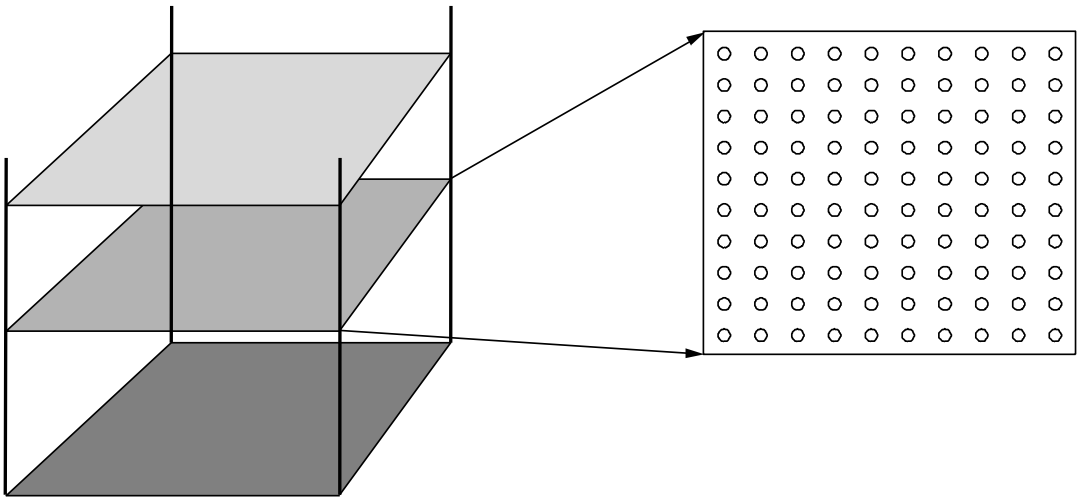
Simulating ocean currents

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 of the work.

- If large number of tasks, the decomposition is called _____.
- If small number of tasks, the decomposition is called _____.

Threads

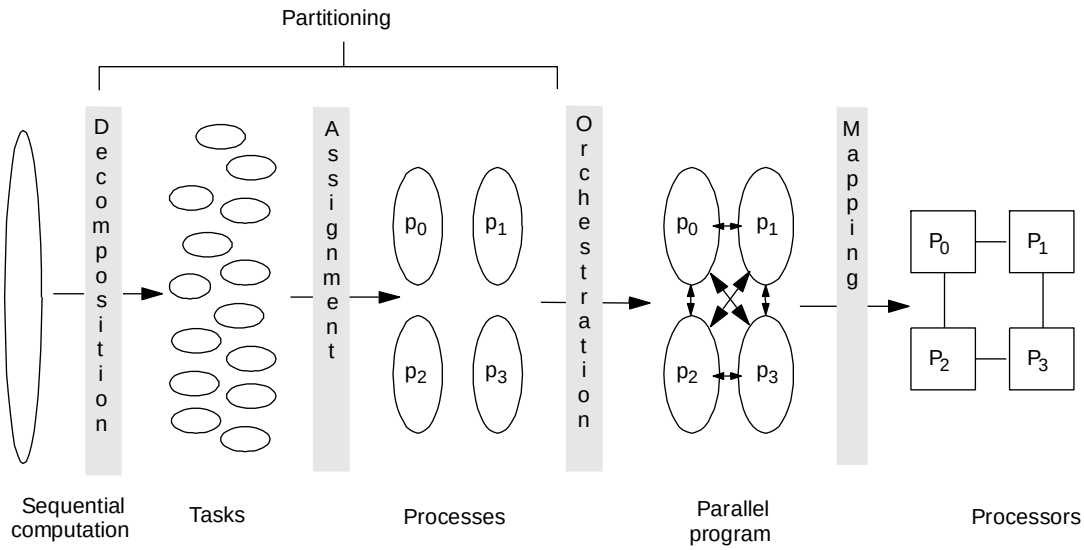
A *thread* is an abstract entity that performs tasks.

- 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 processor.

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

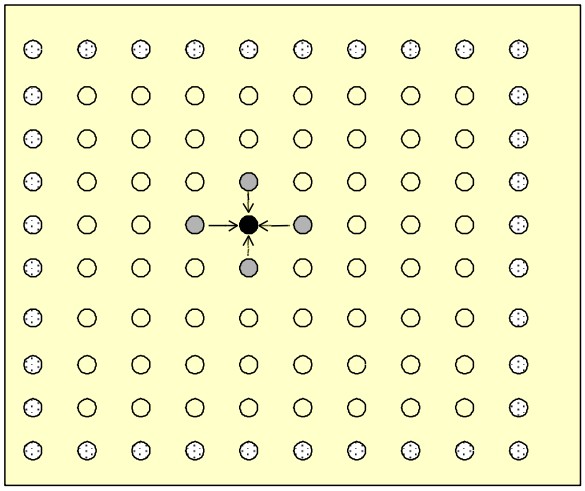
We will consider a parallelization of the kernel of the Ocean application.

The serial program

The equation solver soles 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.

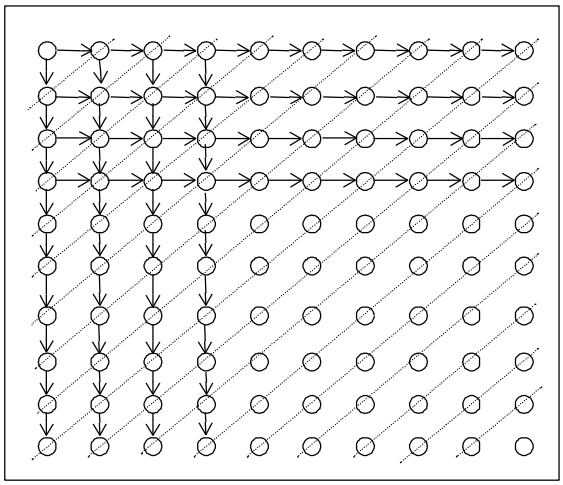
Is there much concurrency in this example? Does the algorithm let us perform more than one sweep concurrently?

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.

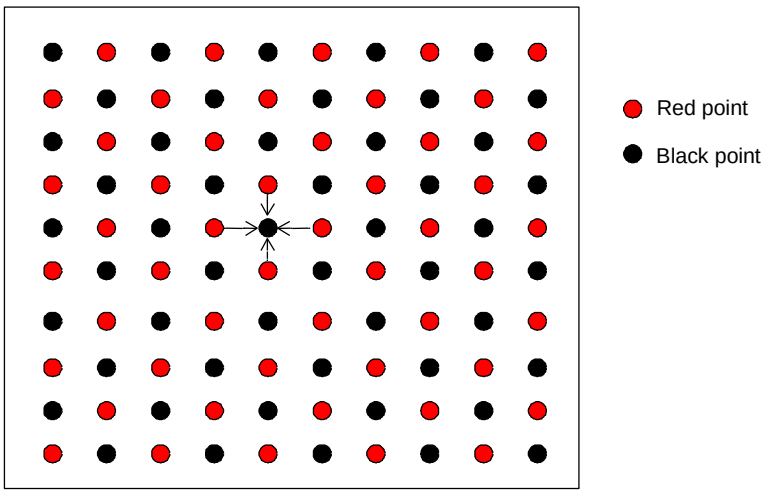
The Gauss-Seidel algorithm doesn’t *require* us to update the points from left to right and top to bottom.

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.

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 2*n* grid points that were computed by other processors; meanwhile, it computes *n* grid points itself.