Introduction To Parallel Algorithms & Architectures
Problem Solution with – Serial approach

For *serial* computation:
- Run on a single computer having a single CPU;
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.
Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs
Parallel approach provides promising research results.
More.... on the use of parallel

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory. For example:

- "Grand Challenge" (en.wikipedia.org/wiki/Grand_Challenge) problems requiring PetaFLOPS and PetaBytes of computing resources.
- Web search engines/databases processing millions of transactions per second.
More...on the use of parallel

Provide concurrency: A single compute resource can only do one operation at a time. Multiple computing resources can be doing many operations simultaneously. For example, the Access Grid (www.accessgrid.org) provides a global collaboration network where people from around the world can meet and conduct work 'virtually'.

Use of non-local resources: Using compute resources on a wide area network, or even the Internet when local compute resources are scarce. For example:

- SETI@home (setiathome.berkeley.edu) uses over 330,000 computers for a compute power over 528 TeraFLOPS (as of August 04, 2008)
- Folding@home (folding stanford edu) uses over 340,000 computers for a compute power of 4.2 PetaFLOPS (as of November 4, 2008)
More..on the use of parallel

**Limits to serial computing:** Both physical and practical reasons pose significant constraints to simply building ever faster serial computers:

- Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
5.1 Approaches to the Design of Parallel Algorithms

Three major approaches to design parallel algorithms:

- Modify an existing sequential algorithm.
- Design a new parallel algorithm.
- Run a sequential algorithm on different processors with different inputs.
5.2 Architecture Constraints
When Designing a Parallel Algorithm

A parallel algorithm comes with constraints imposed by the architecture of the particular computer.

The major constraints are:

1. Single instruction/Multiple instruction and Single data/Multiple data
2. The number and type of processors
3. Shared memory/Distributed memory
4. Communication constraints
5. I/O constraints
Single Instruction/Data vs Multiple Instruction/Data

A computer with n processors can execute n instructions simultaneously, each processor operating on possibly same or different data.

We have four combinations:

- **SISD** → single processor computer
- **SIMD** → Same instruction can be performed on each processor at one cycle (will be used mostly)
- **MIMD** → different instructions to be performed at the same time on different data
- **MISD** → different instructions to be performed at the same time on the same data
SISD

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and even today, the most common type of PC
- Examples: older generation mainframes, minicomputers and workstations;
**SIMD: A type of parallel computer**

- **Single instruct:** All processing units execute the same instruction at any given clock cycle.
- **Multiple data:** Each processing unit can operate on a different data element.
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- **Synchronous and deterministic execution.**
- **Two varieties:** Processor arrays and vector pipelines.
- **Examples:**
  - Processor Arrays: Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.
MISD

- A single data stream is fed into multiple processing units.
- Each process unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.
MIMD

- Currently, the most common type of parallel computer.
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components
The Number and Type of Processors

Approaches to design a parallel algorithm with respect to the number of processors:

First Approach
- The number of processors is fixed
- and is an input to the algorithm.

Second Approach
- The number of processors grows with input size n
- Thus, is a function $P(n)$ of the input size n.
Granularity - fine-or-coarse grain, which is better?

- **Computation / Communication Ratio**: In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- Periods of computation are separated from periods of communication by synchronization events.

- **Fine-grain Parallelism**:
  - Relatively small amounts of computational work are done between communication events
  - Low computation to communication ratio
  - Facilitates load balancing
  - Implies high communication overhead and less opportunity for performance enhancement
  - If granularity is too fine, the overhead required for communications and synchronization between tasks takes longer than the computation.
Granularity - fine-or-coarse grain, which is better?

- **Coarse-grain Parallelism:**
  - Relatively large amounts of computational work are done between communication/synchronization events.
  - High computation to communication ratio.
  - Implies more opportunity for performance improvement.
  - Harder to load balance efficiently.

Which one is better.. Fine grain or coarse grain?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.
Load Balancing for parallel design

Load balancing refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.

Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance. (Just like the critical path analysis!!!)
How to Achieve Load Balance:

1. **Equally partition the work each task receives**
   - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
   - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
   - If a heterogeneous mix of machines with varying performance characteristics are used, make sure to use performance analysis tool to detect any load imbalances. (Adjust work accordingly).
2. Use dynamic work assignment

- Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
  - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
  - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
  - $N$-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.

- When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler - task pool approach. That is, as each task finishes its work, it queues to get a new piece of work.

- It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.
Importance of Communication in parallel design

- Most parallel applications are not quite so simple, and do require tasks to share data with each other...

- There are a number of important factors to consider when designing your program's inter-task communications..

- See next slide for these factors:
Communications

1. Cost of communications

- Inter-task communication virtually always implies overhead.
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.
2. Latency vs. Bandwidth

- **latency** is the time it takes to send a minimal message from point A to point B. (e.g. 10 microsec)

- **bandwidth** is the amount of data that can be communicated per unit of time (e.g. 50 Mbps).

Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.
3. Visibility of communications

- With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.

- With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even know exactly how inter-task communications are accomplished.
Communications

4. Synchronous vs. asynchronous communications

- **Synchronous communication** requires some type of ‘handshaking’ between tasks that are sharing data.

- Synchronous communications are often referred to as ‘blocking’ communications since other work must wait until the communications have completed.

- **Asynchronous communications** allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.

- Asynchronous communications are often referred to as ‘non-blocking’ communications since other work can be done while the communications are taking place.

- Interleaving computation with communication is the single greatest benefit for using asynchronous communications.
Communications

5. Efficiency of communications

- Very often, the programmer will have a choice with regard to factors that can affect communications performance. Only a few are mentioned here.

- Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.

- What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.

