UNIVERSIDAD DEL BÍO-BÍO
FACULTAD DE CIENCIAS EMPRESARIALES

Parallel Programming Patterns Heterogeneous Computing

Professor: Dr. Joel Fuentes - jfuentes@ubiobio.cl

Teaching Assistants:
- Daniel López - daniel.lopez1701@alumnos.ubiobio.cl
- Sebastián González - sebastian.gonzalez1801@alumnos.ubiobio.cl

Course website: http://www.face.ubiobio.cl/~jfuentes/classes/ch

Design patterns

1. Parallel Control Patterns
2. Data Management Patterns
3. Other Patterns

Parallel design patterns

- A design pattern is a combination of recurring tasks that solves a specific problem in designing parallel algorithms.
- Patterns provide a "vocabulary" for algorithm design.
- It can be useful to compare parallel patterns with serial patterns.
- Patterns are universal, they can be used in any parallel programming system.

Parallel Control Patterns

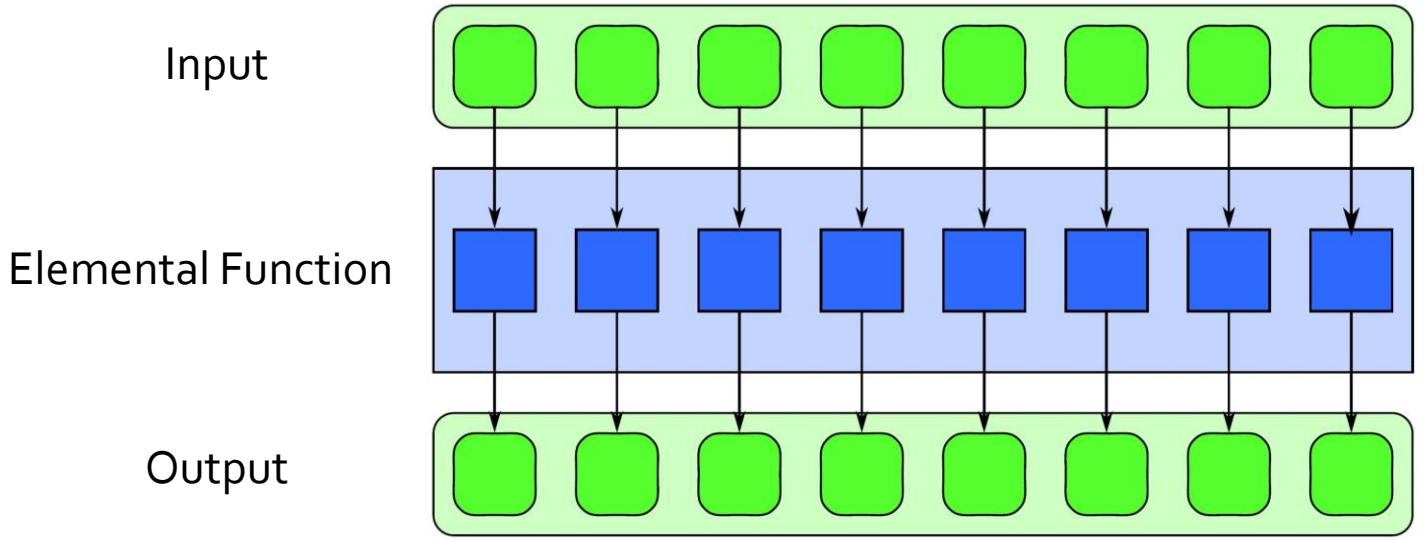
- Extend from serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but with more flexible specifications.
- Parallel control patterns: **fork-join**, **map**, **stencil**, **reduction**, **scan**, **recurrence**

Parallel Control Patterns: Fork-join

- **Fork-join**: allows you to control the flow to multiple parallel flows, and then join them.
- Many programming languages implement this pattern by **spawn** and **sync**
  - The call tree is a tree of parallel calls and functions that are executed in parallel flow (spawned)

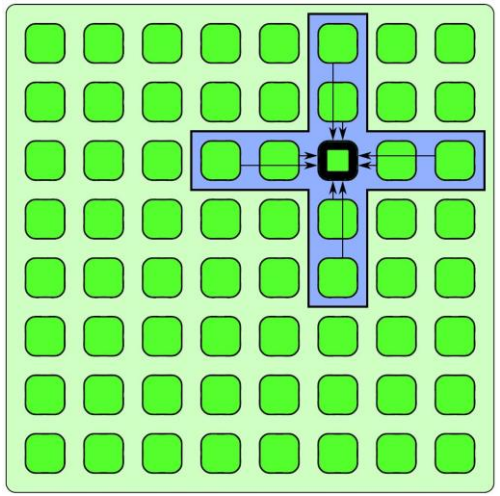
Parallel Control Patterns: Map

- **Map**: executes a function on all items in a collection
- Replicates a serial iteration where each iteration is independent of others. The number of iterations is known, and the computation only depends on the iteration and data from the collection.
- The replicated function is referred to as "elementary function”



Parallel Control Patterns: Stencil

- **Stencil**: Corresponds to a Generalization of Map. Elementary function that accesses a set of "neighbors"”
- Usually combined with iteration
- Edge conditions must be handled carefully.

