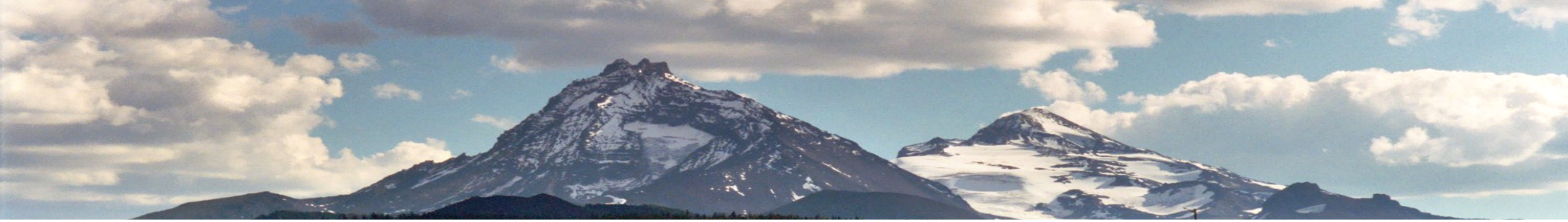


Collectives Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science

UNIVERSITY OF OREGON

Reduce

Summing an Array

```
long sum = 0;
for (int n = 0, n < size; ++n) {
    sum += arr[n];
}
```

Fully sequential

How do we do it in parallel?

Divide and conquer!

- Basic algorithm:
  - Split array into two parts
  - Sum each part (in parallel)
  - Return the sum of the two parts
- Why does this work?
- How can we generalize?

Summing an Array Recursively

```
long sum(int *arr, size_t size) {
    if (size == 0) return 0;
    else return arr[0] + sum(&arr[1], size-1);
}
```

Back to being sequential…

Right to Left Reduction

We can generalize the sum example!

- Work over a triple (T, R, h, z)
  - T is the type of elements in the array
  - R is the type we want to return
  - h is a function T*R→R…that is, it takes a T and an R and returns an R
  - z is an element of type R
- We apply h to the first element of the array, together with the reduction of the rest of the array
- Right to left – computes the sum over the rest of the array first

C++ Right Reduction

```
template<class T, class R, class H> R reduce(H h, R z, T* arr, size_t size)
{
    if (size == 0) return z;
    else return h(arr[0], reduce(h, z&arr[1], size-1));
}
```

Generalization!

```
    sum(arr, size) = reduce([](int n, long m) {return n + m}, 0, arr, size)
```

Note that we need the z: it is how we handle empty arrays.

What about the parallel sum?

We could make sum happen in parallel, can we do this in general?

```
reduce(h, z, [1,2,3,4,5], 4) = h(1, h(2, h(3, h(4, h(5,z)))))
```

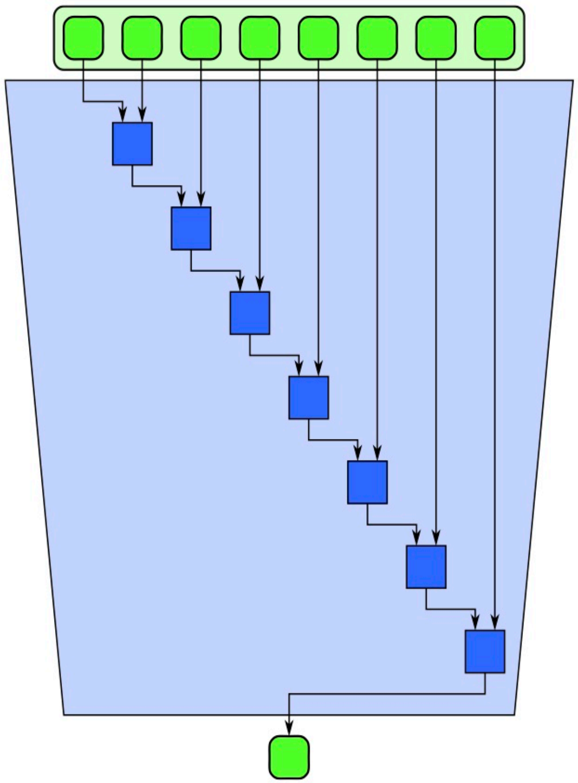
The parallel sum would be more like:

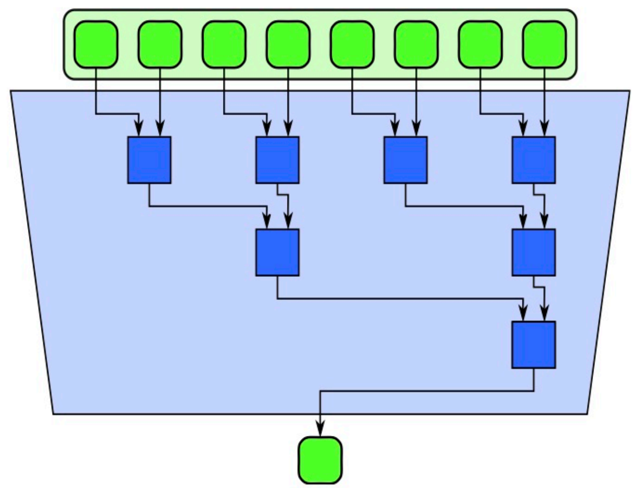
```
h(h(h(1, 2), h(3, 4)), h(5,z))
```

This might compute something totally different. It might not even have the same types (requires T = R).

What is going on?

- Right reduce is *list* structured
- A "parallel reduce" has to be *tree* structured





Sum Simple Algebra

- Associativity Property
  - $(x + y) + z = x + (y + z)$
  - Examples: addition, multiplication
- Commutative Property
  - $x + y = y + z$
  - Examples: addition, multiplication of integers
  - Counter Examples: multiplication of matrices

Monoids

A monoid is a triple (S, +, 0), where S is a set

- S is a set
- \+ is an operation on S (not necessarily addition)
- 0 is an element of S (not necessarily 1)

such that

- Associativity: for all x, y, z in S

  $(x + y) + z = x + (y + z)$
- Identity: for all x in S

  $0 + x = x + 0 = x$
- Not necessarily commutative

Many Many Monoids

- (Z, +, 0) where Z is the integers and +, 0 are the usual meaning of those
- (R, +, 0) where R is the reals
- (Z, *, 1), that is, multiplication of integers
- (Real Valued 2-2 matrics, matrix multiplication, the identity matrix)
- For any set S: ($\{f \mid f : S \rightarrow S\}$, function composition, the identity function)
- (Strings over an alphabet, string concatenation, empty string)
- ….

Many Many Monoids

- (Z, +, 0) where Z is the integers and +, 0 are the usual meaning of those
- (R, +, 0) where R is the reals
- (Z, *, 1), that is, multiplication of integers
- (Real Valued 2-2 matrics, matrix multiplication, the identity matrix)
- For any set S: ($\{f \mid f : S \rightarrow S\}$, function composition, the identity function)
- (Strings over an alphabet, string concatenation, empty string)
- ….

(If you have taken abstract algebra: every **group** is a monoid)

Arbitrary Order Reduction

Performance of Parallel Reduction

Looks like:

- Work: O(n)
- Span: O(n*log(n))

