departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Map and Reduce Patterns

Concurrency and Parallelism — 2017-18

Master in Computer Science

(Mestrado Integrado em Eng. Informática)

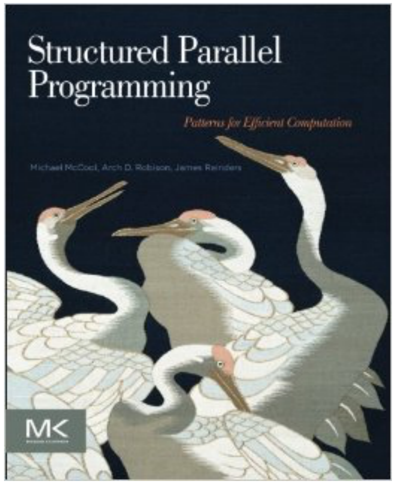
Joao Lourenço <joao.lourenco@fct.unl.pt>

Source: Parallel Computing, CIS 410/510, Department of Computer and Information Science

Outline

- Structured programming patterns overview
  - Concept of programming patterns
  - Serial and parallel control flow patterns
  - Serial and parallel data management patterns

  - Bibliography:
    - **Chapters 4 and 5** of book
      McCool M., Arch M., Reinders J.;
      Structured Parallel Programming: Patterns for
      Efficient Computation;
      Morgan Kaufmann (2012);
      ISBN: 978-0-12-415993-8



Outline

- Map pattern
  - Optimizations
    - sequences of Maps
    - code Fusion
    - cache Fusion
  - Related Patterns
  - Example: Scaled Vector Addition (SAXPY)
- Reduce
  - Example: Dot Product

Mapping

- "Do the same thing many times"

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

- Well-known higher order function in languages like ML, Haskell, Scala

map: $\forall ab.(a \to b) List\langle a \rangle \to List\langle b \rangle$

applies a function to each element in a list and returns a list of results

Example Maps

Add 1 to every item in an array

0	1	2	3	4	5
0	4	5	3	1	0
↓	↓	↓	↓	↓	↓
1	5	6	4	2	1

Double every item in an array

0	1	2	3	4	5
3	7	0	1	4	0
↓	↓	↓	↓	↓	↓
6	14	0	2	8	0

Key Point: An operation is a map if it can be applied to each element without knowledge of its neighbors.

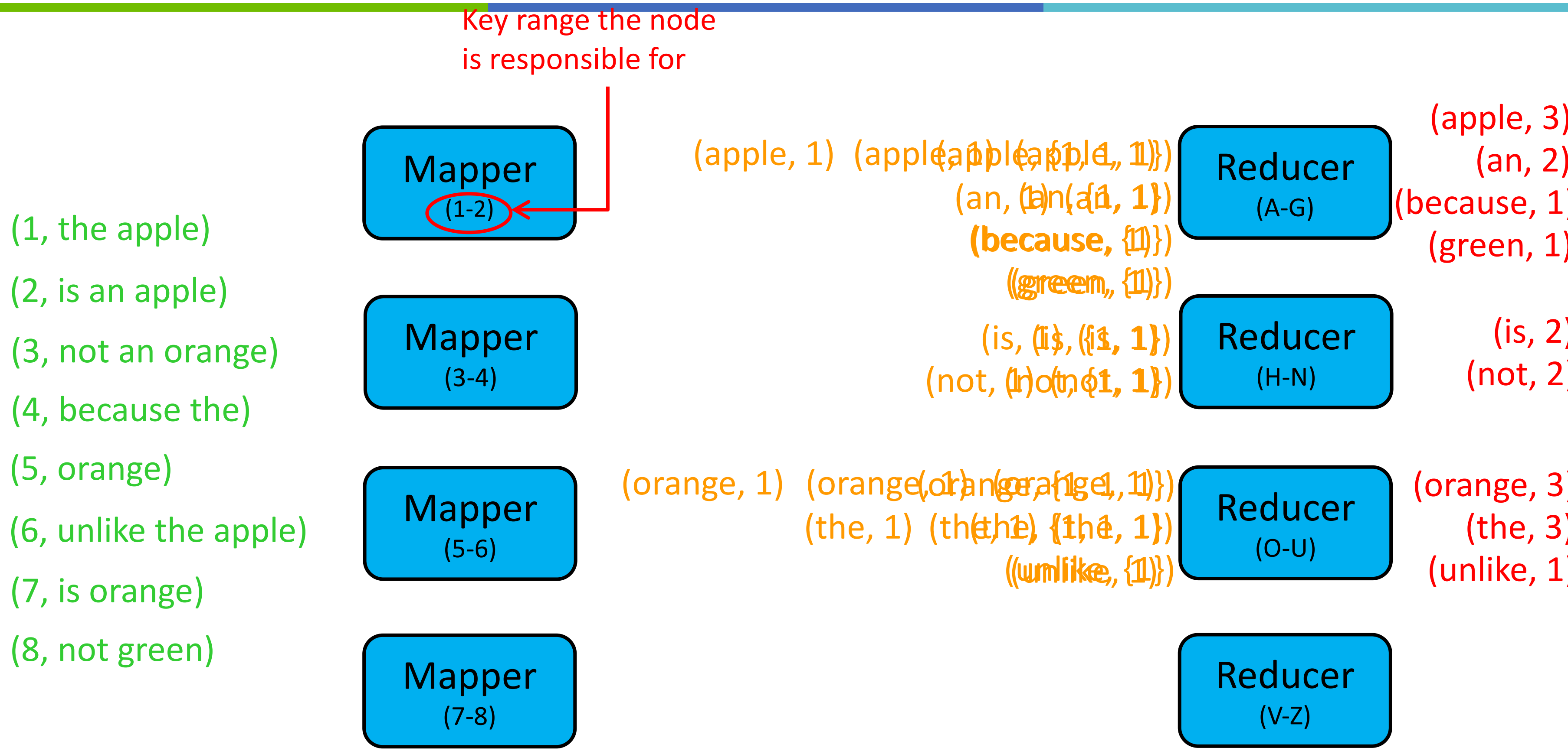
Key Idea

- Map is a “foreach loop” where each iteration is independent

Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups! More precisely: $T(\infty)$ is O(1) plus implementation overhead that is O(log n)…so $T(\infty) \in O(\log n)$.

Simple example: Word count



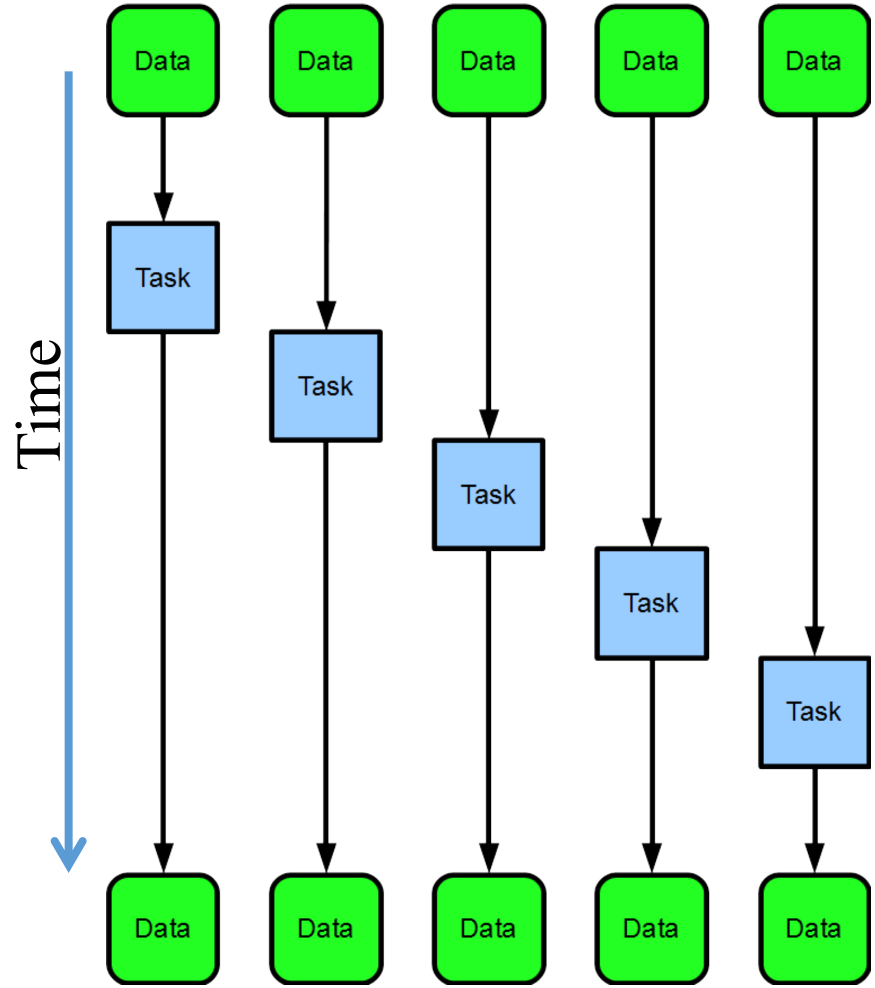
1. Each mapper receives some of the KV-pairs as input
2. The mappers process the KV-pairs one by one
3. Each KV-pair output by the mapper is sent to the reducer that is responsible for it
4. The reducers sort their input by key and group it
5. The reducers process their input one group at a time

Sequential Map

```
for(int n=0;
     n< array.length;
     ++n){

          process(array[n]);

}
```

