

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.

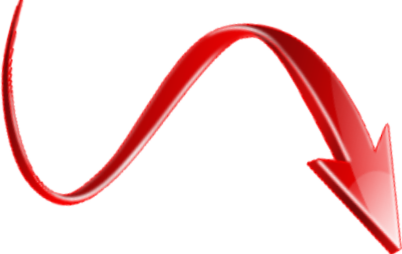
Better performance, but potential waste of system resources.

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

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 */
```

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 ) ;  
}
```

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

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

Speedup of a parallel program

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

Efficiency of a parallel program

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

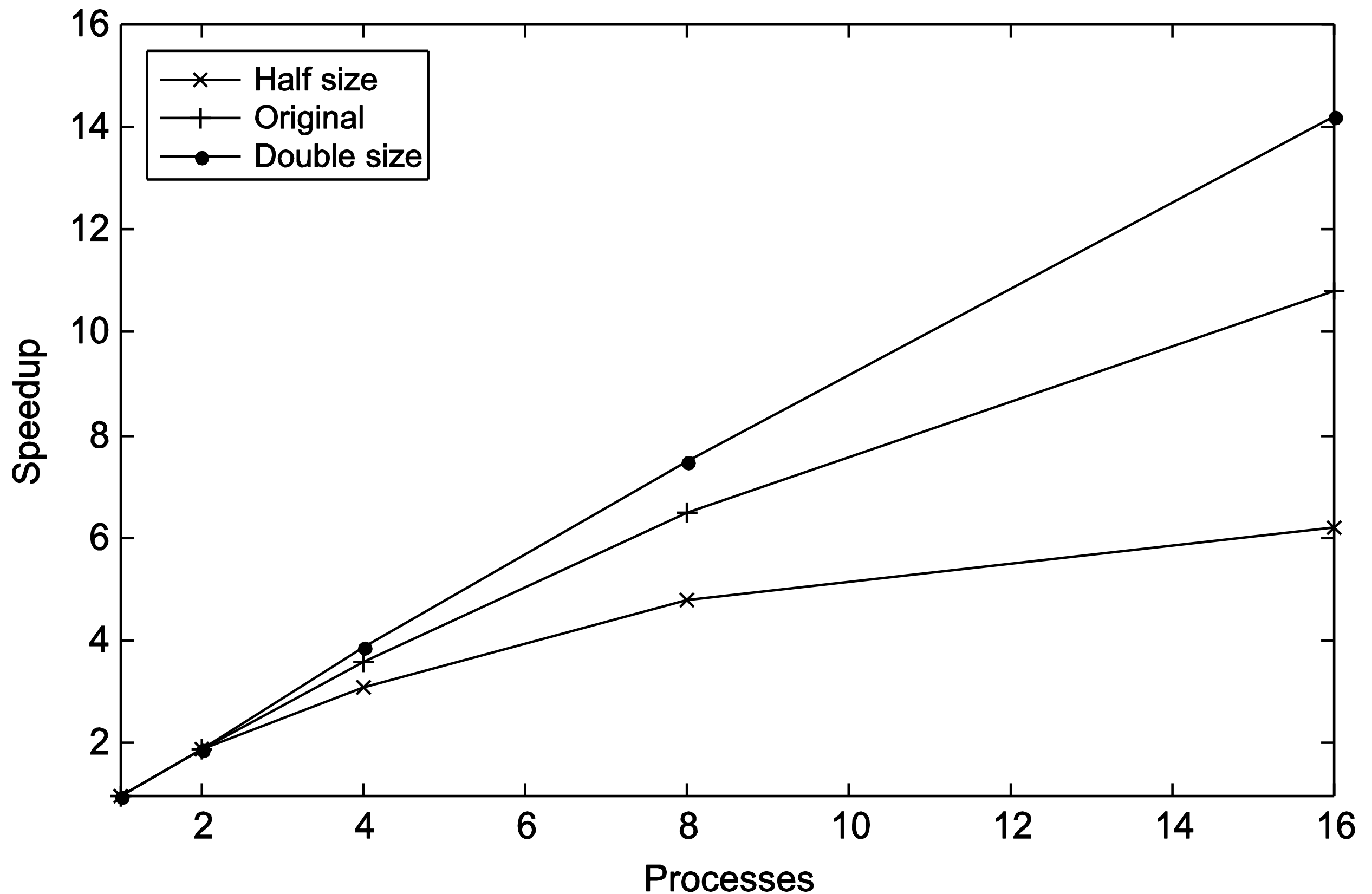
Speedups and efficiencies of a parallel program