- Network media - some platforms may offer more than one network for communications.
6. Overhead and Complexity

it should not be too complicated to say helloworld!!!
Automatic vs. Manual parallelization

Manually developing parallel codes is a time consuming, complex, error-prone and *iterative* process...

Recently, various tools (parallelizing compiler or pre-processor) have been available to assist the programmer with converting serial programs into parallel programs.
A parallelizing compiler works in two different ways:

- **Fully Automatic**
  - The compiler analyzes the source code and identifies opportunities for parallelism.
  - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
  - Loops (do, for) loops are the most frequent target for automatic parallelization.

- **Programmer Directed**
  - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
  - May be able to be used in conjunction with some degree of automatic parallelization also.
Possible problems with parallelizing compilers:

- Wrong results may be produced
- Performance may actually degrade
- Much less flexible than manual parallelization
- Limited to a subset (mostly loops) of code
- May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex

We will focus on manual method of developing parallel codes!!!
Memory Architectures: 1) shared  2) distributed

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.

- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer’s responsibility.
Shared Memory (PRAMs)

Shared Memory is used to estimate the algorithm performance (like its time complexity).

In a shared memory model all processors have equal access to every memory location.

This causes conflicts when two or more processors attempt to access the same memory location.

There are four models to deal with these conflicts.
Four Models (a)

i. **EREW** (Exclusive Read Exclusive Write): only one processor may read or write into a memory location.

ii. **CREW** (Concurrent Read Exclusive Write): only one processor may write to a memory location, multiple processors may read a memory location.

iii. **ERCW** (Exclusive Read Concurrent Write)

iv. **CRCW** (Concurrent Read Concurrent Write)
The concurrent read causes no discrepancies while the concurrent write is further defined as:

- **Common**—all processors write the same value; otherwise is illegal
- **Arbitrary**—only one arbitrary attempt is successful, others retire
- **Priority**—processor rank indicates who gets to write
- **Other**—Another kind of array Reduction operation like SUM, Logical AND or MAX.
Additional Assumptions on Shared Memory PRAMs

- There is no limit on the number of processors in the machine.
- Any memory location is uniformly accessible from any processor.
- There is no limit on the amount of shared memory in the system.
- Resource contention is absent.
- The programs written on these machines are, in general, of type MIMD.

eg. The SB-PRAM developed by Staarland University is a MIMD parallel computer with shared address space and uniform memory access time (CRCW-PRAM-Model)
Searching on a PRAM

Assume a search element X is in the list L[1:n]

Use PRAM computer with n processors P₁,P₂,…,Pₙ

If L[i] = X, then Pᵢ writes i to a memory location.

To prevent conflicts in CW, we may choose the processor having smallest index to be the winner.

Now suppose we use EREW PRAM

a) A temporary array Temp[1:n] and assign the value of X to each array element.

b) All Pᵢ, 1 ≤ i ≤ n, compares L[i] with Temp[i]. Then writes the result of the comparison in Temp[i]. If L[i] = Temp[i] write i, otherwise ∞

c) Result is obtained by reverse broadcasting. To find the minimum entry in the array Temp[1:n], we use a method called Binary Fan-in
a) Broadcasting the value X
b) Compare $L[i]$ with $\text{Temp}[i]$
c) Reverse Broadcasting

Divide the list into pairs and search for the minimum.
Distributed Memory (INTERCONNECTION NETWORKS)
INTRODUCTION

- Dedicated memory for each processor
- No shared memory
- Distributed (parallel) variables
  - Instantiation in every processor
  - P:X denotes the instantiation of X in processor P’s memory
- Information is communicated using messages
INTRODUCTION

Two processors can communicate only if they are adjacent (directly connected)

Communication scenario

Local memory $P_i$

$Y$

$P_i$

Local memory $P_j$

$X$

$P_j$

$P_i:Y \leftrightarrow P_i:X$
Types of Communication Networks

Meshes

- **1D Mesh:** $M_p$
  - $P_i$ and $P_j$ are adjacent iff $|i-j|=1$

- **2D Mesh:** $M_{q,q}$ where $q^2=p$
  - $P_{i,j}$ and $P_{r,s}$ are adjacent iff either $j=s$ and $|i-r|=1$ or $i=r$ and $|j-s|=1$
Complete binary tree: $PT_p$ where $p=2^{k-1}$

- Root is denoted by $\varepsilon$
- Each left movement appends a 0
- Each right movement appends a 1
TYPES OF COMMUNICATION NETWORKS

- **Complete graph:** $K_p$
  - Every processor is directly connected to every other one.

- Other networks include hypergraphs, butterfly networks, shuffle-exchange graphs, etc.
GOODNESS MEASURES - DIAMETER

Diameter(M) = \( \max \{\text{dist}(P_i, P_j): P_i, P_j \text{ are in } M \} \)
and \( \text{dist}(P_i, P_j) \) is the length of shortest route connecting \( P_i \) and \( P_j \).

Diameter is a lower bound for communication complexity.

Although small diameter is good, it is not feasible to build technologically.

