• **Synchronous Iteration** *(Synchronous Parallelism)*:
  – **Barriers**:
    • Counter Barrier Implementation.
    • Tree Barrier Implementation.
    • Butterfly Connection Pattern Message-Passing Barrier.
  – **Synchronous Iteration Program Example**:
    • Iterative Solution of Linear Equations.

• **Dynamic Load Balancing**
  – **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.
  – **Program Example**: Shortest Path Problem.
Synchronous Iteration

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

- For numerical problems, a calculation is repeated and each time, a result is obtained which is used on the next execution. The process is repeated until the desired results are obtained.

- Though iterative methods are sequential in nature, parallel implementation can be successfully employed when there are multiple independent instances of the iteration. 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 but the iterations must be synchronized using point-to-point synchronization, barriers, or other synchronization mechanisms.
Synchronous Iteration
(Synchronous Parallelism)

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

\[
\text{for (j = 0; j < n; j++) /*for each synch. iteration */}
\]
\[
\hspace{1cm} \text{forall (i = 0; i < N; i++)} \{ \hspace{1cm} /*N processes each using*/
\]
\[
\hspace{2cm} \text{body(i);} \hspace{1cm} /* specific value of i */
\]
\[
\}\]

• or:

\[
\text{for (j = 0; j < n; j++)} \{ \hspace{1cm} /*for each synchr. iteration */
\]
\[
\hspace{1cm} \text{i = myrank;} \hspace{1cm} /*find value of i to be used */
\]
\[
\hspace{1cm} \text{body(i);} \hspace{1cm} /*find value of i to be used */
\]
\[
\hspace{1cm} \text{barrier(mygroup);} \hspace{1cm} /*find value of i to be used */
\]
\[
\}\]
Barriers

A synchronization mechanism applicable to 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 in their computation.

- Possible Implementations:
  - Using a counter (linear barrier).
  - Using individual point-to-point synchronization forming:
    - A tree
    - Butterfly connection pattern.
Processes Reaching A Barrier At Different Times
Centralized Counter Barrier Implementation

- Called linear barrier since access to centralized counter is serialized, thus $O(n)$ time complexity.
Message-Passing Counter Implementation of Barriers

If the master process maintains the barrier counter:

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

```
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);
```

**O(n) Time Complexity**
Tree Barrier Implementation

2 log n steps, time complexity $O(\log n)$
Tree Barrier Implementation

• Suppose 8 processes, $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$:
• Arrival phase: $\log_8 = 3$ 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: $2 \log n = 2 \log 8 = 6$
Butterfly Connection Pattern
Message-Passing Barrier

- Butterfly pattern tree construction.
- Log n stages, thus $O(\log n)$ 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: $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

---

Diagram showing the butterfly pattern tree construction for 8 processes.
Message-Passing Local Synchronization

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

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

Process $P_{i+1}$
- recv($P_i$);
- send($P_i$);
Synchronous Iteration Program Example:
Iterative Solution of Linear Equations

• Given a system of n linear equations with n unknowns:
  \[a_{n-1,0} x_0 + a_{n-1,1} x_1 + a_{n-1,2} x_2 \ldots + a_{n-1,n-1} x_{n-1} = b_{n-1}\]
  
  \[\vdots\]
  
  \[a_{1,0} x_0 + a_{1,1} x_1 + a_{1,2} x_2 \ldots + a_{1,n-1} x_{n-1} = b_1\]
  
  \[a_{0,0} x_0 + a_{0,1} x_1 + a_{0,2} x_2 \ldots + a_{0,n-1} x_{n-1} = b_0\]

By rearranging the ith equation:
  \[a_{i,0} x_0 + a_{i,1} x_1 + a_{i,2} x_2 \ldots + a_{i,i-1} x_{i-1} + a_{i,i} x_i + a_{i,i+1} x_{i+1} + \ldots + a_{i,n-1} x_{n-1} = b_i\]

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

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

• 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.
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];
```
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.
- 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.
Partitioning

• **Block allocation:**
  – Allocate groups of \( n/p \) consecutive unknowns to processors in increasing order.

• **Cyclic allocation:**
  – Processors are allocated one unknown in 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.
  \[
t_{\text{comp}} = n/p(2n + 4) \tau \quad \text{Time complexity } O(n^2/p)
\]
- 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
  \[
t_{\text{comm}} = p(t_{\text{startup}} + (n/p)t_{\text{data}}) = (pt_{\text{startup}} + nt_{\text{data}}) \tau
\]
- Overall Time:
  \[
t_p = (n/p(2n + 4) \tau + pt_{\text{startup}} + nt_{\text{data}}) \tau
\]
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:  \( n = ? \)

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

integer \( n/p \)

Minimum execution time occurs when  \( p = 16 \)
Other fully 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.”
• Rules re-applied in subsequent generations 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.
The 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:
  – Every organism with two or three neighboring organisms survives for the next generation.
  – Every organism with four or more neighbors dies from overpopulation.
  – Every organism with one neighbor or none dies from isolation.
  – 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

- 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, mapping or scheduling, is an intractable NP-complete problem, except for specific problems on specific networks.
- Hence heuristics are usually used to select processors for processes.
- Even the best static mapping may not offer the best execution time due to changing conditions at runtime and the process mapping may need to done dynamically.
- 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.
Processor Load Balance & Performance

(a) Imperfect load balancing leading to increased execution time

(b) Perfect load balancing
Centralized Dynamic Load Balancing

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.
Decentralized Dynamic Load Balancing

Distributed Work Pool Using Divide And Conquer

Master, $P_{\text{master}}$

Initial tasks

Process $M_0$

Process $M_{n-1}$

Slaves
Decentralized Dynamic Load Balancing
Distributed Work Pool With Local Queues In Slaves

Tasks could be transferred by:
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 $t$. 
Termination Detection for Decentralized Dynamic Load Balancing

- **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 informing them of global termination if needed.

![Diagram of Ring Termination Detection Algorithm]

Token passed to next processor when reached local termination condition

*Ring termination detection algorithm*
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.

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

Mountain Terrain Map

Corresponding Graph
Representation of Sample Problem Graph

Problem Graph

(a) Adjacency matrix

(b) 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:
  
  ```c
  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.
Steps of Moore’s Algorithm for Example Graph

- Stages in searching the graph:
  - Initial values
    
    | Vertices to consider | Current minimum distances |
    |-----------------------|---------------------------|
    | A                     | 0 10 18 23 34 61         |
    | vertex_queue          |                           |

  - 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.

    | Vertices to consider | Current minimum distances |
    |-----------------------|---------------------------|
    | B                     | 0 10 18 23 34 61         |
    | vertex_queue          |                           |

    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 dist[F]=10+51=61, dist[E]=10+24=34, dist[D]=10+13=23, and 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.
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 */
```
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 */
    recv(P_j, j, dist[j]);                   /* receive new distances */
    append_queue(j);                        /* append vertex to queue */
.

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];             /* get next link around vertex */
    if (newdist_j < dist[j]) {
        dist[j] = newdist_j;             /* send back updated distance */
        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

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[]

dist

Process B

Process C

New distance

Other processes