Review

Discrete optimizations
- Depth first search algorithms
  - Backtrack
  - Branch and Bound
  - A* and IDA*

Parallel implementations:
- Shared memory: maintain two lists of nodes to expand
  - open: not explored
  - closed: explored

Message passing (or distributed)
- Open and closed list are not shared
- May either exchange these lists periodically or use work stealing
- Request (or send) half of the open list at other nodes

Which node to request (or send)
- Asynchronous Round Robin (ARR): each node determines the order to request work from other nodes
- Global Round Robin (GRR): The order of nodes approaches is global
- Random: Each node randomly selects next node to request
CSCE 5160 Parallel Processing

One observation. Parallel Search Overhead

Parallel implementations may do more work because they may search more nodes in the tree.

If \( W_s \) is the work done by one processor, we can equate the work to the number of tree nodes searched \( T_s \).

Likewise \( W_p \) is the work done by \( p \) processor and we an equate this to \( T_p \) as the number of nodes searched.

If \( T_p / T_s < 1 \); we get superlinear speedup.

Otherwise we get less than \( p \) times speedup.

Note this does not include other overheads such as communication and synchronization.

We can still use scalability analyses (Iso-efficiency functions etc)

CSCE5160 April 29, 2019

CSCE 5160 Parallel Processing

How to measure performance of parallel implementation?

Communication overhead (ignoring the size of message at each communication) or the size of open list exchanged.

\( V(R) \) \( \rightarrow \) number of request before each node received at least one request to share list.

Total number of request before a node’s list is too small = \( \log(W) \)

Communication overhead = \( T_o = t_{comm} \times V(p) \times \log(W) \)

\( V(p) \) for ARR = \( O(p^2) \) (worst case)

\( V(p) \) for GRR = \( O(p) \)

\( V(p) \) for random = \( O(p \log p) \) (probabilistic average)
CSCE 5160 Parallel Processing

We are assuming that the work distribution is in fixed-size messages (this can be modified for more complex analysis).

In a hypercube, the average distance between any pair of processors is log p.
So, $t_{comm} = O(\log p)$

Since now we know both $t_{comm}$ and $V(p)$ we can try to solve the iso-efficiency functions.

Note the overhead in parallel computation $T_o = t_{comm} * V(p) * \log (W)$

Iso-efficiency functions:

- Asynchronous Round Robin: $O(p^2 \log^2 p)$
- Global Round Robin: $O(p \log^2 p)$
- Random: $O(p \log^3 p)$

Other ways of stealing work from neighbors

Problem 11.2:

When a processor becomes idle, it selects two processors $p_i$ and $p_{i+1}$ such that the counter at $p_i$ is greater than that at $p_{i+1}$. And the idle processor sends work request to $p_{i+1}$ and $p_{i+1}$ increments its counter if it received more work distributions than $p_{i+1}$.

How to find the counter values. We can consider constructing a logical tree where the leaf nodes (processors) are ordered according the numbers.

We need $O(\log p)$ to find $p_i$ and $p_{i+1}$.

How do we compute $V(p)$ – number of requests by which all processors receive one request?

Note that if $p_{i+1}$ is selected, its counter is incremented (and likely to make it the same as the counter of $p_i$).
And $p_i$ must have received a request previously.

So $V(p) = p$ – the same as Global Round Robin
Consider processors in a Hypercube. We can randomly select nodes that are at a distance \( i \) from requesting node; \( i \) is varied from 1 to \( \log p \).

\[
\text{# nodes at a distance if } i: \text{ that is } i \text{ bits out of } d \text{ bits are different}
\]

\[
\binom{n}{i} = \frac{n!}{i!(n-i)!}
\]

As with random polling, we need to find estimated number of requests such that all processors at a distance \( i \) receive a request. Repeat this for different \( i \) values.

So we have

\[
V(p) = \sum_{i=1}^{d} n_i \log n_i
\]

\( n_i \) is the number of nodes at a distance \( i \) (that is shown above).

Using the simplified equation for random polling for \( V(p) = p \log p \)

For example if \( p = 16, d = 4 \)

\[
V(p) = 4 \log 4 + 6 \log 6 + 4 \log 4
\]

If \( p = 32, d = 5 \)

\[
V(p) = 5 \log 5 + 10 \log 10 + 10 \log 10 + 5 \log 5
\]

Another issue to consider in parallelizing search algorithms is the detection of termination. How do we find termination (when all processors are idle)? Simple Token passing algorithms (by Dijkstra).

