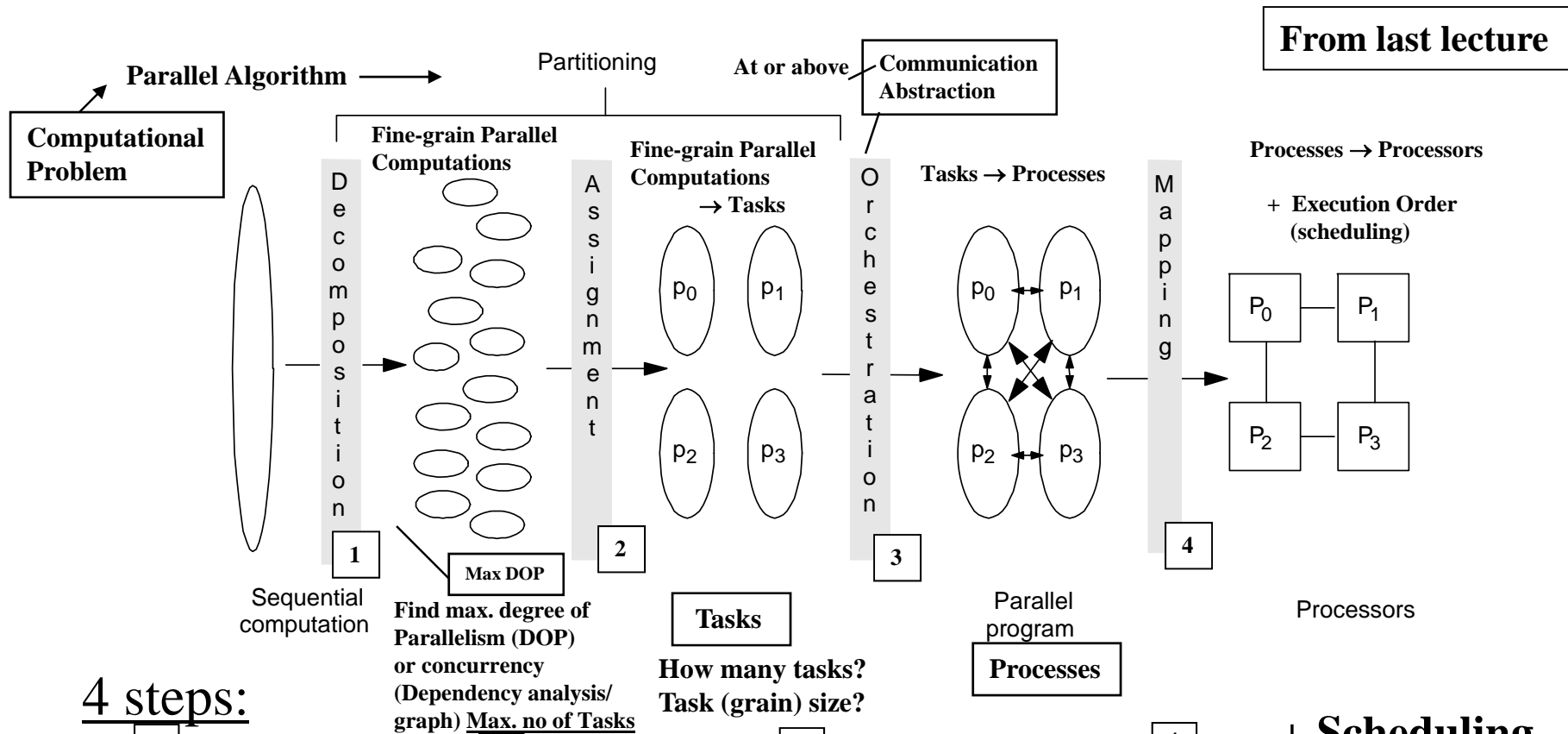


# Steps in Creating a Parallel Program



4 steps:

1 **Decomposition**, 2 **Assignment**, 3 **Orchestration**, 4 **Mapping** + Scheduling

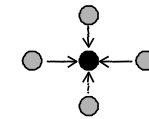
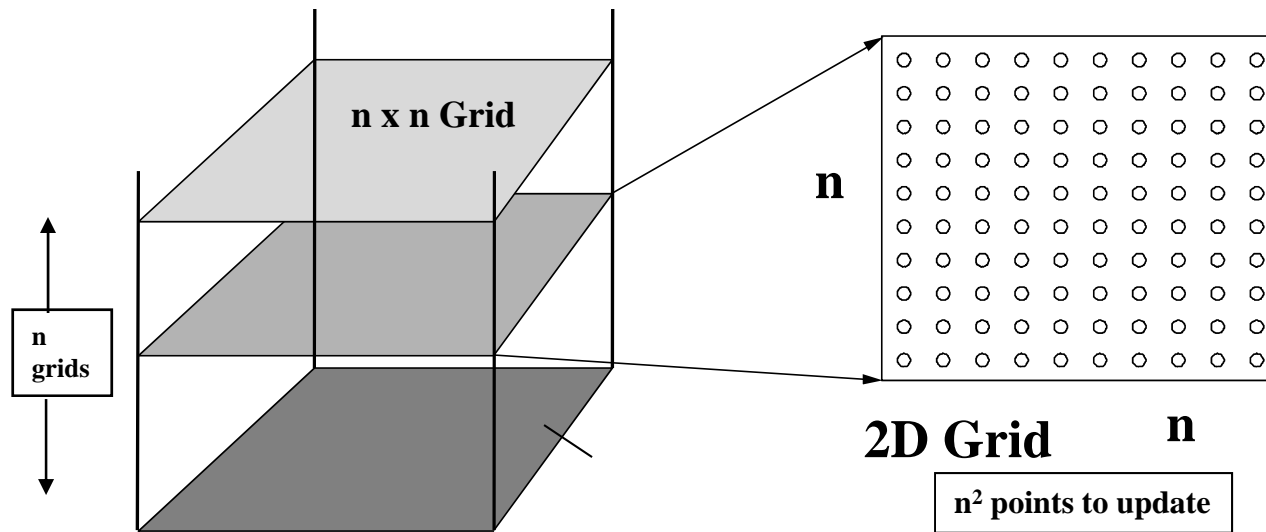
- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly

Vs. implicitly by parallelizing compiler

(PCA Chapter 2.3)

CMPE655 - Shaaban

# Example Motivating Problem: Simulating Ocean Currents/Heat Transfer ...



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j - 1] + A[i - 1, j] + A[i,j + 1] + A[i + 1, j])$$

**Maximum Degree of Parallelism (DOP) or concurrency:  $O(n^2)$  data parallel computations per grid per iteration**

When one task updates/computes one grid element

Total  $O(n^3)$  Computations Per iteration  $n^2$  per grid  $\times$  n grids

(a) Cross sections

(b) Spatial discretization of a cross section

- **Model as two-dimensional “n x n” grids**
- **Discretize in space and time**
  - finer spatial and temporal resolution => greater accuracy
- **Many different computations per time step,  $O(n^2)$  per grid.**
  - set up and solve linear equations iteratively (Gauss-Seidel) .
- **Concurrency across and within grid computations per iteration**
  - $n^2$  parallel computations per grid x number of grids

