transfers work to another processor, the weights are halved in both the processors
- When a processor finishes, weights are returned
- Termination is when processor 0 gets back 1
- Goes with the DFS algorithm; No separate communication steps
- Figure 11.10