In (fully) synchronous applications, all the processes are synchronized at regular points.

### Synchronous Computations

In message-passing systems, barriers are often provided with library routines:

- **MPI**:
  - Barrier routine: `MPI_Barrier()`, with a named communicator being the only parameter.
  - All processes in the group are synchronized by calling `MPI_Barrier()` with a named communicator.

- **PVM**:
  - Barrier routine: `pvm_barrier()`, which is called by each process in the group, blocking until all members of the group have reached the barrier call and only returning then.

A barrier is a basic mechanism for synchronizing processes - inserted at various points in the application to ensure that all processes are synchronized at regular points.
Barrier using a centralized counter.

Barrier();
P1 Barrier();
Pn-1 Barrier();

Counter, C
Increment and check for

Implementation
Centralized counter implementation (sometimes called a linear barrier):

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);

Slave processes:
send(Pmaster);
recv(Pmaster);

Master:
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);

Example code:

// Example code:

// Under the assumption that no process is waiting in the barrier for the entire time, the two-phase strategy

// Above processes move to departure phase and are released.

// A process that enters the phase, does not leave the phase until all processes
// Counter-based barriers often have two phases:

Counter-based barriers often have two phases:

Implementation
Arrival at barrier
Departure from barrier

Figure 6.5 Tree barrier.

Tree implementation:
More efficient. Suppose there are eight processes, \( P_0, P_1, P_2, P_3, P_4, P_5, P_6, \) and \( P_7 \):

**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 \( P_0 \); (\( P_2 \) and \( P_3 \) have reached their barrier)
- \( P_6 \) sends message to \( P_4 \); (\( P_6 \) and \( P_7 \) have reached their barrier)

**Third stage:**
- \( P_4 \) sends message to \( P_0 \); (\( P_4, P_5, P_6, \) and \( P_7 \) have reached their barrier)
- \( P_0 \) terminates arrival phase; (when \( P_0 \) reaches barrier and has received message from \( P_4 \))

Release with a reverse tree construction.

**Butterfly Barrier**

The tree construction can be developed into a so-called butterfly, in which pairs of processes synchronize at each stage:

**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**
- \( P_0 \) \( \leftrightarrow \) \( P_4, P_1 \) \( \leftrightarrow \) \( P_5, P_2 \) \( \leftrightarrow \) \( P_6, P_3 \) \( \leftrightarrow \) \( P_7 \)

**Butterfly Construction**

The tree construction can be developed into a so-called butterfly, in which pairs of processes synchronize at each stage:

**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**
- \( P_0 \) \( \leftrightarrow \) \( P_4, P_1 \) \( \leftrightarrow \) \( P_5, P_2 \) \( \leftrightarrow \) \( P_6, P_3 \) \( \leftrightarrow \) \( P_7 \)

Release with a reverse tree construction.
Local Synchronization

Example

Process $P_i$ needs to be synchronized and to exchange data with process $P_{i-1}$ and $P_{i+1}$ before continuing:

Not a perfect three-process barrier because process $P_{i-1}$ will only synchronize with $P_i$ and continue as soon as $P_i$ allows. Similarly, process $P_{i+1}$ only synchronizes with $P_i$.

```
recv($P_i$); send($P_{i-1}$); recv($P_i$);
send($P_i$); send($P_{i+1}$); send($P_i$);
recv($P_{i-1}$);
recv($P_{i+1}$);
```

Deadlock

When a pair of processes each send and receive from each other, deadlock may occur. Deadlock will occur if both processes perform the same primitive (e.g., `send`) before performing the other primitive (e.g., `recv`).

A solution:

- Process $P_i$ sends first and then receives.
- Process $P_{i-1}$ receives first and then sends.

Combined deadlock-free blocking `sendrecv()` routines

MPI provides routines `MPI_Sendrecv()`, `MPI_Sendrecv_replace()`, and `MPI_Ssendrecv()`.