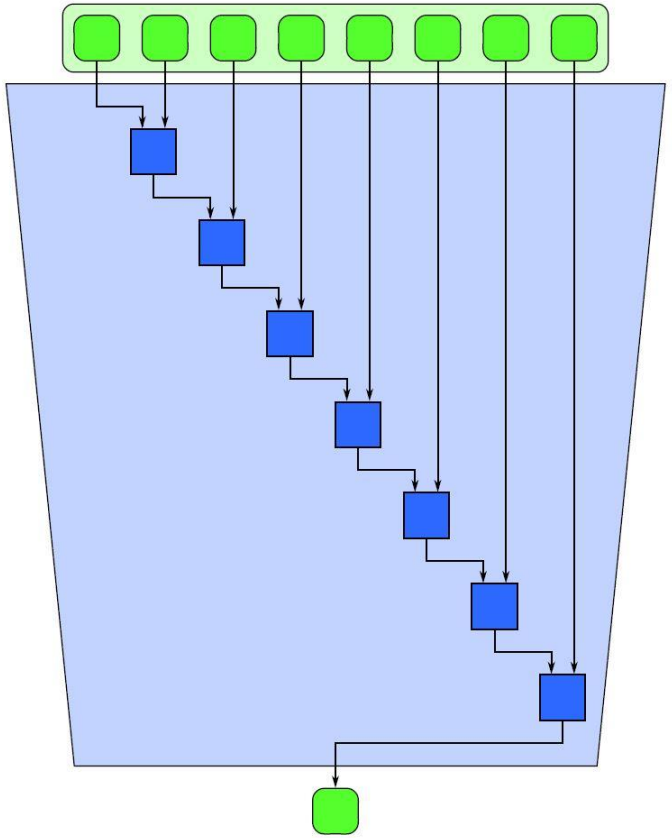


Parallel Control Patterns: Reduction

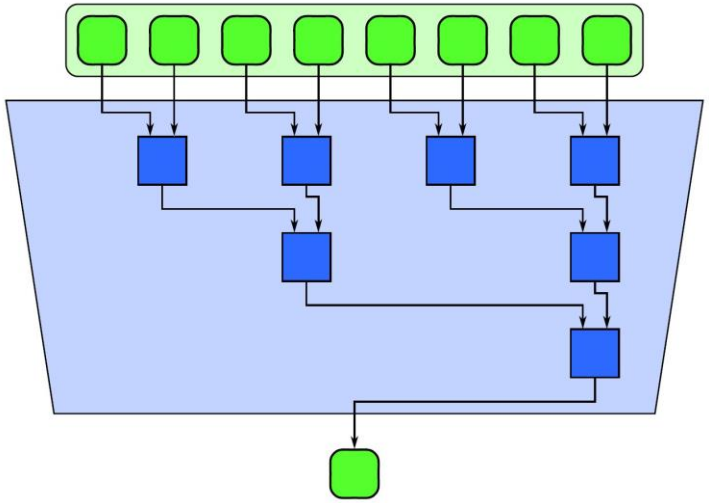
- **Reduction**: Combine each item into a collection using a "merge function"
- Different reduction orders are possible.
- Examples of merge functions: add, mul, max, min, AND, OR, and XOR

