Exam 2: Wednesday April 3, 2019 in F280

Posted some solutions to problems from Chapters 8-10

Review for Exam 2: Primarily from Chapters 8-10

Chapter 10 – Graph Algorithms
- Adjacency matrix vs Adjacency lists
- Shortest spanning tree and Prim’s algorithm
  - parallelization using column striping
  - processors update cost values for nodes assigned to them
  - find local min and then find global min using reduction
  - broadcast the new node to include in tree
- Shortest paths – Dijkstra’s algorithm – similar to Prims
- All to all Shortest paths
  - Warshall’s algorithm
  - Floyd’s algorithm
- BFS and DFS trees (for connected components)

Chapter 9: Sorting algorithms
- Quick sort
- Parallelization using hypercube
- Pivot selection
- Other sorting algorithms, Bucket, Odd-even enumeration
- Sorting networks
- Bitonic sort

Chapter 8: Dense Matrix Algorithms
- Vector * Vector multiplication
- Matrix * Vector
- Matrix * Matrix
  - Cannon’s algorithm
  - Fox’s algorithm
- Solving Linear equations
  - Gaussian Elimination
  - MPI implementation and OpenMP implementation
CSCE 5160 Parallel Processing

Review of Graph Algorithms from last week

```
procedure PRIM
begin

V_T = \{r\};  \# select a root

d[r] = 0;

for all v in (V - V_T) do  \# initialization

if edge (r,v) exists set d[v] = w(r,v);
else d[v] = infinity;

while V_T not equal to V  \# construction not done

begin

find a vertex u such that d[u] is min \{d[v] | v in (V - V_T)\}  \# find a new node to add

V_T = V_T plus u  \# add node u to the tree

for all v in (V - V_T) do

d[v] = min \{d[v], w(u,v)\};  \# update cost values

end

end
```

CSCE 5160 April 1, 2019

CSCE 5160 Parallel Processing

Shortest path

```
procedure Dijkstra.SINGLE.SOURCE.SP(V, E, w, s)
begin

V_T := \{s\};

for all v in (V - V_T) do

if (s, v) exists set l[v] := w(s, v);
else set l[v] := \infty;

while V_T \neq V do

begin

find a vertex u such that l[u] := min[l[v] | v in (V - V_T)];

V_T := V_T \cup \{u\};

for all v in (V - V_T) do

l[v] := min[l[v], l[u] + w(u, v)];

endwhile

end Dijkstra.SINGLE.SOURCE.SP
```

Note that we only change l(v) if the added node reduces path length from source s to v.
All to all shortest paths.

Transitive closure and Warshall’s algorithm

Let us assume that we number vertices and process them in order – or in $i^{th}$ 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 the shortest path between $V_i$ to $V_j$ with the path that may go through nodes $V_0, V_1, ..., V_{k-1}$

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 $V_0, V_1, ..., V_{k-1}$)

We then add $e$ and $f$ to the tree and update $l(v)$ values-- no changes.

<table>
<thead>
<tr>
<th>VT</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(v)</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>inf</td>
<td>inf</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VT</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(v)</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>inf</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VT</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(v)</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VT</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(v)</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

CSCE 5160 Parallel Processing
We can modify the innermost computation to
\[
    d_{i,j}^{k} = \min \{ d_{i,j}^{k-1}, \text{ or } d_{i,k}^{k-1} + d_{k,j}^{k-1} \}
\]
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

1. procedure FLOYD_ALL_PAIRS_SP(A)
2. begin
3. \[ D^{(0)} = A; \]
4. for \( k := 1 \) to \( n \) do
5. \[ i := 1 \) to \( n \) do
6. \[ j := 1 \) to \( n \) 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

Notice the subscript \( k \) in the \( k \)th iteration.
Let us return to the algorithm and see how we can parallelize Floyd’s algorithm.

Likewise compute $D^2$, $D^3$, $D^4$, $D^5$.

All pair wise distances going through node 1
How to parallelize this algorithm.