Example

Linear pipeline, deadlock can be avoided by arranging in the even-numbered processes to send first and the odd-numbered processes to receive first.

```
sendrecv($P_i$); sendrecv($P_{i-1}$);
sendrecv($P_{i+1}$); sendrecv($P_i$);
```

Example

Process $P_{i-1}$ needs to be synchronized with $P_i$ and $P_{i+1}$ (or $P_i$ with $P_{i+1}$ and $P_{i-1}$). Suppose a process $P_i$ needs to be synchronized with $P_{i+1}$ and $P_{i-1}$ before continuing:

```
recv($P_i$); send($P_{i+1}$); recv($P_i$);
send($P_i$); send($P_{i-1}$); send($P_i$);
recv($P_{i+1}$);
recv($P_{i-1}$);
```
In a data parallel computation, the same operation is performed on different data elements simultaneously; i.e., in parallel. Particularly convenient because:

- Ease of programming (essentially only one program).
- Can scale easily to larger problem sizes.
- Many numeric and some non-numeric problems can be cast in a data parallel form.

**Example of a data parallel computation**

To add the same constant to each element of an array:

```c
for (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

The statement `a[i] = a[i] + k` could be executed simultaneously by multiple processors, each using a different index `i` (0 ≤ i ≤ n).

**Forall construct**

A special "parallel" construct in parallel programming languages to specify data parallel operations. Example:

```c
forall (i = 0; i < n; i++) {
    body
}
```

states that `n` instances of the statements of the body can be executed simultaneously. One value of the loop variable `i` is valid in each instance of the body, the first instance has `i` = 0, the next `i` = 1, and so on.

**Example**

To add `k` to each element of an array:

```c
forall (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

Data parallel technique applied to multiprocessors and multicomputers. Example:

```c
i = myrank;
forall (i = 0; i < n; i++)
    a[i] = a[i] + k; /* body */
barrier(mygroup);
```

where `myrank` is a process rank between 0 and `n` - 1.
Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
© Prentice Hall, 1998

Prefix Sum Problem

Given a list of numbers, \( x_0, \ldots, x_{n-1} \), compute all the partial summations (i.e., \( x_0 + x_1; x_0 + x_1 + x_2; \ldots \)).

The prefix calculation can also be defined with associative operations other than addition; for example, subtraction, multiplication, maximum, minimum, and logical (Boolean) operations (AND, OR, exclusive OR, etc.).

Widely studied in connection with various computational models. Practical applications in areas such as processor allocation, data compaction, sorting, and polynomial evaluation.

The sequential code for the prefix problem could be

```c
for (i = 0; i < n; i++) {
    sum[i] = 0;
    for (j = 0; j <= i; j++)
        sum[i] = sum[i] + x[j];
}
```

This is an \( O(n^2) \) algorithm.

Figure 6.8

Data parallel prefix sum operation.

Sequential code might be written as

```c
// a = b + c + d + e;
// a = b + c;
// a = b;
```

Parallel code:

```c
// a = b + c + d + e;
// a = b + c;
// a = b;
```

Given a list of numbers, \( x_0, x_1, \ldots \): Compute the partial summations (i.e., \( x_0 + x_1; x_0 + x_1 + x_2; \ldots \)).

Prefix Sum Problem
Synchronous Iteration

The term synchronous iteration or synchronous parallelism is used to describe solving a problem by iteration where each iteration is composed of several processes that start together at the beginning of each iteration and the next iteration cannot begin until all processes have finished the previous iteration.

The forall construct could be used to specify the parallel bodies of the synchronous iteration:

```c
for (j = 0; j < n; j++) /* for each synchronous iteration */
forall (i = 0; i < N; i++) { /* N processes each executing */
  body(i); /* body using specific value of i */
}
```

In our case:

```c
for (j = 0; j < n; j++) { /* for each synchronous iteration */
  i = myrank; /* find value of i to be used */
  body(i); /* body using specific value of i */
  barrier(mygroup);
}
```

Solving a System of Linear Equations by Iteration

Suppose the equations are of a general form with \( n \) equations and \( n \) unknowns \( x_0, x_1, x_2, \ldots, x_{n-1} \) (\( 0 \leq i < n \)). One way to solve these equations for the unknowns is by iteration. By rearranging the \( i \)th equation:

\[
-x_i = \sum_{j=0}^{i-1} a_{ij} x_j - a_{ii} \sum_{j=i+1}^{n-1} b_j - b_i
\]

or

\[
-x_i = \sum_{j=0}^{i-1} a_{ij} x_j - a_{ii} x_i + \sum_{j=i+1}^{n-1} b_j - b_i
\]

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.

