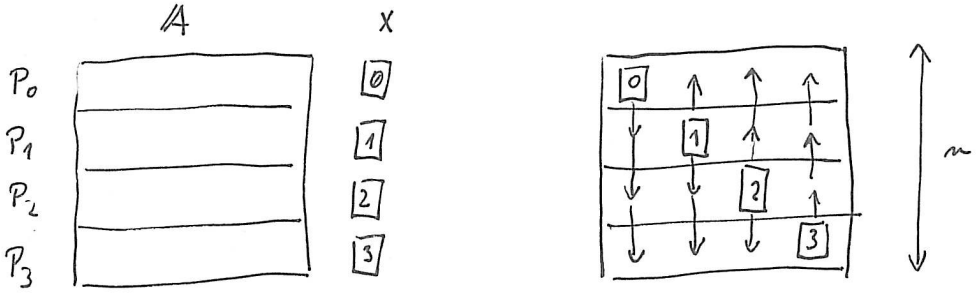
Matrix - vector multiplication $A \cdot x = y$



all-to-all broadcast

broadcast : $t_s \log p + t_w \frac{n}{p}(p-1) \approx t_s \log p + t_w n$

$$T_p = \frac{n^2}{p} + t_s \log p + t_w n$$

$$T_o = \cancel{p n^2} + p t_s \log p + t_w p n - \cancel{n^2}$$

a) $W = K t_w n p$

$n^2 = K t_w p p$

$n^2 = K^2 t_w^2 p^2$

$W = K^2 t_w^2 p^2$

b) $W = K t_s \log p$

The maximum number of processors that can be used cost-optimally

$$p^2 = O(n^2)$$

$$p = O(n) \quad \text{asymptotic upper-bound}$$

Observation:

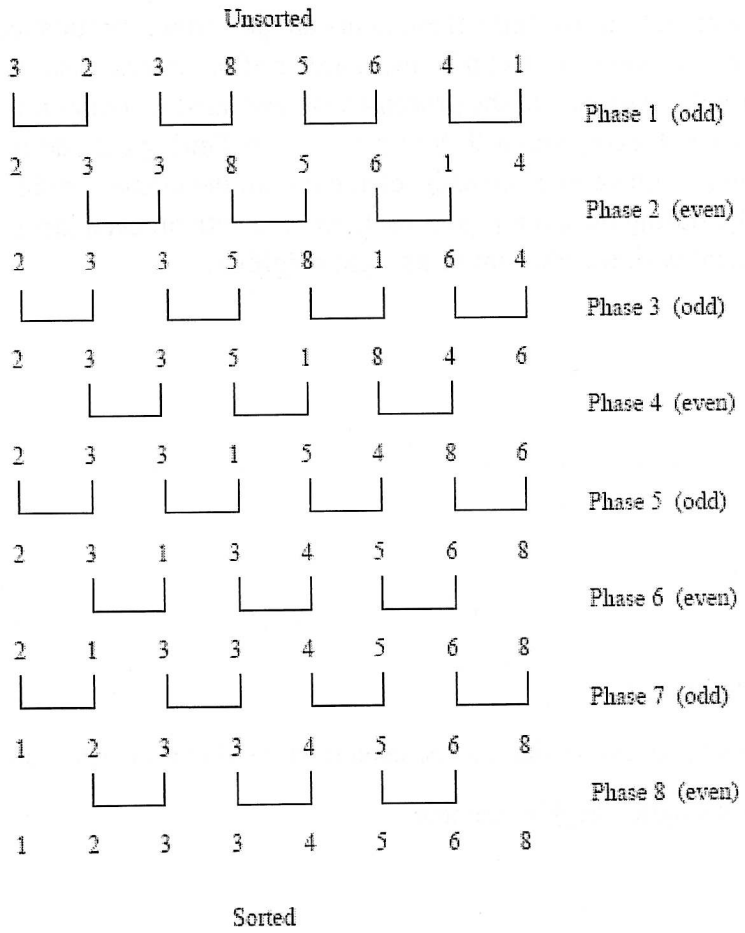
$$S = \frac{T_s}{T_p} = \frac{n^2}{\frac{n^2}{p} + t_s \log p + t_w n}$$

$$E = \frac{S}{p} = \frac{n^2}{n^2 + t_s p \log p + t_w p n} = \left| \begin{matrix} \text{for} \\ p = n \end{matrix} \right| = \frac{n^2}{n^2 + t_s n \log n + t_w n^2}$$

