Solving a System of Linear Equations by Iteration

Suppose that a system of linear equations is not in a special triangular form.

- How to solve? : by iteration
  - Iterative methods are preferred over direct methods when such direct methods require excessive computations.
    - + small memory requirements
    - - may not always converge
- The solution requires global synchronization.

Jacobi Iteration

- all values of x are updated together
- will converge if array of a's is diagonally dominant.
  - the diagonal values of a have an absolute value greater than the sum of the absolute values of other a's on the row

    $\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$ (sufficient but not necessary condition)
- starts with some initial guess ($x_i = b_i$) for all the unknowns.

Termination.

1- Via a termination condition.

$|x_i^t - x_i^{t-1}| <$ error tolerance

This does not guarantee the solution to that accuracy!

- Errors might compound and the computed value could be very significantly different from the final exact value.
- Errors of one computed value will effect the accuracy of other computed values that use it in their calculation.

Other ternination conditions have been proposed.

2- Via a maximum number of iterations.

Consider the tradeoff between using a complex termination calculation with potentially fewer iterations and using a less complex iterations bw checking for termination.

Allow a number of iterations between checking for termination.

Calculations must be sync'd globally.

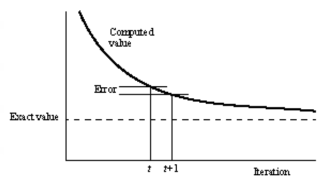


Figure 6.9 Convergence rate.

Sequential Code.

```
for (i=0; i<n; i++)
  x[i] = b[i];             /*initialize unknowns*/

for (it=0; it<limit; it++) {
  for (i=0; i<n; i++) {    /* for each unknown*/
     sum = 0;
     for (j=0; j<n; j++)  /*compute summation of a[][]x[]*/
        if (i != j) 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*/
}
```

Can be more written in a more efficient way!

Parallel Code.

Allocate one process for each unknown, and each process will iterate the same number of times.

```
x[i] = b[i];             /*initialize unknowns*/
for (it=0; it<limit; it++) {
  for (i=0; i<n; i++) {    /* for each unknown*/
     sum = -a[i][i] * x[i];
     for (j=0; j<n; j++)  /*compute summation of a[][]x[]*/
        sum += a[i][j] * x[j];
     new_x[i] = (b[i]-sum)/a[i][i]; /*compute unknown*/
  }
  broadcast_receive(&new_x[i]);
 global_barrier();
}
```

Communication can be done using `send()` and `receive()`'s.

MPI has `MPI_Allgather()` or `MPI_Allgatherv()`

Typically, we want to iterate until the approximations are sufficiently close:

```
it = 0;
do {
   it++;
    ...
} while (tolerance() && (it < limit));
                      why ?
```

Partitioning.

Number of processors is much fewer than the number of data items to be processed.

Assuming *p* processors and *n* unknowns:

- block allocation
  x_0 ... $x_{(n/p)-1}$ to P_0
  $x_{(n/p)}$ ... $x_{(2n/p)-1}$ to P_1 and so on.
- cyclic allocation
  x_0, x_p, x_{2p}, ... $x_{((n/p)-1)p}$ to P_0
  x_1, x_{p+1}, x_{2p+1}, ... $x_{((n/p)-1)p+1}$ to P_1 and so on.
  - more complex to compute indices of unknowns.
  - more effort is needed to group the unknowns in one message.

Analysis.

Suppose *p* processors and *n* equations (unknowns):

- Each processor operates upon *n/p* unknowns
- *t* iterations per processor

Computation.

$t_{comp} = n/p\ (2n+4)\ t$

1 * and 1 + in inner loop --- 1 *, 2 -, and 1 / in outer loop.

Communication.

$t_{comm} = p\ (t_{startup} + (n/p)\ t_{data})t = pt_{startup} + nt_{data})t$

The resulting total execution time has one component that is decreasing function of p and another that is increasing function of p.

We can find the minimum by differentiation.

Heat Distribution Problem

- The solution requires local synchronization.
- We can find the temperature distribution on a sheet of metal by dividing the area into a fine mesh of points.
  - the temperature at an inside point can be taken to be the average of the temperatures of the four neighboring points.
  - *(n-1) x (n-1)* interior points.
  - Edge points are when i=0, i=n, j=0, j=n and have fixed values corresponding to the fixed temperatures of the edges.

We can compute the temperature at each point by iterating the equation:

hi,j = (hi-1,j + hi+1,j + hi,j-1 + hi,j+1) / 4,

where *0<i<n* and *0<j<n*, for a fixed number of iterations or until it satisfies some convergence criteria.

- This equation occurs in several other similar problems:
  - pressure
  - voltage

Each point is an unknown dependent upon a few other unknowns.

Natural order?

Finite Difference method

six neighbors in 3 dimensions

Laplace's equation.

Sequential Code.

```
for (it=0; it<limit; it++) {
   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++)  /* update points*/
         for (j=1; j<n; j++)
            h[i][j] = g[i][j];
}
continue = FALSE;
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      if (!converged(i,j) {
           continue = TRUE;
           break;
      }
} while (continue == TRUE);
```

Parallel Code.

Each point can be visited simultaneously w/o any change to the algorithm.

