Parallel Software

Jinkyu Jeong

Roadmap

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

PARALLEL SOFTWARE

The burden is on software

- **Hardware and compilers can keep up the pace needed.**
- **Software needs to be parallel for routine performance increases**

- **Terminology**
  - In a shared memory program:
    - Start a **single process** and fork **multiple threads**.
    - Threads carry out **a task**.
  - In a distributed memory program:
    - Start **multiple processes**.
    - Processes carry out **a task**.

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

- **In a shared memory program, variables can be shared or private**
  - Implicit communication through shared variables

  - A shared memory program consists of multiple threads

→ dynamic/static threads

  - Multiple threads can access a shared variable concurrently

→ No determinism

Shared Memory (2)

- **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 0 > my_val = 7

Thread 1 > my_val = 19

Nondeterminism

```
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`

Nondeterminism

- **Race condition**
- **Critical section**
- **Mutually exclusive**
- **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 ) ;
```

Mutual exclusion

- **Mutex lock**
  - Only one thread can enter a critical section
  - Other threads sleep until a lock owner exits the critical section
- **Spinlock**
  - Other threads are spinning until a lock owner exits the critical section
- **Semaphore**
  - Allow multiple threads can enter a critical section
- **Monitor**
  - Providing mutual exclusion at a higher-level

Mutual exclusion (2)

- **Read/Write lock**
  - Multiple readers can concurrently enter a critical section
  - A single writer can enter a critical section without readers
- **Variants of spinlock**
  - Ticket lock
    - Spinners (waiters) have its ticket
    - No starvation
  - MCS lock
    - Cacheline movement is minimized
    - Each spinner spins on a different lock variable to each other

Mutual exclusion (3)

- **Read-Copy-Update**
  - Multiple readers and one writer can enter the critical section without any lock being held
  - Writers should be mutually exclusive to each other
  - Update to a shared variable should follow a rule
- **Transactional memory**
  - Multiple memory updates are treated as a transaction
  - Commit or abort memory updates

Thread Safety

- **Thread safe**
  - A code or a function is safe when multiple threads can safely use it
  - strtok() in C is not thread safe
    - Its internal buffer is shared between threads because it is static

```
char* a = “abc def”, *token;
token = strtok(a, “ “);
// token == “abc” ?
token = strtok(NULL, “ “);
// token == “def”?
```

Thread 0

```
char* a = “ghi jkl”, *token;
token = strtok(a, “ “);
// token == “ghi” ?
token = strtok(NULL, “ “);
// token == “jkl” ?
```

Thread 1

  - strtok_r() is thread-safe
    - Strtok_r(char* str, char* delim, char** saveptr)
    - Saveptr is used for a thread-local buffer

Thread Safety (1)

- strtok_r() is thread-safe
  - Strtok_r(char* str, char* delim, char** saveptr)
  - Saveptr is used for a thread-local buffer

```
char* a = “abc def”, *token;
char* ptr;
token = strtok_r(a, “ “, &ptr);
// token == “abc” ? yes
token = strtok(NULL, “ “, &ptr);
// token == “def”? yes
```

Thread 0

```
char* a = “ghi jkl”, *token;
char * ptr;
token = strtok_r(a, “ “, &ptr);
// token == “ghi” ? yes
token = strtok(NULL, “ “, &ptr);
// token == “jkl” ? yes
```

Thread 1

- POSIX.1 specifies thread-safe funtions
  - E.g., read(), write(), printf(), …
  - Just do “man 7 pthread” in UNIX shell

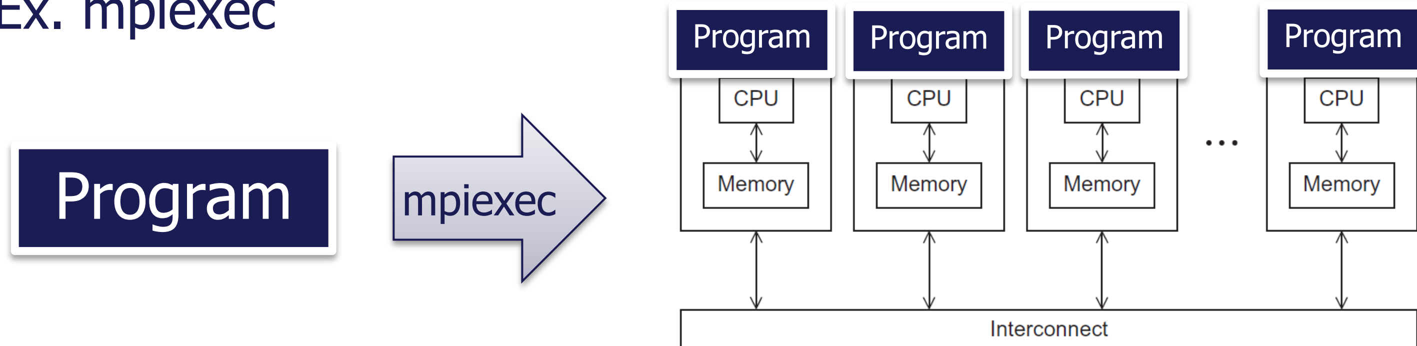
Thread Safety (2)