Parallel Control Patterns: Reduction

Serial Reduction



Parallel Reduction

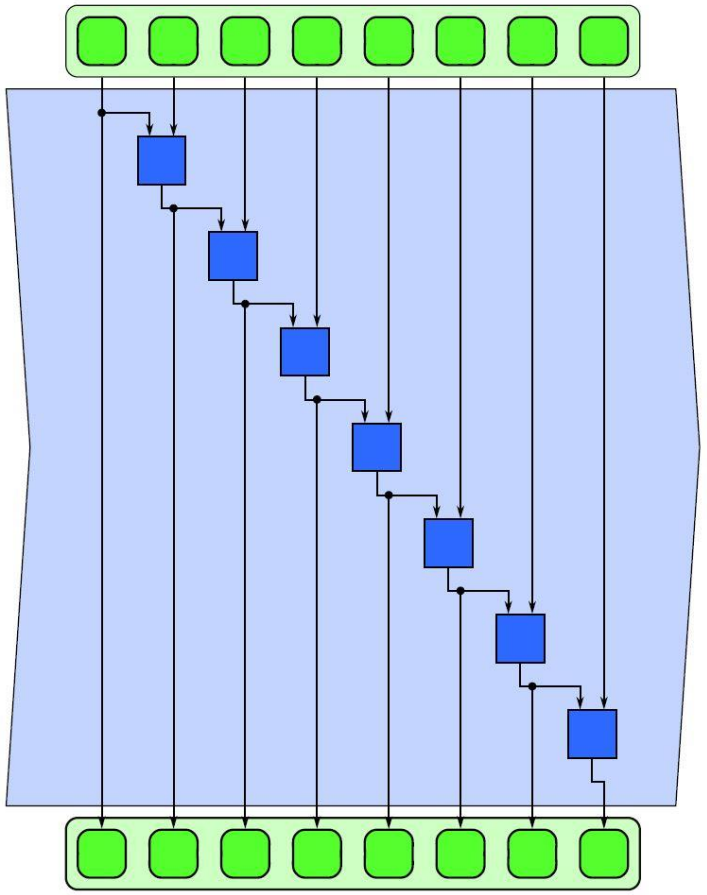


Parallel Control Patterns: Scan

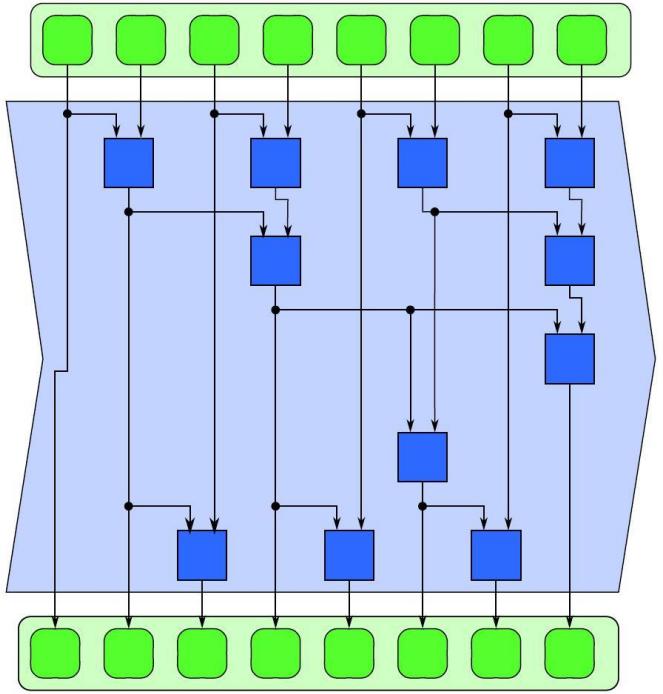
- **Scan**: computes partial reductions in a collection
- For each output in a collection, a reduction in input to that point is executed..
- If the function used is associative, the scan can be parallelized
- Parallelizing the scan is not trivial, as dependencies to previous iterations may exist in the loop.
- A parallel scan will require more operations than the serial version.

Parallel Control Patterns: Scan

Serial Scan



Parallel Scan



Parallel Control Patterns: Recurrence

- **Recurrence**: More complex version than Map, where loop iterations may depend on others
- Similar to Map, but elements can use outputs from adjacent elements as inputs
- For a recurrence to be executable, there must be a serial order of the recurrence of elements such that they can be computed using previously generated outputs..

Design patterns

1. Parallel Control Patterns
2. Data Management Patterns
3. Other Patterns

Parallel Data Patterns

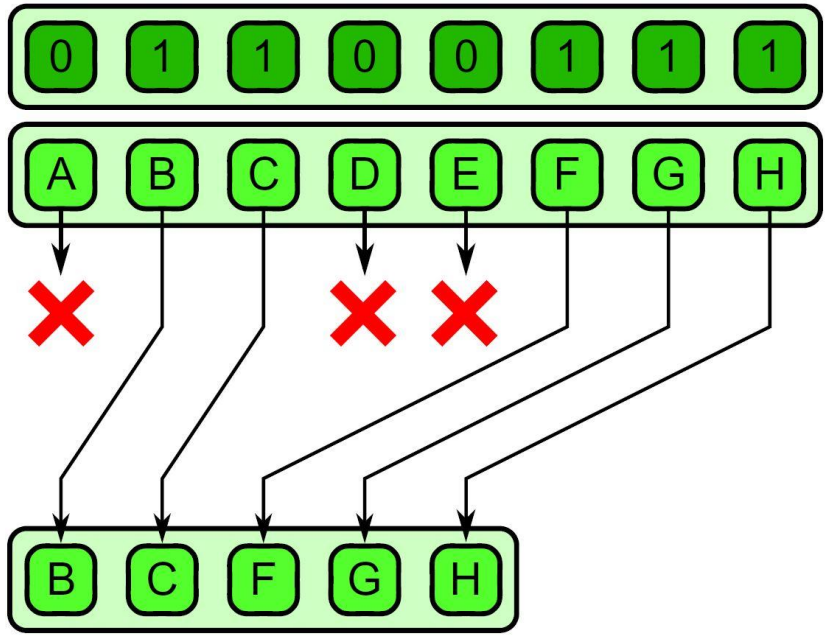
- To avoid problems such as data-race, it is important to know where the data is and if it is shared by multiple processes or threads.
- Some data management patterns help with data localization.
  - Data available when processes need it.
  - Avoid cache misses.
- Parallel data management patterns: **pack**, **pipeline**, **geometric decomposition**, **gather and scatter**

