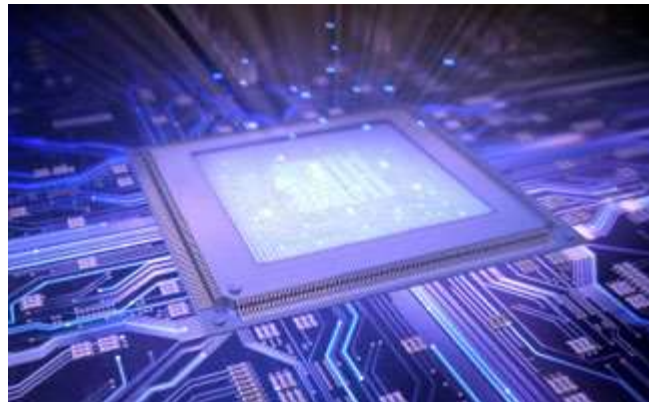




CSCI-UA.0480-003
Parallel Computing

Lecture 4: Parallel Software: Basics

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
<http://www.mzahran.com>



The burden is on software

- From now on...
 - In shared memory programs:
 - Start a single process and fork **threads**.
 - Threads carry out tasks.
 - In distributed memory programs:
 - Start multiple **processes**.
 - Processes carry out tasks.

SPMD - single program multiple data

- A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```
if (I'm thread/process i)
    do this;
else
    do that;
```



Writing Parallel Programs

1. **Divide** the work among the processes/threads
 - (a) so each process/thread gets roughly the same amount of work
 - (b) and communication is minimized.
2. Arrange for the processes/threads to **synchronize**.
3. Arrange for **communication** among processes/threads.

```
double x[n], y[n];  
...  
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

Shared Memory Systems

Shared Memory

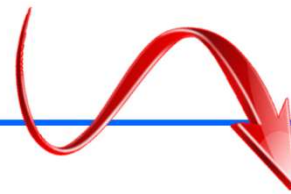
- **Dynamic threads**
 - Master thread waits for work, forks new threads, and when threads are done, they terminate
 - + Efficient use of resources
 - thread creation and termination is time consuming
- **Static threads**
 - Pool of threads created and are allocated work, but do not terminate until cleanup.
 - + Better performance
 - potential waste of system resources

Nondeterminism

```
...  
printf ( "Thread %d > my_val = %d\n" ,  
        my_rank , my_x );  
...
```



Thread 1 > my_val = 19
Thread 0 > my_val = 7



Thread 0 > my_val = 7
Thread 1 > my_val = 19

Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (**mutex**, **semaphore**, ...)

```
my_val = Compute_val ( my_rank ) ;  
Lock(&add_my_val_lock ) ;  
x += my_val ;  
Unlock(&add_my_val_lock ) ;
```


Important!!

What is the relationship between cache coherence and nondeterminism?
Isn't cache coherence enough to ensure determinism?

Busy-waiting

```
ok_for_1 = false;
my_val = Compute_val ( my_rank );
if ( my_rank == 1)
    while ( ! ok_for_1 ); /* Busy-wait loop */
x += my_val ; /* Critical section */
if ( my_rank == 0)
    ok_for_1 = true ; /* Let thread 1 update x */
```