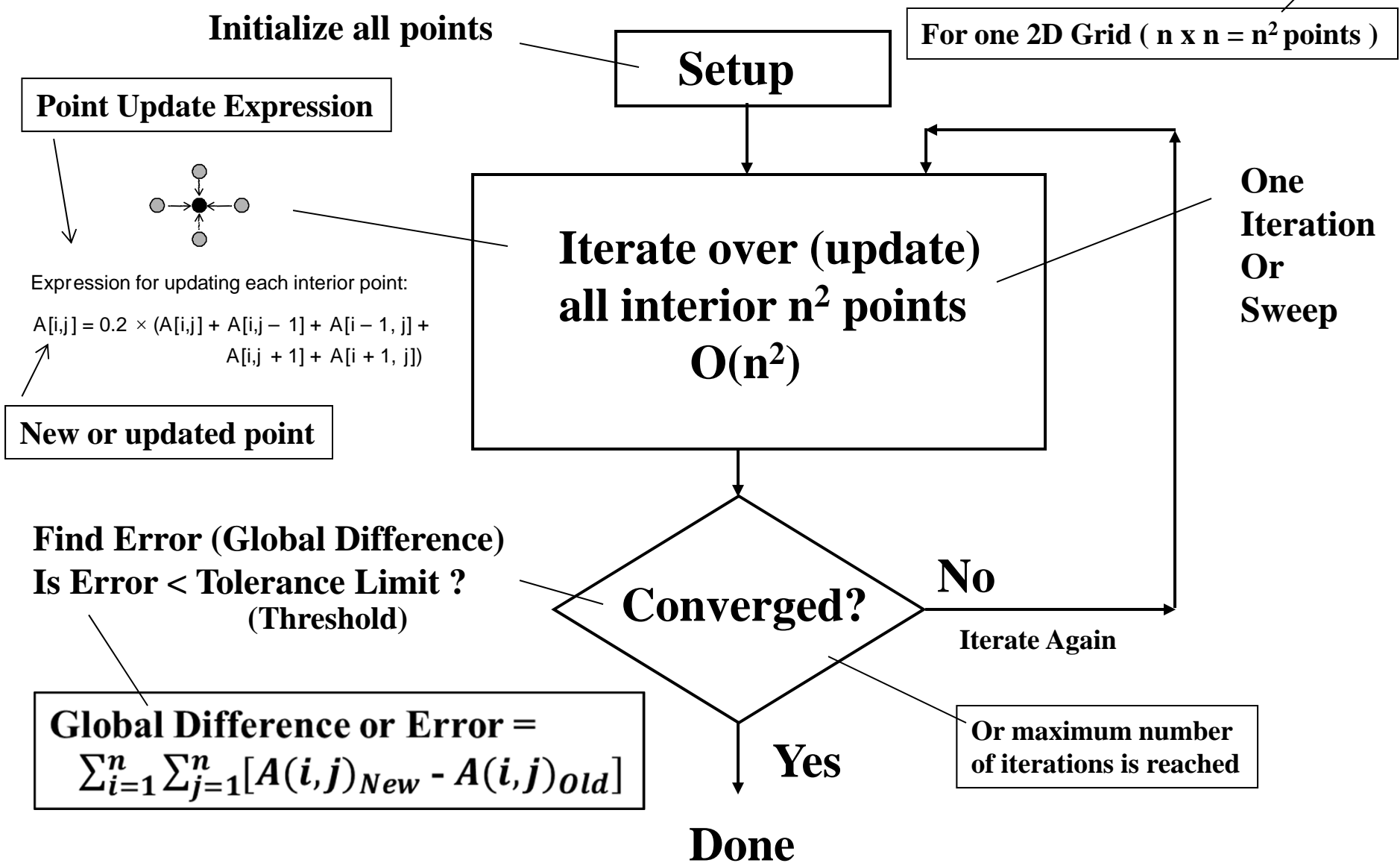
Synchronous iteration

(PCA Chapter 2.3)

More reading: PP Chapter 11.3 (Pages 352-364)

# Solution of Linear System of Equation By Synchronous Iteration

*Iterations are sequential – Parallelism within an iteration  $O(n^2)$*



# Parallelization of An Example Program

Examine a simplified version of a piece of Ocean simulation

- ↓ • Iterative (Gauss-Seidel) linear equation solver Synchronous iteration

**One 2D Grid,  $n \times n = n^2$  points (instead of 3D –  $n$  grids)**

Illustrate parallel program in low-level parallel language:

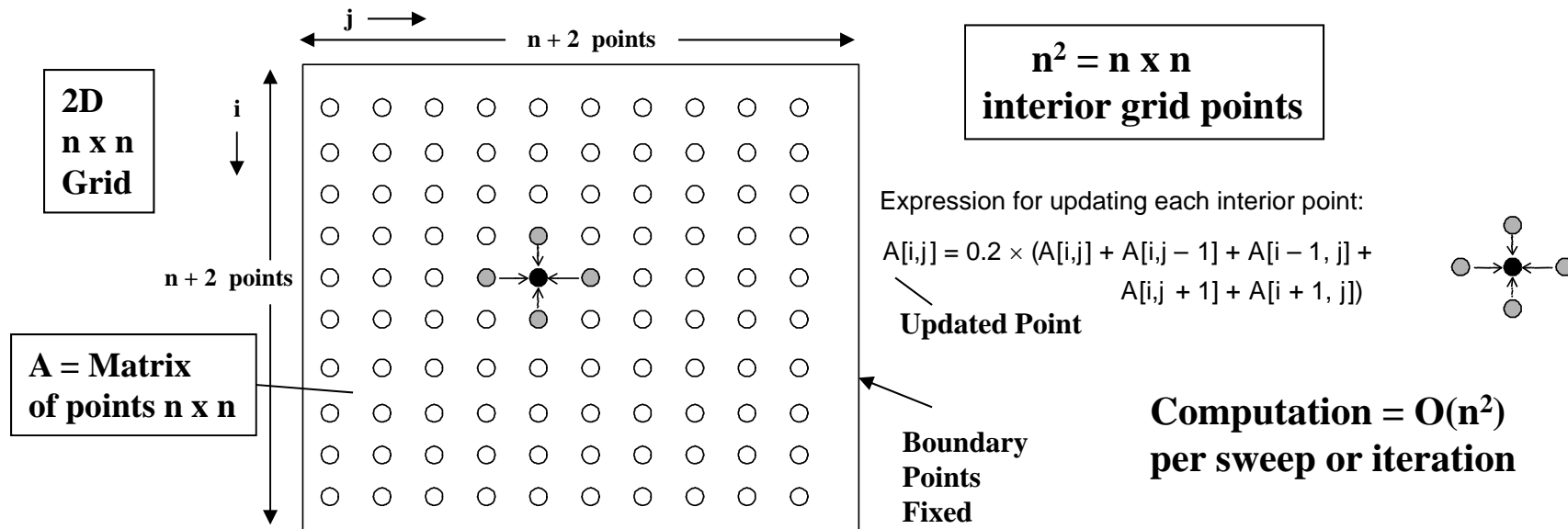
- C-like pseudo-code with simple extensions for parallelism
- Expose basic communication and synchronization primitives that must be supported by parallel programming model.