Pack

- **Pack** is used to delete unused space in a collection.
- Items marked false are discarded/deleted, and the remaining items located in a contiguous sequence in the same order.
- Useful when using Map
- **Unpack** is the reverse pattern and

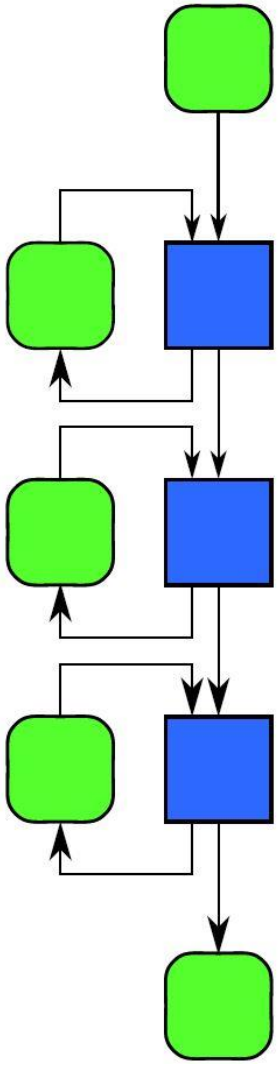
  is used to locate Return

  elements in their original positions



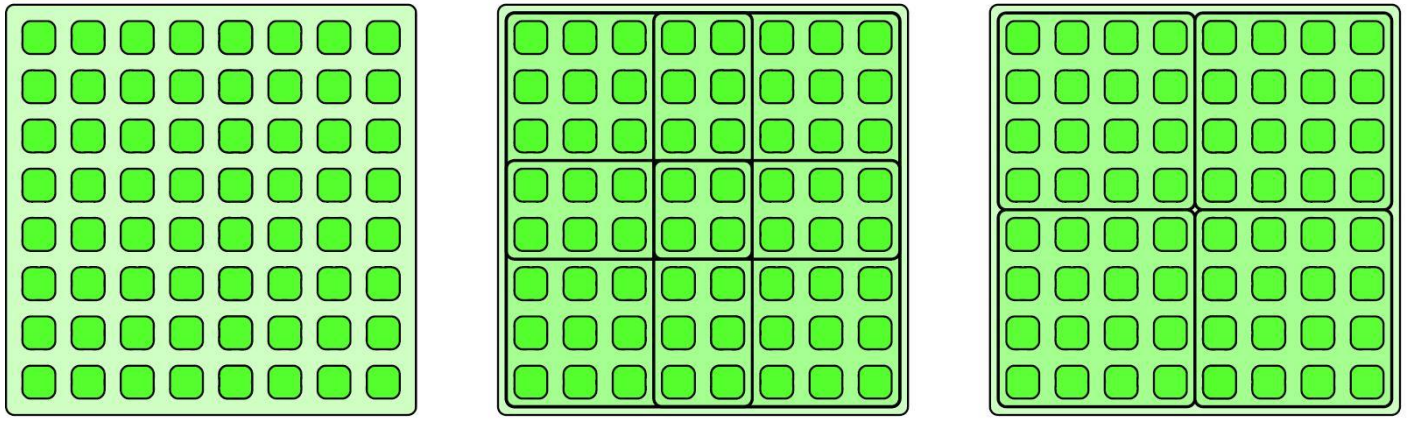
Pipeline

- **Pipeline** connects tasks in a producer-consumer form
- A linear pipeline is the basic idea of the pattern, however variations as in a
- DAG graph is also possible.
- Pipelines are useful when used with other patterns that get higher
- parallelism.



