Parallel Programming COMP5112

Parallel Software

Slides adapted from the lecture notes by Peter Pacheco

Roadmap

- Parallel software
- Input and output
- Performance
- Parallel program design
- Writing and running parallel programs
- Concluding Remarks

PARALLEL SOFTWARE

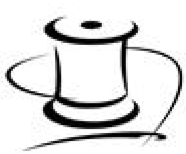
The burden is on software

- Hardware and compilers can keep up the pace needed for parallelism.
- How does parallel software work?
  – 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 program 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

- Dynamic threads
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but 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, but 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

The same input can result in different output.

Race Condition

```
my_val = Compute_val ( my_rank ) ;
x += my_val ;
```

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

The output depends on the timing of concurrent execution.

Critical Section

- Programmer's job to ensure mutually exclusive access to the critical section
  - Mutual exclusion lock (mutex, or simply lock)

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

A block of code that can only be executed by one thread at a time.

busy-waiting

```
my_val = Compute_val ( my_rank ) ;
i f ( my_rank == 1)
    whi l e ( ! ok_for_1 ) ;  /* Busy−wait loop */
x += my_val ;  /* Critical section */
i f ( my_rank == 0)
    ok_for_1 = true ;  /* Let thread 1 update x */
```

message-passing program example

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

Notes on the message-passing program

- It is SPMD. The two processes are using the same executable.
- The variable *message* refers to memory blocks in different processes.
- Process 0 is able to write to stdout.
- Typically both Send and Receive may block until the message is sent/received.
- There are other MPI API functions.

Input and Output

- In distributed memory programs, only process 0 will access *stdin*. In shared memory programs, only the master thread or thread 0 will access *stdin*.

- In both distributed memory and shared memory programs all the processes/threads can access *stdout* and *stderr*.

Input and Output

- However, because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.

- Debug output should always include the rank or id of the process/thread that's generating the output.

Input and Output

- Only a single process/thread will attempt to access any single file other than *stdin*, *stdout*, or *stderr*. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

PERFORMANCE

Speedup of a parallel program

- Number of cores = p
- Serial run-time = T_{serial}
- Parallel run-time = $T_{parallel}$

linear speedup

$$S = \frac{T_{serial}}{T_{parallel}}$$

$$T_{parallel} = T_{serial} \ / \ p$$

Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\dfrac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

