Message-Passing Computing Examples

- Problems with a very large degree of parallelism:
  - Image Transformations:
    - Shifting, Rotation, Clipping etc.
  - Mandelbrot Set:
    - Sequential, static assignment, dynamic work pool assignment.
- Divide-and-conquer Problem Partitioning:
  - Parallel Bucket Sort
  - Numerical Integration:
    - Trapezoidal method using static assignment.
    - Adaptive Quadrature using dynamic assignment.
- Pipelined Computation
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-direction are given by:
    
    \[
    x' = x + \Delta x \quad \quad \quad y' = y + \Delta y
    \]

    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:
    
    \[
    x' = xS_x \quad \quad \quad y' = yS_y
    \]

    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:
    
    \[
    x' = x \cos \theta + y \sin \theta \quad \quad \quad y' = -x \sin \theta + y \cos \theta
    \]

- **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:
    
    \[
    x_1 \leq x \leq x_h \quad \quad \quad y_1 \leq y \leq y_h
    \]

    needs to be true for the point $(x, y)$ to be displayed, otherwise $(x, y)$ is not displayed.
Possible Static Image Partitionings

- Image size: 640x480:
  - To be copied into array: \( \text{map}[ ][ ] \) from image file
  - To be processed by 48 Processes or Tasks

(a) Square region for each process

(b) Row region for each process

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

Master

```c
for (i = 0; i < 8; i++)                         /* for each 48 processes */
    for (j = 0; j < 6; j++) {                         /* for each 48 processes */
        p = i*80;                        /* bit map starting coordinates */
        q = j*80;                        /* load coordinates into array x[],
y[]*/
        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)

```c
recv(Pmaster, c[0] ... c[6400]);
/* receive block of pixels to process */
for (i = 0; i < 6400; i += 2) {
  c[i] = c[i] + delta_x;
  /* shift in x direction */
  c[i+1] = c[i+1] + delta_y;
  /* shift in y direction */
}
send(Pmaster, c[0] ... c[6399]);
/* send transformed pixels to master */
```
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:

\[
t_{\text{comp}} = n^2/p
\]

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:

\[
t_{\text{comm}} = O(n^2)
\]

• So that the overall execution time is given by:

\[
t_p = t_{\text{comp}} + t_{\text{comm}} = O(n^2/p) + O(n^2)\]
Image Computing Example: Mandelbrot Set

- Displaying the Mandelbrot set is another example of processing bit-mapped image processing.
- In this case the image is computed and the computation time is significant.
- Required Computation:
  - The Mandelbrot set is a set of points in a complex plane that are "quasi-stable" (will increase and decrease but not exceed some limit) when computed by iterating a function, usually the function:
    \[ z_n = z_{n-1}^2 + c \]
  - Where \( z_n \) is the \( n \)th iteration of the complex number \( z = a + bi \), and \( z_{n-1} \) is the \((n-1)\)th value of \( z \), and \( c \) is a complex number giving the position of the point in the complex plane.
  - The initial value for \( z \) is zero. The iterations are continued until the value of \( z \) is greater than 2 (which indicates that \( z \) will eventually become infinite) or the number of iterations exceeds some arbitrary limit.
Image Computing Example: Mandelbrot Set

- The value of $z$ is the length of the vector, given by:

$$Z_{\text{length}} = \sqrt{a^2 + b^2}$$

- Computing the complex function, $z_n = z_{n-1}^2 + c$, is simplified by recognizing that if $z = a + bi$:

$$z^2 = a^2 + 2abi + (bi)^2 = a^2 - b^2 + 2abi$$

or a real part of $a^2 - b^2$ and an imaginary part of $2abi$.

- Hence if $\text{z\_real} = \text{the real part of } z$ and $\text{z\_imag} = \text{the imaginary part of } z$, next iteration values can be produced from computing:

$$z_{\text{real}}_n = z_{\text{real}}_{n-1}^2 - z_{\text{imag}}_{n-1}^2 + c_{\text{real}}$$

$$z_{\text{imag}}_n = 2z_{\text{real}}_{n-1} z_{\text{imag}}_{n-1} + c_{\text{imag}}$$
Mandelbrot Set: Sequential code

- The code for iterating one point as a function that returns the number of iterations, could be:

```c
int cal_pixel(float c_real, float c_imag)
{
    int i, max;
    float z_real, z_image, z_real_next, lengthsq;
    max = ???;
    z_real = 0;
    z_imag = 0;
    i = 0;                          /* number of iterations */
    DO {
        z_real_next = (z_real * z_real) - (z_imag * z_imag) + c_real;
        z_imag = (2 * z_real * z_imag) + c_imag;
        z_real = z_real_next;
        lengthsq = (z_real * z_real) + (z_imag * z_imag);
        i++;
    } WHILE (lengthsq < 4.0 AND i < max);
    return i;
}
```
Mandelbrot Set Using Static Assignment

- Grouping, by either square/rectangular regions or by columns/rows is suitable.
- Each process needs to execute the procedure `cal_pixel()`, after being given the coordinates of the pixels to compute.
- Suppose we use the same grouping of 80 x 80 pixels and 48 processes, the code might look like the following:

**Master**

```c
for (i = 0; i < 8; i++) /* for each 48 processes */
for (j = 0; j < 6; j++) {
    x = i*80;               /* bit map starting coordinates */
    y = j*80;
    z = j + 8*i;            /* process number */
    send(Pz, x, y);         /* send starting coordinates to slave*/
}
for (i = 0; i < (640 * 480); i++) { /* from processes, any order */
    recv(ANY, x, y, color); /* receive coordinates/colors */
    display(x, y, color);   /* display pixel on screen */
}
```

**Slave (process i)**

```c
recv(Pmaster, x, y); /* receive starting coordinates */
for (i = x; i < (x + 80); i++)
for (j = y; j < (y + 80); j++) {
    a = -2 + (x * 3.25 / 640); /* scaling factors */
    b = -1.25 + (y * 2.5 / 480);
    color = cal_pixel(a, b);   /* calculate color */
    send(Pmaster, i, j, color); /* send coordinates, color to master */
```
Dynamic Assignment - Work Pool/Processor Farms

Request tasks → Work pool

Return results

\[(x_a, y_a), (x_b, y_b), (x_c, y_c), (x_d, y_d)\]

\[ (x_e, y_e)\]

\[ (x_f, y_f)\]

Figure 3.5 Work pool approach
Mandelbrot Set Using Dynamic Work Pool Assignment

- The message-passing pseudocode for Mandelbrot Set computed using a work pool dynamic assignment could be of the form:

**Master**