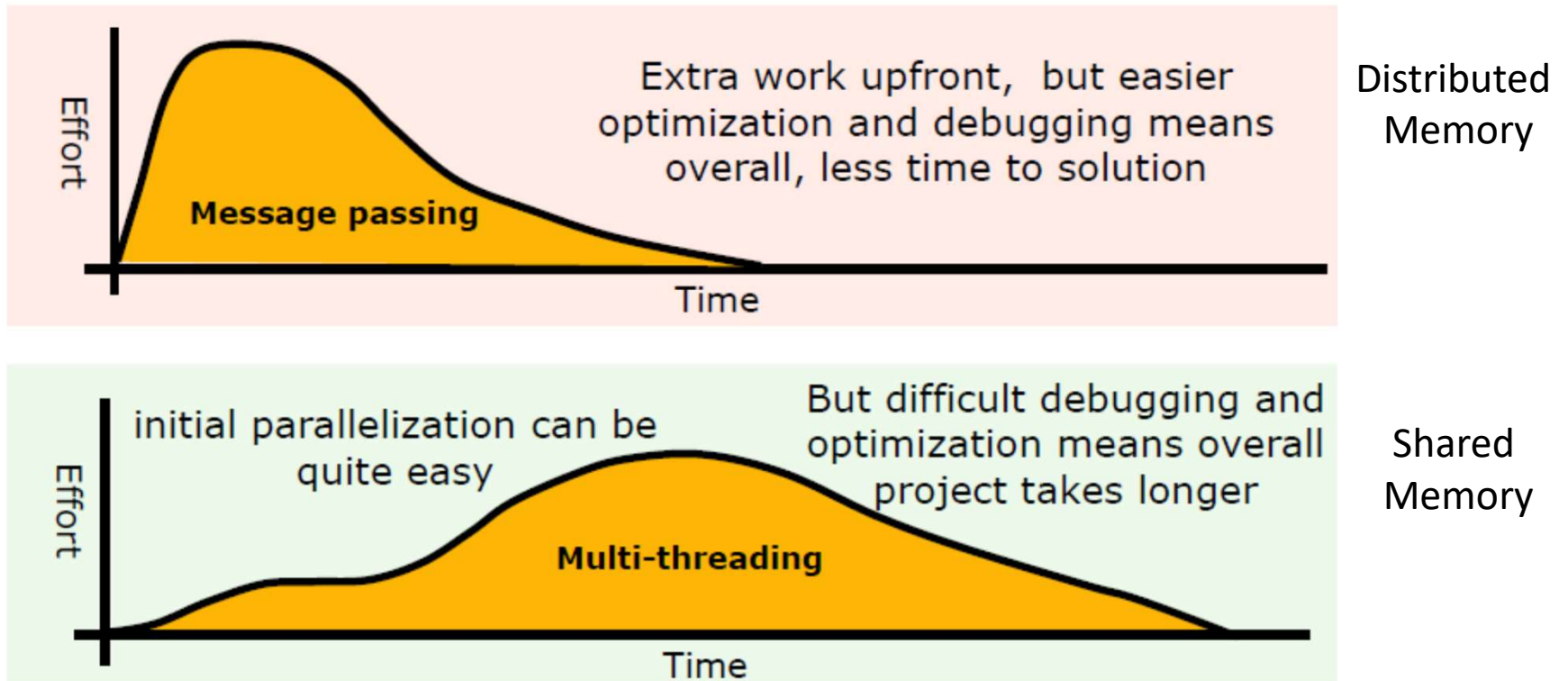
What is wrong with that?

Distributed Memory Systems

Distributed Memory: message-passing

```
char message [100] ;  
...  
my_rank = Get_rank();  
if ( my_rank == 1) {  
    sprintf ( message , "Greetings from process 1" ) ;  
    Send ( message , MSG_CHAR , 100 , 0 ) ;  
} else if ( my_rank == 0) {  
    Receive ( message , MSG_CHAR , 100 , 1 ) ;  
    printf ( "Process 0 > Received: %s\n" , message ) ;  
}
```

How do shared-memory and distributed-memory compare in terms of programmer's effort?



Source: "Many Core Processors ... Opportunities and Challenges" by Tim Mattson

We want to write a parallel program ... Now what?

- We have a serial program.
- How to parallelize it?
- We know that we need to divide work, ensure load balancing, manage synchronization, and reduce communication! → **Nice! How to do that?**
- Unfortunately: there is no mechanical process.
- **Ian Foster** has some nice framework.

Foster's methodology (The PCAM Methodology)

1. **Partitioning**: divide the computation to be performed and the data operated on by the computation into small tasks.

The focus here should be on identifying **tasks that can be executed in parallel**.

This step brings out the parallelism in the algorithm

Foster's methodology (The PCAM Methodology)

2. **Communication:** determine what communication needs to be carried out among the tasks identified in the previous step.



Foster's methodology (The PCAM Methodology)

3. **Agglomeration or aggregation:** combine tasks and communications identified in the first step into larger tasks.