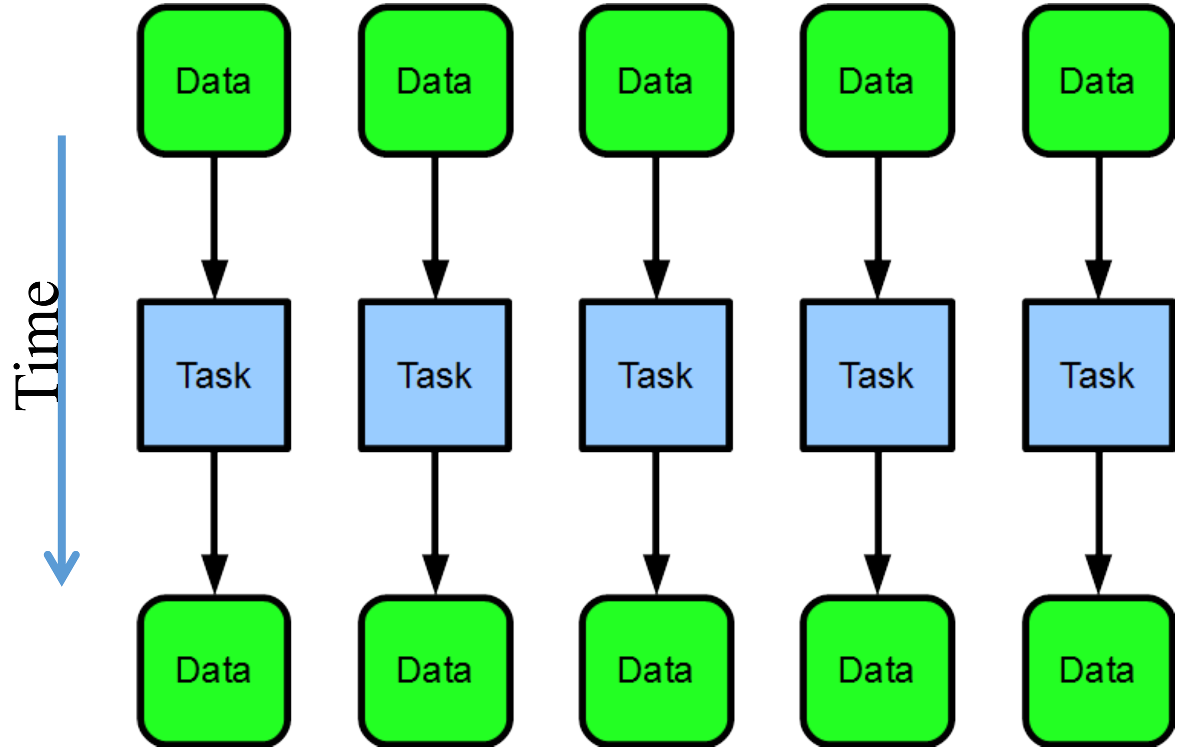


Parallel Map

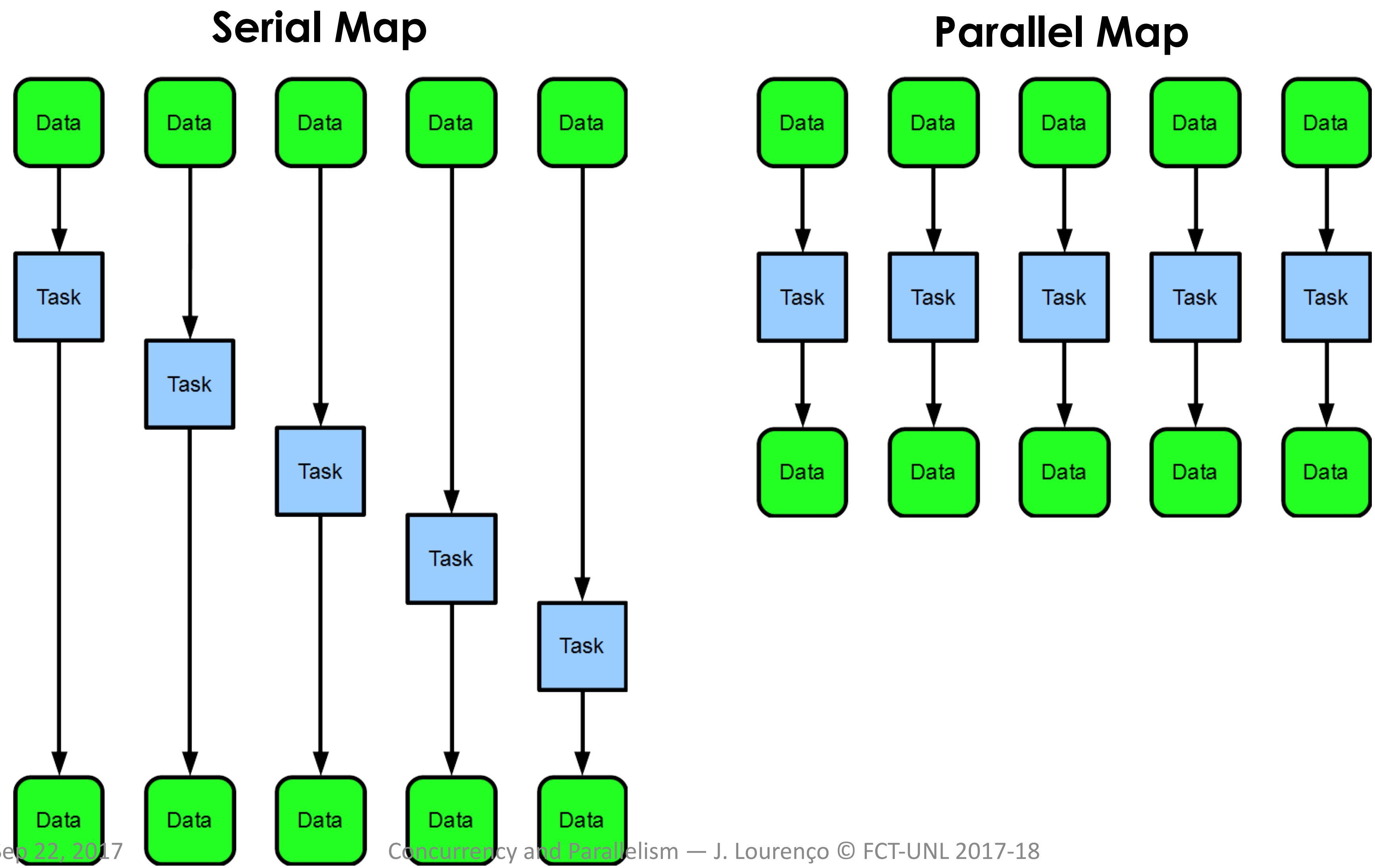
```
parallel_for_each(
    x in array){

        process(x);

}
```

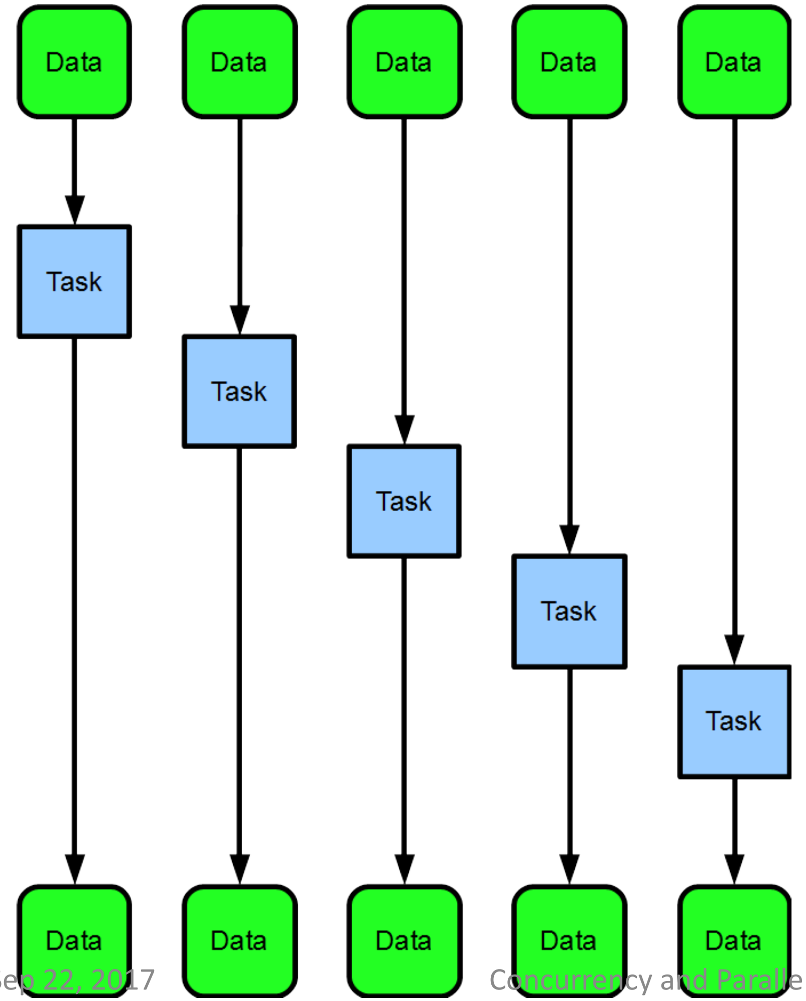


Comparing Maps

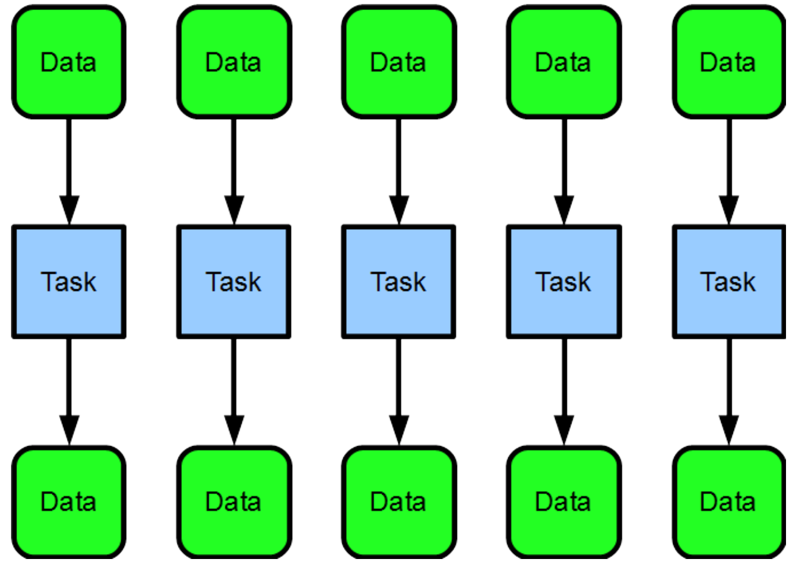


Comparing Maps

Serial Map



Parallel Map



Speedup

The space here is speedup. With the parallel map, our program finished execution early, while the serial map is still running.

