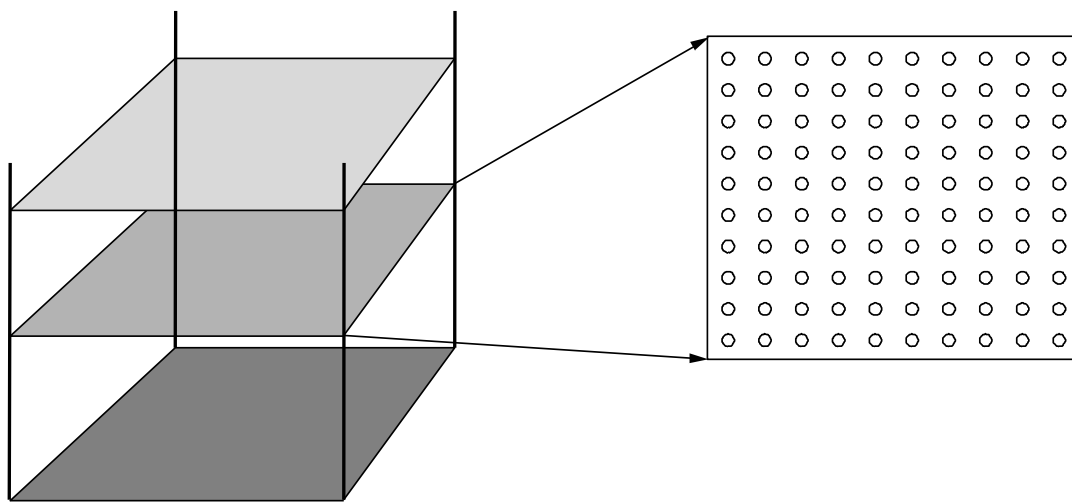


Simulating ocean currents

We will study a parallel application that simulates ocean currents.

Goal: Simulate the motion of water currents in the ocean. Important to climate modeling.

Motion depends on atmospheric forces, friction with ocean floor, and “friction” with ocean walls.



(a) Cross sections

To predict the state of the ocean at any instant, we need to solve complex systems of equations.

The problem is *continuous* in both space and time.
But to solve it, we *discretize* it over both dimensions.

Every important variable, e.g.,

- pressure
- velocity
- currents

has a value at each grid point.

This model uses a set of 2D horizontal cross-sections through the ocean basin.

Equations of motion are solved at all the grid points in one time-step.

- The state of the variables is updated, based on this solution.
- The equations of motion are solved for the next time-step.

Tasks

The first step is to divide the work into *tasks*.

- A task is an arbitrarily defined portion of work.
- It is the smallest unit of concurrency that the program can exploit.

Example: In the ocean simulation, a task can be computations on—

- a single grid point,
- a row of grid points, or
- any arbitrary subset of the grid.

Tasks are chosen to match some natural granularity in the work.

- If the grain is small, the decomposition is called _____.
- If it is large, the decomposition is called _____.

Threads

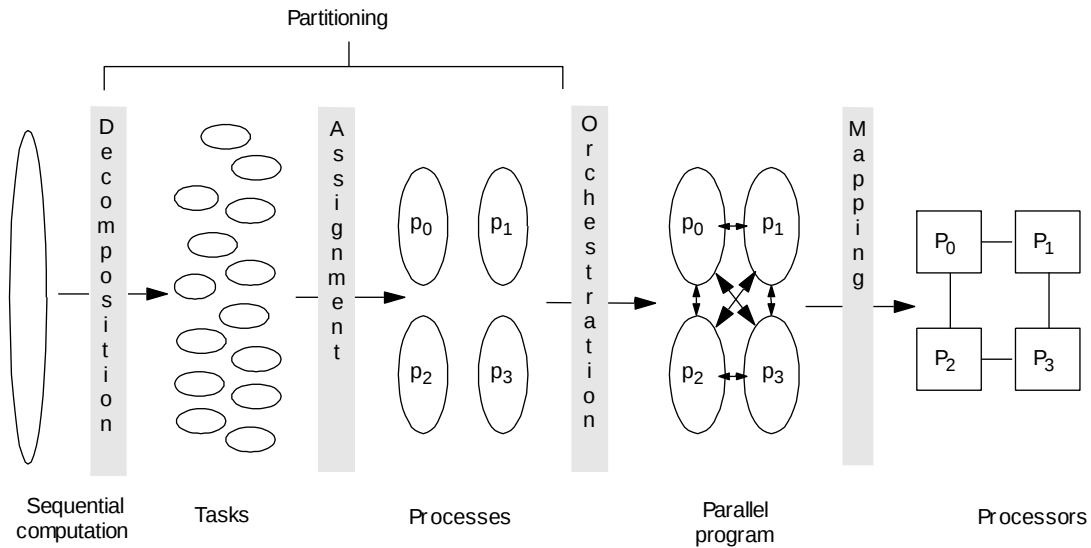
A *thread* is an abstract entity that performs tasks.

