Different Parallel Algorithms

Mahmut Taylan Kandemir

CSE 531

Spring 2023

ACK: A. Grama

Matix Algorithms: Introduction

- Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data-decomposition.
- Typical algorithms rely on input, output, or intermediate data decomposition.
- Most algorithms use one- and two-dimensional block, cyclic, and block-cyclic partitionings.

Matrix-Vector Multiplication

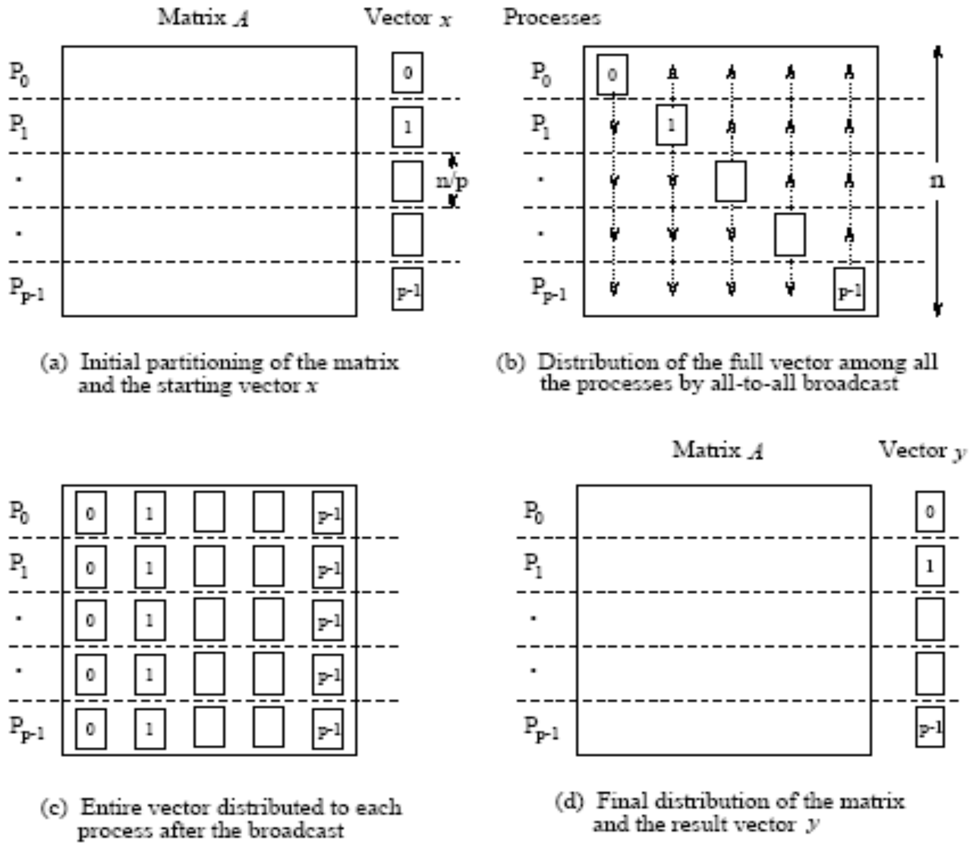
- We aim to multiply a dense n x n matrix A with an n x 1 vector x to yield the n x 1 result vector y.
- The serial algorithm requires n^2 multiplications and additions.

$$W = n^2.$$

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- The n x n matrix is partitioned among n processors, with each processor storing complete row of the matrix.
- The n x 1 vector x is distributed such that each process owns one of its elements.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning


Matrix A
Vector x
Processes
P0
P1
Pp-1
0
1
p-1
n/p
n
(a) Initial partitioning of the matrix and the starting vector x
(b) Distribution of the full vector among all the processes by all-to-all broadcast
(c) Entire vector distributed to each process after the broadcast
Matrix A
Vector y
(d) Final distribution of the matrix and the result vector y

Multiplication of an n x n matrix with an n x 1 vector using rowwise block 1-D partitioning. For the one-row-per-process case, $p = n$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Since each process starts with only one element of x , an all-to-all broadcast is required to distribute all the elements to all the processes.
- Process P_i now computes $y[i] = \Sigma_{j=0}^{n-1}(A[i,j] \times x[j])$.
- The all-to-all broadcast and the computation of $y[i]$ both take time $\Theta(n)$. Therefore, the parallel time is $\Theta(n)$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Consider now the case when $p < n$ and we use block 1D partitioning.
- Each process initially stores $n=p$ complete rows of the matrix and a portion of the vector of size $n=p$.
- The all-to-all broadcast takes place among p processes and involves messages of size $n=p$.
- This is followed by $n=p$ local dot products.
- Thus, the parallel run time of this procedure is

