Performance

Performance

- Speed up and efficiency
- Amdahl’s law
- Scalability

Speedup

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

linear speedup

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

- In practice, it is unlikely to get linear speedup!!

Speedup of a parallel program

- We define the **speedup** of a parallel program to be

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

- For linear speed up S=p

Efficiency of a parallel program

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

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

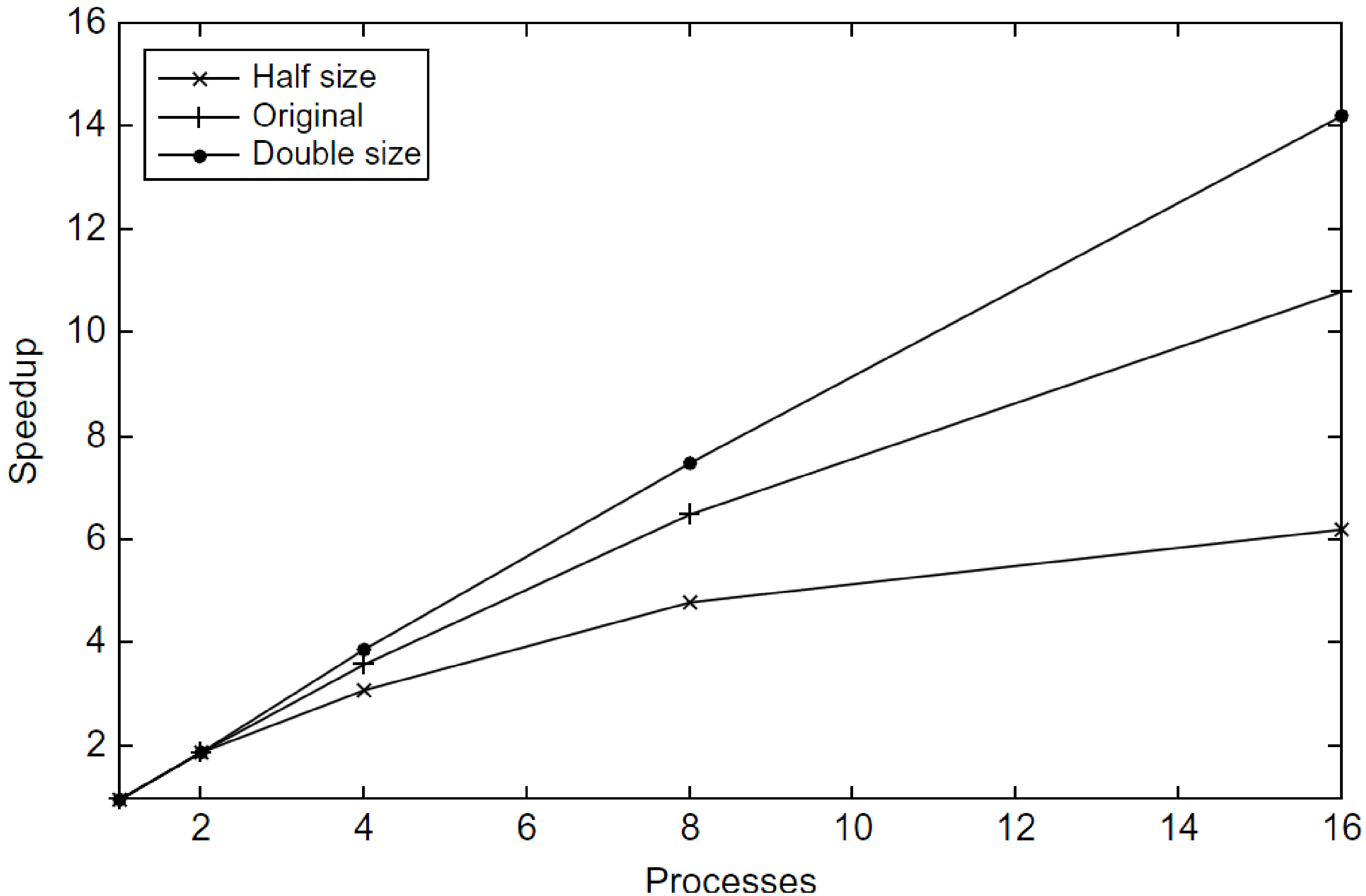
	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Efficiency of a parallel program

- When we increase the problem size, the speedups and the efficiencies increase, while they decrease when we decrease the problem size.
- This behavior is quite common → Parallel programs are developed by dividing the work of the serial program among the processes/threads
- Add in the necessary “parallel overhead” such as mutual exclusion or communication.

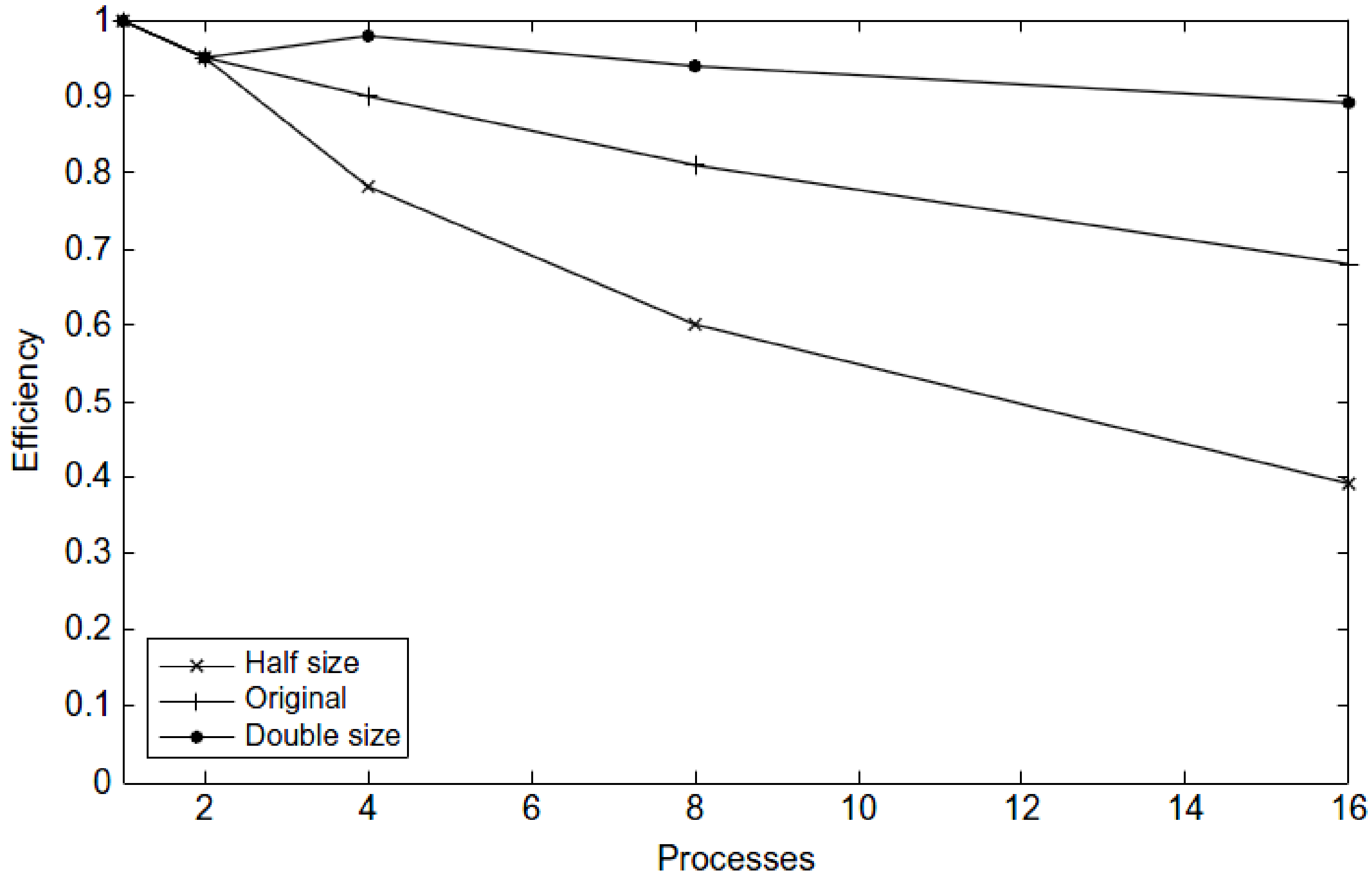
$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}$$

Speed up



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

Amdahl's Law

- 1960s, Gene Amdahl made an observation:

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

- 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

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

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 (when p >> 0, S ≤ 10)

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

Amdahl's Law

- Problems/questions regarding Amdahl's Law:
  - It doesn't take into consideration the problem size
  - Thousands of programs used by scientists obtain huge speedups on large distributed-memory systems
  - Is a small speedup so awful?

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*.

Taking Timings

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

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

```
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

```
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);
```