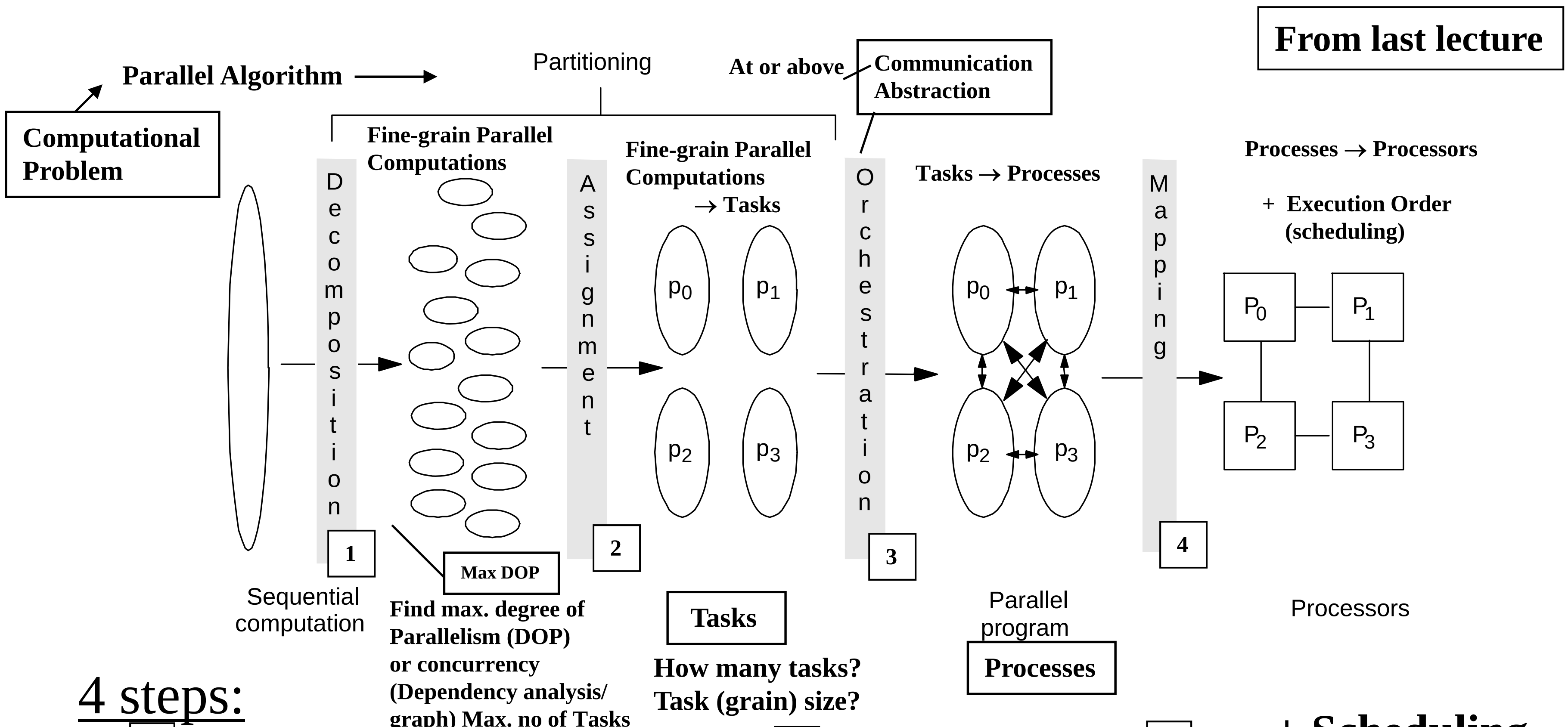
Steps in Creating a Parallel Program



4 steps:

1 2 3 4

+ Scheduling

Task

Decomposition, Assignment, Orchestration, Mapping

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

Vs. implicitly by parallelizing compiler

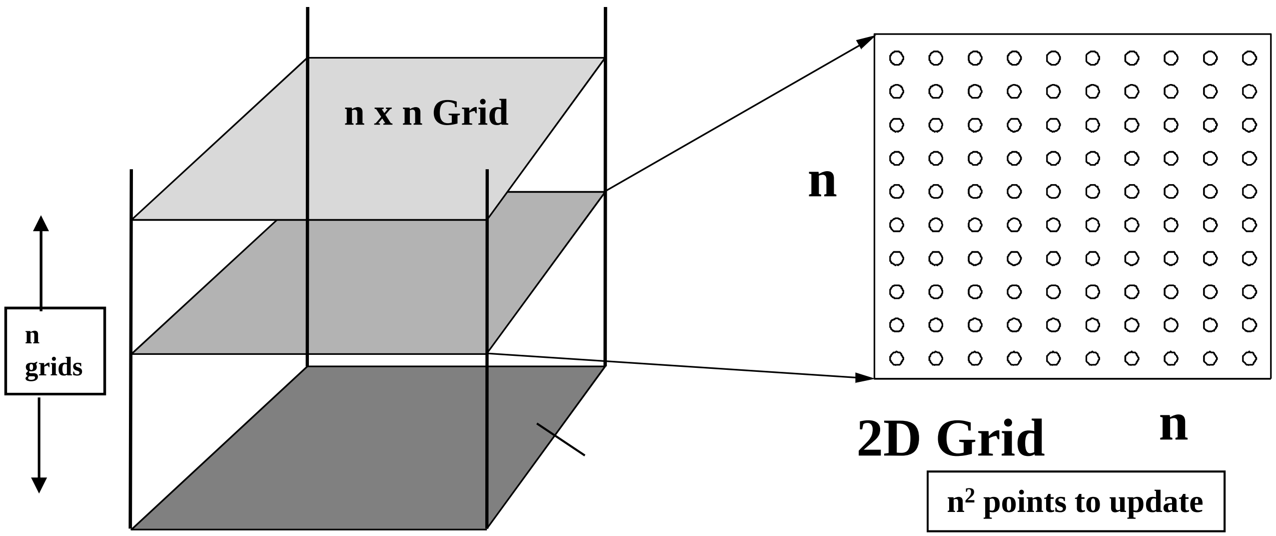
(PCA Chapter 2.3)

CMPE655 - Shaaban

Example Motivating Problem: Simulating Ocean Currents/Heat Transfer

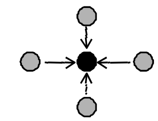
From last lecture

i.e. Grid Computations



(a) Cross sections

(b) Spatial discretization of a cross section



Expression for updating each interior point:

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

Updated Value

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 X n grids

- **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)

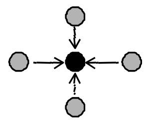
Solution of Linear System of Equation By Synchronous Iteration

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

For one 2D Grid (n x n = n^2 points)