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

This is cost-optimal.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

Scalability Analysis:

- We know that $T_0 = pT_P - W$, therefore, we have,

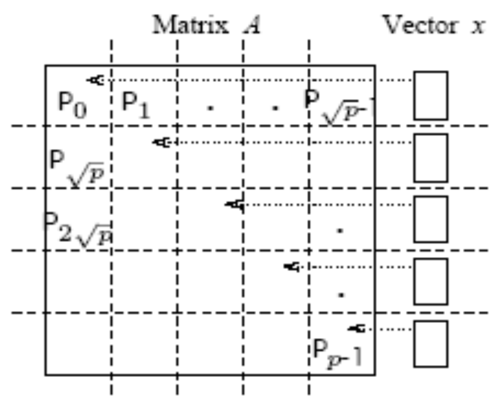
$$T_o = t_s p \log p + t_w np.$$

- For isoefficiency, we have $W = KT_0$, where $K = E/(1 - E)$ for desired efficiency E.
- From this, we have $W = O(p^2)$ (from the t_w term).
- There is also a bound on isoefficiency because of concurrency. In this case, $p < n$, therefore, $W = n^2 = \Omega(p^2)$.
- Overall isoefficiency is $W = O(p^2)$.

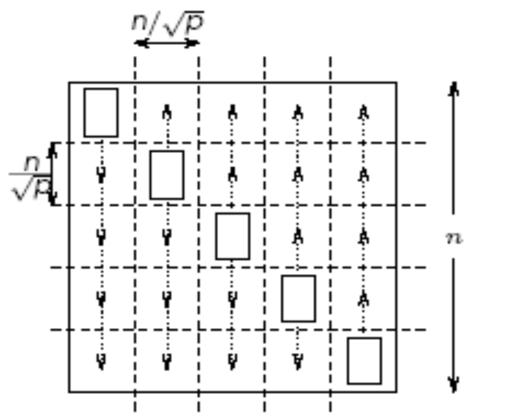
Matrix-Vector Multiplication: 2-D Partitioning

- The n x n matrix is partitioned among n^2 processors such that each processor owns a single element.
- The n x 1 vector x is distributed only in the last column of n processors.

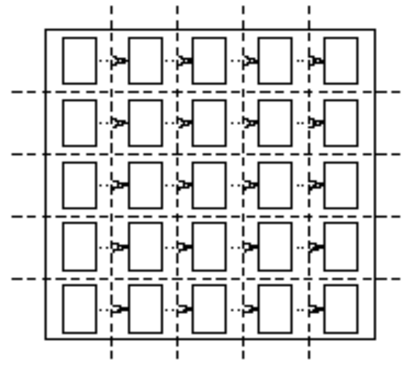
Matrix-Vector Multiplication: 2-D Partitioning



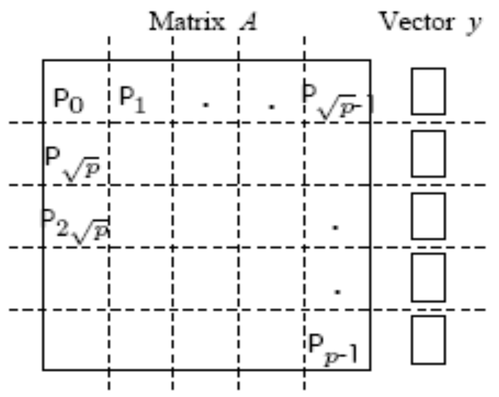
(a) Initial data distribution and communication steps to align the vector along the diagonal



(b) One-to-all broadcast of portions of the vector along process columns



(c) All-to-one reduction of partial results



(d) Final distribution of the result vector

Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, $p = n^2$ if the matrix size is n x n .

Matrix-Vector Multiplication: 2-D Partitioning

- We must first align the vector with the matrix appropriately.
- The first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix.
- The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using n simultaneous broadcasts among all processors in the column.
- Finally, the result vector is computed by performing an all-to-one reduction along the columns.