Independence

- The key to (embarrasing) parallelism is independence

Warning: No shared state!

Map function should be “pure” (or “pure-ish”) and should not modify shared states

- Modifying shared state breaks perfect independence
- Results of accidentally violating independence:
  - non-determinism
  - data-races
  - undefined behavior
  - segfaults

Implementation and API

- OpenMP and CilkPlus contain a parallel ***for*** language construct
- Map is a mode of use of parallel ***for***
- TBB uses **higher order functions** with lambda expressions/"functors"
- Some languages (CilkPlus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

```
A[:] = A[:]*5;
```

is CilkPlus array notation for "multiply every element in A by 5"

Unary Maps

Unary Maps

So far we have only dealt with mapping over a single collection…

Map with 1 Input, 1 Output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
result	6	14	0	2	8	0	0	8	10	6	2	0

```
int  oneToOne ( int x[11] ) {
        return x*2;
}
```

N-ary Maps

N-ary Maps

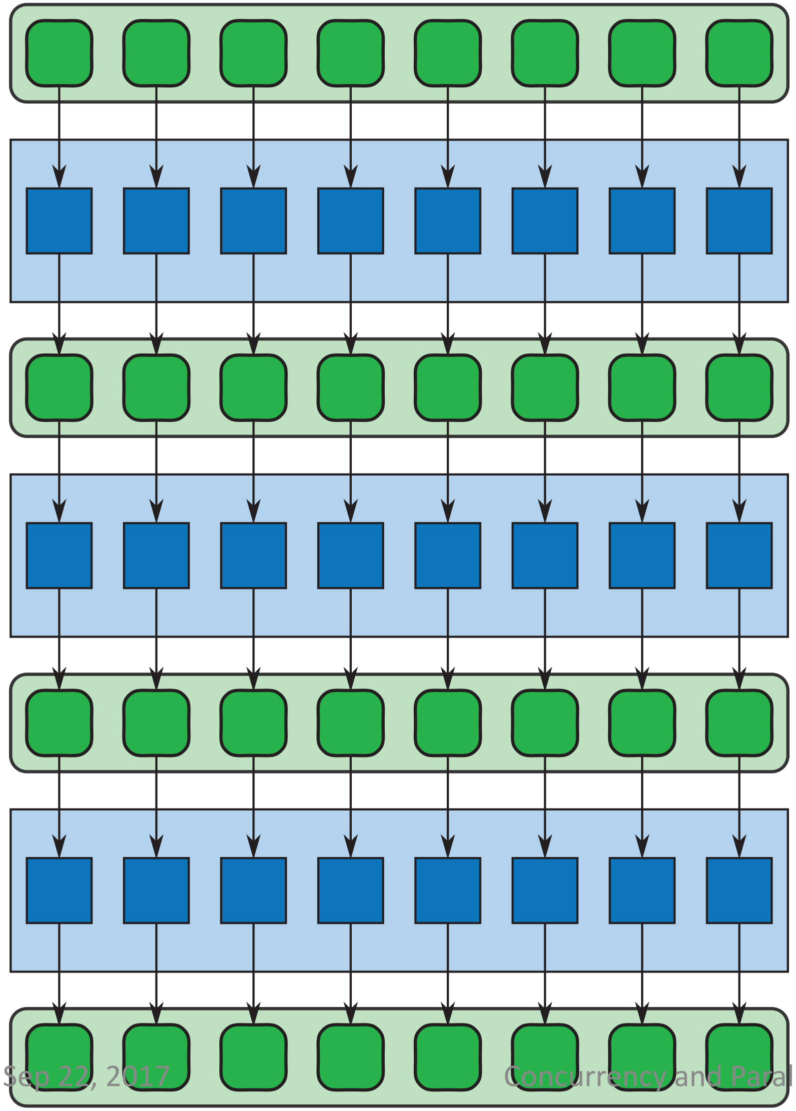
But, sometimes it makes sense to map over multiple collections at once…

Map with 2 Inputs, 1 Output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
y	2	4	2	1	8	3	9	5	5	1	2	1
result	5	11	2	2	12	3	9	9	10	4	3	1

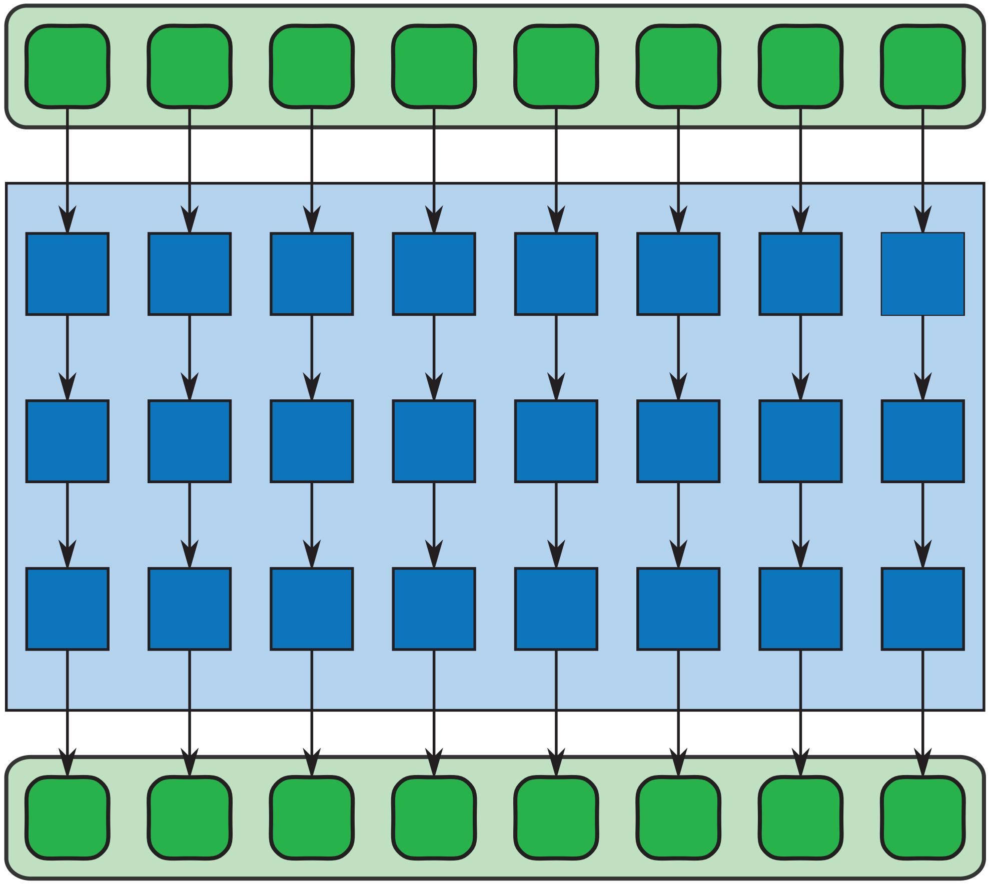
```
int  twoToOne ( int x[11], int y[11] ) {
        return x+y;
}
```

Optimization – Sequences of Maps



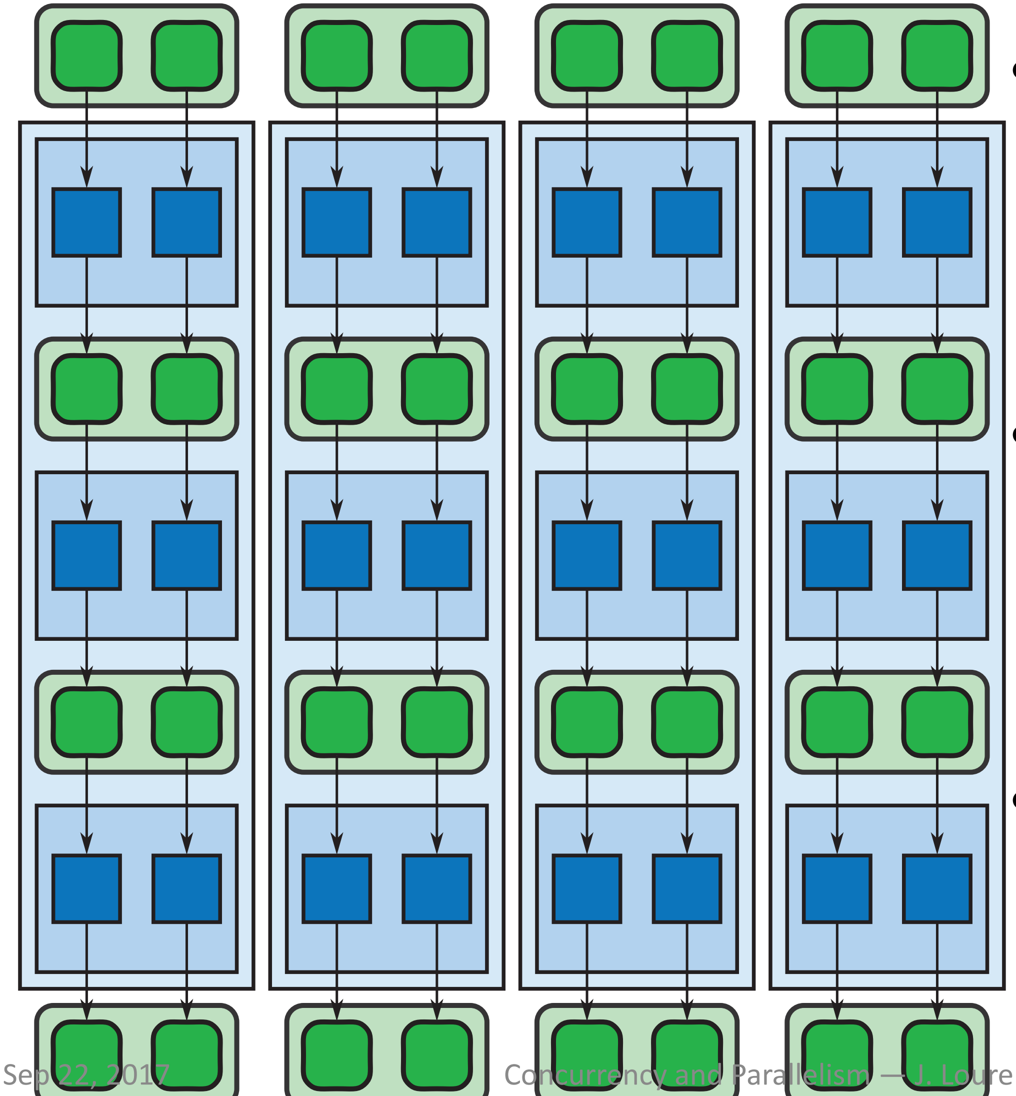
- Often several map operations occur in sequence
  - Vector math consists of many small operations such as additions and multiplications applied as maps
- A naïve implementation may write each intermediate result to memory, wasting memory BW and likely overwhelming the cache