If we assume that once a processor becomes idle, it stays idle. \( p_i \) sends a token when it becomes idle to \( p_{i+1} \). Token starts with \( p_0 \) and when the token from the last processor returns to \( p_0 \) which indicates that all processors are idle.
Dirty and Clean tokens (and dirty and clean processors).
When \( p_0 \) becomes idle, it initiates the termination detection by sending clean token around \( p_0 \) is marked clean.
If \( p_j \) sends work to \( p_i \) and \( j > i \); \( p_j \) is marked dirty.
If \( p_i \) has the token (either dirty or clean) and becomes idle, passes token to \( p_{i+1} \).
If \( p_i \) is dirty then a dirty token is sent; \( p_i \) becomes clean.
If \( p_i \) is clean then a clean token is sent; \( p_i \) stays clean.
If \( p_0 \) receives a clean token and \( p_0 \) itself is clean and idle, termination!

What is the complexity of this algorithm?
If we use simple Dijkstra algorithm: \( O(p) \)
What about the extended algorithm?
The token may have to go around \( p-1 \) times and each time it has to go to \( p \) processors = \( O(p^2) \)

Let us consider best first algorithms (Section 11.5)
Note we explore best possible node to expand.
In sequential implementation, we can maintain a queue of open nodes sorted
or best node to expand is at the head of the queue.

How do we parallelize this?
We can have top \( p \) nodes of the open nodes explored in parallel.
This may waste processing since we may be exploring less than best nodes.

Should we periodically exchange best nodes (all to all communication)?
All processors now have updated best open nodes.
The frequency of exchange is a tradeoff between communication and wasted computation.
Instead of all to all to all may be we should exchange best open nodes among a smaller group of nodes.
CSCE 5160 Parallel Processing

Random: Each processor sends its best nodes to a randomly selected node
Ring: each processor sends best nodes to its node in the logical ring

Or blackboard: All best nodes are posted on the shared memory

Note the search space could take the form of a tree or a graph
- tree like basically means we discard children of a node that is discarded
- graph like means that the children are not deleted because a different path may exist

Even for Graph search, we keep open and closed node lists
- Open nodes: not yet expanded
  - pick the best node from open – move the node to closed
  - and place expanded nodes on the open list

If we parallelize such graph search nodes, it is possible that more than one processor may expand the same node – replication of work

Need to avoid this.

We can assign nodes to specific processors
- we can use a mapping function to assign a node
  (created when other nodes are expanded)

So, when a processor removes a node from open list and places it on the closed
- all the processors assigned to that node will be notified

However, this adds to the communication
Chapter 12: Dynamic Programming
or discrete optimization

Examples: Traveling salesman problem
Finding common subsequences (Bioinformatics)
Pattern matching and pattern discovery
Knapsack problem

In general we can think of dividing the problem into sub-problems
As we discussed, we can have AND or OR sub-problems

We describe our problem as
\[ r = g[f(x_1), f(x_2), \ldots, f(x_n)] \]

In general \( f(x_i) \) itself can be decomposed recursively.
\[ f(x_i) = g[h(y_1), h(y_2), \ldots, h(y_m)] \]

Monadic problems will have only one level of decomposition
Polyadic problems can have multiple levels of recursion or decomposition.

Likewise, if at any level of decomposition, if the solution of a sub-problem only depends on the solution to the decomposition of the problem at next level then we call them Serial. Otherwise we call them non-serial

So we have 4 different classes of Dynamic programming problems

| Serial Monadic | Non-Serial Monadic |
| Serial Polyadic | Non-Serial Polyadic |

Serial Monadic Problems: Shortest Path
We can describe the shortest path from source to destination as sub-problems
At each level we can think of finding shortest paths of length \( i \).

But a more general formulation is shown on page 518 figure
We are trying to find shortest paths from S to R. Let us label a node in level \( l \) as \( v^l_i \).

We can also describe the cost of going from a node \( i \) in level \( l \) to a node \( j \) in level \( l+1 \) as \( c^l_{i,j} \).

