“Parallel Algorithms : Sorting & Graph”

By
Prof. Anand N. Gharu
(Assistant Professor)
PVGCOE Computer Dept.

20th July 2018
Topic Overview

- Issues in Sorting on Parallel Computers
- Bubble Sort and its Variants
- Parallelizing Quick sort
- All-Pairs Shortest Paths
- Algorithm for sparse graph
- Parallel Depth-First Search
- Parallel Best-First Search.
Issues in Sorting of Parallel Computing

- Where the Input and Output Sequences are Stored
- How Comparisons are Performed
  - One Element Per Process
  - More than One Element Per Process

- Sorting can be comparison-based or non comparison-based
- The fundamental operation of comparison-based sorting is compare-exchange
What is the parallel counterpart to a sequential comparator?

- If each processor has one element, the compare exchange operation stores the smaller element at the processor with smaller id. This can be done in $t_s + t_w$ time.

- If we have more than one element per processor, we call this operation a compare split. Assume each of two processors have $n/p$ elements.

- After the compare-split operation, the smaller $n/p$ elements are at processor $P_i$ and the larger $n/p$ elements at $P_j$, where $i < j$.

- The time for a compare-split operation is $(t_s + t_w n/p)$, assuming that the two partial lists were initially sorted.
A parallel compare-exchange operation. Processes $P_i$ and $P_j$ send their elements to each other. Process $P_i$ keeps $\min\{a_i, a_j\}$, and $P_j$ keeps $\max\{a_i, a_j\}$. 

Step 1: 
- $a_i \leftrightarrow a_j$
- $a_i, a_j$

Step 2: 
- $a_j, a_i\min\{a_i, a_j\}$
- $a_i, a_j\max\{a_i, a_j\}$

Step 3: 
- $P_i, P_j$
A compare-split operation. Each process sends its block of size $n/p$ to the other process.

Each process merges the received block with its own block and retains only the appropriate half of the merged block.

In the example, process $P_i$ retains the smaller elements and process $P_j$ retains the larger elements.
Sorting: Parallel Compare Split Operation

Step 1

Step 2

Step 3

Step 4
Bubble Sort and its Variants

❖ The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted.

❖ The complexity of bubble sort is $\Theta(n^2)$.

❖ Bubble sort is difficult to parallelize since the algorithm has no concurrency.

❖ A simple variant, though, uncovers the concurrency.
Bubble Sort

- One of the straightforward sorting methods.
  - Cycles through the list.
  - Compares consecutive elements and swaps them if necessary.
  - Stops when no more out of order pair.
- Slow & inefficient.
- Average performance is $O(n^2)$. 
Bubble Sort:

Original sequence:

Phase 1: Place largest number

Phase 2: Place next largest number

Time
Parallel Bubble Sort:

- Also known as Odd–Even Bubble sort.
- Operates in two alternate phases:

  **Phase–even:**
  
even processes exchange values with right neighbors.

  **Phase–odd:**
  
odd processes exchange values with right neighbors.
Parallel Bubble sort (Odd-Even Sorting)

Unsorted

9 7 3 8 5 6 4 1
7 9 3 8 5 6 1 4
7 3 9 5 8 1 6 4
3 7 5 9 1 8 4 6
3 5 7 1 9 4 8 6
3 5 1 7 4 9 6 8
3 1 5 4 7 6 9 8
1 3 4 5 6 7 8 9

Sorted

Phase 1 (Odd)
Phase 2 (Even)
Phase 3 (Odd)
Phase 4 (Even)
Phase 5 (Odd)
Phase 6 (Even)
Phase 7 (Odd)
Parallel bubble sort: Analysis

- Steps 1-10 is a one big loop that is represented $n$ times.
- Therefore, the parallel time complexity is $O(n)$.
- The algorithm, odd-numbered steps need $(n/2) - 1$ processors and even-numbered steps require $(n/2)$ processors.
- Therefore, this needs $O(n)$ processors.
Bubble Sort and its Variants

The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted:

1. procedure BUBBLE_SORT(n)
2. begin
3. for i := n – 1 downto 1 do
4. for j := 1 to i do
5. compare-exchange(a_j, a_{j+1});
6. end BUBBLE_SORT
Bubble Sort and its Variants