Three parallel programming models targeted for orchestration:

- 
- **Data Parallel**
  - **Shared Address Space (SAS)**
  - **Message Passing**

**(PCA Chapter 2.3)**

# (One) 2D Grid Solver Example



- Simplified version of solver in Ocean simulation 2D (one grid) not 3D
- Gauss-Seidel (near-neighbor) sweeps (iterations) to convergence:
  - 1 – Interior n-by-n points of (n+2)-by-(n+2) updated in each sweep (iteration)
  - 2 – Updates done in-place in grid, and difference from previous value is computed
  - 3 – Accumulate partial differences into a global difference at the end of every sweep or iteration
  - 4 – Check if error (global difference) has converged (to within a tolerance parameter)
 

Iterate

    - If so, exit solver; if not, do another sweep (iteration)
    - Or iterate for a set maximum number of iterations.

# Pseudocode, Sequential Equation Solver

```

1. int n;                                     /*size of matrix: (n + 2-by-n + 2) elements*/
2. float **A, diff = 0;

3. main()
4. begin
5.   read(n) ;                               /*read input parameter: matrix size*/
6.   A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.   initialize(A);                          /*initialize the matrix A somehow*/
8.   Solve (A);                              /*call the routine to solve equation*/
9. end main

10.procedure Solve (A)                       /*solve the equation system*/
11. float **A;                               /*A is an (n + 2)-by-(n + 2) array*/
12.begin
13. int i, j, done = 0;
14. float diff = 0, temp;
15. while (!done) do                         /*outermost loop over sweeps*/
16.   diff = 0;                               /*initialize maximum difference to 0*/
17.   for i ← 1 to n do                       /*sweep over nonborder points of grid*/
18.     for j ← 1 to n do
19.       temp = A[i,j];                      /*save old value of element*/
20.       A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         New value A[i,j+1] + A[i+1,j]); /*compute average*/
22.       diff += abs(A[i,j] - temp);
23.     end for
24.   end for
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
27.end procedure

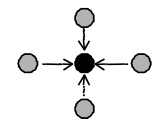
```

**Setup**  
**Initialize grid points**  
**Call equation solver**

**A = Matrix of n x n points**

**Iterate until convergence**

**i.e one iteration**  
**Sweep O(n<sup>2</sup>) computations**



**Update Points**

**diff = Global Difference**

$$\text{Global Difference or Error} = \sum_{i=1}^n \sum_{j=1}^n [A(i,j)_{\text{New}} - A(i,j)_{\text{Old}}]$$

**Done?**

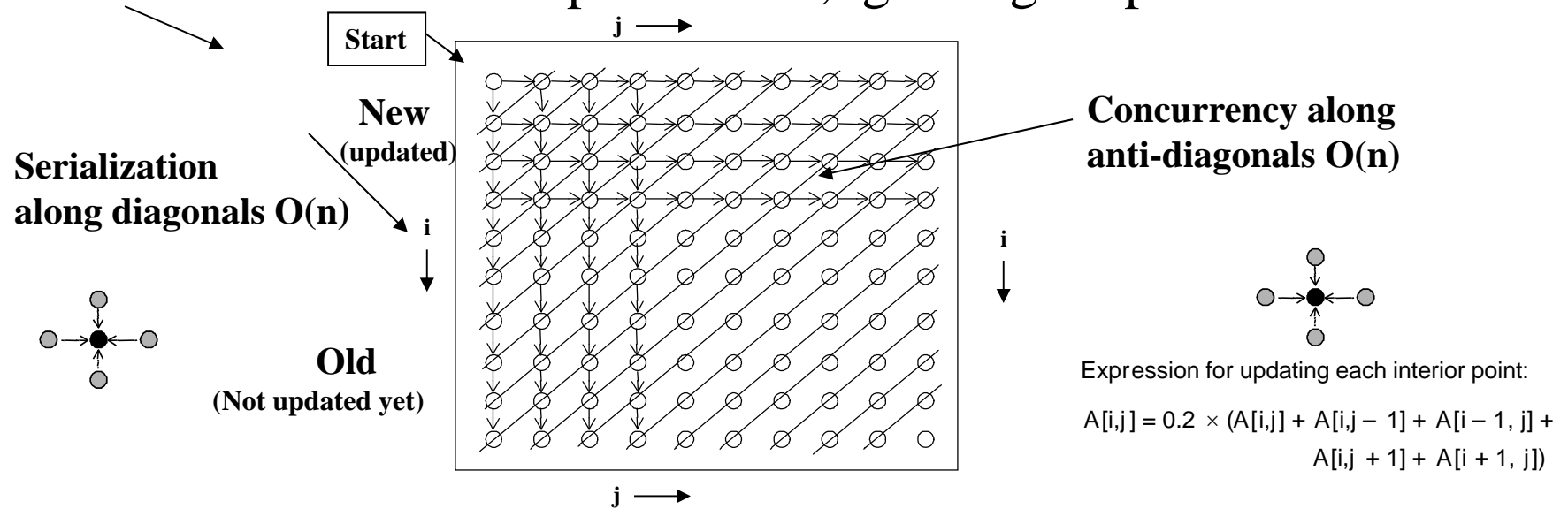
**TOL, tolerance or threshold**

# Decomposition

How?

- Simple way to identify concurrency is to look at loop iterations 1
- 2 *Dependency analysis*; if not enough concurrency is found, then look further into application 3

- 1 • Not much concurrency here at this level (all loops *sequential*)
- 2 • Examine fundamental dependencies, ignoring loop structure



- Concurrency  $O(n)$  along anti-diagonals, serialization  $O(n)$  along diagonal
- Retain loop structure, use pt-to-pt synch; Problem: too many synch ops.
- or • Restructure loops, use global synch; load imbalance and too much synch

