CSCE 5160 Parallel Processing

Exam 2: Wednesday April 3, 2019

Review:
Scatter and gather with variable number of data elements
Bitonic Sorting networks
Graph Algorithms
   Adjacency matrix and adjacency lists
Trees, Spanning trees and minimum spanning trees
Minimum spanning tree: Prim’s algorithm

CSCE5160 March 27, 2019

procedure PRIM
2. begin
3. \( V_T = \{r\}; \) # select a root
4. \( d[r] = 0; \)
5. for all \( v \) in \((V-V_T)\) do # initialization
6.   if edge \((r,v)\) exists set \( d[v] = w(r,v); \)
7.   else \( d[v] = \text{infinity}; \)
8. while \( V_T \) not equal to \( V \) # construction not done
9. begin
10.   find a vertex \( u \) such that \( d[u] \) is min \[d[v]| v \in (V-V_T) \]
     # find a new node to add
11.   \( V_T = V_T \) plus \( u \)
12. for all \( v \) in \((V-V_T)\) do # add node \( u \) to the tree
13.   \( d[v] = \text{min} [d[v], w(u,v)]; \) # update cost values
14. end
15 end

How do we parallelize this algorithm?
Can we think of adding new nodes to the tree in parallel?
What is the problem with this approach?

CSCE5160 March 27, 2019
CSCE 5160 Parallel Processing

We can try to parallelize the computation in finding the minimum cost edge from any of the nodes in the tree

10. find a vertex \( u \) such that \( d[u] \) is min \([d[v]: v \in (V-V_T)]\)

We use tree like reduction for this purpose

We can also parallelize the loop (updating new costs) at lines 12-13

Consider column striping of the adjacency matrix -- assign \( n/p \) nodes to each processor.
Each processor computes the minimum cost edges from the Tree nodes to the nodes assigned to that processor.

Each processor computes the minimum cost edges from the Tree nodes assigned to that processor.
These minimum cost edges are sent to processor \( P_0 \) which performs reduction (to find the overall minimum edge).

The new node to be added (based on the minimum cost edge) is broadcast to all processors.
Each processor computes the min vertex (line 10) for \( n/p \) node of the graph.
This computation can take \( O(n/p) \)

We perform all-to-one reduction (reduction) communication to find the overall minimum
This depends on the network. For hypercube this takes \((t_s + t_w) \log p\)

We then execute a broadcast to broadcast from \( P_0 \) to all nodes -- in hypercube this takes \((t_s + t_w) \log p\)
*(we can also do all to all reduction)*

We repeat the above steps \( n \) times (adding one new node to tree at a time)
Overall Complexity $= O(n^2/p) + O(n \log p)$
the reduction and broadcast of min is repeated n times

Mesh
If we are using a Mesh, the communication costs will differ.
For all-to-one reduction and one-to-all broadcast, we need $O(p^{1/2})$ time units
Total complexity $= O(n^2/p) + O(n \cdot p^{1/2})$

Does it make sense to use 2-D or checkerboard partitioning of Adjacency matrix?
We also need to partition $d(v)$ vector among row processors

Each node now only has a partial information about its neighbors.
Each node now only has a partial information about its neighbors.

Suppose node 4 is added to the tree.
In order to compute d(0), d(1), d(2) and d(3) by first processor in first row,
we need to get the adjacency information from other processors

So the processors in second row (4-7) need to broadcast their values to column processors
Assuming 2-D mesh of processors , cost of this step = $O(p^{1/2})$

How do we find minimum’s? $d(v)$ is distributed among the first row processors
We need to reduction among the row processors
Broadcast the global minimum to row processors
Cost of this $O(p^{1/2})$

We repeat this n times; total communication cost: $O(n \cdot p^{1/2})$

Note this is the same as for the column striping method

Let us look at some other algorithms.
Find shortest paths from given node to all other nodes.
We can use an algorithm very similar to the Minimum Spanning Tree algorithm we just studied.

We start with our source node as the root of the tree.

Pick a new node to add to our set which has a minimum cost edge from the source node.
Then pick a node which has a minimum cost edge to one of the nodes in $V_T$.

