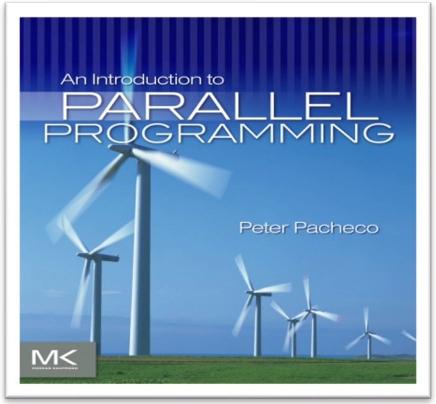
14013204-3 - PARALLEL COMPUTING



Chapter 2

Parallel Software And Performance

Roadmap

- Parallel software
- Measuring Performance
- Sources of performance loss
- Performance trade-offs

PARALLEL SOFTWARE

The burden is on software

- Parallel hardware has arrived.
  - Virtually all desktop and server systems use multicore processors. Nowadays even mobile phones and tablets make use of multicore processors.
- The situation for parallel software is in flux.
  - Most system software makes some use of parallelism, and many widely used application programs (e.g., Excel, Photoshop, Chrome) can also use multiple cores.
  - However, there are still many programs that can only make use of a single core, and there are many programmers with no experience writing parallel programs.

The burden is on software

- Can no longer rely on Hardware and compilers to provide a steady increase in application performance.
- Software developers must learn to write applications that exploit shared- and distributed-memory architectures and MIMD and SIMD systems.
- we'll take a look at some of the issues involved in writing software for parallel systems.
- 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

- We will mainly focus on **SPMD** programs.
- A SPMD program consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.
- Implement both **Task parallel** and **Data parallel** programs.

```
if (I'm thread process i)
    do this;
else
    do that;
```

Coordinating the processes/threads

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. **Assign (allocate)** the work to processes/threads.
3. Arrange for the processes/threads to **synchronize**.
4. Arrange for **communication** among processes/threads.
   - These last two problems are often **interrelated**. For example, in distributed-memory programs, we often implicitly synchronize the processes by communicating among them, and in shared-memory programs, we often communicate among the threads by synchronizing them.

```
double x[n], y[n];
…
for (i = 0; i < n; i++)
    x[i] += y[i];
```

- **Parallelization:** The process of converting a serial program or algorithm into a parallel program.
- **Load Balancing:** The process of dividing the work among the processes/threads so that each process/thread gets roughly the same amount of work.

Shared Memory

- In shared-memory programs, variables can be **shared** or **private**.
  - Shared variables can be read or written by any thread.
  - Private variables can ordinarily only be accessed by one thread.
- Communication among the threads is usually done through shared variables, so communication is **implicit** rather than **explicit**.

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

- In any MIMD system in which the processors execute asynchronously it is likely that there will be nondeterminism.
- A computation is nondeterministic if a given input can result in different outputs.
- If multiple threads are executing independently, the relative rate at which they'll complete statements **varies from run to run**, and hence the **results of the program may be different from run to run**.

```
. . .
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`

- The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location **x**.
- When threads or processes attempt to simultaneously access a shared resource, and the accesses can result in an error, we often say the program has a **race condition**, because the threads or processes are in a "**race**" to carry out an operation.
- That is, **the outcome of the computation depends on which thread wins the race.**

Nondeterminism

- **Critical section:** is a block of code that can only be executed by one thread at a time.
- It's usually our job as programmers to ensure **Mutually Exclusive** access to a critical section.
  - In other words, we need to ensure that if one thread is executing the code in the critical section, then the other threads are excluded.
- **Mutual exclusion lock (mutex, or simply lock):** The most commonly used mechanism for ensuring mutual exclusion. A mutex is a special type of object that has support in the underlying hardware.
  - No pre-determined order on the threads.
  - The code in the critical section is sequential.

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

busy-waiting

- There are alternatives to mutexes. In busy-waiting, a thread **enters a loop, whose sole purpose is to test a condition.**
- In our example, suppose there is a shared variable ok_for_1 that has been initialized to false. Then something like the following code can ensure that thread 1 won't update x until after thread 0 has updated it:

```
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

- In distributed-memory programs, the cores can directly access only their own, private memories.
- There are several APIs that are used. The most widely used is **message-passing**.
- distributed-memory programs are usually executed by starting multiple processes rather than multiple threads.
- A message-passing API provides (at a minimum) a **send** and a **receive** function. Processes typically identify each other by ranks in the range 0, 1, . . . , p − 1, where p is the number of processes.

Message-Passing

sprintf in C is a library function used for formatted output to a string. It works similarly to printf, which prints to the console, but sprintf stores the output in a character buffer (string) specified as its first argument.

```
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 ) ;
}
```

Input and Output

- We'll be making these assumptions and following these rules when our parallel programs need to do I/O:
  - 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*.
    - However, because of the nondeterministic order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout*.
    - The principal exception will be output for debugging a program. In this situation, we'll often have multiple processes/threads writing to *stdout*.
    - Debug output should always include the rank or ID of the process/thread that's generating the output.

