Contents

10 Patterns

10.1 Dependencies

Dependencies are critical when designing parallel programs as they directly influence the program's correctness and potential for parallelism.

Definition 1

A **dependency** arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed.

Non-directly, dependencies **determine the order in which operations must be executed to maintain correctness**.

Why is sequential consistency important?

Sequential Consistency ensures that the results of a parallel program's execution are as if the operations of all processors were executed in some sequential order, while preserving the program order for each processor. It is so important because:

- **Enforcing Dependencies**:
  - **Ordered Execution**. Sequential consistency ensures that operations are executed in an order that respects their dependencies.
  - **Data Integrity**. By maintaining the correct order of dependent operations, sequential consistency preserves the integrity of the data being manipulated in parallel programs.
- **Predictable Behavior**:
  - **Program Order Preservation**. Dependencies define the required order of operations within a program. Sequential consistency ensures each processor's operations appear in sequence, respecting these dependencies.
  - **Non-Interference**. Sequential consistency guarantees that operations from different processors can interleave their executions in many possible sequences, but the final results are consistent with some sequential execution that respects dependencies.

This type of model has already been discussed in Section 8.3 on page 226. However, we can summarize the guarantees it provides:

- ✔ **Simplified Reasoning**. Programmers can reason about parallel programs more easily, as if they were sequential.
- ✔ **Correctness**. It helps maintain the correctness of shared memory operations in concurrent programs.

How to detect dependencies?

By identifying where dependencies occur, we can determine where operations need to be synchronized or reordered to ensure accurate results. But the real dilemma is, "*how do we know when two or more statements are interdependent?*". The short answer is: it is not always that easy. It depends on the complexity of the code, but in general, the definitions are as follows:

- **Independent Statements**: two statements are independent if their **execution order does not affect the computation outcome**.

  The implication is that the order of execution is interchangeable without changing the final state.

- **Dependent Statements**: statements are dependent if the **order of their execution impacts the computation result**.

  The implication is that changing the order of execution alters the final outcome, so the sequence of operations matters.

Type of dependencies

Dependencies can be categorized into several types:

- **Control Dependencies**: execution of one operation depends on the control flow (like `if` statements or loops).
- **Data Dependencies**: one operation requires the result of another operation:
  - **True Dependency** (Read After Write, RAW): a subsequent operation reads data after it's written.
  - **Anti-Dependency** (Write After Read, WAR): a write operation must occur after all preceding read operations.
  - **Output Dependency** (Write After Write, WAW): two write operations must occur in a particular sequence.

Example 1: Dependencies

Some dependency scenarios:

- **Independent Statements**
  - Statement 1: `a=1;`
  - Statement 2: `b=1;`

  Since these two statements do not rely on each other's values, they are independent. Executing them in any order does not change the outcome.

- **True (flow) Dependency**
  - Statement 1: `a=1;`
  - Statement 2: `b=a;`

  `S2` depends on the value of `a` from `S1`. This is a true (or flow) dependency, as `S2` requires `S1` to execute first to produce the correct result.

- **Output Dependency**
  - Statement 1: `a=f(x);`
  - Statement 2: `a=b;`

  Both statements write to `a`, creating an output dependency. The order matters because both statements affect the value of `a`.

- **Anti-Dependency**
  - Statement 1: `a=b;`
  - Statement 2: `b=1;`

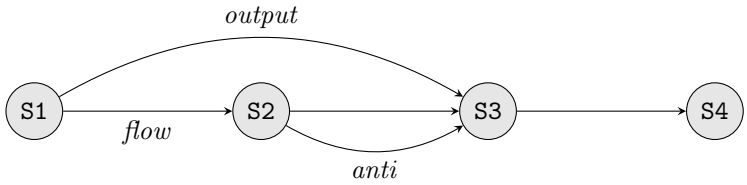
  There is an anti-dependency where `S1` reads `b`, and `S2` writes to `b`. The execution order affects the value read by `S1`.

Dependency Graph

Dependencies can be represented as a graph, also known as a **dependency graph**.

- **Nodes (statements)**. Each node represents a statement in the program. For example:
  - Statement 1: `a = 1;`
  - Statement 2: `b = a;`
  - Statement 3: `a = b + 1;`
  - Statement 4: `c = a;`

  The dependency graph is:



- **Edges (dependencies)**. Arrows between nodes indicate dependencies between statements. Here are the key types of dependencies shown (in the previous dependency graph):

– **True (flow) dependency**:

$$\texttt{S2} \quad \delta \quad \texttt{S3}$$

An arrow from `S2` to `S3` signifies that `S3` depends on `S2` for the value of `a`.

– **Output dependency**:

$$\texttt{S1} \quad \delta^0 \quad \texttt{S3}$$

An arrow from `S1` to `S3` indicates that both `S1` and `S3` write to `a`.

– **Anti dependency**:

$$\texttt{S2} \quad \delta^{-1} \quad \texttt{S3}$$

An arrow from `S2` to `S3` shows that `S2` reads `a`, while `S3` writes to `a`.

Why do we compute dependencies?

Computing data dependencies is important for several reasons, such as optimizing code (compilers can reorder or optimize code better if they know which statements depend on each other), avoiding data hazards (in hardware design, understanding dependencies is essential to avoid hazards), debugging and maintenance (understanding dependencies helps debugging by showing how data flows through the program), etc.

Therefore, by **computing data dependencies, we get a clear picture of how data is used and modified throughout the program**. This systematic approach can lead to more efficient, reliable, and maintainable code.

How do we compute dependencies?

Data dependency relationships can be found by comparing the IN and OUT sets of each node. The IN and OUT sets of a statement S are defined as:

- $IN\,(S)$: set of memory locations (**variables**) that may be **used** in S.
- $OUT\,(S)$: set of memory locations (**variables**) that may be **modified** by S.

In other words, $IN\,(S)$ and $OUT\,(S)$ sets represent **all possible memory locations** (variables) **that might be read from or written to by a statement** S.

Note that the IN and OUT **sets are often larger than the actual number of locations used or modified** in practice.

This conservativeness comes from a need to be cautious. When compiling or analyzing code, it is **better to consider more memory locations than to miss any, ensuring that dependencies are fully captured**. This way, no potential dependencies are overlooked, even if it means including some memory locations that may not be actively used in all scenarios.

Example 2: Simple analogy

Imagine we are listing all the books we might need for a project. We list every possible book we could *potentially* read, even if we end up only needing to read a few of them. This conservative approach ensures we won't miss a book we might actually need later.

Properties when we compute data dependencies

Assuming that there is a path from `S1` to `S2`, the following shows interesting properties about intersection between the IN and OUT sets:

- **Flow Dependency (True Dependency)**:

  $$\text{out}(\texttt{S1}) \cap \text{in}(\texttt{S2}) \neq \emptyset \qquad \texttt{S1}\ \delta\ \texttt{S2}$$

  Statement `S1` writes to a location that statement `S2` reads from later.

- **Anti Dependency**:

  $$\text{in}(\texttt{S1}) \cap \text{out}(\texttt{S2}) \neq \emptyset \qquad \texttt{S1}\ \delta^{-1}\ \texttt{S2}$$

  Statement `S1` reads from a location that statement `S2` writes to later.

- **Output Dependency**:

  $$\text{out}(\texttt{S1}) \cap \text{out}(\texttt{S2}) \neq \emptyset \qquad \texttt{S1}\ \delta^{0}\ \texttt{S2}$$

  Both `S1` and `S2` write to the same location.

Loop Dependencies

In a loop, we generally encounter two main types of dependencies:

- **Loop-Carried Dependencies**. These occur when an **iteration of a loop depends on data from a previous iteration**.

Example 3

In the following loop, each iteration depends on the result of the previous iteration, making it parallelizable only in a pipeline manner.

```
for (i=1; i<100; i++) {
    a[i] = a[i-1] + 1;
}
```

- **Loop-Independent Dependencies**: These exist if **iterations do not rely on results from each other directly**.

Example 4

In the following loop, each iteration can be executed independently, allowing for parallelization.

```
for (i=0; i<100; i++) {
    a[i] = i;
}
```

Furthermore, the dependency is:

- **Lexically Forward Dependency** occurs when the **source statement** (the one that produces the value) **appears *before* the target statement** (the one that consumes the value) within the same iteration of code. Essentially, it follows the natural top-to-bottom order of reading code.

Definition 2: Lexically Forward Dependency

Lexically Forward Dependency, when the source statement appears before the target statement in the loop body.

Example 5: Lexically Forward Dependency

In the following code, `a = b + 1` (the source) comes before `c = a + 2` (the target) within the same loop iteration.

```
for (i=0; i<5; i++) {
    a = b + 1;  // Source statement
    c = a + 2;  // Target statement
}
```

- **Lexically Backward Dependency** occurs when **the source statement comes *after* the target statement** within the same iteration. It's a bit counterintuitive since it involves looking back to an earlier point in the code to resolve a dependency.

Definition 3: Lexically Backward Dependency

Lexically Backward Dependency, when the source comes after the target.

Example 6: Lexically Backward Dependency

In the following code, `c = a + 2` (the target) relies on the value of `a` which is produced later by `a = b + 1` (the source) within the same iteration.

```
for (i=0; i<5; i++) {
    c = a + 2;  // Target statement
    a = b + 1;  // Source statement
}
```

Example 7: Lexical dependency analogy

Lexical dependencies are not an easy topic to understand. Here we offer a simple analogy for better understanding.

Think of a cooking recipe as a loop iteration. Imagine we have a recipe that lists steps to make a dish.

- **Lexically Forward Dependency**: This is like following a recipe where we need to boil water (*source step*) before we can add pasta to it (*target step*). The sequence flows naturally from top to bottom.
- **Lexically Backward Dependency**: Imagine we need to taste the pasta (*target step*) to decide if we should add more salt (*source step*), even though the step to add salt appears later in the list. This means we'd have to look back to the earlier part of the iteration to decide if we need to add more salt.

I really should understand lexical dependencies?

Understanding these dependencies helps when we're optimizing loops for parallel execution. **Lexically forward dependencies are generally simpler to manage since they align with the natural execution order**. Lexically backward dependencies require careful handling to ensure correct results, as they involve "looking back" within the iteration.

Example 8: Loop dependencies and optimization

Consider the following codes:

- First double for loop iteration:

```
for (int i = 0; i < 100; i++)
    for (int j = 1; j < 100; j++)
        a[i][j] = f(a[i][j - 1]);
```

- Second double for loop iteration:

```
for (int j = 1; j < 100; j++)
    for (int i = 0; i < 100; i++)
        a[i][j] = f(a[i][j - 1]);
```

Remark: Cache

When optimizing for cache performance, the goal is to maximize data locality. Data locality can be broken into two types:

1. **Temporal Locality**: Reusing the same data within a short period.
2. **Spatial Locality**: Accessing data locations that are near each other.

We can make the following considerations:

✔ In the **first loop**, the inner loop iterates over `j` while `i` remains constant. This means that within each iteration of the outer loop, we access consecutive elements in the same row (`a[i][...]`).

Since arrays are usually stored in row-major order (consecutive elements of a row are stored in consecutive memory locations), this loop has **good spatial locality**. Elements of `a[i][...]` are likely to be in the cache, reducing cache misses.

✘ In the **second loop**, the inner loop iterates over `i` while j remains constant. This means we're accessing elements in the same column across different rows (`a[...][j]`). Since rows are stored consecutively in memory, accessing elements across rows (column-wise access) leads to poor spatial locality. Each access to `a[i][j]` is more likely to cause a cache miss because it jumps to a different row.

Dependencies and Parallelism

- **Achieving Parallelism with Dependencies**
  - **Concept**: Even when our code has dependencies, we can still achieve parallelism. The key is to identify parts of the code that can run independently and execute those in parallel.
  - **Practice**: Look for independent computations that don't rely on each other's results and schedule them for parallel execution.
  - **Example**: If some parts of our loop iterations dno't depend on others, those can be parallelized while managing the dependent parts separately.
- **Control Ordering of Events on Different Processors (Cores)**
  - **Concept**: Dependencies dictate how different events in our program must be ordered, especially when running on multiple processors or cores.
  - **Practice**: Maintain a partial order of execution, which respects the dependencies and constraints imposed by the code.
  - **Example**: If one computation depends on the result of another, ensure that the dependent computation only starts once the necessary data is available.
- **Use Synchronization Mechanisms**
  - **Concept**: Running code concurrently requires synchronization to ensure that all dependencies and orderings are respected.
  - **Practice**: Use synchronization tools such as locks, semaphores, or barriers to control access to shared resources and manage dependencies.
  - **Example**: If multiple threads are updating a shared variable, synchronization mechanisms ensure that updates occur in the correct order, preventing race conditions.

10.2 Parallel Patterns

What are Parallel Patterns?