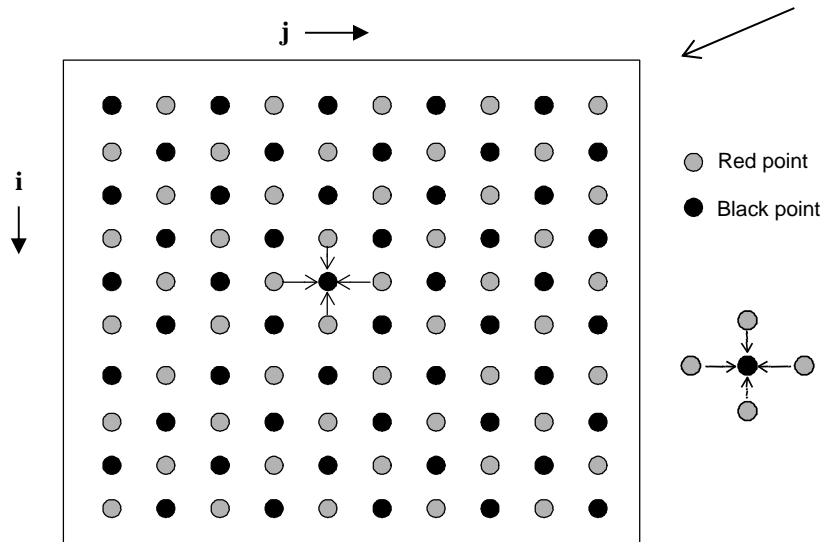
i.e using barriers along diagonals

## Decomposition:

# Exploit Application Knowledge

- Reorder grid traversal: red-black ordering

Two parallel sweeps  
Each with parallel  $n^2/2$  points updates



Maximum Degree of parallelism = DOP =  $O(n^2)$

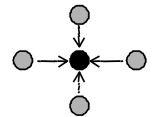
Type of parallelism: Data parallelism

One point update per task ( $n^2$  parallel tasks)

Computation = 1

Communication = 4

Communication-to-Computation ratio = 4



For PRAM with  $O(n^2)$  processors:

Sweep =  $O(1)$

Global Difference =  $O(\log_2 n^2)$

Thus:  $T = O(\log_2 n^2)$

Per Iteration

- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel:
- Global synchronization between them (conservative but convenient)
- Ocean uses red-black; here we use simpler, asynchronous one to illustrate

➔ No red-black sweeps, simply ignore dependencies within a single sweep (iteration) all points can be updated in parallel DOP =  $n^2 = O(n^2)$

Iterations may converge slower than red-black ordering

- Sequential order same as original.

**i.e. Max Software DOP =  $n^2 = O(n^2)$**



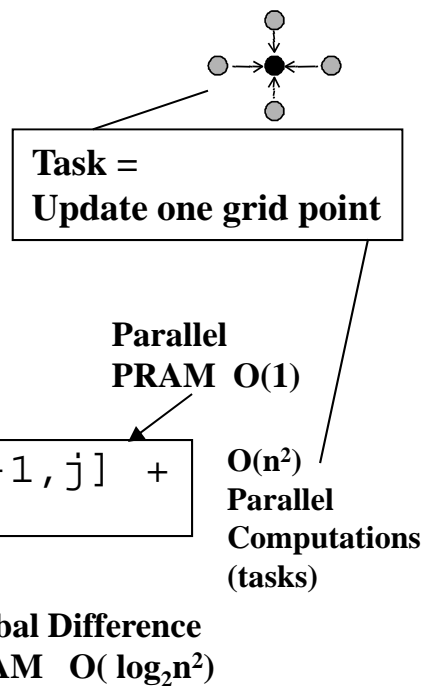
# Decomposition Only

for\_all = in parallel

```

15. while (!done) do /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do /*a parallel loop nest*/
18.     for_all j ← 1 to n do
19.       temp = A[i,j];
20.       Point Update A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         A[i,j+1] + A[i+1,j]);
22.       diff += abs(A[i,j] - temp);
23.     end for_all
24.   end for_all
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
  
```

DOP =  $O(n^2)$



- Task = One row
- Task = update one grid point
- Degree of Parallelism (DOP)
- Decomposition into elements:** degree of concurrency  $n^2$
- To decompose into rows,** make line 18 loop sequential; degree of parallelism (DOP) =  $n$
- for\_all leaves assignment left to system
  - but implicit global synch. at end of for\_all loop

**Fine Grain:**  
 $n^2$  parallel tasks  
 each updates one element  
 DOP =  $O(n^2)$

**Coarser Grain:**  
 $n$  parallel tasks each update a row  
 Task = grid row  
 Computation =  $O(n)$   
 Communication =  $O(n) \sim 2n$   
 Communication to Computation ratio =  $O(1) \sim 2$

Task = update one row of points

The “for\_all” loop construct imply parallel loop computations

# Assignment: (Update $n/p$ rows per task)

i.e Task Assignment

$p$  = number of processes or processors

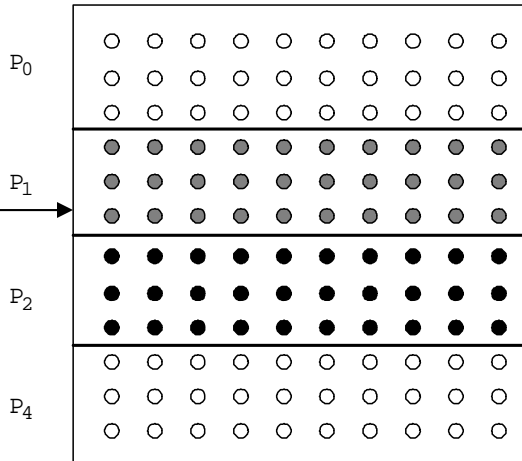
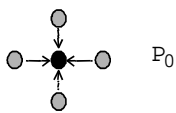
i.e  $n^2/p$  points per task

## • Static assignments (given decomposition into rows)

– Block assignment of rows: Row  $i$  is assigned to process  $\lfloor \frac{i}{p} \rfloor$

– Cyclic assignment of rows: process  $i$  is assigned rows  $i, i+p,$  and

SO ON



Block or strip assignment  
 $n/p$  rows per task

$p$  = number of processors  
(tasks or processes)

$p$  = number of processors  $< n$   
 $p$  tasks or processes  
 Task = updates  $n/p$  rows =  $n^2/p$  elements  
 Computation =  $O(n^2/p)$   
 Communication =  $O(n)$   $\sim 2n$  (2 rows)  
 Communication-to-Computation  
 ratio =  $O(n / (n^2/p)) = O(p/n)$   
 Lower C-to-C ratio is better

## • Dynamic assignment (at runtime):

Not needed here (computation is predictable)

– Get a row index, work on the row, get a new row, and so on

• Static assignment into rows reduces concurrency (from  $n^2$  to  $p$ )

$p$  tasks  
Instead of  $n^2$

– concurrency (DOP) =  $n$  for one row per task C-to-C =  $O(1)$

/ In one task

– Block assign. reduces communication by keeping adjacent rows together

Why Block Assignment

• Let's examine orchestration under three programming models:

1- Data Parallel

2- Shared Address Space (SAS)

3- Message Passing

# Data Parallel Solver

```
1. int n, nprocs; /*grid size (n + 2-by-n + 2) and number of processes*/
2. float **A, diff = 0;
```

**nprocs = number of processes = p**

```
3. main()
4. begin
5.   read(n); read(nprocs); /*read input grid size and number of processes*/
6.   A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.   initialize(A); /*initialize the matrix A somehow*/
8.   Solve (A); /*call the routine to solve equation*/
9. end main
```

