Multicore Computing
Lecture18 – Structured Patterns

남 범 석

bnam@skku.edu

Parallel Patterns

- **Parallel Patterns**: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a "vocabulary" for algorithm design
- Patterns are universal – they can be used in *any* parallel programming system

Structured Parallel Patterns

The following additional parallel patterns can be used for "**structured parallel programming**":

- Map
- Recurrence
- Scan
- Reduce
- Pack/expand
- Fork/join
- Pipeline
- Partition
- Segmentation
- Stencil
- Search/match
- Gather
- Merge scatter
- Priority scatter
- *Permutation scatter
- !Atomic scatter

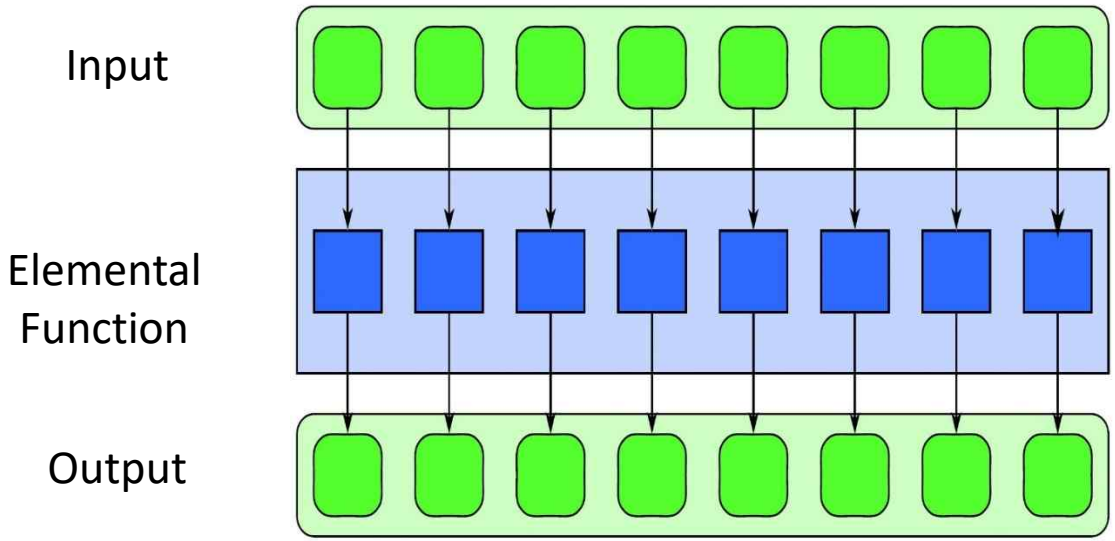
Using these patterns, threads and vector intrinsics can (mostly) be eliminated and the maintainability of software improved.

Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection

```
for (i=0; i<n; ++i) {
   process(x);
}
```

```
parfor(x in array){
   process(x);
}
```



Parallel Control Patterns: Map

- "Do the same thing many times"

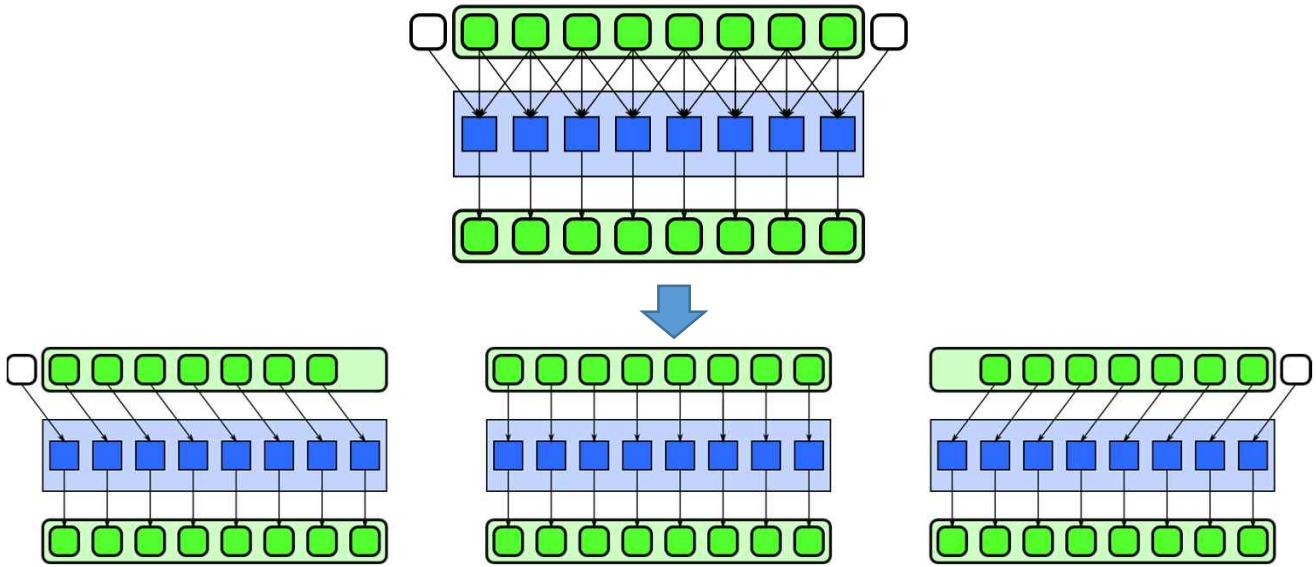
```
foreach i in foo:
    do something
```

- Well-known higher order function in languages like ML, Haskell, Scala applies a function to each element in a list and returns a list of results

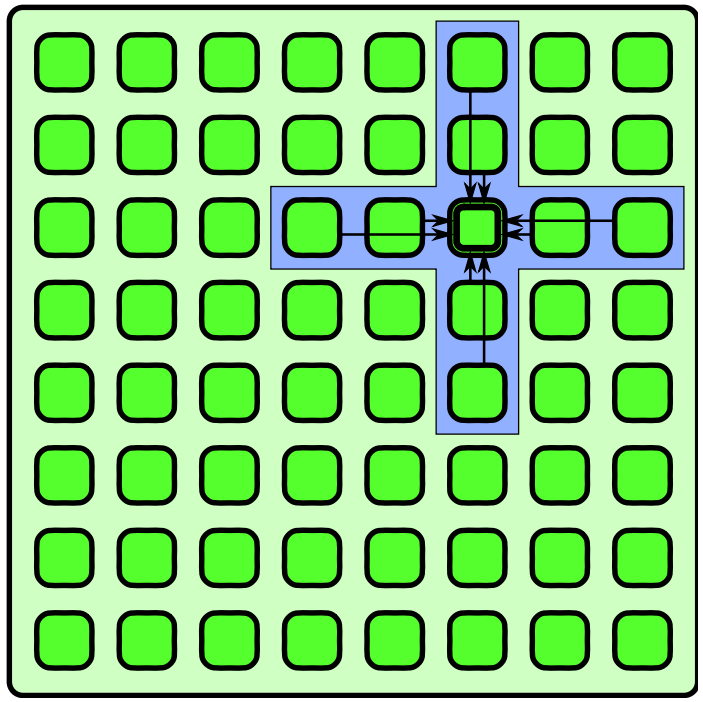
```
ghci> map (+3) [1,5,3,1,6]

[4,8,6,4,9]
```

Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of "neighbors", stencil is a generalization of map
- A stencil output is a function of a "neighborhood" of elements in an input collection



Parallel Control Patterns: nD Stencil

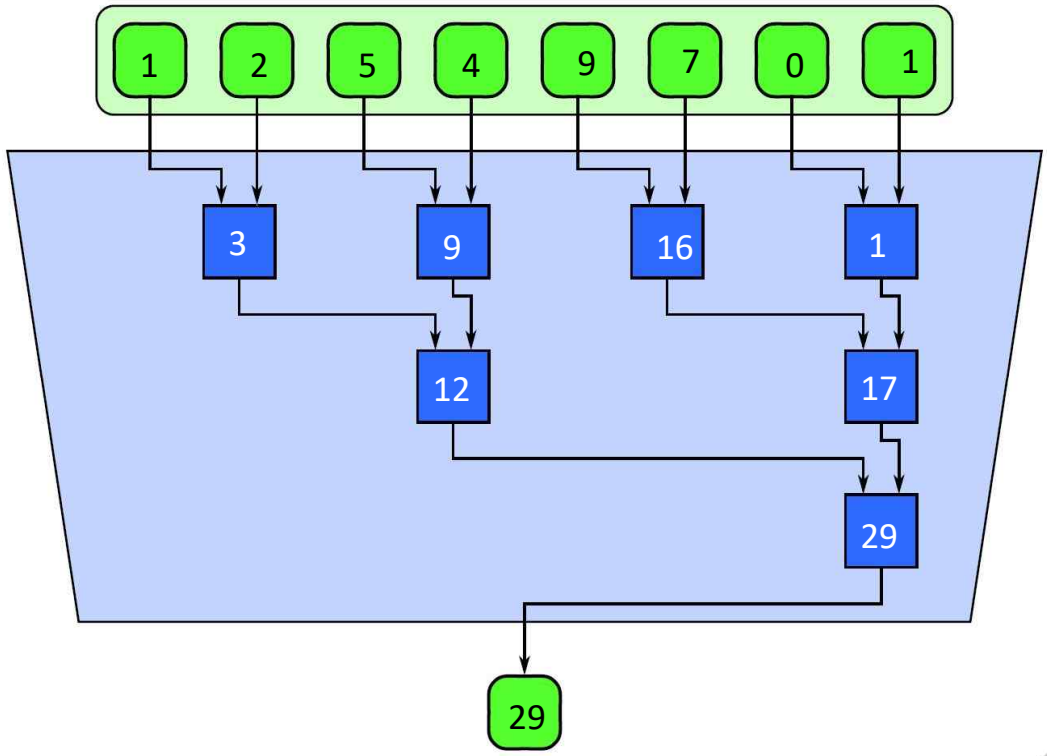


- *nD Stencil* applies a function to neighbourhoods of an nD array
- Neighbourhoods are given by set of relative offsets
- Boundary conditions need to be considered