For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

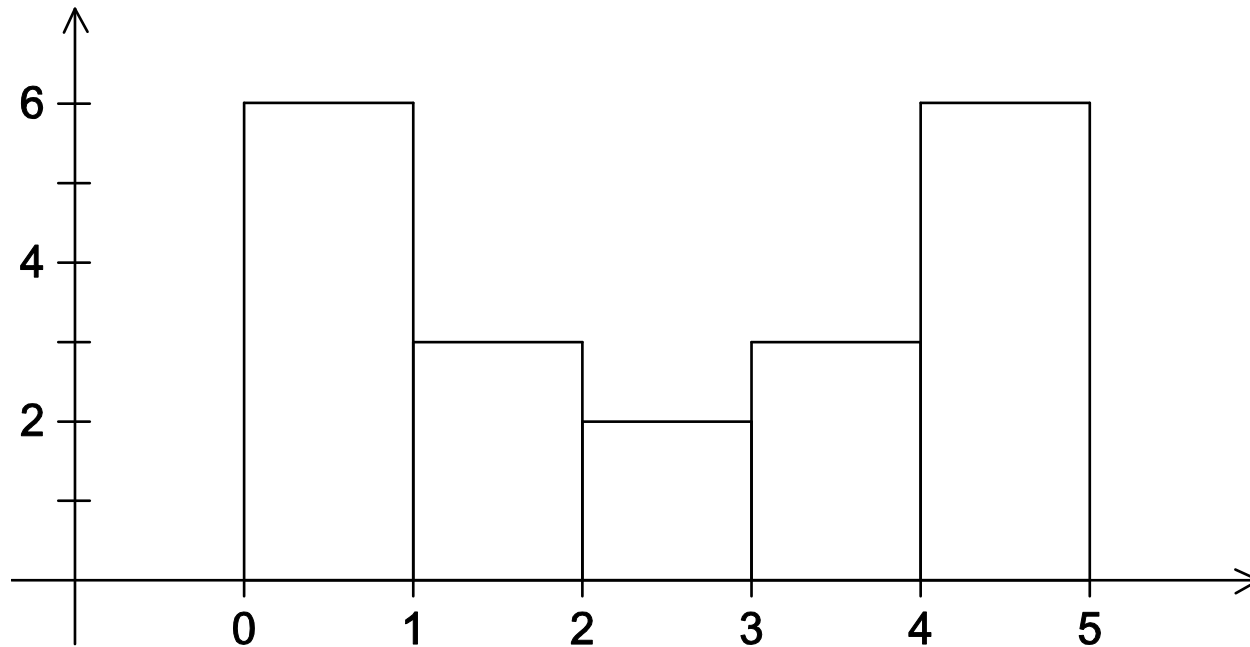
Foster's methodology (The PCAM Methodology)

4. **Mapping**: assign the composite tasks identified in the previous step to processes/threads.

This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Example - histogram

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



Serial program - input

1. The number of measurements: `data_count`
2. An array of `data_count` floats: `data`
3. The minimum value for the bin containing the smallest values: `min_meas`
4. The maximum value for the bin containing the largest values: `max_meas`
5. The number of bins: `bin_count`

- Data[0] = 1.3
- Data[1] = 2.9
- Data[2] = 0.4
- Data[3] = 0.3
- Data[4] = 1.3
- Data[5] = 4.4
- Data[6] = 1.7
- Data[7] = 0.4
- Data[8] = 3.2
- Data[9] = 0.3
- Data[10] = 4.9
- Data[11] = 2.4
- Data[12] = 3.1
- Data[13] = 4.4
- Data[14] = 3.9,
- Data[15] = 0.4
- Data[16] = 4.2
- Data[17] = 4.5
- Data[18] = 4.9
- Data[19] = 0.9