\( \text{Dia}(M_p) = p - 1 \) where \( P_i = P_1 \) and \( P_j = P_p \)
DIAMETER

\[ \text{Dia}(M_{q,q}) = 2q - 2 \]
where \( P_i = P_{1,1} \)
and \( P_j = P_{q,q} \)

\[ \text{Dia} (PT_p) = 2 \left\lfloor \log_2 p \right\rfloor \]

\[ \text{Dia} (K_p) = 1 \]
GOODNESS MEASURES – MAXIMUM DEGREE

Max-degree(M) = max{degree(P): P is in M}
where degree of a processor is the number of links incident with P

Max-degree(M_p) = 2
Max-degree(M_q,q) = 4
MAXIMUM DEGREE

- Max-degree($PT_p$) = 3
- Max-degree($K_p$) = $p-1$
GOODNESS MEASURES – BISECTION WIDTH

Bisection-width(M) = \( \min \{|C(X,Y)|: \text{abs}(|X|-|Y|\leq 1)\} \)

- X and Y are partitions of M with almost equal size and \( X+Y=M \)
- \( C(X,Y) \) is the number of links joining a processor in X to a processor in Y

Small bisection-width implies congestion during communication.
Bisection-Width

Bisection-width($M_p$) = 1

Bisection-width($M_{q,q}$) = $q + (q^2 \mod 2)$
Bisection-Width

Bisection-width($PT_p$) = 1

Bisection-width($K_p$) = $p/2 \times p/2 = p^2/4$
Assume search element $X$ is stored initially on $P_1$.

Broadcasting $X$ requires $O(n)$ time due to communication constraints.

Broadcasting procedure used for EREW PRAM cannot be used since it involves communication between nonadjacent processors.
SEARCHING ON A 2D MESH

- Assume list size $n$ is equal to $q^2$
- Initially $X$ is stored in $P_{1,1}$
- List is stored in mesh in row major order.
  - $P_{i,j}$ stores $L[k]$ where $k=(i-1)q+j$
SEARCHING ON A 2D MESH

1. Broadcast $P_{1,1}:X$ to all processors
   1. $P_{1,i}$ sends $X$ to $P_{1,i+1}$, $1 \leq i \leq q-1$
   2. $P_{i,j}$ send $X$ to $P_{i+1,j}$, $1 \leq j \leq q-1$

2. Each processor compares $P_{ij}:X$ with $P_{ij}:L$ and if they are not equal writes infinity (Inf) to $P_{ij}:Index$

3. Reverse broadcast minimum index value
   1. Compute column minimums and write to first row
      Find min($P_{ij}:Index, P_{i-1,j}:Index$), $1 \leq i \leq q-1$
   2. Find minimum of row 1
      Find min($P_{1,i}:Index, P_{1,i+1}:Index$), $1 \leq i \leq q-1$

Communication Complexity: $\Theta(n) = 4q - 4 = \Theta(\sqrt{n})$

Speedup: $S(n) = n/(4\sqrt{n} - 4)$
The list $L[1:n]$, $n=2^{k-1}$ is stored in leaves.

Processor $P_a$ stores $L[i]$ where $a$ is equal to $k-1$ digit representation of $i-1$.

Search key $X$ is initially on $P_\epsilon$ (root).

All index values are initialized to infinity.
1. Starting from the root, broadcast X to child processors until all leaf processors get X. 2 steps for each processor to broadcast to its children.

2. Compare P:X with P:L and write index value to P:Index if they are equal, otherwise leave P:Index as infinity.

3. Use binary fan-in technique to calculate minimum index value at root.

Communication Complexity: $\Theta(3 \log_2 n + 1) = \Theta(\log_2 n)$

Speedup: $S(n) = n/(\log_2 n)$
SEARCHING ON PT<sub>p</sub>

Total steps: 3log<sub>n</sub>+1=10
Chapter
5.5.5, 5.6, 5.7, 5.8
5.5.5 Example: Pseudocode for searching on the Two-Dimensional Mesh

```
procedure Broadcast2DMesh ( P_{1,1} : X, n)

Model: two-dimensional mesh M_{q,q}, with p=n=q^2 processors

Input: P_{1,1} : X (element to be broadcast)

Output: P_{1,1} : X is broadcast to each processor in M_{q,q}

for i:= 1 to q-1 do
    P_{i,j+1} : X ← P_{1,j} : X \{propagate X to the right across first row\}
endfor

for i:= 1 to q-1 do
    for P_{i,j} \ 1 ≤ j ≤ q \ do in parallel
        P_{i+1,j} : X ← P_{i,j} : X \{propagate X down row by row\}
    end in parallel
end for

end Broadcast2DMesh
```
5.5.5 Example: Pseudocode for searching on the Two-Dimensional Mesh

```plaintext
function Min2DMesh ( X, n )

Model: two-dimensional mesh $M_{q,q}$, with $p=n=q^2$ processors

Input: $X$ (a list of $n$ real numbers $x_1, x_2, ..., x_n$) range: $P_{i,j}$, $1 \leq i,j \leq q$

Output: $\min \{x_1, x_2, ..., x_n\}$

for $Row := q-1$ downto 1 do  
  for $P_{i,j}$, $i=Row$ .and. $1 \leq j \leq q$ do in parallel
    { compute column minimums }
    $P_{i,j} : Temp \leftarrow P_{i+1,j} : X$  
    { communicate up from $X$ to $Temp$ }
    $X := \min \{X, Temp\}$  
    { compute min of $P_{i,j} : X$ and $P_{i,j} : Temp$ }
  end in parallel
endfor

for $Column := q-1$ downto 1 do  
  for $P_{i,j}$, $i = 1$ .and. $j = Column$ do in parallel
    { compute first row minimum sequentially }
    $P_{i,j} : Temp \leftarrow P_{i,j+1} : X$  
    { communicate left from $X$ to $Temp$ }
    $X := \min \{X, Temp\}$  
    { only $P_{1,j}$ is active }
  end in parallel
endfor
return ( $P_{1,1} : X$ )

end Min2DMesh
```
5.5.5 Example: Pseudocode for searching on the Two-Dimensional Mesh

function Search2DMesh ( L, n, x, Index )

Model: two-dimensional mesh \( M_{q,q} \), with \( p=n=q^2 \) processors

Input:  
  \( L \) (a list of \( n \) elements)  
  \( x \) (a search element)  
  \( \text{Index} (P_{i,j} : \text{Index} = (q-1)i + j) \)

  range: \( P_{i,j} \), \( 1 \leq i,j \leq q \)  
  front end variable  

Output: the smallest row-major index where \( x \) occurs in \( L \), or \( \infty \) if \( x \) is not in \( L \)

\( P_{1,1} : X := x \)

call Broadcast2DMesh ( \( P_{1,1}:X \), \( n \) )

for \( 1 \leq i, j \leq q \) do in parallel
  if \( P_{i,j} : L \neq P_{i,j} : X \) then
    \( P_{i,j} : \text{Index} := \infty \)
  endif
end in parallel

return ( Min2DMesh(\( \text{Index}, n \) ) )

end Search2DMesh
5.6 Performance Measures of Parallel Algorithms

- The best-case, average and worst-case complexities of a parallel algorithm are defined in terms of the number of parallel basic operations performed.

