Basic Techniques of Parallel Programming & Examples

- Problems with a very large degree of (data) parallelism: (PP ch. 3)
  - Image Transformations:
    • Shifting, Rotation, Clipping etc.
  - Pixel-level Image Processing: (PP ch. 12)

- Divide-and-conquer Problem Partitioning: (pp ch. 4)
  - Parallel Bucket Sort
  - Numerical Integration:
    • Trapezoidal method using static assignment.
    • Adaptive Quadrature using dynamic assignment.

- Pipelined Computation (pp ch. 5)
  - Pipelined Addition
  - Pipelined Insertion Sort
  - Pipelined Solution of A Set of Upper-Triangular Linear Equations

To improve throughput of a number of instances of the same problem

Data parallelism (DOP) scale well with size of problem

Parallel Programming (PP) book, Chapters 3-7, 12
Basic Techniques of Parallel Programming & Examples

• **Synchronous Iteration** (Synchronous Parallelism) : (PP ch. 6)
  
  – **Barriers:** Implementations
    1. Counter Barrier Implementation.
    2. Tree Barrier Implementation.

  – **Synchronous Iteration Program Example:**
    • Iterative Solution of Linear Equations (Jacobi iteration)

• **Dynamic Load Balancing** (PP ch. 7)
  
  – Centralized Dynamic Load Balancing.
  – Decentralized Dynamic Load Balancing:
    • Distributed Work Pool Using Divide And Conquer.
    • Distributed Work Pool With Local Queues In Slaves.
    • Termination Detection for Decentralized Dynamic Load Balancing.
  
  – **Example:** Shortest Path Problem (Moore’s Algorithm).
Problems with large degree of (data) parallelism:

Example: Image Transformations

Common Pixel-Level Image Transformations:

- **Shifting:**
  - The coordinates of a two-dimensional object shifted by $\Delta x$ in the x-direction and $\Delta y$ in the y-dimension are given by:
    \[
    x' = x + \Delta x \\ y' = y + \Delta y
    \]
    where $x$ and $y$ are the original, and $x'$ and $y'$ are the new coordinates.

- **Scaling:**
  - The coordinates of an object magnified by a factor $S_x$ in the x direction and $S_y$ in the y direction are given by:
    \[
    x' = xS_x \\ y' = yS_y
    \]
    where $S_x$ and $S_y$ are greater than 1. The object is reduced in size if $S_x$ and $S_y$ are between 0 and 1. The magnification or reduction need not be the same in both x and y directions.

- **Rotation:**
  - The coordinates of an object rotated through an angle $\theta$ about the origin of the coordinate system are given by:
    \[
    x' = x \cos \theta + y \sin \theta \\ y' = -x \sin \theta + y \cos \theta
    \]

- **Clipping:**
  - Deletes from the displayed picture those points outside a defined rectangular area. If the lowest values of $x$, $y$ in the area to be display are $x_1$, $y_1$, and the highest values of $x$, $y$ are $x_h$, $y_h$, then:
    \[
    x_1 \leq x \leq x_h \\ y_1 \leq y \leq y_h
    \]
    needs to be true for the point $(x, y)$ to be displayed, otherwise $(x, y)$ is not displayed.

Parallel Programming book, Chapter 3, Chapter 12 (pixel-based image processing)

Also low-level pixel-based image processing

DOP = $O(n^2)$
Domain decomposition partitioning used
(similar to 2-d grid ocean example)

Possible Static Image Partitionings

- Image size: 640x480:
- To be copied into array: 
  `map[][ ]` from image file
- To be processed by 48 Processes or Tasks

**Communication** = \( \frac{4n}{\sqrt{p}} \)

**Computation** = \( \frac{n^2}{p} \)

\[ C = \frac{4 \times \sqrt{p}}{n} \]

**Communication to Computation Ratio** = \( \frac{2p}{n} \)

**Pixel-based image processing Example: Sharpening Filter**

Updated \( X_4 = (8X_4 - X_0 - X_1 - X_2 - X_3 - X_5 - X_6 - X_7 - X_8)/9 \)

**Communication** = 2n

**Computation** = \( \frac{n^2}{p} \)

**c-to-c ratio** = \( \frac{2p}{n} \)

**Possible Static Image Partitioning**

- 80x80 blocks
- 10x640 strips (of image rows)

**Block Assignment:**

**Strip Assignment:**

- **Regional Processing Example:**
- **Parallel Programming book, Chapters 12**

More on pixel-based image processing

Figure 3.3 Partitioning into region for individual processes
Message Passing Image Shift Pseudocode Example  (48, 10x640 strip partitions)

Master

for (i = 0; i < 8; i++)                         /* for each 48 processes */
for (j = 0; j < 6; j++) {
    p = i*80;                               /* bit map starting coordinates */
    q = j*80;
    for (i = 0; i < 80; i++)                /* load coordinates into array x[], y[]*/
        x[i] = p + i;
        y[i] = q + j;
    }
    z = j + 8*i;                            /* process number */
    send(Pz, x[0], y[0], x[1], y[1] ... x[6399], y[6399]);
    /* send coords to slave*/
}

for (i = 0; i < 8; i++)                         /* for each 48 processes */
for (j = 0; j < 6; j++) {                       /* accept new coordinates */
    z = j + 8*i;                            /* process number */
    recv(Pz, a[0], b[0], a[1], b[1] ... a[6399], b[6399]);
    /* receive new coords */
    for (i = 0; i < 6400; i += 2) {            /* update bit map */
        map[ a[i] ][ b[i] ] = map[ x[i] ][ y[i] ];
    }
Message Passing Image Shift Pseudocode

Example (48, 10x640 strip partitions)

Slave (process i)

recv(Pmaster, c[0] ... c[6400]);
for (i = 0; i < 6400; i += 2) {
  c[i] = c[i] + delta_x;
  c[i+1] = c[i+1] + delta_y;
}
send(Pmaster, c[0] ... c[6399]);

/* receive block of pixels to process */
/* transform pixels */
/* shift in x direction */
/* shift in y direction */
/* send transformed pixels to master */

i.e Get pixel coordinates to work on from master process

Update points (data parallel comp.)
Send results to master process

Or other pixel-based computation

More on pixel-based image processing
Parallel Programming book, Chapters 12
Image Transformation Performance Analysis

- Suppose each pixel requires one computational step and there are \( n \times n \) pixels. If the transformations are done sequentially, there would be \( n \times n \) steps so that:
  \[ t_s = n^2 \]
  and a time complexity of \( O(n^2) \).

- Suppose we have \( p \) processors. The parallel implementation (column/row or square/rectangular) divides the region into groups of \( n^2/p \) pixels. The parallel computation time is given by:
  \[ t_{comp} = \frac{n^2}{p} \]
  which has a time complexity of \( O(n^2/p) \).

- Before the computation starts, the bit map must be sent to the processes. If sending each group cannot be overlapped in time, essentially we need to broadcast all pixels, which may be most efficiently done with a single bcast() routine.