After adding a new node, we will check if a path from the new node to other nodes with smaller cost exist.
Modifying Prim’s algorithm to compute shortest paths from source to all other nodes (or Dijkstra’s algorithm)

---

**Algorithm 10.2** Dijkstra’s sequential single-source shortest path algorithm.

Note that we only change $l(v)$ if the added note reduces path length from source $s$ to $v$.
CSCE 5160 Parallel Processing

Is there any difference between Shortest paths’ algorithm and Minimum Spanning Tree algorithm? – the order in which we add nodes may change

Does Prim’s algorithm guarantee shortest paths from a given node? From root to all other nodes?

Parallelization of Dijkstra’s shortest paths algorithm -- very similar to that of Prim’s algorithm (page 437)

All to all shortest paths.

Can we extend Dijkstra’s algorithm to compute all pairs shortest paths? Repeat the algorithm with different root (so the algorithm on page 437 is repeated n times)

Somewhat of a related problem is called Transitive Closure which tries to find if a path exists between every pair of nodes. Excluding circuits, if a path exists between two nodes its length can at most be n-1.

Let us see what happens if start with our adjacency matrix A, and find A*A. What does A² represent?

If A is un-weighted, then it presents all paths of length 2.

a²[i,j] represents the number of paths from node vi to vj of length 2.

Likewise Aⁿ represents all paths of length n-1. Now if we look at the sum

R= A + A² +... + Aⁿ⁻¹ we are looking at all possible paths.

Note each A’ is like a matrix multiplication -- O(n³) and computing R is O(n⁴)

Can we modify the matrix multiplication to find all shortest paths in a graph?
Let us modify Matrix multiplication algorithms to compute Transitive closure.

In matrix multiplication we use
\[ C[i,j] = C[i,j] + A[i,k] \times B[k,j] \] /* inside the innermost loop

For our Transitive closure we are dealing with the same matrix for A and B -- the adjacency list.

If we have un-weighted arcs, A will consist of 0’s and 1’s. So, our modified algorithm will consist of the following


A more efficient algorithm (for sequential case) is known as Warshall’s algorithm.

Replace multiplication and addition of a Matrix multiplication (innermost loop)


With a binary operation

if not \( R[i,j] \) and if \( A[i,k] \) and \( A[k,j] \) then \( R[i,j] = true \)

Now computing new R is \( O(n^3) \) not \( O(n^4) \).

Consider how we can use the transitive closure idea for finding all shortest paths.

If we have a shortest path between a pair of nodes i and j, its length must be less than \( n-1 \) (assuming no loops -- particularly negative cost loops).

So, we try to find paths of length 1, 2, ..., \( n-1 \). Each time, if we find a path between the two i, and j that is shorter, we update our matrix.

Let \( D_m \) be a matrix for keeping track of shortest paths.
\[ d^m[i,j] = \text{the shortest path between i and j of length } m \text{ or less} \]

Consider how we can modify our Transitive closure algorithm

\[ d^m[i,j] = \min \{ d^{m-1}[i,j], \text{or } d^{m-1}[i,k] + d^{m-1}[k,j] \} \]

Actually, we can do even better.
CSCE 5160 Parallel Processing

Let us assume that we number vertices and process them in order – or in \( p \)th iteration, we look at paths going through node \( i \).

Consider the case the D matrix, particularly in iteration \( k-1 \); \( d^{k-1}[i,j] \) – here we interpret D matrix differently.

This element \( d^{k-1}[i,j] \) is the shortest path between \( V_i \) to \( V_j \) with the path that may go through nodes \( V_0, V_1, \ldots, V_{k-1} \).

So in the next step to compute \( D^k \), we only need to see if there a shorter path that goes through node \( V_k \) (using previously computed shortest paths that go through nodes \( V_0, V_1, \ldots, V_{k-1} \)).

We can modify the innermost computation to

\[
d^k[i,j] = \min \{ d^{k-1}[i,j], d^{k-1}[i,k] + d^{k-1}[k,j] \}
\]

Note the indexes \( k \) and the length of the path \( k \).

