High Performance Computing
Lecture 41

Matthew Jacob

Indian Institute of Science

Example: MPI Pi Calculating Program

/Each process initializes, determines the communicator size and its own rank

```
MPI_Init (&argc, &argv);
MPI_Comm_size ( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank ( MPI_COMM_WORLD, &myid);
```

/The master process (P_0) takes input from the user

```
if (myid == 0){
   printf("Enter the number of intervals");
   scanf("%d", &n);
}
```

/The master process broadcasts the value of n

```
MPI_Bcast (&n,1,MPI_INT,0, MPI_COMM_WORLD);
```

Example: MPI Pi Calculating Program

```
if (n == 0) {/* master process */}
else { /* each slave process does some work */
   h = 1.0 / (double) n;
   sum = 0.0;
   for (i = myid+1; i <= n; i += numprocs) {
         x = h * ((double) i - 0.5);
         sum += (4.0 / (1.0 + x*x));
   }
   mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1,MPI_DOUBLE, MPI_SUM,
   0, MPI_COMM_WORLD);
}
MPI_Finalize();
```

Parallelizing a Program

Given a sequential program/algorithm, how to go about producing a parallel version

Four steps in program parallelization

1. Decomposition

   Identifying parallel tasks with large extent of possible parallel activity

2. Assignment

   Grouping the tasks into processes with best load balancing

3. Orchestration

   Reducing synchronization and communication costs

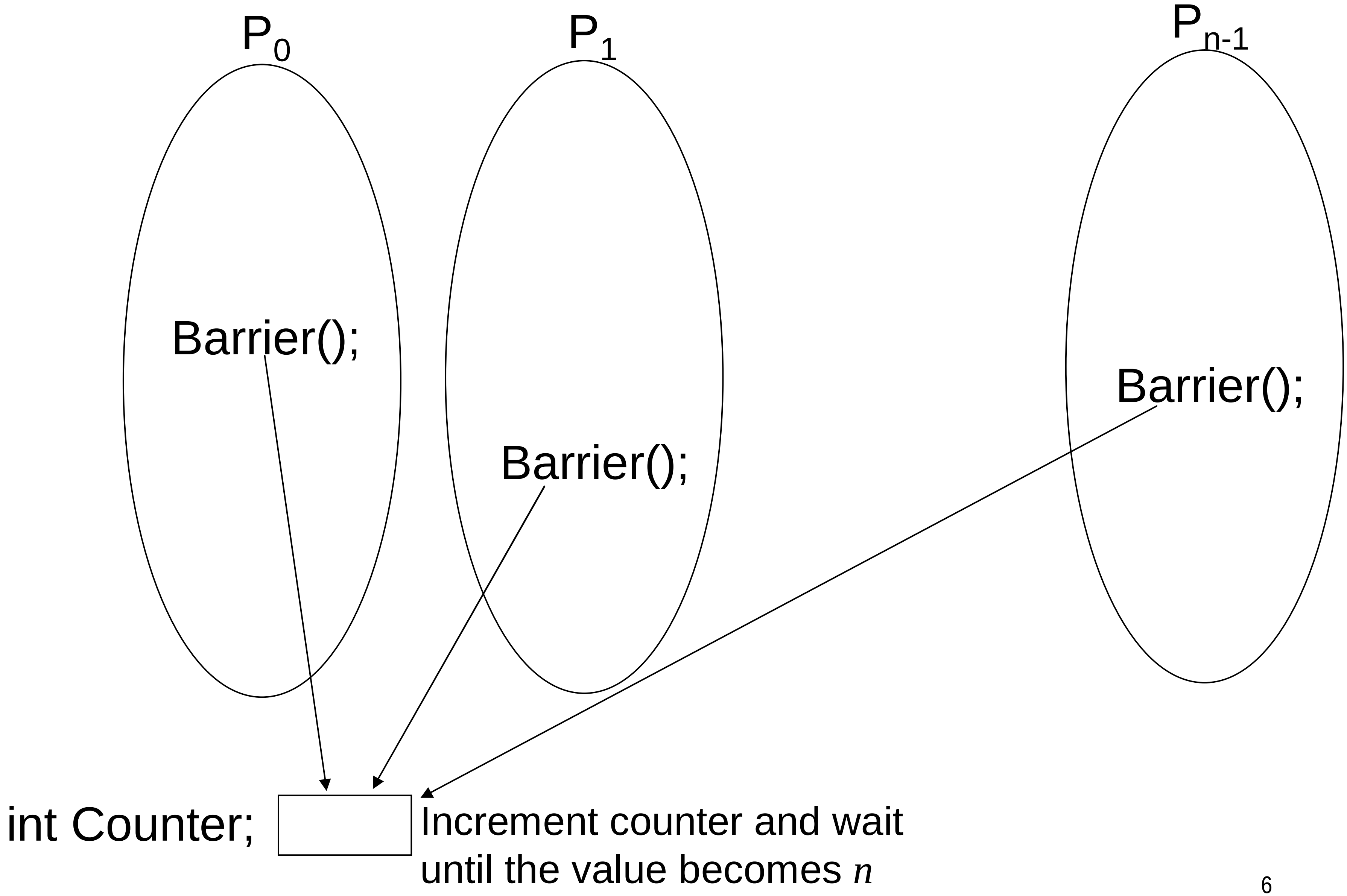
4. Mapping

   Mapping of processes to processors

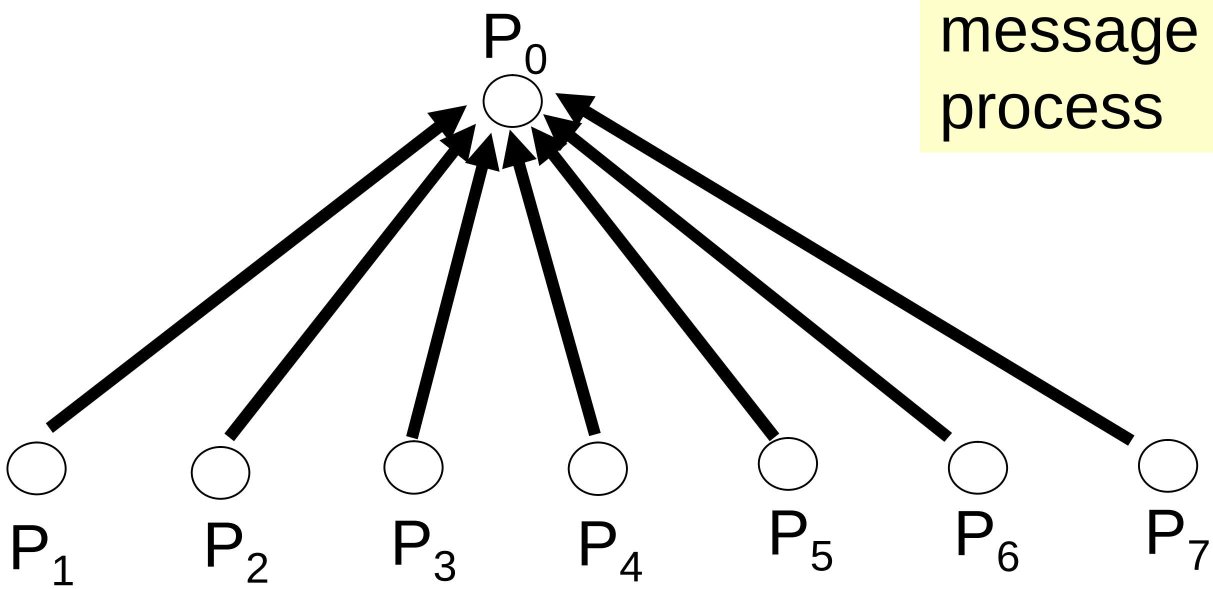
Example 1: Barrier Implementation

- What is a barrier?
  - A process synchronization primitive
  - If n cooperating processes all include a call to the barrier primitive …
  - Each entering process gets blocked on the barrier call until all the n processes have reached the barrier call
  - Thus, the n processes are synchronized on departure from the barrier call