Initialize all points

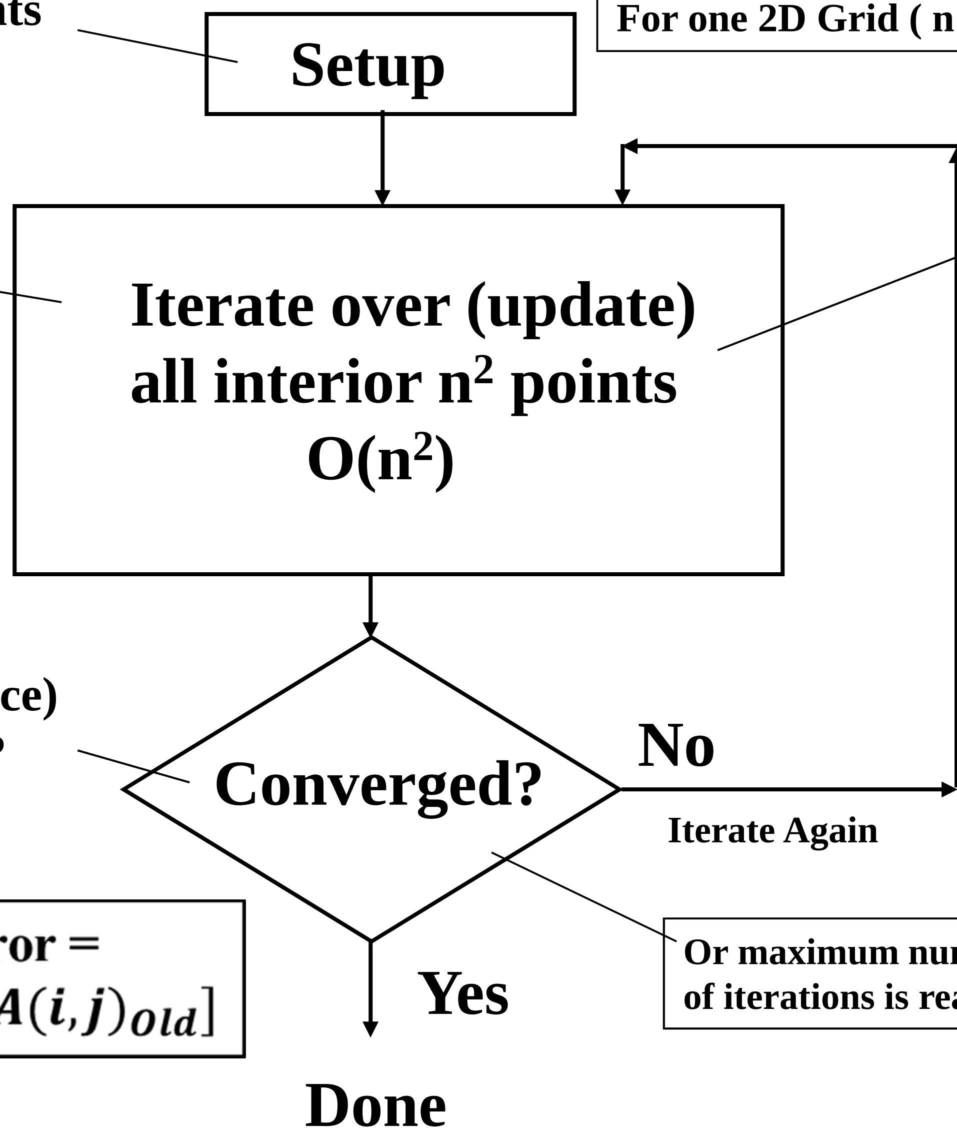
Point Update Expression



Expression for updating each interior point:

A[i,j] = 0.2 × (A[i,j] + A[i,j – 1] + A[i – 1, j] + A[i,j + 1] + A[i + 1, j])

New or updated point



One Iteration Or Sweep

Find Error (Global Difference)
Is Error < Tolerance Limit ?
(Threshold)

Or maximum number of iterations is reached

Global Difference or Error =

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

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 x n = n^2 points (instead of 3D – n grids)

Illustrate parallel program in low-level parallel language:

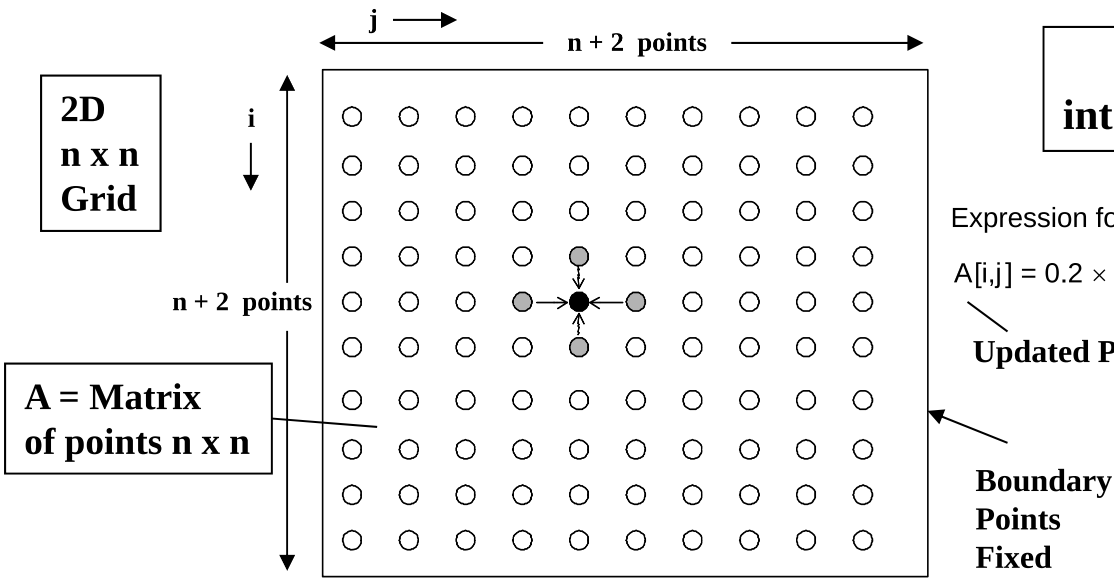
- C-like pseudo-code with **simple extensions for parallelism**
- Expose basic communication and synchronization primitives that must be supported by parallel programming model.