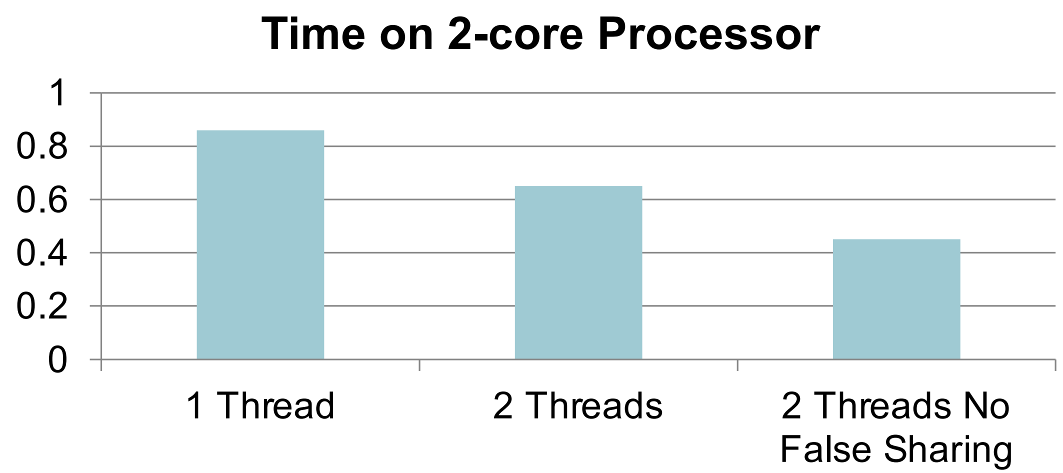
Input and Output

- 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

Measuring Performance

- Basic measure is *execution time*



Measuring Performance

- Performance is usually main aim of parallel computing
  - Some problems take too long on single processor.
- Usually, the best we can hope to do is to equally divide the work among the cores, with no additional work for the cores.
- If we succeed in doing this, and we run our parallel program with p **cores**, one thread or process on each core, then our **parallel program** will run p **times faster** than the **serial program**.
- This is often **difficult to achieve**, e.g.
  - Splitting work, Data dependencies, Load balancing, False sharing, etc.
- Let's define performance formally.

Measuring Performance

- If we call the serial **run-time** T_{serial} and our parallel run-time $T_{parallel}$, then the best we can hope for is

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

- When this happens, we say that our parallel program has **linear speedup**.
- In practice, we usually **don't get perfect linear speedup**, because the use of multiple processes/threads almost invariably introduces some **overhead** ((*The amount of time required to coordinate parallel tasks, as opposed to do useful work*)).
  - It's likely that the overheads will increase as we increase the number of processes or threads.

Sequential Code / Parallel Code

```
int sum = 0;
for(int i = 0; i < size; i++) {
        sum += array[i];
    //sum=sum+array[i];
}
```

```
int numThreads = 2; //Assume one thread per core, and 2
    cores
int sum = 0;
int i = 0;
int middleSum[numThreads];
int threadSetSize = size/numThreads
//Each thread will execute this code with a different
    threadID
for( i = threadID*threadSetSize; i <
    (threadID+1)*threadSetSize; i++)
{
        middleSum[threadID] += array[i];
}
waitForAllThreads(); //Wait for all threads
//Only thread 0 will execute this code
if (threadID==0) {
    for(i = 0; i < numThreads; i++) {
        sum += middleSum[i];}}
```

Measuring Performance

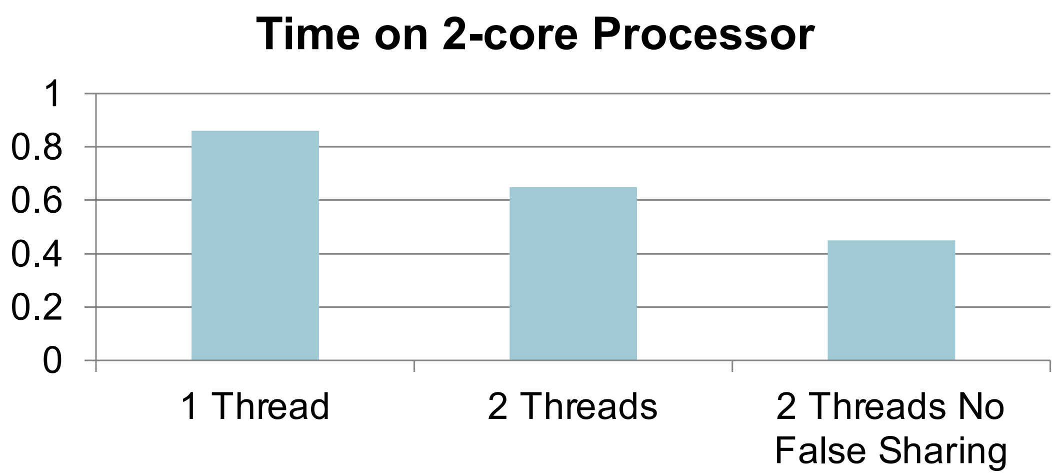
- Number of cores = p
- Serial run-time = T_{serial}
- Parallel run-time = $T_{parallel}$

linear speedup

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

Measuring Performance

- Execution time is not enough
- How do we relate different times?
  - Maybe we know one time is better than another
  - But how much better? What is the relationship?



