Solving Linear Equations Iteratively & In Parallel

Our Matrix-Vector Multiply can be used to solve linear equations

$$
\begin{aligned}
a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \ldots + a_{0,n-1}x_{n-1} &= b_0 \\
a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \ldots + a_{1,n-1}x_{n-1} &= b_1 \\
a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + \ldots + a_{2,n-1}x_{n-1} &= b_2 \\
&\vdots \\
a_{m-1,0}x_0 + a_{m-1,1}x_1 + a_{m-1,2}x_2 + \ldots + a_{m-1,n-1}x_{n-1} &= b_{m-1}
\end{aligned}
$$

can often be solved iteratively. A sufficient condition is **diagonal dominance**: The diagonal term is greater than the sum of the abs of the off-diagonal terms.

Our Matrix-Vector Multiply can be used to solve linear equations

First symbolically solve the i^{th} equation for x_i:

$$
\begin{aligned}
x_0 &= (b_0 - (a_{01}x_1 + a_{02}x_2 + \ldots + a_{0,n-1}x_{n-1}))/a_{00} \\
x_1 &= (b_1 - (a_{10}x_0 + a_{12}x_2 + \ldots + a_{1,n-1}x_{n-1}))/a_{11} \\
x_2 &= (b_2 - (a_{20}x_0 + a_{21}x_2 + \ldots + a_{2,n-1}x_{n-1}))/a_{22} \\
&\ldots \\
x_{m-1} &= (b_{m-1} - (a_{m-1,0}x_0 + a_{m-1,1}x_2 + \ldots + a_{m-1,n-2}x_{n-2}))/a_{m-1,m-1}
\end{aligned}
$$

- Each x on the LHS depends on all other x's except itself.
- The RHS entails a matrix-vector multiplication CX.

Matrix Equation

- The previous equation in matrix form is:

  $$X = B - CX$$

- Starting with an initial approximation to X (say all 0's), we repeatedly iterate:

  $$X := B - CX;$$

  until the change in X (say, Euclidean distance), is within a desired tolerance.

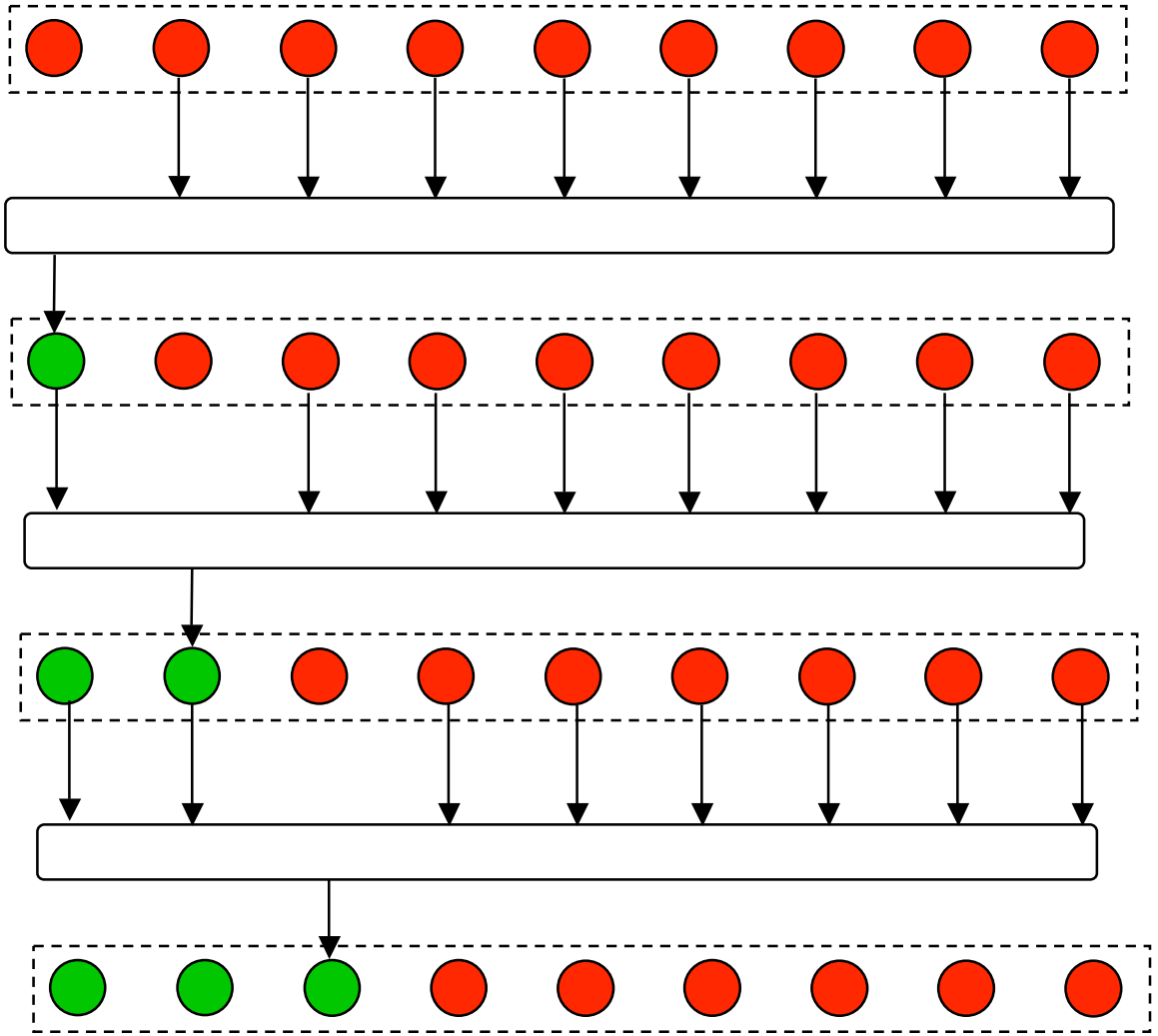
Jacobi Method

- What we have just described is the Jacobi method for solving simultaneous equations.
- It is one of several forms of “relaxation”.
- The distinguishing feature of Jacobi is that all the new LHS’s are computed **before** installing them in place of the Xs.

Gauss-Seidel Method

- In this variation, the new LHS's are installed as soon as they are computed, assuming that this is done sequentially.
- In its pure form, this method appears unattractive for parallel computing; it is too sequential.
- The following graph shows this.

Gauss-Seidel Method



= “old”

= “new”

Gauss-Seidel v. Jacobi

- On the other hand, Gauss-Seidel **converges** about twice as fast as Jacobi.

- Intuitively, this is because the effect of changes propagates faster.

Gauss-Seidel Method

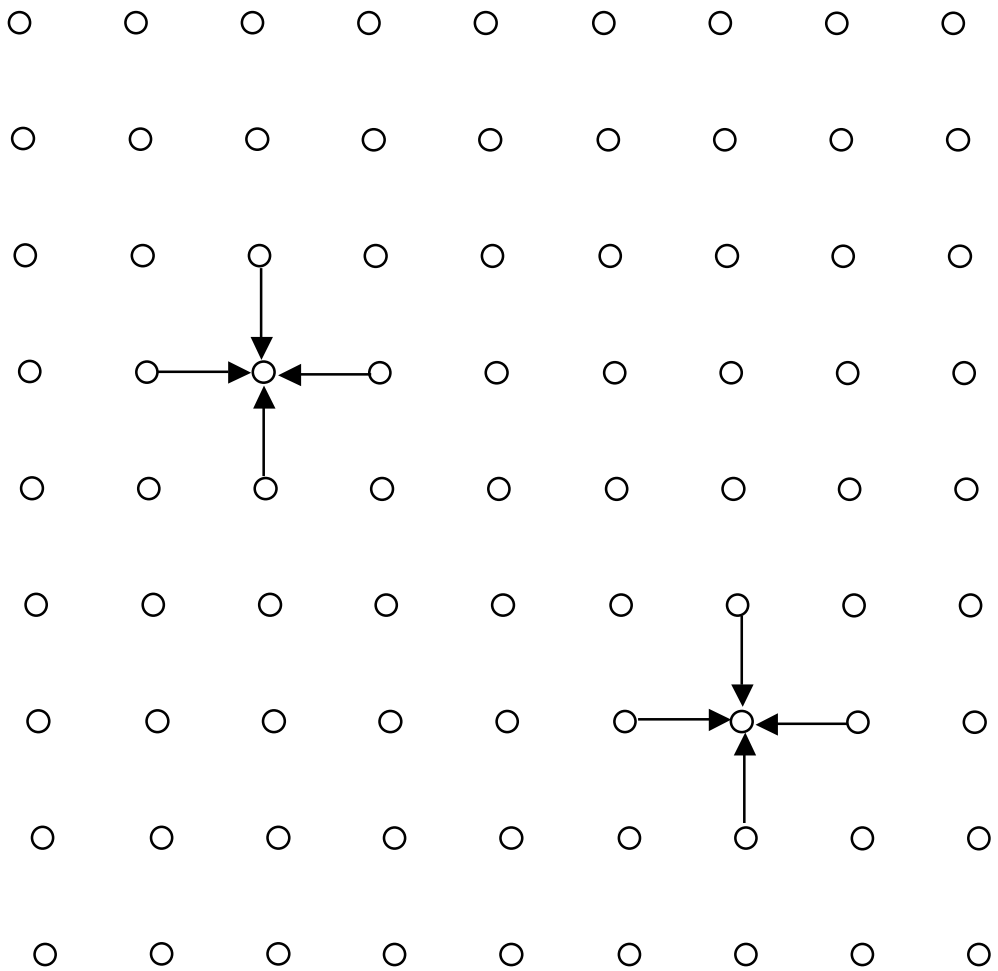
- If the matrix is sparse and structured in certain regular ways, there is a way to pipeline this computation so that it is not so bad once the pipe is full.

