CHAPTER 5

UNIT V

Parallel Algorithms - Sorting and Graph

Syllabus Topics

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.

Syllabus Topic : Issues in Sorting on Parallel Computers

5.1 Issues in Sorting on Parallel Computers

Q. 5.1.1 What are the issues in sorting on parallel computers ? (Refer sections 5.1 to 5.1.2) **(7 Marks)**

- The parallel version of the sorting algorithm is based on the task in which the elements to be sorted are distributed onto the number of available processes for sorting.
- There are number of points required to be focused to make it clear about the working of parallel sorting algorithms.

5.1.1 Where the Input and Output Sequences are Stored ?

- The input upon which sorting algorithm performs and the sorted sequences are stored in the process's memory in sequential sorting algorithm.
- On the other side in parallel programming the sequences are stored at two places, only one of the processes may contain sequences or distributed among the processes in case of distributed sorting approach is applied.
- We need to consider the distribution of the sorted output sequence among the processes precisely.
- A method in general considered is based on the enumeration of processes and thereafter this enumeration is specified as a global ordering for the sorted sequence.
- Once the enumeration of processes is decided the sequence will be sorted based on the enumeration.
- For example, consider two processes P_1 and P_2 are in the order P_1 followed by P_2 in the enumeration.
- In this case all elements stored in P_1 will be smaller than the elements stored in P_2. It is not restricted on only one approach but we can have number of ways possible for the enumeration of processes.
- Certain enumeration is decided based on the efficiency of the algorithm on that particular domain.

5.1.2 How Comparisons are Performed ?

- In sequential sorting approach two elements are compared and exchanged very easily because elements are stored in the process's memory locally.
- On the other hand this step is complex in the parallel algorithm design. The complexity of comparison and exchange occurs because elements reside on different processes.

5.1.2(A) One Element Per Process

- The simplest case to be considered is that the only one element to be sorted is hold by each process.
- It may happen that a pair of processes P_1 and P_2 need to compare their respective elements a_1 and a_2 at some point in their executions.
- Once the comparison is done the process P_1 will hold smaller element and P_2 will hold larger element of (a_1, a_2). In this simplest case, each process compares received element with its own and retains the appropriate element.
- According to our example, P_1 will keep smaller and larger element is kept by P_2.
- This operation is referred as compare and exchange approach in the sequential sorting algorithm design.
- The Fig. 5.1.1 describes the each compare-exchange operation which performs one comparison step and one communication step.

Fig. 5.1.1

5.1.2(B) More than One Element Per Process

- The parallel sorting algorithm can be applied on a large sequence with a small number of processes in comparison with the sequential algorithm. A block of n/p elements is assigned to each process. Here **p** is the number of processes and **n** represents the number of elements to be sorted.
- As we have discussed in the one element per process case, let us consider two processes P_1 and P_2 redistribute their blocks of n/p elements.
- After redistribution, assume one will get the smaller n/p elements and other process will get the larger n/p elements.
- This operation is called compare - split operation and is described in the Fig. 5.1.2.
- According to 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 this example, process P_1 retains the smaller elements an process P_2 retains the larger elements.

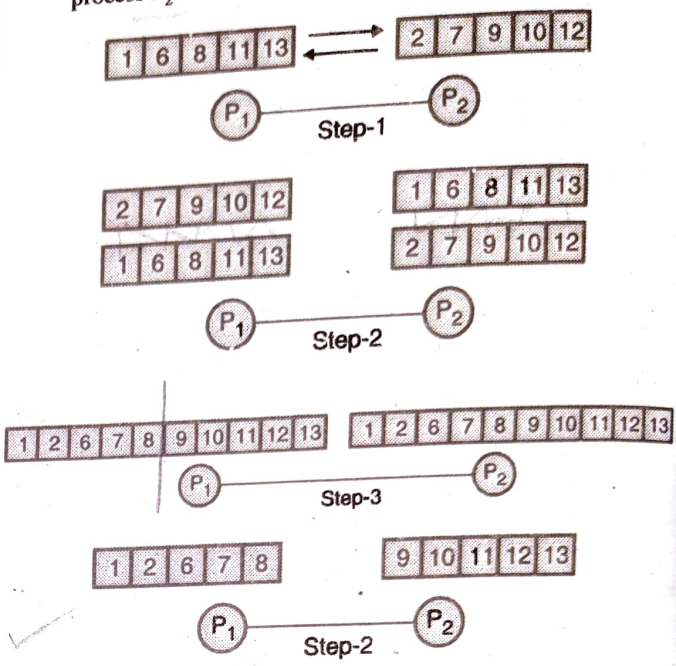


Fig. 5.1.2

Syllabus Topic : Bubble Sort and its Variants

5.2 Bubble Sort and its Variants

Q. 5.2.1 Explain Bubble sort with algorithm. (Refer section 5.2) **(5 Marks)**

- The adjacent elements in a sequence to be sorted are compared and exchanged in the sequential bubble sort algorithm. Suppose a given sequence to be sorted is $<a_1, a_2, \ldots, a_n>$, the bubble sort algorithm first performs n – 1 compare exchange operation in the following order :

$(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$

- This step moves the largest element to the end of the sequence. The last element in the transformed sequence is then ignored, and the sequence of compare-exchanges is applied to the resulting sequence.

Sequential bubble sort algorithm

- The sequential bubble sort algorithm is given below :

```
Algorithm Bubble_sort(n)
  {
    for(i = n- 1; i>= 1; i--)
    {
```

```
        for(j = 1; j<=i; j++)
        {
            Perform exchange (a_j + a_j+1);
        }
    }
}
```

5.2.1 Odd-Even Transposition

- There are two phases in this algorithm called as odd and even phases. In this algorithm, n elements are sorted in n phases where n is even.
- Consider a sequence to be sorted is $<a_1, a_2, \ldots\ldots, a_n>$. The odd phase works on the basis that, the elements with odd indices are compared with their neighbors, and found as out of sequence they are exchanged. This means the pairs with odd indices and their neighbor are compare - exchanged, for example, the pairs (a_1,a_2), (a_3,a_4),.....(a_{n-1}, a_n) are compare and exchanged if not in proper sequence, assume n as even here.
- In the similar fashion, consider the case with even phase. In this phase elements with even indices are compared with their right neighbors. After comparison among a pair and if found as out of sequence, they are exchanged. This means the pairs (a_2, a_3), (a_4, a_5).....(a_{n-2}, a_{n-1}) are compare - exchanged. The sequence is sorted after performing n - phases of odd - even exchanges.

Sequential odd-even transposition sort algorithm

- The pseudo code of the Sequential odd-even transposition sort algorithm is given below :

```
Algorithm even_odd(n)
{
  for(i = 1; i<= n; i++)
  {
  if( (i % 2) != 0) /*i is odd*/
  {
     for (j=0; j<= (n/2-1)
           perform_exchange (a_2j+1, a_2j+2) ;
  }
  else /*i is even*/
  {
     for (j=0; j<=(n/2-1)
        perform_exchange (a_2j, a_2j+1) ;
  }
}
```

- After n phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires O(n) comparisons, Serial complexity is $O(n^2)$. The Fig. 5.2.1 shows the working of the algorithm based on Sorting of (n = 8) elements.

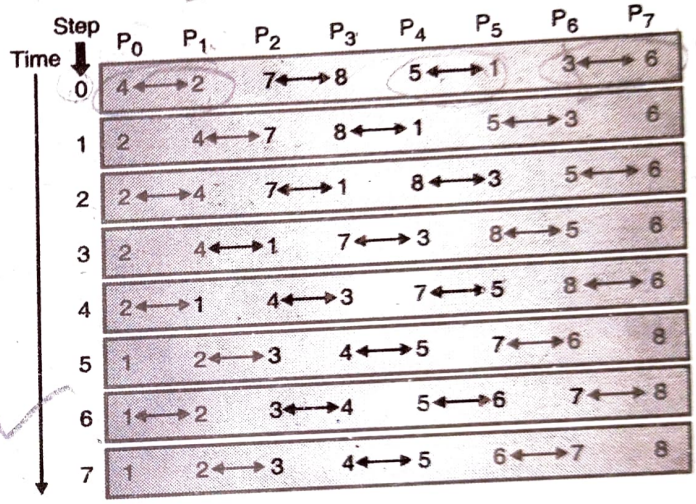


Fig. 5.2.1

5.2.1(A) Parallel Formulation

- The parallel version of the odd – even transposition sort can be done easily. The compare – exchange operations are performed simultaneously on pairs of elements during each phase of the algorithm.
- Consider the one – element – per process case and assume n be the number of elements to sorted and also the number of processes involved in the sorting.
- Here we will also assume that the processes are arranged in the one dimensional array. Initially all elements ai reside on processes P_i where i = 1,2,3..n. As discussed previously that each process with odd label performs compare – exchange its elements with the elements of the process reside as its neighbor on the right.