Speedup of a parallel program

- **Speedup** measures increase in running time due to parallelism.
- Execution time of sequential program divided by execution time of parallel program (for the same problem).

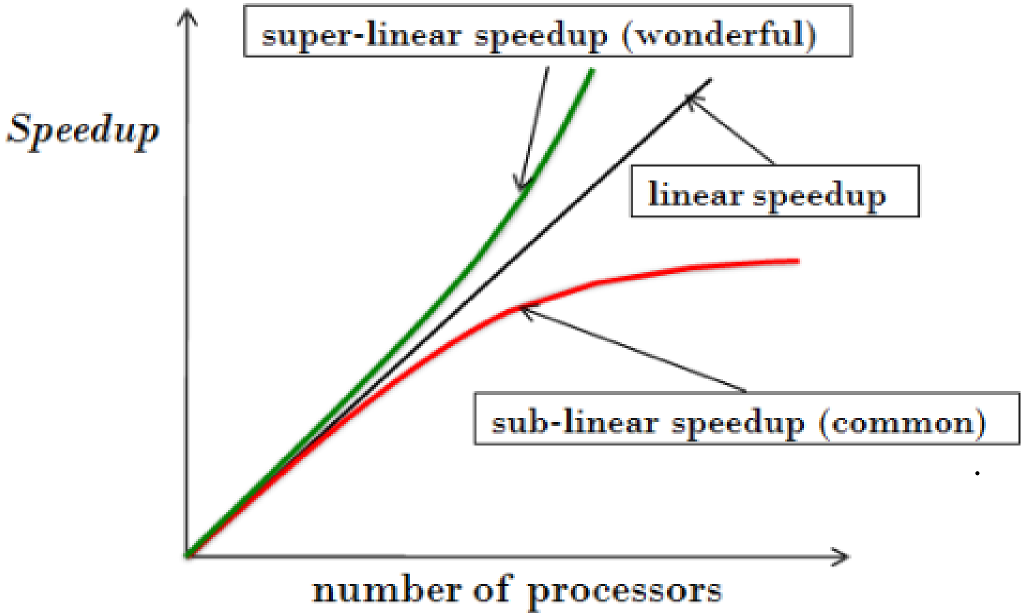
$$S = \frac{T_{serial}}{T_{parallel}}$$

- $T_{serial}(s)$ = Sequential time
- $T_{parallel}(p)$ = Parallel time on P processors
- E.g., T2 is parallel time on 2 processors
- Linear speedup has **S = p**. **Ideally**, **speedup** is linear (perfect). E.g.
  - $\frac{T_s}{T_2} = 2$
  - $\frac{T_s}{T_4} = 4$

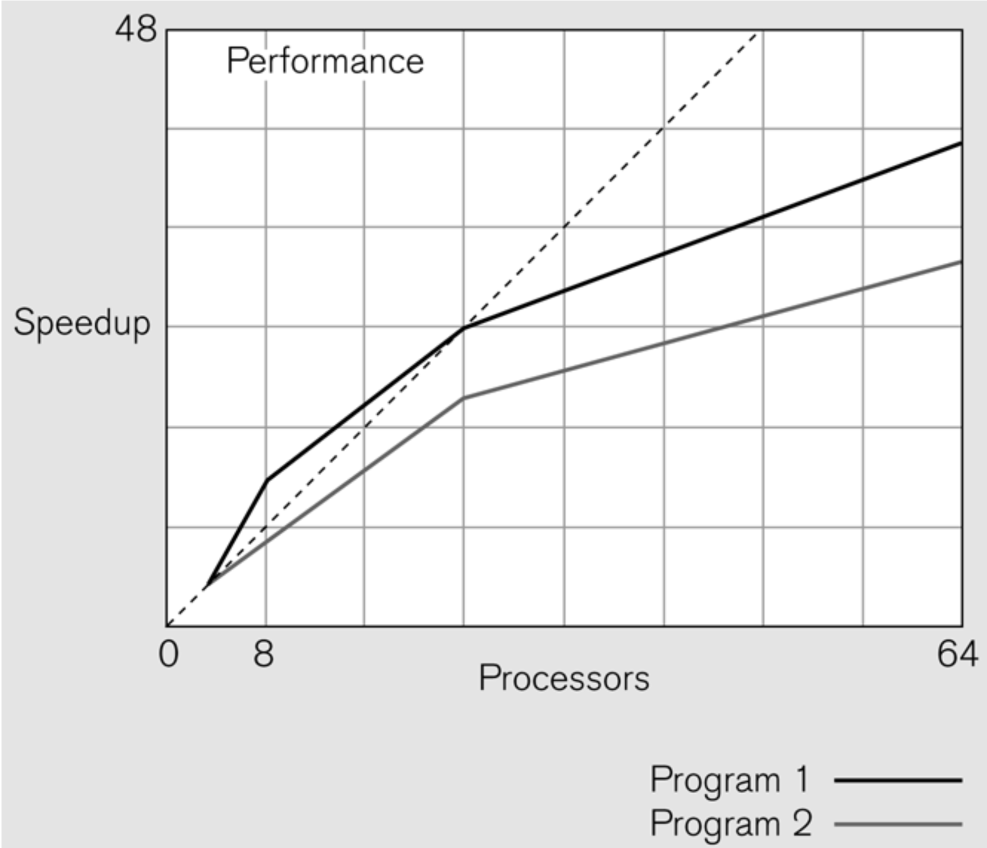
➢ But **this rarely happens**. There are several sources of performance loss. Will be discussed later.