- **Download and test the codes**
  - http://csl.skku.edu/uploads/SSE3054S14/thread_safe_strtok.txt
  - http://csl.skku.edu/uploads/SSE3054S14/thread_unsafe_strtok.txt

Distributed memory

- **Each process can access only its private (local) memory**

- **Application framework for building a program in distributed memory**
  - Ex. MPI(message passing interface), Hadoop
  - APIs for sending/receiving messages
  - Running multiple programs at once
    - Ex. mpiexec



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 ) ;
}
```

Distributed memory (2)

- **Shared-memory programming easy and appealing to programmers**
- **Distributed shared memory (DSM)**
  - Compiler of operating system provide a transparent view of a shared-memory system upon a distributed memory system
  - Shared memory accesses are translated to sending messages
  - Ex. Treadmarks, single system image, partitioned global address space languages

Partitioned Global Address Space Languages

```
shared int n = . . . ;
shared double x[n], y[n] ;
private int i , my_first_element , my_last_element ;
my_first_element = . . . ;
my_last_element = . . . ;
/ * Initialize x and y  */
. . .
for ( i = my_first_element ; i <= my_last_element ; i++)
     x[i] += y[i] ;
```

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

- **Number of cores = p**
- **Serial run-time = T_{serial}**
- **Parallel run-time = $T_{parallel}$**
- **Speedup =** $S = \frac{T_{serial}}{T_{parallel}}$
- **Linear speedup =** $T_{parallel} = T_{serial} / p$

Efficiency of a parallel program

- **Efficiency**

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

- **Effect of overhead**
  - Spawning multiple threads/processes
  - Message passing overhead

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