Parallel Control Patterns: Reduction

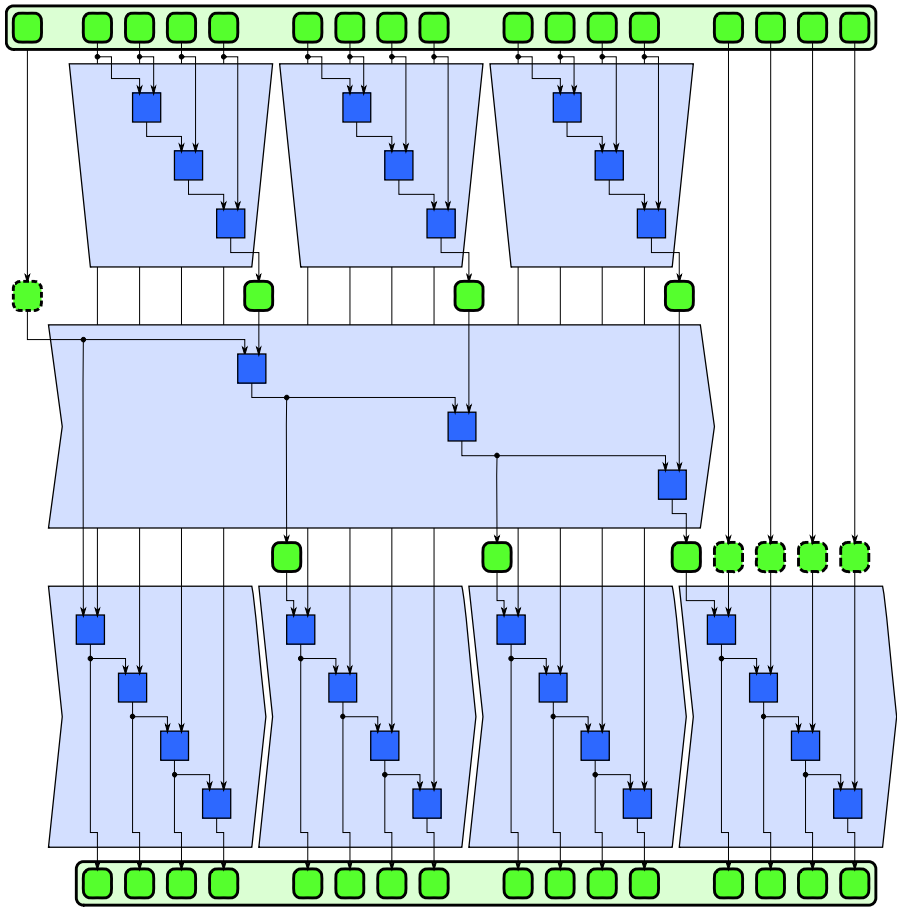
- **Reduction**: Combines every element in a collection using an associative "combiner function"
  - Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

```
b = 0;
for (i=0; i<n; ++i) {
    b += f(B[i]);
}
```



Parallel Control Patterns: Scan

- **Scan (Prefix Sum)**: a reduction of the input up to that point



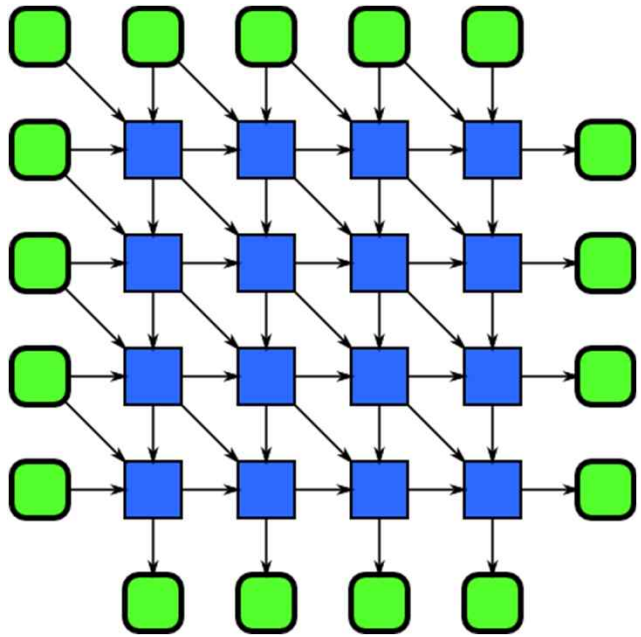
- *Scan* computes all partial reductions of a collection

```
A[0] = B[0] + init;
for (i=1; i<n; ++i) {
   A[i] = B[i] + A[i-1];
}
```

Parallel Control Patterns: Recurrence

- **Recurrence**: More complex version of map, where the loop iterations depend on one another
  - Similar to map, but elements can use outputs of adjacent elements as inputs
- For a recurrence to be computable, there *must* be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs

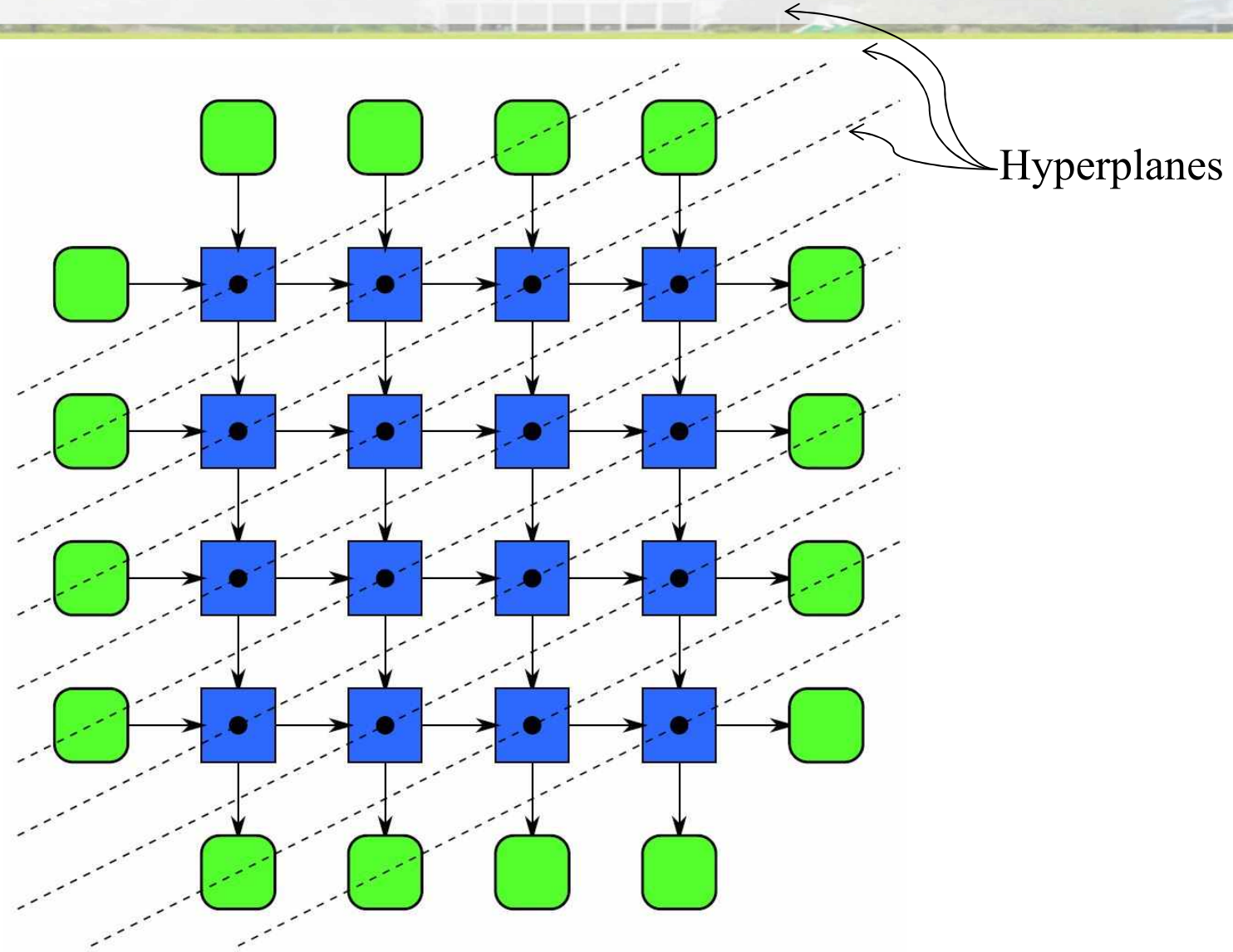
```
for (int i = 1; i < N; i++) {
   for (int j = 1; j < M; j++) {
      A[i][j] = f(
         A[i-1][j],
         A[i][j-1],
         A[i-1][j-1],
         B[i][j]);
   }
}
```



Parallel Control Patterns: Recurrence

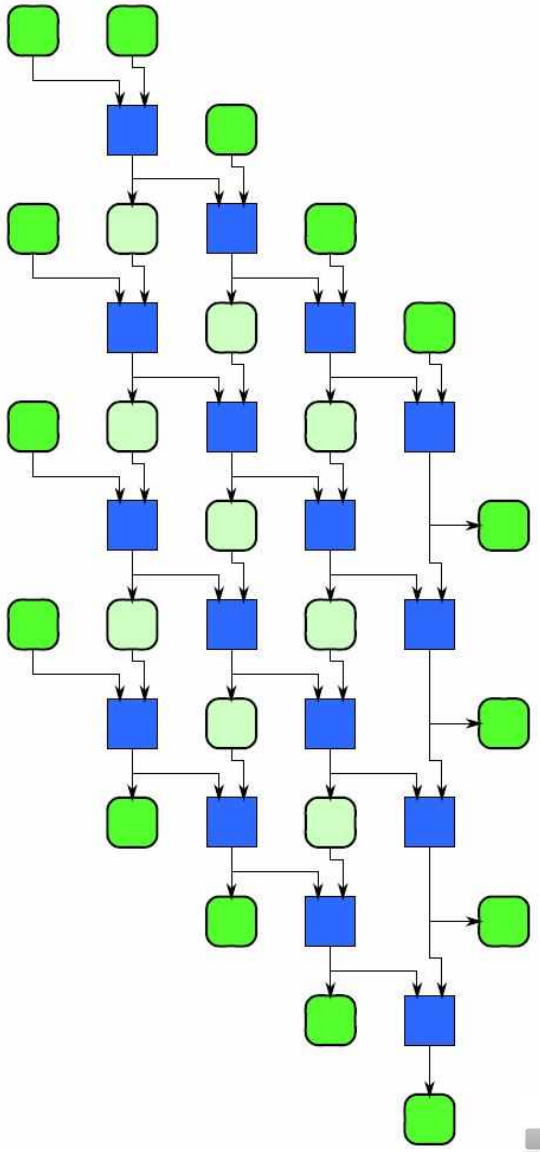
- This can still be parallelized!
- Trick: find a plane that cuts through grid of intermediate results
  - Previously computed values on one side of plane
  - Values to still be computed on other side of plane
  - Computation proceeds perpendicular to plane through time (this is known as a sweep)
- This plane is called a separating hyperplane