- One such case is the Laplace equation.

Laplace Equation

- Consider the case where the array of unknowns is itself two-dimensional.
- This occurs for example in Laplace's equation (aka Heat Equation).
- The update rule is that each point depends only on the previous value of its immediate neighbors.

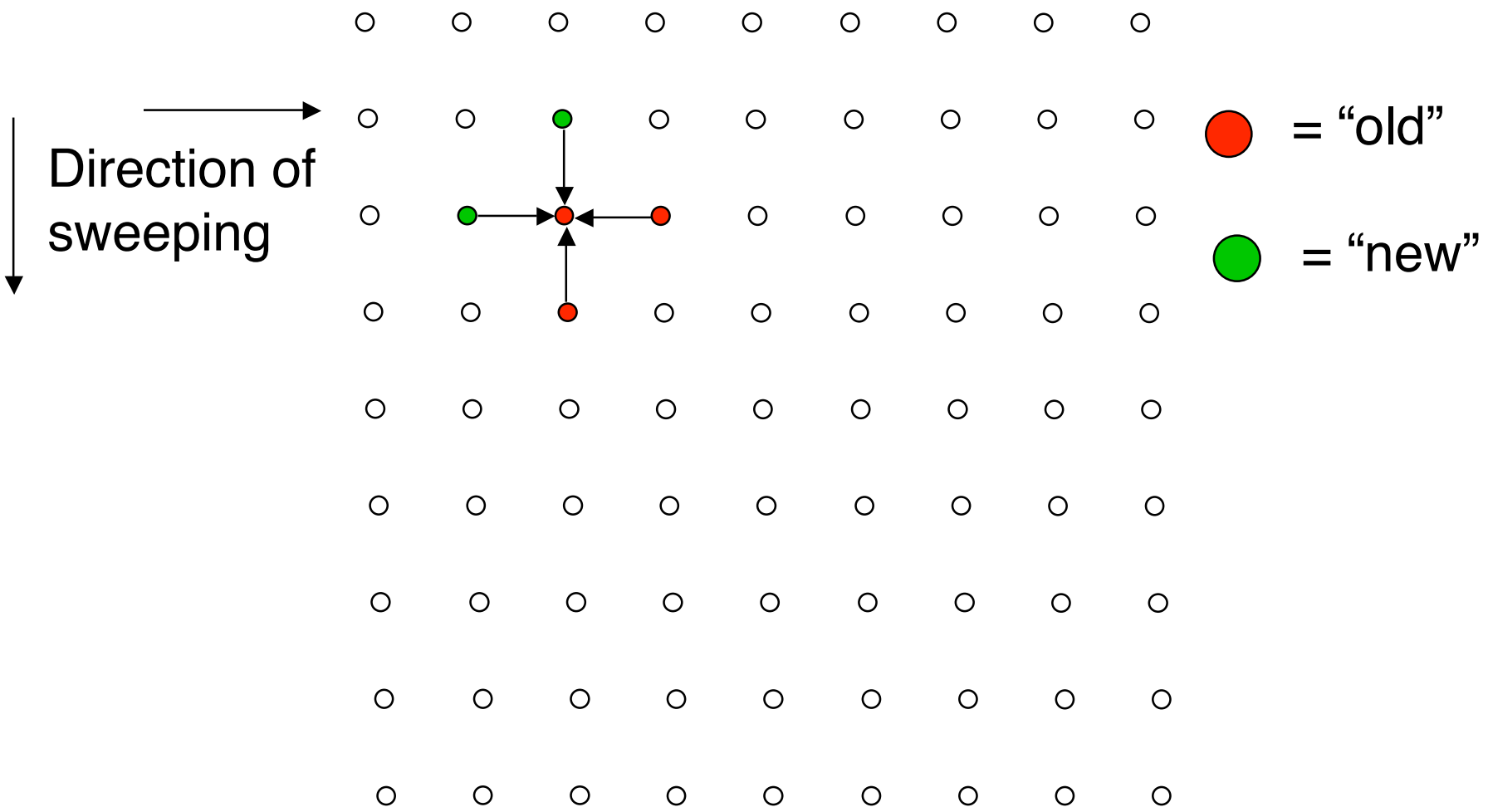
Laplace Equation



Laplace Equation

- Typically the boundary values are fixed.
- Only the interior points are updated.
- So this is a “boundary value” problem.
- The algebraic problem is derived by discretizing a partial differential equation (Laplace’s equation).
- Let’s look at the Gauss-Seidel update.

Gauss-Seidel Method



Gauss-Seidel Method

- As soon as a sufficiently-many elements of the current sweep are finished, the next sweep can be started in parallel!
- This is pipelining the matrix through a series of sweeps.
- Once the pipeline is “full”, all processors can be kept busy until the end.

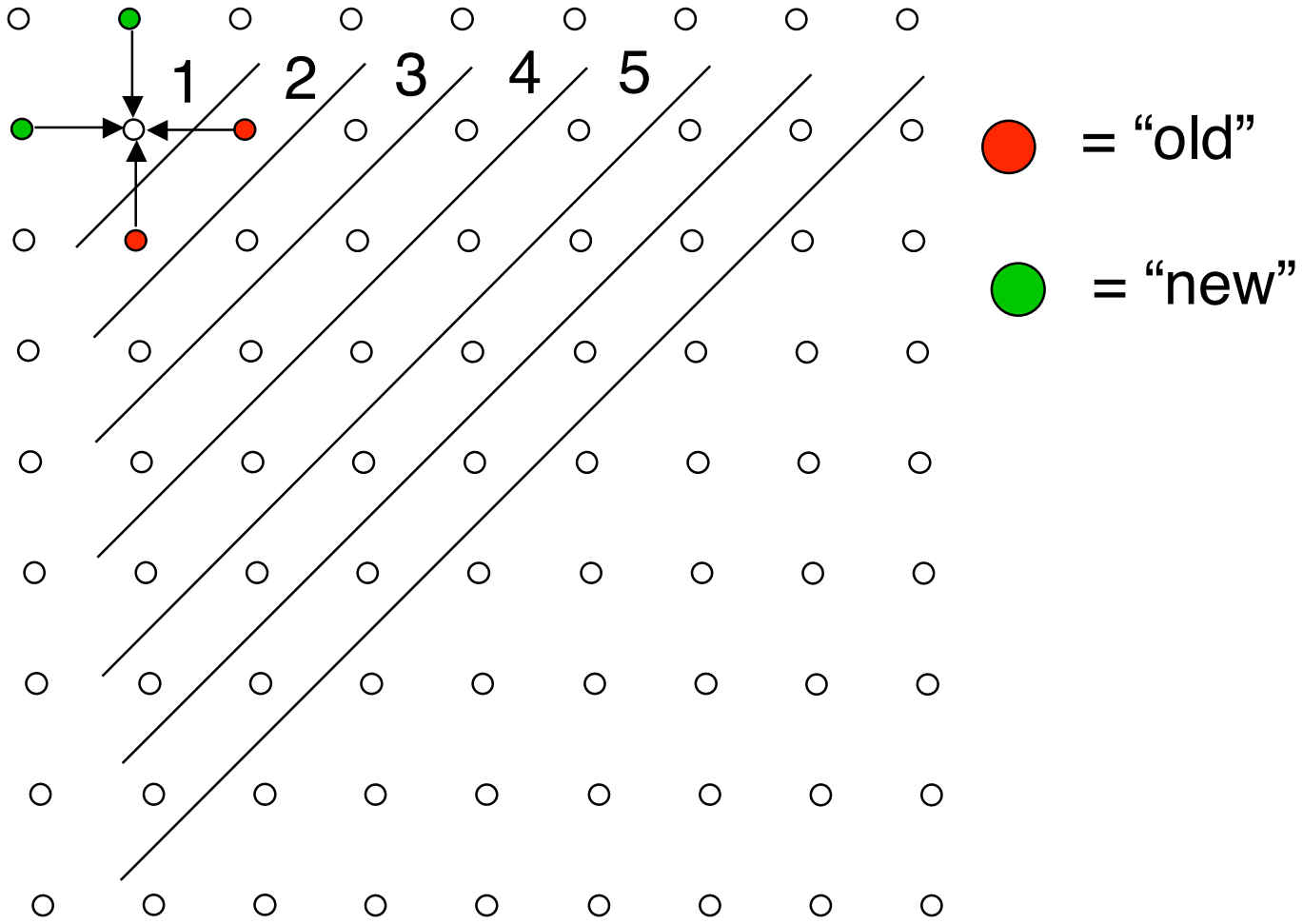
Gauss-Seidel Method

- The alternate way of thinking about this is as a “wave-front”.