Speedups and efficiencies of a parallel program

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

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

- **$T_{overhead}$**
  - Usually proportional to the number of process/threads

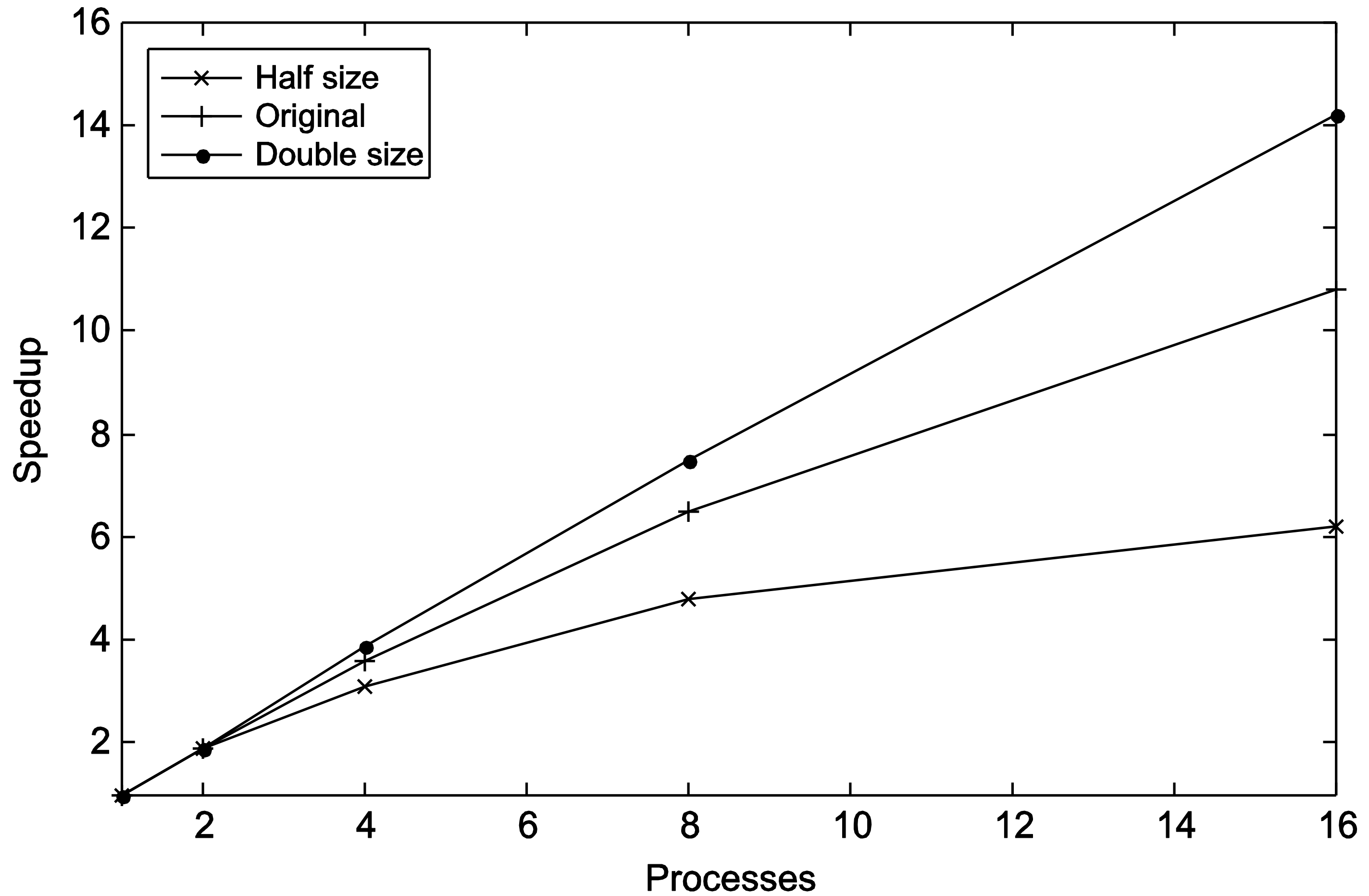
Speedups and efficiencies of parallel program on different problem sizes

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

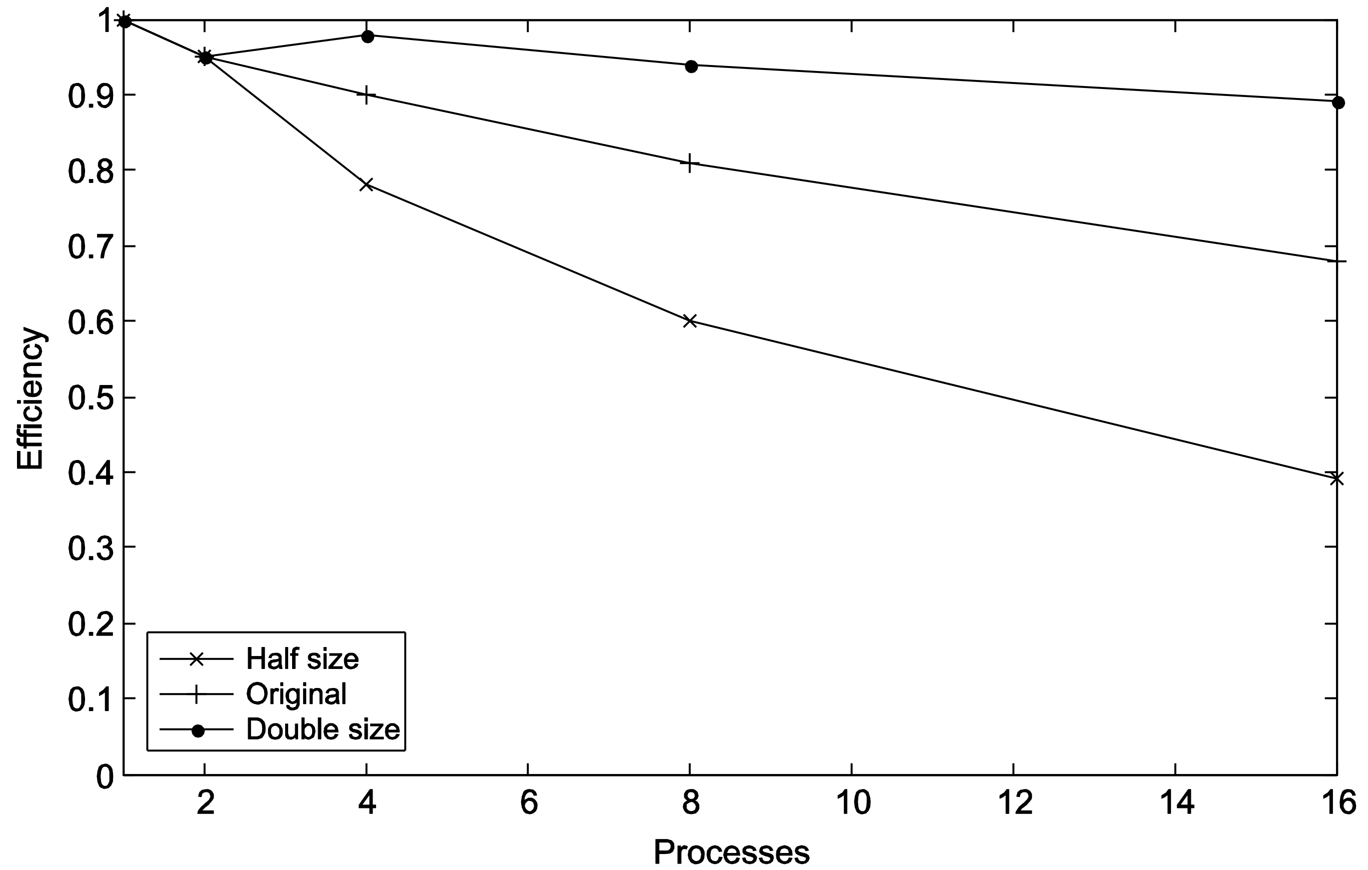
	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

- **With the same $T_{overhead}$, increasing the size of problem decreases the portion of $T_{overhead}$**

Speedup



Efficiency



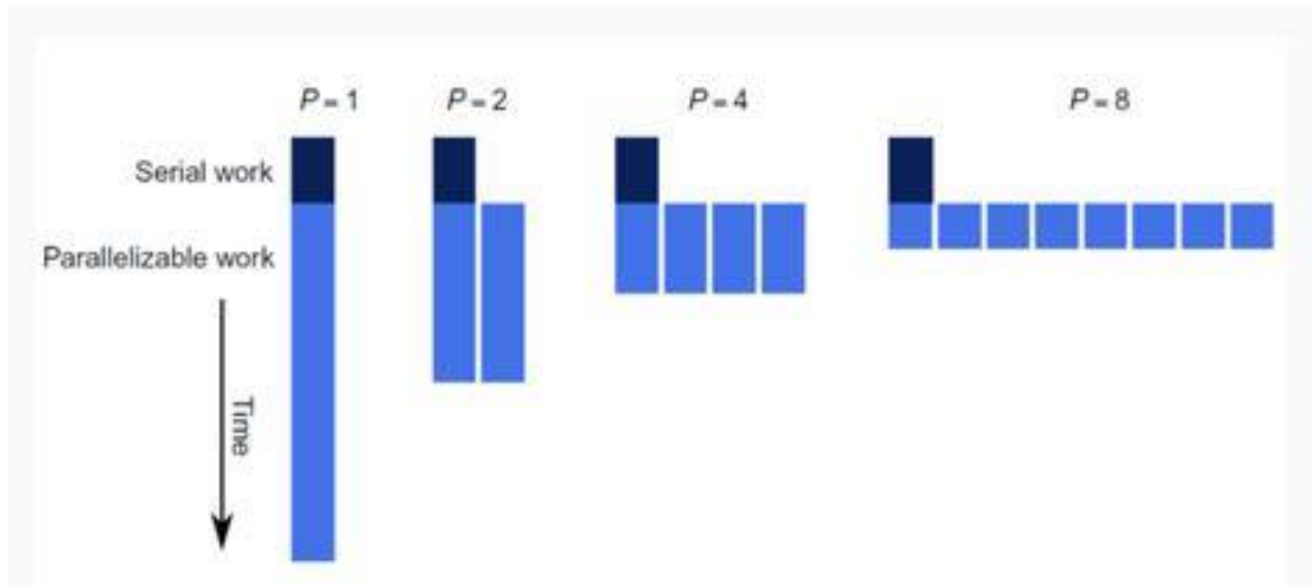
Amdahl's Law

- The possible speedup is going to be very limited — regardless of the number of cores available.

$n \in \mathbb{N}$ # of threads

$B \in [0, 1]$ Fraction of serial part