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

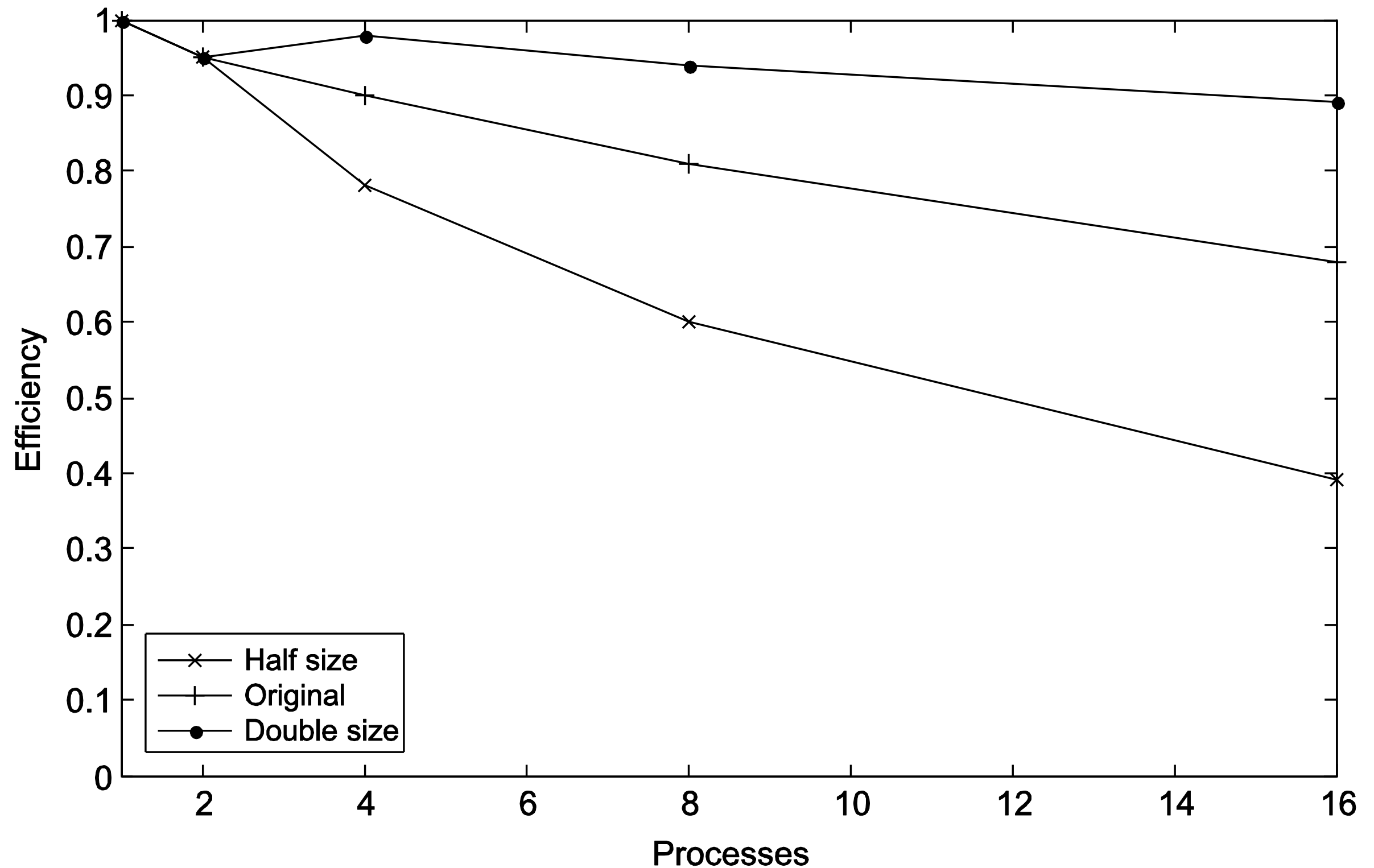
Speedups and efficiencies of parallel program on different problem sizes

	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

Speed-up



Efficiency



Effect of overhead

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

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.

Example

We can parallelize 90% of a serial program.

Parallelization is “perfect” regardless of the number of cores p we use.

$T_{\text{serial}} = 20$ seconds

Runtime of parallelizable part is

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

Example (cont.)

Runtime of “unparallelizable” part is

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

Overall parallel run-time is

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

Example (cont.)

Speed up

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

Scalability

Suppose that we increase the number of processes/threads, if we can find a corresponding rate of increase in the problem size so that the program always has efficiency E , then the program is **scalable**.

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

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

MPI_Wtime

omp_get_wtime

Taking Timings

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

Taking Timings

```
1 shared double global_elapsed;
2 private double my_start, my_finish, my_elapsed;
3 ...
4 /* Synchronize all processes/threads */
5 Barrier();
6 my_start = Get_current_time();
7
8 /* Code that we want to time */
9 ...
10
11 my_finish = Get_current_time();
12 my_elapsed = my_finish - my_start;
13
14 /* Find the max across all processes/threads */
15 global_elapsed = Global_max(my_elapsed);
16 if (my_rank == 0)
17     printf ("The elapsed time = %e seconds \n", global_elapsed);
```