- A program is composed of cooperating threads.
- Each thread is assigned to a processor.
- Threads need not correspond 1-to-1 with processors!

Example: In the ocean simulation, an equal number of rows may be assigned to each thread.

Four steps in parallelizing a program:

- *Decomposition* of the computation into tasks.
- *Assignment* of tasks to threads.
- *Orchestration* of the necessary data access, communication, and synchronization among threads.
- *Mapping* of threads to processors.



Together, decomposition and assignment are called *partitioning*.

They break up the computation into tasks to be divided among threads.

The number of tasks available at a time is an upper bound on the achievable parallelism.

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Parallelization of an Example Program

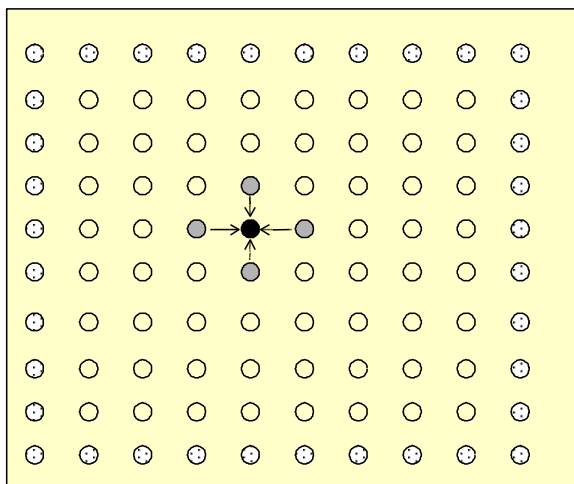
[§2.3] In this lecture, we will consider a parallelization of the kernel of the Ocean application.

The serial program

The equation solver solves a PDE on a grid.

It operates on a regular 2D grid of $(n+2)$ by $(n+2)$ elements.

- The *boundary elements* in the border rows and columns do not change.
- The interior n -by- n points are updated, starting from their initial values.



Expression for updating each interior point:

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

- The old value at each point is replaced by the weighted average of itself and its 4 nearest-neighbor points.
- Updates are done from left to right, top to bottom.
 - The update computation for a point sees the new values of points above and to the left, and
 - the old values of points below and to the right.

This form of update is called the Gauss-Seidel method.

During each sweep, the solver computes how much each element has changed since the last sweep.

- If this difference is less than a “tolerance” parameter, the solution has converged.
- If so, we exit solver; if not, we do another sweep.

Here is the code for the solver.

```

1. int n;                                /*size of matrix: (n + 2)-by-(n + 2) elements*/
2. double **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.  double **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.          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

```

Decomposition

A simple way to identify concurrency is to look at loop iterations.

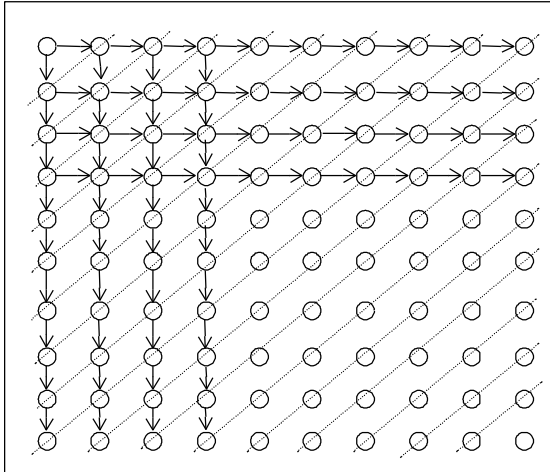
Is there much concurrency in this example? Does the algorithm let us perform more than one sweep concurrently?

Note that—

- Computation proceeds from left to right and top to bottom.

- Thus, to compute a point, we use
 - the updated values from the point above and the point to the left, but
 - the “old” values of the point itself and its neighbors below and to the right.

Here is a diagram that illustrates the dependences.



The horizontal and vertical lines with arrows indicate dependences.