B. Odd-Even Transposition

The odd-even transposition algorithm sorts n elements in n phases (n is even), each of which requires n/2 compare-exchange operations. This algorithm alternates between two phases, called the odd and even phases. Let $\langle a_1, a_2, \ldots, a_n \rangle$ be the sequence to be sorted. During the odd phase, elements with odd indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n) are compare-exchanged (assuming n is even). Similarly, during the even phase, elements with even indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1}) are compare-exchanged. After n phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons, and there are a total of n phases; thus, the sequential complexity is $\Theta(n^2)$. The odd-even transposition sort is shown in Algorithm 9.3 and is illustrated in Figure 9.13.

Figure 9.13. Sorting n = 8 elements, using the odd-even transposition sort algorithm. During each phase, n = 8 elements are compared.



9.3 Sequential odd-even transposition sort algorithm.

```
1.   procedure 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.                   compare-exchange(a_{2j+1}, a_{2j+2});
8.           if i is even then
9.               for j := 1 to n/2 − 1 do
10.                  compare-exchange(a_{2j}, a_{2j+1});
11.      end for
12.  end ODD-EVEN
```

Parallel Formulation

It is easy to parallelize odd-even transposition sort. During each phase of the algorithm, compare-exchange operations on pairs of elements are performed simultaneously. Consider the one-element-per-process case. Let n be the number of processes (also the number of elements to be sorted). Assume that the processes are arranged in a one-dimensional array. Element a_i initially resides on process P_i for i = 1, 2, ..., n. During the odd phase, each process that has an odd label compare-exchanges its element with the element residing on its right neighbor. Similarly, during the even phase, each process with an even label compare-exchanges its element with the element of its right neighbor.

$T_p = \ldots \Theta\left(\frac{n}{p}\log\frac{n}{p} + p\frac{n}{p} + p\frac{n}{p}\right)$

$W = \ldots \Theta(n \log n)$

$T_o = \ldots \Theta(n\log n - n\log p + pn - n\log n) = \Theta(pn)$

Isoefficiency function: $\ldots \Theta(p\,2^p)$

Maximum number of processors that can be used to solve this problem cost-optimally: $\ldots p = O(\log n)$

Scalability: poorly scalable / ~~highly scalable~~

$\Theta(n\log n) = \Theta(pn)$

$\Theta(n) = \Theta(2^p)$

$$S = \frac{n\log n}{\frac{n}{p}\log\frac{n}{p} + n}$$

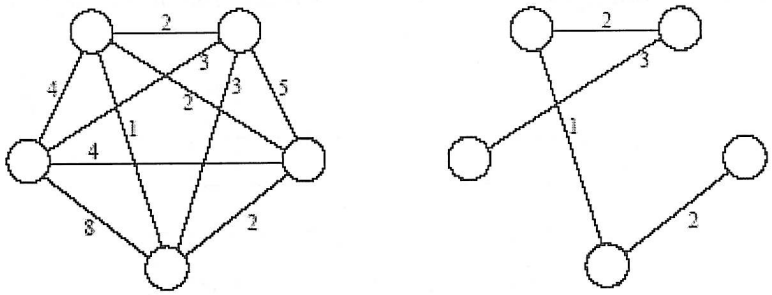
$$E = \frac{n\log n}{n\log\frac{n}{p} + np} \approx \frac{n\log n}{n\log n - n\log\log n + n\log n}$$

$$\boxed{p = \log n}$$

A. Minimum Spanning Tree: Prim's Algorithm

A spanning tree of an undirected graph G is a subgraph of G that is a tree containing all the vertices of G. In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph. A minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight. ~~Many problems require finding an MST of an undirected graph. For example, the minimum length of cable necessary to connect a set of computers in a network can be determined by finding the MST of the undirected graph containing all the possible connections~~. Figure 10.4 shows an MST of an undirected graph.