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{1}{B + \frac{1}{n}(1-B)}$$



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**
- **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**

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

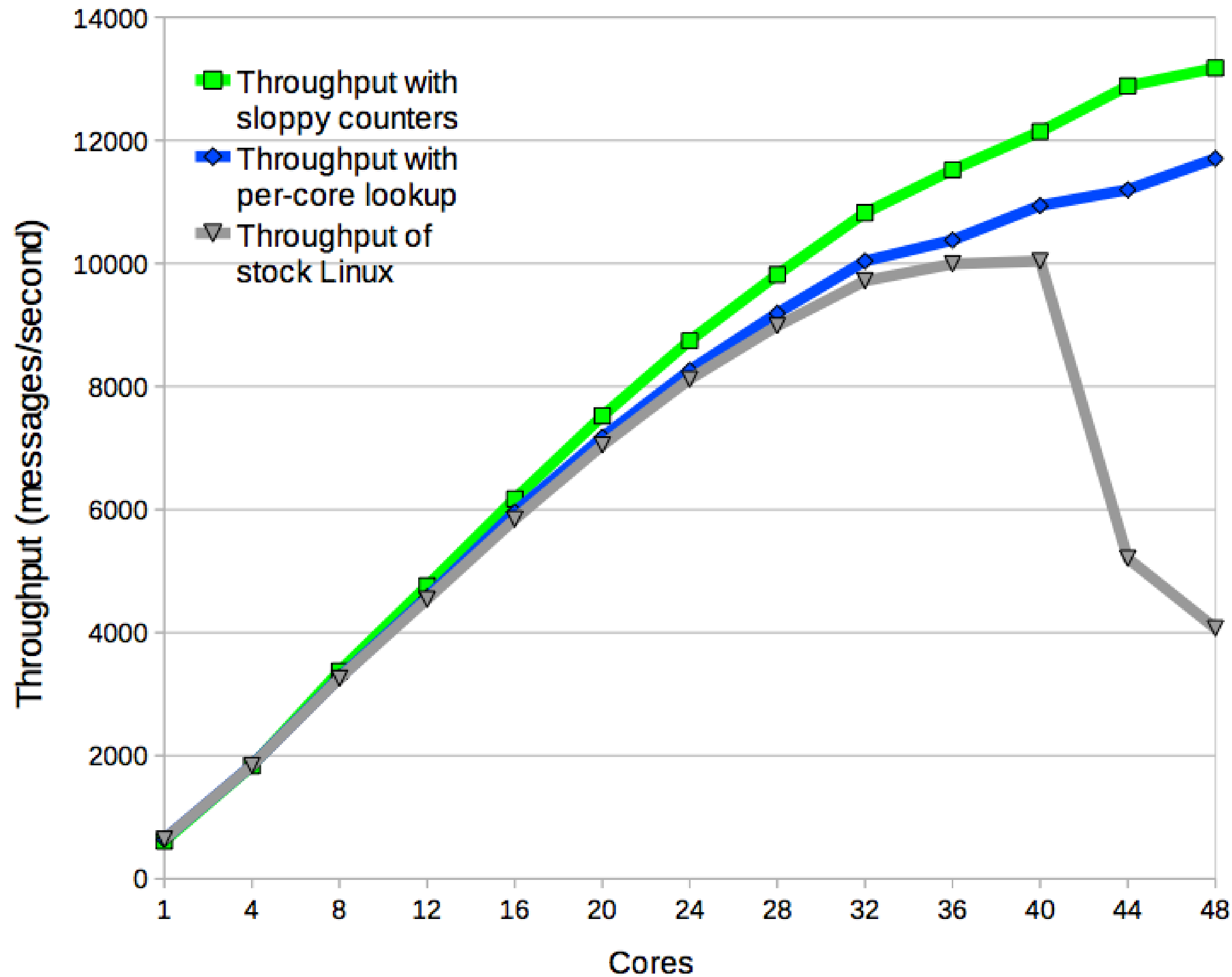
- **Speed up is**

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

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***.

Scalability issues in Commodity OS



[S. Boyd-Wickizer, et. al., OSDI 2010]

Taking Timings

- **What is time?**
- **Start to finish?**
- **A program segment of interest?**
- **CPU time?**
- **Wall clock time?**

Wall clock time

Usage

Core 1 | Idle | Core 1 | Idle | Core 1

Core 0 | I/O wait | Core 0 | Idle | Core 0

Time

Program starts

Program finishes

Core 1 | Core 1 | Core 1 | Core 0 | Core 0 | Core 0

CPU time

Taking Timings