The iterative method described is called a Jacobi iteration – all values of \( x \) are updated together.
A simple common approach is to compare values computed in each iteration to their differences. If all $|x_i^{(t)} - x_i^{(t-1)}| < \epsilon$ for all $i$, then the iteration is stopped.

Sequential Code:

```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++) { /* for each unknown */
        sum = 0;
        for (j = 0; j < n; j++) /* compute summation of a[i][j]x[j] */
            if (i != j) sum = sum + a[i][j] * x[j];
        new_x[i] = (b[i] - sum) / a[i][i]; /* compute unknown */
    }
    for (i = 0; i < n; i++)
        x[i] = new_x[i]; /* update values */
}
```
Parallel Code

The broadcast routine, `broadcast_receive()`, sends the newly computed value of \( x[i] \) from process \( i \) to every other process and collects data broadcast from the other processes. An alternative simple solution is to return to basic `send()`s and `recv()`s for `broadcast_receive()`; i.e., process \( i \) might have

```c
for (j = 0; j < n; j++) if (i != j) send(&x[i], Pj);
for (j = 0; j < n; j++) if (i != j) recv(&x[j], Pj);
```

Figure 6.10

Allgather operation.

Broadcast and gather values in one composite construction - Allgather
Typically, we want to Iterate until the approximations are sufficiently close, rather than for a fixed number of times (which may not provide a sufficiently accurate solution). Each process could check its own computed value with, say,

```c
x[i] = b[i]; /*initialize unknown*/
iteration = 0;
do {
    iteration++;
    sum = -a[i][i] * x[i];
    for (j = 1; j < n; j++)  /* compute summation */
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i]; /* compute unknown */
    broadcast_receive(&new_x[i]);  /* broadcast value and wait */
} while (tolerance() && (iteration < limit));
```

where `tolerance()` returns `FALSE` if ready to terminate; otherwise it returns `TRUE`. We need to decide how to distribute the work among the processors. The goal is to partition the problem so that each processor is allocated a fraction of the total work.

**Partitioning**

Two common partitioning techniques are **block allocation** and **cyclic allocation**.

- **Block Allocation**: Allocated `n/p` unknowns to `p` processors in simple increasing order; i.e., with `p` processors and `n` unknowns.

- **Cyclic Allocation**: Processors are allocated one unknown in order; i.e., processor `P_0` is allocated `x_0`, `x_{p}`, `x_2`, ... , `x_{(n/p)-1}` , processor `P_1` is allocated `x_1`, `x_{p+1}`, `x_2` , ... , `x_{(n/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).

In general, it is not easy to formulate a simple rule for partitioning the work among the processors. Each process must check its own computed value to determine if the approximations are sufficiently close.
Figure 6.11: Effects of computation and communication in Jacobi iteration.

Overall
Communication

Analysis
Figure 6.13
Natural ordering of heat distribution problem.

Actually, we are solving a system of linear equations. Each point is an unknown dependent upon a few other unknowns, rather than all the other unknowns in the general case.

To clarify this relationship, consider the array of points as numbered in so-called 
*natural order*, starting at zero at the top left corner and in rows of one point:

The points are numbered from 1 for convenience and include those representing the edges. Each point will then use the equation

\[
\begin{align*}
&x_{i-1} - 4x_i + x_{i+1} = 0 \\
&x_{i+k} - 4x_i + x_{i-k} = 0
\end{align*}
\]

This could be written as a linear equation containing the unknowns \(x_i \pm k\), \(x_i \pm 1\), and \(x_i\):

\[
\frac{\partial^2 u}{\partial x^2} = 0
\]

We are also solving Laplace’s equation.

Sequential Code

```c
while (continue == TRUE) {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];
    continue = FALSE; /* indicates whether to continue */
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            if (!converged(i,j)) {
                continue = TRUE;
                break;
            }
} while (continue == TRUE);
```

To stop at some precision:

```c
while (continue == TRUE) {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + ... h[i][j+1]);
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];
    continue = FALSE; /* indicates whether to continue */
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            if (!converged(i,j))
                continue = TRUE;
} while (continue == TRUE);
```
Parallel Code

Version with a fixed number of iterations, process $P_i,j$ (except for the boundary points):

```c
for (iteration = 0; iteration < limit; iteration++) {
    g = 0.25 * (w + x + y + z);
    send(&g, Pi-1,j); /* non-blocking sends */
    send(&g, Pi+1,j);
    send(&g, Pi,j-1);
    send(&g, Pi,j+1);
    recv(&w, Pi-1,j); /* synchronous receives */
    recv(&x, Pi+1,j);
    recv(&y, Pi,j-1);
    recv(&z, Pi,j+1);
}
```

after suitable initialization of $w$, $x$, $y$, and $z$.