We do not have to examine for all values of \( k \)-- We only examine one node \( k \) in the \( k \)th iteration. 

*In other words, at iteration \( k \), we need only one row and one column element from the matrix \( D^{k-1} \).*

This is called Floyd (or Warshall-Floyd) algorithm.

---

CSCE 5160 Parallel Processing

Why It Works

\[
\begin{align*}
&\text{Shortest path from } i \text{ to } k \\
&\text{through } 0, 1, \ldots, k-1 \\
&\text{Computation in previous iterations}
\end{align*}
\]

\[
\begin{align*}
&\text{Shortest path from } i \text{ to } j \\
&\text{through } 0, 1, \ldots, k-1
\end{align*}
\]

\[
\begin{align*}
&\text{Shortest path from } k \text{ to } j \\
&\text{through } 0, 1, \ldots, k-1
\end{align*}
\]

*In \( k \)th iteration, we do not have to look at nodes 0,1,..,k-1

*Only nodes \( k, k+1, \ldots, n-1 \)*
CSCE 5160 Parallel Processing

Consider the case \( d^m[i,j] \)

This consists of shortest paths of length \( m \) or less from node \( i \) to node \( j \).
That means, we have already examined several intermediate nodes (to be exact \( m-2 \)) in constructing the paths of length \( m \) or less.

We do not have to compute the paths involving these any longer. We can modify the innermost computation to

\[
    d^k[i,j] = \min \{ d^{k-1}[i,j], \text{or} d^{k-1}[i,k] + d^{k-1}[k,j] \}
\]

Note the indexes \( k \) indicates length of the path \( k \).

Since we try to find a path from node \( i \) to node \( j \), and we examine node \( j \) in order from 1, 2,…, \( k \)…n, at \( k \)th iteration, we have looked at nodes \( j=1,\ldots,k-1 \)

In other words, at iteration \( k \), we need only one row and one column element from the matrix \( D^{k-1} \).

This is called Floyd (or Warshall-Floyd) algorithm

CSCE5160 March 27, 2019

---

CSCE 5160 Parallel Processing

1. procedure FLOYD_ALL_PAIRS_SP(A)
2. begin
3. \( D^{(0)} = A; \)
4. for \( k := 1 \) to \( n \) do
5. \hspace{1em} for \( i := 1 \) to \( n \) do
6. \hspace{2em} for \( j := 1 \) to \( n \) do
7. \hspace{3em} \( d^{(k)}_{i,j} := \min (d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}); \)
8. end FLOYD_ALL_PAIRS_SP

Notice the subscript \( k \) in the \( k \)th iteration.

CSCE5160 March 27, 2019
Let us compute all to all shortest distances
Let us label nodes as 1,2,3,4,5 instead of A,B,C,D,E

\[
D^0 = A
\]

\[
d_{1,1} = \min\{d_{1,2}, d_{1,3} + d_{1,2}\} = \min\{6, 0 + 0\} = 6
\]

\[
d_{1,2} = \min\{d_{2,3}, d_{1,3} + d_{1,2}\} = \min\{3, 0 + 3\} = 3
\]

\[
d_{1,3} = \min\{d_{1,4}, d_{1,1} + d_{1,3}\} = \min\{\infty, 0 + \infty\} = \infty
\]

\[
d_{1,4} = \min\{d_{2,5}, d_{1,5} + d_{1,4}\} = \min\{\infty, 0 + \infty\} = \infty
\]

\[
d_{1,5} = \min\{d_{3,1}, d_{3,1} + d_{1,5}\} = \min\{\infty, \infty + \infty\} = \infty
\]

\[
d_{2,1} = \min\{d_{2,1}, d_{2,2} + d_{1,1}\} = \min\{4, 4 + 0\} = 4
\]

\[
d_{2,2} = \min\{d_{2,3}, d_{2,2} + d_{1,2}\} = \min\{\infty, 4 + 3\} = 7
\]

\[
d_{2,3} = \min\{d_{3,2}, d_{2,2} + d_{1,3}\} = \min\{1, \infty + \infty\} = 1
\]