- One measure of the performance of a parallel algorithm results from comparing $W(n)$ to the smallest worst-case complexity $W^*(n)$ over all sequential algorithms for the problem.

**Speedup of a Parallel Algorithm, $S(n)$:**

$$S(n) = \frac{W^*(n)}{W(n)}$$
5.6 Performance Measures of Parallel Algorithms

- Good speedup usually comes with the additional cost of using many processors.
- When measuring the efficiency of a parallel algorithm, it is important to consider both the worst-case complexity $W(n)$ and the number of processors $p(n)$ used.

Cost of a parallel algorithm:

$$C(n) = p(n) \times W(n)$$
5.6 Performance Measures of Parallel Algorithms

- A parallel algorithm is *cost optimal* if \( C(n) = W^*(n) \).
- A parallel algorithm is considered cost efficient if its cost \( C(n) \) is within polylogarithmic (belonging \( O(\log^k n) \)) factor of being cost optimal.

**Efficiency:**

\[
E(n) = \frac{W^*(n)}{C(n)} \quad \quad \quad \quad E(n) = \frac{S(n)}{p(n)}
\]

- \( E(n) \leq 1 \)
- A parallel algorithm is cost optimal if, and only if, \( E(n) = 1 \)
5.6 Performance Measures of Parallel Algorithms

Performance Measurement of MinPRAM:

- MinPRAM has complexity $W(n) = \log_2 n$
- The best sequential algorithm for finding the minimum of $n$ elements has complexity $W^*(n) = n - 1$

$$S(n) = \frac{n - 1}{\log_2 n}$$

- MinPRAM utilizes $n/2$ processors

$$C(n) = \left(\frac{n}{2}\right) \log_2 n$$

$$E(n) = \frac{2(n - 1)}{n \log_2 n}$$
5.6 Performance Measures of Parallel Algorithms

Performance Measurement of MinCRCW:

Algorithm:

- In one parallel step a shared memory array Win[1:n] is initialized to 0.
- For each pair of numbers L[i] and L[j], i < j, P_{i,j} reads L[i] and L[j], compares them, writes a 1 to Win[i] if L[i] > L[j], and writes a 1 to Win[j] otherwise.
- Only L[k] loses each of the n-1 comparisons.
- Win[i] = 1, i ≠ k, and Win[k] = 0.

\[
\begin{align*}
S(n) &= n - 1 \\
C(n) &= \frac{n^2 - n}{2} \\
E(n) &= \frac{2}{n}
\end{align*}
\]

\[
\left(\frac{n}{2}\right) = \frac{n^2 - n}{2}
\]
function \textit{MinCRCW}(L[1:n])

Model: CRCW PRAM with \( p = \frac{n^2 - n}{2} \) processors

Input: \( L[1:n] \) (a list of size \( n \))
Output: the minimum value of a list element in \( L \)

\begin{align*}
&\text{for } 1 \leq i \leq n \text{ do in parallel} \\
&\quad \text{Win}[i] := 0 \\
&\text{end in parallel} \\
&\text{for } 1 \leq i, j \leq n \text{ and } i < j \text{ do in parallel} \{ \text{Pi,j reads and compares } L[i] \text{ and } L[j] \} \\
&\quad \text{if } L[i] > L[j] \text{ then} \\
&\quad\quad \text{Win}[i] := 1 \{ \text{processors Pi,j concurrently write 1 to Win}[i] \} \\
&\quad\quad \text{else} \\
&\quad\quad \quad \text{Win}[j] := 1 \{ \text{processors Pi,j concurrently write 1 to Win}[j] \} \\
&\quad\quad \text{endif} \\
&\text{end in parallel} \\
&\text{for } 1 \leq i \leq n \text{ do in parallel} \\
&\quad \text{if } \text{Win}[i] = 0 \text{ then } \text{IndexMin} := i \text{ endif} \\
&\text{end in parallel} \\
&\text{return } ( L[\text{IndexMin}] ) \\
\end{align*}