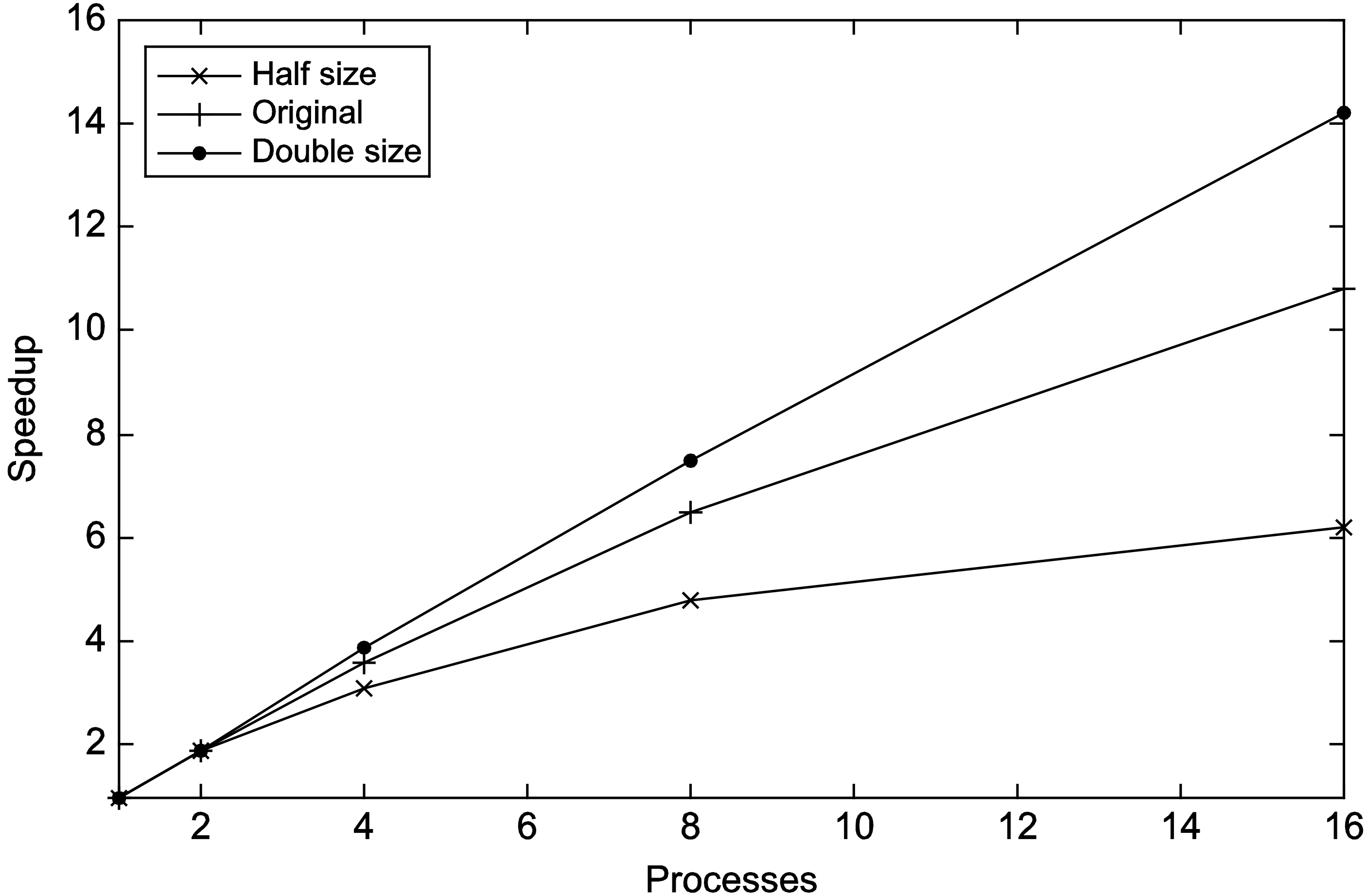
Speedup and efficiency example

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

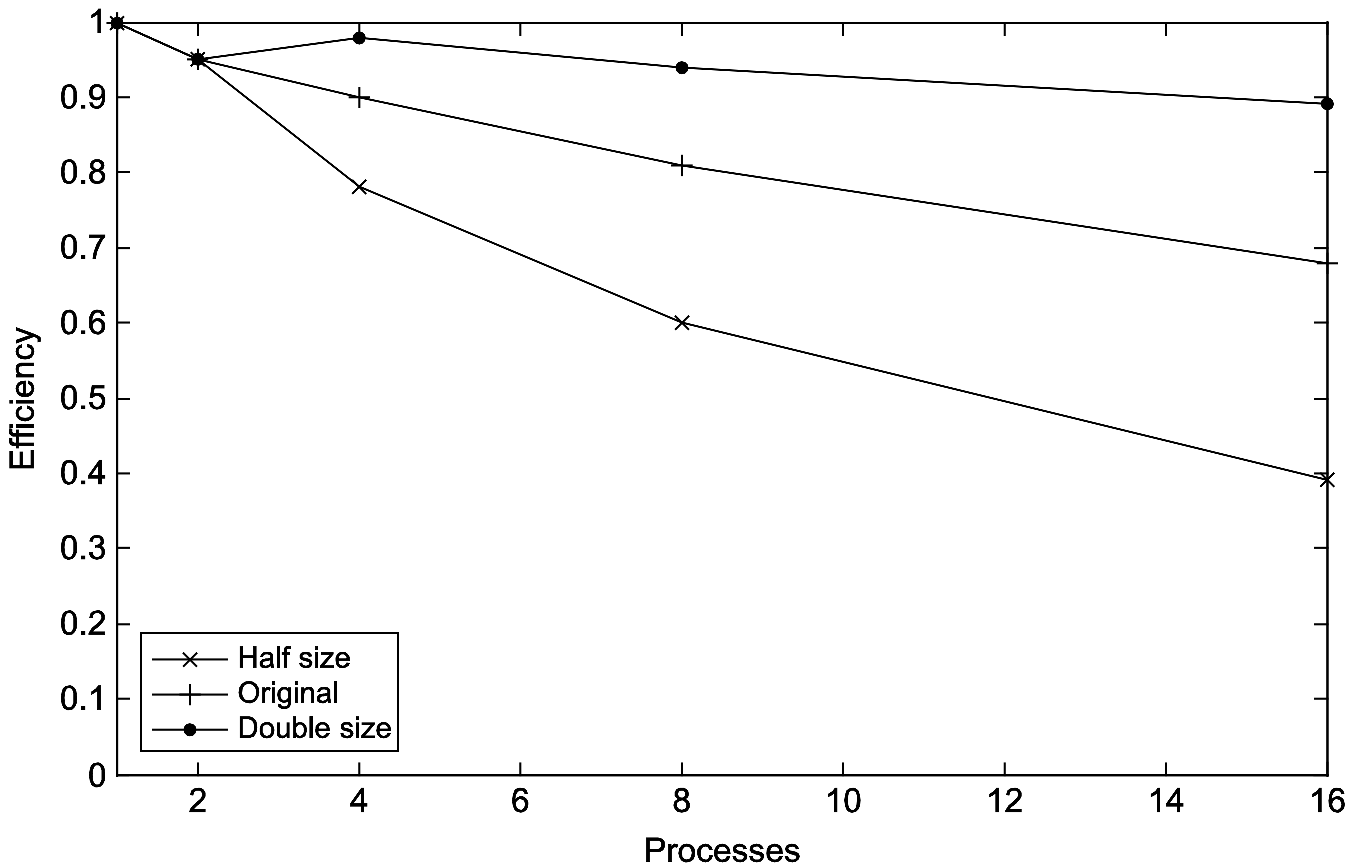
Speedups and efficiencies of parallel program on different problem sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Example Speedup Figure



Example Efficiency Figure



Parallel program overhead

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

Amdahl's Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.

n: number of cores; p: percentage of the serial time that the parallelizable part takes

$$\text{Speedup} = \frac{T_{serial}}{p \times T_{serial} / n + (1\text{-}p) \times T_{serial}} = \frac{1}{1 - p + p/n}$$

Example

- We can parallelize 90% of a serial program.
- Parallelization is “perfect” regardless of the number of cores *p* we use.
- T_{serial} = 20 seconds
- Parallel runtime of parallelizable part is

$$0.9 \times T_{serial} / p = 18 / p$$

Example (cont.)

- Runtime of "unparallelizable" part is

$$0.1 \times T_{serial} = 2$$

- Overall parallel run-time is

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

Example (cont.)

- Speedup

$$S = \frac{T_{serial}}{0.9 \times T_{serial} / p + 0.1 \times T_{serial}} = \frac{20}{18 / p + 2}$$

Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.

Time Measurement

- Wall clock time
- CPU time (reported by c function *clock*) ?
- Start to finish (Unix shell command *time*)?
- A program segment of interest?

Measure Elapsed Time

Pseudo function name

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

MPI_Wtime

omp_get_wtime

Taking Timings in Parallel Programs

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

Elapsed Time in Parallel Programs

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

PARALLEL PROGRAM DESIGN

Foster's 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.

Foster's methodology

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

Foster's 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.