- If we update a point in Gauss-Seidel as soon as it is “ready” (when its predecessors have been computed), we get the following phenomenon.

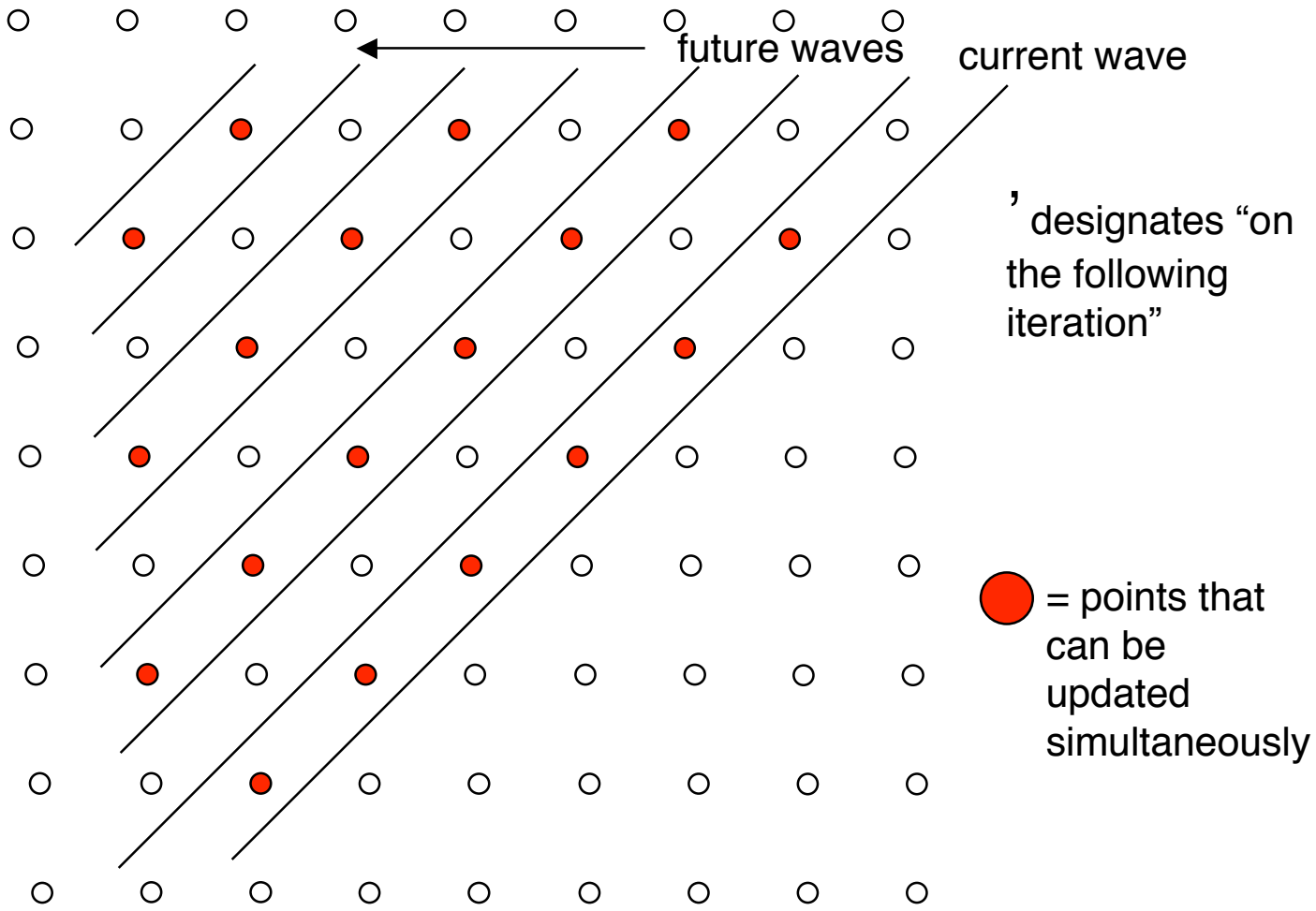
Gauss-Seidel Wavefront

Numbers show order of updating



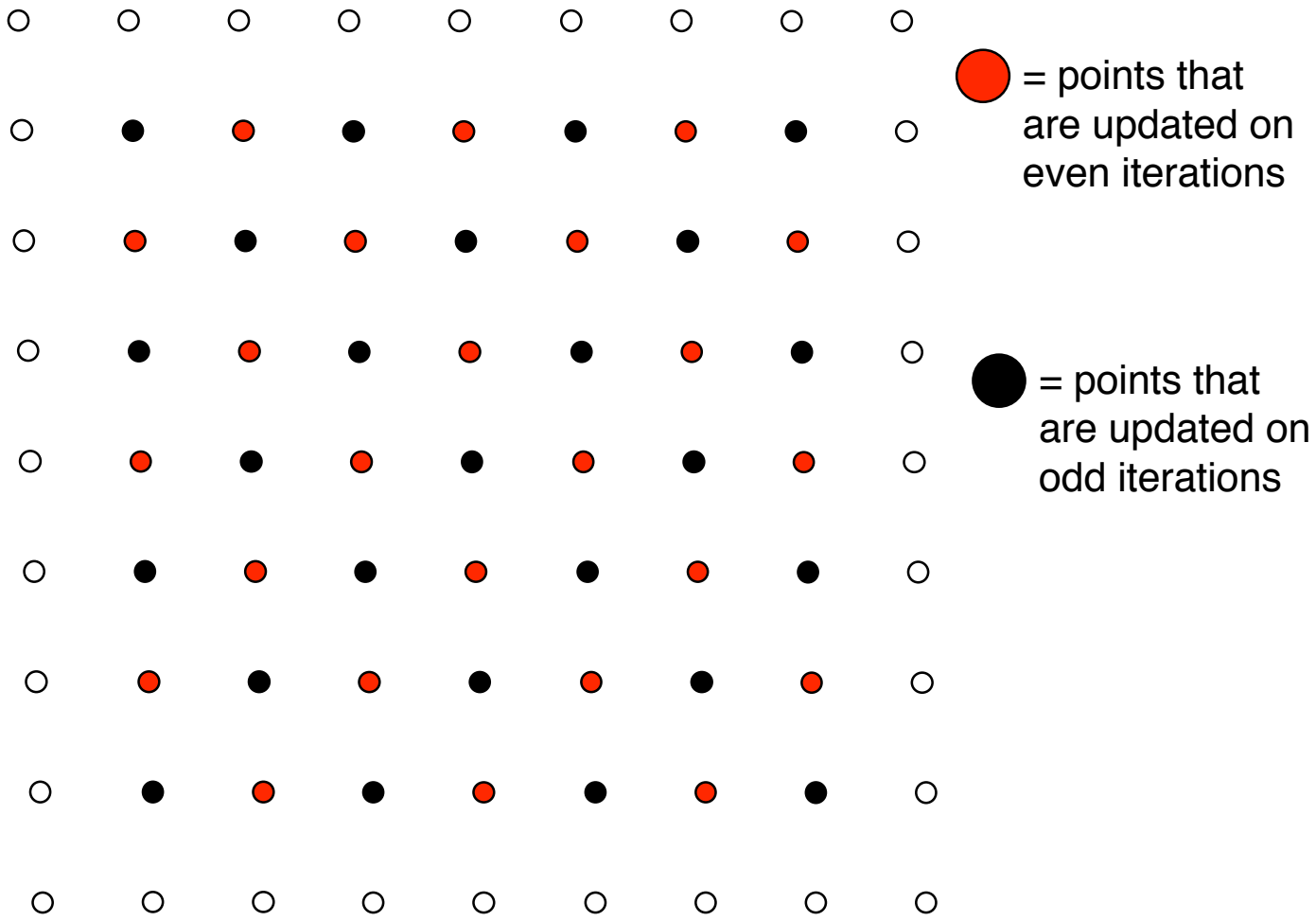
Gauss-Seidel Wavefront

As soon as the wave is one diagonal away, the next wave could be started.



Red-Black Method

Since half the points are being updated at any time, just do all half; don't wait for a wave to arrive.