Matrix-Vector Multiplication: 2-D Partitioning

- Three basic communication operations are used in this algorithm: one-to-one communication to align the vector along the main diagonal, one-to-all broadcast of each vector element among the n processes of each column, and all-to-one reduction in each row.
- Each of these operations takes $\Theta(\log n)$ time and the parallel time is $\Theta(\log n)$.
- The cost (process-time product) is $\Theta(n^2 \log n)$; hence, the algorithm is not cost-optimal.

Matrix-Vector Multiplication: 2-D Partitioning

- When using fewer than n^2 processors, each process owns a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix.
- The vector is distributed in portions of $n/\sqrt{p}$ elements in the last process-column only.
- In this case, the message sizes for the alignment, broadcast, and reduction are all $n/\sqrt{p}$.
- The computation is a product of an $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrix with a vector of length $n/\sqrt{p}$.

Matrix-Vector Multiplication: 2-D Partitioning

- The first alignment step takes time

$$t_s + t_w n/\sqrt{p}$$

- The broadcast and reductions take time

$$(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$$

- Local matrix-vector products take time

$$t_c n^2/p$$

- Total time is

$$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$$

Matrix-Vector Multiplication: 2-D Partitioning

- Scalability Analysis:

- $T_o = pT_p - W = t_s p \log p + t_w n \sqrt{p} \log p$
- Equating T_0 with W, term by term, for isoefficiency, we have, $W = K^2 t_w^2 p \log^2 p$ as the dominant term.
- The isoefficiency due to concurrency is $O(p)$.
- The overall isoefficiency is $O(p \log^2 p)$ (due to the network bandwidth).
- For cost optimality, we have, $W = n^2 = p \log^2 p$. For this, we have, $p = O\left(\frac{n^2}{\log^2 n}\right)$

Matrix-Matrix Multiplication

- Consider the problem of multiplying two n x n dense, square matrices A and B to yield the product matrix $C = A$ x B.
- The serial complexity is $O(n^3)$.
- We do not consider better serial algorithms (Strassen's method), although, these can be used as serial kernels in the parallel algorithms.
- A useful concept in this case is called *block* operations. In this view, an n x n matrix A can be regarded as a q x q array of blocks $A_{i,j}$ ($0 \leq i, j < q$) such that each block is an (n/q) x (n/q) submatrix.
- In this view, we perform q^3 matrix multiplications, each involving (n/q) x (n/q) matrices.

Matrix-Matrix Multiplication

- Consider two n x n matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ ($0 \leq i, j < \sqrt{p}$) of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- 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 $0 \leq k < \sqrt{p}$.
- All-to-all broadcast blocks of A along rows and B along columns.
- Perform local submatrix multiplication.

Matrix-Matrix Multiplication

- The two broadcasts take time
$$2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p} - 1))$$
- The computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices.
- The parallel run time is approximately
$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}.$$
- The algorithm is cost optimal and the isoefficiency is $O(p^{1.5})$ due to bandwidth term t_w and concurrency.
- Major drawback of the algorithm is that it is not memory optimal.

Other Parallel Matrix-Matrix Multiplication Algorithms

- Cannon’s algorithm
- DNS algorithm

- Solving Systems of Equations in Parallel
  - Can we parallelize Gaussian Elimination? How?

Sorting: Overview

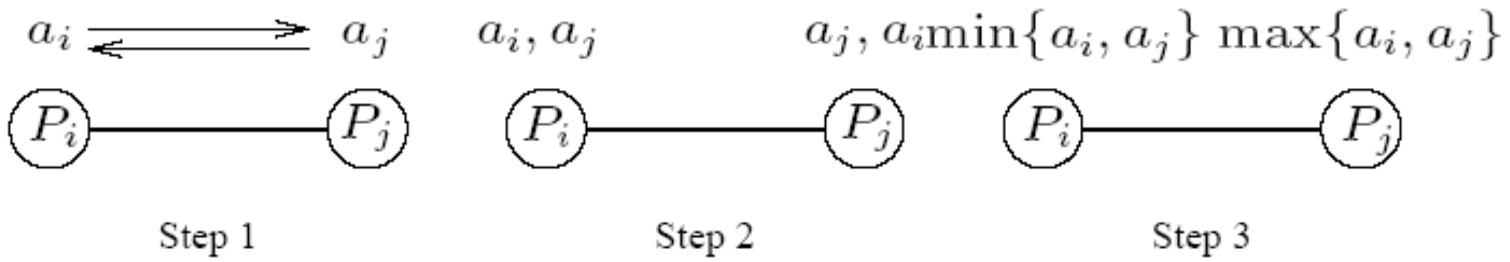
- One of the most commonly-used and well-studied kernels.
- Sorting can be *comparison-based* or *noncomparison-based*.
- The fundamental operation of comparison-based sorting is *compare-exchange*.
- The lower bound on any comparison-based sort of *n* numbers is $\Theta(n\log n)$.
- We focus here on comparison-based sorting algorithms.

