CS 6068
Parallel Computing
Fall 2014
Lecture 2 – Sept 8
Communication Patterns

Prof. Fred Annexstein   @proffreda
fred.annexstein@uc.edu
Office Hours: 11-12pm MW or by appointment
Tel: 513-556-1807
Meeting: Mondays 6:00-8:50PM Baldwin 645

Lecture 2: Plan

- Review – Programming Model
- Very Common Communication Patterns

  Map, Scatter, Gather, Stencil

- Work and Step Complexity Analysis

  Watch for falling trees and logs….

- Basic Parallel Operations and Algorithms

  Reduce, Scan, Histogram

- Expressive Language Descriptions of Algorithm

  PRAM and Recursion

Review Programming Model

GPU/CUDA Model:

- Arrays of lightweight parallel thread deployed by invoking a kernel function
- kernel calls specify size blockDim and gridDim
- threadIdx and blockIdx are used for indexing of threads
- Memory management, Global memory, device and host arrays and faster shared memory

The Map Operator

- A basic function on arrays that takes another function as argument
- map applies function arg as operation on each array element
- Built natively into python and other functional PLs

```
>>> map(add1, [1, 2 ,3])
        [2, 3, 4]
```

- In CUDA a map is code that assigns each thread in an array the same task on its own local memory

Scatter and Gather Communication Operations

- Scatter is an data communication operation that moves or permutes data based on an array of location addresses

- Gather is a operation that specifies that each thread does a computation using data from multiple locations

Stencil Communication Operation

- Given a fixed sized stencil window to cover and centered on every array element
- Applications often use small 2-D stencil applied to large 2D image array.
- Common class of applications related to convolution filtering

HW#2 Gaussian Blurring as Map Application

Gaussian blurring is defined as exponentially weighted averaging, typically using square stencil at every pixel.

In HW#2 you will write a kernel that applies a Gaussian filter that provides a weighted averaging of each color channel. One kernel call per color. Also allocate memory and write a kernel to separate colors. Set appropriate grid and block sizes.

Work and Step Complexity

- *Work complexity* is defined as the total number of operations executed by a computation (can assume only one processor).

- *Step (or depth) complexity* is defined as the longest chain of sequential dependencies in the computation (can assume infinite number of parallel processors).

Complexity Example

- Consider, for example, summing *16* numbers using a balanced binary tree.
- The work required by this computation is *15* operations (the 15 additions).
- The steps required is *4* operations since the longest chain of dependencies is the depth of the summation tree--the sums need to be calculated starting at the leaves and going down one level at a time.
- In general summing *n* numbers using a balanced tree pattern requires *n-1* work and *log n steps* .

A Recursive Language Description using Python

```
def qsort(list):
    if list == []:
        return []
    else:
        pivot = list[0]
        lesser = [x for x in list[1:] if x < pivot])
        greater = [x for x in list[1:] if x >= pivot])
    return qsort(lesser) + [pivot] + qsort(greater)
```

[Exercise: write a python function for summing.]

Quicksort

- Quicksort written in this manner is not hard to parallelize.
- We can execute the two recursive calls in parallel
- And, all the pivot comparisons can be done in parallel.
- Complete analysis is done by considering the recursion tree using recurrence relations. (Come back to this later.)

The Reduce Operation

- sum_reduce(x) is a function that takes a list and returns the sum of the elements.
- Described as a simple recursive function and related to the binary tree reduction seen above.

```
def sum_reduce(x):
   if len(x)== 1:
        return
   else:
      sumleft = sum_reduce (x[0:len(x)/2] )
      sumright = sum_reduce (x[len(x)/2 +1:])
      return    sumleft + sumright
```

Work and Step Complexity of Reduce using Recurrence Relations

- $W(n) = 2 * W(n/2) + 1; W(1) = 0$

- $S(n) = S(n/2) + 1 ; S(1) = 0$

- Next: CUDA Coding Reduce

```
__global__ void global_reduce_kernel(float * d_out, float * d_in)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid  = threadIdx.x;

    // do reduction in global mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            d_in[myId] += d_in[myId + s];
        }
        __syncthreads();        // make sure all adds at one stage are done!
    }

    // only thread 0 writes result for this block back to global mem
    if (tid == 0)
    {
        d_out[blockIdx.x] = d_in[myId];
    }
}
```

```
__global__ void shmem_reduce_kernel(float * d_out, const float * d_in)
{
    // sdata is allocated in the kernel call: 3rd arg to <<<b, t, shmem>>>
    extern __shared__ float sdata[];

    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid  = threadIdx.x;

    // load shared mem from global mem
    sdata[tid] = d_in[myId];
    __syncthreads();            // make sure entire block is loaded!

    // do reduction in shared mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();        // make sure all adds at one stage are done!
    }
```

The Scan Operation

- One of most important data parallel primitives
- Inclusive and Exclusive versions of running sum

1 2 3 4 5 6 7 8

Coding the Scan:

Iterative and Recursive versions

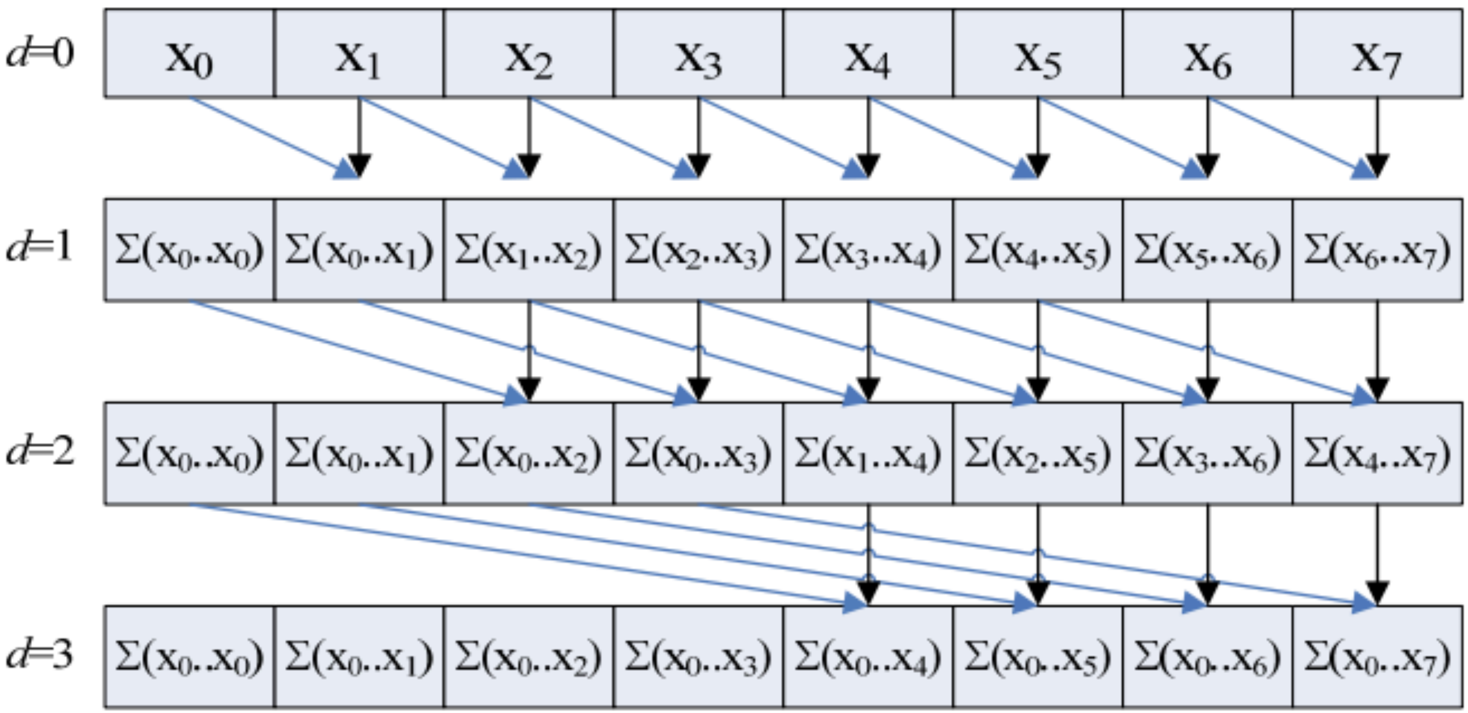
PRAM Algorithms of Hillis-Steele and Blelloch

Coding the Scan Operation: Recursive

```
def scan(add,x):
    if len(x)== 1:
        return x
    else:
        firsthalf = scan(add, x[0:len(x)/2])
        secondhalf = scan(add, x[len(x)/2:])
        s = map(add(firsthalf[-1]), secondhalf)
        res = firsthalf + s
        return res
```

Hillis and Steele PRAM version

(PRAM is like CUDA Block with unlimited threads)



Hillis and Steele PRAM version

For i= 0 to log n steps:

thread x adds its current held value

// sum of 2^i previous //

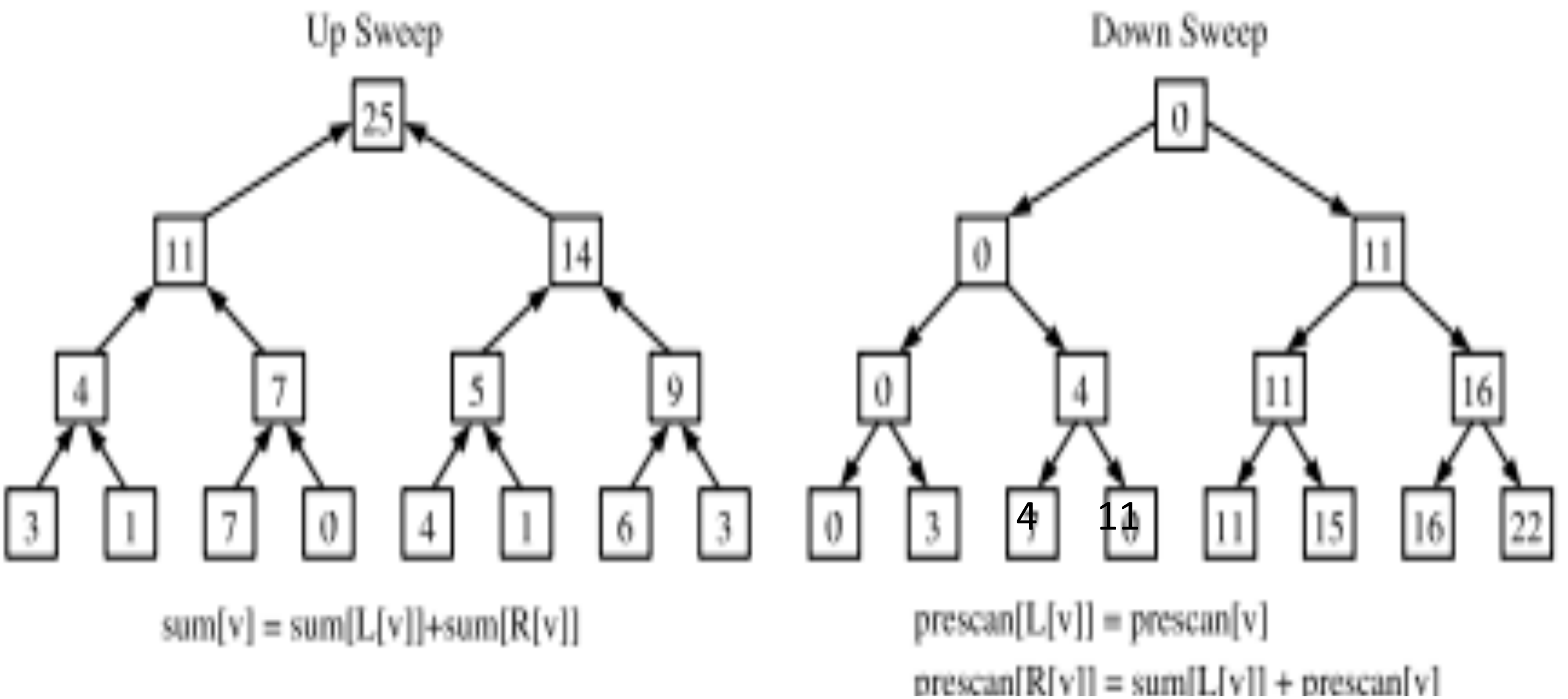
to

value held by thread x - (2^i)

// thus doubling the length of sum//

Review the work and step complexity of this.....

Blelloch PRAM version



```
procedure down-sweep(A)
  a[n − 1] ← 0                                  % Set the identity
  for d from (lg n) − 1 downto 0
    in parallel for i from 0 to n − 1 by 2^(d+1)
      t ← a[i + 2^d − 1]                        % Save in temporary
      a[i + 2^d − 1] ← a[i + 2^(d+1) − 1]       % Set left child
      a[i + 2^(d+1) − 1] ← t + a[i + 2^(d+1) − 1] % Set right child
```

	Step	Vector in Memory							
	0	[$\boxed{3}$	$\boxed{1}$	$\boxed{7}$	$\boxed{0}$	$\boxed{4}$	$\boxed{1}$	$\boxed{6}$	$\boxed{3}$]
up	1	[3	$\boxed{4}$	7	$\boxed{7}$	4	$\boxed{5}$	6	$\boxed{9}$]
	2	[3	4	7	$\boxed{11}$	4	5	6	$\boxed{14}$]
	3	[3	4	7	11	4	5	6	$\boxed{25}$]
clear	4	[3	4	7	11	4	5	6	$\boxed{0}$]
down	5	[3	4	7	$\boxed{0}$	4	5	6	$\boxed{11}$]
	6	[3	$\boxed{0}$	7	$\boxed{4}$	4	$\boxed{11}$	6	$\boxed{16}$]
	7	[$\boxed{0}$	$\boxed{3}$	$\boxed{4}$	$\boxed{11}$	$\boxed{11}$	$\boxed{15}$	$\boxed{16}$	$\boxed{22}$]

Comparing PRAM Scan Algorithms

- Hillis-Steele

  Work = $O(n \log n)$ Steps = $\log n$

- Blelloch

  Work = $O(n)$ Steps = $2 \log n$

Work efficient versus step efficient – when to choose?

Histograms / Data Binning

- Sequential algorithm

Loop through all data items

compute appropriate bin for each item

increment bin value +1

- Parallel algorithm

Each Parallel Design must deal with race conditions and performance bottlenecks.

Use atomics?

Use reduce or scan?

Local bins for each thread

Looking Ahead
HW#3 Tone Mapping

- Use of reduce, scan, and histogram

To map pixel intensities via tone mapping.

Start with array of brightness values

Compute minimum and maximum (reduce)

Compute a histogram based on these values