**Setup/Initialize Points**

**A = Matrix of n x n points**

```
10. procedure Solve(A) /*solve the equation system*/
11.   float **A; /*A is an (n + 2-by-n + 2) array*/
```

**Block decomposition by row**

**n/p rows per processor**

```
12. begin
13.   int i, j, done = 0;
14.   float mydiff = 0, temp;
14a.   DECOMP A[BLOCK,*, nprocs];
15.   while (!done) do /*outermost loop over sweeps*/
16.     mydiff = 0; /*initialize maximum difference to 0*/
17.     for_all i ← 1 to n do /*sweep over non-border points of grid*/
18.       for_all j ← 1 to n do
19.         temp = A[i,j]; /*save old value of element*/
20.         A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]); /*compute average*/
22.         mydiff += abs(A[i,j] - temp);
23.       end for_all
24.     end for_all / mydiff = Local Difference diff = Global Difference
```

**Sweep/Iteration:**

**T = O(n<sup>2</sup>/p)**

↑ In Parallel ↓

```
24a.   REDUCE (mydiff, diff, ADD);
25.   if (diff/(n*n) < TOL) then done = 1;
26.   end while
27. end procedure
```

**Add all local differences (REDUCE)**

**cost depends on architecture**

**and implementation of REDUCE**

**best: O(log<sub>2</sub>p) using binary tree reduction**

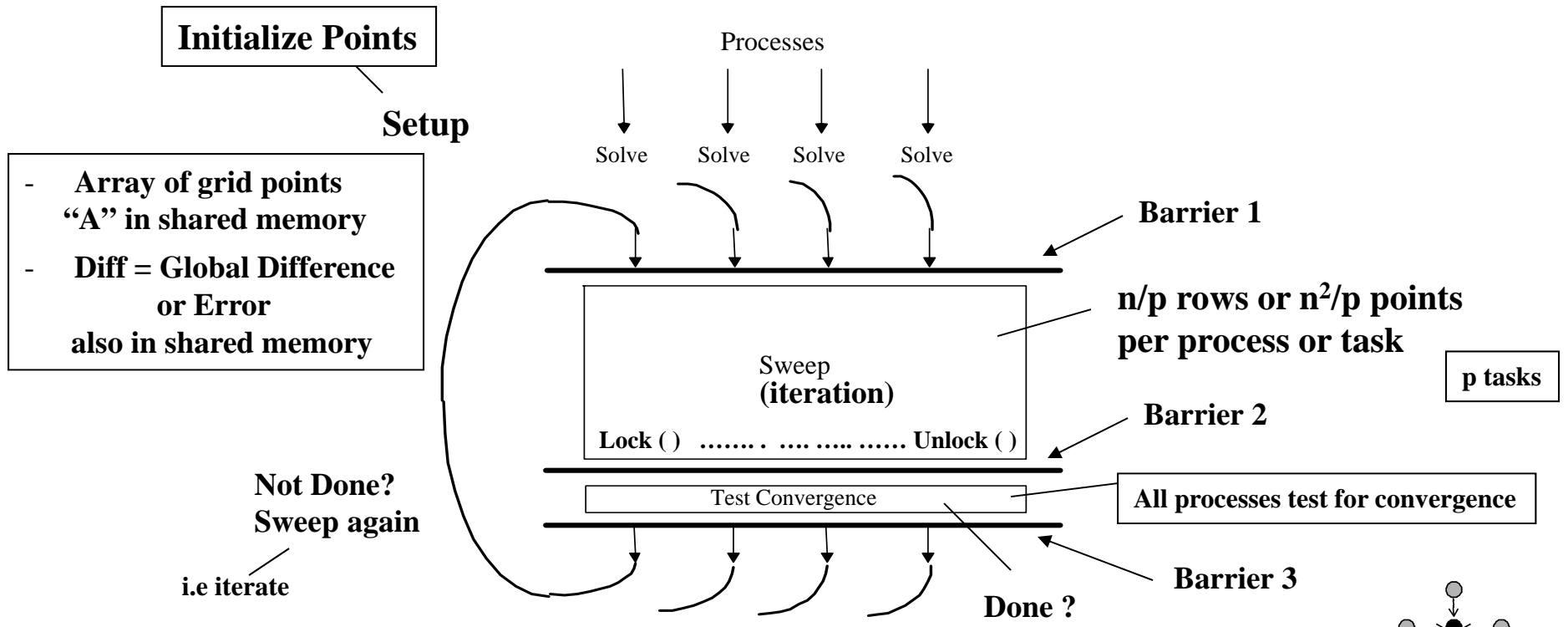
**Worst: O(p) sequentially**

**O(n<sup>2</sup>/p + log<sub>2</sub>p) ≤ T(iteration) ≤ O(n<sup>2</sup>/p + p)**

SAS

# Shared Address Space Solver

Single Program Multiple Data (SPMD) ← Still MIMD



i.e Which n/p rows to update for a task or process with a given process ID (PID)

- Row Assignment controlled by values of variables used as loop bounds and individual process ID (PID)

For process

As shown next slide →

# Pseudo-code, Parallel Equation Solver for Shared Address Space (SAS)

**SAS**

```

1.  int n, nprocs;           /*matrix dimension and number of processors to be used*/
2a. float **A, diff;        /*A is global (shared) array representing the grid*/
                               /*diff is global (shared) maximum difference in current
                               sweep*/
2b.  LOCKDEC(diff_lock);    /*declaration of lock to enforce mutual exclusion*/
2c.  BARDEC(bar1);          /*barrier declaration for global synchronization between
                               sweeps*/

Main process or thread → 3.  main()
                           4.  begin
                           5.      read(n); read(nprocs); /*read input matrix size and number of processes*/
                           6.      A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
                           7.      initialize(A); /*initialize A in an unspecified way*/
                           8a.     CREATE (nprocs-1, Solve, A);
                           8.      Solve(A); /*main process becomes a worker too*/
                           8b.     WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
                           9.  end main

Create p-1 processes →

10. procedure Solve(A)
11.     float **A; /*A is entire n+2-by-n+2 shared array, as in the sequential program*/

More Setup →
12.     begin
13.         int i,j, pid, done = 0;
14.         float temp, mydiff = 0; /*private variables*/
14a.        int mymin = 1 + (pid * n/nprocs); /*assume that n is exactly divisible by*/
14b.        int mymax = mymin + n/nprocs - 1 /*nprocs for simplicity here*/

Which rows? →

15.         while (!done) do /*outer loop over all diagonal elements*/
16.             mydiff = diff = 0; /*set global diff to 0 (okay for all to do it)*/
16a.            BARRIER(bar1, nprocs); /*ensure all reach here before anyone modifies diff*/
17.             for i ← mymin to mymax do /*for each of my rows*/
18.                 for j ← 1 to n do /*for all nonborder elements in that row*/
19.                     temp = A[i,j];
20.                     A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);
22.                     mydiff += abs(A[i,j] - temp);
23.                 endfor
24.             endfor
25a.            LOCK(diff_lock); /*Mutual Exclusion (lock) for global difference
25b.            diff += mydiff; /*update global diff if necessary*/
25c.            UNLOCK(diff_lock); /*Critical Section: global difference

Barrier 1 → (Start sweep)
Barrier 2 →

25d.            BARRIER(bar1, nprocs); /*ensure all reach here before checking if done*/
25e.            if (diff/(n*n) < TOL) then done = 1; /*check convergence; all get same answer*/

Barrier 3 →
25f.            BARRIER(bar1, nprocs);
26.         endwhile
27.     end procedure
    
```