data_count = 20

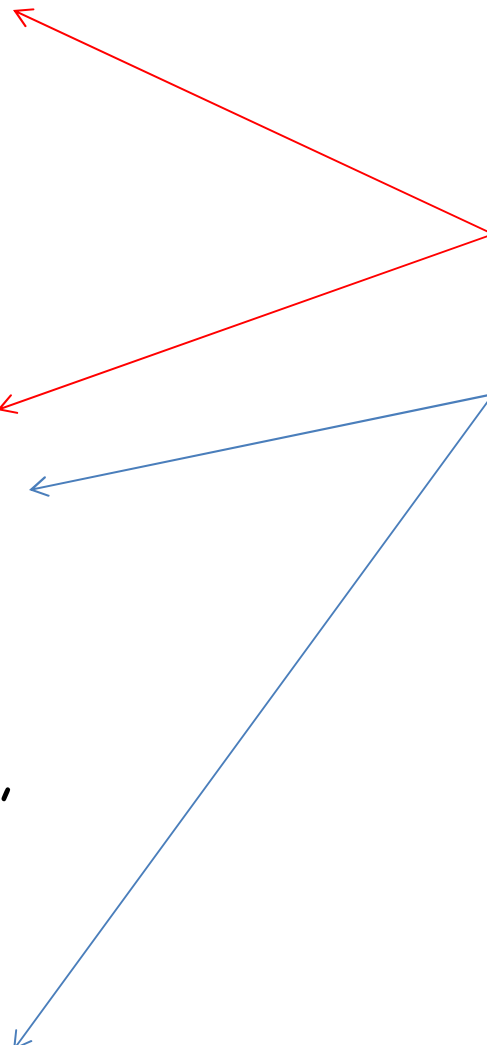
- Data[0] = 1.3
- Data[1] = 2.9
- Data[2] = 0.4
- Data[3] = 0.3
- Data[4] = 1.3
- Data[5] = 4.4
- Data[6] = 1.7
- Data[7] = 0.4
- Data[8] = 3.2
- Data[9] = 0.3
- Data[10] = 4.9
- Data[11] = 2.4
- Data[12] = 3.1
- Data[13] = 4.4
- Data[14] = 3.9,
- Data[15] = 0.4
- Data[16] = 4.2
- Data[17] = 4.5
- Data[18] = 4.9
- Data[19] = 0.9

data_count = 20

min_meas = 0.3

max_meas = 4.9

bin_count = 5



Serial program - output

1. **bin_maxes** : an array of bin_count floats → store the upper bound of each bin
2. **bin_counts** : an array of bin_count ints → stores the number of elements in each bin

- Data[0] = 1.3
- Data[1] = 2.9
- Data[2] = 0.4
- Data[3] = 0.3
- Data[4] = 1.3
- Data[5] = 4.4
- Data[6] = 1.7
- Data[7] = 0.4
- Data[8] = 3.2
- Data[9] = 0.3
- Data[10] = 4.9
- Data[11] = 2.4
- Data[12] = 3.1
- Data[13] = 4.4
- Data[14] = 3.9,
- Data[15] = 0.4
- Data[16] = 4.2
- Data[17] = 4.5
- Data[18] = 4.9
- Data[19] = 0.9

