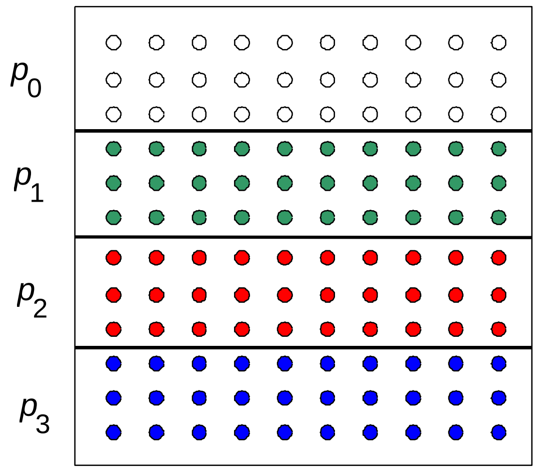
Assignment

How can we statically assign elements to processes?

- One option is “block assignment”—Row *i* is assigned to process $\lfloor i/p \rfloor$.



- Another option is “cyclic assignment—Process *i* is assigned rows *i*, *i*+*p*, *i*+2*p*, etc.
- Another option is 2D contiguous block partitioning.

We could instead use dynamic assignment, where a process gets an index, works on the row, then gets a new index, etc. Is there any advantage to this?

What are advantages and disadvantages of these partitionings?

Static assignment of rows to processes reduces concurrency

But block assignment reduces communication, by assigning adjacent rows to the same processor.

How many rows now need to be accessed from other processors?

So the communication-to-computation ratio is now only O(_____).

Orchestration

Once we move on to the orchestration phase, the computation model constrains our decisions.

Data-parallel model

In the code below, we assume that global declarations are used for shared data, and that any data declared within a procedure is private.

Global data is allocated with *g_malloc*.

Differences from sequential program:

- **for_all** loops
- **decomp** statement
- *mydiff* variable, private to each process
- **reduce** statement

```
int n, nprocs;                         /*grid size (n+2×n+2) and # of processes*/
double **A, diff = 0;

main()
begin
  read(n); read(nprocs);               /*read input grid size and # 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*/
    double **A;                        /* A is an (n+2×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
```

The **decomp** statement has a twofold purpose.

- It specifies the assignment of iterations to processes.

  The first dimension (rows) is partitioned into *nprocs* contiguous blocks. The second dimension is not partitioned at all.

  Specifying **`[CYCLIC, *, nprocs]`** would have caused a cyclic partitioning of rows among **`nprocs`** processes.

  Specifying **`[*,CYCLIC, nprocs]`** would have caused a cyclic partitioning of columns among **`nprocs`** processes.

  Specifying **`[BLOCK, BLOCK, nprocs]`** would have implied a 2D contiguous block partitioning.

  For all of these partitionings, tell which processing element in a 64-PE system would compute A[33, 65].

- It specifies the assignment of grid data to memories on a distributed-memory machine. (Follows the *owner-computes* rule.)

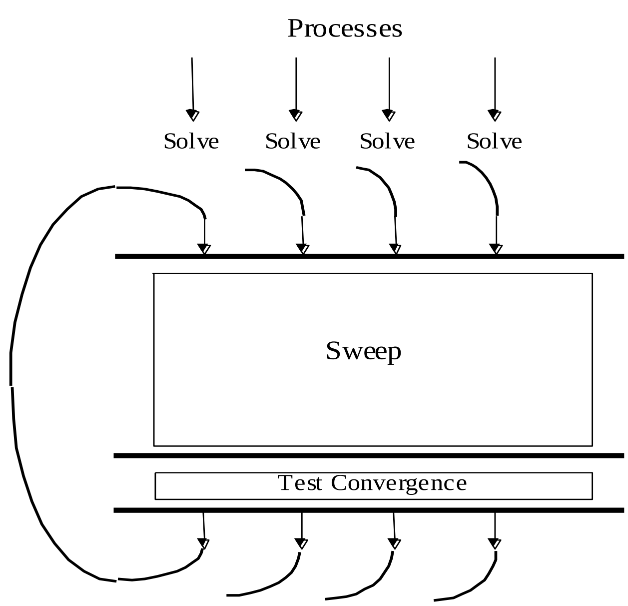
The *mydiff* variable allows local sums to be computed.