Parallel Control Patterns: Recurrence



Parallel Control Patterns: Recurrence

- Same grid of intermediate results
- Each level corresponds to a loop iteration
- Computation proceeds downward



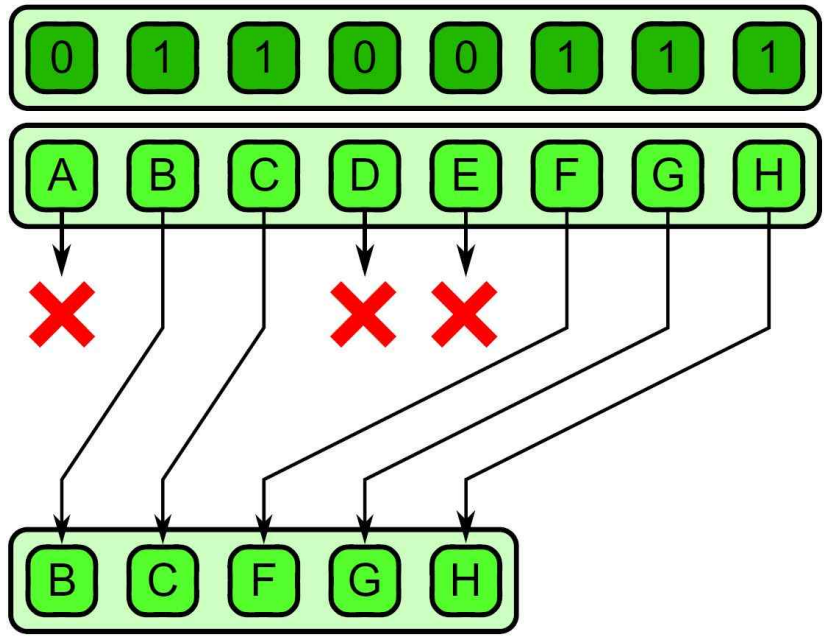
Parallel Data Management Patterns: Pack

- **Pack** is used to eliminate unused space in a collection
- Elements marked *false* are discarded, the remaining elements are placed in a contiguous sequence in the same order
- Useful when used with map
- **Unpack** is the inverse

  and is used to place

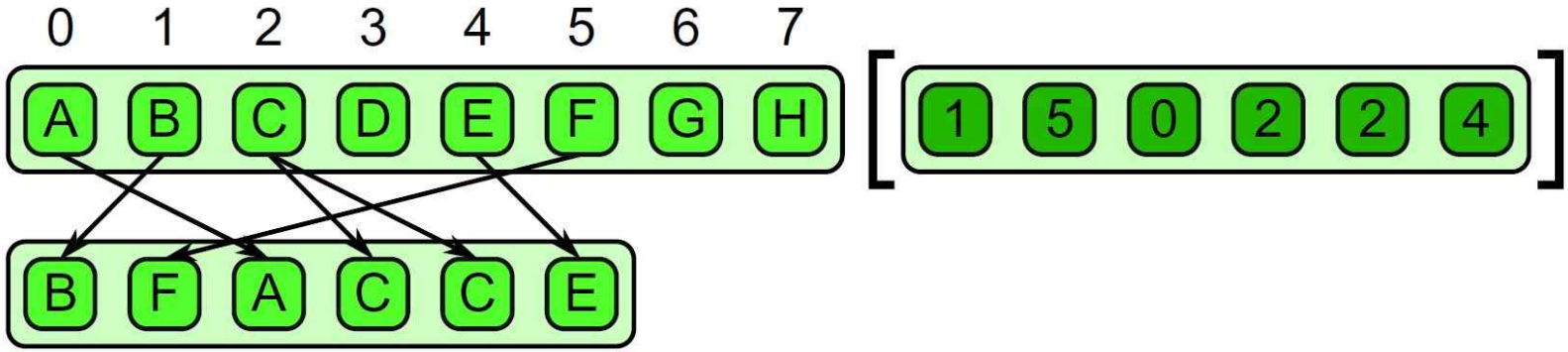
  elements back in their

  original locations

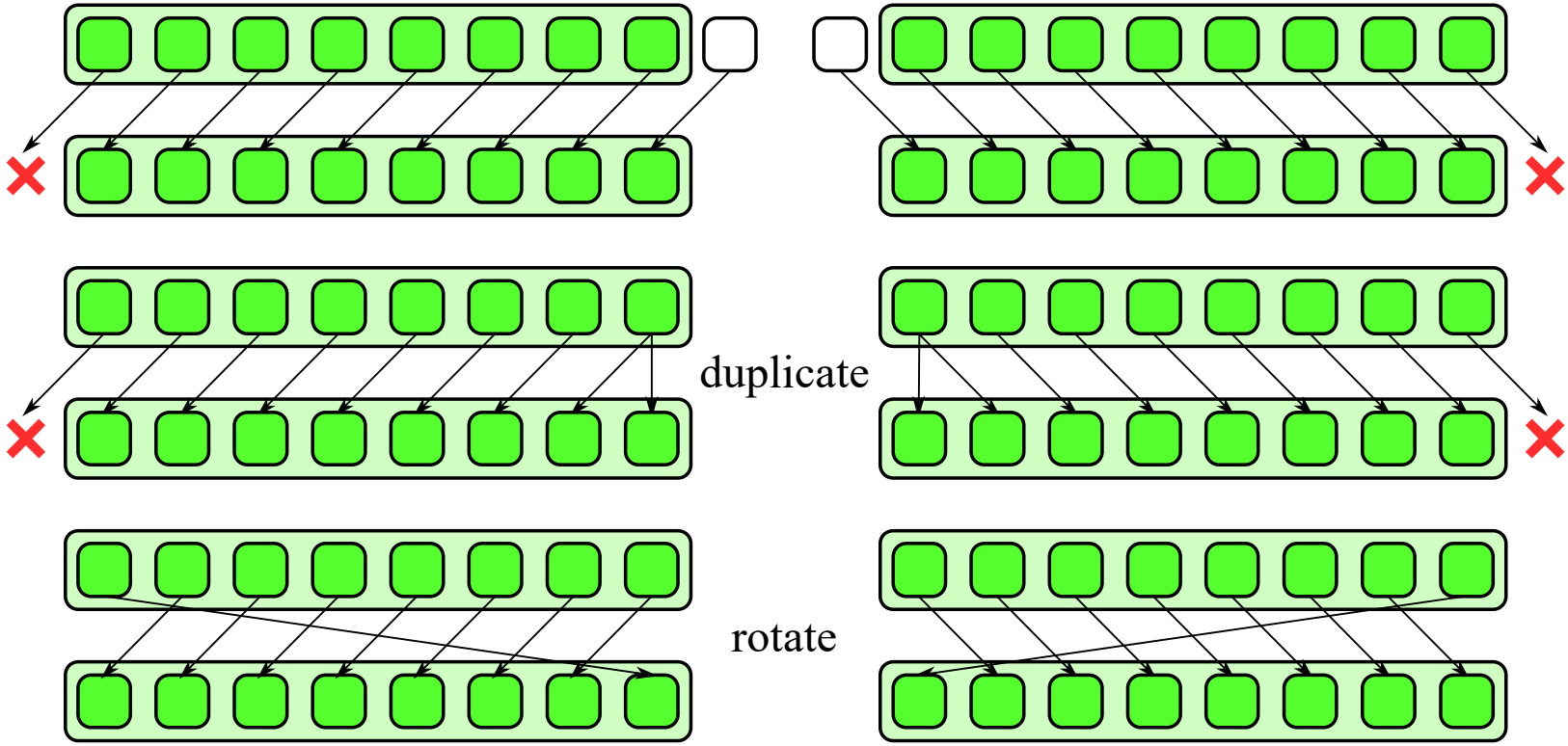


Parallel Data Management Patterns: Gather

- **Gather** reads a collection of data given a collection of indices
- Think of a combination of map and random serial reads
- The output collection shares the same type as the input collection, but it share the same shape as the indices collection