Figure 10.4. An undirected graph and its minimum spanning tree.



Algorithm 10.1 Prim's sequential minimum spanning tree algorithm.

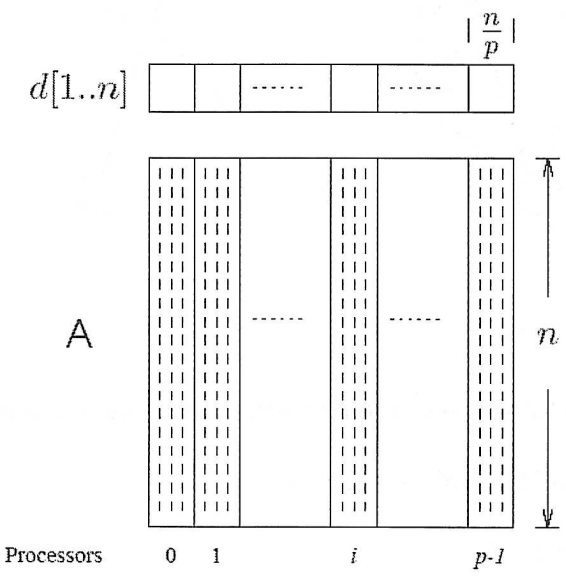
```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};
12.           for all v ∈ (V − V_T) do
13.               d[v] := min{d[v], w(u, v)};
14.       endwhile
15.   end PRIM_MST
```

Parallel Formulation

Prim's algorithm is iterative (see Algorithm 10.1). Each iteration adds a new vertex to the minimum spanning tree. Since the value of d[v] for a vertex v may change every time a new vertex u is added in V_T, it is hard to select more than one vertex to include in the minimum spanning tree. However, each iteration can be performed in parallel as follows.

Let p be the number of processes, and let n be the number of vertices in the graph. The set V is partitioned into p subsets using the 1-D block mapping (Section 3.4.1). Each subset has n/p consecutive vertices, and the work associated with each subset is assigned to a different process. Let V_i be the subset of vertices assigned to process P_i for $i = 0, 1, ..., p - 1$. Each process P_i stores the part of the array d that corresponds to V_i (that is, process P_i stores d [v] such that $v \in V_i$). Figure 10.6(a) illustrates the partitioning. Each process P_i computes $d_i[u] = \min\{d_i[v] | v \in (V - V_T) \cap V_i\}$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by using the all-to-one reduction operation (Section 4.1) and is stored in process P_0. Process P_0 now holds the new vertex u, which will be inserted into V_T. Process P_0 broadcasts u to all processes by using one-to-all broadcast (Section 4.1). The process P_i responsible for vertex u marks u as belonging to set V_T. Finally, each process updates the values of d[v] for its local vertices.

Figure 10.6. The partitioning of the distance array d and the adjacency matrix A among p processes.



$$S = \frac{n^2}{\frac{n^2}{p} + n \log p}$$

$$E = \frac{n^2}{n^2 + n p \log p} = \frac{n^2}{n^2 + n \frac{n}{\log n} \cdot \log n}$$

$$\left(p = \frac{n}{\log n}, \log p \approx \log n \right)$$

When a new vertex u is inserted into V_T, the values of d[v] for $v \in (V - V_T)$ must be updated. The process responsible for v must know the weight of the edge (u, v). Hence, each process P_i needs to store the columns of the weighted adjacency matrix corresponding to set V_i of vertices assigned to it. This corresponds to 1-D block mapping of the matrix (Section 3.4.1). The space to store the required part of the adjacency matrix at each process is $\Theta(n^2/p)$. Figure 10.6(b) illustrates the partitioning of the weighted adjacency matrix.