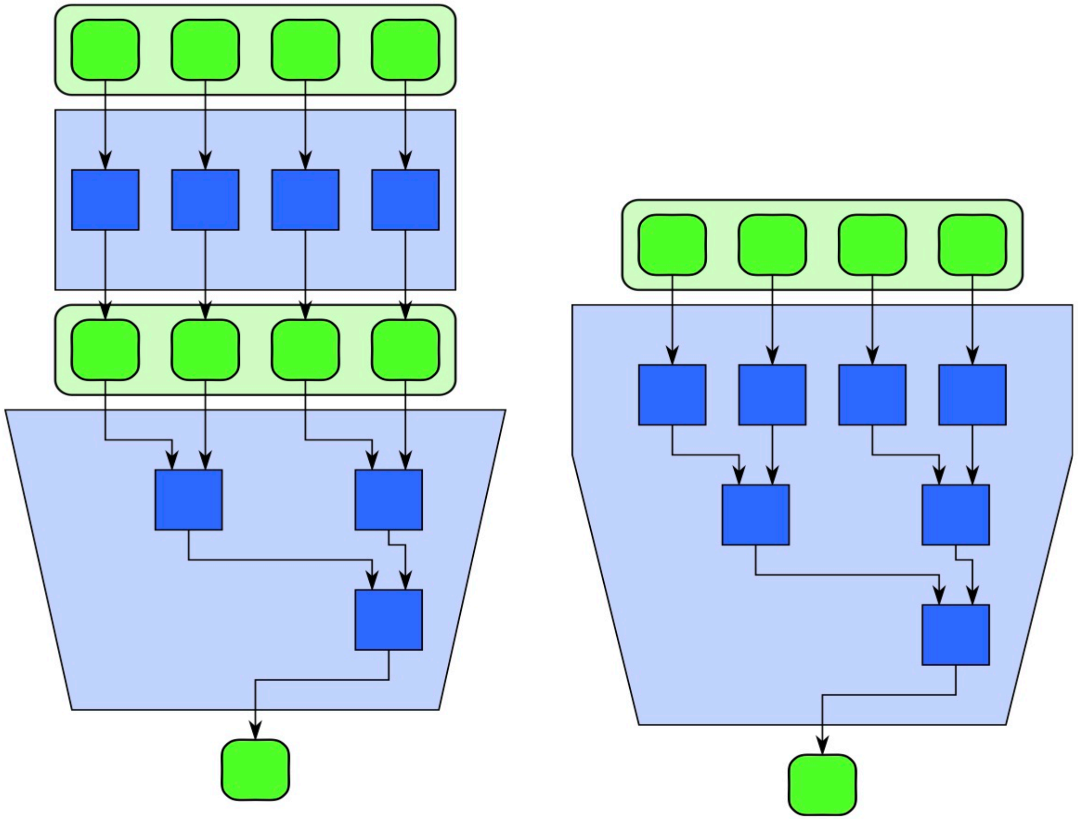
But slight nuance…

These numbers are not *the number of calls to worker function (+)* **not** the total time

We will come back to this!

Map/Reduce Fusion

- Map then fuse pattern very common
- Fuse them!



Example: Dot Product

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

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

Dot Product in TBB

```
float tbb_sprod(
   size_t n,
   const float *a,
   const float *b
) {
   return tbb::parallel_reduce(
      tbb::blocked_range<size_t>(0,n),
      float(0),
      [=]( // lambda expression
         tbb::blocked_range<size_t>& r,
         float in
      ) {
         return std::inner_product(
            a+r.begin(), a+r.end(),
            b+r.begin(), in );
      },
      std::plus<float>()
   );
}
```

Merge Sort as a reduction

Define the monoid (S, <>, []) where

- S is the set of in order vectors over some type
- <> is the merge operation: [1,3,5,7] <> [2,6,15] = [1,2,3,5,6,7,15]
- [] is the empty list

We can sort an array via a pair of a map and a reduce

- Map each element into a vector containing just that element
- Reduce using the monoid above

How fast is this?

Right Biased Sort

Start with [14,3,4,8,7,52,1]

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

Reduce:

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

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

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

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

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

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

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

Right Biased Sort Cont

- How long did that take?
- Well we did O(n) merges…but each one took O(n) time
- $O(n^2)$
- We wanted merge sort, but instead we got insertion sort!

Tree Shape Sort

Start with [14,3,4,8,7,52,1]

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

Reduce:

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

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

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

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

Tree Shaped Sort Performance

- Even if we only had a single processor this is better
  - We do O(log n) merges
  - Each one is O(n)
  - So O(n*log(n))
- But opportunity for parallelism is not so great
  - O(n) assuming sequential merge
- We will explore parallel merging later in the class
- Takeaway: the shape of reduction matters!