end \textit{MinCRCW}
Action of MinCRCW for the sample input list \([ L: 95, 10, 6, 15 ]\)
## 5.6 Performance Measures of Parallel Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Basic operations</th>
<th>P(n)</th>
<th>W(n)</th>
<th>S(n)</th>
<th>C(n)</th>
<th>E(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SumPRAM</td>
<td>+</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>n/logn</td>
<td>1/logn</td>
</tr>
<tr>
<td>MinPRAM</td>
<td>&lt;</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>n/logn</td>
<td>1/logn</td>
</tr>
<tr>
<td>SearchPRAM</td>
<td>&lt;</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>n/logn</td>
<td>1/logn</td>
</tr>
<tr>
<td>Sum2DMesh</td>
<td>+</td>
<td>n</td>
<td>(\sqrt{n})</td>
<td>(\sqrt{n})</td>
<td>(n^{3/2})</td>
<td>1/(\sqrt{n})</td>
</tr>
<tr>
<td>Min2DMesh</td>
<td>&lt;</td>
<td>n</td>
<td>(\sqrt{n})</td>
<td>(\sqrt{n})</td>
<td>(n^{3/2})</td>
<td>1/(\sqrt{n})</td>
</tr>
<tr>
<td>Search2DMesh</td>
<td>&lt;</td>
<td>n</td>
<td>(\sqrt{n})</td>
<td>(\sqrt{n})</td>
<td>(n^{3/2})</td>
<td>1/(\sqrt{n})</td>
</tr>
<tr>
<td>SumPT</td>
<td>+</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>n/logn</td>
<td>1/logn</td>
</tr>
<tr>
<td>MinPT</td>
<td>&lt;</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>n/logn</td>
<td>1/logn</td>
</tr>
<tr>
<td>SearchPT</td>
<td>&lt;</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>n/logn</td>
<td>1/logn</td>
</tr>
<tr>
<td>MinCRCW</td>
<td>+</td>
<td>(n^2)</td>
<td>1</td>
<td>n</td>
<td>(n^2)</td>
<td>1/n</td>
</tr>
<tr>
<td>DotProdPRAM</td>
<td>+</td>
<td>n</td>
<td>logn</td>
<td>n/logn</td>
<td>Nlogn</td>
<td>1/logn</td>
</tr>
<tr>
<td>DotProd2DMesh</td>
<td>+</td>
<td>n</td>
<td>(\sqrt{n})</td>
<td>(\sqrt{n})</td>
<td>(n^{3/2})</td>
<td>1/(\sqrt{n})</td>
</tr>
</tbody>
</table>
5.7 Speedup and Amdahl's Law

- Most problems have an inherently sequential component and therefore cannot be completely parallelized.

- If a fraction $f$ of the basic operations must be performed sequentially for any input, then

$$W(n) = \left( f + \frac{1-f}{p} \right) \times W^*(n)$$

$$S \leq \frac{1}{f + \frac{1-f}{p}}$$
5.8 Closing Remarks

- In the case of PRAMs, algorithms designed for CRCW and CREW models with \( p \) processors can be simulated on the EREW PRAM with \( p \) processors at a cost of a multiplicative complexity factor of \( \log p \).

- In the case of interconnection network models, the portability question is usually handled by establishing efficient ways in which to embed one interconnection network model into another.

- A sequential algorithm is entirely impractical unless it has polynomial complexity \( O(n^k) \).

- Sequential algorithms are sometimes called time efficient if they have polynomial worst-case complexity.

- A parallel algorithm on PRAM is considered to be time efficient if it has polylogarithmic \( \left( O((\log n)^k) \right) \) worst-case complexity.

- A parallel algorithm for solving a given problem, which can be solved by a sequential algorithm in polynomial time (belonging to the class \( P \)), is said to be in the class \( NC \) (Nick Pippenger’s Class).

- Class NC assumes more than a linear number of processors is impractical for large \( n \) given the present state of technology.
Chapter 6.1 & 6.2
(Parallel Sorting)

6.1. SORTING ON THE CRCW AND CREW PRAMS
6.2. ODD-EVEN MERGE SORT ON THE EREW PRAM
6.1 SORTING ON THE CRCW AND CREW PRAMS

SORTING ON CRCW

Procedure SortCRCW;

- Computes position of L[i] in the sorted order of L, determining number of elements in L, that are smaller than L[i], i = 1, ..., n

- Uses:
  - \((n^2 - n)/2\) processors
  - An auxiliary array Win[1:n] where Win[i] computes total number of wins of L[i]
6.1. (continued)

\textbf{procedure} SortCRCW(L[1:n])
\textbf{Model:} CRCW PRAM with p=(n^2 \ - n)/2 processors, write conflicts resolved by summing
\textbf{Input:} L[1:n]
\textbf{Output:} L[1:n] (L sorted in increasing order)
for 1 \leq i \leq n \textbf{ do in parallel}
\hspace{1em} Win[i] := 0
\textbf{end in parallel}
for 1 \leq i, j \leq n .and. i < j \textbf{ do in parallel}
\hspace{1em} if L[i] > L[j] then
\hspace{2em} processors P_{ij} concurrently write 1 tp Win[i]
\hspace{1em} else
\hspace{2em} processors P_{ij} concurrently write 1 tp Win[j]
\hspace{1em} endif
\textbf{end in parallel}
for i := 1 to n \textbf{ do in parallel}
\hspace{1em} P_{j1} reads Win[i] and writes L[i] to position 1 + Win[i] of L
\textbf{end in parallel}
\textbf{end Sort CRCW}
procedure SortCRCW(L[1:n])

Model: CRCW PRAM with p=(n^2 - n)/2 processors, write conflicts resolved by summing

Input: L[1:n]
Output: L[1:n] (L sorted in increasing order)

for 1≤i≤n do in parallel
    Win[i] := 0
end in parallel

for 1≤i,j≤n. and. i<j do in parallel
    if L[i] > L[j] then
        processors P_{ij} concurrently write 1 tp Win[i]
    else
        processors P_{ij} concurrently write 1 tp Win[j]
    endif
end in parallel

for i:=1 to n do in parallel
    P_{ji} reads Win[i] and writes L[i] to position 1 + Win[i] of L
end in parallel

end Sort CRCW
procedure SortCRCW(L[1:n])

Model: CRCW PRAM with p=(n^2 − n)/2 processors, write conflicts resolved by summing