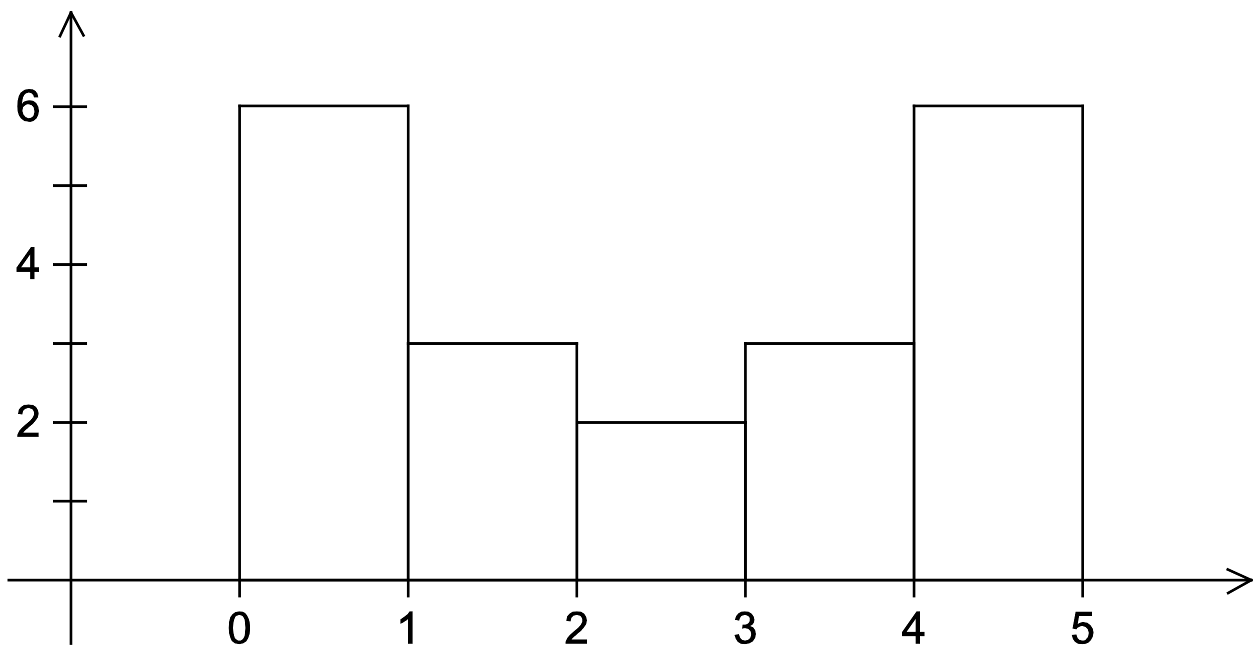
Foster's methodology

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 building

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

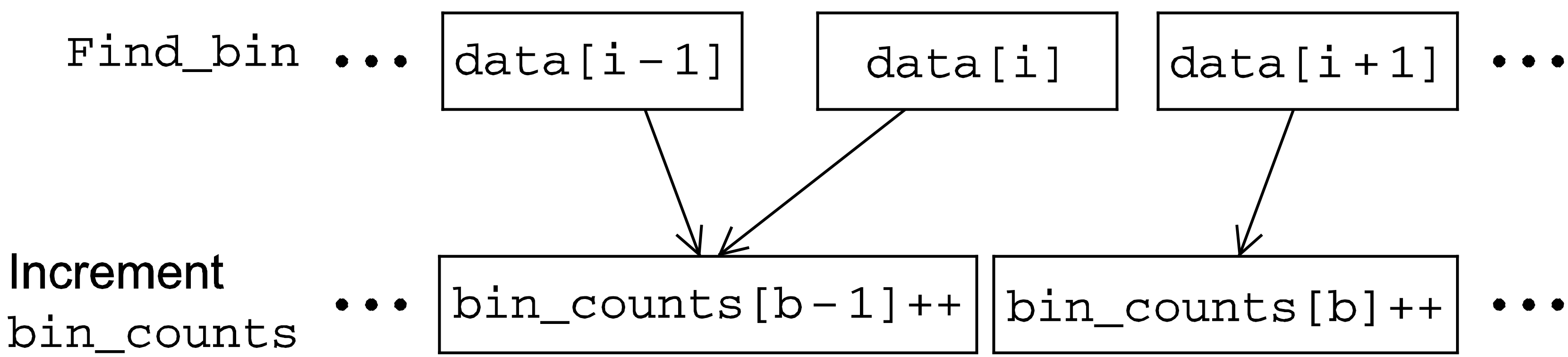
1. The number of measurements: data_count
2. An array of data_count floats: data
3. The minimum value for the bin containing the smallest values: min_meas
4. The maximum value for the bin containing the largest values: max_meas
5. The number of bins: bin_count