```c
for (k = 0; k < nproc; k++) { /* send coordinates to processes */
    x = k * 80;
    y = k * 80;
    send(Pk, data_tag, x, y);
}
unsent_blocks = nproc;
while (unsent_blocks < BLOCKS) {
    recv (Pi, ANY, result_tag, x, y, color);
    display (x, y, color); /* display pixel on screen */
    x = unsent_blocks * 80;
    y = unsent_blocks * 80;
    send(Pi, data_tag, x, y); /* send more coordinates */
    unsent_blocks++;
}
for (last = 0; last < nproc; last++) { /* receive final results */
    recv(Pi, ANY, result_tag, x, y, color);
    display(x, y, color);
    send(Pi, ANY, terminator_tag);
}
```
Mandelbrot Set Using Dynamic Work Pool Assignment

Slave (process i)

while (!nrecv(Pmaster, terminator_tag)) {
    /* while more pixels to take */
    recv(Pmaster, data_tag, x, y);  /* receive starting */
    /* coordinates */
    for (i = x; i < (x + 80); i++)
        for (j = y; j < (y + 80); j++) {
            a = -2 + (x * 3.25 / 640);  /* scaling factors */
            b = -1.25 + (y * 2.5 / 480);
            color = cal_pixel(a, b);  /* calculate color */
            send(Pmaster, result_tag, i, j, color);
            /* send coordinates, color to master */
        }
}
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 Divide and conquer: A task is divided into M parts at each stage of the divide phase (a tree node has M children).
Divide-and-Conquer Example

Bucket Sort

- On a sequential computer, it requires $n$ steps to place the $n$ numbers into $m$ buckets (by dividing each number by $m$).
- If the numbers are uniformly distributed, there should be about $n/m$ 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 $(n/m) \log_2 (n/m)$ steps to sort the $n/m$ numbers in each bucket using a sequential sorting algorithm such as Quicksort or Mergesort, leading to sequential time of:
  $$t_s = n + m((n/m) \log_2 (n/m)) = n + n \log_2 (n/m) = O(n \log_2 (n/m))$$
- If $n = km$ where $k$ is a constant, we get a linear complexity of $O(n)$. 

Sequential Bucket Sort

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 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.
  - 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

Sorted numbers

Figure 4.6 Parallel version of bucket sort
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:

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

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 these formulae.

• If the numbers are not uniformly distributed, some buckets would have more numbers than others and sorting them would dominate the overall computation time.

• 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. \( t_{comp1} = n \)
  - Communication time using a broadcast or scatter:
    \( t_{comm1} = t_{startup} + t_{data}n \)

- **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: \( t_{comp2} = n/p \)

- **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()
    \( t_{comm3} = p(1-p)(t_{startup} + (n/p^2) t_{data}) \)
    - Communication time from different processes fully overlap:
      \( t_{comm3} = (1-p)(t_{startup} + (n/p^2) t_{data}) \)

- **Phase 4**: Sorting large buckets in parallel. No communication.
  - Each bucket contains \( n/p \) numbers
    \( t_{comp4} = (n/p) \log(n/P) \)

**Overall time:** \( t_p = t_{startup} + t_{data}n + n/p + (1-p)(t_{startup} + (n/p^2) t_{data}) + (n/p) \log(n/P) \)
Numerical Integration
Using Rectangles
More Accurate Numerical Integration Using Rectangles

\[ f(x) \]

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

\[ a \quad p \quad q \quad b \quad x \]
Numerical Integration
Using The Trapezoidal Method

Each region is calculated as
\[
\frac{1}{2}(f(p) + f(q)) \delta
\]
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, Pgroup); /* 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 */
Numerical Integration Using The Trapezoidal Method: Static Assignment Message-Passing

• We can simplify the calculation somewhat by algebraic manipulation as follows:

\[
Area = \frac{\delta(f(\alpha) + f(\alpha + \delta))}{2} + \frac{\delta(f(\alpha + \delta) + f(\alpha + 2\delta))}{2} + \cdots + \frac{\delta(f(\alpha + n\delta) + f(b))}{2} = \\
= \delta \left( \frac{f(\alpha)}{2} + f(\alpha + \delta) + f(2\alpha + \delta) + \cdots + f(\alpha + n\delta) + \frac{f(b)}{2} \right)
\]

so that the inner summation can be formed and then multiplied by the interval.

• One implementation would be to use this formula for the region handled by each process:

\[
area = 0.5 \times (f(start) + f(end)); \\
for \ (x = start + d; x < end; x = x + d) \\
area = area + f(x); \\
area = area \times d
\]
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 the area computed for the largest of A or B is close to the sum of the other two areas, or 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.
Reducing the size of $\delta$ is stopped when the area computed for the largest of $A$ or $B$ is close to the sum of the other two areas, or when $C$ is small.
Gravitational N-Body Problem

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

\[ F = \frac{(Gm_a m_b)}{r^2} \quad \text{F} = \text{mass} \times \text{acceleration} \]

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

For computer simulation:

\[ F = m \frac{(v^{t+1} - v^t)}{\Delta t} \]

\[ v^{t+1} = v^t + \frac{F \Delta t}{m} \]

\[ x^{t+1} - x^t = v^{t+1/2} \Delta t \]

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
}
```
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 $d \times d \times d$:
    \[
    r \geq d/\theta
    \]
    \(r\) = distance to the center of mass
    \(\theta\) = 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

- Compute forces
- Update properties
- Build tree
- Compute moments of cells
- Traverse tree to compute forces
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.
Pipelined Computations

• Given the problem can be divided into a series of sequential operations, the pipelined approach can provide increase speed under any of the following three "types" of computations:

1. If more than one instance of the complete problem is to be executed.

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.
Pipelined Computations

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

Pipeline for a frequency filter
Pipelined Computations

Pipeline Space-Time Diagram

Pipelining allows for concurrent execution of multiple instances of a computation on different processors. The diagram shows a space-time representation of pipelining, where each processor (P_0 to P_5) is involved in the computation process, and the instances are spread across time to maximize efficiency.

Time progression is indicated from left to right, with each processor handling a portion of the computation. The instances are labeled sequentially to illustrate the flow and concurrency within the pipeline.
Pipelined Computations

Alternate Pipeline Space-Time Diagram
Pipeline Processing Where Information Passes To Next Stage Before End of Process

Partitioning pipelines processes onto processors
Pipelined Addition

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

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

  $\text{accumulation} += \text{number};$

  $\text{send}(P_{i+1}, \text{accumulation});$
Pipelined Addition: Analysis

- \( t_{\text{total}} = \) pipeline cycle x number of cycles
  \[ = (t_{\text{comp}} + t_{\text{comm}})(m + p - 1) \]
  for \( m \) instances and \( p \) pipeline stages

- For single instance:
  \[ t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}})+1)n \]
  Time complexity \( O(n) \)

- For \( m \) instances of \( n \) numbers:
  \[ t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}})+1)(m+n-1) \]
  For large \( m \), average execution time \( t_a \):
  \[ t_a = t_{\text{total}}/m = 2(t_{\text{startup}} + t_{\text{data}})+1 \]