bin_maxes[0] = 0.9

bin_maxes[1] = 1.7

bin_maxes[2] = 2.9

bin_maxes[3] = 3.9

bin_maxes[4] = 4.9

bin_counts[0] = 6

bin_counts[1] = 3

bin_counts[2] = 2

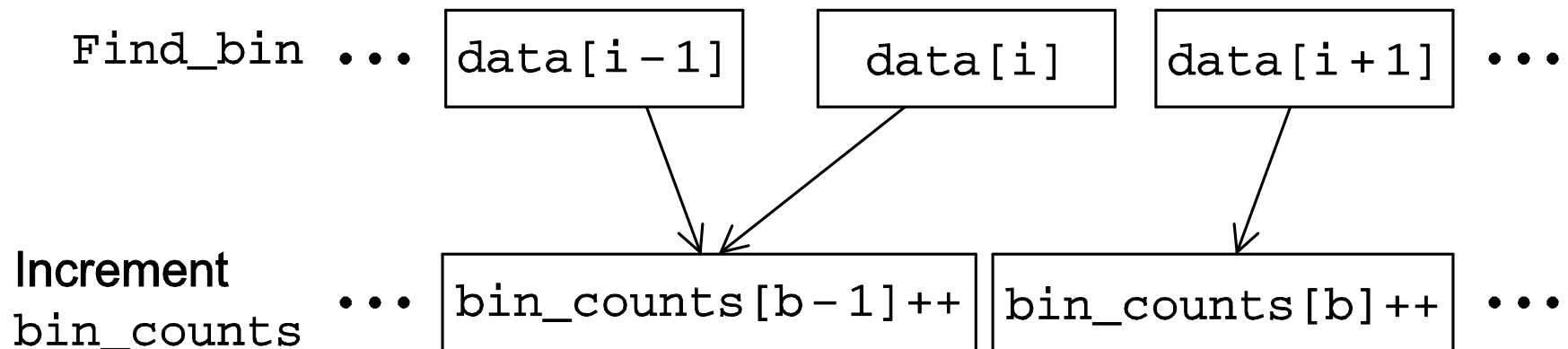
bin_counts[3] = 3

bin_counts[4] = 6

Serial Program

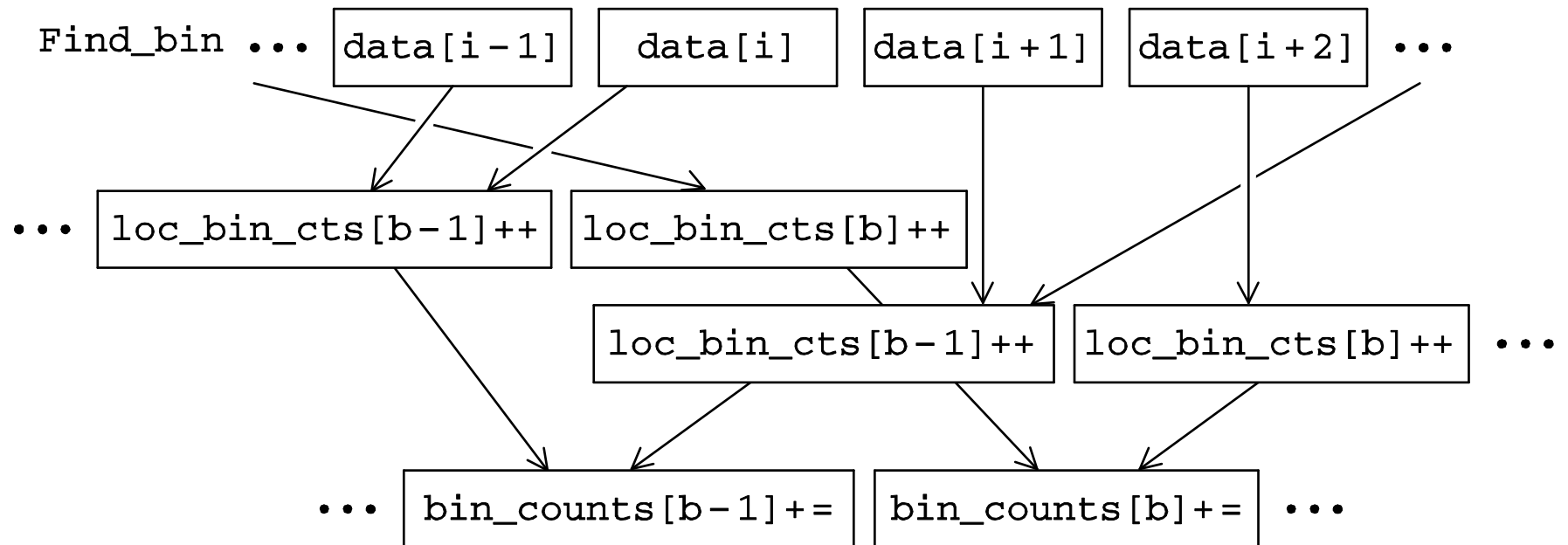
```
int bin = 0;
for( i = 0; i < data_count; i++){
    bin = find_bin(data[i], ...);
    bin_counts[bin]++;
}
```