Simple Algorithm: Assign one process to each point.

```
for (it=0; it<limit; it++) {
     g = 0.25 *(w + x + y + z);
     send(&g, P_{i-1,j});   /* nonblocking sends */
     send(&g, P_{i+1,j});
     send(&g, P_{i,j-1});
     send(&g, P_{i,j+1});
     recv(&g, P_{i-1,j});   /* synchronous receives */
     recv(&g, P_{i+1,j});
     recv(&g, P_{i,j-1});
     recv(&g, P_{i,j+1});
}
```

Local Barrier

Processes stop when they reach their required precision:

- a master process needs to be modified when all processes have stopped.

```
it = 0;
do {
      it++;
      g = 0.25 *(w + x + y + z);
      send(&g, P_{i-1,j});   /* locally blocking sends */
      send(&g, P_{i+1,j});
      send(&g, P_{i,j-1});
      send(&g, P_{i,j+1});
      recv(&g, P_{i-1,j});   /* locally blocking receives */
      recv(&g, P_{i+1,j});
      recv(&g, P_{i,j-1});
      recv(&g, P_{i,j+1});
} while ((!converged(i,j)) || (iteration==limit));
send(&g, &i, &j, &it, P_{master});
```

Handling the processes operating at the edges:

```
if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
it = 0;
do {  it++;
      g = 0.25 *(w + x + y + z);
      if (!first_row) send(&g, P_{i-1,j});
     if (!last_row) send(&g, P_{i+1,j});
     if (!first_column) send(&g, P_{i,j-1});
     if (!last_column) send(&g, P_{i,j+1});
     if (!first_row) recv(&g, P_{i-1,j});
     if (!last_row) recv(&g, P_{i+1,j});
     if (!first_column) recv(&g, P_{i,j-1});
     if (!last_column) recv(&g, P_{i,j+1});
} while ((!converged(i,j)) || (iteration==limit));
send(&g, &i, &j, &it, P_{master});
```

Partitioning.

- Block partition (square blocks)
- Strip partition (row or column strips)

- The communication times will be heavily influenced by startup time.
  - In general, the strip partition is best for a large startup time, and a block partition is best for a small startup time.
  - The startup time will be large in most systems, especially in workstation clusters.

Implementation Details.

A complete column of points needs to be sent to adjacent process in one message.

- When the array is stored in row-major order as in C, then a row-strip partitioning can be used.
  - Each process will have an additional row of points at each edge, called *ghost points*, that hold the values from the adjacent edge.

```
for (i=1; i<m; i++)
   for (j=1; j<n/p; 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<m; i++)       /* update points*/
   for (j=1; j<n/p; j++)
      h[i][j] = g[i][j];
send(&g[1][1], &m, P_{i-1});  /* send rows */
send(&g[1][m], &m, P_{i+1});
recv(&h[1][0], &m, P_{i-1});  /* receive rows */
recv(&h[1][m+1],&m, P_{i+1});
```

Safety and Deadlock.

The arrangement when all processes send their messages first and then receive all of their messages is described "unsafe" in the MPI literature.

- Because, the amount of buffering is not specified in MPI
- A **`send()`** may block if buffer storage is insufficient
  - Hence a locally blocking send may behave as a sync.send.
  - Since a matching receive would never be executed if all the sends are sync., deadlock would occur.

Solution: make the code safe by alternating the order of sends and receives in adjacent processors.

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

MPI offers several alternative methods for safe communication:

- Combined send/receives: `MPI_Sendrecv()`
- Buffered sends: `MPI_Bsend()`
- Nonblocking routines: `MPI_Isend()` and `MPI_Irecv()` followed by `MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, or `MPI_Testany()`.

```
MPI_Isend(&g[1][1], &m, P i-1);
MPI_Isend(&g[1][m], &m, P i+1);
MPI_Irecv(&h[1][0], &m, P i-1);
MPI_Irecv(&h[1][m+1],&m, P i+1);
```

Cellular Automata

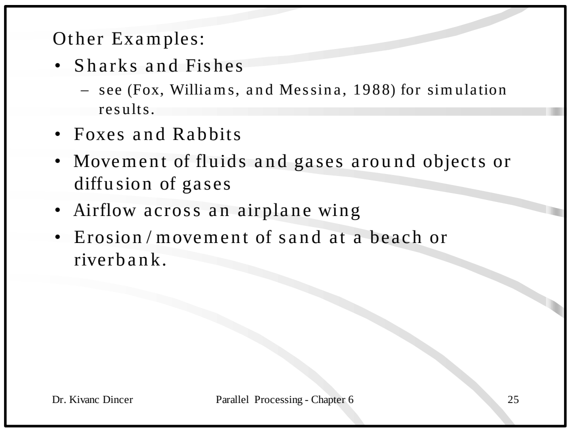
Cellular automaton is paritularly suitable for synchronous iteration.

- Problem space is divided into cells
  - Each cell is can be in one of a finite number of states.
- Cells are affected from their neighbors according to certain rules, and all cells are affected simultaneously in a "generation."
- The rules are reapplied in subsequent generations so that cells evolve, or change state, from generation to generation.

Game of Life

The most famous cellular automata.

- We have a board that consists of 2D array of cells.
- Each cell can hold one "organism" and has eight neighboring cells.
- Initially some cells are occupied.
- Following rules apply:
  - Every organism with 2/3 neighboring organisms survives for the next generation.
  - Every organism with 4/more neighboring organisms dies from overpopulation.
  - Every organism with 0/1 neighboring organisms dies from isolation.
  - Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.



Other Examples:

- Sharks and Fishes
  - see (Fox, Williams, and Messina, 1988) for simulation results.
- Foxes and Rabbits
- Movement of fluids and gases around objects or diffusion of gases
- Airflow across an airplane wing
- Erosion / movement of sand at a beach or riverbank.