Serial program - output

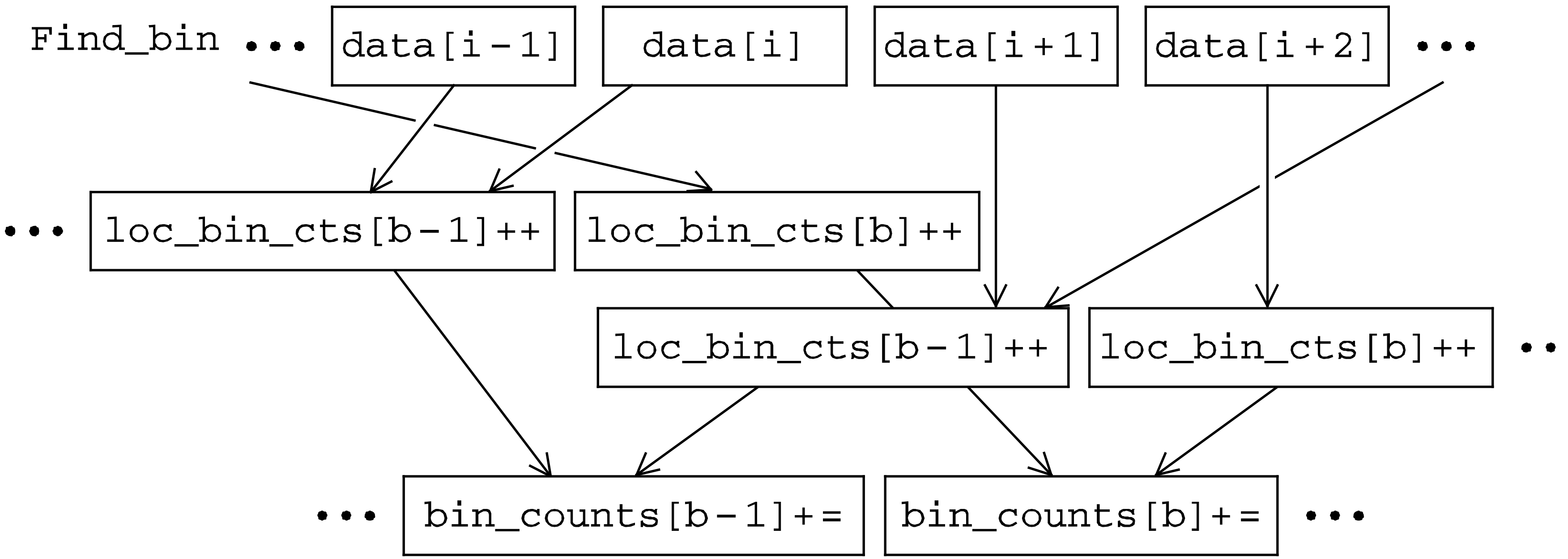
1. bin_maxes : an array of bin_count floats

