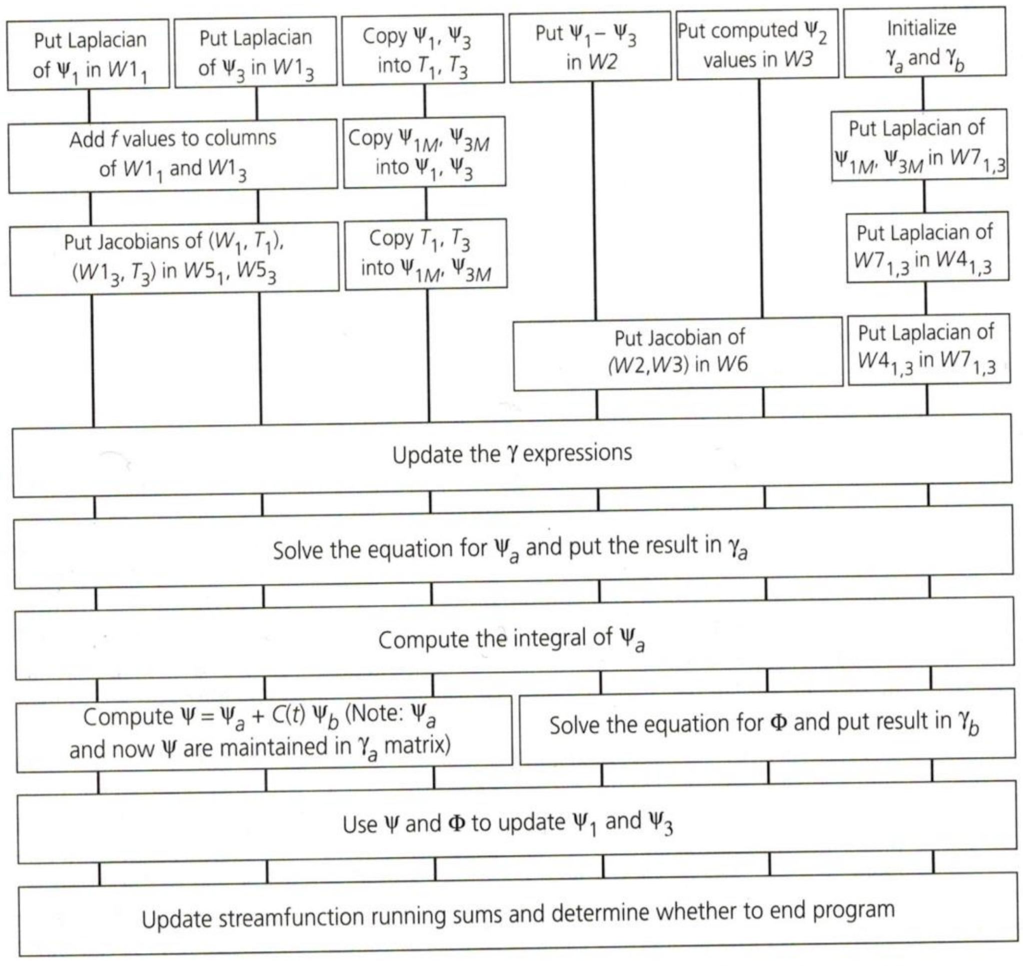
Simulating ocean currents

We will study a parallel application that simulates ocean currents.

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

The overall structure of the program looks like this:



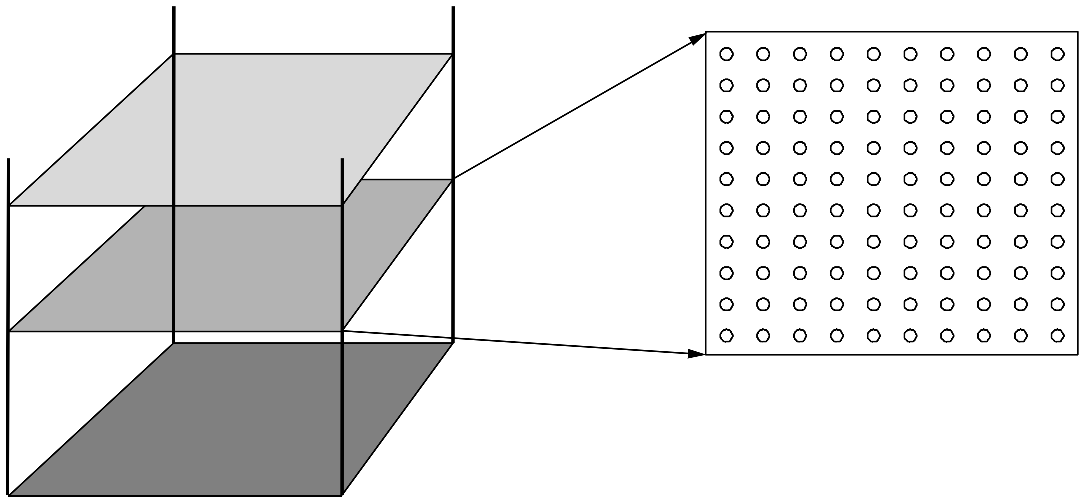
Put Laplacian of Ψ_1 in $W1_1$
Put Laplacian of Ψ_3 in $W1_3$
Copy Ψ_1, Ψ_3 into T_1, T_3
Put $\Psi_1 - \Psi_3$ in $W2$
Put computed Ψ_2 values in $W3$
Initialize γ_a and γ_b
Add f values to columns of $W1_1$ and $W1_3$
Copy Ψ_{1M}, Ψ_{3M} into Ψ_1, Ψ_3
Put Laplacian of Ψ_{1M}, Ψ_{3M} in $W7_{1,3}$
Put Jacobians of (W_1, T_1), ($W1_3$, T_3) in $W5_1$, $W5_3$
Copy T_1, T_3 into Ψ_{1M}, Ψ_{3M}
Put Laplacian of $W7_{1,3}$ in $W4_{1,3}$
Put Jacobian of ($W2$,$W3$) in $W6$
Put Laplacian of $W4_{1,3}$ in $W7_{1,3}$
Update the γ expressions
Solve the equation for Ψ_a and put the result in γ_a
Compute the integral of Ψ_a
Compute $\Psi = \Psi_a + C(t)\,\Psi_b$ (Note: Ψ_a and now Ψ are maintained in γ_a matrix)
Solve the equation for Φ and put result in γ_b
Use Ψ and Φ to update Ψ_1 and Ψ_3
Update streamfunction running sums and determine whether to end program