**A = Matrix of n x n points allocated in shared memory**

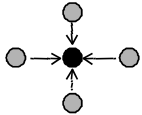
**Setup**

**Array "A" is shared (all grid points)**

**# of processors = p = nprocs**  
**pid = process ID, 0 .... P-1**

**Loop Bounds/Which n/p Rows?**  
**mymin = low row**  
**mymax = high row**

**Private Variables**



**Sweep:**  
**T = O(n<sup>2</sup>/p)**

**T(p) = O(n<sup>2</sup>/p + p)**

**T = O(p)** Serialized update of global difference

**Check/test convergence: all processes do it**

**Done?**

# Notes on SAS Program

- **SPMD: not lockstep** (i.e. still MIMD not SIMD) or even necessarily same instructions.

SPMD = Single Program Multiple Data

- **Row Assignment controlled by values of variables used as loop bounds and process ID (pid)** (i.e. mymin, mymax)

Which n/p rows?

- Unique pid per process, used to control assignment of blocks of rows to processes.

- **Done condition (convergence test) evaluated redundantly by all processes**

By checking if Global Difference = diff < Threshold

- Code that does the update identical to sequential program

**But**

- Each process has private mydiff variable

Why?

Otherwise each process must enter the shared global difference critical section for each point,  $n^2/p$  times ( $n^2$  times total) instead of just p times per iteration for all processes

- Most interesting special operations needed are for synchronization

- Accumulations of local differences (mydiff) into shared global difference (diff) have to be **mutually exclusive**

Using LOCK () ... UNLOCK ()

- **Why the need for all the barriers?**

In a critical section

# Need for Mutual Exclusion

- Code each process executes:

diff = Global Difference

load the value of diff into register r1  
 add the register r2 to register r1  
 store the value of register r1 into diff

- A possible interleaving:

i.e relative operations ordering in time

Local Difference in r2

Time



P1

r1 ← diff

r1 ← r1+r2

diff ← r1

diff = Global Difference (in shared memory)

P2

Load

r1 ← diff

r1 ← r1+r2

Store

diff ← r1

{P1 gets 0 in its r1}

{P2 also gets 0}

{P1 sets its r1 to 1}

{P2 sets its r1 to 1}

{P1 sets cell\_cost to 1}

{P2 also sets cell\_cost to 1}

r2 = mydiff = Local Difference

- Need the sets of operations to be **atomic** (**mutually exclusive**)

**Fix ?**

# Mutual Exclusion



No order guarantee provided

Provided by **LOCK-UNLOCK** around critical section

- Set of operations we want to execute atomically
- **Implementation of LOCK/UNLOCK must guarantee mutual exclusion.**

However, no order guarantee

i.e one task/process or processor at a time in critical section

Can lead to significant serialization if contended (many tasks want to enter critical section at the same time)

- Especially costly since many accesses in critical section are non-local
- Main reason to use private mydiff for partial accumulation:
  - ↘ Reduce the number times needed to enter critical section by each process to update global difference:

$O(p)$  total number of accesses to critical section

- **Once per iteration vs.  $n^2/p$  times per process without mydiff**

Or  $O(n^2)$  total number of accesses to critical section by all processes  
i.e. Enter critical section once for each point update (Without private mydiff)



# Global (or group) Event Synchronization

BARRIER(nprocs): wait here till nprocs processes get here

- Built using lower level primitives i.e locks, semaphores
- Global sum example: wait for all to accumulate before using sum
- Often used to separate phases of computation

<u>Process P_1</u>	<u>Process P_2</u>	<u>Process P_nprocs</u>	
set up eqn system	set up eqn system	set up eqn system	<b>Setup</b>
<b>Barrier</b> (name, nprocs)	<b>Barrier</b> (name, nprocs)	<b>Barrier</b> (name, nprocs)	
solve eqn system	solve eqn system	solve eqn system	<b>Update Points</b>
Lock() ..... Unlock ()	Lock() ..... Unlock ()	Lock() ..... Unlock ()	<b>Enter Critical Section</b>
<b>Barrier</b> (name, nprocs)	<b>Barrier</b> (name, nprocs)	<b>Barrier</b> (name, nprocs)	
apply results	apply results	apply results	<b>Convergence Test</b>
<b>Barrier</b> (name, nprocs)	<b>Barrier</b> (name, nprocs)	<b>Barrier</b> (name, nprocs)	

- Conservative form of preserving dependencies, but easy to use

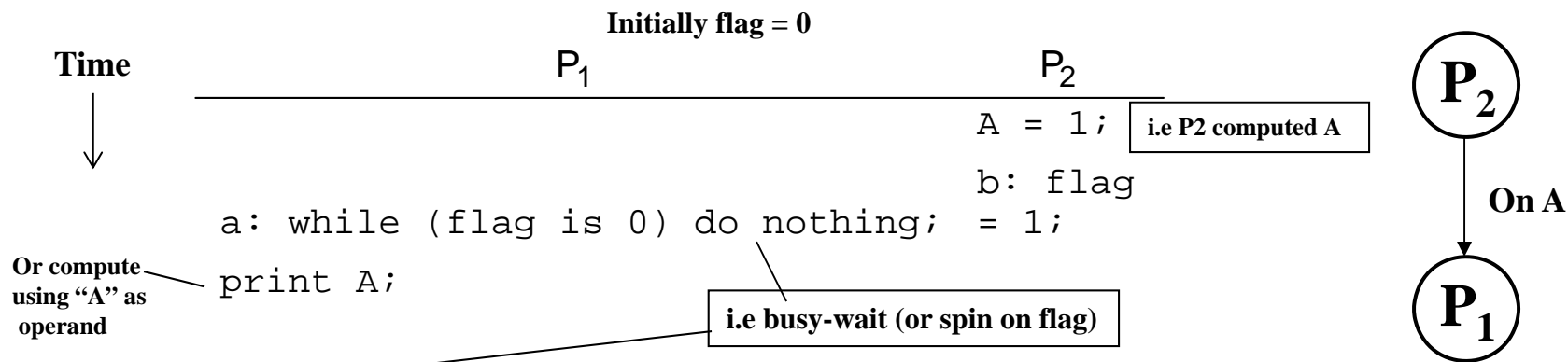
Done by all processes

**Most Critical Barrier?**

# Point-to-point (Ordering) Event Synchronization SAS (Not Used or Needed Here)

One process notifies another of an event so it can proceed:

- Needed for **task ordering** according to **data dependence between tasks**
- Common example: producer-consumer (bounded buffer)
- Concurrent programming on uniprocessor: semaphores
- Shared address space parallel programs: semaphores, or use ordinary variables as **flags** in shared address space



- *Busy-waiting (i.e. spinning)*
  - *Or block process (better for uniprocessors?)*

# Message Passing Grid Solver

- Cannot declare A to be a shared array any more

Thus

No shared address space

- Need to compose it logically from per-process private arrays

myA arrays

- Usually allocated in accordance with the assignment of work
- Process assigned a set of rows allocates them locally

Each n/p rows in local memory

n/p rows in this case myA

- Explicit transfers (communication) of entire border or “Ghost” rows between tasks is needed (as shown next slide)

At start of each iteration

- Structurally similar to SAS (e.g. SPMD), but **orchestration is different**

- Data structures and data access/naming e.g Local arrays vs. shared array

Explicit

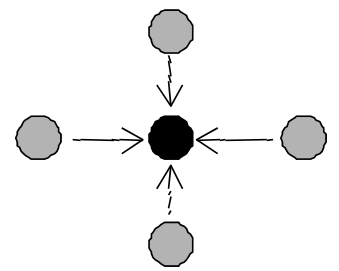
– Communication

Implicit

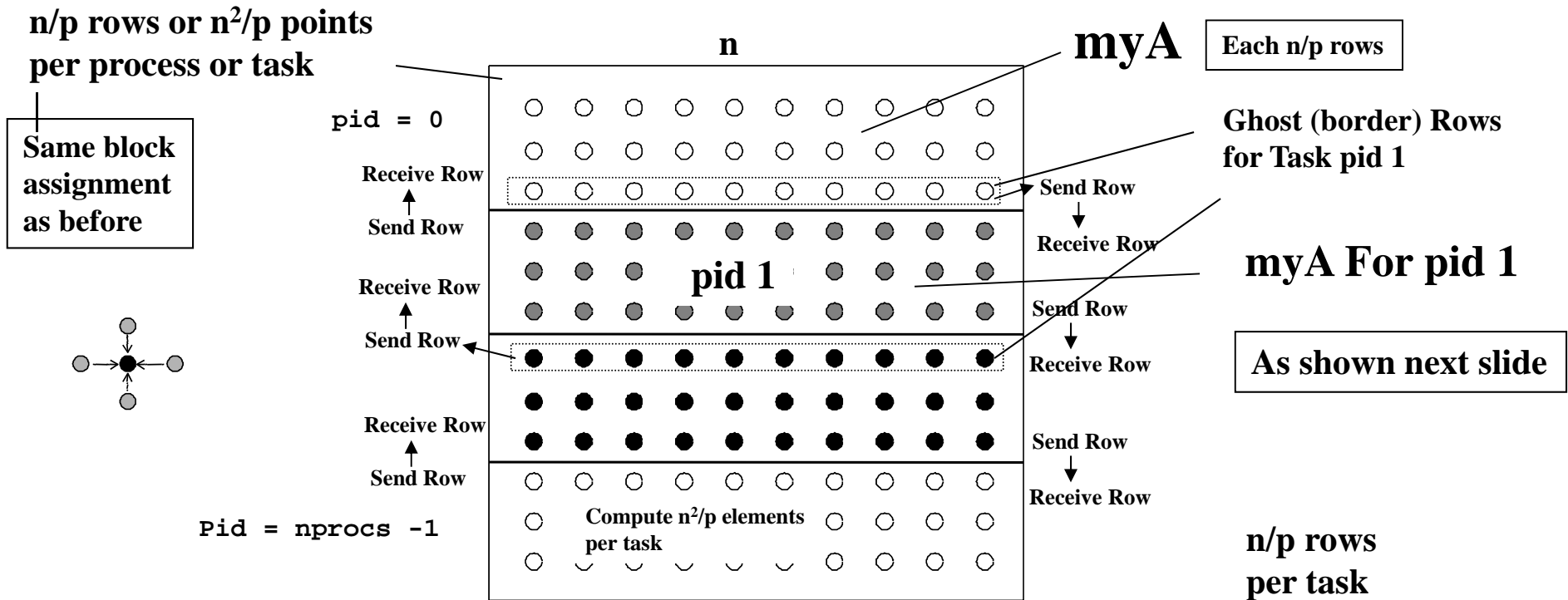
– Synchronization

} Via Send/receive pairs

+



# Message Passing Grid Solver



- Parallel Computation =  $O(n^2/p)$
- Communication of rows =  $O(n)$
- Communication of local DIFF =  $O(p)$
- Computation =  $O(n^2/p)$
- Communication =  $O(n + p)$
- Communication-to-Computation Ratio =  $O((n+p)/(n^2/p)) = O((np + p^2) / n^2)$

**Time per iteration:**  
 $T = T(\text{computation}) + T(\text{communication})$   
 $T = O(n^2/p + n + p)$

nprocs = number of processes = number of processors = p

## Pseudo-code, Parallel Equation Solver for Message Passing

Create p-1 processes

Done by master process (pid = 0)

```

1. int pid, n, b; /*process id, matrix dimension and number of processors to be used*/
2. float **myA;
3. main()
4. begin
5.     read(n); read(nprocs); /*read input matrix size and number of processes*/
8a.    CREATE (nprocs-1, Solve);
8b.    Solve(); /*main process becomes a worker too*/
8c.    WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
9. end main

```

# of processors = p = nprocs

## Message Passing

Initialize local rows myA

```

10. procedure Solve()
11. begin
13.     int i, j, pid, n' = n/nprocs, done = 0;
14.     float temp, tempdiff, mydiff = 0; /*private variables*/
6.     myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2); /*my assigned rows of A*/
7. initialize(myA); /*initialize my rows of A, in an unspecified way*/

```

Initialize myA (Local rows)

Send one or two ghost rows

Exchange ghost rows (send/receive)

Receive one or two ghost rows

```

15. while (!done) do
16.     mydiff = 0; /*set local diff to 0*/
16a.    if (pid != 0) then SEND (&myA[1,0], n*sizeof(float), pid-1, ROW);
16b.    if (pid == nprocs-1) then SEND (&myA[n',0], n*sizeof(float), pid+1, ROW);
16c.    if (pid != 0) then RECEIVE (&myA[0,0], n*sizeof(float), pid-1, ROW);
16d.    if (pid != nprocs-1) then RECEIVE (&myA[n'+1,0], n*sizeof(float), pid+1, ROW);
/*border rows of neighbors have now been copied into myA[0,*] and myA[n'+1,*]*/

```

Communication  $O(n)$  exchange ghost rows

Before start of iteration

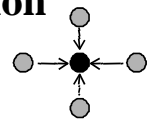
Sweep over n/p rows = n<sup>2</sup>/p points per task  
T = O(n<sup>2</sup>/p)

```

17. for i ← 1 to n' do /*for each of my (nonghost) rows*/
18.     for j ← 1 to n do /*for all nonborder elements in that row*/
19.         temp = myA[i,j];
20.         myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.             myA[i,j+1] + myA[i+1,j]);
22.         mydiff += abs(myA[i,j] - temp);
23.     endfor
24. endfor