Red-Black Method

- Degree of parallelism = 1/2 degree for Jacobi (insignificant drawback if large number of points)
- No extra storage required as with Jacobi
- Convergence = Gauss Seidel
- Program similar to iterated matrix-vector, except that red and black iterations alternate

Partitioning

- To partition the previous methods on a distributed-memory system, the number of points should be large compared to the number of processors.
- The grid is divided into per-processor regions.
- Only the **borders** of these regions need to be communicated to processors of neighboring regions.

Communication across the Border
("ghost points")

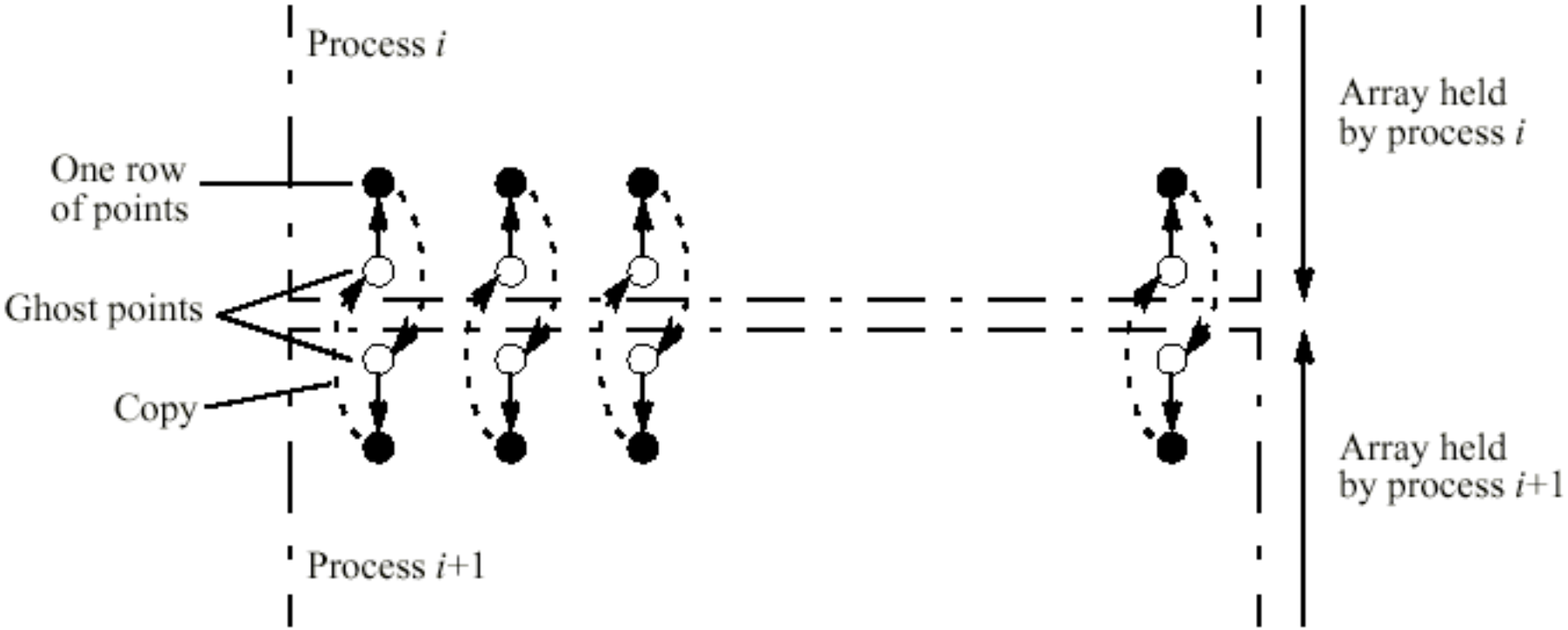


Figure 6.18 Configurating array into contiguous rows for each process, with ghost points.

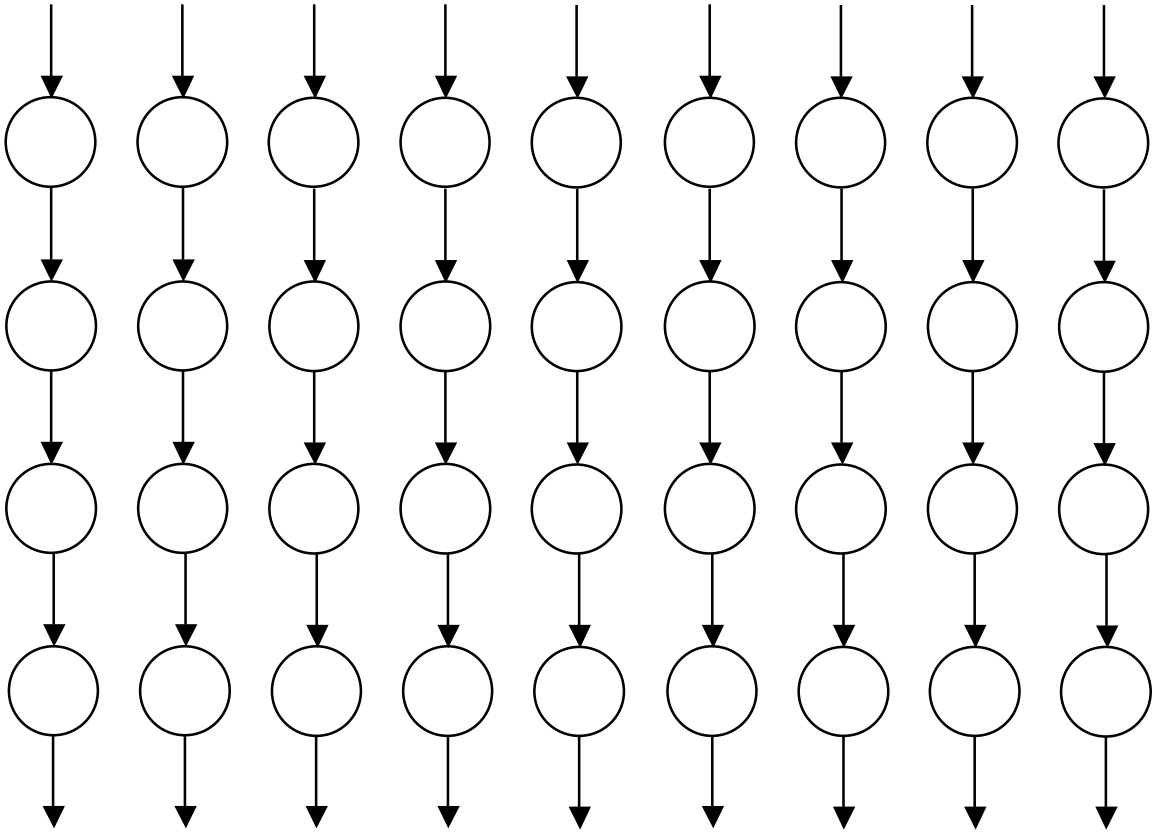
Embarassingly-Parallel Problems

Pleasantly-Parallel Problems

Pleasantly-Parallel Problems

- The degree of parallelism is very high (grows linearly or better with problem size).
- There is little or no communication between operations or threads.

Graph of a Pleasantly-Parallel Problem



Deja Vu

- The *initial segment* of cost-optimal versions of problems such as:
  - vector inner-product
  - prefix sum
  - polynomial evaluation
- Matrix-Vector multiplication, if distribution cost ignored

Examples

- Image Transformation
- IFS (Iterated Function Systems), e.g. “Mandelbrot Set” and other Fractals
- Monte Carlo methods

Image Transformation

- 2-D array (say 512 x 512) of pixels (“pixmap”)
- Pixels are grayscale or RGB
- Typical function produces a new pixmap from an old one, by:
  - Coordinate translation, scaling, rotation, clipping, etc. (move pixels to new locations)
  - Masking, thinning (change pixels in place)

Image Transformation

- Since the typical operations tend to be fine grain, it is important to partition the pixmaps into large chunks to reduce communication overhead.