Sorting: Basics

What is a parallel sorted sequence? Where are the input and output lists stored?

- We assume that the input and output lists are distributed.
- The sorted list is partitioned with the property that each partitioned list is sorted and each element in processor P_i's list is less than that in P_j's list if $i < j$.

Sorting: Parallel Compare-Exchange Operation



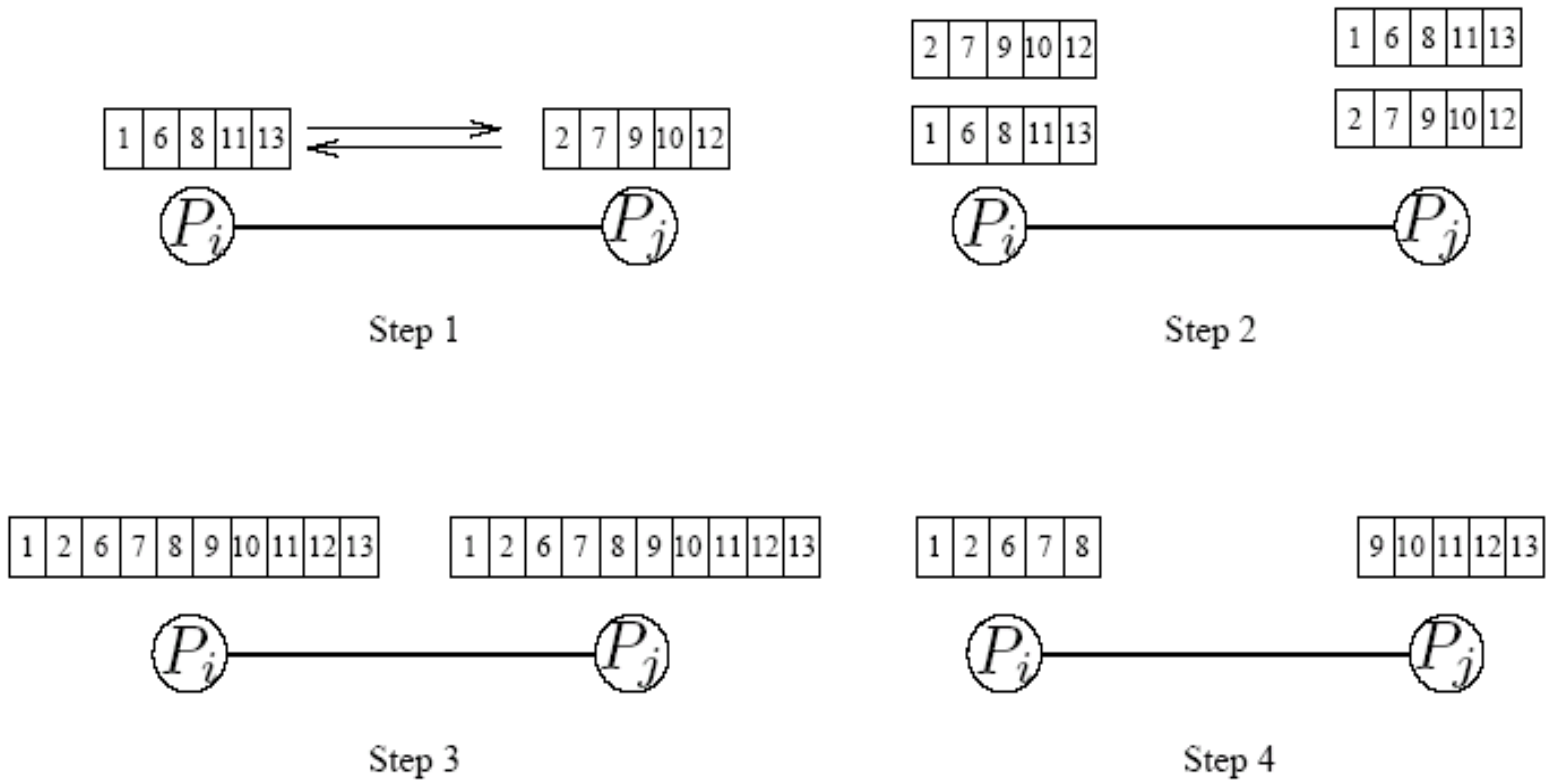
A parallel **compare-exchange** operation. Processes P_i and P_j send their elements to each other. Process P_i keeps $\min\{a_i, a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.