Each process has its own iteration loop. The number of iterations must be sent to each process.

It is important to use `send()`s that do not block while waiting for the `recv()`s; otherwise, the processes would deadlock, each waiting for a `recv()` before moving on.

The `recv()`s must be synchronous and wait for the `send()`s. Each process will be synchronized with its four neighbors by the `recv()`s.

Local barrier

![Diagram showing message passing for heat distribution problem.](image-url)
Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers

Barry Wilkinson and Michael Allen

Prentice Hall, 1998

Version where processes stop when they reach their required precision:

\[
\text{iteration} = 0; \\
\text{do} \quad \text{\{}
\text{iteration}++; \\
g = 0.25 \times (w + x + y + z); \\
\text{send}(&g, P_{i-1}, j); \quad \text{/* locally blocking sends */} \\
\text{send}(&g, P_{i+1}, j); \\
\text{send}(&g, P_i, j-1); \\
\text{send}(&g, P_i, j+1); \\
\text{recv}(&w, P_{i-1}, j); \quad \text{/* locally blocking receives */} \\
\text{recv}(&x, P_{i+1}, j); \\
\text{recv}(&y, P_i, j-1); \\
\text{recv}(&z, P_i, j+1); \\
\text{\} while((!\text{converged}(i, j)) \text{ or (iteration < limit))}; \\
\text{send}(&g, \&i, \&j, \&iteration, P_{\text{master}}); \\
\]

To handle the processes operating at the edges, we could use the process ID to determine the location of the process in the array, leading to code such as

\[
\text{if (last \_\_row)} \quad w = \text{bottom \_\_value}; \\
\text{if (first \_\_row)} \quad x = \text{top \_\_value}; \\
\text{if (first \_\_column)} \quad y = \text{left \_\_value}; \\
\text{if (last \_\_column)} \quad z = \text{right \_\_value}; \\
\text{version where processes stop when they reach their required precision:}
\]

\[
\text{iteration} = 0; \\
\text{do} \quad \text{\{}
\text{iteration}++; \\
g = 0.25 \times (w + x + y + z); \\
\text{if !(first \_\_row)} \quad \text{send}(&g, P_{i-1}, j); \\
\text{if !(last \_\_row)} \quad \text{send}(&g, P_{i+1}, j); \\
\text{if !(first \_\_column)} \quad \text{send}(&g, P_i, j-1); \\
\text{if !(last \_\_column)} \quad \text{send}(&g, P_i, j+1); \\
\text{if !(last \_\_row)} \quad \text{recv}(&w, P_{i-1}, j); \\
\text{if !(first \_\_row)} \quad \text{recv}(&x, P_{i+1}, j); \\
\text{if !(first \_\_column)} \quad \text{recv}(&y, P_i, j-1); \\
\text{if !(last \_\_column)} \quad \text{recv}(&z, P_i, j+1); \\
\text{\} while((!\text{converged}) \text{ or (iteration < limit))}; \\
\text{send}(&g, \&i, \&j, \&iteration, P_{\text{master}}); \\
\]

\[\text{Figure 6.15}
\text{Partitioning heat distribution problem.}\]

\[\text{Blocks (rows)}\]

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>A_2</td>
<td>A_3</td>
<td>\ldots</td>
</tr>
<tr>
<td>A_2</td>
<td>A_3</td>
<td>A_4</td>
<td>\ldots</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\ddots</td>
</tr>
</tbody>
</table>

\[\text{Strips (columns)}\]

0_{-1} P_0 \quad P_1 \quad P_2 \quad \ldots

0_{-1} A_1 \quad A_2 \quad A_3 \quad \ldots

0_{-1} A_2 \quad A_3 \quad A_4 \quad \ldots

0_{-1} \vdots | \vdots | \vdots | \ddots

0_{-1} \text{Blocks (rows)}

\text{To handle the processes operating at the edges, we could use the process ID to determine the location of the processes in the array, leading to code such as...}
Square blocks

**Figure 6.16**

Communication consequences of partitioning.