Three parallel programming models targeted for orchestration:

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

(PCA Chapter 2.3)

(One) 2D Grid Solver Example

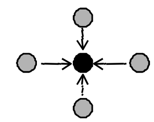


n^2 = n x n
interior grid points

Expression for updating each interior point:

A[i,j] = 0.2 × (A[i,j] + A[i,j – 1] + A[i – 1, j] + A[i,j + 1] + A[i + 1, j])

Updated Point



Computation = O(n^2)
per sweep or iteration

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

Iterate

Pseudocode, Sequential Equation Solver

Setup

Initialize grid points

Call equation solver

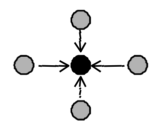
A = Matrix of n x n points

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

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

procedure Solve (A)                      /*solve the equation system*/
   float **A;                            /*A is an (n + 2)-by-(n + 2) array*/
begin
   int i, j, done = 0;
   float diff = 0, temp;
   while (!done) do                      /*outermost loop over sweeps*/
      diff = 0;                          /*initialize maximum difference to 0*/
      for i ← 1 to n do                  /*sweep over nonborder points of grid*/
         for j ← 1 to n do
            temp = A[i,j];               /*save old value of element*/
            A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
               A[i,j+1] + A[i+1,j]);     /*compute average*/
            diff += abs(A[i,j] - temp);
         end for
      end for
      if (diff/(n*n) < TOL) then done = 1;
   end while
end procedure
```

Iterate until convergence

i.e one iteration

Sweep O(n^2) computations



Update Points

New value

New value

Old value

Done?

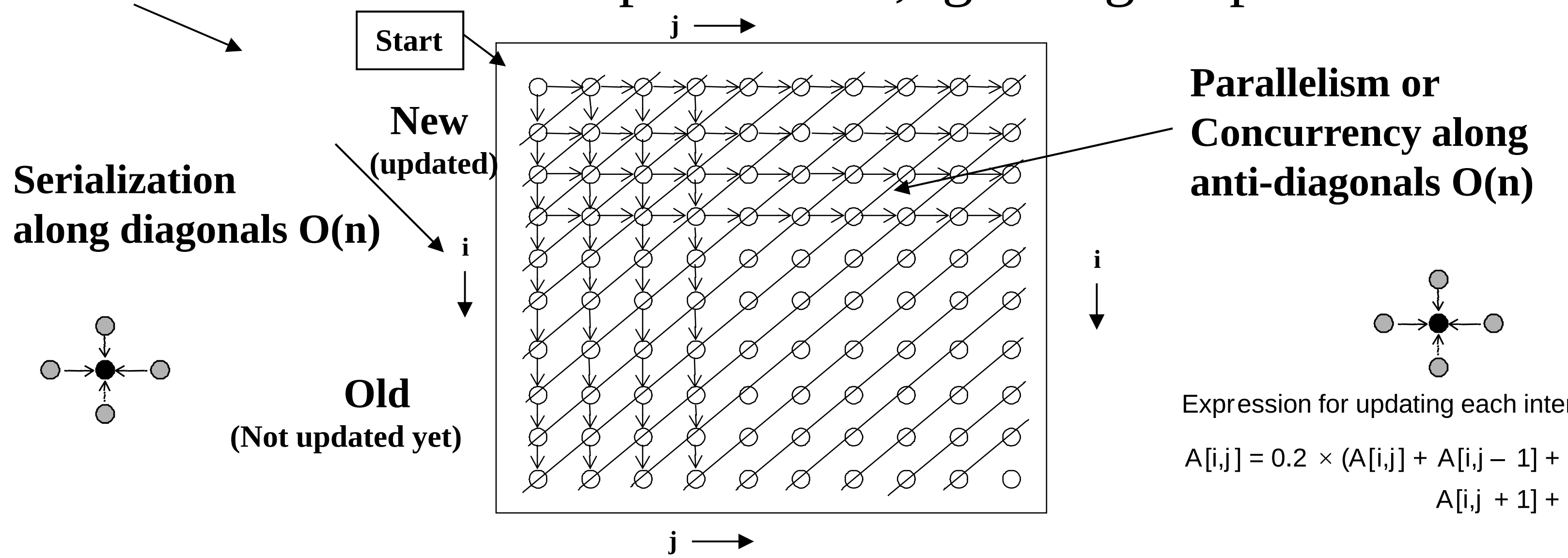
TOL, tolerance or threshold

diff = Global Difference

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

Decomposition **How?**

- Simple way to identify concurrency is to look at loop iterations 1 — Maximum
  - 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



Expression for updating each interior point:

A[i,j] = 0.2 × (A[i,j] + A[i,j – 1] + A[i – 1, j] + A[i,j + 1] + A[i + 1, j])

- Concurrency *O(n)* along anti-diagonals, serialization *O(n)* along diagonal
- Retain loop structure, use pt-to-pt synch; Problem: too many synch ops.

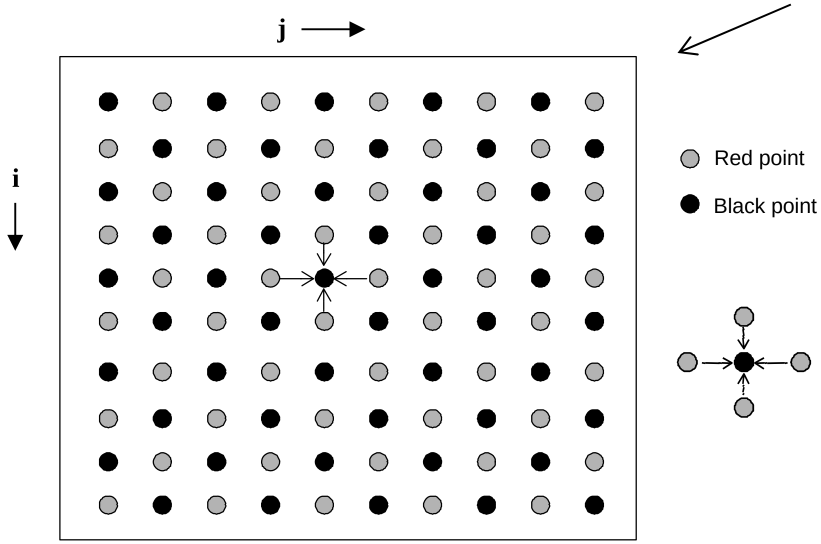
Or • Restructure loops, use global synch; load imbalance and too much synch

i.e using barriers along diagonals

Decomposition: Exploit Application Knowledge

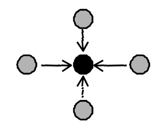
- Reorder grid traversal: red-black ordering

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**



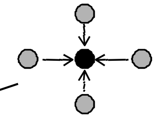
For PRAM with $O(n^2)$ processors:
Sweep = O(1)
Global Difference = $O(\log_2 n^2)$
Thus: $T = O(\log_2 n^2)$ **Per Iteration**

- 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