Sorting: Basics

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.

Sorting: Parallel Compare Split Operation



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 this example, process P_i retains the smaller elements and process P_i retains the larger elements.

Bubble Sort and its Variants

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

```
procedure BUBBLE_SORT($n$)
begin
    for $i := n - 1$ downto $1$ do
        for $j := 1$ to $i$ do
            compare-exchange($a_j, a_{j+1}$);
end BUBBLE_SORT
```

Sequential bubble sort algorithm.

Bubble Sort and its Variants

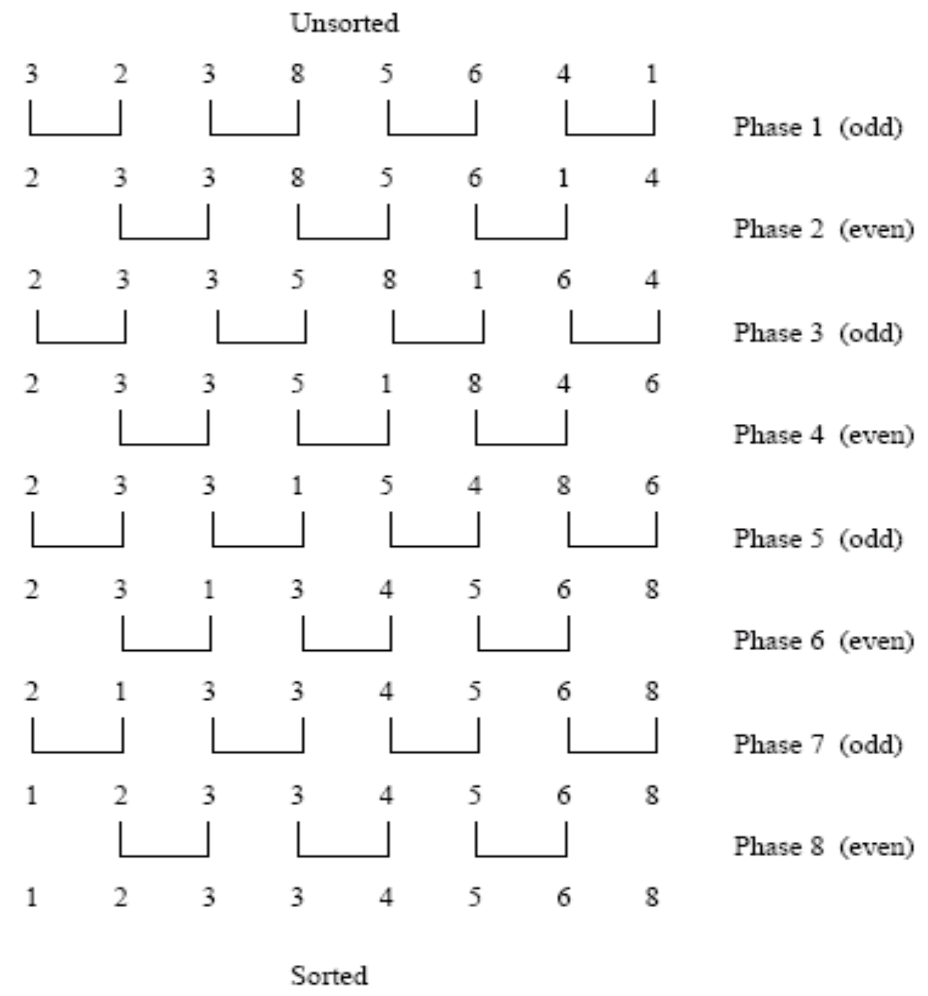
- The complexity of bubble sort is $\Theta(n^2)$.
- Bubble sort is *difficult* to parallelize since the original algorithm has *no* concurrency.
- A simple variant, though, uncovers the concurrency!

Sequential Odd-Even Transposition

```
procedure ODD-EVEN($n$)
begin
    for $i := 1$ to $n$ do
    begin
        if $i$ is odd then
            for $j := 0$ to $n/2 - 1$ do
                compare-exchange($a_{2j+1}, a_{2j+2}$);
        if $i$ is even then
            for $j := 1$ to $n/2 - 1$ do
                compare-exchange($a_{2j}, a_{2j+1}$);
    end for
end ODD-EVEN
```

Sequential odd-even transposition sort algorithm.