Example Speedup Graph

- super-linear speedup: S > p
- linear speedup: S = p
- sub-linear speedup: S < p



Example Speedup Graph



Speedup

- T_s is time of sequential program
- NOT parallel program with 1 process/thread.
  - Parallel program has extra **overheads**
  - Example: array sum results present only parallel program with different number of threads.
    - **Sequential program**: 0.52 seconds (T_s)
    - **Parallel program, 1 thread**: 0.86 seconds (T_1)

Speedup

- Parallel program, 1 thread is T_1 != T_s
  - **Absolute speedup** = T_s / T_2 = 0.52/0.45 = 1.16
  - **Relative speedup** = T_1 / T_2 = 0.86 / 0.45 = 1.91

Efficiency of a parallel program

linear speedup has $S = p$, which is unusual.

- As p **increases**, we expect S **to become a smaller and smaller fraction** of the ideal, linear speedup p.
- Another way of saying this is that **S/p** will probably get smaller and smaller as p increases.
- This value, **S/p**, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for S, we see that the efficiency is

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

Efficiency of a parallel program

- If the serial run-time has been taken on the same type of core that the parallel system is using, we can think of **efficiency** as **the average utilization of the parallel cores on solving the problem**.
- That is, the efficiency can be thought of as **the fraction of the parallel run-time that's spent, on average, by each core working on solving the original problem**. ((To measure how effectively each processor used)).
- The remainder of the parallel run-time is the **parallel overhead**.
- If **efficiency = 1**, then **linear speedup**
- If **efficiency < 1**
  - Processors **not fully used** (to solve problem).
  - This is the normal case.
  - Many reasons for performance loss.
  - Also, efficiency falls as processors increased.

Effect of overhead

- For example, suppose we have T_{serial} = 24 ms, p = 8, and $T_{parallel}$ = 4 ms. Then,

$$E = \frac{24}{8 \cdot 4} = \frac{3}{4},$$

- On average, each process/thread spends **3/4 · 4 = 3 ms** on solving the original problem, and **4−3 = 1 ms** in **parallel overhead**.
- Many parallel programs are developed by explicitly dividing the work of the serial
- program among the processes/threads and adding in the necessary "parallel overhead," such as mutual exclusion or communication.
- Therefore if $T_{overhead}$ denotes this parallel overhead, it's often the case that:

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

Speedups and efficiencies of a parallel program

- We've already seen that $T_{parallel}$, S, and E, depend on **p**, the number of processes or threads.

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

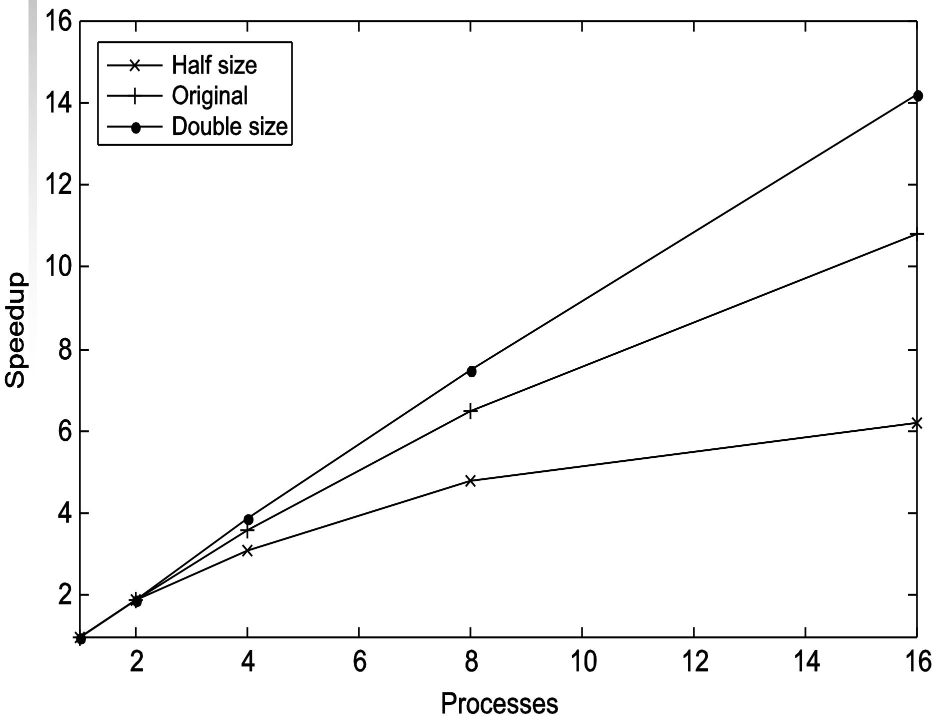
Speedups and efficiencies of parallel program on different problem sizes