Task = Update one grid point

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.       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 = $O(n^2)$

Point Update

Parallel PRAM O(1)

$O(n^2)$ Parallel Computations (tasks)

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

Task = One row

Task = update one grid point

Degree of Parallelism (DOP)

- **<u>Decomposition into elements:</u>** degree of concurrency n^2
- **<u>To decompose into rows</u>**, 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

Task = update one row of points

<u>Fine Grain:</u>
n^2 parallel tasks each updates one element

DOP = $O(n^2)$

<u>Coarser Grain:</u>
n parallel tasks each update a row
<u>Task = grid row</u>
Computation = O(n)
Communication = O(n) ~ 2n
Communication to Computation ratio = O(1) ~ 2

The "for_all" loop construct imply parallel loop computations

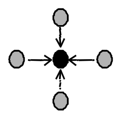
<u>**Assignment:**</u> (Update n/p rows per task)

i.e Task Assignment

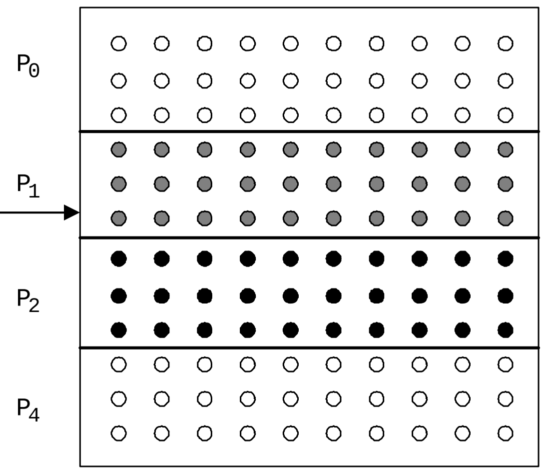
p = number of processes or processors

i.e n^2/p points per task

- **<u>Static assignments</u>** (given decomposition into rows)
  - <u>Block assignment of rows:</u> Row i is assigned to process $\left\lfloor \frac{i}{p} \right\rfloor$
  - <u>Cyclic assignment of rows:</u> 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 = $O(n)$ **~ 2n (2 rows)**
Communication-to-Computation

ratio = $O(n / (n^2/p)) = O(p/n)$

Lower C-to-C ratio is better

- **<u>Dynamic assignment (at runtime):</u>** **<u>Not needed here (computation is predictable)</u>**
  - 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 <u>**orchestration**</u> under three programming models:

1- Data Parallel **2- Shared Address Space (SAS)** **3- Message Passing**

Data Parallel Solver

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

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

procedure Solve(A)                               /*solve the equation system*/
    float **A;                                   /*A is an (n + 2-by-n + 2) array*/
  begin
  int i, j, done = 0;
  float mydiff = 0, temp;
    DECOMP A[BLOCK,*, nprocs];
  while (!done) do                               /*outermost loop over sweeps*/
    mydiff = 0;                                  /*initialize maximum difference to 0*/
    for_all i ← 1 to n do                        /*sweep over non-border points of grid*/
      for_all j ← 1 to n do
        temp = A[i,j];                           /*save old value of element*/
        A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
          A[i,j+1] + A[i+1,j]);                  /*compute average*/
        mydiff += abs(A[i,j] - temp);
      end for_all
    end for_all
      REDUCE (mydiff, diff, ADD);
    if (diff/(n*n) < TOL) then done = 1;
  end while
end procedure
```