If we have $n \times n$ processors, we can use each processor to compute the elements of $d[i,j]$ in parallel.

```
1. procedure FLOYD_ALLPAIRS_SP(A)
2. begin
3. $D^{(0)} = A$;
4. for $k := 1$ to $n$ do
5. for $i := 1$ to $n$ do
6. for $j := 1$ to $n$ do
7. $d^{(k)}_{i,j} := \min (d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j})$;
8. end FLOYD_ALLPAIRS_SP
```
Let us consider the case where we have \( p \) processor, 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:

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

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

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

**Homework #8: Implement Floyd’s algorithms in MPI (Due April 10, 2019)**
- You can use either row or column striping
- Try with graphs containing 16, 32 and 64 (or more) nodes
- You can use distances between cities if you want

---

**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 \( (n \text{ elements}) \) 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.

![Partitioned Matrix](image)

Data in the blue processors are needed: (kth column and kth row elements)

---

**Parallel Floyd algorithm (page 443)**

Procedure Floyd

for($k=0; k<n; k++)$

{ each process $P_{ij}$ that has a *segment of the $k^{th}$ row* of $D^{k-1}$
  broadcast it to the $P_{ji}$ processes
  
  #to every process in its column

  each process $P_{ij}$ that has a *segment of the $k^{th}$ column* of $D^{k-1}$
  broadcast it to the $P_{ij}$ processes
  
  #to every process in its row

  each process computes $D^k$
}

---
CSCE 5160 Parallel Processing

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 \( \mathcal{O}(\frac{n}{p^{1/2}} \log p) \) for communication.

Since each processor has to compute updated values for \( \frac{n^2}{p} \) elements, the total computation cost = \( \mathcal{O}(\frac{n^2}{p}) \).

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 = \( \mathcal{O}(\frac{n^3}{p}) + \mathcal{O}(\frac{n^2}{p^{1/2}} \log p) \) (see page 442)