Example 1: Barrier Implementation



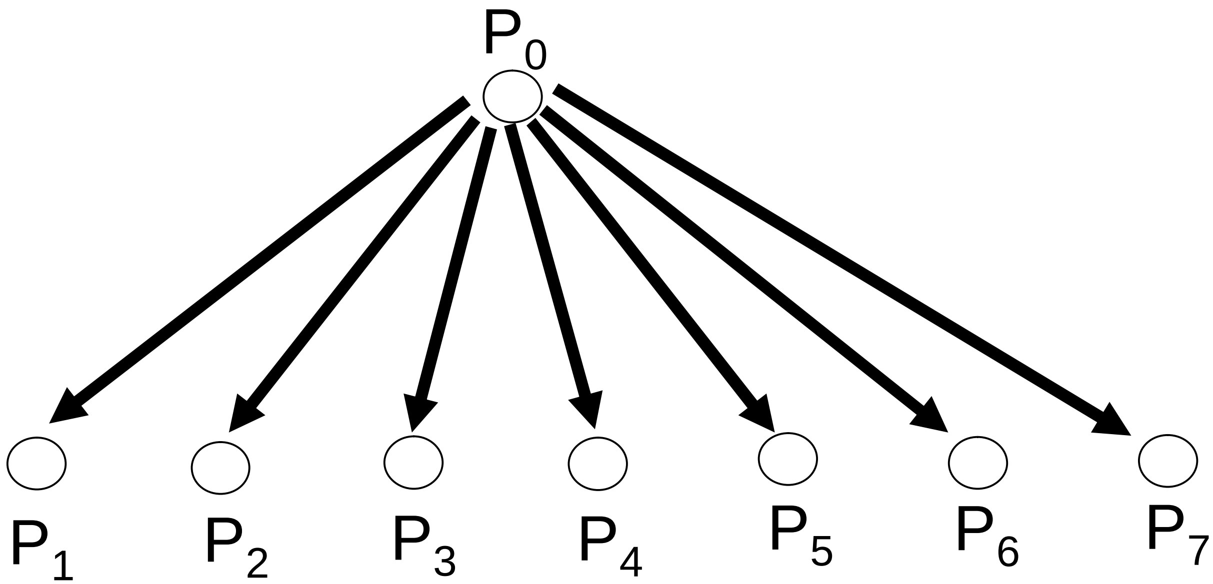
Linear Barrier Pseudocode



When a process reaches the barrier call, it sends a message to the master process

Linear Barrier Pseudocode

When the master process has received n messages, it sends a message to each of the participating processes to go ahead



Linear Barrier Pseudocode

Master:

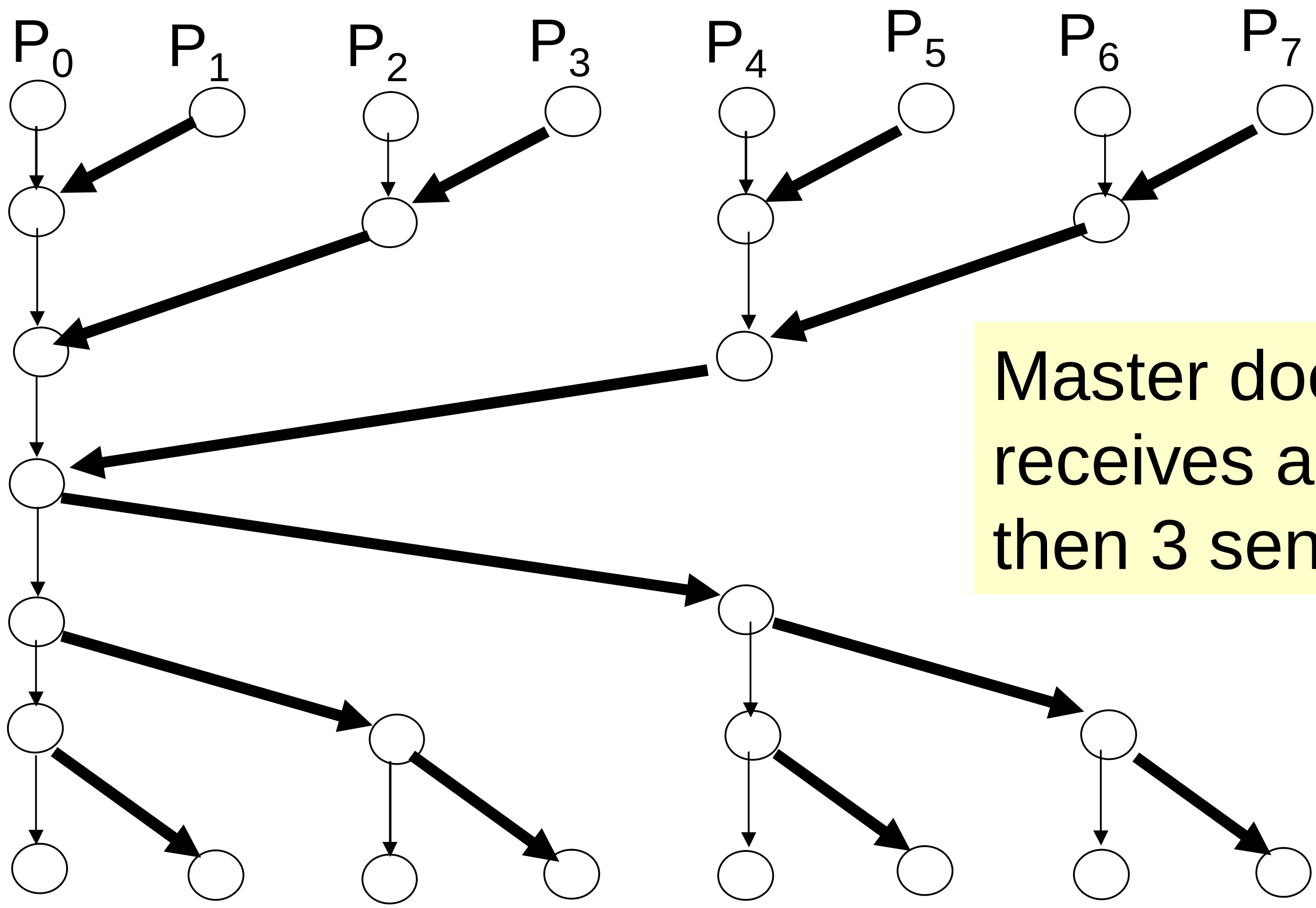
```
for (i = 0; i < n; i++)    /receive messages from slaves/
    receive (P_any );
for (i = 0; i < n; i++)    /release slaves/
    send (P_i );
```