Optimization – Code Fusion



- Can sometimes “fuse” together the operations to perform them at once
- Adds arithmetic intensity, reduces memory/cache usage
- Ideally, operations can be performed using registers alone

Optimization – Cache Fusion



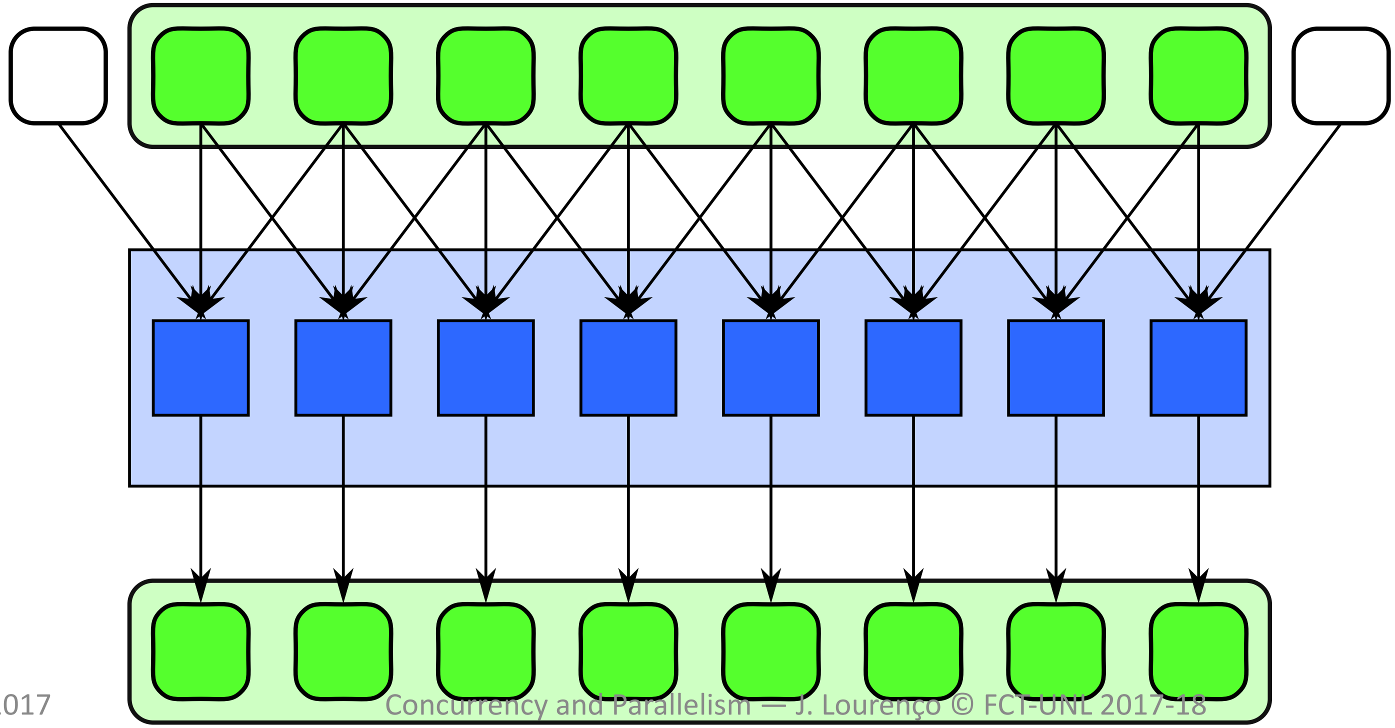
- Sometimes impractical to fuse together the map operations
- Can instead break the work into blocks, giving each CPU one block at a time
- Hopefully, operations use cache alone

Related Patterns

- Three patterns related to map are now discussed here:
  - Stencil
  - Workpile
  - Divide-and-Conquer

Stencil

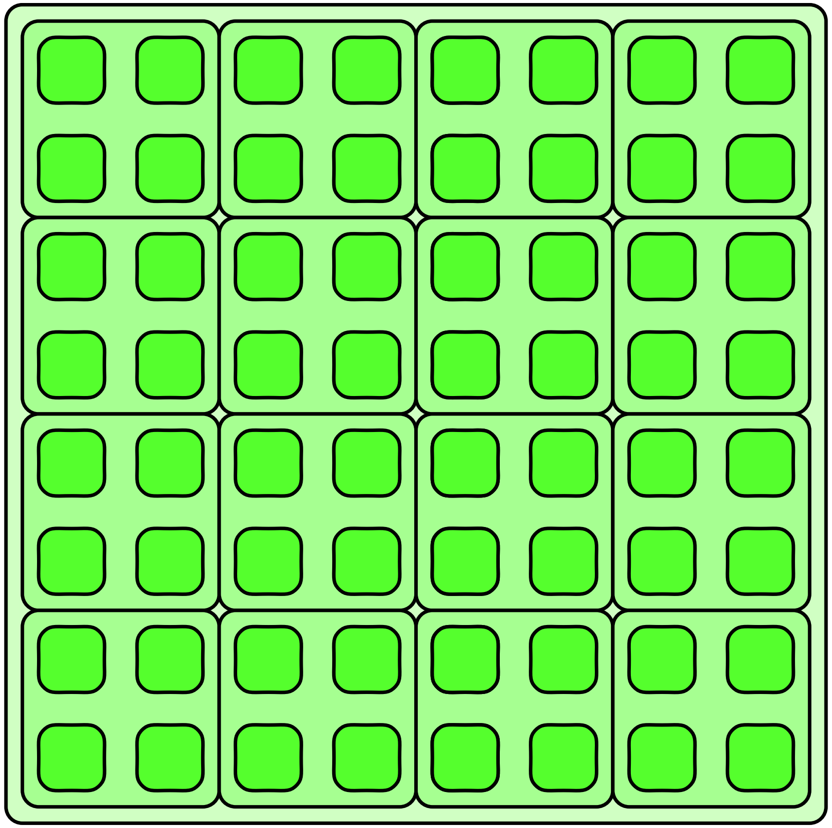
- Each instance of the map function accesses neighbors of its input, offset from its usual input
- Common in imaging and PDE solvers



Workpile

- Work items can be added to the map while it is in progress, from inside map function instances
- Work grows and is consumed by the map
- Workpile pattern terminates when no more work is available

Divide-and-Conquer



- Applies if a problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially

Example: Scaled Vector Addition (SAXPY)

- $y \leftarrow ax + y$
  - Scales vector x by a and adds it to vector y
  - Result is stored in input vector y
- Comes from the BLAS (Basic Linear Algebra Subprograms) library
- **Every element in vector x and vector y are independent**

What does $y \leftarrow ax + y$ look like?

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Visual: $y \leftarrow ax + y$

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Twelve processors used → one for each element in the vector

Visual: $y \leftarrow ax + y$

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Six processors used → one for every two elements in the vector

Visual: $y \leftarrow ax + y$

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Two processors used → one for every six elements in the vector

Serial SAXPY Implementation

```
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];
}
```

TBB SAXPY Implementation

The image part with relationship ID rId2 was not found in the file.

```
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];
        }
    );
}
```