The **reduce** statement tells the system to add together all the *mydiff* variables into the shared *diff* variable.

Shared-memory model

In this model, we need mechanisms to create processes and manage them.

After we create the processes, they interact as shown on the right.



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

main()
begin
    read(n); read(nprocs);       /*read input matrix size and number of processes */
    A ← -     (a two-dimensional array of size n+2 by n+2 doubles);
    initialize(A);               /*initialize A in an unspecified way*/
    CREATE (nprocs–1, Solve, A);
  Solve(A);                      /*main process becomes a worker
    WAIT_FOR_END (nprocs–1);     /*wait for all child processes created to terminate*/
end main

procedure Solve(A)
  double**A;                     /*A is entire n+2-by-n+2 shared array,
                                 as in the sequential program*/
begin
  int i,j, pid, done = 0;
  float temp, mydiff = 0;        /*private variables*/
  int mymin = 1 + (pid * n/nprocs);  /*assume that n is exactly divisible by*/
  int mymax = mymin + n/nprocs - 1   /*nprocs for simplicity here*/

  while (!done) do               /* outer loop over all diagonal elements*/
    mydiff = diff = 0;           /*set global diff to 0 (okay for all to do it)*/
  BARRIER(bar1, nprocs);         /*ensure all reach here before anyone modifies diff*/
    for i ← mymin to mymax do    /*for each of my rows */
        for j ← 1 to n do        /*for all nonborder elements in that row*/
        temp = A[i,j];
        A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
            A[i,j+1] + A[i+1,j]);
        mydiff += abs(A[i,j] - temp);
      endfor
    endfor
    LOCK(diff_lock);             /*update global diff if necessary*/
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER(bar1, nprocs);       /*ensure all reach here before checking if done*/
    if (diff/(n*n) < TOL) then done = 1;   /*check convergence; all get
                                           same answer*/
    BARRIER(bar1, nprocs);
  endwhile
end procedure
```

What are the main differences between the serial program and this program?

- The first process creates *nprocs*–1 worker processes. All *n* processes execute *Solve*.

  All processes execute the same code.

  But all do *not* execute the same instructions at the same time.

- Private variables like *mymin* and *mymax* are used to control loop bounds.
- All processors need to—

- complete an iteration before any process tests for convergence. Why?
- test for convergence before any process starts the next iteration. Why?

Notice the use of *barrier synchronization* to achieve this.

What could happen if the barrier at Line 16a was removed?

What could happen if the barrier at Line 25d was removed?

What could happen if the barrier at Line 25f was removed?

- Locks must be plsaced around updates to *diff*, so that no two processors update it at once. Otherwise, inconsistent results could ensue.

p_1	p_2	
`r1 ← diff`		{ p_1 gets 0 in its `r1`}
	`r1 ← diff`	{ p_2 also gets 0}
`r1 ← r1+r2`		{ p_1 sets its `r1` to 1}
	`r1 ← r1+r2`	{ p_2 sets its `r1` to 1}
`diff ← r1`		{ p_1 sets *diff* to 1}
	`diff ← r1`	{ p_2 also sets *diff* to 1}

If we allow only one processor at a time to access *diff*, we can avoid this *race condition*.

What is one performance problem with using locks?

Note that at least some processors need to access *diff* as a non-local variable.

What is one technique that our shared-memory program uses to diminish this problem of serialization?

Message-passing model

The program for the message-passing model is also similar, but again there are several differences.

- There's no shared address space, so we can't declare array *A* to be shared.

  Instead, each processor holds the rows of *A* that it is working on.

- The subarrays are of size $(n/nprocs + 2) \times (n + 2)$.
  This allows each subarray to have a copy of the boundary rows from neighboring processors. Why is this done?

  These *ghost* rows must be copied explicitly, via **send** and **receive** operations.

  Note that **send** is not synchronous; that is, it doesn't make the process wait until a corresponding **receive** has been executed.

  What problem would occur if it did?

- Since the rows are copied and then not updated by the processors they have been copied from, the boundary values are more out-of-date than they are in the sequential version of the program.

  This may or may not cause more sweeps to be needed for convergence.

- The indexes used to reference variables are *local* indexes, not the "real" indexes that would be used if array *A* were a single shared array.

```
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

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*/

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

There are one or more typos in the **if** statements involving **pid**s. Which statement(s)? What are the error(s)?