Slaves:

```
send (P_master );
receive (P_master)
```

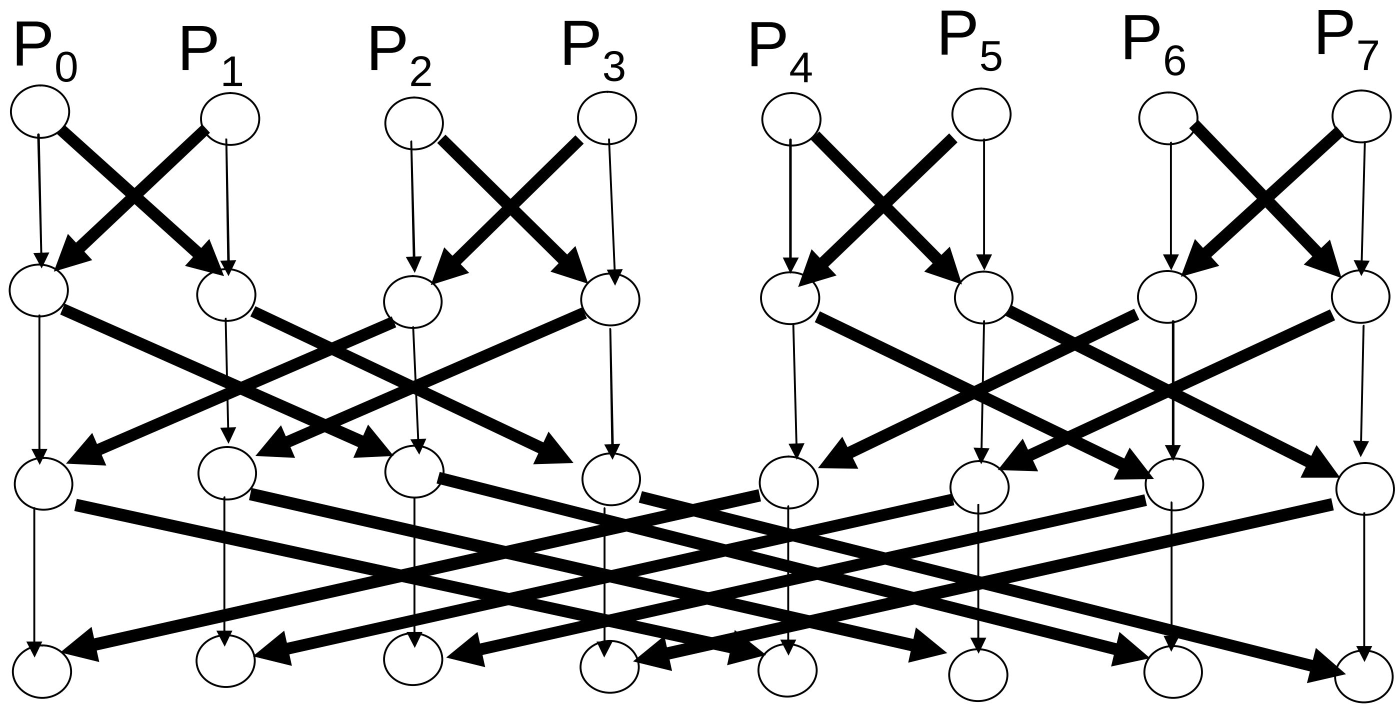
Master does n receives and then n sends

Alternatively …Tree Barrier



Master does 3 receives and then 3 sends

Alternatively …Butterfly Barrier



Stage 1: P0-P1, P2-P3, P4-P5, P6-P7

Stage 2: P0-P2, P1-P3, P4-P6, P5-P7

Stage 3: P0-P4, P1-P5, P2-P6, P3-P7

Each process does 3 send-receives

Example 2

Given a 2-d array of float values, repeatedly average each elements with its immediate neighbours until the difference between two iterations is less than some tolerance value