- The individual processes have to send back the transformed coordinates of their group of pixels requiring individual send()s or a gather() routine. Hence the communication time is:
  \[ t_{comm} = O(n^2) \]

- So that the overall execution time is given by:
  \[ t_p = t_{comp} + t_{comm} = O(n^2/p) + O(n^2) \]

- C-to-C Ratio = \( p \)
Divide-and-Conquer

- One of the most fundamental techniques in parallel programming.
- The problem is simply divided into separate smaller subproblems usually of the same form as the larger problem and each part is computed separately.
- Further divisions done by recursion.
- Once the simple tasks are performed, the results are combined leading to larger and fewer tasks.
- M-ary (or M-way) Divide and conquer: A task is divided into M parts at each stage of the divide phase (a tree node has M children).

Parallel Programming book, Chapter 4
Divide-and-Conquer Example

Bucket Sort

- On a sequential computer, it requires \( n \) steps to place the \( n \) numbers to be sorted into \( m \) buckets (e.g. by dividing each number by \( m \)).

- If the numbers are uniformly distributed, there should be about \( \frac{n}{m} \) numbers in each bucket.

- Next the numbers in each bucket must be sorted: Sequential sorting algorithms such as Quicksort or Mergesort have a time complexity of \( O(n \log_2 n) \) to sort \( n \) numbers.
  
  - Then it will take typically \( (\frac{n}{m}) \log_2 (\frac{n}{m}) \) steps to sort the \( \frac{n}{m} \) numbers in each bucket, leading to sequential time of:

\[
t_s = n + m((\frac{n}{m}) \log_2 (\frac{n}{m})) = n + n \log_2 (\frac{n}{m}) = O(n \log_2 (\frac{n}{m}))
\]
Sequential Bucket Sort

Divide Numbers into Buckets

Sort Numbers in each Bucket

O(n)

n Unsorted numbers

m Buckets

Sort contents of buckets

Merge lists

Sorted numbers

O(nlog₂ (n/m))

Ideally n/m numbers per bucket (range)

Figure 4.5 Bucket sort
Parallel Bucket Sort

- Bucket sort can be parallelized by assigning one processor for each bucket this reduces the sort time to \((n/p)\log(n/p)\) (\(m = p\) processors).
- Can be further improved by having processors remove numbers from the list into their buckets, so that these numbers are not considered by other processors.
- Can be further parallelized by partitioning the original sequence into \(m\) regions, one region for each processor.
- Each processor maintains \(p\) “small” buckets and separates the numbers in its region into its small buckets.
- These small buckets are then emptied into the \(p\) final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket \(i\) to processor \(i\)).
- Phases:
  - **Phase 1:** Partition numbers among processors. (\(m = p\) processors)
  - **Phase 2:** Separate numbers into small buckets in each processor.
  - **Phase 3:** Send to large buckets.
  - **Phase 4:** Sort large buckets in each processor.
Parallel Version of Bucket Sort

Phase 1

Phase 2

Phase 3

Phase 4

Computation $O(n/m)$

Communication $O((m - 1)(n/m^2))$ ~ $O(n/m)$

Computation $O((n/m)\log_2(n/m))$

Ideally: Each large bucket has $n/m$ numbers
Each small bucket has $n/m^2$ numbers
Performance of Message-Passing Bucket Sort

- Each small bucket will have about \( \frac{n}{m^2} \) numbers, and the contents of \( m - 1 \) small buckets must be sent (one bucket being held for its own large bucket). Hence we have:

\[
\begin{align*}
  t_{\text{comm}} &= (m - 1)\left(\frac{n}{m^2}\right) \\
  \text{and} \\
  t_{\text{comp}} &= \frac{n}{m^2} + \left(\frac{n}{m}\right)\log_2\left(\frac{n}{m}\right)
\end{align*}
\]

and the overall run time including message passing is:

\[
t_p = \frac{n}{m} + (m - 1)\left(\frac{n}{m^2}\right) + \left(\frac{n}{m}\right)\log_2\left(\frac{n}{m}\right)
\]

- Note that it is assumed that the numbers are uniformly distributed to obtain the above performance.

- If the numbers are not uniformly distributed, some buckets would have more numbers than others and sorting them would dominate the overall computation time. This leads to load imbalance among processors.

- The worst-case scenario would be when all the numbers fall into one bucket.
More Detailed Performance Analysis of Parallel Bucket Sort

- **Phase 1. Partition numbers among processors:**
  - Involves computation and communication
  - \( n \) computational steps for a simple partitioning into \( p \) portions each containing \( n/p \) numbers. \( t_{\text{comp}1} = n \)
  - Communication time using a broadcast or scatter:
    \[ t_{\text{comm}1} = t_{\text{startup}} + t_{\text{data}}n \]

- **Phase 2. Separate numbers into small buckets in each processor:**
  - Computation only to separate each partition of \( n/p \) numbers into \( p \) small buckets in each processor: \( t_{\text{comp}2} = n/p \)

- **Phase 3. Small buckets are distributed. No computation**
  - Each bucket has \( n/p^2 \) numbers (with uniform distribution).
  - Each process must send out the contents of \( p-1 \) small buckets.
  - Communication cost with no overlap - using individual send()!
    - Upper bound: \( t_{\text{comm}3} = p(1-p)(t_{\text{startup}} + (n/p^2)t_{\text{data}}) \)
    - Communication time from different processes fully overlap:
      - Lower bound: \( t_{\text{comm}3} = (1-p)(t_{\text{startup}} + (n/p^2)t_{\text{data}}) \)

- **Phase 4. Sorting large buckets in parallel. No communication.**
  - Each bucket contains \( n/p \) numbers
    \[ t_{\text{comp}4} = (n/p)\log(n/P) \]

Overall time: \( t_p = t_{\text{startup}} + t_{\text{data}}n + n/p + (1-p)(t_{\text{startup}} + (n/p^2)t_{\text{data}}) + (n/p)\log(n/P) \)
Divide-and-Conquer Example
Numerical Integration Using Rectangles

\[ f(x) \]

- \( n \) intervals
- \( p \) processes or processors

\[ f(p) \quad f(q) \]

\[ a \quad p \quad \delta \quad q \quad b \quad x \]

Comp = \( \frac{n}{p} \)
Comm = \( O(p) \)
C-to-C = \( O(P^2 / n) \)

Also covered in lecture 5 (MPI example)

Parallel Programming book, Chapter 4
More Accurate Numerical Integration Using Rectangles

Also covered in lecture 5 (MPI example)
Numerical Integration
Using The Trapezoidal Method

Each region is calculated as
\[ \frac{1}{2}(f(p) + f(q)) \delta \]

\[ f(x) \]

\[ a \quad p \quad \delta \quad q \quad b \quad x \]

\[ n \text{ intervals} \]
Numerical Integration Using The Trapezoidal Method:
Static Assignment Message-Passing