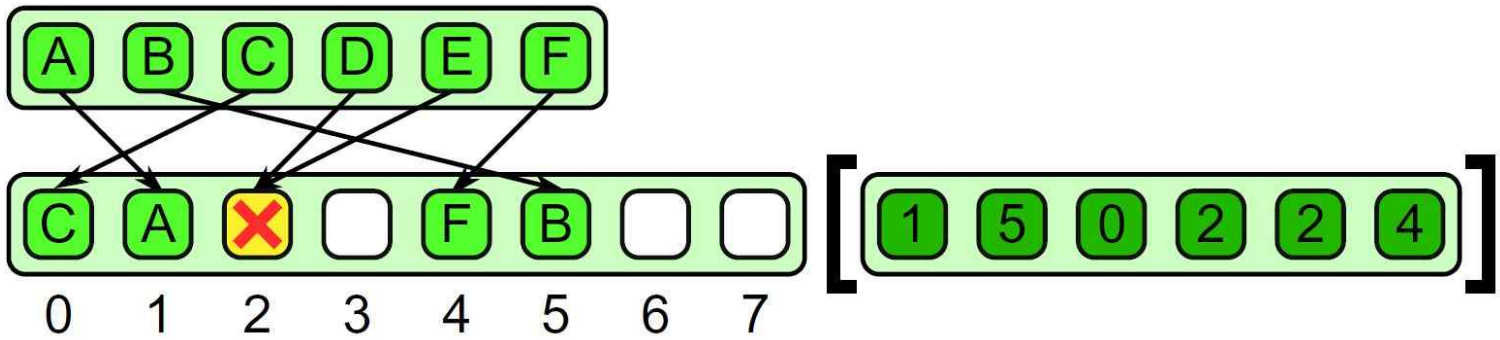
Special Case of Gather: Shifts



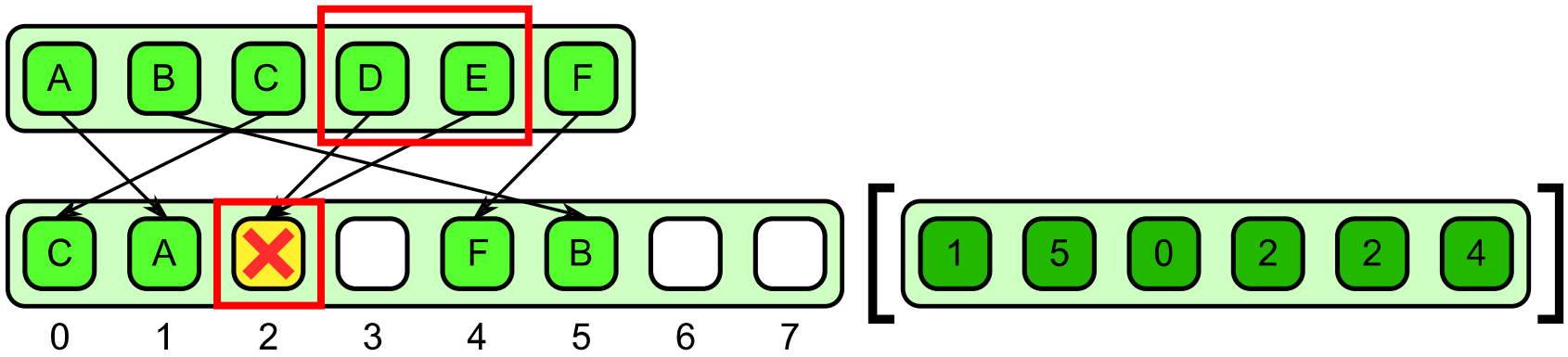
- Moves data to the left or right in memory
- Data accesses are offset by fixed distances

Parallel Data Management Patterns: Scatter

- **Scatter** is the inverse of gather
- A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index
- Race conditions can occur when we have two writes to the same location!



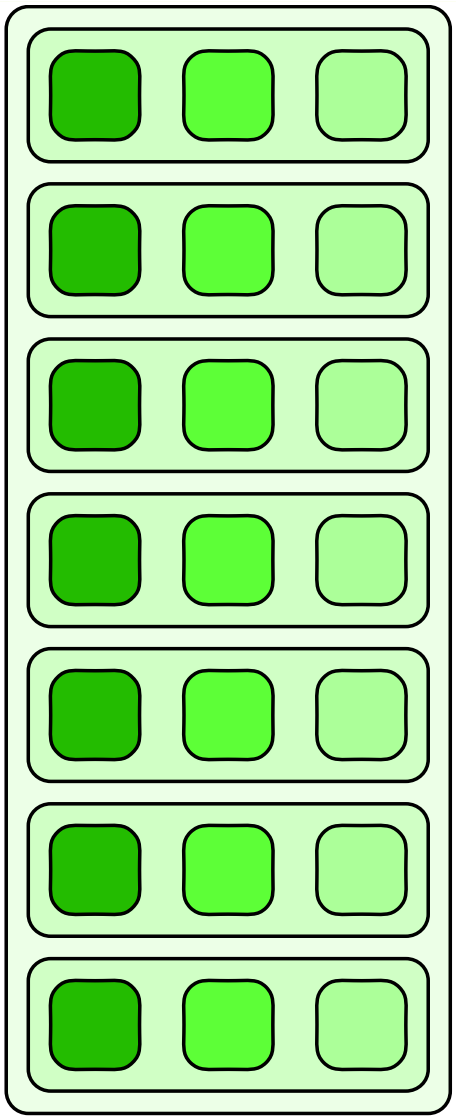
Scatter: Race Conditions



Given a collection of input data
and a collection of write locations
scatter data to the output collection

Race Condition: Two (or more) values being written to the same location in output collection.
Result is undefined unless enforce rules.
Need rules to resolve collisions!

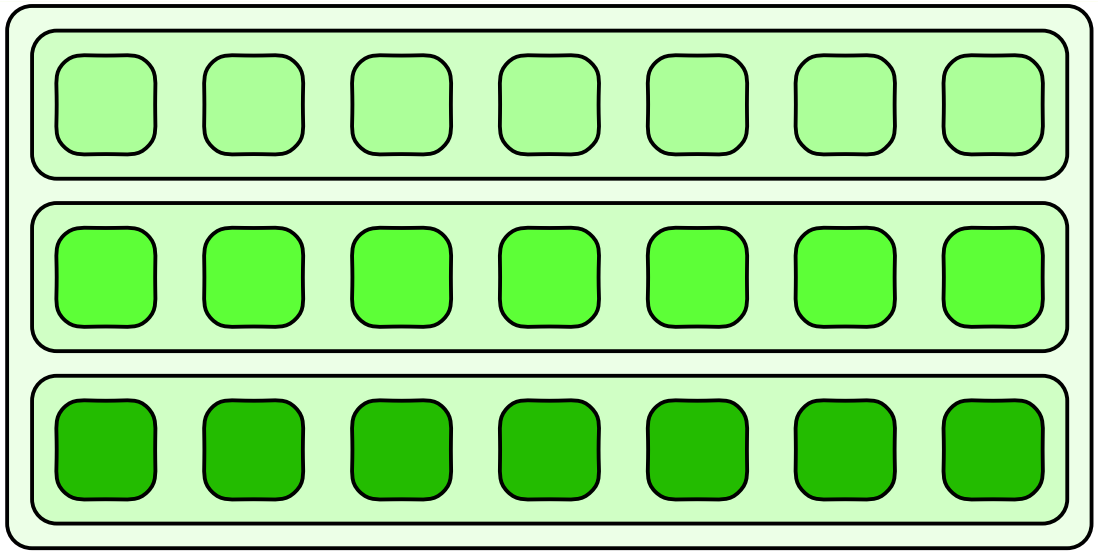
Array of Structures (AoS)



```
struct point{
    float x, y;
};
struct point set[100];
```

- Pros:
  - May lead to better cache utilization if data is accessed randomly
- Cons:
  - Does not align data well for vectorization

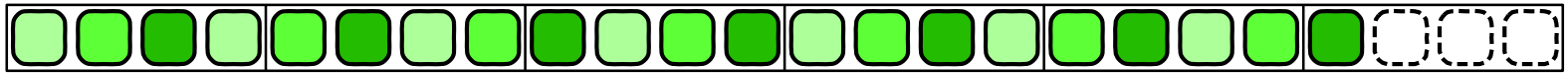
Structures of Arrays (SoA)



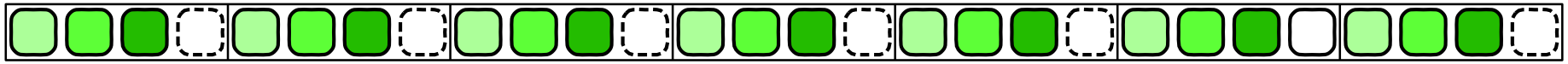
```
struct point{
    float x[100], y[100];
};
struct point set;
```

Data Layout Options

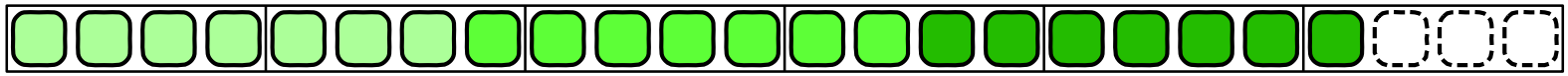
Array of Structures (AoS), padding at end



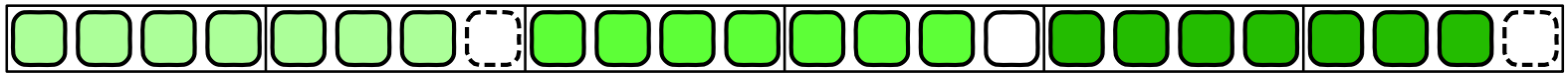
Array of Structures (AoS), padding after each structure



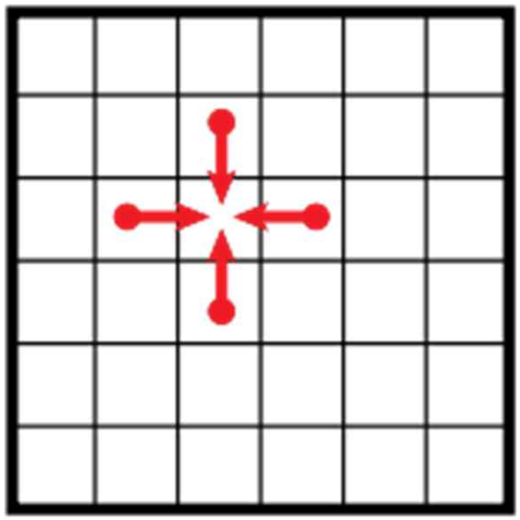
Structure of Arrays (SoA), padding at end