• The complexity of bubble sort is $\Theta(n^2)$.
• Bubble sort is difficult to parallelize since the algorithm has no concurrency.
• A simple variant, though, uncovers the concurrency.
Odd-Even Transposition

1. procedure \texttt{ODD-EVEN}(n)
2. begin
3. for \(i := 1\) to \(n\) do
4. begin
5. if \(i\) is odd then
6. for \(j := 0\) to \(n/2 - 1\) do
7. \texttt{compare-exchange}(a_{2j+1}, a_{2j+2});
8. if \(i\) is even then
9. for \(j := 1\) to \(n/2 - 1\) do
10. \texttt{compare-exchange}(a_{2j}, a_{2j+1});
11. end for
12. end \texttt{ODD-EVEN}

Sequential odd-even transposition sort algorithm.
Odd-Even Transposition

Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.
Odd-Even Transposition

• After $n$ phases of odd-even exchanges, the sequence is sorted.
• Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
• Serial complexity is $\Theta(n^2)$. 
Parallel Odd-Even Transposition

- Consider the one item per processor case.
- There are $n$ iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not the optimal one.
Parallel Odd-Even Transposition

```plaintext
procedure ODD-EVEN_PAR(n)
begin
    id := process’s label
    for i := 1 to n do
    begin
        if i is odd then
            if id is odd then
                compare-exchange_min(id + 1);
            else
                compare-exchange_max(id - 1);
        else
            if i is even then
                if id is even then
                    compare-exchange_min(id + 1);
                else
                    compare-exchange_max(id - 1);
    end for
end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.
Parallel Odd-Even Transposition

- Consider a block of $n/p$ elements per processor.
- The first step is a local sort.
- In each subsequent step, the compare exchange operation is replaced by the compare split operation.
- The parallel run time of the formulation is

$$T_P = \Theta \left( \frac{n}{p} \log \frac{n}{p} \right) + \Theta(n) + \Theta(n).$$
Parallel Odd-Even Transposition

• The parallel formulation is cost-optimal for $p = O(\log n)$.
• The isoefficiency function of this parallel formulation is $\Theta(p2^p)$. 
Shell sort

- The main limitation of odd-even transposition sort is that it moves elements only one position at a time. If a sequence has just a few elements out of order, and if they are N distance from their proper positions, then the sequential algorithm still requires N^2 time to sort the sequence. To make a substantial improvement over odd-even transposition sort, we need an algorithm that moves elements long distances. Shell sort is one such serial sorting algorithm.
Shell sort

- Let \( n \) be the number of elements to be sorted and \( p \) be the number of processes.
- During the first phase, processes that are far away from each other in the array compare-split their elements.
- During the second phase, the algorithm switches to an odd-even transposition sort.
Parallel Shell sort

• Initially, each process sorts its block of $n/p$ elements internally.

• Each process is now paired with its corresponding process in the reverse order of the array. That is, process $P_i$, where $i < p/2$, is paired with process $P_{p-i-1}$.

• A compare-split operation is performed.

• The processes are split into two groups of size $p/2$ each and the process repeated in each group.

• Example: 

  https://slideplayer.com/slide/4927085/68/video/Chapter+%3A+Sorting+Algorithms.mp4
Parallel Shell sort

An example of the first phase of parallel shell sort on an eight-process array.
Shell sort: Parallel algorithm...

Example:

Iteration 1

On completion of iteration 2

Iteration 2
Figure 10.3 shows the example of sorting the array, which consists of 16 elements by means of the discussed method. It should be noted that the data appears to be sorted after the completion of the first stage, and there is no need to execute the odd-even transposition iterations.

![Diagram of sorting process](image)