- For partitioned multiple instances:
  \[ t_{\text{comp}} = d \]
  \[ t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}}) \]
  \[ t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + d)(m + n/d - 1) \]
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:

  ```
  recv($P_{i-1}$, number);
  IF (number > $x$) {
    send($P_{i+1}$, $x$);
    $x$ = number;
  }
  ELSE send($P_{i+1}$, number);
  ```

![Diagram of the algorithm](image)
Pipelined Insertion Sort

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

recv(P_{i-1}, x);
for (j = 0; j < (n-i); j++) {
    recv(P_{i-1}, number);
    IF (number > x) {
        send(P_{i+1}, x);
        x = number;
    } ELSE send(P_{i+1}, number);
}
Pipelined Insertion Sort Example
Pipelin ed Insertion Sort: Analysis

• Sequential implementation:
\[ t_s = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]

• 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 ( )
    • \( t_{comp} = 1 \quad t_{comm} = 2(t_{startup} + t_{data}) \)
    • \( t_{total} = \text{cycle time} \times \text{number of cycles} \)
      \[ = (1 + 2(t_{startup} + t_{data}))(2n - 1) \]
Pipelined Insertion Sort

Master process

$\begin{array}{c} d_{n-1} \ldots d_2 d_1 d_0 \\ \text{Sorted sequence} \end{array}$

Sorting phase

$2n - 1$

Returning sorted numbers

$\begin{array}{c} P_4 \\ P_3 \\ P_2 \\ P_1 \\ P_0 \end{array}$

Shown for $n = 5$

Time
Solving A Set of Upper-Triangular Linear Equations

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

\[ x_1 = \frac{b_1}{a_{11}} \]
\[ x_2 = \frac{b_2 - a_{21}x_1}{a_{22}} \]
\[ x_3 = \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \]

\[ x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}} \]
Sequential Code:

- Given the constants a and b are stored in arrays and the value for unknowns also to be stored in an array, the sequential code could be:

```c
for (i = 1; i <= n; i++) {
    sum = 0;
    for (j = 1; j < i; j++) {
        sum += a[i][j]*x[j];
        x[i] = (b[i] - sum)/a[i][j];
    }
}
```
Solving A Set of Upper-Triangular Linear Equations

Parallel Code:

• The pseudo code of process $P_i$ of the pipelined version could be:

\[
\text{for } (j = 1; j < i; j++) \{ \\
\text{recv}(P_{i-1}, x[j]); \\
\text{send}(P_{i+1}, x[j]); \\
\}
\]

\[
\text{sum} = 0; \\
\text{for } (j = 1; j < i; j++) \\
\text{sum} += a[i][j]*x[j]; \\
x[j] = (b[i] - \text{sum})/a[i][j]; \\
\text{send}(P_{i+1}, x[j]);
\]
Solving A Set of Upper-Triangular Linear Equations

Pipeline processing using back substitution

First value passed onward

Final computed value

 Processes

P0
P1
P2
P3
P4
P5
Solving A Set of Upper-Triangular Linear Equations: Analysis

Communication:
• Each process in the pipelined version performs $i \text{ rec}(\ )s$, $i + 1 \text{ send}(\ )s$, 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 effects of the parallel version.
• Lester quotes a value of $0.37n$ for his simulator but it would depend heavily on the actual system parameters.
## 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($x_0$) $\Rightarrow$</td>
<td>send($x_0$) $\Rightarrow$</td>
<td>send($x_0$) $\Rightarrow$</td>
<td>send($x_0$) $\Rightarrow$</td>
<td>send($x_0$) $\Rightarrow$</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($x_1$) $\Rightarrow$</td>
<td>multiply/add</td>
<td>send($x_1$) $\Rightarrow$</td>
<td>send($x_1$) $\Rightarrow$</td>
</tr>
<tr>
<td>send($x_2$) $\Rightarrow$</td>
<td>recv($x_1$)</td>
<td>divide/subtract</td>
<td>recv($x_2$)</td>
<td>multiply/add</td>
</tr>
<tr>
<td>end</td>
<td>multiply/add</td>
<td>send($x_1$) $\Rightarrow$</td>
<td>send($x_2$) $\Rightarrow$</td>
<td>send($x_2$) $\Rightarrow$</td>
</tr>
<tr>
<td>divide/subtract</td>
<td>recv($x_1$)</td>
<td>multiply/add</td>
<td>recv($x_3$)</td>
<td>multiply/add</td>
</tr>
<tr>
<td>send($x_3$) $\Rightarrow$</td>
<td>multiply/add</td>
<td>send($x_4$) $\Rightarrow$</td>
<td>send($x_3$) $\Rightarrow$</td>
<td>send($x_3$) $\Rightarrow$</td>
</tr>
<tr>
<td>end</td>
<td>divide/subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>