Parallel Patterns are essentially **recurring combinations of task distribution and data access strategies that can be applied to solve specific problems** in parallel computing. These patterns act as a "vocabulary" for algorithm design, making it easier to communicate and understand various approaches to parallelism.

By applying these patterns, developers can achieve efficient and effective parallelism, ensuring that tasks are executed concurrently without conflicts and with optimal performance.

- ✔ **Simplicity**. They provide clear, reusable templates for designing parallel algorithms.
- ✔ **Efficiency**. They help optimize performance by utilizing parallelism effectively.
- ✔ **Universality**. They are applicable across different parallel programming systems and platforms.

In essence, **parallel patterns help streamline the process of designing and implementing parallel algorithms**, making it easier to manage dependencies and achieve concurrent execution.

Roughly speaking, parallel patterns are **similar to design patterns** in computer science, but they focus specifically on parallel computing. They provide reusable solutions to common problems in parallel programming, with the goal of simplifying and streamlining the development process.

Types of Parallel Patterns

1. **Nesting Pattern**: Combining different parallel patterns within one another.
2. **Serial/Parallel Control Patterns**: Handling control flow in parallel and serial region of code.
3. **Serial/Parallel Data Management Patterns**: Managing data access and modifications in parallel and serial contexts.
4. Other Patterns:
   - Superscalar Sequences
   - Futures
   - Speculative Selection
   - Workpile
   - Search
   - Segmentation
   - Expand
   - Category Reduction
5. **Programming Model Support for Patterns**: Tools and models that support implementing these patterns.

10.2.1 Nesting Pattern

The **Nesting Pattern** is a compositional pattern[13] that **allows for the hierarchical composition of patterns**. This is used in both serial and parallel algorithms. The main idea is to **use "pattern diagrams"[14] to visually show how tasks can be organized and composed in a hierarchy**.

Key Points

- **Hierarchical Composition**: The Nesting Pattern **allows patterns to be nested within each other**, creating a hierarchy of patterns.
- **Task Blocks**: Each **"task block" in a pattern diagram represents a location of general code in an algorithm**.
- **Replaceability**: **Any task block in the diagram can be replaced with a pattern** that has the same input/output and dependencies.

In the following figure, we can see a diagram showing a multiple task blocks. Some of these blocks are highlighted in green to indicate that they can be replaced with other patterns. The goal is to illustrate how we can nest patterns to build more complex algorithms.

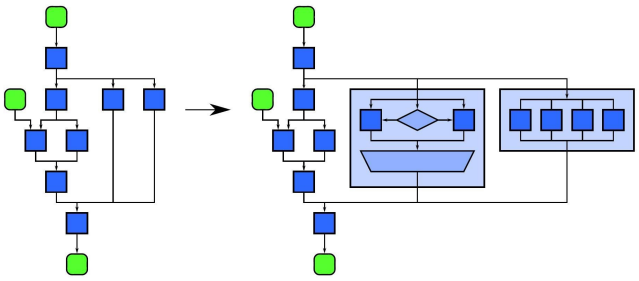


Figure 39: Graphical example of a nesting pattern.

[13] **Compositional Patterns** are **design patterns that allow us to compose complex systems from simpler, reusable components**. These patterns emphasize the importance of combining smaller pieces of functionality in a structured and predictable way. They are particularly useful in managing the complexity of large systems by breaking them down into manageable parts.

[14] **Pattern Diagrams** are **visual tools used to represent and illustrate patterns**, particularly in the context of software engineering and algorithm design. They help in understanding how different components or tasks fit together to form a complete system or process.

Example 9: Pattern Diagrams

Suppose we are designing a parallel algorithm to process large data sets. A pattern diagram can help us visualize the composition of different patterns, such as data partitioning, task scheduling, and data aggregation. Each of these patterns can be represented as a set of nodes and edges nested within a larger pattern that represents the entire algorithm.

10.2.2 Serial Control Patterns

Serial Control Patterns are **fundamental programming constructs that dictate the sequence of execution in a program**. They are the basic building blocks of any algorithm, defining how the flow of control moves through the program from start to finish. In other words, they determine the order in which tasks are performed.

Why "serial" control patterns?

These patterns are called "serial" because the **tasks** they control are **executed one after the other**, in a specific order, without overlapping.

Types

1. **Sequence Pattern**

   Definition. An ordered list of tasks that are executed in a specific order.

   Assumption. The program text ordering will be followed.

   Example 10: Sequence Pattern

   In the following example, each task must be completed before the next one begins.

   ```
   int A = 10;
   int T = foo(A);
   int S = bar(T);
   int B = delta(S);
   ```

2. **Selection Pattern**

   Definition. A condition is evaluated first; depending on the result, one of two tasks is executed.

   Assumption. Only one of the tasks will be executed, not both.

   Example 11: Selection Pattern

   In the following example, the function `a` is executed if condition `c` is true; otherwise, function `b` is executed.

   ```
   if (c) {
       a();
   } else {
       b();
   }
   ```

3. **Iteration Pattern**

   Definition. A condition is evaluated, and if true, a task is executed repeatedly until the condition becomes false.

   Complication in Parallelizing. Dependencies may exist between iterations.

   Example 12: Iteration Pattern

   In the following example, function `a` is executed `n` times. The condition `i < n` is evaluated, and if true, the function `a` is repeated until `i < n` becomes false (then `i = n`).

```
for (int i = 0; i < n; ++i) {
    a();
}
```

4. **Recursion Pattern**

   Definition. A dynamic form of nesting where functions call themselves.

   Special Case. Tail recursion can be converted into iteration, which is important for functional languages.

   Example 13: Recursion Pattern

```
int recursive_function(int n) {
    if n > 0
        return n * recursive_function(n - 1);
    return 1; // base case
}
```

These patterns are the **foundation of structured serial programming**. Also, when parallelizing serial algorithm, it's crucial to understand these patterns to effectively manage dependencies and concurrency.

10.2.3 Parallel Control Patterns

Parallel Control Patterns are **design patterns used in parallel computing to manage the execution flow of tasks that can run simultaneously**. These patterns help in organizing and structuring parallel code to ensure efficient execution, resource management, and scalability. They are essential for leveraging the power of multi-core processors and distributed computing systems.

In essence, parallel control patterns provide the blueprint for designing and implementing parallel algorithms, ensuring that tasks are executed efficiently and effectively.

Types

1. **Fork-Join Parallel Pattern**

   Definition. The Fork-Join pattern allows control flow to fork into multiple parallel flows and then rejoin later.

   Implementation. Languages like Cilk Plus use `spawn` and `sync` to implement this pattern. When a function is spawned, it runs in parallel with the caller. The caller then uses `sync` to wait for all spawned functions to complete before proceeding.

   Difference from Barrier. A "join" involves only one thread continuing, while a "barrier" allows all threads to continue.

2. **Map Parallel Pattern**

   Definition. The Map pattern applies a function to each element of a collection independently, producing a new collection of results.

   Usage. It replaces serial iterations where each iteration is independent.

   Elemental Function. The function applied to each element is called an "elemental function".

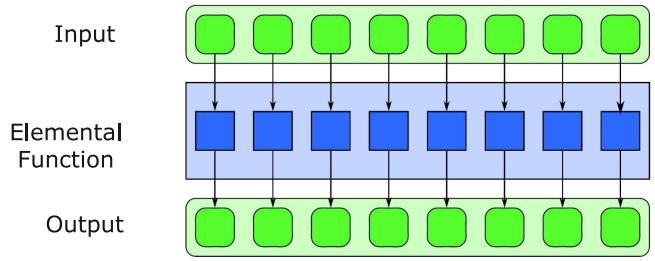


Figure 40: Visual representation of the Map Parallel Pattern.

Example 14: Map analogy

A good analogy is the `map` method used in Javascript. The difference is that the `map` in Javascript is done sequentially, instead the map pattern is done in parallel. However, as we can see from the example provided in the official documentation, a function is applied to each element of the input:

```
// Elemental Function
function elementalFunction(num) {
  return num * 2;
}

// Example array
const array1 = [1, 4, 9, 16];

// Pass an elementalFunction to map
const map1 = array1.map((x) => elementalFunction(x));

console.log(map1);
// Expected output: Array [2, 8, 18, 32]
```

Here the official documentation:

3. **Stencil Parallel Pattern**

   Definition. The Stencil pattern extends the Map pattern by allowing the elemental function to access neighboring elements.

   Usage. Commonly used in iterative solvers or simulations where the state of an element depends on its neighbors.

   Consideration. Boundary conditions must be handled carefully.

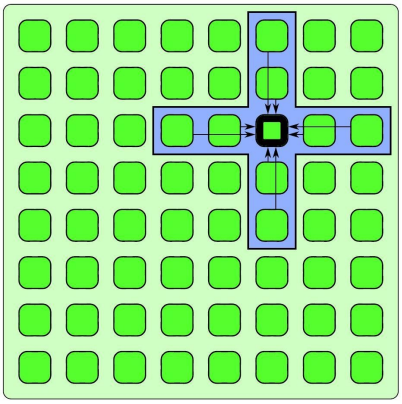


Figure 41: Visual representation of the Stencil Parallel Pattern.

4. **Reduction Parallel Pattern**

Definition. The Reduction pattern combines elements of a collection using an associative "combiner function" (e.g., addition, multiplication, max, min).

Parallelism. Different orderings of the reduction are possible due to the associative property, making it suitable for parallel execution.

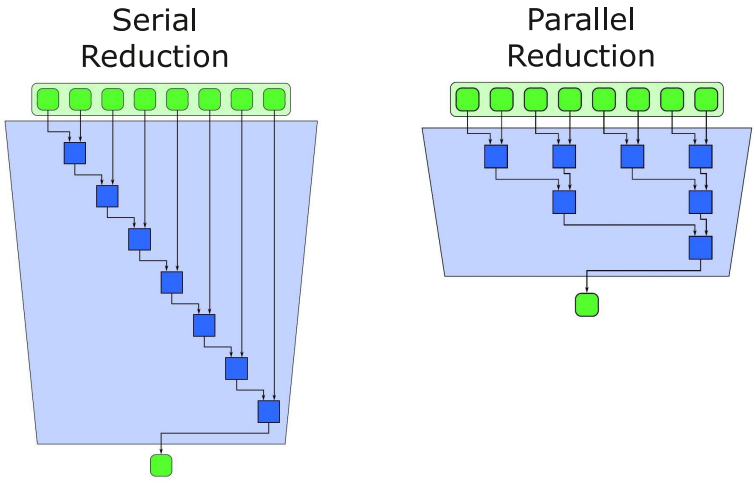


Figure 42: Visual representation of the Reduction Parallel Pattern.

Example 15: Reduction analogy

A good analogy is the `reduce` method used in Javascript. The difference is that the `reduce` in Javascript is done sequentially, instead the reduce pattern is done in parallel.

```
function combinerFun(acc, curVal) {
    return acc + curVal;
}

// Example array
const arrayToReduce = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
// steps:
// 0: 0 + 1
// 1: 1 + 2
// 2: 3 + 3
// 3: 6 + 4
// result = 10
const initialValue = 0;
const sumOfArray = arrayToReduce.reduce(
  (acc, curVal) => combinerFun(acc, curVal),
  initialValue
);

console.log(sumOfArray);
// Expected output: 10
```

Here the official documentation:

5. **Scan Parallel Pattern**

   Definition. The Scan pattern computes all partial reductions of a collection.

   Usage. For each element in the output collection, it computes a reduction of the input elements up to that point.

   Parallelization. Although it has dependencies, the associative nature of the function allows parallelization.

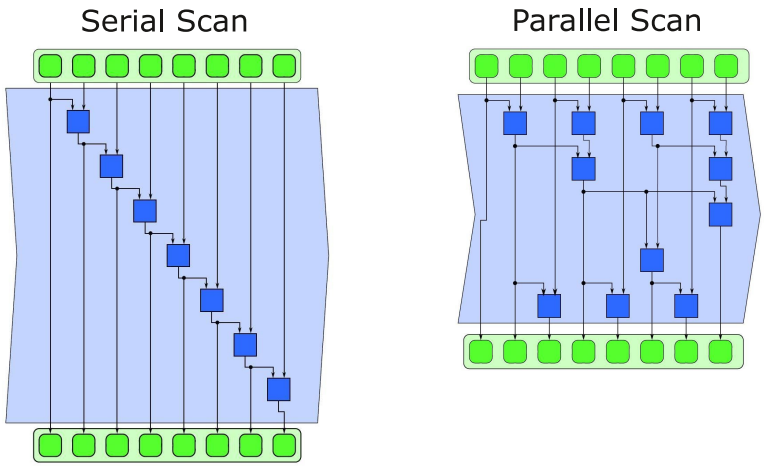


Figure 43: Visual representation of the Scan Parallel Pattern.