First two stages of Foster's Methodology

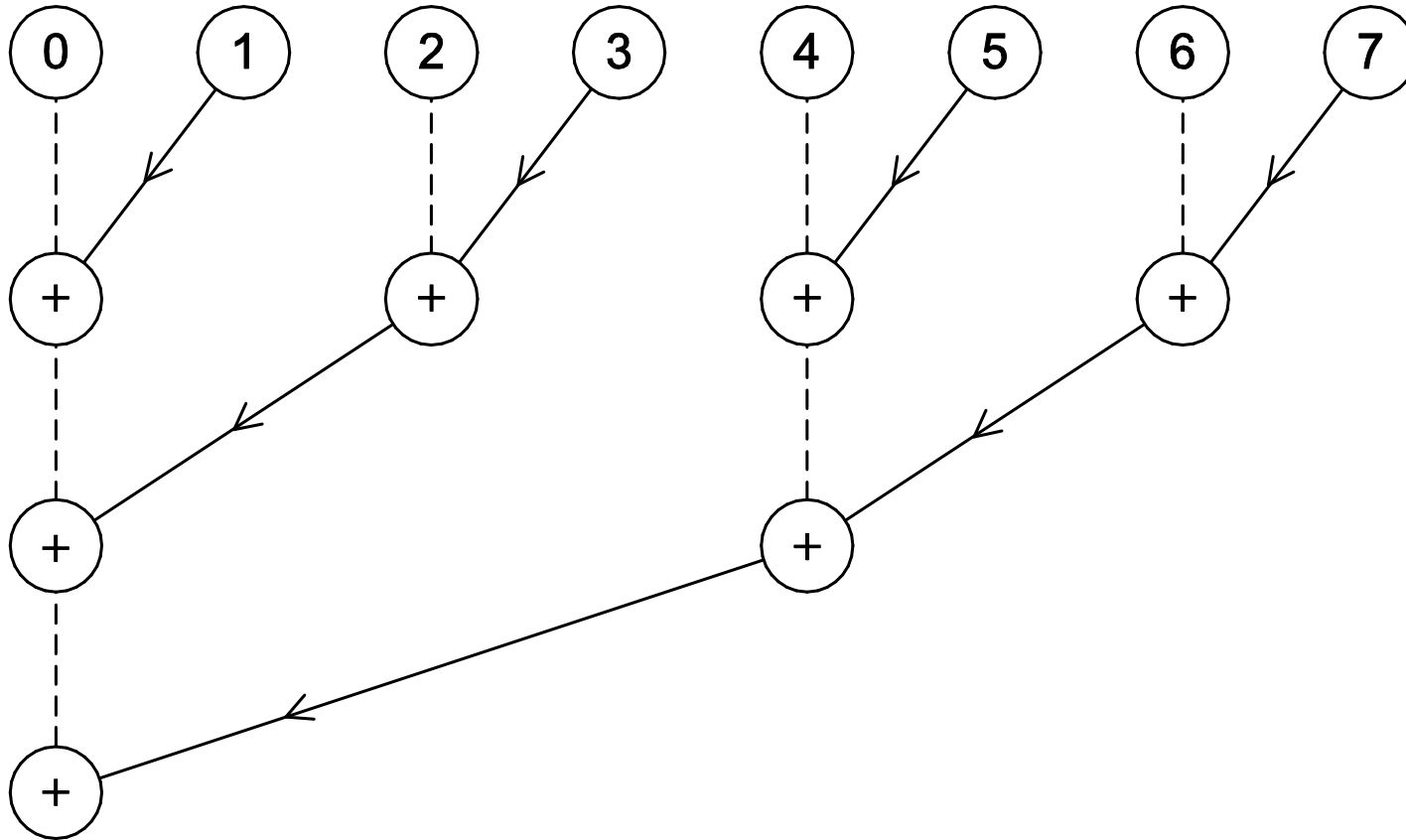


Find_bin returns the bin that `data[i]` belongs to.

Alternative definition of tasks and communication



Adding the local arrays



Conclusions

- Parallel software
 - We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.
 - Later we will look at GPUs
- Parallel Program Design
 - Partition
 - Determine communication
 - Aggregate (if needed)
 - Map