- The similar operation performs on the even phase where each process with even label compare – exchanges with the elements residing on its right.

☞ **Parallel version of odd-even transposition sort**

- This parallel approach is shown in the following algorithm

```
Algorithm parallel_even_odd(n)
{
  id = process_identifier;
  for(i = 1; i<= n; i++)
  {
    if( (i % 2) != 0) /*i is odd*/
    {
      if( (id % 2) != 0) /*id is odd*/
        perform_exchange_min(id+1);
      else
        perform_exchange_max(id-1);
    }
    if( (i % 2) == 0) /*i is even*/
    {
      if( (id% 2) == 0) /*id is even*/
        perform_exchange_min(id+1);
    else
      perform_exchange_max(id-1);
      }
  }
}
```

5.2.1(B) Shell sort

Q. 5.2.2 Write short note on Shell sort with example. (Refer section 5.2.1(B)) **(8 Marks)**

- This is one of the oldest sorting algorithms first invented by **Donald L. Shell** in 1959. This algorithm's complexity analysis is more sophisticated but it is easier in understanding and, implementation point of view. One of the significant merits of this algorithm is it is fast in nature.

- The approach of shell sort algorithm is described in following points:
  (a) The data sequence is arranged in a two dimensional array
  (b) Sorts the columns of the array.

- The data sequence automatically becomes in a partially sorted fashion because of the effects of the above points. The above process is repeated number of times but each time with a narrow array, i.e., with a smaller number of columns. At the end of the last step, the array consists of only one column. The sortedness of the sequence is increased in each step and in the last step it becomes a completely sorted sequence.

☞ **Example**

Consider a sequence of numbers to be sorted

3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

Step 1 : The sequence is arranged in an array with 7 columns (left), then the columns are sorted as shown on the right side.

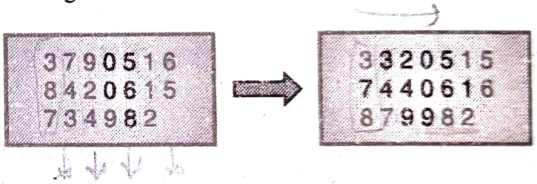


Fig. 5.2.2

- The observation from the first step is that, the data elements 8 and 9 are now reached to the end of the sequence but smaller elements 2 is still there.

Step-2 : The sequence is now arranged in three columns which are again sorted.

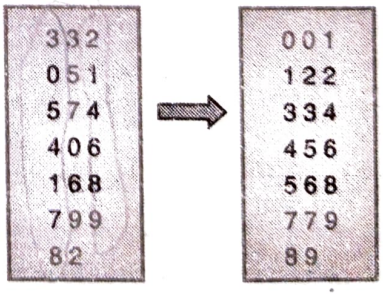


Fig. 5.2.3

- The sequence is almost sorted at this stage, when it will be arranged in one column in the last step. At this stage only elements 6, 8 and 9 have to move a little bit to their correct position.

5.2.1(B)1 General Scheme of Shell Sort Method

- The general scheme of the method is described below.
- The elements of n/2 pairs $(a_i, a_{n/2+i})$ for $1 \le i \le n/2$ are sorted during the first step of the algorithm. The elements of n/4 groups of four elements each $(a_i, a_{n/4+1}, a_{n/2+1}, a_{3n/4+1})$ for $1 \le i \le n/4$ are sorted during the second step. During the third step the elements of n/8 groups of eight elements each are sorted etc. All the elements of the array $(a_1, a_2, \ldots, a_n)$ are sorted at the last step. The insertion sort method is used at each step for sorting elements in groups. As it can be noted the total number of iterations of the Shell algorithm is equal to $\log_2 n$.

Sequential algorithm of Shell sorting

- The pseudo code of the shell algorithm is shown below :

```
// Sequential algorithm of Shell sorting
ShellSort(double A[], int n)
{
int incr = n/2;
while(incr> 0 )
{
   for ( int i = incr+1; i < n; i++)
   {
         j = i-incr;
         while ( j > 0 )
         if ( A[j] > A[j+incrj ) )
         {
            swap (A[j], A[j+incr]);
            j = j - incr;
         }
         else j = 0;
   }
   incr = incr/2;
}
}
```

5.2.1(B)2 Shell Sort Parallel Algorithm

- The shell sort method can be done in parallel if the communication network topology used is N – dimensional hypercube (if the number of processors is equal to $p = 2^N$). These processors may appear to be rather far from each other in case of linear enumeration. The required mapping the hypercube topology into the linear array structure may be implemented using the Gray code.
- In this case the sorting method can be performed into two divided sequential stages separately. At the first stage interaction of the processors in the hypercube structure takes place. The following simple rule can be used for forming pairs of processors interacting with each other during the compare – split operation : the processors whose bit codes of their numbers differ only in position (N – i) are paired at each iteration i, $0 \le i < N$.
- The usual interaction of the parallel odd-even transposition algorithm are performed at the second stage. The iterations of this stage are executed up to the actual termination of changes of the data being sorted. Thus, the total number L of such iterations may vary from 2 to p.
- The Fig. 5.2.4 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. This example is based for four processors and they are marked by circles. The binary number representation are used to give processor numbers.

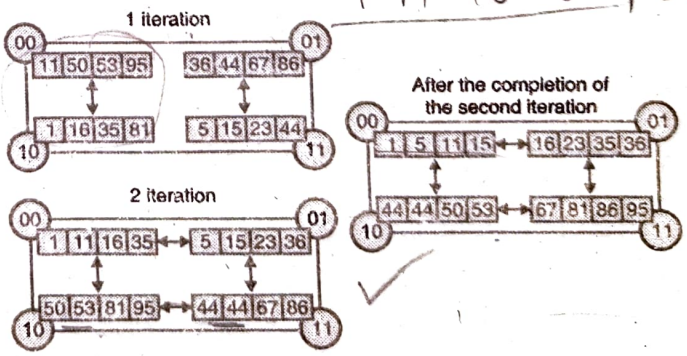


Fig. 5.2.4: The example of the use of the parallel Shell algorithm for 4 processors (the processors are marked by circles, the processor numbers are given in their binary representation)

- 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 coincides 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.

☞ **Efficiency analysis**

- The relations obtained for parallel bubble sort method of be used for estimating the efficiency of the parallel variant of the Shell algorithm.
- It is only necessary to take into account the two stages of the Shell algorithm. With regard to this peculiarity the total execution time for the new parallel method may be determined by means of the following expression:

$$T_p = \left(\frac{n}{p}\right) \log_2 \left(\frac{n}{p}\right) \tau$$

$$+ (\log_2 p + L)\left[\left(\frac{2n}{p}\right)\tau + \left(\alpha + \omega\left(\frac{n}{p}\right)/\beta\right)\right]$$

- As it can be noted, the efficiency of the parallel variant of Shell sorting depends considerably on the value L. If the value L is small, the new parallel sorting method is executed more quickly than the previously described odd-even transposition algorithm.

5.3 Quick Sort

- In the quick sort method the general consideration is that, the quick sort method is based on the sequential subdividing the sorted data into blocks of smaller sizes in such a way that the ordering relation is provided among the values of different blocks. Here it is noted that, for any pair of blocks all the values of one of the blocks do not exceed the values of the other one.
- The division of the original data into the first two parts is performed at the first iteration of the method.
- A certain pivot element is selected for providing this division, and all the values of the data, which are smaller that the pivot element, are transferred to the first block being formed.
- All the rest of the values form the second block of the sorted data. These rules are applied recursively for the two created blocks on the second iteration of the sorting etc.
- If the choice of the pivot elements is adequate, than the initial data array appears to be sorted after the execution of $\log_2 n$ iterations.
- The pseudo code for the quicksort scheme is described below. In this algorithm it is assumed that the pivot element is determined by the first element value of the sorted data.