Example 16: Scan analogy

```
// JavaScript doesn't provide a native implementation of the scan,
// but we can build it manually
// * array - input
// * fn - reduction to apply to input values
// * initialValue - initial value of the accumulator
function scan(array, fn, initialValue) {
    // create the output array
    const result = [];
    // The scan method is a partial reduction,
    // so we need to apply the partial reduce
    array.reduce((acc, value, index) => {
        // combiner function to apply
        const newAcc = fn(acc, value);
        // save the calculated value
```

```
        // at the i-th iteration
        result[index] = newAcc;
        // returns the calculated value,
        // since it will be the new accumulator
        return newAcc;
    }, initialValue);
    // return the output
    return result;
}

// combiner function of the reduction pattern
function combinerFunc(accumulator, currentValue) {
    return accumulator + currentValue;
}

// Example array
const arr = [1, 2, 3, 4];
// Scan
const sumRes = scan(
    arr, (a, b) => combinerFunc(a, b), 0
);

console.log(sumRes);
// Expected output: [1, 3, 6, 10]
```

To test the execution, here there is a free console (or just open the browser console):

6. **Recurrence Parallel Pattern**

   Definition. The Recurrence pattern is a more complex version of Map, where elements can depend on the outputs of adjacent elements.

   Usage. Similar to Map, but requires a serial ordering of elements for computability.

10.2.4 Serial Data Management Patterns

Serial Data Management Patterns refer to the various **methods and strategies used by serial programs to manage data**. These patterns deal with how data is allocated, shared, read, written, and copied in a program. Understanding these patterns is crucial for writing efficient and maintainable code.

Types

1. **Random Read and Write Pattern**

   Definition. **Memory locations are accessed using addresses**, often through pointers. This allows for random access to data, meaning any memory location can be read from or written to at any time.

   Challenges. **Aliasing** (uncertainty about whether two pointers refer to the same memory location) can cause problems when converting serial code to parallel code.

Example 17: Random Read and Write Pattern

```cpp
#include <iostream>

using namespace std;

int main() {
    int array[10];

    // Write to random positions
    array[3] = 5;
    array[7] = 10;

    // Read from random positions
    cout << "Value at position 3: " << array[3] << endl;
    cout << "Value at position 7: " << array[7] << endl;

    return 0;
}
```

2. **Stack Allocation Pattern**

   Definition. **Stack allocation is used for dynamically allocating data** in a Last-In-First-Out (LIFO) manner. This is very efficient because allocation and deallocation can be done in constant time.

   Benefits. **Preserves locality, making data access faster**. In parallel programming, each thread usually has its own stack, preserving thread locality.

Example 18: Stack Allocation Pattern

```cpp
#include <iostream>

using namespace std;

void functionB() {
    int b = 10; // Stack allocation
    cout << "Function B, value of b: " << b << endl;
}

void functionA() {
    int a = 5; // Stack allocation
    cout << "Function A, value of a: " << a << endl;
    functionB();
}

int main() {
    functionA();
    return 0;
}
```

3. **Heap Allocation Pattern**

   Definition. **Used when data cannot be allocated in a LIFO manner**. Heap allocation allows for dynamic memory allocation at any time.

   Challenges. **Slower and more complex than stack allocation**. In parallel programming, a parallelized heap allocator should be used to keep separate pools for each parallel worker.

Example 19

```cpp
#include <iostream>

using namespace std;

int main() {
    int* ptr = new int; // Heap allocation
    *ptr = 10;
    cout << "Value in heap: " << *ptr << endl;
    delete ptr; // Free the allocated memory
    return 0;
}
```

4. Objects Pattern

Definition. **Objects are language constructs that associate data with code to manipulate and manage that data**. They can have member functions and belong to a class of objects.

Usage. In parallel programming, objects are often generalized in various ways to support concurrent execution.

Example 20

```
#include <iostream>

using namespace std;

class MyObject {
public:
    int value;

    void display() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    MyObject obj; // Stack allocation of an object
    obj.value = 15;
    obj.display();

    MyObject* objPtr = new MyObject(); // Heap
    allocation of an object
    objPtr->value = 20;
    objPtr->display();
    delete objPtr; // Free the allocated memory

    return 0;
}
```

10.2.5 Parallel Data Management Patterns

Parallel Data Management Patterns are **strategies used to manage data in parallel computing environments**. These patterns ensure that data is properly managed, avoiding issues like race conditions and ensuring efficient data locality. They are crucial for optimizing the performance of parallel programs by managing how data is allocated, shared, accessed, and manipulated across multiple parallel workers.

Types

1. **Pack Parallel Pattern**

   Description. Pack is used to **eliminate unused space in a collection by discarding elements marked as false and placing the remaining elements in a contiguous sequence**.

   Usage. Commonly used with the Map pattern.

   Inverse Operation. Unpack is the inverse operation, placing elements back in their original locations.

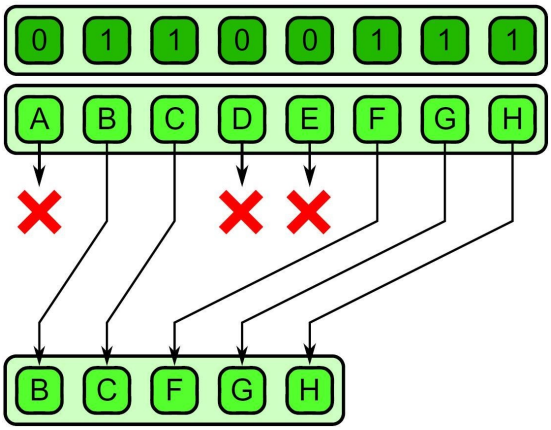


Figure 44: Visual representation of the Pack Parallel Pattern.

2. **Pipeline Parallel Pattern**

   Description. **Pipeline connects tasks in a producer consumer manner**. Tasks are organized in a linear sequence where the output of one task becomes the input of the next.

   Usage. Useful when combined with other patterns to multiply available parallelism. Pipelines can also be structured as Directed Acyclic Graphs (DAGs).

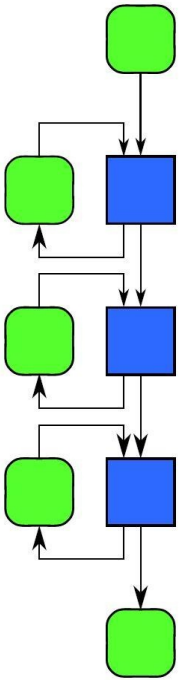


Figure 45: Visual representation of the Pipeline Parallel Pattern.

3. **Geometric Decomposition Parallel Pattern**

   Description. **Geometric Decomposition arranges data into subcollections**, which can be overlapping or non-overlapping.

   Usage. This pattern provides a different view of the data without necessarily moving it. It's commonly used in grid-based computations like finite element analysis or image processing.

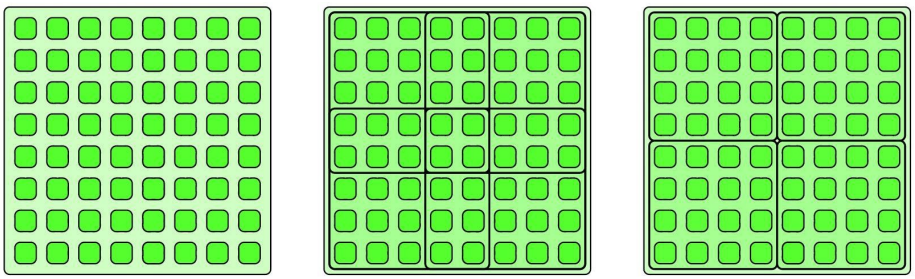


Figure 46: Visual representation of the Geometric Decomposition Parallel Pattern.

4. **Gather Parallel Pattern**

   Description. **Gather reads a collection of data given a collection of indices**, similar to a combination of Map and random serial reads.

   Usage. The output collection shares the same type as the input collection but has the same shape as the indices collection.

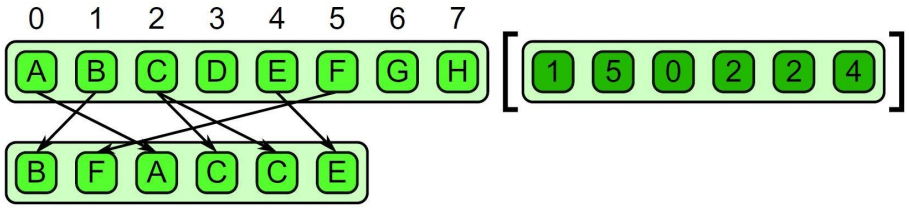


Figure 47: Visual representation of the Gather Parallel Pattern.

5. **Scatter Parallel Pattern**

   Description. Scatter is the inverse of Gather. It **writes each element of the input to the output at the given index specified by a collection of indices**.

   Usage. Race conditions can occur when multiple writes target the same location, so care must be taken to avoid conflicts.

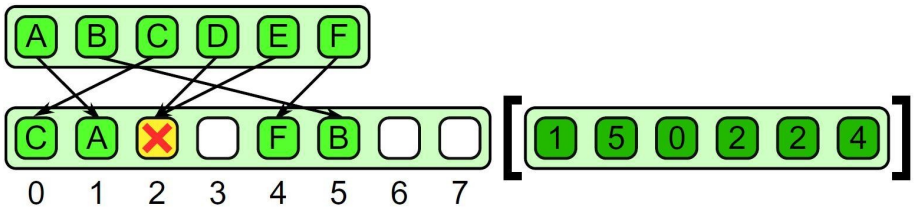


Figure 48: Visual representation of the Scatter Parallel Pattern.

10.2.6 Other Parallel Patterns

There are other important parallel patterns that we don't introduce. The following lists some of them with a brief description.

- **Superscalar Sequences Parallel Pattern**
  - **Description**. Write a sequence of tasks, ordered only by dependencies.
  - **Usage**. Helps in organizing tasks such that they can be executed out of order as long as dependencies are respected, increasing parallelism.
- **Futures Parallel Pattern**
  - **Description**. Similar to fork-join, but tasks do not need to be nested hierarchically.
  - **Usage**. Allows asynchronous execution of tasks, where a "future" represents a value that will be computed and available in the future (a similar logic exists in Python: official documentation).
- **Speculative Selection Parallel Pattern**
  - **Description**. A general version of serial selection where the condition and both outcomes can all run in parallel.
  - **Usage**. Used when it's beneficial to execute multiple branches in parallel and discard the results of the non-selected branch once the condition is evaluated.
- **Workpile Parallel Pattern**
  - **Description**. A general map pattern where each instance of the elemental function can generate more instances, adding to the "pile" of work.
  - **Usage**. Efficient for dynamic task generation, where new tasks are created as the algorithm progresses.
- **Search Parallel Pattern**
  - **Description**. Finds some data in a collection that meets specific criteria.
  - **Usage**. Utilized in searching algorithms where multiple search operations can run in parallel to speed up the process.
- **Segmentation Parallel Pattern**
  - **Description**. Operations on subdivided, non-overlapping, non-uniformly sized partitions of 1D collections.
  - **Usage**. Effective for processing large datasets by dividing them into manageable segments that can be processed in parallel.

- **Expand Parallel Pattern**

  Description. A combination of pack and map patterns.

  Usage. Allows elements to be expanded and processed in parallel, useful for operations that generate variable amounts of output from each input element.

- **Category Reduction Parallel Pattern**

  Description. Given a collection of elements each with a label, find all elements with the same label and reduce them.

  Usage. Common in data aggregation tasks where elements are grouped by a key (label) and then reduced (aggregated) by some operation.

10.3 Map Pattern

10.3.1 What is a Map?

The **Map Pattern** involves **applying a function to each element in a collection independently (without knowledge of neighbors), producing a new collection of results**. This pattern is highly parallelizable because each element is processed independently of the others.

Key Points

- **Independence**. **Each iteration of the map operation is independent of the others**, making it highly parallelizable.

  More precisely, assuming an infinite number of processors ∞ available, the operation takes constant time $O(1)$, but the overhead for managing these parallel tasks grows logarithmically with input size $O(\log n)$; however, even with the implementation overhead, the time complexity remains very efficient and grows slowly with input size.

  Modifying shared state breaks parallelism independence, and the violation can cause some problems: non-determinism, data races, undefined behavior, segmentation faults.

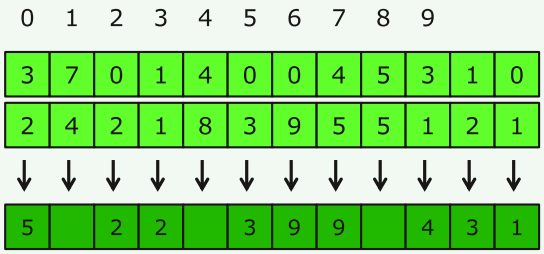
- **Unary Maps**. Maps can **operate over a single collection** (one input, one output).