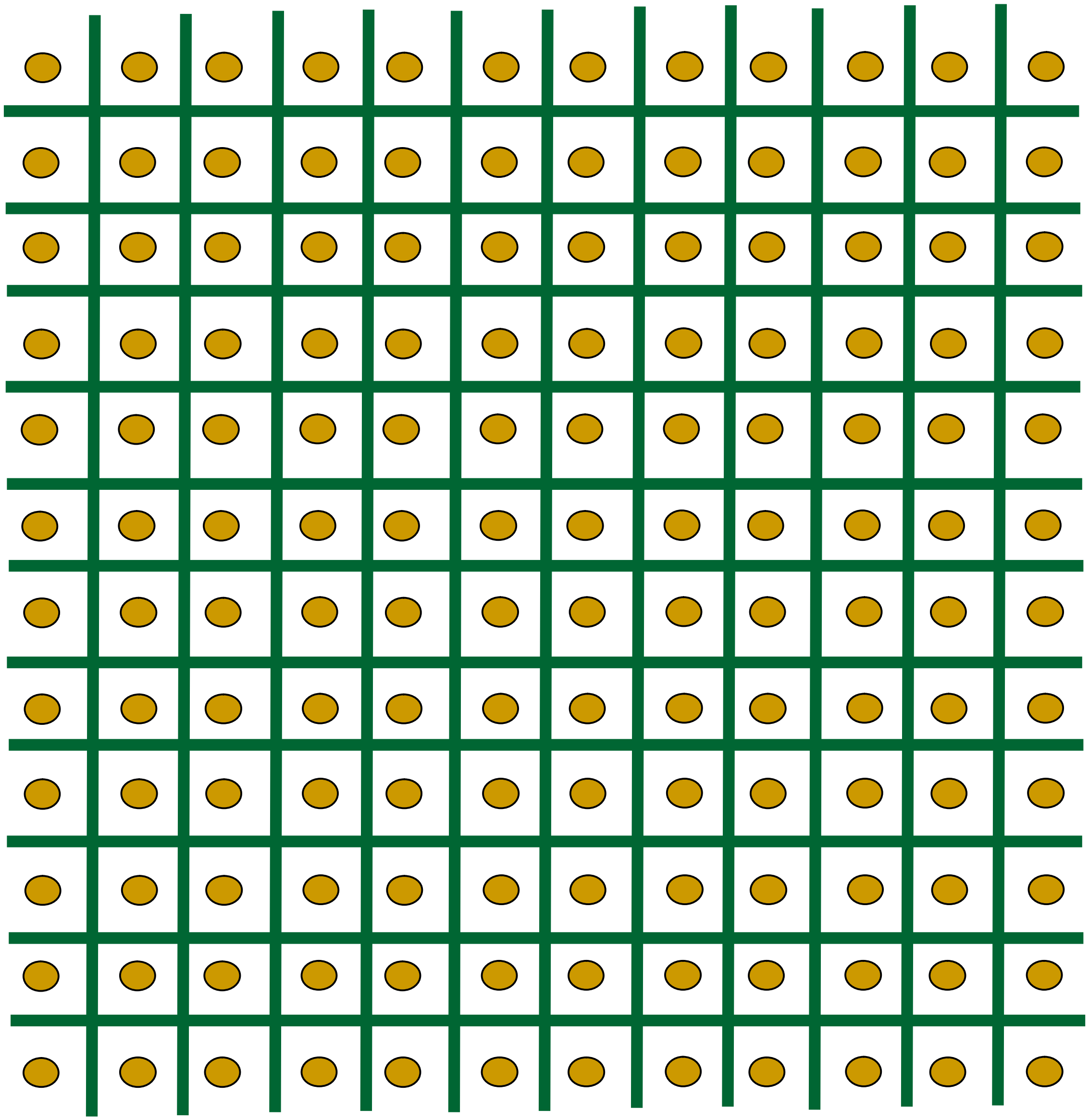
```
diff = 0.0
for (i=0; i < n; i++)
   for (j=0; j < n, j++){
      temp = A[i] [j];
      A[i][j] = average (neighbours);
      diff += abs (A[i][j] – temp);
   }
if (diff < tolerance) done;
```

	A[i-1][j]	
A[i][j-1]	A[i][j]	A[i][j+1]
	A[i+1][j]	