```

Computation  $O(n^2/p)$



Local Difference

/\*communicate local diff values and determine if done; can be replaced by reduction and broadcast\*/

Send mydiff to pid 0  
Receive test result from pid 0

```

25a.    if (pid != 0) then /*process 0 holds global total diff*/
25b.        SEND (mydiff, sizeof(float), 0, DIFF);
25c.        RECEIVE (done, sizeof(int), 0, DONE);
25d.    else /*pid 0 does this*/
25e.        for i ← 1 to nprocs-1 do /*for each other process*/
25f.            RECEIVE (tempdiff, sizeof(float), *, DIFF);
25g.            mydiff += tempdiff; /*accumulate into total*/
25h.        endfor
25i.        if (mydiff/(n*n) < TOL) then done = 1;
25j.        for i ← 1 to nprocs-1 do /*for each other process*/
25k.            SEND (done, sizeof(int), i, DONE);
25l.        endfor
25m.    endif
26. endwhile
27. end procedure

```

Pid 0

Only Pid 0 tests for convergence

+

$O(p)$  Communication

T = O(n<sup>2</sup>/p + n + p)

Pid 0:

Done?

calculate global difference and test for convergence

send test result to all processes

# Notes on Message Passing Program

- Use of ghost rows. Or border rows i.e Two-sided communication
- Receive does not transfer data, send does (sender-initiated)
  - Unlike **SAS** which is usually receiver-initiated (load fetches data) i.e One-sided communication
- Communication done at beginning of iteration (exchange of ghost rows).
- Explicit communication in whole rows, not one element at a time (SAS)
- Core similar, but indices/bounds in local space rather than global space.
- Synchronization through sends and blocking receives (implicit)
  - Update of global difference and event synch for done condition
  - Could implement locks and barriers with messages
- Only one process (pid = 0) checks convergence (done condition).
- Can use REDUCE and BROADCAST library calls to simplify code:

```
/*communicate local diff values and determine if done, using reduction and broadcast*/
25b.   REDUCE(0,mydiff,sizeof(float),ADD); Compute global difference
25c.   if (pid == 0) then
25i.     if (mydiff/(n*n) < TOL) then done = 1;
25k.   endif
25m.     BROADCAST(0,done,sizeof(int),DONE); Broadcast convergence test result to all processes
      \
      Tell all tasks if done
```

# Message-Passing Modes: Send and Receive Alternatives

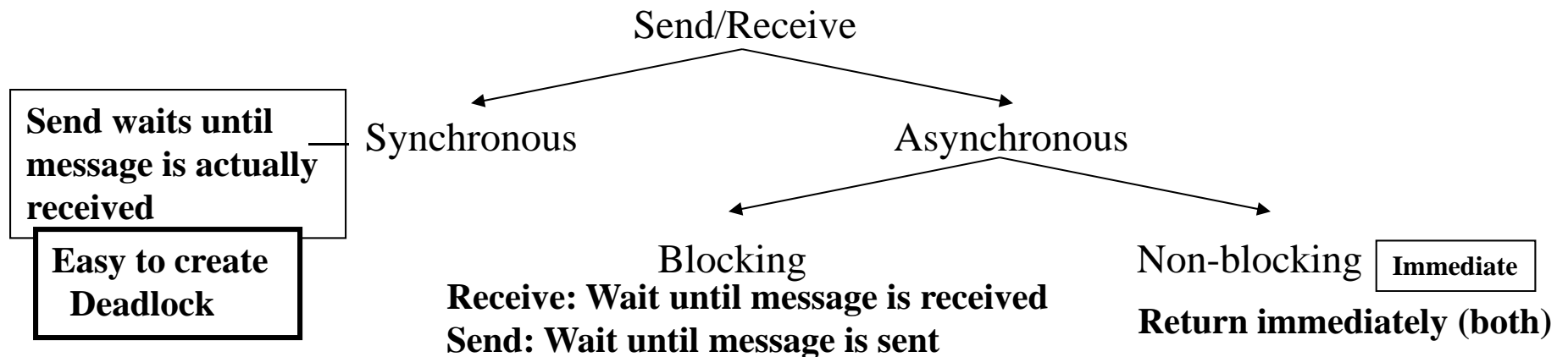
Point-to-Point Communication

Can extend functionality: stride, scatter-gather, groups

All can be implemented using send/receive primitives

Semantic flavors: **based on when control is returned**

Affect when data structures or buffers can be reused at either end



- Affect event synch (mutual exclusion implied: only one process touches data)
- Affect ease of programming and performance

Synchronous messages provide built-in synch. through match

- Separate event synchronization needed with asynch. messages

With synchronous messages, our code is deadlocked. Fix?

Use asynchronous blocking sends/receives

# Message-Passing Modes: Send and Receive Alternatives

Synchronous Message Passing: In MPI: MPI\_Ssend () MPI\_Srecv()

Process X executing a synchronous send to process Y has to wait until process Y has executed a synchronous receive from X.

Asynchronous Message Passing:

Blocking Send/Receive: In MPI: MPI\_Send () MPI\_Recv()

Most  
Common  
Type

A blocking send is executed when a process reaches it without waiting for a corresponding receive. Returns when the message is sent. A blocking receive is executed when a process reaches it and only returns after the message has been received.

Non-Blocking Send/Receive: In MPI: MPI\_Isend () MPI\_Irecv()

A non-blocking send is executed when reached by the process without waiting for a corresponding receive. A non-blocking receive is executed when a process reaches it without waiting for a corresponding send. Both return immediately.



# Orchestration: Summary

## Shared address space

- Shared and private data explicitly separate
- Communication implicit in access patterns
- No *correctness* need for data distribution
- Synchronization via atomic operations on shared data
- Synchronization explicit and **distinct** from data communication

Here Critical Section + Barriers

## Message passing

- Data distribution among local address spaces needed **myA's**
- No explicit shared structures (implicit in communication patterns)
- Communication is explicit
- Synchronization implicit in communication (at least in synch. case)
  - Mutual exclusion implied **No SAS**

# Correctness in Grid Solver Program

**Decomposition** and <sup>Task</sup>**Assignment** similar in SAS and message-passing  
Orchestration is different:

- Data structures, data access/naming, communication, synchronization

<u>AKA shared?</u>	<u>SAS</u>	<u>Msg-Passing</u>
Explicit global data structure?	<b>Yes</b>	<b>No</b>
Assignment indept of data layout?	<b>Yes</b>	<b>No</b>
Communication	<b>Implicit</b>	<b>Explicit</b>
Synchronization	<b>Explicit</b>	<b>Implicit</b>
Explicit replication of border rows? i.e. ghost rows	<b>No</b>	<b>Yes</b>

Lock/unlock  
Barriers

Via  
Send/  
Receive  
Pairs

Ghost  
Rows

Requirements for performance are another story ...