☞ **Sequential algorithm of quick sort**

Q. 5.3.1 Write Sequential algorithm of quick sort. (Refer section 5.3) **(8 Marks)**

```
// The sequential Algorithm of Quick Sorting
QuickSort(double A[], int i1, int i2)
{
  if ( i1 < i2 )
  {
    double pivot = A[i1];
    int is = i1;
    for ( inti = i+1; i<i2;i++)
    if ( A[i] ≤ pivot )
    {
      is = is + 1;
      swap(A[is], A[i]);
    }
    swap(A[i1], A[is]);
    QuickSort (A, i1, is);
    QuickSort (A, is+1, i2);
  }
}
```

Syllabus Topic : Parallelizing Quick Sort

5.3.1 The Parallel Quick Sort Algorithm

- The parallel quicksort algorithm can be easily created for the computer system with topology of N-dimensional hypercube (i.e. $p = 2^N$).
- Assume initial data are divided in the blocks of same size n/p and distributed among processors. The enumeration of the hypercube processors used to decide the location of the data blocks.
- The following is the possible method to execute the first iteration of the parallel method :
  - Select the pivot element and broadcast it to all the processors (for instance, the arithmetic mean of the elements of some pivot processor may be chosen as the pivot element);
  - Subdivide the data block available on each processor into two parts using the pivot element;
  - Form the pairs of processors, for which the bit presentation of the numbers differs only in N position. After that the exchange of the data among these processors should be executed. As a result of these data transmissions, the parts of the blocks with the values smaller than the pivot element must appear on the processors, for which the bit position N of the processor numbers are equal to 0. The processors with the numbers in which the bit N is equal to 1 must collect correspondingly all the data values exceeding the value of the pivot element.
- The initial data appear to be subdivided into two parts after execution of the initial iteration.
- One of them (with the values smaller than the pivot element value) is located on the processors, whose numbers hold 0 in the N-th bit. There are only p/2 such processors.
- Thus, the initial N-dimensional hypercube also is subdivided into two sub-hypercubes of N – 1 dimension.
- The above described procedure may also be applied to these sub-hypercubes. After executing N such iterations, it is sufficient to sort the data blocks which have been formed on each separate processor to terminate the method.
- The Fig. 5.3.1 illustrates the parallel working of quick sort algorithm. It is based on the sorting data where n = 16, p = 4. This means each processor block holds four elements. The following notations used in the diagram:
- In the Fig. 5.3.1 processors are represented by 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 data partitioning is based upon the chosen optimal values of pivot elements. The value 0 was used for all the processors at the first iteration. The pivot value was equal to 4 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.

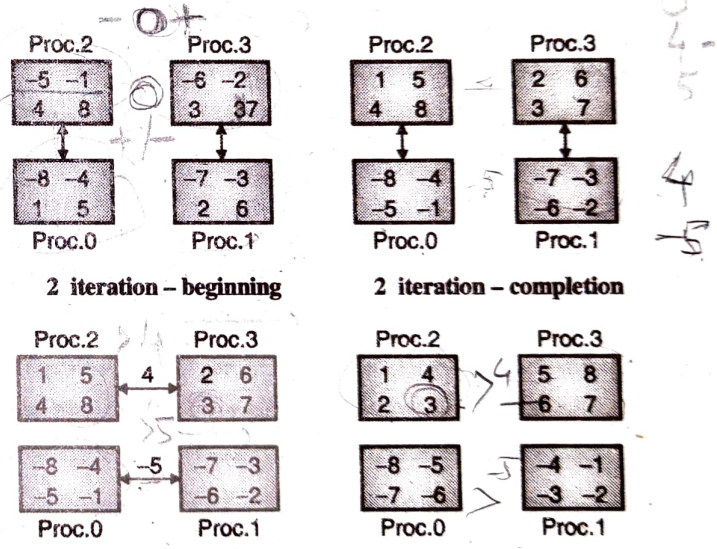


Fig. 5.3.1 : The example of sorting data by the parallel quick sort method

- As previously, the basic computational subtask may be the compare-split operation.
- The number of the subtasks coincides with the number of the processors used.
- The distribution of the subtasks among the processors should be done with regard to the efficient algorithm execution for the hypercube network topology.

Syllabus Topic : All-Pairs Shortest Paths

5.4 All-Pairs Shortest Paths

- The weighted graph G = (V,R) is used to describe the problem of finding all shortest paths. This graph G contains n number of vertices. The non – negative weight is assigned to each arc of the graph. It is also assumed that the graph considered here is oriented, i.e. if there is an arc from vertex A to vertex B, then there should not be an arc from the vertex B to the vertex A.
- The problem to be considered here is, to find the minimum paths among each pair of the vertices of the given graph G. As a practical example we may use the problem of working out the transport route between various cities, if the distances between them are given, and all the problems alike.
- We will use the Floyd algorithm for solving the problem of searching all the shortest paths.

5.4.1 Dijkstra's Algorithm

Q. 5.4.1 Write and explain Dijkstra's algorithm for finding the shortest path. (Refer section 5.4.1) **(5 Marks)**

- Dijkstra's algorithm originally used to find out shortest path from a single vertex to all other vertex in a graph.
- The same algorithm can be used for the all - pairs shortest - path problem by the simple expedient of applying it N times, once to each vertex V_0, ..., V_{n-1}.

Pseudo code for Dijkstra's sequential algorithm

- The pseudo code for the Dijkstra's sequential algorithm is given below :

```
Algorithm Seq Dijkstra (N)
{
    d_s = 0;
    d_i = infinity;
    while ( i!= s)
    {
        T= V;
        for(i=0; i< N-1; i++)
        { Search (V_m ∈ T with minimum d_m);
          For each edge (V_m, V_t) with V_t ∈ T {
          if (d_t > d_m + length ((V_m , V_t)))
              d_t = d_m + length ((V_m, V_t))
          }
          T = T – V_m;
        }
    }
}
```

- This algorithm maintains a vector T which contains vertices for which shortest paths have not been found. The shortest path from vertex V_s to vertex V_i and it is maintained in d_i.
- Initially T is initialized with V and all d_i with infinity. After performing each step of the algorithm, the vertex with the smallest value of d is removed from T.
- The path from the vertex V_i to its neighbor in T is examined to check a path through V_i would be shorter than the currently known path. Consider the following graph just to describe about the comparisons of various paths to find single source shortest path. Here the best known path from the vertex V_s to vertex V_t is compared with the path that leads from V_s to V_m and then to V_t.

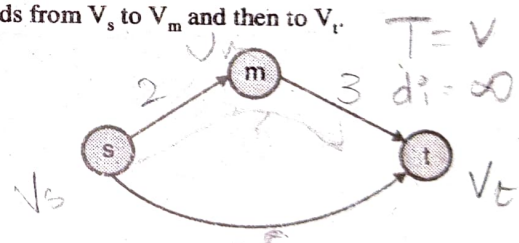


Fig. 5.4.1

- The single – source shortest path represented by the pseudo code shown above executes N times, once for each vertex. The total number of comparisons in this algorithm becomes $O(N^3)$.

Parallel version of the Dijkstra's algorithm

- There are two ways in which Dijkstra's algorithm can be performed in parallel. In the first version of the Dijkstra's parallel algorithm the graph is replicated in each of the P tasks. Here the sequential algorithm for N/P vertices is executed by each task.

- There is no communications requires in this algorithm but it can utilize at most N processors. This approach is called as source – partitioned approach for parallel formulation of the sequential algorithm.
- The second parallel version of the Dijkstra's algorithm is suitable for the case when P is greater than N. In this case we define N sets of P/N tasks and each set of tasks is given the entire graph and is responsible for computing shortest paths for a single vertex.
- The Fig. 5.4.2 shows the approach, here we have N = 9 and P = 36. This Fig. 5.4.2 also shows one set of P/N = 4 tasks in shaded form.

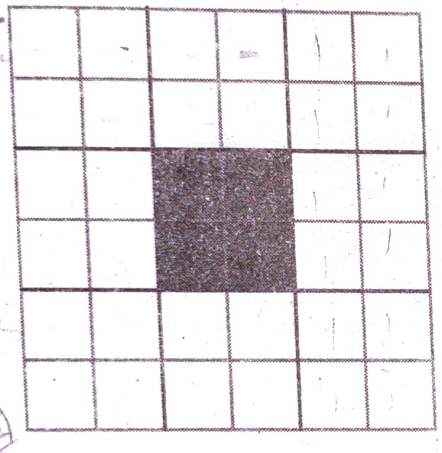


Fig. 5.4.2 : The second parallel Dijkstra algorithm allocates P/N tasks to each of N instantiations of Dijkstra's single-source shortest-path algorithm.

5.4.2 Floyd Algorithm

Generally the pseudo code of the algorithm proposed by Floyd for searching the minimum paths between all the pairs of vertices is given below. The complexity of this algorithm is in the order of n^3 where n is the number of vertices in the graph.