• Before the start of computation, one process is statically assigned to compute each region.
• Since each calculation is of the same form an SPMD model is appropriate.
• To sum the area from x = a to x=b using p processes numbered 0 to p-1, the size of the region for each process is (b-a)/p.
• A section of SMPD code to calculate the area:

Process Pi
if (i == master) { /* broadcast interval to all processes */
    printf("Enter number of intervals ");
    scanf("%d", &n);
}
bcast(&n, P_group); /* broadcast interval to all processes */
region = (b-a)/p; /* length of region for each process */
start = a + region * i; /* starting x coordinate for process */
end = start + region; /* ending x coordinate for process */
d = (b-a)/n; /* size of interval */
area = 0.0;
for (x = start; x < end; x = x + d)
    area = area + 0.5 * (f(x) + f(x+d)) * d;
reduce_add(&integral, &area, Pgroup); /* form sum of areas */

Computation = O(n/p)  Communication ~ O(p)
C-to-C ratio = O(p / (n/p) = O(p^2 /n)
Example:  n = 1000  p = 8  C-to-C =  64/1000 = 0.064
Numerical Integration And Dynamic Assignment: Adaptive Quadrature

- To obtain a better numerical approximation:
  - An initial interval $\delta$ is selected.
  - $\delta$ is modified depending on the behavior of function $f(x)$ in the region being computed, resulting in different $\delta$ for different regions.
  - The area of a region is recomputed using different intervals $\delta$ until a good $\delta$ proving a close approximation is found.

- One approach is to double the number of regions successively until two successive approximations are sufficiently close.

- Termination of the reduction of $\delta$ may use three areas A, B, C, where the refinement of $\delta$ in a region is stopped when 1- the area computed for the largest of A or B is close to the sum of the other two areas, or 2- when C is small.

- Such methods to vary are known as Adaptive Quadrature.

- Computation of areas under slowly varying parts of $f(x)$ require less computation those under rapidly changing regions requiring dynamic assignment of work to achieve a balanced load and efficient utilization of the processors.

How?
Reducing the size of $\delta$ is stopped when 1- the area computed for the largest of A or B is close to the sum of the other two areas, or 2- when C is small.
Gravitational N-Body Problem

• To find the positions and movements of bodies in space that are subject to gravitational forces. Newton Laws:

\[ F = \frac{Gm_am_b}{r^2} \quad F = \text{mass} \times \text{acceleration} \]

\[ F = m \frac{dv}{dt} \quad v = \frac{dx}{dt} \]

For computer simulation:

\[ F = m \left( \frac{v_{t+1} - v_t}{\Delta t} \right) \]

\[ v_{t+1} = v_t + F \frac{\Delta t}{m} \]

\[ x_{t+1} - x_t = \frac{v_{t+1}}{\Delta t} \Delta t \]

Sequential Code:

```c
for (t = 0; t < tmax; t++) /* for each time period */
    for (i = 0; i < n; i++) { /* for each body */
        F = Force_routine(i); /* compute force on body i */
        v[i]_new = v[i] + F * dt; /* compute new velocity and */
        x[i]_new = x[i] + v[i]_new * dt /* new position */
    }
    for (i = 0; i < nmax; i++) { /* for each body */
        v[i] = v[i]_new /* update velocity, position */
        x[i] = x[i]_new
    }
```

Parallel Programming book, Chapter 4
Gravitational N-Body Problem: Barnes-Hut Algorithm

- To parallelize problem: Groups of bodies partitioned among processors. Forces communicated by messages between processors.
  - Large number of messages, $O(N^2)$ for one iteration.
- Approximate a cluster of distant bodies as one body with their total mass.
- This clustering process can be applies recursively.
- **Barnes Hut**: Uses divide-and-conquer clustering. For 3 dimensions:
  - Initially, one cube contains all bodies.
  - Divide into 8 sub-cubes. (4 parts in two dimensional case).
  - If a sub-cube has no bodies, delete it from further consideration.
  - If a cube contains more than one body, recursively divide until each cube has one body.
  - This creates an oct-tree which is very unbalanced in general.
  - After the tree has been constructed, the total mass and center of gravity is stored in each cube.
  - The force on each body is found by traversing the tree starting at the root stopping at a node when clustering can be used.
  - The criterion when to invoke clustering in a cube of size $d \times d \times d$:
    \[ r \geq d/\theta \]
    \[ r = \text{distance to the center of mass} \]
    \[ \theta = \text{a constant, 1.0 or less, opening angle} \]
  - Once the new positions and velocities of all bodies is computed, the process is repeated for each time period requiring the oct-tree to be reconstructed.
Two-Dimensional Barnes-Hut

Recursive Division of Two-dimensional Space

Locality Goal:

Bodies close together in space should be on same processor
Barnes-Hut Algorithm

- Main data structures: array of bodies, of cells, and of pointers to them
  - Each body/cell has several fields: mass, position, pointers to others
  - Pointers are assigned to processes
N-Body Problem: A Balanced Partitioning Approach: Orthogonal Recursive Bisection (ORB)

For a two-dimensional square:

- A vertical line is found that created two areas with equal number of bodies.
- For each area, a horizontal line is found that divides into two areas with an equal number of bodies.
- This is repeated recursively until there are as many areas as processors.
- One processor is assigned to each area.
- Drawback: High overhead for large number of processors.

ORB is a form of domain decomposition
Pipelined Computations

- Given the problem can be divided into a series of sequential operations (processes), the pipelined approach can provide increased speed “problem instance throughput” under any of the following three "types" of computations:

  1. If more than one instance of the complete problem is to be executed. **Most common and/or Types 2, 3 below**
  2. A series of data items must be processed with multiple operations.
  3. If information to start the next process can be passed forward before the process has completed all its internal operations.

Implements problem throughput: instances/second

Does not improve the time for a problem instance (usually).

(similar to instruction pipelining)

Parallel Programming book, Chapter 5
Pipelined Computations Examples

Pipeline for unfolding the loop:
for (i = 0; i < n; i++)
sum = sum + a[i]

Pipeline for a frequency filter

Pipeline for a frequency filter

Signal without frequency $f_0$
Signal without frequency $f_1$
Signal without frequency $f_2$
Signal without frequency $f_3$

Filtered signal

Pipelined Sum

sum = 0 initially
Pipelined Computations

Pipeline Space-Time Diagram

Goal: Improve problem instance throughput: instances/sec
Ideal throughput improvement = number of stages = p

Type 1
Multiple instances of the complete problem

Each pipeline stage is a process or task

Time for m instances = (pipeline fill + number of instances) x stage delay
= (p - 1 + m) x stage delay

Stage delay = pipeline cycle

Number of stages = p  here  p = 6
Number of problem instances = m
Pipelined Computations