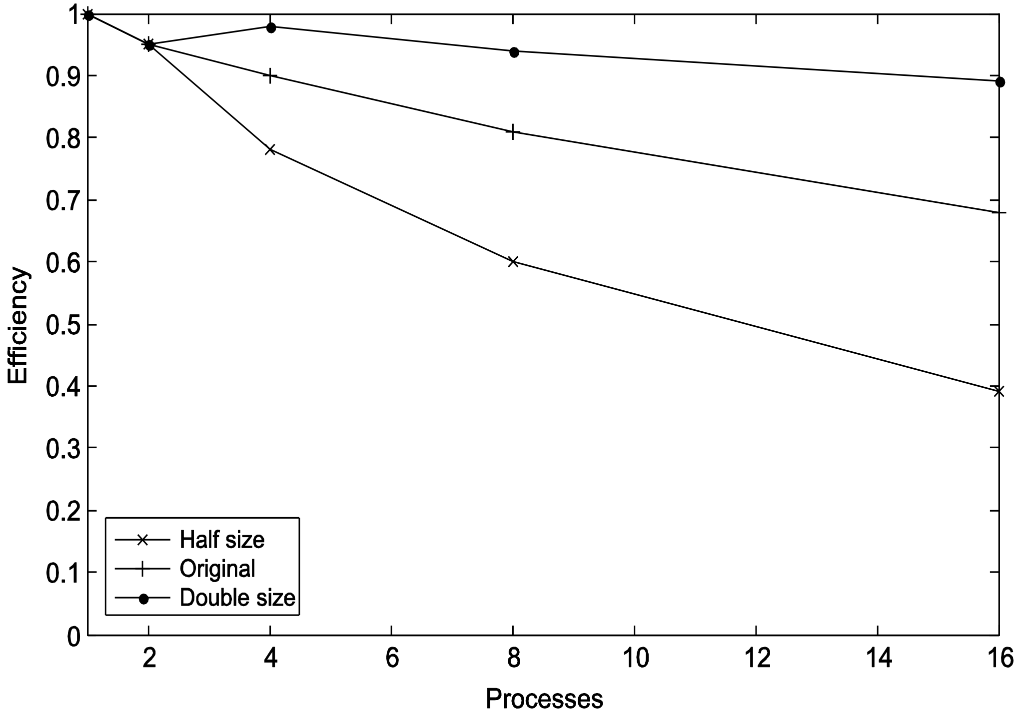
- We also need to keep in mind that $T_{parallel}$, ***S*, *E*, and** T_{serial} all depend on **the problem size**.
- For example, if we **halve and double** the problem size of the program, whose speedups are in the previous slide, we get the speedups and efficiencies shown:

- 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, because in many parallel programs, as the problem size is increased but the number of processes/threads is fixed, the parallel **overhead** $T_{overhead}$ **grows much more slowly than the time spent in solving the original problem.**
  - There's more **computation** work for the processes/threads to do, so the relative amount of **coordination** time should be less.

	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 and Efficiency





Problem Size

- **Fixed-size speedup**
  - Use same problem size for all processor counts.
  - Problem: If large range of processors, then small problem size that fits in memory of one processor may be too small for, e.g., 100000 processors = high overhead = reduced speedup
  - Solution: scaled speedup
- **Scaled speedup**
  - Increase problem size with number of processors.
  - Not straightforward. E.g., how is larger problem size affected by memory and interconnect architecture?

Taking Timings

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

Taking Timings

- We are **interested in the time spent doing specific computation**, **not the entire execution of the parallel program**.
- To measure the **wall clock time** for specific computation:

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

- ***Get current time()*** is a function that's supposed to return the number of seconds that have elapsed since some fixed time in the past.