nprocs = number of processes = p

Setup/Initialize Points

A = Matrix of n x n points

Block decomposition by row

n/p rows per processor

In Parallel

New Value

Sweep/Iteration: $T = O(n^2/p)$

mydiff = Local Difference

diff = Global Difference

Add all local differences (REDUCE) cost depends on architecture and implementation of REDUCE
best: $O(\log_2 p)$ using binary tree reduction
Worst: $O(p)$ sequentially

$$\boxed{O(n^2/p + \log_2 p) \le T(\text{iteration}) \le O(n^2/p + p)}$$

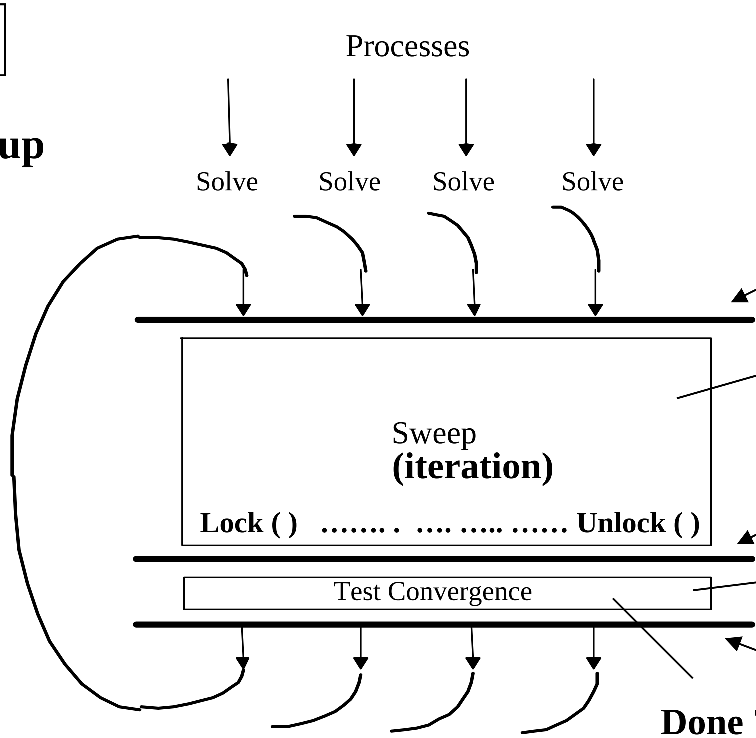
SAS Shared Address Space Solver

Single Program Multiple Data (SPMD) ← **Still MIMD**

Initialize Points

Setup

- **Array of grid points "A" in shared memory**
- **Diff = Global Difference or Error also in shared memory**



Barrier 1

n/p rows or n^2/p points per process or task

p tasks

Barrier 2

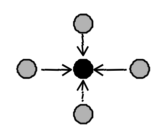
All processes test for convergence

Not Done? Sweep again

i.e iterate

Barrier 3

Done ?



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

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

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

Array "A" is shared (all grid points)

```
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*/

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

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

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);              /*update global diff if necessary*/
25b.    diff += mydiff;
25c.    UNLOCK(diff_lock);
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*/
25f.    BARRIER(bar1, nprocs);
26.   endwhile
27. end procedure
```

Main process or thread → main()

Setup

Create p-1 processes

More Setup

Which rows?

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

Private Variables

Barrier 1 → 16a.