Some Decomposition Options

1. A parallel task for each element update

Option 1



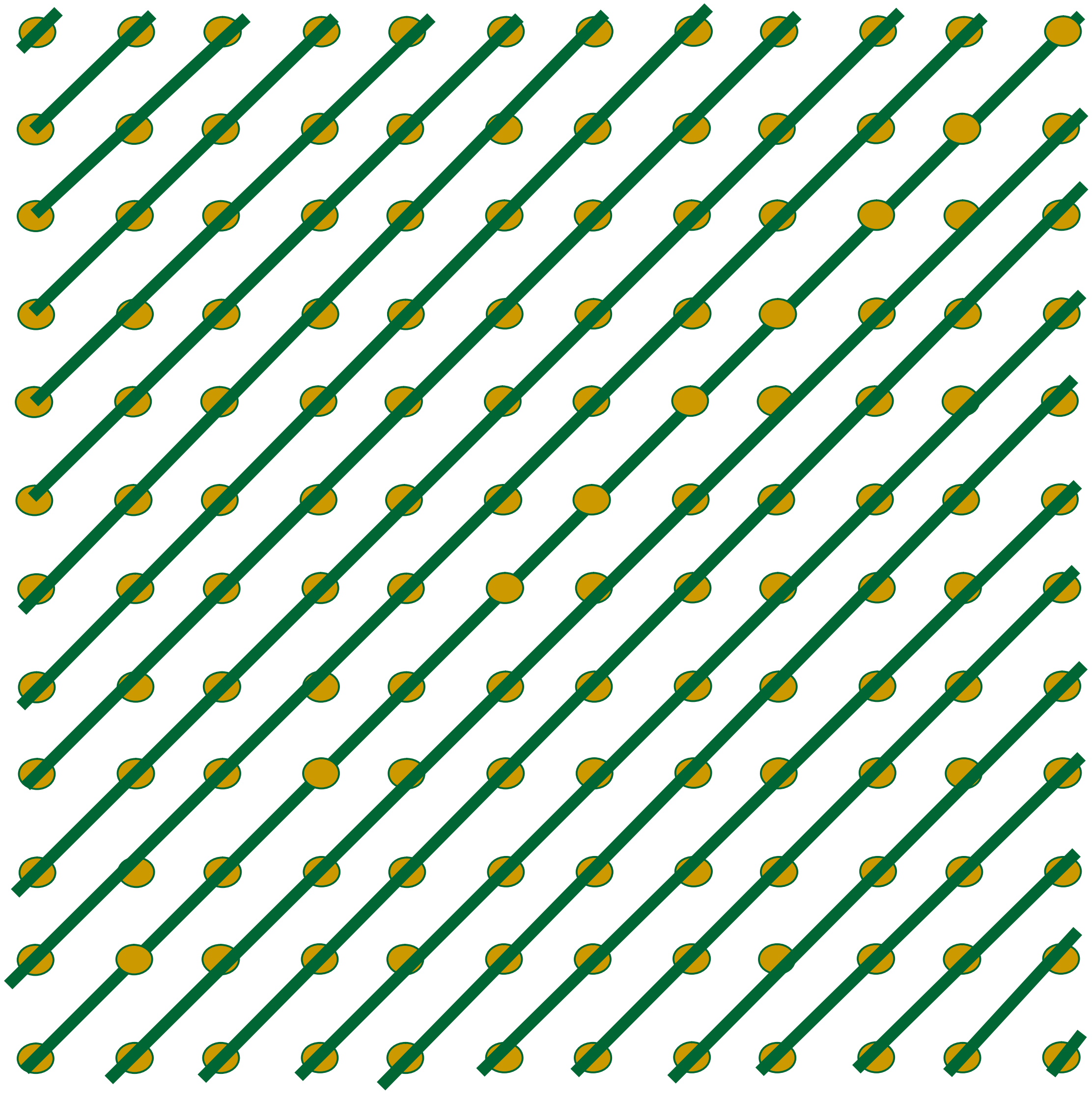
Some Decomposition Options.

1. A parallel task for each element update
   - Maximum parallelism: n^2
   - Synchronization required: wait for left & top values
   - High synchronization cost

Some Decomposition Options..

1. A parallel task for each element update
   - Maximum parallelism: n^2
   - Synchronization required: wait for left & top values
   - High synchronization cost
2. A parallel task for each anti-diagonal