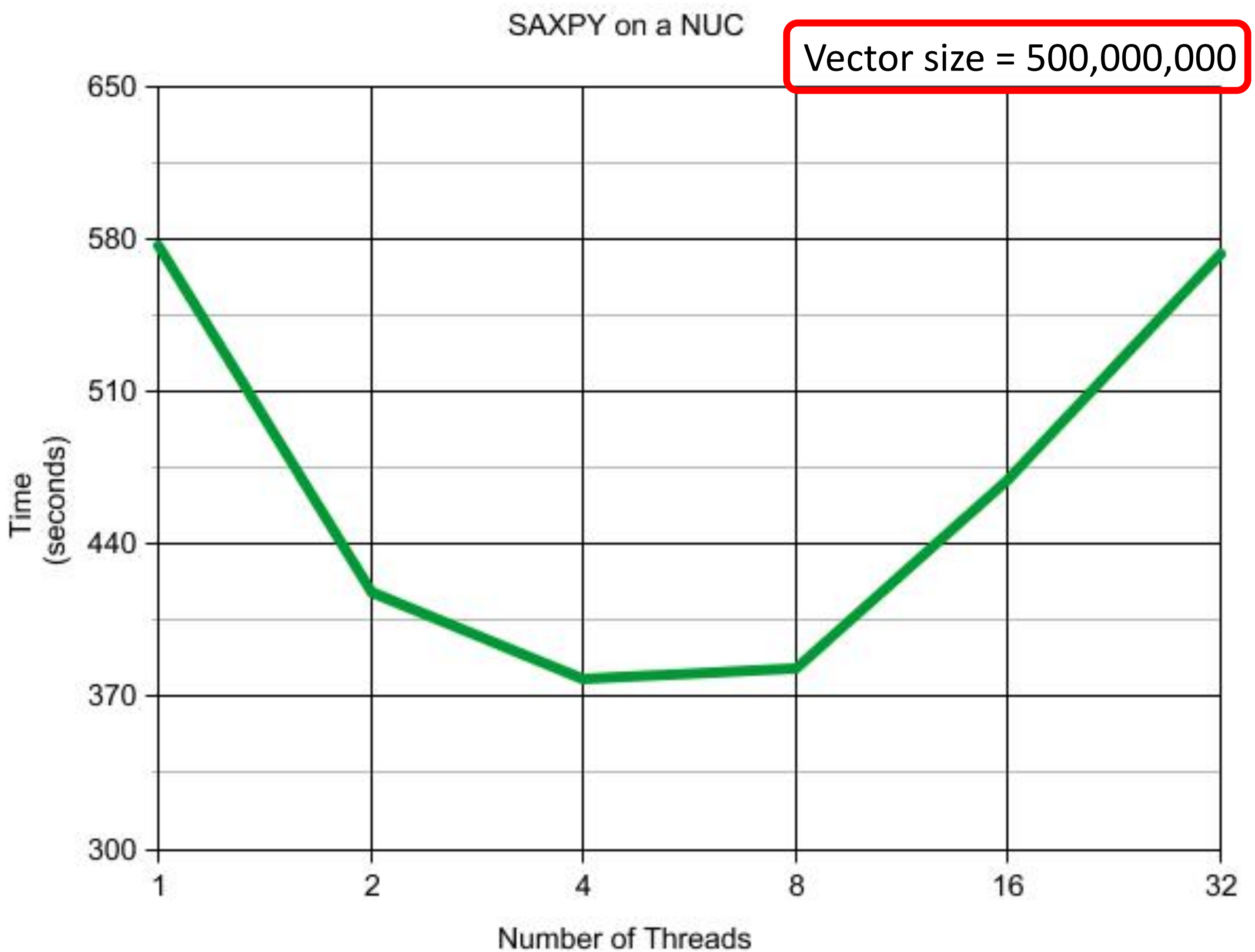
Cilk Plus SAXPY Implementation

```
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 (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

OpenMP SAXPY Implentation

```
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];
}
```

OpenMP SAXPY Performance

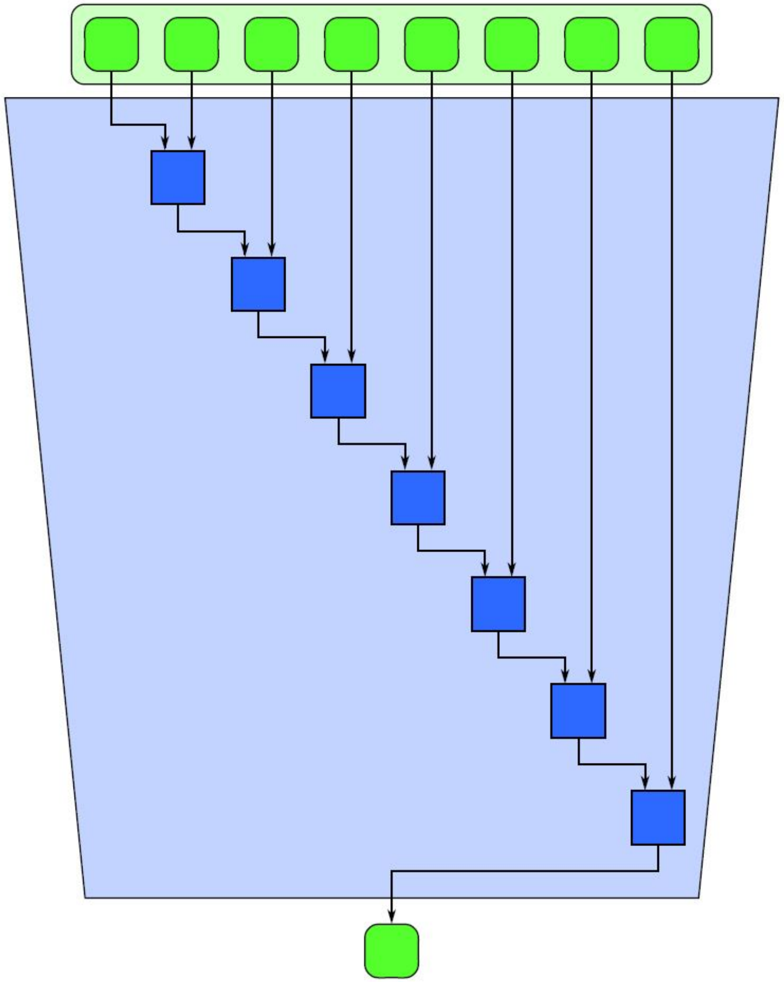


Reduce

- **Reduce** is used to combine a collection of elements into one summary value
- A combiner function combines elements pairwise
- A combiner function only needs to be *associative* to be parallelizable
- Example combiner functions:
  - Addition
  - Multiplication
  - Maximum / Minimum

Reduce

Serial Reduction

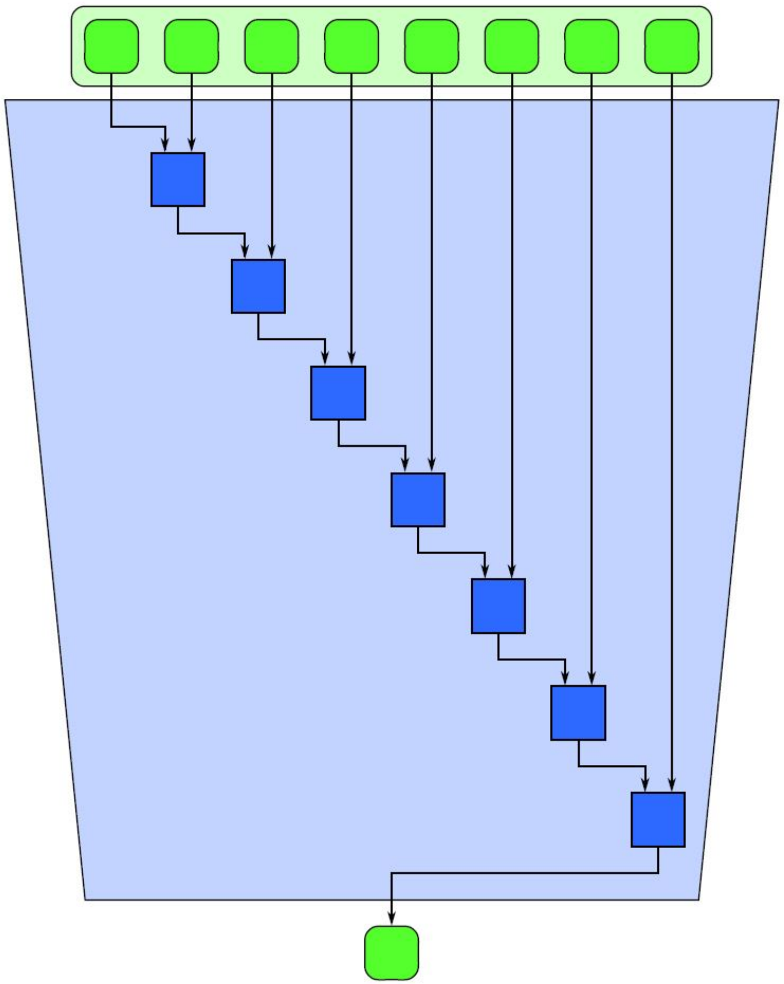