- **Block partition:**
  - Four edges where data points are exchanged. Communication time is given by
  - The equation is only valid for \( p \geq 9 \) when at least one block has four communicating edges.

- **Strip partition:**
  - Two edges where data points are exchanged. Communication time is given by
  - In general, the strip partition is better for \( \frac{t_{data}}{t_{startup}} > \frac{1}{2} \).

---

**Figure 6.17**

Startup times for block and strip partitions.

- In general, the strip partition is best for a large startup time, and a block partition is best for a small startup time. With the previous equations, the block partition has a larger communication time if
  - \( \frac{t_{startup}}{n} + \frac{t_{data}}{np} \) or
  - \( \left( \frac{t_{startup}}{n} + \frac{t_{data}}{np} \right) < \left( \frac{t_{startup}}{n} + \frac{t_{data}}{np} \right) \)

Optimum

Two edges where data points are exchanged. Communication time is given by

**Strip partition:**

The equation is only valid for \( \frac{d}{n} \geq 2 \) when an edge and blocks have noncommunicating edges.

**Block partition:**

Four edges where data points are exchanged. Communication time is given by...
Convenient to arrange a additional row of points at each edge, called 'ghost points', that hold the values from the adjacent edge. Each array of points is increased to accommodate the ghost rows.

A way of making the code safe is to alter the order of the send()s and recv()s, as follows:

```c
if ((myid % 2) == 0) { /* even processes */
    send(&g[m][1], &m, Pi-1);
    recv(&h[m][1], &m, Pi-1);
    send(&g[m+1][1], &m, Pi+1);
    recv(&h[m][m+1], &m, Pi+1);
} else { /* odd numbered processes */
    recv(&h[m][1], &m, Pi-1);
    send(&g[m][1], &m, Pi-1);
    recv(&h[m][m+1], &m, Pi+1);
    send(&g[m+1][1], &m, Pi+1);
}
```

This method can be seen without modification when the sending process sends (and the receiving process receives) the corresponding message. Since the sending process sends the same message as the receiving process receives, the deadlock can be avoided because each process sends a message to the process that it has received. In this way, the deadlock problem is not prevented. The solution is to exchange the order of the send()s and recv()s, as shown in the code above. This method is called the 'safety and deadlock avoidance' method.
MPI Safe message Passing Routines

MPI offers several alternative methods for safe communication:

- Combined send and receive routines: MPI_Sendrecv()
  - Guaranteed not to deadlock

- Buffered send()s: MPI_Bsend()
  - User provides explicit storage space

- Nonblocking routines: MPI_Isend() and MPI_Irecv()
  - Return immediately, separate routine used to establish whether the message has been received (MPI_Wait(), MPI_Waitall(), MPI_Waitany(), MPI_Test(), MPI_Testall(), MPI_Testany())

A pseudocode segment using the third method is:

```plaintext
isend(&g[1][1], &m, Pi-1);
isend(&g[1,m], &m, Pi+1);
irecv(&h[1][0], &m, Pi-1);
irecv(&h[1][m+1], &m, Pi+1);
waitall(4);
```

Essentially, the wait routine becomes a barrier, waiting for all the message-passing routines to complete.

Cellular Automata

In this approach, the problem space is first divided into a two-dimensional array of cells. Each cell can hold one of a finite number of states. The arrangement of a neighborhood of adjacent cells, usually a square, affects the state of the cell.

The Game of Life

The Game of Life, conceived by John Horton Conway, and published by Gardner (Gardner, 1967), is a two-dimensional cellular automaton. Each cell in the grid can be in one of two states: dead or alive. Initially, the grid is partially filled with living cells.

The rules of the Game of Life are:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The Game of Life is a classic example of a cellular automaton.
"Sharks and Fishes" in the sea, each with different behavior rules.

An ocean could be modeled as a three-dimensional array of cells.

Fish might move around according to these rules:
1. If there is one empty adjacent cell, the fish moves to this cell.
2. If there is more than one empty adjacent cell, the fish moves to one cell chosen at random.
3. If all adjacent cells are occupied, the fish stays where it is.
4. If the fish moves and has reached its breeding age, it gives birth to a baby fish, which is left in the vacating cell.
5. Fish die after $x$ generations.