Sequential Odd-Even Transposition



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

Sequential 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.
- Thus, the serial (sequential) 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 only 1 compare-exchange.
- Hence, 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

```
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($id - 1$);
        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 with p<n

- 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

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

Parallel Odd-Even Transposition

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

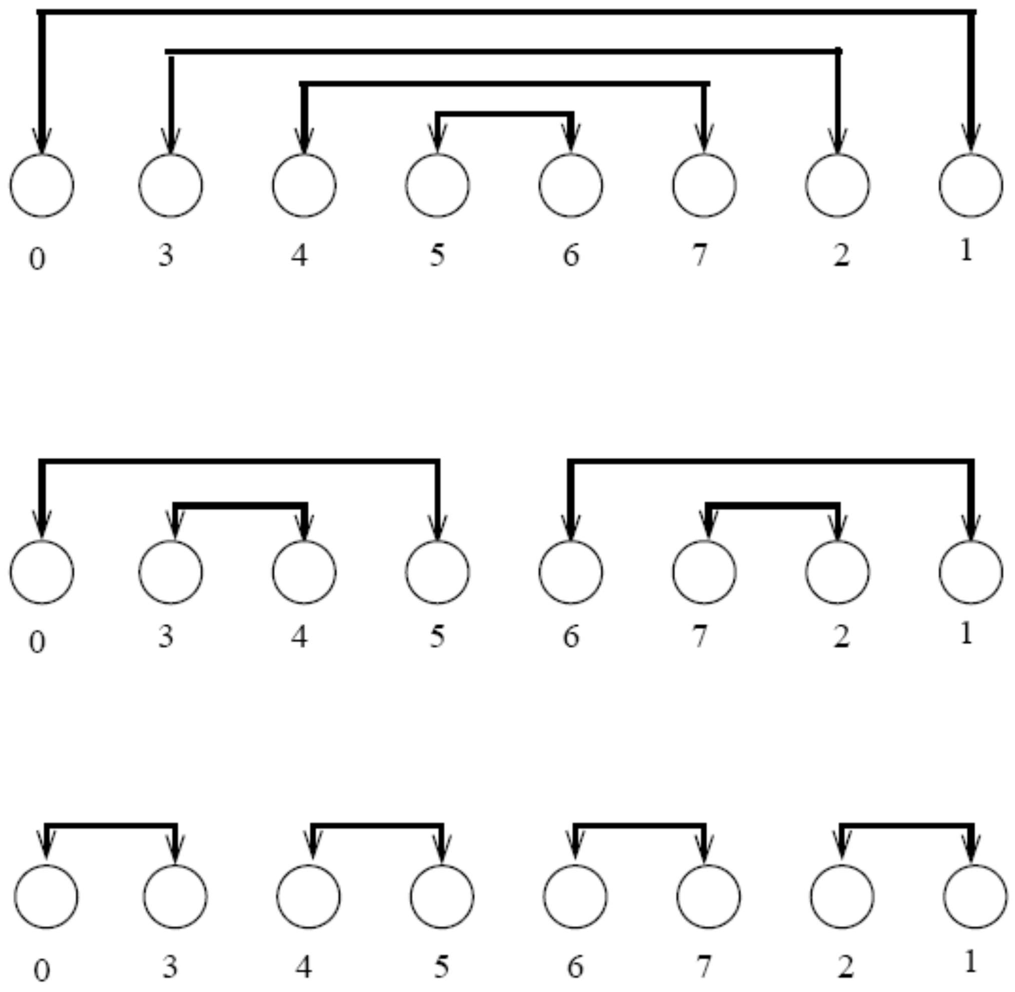
Shellsort

- 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 Shellsort

- 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.

Parallel Shellsort



An example of the first phase of parallel shellsort on an eight-process array.

Parallel Shellsort

- 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:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right)}^{\text{first phase}} + \overbrace{\Theta\left(l\frac{n}{p}\right)}^{\text{second phase}}.$$

Other Sorting Algorithms

- Can Quicksort be parallelized? How?
- Can Bucket Sort be parallelized? How?

Tips for the Final Exam

- Go over the slides very carefully
- Send me/Scott email about things that are not clear
- Read the research papers in Canvas
- Solve the practice questions (by yourself)