Taking Timings

- we're usually interested in is a single time: **the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code.**

```
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);
```

Pitfalls (beware)

- **Do not compare speedup on different processors.**
- E.g.
  - Sequential on Pentium 4, T_s = 1 second
  - Parallel on 8x Pentium 1, T_8 = 10 seconds
  - Speedup = T_s/T_8 = 0.1
  - Efficiency = Speedup/P = 0.1/8 = 0.0125
- **These results are meaningless**
  - Must use same hardware

Pitfalls (beware)

- **Cold starts**
  - First run of a program usually slow
  - Page table misses (virtual memory)
  - Cache misses
  - Second, third, ++ runs are faster
  - This means, **warm up memory/cache before you start measuring performance.**

Pitfalls (beware)

- **Related issue: single-run performance**
  - Don't calculate **speedup based on one run**, as runtimes vary
  - Need to **take average of several runs**, e.g. report either a **mean** or a **median** run-time:
    - **Mean**: add up all the data, and then divide this total by the number of values in the data.
    - **Median**: put the values in order, then find the middle value.
  - **Example:** Find the mean, and median for the following list of values:

    13, 18, 13, 14, 13, 16, 14, 21, 13

1. **Mean**: (13 + 18 + 13 + 14 + 13 + 16 + 14 + 21 + 13) ÷ 9 = 15
2. **Median**: Rewrite the list in order: 13, 13, 13, 13, **14**, 14, 16, 18, 21

Pitfalls (beware)

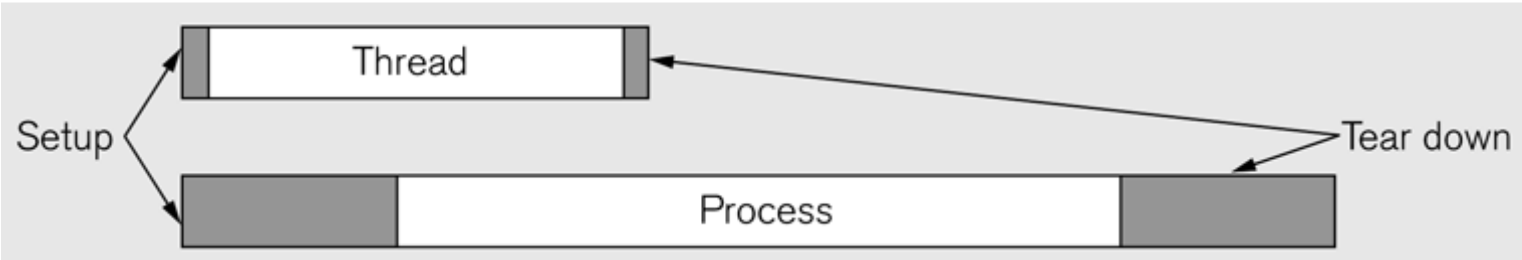
- **I/O Accesses**
  - **Network, hard disk accesses take a lot of time**
  - If program is regularly making such accesses, **performance is difficult to measure**, because **execution times vary** a lot.
  - E.g.
    - Sequential: 6s
    - Parallel 2 threads:
      - 1st run: 5s (**cold start)**
      - 2nd run: 2s (**warm up)**
      - 3rd run: 4s (**another program accessing disk**)
  - As a practical matter, since our programs won't be designed for high-performance I/O, we'll usually not include I/O in our reported run-times.

Sources of Performance Loss

- Overhead
- Contention for resources
- Idle time
- Non-parallelizable computation

Overhead

- **Any performance cost of parallelizing** a sequential program
- Any cost that is incurred in **the parallel solution** but not in the sequential one.
- **Process + thread creation/destruction**
  - Sequential program only creates one process.
  - Parallel program creates ***at least* one process (maybe more)**, and ***at least* one thread (usually more).** Also have to destroy them at the end.
  - **This results in reduced performance**.
    - with array sum: Ts = 0.52, T1 = 0.86



Overhead

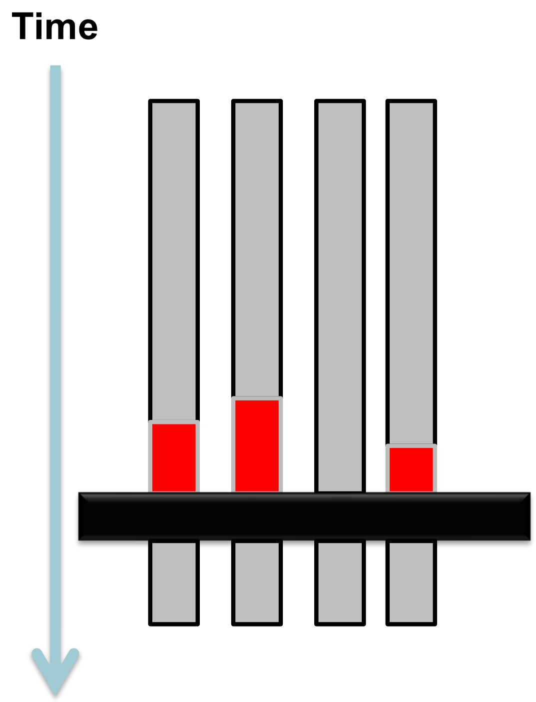
- Four main sources
  1. Communication
  2. Synchronization
  3. Computation
  4. Memory

Overhead

- **Communication**
  - **Threads and processes** communicate.
  - Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access.
  - False sharing in shared memory systems is also communication. (**unwanted**)
  - Sequential program doesn't use communication (1 process).
- **Synchronization**
  - Usually **causes one thread to wait for another.**
  - Shared-memory programs will almost always have **critical sections.**
    - Which will require that we use some **mutual exclusion** mechanism, such as a **mutex**. The calls to the mutex functions are the overhead.
  - The use of the such synchronization mechanisms forces the parallel program to serialize execution of the critical section.
  - Sequential program doesn't use synchronization (1 process)
- The overheads **will increase as we increase the number of processes or threads**. For example, more threads will probably mean more threads need to access a critical section, and more processes will probably mean more data needs to be transmitted across the network.