$T_p = \Theta\left(n \left(\frac{n}{p}\right) + n \log p\right)$

$W = \Theta(n^2)$

$T_o = \Theta(n p \log p)$

$\Theta(n^2) = \Theta(n p \log p)$

$\Theta(n) = \Theta(p \log p)$

$\Theta(n^2) = \Theta(p^2 \log^2 p)$

Isoefficiency function: $\Theta(p^2 \log^2 p)$

Maximum number of processors that can be used to solve this problem cost-optimally: ...

Scalability: ~~poorly scalable~~ / highly scalable

$\Theta(n) = \Theta(p \log p)$

$\Theta(\log n) = \Theta(\log p + \log \log p)$

$\Theta(\log n) \approx \Theta(\log p)$

$\Theta(n) = \Theta(p \log p)$

$\Theta\left(\frac{n}{\log p}\right) = \Theta(p) \Rightarrow p = O\left(\frac{n}{\log n}\right)$

C. Matrix-Matrix Multiplication - A Simple Parallel Algorithm

Consider two n x n matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ $0 \leq k < \sqrt{k}$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. These blocks are mapped onto a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes. The processes are labeled from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix. Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $(0 \leq i, j < \sqrt{p})$. To acquire all the required blocks, an all-to-all broadcast of matrix A's blocks is performed in each row of processes, and an all-to-all broadcast of matrix B's blocks is performed in each column. After $P_{i,j}$ acquires $A_{i,0}, \ldots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, \ldots, A_{\sqrt{p}-1,j}$, it performs the submatrix multiplication and addition step of lines 7 and 8 in Algorithm 8.3.

Performance and Scalability Analysis The algorithm requires two all-to-all broadcast steps (each consisting of $\sqrt{p}$ concurrent broadcasts in all rows and columns of the process mesh) among groups of $\sqrt{p}$ processes. The messages consist of submatrices of n^2/p elements. From Table 4.1, the total communication time is $2(t_s \log(\sqrt{p}) + t_w (n^2/p)(\sqrt{p}-1))$. After the communication step, each process computes a submatrix $C_{i,j}$, which requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices (lines 7 and 8 of Algorithm 8.3 with $q = \sqrt{p}$.

Algorithm 8.3 The block matrix multiplication algorithm for n x n matrices with a block size of (n/q) x (n/q).

```
1.   procedure BLOCK_MAT_MULT (A, B, C)
2.   begin
3.      for i := 0 to q - 1 do
4.          for j := 0 to q - 1 do
5.              begin
6.                 Initialize all elements of Ci,j to zero;
7.                 for k := 0 to q - 1 do
8.                     Ci,j := Ci,j + Ai,k x Bk,j;
9.              endfor;
10. end BLOCK_MAT_MULT
```

$$S = \frac{n^3}{\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} + \log p}$$

$$E = \frac{n^3}{n^3 + n^2\sqrt{p} + p \log p}$$

$T_p = \ldots \Theta\left(\log p + \frac{n^2}{\sqrt{p}} - \frac{n^2}{p} + \frac{n^3}{p}\right) = \Theta\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} + \log p\right)$

$W = \ldots \Theta(n^3)$

$T_o = \ldots \Theta(n^3 + n^2\sqrt{p} + p \log p - n^3)$

Isoefficiency function: $\ldots \Theta(p^{\frac{3}{2}})$

Maximum number of processors that can be used to solve this problem cost-optimally: $\ldots p = O(n^2)$

Scalability: ~~poorly scalable~~ / highly scalable

a) $\Theta(n^3) = \Theta(n^2\sqrt{p})$

$\Theta(n^3) = \Theta(p^{\frac{3}{2}})$

b) $\Theta(n^3) = \Theta(p \log p)$

$\Theta(n^2) = \Theta(p)$