☞ Sequential Floyd algorithm

```
// The Sequential Floyd Algorithm
  for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
      A[i, j] = min(A[i, j],A[i, k]+A[k, j]);
```

- The minimum value operation named as min() must be performed here with taking into account the method of describing nonexistent graph arcs in the adjacency matrix.
- The adjacency matrix R changes while the algorithm is executed. At the end of the computation the required result will be stored in the matrix R.

☞ Computation Decomposition

- According to the general scheme of the Floyd algorithm, the task which takes the main computational load is choosing the minimum values in the problem of searching the shortest paths.
- It is not worthwhile to parallelize this rather simple operation, as it will not speed up the computation significantly. It is more efficient to update the values of matrix A simultaneously, as it will make parallel computations more effective.
- Let us illustrate the correctness of this method of parallelism implementation.
- For this we have to prove that the operation of updating the values of the matrix A during one iteration of the outer cycle k may be performed independently. In other words, we must prove that the elements A[i, k] and A[k, j] do not change at iteration k for any pair of indices (i, j). Let us analyze the expression, according to which the change of the elements of the matrix A happens :

$$A[i, j] \leftarrow \min (A[i, j], A[i, k] + A[k, j]).$$

For i = k we will have

$$A[k, j] \leftarrow \min (A[k, j] , A[k, k] + A[k, j]),$$

but then value A[k, j] will not change as A[k, k] = 0.

- For j = k the expression is transformed and reduced to $A[i, k] \leftarrow \min (A[i, k], A[i, k] + A[k, k])$
- Which also shows that values A[i, k] are unchangeable. As a result, the necessary conditions for parallel computations take place. Thus, the operation of updating the elements of the matrix A may be used as the basic computational subtask (to identify subtasks we will use the indices of the elements, which are updated in them).

☞ Analysis of Information Dependencies

- The elements of the matrix A are divided and assigned to different subtasks to perform the computations in parallel, for example subtask(i, j) contains elements A[i, j], A[i, k],

A[k, j] of the matrix A, which are necessary for the computations.

- The possible data duplication can be eliminated by assigning only element A[i, j] in the subtask (i, j). Other values required to perform the computations can be obtained by means of data transmission.
- In this way, each element A[k, j] of the row k of the matrix A must be transmitted to all the subtasks (k, j), $1 \leq j \leq n$, and each element A[i, k] of the column k of the matrix A must be transmitted to all the subtasks (i, k), $1 \leq i \leq n$.
- This is depicted in the Fig. 5.4.3 where the arrows show the direction of exchanging values at iteration k.

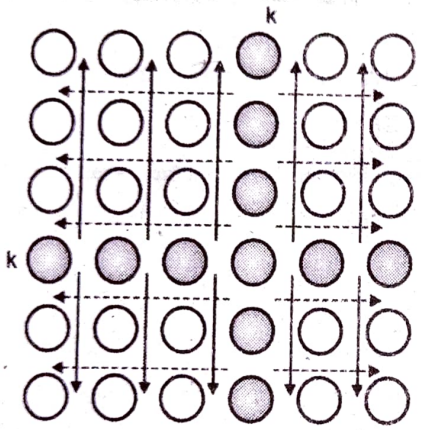


Fig. 5.4.3 : The information dependencies of the basic subtasks

Scaling and Distributing Subtasks among Processors

- The number of basic subtasks n^2 are always larger than the available processors P on which tasks can be performed ($p << n^2$).
- The possible way to perform the aggregation of the computation is the use of block-striped scheme decomposition of the matrix A. Here the computations related with updating the elements of one or several rows (rowwise or horizontal partitioning) or columns (columnwise or vertical partitioning) of the matrix A are united into one basic subtask.
- The two decomposition schemes described here are practically equal. We will further apply only partitioning the matrix A into horizontal stripes because the fact is arrays are located rowwise in C programming language.
- In this method of data decomposition it is necessary to transmit among the subtasks only the elements of one of rows of the matrix A. The network topology for efficient execution of this communications is a hypercube or a complete graph.

Efficiency Analysis

- The efficiency of the Floyd algorithm is analyzed and don into two stages. In the first stage, the order of the algorithm computational complexity will be estimated.
- The second stage is based on the specification and estimation of the time complexity of the data computations.
- As it is mentioned previously that the total complexity of the sequential algorithm is n^3. Each processor performs the update of the matrix A elements at each iteration of the parallel algorithm.
- Each subtask contains n^3/p such elements. The total number of iterations performed by the parallel algorithm is equal to n. Therefore, the speedup and efficiency characteristics of the Floyd algorithm look as follows:

$$S_p = \frac{n^3}{\left(\frac{n^3}{p}\right)} = p \quad \text{and} \quad E_p = \frac{n^3}{p \cdot \left(\frac{n^3}{p}\right)} = 1$$

- These are the ideal characteristics of the parallel computations performed in the algorithm. To specify the obtained relations we will introduce the execution time of the basic operation of choosing the minimum value into the obtained expressions. We will also take into account the overhead of data communications among the processors.
- The communication operation consists of transmitting a row of the matrix A to all the processors at each iteration of the algorithm. The execution of this operation is done in log p steps and the total duration of the data communication can be defined by the following expressions :

$$T_p(\text{comm}) = n \lceil \log_2 p \rceil (\alpha + \omega n/\beta)$$

- Here,
  - α represents the latency of the transmission network,
  - β represents its bandwidth,
  - ω is the matrix element size in bytes

- The total execution time for the Floyd parallel algorithm may be defined in the following way :

$$T_p = n^2 \cdot \left\lceil \frac{n}{p} \right\rceil \cdot \tau + n \cdot \lceil \log_2 (p) \rceil (\alpha + \omega \cdot n/\beta)$$

where τ is the execution time of choosing the minimum value.

☞ **Designing parallel algorithm partitioning**

The overall steps of designing the parallel algorithm based on the Floyd sequential approach of finding all pairs shortest path is given below :

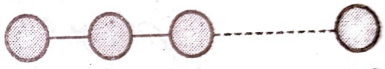
- Choose either domain decomposition or functional decomposition
- Same assignment statement executed n^3 times
  - No functional parallelism
- Easy to do domain decomposition
  - Divide matrix A into n^2 elements
  - Associate a primitive task with each element
- Communication
  - Each update of element a[i][j] requires access to elements a[i][k] and a[k][j]
  - For a given value of k
    - Element a[k,m] is needed by every task associated with elements in column m
    - Element a[m,k] is needed by every task associated with elements in row m
  - In iteration k, each element in row k gets broadcast to the tasks in the same column
  - Each element in column k gets broadcast to tasks in the same row
  - Do we need to update every element of matrix concurrently?
    - Values of a[i][k] and a[k][j] do not change during iteration k
    - Update to a[i][k] is a[i][k] = min (a[i,k], a[i,k] + a[k,k]);
    - Update to a[k][j] is a[k][j] = min (a[k,j], a[k,k] + a[k,j]);
    - In both the above updates, a[i][k] and a[k][j] cannot decrease (all numbers are positive)
    - Hence, no dependence between updates of a[i][j] and the updates of a[i][k] and a[k][j]
    - For each iteration k of the outer loop, perform broadcasts and update every element of matrix in parallel

Syllabus Topic : Algorithms for Sparse Graphs

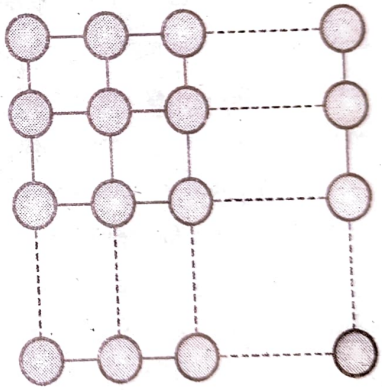
5.5 Algorithms for Sparse Graphs

Q. 5.5.1 Explain sparse graphs with suitable diagram. (Refer section 5.5) **(4 Marks)**

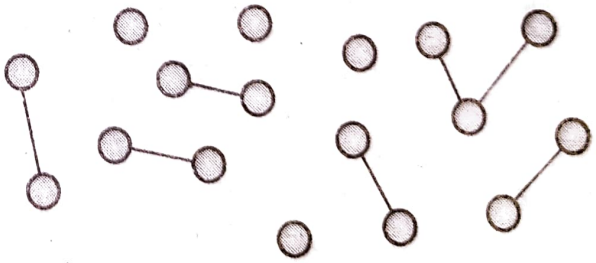
- A graph *in which the number of* edges *is much less than the possible number of edges.*
- Some of the sparse graph examples are shown in the Fig. 5.5.1.