The dashed lines along the antidiagonal connect points with no dependences that can be computed in parallel.

Of the $O(\text{---})$ work in each sweep, $\exists$ concurrency proportional to --- along antidiagonals.

How could we exploit this parallelism?

- We can *leave loop structure alone* and let loops run in parallel, inserting *synchronization ops* to make sure a value is computed before it is used.

Why isn't this a good idea?

- We can *change the loop structure*, making
 - the outer **for** loop (line 17) iterate over anti-diagonals, and
 - the inner **for** loop (line 18) iterate over elements within an antidiagonal.

Why isn't this a good idea?

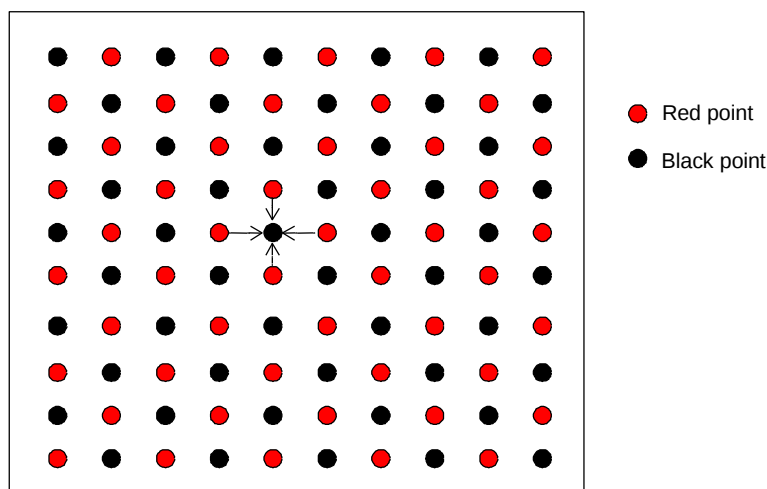
The Gauss-Seidel algorithm doesn't *require* us to update the points from left to right and top to bottom.

It is just a convenient way to program on a uniprocessor.

We can compute the points in another order, as long as we use updated values frequently enough (if we don't, the solution will converge, but more slowly).

Red-black ordering

Let's divide the points into alternating "red" and "black" points:



To compute a red point, we don't need the updated value of any other red point. But we need the updated values of 2 black points.

And similarly for computing black points.

Thus, we can divide each sweep into two phases.

- First we compute all red points.
- Then we compute all black points.

True, we don't use any updated black values in computing red points.

But we use *all* updated red values in computing black points.

Whether this converges more slowly or faster than the original ordering depends on the problem.

But it does have important advantages for parallelism.

- How many red points can be computed in parallel?
- How many black points can be computed in parallel?

Red-black ordering is effective, but it doesn't produce code that can fit on a single display screen.

A simpler decomposition

Another ordering that is simpler but still works reasonably well is just to ignore dependences between grid points within a sweep.

A sweep just updates points based on their nearest neighbors, regardless of whether the neighbors have been updated yet.

Global synchronization is still used between sweeps, however.

Now execution is no longer deterministic.

The number of sweeps needed, and the results, may depend on the number of processors used.

But for most reasonable assignments of processors, the number of sweeps will not vary much.

Let's look at the code for this.

```
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
```

The *only* difference is that **for** has been replaced by **for_all**.

A **for_all** just tells the system that all iterations can be executed in parallel.

With **for_all** in both loops, all n^2 iterations of the nested loop can be executed in parallel.

We could write the program so that the computation of one row of grid points must be assigned to a single processor. How would we do this?

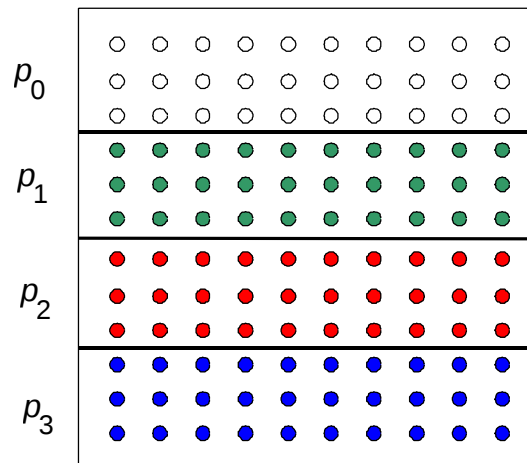
With each row assigned to a different processor, each task has to access about $2n$ grid points that were computed by other processors; meanwhile, it computes n grid points itself.