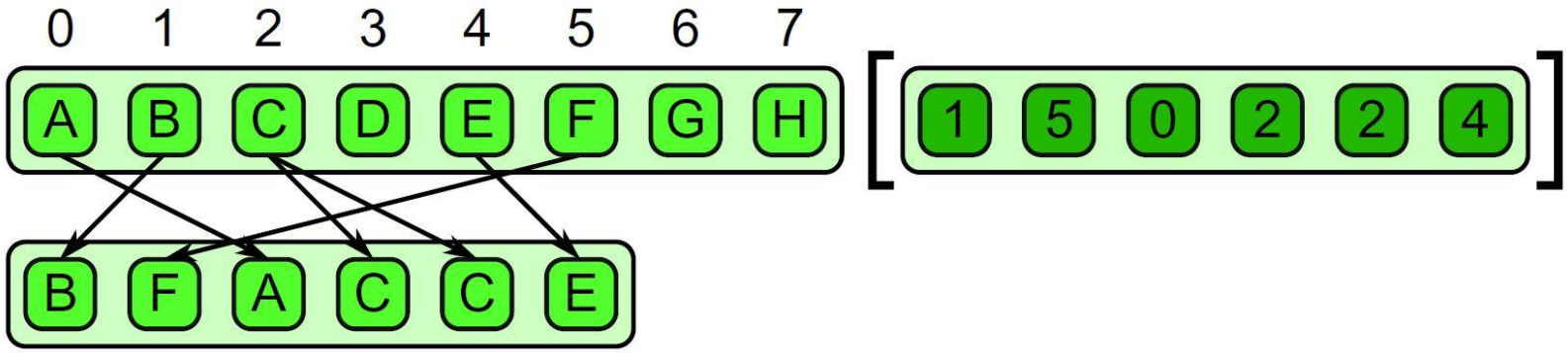
Geometric Decomposition

- **Geometric Decomposition**– organizes data into sub-collections.
- Decomposition with overlap and without overlap are possible.
- This pattern does not necessarily move data, it only gives us another view of this.



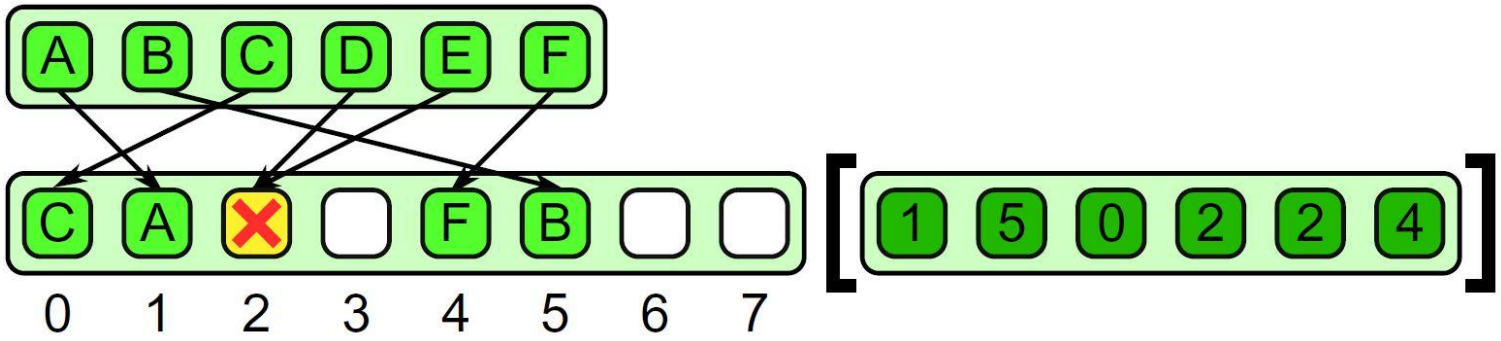
Gather

- **Gather** reads a collection of data given a collection of indexes.
- It can be imagined as a combination of map and random serial readings.
- the resulting collection shares the same type of the input collection but the same size as the index collection.



Scatter

- **Scatter** is the inverse of gather
- A dataset and an index set are required. Each element of the entry is written as a result in the index delivered for that position.
- Data-race conditions can occur when two elements are written to the same location.