```
template<typename T>
T reduce(
    T (*f)(T,T),  // combiner function
    size_t n,  // number of elements in input array
    T a[]        // input array
) {
    assert(n > 0);
    T accum = a[0];
    for (size_t i = 1; i < n; i++) {
        accum = f(accum, a[i]);
    }
    return accum;
}
```

The input array cannot be empty!

Reduce

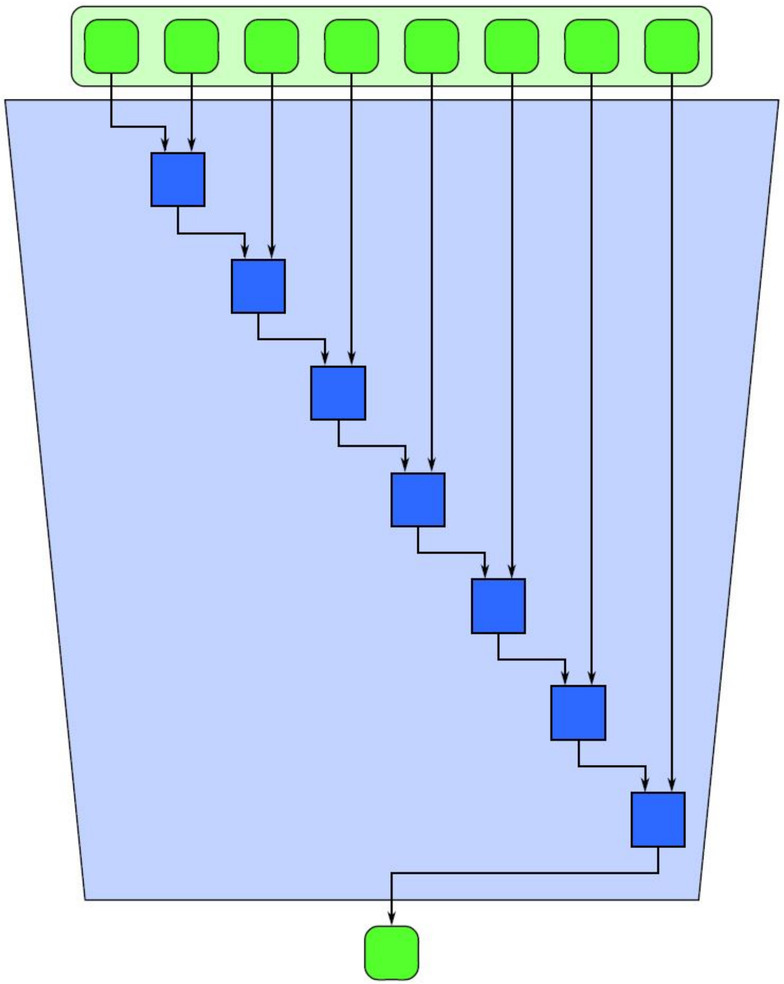
Serial Reduction



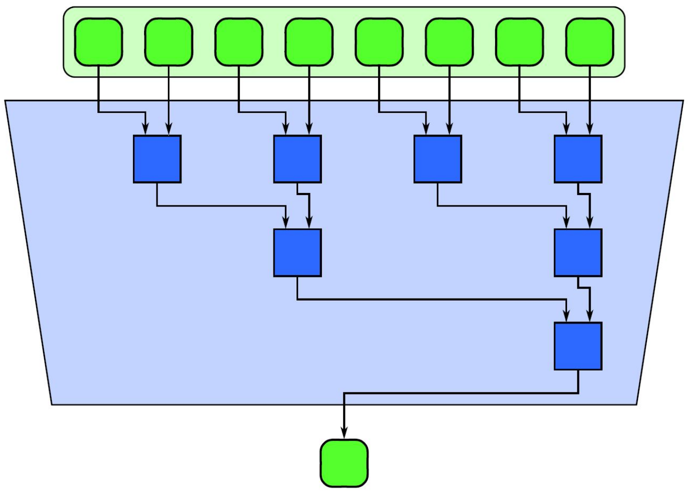
```
template<typename T>
T reduce(
    T (*f)(T,T),  // combiner function
    size_t n,  // number of elements in input array
    T a[],       // input array
    T identity // identity of combiner function
) {
    T accum = identity;
    for (size_t i = 0; i < n; ++i) {
        accum = f(accum, a[i]);
    }
    return accum;
}
```

Reduce

Serial Reduction



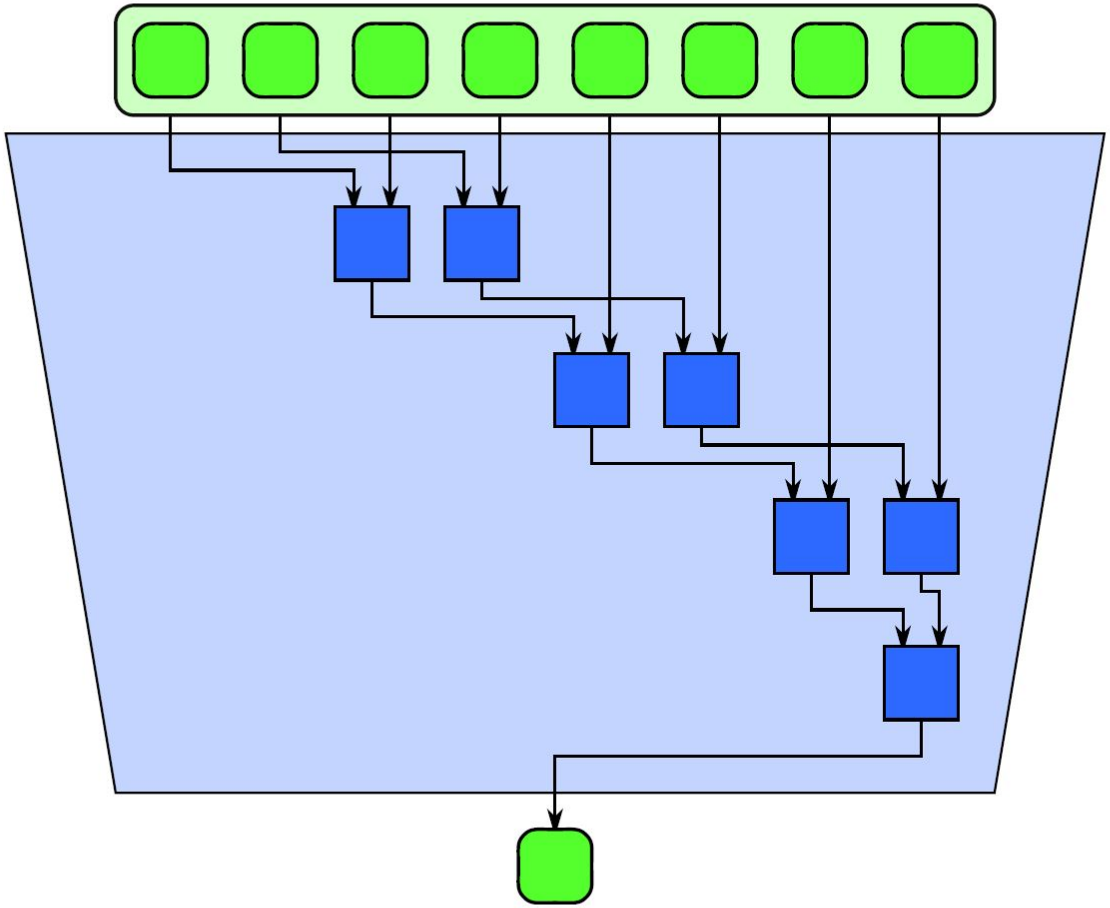
Parallel Reduction



Implementation later…

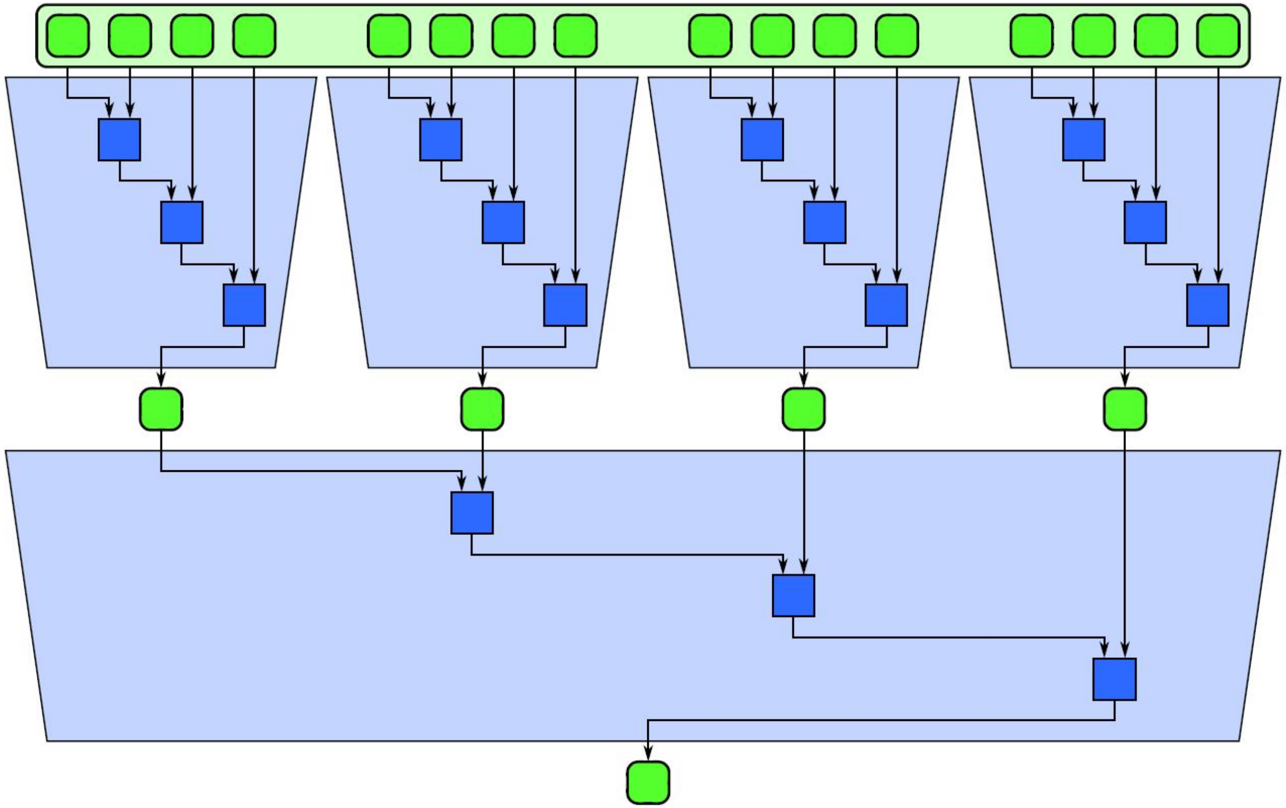
Reduce

- Vectorization

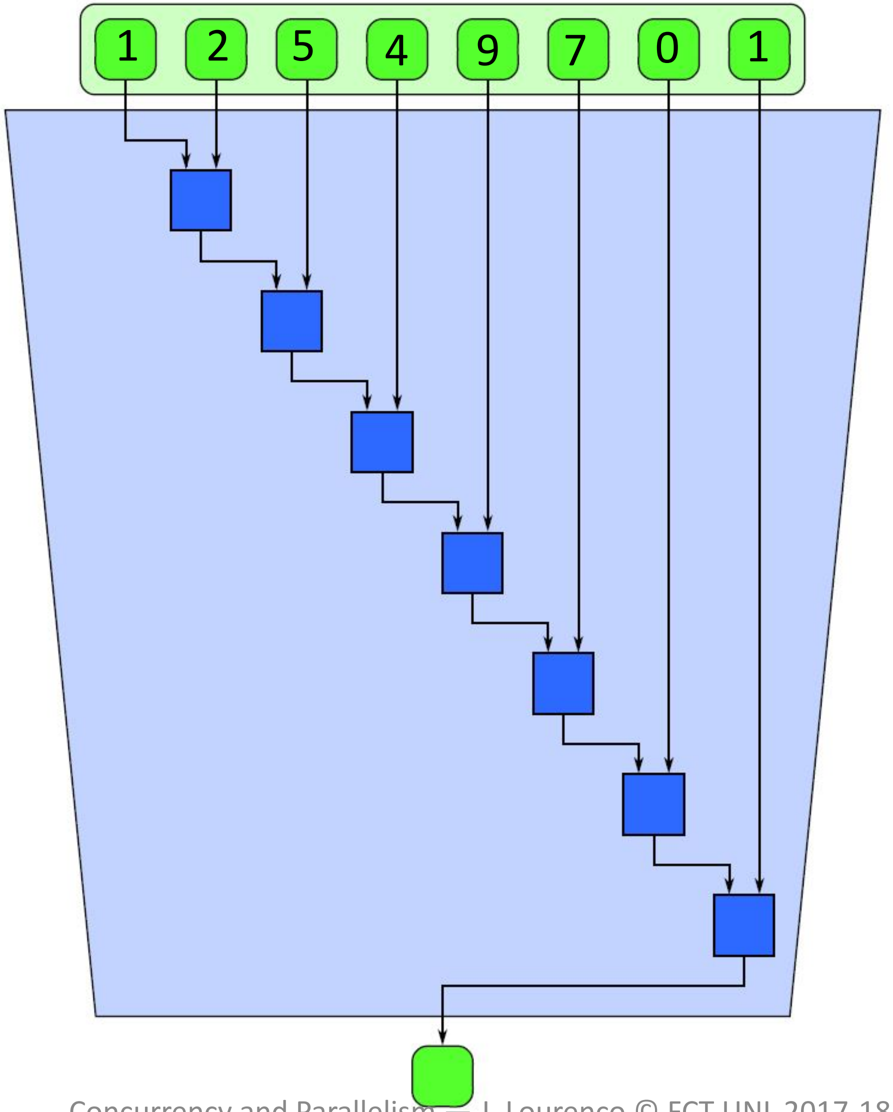


Reduce

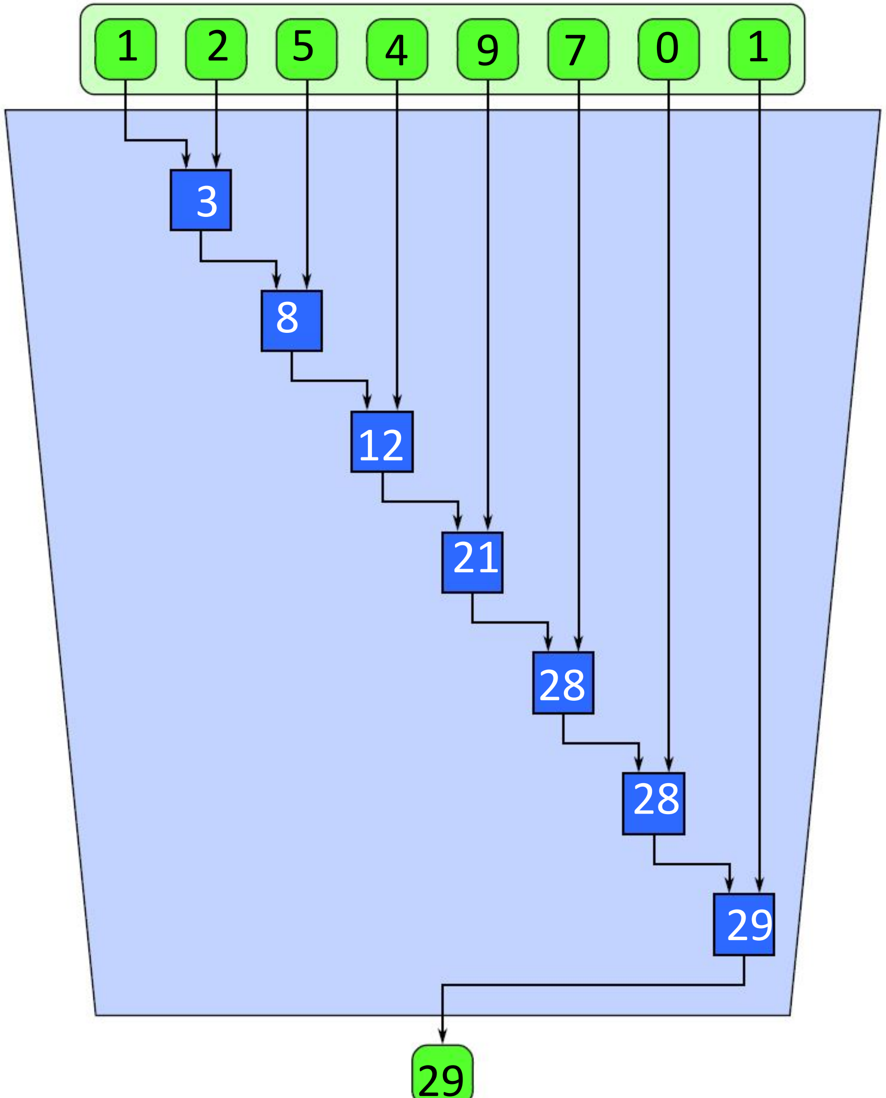
- **Tiling** is used to break chunks of work up for workers to reduce serially



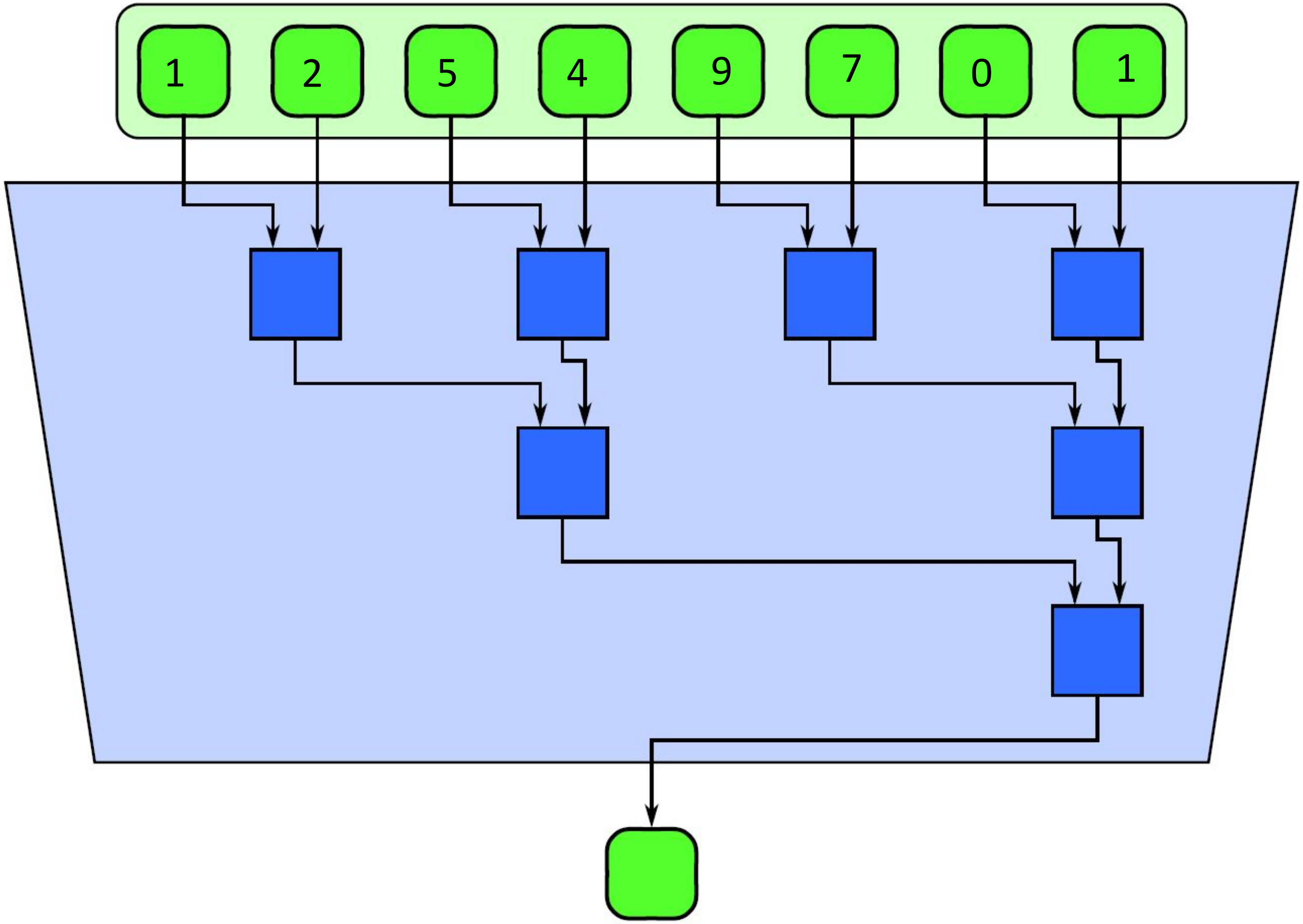
Reduce – Add Example



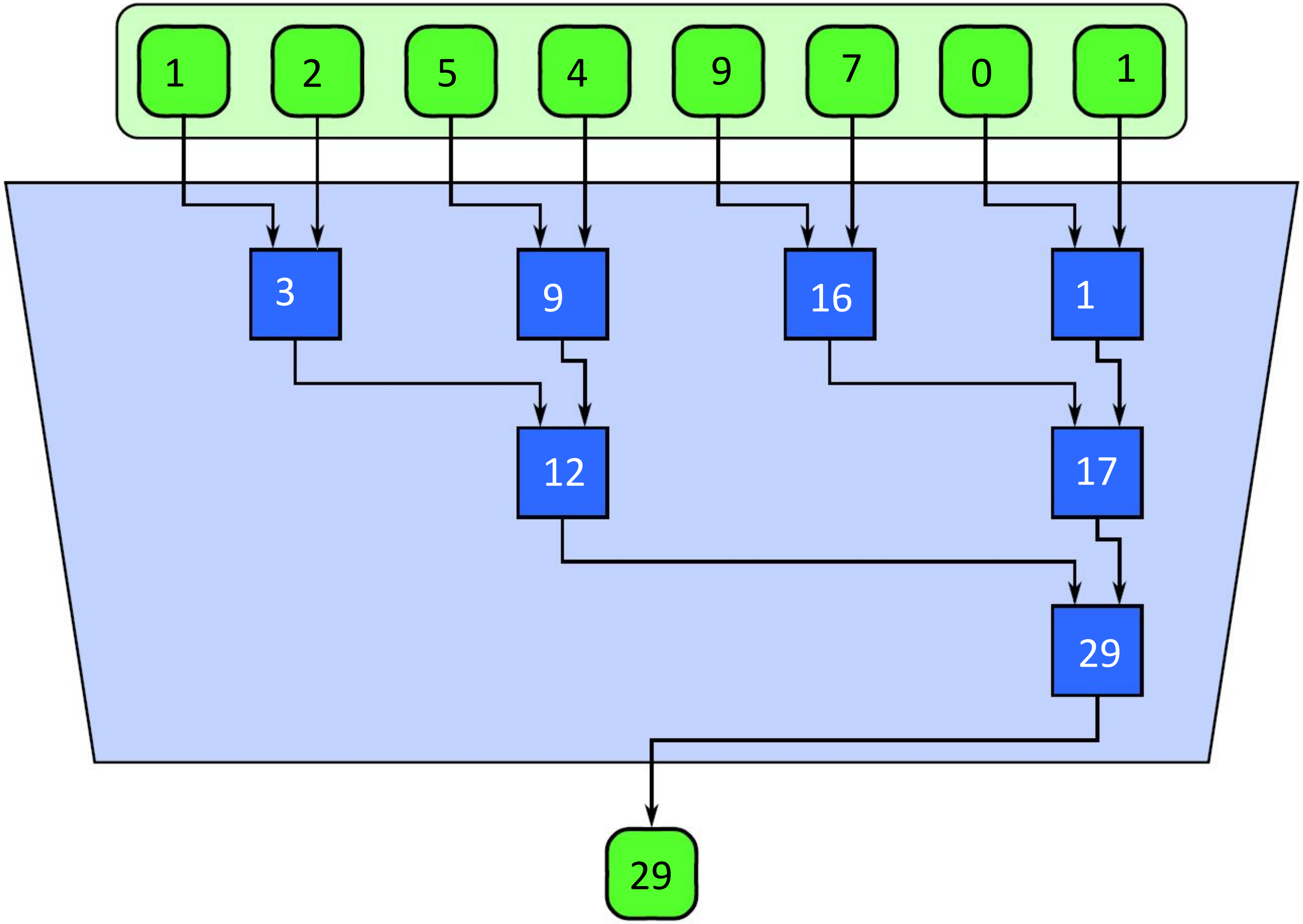
Reduce – Add Example



Reduce – Add Example

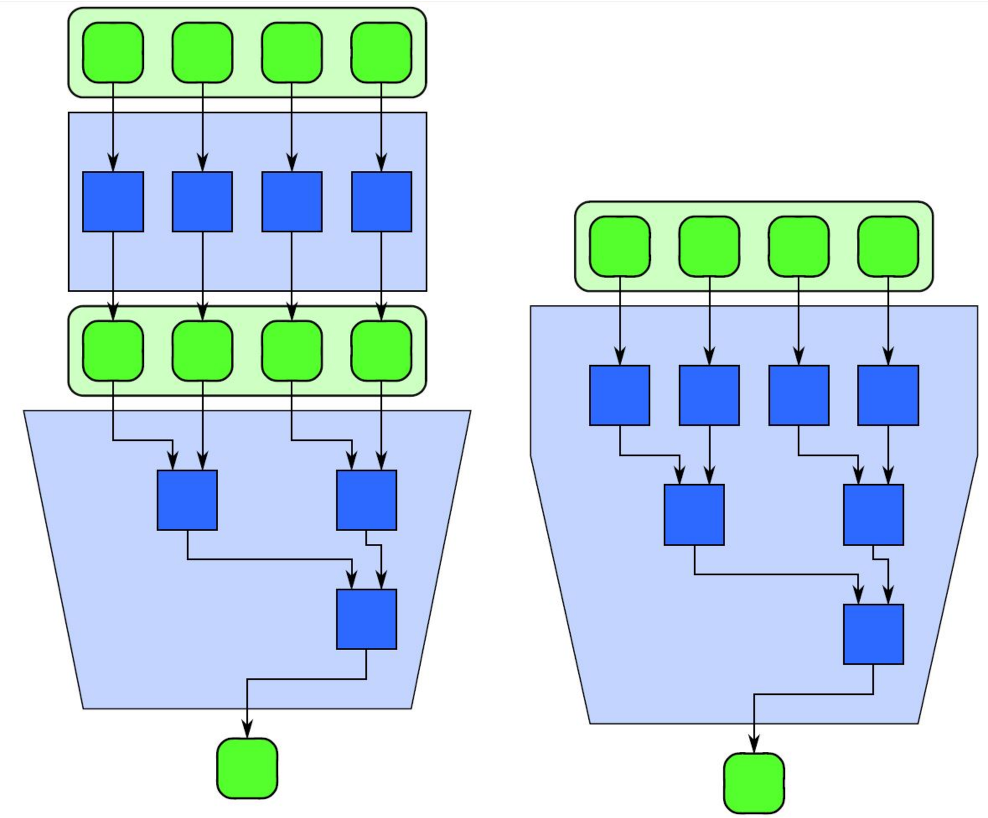