**Figure 10.3.** The example of the use of the parallel Shell algorithm for 4 processors (the processors are marked by circles, the processor numbers are given in their binary representation)

With regard to the given description the same decomposition approach can be applied and define the compare-split operation as the basic computational subtask. As a result, the number of subtasks will coincide with the number of the available processors (the size of the data blocks in the subtasks is equal to $n/p$). As a result, scaling the computations is not needed again. The distribution of the sorted data among the processors should be selected with regard to the efficient implementation of the compare-split operation in the hypercube network topology.
Parallel Shell sort

- Each process performs $d = \log p$ compare-split operations.
- With $O(p)$ bisection width, each communication can be performed in
time $\Theta(n/p)$ for a total time of $\Theta((n \log p)/p)$.
- In the second phase, $l$ odd and even phases are performed, each
requiring time $\Theta(n/p)$.
- The parallel run time of the algorithm is:

\[
T_P = \Theta\left(\frac{n}{p} \log \frac{n}{p}\right) + \Theta\left(\frac{n}{p} \log p\right) + \Theta\left(\frac{n}{l \cdot p}\right). \tag{3}
\]
Quicksort

- Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average complexity.

- Quicksort selects one of the entries in the sequence to be the pivot and divides the sequence into two - one with all elements less than the pivot and other greater.

- The process is recursively applied to each of the sublists.
procedure QUICKSORT (A, q, r)
begin
if q < r then
begin
    x := A[q];
    s := q;
    for i := q + 1 to r do
        if A[i] ≤ x then
            begin
                s := s + 1;
                swap(A[s], A[i]);
            end if
    swap(A[q], A[s]);
    QUICKSORT (A, q, s);
    QUICKSORT (A, s + 1, r);
end if
end QUICKSORT

The sequential quicksort algorithm.
Example of the quicksort algorithm sorting a sequence of size $n = 8$

https://youtu.be/cnzIChs03cc : Parallel Quick Sort example
https://youtu.be/zE9N-KrsMBc : Parallel Quick Sort Example
Quicksort

- The performance of quicksort depends critically on the quality of the pivot.
- In the best case, the pivot divides the list in such a way that the larger of the two lists does not have more than $an$ elements (for some constant $a$).
- In this case, the complexity of quicksort is $O(n\log n)$. 
To illustrate the parallel quick algorithm, Figure 10.6 shows the example of sorting data when $n = 16, p = 4$ (i.e. each processor block holds four elements). The processors are shown as rectangles, the data blocks being sorted are shown inside the rectangles. The block values are given at the beginning and at the completion of each sorting iteration. The interacting pairs of processors are linked by double-headed arrows. The optimal values of the pivot elements were chosen for data partitioning. At the first iteration, the value 0 was used for all the processors. At the second iteration, for the pair of processors (0, 1) the pivot element was equal to 4, for the pair (2, 3) the value was chosen to be equal to -5.

Figure 10.6. The example of sorting data by the parallel quick sort method (the results of local block sorting are not included)
Parallel all-pairs shortest path algorithm

Single-Source Shortest Paths

• “A central problem in algorithmic graph theory is the shortest path problem. Hereby, the problem of finding the shortest path between every pair of nodes is known as all-pair-shortest-paths (APSP) problem.”

• The single source shortest path algorithm (for arbitrary weight positive or negative) is also known Bellman-Ford algorithm is used to find minimum distance from source vertex to any other vertex. The main difference between this algorithm with Dijkstra’s algorithm is, in Dijkstra’s algorithm we cannot handle the negative weight, but here we can handle it easily.

• Bellman-Ford algorithm finds the distance in bottom up manner. At first it finds those distances which have only one edge in the path. After that increase the path length to find all possible solutions.