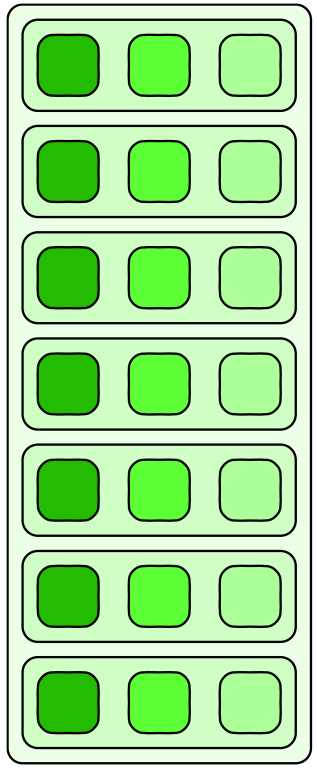


Design patterns

1. Parallel Control Patterns
2. Data Management Patterns
3. Other Patterns

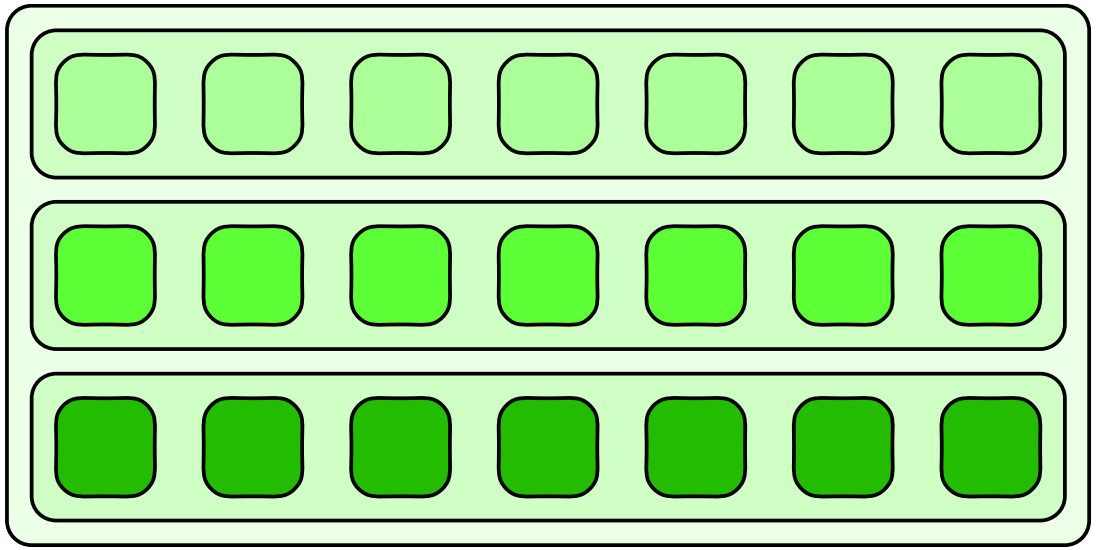
Other Parallel Patterns: AoS vs SoA

- **Array of Structures (AoS)**
  - Can deliver better cache utilization if data is accessed randomly.



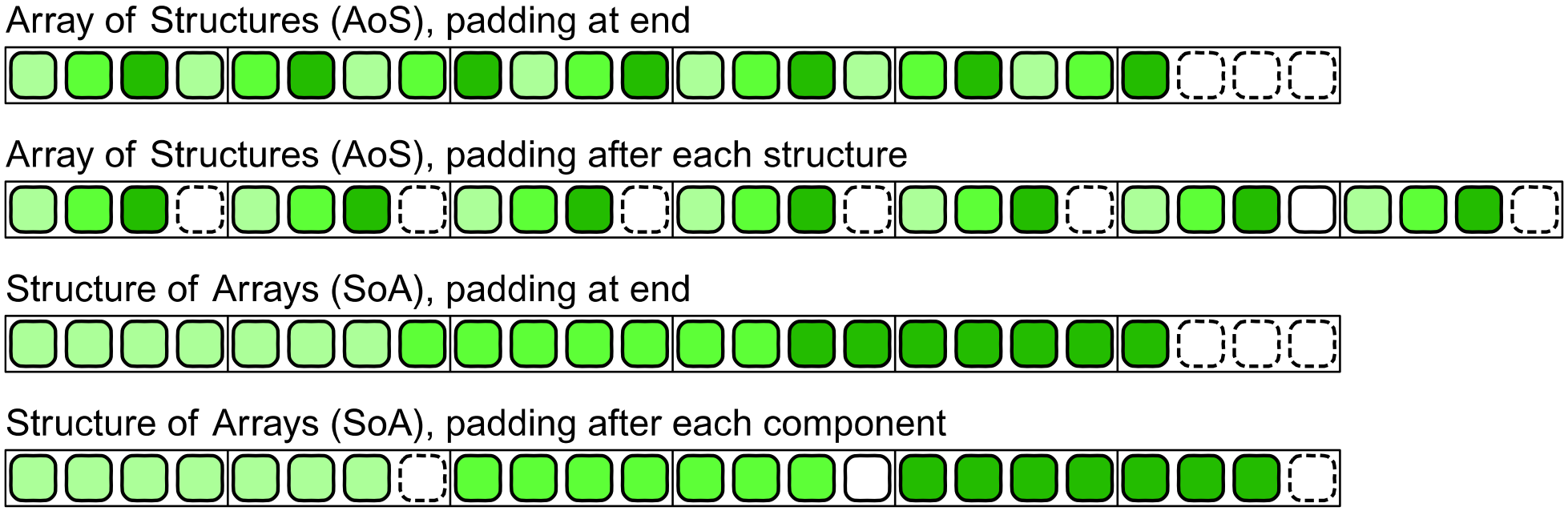
Other Parallel Patterns: AoS vs SoA

- **Structures of Arrays (SoA)**
  - Typically better for vectorization.



