Overview: Synchronous Computations

- definition
- synchronous computation example 1: solving linear systems with Jacobi Iteration
  - solution of linear systems (problem definition)
  - fixed-point iterative linear system solvers and Jacobi iteration
  - serial and parallel code
  - partitioning and performance model
- synchronous computation example 2: solving the Heat Equation in 2D
  - problem definition and finite-difference discretization
  - serial and parallel code
  - partitioning: strip (1D) versus block (2D) partitioning; performance modelling
  - the concept of ghost layer of points (aka halo)
  - avoiding deadlocks
  - early termination
- synchronous computation example 3: Advection Equation in 2D-Assignment 1 (not covered in the lecture, similar to example 2)

Ref: Chapter 6: Wilkinson and Allen

Synchronous computations (definition)

- computations in which a group of processes perform local independent work BUT *must periodically wait for each of other* (i.e., synchronize) before proceeding
- (low-level) example: in SIMD computers the same instruction is executed on several processors on different data before proceeding with the next instruction
- in many cases, synchronization is a consequence of data exchange (e.g., to satisfy data dependency among steps)
- synchronous iteration (this lecture) is an important class of synchronous computations
  - to solve problems iteratively in such a way that several processes start together at the beginning of each iteration and the next iteration cannot begin until all processes have finished the preceding iteration
- we will illustrate synchronous iteration with two examples: Jacobi iteration and solution of the 2D Heat Equation in 2D

Solution of Linear Systems (problem definition)

- we aim at finding $x \in \mathbb{R}^n$ such that:

$$Ax = b$$

  with $A \in \mathbb{R}^{n\times n}$ (nonsingular matrix) and $b \in \mathbb{R}^n$ (right-hand-side vector) given
- in component-wise form, this problem reads (assuming 0-based indexing):

$$\underbrace{\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}}_{b}$$

- n is the number of equations/unknowns in the system
- ubiquitous problem in computational science and engineering (CSE) applications (e.g., numerical solution of PDEs using the finite element method)
- high quality parallel message-passing libraries around (e.g., PETSc, Hypre, Trilinos)

Fixed-point Iterative Linear solvers and Jacobi Iteration

- the most basic iterative solvers are the so-called linear fixed-point methods
- in such methods, A is split as $A = M - N$, with M being nonsingular
- starting from initial approximate solution $x^{(0)}$, they iterate the recurrence given by:

$$x^{(k+1)} = x^{(k)} + M^{-1}\underbrace{(b - Ax^{(k)})}_{\text{residual}}$$

  till some termination criterion is fulfilled (e.g., max # of iterations reached or distance among $x^{(k+1)}$ and $x^{(k)}$ "sufficiently small")
- in practice one uses a cheap-to-invert approximation $M^{-1} \approx A^{-1}$ (note that if $M^{-1} = A^{-1}$ then $x^{(1)}$ is already the solution)
- if they convergence, they are guaranteed to converge to x; however, they don't always converge (they converge if and only if $\rho(I - M^{-1}A) < 1$, with $\rho(B)$ being the max eigenvalue of B in absolute value)
- Jacobi iteration (our example) choose $M^{-1} = D^{-1}$, with D being the diagonal of A (a quite rough approximation of A^{-1}!)

Sequential Jacobi Iteration

the Jacobi recurrence in matrix form:

$$x^{(k+1)} = x^{(k)} + D^{-1}(b - Ax^{(k)})$$

can be written in component-wise form as:

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - \sum_{j=0}^{n-1} a_{ij}x_j^{(k)}\right), \; i = 0, \ldots, n-1$$

```
...  // Init vector x
for (iter=0; iter<max_iter; iter++)
{
 for (i=0; i<n; i++) {
   sum=0.0
   for (j=0; j<n; j++) {
    sum = sum + a[i][j]*x[j];
   }
   new_x[i]=x[i]+(b[i]-sum)/a[i][i];
 }
 for (i=0; i<n; i++)
   x[i]=new_x[i];
}
```

- arrays `b[]` and `a[][]` hold b and A
- arrays `x[]` and `new_x[]` hold $x^{(k)}$ and $x^{(k+1)}$
- for simplicity, we ignore early stopping condition (typically based on "sufficiently small" distance among $x^{(k+1)}$ and $x^{(k)}$)

Parallel Jacobi Iteration

- consider a row-wise partition of A and a (naive!) static mapping of a single row per process, i.e., $p = n$ (i.e., we have as many rows as processes)
- the vector b is partitioned/mapped accordingly to the rows of A
- HOWEVER, the vectors $x^{(k)}$ and $x^{(k+1)}$ are not partitioned/mapped to the processes, but replicated in all processes (why?)

```
... // Init vector x
    // (consistently in all processes!)
i=process_rank_id()
for (iter=0; iter<max_iter; iter++)
{
  sum=0.0
  for (j=0; j<n; j++) {
    sum = sum + a[i][j]*x[j];
  }
  new_x[i]=x[i]+(b[i]-sum)/a[i][i];
  ... // collective comm here!
  for (i=0; i<n; i++)
    x[i]=new_x[i];
}
```

message-passing parallel program (remainder: SPMD execution)