FIGURE 3.14 Ocean: The phases in a time-step and the dependences among grid computations. Each box is a grid computation (or pair of similar computations). Computations connected by vertical lines are dependent while others, such as those in the same row, are independent. The parallel program treats each horizontal row as a phase and synchronizes between phases.

The program offers opportunities for function parallelism (______________________) and data parallelism (______________________).

We will concentrate on solving the equation for ψ_a (data parallelism).

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

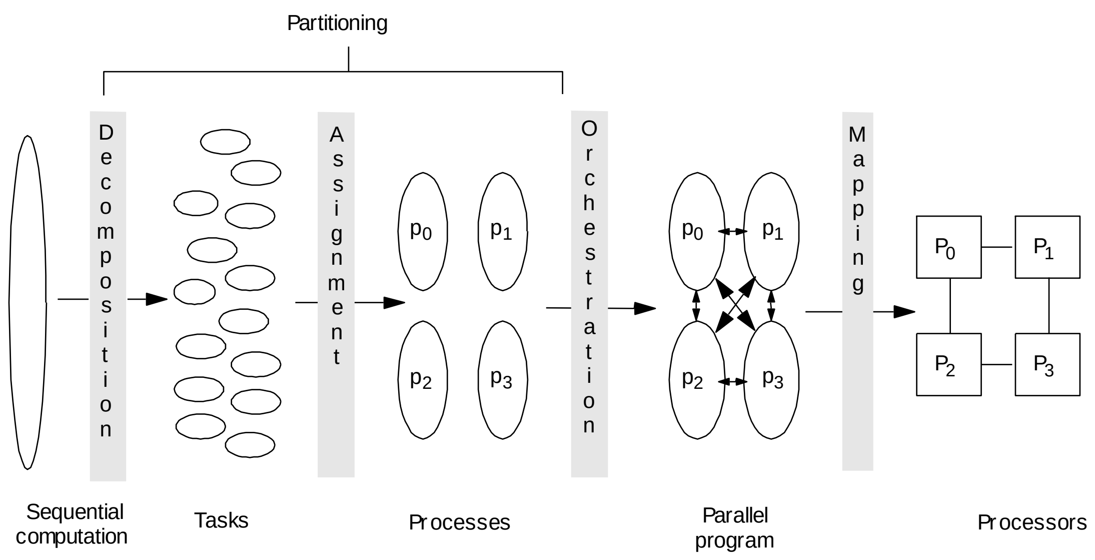
A *thread* is an abstract entity that performs tasks.

- 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

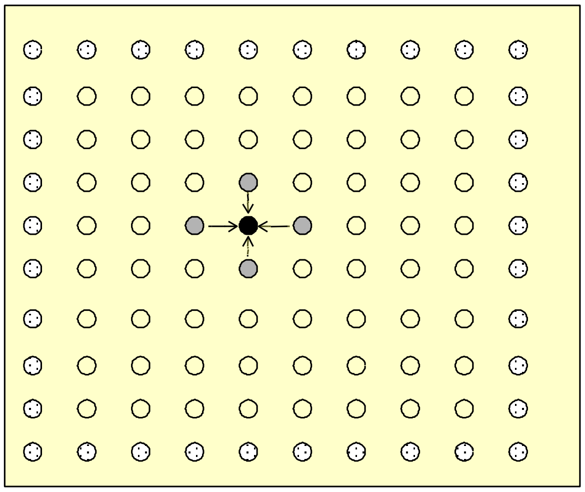
[§2.3] In this lecture, we will consider a parallelization of the kernel of the Ocean application.