\[
d_{2,4} = \min\{d_{2,5}, d_{2,5} + d_{1,4}\} = \min\{\infty, 0 + \infty\} = \infty
\]

\[
d_{2,5} = \min\{d_{3,5}, d_{3,5} + d_{1,5}\} = \min\{\infty, \infty + \infty\} = \infty
\]

\[
d_{3,1} = \min\{d_{3,1}, d_{3,3} + d_{1,1}\} = \min\{\infty, \infty + 0\} = \infty
\]

\[
d_{3,2} = \min\{d_{3,2}, d_{3,3} + d_{1,2}\} = \min\{\infty, \infty + 6\} = \infty
\]

\[
d_{3,3} = \min\{d_{3,4}, d_{3,4} + d_{1,3}\} = \min\{5, \infty + \infty\} = 5
\]

\[
d_{3,4} = \min\{d_{3,5}, d_{3,5} + d_{1,4}\} = \min\{1, \infty + \infty\} = 1
\]
Continue and compute paths from node 4 and 5
To all other nodes, examining paths through node 1
No change since there is no path from 4 to 1 or 5

Let us compute $D^{i}$ paths going through node B

- $d_{1,2}^{i} = \min\{d_{1,3}^{i}, d_{1,2}^{i}+d_{2,2}^{i}\} = \min\{6, 6+0\} = 6$
- $d_{1,3}^{i} = \min\{d_{1,3}^{i}, d_{1,2}^{i}+d_{2,2}^{i}\} = \min\{3, 6+\infty\} = 3$
- $d_{1,4}^{i} = \min\{d_{1,4}^{i}, d_{1,2}^{i}+d_{2,2}^{i}\} = \min\{\infty, 6+1\} = 7$
- $d_{1,5}^{i} = \min\{d_{1,5}^{i}, d_{1,2}^{i}+d_{2,2}^{i}\} = \min\{\infty, 6+\infty\} = \infty$
Continuing
\[ d_{1,3} = \min \{ d_{1,1} + d_{1,3} + d_{1,5} \} = \min \{ \text{inf}, \text{inf}+4 \} = \text{inf} \]
\[ d_{1,2} = \min \{ d_{1,1} + d_{1,2} + d_{1,2} \} = \min \{ \text{inf}, \text{inf}+0 \} = \text{inf} \]
\[ d_{1,4} = \min \{ d_{1,1} + d_{1,2} + d_{1,4} \} = \min \{ 5, \text{inf}+1 \} = 5 \]
\[ d_{1,5} = \min \{ d_{1,5}, d_{1,2} + d_{1,5} \} = \min \{ 1, \text{inf}+\text{inf} \} = 1 \]

Need to compute paths for node 4 and 5 to all other nodes going through node 2
Then repeat for \( D^3 \) paths going through node 3; \( D^4 \) going through node 4
And \( D^5 \)

Note, everytime we only see if there is a new shorter path going through one new node
Likewise compute $D^3$, $D^4$, $D^5$.

Let us return to the algorithm and see how we can **parallelize** Floyd’s algorithm.

---

**Procedure FLOYD_ALL_PAIRS_SP(A)**

1. **procedure FLOYD_ALL_PAIRS_SP(A)***
2. begin
3. \[ D^{(0)} = A; \]
4. for \( k := 1 \) to \( n \) do
5. \[ \text{for } i := 1 \text{ to } n \text{ do} \]
6. \[ \text{for } j := 1 \text{ to } n \text{ do} \]
7. \[ d_{i,j}^{(k)} := \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}); \]
8. end FLOYD_ALL_PAIRS_SP

---

If we have \( n^2 \) processors, we can use each processor to compute the elements of \( d[i,j] \) in parallel.
If we are dealing with un-directed graphs then we need only \(\frac{n^2}{2}\) processors. However, in computing \(d^k[i,j]\), a processor needs access to \(d^{k-1}[i,k]\) and \(d^{k-1}[k,j]\) -- that is, we need ith row and jth column of \(D^{k-1}\).

We can use all-to-all personalized broadcast among the processors.