Example 21: Unary Map

```
// Unary Map takes a single value
// (a single position of the input array)
// and returns a single value
// (the new value of the output array)
function elementalFunction(val) {
    return val * 2;
}

// input
const inArr = [1, 2, 3, 4];

// apply unary map
const outArr = inArr.map((inVal) => elementalFunction(
    inVal));

// expected output: [2, 4, 6, 8]
console.log(outArr);
```

- **N-ary Maps**. Maps can also **operate with multiple inputs and outputs**, such as performing element-wise operations on two arrays.

Example 22: N-ary Map




```
int twoToOne(int x, int y) {
    return x + y;
}
```

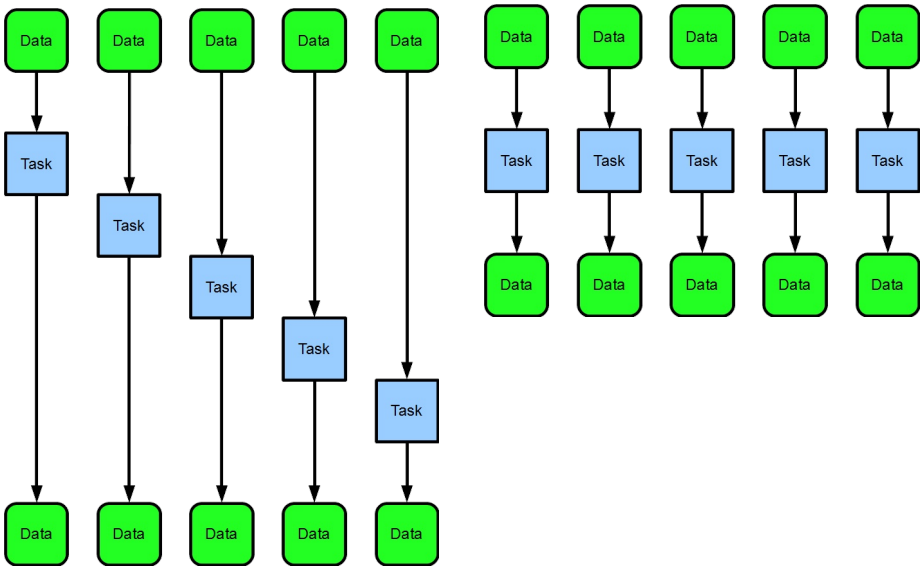


Figure 49: A sequential Map on the left and a parallel Map on the right.

Advantages

- **Easy to Parallelize**. Since each element is processed independently, map operations can be easily parallelized to improve performance.
- **Local State**. Each element has its own input and output, avoiding shared state issues.

10.3.2 Optimizations

10.3.2.1 Sequences of Maps

Sequences of Maps refer to a **series of map operations that are applied one after another in a pipeline**. Each operation in the sequence takes the output of the previous operation as its input. This pattern is common in tasks involving vector math, where multiple small operations like additions and multiplications are applied in sequence to elements of an array or collection.

Key Points

- **Independent Operations**. **Each map operation is independent and can be parallelized**. However, when several maps are applied in sequence, each intermediate result might be written to memory.
- **Vector Math**. Often, tasks in vector math consist of multiple small operations applied as maps, such as `a[i] = f(a[i])` followed by `a[i] = g(a[i])`, and so on.

✖ **Memory Inefficiency**. A naïve implementation that writes each intermediate result to memory can **waste memory bandwidth** and **potentially overwhelm the cache**.

How to Improve Performance? In the following pages, we introduce two techniques, code fusion and cache fusion, that can be used to reduce memory bandwidth usage and improve cache efficiency.

Example 23: Sequences of Maps

Image we have two map operations to apply to an array: one to double each element and another to add 3 to each element.

```
for (int i = 0; i < n; ++i) {
    a[i] = a[i] * 2;
}

for (int i = 0; i < n; ++i) {
    a[i] = a[i] + 3;
}
```

In this example, each element of the array `a` is first doubled, and then `3` is added to each element.

10.3.2.2 Code Fusion

Code Fusion is an **optimization technique** used to improve the performance of programs, particularly in parallel computing. It involves **combining multiple operations into a single**, more efficient operation. By doing so, code fusion increases arithmetic intensity, reduces memory/cache usage, and enhances overall computational efficiency.

Key Points

- **Combining Operations**. Code fusion ***fuses* together multiple operations so that they are performed simultaneously**, rather than sequentially.

  Example 24: Code Fusion

  Instead of performing a series of separate operations on an array, we combine them into one loop, reducing the number of passes over the data.

- **Increasing Arithmetic Intensity**. Arithmetic intensity refers to the **ratio of computational operations to memory operations**. By fusing operations, we increase this ratio, meaning **more calculations are done per memory access**.

  Benefit. Higher arithmetic intensity leads to better utilization of the CPU and other computational resources.

- **Reducing Memory/Cache Usage**. By reducing the number of intermediate results that need to be stored in memory, code fusion **minimizes memory access and optimizes cache usage**.

- **Use of Registers**. Ideally, **operations can be performed using registers alone**, which are the fastest form of storage in a CPU. By keeping data in registers and reducing memory access, code fusion achieves maximum computational efficiency.

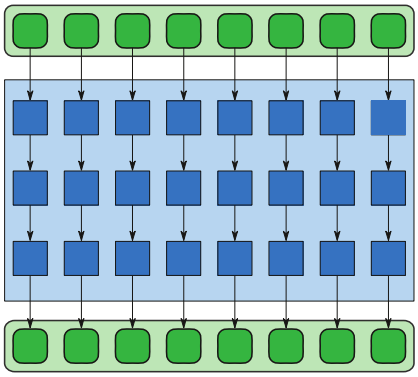


Figure 50: Graphical example of a code fusion.

10.3.2.3 Cache Fusion

Cache Fusion is an optimization **technique** that improves the efficiency of memory access by **ensuring that data is accessed and processed in a way that takes full advantage of the CPU cache**. This is particularly important in parallel computing, where efficient use of cache can significantly enhance performance.

Key Points

- **Optimizing Cache Usage**. Cache fusion involves **structuring code and data access patterns to maximize the cache's effectiveness**. This means minimizing cache misses and ensuring that data stays in the cache as long as needed.
- **Reducing Memory Bandwidth Usage**. By optimizing how data is accessed, cache fusion **reduces the number of memory accesses needed**, thus **decreasing memory bandwidth usage**.

How does it work?

1. **Breaking Operations into Smaller Blocs**. Instead of processing large blocks of data all at once, **cache fusion breaks these operations into smaller blocks that fit into the cache**.

   This ensures that data accessed remains in the cache, reducing the need to repeatedly access main memory.

2. **Efficient Data Movement**. The **smaller blocks are processed one at a time**, with each block being loaded into the cache, processed, and then written back to memory.

   This minimizes the movement of data between the cache and main memory, improving overall performance.

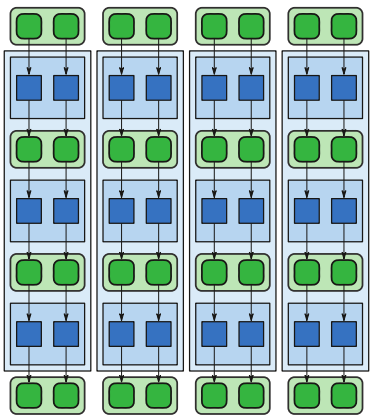


Figure 51: Graphical example of a cache fusion.

10.3.3 Related Patterns

Related Patterns are **design patterns that share similarities in structure, function, or purpose**. They often **address similar problems or solutions in different ways**, complementing each other within a broader context of software or algorithm design. Understanding related patterns helps in selecting the most appropriate pattern for a specific task and recognizing how different patterns can be combined or interchanged to achieve optimal results.

Patterns related to Map

- **Stencil Pattern related to Map Pattern**

  Description. Each instance of the map function accesses neighbors of its input, offset from its usual input.
  In other words, it is used when we need to **apply a function to a collection of elements, but each function's input may also involve neighboring elements**.

  Usage. Common in tasks like image processing or solving partial differential equations (PDEs), where the value of each element depends not only on itself but also on its surrounding elements.

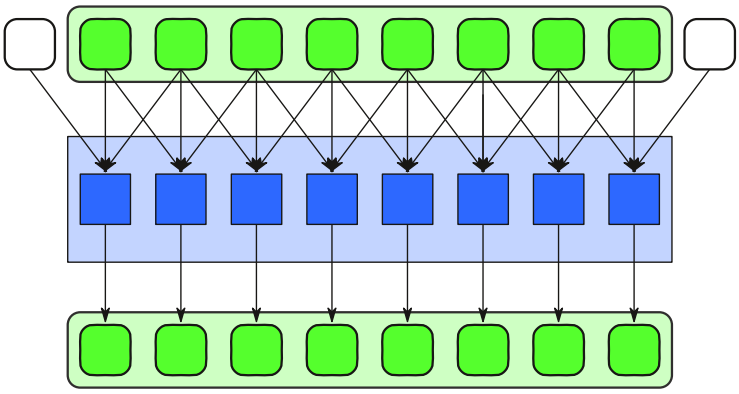


Figure 52: Visual representation of the Stencil Pattern.

Example 25: Stencil Pattern related to Map Pattern

In a 2D grid, the function applied to each cell may also read values from its neighboring cells (e.g., for edge detection in image processing).

- **Workpile Pattern related to Map Pattern**

  Description. Workpile pattern **allows work items to be added to the map while it is in progress, from inside map function instances**.

  Usage. The work grows and is consumed by the map. It terminates when no more work is available.

Example 26: Workpile Pattern related to Map Pattern

Useful in tasks like graph traversal or dynamic task generation where new tasks are created during the execution.

- **Divide-and-Conquer Pattern related to Map Pattern**

  Description. This pattern applies if a **problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially**.

  Usage. Common in algorithms like mergesort, quicksort, and solving mathematical problems such as the Fibonacci sequence or matrix multiplication.

Example 27: Divide-and-Conquer Pattern related to Map Pattern

Divide an array into smaller subarrays, sort them individually, and then merge the sorted subarrays.

10.3.4 Scaled Vector Addition (SAXPY)

SAXPY stands for **Single-Precision A*X Plus Y**. It **is a fundamental operation in linear algebra** and is part of the Basic Linear Algebra Subprograms (BLAS) library. The operation is defined as:

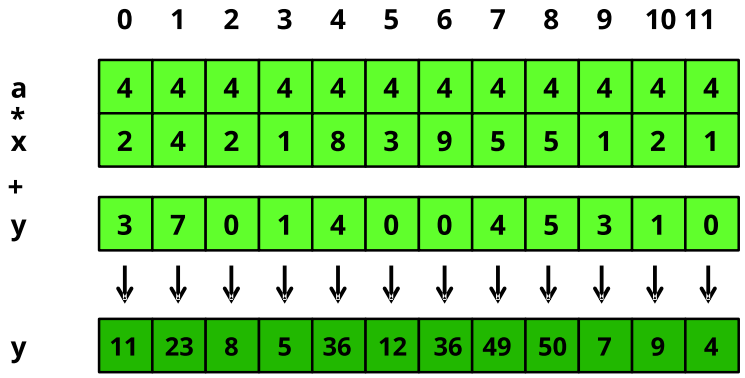
$$y = \alpha x + y \tag{15}$$

Where:

- α is a scalar (a single value).
- x and y are vectors (arrays of numbers).

The main observations that we can made are: each element in vector x is scaled by α; the scaled elements of x are added to the corresponding elements of y; but the most important thing is that each element in the vectors x and y is processed independently, making SAXPY **highly parallelizable**.

Maybe, a visual representation, can be useful to identify which pattern can be applied:



Implementations

- A **serial implementation** of SAXPY written:

```
void saxpy_serial(
    size_t n,        // the number of elements in the vectors
    float a,         // scale factor
    const float x[],// the first input vector
    float y[]        // the output vector and second input
    vector
) {
    for (size_t i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

- **Threading Building Blocks** (TBB, parallel) implementation:

```
void saxpy_tbb(
    int n,      // the number of elements in the vectors
    float a,    // scale factor
    float x[],  // the first input vector
    float y[]   // the output vector and second input vector
) {
    tbb::parallel_for(
        tbb::blocked_range<int>(0, n),
        [&](tbb::blocked_range<int> r) {
            for (size_t i = r.begin(); i != r.end(); i++)
                y[i] = a * x[i] + y[i];
        }
    );
}
```

  Where:

  - `tbb::parallel_for(range, function)`
    - `range`: Specifies the range of indices to be processed (`0` to `n`).
    - `function`: A lambda function that defines the operation for each subrange.
  - Lambda function: `[&](tbb::blocked_range<int> r)`
    - `[&]`: Capture clause that captures all variables by reference.
    - `(tbb::blocked_range<int> r)`: Parameter list; in this case, the range `r` to be processed.
    - Function body:

```
for (size_t i = r.begin(); i != r.end(); i++)
    y[i] = a * x[i] + y[i];
```

      Each element `y[i]` is updated independently within the subrange.

  It works like this:

  - **Range Splitting**: `tbb::blocked_range` divides the index range into subranges, which are processed in parallel.
  - **Dynamic Load Balancing**: TBB's scheduler dynamically assigns subranges to available threads, ensuring efficient load balancing.
  - **Data Independence**: Each update to `y[i]` is independent of other elements, making it safe for parallel execution.

- Cilk Plus (**parallel**) **implementation**:

```
void saxpy_cilk(
    int n,      // the number of elements in the vectors
    float a,    // scale factor
    float x[],  // the first input vector
    float y[]   // the output vector and second input vector
) {
    cilk_for (size_t i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

It works like this:

- **Automatic Parallelization**: `cilk_for` divides the loop iterations among available threads, balancing the workload dynamically.
- **Data Independence**: Each update to `y[i]` is independent of other elements, making it safe for parallel execution.
- **Simplified Syntax**: `cilk_for` provides a simple and familiar loop syntax for parallel loops, making it easy to convert serial loops into parallel loops.