Input: L[1:n]
Output: L[1:n] (L sorted in increasing order)

for 1 ≤ i ≤ n do in parallel
    Win[i] := 0
end in parallel

for 1 ≤ i, j ≤ n, and i < j do in parallel
    if L[i] > L[j] then
        processors P_{ij} concurrently write 1 to Win[i]
    else
        processors P_{ij} concurrently write 1 to Win[j]
    endif
end in parallel

for i := 1 to n do in parallel
    P_{j1} reads Win[i] and writes L[i] to position 1 + Win[i] of L
end in parallel

end Sort CRCW
procedure SortCRCW(L[1:n])

Model: CRCW PRAM with p=(n^2 - n)/2 processors, write conflicts resolved by summing

Input: L[1:n]
Output: L[1:n] (L sorted in increasing order)

for 1 ≤ i ≤ n do in parallel
    Win[i] := 0
end in parallel

for 1 ≤ i, j ≤ n, and i < j do in parallel
    if L[i] > L[j] then
        processors P_{ij} concurrently write 1 tp Win[i]
    else
        processors P_{ij} concurrently write 1 tp Win[j]
    endif
end in parallel

for i := 1 to n do in parallel
    P_{j1} reads Win[i] and writes L[i] to position 1 + Win[i] of L
end in parallel

end Sort CRCW
procedure SortCRCW(L[1:n])

Model: CRCW PRAM with \( p = \frac{n^2 - n}{2} \) processors, write conflicts resolved by summing

Input: \( L[1:n] \)

Output: \( L[1:n] \) (\( L \) sorted in increasing order)

for \( 1 \leq i \leq n \) do in parallel
    \( \text{Win}[i] := 0 \)
end in parallel

for \( 1 \leq i, j \leq n. \text{and. } i < j \) do in parallel
    if \( L[i] > L[j] \) then
        processors \( P_{ij} \) concurrently write 1 tp \( \text{Win}[i] \)
    else
        processors \( P_{ij} \) concurrently write 1 tp \( \text{Win}[j] \)
    endif
end in parallel

for \( i := 1 \) to \( n \) do in parallel
    \( P_{j_1} \) reads \( \text{Win}[i] \) and writes \( L[i] \) to position \( 1 + \text{Win}[i] \) of \( L \)
end in parallel

end Sort CRCW
6.1. (continued)

**procedure** SortCRCW(L[1:n])

*Model:* CRCW PRAM with $p = \frac{n^2 - n}{2}$ processors, write conflicts resolved by summing

*Input:* L[1:n]

*Output:* L[1:n] (L sorted in increasing order)

for $1 \leq i \leq n$ do in parallel
  Win[i] := 0
end in parallel

for $1 \leq i, j \leq n$ and $i < j$ do in parallel
  if L[i] > L[j] then
    processors $P_{ij}$ concurrently write 1 to Win[i]
  else
    processors $P_{ij}$ concurrently write 1 to Win[j]
  endif
end in parallel

for $i := 1$ to $n$ do in parallel
  $P_{ji}$ reads Win[i] and writes L[i] to position $1 + Win[i]$ of L
end in parallel

end Sort CRCW
Analysis

Sorting in CRCW: constant time

2 Major Drawbacks:

- Necessity of powerful concurrent write resolution model
- Requirement of large number of processors

\[ c(n) = \frac{n^2 - n}{2} \in \Theta(n^2), \text{ where } W^*(n) = \Theta(n\log n). \]

How can we enhance the efficiency?

by transforming SORTCRCW into SORTCREW (c(logn)) or to SORTEREW (\(\Theta(logn)\))
SORT CREW

- Uses $W[1:n:1:n]$ , an auxiliary 2D array
- Writes the comparison result of $L[i]$ and $L[j]$ to $W[i,j]$ or $W[j,i]$
- To find position of $L[i]$, we sum the numbers in the $i$th row of $W[j,i]$ and add 1 to the result
- These sums take : $\log n$ parallel steps in addition to $\Theta(n^2)$ processors.
ODD-EVEN MERGE SORT ON THE EREW PRAM

- Sorts list of size \(n\) on EREW PRAM using \(n\) processors
- Complexity is \(\Theta(\log^2 n)\) and cost is \(\Theta(n\log^2 n)\)
- Recall MergeSort!!! (hint: divide and conquer)

Procedure MergePRAM : for merging the sorted sublists \(L[1: n/2]\) and \(L[n/2 +1:n]\)
(suppose that \(n\) is a power of 2)
**PSEUDOCODE**

**Procedure** MergeSort PRAM(L[1:n]) **recursive**

**Model:** EREW PRAM (the number of processors needed depends on the implementation of MergePRAM)

**Input:** L[1:n] (a list of size n, n is a power of 2)

**Output:** L[1:n] (sorted list)

**end MergeSortPRAM**
ODD EVEN MERGE PRAM

Example of oddevenmerge PRAM
Example of oddevenmerge PRAM

Parallel recursive calls
Example of oddevenmerge PRAM

Parallel recursive calls
Parallel resolution of recursive calls
Parallel resolution of recursive calls

Compare and exchange
Parallel resolution of recursive calls

Compare and exchange
Parallel resolution of recursive calls

Compare and exchange

interleave
Parallel resolution of recursive calls

Compare and exchange

interleave

Compare and exchange
Parallel resolution of recursive calls

Compare and exchange

Interleave

Compare and exchange
Parallel resolution of recursive calls
Parallel resolution of recursive calls

Compare and exchange

Interleave

Compare and exchange

Interleave

2 8  23 11  17 10  55 79

2 8  11 23  10 17  55 79

2 11 8  23  10 55 17 79

2 8 11 23  10 17 55 79

2 10 8  17 11  55 23 79
Parallel resolution of recursive calls
Parallel resolution of recursive calls