Problem Instances

<table>
<thead>
<tr>
<th>Instance 0</th>
<th>Instance 1</th>
<th>Instance 2</th>
<th>Instance 3</th>
<th>Instance 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$P_0$</td>
<td>$P_0$</td>
<td>$P_0$</td>
<td>$P_0$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_1$</td>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$P_2$</td>
<td>$P_2$</td>
<td>$P_2$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$P_3$</td>
<td>$P_3$</td>
<td>$P_3$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$P_4$</td>
<td>$P_4$</td>
<td>$P_4$</td>
<td>$P_4$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$P_5$</td>
<td>$P_5$</td>
<td>$P_5$</td>
<td>$P_5$</td>
</tr>
</tbody>
</table>

Time for $m$ instances = (pipeline fill + number of instances) x stage delay

= (p - 1 + m) x stage delay

Goal: Improve problem instance throughput: instances/sec

Ideal throughput improvement = number of stages = p

Alternate Pipeline Space-Time Diagram

Stage delay = pipeline cycle

Ideal Problem Instance Throughput = 1 / stage delay
Pipelined Computations: Type 3 (i.e overlap pipeline stages)

Information transfer sufficient to start next process

Staircase effect due to overlapping stages

(a) Processes with the same execution time

(b) Processes not with the same execution time

Pipelined Processing Where Information Passes To Next Stage Before End of Process

Partitioning pipeline processes onto processors to balance stages (delays)
Pipelined Addition

- The basic code for process $P_i$ is simply:

1. \texttt{recv($P_{i-1}$, accumulation);}  
2. accumulation += number;  
3. \texttt{send($P_{i+1}$, accumulation);} 

Or several numbers assigned to $P_i$

Pipelined Computations: Type 1 Example

Parallel Programming book, Chapter 5
Pipelined Addition: Analysis

- $t_{total} = \text{pipeline cycle x number of cycles}
  = (t_{comp} + t_{comm})(m + p -1)$ for $m$ instances and $p$ pipeline stages.
- For single instance of adding $n$ numbers:
  $t_{total} = (2(t_{startup} + t_{data})+1)n$
  Time complexity $O(n)$
- For $m$ instances of $n$ numbers:
  $t_{total} = (2(t_{startup} + t_{data})+1)(m+n-1)$
  For large $m$, average execution time $t_a$ per instance:
  $t_a = \frac{t_{total}}{m} = 2(t_{startup} + t_{data}) +1 = \text{Stage delay or cycle time}$
- For partitioned multiple instances:
  $t_{comp} = d$
  $t_{comm} = 2(t_{startup} + t_{data})$
  $t_{total} = (2(t_{startup} + t_{data}) + d)(m + n/d -1)$

$P = n = \text{numbers to be added}$
$T_{comp} = 1$  Each stage adds one number
$T_{comm} = \text{send + receive}$
$2(t_{startup} + t_{data})$
Stage Delay = $T_{comp} + T_{comm}$
= $T_{comp} = 2(t_{startup} + t_{data}) + 1$

Fill Cycles ignored

Pipeline stage delay

Each stage adds $d$ numbers
Number of stages = $n/d$

$Pipelined Computations: Type 1 Example$

$T_{comp} = \frac{1}{\text{problem instance throughout}}$

Fill Cycles

$m = \text{Number of instances}$

$\text{Number of stages} = \frac{n}{d}$
Pipelined Addition

Using a master process and a ring configuration

Master with direct access to slave processes
Pipelined Insertion Sort

- The basic algorithm for process $P_i$ is:

  $$\text{recv}(P_{i-1}, \text{number});$$

  IF (number > $x$) {
      $$\text{send}(P_{i+1}, x);$$
      $$x = \text{number};$$
  } ELSE send($P_{i+1}$, number);

Pipelined Computations: Type 2 Example

- Type 2: Series of data items
  Processed with multiple operations

- Series of numbers $x_{n-1} \ldots x_1 x_0$
  to be sorted

- Largest number (i.e. keep largest number)
- Next largest number

- Receive → Compare/Exchange → Send

Parallel Programming book, Chapter 5
Pipelined Insertion Sort

- Each process must continue to accept numbers and send on smaller numbers for all the numbers to be sorted, for n numbers, a simple loop could be used:

\[
\text{recv}(P_{i-1}, x); \\
\text{for} \ (j = 0; j < (n-i); j++) \ { \\
\text{recv}(P_{i-1}, \text{number}); \\
\text{IF} \ (\text{number} > x) \ { \\
\text{send}(P_{i+1}, x); \\
x = \text{number}; \\
} \ \text{ELSE} \ \text{send}(P_{i+1}, \text{number}); \\
} \\
\]
Pipelined Insertion Sort Example

Sorting phase = 2n -1 = 9 cycles or stage delays

Here: n = 5

Number of stages
Pipelined Insertion Sort: Analysis

- **Sequential implementation:**
  \[ t_s = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \quad \text{O}(n^2) \]

- **Pipelined:**
  - Takes \( n + n - 1 = 2n - 1 \) pipeline cycles for sorting using \( n \) pipeline stages and \( n \) numbers.
  - Each pipeline cycle has one compare and exchange operation:
    - Communication is one `recv()`, and one `send()`
    - \( t_{\text{comp}} = 1 \quad t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}}) \)
    - \( t_{\text{total}} = \text{cycle time} \times \text{number of cycles} \)
      \[ = (1 + 2(t_{\text{startup}} + t_{\text{data}}))(2n - 1) \quad \sim \text{O}(n) \]
Pipelined Insertion Sort

Unsorted Sequence

Sorting phase = 2n -1 = 9 cycles or stage delays
Stage delay = 1 + 2(t_{startup} + t_{data})

Unsorted Sequence

Pipelined Computations: Type 2 Example

Type 2: Series of data items processed with multiple operations
Solving A Set of Upper-Triangular Linear Equations (Back Substitution)

\[ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 \ldots + a_{nn}x_n = b_n \]
\[ \ldots \]
\[ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \]
\[ a_{21}x_1 + a_{22}x_2 = b_2 \]
\[ a_{11}x_1 = b_1 \]

\[ x_1 = \frac{b_1}{a_{11}} \]
\[ x_2 = \frac{b_2 - a_{21}x_1}{a_{22}} \]
\[ x_3 = \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \]
\[ \ldots \]
\[ x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}} \]

Here \( i = 1 \) to \( n \)

\( n = \) number of equations

Parallel Programming book, Chapter 5
Solving A Set of Upper-Triangular Linear Equations (Back Substitution)

Sequential Code:

- Given the constants a and b are stored in arrays and the value for unknowns $x_i$ (here $i=0$ to $n-1$) also to be stored in an array, the sequential code could be:

\[
\begin{align*}
x[0] &= b[0]/a[0][0] \\
\text{for} \ (i = 1; \ i < n; \ i++) \ {\{ \\
    \quad \text{sum} &= 0; \\
    \quad \text{for} \ (j = 0; \ j < i; \ j++) \ {\{ \\
    \quad \quad \text{sum} &= \text{sum} + a[i][j]*x[j]; \\
    \quad \quad x[i] &= (b[i] - \text{sum})/a[i][i]; \\
    \quad \}} \\
\}} \\
\end{align*}
\]