Structure of Arrays (SoA), padding after each component

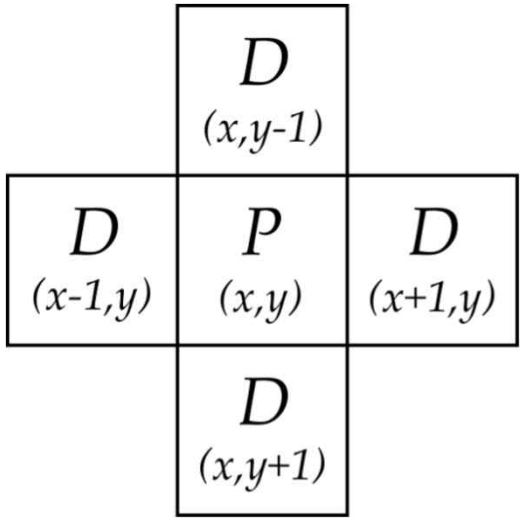


2-Dimensional Stencils



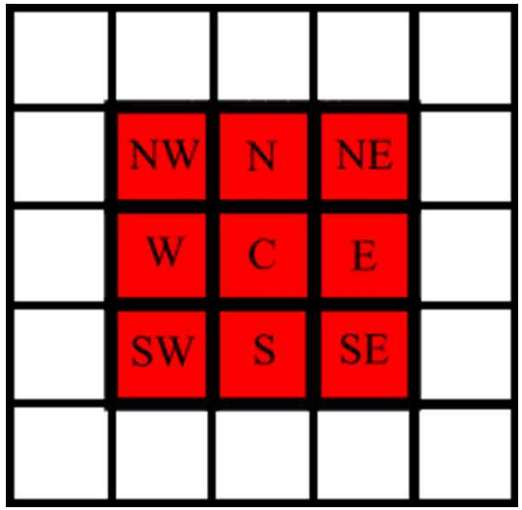
4-point stencil

Center cell (P)
is not used



5-point stencil

Center cell (P)
is used as well

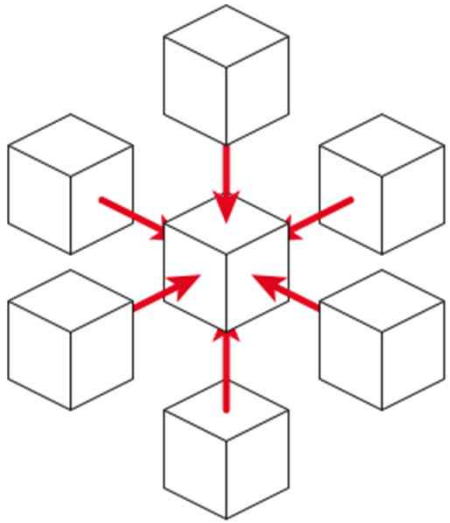


9-point stencil

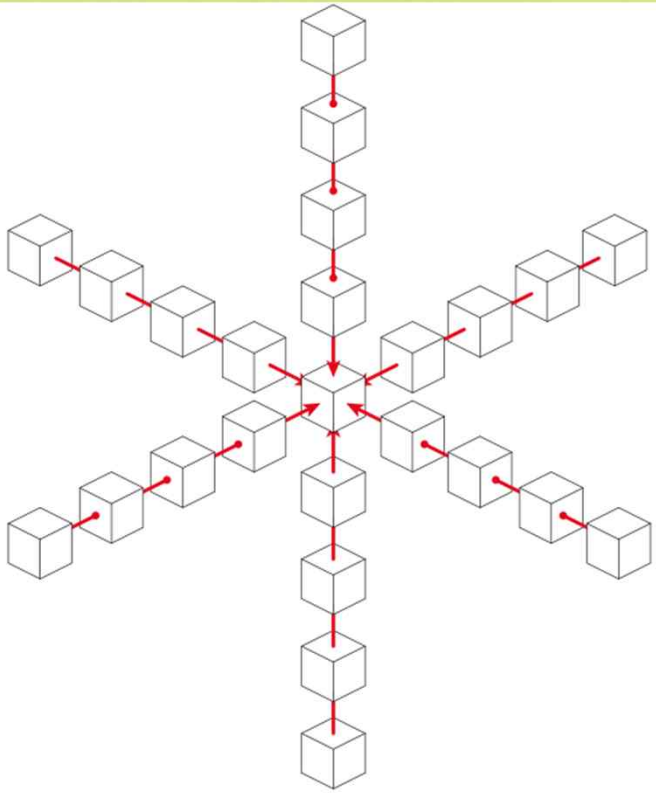
Center cell (C)
is used as well

Source: http://en.wikipedia.org/wiki/Stencil_code

3-Dimensional Stencils



6-point stencil
(7-point stencil)

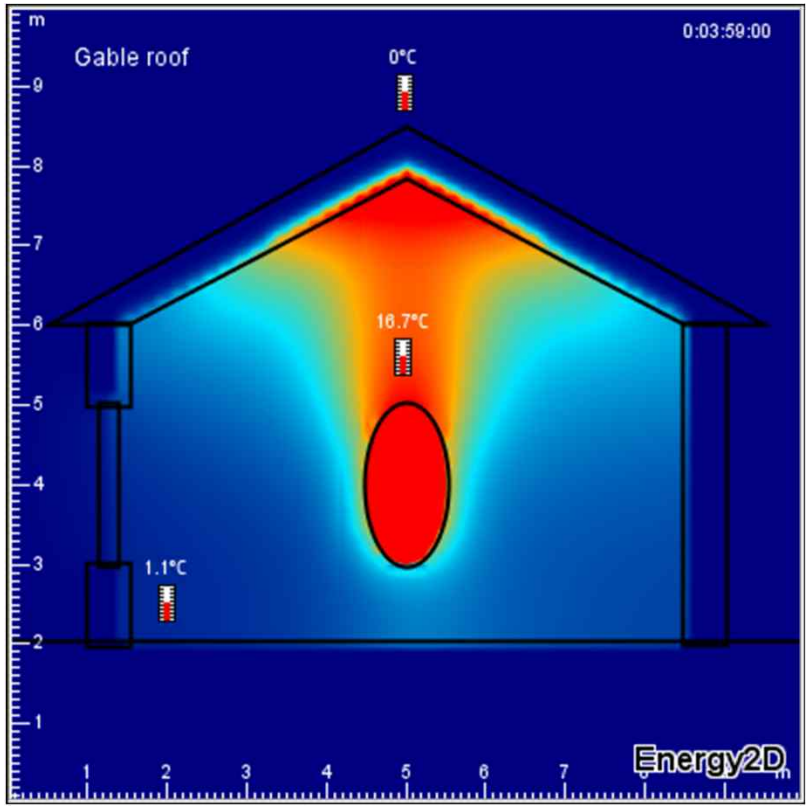


24-point stencil
(25-point stencil)

Source: http://en.wikipedia.org/wiki/Stencil_code

Stencil Example: Heat Transfer Simulation

- Here is our array, A



A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

Stencil Example: Heat Transfer Simulation

- Here is our array, A
- B is the output array
  - Initialize to all 0
- Apply a stencil operation to the inner square of the form:
- B(i,j) = avg(A(i,j),
  A(i-1,j), A(i+1,j),
  A(i,j-1), A(i,j+1))

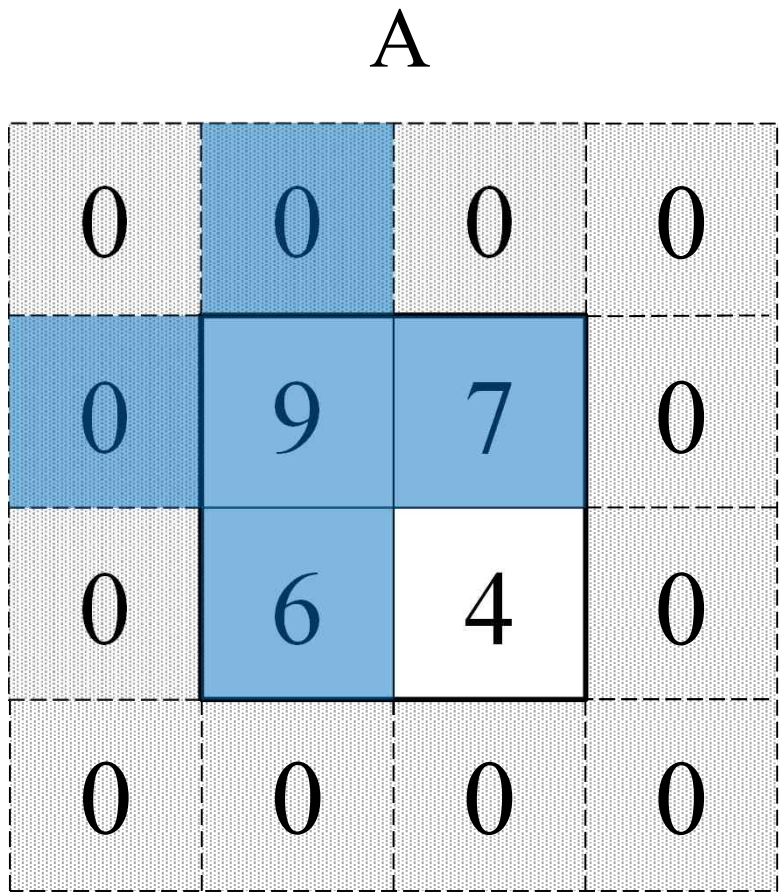
A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

What is the output?