Other Parallel Patterns: AoS vs SoA

- Organization in memory:



Other Parallel Patterns: AoS vs SoA

AoS Code

```
struct node {
  float x, y, z;
};
struct node NODES[1024];

float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

SoA Code

```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

Other Parallel Patterns: AoS vs SoA

AoS Code

```
struct node {
  float x, y, z;
};
struct node NODES[1024];

float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

- More logical type of organization.
- Extremely difficult to access by gathers and scatters.
- Not very useful for vectorization.

Other Parallel Patterns: AoS vs SoA

- Separate arrangements for each field of the structure.
- Maintains contiguous memory access when vectorization.

SoA Code

```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

Summary

- **Parallel control patterns**
  - **fork-join**, **map**, **stencil**, **reduction**, **scan**, **recurrence**
- **Data management patterns**
  - **pack**, **pipeline**, **geometric decomposition**, **gather** and **scatter**
- **Other patterns**

Examples using patterns

- **Merge sort with reductions**
- Sort an array of integers using map and reduction.
- Idea: Map each element into a single-element vector and then apply the merge operation between vectors.
  - <> is the merge operation : [1,3,5,7] <> [2,6,15] = [1,2,3,5,6,7,15]
  - [] is the empty list.

Examples using patterns

Input: [14,3,4,8,7,52,1]

Mapped to [[14],[3],[4],[8],[7],[52],[1]]

Reduction from the right:

[14] <> ([3] <> ([4] <> ([8] <> ([7] <> ([52] <> [1])))))

= [14] <> ([3] <> ([4] <> ([8] <> ([7] <> [1,52]))))

= [14] <> ([3] <> ([4] <> ([8] <> [1,7,52])))

= [14] <> ([3] <> ([4] <> [1,7,8,52]))

= [14] <> ([3] <> [1,4,7,8,52])

= [14] <> [1,3,4,7,8,52]

= [1,3,4,7,8,14,52]

O(n) merge operations are performed, but each one takes O(n) = O(n^2)

Examples using patterns

Input: [14,3,4,8,7,52,1]

Mapped to [[14],[3],[4],[8],[7],[52],[1]]

Reduction as a tree:

(([14] <> [3]) <> ([4] <> [8])) <> (([7] <> [52]) <> [1])

= ([3,14] <> [4,8]) <> ([7,52] <> [1])

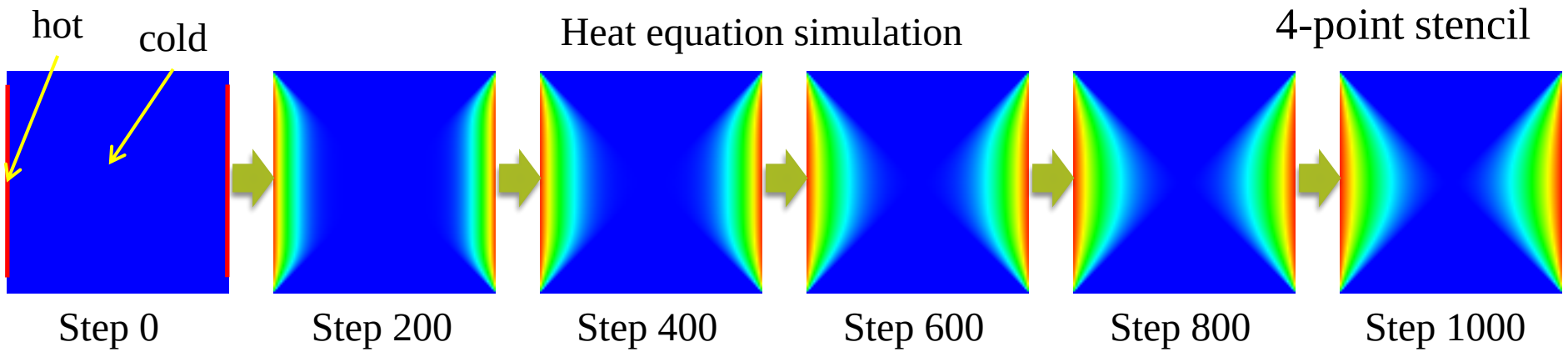
= [3,4,8,14] <> [1,7,52]

= [1,3,4,7,8,14,52]

O(logn) merge operations are performed, but each takes O(n) = O(nlogn)