Complexity $O(n^2)$
Parallel Code:
• The pseudo code of process $P_i$ of the pipelined version could be:

```c
for (j = 0; j < i; j++) {
    recv($P_{i-1}$, $x[j]$);
    send($P_{i+1}$, $x[j]$);
}
sum = 0;
for (j = 0; j < i; j++) {
    sum = sum + $a[i][j]*x[j]$;
    $x[i] = (b[i] - sum)/a[i][i]$;
    send($P_{i+1}$, $x[j]$);
}
```

$1 < i < p = n$

Receive $x_0, x_1, ..., x_{i-1}$ from $P_{i-1}$
Send $x_0, x_1, ..., x_{i-1}$ to $P_{i+1}$

Compute sum term
Compute $x_i$
Send $x_i$ to $P_{i+1}$
Modified (for better overlap) Pipelined Solution

- The pseudo code of process $P_i$ of the pipelined version can be modified to start computing the sum term as soon as the values of $x$ are being received from $P_{i-1}$ and resend to $P_{i+1}$:

\[
\text{sum} = 0; \\
\text{for (j = 0; j < i; j++)} \{ \\
  \text{recv}(P_{i-1}, x[j]); \\
  \text{send}(P_{i+1}, x[j]); \\
  \text{sum} = \text{sum} + a[i][j]*x[j]; \\
\} \\
\text{x[i] = (b[i] - sum)/a[i][i];} \\
\text{send}(P_{i+1}, x[i]);
\]

This results in better overlap between pipeline stages as shown next.
Pipelined Solution of A Set of Upper-Triangular Linear Equations

Pipelined Computations: Type 3 (i.e overlap pipeline stages) Example

Pipeline processing using back substitution

Staircase effect due to overlapping stages

First value passed onward

Final computed value

Time
### Operation of Back-Substitution Pipeline

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>divide</td>
<td>recv($x_0$)</td>
<td>recv($x_0$)</td>
<td>recv($x_0$)</td>
<td>recv($x_0$)</td>
</tr>
<tr>
<td>send($x_0$)</td>
<td>send($x_0$)</td>
<td>send($x_0$)</td>
<td>send($x_0$)</td>
<td>send($x_0$)</td>
</tr>
<tr>
<td>end</td>
<td>multiply/add</td>
<td>recv($x_1$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>divide/subtract</td>
<td>send($x_1$)</td>
<td>send($x_1$)</td>
<td>send($x_1$)</td>
<td>send($x_1$)</td>
</tr>
<tr>
<td>send($x_1$)</td>
<td>multiply/add</td>
<td>recv($x_1$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>end</td>
<td>divide/subtract</td>
<td>send($x_2$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>send($x_2$)</td>
<td>multiply/add</td>
<td>recv($x_2$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>end</td>
<td>divide/subtract</td>
<td>send($x_3$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>send($x_3$)</td>
<td>multiply/add</td>
<td>recv($x_3$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>end</td>
<td>divide/subtract</td>
<td>send($x_4$)</td>
<td>multiply/add</td>
<td>multiply/add</td>
</tr>
<tr>
<td>send($x_4$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Staircase effect due to overlapping stages**

### Pipelined Computations: Type 3 (i.e overlap pipeline stages) Example
Pipelined Solution of A Set of Upper-Triangular Linear Equations: Analysis

Communication:
• Each process i in the pipelined version performs i recv()s, i + 1 send()s, where the maximum value for i is n. Hence the communication time complexity is O(n).

Computation:
• Each process in the pipelined version performs i multiplications, i additions, one subtraction, and one division, leading to a time complexity of O(n).
• The sequential version has a time complexity of O(n^2). The actual speed-up is not n however because of the communication overhead and the staircase effect from overlapping the stages of the pipelined parallel version.

Pipelined Computations: Type 3
(i.e overlap pipeline stages) Example
Synchronous Computations (Iteration)

- Iteration-based computation is a powerful method for solving numerical (and some non-numerical) problems.

- For numerical problems, a calculation is repeated in each iteration, a result is obtained which is used on the next iteration. The process is repeated until the desired results are obtained (i.e. convergence).
  - Similar to ocean 2d grid example

- Though iterative methods (between iterations) are sequential in nature (between iterations), parallel implementation can be successfully employed when there are multiple independent instances of the iteration or a single iteration is split into parallel processes using data parallelism (e.g. ocean). In some cases this is part of the problem specification and sometimes one must rearrange the problem to obtain multiple independent instances.

- The term "synchronous iteration" is used to describe solving a problem by iteration where different tasks may be performing separate iterations or parallel parts of the same iteration (e.g. ocean example) but the iterations must be synchronized using point-to-point synchronization, barriers, or other synchronization mechanisms.
  
  i.e. Fine grain event synch.  
  i.e. Conservative (group) synch.
Data Parallel Synchronous Iteration

- Each iteration composed of several parallel processes that start together at the beginning of each iteration. Next iteration cannot begin until all processes have finished previous iteration. Using forall:

\[
\text{for } (j = 0; j < n; j++) \quad /* \text{for each synch. iteration} */ \\
\quad \text{forall } (i = 0; i < N; i++) \quad /* N processes each using */ \\
\quad \quad \text{body}(i); \quad /* \text{specific value of } i */ \\
\quad \}
\]

- or:

\[
\text{for } (j = 0; j < n; j++) \quad /* \text{for each synchronous iteration} */ \\
\quad i = \text{myrank}; \quad /* \text{find value of } i \text{ to be used} */ \\
\quad \text{body}(i); \\
\quad \text{barrier}(\text{mygroup}); \quad /* \text{Part of the iteration given to } Pi */ \\
\quad \}
\]

Similar to ocean 2d grid computation (lecture 4)
Barrier Implementations

A conservative group synchronization mechanism applicable to both shared-memory as well as message-passing, [pvm_barrier(), MPI_barrier()] where each process must wait until all members of a specific process group reach a specific reference point “barrier” in their Computation.

- Possible Barrier Implementations:
  1. Using a counter (linear barrier). $O(n)$
  2. Using individual point-to-point synchronization forming:
     - A tree: $2 \log_2 n$ steps thus $O(\log_2 n)$
     - Butterfly connection pattern: $\log_2 n$ steps thus $O(\log_2 n)$
Processes Reaching A Barrier at Different Times

i.e synch wait time
Centralized Counter Barrier Implementation

- Called linear barrier since access to centralized counter is serialized, thus $O(n)$ time complexity.