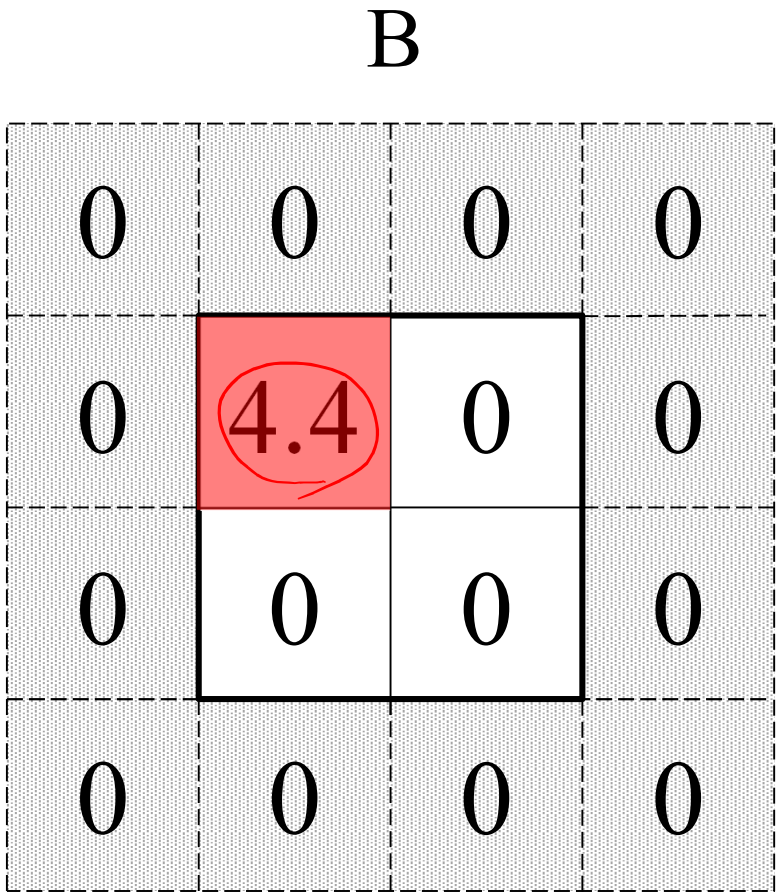
Stencil Example: Heat Transfer Simulation

- Average all blue squares

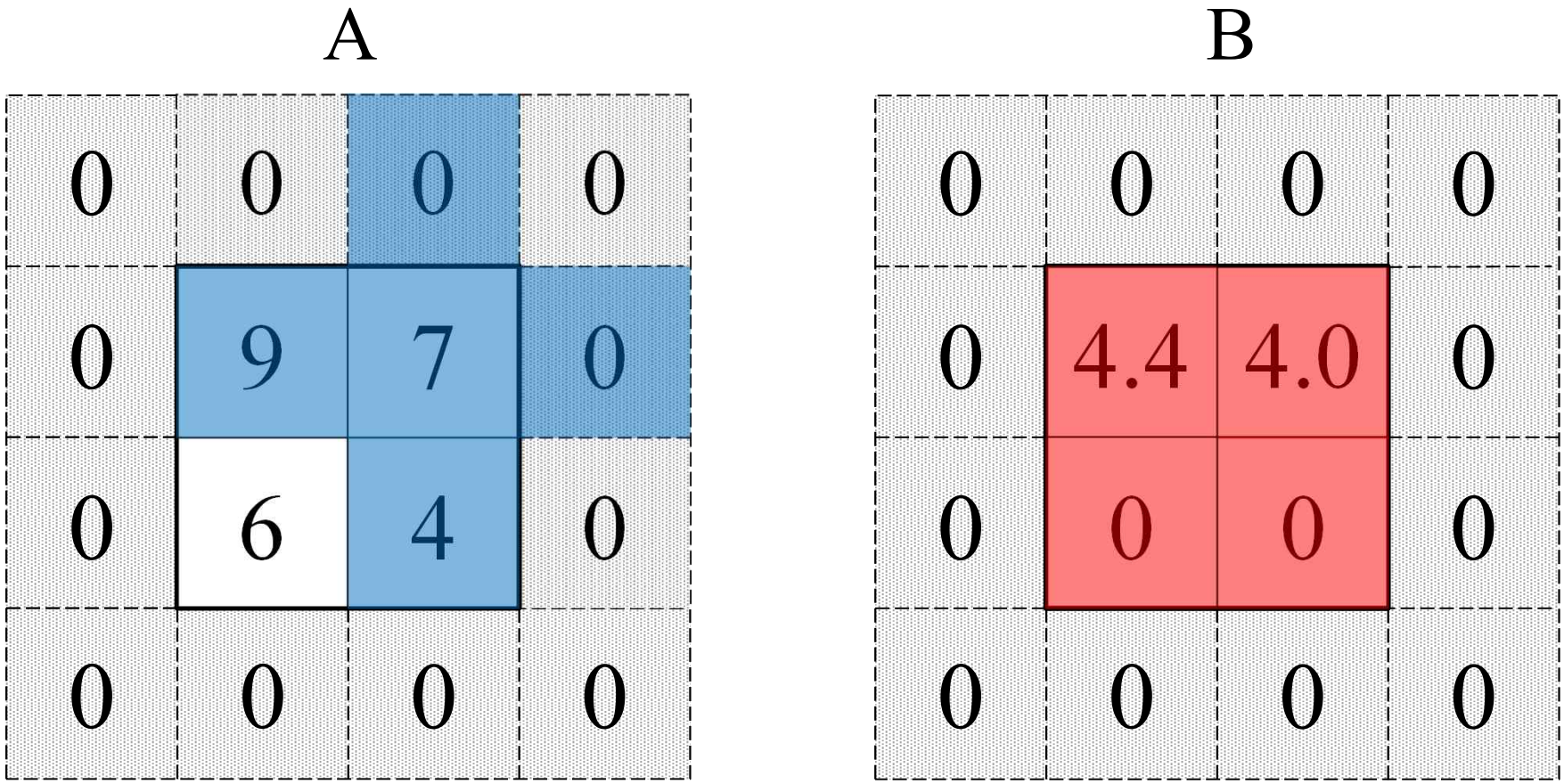


Stencil Example: Heat Transfer Simulation

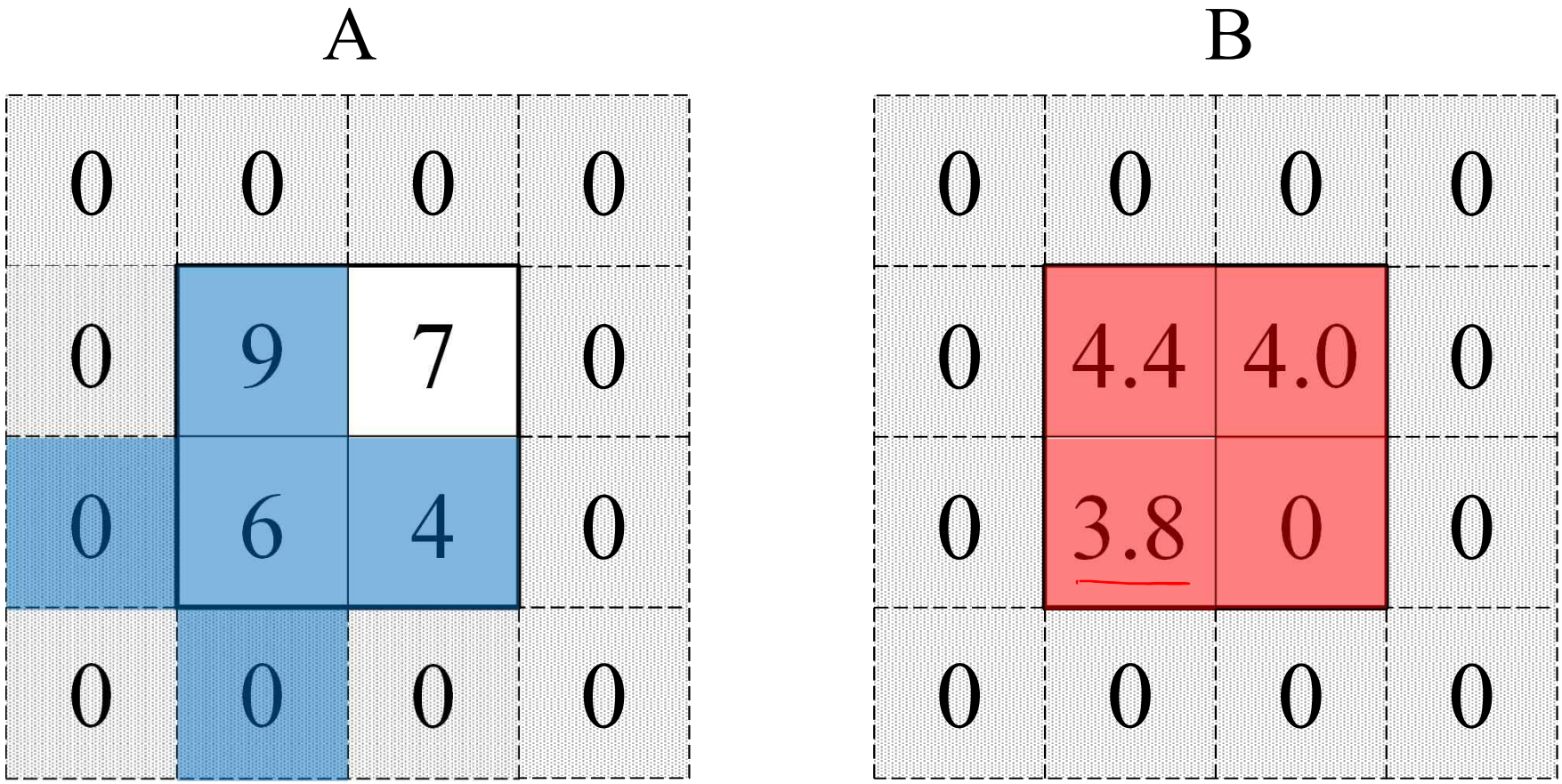
- Average all blue squares
- Store result in B