Shape Matters

- Often tree based reductions are slower
- In fact, any reduction can be rewritten using monoids…but the cost of the operation **won't be a constant**
- Parallel reduction is suitable for problems where you have
  - Cost free associativity (like adding number)
  - Or tree based reduction is better (like merging)

Scan

Prefix Scans

What if instead of summing a list we wanted to compute all the **prefix sums**?

```
int * results = malloc (sizeof(int)*size);
int temp = 0;
for (int n = 0, n < size, ++n) {
    temp += arr[n];
    results[n] = temp;
}
```

[1,2,3,4,5] → [1,3,6,10,15]

Semi-Groups

A pair (S, +) where

S is a set

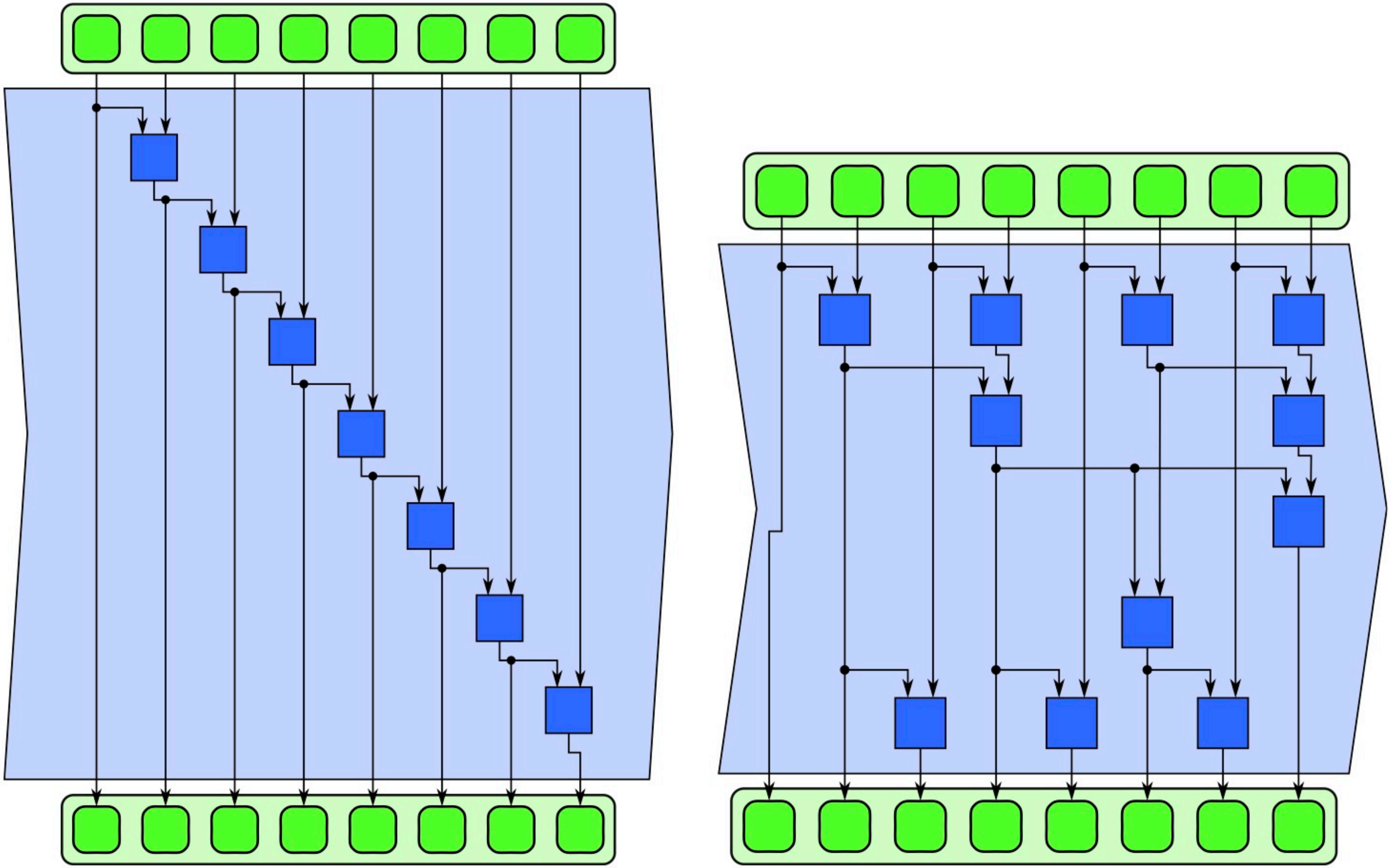
+ is a binary operation on S

- Is a semi-group if + is associative
- Unlike monoids, we don't require identity
- Every monoid is a semi-group, but not vice versa!