• Dijkstra example : https://youtu.be/pVfj6mxhdMw
Dijkstra’s algorithms find a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

The graph has the following:

- vertices, or nodes, denoted in the algorithm by v or u;
- weighted edges that connect two nodes: (u,v) denotes an edge, and w(u,v) denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- dist, an array of distances from the source node s to each node in the graph, initialized the following way: \( \text{dist}(s) = 0 \); and for all other nodes \( v \), \( \text{dist}(v) = \infty \). This is done at the beginning because as the algorithm proceeds, the dist from the source to each node \( v \) in the graph will be recalculated and finalized when the shortest distance to \( v \) is found
- \( Q \), a queue of all nodes in the graph. At the end of the algorithm's progress, \( Q \) will be empty
- \( S \), an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, \( S \) will contain all the nodes of the graph

Dijkstra’s Example: https://youtu.be/wtdtkJgcYUM
Dijkstra’s Algorithms

The algorithm proceeds as follows:

1. While Q is not empty, pop the node v, that is not already in S, from Q with the smallest dist(v). In the first run, source node s will be chosen because dist(s) was initialized to 0. In the next run, the next node with the smallest dist value is chosen.

2. Add node v to S, to indicate that v has been visited

3. Update dist values of adjacent nodes of the current node v as follows: for each new adjacent node u,
   - if dist(v) + weight(u,v) < dist(u), there is a new minimal distance found for u, so update dist(u) to the new minimal distance value;
   - otherwise, no updates are made to dist(u).

The algorithm has visited all nodes in the graph and found the smallest distance to each node. dist now contains the shortest path tree from source S.

Note: The weight of an edge (u,v) is taken from the value associated with (u,v) on the graph

Example: https://www.gatevidyalay.com/dijkstras-algorithm-shortest-path-algorithm/
Dijkstra’s Algorithms

1. procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2. begin
3. \( V_T := \{s\} \);
4. for all \( v \in (V - V_T) \) do
5. \( \text{if} \ (s, v) \text{ exists set } l[v] := w(s, v) \);
6. \( \text{else set } l[v] := \infty \);
7. while \( V_T \neq V \) do
8. begin
9. find a vertex \( u \) such that \( l[u] := \min\{l[v] | v \in (V - V_T)\} \);
10. \( V_T := V_T \cup \{u\} \);
11. for all \( v \in (V - V_T) \) do
12. \( l[v] := \min\{l[v], l[u] + w(u, v)\} \);
13. endwhile
14. end DIJKSTRA_SINGLE_SOURCE_SP

Example: https://www.gatevidyalay.com/dijkstras-algorithm-shortest-path-algorithm/
Dijkstra's Algorithms e.g.

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
Floyd’s Algorithm is also called as All Pairs Shortest Path Algorithm.

In a given connected, directed and weighted Graph G, the objective of this Algorithm is to find the minimum cost from any vertex to any other vertex in the Graph G.

From a vertex i to vertex j, there may be so many paths, but we have to find a path with minimum value and Dynamic Programming Principle is applied for this.

A vertex k is considered as a vertex which comes in between the path from vertex i to vertex j. Now, the path value \( \text{cost} (i, k) + \text{cost} (k, j) \) is calculated and checked if it is minimum.

This process is repeated for all the possible values of k to get the All Pairs Shortest Path matrix.

This is given by the following recurrence relation:

\[
A^k(i, j) = \min \{ A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j) \}
\]
Floyd Algorithms

• The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

• For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

  1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

  2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

• The following figure shows the above optimal substructure property in the all-pairs shortest path problem.

Example: https://www.gatevidyalay.com/floyd-warshall-algorithm-shortest-path-algorithm/
Floyd Algorithms

Example:

Input:

```plaintext
graph[][] = {{0, 5, INF, 10},
             {INF, 0, 3, INF},
             {INF, INF, 0, 1},
             {INF, INF, INF, 0}}
```

which represents the following graph:

```
10
(0)-------->(3)
    /|
   5 |
    /|
   1 |
    \
(1)-------->(2)
    3
```

Note that the value of `graph[i][j]` is 0 if i is equal to j.
And `graph[i][j]` is INF (infinite) if there is no edge from vertex i to j.

Output:

Shortest distance matrix

```
  0  5  8  9
INF 0  3  4
INF INF 0  1
INF INF INF 0
```

Example: https://youtu.be/oNl0rf2P9gE
Example: https://youtu.be/g0ZDX5-h0hY
Example: https://youtu.be/5KoFK_jYGaQ
Floyd-Algorithms

let \( V \) = number of vertices in graph
let \( \text{dist} = V \times V \) array of minimum distances

for each vertex \( v \)
    \( \text{dist}[v][v] \leftarrow 0 \)

for each edge \((u,v)\)
    \( \text{dist}[u][v] \leftarrow \text{weight}(u,v) \)

for \( k \) from 1 to \( V \)
    for \( i \) from 1 to \( V \)
        for \( j \) from 1 to \( V \)
            if \( \text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j] \)
                \( \text{dist}[i][j] \leftarrow \text{dist}[i][k] + \text{dist}[k][j] \)
            end if

return \( A \)

Floyd-Warshall Algorithm

\( n \) = no of vertices
\( A \) = matrix of dimension \( n \times n \)

for \( k = 1 \) to \( n \)
    for \( i = 1 \) to \( n \)
        for \( j = 1 \) to \( n \)
            \( A^k[i, j] = \min (A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j]) \)
        end for
    end for
end for

return \( A \)
Definition Sparse Graph: A graph in which the number of edges is much less than the possible number of edges.

Definition Dense Graph: A graph in which the number of edges is close to the possible number of edges.

There is no strict distinction between sparse and dense graphs. Typically, a sparse (connected) graph has about as many edges as vertices, and a dense graph has nearly the maximum number of edges.
Examples of sparse graphs:
(a) a linear graph, in which each vertex has two incident edges;
(b) a grid graph, in which each vertex has four incident vertices; and
(c) a random sparse graph.
Algorithms for Sparse Graphs

Example: [https://youtu.be/ShYMsLp8rAA](https://youtu.be/ShYMsLp8rAA)

Example: [https://youtu.be/wJVbvJiQ-rg](https://youtu.be/wJVbvJiQ-rg)
Finding a Maximal Independent Set

Consider the problem of finding a maximal independent set (MIS) of vertices of a graph.

- We are given a sparse undirected graph $G = (V, E)$. A set of vertices $I \subseteq V$ is called independent if no pair of vertices in $I$ is connected via an edge in $G$.
- An independent set is called maximal if by including any other vertex not in $I$, the independence property is violated.
Finding a Maximal Independent Set

Note that as the example illustrates, **maximal independent sets** are not unique.

Maximal independent sets of vertices can be used to determine which computations can be done in parallel in certain types of task graphs.

For example, maximal independent sets can be used to determine the sets of rows that can be factored concurrently in parallel incomplete factorization algorithms, and to compute a coloring of a graph in parallel.
Parallel Merge Sort

Merge sort first divides the unsorted list into smallest possible sub-lists, compares it with the adjacent list, and merges it in a sorted order. It implements parallelism very nicely by following the divide and conquer algorithm.
Parallel Merge Sort

- If the input sequence has fewer than two elements, return.
- Partition the input sequence into two halves.
- Sort the two subsequences using the same algorithm.
- Merge the two sorted subsequences to form the output sequence.

Compare-Exchange.

A compare-exchange operation merges two sorted sequences of length $M$, contained in tasks A and B. Upon completion of the operation, both tasks have $M$ data, and all elements in task A are less than or equal to all elements in task B. As illustrated in Figure, each task sends its data to the other task. Task A identifies the $M$ lowest elements and discards the remainder; this process requires at least $M/2$ and at most $M$ comparisons. Similarly, task B identifies the $M$ highest elements.
The compare-exchange algorithm, with \( M = 4 \).

(a) Tasks A and B exchange their sorted subsequences.

(b) They perform a merge operation to identify the lowest and highest \( M \) elements, respectively.

(c) Other elements are discarded, leaving a single sorted sequence partitioned over the two tasks.
Parallel Merge Sort

Algorithm
procedure parallelmergesort(id, n, data, newdata)
begin
    data = sequentialmergesort(data)

    for dim = 1 to n
        data = parallelmerge(id, dim, data)
    endfor

    newdata = data
end
## Compare Sequential Merge Sort and Parallel Merge Sort

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Sequential Merge</th>
<th>Parallel Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Merge sort is divide and conquer based approach</td>
<td>Merge sort is compare-exchange based approach</td>
</tr>
<tr>
<td>2</td>
<td>Single processor is used</td>
<td>Multiple processor are used</td>
</tr>
<tr>
<td>3</td>
<td>Slower execution</td>
<td>Faster execution</td>
</tr>
<tr>
<td>4</td>
<td>Less efficient</td>
<td>More efficient</td>
</tr>
<tr>
<td>5</td>
<td>Can solve problem in sequential manner</td>
<td>Can solve problem in parallel manner</td>
</tr>
<tr>
<td>6</td>
<td>$O(n \log n)$ in all three cases</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Parallel Depth First Search

• The critical issue in parallel depth-first search algorithms is the distribution of the search space among the processors.