(a) A linear graph, in which each vertex has two incident edges



(b) A grid graph, in which each vertex has four incident vertices



(c)

Fig. 5.5.1

☞ **A random sparse graph**

- The algorithms invented for the dense graphs also work for sparse graphs but the sparseness nature is considered in the algorithm design shows significant performance improvements.

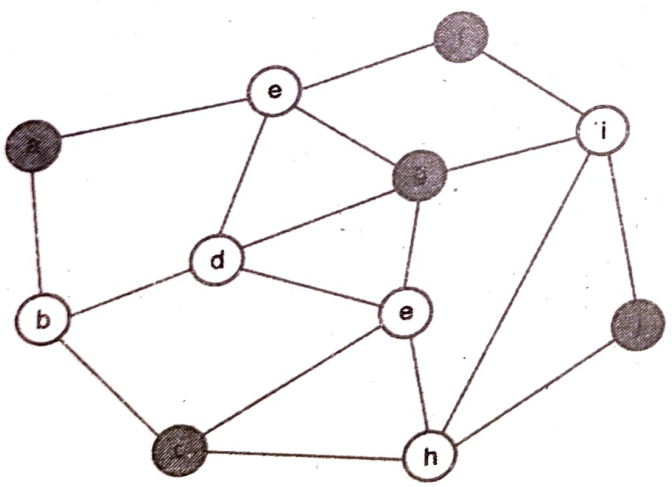
- The use of adjacency list instead of adjacency matrix is adopted in the algorithm development for sparse graphs. This change is important to be considered because the complexity of the matrix based algorithms is usually $W(n^2)$ and it does not depend upon the number of edges also. As another side the complexity of the adjacency list based algorithms is usually W(n + |E|), which is based on the sparseness characteristics of the graph.

5.5.1 Finding a Maximal Independent Set

- Assume G is a graph of set of vertices and edges. An independent set in G is a set of vertices I in G such that no two vertices in I are adjacent or neighbors.
- The problem of finding a maximum independent set is described as, to compute an independent set of maximum size that is maximum number of vertices in a given graph G.

☞ Example

- The Fig. 5.5.2 is used to describe the independent set and the maximal independent set in a given graph. In this example we found two maximal independent sets and because of that it is stated that the maximal independent sets are not unique. We can determine which computations can be done in parallel using the maximal independent sets of vertices. As an example, the sets of rows that can be factored concurrently in parallel incomplete factorization algorithms.



{a, d, i, h} is an independent set
{a, c, j, f, g} is a maximal independent set
{a, d, h, f} is a maximal independent set
Fig. 5.5.2

- In the starting of the simplest algorithm for computing a maximal independent set of vertices, 'I' is initially set to empty, and all vertices are assigned in a set C considered as a candidate set of vertices for inclusion in 'I'. In the further step of the algorithm a vertex v from C is moved into 'I' repeatedly. In this way all vertices adjacent to v from C are removed until C becomes empty. The process terminates when C becomes empty and in this case 'I' is a maximal independent set.
- The set I will contain an independent set because a vertex is removed from C and added into I repeatedly if its subsequent inclusion violates independence condition. The resultant set becomes a maximal set also because any other vertex that is not already in *I* is adjacent to at least one of the vertices in *I*.

The algorithm is described in the following ways :

☞ Maximal Indpendent set algorithm

```
Algorithm MIS(G)
{
i = ∅, V' = V
while (V' == ∅ )
{
   Choose any v ∈ V' (in lexicographically order)
   Set i = I ∪ v
   Set V' = V'\(v ∪ Neighbor(v))
}
Return i
}
```

☞ Luby's randomized maximal independent set algorithm

- This algorithm described above is simple but does not suit well for parallel processing as it is serial in nature. The algorithm originally developed by Luby for graph coloring is adopted for performing MIS computation in parallel.
- A maximal independent set computation using Luby's algorithm works in incremental fashion and described in the following ways.
- Initially, the set I is initialized to be empty and the set of candidate vertices C is set to be equal to V. Each vertex included in the candidate set C is assigned with a unique

random number, and a vertex will be considered for inclusion in 'I' if the random number assigned into it is smaller than the random numbers of all the adjacent vertices.

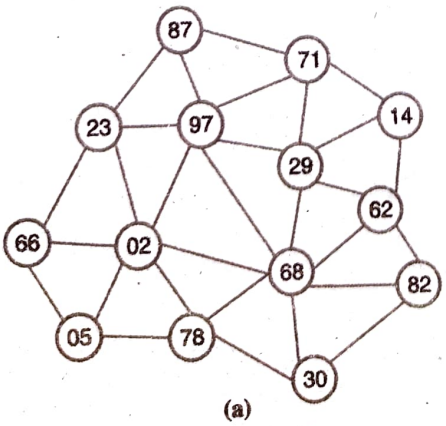
- The steps of random number assignment and vertex selection are repeated for the remaining vertices in C, and similarly vertices are added in set I. Addition of vertices on I ends when there is no vertex left in the candidate set C.

```
Algorithm Maxirnal_independent_set{G)
/*Given a graph find a maximal independent set.*/
{
 I= ϕ, G' = G
while ( isEmpty{G')!= True) /*G' is not the empty graph*/
{
1. Choose a random set of vertices S ∈ G' by selecting each
   vertex v independently with probability 1/⊆ (2d(v))
2. For every edge (u, v) ∈ E{G') if both endpoints are in S
   then remove the vertex of lower degree from S (Break ties
   arbitrarily). Call this new set S'.
3. I = I ∪ S' . G' = G' \ (S' ∪ N( S')), (G' is the induced
   subgraph on V'\(S'∪ N(S'))
}
return I
 }
```

- The different steps in Luby's algorithm for maximal independent set is shown in Fig. 5.5.3. The numbers inside each vertexcorrespond to the random number assigned to the vertex.

☞ **Steps for Luby's randomized MIS**

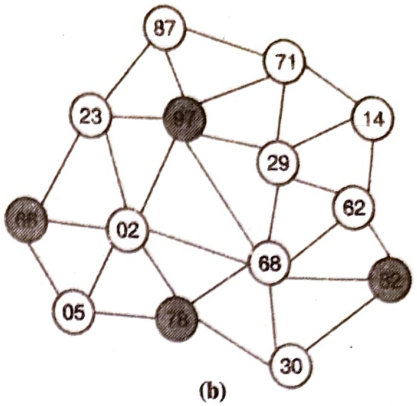
(1) Initially assigned random numbers



(a)

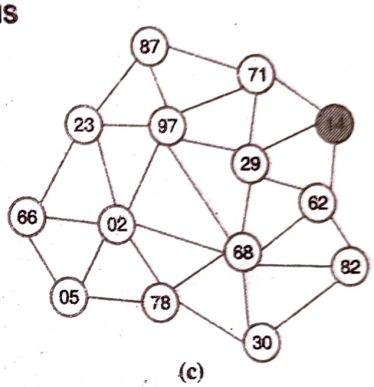
Fig. 5.5.3 : Contd...

(2) Find maximal independent set



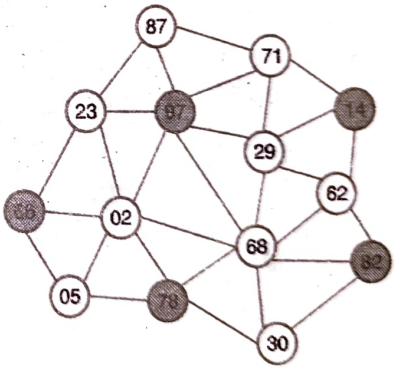
(b)

(3) Find MIS



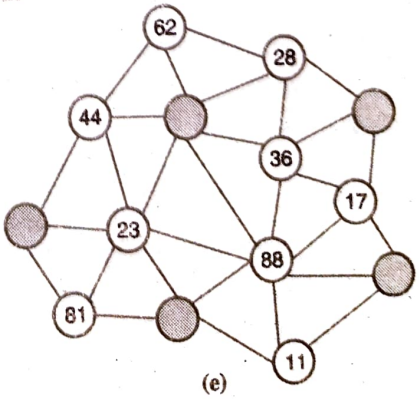
(c)

(4) Find MIS, color all the same color



(d)

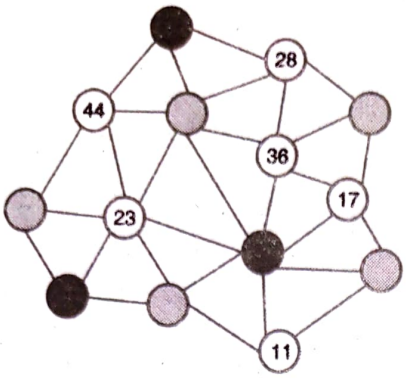
(5) Repeat



(e)

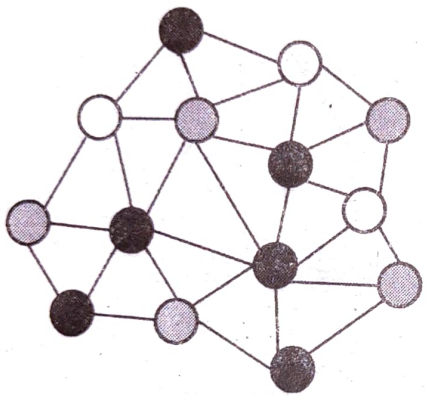
Fig. 5.5.3 : Contd...