Reduce – Add Example



Reduce

- We can "fuse" the map and reduce patterns



Reduce

- Precision can become a problem with reductions on floating point data
- Different orderings of floating point data can change the reduction value

Reduce Example: Dot Product

- 2 vectors of same length
- Map (x) to multiply the components
- Then reduce with (+) to get the final answer

$$a \cdot b = \sum_{i=0}^{n-1} a_i b_i$$

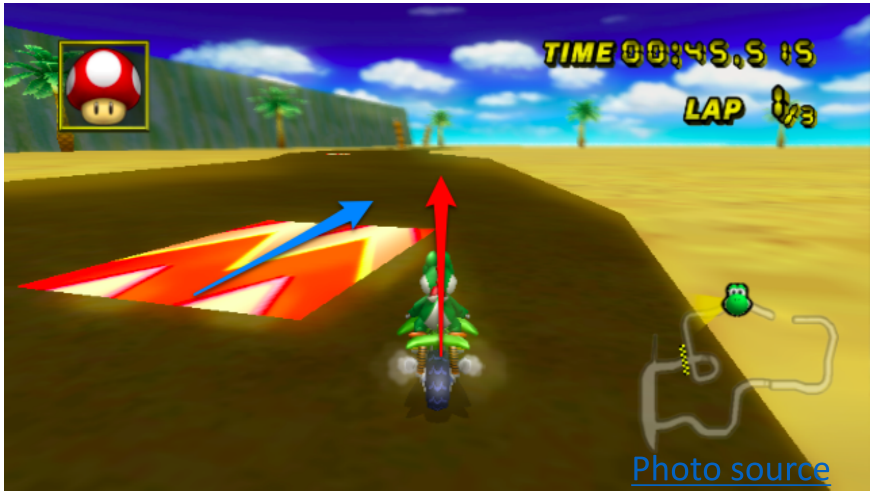
Also: $\vec{a} \cdot \vec{b} = |\vec{a}| \cos(\theta) |\vec{b}|$

Dot Product – Example Uses

- Essential operation in physics, graphics, video games,…
- Gaming analogy: in Mario Kart, there are "boost pads" on the ground that increase your speed
  - red vector is your speed (x and y direction)
  - blue vector is the orientation of the boost pad (x and y direction). Larger numbers are more power.

How much boost will you get? For the analogy, imagine the pad multiplies your speed:

- If you come in going 0, you'll get nothing
- If you cross the pad perpendicularly, you'll get 0 [just like the banana obliteration, it will give you 0x boost in the perpendicular direction]



$$Total = speed_x \cdot boost_x + speed_y \cdot boost_y$$

Ref: http://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/

Dot Product – Serial implem.

$$a \cdot b = \sum_{i=0}^{n-1} a_i b_i$$