- OpenCL (**parallel**) **implementation**:

```
__kernel void saxpy_opencl(
    __constant float a,
    __constant float* x,
    __constant float* y
) {
    int i = get_global_id(0);
    y[i] = a * x[i] + y[i];
}
```

It works much like CUDA.

- OpenMP (**parallel**) **implementation**:

```
void saxpy_openmp(
    int n,      // the number of elements in the vectors
    float a,    // scale factor
    float x[],  // the first input vector
    float y[]   // the output vector and second input vector
) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

10.4 Collectives operations

Collective operations are fundamental to parallel computing. They **involve communication patterns that enable data to be distributed among multiple processes, combined, or transformed in various ways**.

In essence, collectives are operations where all processes in a parallel program participate.

Collective Patterns

- **Reduce**: Combines data from all processes into a single result.
- **Scan**: Computes cumulative operations (such as prefix sum) across processes.
- **Partition**: Divides data into subsets and distributes them among processes.
- **Scatter**: Distributes distinct pieces of data from one process to all other processes.
- **Gather**: Collects distinct pieces of data from all processes into one process.

Collectives are used in parallel algorithms to **ensure efficient and effective data handling across multiple processes**. They help in achieving tasks such as data aggregation, distribution, synchronization, and more.

10.4.1 Reduce (or Reduction) Pattern

The **Reduce Pattern** (or commonly called **Reduction Pattern**) is used to **combine a collection of elements into a summary value using a combiner function**. This pattern has already been introduced on page 287.

The **combiner function** combines elements pairwise. It must be associative to be parallelizable. Some common combiner functions include addition, multiplication, and finding the maximum or minimum value.

Is it really important that the combiner function must be associative?

It is not important, it is essential. Before answering, we need to recall a bit of theory.

A **binary operation** (like addition) **is associative if the order in which the operations are performed does not change the result**.

Therefore, for a reduction operation to be parallelizable, the combiner function must be associative. This is because parallel reduction involves combining elements in pairs, **often in different order**, across multiple operations. If the operation is associative, **we can divide the work between processes and combine the results in any order without changing the final result**.

In summary, the combiner function must be associative to ensure that the end result is the same regardless of how the elements are grouped or combined in parallel.

If the reader is still not convinced, we present the following example as proof.

Example 28: Why associative property is important

It is well know that the addition is associative because:

$$(A + B) + C = A + (B + C)$$

This allows us to parallelize addition across multiple processes. The order in which the sums are performed doesn't matter, because the property guarantees that the result is always the same. Instead, subtraction is non-associative because:

$$(A - B) - C \neq A - (B - C)$$

This means that we cannot parallelize subtraction in the same way. In this scenario, different executions lead to different execution orders and different results.

Types of Reduction

1. **Serial Reduction**. **Combines elements in a linear fashion**, one after the other.

   Example 29: Serial Reduction Pattern

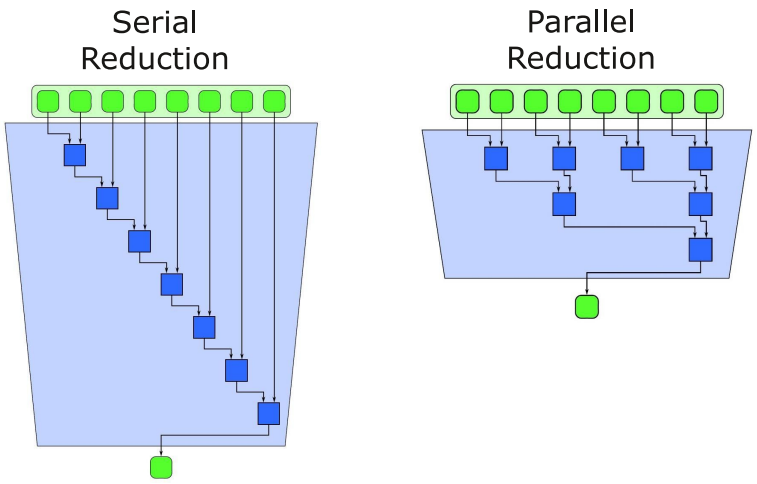
   Summing elements `[A, B, C, D]` would be proceed as:

   $$((\texttt{A} + \texttt{B}) + \texttt{C}) + \texttt{D}$$

2. **Parallel Reduction**. **Combines elements in a tree-like structure**, allowing multiple operations to be performed simultaneously.

   Example 30: Parallel Reduction Pattern

   Summing elements `[A, B, C, D]` in parallel could proceed as $(\texttt{A} + \texttt{B})$ and $(\texttt{C} + \texttt{D})$ in one step, followed by the final sum of these results.



Vectorization

In general, **Vectorization** refers to the **process of converting an algorithm from operating on single elements at a time to operating on a set of elements simultaneously using vectorized instructions**.

Modern processors have special vector processing units (such as SIMD - Single Instruction, Multiple Data) that can handle multiple data elements in a single instruction.

The Reduction Pattern combines elements in pairs until a single result is obtained. **Vectorization can be applied to this process to combine multiple pairs simultaneously**.

Example 31: Reduce Pattern Vectorization

Suppose we have an array of numbers that we want to sum. Instead of summing each pair one by one, we can use vectorized instructions to sum several pairs at once. This is especially beneficial for large datasets.

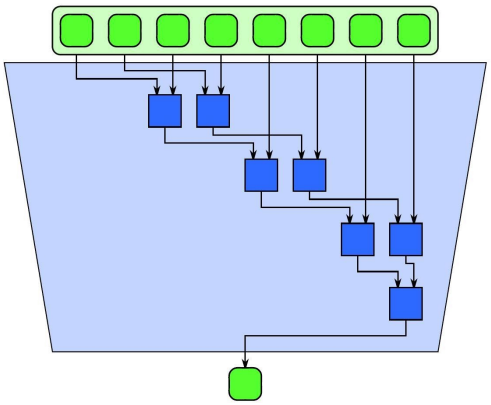


Figure 53: Graphical example of the vectorization technique.

Advantages

- **Speed**. Vectorization can greatly speed up the reduction process by making better use of CPU resources.
- **Parallelism**. It enhances the inherent parallelism of the reduction operation, further improving performance.

Tiling

Tiling (also known as blocking) is a **technique where the workload is divided into smaller, more manageable chunks or "tiles"**. Each tile can be processed independently, and the results can then be combined. We have already seen this topic in the CUDA environment, on page 136.

Goal. The main goal of tiling is to improve cache efficiency and reduce memory access latency by keeping frequently accessed data within the faster levels of the memory hierarchy (like the CPU cache).

The tiling in the reduction pattern is applied as follows:

1. **Divide Data**. The **entire dataset is divided into smaller tiles**. Each tile contains a subset of the data.
2. **Process Tiles**. Each tile is processed independently. For the reduction pattern, this means **performing the reduction operation on the elements within each tile**.
3. **Combine Results**. Once all tiles have been processed, the **results from each tile are combined to obtain the final result**.

Example 32: Tiling with Reduce Pattern

Imagine we have an array of numbers that we want to sum using tiling:

- Divide the array into smaller tiles (e.g., blocks of 100 elements each).
- Perform the summation within each tile in parallel.
- Combine the sums of all tiles to get the final result.

Advantages

- **Cache Efficiency**. By working on smaller tiles, the data can fit into the CPU cache, reducing the need to fetch data from slower main memory.
- **Parallelism**. Each tile can be processed in parallel, making it easier to distribute the workload across multiple processes or threads.
- **Scalability**. Tiling allows for better scalability, especially when dealing with large datasets.

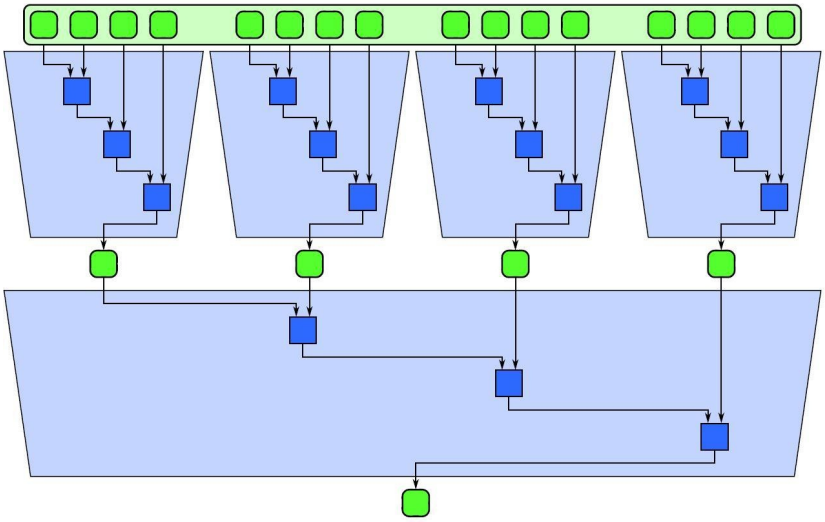


Figure 54: Graphical example of the tiling technique applied to the reduce pattern.

⚠ Limitations

Unfortunately, the reduce pattern has two major limitations: precision issues and order floating-point dependency.

✖ **Precision Issues**. Small rounding errors occur because some numbers cannot be represented exactly. For example, the number 0.1 cannot be represented precisely in binary floating-point format.[15]

When performing a reduction operation (e.g., summing a large array of floating-point numbers), these **small rounding errors can accumulate**. Each addition introduces a tiny error, and the more additions we perform, the larger the cumulative error becomes.

Example 33: Precision Issues

Suppose we sum the numbers `[0.1, 0.2, 0.3, 0.4]`. Ideally, the result should be `1.0`. However, due to rounding errors, we might get something like `0.9999999999999999` instead.
In large datasets, this effect can be more pronounced, leading to significant deviations from the expected result.

[15] Floating-point numbers are represented in a way that allows for a wide range of values but introduces some imprecision. This is because they are stored in a fixed number of bits, which can only approximate most real numbers.

✖ **Order Floating-Point Dependency**. Floating-point addition is not associative. This means that the order in which we add the numbers can affect the result.

In a parallel reduction, elements are combined in different orders depending on how they are distributed across processes. **Different orders of combination can lead to different results due to the non-associative nature of floating-point addition**.

Example 34: Order Floating-Point Dependency

Suppose we have three numbers `1e20`, `-1e20`, and `1.0`. If we add them as:

$$\texttt{(1e20 + -1e20) + 1.0} = \texttt{0.0 + 1.0} = 1.0$$

But if we add them as:

$$\texttt{1e20 + (-1e20 + 1.0)} \approx \texttt{1.0}$$

Due to the large magnitude difference between `1e20` and `1.0`.

These problems can be partially solved using some intelligent techniques such as:

- Kahan Summation Algorithm:
- Higher Precision:
- Pairwise Summation:
- Consistent Order:

However, in this course we don't delve into these topics.

10.4.2 Scan Pattern

The **Scan Pattern** is used to **compute all partial results of an input sequence using a binary associative operation**. It generates a new sequence where each element is the result of applying the operation to all preceding elements in the input sequence.

Types of Scans

- **Inclusive Scan Pattern**
  - **Definition**. Each element in the result sequence includes the value of the current element in the input sequence.
  - **Result**. The **result for each position is the combination** (using the associative operation) **of all elements up to and including the current element**.
  - **Use Case**. It is useful when the result must contain the current element at every position. It can be used in parallel algorithms where each step depends on all previous steps, including the current one.

Example 35: Inclusive Scan Pattern

For an input array `[A, B, C, D]`, an inclusive scan with an associative operation + results in:

```
[A, A + B, A + B + C, A + B + C + D]
```

For numbers:

- Input: $[1, 2, 3, 4]$
- Output: $[1, 3, 6, 10]$

- **Exclusive Scan Pattern**
  - **Definition**. Each element in the result sequence excludes the value of the current element from the input sequence. Instead, it includes the combination of all prior elements.
  - **Result**. The **result for each position is the combination** (using the associative operation) **of all elements before the current element**.
  - **Use Case**. It is useful for scenarios where the result at each position should exclude the current element and provide the cumulative effect of all previous elements. It's often used in algorithms that require a shift in position, such as calculating running sums starting from the next element.

Example 36: Exclusive Scan Pattern

For an input array `[A, B, C, D]`, an exclusive scan with an associative operation `+` results in:

`[×, A, A + B, A + B + C]`

Where $\times$ is the identity element of the operation. For numbers:

- Input: $[1, 2, 3, 4]$
- Output: $[0, (1), (1 + 2), (1 + 2 + 3)] = [0, 1, 3, 6]$

Techniques for parallelizing Scan Pattern

- **Prefix Maximum Algorithm with Up and Down Sweep**

  Definition. Prefix Maximum Algorithm with Up and Down Sweep is a special **implementation of the scan pattern**, **designed to find the maximum value within an array using parallel processing**.

  It calculates the maximum value at the i-th index of the input array and contains the maximum value of the array at the last position.

  For **example**, given the array:

  `[1, 4, 0, 2, 7, 2, 4, 3]`

  It returns:

  `[1, 4, 4, 4, 7, 7, 7, 7]`

  Where indicates that at position 2, the sub-array `[1, 4, 0]`, the maximum value that can be found is `4`.

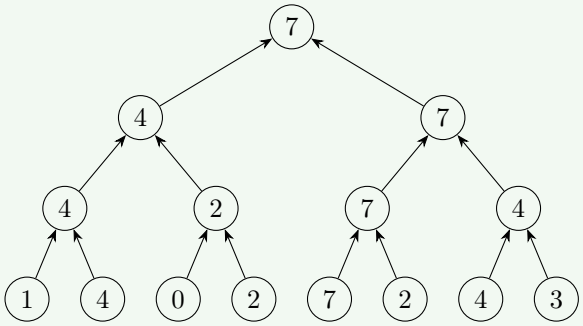
  It consists of two main phases: Up-Sweep (Reduce) Phase and Down-Sweep Phase.

  Algorithm

  1. **Up-Sweep (Reduce) Phase** . Create a balanced binary tree of partial results, focusing on **finding the maximum value**.
     (a) **Pairwise Combination**: Begin by combining elements in pairs to find the maximum of each pair.
     (b) **Level-by-Level Combination**: Continue combining results at each level of the tree until a single result (the maximum value) is obtained.

Example 37: Up-Sweep (Reduce) technique

We follow a tree approach to manually compute the up-sweep. We divide the input into pairs and evaluate each pair with the max operator. The result is the next node. For an array `[1, 4, 0, 2, 7, 2, 4, 3]`:



The comparison made are:

(a) Level 1:

* Pairs:
  - `max(1, 4)`
  - `max(0, 2)`
  - `max(7, 2)`
  - `max(4, 3)`
* Result: `[4, 2, 7, 4]`

(b) Level 2:

* Pairs:
  - `max(4, 2)`
  - `max(7, 4)`
* Result: `[4, 7]`

(c) Level 3:

* Pair: `max(4, 7)`
* Result: `[7]`

At this point, the Up-Sweep phase has found the maximum value of the entire array.

2. **Down-Sweep Phase**. Use the partial results from the Up-Sweep phase to **compute the final scan results for all elements in the array**.
   (a) **Initialization**: Begin with an initial value, typically the identity element (for maximum, it could be negative infinity $-\infty$).

(b) **Propagate Results**: Propagate the maximum values back down the tree, updating the results for each element.

Example 37 (continue): Down-Sweep technique

Continuing from the Up-Sweep phase result `[7]` and the original array `[1, 4, 0, 2, 7, 2, 4, 3]` (page 317).

The main rule followed during the down-sweep technique is the following: the left leaf always inserts the value of the parent node, while the right leaf inserts the value calculated after applying the `max` operation to the value of the parent node and the left side just removed (which we find on the up-sweep graph).

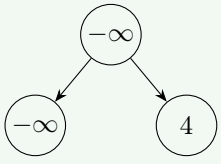
On the last level of the tree, there is an exception: before inserting the left leaf directly, the `max` operation of the inherited value and the input value must be performed.

The steps are detailed:

(a) Replace the maximum value on the root (7) and insert the identity element, in our case the $-\infty$.

(b) Level 1:

* Insert the parent node on the left leaf, so $-\infty$.
* On the right leaf, insert the performed operation between the parent's value and the left leaf of the up-sweep tree. So the values to compare are $-\infty$ and 4: `max(`$-\infty$`, 4) = 4`



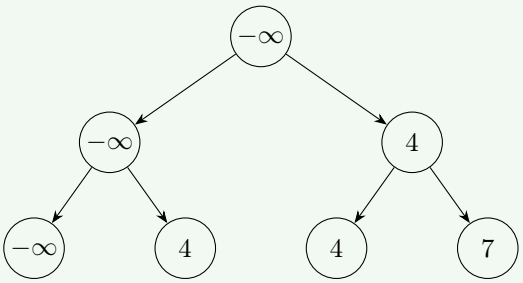
(c) Level 2 ((from left to the right)):

* Vertex $-\infty$:
  - Insert the parent node on the **left leaf**, so $-\infty$. It's not the last leaf of the tree, so we don't do any checking.
  - The **right leaf** comparison is between $-\infty$ (the father's value) and the left leaf on the up-sweep tree (the value just replaced by the $-\infty$), so 4. Again, the comparison is:

    `max(`$-\infty$`, 4) = 4`

* Vertex 4:
  - Insert the parent node on the **left leaf**, so 4.
  - The **right leaf** comparison is between 4 (the father's value) and the left leaf on the up-sweep tree (the value just replaced by 4), so 7. The comparison is:

    `max(4, 7) = 7`



(d) Level 3 (from left to the right):

* Vertex $-\infty$:
  - **Left leaf**. Since this is the last level, we need to compute an extra check. We need to compute:

    `max(inherited val, input val)` (16)

    In our case, the inherited value is the $-\infty$, but the input value on the first position is 1. Therefore:

    `max(`$-\infty$`, 1) = 1`

- **Right leaf**. We compare the value we just removed, 1, to the value of the parent: $-\infty$.

  `max(1, −∞) = 1`

  The result should be 1 BUT we are at the last level of the tree. So we need to do an extra check between the inherited value (what we want to place) and the input value on the second position (our case 4):

  `max(1, 4) = 4`

* Vertex 4:
  - **Left leaf**. We apply formula 2d on page 319, the idea is the same as before:

    `max(4, 0) = 4`

  - **Right leaf**. We compare the value just removed, 0, to the value of parent: 4.

    `max(0, 4) = 4`

    And the value obtained with the input value on the array:

    `max(4, 2) = 4`

* Vertex 4:
  - **Left leaf**. We apply formula 2d on page 319, the idea is the same as before:

    `max(4, 7) = 7`

  - **Right leaf**. We compare the value just removed, 7, to the value of parent: 4.

    `max(7, 4) = 7`

    And the value obtained with the input value on the array:

    `max(7, 2) = 7`

* Vertex 7:
  - **Left leaf**. We apply formula 2d on page 319, the idea is the same as before:

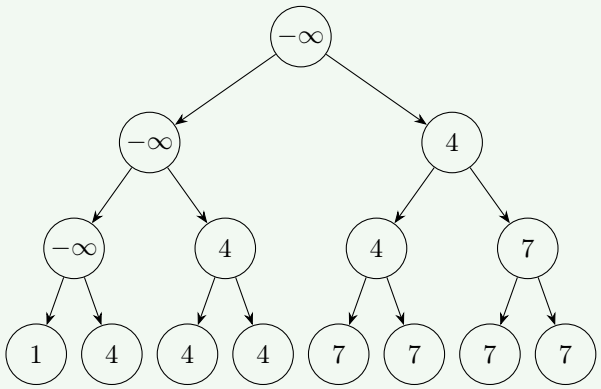
    ```
    max(7, 4) = 7
    ```

  - **Right leaf**. We compare the value just removed, 4, to the value of parent: 7.

    ```
    max(4, 7) = 7
    ```

    And the value obtained with the input value on the array:

    ```
    max(7, 3) = 7
    ```



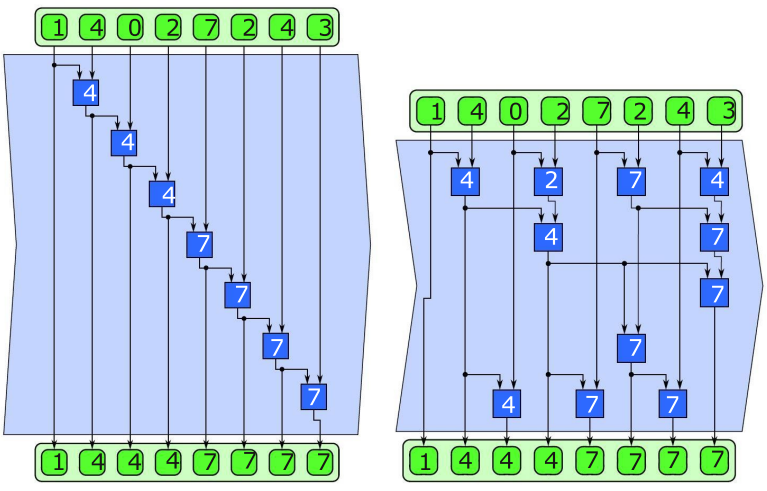


Figure 55: Graphical example of the Maximum Algorithm with Up and Down Sweep. On the left a sequential version and on the right a parallel version. The parallel version was analyzed on the previous pages.

- **Three-Phase Scan with Tiling**

  Definition. Three-Phase Scan with Tiling is a technique that **divides the input array into smaller tiles and processes them in three distinct phases** to efficiently perform the scan operation. The main goal is to improve parallelism and cache efficiency by splitting the input array into smaller tiles and processing them independently.

  Algorithm

  1. **Tiling and Local Scan**. Divide the input array into smaller tiles and perform the scan operation on each tile independently.
     (a) **Divide the Input Array**: Break the input array into smaller blocks or tiles.
     (b) **Local Scan** (Inclusive): Compute the scan for each tile independently.

     Example 38: Tiling and Local Scan

     For an array `[1, 2, 3, 4, 5, 6, 7, 8]` with a tile size of 4:

     * Tiles: `[1, 2, 3, 4]` and `[5, 6, 7, 8]`
     * Local Scans:
       · First tile: `[1, 3, 6, 10]`
       · Second tile: `[5, 11, 18, 26]`

  2. **Scan of Tile Results**. Compute the **scan of the final elements of each tile to handle dependencies between tiles**.
     (a) **Extract Final Elements**: Take the last element of each tile.
     (b) **Global Scan**: Perform a scan operation on these final elements to propagate the results across tiles.

     Example 39: Scan of Tile Results

     * Final Elements: `10` (from the first tile) and `26` (from the second tile)
     * Global Scan: `[10, 36]` (assuming addition and identity element `0`)

  3. **Distribution of Tile Results**. **Distribute the results of the scanned tile results to all elements in their respective tiles**.
     (a) **Distribute Results**: Add the scan results of the previous tiles to the elements of the current tile.

Example 40: Distribution of Tile Results

* First Tile Remains the Same: `[1, 3, 6, 10]`
* Second Tile: add the scan result `10` from the first tile's last element:
  - `5 + 10 = 15`
  - `11 + 10 = 21`
  - `18 + 10 = 28`
  - `26 + 10 = 36`

  `[15, 21, 28, 36]`

Combining the results of the two tiles, we get:

`[1, 3, 6, 10, 15, 21, 28, 36]`

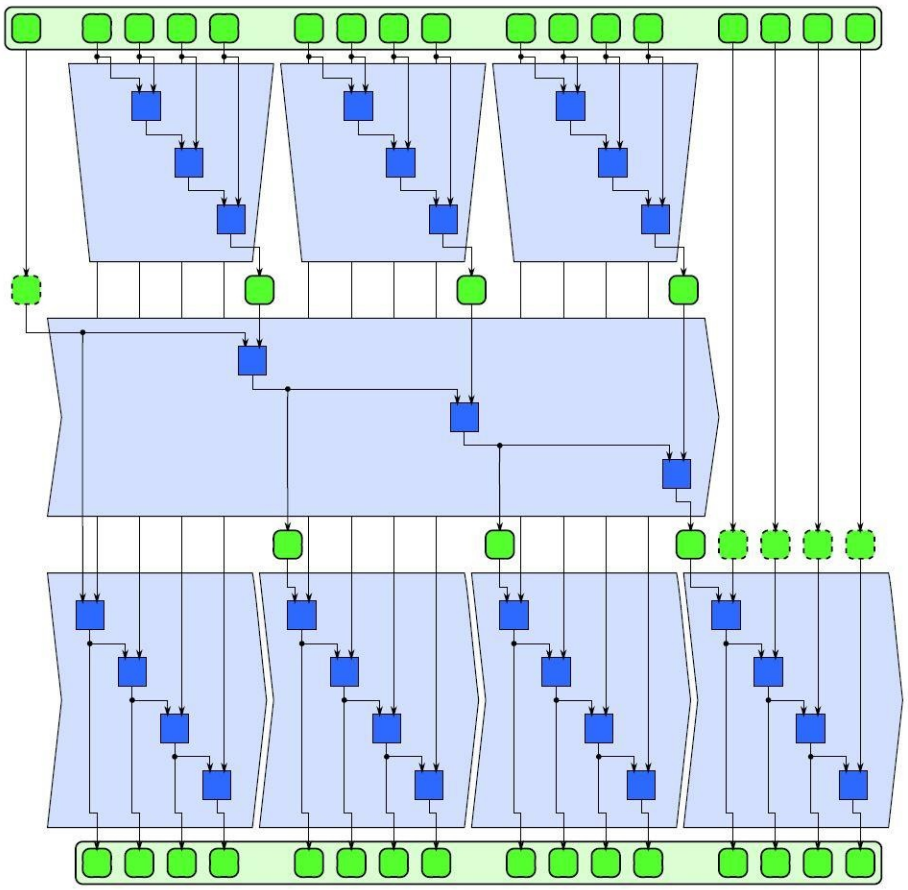


Figure 56: Graphical example of the Three-Phase Scan with Tiling.

- **Fusion of Map Pattern with Scan Pattern**

  Definition. Combine the transformation capabilities of the map pattern with the cumulative operations of the scan pattern to **achieve more complex and efficient parallel computations**.

  When we fuse the map and scan patterns, we **first apply the map function to transform each element in the input sequence**, and **then apply the scan operation to the transformed sequence**.

  Algorithm

  1. **Apply Map Function**: Transform each element in the input sequence using the map function.
  2. **Apply Scan Operation**: Perform the scan operation on the transformed sequence to compute cumulative results.

Example 41: Fusion of Map Pattern with Scan Pattern

Let's consider a practical example where we want to compute the prefix sums of the squares of an input array `[1, 2, 3, 4]`.

1. **Map Function**
   (a) Define the map function as $f(x) = x^2$.
   (b) Apply the map function: `[1, 4, 9, 16]` (squares of the input elements),
2. **Scan Operation**
   (a) Perform an inclusive scan on the transformed sequence `[1, 4, 9, 16]`.
   (b) Inclusive scan result: `[1, 5, 14, 30]`.

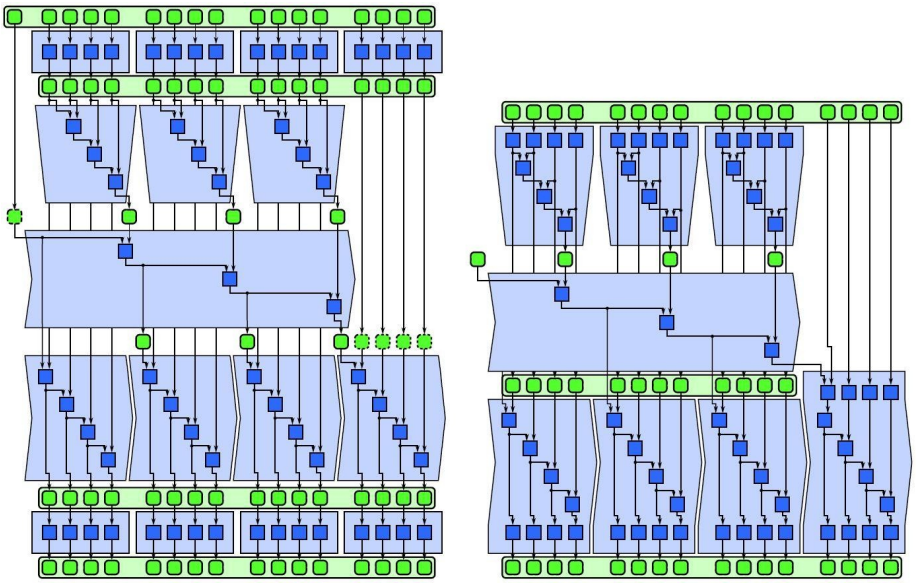


Figure 57: Graphical example of the Fusion of Map Pattern with Scan Pattern.

10.5 Gather Pattern

10.5.1 What is a Gather?

Why is the need for the gather pattern born?

There are two main reasons for this:

1. **Data Movement**.

   Problem. Data movement is often more costly than computation, especially when transferring data across memory layers or networks. This includes factors like:

   - Locality optimization. Reducing data access times by keeping frequently accessed data close to the processing unit.
   - Efficient data access. Improving performance by minimizing the cost of accessing scattered data locations.

   Solution. The gather pattern addresses these issues by efficiently collecting data from multiple, scattered memory locations into a contiguous structure.

2. **Parallel Data Reorganization**.

   Problem. To achieve performance gains in parallel algorithms, data often needs to be reorganized efficiently. This involves:

   - Parallel data movement. Moving large datasets simultaneously rather than sequentially.
   - Consistency management. Ensuring data integrity while reorganizing in parallel.
   - Intermediate data structures. Holding temporary results during reorganization.

   Solution. The gather pattern is crucial for parallel data reorganization because it:

   - Enables parallel data movement, allowing multiple data points to be gathered simultaneously from different sources.
   - Maintains data consistency, ensuring that the reorganization process doesn't introduce errors.
   - Supports intermediate structures, making it easier to manage data during complex transformations.

What is a Gather Pattern?

The **Gather Pattern** is a data movement operation that **creates a new (*source*) collection of data by reading elements from an existing (*input*) data collection**. It is commonly used in parallel computing to efficiently reorganize or extract data based on specific indices. At first it might seem complicated, but with some simple examples it will be very easy.

In general, it works like this:

1. **Read** data from the input collection at positions specified.
2. **Write** the gathered data to the output collection in the same order as the indexes specified in Input.
3. Returns an output collection that has the same shape (or dimensionality) as the index collection.

It can be seen as a **combination** of `map` and `random read` operations because it **performs multiple reads from non-contiguous memory locations**.

Serial Implementation

The following pseudocode shows a serial implementation of the gather pattern:

```
template<typename Data, typename Idx>
void gather(
    size_t n,     // number of elements in data collection
    size_t m,     // number of elements in index collection
    Data a[],     // input data collection (n elements)
    Data A[],     // output data collection (m elements)
                  // used to modify the output in place
    Idx idx[]     // input index collection (m elements)
) {
    // Iterate over index collection
    for (size_t i = 0; i < m; ++i) {
        // Get the i-th index
        size_t j = idx[i];
        // Ensure index is within bounds;
        // It avoids buffer overflow,
        // and ensure memory safety
        assert(0 <= j && j < n);
        // Perform random read and write to output
        A[i] = a[j];
    }
}
```

The pseudo-signature of the function is:

- Input:
  - `a[]`: original data collection (size `n`).
  - `idx[]`: index collection specifying which elements to gather (size `m`).
- Output:
  - `A[]`: output collection to store gathered elements (size `m`, trivial since we iterate over the input indices `idx`)

The process performs the following operations:

1. **Loop through each index** `i` in the index collection (`idx[]`).
2. **Access the data** at position `idx[i]` in `a[]`.
3. **Assertion** ensures that the index is valid (within the bounds of `a[]`).
4. **Copy the data** from `a[idx[i]]` to the corresponding position `A[i]` in the output collection.

It is interesting to note that there are good **opportunities to parallelize** the code thanks to the for loop. In fact, each iteration is independent of the others (no data dependencies between iterations). More precisely, each thread writes to a unique position in the output array (collection) `A`. Also, each position is valid thanks to the assert boundary check.

Example 42: Gather Pattern

Let the following arguments:

- Source Array, contains the original data elements:

  `A = [A, B, C, D, E, F, G, H]`

  Indices `0` through `7`, so the `n` argument is `8`.

- Index Array, specifies which elements from the source array to retrieve:

  `idx = [1, 5, 0, 2, 2, 4]`

  Indices `0` through `5`, so the `m` argument is `6`.

- Output Array, will store the gathered elements in the order of the indices `idx`:

  `B = []`

  Empty initially, will be filled based on `idx`.

The function calculates:

1. First iteration:
   (a) Get *0*-th index: `idx[0] = 1`
   (b) Boundary check? `0 <= idx[0]` $\wedge$ `idx[0] < 8` ✔
   (c) Perform random read: `B[0] = A[idx[0]] = A[1] = B`
2. Second iteration:
   (a) Get *1*-th index: `idx[1] = 5`
   (b) Boundary check? `0 <= idx[1]` $\wedge$ `idx[1] < 8` ✔
   (c) Perform random read: `B[1] = A[idx[1]] = A[5] = F`

3. Third iteration:
   (a) Get *2*-th index: `idx[2] = 0`
   (b) Boundary check? `0 <= idx[2]` $\wedge$ `idx[2] < 8` ✔
   (c) Perform random read: `B[2] = A[idx[2]] = A[0] = A`
4. Fourth iteration:
   (a) Get *3*-th index: `idx[3] = 2`
   (b) Boundary check? `0 <= idx[3]` $\wedge$ `idx[3] < 8` ✔
   (c) Perform random read: `B[3] = A[idx[3]] = A[2] = C`
5. Fifth iteration:
   (a) Get *4*-th index: `idx[4] = 2`
   (b) Boundary check? `0 <= idx[4]` $\wedge$ `idx[4] < 8` ✔
   (c) Perform random read: `B[4] = A[idx[4]] = A[2] = C`
6. Sixth iteration:
   (a) Get *5*-th index: `idx[5] = 4`
   (b) Boundary check? `0 <= idx[5]` $\wedge$ `idx[5] < 8` ✔
   (c) Perform random read: `B[5] = A[idx[5]] = A[4] = E`

The returned output collection is:

`B = [B, F, A, C, C, E]`

Some interesting observations we can make are:

- **Same dimensionality**: output data collection (array `B`) has the same number of elements as the number of indexes in the index collection (value `m`).
- **Same types**: elements of the output collection are of the same type as the input data collection.

10.5.2 Shift

The **Shift operation** is a specialized case of the gather pattern where **data is moved uniformly left or right in memory**. Unlike random gathers (already seen in section 10.5.1, page 326), shifts involve **regular**, **predictable data movement** with fixed offsets.

Key Features

- **Data Movement**. Elements are shifted by a fixed distance.
- **Offset Access**. Data accesses are offset uniformly, making the operation predictable.

Boundary Conditions

With the shift operation, the main problem we can encounter is mainly one: boundary conditions.

When shifting data, **handling the edge elements** (out-of-bounds data) **is crucial**. There are several strategies:

- **Default Value**. Fill the vacant position with a default (e.g., `0` or `null`).

Example 43: Default Boundary Condition

Input array:

```
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Shift with default boundary condition:

- Left shift:

```
A = [2, 3, 4, 5, 6, 7, 8, 9, 10, 0]
```

- Right shift:

```
A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Duplicate**. Replicate the edge element to fill the gap.

Example 44: Duplicate Boundary Condition

Input array:

```
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Shift with duplicate boundary condition:

- Left shift:

```
A = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10]
```

– Right shift:

```
A = [1, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

✔ **Rotate**. Wrap around the array, moving the last element to the first position (circular shift).

Example 45: Rotate Boundary Condition

Input array:

```
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Shift with rotate boundary condition:

– Left shift:

```
A = [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
```

– Right shift:

```
A = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Advantages & Benefits

- Shifts are highly efficient with **vector instructions** due to their regularity.
- They allow **parallel shifting** of multiple elements simultaneously.
- Good **data locality** enhances performance, as adjacent elements are accessed together.

10.5.3 Zip

The **Zip operation** is a special case of the gather pattern where **two (or more) arrays are combined by interleaving their elements**. It functions like a zipper, **taking one element from each array in sequence to form a new combined array**. It is important to note that it **works with different types**, so we can zip elements of different types, like integers and floats, or even complex objects.

How does it work?

The operation **takes an element from the first array**, then **one from the second** array, another from the third, and so on, and **repeats the process**. The **output is the combined sequence**.

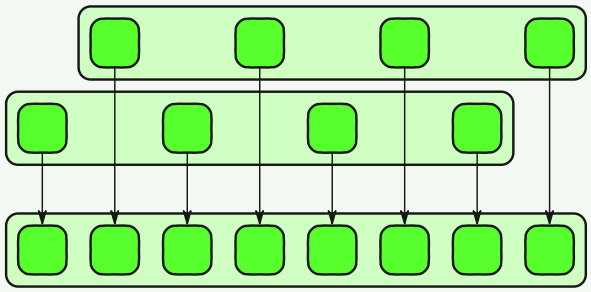
Example 46: zip operation

Consider two arrays:

1. Real Parts: contains real numbers.
2. Imaginary Parts: contains imaginary numbers.

The output is a combined sequence like:

```
[Real0, Imag0, Real1, Imag1, Real2, Imag2, ...]
```



Parallelism

Each pair of elements (one from each array) can be **combined independently**. This independence allows **parallel execution since there's no dependency between the operations for different pairs**.

10.5.4 Unzip

The **Unzip operation** is essentially the **reverse of the zip operation**. While zip combines multiple arrays into one by interleaving their elements, **unzip separates a single interleaved array back into its original components**.

How does it work?

The operation **extracts sub-arrays at regular offsets** (strides) **to separate the elements into their original groups**. The input is a combined sequence (e.g., alternating real and imaginary parts).

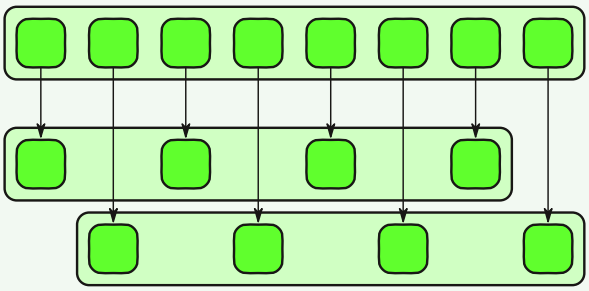
Example 47: unzip operation

Consider the input array:

```
[Real0, Imag0, Real1, Imag1, Real2, Imag2, ...]
```

The output are:

- Array of Real Parts: `[Real0, Real1, Real2]`
- Array of Imaginary Parts: `[Imag0, Imag1, Imag2]`



Parallelism

Each element extraction is independent. This allows for **parallel data access** because there's no dependency between different extractions.

However, the unzip operation is an **efficient data extraction because it takes advantage of stride-based memory access**, which can be optimized for performance.

10.6 Scatter Pattern

10.6.1 What is a Scatter?

The **Scatter Pattern** is a data movement pattern where **elements from a source array are distributed (or scattered) to various locations in a destination array based on specified indices**. Essentially, it's about writing data to random locations.

⭐ Key Features

In general, the **input** of the scatter pattern is:

- **Source data array**: value to be written.
- **Index array**: specifies where each value should be written in the destination.

Each element from the source is written to the position in the target array specified by the corresponding index. The **output is a target array where the data is scattered across the positions**.

⚖ Gather vs. Scatter

The main differences between gather and scatter are:

- **Gather** (read focused): Involves **random reads** where the read **locations are provided as input**.
- **Scatter** (write focused): Involves **random writes** with write **locations provided as input**, which can lead to race conditions.

🛠 Serial implementation

The following pseudocode shows a serial implementation of the scatter pattern:

```
template<typename Data, typename Idx>
void scatter(
    size_t n,      // Number of elements
                   // in the output data collection
    size_t m,      // Number of elements
                   // in the input data and index collection
    Data a[],      // Input data collection (m elements)
    Data A[],      // Output data collection (n elements)
    Idx idx[]      // Input index collection (m elements)
) {
    // Loop through each input element
    for (size_t i = 0; i < m; ++i) {
        // Get the target index from the index array
        size_t j = idx[i];
        // Ensure the index is within output array bounds
        assert(0 <= j && j < n);
        // Perform the scatter: write a[i] to A[j]
        A[j] = a[i];
    }
}
```

The method is characterized by:

- **Loop** (`for`): Iterates through each element in the input array `a[]` (length `m`).
- **Indexing** (`j = idx[i]`): Determines **where to place** `a[i]` in the output array `A[]`.
- **Boundary Check** (`assert`): Ensures the index `j` is **valid** (avoids out-of-bounds errors).
- **Write Operation** (`A[j] = a[i]`): **Scatters** the data from `a[i]` to `A[j]`.

The possibilities to parallelize the code are interesting, but instead of the gather pattern, here we have a problem with race conditions on write operations. So the code can be parallelized (**parallelize over for loop to perform random write**), but we need to find some strategies to avoid race conditions.

Race Conditions in Scatter

Unfortunately, the scatter pattern can lead to race conditions because of the way it handles writes to memory. As we know, a race condition occurs when two or more operations try to access and modify the same data at the same time, and the final result depends on the order in which the operations are performed.

In the scatter pattern, we're performing random writes to different locations in memory. The **problem arises when multiple write operations target the same location at the same time**. Because the writes happen in parallel, we can't predict:

1. Which write will happen first
2. Which value will be stored last

Example 48: Race Condition in Scatter

Consider this:

- Source Data:

```
A = [A, B, C, D, E, F]
```

- Index Array (write locations):

```
idx = [1, 5, 0, 2, 2, 4]
```

This means that the two threads that will handle the 3rd and 4th positions on the index array will suffer from a race condition. Because if they try to write at the same time, the result will depend on which thread finishes last:

- If thread A (writing `D`) finishes last, position `2` will be equal to `D`.
- If thread B (writing `E`) finishes last, position `2` will be equal to `E`.

10.6.2 Avoid race conditions

10.6.2.1 Atomic Scatter

Atomic Scatter is a **collision resolution strategy** used when multiple threads attempt to write to the same memory location simultaneously. It is based on one of the most common and famous topic: **atomic operations**.

How does it work?

✔ **Atomic Writes**. **Each write is atomic**, meaning it either completes fully or doesn't happen at all. No partial writes.

✖ **Non-Deterministic Outcome**. Since it relies solely on atomic writes, this doesn't mean that there is a mechanism to check which thread should write before another. When a collision occurs, the outcome depends on which write completes first. There's no predefined rule for determining which value to store.

Example 49: Atomic Scatter

Consider this:

- Source Data:

```
A = [A, B, C, D, E, F]
```

- Index Array (write locations):

```
idx = [1, 5, 0, 2, 2, 4]
```

Notice that each thread is assigned to every position of the `idx` array. This means that threads #4 and #5 must write to the same position on the output array. However, we adopt the atomic strategies so that what happens depends on the order of writing.

A possibles scenario:

1. Thread #4 asks to do an atomic write to position 2 of the output array. The operation is marked atomic and can be done safely. The value written to `A[2]` is `D`.
2. Then thread #5 performs the same steps and overwrites the value written by thread #4 and writes the value `E` to `A[2]`.

This is non-deterministic, so this is a possible scenario, but it could also be the other way around and the final value could be `D`.

Advantages

- ✔ **Simple to implement**. No need for complex locking mechanisms.
- ✔ **Fast**. Atomic operations are optimized in hardware, making them efficient for parallel execution.

Disadvantages

- ✖ **Unpredictable results**. Non-determinism can be problematic for algorithms that require consistent outputs.
- ✖ **Data loss**. One of the colliding values will be lost without any mechanism to combine or preserve both.

When to use Atomic Scatter?

Atomic scatter is suitable when:

- **Collisions are rare**: so the occasional data loss or non-determinism is acceptable.
- **We don't care which value is kept**: for example, in random sampling or approximate algorithms.

10.6.2.2 Permutation Scatter

Permutation Scatter is a special type of scatter pattern where **collisions are strictly prohibited**. This means that **every input element must be placed in a unique output position**, ensuring that the output is a permutation (reordering) of the input.

Key Features

- **Collisions are illegal**. Unlike atomic scatter, where collisions are resolved non-deterministically, **permutation scatter ensures that no two elements try to write to the same location**.
- **Output is a permutation of the input**. The output array consists of the same elements as the input but in a different order, **without any duplicates or missing values**.
- **Collision checking in advance**. To guarantee uniqueness, a **pre-check must be done** to detect any duplicate indices **before executing the scatter operation**.

What if there is a collision during the pre-check?

If collisions exist, scatter cannot proceed as-is and may **need to be transformed into a gather operation** instead.

Why convert a scatter to a gather?

Instead of letting multiple elements write to the same index in parallel (scatter), we can:

1. Detect collisions beforehand by checking for duplicate indices in the index array.
2. Reformulate the operation as a gather, where **each output index reads from a unique input location**, instead of multiple inputs writing to the same index.
3. Ensure correctness by enforcing a one-to-one mapping between input and output locations.

So it is clear that the **gather ensures that each output location reads from only one input, eliminating write conflicts**.

Example 50: Convert Scatter to Gather

Image we have an input array and an index array:

- Input: `A = [A, B, C, D]`
- Index: `idx = [1, 3, 3, 2]`

Here we have a conflict at index 3 because both B and C want to write here. Therefore, the pre-check reports this conflict; the operation has been marked as undefined behavior; we convert the operation from a scatter to a gather.

Now the approaches we can make are two: we **convert the whole operation** into a gather and we do that in parallel (to achieve high performance); or the **position where the permutation cannot guarantee the atomic write, are passed to a gather function to guarantee atomicity**.

The implementation depends on the programmer and the behavior we want. What we need to know is that: **if a conflict is detected during permutation, the conversion to gather is the best choice that we can make**.

Permutation scatter requires a one-to-one mapping between input and output indices. If collisions exist, scatter must be **turned into gather so that instead of multiple inputs writing to the same location, each output position retrieves its correct value without conflicts**.

⚖ Permutation vs. Atomic Scatter

- **Collision**
  - **Atomic**: Allowed, resolved arbitrarily.
  - **Permutation**: Not allowed.
- **Write conflicts**
  - **Atomic**: Handled using atomic operations.
  - **Permutation**: Avoided by pre-checking
- **Output structure**
  - **Atomic**: Can overwrite values.
  - **Permutation**: Always a permutation of input.
- **Use Cases**
  - **Atomic**: Hash tables, parallel memory writes.
  - **Permutation**: Matrix transposition, FFT scrambling.

10.6.2.3 Merge Scatter

The **Merge Scatter** strategy is a way to resolve write conflicts in a scatter operation by **applying a merge operator that combines values instead of discarding or randomly selecting one**.

How does it work?

Instead of letting one thread overwrite another's data, Merge Scatter **applies a merge function to combine values when a conflict occurs**. The function must be:

- **Associative**, order of grouping doesn't change result:

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

- **Commutative**, order of values doesn't change result:

$$a \oplus b = b \oplus a$$

Why these properties? Because parallel execution means that scatter operations happen in arbitrary order, so the **merge result must be independent of the order in which values arrive**. In other words, it guarantees deterministic behavior (because we always know what the result will be).

Example 51: Merge Scatter

Suppose we have the following data:

Input	`[2, 3, 1, 1, 5, 6]`
Indices	`[1, 5, 0, 2, 2, 4]`

Positions `3` and `4` have multiple writes, causing collisions. Therefore, we use the merge scatter strategy and we define the merge function as: addition.

So instead of overwriting, we sum the values in case of a conflict:

`[1, 2, 6, _, 6, 3]`

At the second position, the index where there was a collision, the merge function was applied between the value `1` and `5` (`1 + 5 = 6`).

Advantages

- **Preserves Data**. Unlike atomic scatter (where some values are lost), merge scatter ensures **all data contributes to the final output**.
- **Parallel-Friendly**. Works well in parallel settings since **order doesn't affect the result**.

✔ **Useful in some scenario**: histogram computation, sparse matrix operations, parallel reductions.

Disadvantages

✖ **Increased Computational Overhead**. Instead of a simple write operation, we now have to **perform a merge operation** (e.g., addition, `max`, `min`). This **adds extra computational cost**, especially if the merge function is complex. Also, some architectures may require atomic operations or locks to ensure safe merging, which can slow down execution.

In summary, the **more conflicts there are, the more overhead is introduced by the merge function**.

✖ **Not Always Deterministic**. If the merge **function is not associative and commutative**, the **result can depend on the order of execution**, which is unpredictable in parallel settings.

Note that this is very common in real-world scenarios; for **example**, when using floating-point addition, different execution orders can result in small numerical inconsistencies due to floating-point precision issues.

✖ **Possible Loss of Information**. If the **merge function is not a lossless operation**, **some original data might be lost**.

Example 52: Loss of Information

For example, if the merge function is `max()`, only the largest value survives, and all others are discarded. This mean **we cannot reconstruct the original inputs from the output**.

✖ **Limited Applicability**. Merge Scatter is **only applicable when a meaningful merge function exists**. Some applications require exact values, and merging (e.g., summing or taking max) may not be acceptable.

Example 53: Limited Applicability

For example, if each thread writes a different pixel color in an image processing task, merging colors with sum/max may produce unintended artifacts.

So we can summarize **when we need to avoid merge scatter**:

✖ When we need **exact values** without modification.

✖ When the merge function is **not associative or commutative**.

✖ When merging **significantly increases computational complexity**.