Scans

- Scan operation generalizes prefix sum
- Parameterized by a semi-group
- Scan is often the trick to turn algorithm that seems to have **sequential data dependencies** into a parallel algorithm

Scans



Binary Addition

Given two bit vectors of length n, compute their sum

$1010 + 0111 = 10001$

Standard algorithm you learned in elementary school

- Add from least significant digit to most significant digit
- Keep track of carrying
- Very sequential: O(n)

Can we do better?

UNIVERSITY OF OREGON

Binary Addition of Single Bits

Single bit addition

$0 + 0 \rightarrow 00$

$0 + 1 \rightarrow 01$

$1 + 0 \rightarrow 01$

$1 + 1 \rightarrow 10$

Note that we produce carries.

Binary Addition of Single Bits

Single bit addition

$0 + 0 \rightarrow 00$

$0 + 1 \rightarrow 01$

$1 + 0 \rightarrow 01$

$1 + 1 \rightarrow 10$

Clever Carrying

Single bit addition assumes we don't have a carry

What if we do?

Single bit addition assuming a carry

$0 + 0 \rightarrow 01$

$0 + 1 \rightarrow 10$

$1 + 0 \rightarrow 10$

$1 + 1 \rightarrow 11$

Note that we produce carries.

Clever Carrying Part 2

Single bit addition behavior depends on if we had a carry!

Single bit with and without carries

- $0 + 0 \rightarrow 00, 01$
- $0 + 1 \rightarrow 01, 10$
- $1 + 0 \rightarrow 01, 10$
- $1 + 1 \rightarrow 10, 11$

Clever Carrying part 3

We can classify these pairs in terms of

- "generates a carry" (G)
- "preserves the carry given to it, but does not produce one" (P)
- "does not produce a carry" (N)

Single bit carrying behavior:

$0 + 0 \rightarrow N$

$0 + 1 \rightarrow P$

$1 + 0 \rightarrow P$

$1 + 1 \rightarrow G$

The Carrying Semi-Group

The set {P, N, G} forms a semi-group under the operation <>

- P <> x → x
- G <> _ → G
- N <> _ → N

Here _ means “any element”

Multi Bit Carrying

We can expand this to multiple bits

11 + 01 always generates a carry (G)

01 + 01 never generates a carry (N)

01 + 10 preserves a carry (P)

AB + CD works in the following way:

A + B → G means AB + CD → G

A + B → N means AB + CN → N

A + B → P means AB + CD does the same thing as ever B + D

AB + CD

Computing the Carry

If you want to compute the carry behavior of the sum of binary numbers

1010100101 + 0101011100

We can do it first computing the carries pairwise

PPPPPPPGNP

And then using the semi-group structure

((P <> P) <> (P <> P)) <> (((P <> P) <> (P <> G)) <> (N <> P))

= (P <> P) <> ((P <> G) <> N) = P <> (G <> N) = P <> G

= G

Thus we know this example produces a carry.

Look Ahead Carry Addition

- We can take this idea to produce an algorithm for binary addition
- We write all numbers "backwards" (least significant digit to most significant)
- Use map to compute a {P, N, G} array
- Scan over this array with <> producing an array C
- Use map to compute the result

```
parallel_for(int n = 1, n < len, ++n) {
    if (R[n-1] == G) R[n] = A[n]^B[n];
    else R[n] = ! (A[n]^B[n]);
}
```

Scan Performance

- Sequential scan calls the function O(n) times
- Parallel scan is calls the function
  - Span: O(log n)
  - Work: O(n*log n)
- Unlike other patterns we have to do extra work to be parallel
- But, scan is broadly applicable. If you can't make an algorithm parallel, scan is often a good place to look