```
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);
```

theoretical function

MPI_Wtime

omp_get_wtime

Taking Timings

```
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.

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.

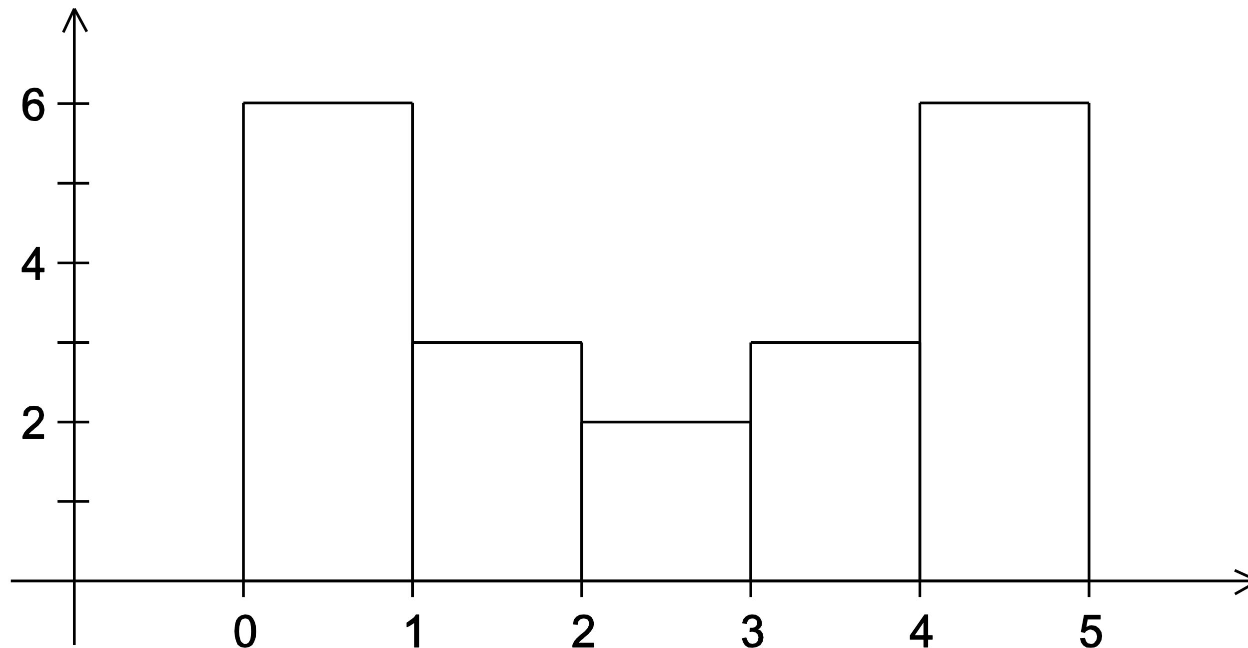
Foster's methodology

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

- 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

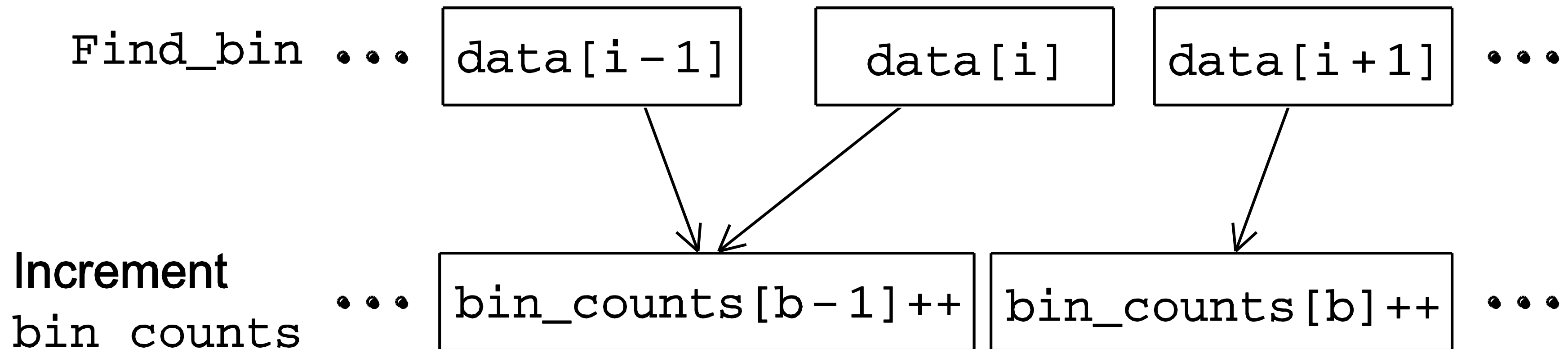


Serial program

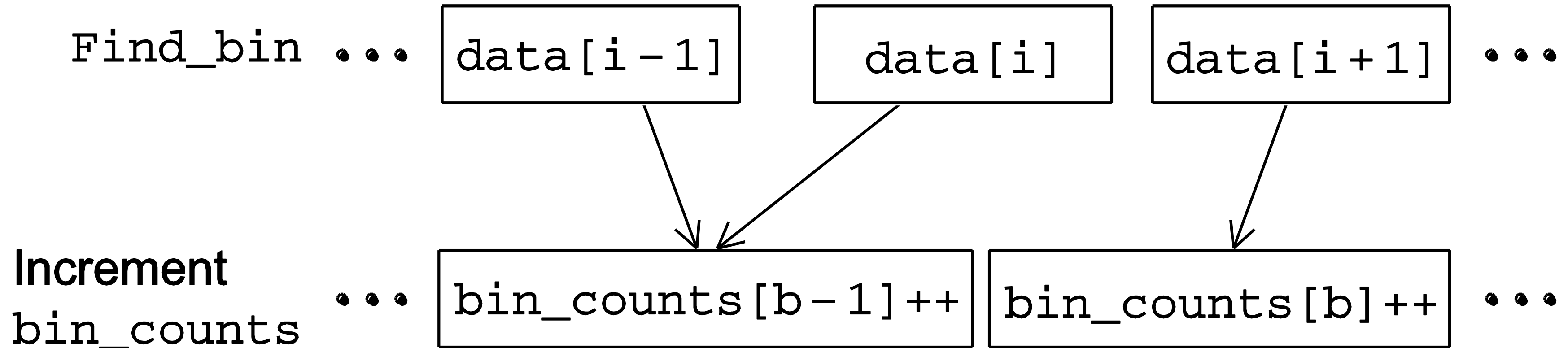
1. The number of measurements: data_count
2. An array of data_count floats: data
3. The number of bins: bin_count
4. bin_maxes : an array of bin_count floats

   Ex. bin_maxes = [1, 2, 3, 4, 5]
5. bin_counts : an array of bin_count ints

   Ex. bin_counts = [6, 3, 2, 3, 6]