---

**CSCE 5160 Parallel Processing**

**CSCE5160 March 27, 2019**

27

---

**CSCE 5160 Parallel Processing**

**CSCE5160 March 27, 2019**

28
Let us consider the case where we have $p$ processors, each processor will be assigned \((n \times n)/p\) elements of $D^k$ to compute.

Depending on how we assign Adjacency matrix elements to processors we can eliminate some broadcasts.

- **Column wise block striped**
  - Broadcast within columns eliminated

- **Row wise block striped**
  - Broadcast within rows eliminated
  - Reading matrix from file simpler

What about checkerboard?
- We need both column and row broadcasts

---

**Complexity Analysis**

**Sequential complexity** = $O(n^3)$ – similar to Matrix multiplication but we use Washall’s method and repeat $n$ times

**Parallel complexity** (assume row striping – *one row is broadcast in each iteration*):

Computation: $O(n^2/p)$ on each iteration and repeated $n$ times
- **Total computational complexity** = $O(n^3/p)$

Communication: Each broadcast can be done in log $(p)$ units using hypercube
- But we are sending $n$ elements in each broadcast
  - *note one row ($n$ elements) are broadcast by one processor*
- So, each broadcast complexity = $O(n \times (\log p))$
  - We repeat the broadcast $n$ times
- **Communication complexity** = $O(n^2 \log (p))$

**Total complexity** = $O(n^3/p) + O(n^2 \log (p))$
Let us assume checkerboard partitioning as is done in Textbook

Again to compute the \( d^k[i,j] \) for the elements assigned to processor \( P_i \), we need the \( i \)th row and \( j \)th column of \( D^{k-1} \).

We can use all-to-all broadcast or use a carefully synchronized approach where only a processor in \( i \)th row (or \( j \)th column) will broadcast its data to every other processor in its row (or column).

Let us examine how we partitioned our matrix -- see page 442.

Data in the blue processors are needed (\( k \)th column and \( k \)th row elements)

---

Parallel Floyd algorithm (page 443)

Procedure Floyd

\[
\text{for}(k=0; \ n<; \ k++) \\
\{ \text{each process } P_{i,j} \text{ that has a segment of the } k^{\text{th}} \text{ row of } D^{(k-1)} \text{ broadcast it to the } P_{*,j} \text{ processes} \\
\quad \text{#to every process in its column} \\
\text{each process } P_{i,j} \text{ that has a segment of the } k^{\text{th}} \text{ column of } D^{(k-1)} \text{ broadcast it to the } P_{i,*} \text{ processes} \\
\quad \text{#to every process in its row} \\
\text{each process computes } D^k \}
\]
At each broadcast, a processor will send only a single row or a single column and thus we are dealing with $\frac{n}{p^{1/2}}$ elements in each communication.

One-to-all broadcast of 1 unit of data takes $\log p$ time units.

So we have $O\left(\frac{n}{p^{1/2}} \log p\right)$ for communication.

Since each processor has to compute updated values for $(n^2)/p$ elements, the total computation cost = $O(n^2/p)$.

However, we need to compute the values for different $k$, for $k=1,..,n$.

(repeat the above communication and computation $n$ times)

The total complexity = $O\left(n^2/p + \frac{n^2}{p^{1/2}} \log p\right)$ (see page 442)

Can we think of OpenMP implementation of Floyd?

```c
#pragma omp parallel for
for (j=0; j<n; j++) /* one column per thread */
for (k=0; k<n; k++) /* same as k loop */
for (i=0; i<n; i++) {
    D[i][j] = Min(D[i][j], D[i][k] + D[k][j]);
}
```

/* a barrier is needed */
We can use transitive closure to find if a graph is connected
(is there a path from any node s to any other node t)

If the graph is **not fully** connected, we can think of the graph as several different connected graphs

We want to analyze the graph to find all "connected components"

We can also use either a depth first or breadth first search of the graph
Create depth first or breadth first trees
Each connected component will be a separate tree.

Either Depth first or Breadth first approach will find all **connected components** of a graph.

How can we parallelize dfs and bfs algorithms?