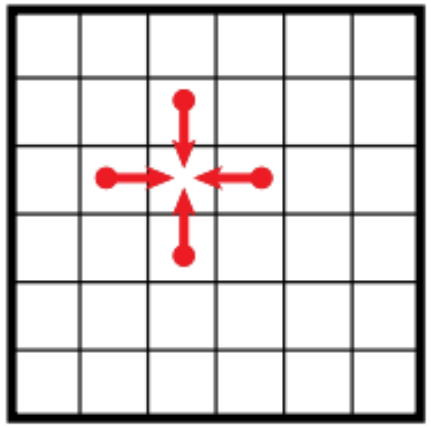
Examples using patterns

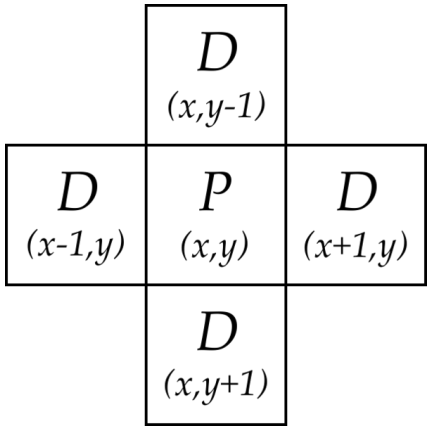
- **Heat propagation simulations with stencil**
- Calculate heat propagation on a metal plate using the Laplace equation and Jacobi iteration.

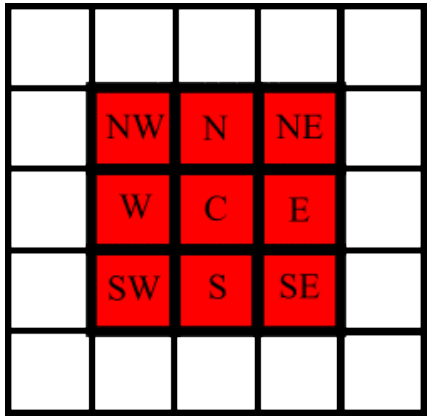


Examples using patterns

- **Heat propagation simulations with stencil**
- Calculate heat propagation on a metal plate using the Laplace equation and Jacobi iteration.
- the operation consists of calculating the average for all the cells on the surface and iterating until they converge.

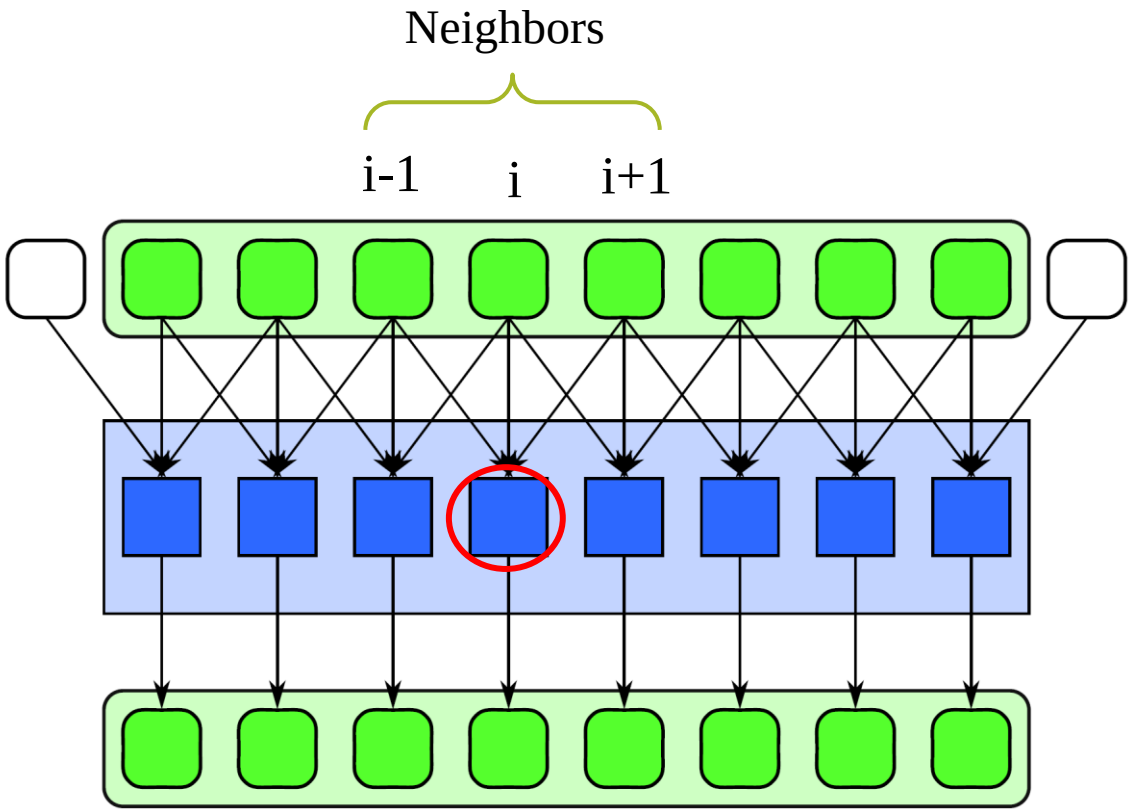






Examples using patterns

- **Heat propagation simulations with stencil**



References

- Parallel Computing Center. University of Oregon http://ipcc.cs.uoregon.edu/index.html