So the communication-to-computation ratio is $O(1)$.

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 $i, i+p, i+2p$, 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 $O(\text{_____})$.

Orchestration

Once we move on to the orchestration phase, the computation model affects 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

```

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

3.  main()
4.  begin
5.      read(n); read(nprocs);                    /*read input grid size and # 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

10. procedure Solve(A)                             /*solve the equation system*/
11.     double **A;                                  /* A is an (n+2×n+2) array*/
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
24a.        REDUCE (mydiff, diff, ADD);
25.         if (diff/(n*n) < TOL) then done = 1;
26.     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.

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

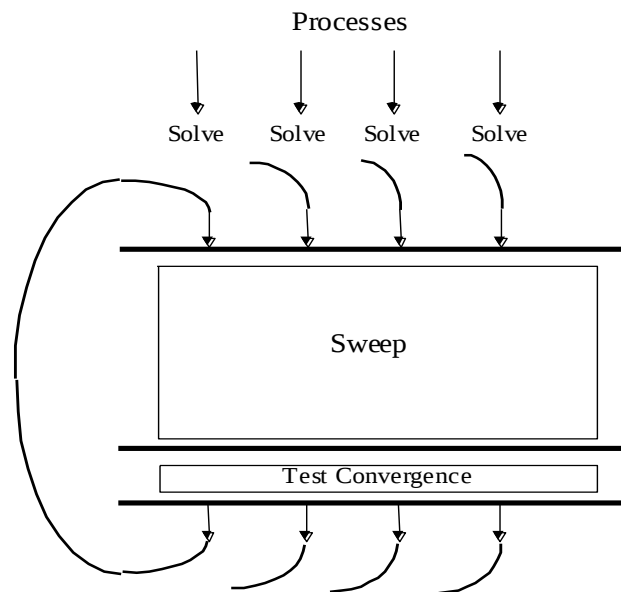
The *mydiff* variable allows local sums to be computed.

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.



```

1.      int n, nprocs;           /*matrix dimension and number of processors to be used*/
2a.     double**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
8b.     WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
9.      end main

10.     procedure Solve(A)
11.         double**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              /*update global diff if necessary*/
25a.        LOCK(diff_lock);
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

```

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.

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

<u>p_1</u>	<u>p_2</u>
$r1 \leftarrow diff$	{ p_1 gets 0 in its $r1$ }
	$r1 \leftarrow diff$
	{ p_2 also gets 0}
$r1 \leftarrow r1+r2$	{ p_1 sets its $r1$ to 1}
	$r1 \leftarrow r1+r2$
	{ p_2 sets its $r1$ to 1}
$diff \leftarrow r1$	{ p_1 sets $diff$ to 1}
	$diff \leftarrow r1$
	{ 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*/
6a.    CREATE (nprocs-1, Solve);
6b.    Solve();                                  /*main process becomes a worker too*/
6c.    WAIT_FOR_END (nprocs-1);                  /*wait for all child processes created to terminate*/
7. end main

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

16. while (!done) do
17.     mydiff = 0;                                /*set local diff to 0*/
18a.    if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
18b.    if (pid != nprocs-1) then
19.        SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
20c.    if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
20d.    if (pid != nprocs-1) then
21.        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,*]*/
22. for i ← 1 to n' do                            /*for each of my (nonghost) rows*/
23.     for j ← 1 to n do                          /*for all nonborder elements in that row*/
24.         temp = myA[i,j];
25.         myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
26.             myA[i,j+1] + myA[i+1,j]);
27.         mydiff += abs(myA[i,j] - temp);
28.     endfor
29. endfor

                                                /*communicate local diff values and determine if
                                                done; can be replaced by reduction and broadcast*/
30a.    if (pid != 0) then                        /*process 0 holds global total diff*/
31b.        SEND(mydiff,sizeof(float),0,DIFF);
31c.        RECEIVE(done,sizeof(int),0,DONE);
31d.    else                                        /*pid 0 does this*/
32e.        for i ← 1 to nprocs-1 do              /*for each other process*/
32f.            RECEIVE(tempdiff,sizeof(float),*,DIFF);
32g.            mydiff += tempdiff;                /*accumulate into total*/
32h.        endfor
32i.        if (mydiff/(n*n) < TOL) then          done = 1;
32j.            for i ← 1 to nprocs-1 do          /*for each other process*/
32k.                SEND(done,sizeof(int),i,DONE);
32l.            endfor
32m.        endif
33. endwhile
34. end procedure

```