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 the sum of these differences 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
```

Answer these questions about the solver.

Why is the array size (n+2)×(n+2) rather than n×n?

Why is it necessary to use a *temp* variable?

Why is the denominator in Line 25 n*n?

Decomposition

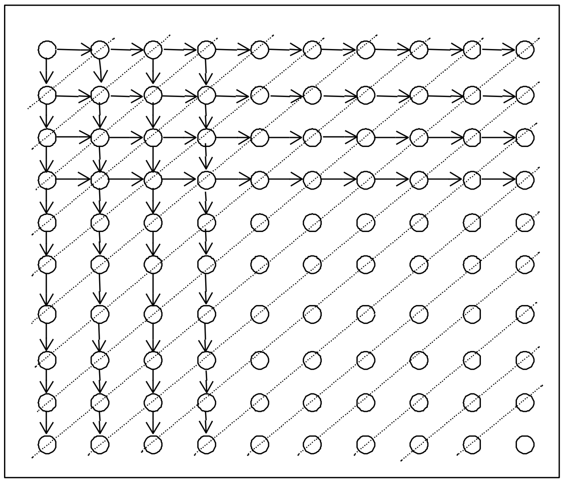
A simple way to identify concurrency is to look at loop iterations.

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.

Check: If `A[3,4]` is being computed, which updated values are used in the calculation?

Which of the following points can be updated in parallel?

Of the *O*(___) work in each sweep, ∃ concurrency proportional to along antidiagonals. (Give your answer in terms of *n*; how many points along an antidiagonal can be computed in parallel?)

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?

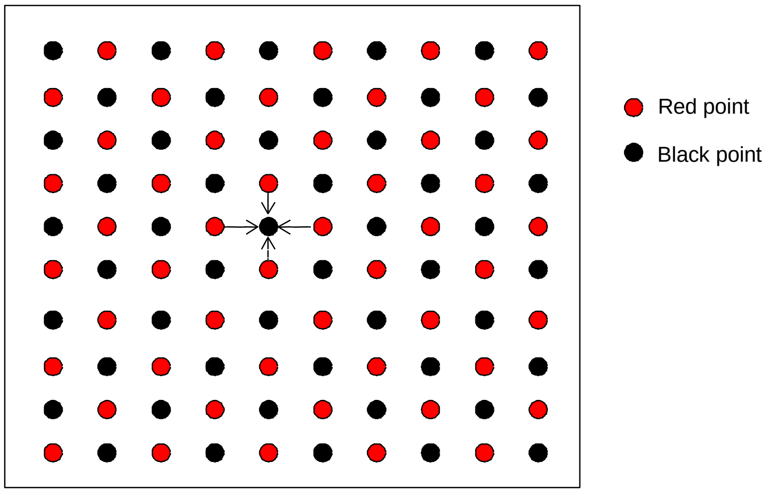
The Gauss-Seidel algorithm doesn't *require* us to update the points from left to right and top to bottom.

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.

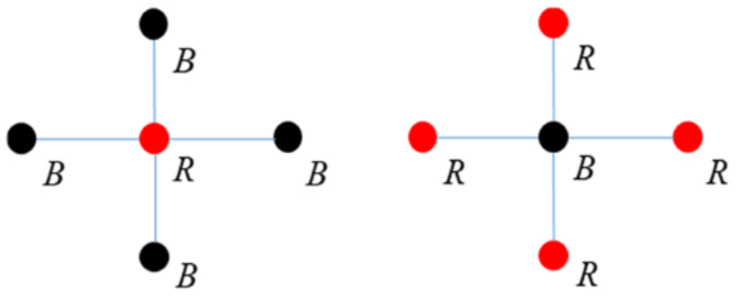
And similarly for computing black points.

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.

- Which points can be computed in parallel?
- Altogether, 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. (Does this matter?)

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.

```
while (!done) do                    /*a sequential loop*/
  diff = 0;
  for_all i ← 1 to n do             /*a parallel loop nest*/
    for_all j ← 1 to n do
      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]);
      diff += abs(A[i,j] - temp);
    end for_all
  end for_all
  if (diff/(n*n) < TOL) then done = 1;
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?

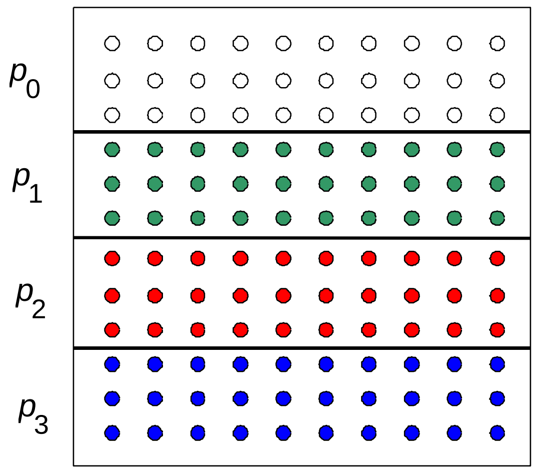
With each row assigned to a different processor, each task has to access about 2*n* grid points that were computed by other processors; meanwhile, it computes *n* grid points itself.

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*+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*(_____).