2. bin_counts : an array of bin_count ints

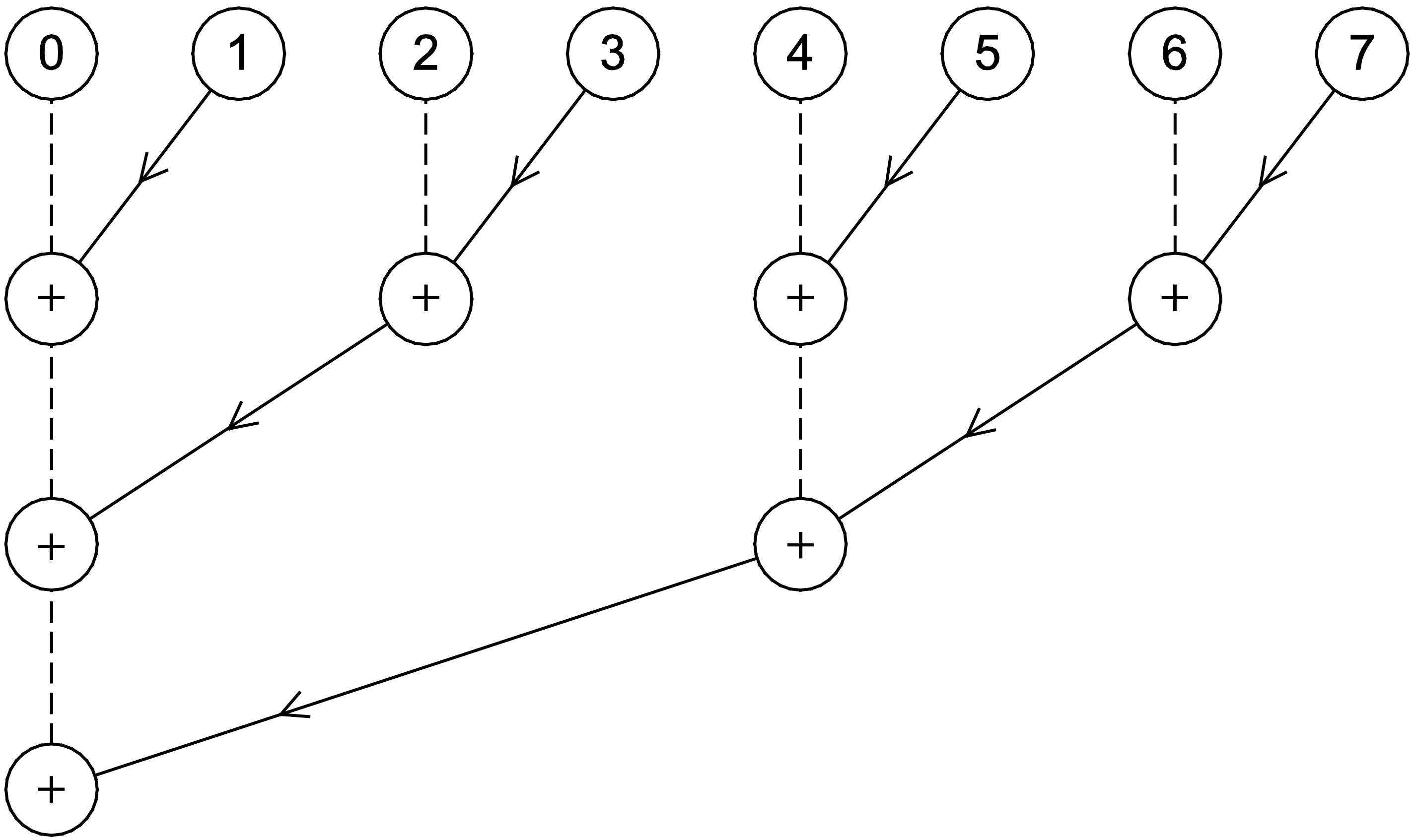
First two stages of Foster's Methodology



Alternative definition of tasks and communication



Adding the local arrays



Concluding Remarks (1)

- Serial systems
  - The standard model of computer hardware has been the von Neumann architecture.
- Parallel hardware
  - Flynn's taxonomy.
- Parallel software
  - We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching (SPMD).

Concluding Remarks (2)

- Input and Output
  - We'll write programs in which one process or thread can access stdin, and all processes can access stdout and stderr.
  - However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing stdout.

Concluding Remarks (3)

- Performance
  - Speedup
  - Efficiency
  - Amdahl’s law
  - Scalability
- Parallel Program Design
  - Foster’s methodology