- at each outer loop iteration, each process with rank i, computes $x_i^{(k+1)}$ (entry of next iterate mapped to it)
- collective communication acts as a synchronization point
- this communication is such that all processes end up in `new_x[]` with the entries of $x^{(k+1)}$ computed by any other processes
- let us discuss how to realize this communication step (next slides)

Parallel Jacobi Iteration (communication)

- most naive approach: p broadcasts naively implemented using point-to-point communication (not the way to go)

```
i = process_rank_id();
for (j=0; j<n; j++)
 if (i!=j) send(&new_x[i],1,j);
for (j=0; j<n; j++)
 if (i!=j) recv(&new_x[j],1,j);
```

Alternative 1 (deadlock-free?)

```
i = process_rank_id();
for (root=0; root<n; root++)
 if (i==root)
  for (j = 0; j < n; j++)
    if (i!=j) send(&new_x[i],1,j);
 else
  recv(&new_x[root], 1, root);
```

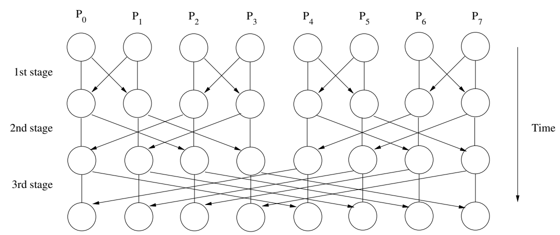
Alternative 2-reorder sends/recvs (deadlock-free but still naive)

- less naive approach: p broadcasts implemented using broadcast collective (but still not the way to go)

```
for (root=0; root<n; root++)
  broadcast(&new_x[root], 1, root);
```


Parallel Jacobi Iteration (communication)

- smarter approach (but still not the way to go): butterfly pattern (aka recursive doubling) using point-to-point communication

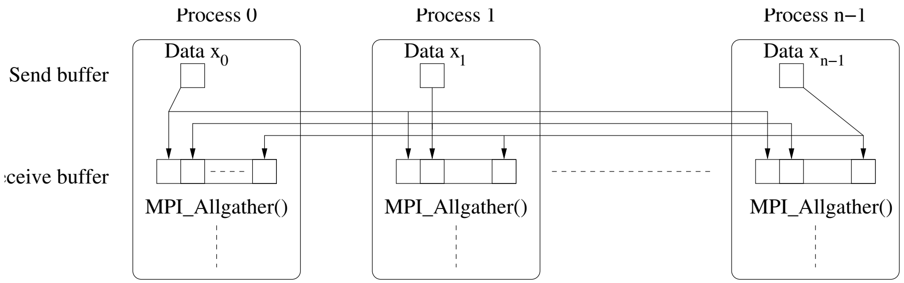


$p = 8$ (thus $s = 3$)

- completes in $s = \log_2(p)$ steps
- at each stage, we have $\frac{p}{2}$ pairs of communication process
- at each stage, message size doubles (why?)

Parallel Jacobi Iteration (communication)

- smartest approach (the way to go): use `MPI_Allgather` collective (it opens the door for exploiting a highly optimized algorithm available at the MPI implementation for the particular underlying high speed network at hand)



Partitioning and Parallel Cost Analysis of Jacobi iteration

- let us be more clever, and partition A (and b) into blocks of $\frac{n}{p}$ rows each
- let us denote by τ the number of Jacobi iterations
- as usual, t_f is the time/flop, t_s message start-up time, t_w per-word time
- sequential algorithm time (2 flops/inner loop + 3 flops/outer loop):

  $t_{\text{seq}} = \tau\, n(2n+3)t_f$
- parallel computation (decreases linearly with p):

  $t_{\text{comp}} = \tau\, \frac{n}{p}(2n+3)t_f$
- parallel communication (increases linearly with p):

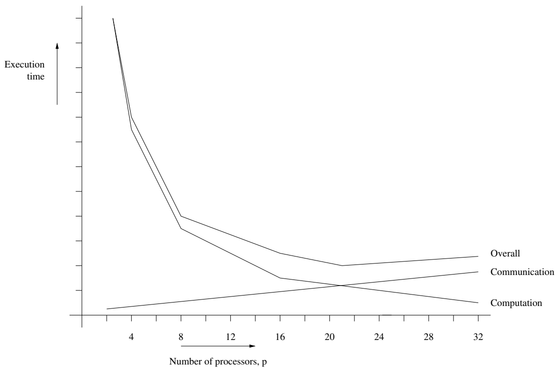
  $t_{\text{comm}} = \tau p(t_s + \frac{n}{p}t_w) = \tau(pt_s + nt_w)$
- parallel algorithm time:

  $t_{\text{par}} = t_{\text{comp}} + t_{\text{comm}}$

Assumptions:

- neglect the effect of the number of links and t_h
- communication implemented inefficiently with p broadcasts
- communication cost of a broadcast equivalent to a single point-to-point communication

Instantiating the Parallel Jacobi Iteration Time Model



instantiating the model parameters as:

- $t_s = 10^5 t_f$
- $t_w = 50 t_f$
- fixed problem size of $n = 1000$ (strong scaling)