(6) Repeat



(f)

(7) Repeat



(g)

(8) Repeat

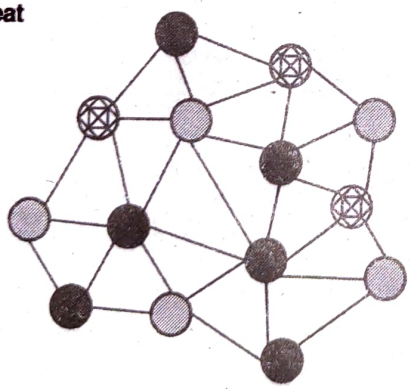


Fig. 5.5.3(h) : Luby's randomized MIS

5.5.2 Parallel Formulation of Luby's Algorithm in Shared-Address-Space Computer

- The process of Luby's algorithm parallel formulation on parallel computer is described as follows:
- Assume an array I of size equal to number of vertices in the graph (i.e. |V|). When the algorithm terminates the ith position of the array I will store one (i.e., I[i]) if vertex vi is the part of MIS, otherwise it contains value zero.
- All the elements in I are initially set to 0, some of the values are changed to 1 from 0 during each iteration of the Luby's algorithm.
- Also assume another array C which is also of size |V| and all elements of C are set to one initially. When algorithm executes the values in array will contain one at a particular element position if vertex vi is part of the candidate set, or will contain zero otherwise.
- Let's finally assume an array R of size |V| and the random numbers assigned to each vertex are stored in it.
- The set C is partitioned logically and assigned to P processes for processing during each iteration of the algorithm. A random number will be generated by each process for each and every vertices assigned from C. The step for which vertex can be included in the array I will be determined by the processes when random number generation for all the vertices are completed. Particularly, for each vertex assigned to them, they check to see if the random number assigned to it is smaller than the random numbers assigned to all of its adjacent vertices. If it is true, they set the corresponding entry in *I* to one. Because *R* is shared and can be accessed by all the processes, determining whether or not a particular vertex can be included in *I* is quite straightforward.
- Array *C* can also be updated in a straightforward fashion as follows. Each process, as soon as it determines that a particular vertex *v* will be part of *I*, will set to zero the entries of Corresponding to its adjacent vertices. Note that even though more than one process may be setting to zero the same entry of *C* (because it may be adjacent to more than one vertex that was inserted in *I*), such concurrent writes will not affect the correctness of the results, because the value that gets concurrently written is the same.

5.5.2(A) Single-Source Shortest Paths

Q. 5.5.2 Explain Single-Source Shortest Paths with algorithm. (Refer section 5.5.2(A)) **(5 Marks)**

- **Johnson's** algorithm is the modified version of the Dijkstra's single source shortest path algorithm for funding shortest path in sparse graphs efficiently.

– A priority queue Q is used to store the value I[V] for each vertex V belongs to (V - V_T). The smallest value in I is always kept at the front of the priority queue.

Dijkstra's single source shortest path

– The pseudo code of the algorithm is shown below :

```
Algorithm Johnson_SingleSource(V,E,c)
{
    Q = V;
    for (all v in Q)
    I[v] = infinity;
    I[s] = 0;
    while (Q!== 0)
    {
    u = get_min(Q);
    for ( each v ∈ Adj[u])
        if( ((v ∈ Q)&& (I[u] + w(u, v))) < I[v])
        I[v] = I[u] + w(u, v);
    }
}
```

– The value I[V] = infinity is inserted in the priority queue initially for each vertex V other than the source vertex. The value zero is inserted for the source vertex S (i.e. I[S] = 0). The vertex with the minimum value in I is removed from the priority queue at each step of the algorithm.

– The algorithm uses heap because it improves the overall execution time of the algorithm. The adjacency list is traversed for the node u and the distance I[V] to vertex v is updated in the heap. The total number of updates is equal to the number of edges.

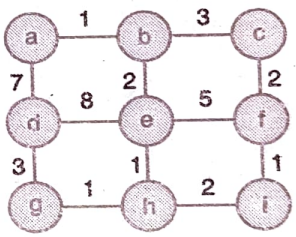
5.5.3 Parallel Johnson's Algorithm with Centralized Queue

– The steps of the algorithm are given below :

```
Algorithm parallel_Johnson_SingleQ()
{
Priority Q is kept by one processor P1
Other_Processors (Pn-1) do update (I[v] for v ∈ (V –VT))
Send (updated values to P1 by Pn-1)
/*
In each iteration the algorithm updates roughly |E|/|V|
vertices → a maximum of |E| / |V| proc.
For each new edge the queue is updated in O(log n) time (take
the old I[v] away and put a new value in)
A total of |E| edges requires O(|E| log n)time. The same
order as the sequential algorithm
*/
}
```

– Maintaining strict order of Johnson's algorithm generally leads to a very restrictive class of parallel algorithms. We need to allow exploration of multiple nodes concurrently.

– This is done by simultaneously extracting P nodes from the priority queue, updating the neighbors' cost, and augmenting the shortest path. If an error is made, it can be discovered (as a shorter path) and the node can be reinserted with this shorter path.



Priority Queue

Infinity → inf

(1) b : 1, d : 7, c:inf, e:inf, f:inf, h:inf, i:inf

(2) e : 3, c : 4, g : 10, f : inf, h : inf, i : inf

(3) h : 4, f : 6, i : inf

(4) G : 5, i : 6

a	b	c	d	e	f	g	h	i
0	1	inf	7	inf	inf	inf	inf	inf
0	1	4	7	3	inf	10	inf	inf
0	1	4	7	3	6	10	4	inf
0	1	4	7	3	6	5	4	6

Fig. 5.5.4

– Even if we can extract and process multiple nodes from the queue, the queue itself is a major bottleneck. For this reason, we use multiple queues, one for each processor. Each processor builds its priority queue only using its own vertices. When process P_i extracts the vertex $u \in V_i$, it sends a message to processes that store vertices adjacent to u. Process P_j, upon receiving this message, sets the value of $l[v]$ stored in its priority queue to $\min\{l[v], l[u] + w(u,v)\}$

- If a shorter path has been discovered to node v, it is reinserted back into the local priority queue. The algorithm terminates only when all the queues become empty. A number of node partitioning schemes can be used to exploit graph structure for performance.

5.5.4 Parallel Johnson's Algorithm with Distributed Queue

- The steps performed by this algorithm are given below:

```
Algorithm parallel_Johnson_DistributedQ()
{
  Distribute vertices V on different processors in p disjoint
  sets such that P_i has V_i.
  Each processor
  - has a priority queue Q_i for their own vertices
  - has a vector sp, where in the end sp[v] will be the cost
    for the shortest path from s to v
  - executes Johnson's algorithm on their own subgraph
    Initially l[V] = ∞   for all vertices excepts (l[s] = 0)
    Each time a vertex v is removed from the queue set
    sp[v] = l[v]
}
```

- Assume p represents the number of processes and a sparse graph is represented by G = (V,E). The set of vertices V is partitioned into V_1, V_2,...,V_p disjoint sets. Here one process of the p processes is assigned with a set of vertices and its associated adjacency lists.

A priority queue is maintained by each process where it stores the vertices assigned to it.

The shortest path is computed by each process from the source to the vertices assigned into it. Therefore, the disjoint priority queues Q_1, Q_2,..., Q_p is created for each process by partitioning the priority queue.

Each partitioned priority queue Q_1, Q_2,...,Q_p is assigned to a separate process. Each process maintains an array named as sp to store the shortest path from the source vertex to v for each vertex v in V_i.

That is the array sp[v] stores the cost of the shortest path from source to V_i.