Translation

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

Scaling

$$x' = xS_x$$

$$y' = yS_y$$

Rotation

$$x' = x\cos\theta + y\sin\theta$$

$$y' = -x\sin\theta + y\cos\theta$$

IFS (Iterated Function Systems)

- e.g. “Mandelbrot Set”
- The *original* naïve parallelism problem
- 2-D array of complex values
- Points are **updated in place** for some large number of iterations, to determine whether each point is “in” the set or not.
- **No communication** with other points

Mandelbrot Set & Iteration

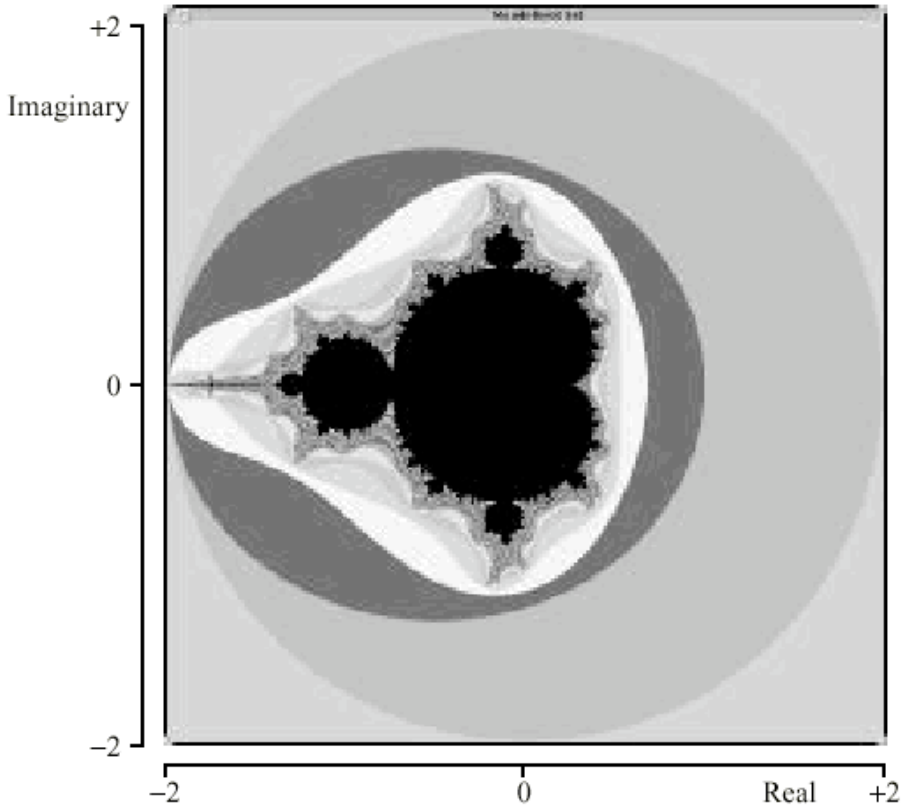


Figure 3.4 Mandelbrot set.

$$z_{real} = z_{real}^2 - z_{imag}^2 + c_{real}$$

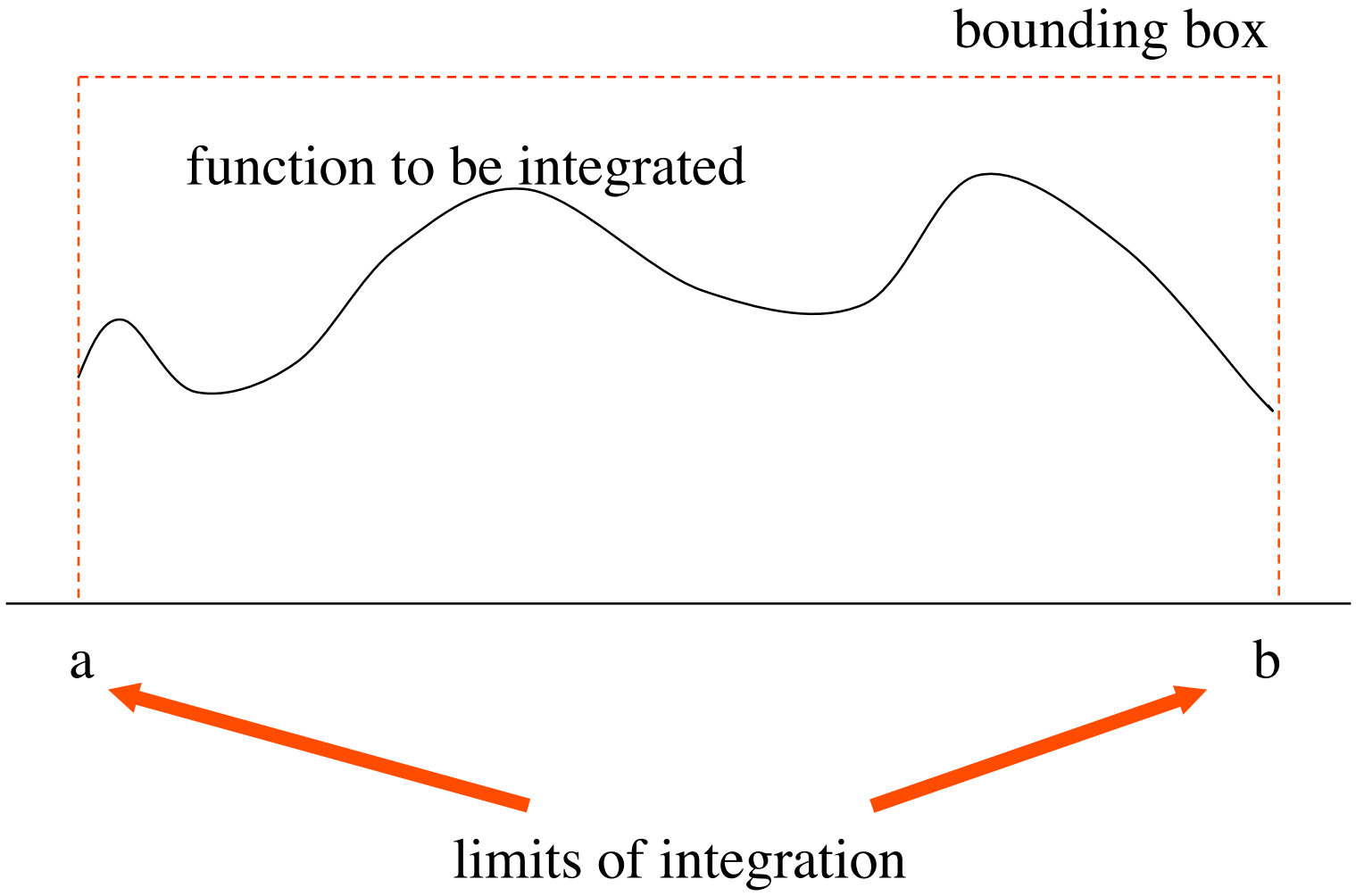
$$z_{imag} = 2 z_{real} z_{imag} + c_{imag}$$

If |z| < 2 after certain large number of iterations, point is declared to be in the set.

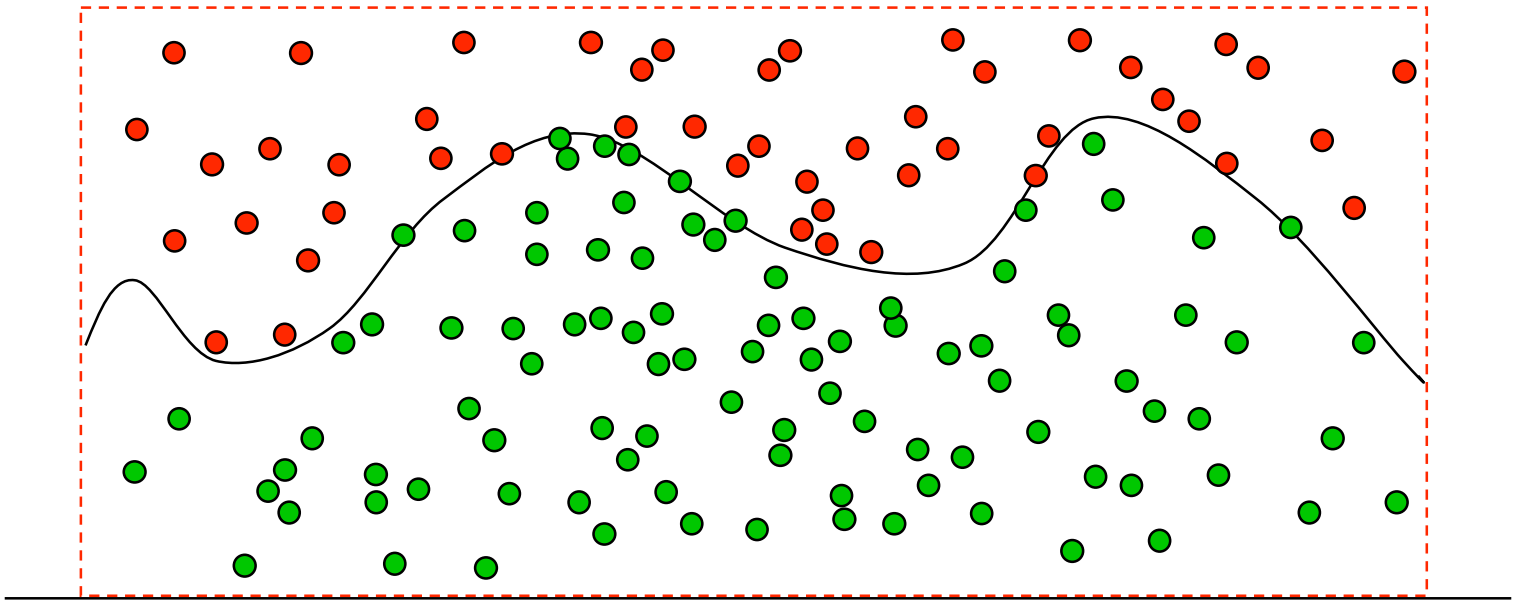
Monte Carlo Methods

- Random numbers for fun and profit
- Embarassingly parallel
- Embarassingly slow
- Sometimes the only recourse for complex problems

To Integrate



To Integrate



Given random (x, y) it is trivial to determine whether f(x) $\leq$ y.