2  8

23 11

17 10

55 79

Compare and exchange

2  8

11 23

10 17

55 79

interleave

2  11  8  23

10  55  17  79

Compare and exchange

2  8  11  23

10  17  55  79

interleave

2  10  8  17  11  55  23  79

Compare and exchange

2  8  10  11  17  23  55  79
Parallel resolution of recursive calls

Resolve level 2
Parallel resolution of recursive calls

Resolve level 2

Compare and exchange

Interleave

Compare and exchange

Interleave

Compare and exchange

Resolve level 1
Parallel resolution of recursive calls

Resolve level 2

Compare and exchange

Resolve level 1

interleave

Compare and exchange

interleave

Resolve level 0

Compare and exchange
COMPLEXITY ANALYSIS

- For an input list of size $n$, OddEvenMergeSortPRAM makes $\log n$ parallel calls each involving concurrent calls to OddEvenMergePRAM.
- Each of the input sublists involved in the $i$th parallel call has size $2^{(i-1)}$.
- Since OddEvenMergePRAM performs $i$ parallel comparisons for an input list of size $2^i$, the total number of parallel comparisons performed by OddEvenMergeSort PRAM is:
  - $W(n) = 1 + 2 + 3 + \ldots + \log n = \Theta((\log n)^2)$
  - And the cost $C(n) = \Theta(n \log^2 n)$
6.3 Sorting on the One-Dimensional Mesh

- Any comparison based parallel sorting algorithm on $M_p$, $p=p(n)=n$ performs at least $(n-1)$ communication steps.

- A speedup of at most $(\log n)$ can be achieved.

- Two sorting algorithms on $M_p$ achieving this speedup are:
  - $InsertionSort1DMesh$
  - $OddEvenSort1DMesh$
6.3.1 Parallelized *InsertionSort* on the One-Dimensional Mesh $M_p$

- InsertionSort1DMesh is a parallelization of the forward scan version of *InsertionSort*.
- Multiple elements are inserted concurrently.
- The algorithm inputs all values in the list to the processor $P_1$.
- Each processor contains two variables, namely $A$ and $B$.
- $P_i$: $A$ is initialized to $+\infty$, $1 \leq i < n+1$
- Performs two phases.
Phase 1: One more processor on the right becomes active in each step.

In each step:

- Read the next list element into $P_1$: B.
- Compare A and B in each of the active processors. Interchange (swap) if $A > B$.
- Communicate B to the next processor (propagate B) making it active.

Total number of steps = n.
Phase 2: One more processor on the left becomes idle in each step.

In each step:

- Compare A and B in each of the active processors. Interchange (swap) if A > B.
- Communicate B to the next processor (propagate B).

Total number of steps = (n-1).
**Action of InsertionSort1DMesh with list 23,2,40,1,9:**

Processor \( P_i \):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialize A to +( \infty )</strong></td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
</tbody>
</table>

### Phase 1

<table>
<thead>
<tr>
<th>Step 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read into B:</td>
<td>+( \infty )</td>
<td>23</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
<tr>
<td>Compare &amp; swap:</td>
<td>23</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
<tr>
<td>Propagate B:</td>
<td>23</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read into B:</td>
<td>23</td>
<td>2</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
<tr>
<td>Compare &amp; swap:</td>
<td>2</td>
<td>23</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
<tr>
<td>Propagate B:</td>
<td>2</td>
<td>23</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read into B:</td>
<td>2</td>
<td>40</td>
<td>+( \infty )</td>
<td>23</td>
<td>+( \infty )</td>
</tr>
<tr>
<td>Compare &amp; swap:</td>
<td>2</td>
<td>40</td>
<td>23</td>
<td>+( \infty )</td>
<td>+( \infty )</td>
</tr>
<tr>
<td>Propagate B:</td>
<td>2</td>
<td>40</td>
<td>23</td>
<td>40</td>
<td>+( \infty )</td>
</tr>
</tbody>
</table>

Note: The values in the cells represent the elements of the list at each step of the algorithm.
Read into B: [2, 1, 23, 40, +∞, +∞, +∞, +∞, +∞, +∞]
Compare & swap: [1, 2, 23, 40, +∞, +∞, +∞, +∞, +∞, +∞]
Propagate B: [1, 2, 23, 2, +∞, 40, +∞, +∞, +∞, +∞]

Read into B: [1, 9, 23, 2, +∞, 40, +∞, +∞, +∞, +∞]
Compare & swap: [1, 9, 2, 23, 40, +∞, +∞, +∞, +∞, +∞]
Propagate B: [1, 9, 2, 9, 40, 23, +∞, +∞, +∞, +∞]
Phase 2

Propagate B:

Compare & swap: 

Propagate B:

Compare & swap: 

Propagate B:

Compare & swap: 

Sorted list is in $P_i$: $A$, $1 \leq i < n+1$
- *InsertionSort* performs $2n-1$ in parallel comparison steps.
- $W(n) = 2n-1$ (n steps for phase 1, n-1 steps for phase 2)
- $C(n) = n(2n-1)$
- $S(n) \in \Theta(\log n)$
procedure InsertionSort1DMesh (n)
Model: one dimensional mesh $M_p$ with $p=n+1$ processors
External input: $n$ list elements read into $P_1:B$ one element at a time
Output: sorted list resides in $A$

for $P_i$, $1 \leq i < n+1$ do in parallel
    $A := +\infty$
end in parallel
{ phase 1 }
for $i:=1$ to $n$ do
    read ($P_i:B$) {read $i^{th}$ list element}
    for $P_j$, $1 \leq j \leq i$ do in parallel
        if $A > B$ then
            call interchange($A,B$)
        endif
        $P_{j+1} : B \leq P_j : B$ {propagate $B$ to the right}
    end in parallel
endfor
{phase 2}
for $i:=2$ to $n$ do
    for $P_j$, $i \leq j \leq n$ do in parallel
        if $A > B$ then
            call interchange($A,B$)
        endif
        $P_{j+1} : B \leq P_j : B$ {propagate $B$ to the right}
    end in parallel
end for
end InsertionSort1DMesh
6.3.2 Odd-Even Transposition Sort on a One-Dimensional Mesh