- Each time a vertex v is extracted from the priority queue the respective cost from sp[V] is updated to l[V]. Initially all positions of the array sp is initialized with ∞, that sp[v] = ∞, for every vertex v other than the source vertex, and we insert $l[s]$ into the appropriate priority queue for the source vertex s.
- Now johnson's algorithm is executed by each process in its local priority queue. After completion of the algorithm, the length of the shortest path from source to vertex v is stored in sp[v].
- Consider a process P_1 extracts the vertex u on the basis that it has the smallest value of l[u] from Q_i. At this point, the l values of vertices assigned to processes other than P_1 need to be updated. The notification message from process P_1 about the new values received by all processes that store vertices adjacent to u.
- The values of l are updated by the processes upon receiving the new values. For example, assume an edge (u,v) such that u is in V_1 and v is in V_2, and a process P_1 has just now extracted vertex u from its priority queue.
- Then the message containing potential new value of l[V], which is l[u] + w(u, v) is sent by process P_1 to process P_2. Process P_2, upon receiving this message, sets the value of $l[v]$ stored in its priority queue to $\min\{l[v], l[u] + w(u, v)\}$.
- It may be possible that process P_2 has already extracted vertex v from its priority queue because both the processes P_1 and P_2 execute johnson's algorithm. This indicates that the sp[V] representing the shortest path from source to vertex V is already computed by the process P_2. At this stage there will be two possibilities : either $sp[v] <= l[u] + w(u, v)$, or $sp[v] > l[u] + w(u, v)$.
- This means the first case indicates that the longer path to vertex v passing through u and second case indicates shorter path. Process P_2 does not need to do anything in the first case because the shortest path to v does not change. Whereas in the second case it is necessary for process P_2 to update the cost of the shortest path to vertex v. Here P_2 update by inserting the vertex v back into the priority queue with $l[v] = l[u] + w(u, v)$ and disregarding the value of $sp[v]$.Since a vertex v can be reinserted into the priority queue, the algorithm will be terminated only when the situation, where all queues will not contain any values.

Dependencies in Johnson's algorithm

- If $u \in V_i$ and $v \in V_j$ when P_i removes u from Q_i then P_i sends information to P_j about that $l[v]$ possibly could have the a value $l[u] + w(u, v)$. P_j sets $l[v] = \min\{l[v], l[u] + w(u,v)\}$.
- Since P_j also executes Johnson's algorithm P_j may already have removed v from the queue Q_j. Then we can have two cases:
  - If $l[u] + w(u,w) \geq sp[v]$ then the already found path is shorter. Then P_j does not have to do anything.
  - If $sp[v] > l[u] + w(u,w)$ then the path via u is shorter than the shortest path found so far. P_j lets $l[v] = l[u] + w(u,w)$, removes the value of sp[v] and puts back v into Q_j.
  - The algorithm is not terminated until all queues are empty. Note, that due to the distributed queue the removal of values from the queue is not always done in the "right" order and then "unnecessary" work is done by the parallel algorithm.

Syllabus Topic : Parallel Depth First Search (DFS)

5.6 Parallel Depth First Search (DFS)

Q. 5.6.1 Explain Parallel Depth First Search (DFS). (Refer section 5.6) (7 Marks)

- Depth – first search algorithm is a tree – based graph traversal algorithm. This algorithm is used to search a graph. In DFS approach the traversal in a graph nor tree starts from the parent vertex down to its children and grandchildren vertices in a single path till the dead – end.
- In case when it is found that, no more vertices present in a path, the DFS algorithm backtrack to a point where it can choose another path to take. This process is repeated again and again until all vertices have been visited.
- The distribution of the search space among the processors is one of the most crucial issues in parallel depth – first search algorithm. This can be illustrated using the diagram as shown in the Fig. 5.6.1. The parallel search process can be performed on the left subtree rooted at A and right subtree rooted at B.

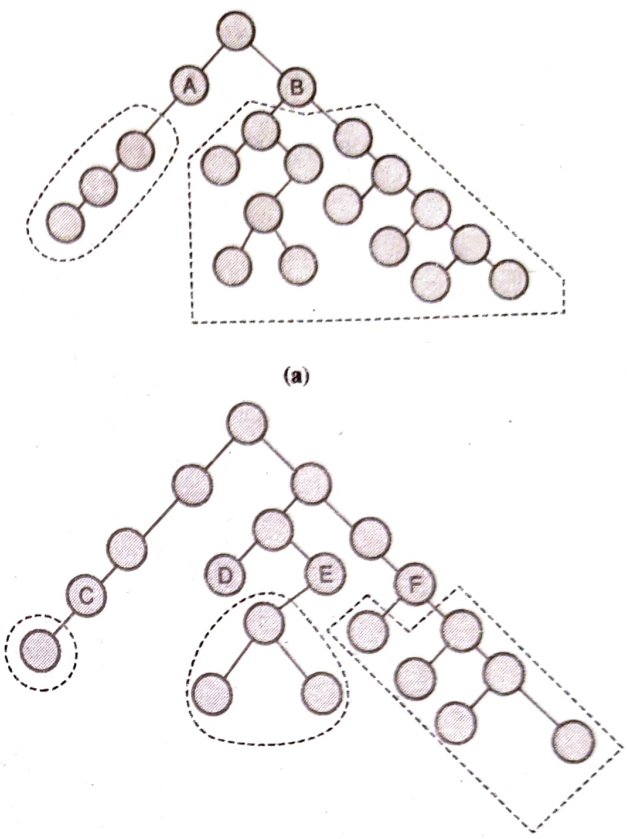


Fig. 5.6.1 : Parallel depth - first search

- A node in the tree is statically assigned to a processor and therefore, it is possible to expand the whole subtree which is rooted at that node without requiring communication with another processor. This way a good parallel search algorithm can be achieved based on static allocation.
- Now we will see when this approach is applied in the tree given above. Also assume that we have two processors. The two nodes A and B are assigned to two different processors after expansion of root node of the tree.
- Now the simultaneous search procedure starts in each subtree rooted at a particular node.
- This approach suffers with the problem of processor idleness. A processor becomes idle for a significant amount of time because of the fewer nodes in a particular subtree, for example in the Fig. 5.6.1 subtree rooted at node.
- A has fewer nodes as compare to subtree rooted at B. The actual reason for the processor idleness is due to the workload imbalance. The situation becomes worst if the number of processors increases.

- 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 performance of the static partitioning is poor in the tree which is considered as unstructured tree because of the variations in the partition sizes of the search space rooted at different nodes.
- The search space is generated dynamically and because of that it becomes necessary to balance the search space among processors dynamically.
- Dynamic load balancing is used when one processor becomes idle and other is heavily loaded.
- The idle processor get the work from other processor and load balancing is handled. For example the two partitions of the tree shown in the Fig. 5.6.1 where nodes A and B are assigned to two processors.
- When the processor in which A is assigned becomes idle, it gets work from the other processor.
- The dynamic distribution approach reduces the work imbalance among processors but because of the communication from one to another for work transfer it may suffer with overheads.
- In spite of this communication overhead issue the approach is able to satisfy work balance among processors.

Parallel formulation of DFS based on dynamic load balancing

- The search space for each processor is assigned in a disjoint fashion and each processor performs DFS on its own search space. 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.
- The comparison of this approach can be done with the message - passing architecture and shared address space machines. In this approach work request and response are like message request and response used in the message passing architecture.
- In the similar ways locking and extracting work in shared address space machines are also like work request and response. In this approach, all the processors terminates whenever a processor finds a goal node. If no solution is found of the problem for the finite search space, then all the processors eventually run out of work.

5.7 Best-First Search

- The best – first search algorithm traverses a graph to reach a target in the shortest possible path. Best-First Search follows an evaluation function to determine which node is the most appropriate to traverse next.

Steps of the best – first search algorithm

Step 1 : Start with the root node and mark it as visited.

Step 2 : Find the next appropriate node from the root node and mark it visited.

Step 3 : Go to the next level and find the appropriate node and mark it visited.

Step 4 : Continue this process until the target is reached.