---

CSCE 5160 Parallel Processing

Can we think of OpenMP implementation of Floyd?

```c
#procedure FLOYD_ALL_PAIRS_SP(A) begin
1. \( D^{(0)} = A; \)
2. for \( k := 1 \) to \( n \) do
3.  \( d_{i,j}^{(k)} := \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right); \)
end FLOYD_ALL_PAIRS_SP

....
```

```c
#pragma omp parallel for
/* one column per thread */
for (j=0; j<n; j++)
    for (k=0; k<n; k++)
        \( d_{i,j}^{(k)} = \text{Min}(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}); \)
#pragma omp barrier
/* 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?

Note DFS and BFS trees are not unique
CSCE 5160 Parallel Processing

Consider Parallelizing Breadth First Approach

If we assign a node to each processor, we can think of finding trees with different roots of depth 1 in parallel (we will call them $T(u,0)$ – root $u$ and depth of 1)

What do we do now?
Try to merge a pair trees to construct larger tree of the form $T(u,1)$

Consider building $T(1,1)$. We will try to merge $T(2,0)$ with $T(1,0)$

For each edge $(i,j)$, in $T(2,0)$, if $i$ and $j$ are already in $T(1,0)$, do nothing.
Otherwise add the edge to the tree $T(1,1)$ $(i,j)$, - do not add edges to nodes already in $T(1,0)$

Now let us add $T(3,0)$

CSCE 5160 Parallel Processing

Note that if we try to merge any other T(u,0) trees with T(1,0) we cannot add any new nodes since T(1,1) can have only depth of 2.

In a similar manner we can build T(2,1), T(3,1), T(4,1), ….., T(9,1)

T(v,2) trees -- depth of 3

Consider T(1,1) -- only 3 nodes at leaves -- 4, 5, 6. We can only consider merging T(4,1), T(5,1) and T(6,1)
Similarly we can build other $T(v, 2)$ trees. We need only go up to $T(v, \log n)$ levels.

In this example you notice that $T(1, 2)$ is the breadth first tree with 1 as root.
If we try to generate $T(1, 3)$ and $T(1, 4)$ you will not grow the tree anymore.

What is the complexity?
What is the complexity of the merge?
To merge two trees we need $O(\log n)$ since $T(u,k)$ can have at most $\log n$ nodes in that tree.

If we have $n$ processors to merge all trees $T(v,k)$ on to a single tree $T(u,k)$, we can complete the merge to generate $T(u,k+1)$ in $O(\log n)$.

Since we have to perform this merge operation for all nodes to generate $T(v,k+1)$, we can use $O(n^*n)$ processors.

We repeat the merge operation for $\log (n)$ times.

**Total complexity** = $O(\log^2 n)$ or $O(\log n \times \log n)$ if we have $O(n^*n)$ processors.

Sequential complexity = $O(n^*n)$ -- not cost optimal.
CSCE 5160 Parallel Processing

Depth First

A similar approach to develop a parallel depth first approach will not work. Why?

Consider creating depth first forests of type T(u,k) and merge them.

Instead we will use a different approach.

We use row striping to assign n/p rows of the adjacency matrix to each processor.

Each processor will have only a subset of edges (corresponding to the rows it has).

*Does not see the entire graph*

Each processor will build a depth first tree with its links.

We merge the tree together to form a single depth first tree.

Let us consider a concrete example.
After finding individual depth first trees, we have dfs forests
Now we need to merge these into a single tree
CSCE 5160 Parallel Processing

Not exactly what we would expect from a sequential depth first search but still a df tree

Complexity of the algorithm

The merge operation requires $O(n)$ operations since each dfs forest can have at most $O(n)$ nodes to merge and we need to merge $p$ forests, we need $O(p*n)$ operations

If we perform pair-wise merging, we need only $O(n^* \log p)$ operations

Each processor needs to construct a dfs with $|E|/p$ edges.
If we assume $|E|= n^*n$ then DFS construction would be $O(n^*/p)$ computations

Total complexity = $O(n^*/p) + O(n \log p)$ and if $p = n$, then complexity = $O(n \log n)$
Sequential complexity = $O(n^*)$
Parallel implementation is not cost-optimal

CSCE 5160 Parallel Processing

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

If a graph has very few edges we will use adjacency lists (instead of adjacency matrix) for graph algorithms

if connected, a graph must have at least $n-1$ edges (minimum number of edges)
if strongly connected will have $n*(n-1)/2$ edges (maximum)

Parallelization of adjacency lists-based algorithms is very difficult

How do we balance load?

If we distribute vertices (and corresponding lists) to processors, some processors may have more edges than others → we may need to scatter different number of elements
If we try to distribute equal number of edges, involves more communication

Note we already have seen how to scatter and gather different number of data items
Another important algorithm

Maximal independent set

Somewhat related to the famous 4-Color theorem

MIS is a set of vertices of a graph such that no edges among these nodes exist

The nodes in this set are not adjacent to each other

Maximal mean, if you add one more node to this set, it will no longer be independent

Consider an example Figure 10.15 on page 452

MIS are not unique

How do we find MIS (sequential algorithm first)

V is the set of all vertices

MIS is the set of independent vertices

Initialization: MIS = null

Repeat

pick a node u from V

add u to MIS and remove it from V

remove all nodes v from V which are adjacent to u

Until V is empty

Complexity = O(n)

How to parallelize?

If we add one node at time, we do not get much parallelism

Consider a different approach so that we can add more than one node to MIS at a time

We will assign random values to each node

Pick nodes with lowest value compared to their neighbors

add these nodes to MIS and remove their neighbors from V
This can be parallelized using shared memory implementation
Divide the nodes into threads (each getting $|V|/p$ nodes)
Each node assigns a random number to its nodes
Picks nodes with minimum values for insertion into MIS
Note all random values for all nodes are in shared memory

We still need synchronization when random values are assigned

If a graph has very few edges we will use adjacency lists (instead of adjacency matrix) for graph algorithms

if connected, a graph must have at least $n-1$ edges (minimum number of edges)
if strongly connected will have $n*(n-1)/2$ edges (maximum)

Parallelization of adjacency lists-based algorithms is very difficult
How do we balance load?
If we distribute vertices (and corresponding lists) to processors, some processors may have more edges than others
If we try to distribute equal number of edges, involves more communication
If we assume a special structure for graph, say that on average all nodes have the same number of edges -- grid graphs (see for example Fig. 10.14 on page 452)
We will see similar grid graphs in several algorithms that use sparse matrices and sparse graphs
Figure 10.14 A street map (a) can be represented by a graph (b). In the graph shown in (b), each street intersection is a vertex and each edge is a street segment. The vertices of (b) are the intersections of (a) marked by dots.