```
float sprod(
    size_t n,
    const float a[],
    const float b[]
) {
    float res = 0.0f;
    for (size_t i = 0; i < n; i++) {
        res += a[i] * b[i];
    }
    return res;
}
```

Dot Product – Vectorization with SSE

```c
float sse_sprod(
    size_t n,
    const float a[],
    const float b[]
) {
    assert(0 == n % 4); // only works for N, a multiple of 4
    __m128 res, prd, ma, mb;
    res = _mm_setzero_ps();
    for (size_t i = 0; i < n; i += 4) {
        ma = _mm_loadu_ps(&a[i]); // load 4 elements from a
        mb = _mm_loadu_ps(&b[i]); // load 4 elements from b
        prd = _mm_mul_ps(ma,mb); // multiple 4 values elementwise
        res = _mm_add_ps(prd,res); // accumulate partial sums over 4-tuples
    }
    prd = _mm_setzero_ps();
    res = _mm_hadd_ps(res, prd); // horizontal addition
    res = _mm_hadd_ps(res, prd); // horizontal addition
    float tmp;
    _mm_store_ss(&tmp, res);
    return tmp;
}
```

$$a \cdot b = \sum_{i=0}^{n-1} a_i b_i$$

Dot Product – Cilk+ with Array Notation

$$a \cdot b = \sum_{i=0}^{n-1} a_i b_i$$

```
float cilkplus_sprod(
    size_t n,
    const float a[],
    const float b[]
) {
    return __sec_reduce_add(a[0:n] * b[0:n]);
}
```

Not implemented in gcc

Dot Product – Cilk+ with Explicit Tiling

$$a \cdot b = \sum_{i=0}^{n-1} a_i b_i$$

```
float cilkplus_sprod_tiled(
    size_t n,
    const float a[],
    const float b[]
) {
    size_t tilesize = 4096;
    cilk::reducer_opadd<float> res(0);
    cilk_for (size_t i = 0; i < n; i+=tilesize) {
        size_t m = std::min(tilesize,n-i);
        res += __sec_reduce_add(a[i:m] * b[i:m]);
    }
    return res.get_value();
}
```

Dot Product – OpenMP

$$a \cdot b = \sum_{i=0}^{n-1} a_i b_i$$

```
float openmp_sprod(
    size_t n,
    const float *a,
    const float *b
) {
    float res = 0.0f;
#pragma omp parallel for reduction(+:res)
    for (size_t i = 0; i < n; i++) {
        res += a[i] * b[i];
    }
    return res;
}
```

The END