Synchronization Cost Example

- **Barrier overhead**
  - **Gray: threads working**
  - **Red: threads waiting**
    - Load balance
  - **Black: communication**:
    - message all threads to resume



Overhead

- **Computation**
  - Parallel solution usually **does extra computation.**
  - Example from array sum: *middle sums.*
- **Memory**
  - **Parallel solution** may use **more memory.**
  - Array sum example: **padding memory** to remove false sharing.
  - This actually improves performance but increases **cost of memory.** May not always be possible if memory limit reached by program.

Contention

- **Competition for a shared resource**
  - E.g., bus in the computer's architecture.
- **Parallel program increases contention**
  - More processors/threads accessing shared resources.
  - Can affect processors that are working on different areas of the program.
  - E.g., imagine **two processors causing contention on a bus** due to **false sharing.**
  - A third processor's **memory request latency is increased**, even if it is not involved in the false sharing.

Idle Time

- **Ideally, processors working all the time.**
- Not usually the case
  - **Waiting for data from memory**
  - **Load imbalance (not enough work)**

Non-Parallelizable Code

- **Some code cannot be parallelized**
  - Code with **dependencies** is one example
  - This code **has to be executed sequentially**
  - This code **limits the benefit of parallelization**

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:** Suppose that we can parallelize 90% of a serial program.
- Suppose that the speedup of the parallelized part is perfect, which means that the speedup is always **p** regardless of the number of **p**.
  - If the sequential run-time is $\mathbf{T_{serial} = 20}$ seconds
  - Runtime of parallelizable part is $\mathbf{0.9 \times T_{serial} / p = 18 / p}$
  - Runtime of "unparallelizable" part is $\mathbf{0.1 \times T_{serial} = 2}$
  - **Overall parallel run-time is:** $\mathbf{T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2}$
  - **Speed up:**

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

Amdahl's Law

- Now **as p gets larger and larger** $0.9 \times \frac{T_{sequential}}{p} = \frac{18}{p}$ **gets closer and closer to 0**.
- So **the total parallel runtime** can't be **smaller than** $0.1 \times \mathbf{T_{sequential}} = 2$
- That is, the **denominator in S** can't be smaller than $0.1 \times \mathbf{T_{sequential}} = 2$

  The fraction S must therefore be smaller than: $S \leq \frac{T_{sequential}}{(0.1 \times T_{sequential}\}} = \frac{20}{2}$