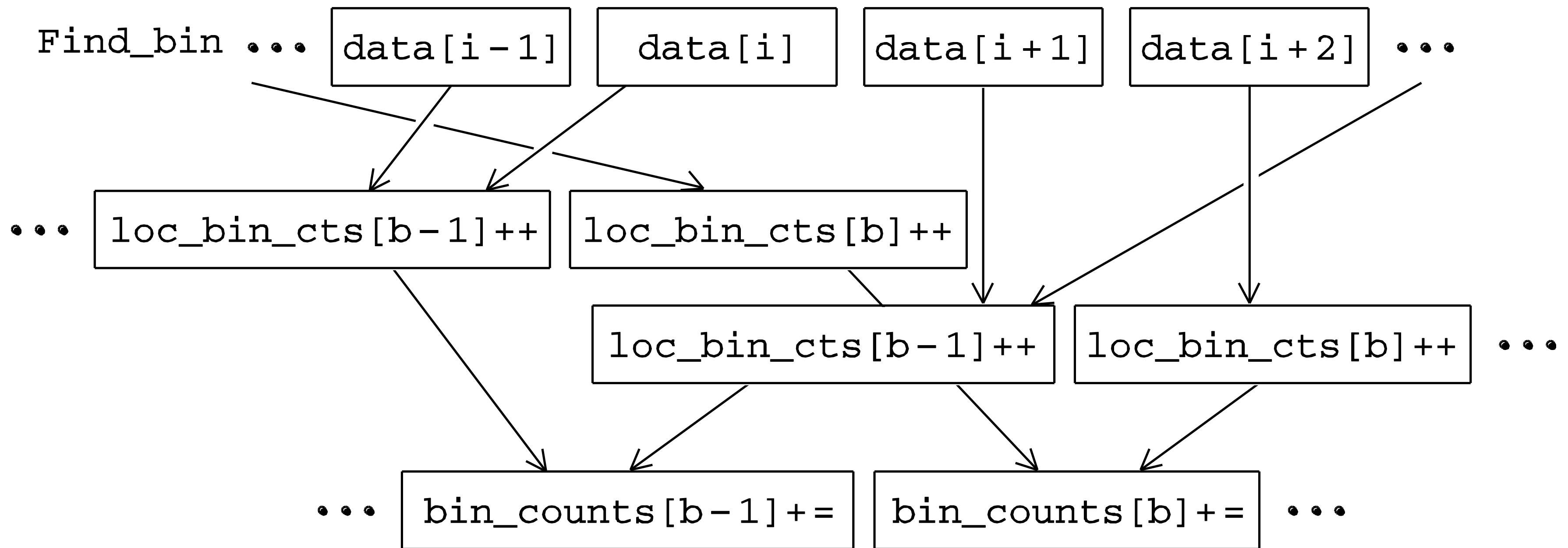


First two stages of Foster's Methodology



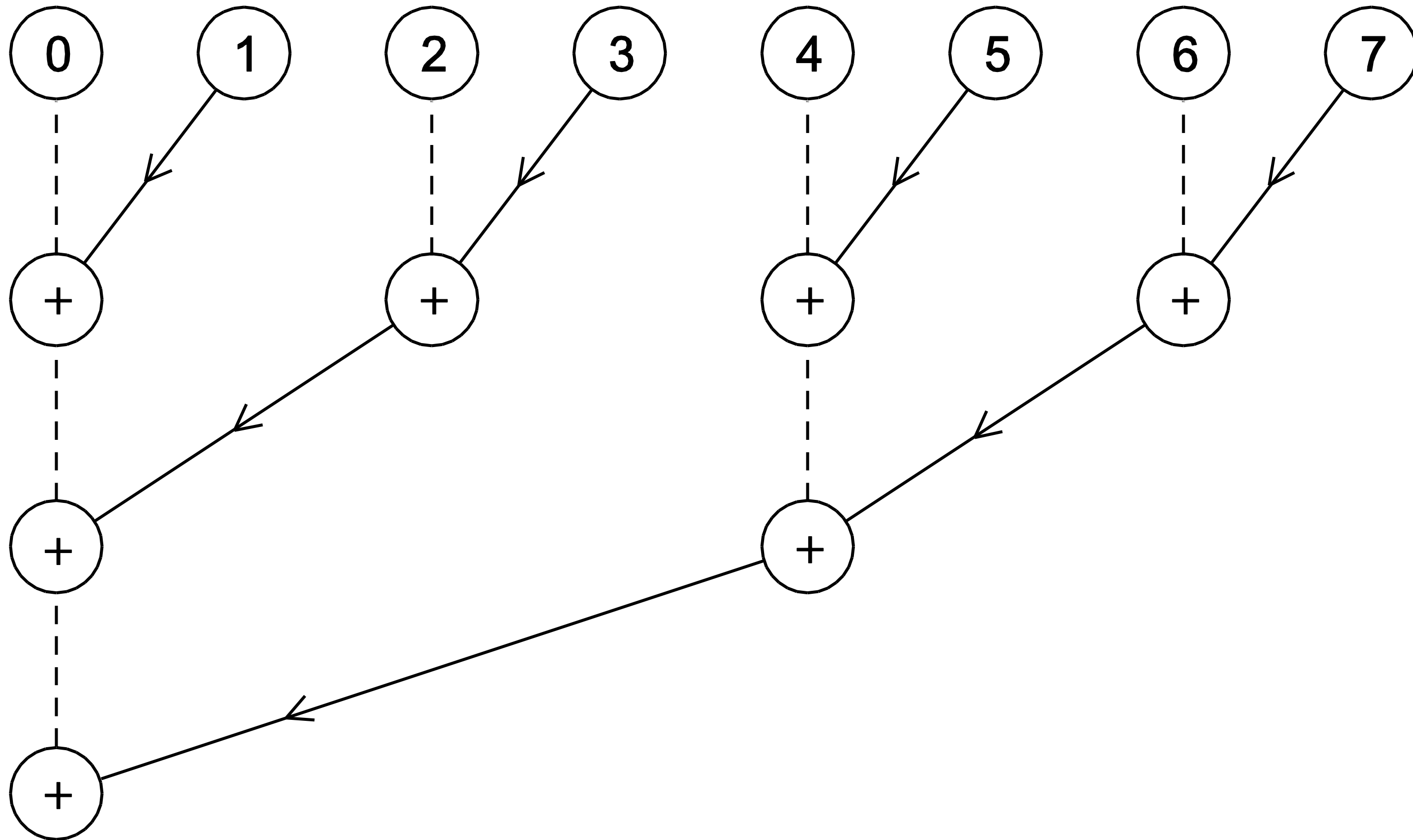
- **Partition data array only**
  - Core 0 works on data[0] ~ data[n/p – 1]
  - Core 1 works on data[n/p] ~ data[2n/p – 1]
  - …
  - bin_counts is shared
  - Enormous communication for each update of bin_counts
  - We have to aggregate the task for updating bin_counts to each thread

Alternative definition of tasks and communication



- **Partition data array and bin_counts array**
  - Core 0 works on data[0] ~ data[n/p – 1]
  - …
  - Each thread has private loc_bin_cts
  - Communication is only to merge multiple loc_bin_cts variables

Adding the local arrays



Concluding Remarks (1)

- **Parallel software**
  - We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.
  - SPMD programs.
- **Shared/Distributed memory system**
  - Communication/synchronization
  - Nondeterminism
- **Input and Output**
  - Be careful of using stdin/out/err and file accesses

Concluding Remarks (2)

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