$n$ is number of processes in the group
Message-Passing Counter Implementation of Barriers

The master process maintains the barrier counter:

- It counts the messages received from slave processes as they reach their barrier during arrival phase.
- Release slave processes during departure (or release) phase after all the processes have arrived.

```c
for (i = 0; i < n; i++) /* count slaves as they reach their barrier */
    recv(Pany);
for (i = 0; i < n; i++) /* release slaves */
    send(Pi);
```

2 phases:
1. Arrival
2. Departure (release)

Each phase n steps
Thus $O(n)$ time complexity

Can also use broadcast for release

More detailed operation of centralized counter barrier

For $O(n)$ time complexity

Barrier Implementations:

1. 2 phases:
   1. Arrival
   2. Departure (release)

2. Each phase n steps
   Thus $O(n)$ time complexity

Can also use broadcast for release

More detailed operation of centralized counter barrier

O(n) Time Complexity
Barrier Implementations:

1. Arrival Phase
   - log₂ n steps
   - Arrival at barrier

2. Departure Phase
   - Reverse of arrival phase
   - Departure from barrier
   - log₂ n steps

Tree Barrier Implementation:

- 2 phases:
  1. Arrival
  2. Departure (release)
- Each phase log₂ n steps
- Thus O(log₂ n) time complexity

Uses tree-structure point-to-point synchronization/messages to reduce congestion (vs. counter-based barrier) and improve time complexity - O(log₂ n) vs. O(n).

Arrival phase can also be used to implement reduction

2 log₂ n steps, time complexity O(log₂ n)

The departure phase also represents a possible implementation of broadcasts
Tree Barrier Implementation

Using point-to-point messages (message passing)

• Suppose 8 processes, $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$:
• **Arrival phase**: $\log_2(8) = 3$ stages
  – **First stage**:
    • $P_1$ sends message to $P_0$; (when $P_1$ reaches its barrier)
    • $P_3$ sends message to $P_2$; (when $P_3$ reaches its barrier)
    • $P_5$ sends message to $P_4$; (when $P_5$ reaches its barrier)
    • $P_7$ sends message to $P_6$; (when $P_7$ reaches its barrier)
  – **Second stage**:
    • $P_2$ sends message to $P_0$; ($P_2$ & $P_3$ reached their barrier)
    • $P_6$ sends message to $P_4$; ($P_6$ & $P_7$ reached their barrier)
• **Third stage**:
  • $P_4$ sends message to $P_0$; ($P_4$, $P_5$, $P_6$, & $P_7$ reached barrier)
  • $P_0$ terminates arrival phase; (when $P_0$ reaches barrier & received message from $P_4$)
• **Release phase** also 3 stages with a reverse tree construction.
• Total number of steps: $2 \log_2 n = 2 \log_2 8 = 6$ steps

$O(\log_2 n)$
Butterfly Connection Pattern

Message-Passing Barrier Implementation

- Butterfly pattern tree construction.
- Also uses point-to-point synchronization/messages (similar to normal tree barrier), but ..
- Has one phase only: Combines arrival with departure in one phase.
- $\log_2 n$ stages or steps, thus $O(\log_2 n)$ time complexity.
- Pairs of processes synchronize at each stage [two pairs of send() / receive()].
- For 8 processes:

  First stage: $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
  Second stage: $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
  Third stage: $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

Advantage over tree implementation: No separate arrival and release phases, $\log_2 n$ stages or steps vs. $2 \log_2 n$ steps
Synchronous Iteration Example: Iterative Solution of Linear Equations

Using Jacobi Iteration

• Given a system of \( n \) linear equations with \( n \) unknowns:

\[
\begin{align*}
  a_{n-1,0} x_0 + a_{n-1,1} x_1 + \ldots + a_{n-1,n-1} x_{n-1} &= b_{n-1} \\
  \vdots \\
  a_{1,0} x_0 + a_{1,1} x_1 + a_{1,2} x_2 + \ldots + a_{1,n-1} x_{n-1} &= b_1 \\
  a_{0,0} x_0 + a_{0,1} x_1 + a_{0,2} x_2 + \ldots + a_{0,n-1} x_{n-1} &= b_0 \\
\end{align*}
\]

By rearranging the \( i \)th equation:

\[
  a_{i,0} x_0 + a_{i,1} x_1 + a_{i,2} x_2 + \ldots + a_{i,n-1} x_{n-1} = b_i
\]

to:

\[
  x_i = \frac{1}{a_{i,i}} [b_i - (a_{i,0} x_0 + a_{i,1} x_1 + a_{i,2} x_2 + \ldots + a_{i,i-1} x_{i-1} + a_{i,i+1} x_{i+1} + a_{i,n-1} x_{n-1})]
\]

or

\[
  x_i = \frac{1}{a_{i,i}} \left[ b_i - \sum_{j \neq i} a_{i,j} x_j \right] 
\]

Possible initial value of \( x_i = b_i \)

Computing each \( x_i \) (\( i = 0 \) to \( n-1 \)) is \( O(n) \)
Thus overall complexity is \( O(n^2) \) per iteration

• This equation can be used as an iteration formula for each of the unknowns to obtain a better approximation.

• Jacobi Iteration: All the values of \( x \) are updated at once in parallel.

(i.e. updated values of \( x \) not used in current iteration)

Jacobi Iteration
Iterative Solution of Linear Equations

Jacobi Iteration Sequential Code:

Given the arrays `a[][]` and `b[]` holding the constants in the equations, `x[]` provided to hold the unknowns, and a fixed number of iterations, the code might look like:

```c
for (i = 0; i < n; i++)
    x[i] = b[i]; /* initialize unknowns */
for (iteration = 0; iteration < limit; iteration++)
    for (i = 0; i < n; i++)
        sum = 0;
        for (j = 0; j < n; j++)    /* compute summation of a[][]x[] */
            if (i != j) {
                sum = sum + a[i][j] * x[j];
                new_x[i] = (b[i] - sum) / a[i][i]; /* Update unknown */
            }
    for (i = 0; i < n; i++)          /* update values */
        x[i] = new_x[i];
```

Max. number of iterations $O(n^2)$ per iteration.
Iterative Solution of Linear Equations

Jacobi Iteration Parallel Code:

- In the sequential code, the for loop is a natural "barrier" between iterations.
- In parallel code, we have to insert a specific barrier. Also all the newly computed values of the unknowns need to be broadcast to all the other processes. \( \text{Comm} = O(n) \)
- Process \( P_i \) could be of the form:

```c
x[i] = b[i];                          /* initialize values */
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 1; j < n; j++)         /* compute summation of a[][]x[] */
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];    /* compute unknown */
    broadcast_receive(&new_x[i]);    /* broadcast values */
    global_barrier();             /* wait for all processes */
}
```

- The broadcast routine, broadcast_receive(), sends the newly computed value of \( x[i] \) from process \( i \) to other processes and collects data broadcast from other processes.

broadcast_receive() can be implemented by using \( n \) broadcast calls
Parallel Jacobi Iteration: Partitioning

• **Block allocation of unknowns:**
  – Allocate groups of $\frac{n}{p}$ consecutive unknowns to processors/processes in increasing order.