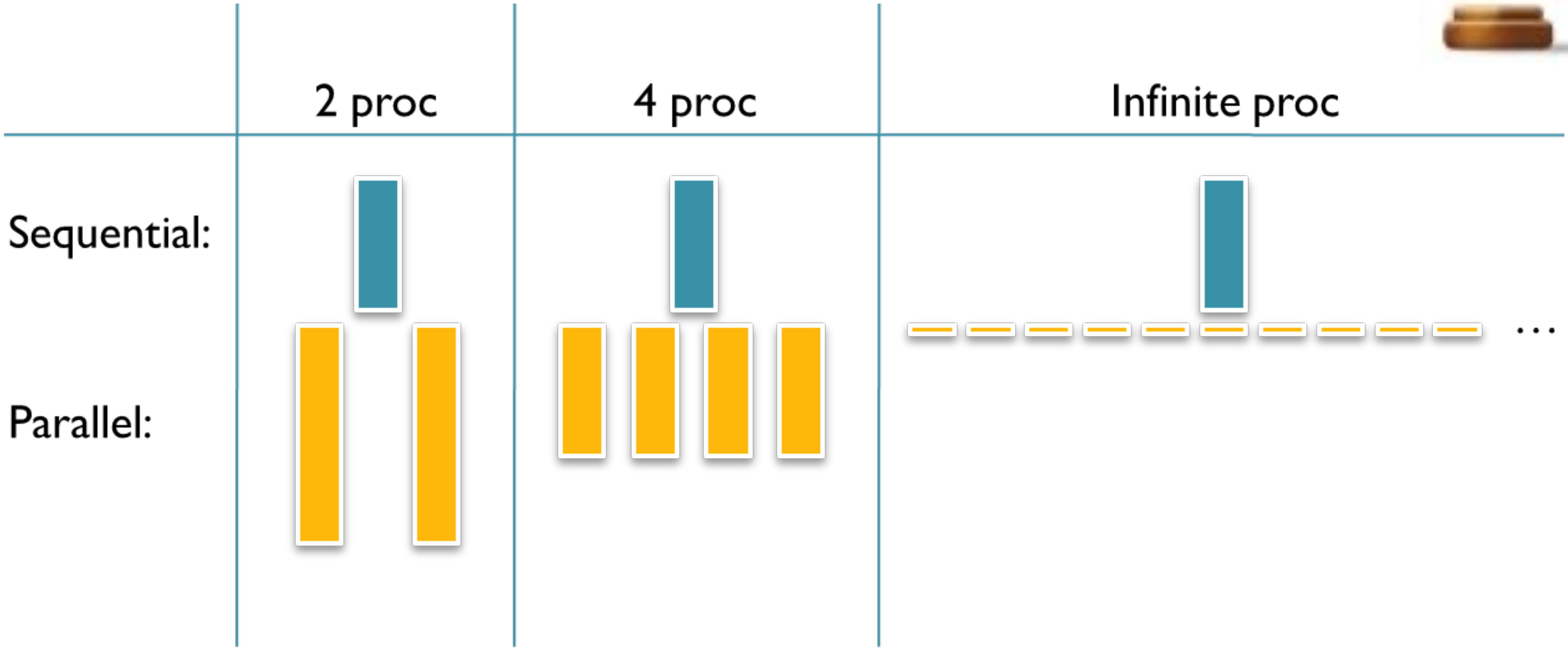
  $= 10$

  that is $\boldsymbol{S} \leq \mathbf{10}$
- This is saying **that even though we've done a perfect job in parallelizing 90% of the program**, and even if we have, say, **1000 cores**, **we'll never get a speedup better than 10**.

Amdahl's Law

- **More generally**, if a **fraction r** of our **sequential program remains un-parallelized**, then Amdahl's law says **we can't get a speedup better than $\frac{1}{r}$.**
  In our example, $r = 1 - 0.9 = \frac{1}{10}$ so we couldn't get a **speedup better than 10**.
- Thus, even if $\boldsymbol{r}$ is quite small, e.g. say $\frac{1}{100}$, and we have a system with **thousands** of cores, we can't possibly get a speedup better than 100.

Amdahl's Law



Amdahl’s Law

- Shall we give up? **No**
- There are **several reasons** not to be too **worried by Amdahl’s law**:

1. **it doesn’t take into consideration the problem size**.
    - **Often when we have more processors, we also increase the problem size**.
    - E.g. **increase amount of data to be computed**.
    - This reduces sequential part of the total problem, so we can get more speedup.
2. There are **thousands of programs** used by scientists and engineers that routinely **obtain huge speedups on large distributed-memory systems**.
3. In many cases, **obtaining a speedup of 5 or 10 is more than adequate**, especially if the effort involved in developing the parallel program wasn’t very large.

Scalability

- The word "scalable" has a wide variety of informal uses.
- A program is scalable if, by increasing the power of the system it's run on (e.g., increasing the number of cores), we can obtain speedups over the program when it's run on a less powerful system (e.g., a system with fewer cores).
- Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain an efficiency E.
- Suppose we now increase the number of processes/threads that are used by the program. 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 program is *strongly scalable*.
  - If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is *weakly scalable*.

Performance Tradeoff

- We have seen there are many factors that make it difficult to **write high performance parallel programs**
  - But it gets worse (more complex). . .
  - **Overheads**, **Amdahl's Law**, **idle time**, etc. reduce performance.
  - However, trying **to reduce the effect of one factor can increase the effect of another factor.**

Performance Tradeoff

- Three main trade-offs
  - Communication vs. computation
  - Memory vs. parallelism
  - Overhead vs. parallelism

Communication vs. Computation

- **Communication** can **add large overhead**. E.g., NUMA
- **Overlap communication and computation**
  - Identify communication that is independent of the computation.
  - Send communication message, and then do computation while waiting for message reply. Hide communication latency.
- **Can be reduced by extra computation**
  - Redundant/extra computation
    - Rather than requesting some data from memory, processor calculates data values by itself (if this is possible).
    - Objective: do extra computation, if it costs less than communication

Memory vs. Parallelism

- We saw that if we use **more memory, we can improve parallelism**
  - **Privatization**
    - Make **variables** to **remove false dependencies**
  - **Padding**
    - **Spread variables out in memory** (by allocating extra memory) to remove false sharing
    - Sometimes the wasted space can be used for other data, but makes code more complex.
- Improving memory efficiency can reduce parallelism.

Overhead vs. Parallelism

- Want to **minimize overhead**
- Increasing parallelism increases overhead
  - E.g., **create more threads to use more procs**.
  - But creating threads adds overhead
  - **More threads increase contention**
  - If the problem size is fixed then:
    - **Increase threads**: less work per thread.
    - **Overhead larger part** of each thread's time.
    - **Less chance** to hide communication cost.

Summary

- **Parallel software**
  - SPMD
  - Coordinating the processes/threads
  - Shared-memory
  - Distributed-memory
  - Input and output
- **Measuring performance**
  - Speedup = Ts / Tp
  - Efficiency = speedup / P
  - Amdahl's law
  - Scalability
  - Taking timings of MIMD programs

Summary

- **Sources of performance loss**
  - **Overheads**
    - Communication
    - Synchronization
    - