PARALLEL SOFTWARE

Dr. Steve Petruzza

Reference material for part of this presentation :

An Introduction to Parallel Programming by Peter Pacheco

UtahStateUniversity

The burden is on software

- Hardware and compilers can keep up the pace needed.
- 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

- 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

See code example

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 ) ;
```

busy-waiting

```
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 */
```

Approaches to distributed memory access

- Message passing
- Partitioned Global Address Space (PGAS)

message-passing

```
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 ) ;
}
```

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  */
. . .
f  o r ( 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 = $T_{parallel}$

linear speedup

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

Speedup of a parallel program

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

Efficiency of a parallel program

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

Speedups and efficiencies of a parallel program

Speedup matrix vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

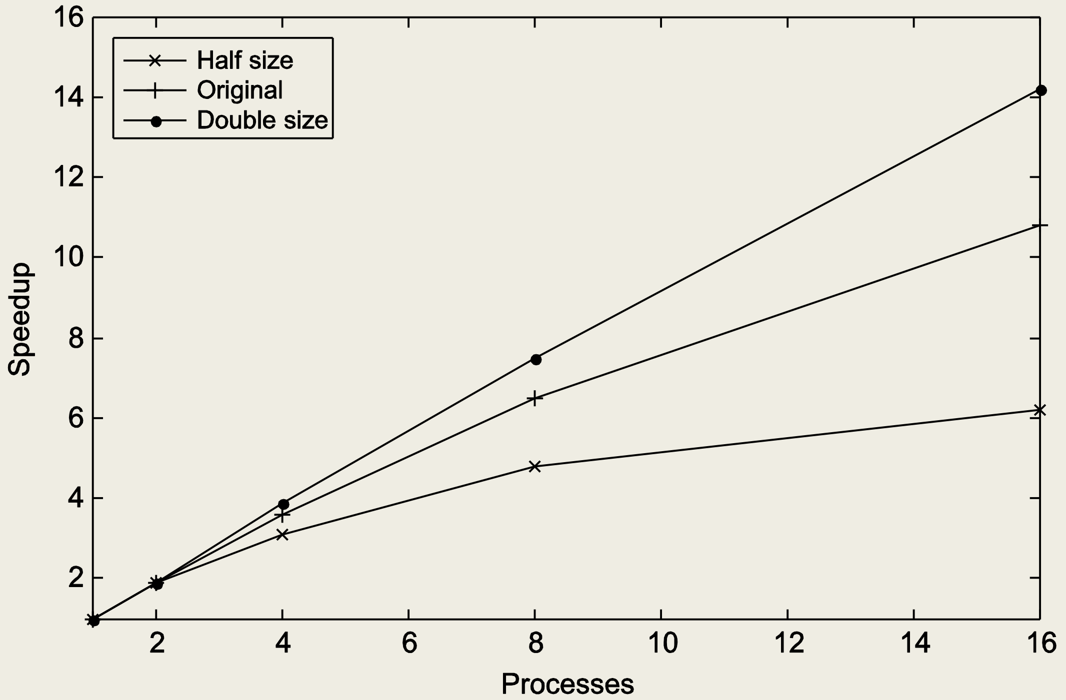
Speed up and efficiency

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

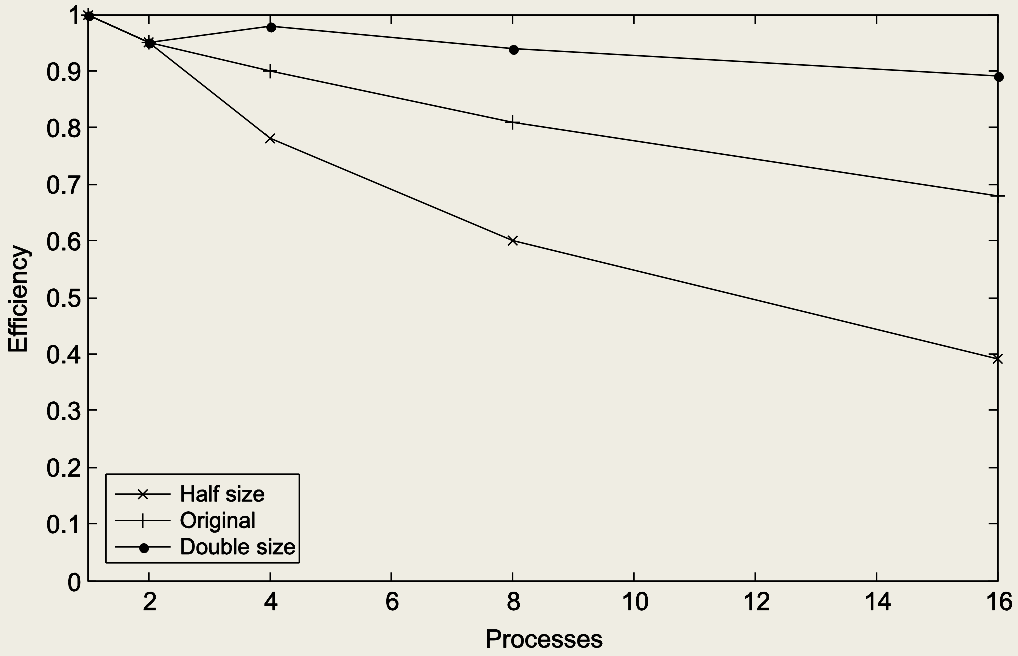
	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Speedup



Half size
Original
Double size
Speedup
Processes
2
4
6
8
10
12
14
16

Efficiency

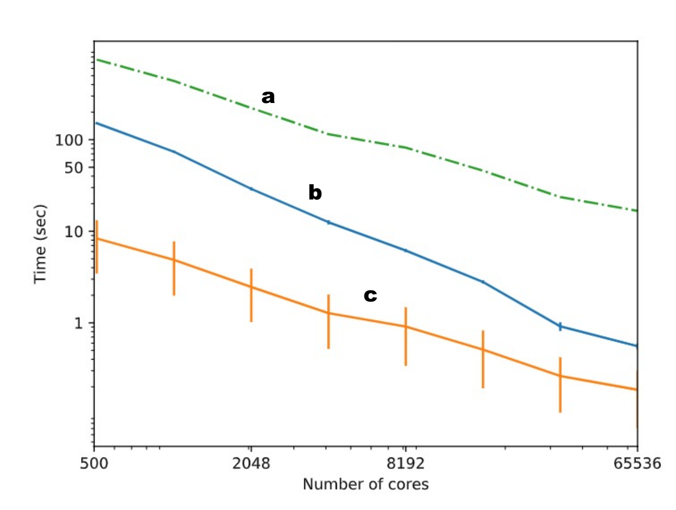


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Efficiency
2
4
6
8
10
12
14
16
Processes
Half size
Original
Double size

Effect of overhead

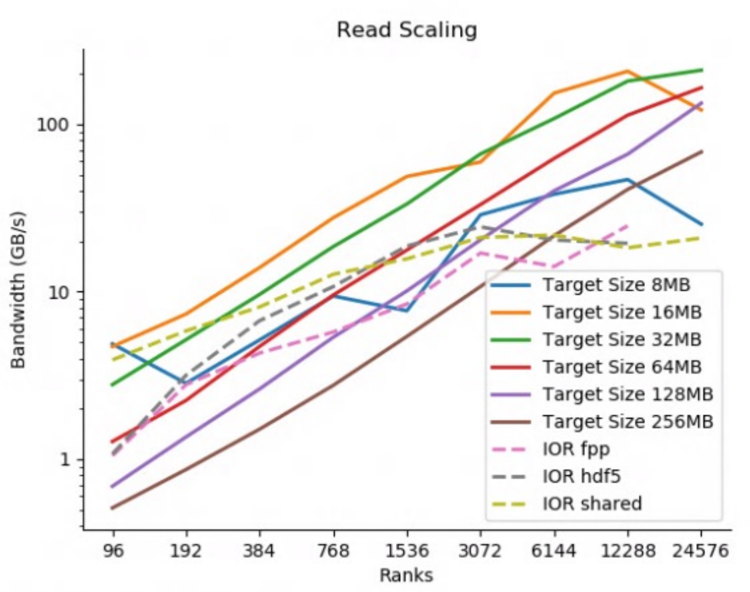
$$T_{parallel} = T_{serial} / p + T_{overhead}$$

Execution time



Which one has better performance?

Throughput/bandwidth performance



Which one has better performance?

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.

- Example, can we parallelize this serial program?

a = x + y

c = b + a

d = c + a

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

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

Example (cont.)

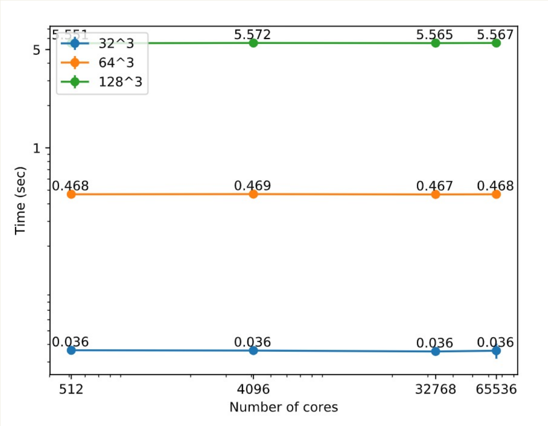
- Speed up

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

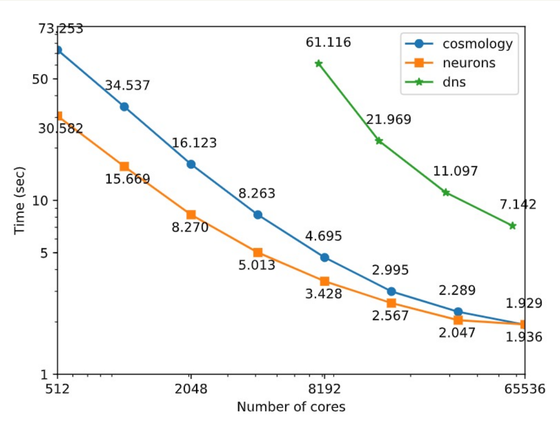
Scalability

- 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*.

Weak and strong scalability, which is which? (assuming both scale well)



A



B

Taking Timings

- What is the run-time of a program?
- Start to finish?
- A program segment of interest?
- CPU time? (*clock*() function, does not include the time when the program was idle)
- Wall clock time? (beginning to end of execution)
- *time* linux command:
  - *real: wall clock time*
  - *user: code running in user mode*
  - *sys: code running in kernel mode (e.g., file I/O, start process)*

Taking Timings

theoretical

function

```
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 a parallel environment

```
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);
```

Taking Timings in a parallel environment

```
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.

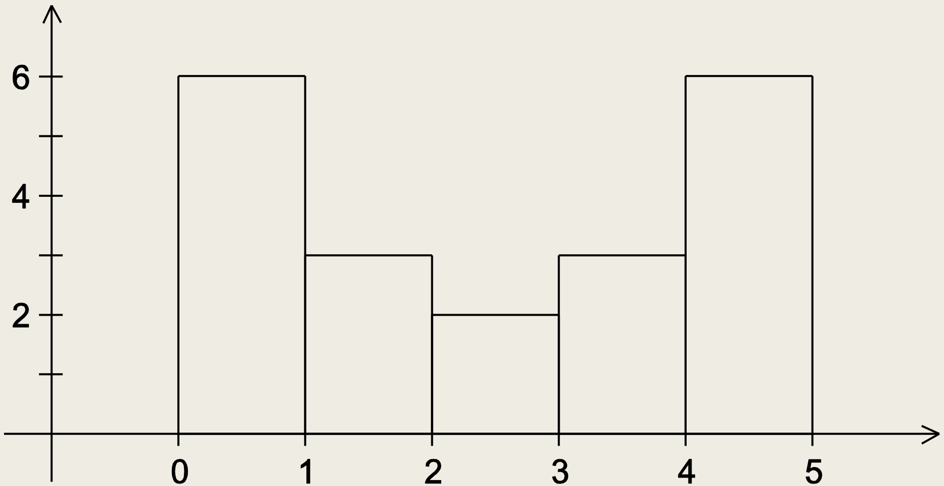
Foster's methodology

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

- 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



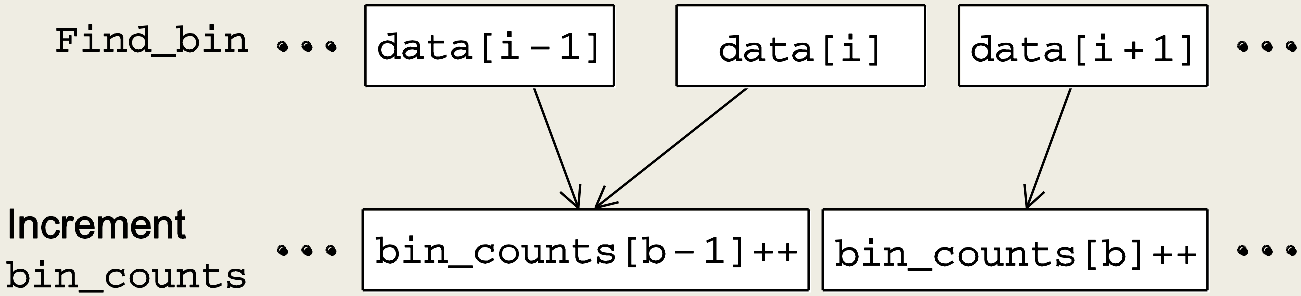
Serial program - input

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

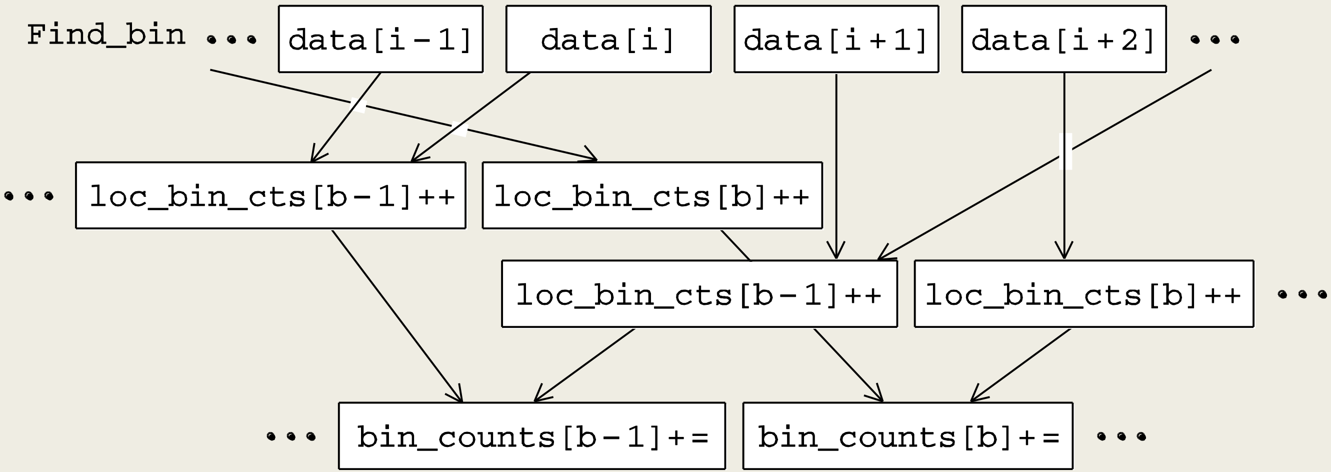
Serial program - output

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



Each core updates its local bins (arrays)

Adding the local arrays