- Not a straightforward parallelization of the sequential version.
- Requires $n$ iterations.
- In each iteration the following steps are performed:
  - **Odd-even exchange**: For all odd $i$, interchange $P_i:L$ and $P_{i+1}:L$ if the former is greater than the latter.
  - **Even-odd exchange**: For all even $i$, interchange $P_i:L$ and $P_{i+1}:L$ if the former is greater than the latter.
Action of *OddEvenSort1DMesh* with list 23, 2, 40, 1, 9, -2:

<table>
<thead>
<tr>
<th>Processors</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>23</td>
<td>2</td>
<td>40</td>
<td>1</td>
<td>9</td>
<td>-2</td>
</tr>
<tr>
<td>Step 1</td>
<td>2</td>
<td>23</td>
<td>1</td>
<td>40</td>
<td>-2</td>
<td>9</td>
</tr>
<tr>
<td>Step 2</td>
<td>2</td>
<td>1</td>
<td>23</td>
<td>-2</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>Step 3</td>
<td>1</td>
<td>2</td>
<td>-2</td>
<td>23</td>
<td>9</td>
<td>40</td>
</tr>
<tr>
<td>Step 4</td>
<td>1</td>
<td>-2</td>
<td>2</td>
<td>9</td>
<td>23</td>
<td>40</td>
</tr>
<tr>
<td>Step 5</td>
<td>-2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>23</td>
<td>40</td>
</tr>
<tr>
<td>Step 6</td>
<td>-2</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>23</td>
<td>40</td>
</tr>
</tbody>
</table>
Oddevensort-1Dmesh is optimal comparison based sorting algorithm on 1D mesh.

- $W(n) = n$
- $C(n) = n^2$
- $S(n) \in \Theta(\log n)$

Its cost and speedup are not promising for large $n$!!!

How about 2D mesh???
procedure OddEvenSort1DMesh (L,n)
Model: one dimensional mesh M_p with p=n processors
Input: L (a list of size n)
Output: L sorted in nondecreasing order
for Step:=1 to n do
if odd(Step) then
    for P_j, 1≤j≤n-1 .and. odd(i) do in parallel {odd-even exchange}
        P_i: Temp <= P_j+1: L {communicate left from L to Temp}
        if L>Temp then
            P_{j+1}: L <= P_j: L {propagate L to the right}
            L := Temp
        endif
    end in parallel
else {Step is even}
    for P_i, 2≤i≤n-2 .and. even(i) do in parallel {even-odd exchange}
        P_i: Temp <= P_{i+1}: L {communicate left from L to Temp}
        if L>Temp then
            P_{i+1}: L <= P_i: L {propagate L to the right}
            L := Temp
        endif
    end in parallel
endif
end for
end OddEvenSort1DMesh
6.4 Sorting on the Two-Dimensional Mesh

- A new type of ordering “snake ordering” (or raw-major ordering)

- Processor $P_{i,j}$ is the $k^{th}$ processor where:

\[
k = \begin{cases} 
q(i-1)+j & \text{if } i \text{ is odd, } 1 \leq i, j \leq q \\
qi-j+1 & \text{if } i \text{ is even, } 1 \leq i, j \leq q 
\end{cases}
\]
• On the two dimensional mesh $M_{q,q}$ assume $n=q^2$ for convenience.

• The algorithm involves $\lceil \log_2 n \rceil + 1$ steps.

• Each odd-numbered step is a call to $OddEvenRowSort$.
  - Achieved by emulating $OddEvenSort1DMesh$ for $M_q$.
  - If the row number, $i$, is odd then sort row $i$ in increasing order.
  - If it is even then row $i$ is sorted in decreasing order.

• Each even numbered step is a call to $OddEvenColumnSort$.
  - Column $j$ is sorted in increasing order.
Action of \textit{Sort2DMesh} with list 23,2,40,1,9,-2,2,43,-7,-4,23,90,3,15,16,76:

**Step 1: row sort**
- Odd-row increasing order
- Even row decreasing order

**Step 2: Column sort**
- Columns in increasing order

**Step 3: row sort**
- Odd-row increasing order
- Even row decreasing order

**Step 4: Column sort**
- Columns in increasing order

**Step 5: row sort**
- Odd-row increasing order
- Even row decreasing order
Both *OddEvenRowSort* and *OddEvenColumnSort* perform $q = \sqrt{n}$ parallel comparisons.

- $W(n) = (\lceil \log_2 n \rceil + 1)\sqrt{n}$
- $C(n) = n(\lceil \log_2 n \rceil + 1)\sqrt{n}$
- $S(n) \in \Theta(\sqrt{n})$

*Sort2DMesh* is superior to *OddEvenSort1DMesh*

*Shearsort, Bitonicmergesort, LS3 sort* by Lang, Schimmler, Schmeck and Schröder, and many others...

*Eg.*
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/twodim/shear/shearsorten.htm
Pseudocode for \textit{Sort2DMesh}:

\begin{verbatim}
procedure Sort2DMesh (L,n)
    Model: two dimensional mesh $M_{q,q}$ with p=n=q^2 processors
    Input: L (a list of size n)
    Output: L sorted in snake order
    for Step:=1 to log_2 n +1 do
        if odd(Step) then
            call OddEvenRowSort (L,n)
        else {Step is even}
            call OddEvenColumnSort (L,n)
        endif
    end for
end Sort2DMesh
\end{verbatim}