Let us use \( C^l_i \) as the shortest path from node \( i \) in level \( l \) to R and 
\[
C^l = [C^l_0, C^l_1, C^l_2, \ldots, C^l_{n-1}]
\]

So what we need to find for solving shortest path from S to R is \( C^0 \).

We can start from the last level. Note there will be only one link from the last level nodes to R.

We can start from the last level. Note there will be only one link from the last level nodes to R.

In other words, we check the minimum cost of going from any node in level \( l+1 \).

We can start from the last level. Note there will be only one link from the last level nodes to R.

\[
C^l = [c^{l+1}_0, c^{l+1}_1, \ldots, c^{l+1}_{n-1}]
\]
CSCE 5160 Parallel Processing

Now we can recurse back to level \( r-1, r-2, \ldots \) to level 0

Consider the following

\[
C^0_0 = \min \{(c_{0,0}^0 + C^1_0), (c_{0,1}^0 + C^1_1), \ldots, (c_{0,n-1}^0 + C^1_{n-1})\}
\]

\[
C^1_0 = \min \{(c_{1,0}^1 + C^2_0), (c_{1,1}^1 + C^2_1), (c_{1,2}^1 + C^2_2), \ldots, (c_{1,n-1}^1 + C^2_{n-1})\}
\]

\[
C^1_1 = \min \{(c_{2,0}^1 + C^2_0), (c_{2,1}^1 + C^2_1), (c_{2,2}^1 + C^2_2), \ldots, (c_{2,n-1}^1 + C^2_{n-1})\}
\]

\[
C^1_{n-1} = \min \{(c_{n-1,0}^1 + C^2_0), (c_{n-1,1}^1 + C^2_1), \ldots, (c_{n-1,n-1}^1 + C^2_{n-1})\}
\]

We can actually write this as matrix * vector products

\[
C^i = M_{i,i+1} * C^{i+1}
\]

Note \( M_{i,i+1} \) looks like

\[
\begin{array}{cccc}
C^i_{0,0} & C^i_{0,1} & \cdots & C^i_{0,n-1} \\
C^i_{1,0} & C^i_{1,1} & \cdots & C^i_{1,n-1} \\
& & \cdots & \\
C^i_{n-1,0} & C^i_{n-1,1} & \cdots & C^i_{n-1,n-1}
\end{array}
\]

CSCE 5160 Parallel Processing

Unlike matrix * vector, we replace the addition is replaced by “\( \min \)” and multiplication is replaced by “addition”

\[
A^\times x = [a_{00}x_0 + a_{01}x_1 + \ldots + a_{0n-1}x_{n-1}] + [a_{10}x_0 + a_{11}x_1 + \ldots + a_{1n-1}x_{n-1}],
\]

\[
\ldots + [a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1}]
\]

Consider an example

Let us try to find shortest path from a to e

Let us compute cost backwards

The last stage to reach e
Now consider reaching the nodes in the penultimate level from b, c, d, f

Note I am using minimum cost from each node at the penultimate level to e

Continuing to next level

I am only using minimum costs from previous level
I deleted actual paths and kept only the costs
Shortest path from a to e is 4 (by going through c)
How to parallelize this algorithm?

Consider column striping of this matrix

What type of a communication do we need?

Any other type of parallelization?

Divide the text string into n/p substrings and assign each substring to a processor

May consider assign boundary strings to multiple processors

---

The longest common subsequence

Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ be two sequences. The longest common subsequence is the longest sequence of common characters. *Need not be consecutive strings – but preserve the order*

Example: $A = (c, a, d, b, r, z)$ and $B = (a, s, b, z)$

The LCS = $(a, b, z)$

Problem formulation.

Let $F[i,j]$ is the length of the longest common subsequence of the first $i$ elements of $A$ and first $j$ elements of $B$

$$F[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ F[i-1,j-1] + 1 & \text{if } i, j >0 \text{ and } a_i = b_j \\ \max\{F[i,j-1], F[i-1,j]\} & \text{if } i, j >0 \text{ and } a_i \neq b_j \end{cases}$$

We can use two pointers, one moving along $A$ and another moving along $B$.
Computing Longest Common Subsequence (LCS) of two amino acids
See figure 12.6 on page 525