(Start sweep)

Global Difference (Shared)

Critical Section

(sweep done)

MyDiff = Local Difference (Private)

Mutual Exclusion (lock) for global difference

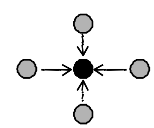
Critical Section: global difference

Barrier 2 → 25d.

Barrier 3 → 25f.

Done?

Sweep:
T = O(n^2/p)



T(p) = O(n^2/p + p)

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

Check/test convergence: all processes do it

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 (0, 1, 2, … p-1) 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**
  - **Why the need for all the barriers?**

Using LOCK ( ) …. UNLOCK ( )

In a critical section

Need for Mutual Exclusion

- Code each process executes:

diff = Global Difference

r1 , r2 are registers

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

Local Difference in r2

- A possible interleaving: i.e relative operations ordering in time

Time

P1	P2	
r1 ← diff		{P1 gets 0 in its r1}
	r1 ← diff (Load)	{P2 also gets 0}
r1 ← r1+r2		{P1 sets its r1 to 1}
	r1 ← r1+r2	{P2 sets its r1 to 1}
diff ← r1		{P1 sets cell_cost to 1}
	diff ← r1 (Store)	{P2 also sets cell_cost to 1}

diff = Global Difference (in shared memory)

r2 = mydiff = Local Difference

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

Fix ?

Mutual Exclusion

Lock $\xrightarrow{\text{Enter}}$ **Critical Section** $\xrightarrow{\text{Exit}}$ Unlock

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 locally:
  - 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

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

- Conservative form of preserving dependencies, but easy to use

Done by all processes

Most Critical Barrier?

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

SAS

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**

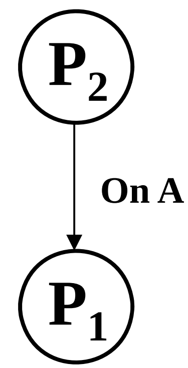
Initially flag = 0

Time

```
                  P1                                P2
                                             A = 1;      i.e P2 computed A
                                             b: flag
a: while (flag is 0) do nothing;             = 1;
print A;
```

Or compute using "A" as operand

i.e busy-wait (or spin on flag)