Pseudo code of the algorithm

```
Algorithm BFS(m)
{
    Insert (m at start node)
    Insert( m.StartNode)
    while (isEmpty(PQ) !=True)/* Until PriorityQueue is
    empty*/
    {
    c ←-DeleteMin(PQ);/*Delete minimum from priority
    queue*/ if (found(c)) /*c is the goal*/
        exit();
    else
        for(each neighbor n of c)
        if (visited(n) != True)
            Mark n "Visited"
            Insert( n)
            Mark c "Examined"
    }
}
```

- The best first search algorithm can be applied on graphs and trees. Best-First makes use of a heuristic (or quickly computed estimate of the cost to reach the goal from each node), called $\hat{h}$, to guide its search. The idea behind using a heuristic to guide the search is that the algorithm will not waste time probing paths that do not seem likely to lead to the goal state (node).
- However, this means that an inaccurate $\hat{h}$ function can misguide the search, which can result in the search finding a path to the goal that is not optimal. For this reason, one of the greatest challenges in graph searching is to develop a heuristic function that can be quickly computed and that will be a very close estimate of the actual cost to the goal.
- The Best-First Search Algorithm maintains an open queue of nodes that is sorted in ascending order according to the each node's $\hat{h}$ value. Thus, the Best-First Search only uses the heuristic function when deciding which node to close next.
- This reliance on the $\hat{h}$ function can cause the Best-First Search Algorithm to close a node with greater-than-optimal cost, as in the following simple example:

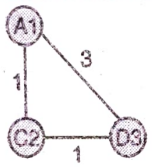


Fig. 5.7.1

- In the above diagram, when node D is closed, node A will be added to the open queue with a value of $\hat{h}(A) = 1$, and node C will be added to the open queue with a value of $\hat{h}(C) = 2$. Since $1 < 2$, node A will be closed with the cost of 3. However, the shortest path to node A is DCA, which has a cost of 2.
- Because the Best-First Search Algorithm only relies upon $\hat{h}$ when choosing the next node to close, it closed node *A* before finding the shortest path.
- Therefore, the Best-First Search is not guaranteed to find the optimal path from the start node to the goal node and can be misled by an errant heuristic function.

Syllabus Topic : Parallel Best-First Search

5.7.1 Parallel Best-First Search

- As we have discussed that the open list maintains the unexpected nodes in the search graph which is ordered according to their $\hat{h}$-values. The most promising node from the open list is removed and expanded in the sequential Best – First – Search algorithm. Thereafter the newly generated nodes are added in the open list.

Centralized strategy

- Different nodes from the open list concurrently expanded by different processors in the parallel implementation of the BFS algorithm. In the centralized strategy each processor is assigned to work on one of the current best nodes on the open list. In this approach a single global open list is used from where each processor gets work.

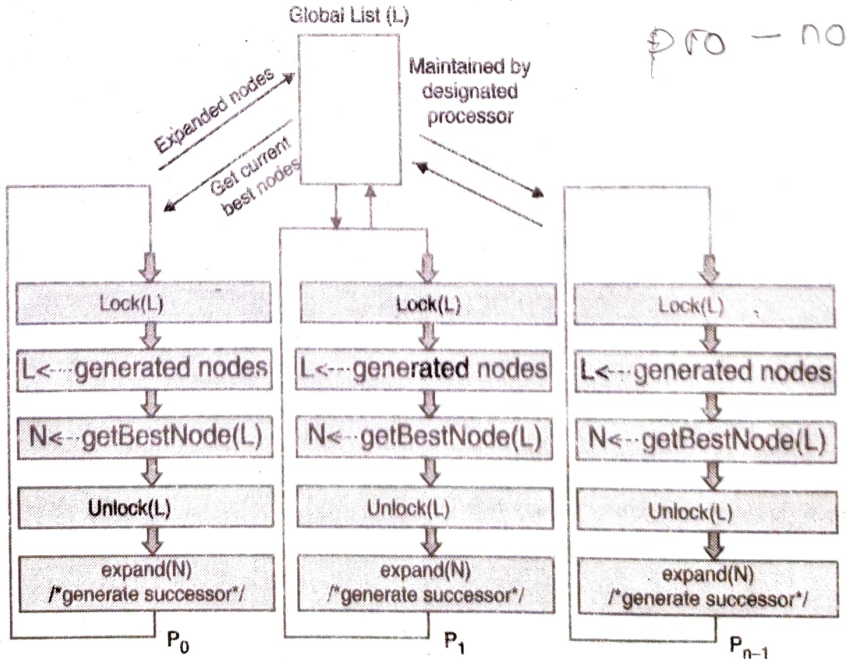


Fig. 5.7.2 : A general schematic for parallel best-first search using centralized strategy

- This parallel approach is used to expand more than one node at a time, and therefore it is possible to expand nodes that would not be expanded by a sequential algorithm. Let's consider a case where first node on the open list represents a solution
- The parallel working of best – first – search algorithm expands the first P nodes from the open list. In this case even-though more nodes have been expanded, the amount of extra work is limited because it always picks up best P nodes.
- Fig. 5.7.2 illustrates the centralized strategy.

Communication Strategies for Parallel Best-First Tree Search

In this section we will discuss three strategies which are used to ensure that nodes with good I – values are distributed evenly among processors.

1. Random communication strategy

- In this approach each processor sends some of its best nodes to the open list of a randomly selected processor.
- This strategy ensures that, the part of the search space get by others will be good if a processor stores a good part of the search.
- The search overhead factor will be reduced if nodes are transferred frequently; otherwise it can become quite large.
- The best node transfer frequency depends upon the communication cost. It is best to communicate after every node expansion, if the communication cost is low.

2. Ring communication strategy

- In this approach all processors are arranged in a virtual ring. Some of the best nodes are exchanged by each processor with the open list of its neighbors in the ring periodically.
- The message passing and shared space machines can have implementation of this strategy where processors are organized into a logical ring.
- The Fig. 5.7.3 describes the ring communication strategy.
- The search overhead factor of this scheme is very high if search space is not highly uniform. The overhead occurs because the scheme takes a long time to distribute good nodes from one processor to all other processors.

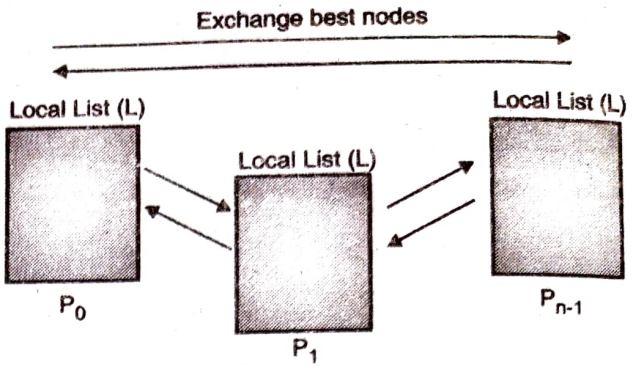


Fig. 5.7.3 : A message-passing implementation of parallel best first search using the ring communication strategy

3. Blackboard communication strategy

- In this strategy nodes are switched among processors through a shared blackboard. First a processor selects a best node from its open list and compares its I-value with the best node on the blackboard.
- The processor will expand that node if it is found that its I-value is within a tolerable limit of the best node on the blackboard.
- The processor sends some of its best nodes to the blackboard before expanding the current node, if it is found that the selected node is better than the best node present on the blackboard.
- On the other hand if it is found that, the selected node is much worse than the best node on the blackboard, some good nodes are selected by the processor from the blackboard and reselects a node for expansion.
- This strategy is based on the value of the best node in the blackboard has to be checked with each node expansion makes it only suitable for the shared space machines.

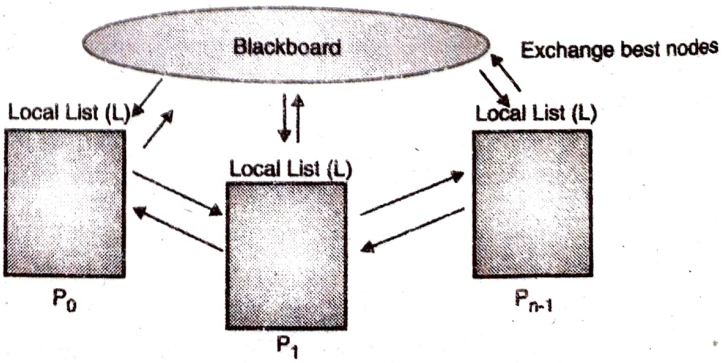


Fig. 5.7.4 : An implementation of parallel best-first search using the blackboard communication strategy

Communication Strategies for Parallel Best-First Graph Search

- An algorithm has to check for node replication while performing searching on graphs. The task for replication checking is distributed among processors.
- The replication checking is implemented using hashing. In this replication checking method each node is maps with a specific processor.
- In later stage, whenever a node is generated, it is mapped with the same processor, which checks for replication locally.
- The hash function used in this technique takes a node as an input and returns a processor label. When a node is generated, it is sent to the processor whose label is returned by the hash function for that node.
- A processor now checks after receiving a node whether it is already present in the local open and closed lists of that processor.
- The node is inserted in the open list if it does not present in both the lists. If the node already present and it is found that new node has better cost associated with it as compare to the existing node, then the previous version of the node is replaced by the new node on the *open* list.