Simply **counting** the number of such points and dividing by the total number of points approximates the integral.

Random Number Considerations

- Ideally (from mathematical point of view) one central RNG (random-number generator)
- Central RNG would be bottleneck
- For parallelism, use separate independent RNGs.
- Problem is to keep these from getting synchronized, producing non-random results overall.

Typical RNG Iteration

$$x_{i+1} = (ax_i + c) \bmod m$$

Parallel version, k processors, $A = a^k$ and $C = c(a^{k-1} + \ldots + a^0)$:

$$x_{i+1} = (ax_i + c) \bmod m$$

$$x_{i+k} = (Ax_i + C) \bmod m$$

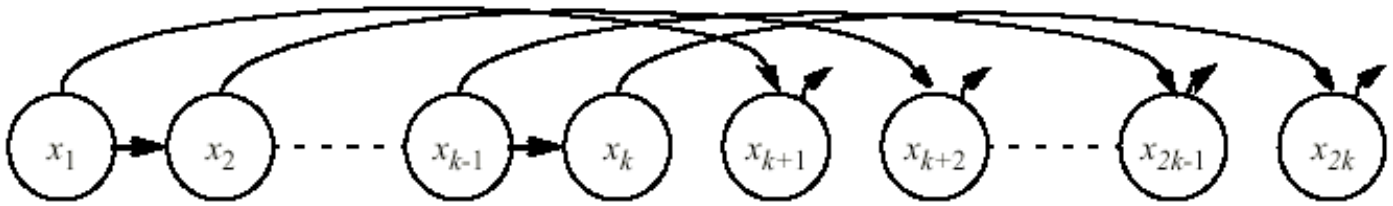


Figure 3.10 Parallel computation of a sequence.

initialized sequentially computed from predecessors

Processor/Work Pool Pattern

- aka Worker Model, or Processor Farm
- Can convert many relatively fine-grain problems to be somewhat pleasingly parallel
- Identify units of work, of which there should be a large number
- A unit of work is represented by an Object, such as one containing the parameters to a procedure

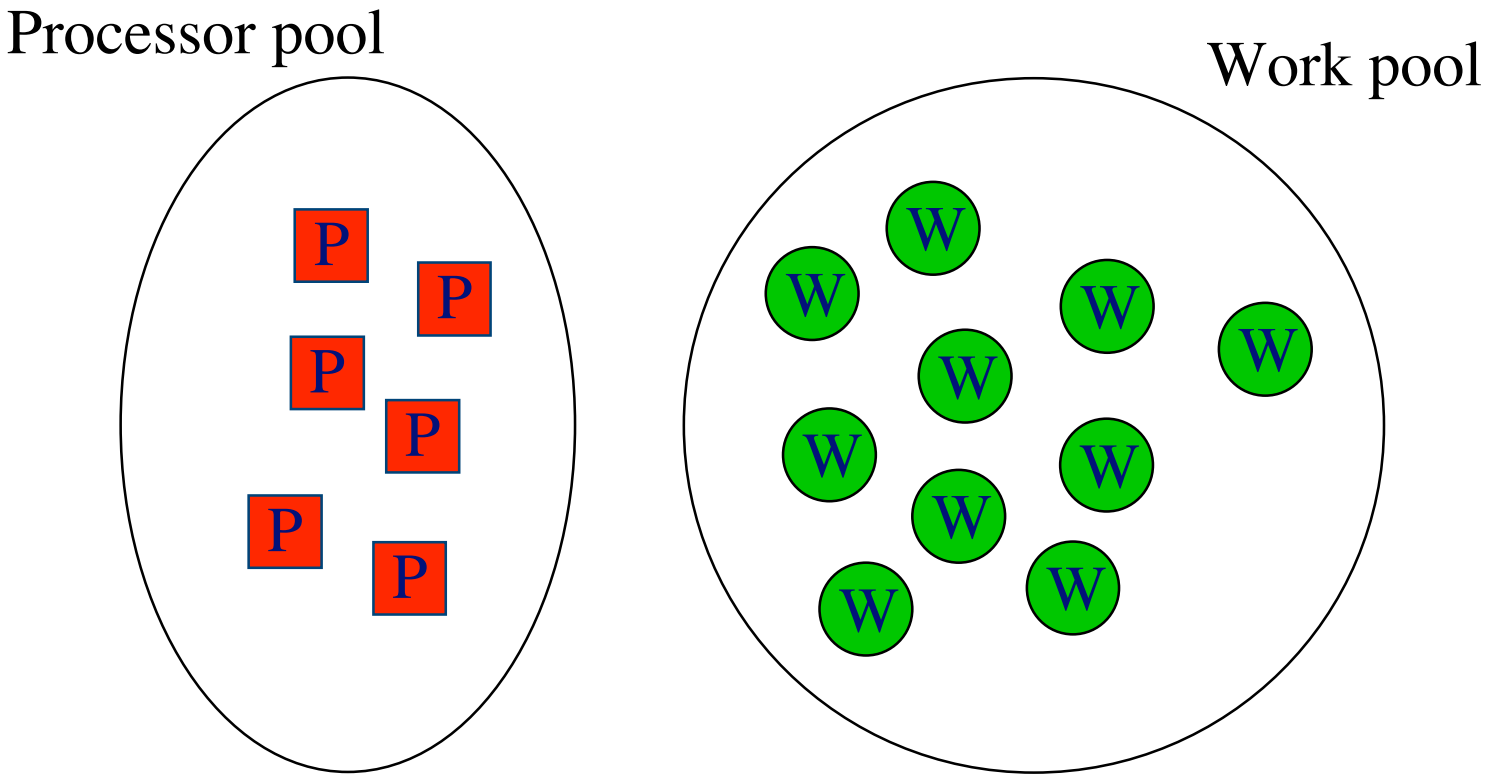
Processor/Work Pool Model

- Typically one unit of work will spawn others.
- If a processor is available in the pool, the new work will be taken up by it.
- If no processor is available, the new work is put in the work pool until a processor completes some work.