- *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**

  Each n/p rows in local memory

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

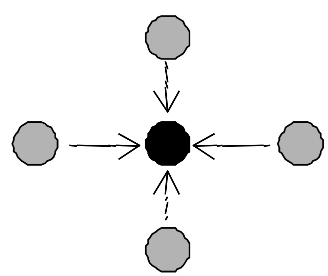
  n/p rows in this case **myA**

- 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

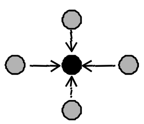
  } **Via Send/receive pairs** +

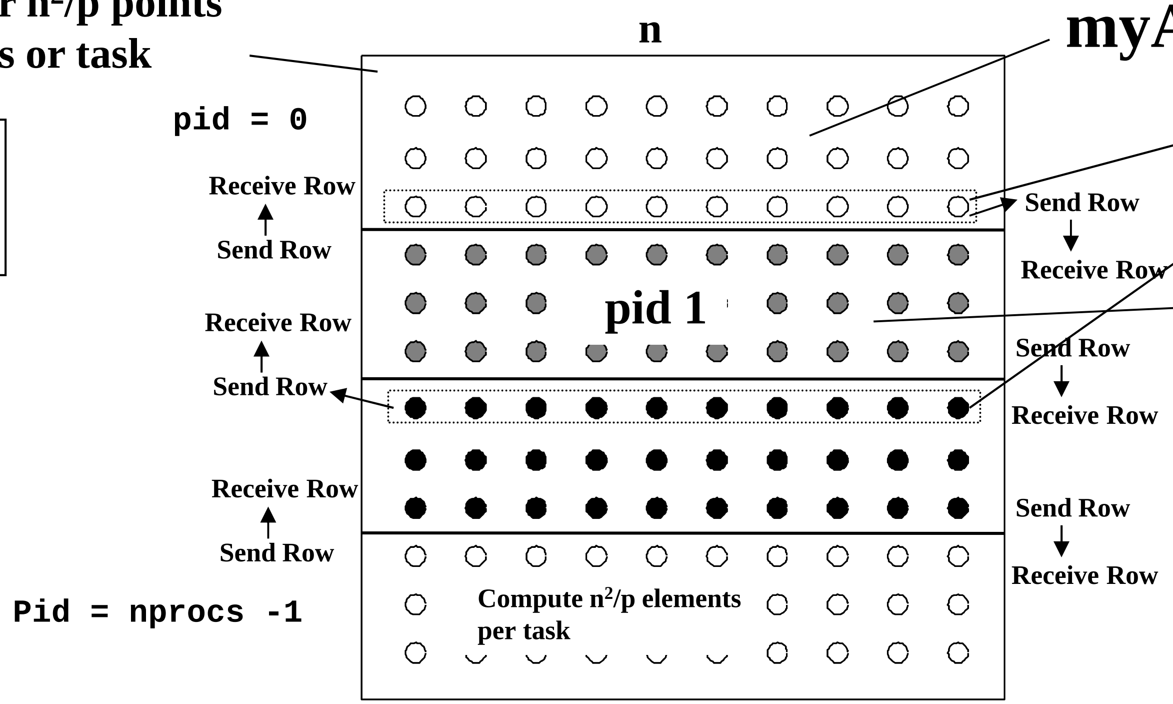


Message Passing Grid Solver

n/p rows or n^2/p points per process or task

Same block assignment as before





myA

Each n/p rows

Ghost (border) Rows for Task pid 1

myA For pid 1

As shown next slide

n/p rows per task

- 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(computation) + T(communication)
T = $O(n^2/p + n + p)$

nprocs = number of processes = number of processors = p

Pseudo-code, Parallel Equation Solver for Message Passing

of processors = p = nprocs

Message Passing

Create p-1 processes

Done by master process (pid =0)

Initialize local rows myA

Initialize myA (Local rows)

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

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

while (!done) do
     mydiff = 0;                    /*set local diff to 0*/
     if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
     if (pid = nprocs-1) then
        SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
     if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
     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,*]*/
     for i ← 1 to n' do             /*for each of my (nonghost) rows*/
        for j ← 1 to n do           /*for all nonborder elements in that row*/
           temp = myA[i,j];
           myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
              myA[i,j+1] + myA[i+1,j]);
           mydiff += abs(myA[i,j] - temp);
        endfor
     endfor
                                    /*communicate local diff values and determine if
                                      done; can be replaced by reduction and broadcast*/
        if (pid != 0) then                /*process 0 holds global total diff*/
           SEND(mydiff,sizeof(float),0,DIFF);
           RECEIVE(done,sizeof(int),0,DONE);
        else                              /*pid 0 does this*/
           for i ← 1 to nprocs-1 do       /*for each other process*/
              RECEIVE(tempdiff,sizeof(float),*,DIFF);
           mydiff += tempdiff;            /*accumulate into total*/
        endfor
        if (mydiff/(n*n) < TOL) then      done = 1;
           for i ← 1 to nprocs-1 do       /*for each other process*/
              SEND(done,sizeof(int),i,DONE);
           endfor
     endif
  endwhile
  end procedure
```

Send one or two ghost rows

Exchange ghost rows (send/receive)

Receive one or two ghost rows

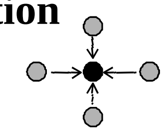
Communication $O(n)$ exchange ghost rows

Before start of iteration

+

Sweep over n/p rows $= n^2/p$ points per task $T = O(n^2/p)$

Computation $O(n^2/p)$



Update Points of myA

Local Difference

+

Send mydiff to pid 0

Receive test result from pid 0

Pid 0: Done?

calculate global difference and test for convergence

send test result to all processes

Pid 0

Only Pid 0 tests for convergence

$O(p)$ Communication

$T = O(n^2/p + n + p)$

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);
25c.    if (pid == 0) then
25i.      if (mydiff/(n*n) < TOL) then done = 1;
25k.    endif
25m.      BROADCAST(0,done,sizeof(int),DONE);
```

Compute global difference

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

Send/Receive

- Synchronous — **Send waits until message is actually received**
  - **Easy to create Deadlock**
- Asynchronous
  - Blocking
    - **Receive: Wait until message is received**
    - **Send: Wait until message is sent**
  - Non-blocking — **Immediate**
    - **Return immediately (both)**

- 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. <u>Both return immediately.</u>

MPI = Message Passing Interface

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

Task

Decomposition and **Assignment** **similar** in SAS and message-passing

i.e. n/p block of rows

Orchestration is different:

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

AKA shared?

	SAS	Msg-Passing
Explicit global data structure?	**Yes**	**No**
Assignment indept of data layout?	**Yes**	**No**
Communication	**Implicit**	**Explicit**
Synchronization	**Explicit**	**Implicit**
Explicit replication of border rows?	**No**	**Yes**

Lock/unlock Barriers

Via Send/ Receive Pairs

i.e. ghost rows

Ghost Rows

Requirements for performance are another story ...