• **Cyclic allocation of unknowns:**
  – Processors/processes are allocated one unknown in cyclic order;
  – i.e., processor P0 is allocated $x_0, x_p, x_{2p}, \ldots, x_{(n/p)-1}p$, processor P1 is allocated $x_1, x_{p+1}, x_{2p+1}, \ldots, x_{(n/p)-1}p+1$, and so on.
  – Cyclic allocation has no particular advantage here (Indeed, may be disadvantageous because the indices of unknowns have to be computed in a more complex way).
Jacobi Iteration: Analysis

- Sequential Time equals iteration time * number of iterations. O(n^2) for each iteration.
- Parallel execution time is the time of one processor each operating over n/p unknowns.
- Computation for \( \tau \) iterations:
  - Inner loop with n iterations, outer loop with n/p iterations
  - Inner loop: a multiplication and an addition.
  - Outer loop: a multiplication and a subtraction before inner loop and a subtraction and division after inner loop.

\[
t_{\text{comp}} = \frac{n}{p}(2n + 4) \tau \quad \text{Time complexity } O(n^2/p)
\]

- Communication:
  - Occurs at the end of each iteration, multiple broadcasts.
  - p broadcasts each of size n/p require \( t_{\text{data}} \) to send each item

\[
t_{\text{comm}} = p(t_{\text{startup}} + \frac{n}{p}t_{\text{data}}) = (pt_{\text{startup}} + nt_{\text{data}}) \tau \quad O(n)
\]

- Overall Time:

\[
t_p = \frac{n}{p}(2n + 4) \tau + pt_{\text{startup}} + nt_{\text{data}} \tau
\]

C-to-C Ratio approximately = \( n/(n^2/p) = p/n \)
Effects of Computation And Communication in Jacobi Iteration

For one iteration:

\[ t_p = \frac{n}{p}(2n + 4) \tau + pt_{\text{startup}} + nt_{\text{data}} \]

**Given:** \( n = ? \) (value not provided in textbook)

\( t_{\text{startup}} = 10000 \quad t_{\text{data}} = 50 \)

integer \( n/p \)

Here, minimum execution time occurs when \( p = 16 \)

Parallel Programming book, Chapter 6 (page 180)
Other Synchronous Problems: Cellular Automata

- The problem space is divided into cells.
- Each cell can be in one of a finite number of states.
- Cells affected by their neighbors according to certain rules, and all cells are affected simultaneously in a “generation.”
  - Thus predictable, near-neighbor, data parallel computations within an iteration (suitable for partitioning using static domain decomposition).
- Rules re-applied in subsequent generations (iterations) so that cells evolve, or change state, from generation to generation.
- Most famous cellular automata is the “Game of Life” devised by John Horton Conway, a Cambridge mathematician.

\[ \begin{array}{ccc}
X_0 & X_1 & X_2 \\
X_3 & X_4 & X_5 \\
X_6 & X_7 & X_8 \\
\end{array} \]

Cell state update:
- near-neighbor
- data parallel computation
Conway’s Game of Life

• Board game - theoretically infinite two-dimensional array of cells.
• Each cell can hold one “organism” and has eight neighboring cells, including those diagonally adjacent. Initially, some cells occupied.
• The following rules apply:
  1 – Every organism with two or three neighboring organisms survives for the next generation.
  2 – Every organism with four or more neighbors dies from overpopulation.
  3 – Every organism with one neighbor or none dies from isolation.
  4 – Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.
• These rules were derived by Conway “after a long period of experimentation.”
“Serious” Applications for Cellular Automata

- Fluid/gas dynamics.
- The movement of fluids and gases around objects.
- Diffusion of gases.
- Biological growth.
- Airflow across an airplane wing.
- Erosion/movement of sand at a beach or riverbank.
- ........
Dynamic Load Balancing: Dynamic Tasking

• To achieve best performance of a parallel computing system running a parallel problem, it’s essential to maximize processor utilization by distributing the computation load evenly or balancing the load among the available processors while minimizing overheads.

• Optimal static load balancing, partitioning/mapping, is an intractable NP-complete problem, except for specific problems with regular and predictable parallelism on specific networks.
  – In such cases heuristics are usually used to select processors for processes (e.g. Domain decomposition) Covered in lecture 6

• Even the best static mapping may not offer the best execution time due to possibly changing conditions at runtime and the process mapping may need to done dynamically (depends on nature of parallel algorithm) (e.g. N-body, Ray tracing) i.e. predictability of computation

• The methods used for balancing the computational load dynamically among processors can be broadly classified as:

  1. Centralized dynamic load balancing.
  2. Decentralized dynamic load balancing.

Parallel Programming book, Chapter 7
Processor Load Balance & Performance

(a) Imperfect load balancing leading to increased execution time

(b) Perfect load balancing
Centralized Dynamic Load Balancing

One Task Queue (maintained by one master process/processor)

Advantage of centralized approach for computation termination:

- The master process terminates the computation when:
  1. The task queue is empty, and
  2. Every process has made a request for more tasks without any new tasks been generated.

Potential disadvantages (Due to centralized nature):

- High task queue management overheads/load on master process.
- Contention over access to single queue may lead to excessive contention delays.

In particular for a large number of tasks/processors and/or for small work unit size (task grain size)
Decentralized Dynamic Load Balancing

A Number of Distributed Task Queues

Distributed Work Pool Using Divide And Conquer

Advantage Over Centralized Task Queue (Due to distributed/decentralized nature):
- Less effective dynamic tasking overheads (multiple processors manage queues).
- Less contention and contention delays than access to one task queue.

Disadvantage compared to Centralized Task Queue:
- Harder to detect/determine parallel computation termination, requiring a termination detection algorithm.
Decentralized Dynamic Load Balancing
Distributed Work Pool With Local Queues In Slaves

Tasks could be transferred by one of two methods:

1. Receiver-initiated method.
2. Sender-initiated method.

Termination Conditions for Decentralized Dynamic Load Balancing:

In general, termination at time “t” requires two conditions to be satisfied:

1. Application-specific local termination conditions exist throughout the collection of processes, at time “t”, and
2. There are no messages in transit between processes at time “t”.

Disadvantage compared to Centralized Task Queue: Harder to detect/determine parallel computation termination, requiring a termination detection algorithm.
Termination Detection for Decentralized Dynamic Load Balancing

- Detection of parallel computation termination is harder when utilizing distributed tasks queues compared to using a centralized task queue, requiring a termination detection algorithm. One such algorithm is outlined below:

- **Ring Termination Algorithm:**
  - Processes organized in ring structure.
  - When $P_0$ terminated it generates a token to $P_1$.
  - When $P_i$ receives the token and has already terminated, it passes the token to $P_{i+1}$. $P_{n-1}$ passes the token to $P_0$.
  - When $P_0$ receives the token it knows that all processes in ring have terminated. A message can be sent to all processes (using broadcast) informing them of global termination if needed.