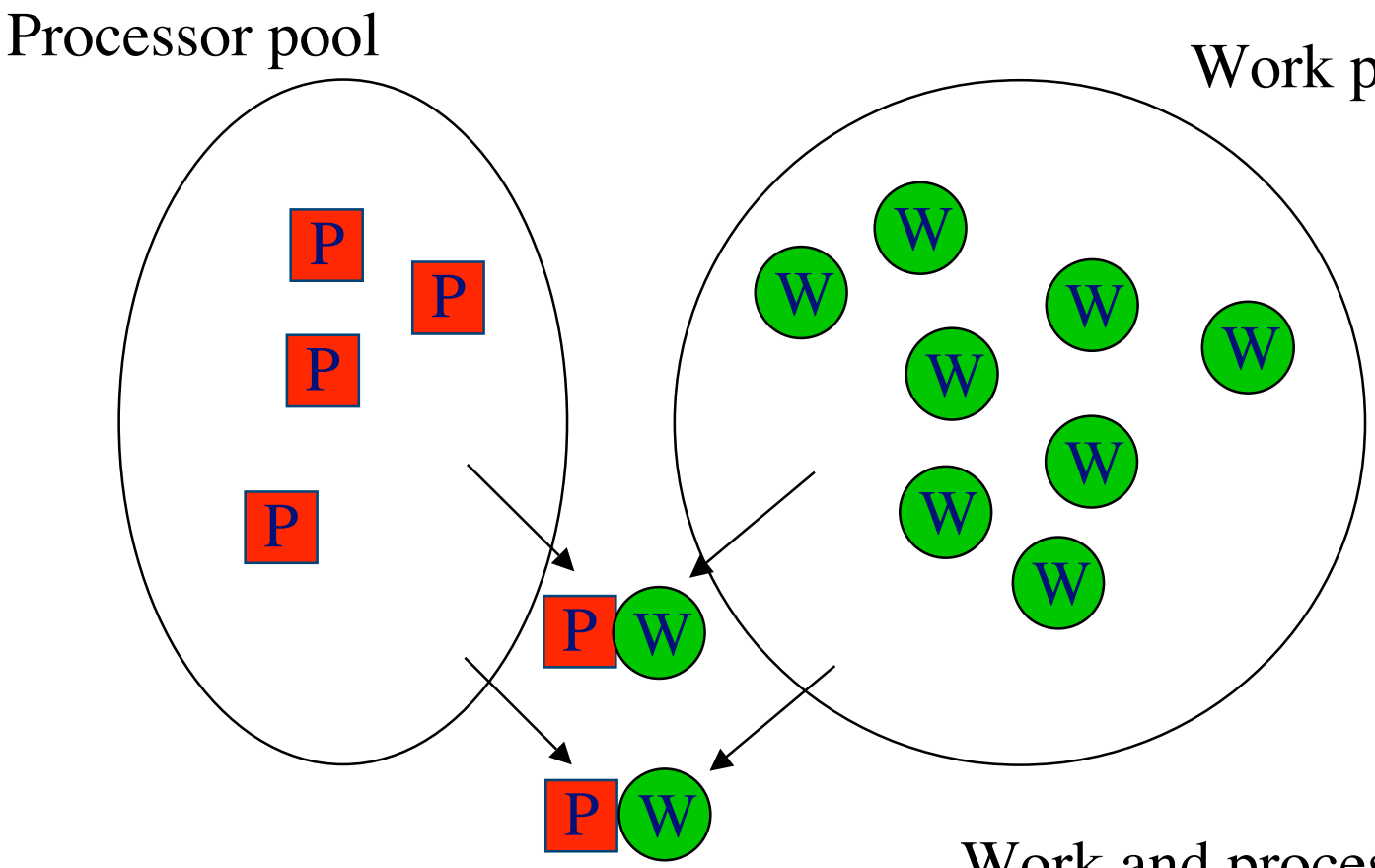
Processor/Work Pool Model

- At steady state, we have that at all times either:
  - The work pool is empty, or
  - The processor pool is empty
- Having both work and processors is an unstable condition: the processor should take on the work.
- This condition should only exist at start-up or transients.

Processor/Work Pool Model (startup configuration)



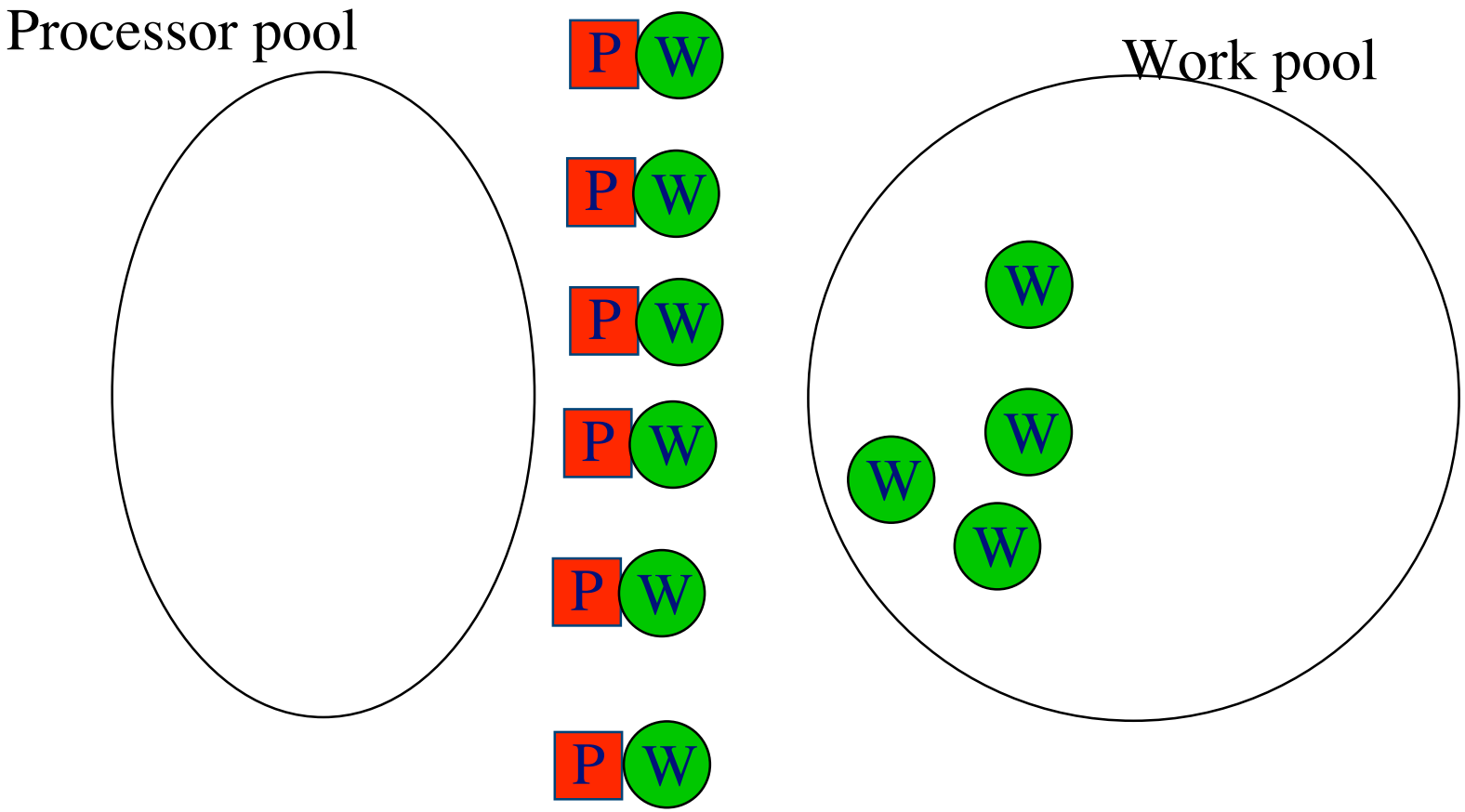
Processor/Work Pool Model
(startup configuration)



Work and processors are like matter and anti-matter; they want to “annihilate” each other.

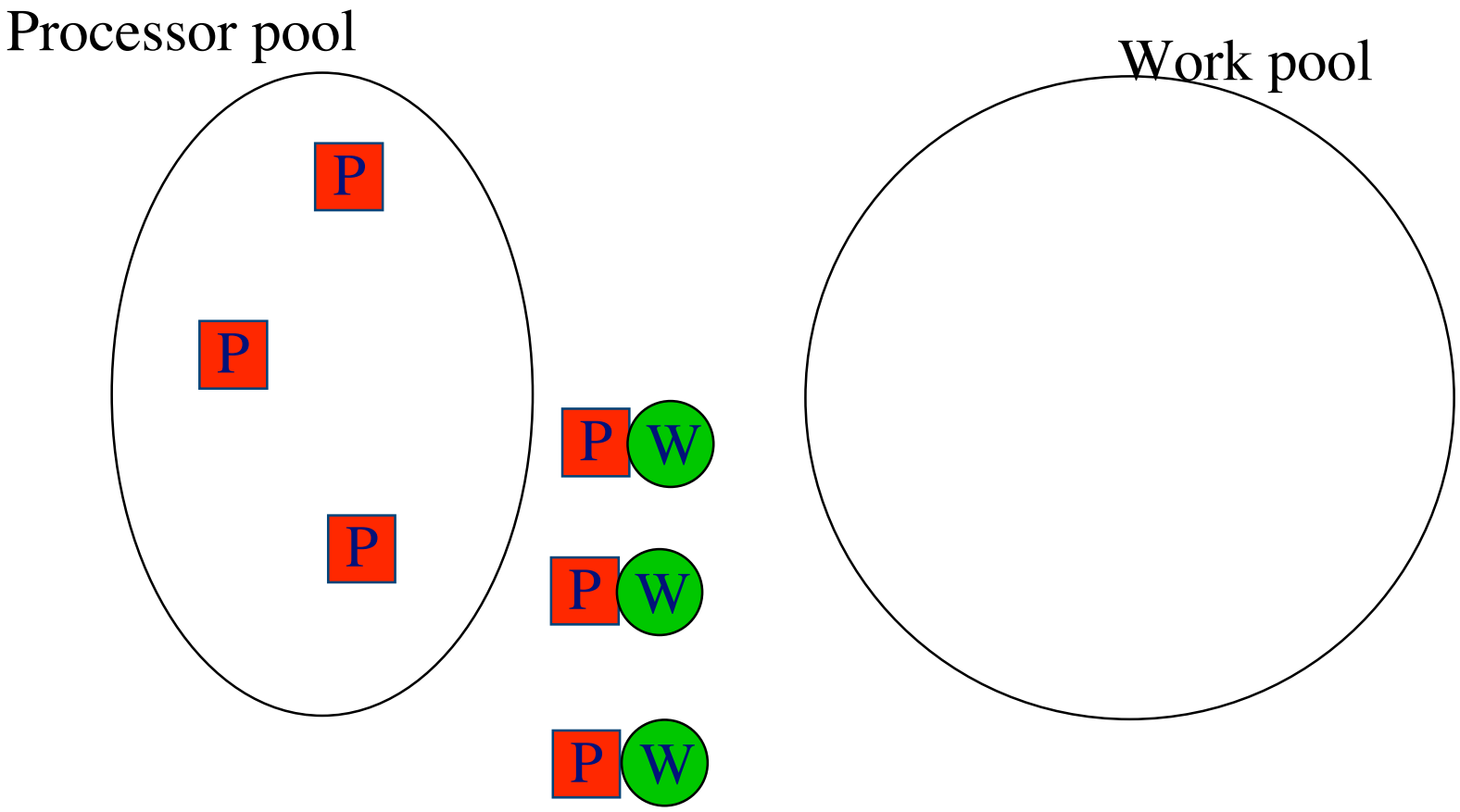
Processor/Work Pool Model

(surplus-work configuration)