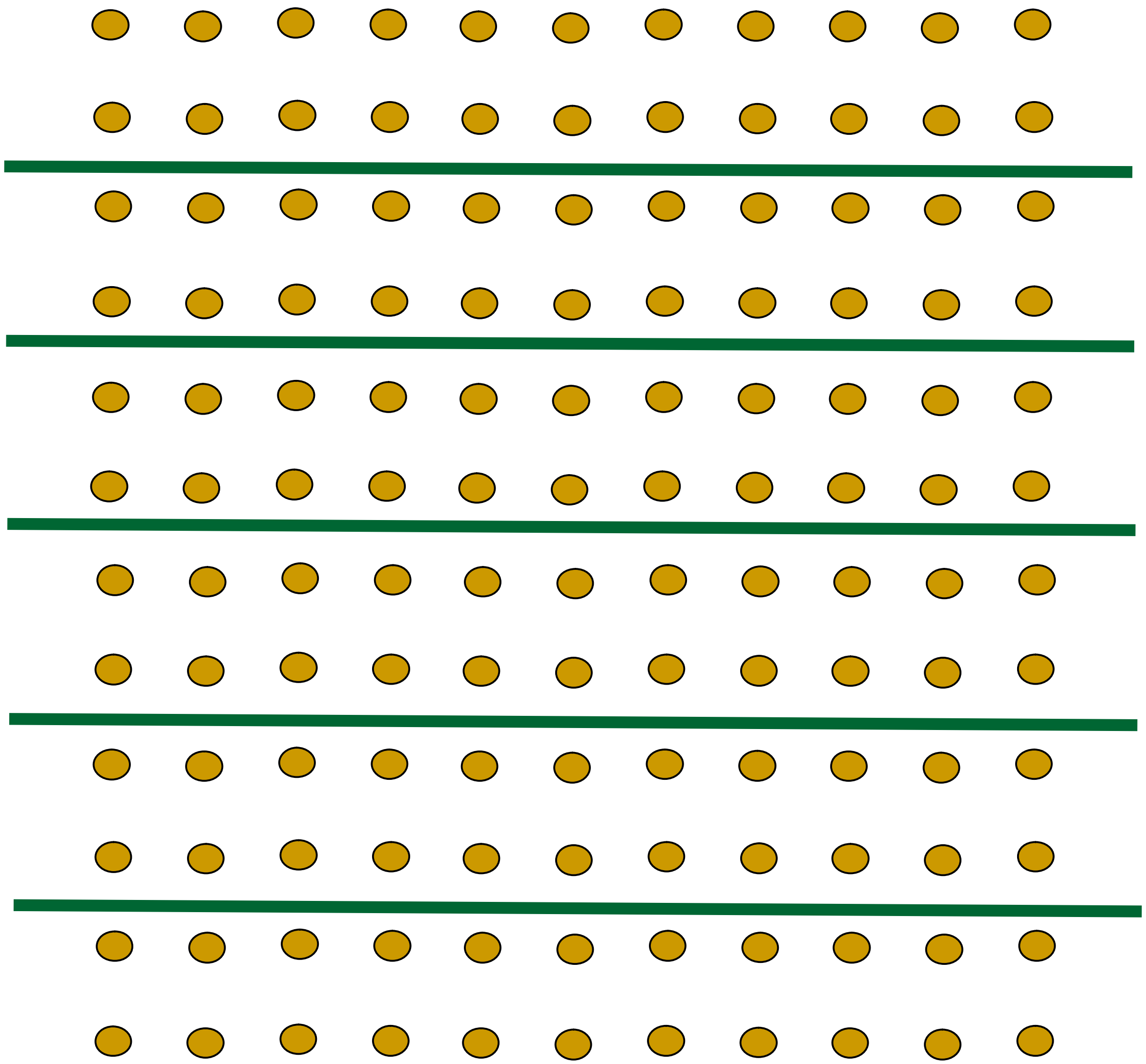
Option 2 Anti-diagonals



Some Decomposition Options…

1. A parallel task for each element update
   - Maximum parallelism: n^2
   - Synchronization required: wait for left & top values
   - High synchronization cost
2. A parallel task for each anti-diagonal
   - No dependence among elements in task
   - Maximum parallelism: 2n-1
   - Synchronization: must wait for previous anti-diagonal values; less cost than for the previous scheme
3. A parallel task for each block of rows

Option 3   Blocks of rows



High Performance Computing

1. Program execution: Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. Computer organization: Memory, Registers, Instruction set architecture, Instruction processing (6)
3. Virtual memory: Address translation, Paging (4)
4. Operating system: Processes, System calls, Process management (6)
5. Pipelined processors: Structural, data and control hazards, impact on programming (4)
6. Cache memory: Organization, impact on programming (5)
7. Program profiling (2)
8. File systems: Disk management, Name management, Protection (4)
9. Parallel programming: Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI (5)