• Consider the tree shown in Figure. Note that the left subtree (rooted at node A) can be searched in parallel with the right subtree (rooted at node B).

• By statically assigning a node in the tree to a processor, it is possible to expand the whole subtree rooted at that node without communicating with another processor. Thus, it seems that such a static allocation yields a good parallel search algorithm.
Parallel Depth First Search

Example DFS: https://youtu.be/dkp9KvUtrWo

Figure: The unstructured nature of tree search and the imbalance resulting from static partitioning.
Parallel Depth First Search

• Assume that we have two processors. The root node is expanded to generate two nodes (A and B), and each of these nodes is assigned to one of the processors.

• Each processor now searches the subtrees rooted at its assigned node independently. At this point, the problem with static node assignment becomes apparent.

• The processor exploring the subtree rooted at node A expands considerably fewer nodes than does the other processor. Due to this imbalance in the workload, one processor is idle for a significant amount of time, reducing efficiency.
Parallel Depth First Search

- Consider the partitioning of the tree for **four processors**. Nodes A and B are expanded to generate nodes C, D, E, and F. Assume that each of these nodes is assigned to one of the four processors. Now the processor searching the subtree rooted at node E does most of the work, and those searching the subtrees rooted at nodes C and D spend most of their time idle. The static partitioning of unstructured trees yields poor performance because of substantial variation in the size of partitions of the search space rooted at different nodes. Furthermore, since the search space is usually generated dynamically, it is difficult to get a good estimate of the size of the search space beforehand. Therefore, it is necessary to balance the search space among processors dynamically.
Parallel Depth First Search

• A parallel formulation of DFS based on dynamic load balancing is as follows. Each processor performs DFS on a disjoint part of the search space.

• When a processor finishes searching its part of the search space, it requests an unsearched part from other processors. This takes the form of work request and response messages in message passing architectures, and locking and extracting work in shared address space machines.

• Whenever a processor finds a goal node, all the processors terminate. If the search space is finite and the problem has no solutions, then all the processors eventually run out of work, and the algorithm terminates.
Parallel Depth First Search

• A generic scheme for **dynamic load balancing**.
Parallel Best First Search

Example DFS: [https://youtu.be/alxqDHJg_q0](https://youtu.be/alxqDHJg_q0)

- An important component of best-first search (BFS) algorithms is the *open* list. It maintains the unexpanded nodes in the search graph, ordered according to their $l$-value. In the sequential algorithm, the most promising node from the *open* list is removed and expanded, and newly generated nodes are added to the *open* list.

- In most parallel formulations of BFS, different processors concurrently expand different nodes from the open list. These formulations differ according to the data structures they use to implement the open list. Given $p$ processors, the simplest strategy assigns each processor to work on one of the current best nodes on the open list. This is called the centralized strategy because each processor gets work from a single global open list. Since this formulation of parallel BFS expands more than one node at a time, it may expand nodes that would not be expanded by a sequential algorithm.
Parallel Best First Search

Global list maintained at designated processor

Put expanded nodes
Get current best node

Lock the list
Place generated nodes in the list
Pick the best node from the list
Unlock the list
Expand the node to generate successors

$P_0$

Lock the list
Place generated nodes in the list
Pick the best node from the list
Unlock the list
Expand the node to generate successors

$P_1$

Lock the list
Place generated nodes in the list
Pick the best node from the list
Unlock the list
Expand the node to generate successors

$P_{p-1}$
Parallel Depth First Search

- Communication Strategies for Parallel Best-First Tree Search:

In the *random communication strategy*, each processor periodically sends some of its best nodes to the *open* list of a randomly selected processor. This strategy ensures that, if a processor stores a good part of the search space, the others get part of it. If nodes are transferred frequently, the search overhead factor can be made very small; otherwise it can become quite large. The communication cost determines the best node transfer frequency. If the communication cost is low, it is best to communicate after every node expansion.
Parallel Depth First Search

• Communication Strategies for Parallel Best-First Tree Search:

In the *ring communication strategy*, the processors are mapped in a virtual ring. Each processor periodically exchanges some of its best nodes with the *open* lists of its neighbors in the ring. This strategy can be implemented on message passing as well as shared address space machines with the processors organized into a logical ring. As before, the cost of communication determines the node transfer frequency.
Parallel Depth First Search

- Communication Strategies for Parallel Best-First Tree Search:

Figure: A message-passing implementation of parallel best-first search using the ring communication strategy.
THANK YOU !!!!!

My Blog: https://anandgharu.wordpress.com/

Email: gharu.anand@gmail.com