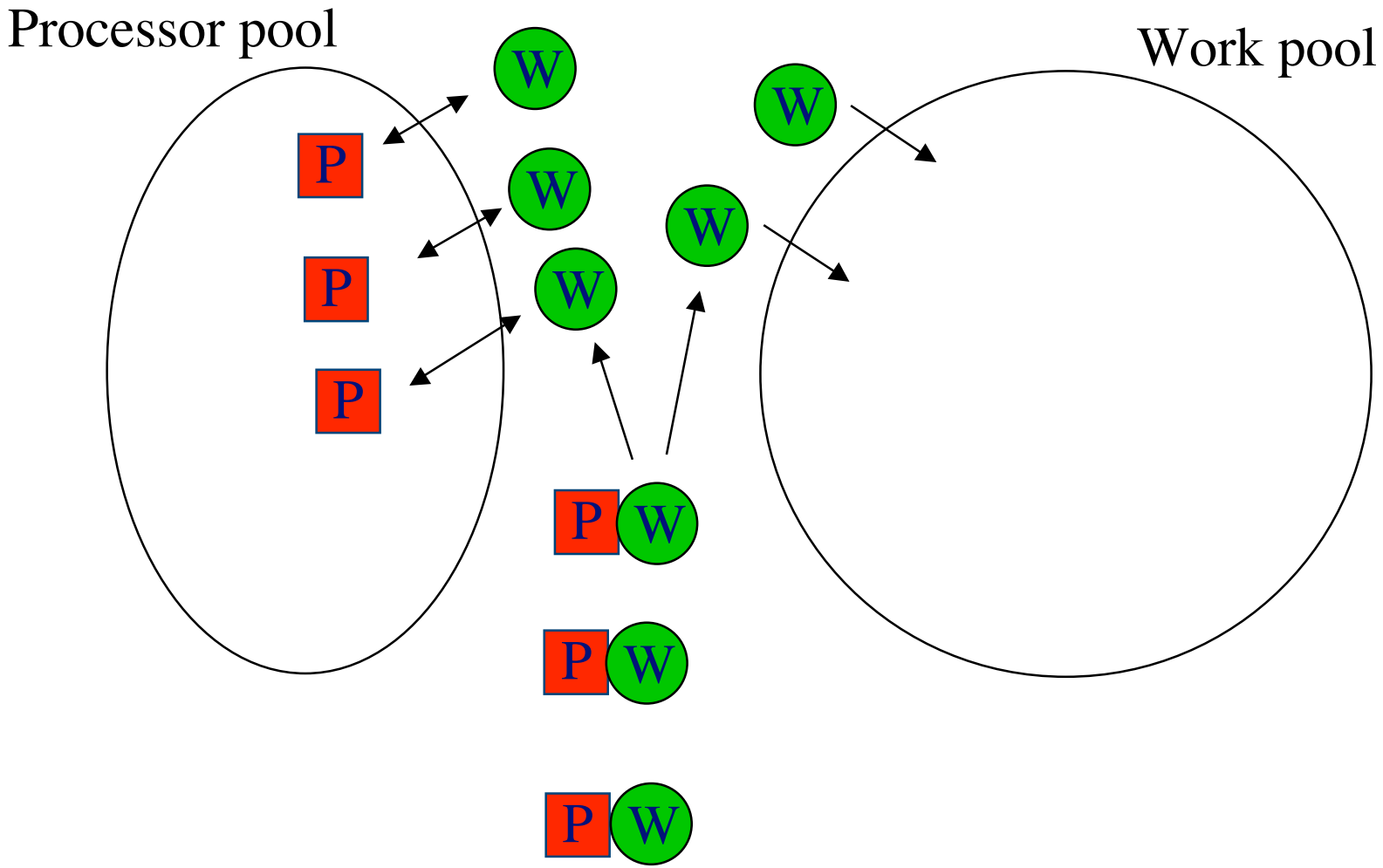
Processor/Work Pool Model

(surplus-processor configuration)



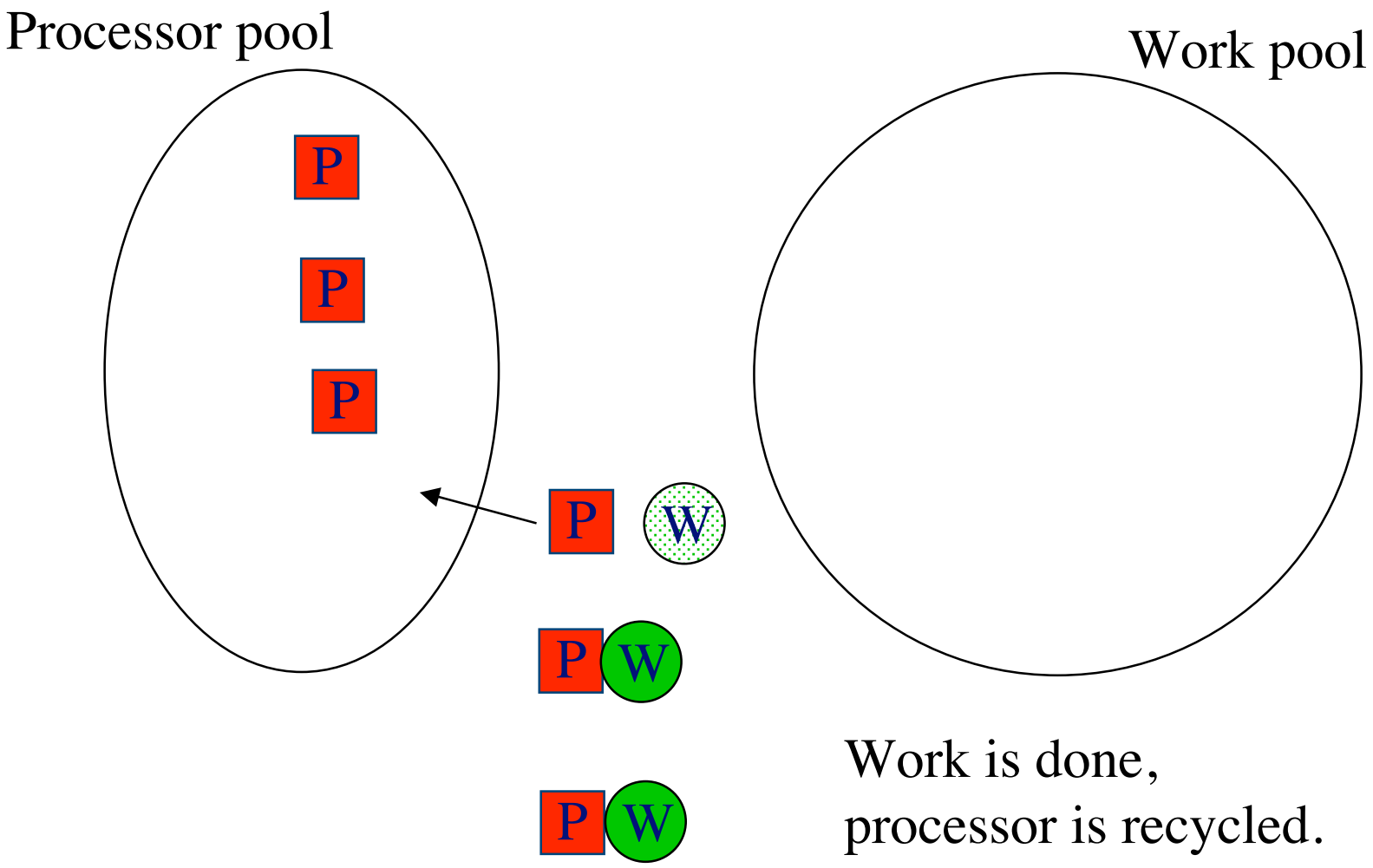
Processor/Work Pool Model

(new work being spawned)



Processor/Work Pool Model

(processor becoming idle due to work completed)



More Deja Vu

- This model used to schedule threads and processes.
- The work units can be other.
- The “processors” can be processes or threads themselves.

Processor/Work Pool Model

Advantages

Disadvantages

Processor/Work Pool Model

Advantages

- Low sensitivity to number of processors and work units
- Finer-grain work units simplify balancing of load
- Good way to hide latency
- Possibly a good basis for fault tolerance

Disadvantages

- Some overhead in managing work units
- May create a bottleneck if pools centralized