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.
Barriers

A synchronization mechanism applicable to shared-memory as well as message-passing, where each process must wait until all members of a specific process group reach a specific reference point in their computation.

- Possible Implementations:
  - A library call possibly implemented using a counter
  - Using individual point-to-point synchronization forming:
    - A tree
    - Butterfly connection pattern.
Processes Reaching A Barrier At Different Times

Time

Active

Waiting

Barrier

Processes

$P_0$ $P_1$ $P_2$ $P_{n-1}$
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.

```c
for (i = 0; i < n; i++) /* count slaves as they reach their barrier */
    recv(P_{any});
for (i = 0; i < n; i++) /* release slaves */
    send(P_i);
```

![Diagram showing the message-passing counter implementation of barriers]

Master

Slave processes
Tree Barrier Implementation

- Arrival at barrier
- Departure from barrier
- Synchronizing message
Butterfly Connection Pattern
Message-Passing Barrier

1st stage

2nd stage

3rd stage

Time

$P_0$ $P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$ $P_7$
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 \( n \) linear equations with \( n \) unknowns:

\[
\begin{align*}
\begin{align*}
\begin{align*}
\sum_{k=0}^{n-1} a_{n-1,k} x_k &= b_{n-1} \\
\vdots &
\sum_{k=0}^{n-1} a_{1,k} x_k &= b_1 \\
\sum_{k=0}^{n-1} a_{0,k} x_k &= b_0
\end{align*}
\end{align*}
\end{align*}
\]

By rearranging the \( i \)th equation:

\[
\sum_{k=0}^{n-1} a_{i,k} x_k = b_i
\]

to:

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

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.
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

$$t_{\text{comm}} = p(t_{\text{startup}} + \frac{n}{p}t_{\text{data}}) = (pt_{\text{startup}} + nt_{\text{data}}) \tau$$

- Overall Time:

$$t_p = (\frac{n}{p}(2n + 4) \tau + pt_{\text{startup}} + nt_{\text{data}}) \tau$$
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 \( p = 16 \)
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.

- Optimal static load balancing, optimal 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 offer the best execution time due to changing conditions at runtime and the process may need to be 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 $t$. 
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:
  
  ```
  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
    
    \[
    \begin{array}{|c|c|c|c|}
    \hline
    \text{Vertices to consider} & \text{Current minimum distances} \\
    \hline
    A & 0 & \infty & \infty & \infty & \infty & \infty \\
    \hline
    \text{vertex_queue} & \text{dist[]} \\
    \hline
    \end{array}
    \]

  - 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.

    \[
    \begin{array}{|c|c|c|c|}
    \hline
    \text{Vertices to consider} & \text{Current minimum distances} \\
    \hline
    B & 0 & 10 & \infty & \infty & \infty & \infty \\
    \hline
    \text{vertex_queue} & \text{dist[]} \\
    \hline
    \end{array}
    \]

    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
    \[\text{dist}[F]=10+51=61, \quad \text{dist}[E]=10+24=34,\]
    \[\text{dist}[D]=10+13=23, \quad \text{dist}[C]=10+8=18.\]
    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.
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 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 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:

```
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];          /* add vertex to queue if not there */
        if (newdist_j < dist[j]) {
            dist[j]=newdist_j;
            append_queue(j);
        }
    }                                          /* 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_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];
    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

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[]

dist

Process A

New distance

Process B

New distance

Vertex w[]

dist

Process C

Other processes