The sharks might be governed by the following rules:
1. If one adjacent cell is occupied by a fish, the shark moves to this cell and eats the fish.
2. If more than one adjacent cell is occupied by fish, the shark chooses one fish at random and moves to the cell occupied by the fish, and eats the fish.
3. If no fish are in adjacent cells, the shark chooses an unoccupied adjacent cell to move to in a similar manner as fish move.
4. If the shark moves and has reached its breeding age, it gives birth to a baby shark, which is left in the vacating cell.
5. If a shark has not eaten for $y$ generations, it dies.

Similar examples: “foxes and rabbits” - The behavior of the rabbits is to move around happily whereas the behavior of the foxes is to eat any rabbits they come across.

Serious Applications for Cellular Automata

Examples - fluid dynamics, the movement of fluids and gases around objects, or diffusion of gases, biological growth, airflow across an airplane wing, erosion/movement of sand at a beach or oyster bed.

PROBLEMS

Scientific/Numerical

6-1. Implement the counter barrier described in Figure 6.4, and test it. Is it necessary to use blocking or synchronous routines for both send and receive? Explain.

6-2. Write barrier(procno), which will block (if necessary) and then release the processes. Allow for the barrier to be called with different numbers of processes and with different values for procno.

6-3. Investigate the time that a barrier takes to operate by using code such as:

6-4. Write code to implement an equivalent barrier routine described in Section 6.1.4, and compare with any available barrier calls.

6-5. Implement the butterfly barrier described in Section 6.1.4, and compare with any available barrier calls.

6-6. Determine experimentally at what point in your system the limit to buffering is reached when using nonblocking sends. Establish the effects of requesting more buffering than is available.

6-7. Can noncommutative operators such as division be used in the prefix calculation of Figure 6.8?

6-8. Determine the efficiency of the prefix calculation of Figure 6.8.

6-9. Given a fixed rectangular area with sides $x$ and $y$ and a communication that is proportional to the perimeter, $2(x + y)$, show that the minimum communication is given by $x = y$ (i.e., a square).

6-10. Write a parallel program to solve the one-dimensional problem based on finite difference equation:

6-11. In the text, we have assumed a square array for the heat distribution problem of Section 6.3.2. What are the mathematical conditions for choosing blocks or strips as the partition of the array for $0 < x < 1000$, $y = 10$ and $t_{max} = 250$?
Investigate the accuracy of convergence of the heat distribution problem using different termination methods described in Section 6.12. Determine whether it is sufficient to use the difference between the present and previous solutions. If each point is computed until each is within 1% (say) of its previous computed value, what is the accuracy of the solution?}

Write a parallel program to simulate the Game of Life as described in Section 6.13 and experiment with different initial populations.

Figure 6.19 shows a room that has four walls and a fireplace. The temperature of the wall is 20°C, and the temperature of the fireplace is 100°C. Write a parallel program using Jacobi iteration to compute the temperature inside the room and plot (preferably in color) temperature contours at 10°C intervals using Xlib calls or similar graphics calls as available on your system. Instrument the code so that the elapsed time is displayed. (This programming assignment is convenient after a Mandelbrot assignment because it can use the same graphics calls.)

Repeat Problem 6-14 but with a round room of diameter 20 ft and a point heat source in the center at 100°C; the walls are at 20°C.
6-16. Simulate a road junction controlled by traffic lights as shown in Figure 6.20. Vehicles come from all four directions along the roads and either wish to pass straight through the junction to the other side, or turn left or right at the approach, and implement them in a parallel program using your own test data (vehicle numbers and positions).

6-17. Write a parallel program to simulate the actions of the sharks and fish as described in Section 6. The parameters that are input are size of ocean, number of fish and sharks, their initial placement in the ocean, and number of times the simulation is to be run. The simulations are repeated 20 times to ensure all areas of the ocean are covered. The position of each fish is recorded at each time step. The age of each fish is increased by one after each time step. The program should simulate the movement of each fish and update their positions. The program should also keep track of the number of times each fish is caught by the sharks. The program should output the total number of times each fish is caught and the number of times each shark is caught.

6-23. (A research assignment) Develop the rules necessary to model the airflow across a wing as shown in Figure 6.21 (two dimensions). Select your own dimensions for the solution space and object. Select the number of grid points and write code to solve the problem.