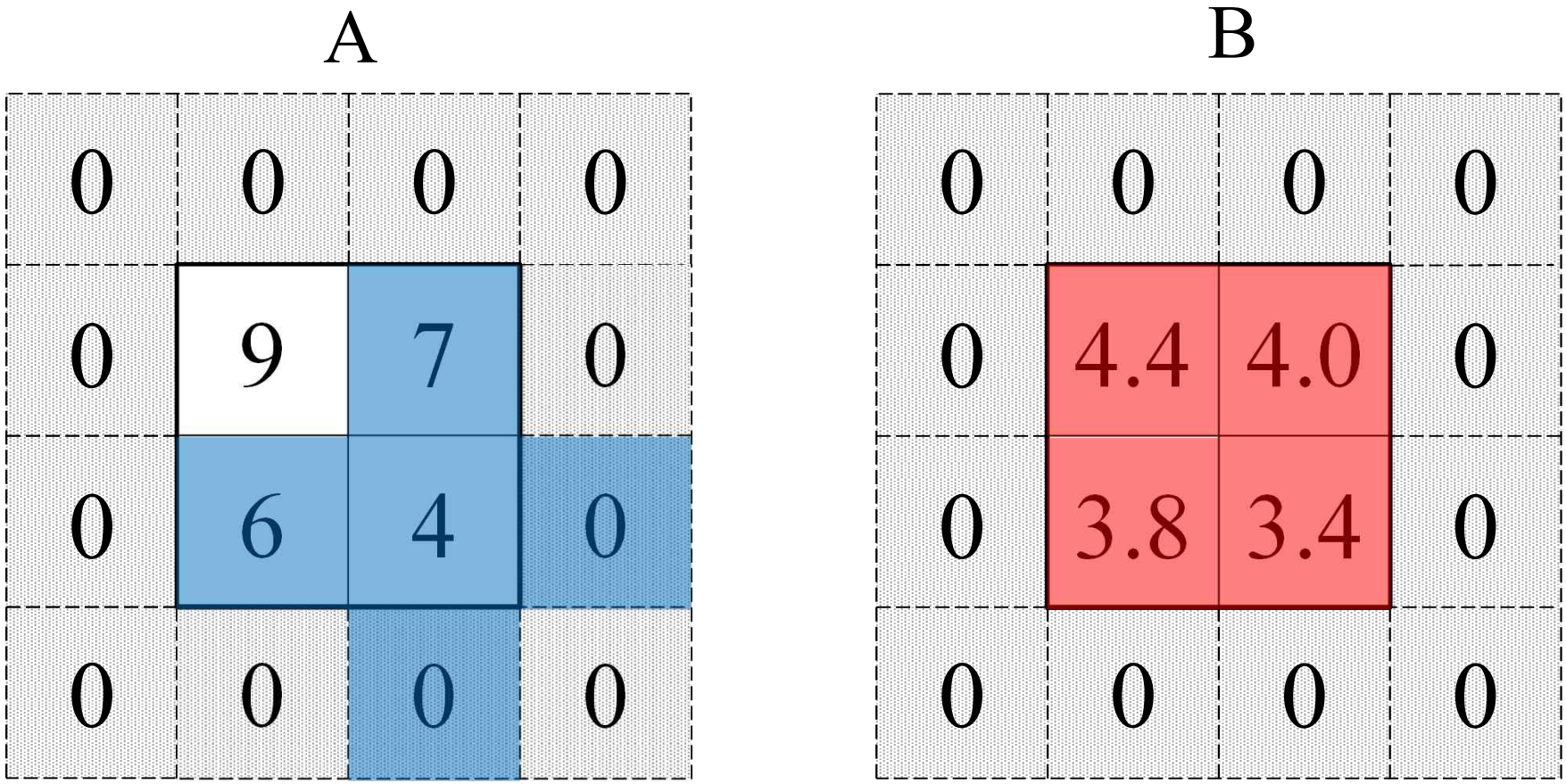
Stencil Example: Heat Transfer Simulation



Stencil Example: Heat Transfer Simulation



Stencil Example: Heat Transfer Simulation



Iterative Methods

- Iterative codes are ones that update their data in steps
  - At each step, a new value of an element is computed using a formula based on other elements
  - Once all elements are updated, the computation proceeds to the next step or completes
- Iterative methods are most commonly found in computer simulations of physical systems for scientific and engineering applications
  - Computational fluid dynamics
  - Electromagnetics modeling
- They are often applied to solve very large linear equations
  - Jacobi iteration
  - Gauss-Seidel iteration
  - Successive over relaxation

Jacobi Iterative Methods

- Ax=B
- $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$
  $a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$
  $\ddot{a}_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$
- To begin, solve the 1st equation for x_1, the 2nd equation for x_2, and so on to obtain the rewritten equations:
  - $x_1 = 1/a_{11}(b_1 - a_{12}x_2 - a_{13}x_3 - \ldots a_{1n}x_n)$
    $x_2 = 1/a_{22}(b_2 - a_{22}x_1 - a_{23}x_3 - \ldots a_{2n}x_n)$
    ...
    $x_n = 1/a_{nn}(b_n - a_{n2}x_1 - a_{n3}x_2 - \ldots a_{nn-1}x_{n-1})$
- Then make an initial guess of the solution
  $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \ldots x_n^{(0)})$
- Substitute these values into the right hand side of the rewritten equations to obtain the first approximation
  $x^{(1)} = (x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \ldots x_n^{(1)})$ // output of the first iteration
- In the same way, the second approximation is computed.
- By repeated iterations, we form a sequence of approximation

Jacobi Iterative Methods

- Example:
  - Apply the Jacobi method to solve

$$5x_1 - 2x_2 + 3x_n = -1$$
$$-3x_1 + 9x_2 + x_n = 2$$
$$2x_1 - x_2 - 7x_n = 3$$

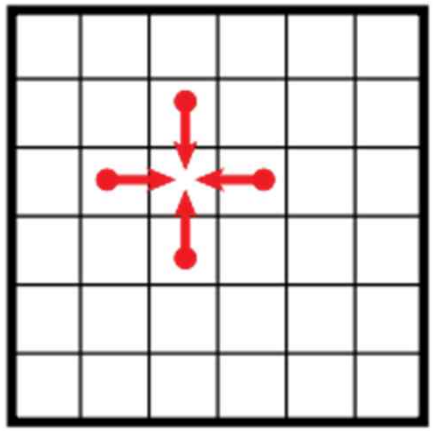
$$\rightarrow$$

$$x_1 = -1/5 + 2/5\, x_2 - 3/5\, x_3$$
$$x_2 = 2/9 + 3/9\, x_1 - 1/9\, x_3$$
$$x_3 = -3/7 + 2/7\, x_1 - 1/7\, x_2$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$X_1^{(k)}$	0.000	-0.200	0.146	0.192			
$X_2^{(k)}$	0.000	0.222	0.203	0.328			
$X_3^{(k)}$	0.000	-0.429	-0.517	-0.416			

2-Dimension Jacobi Iteration

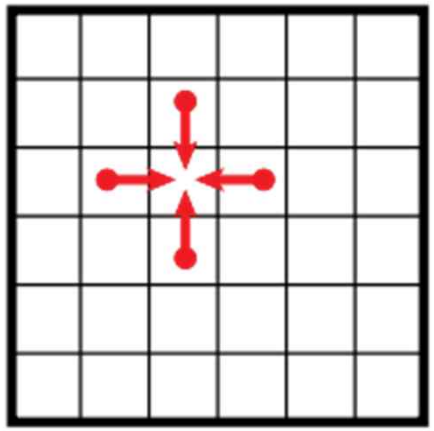
- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge



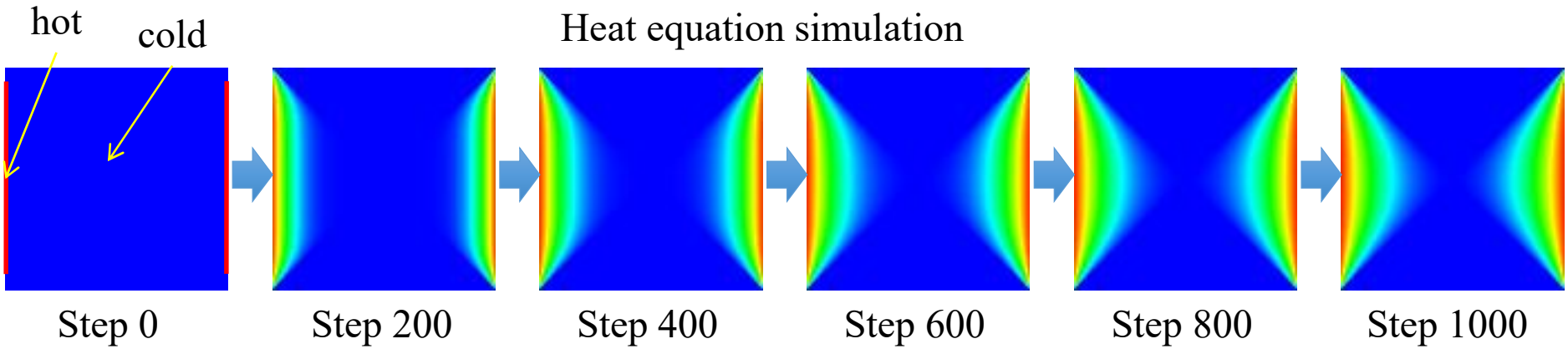
4-point stencil

2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge



4-point stencil



Gauss-Seidel Method

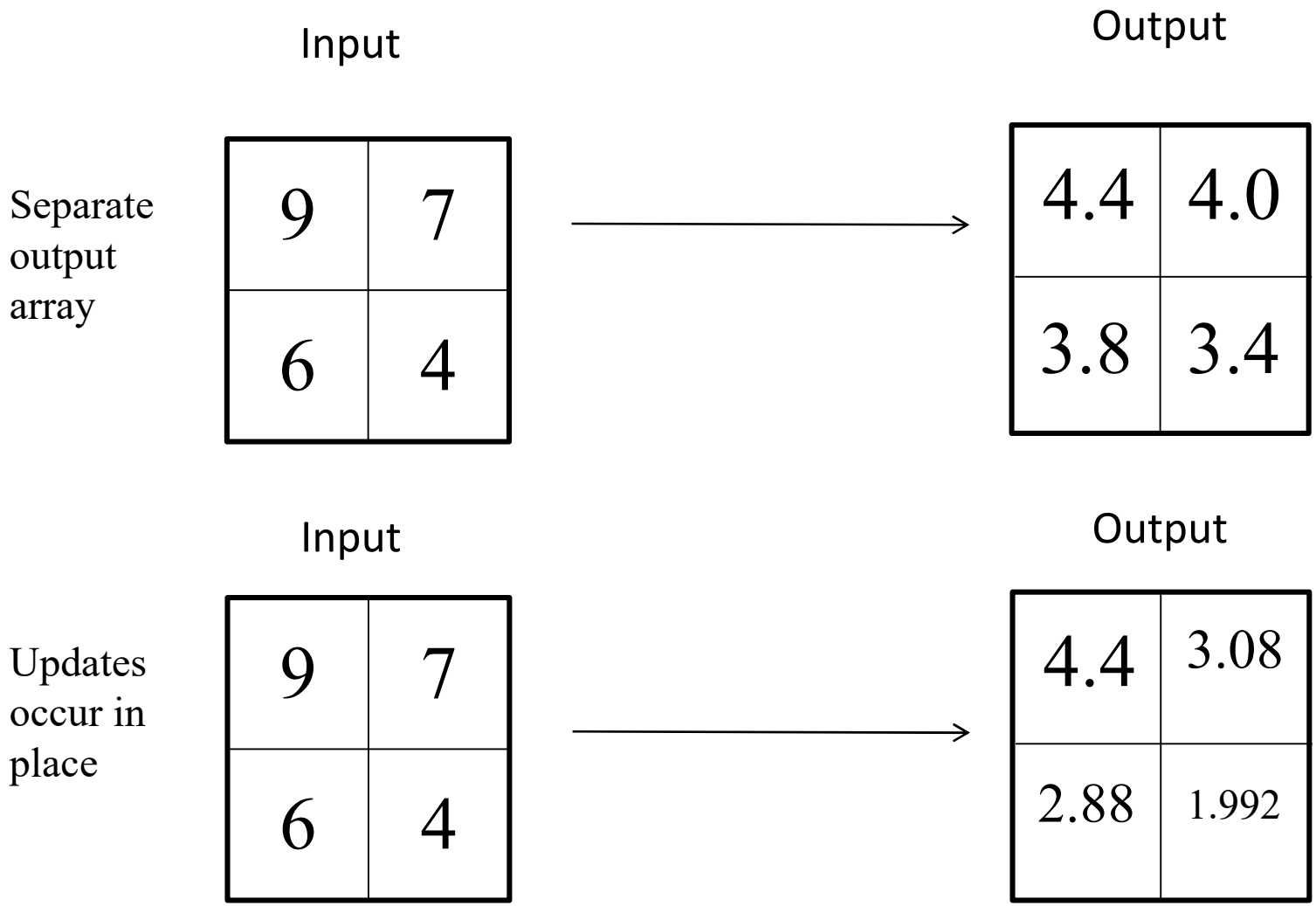
- With the Jacobi method, the values of $x_i^{(k)}$ obtained in the k^{th} iteration need to remain unchanged until the entire $(k+1)^{th}$ iteration has been calculated.
- With the Gauss-Seidel method, we use the new values $x_i^{(k+1)}$ as soon as they are known.
- For example, once we have computed $x_1^{(k+1)}$ from the first iteration, its value is then used in the second equation to obtain the new $x_2^{(k+1)}$, and so on.

Serial Stencil Example

```
15      // array to hold neighbors
16      In neighborhood[NumOff];
17      // loop over all output locations
18      for (int i = 0; i < n; ++i) {
19          // loop over all offsets and gather neighborhood
20          for (int j = 0; j < NumOff; ++j) {
21              // get index of jth input location
22              int k = i+offsets[j];
23              if (0 <= k && k < n) {
24                  // read input location
25                  neighborhood[j] = a[k];
26              } else {
27                  // handle boundary case
28                  neighborhood[j] = b;
29              }
30          }
31          // compute output value from input neighborhood
32  a[i]    r[i] = func(neighborhood);
33      }
34  }
```

Updates occur in place!!!

Different Cases: Jacobi vs Gauss-Seidel



Gauss-Seidel method

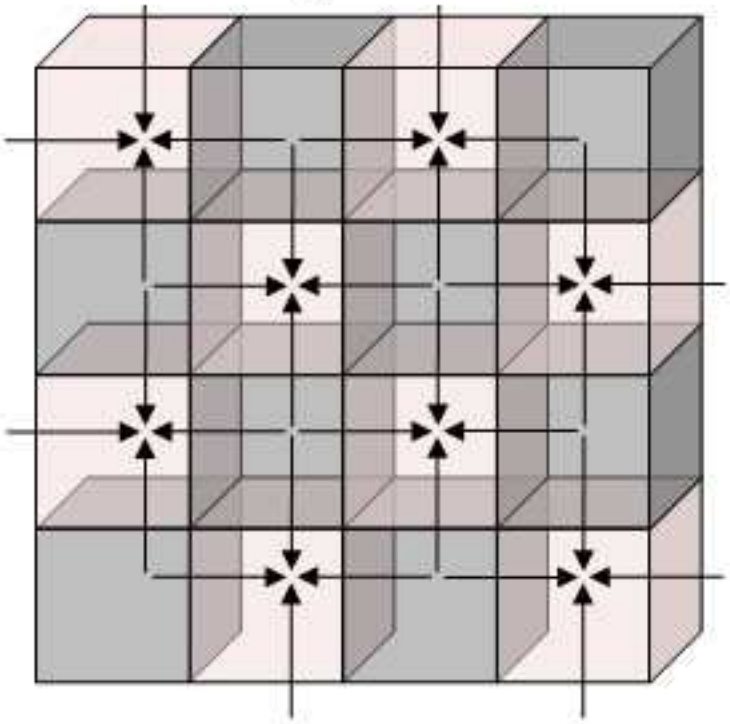
```
// Parallel Gauss-Seidel method
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
 dmax = 0; // maximum variation of values u
#pragma omp parallel for shared(u,n,dmax) private(i,temp,d)
  for (i=1; i<N+1;i++){
#pragma omp parallel for shared(u,n,dmax) private(j,temp,d)
      for ( j=1; j<N+1;j++) {
            temp = u[i][j];
            u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] – h*h*f[i][j]);
            d = fabs (temp-u[i][j])
            omp set_lock(dmax_lock);
                if (dmax < d) dmax = d;
            omp_unset_lock(dmax_lock);
      } // end of inner parallel region
  } // end of outer parallel region
} while ( dmax > eps );
```


Successive Over Relaxation (SOR)

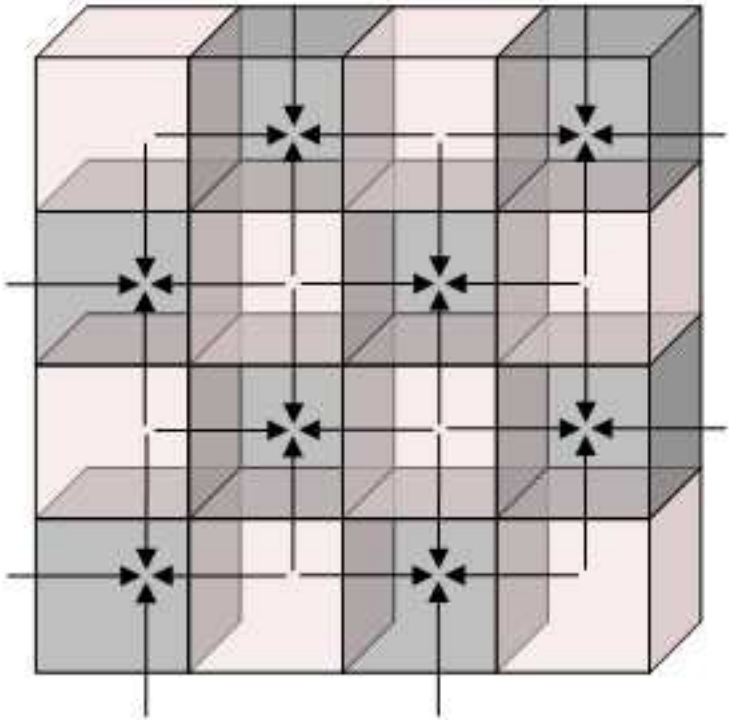
- SOR is an alternate method of solving linear equations
- While the Jacobi iteration scheme is very simple and parallelizable, its slow convergent rate renders it impractical for any "real world" applications
- One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation
- It also allows a method known as Red-Black SOR to be used to enable parallel updates in place

Red / Black SOR

Pass 1:Writing to red cells,
reading from black



Pass 2:Writing to black cells,
reading from red



Programming Model Support for Parallel Patterns

Table 3.2 Summary of programming model support for the patterns discussed in this book. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. P: Implementations of one pattern in terms of others, listed under the pattern being implemented. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily). When examples exist in this book of a particular pattern with a particular model, section references are given.

Parallel Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
Parallel nesting	F	F			
Map	F 4.2.3; 4.3.3 11	F 4.2.4;4.2.5; 4.3.4;4.3.5 11	F 4.2.6; 4.3.6	F 4.2.7;4.2.8; 4.3.7	F 4.2.9; 4.3.8
Stencil	I 10	I 10	I	F 10	I
Workpile	F				I
Reduction	F 5.3.4 11	F 5.3.5 11	F 5.3.6	F 5.3.7	I
Scan	F 5.6.5 14	I 5.6.3 *P 8.11* 14	I 5.6.4 *P 5.4.4*	F 5.6.6	I
Fork–join	F 8.9.2 13	F 8.7; 8.9.1 13	I		
Recurrence		*P 8.12*			
Superscalar sequence					F
Futures					
Speculative selection					
Pack	I 14	I 14	I	F	I
Expand	I	I	I	I	I
Pipeline	F 12	I 12	I		
Geometric decomposition	I 15	I 15	I	I	I
Search	I	I	I	I	I
Category reduction	I	I	I	I	I
Gather	I	F	I	F	I
Atomic scatter	F	I	I		I
Permutation scatter	F	F	F	F	F
Merge scatter	I	I	I	F	I
Priority scatter					

Programming Model Support for Patterns

Table 3.3 Additional patterns discussed. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily).

Parallel Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
Superscalar sequence	I	I	I		F
Futures	I	I	I		I
Speculative selection	I				
Workpile	F	I	I		I
Expand	I	I	I	I	I
Search	I	I	I	I	I
Category reduction	I	I	I	I	I
Atomic scatter	F	I	I		I
Permutation scatter	F	F	F	F	F
Merge scatter	I	I	I	F	I
Priority scatter