Token passed to next processor when reached local termination condition
Program Example: Shortest Path Algorithm

- Given a set of interconnected vertices or nodes where the links between nodes have associated weights or “distances”, find the path from one specific node to another specific node that has the smallest accumulated weights.
  
  i.e shortest path

- One instance of the above problem below:
  
  “Find the best way to climb a mountain given a terrain map.”

Source

Destination

Possible intermediate camps

Mountain Terrain Map

A = Source

F = Destination

Corresponding Directed Graph
Representation of Sample Problem Graph

Directed Problem Graph

Source = A
Destination = F

Non-symmetric for directed graphs

Adjacency Matrix

Directed Problem Graph Representations

Adjacency List
Moore’s Single-source Shortest-path Algorithm

- Starting with the source, the basic algorithm implemented when vertex i is being considered is as follows.
  - Find the distance to vertex j through vertex i and compare with the current distance directly to vertex j.
  - Change the minimum distance if the distance through vertex j is shorter. If \( d_i \) is the distance to vertex i, and \( w_{ij} \) is the weight of the link from vertex i to vertex j, we have:
    \[
    d_j = \min(d_j, d_i + w_{ij})
    \]

- The code could be of the form:
  
  ```
  newdist_j = dist[i]+w[i][j];
  if(newdist_j < dist[j])
    dist[j] = newdist_j;
  ```

- When a new distance is found to vertex j, vertex j is added to the queue (if not already in the queue), which will cause this vertex to be examined again.

Parallel Programming book, Chapter 7
Steps of Moore’s Algorithm for Example Graph

- **Stages in searching the graph:**
  - **Initial values**

  ![Vertex Queue and Distances](image)

  - **Source = A**
  - **Destination = F**

  Start with only source node to consider

  - **Each edge from vertex A is examined starting with B**

  The weight to vertex B is 10, which will provide the first (and actually the only distance) to vertex B. Both data structures, vertex_queue and dist[], are updated.

  ![Vertex Queue and Distances](image)

  - **Source = A**
  - **Destination = F**

  F is destination

  - **Once a new vertex, B, is placed in the vertex queue, the task of searching around vertex B begins.**

  The distances through vertex B to the vertices are:
  - \( \text{dist}[F] = 10 + 51 = 61 \)
  - \( \text{dist}[E] = 10 + 24 = 34 \)
  - \( \text{dist}[D] = 10 + 13 = 23 \)
  - \( \text{dist}[C] = 10 + 8 = 18 \)

  Since all were new distances, all the vertices are added to the queue (except F).

  - **Vertex F need not to be added because it is the destination with no outgoing edges and requires no processing.**

  - **Destination not added to vertex queue**

  E, D, E have lower Distances thus appended to vertex_queue to examine
Steps of Moore’s Algorithm for Example Graph

- Starting with vertex E:
  - It has one link to vertex F with the weight of 17, the distance to vertex F through vertex E is \( \text{dist}[E] + 17 = 34 + 17 = 51 \) which is less than the current distance to vertex F and replaces this distance.

- Next is vertex D:
  - There is one link to vertex E with the weight of 9 giving the distance to vertex E through vertex D of \( \text{dist}[D] + 9 = 23 + 9 = 32 \) which is less than the current distance to vertex E and replaces this distance.
  - Vertex E is added to the queue.
Steps of Moore’s Algorithm for Example Graph

• Next is vertex C:
  – We have one link to vertex D with the weight of 14.
  – Hence the (current) distance to vertex D through vertex C of \( \text{dist}[C] + 14 = 18 + 14 = 32 \). This is greater than the current distance to vertex D, \( \text{dist}[D] \), of 23, so 23 is left stored.

• Next is vertex E (again):
  – There is one link to vertex F with the weight of 17 giving the distance to vertex F through vertex E of \( \text{dist}[E] + 17 = 32 + 17 = 49 \) which is less than the current distance to vertex F and replaces this distance, as shown below:

There are no more vertices to consider and we have the minimum distance from vertex A to each of the other vertices, including the destination vertex, F.

Usually the actual path is also required in addition to the distance and the path needs to be stored as the distances are recorded.

The path in our case is ABDE F.
Moore’s Single-source Shortest-path Algorithm

Sequential Code:

- The specific details of maintaining the vertex queue are omitted.
- Let next_vertex() return the next vertex from the vertex queue or no_vertex if none, and let next_edge() return the next link around a vertex to be considered. (Either an adjacency matrix or an adjacency list would be used to implement next_edge()).

The sequential code could be of the form:

```c
while ((i=next_vertex())!=no_vertex)            /* while there is a vertex */
    while (j=next_edge(vertex)!=no_edge) {      /* get next edge around vertex */
        newdist_j=dist[i] + w[i][j];
        if (newdist_j < dist[j]) {
            dist[j]=newdist_j;
            append_queue(j);               /* add vertex to queue if not there */
        }
    }                                          /* no more vertices to consider */
/* no more vertices to consider */
```
Moore’s Single-source Shortest-path Algorithm
Parallel Implementation, Centralized Work Pool

The code could be of the form:

Master

recv(any, P_i);                          /* request for task from process P_i */
if ((i= next_edge()!= no_edge)
    send(P_i, i, dist[i]);               /* send next vertex, and */
    /* current distance to vertex */
    recv(P_j, j, dist[j]);                   /* receive new distances */
    /* append vertex to queue */
    append_queue(j);

Slave (process i)

send(P_master, P_i);                       /* send a request for task */
recv(P_master, i, d);                     /* get vertex number and distance */
while (j=next_edge(vertex)!= no_edge) {  /* get next link around vertex */
    newdist_j = d + w[i][j];
    if (newdist_j < dist[j]) {
        dist[j]=newdist_j;
        send(P_master, j, dist[j]);       /* send back updated distance */
    }
}
/* no more vertices to consider */
Moore’s Single-source Shortest-path Algorithm
Parallel Implementation, Decentralized Work Pool

The code could be of the form:

Master

```c
if ((i = next_vertex() != no_vertex)
    send(P_i, "start"); /* start up slave process i */

Slave (process i)

if (recv(P_j, msgtag = 1)) /* asking for distance */
    send(P_j, msgtag = 2, dist[i]); /* sending current distance */

if (nrecv(P_master) { /* if start-up message */
    while (j=next_edge(vertex) != no_edge) { /* get next link around vertex */
        newdist_j = dist[i] + w[j];
        send(P_j, msgtag=1); /* Give me the distance */
        recv(P_i, msgtag = 2, dist[j]); /* Thank you */
        if (newdist_j > dist[j]) {
            dist[j] = newdist_j;
            send(P_j, msgtag=3, dist[j]); /* send updated distance to proc. j */
        }
    }
}
```

where w[j] hold the weight for link from vertex i to vertex j.
Moore’s Single-source Shortest-path Algorithm

Distributed Graph Search

Master process

Start at source vertex

Vertex w[]

Process A

dist

New distance

Vertex w[]

Process C

dist

New distance

Other processes

Process B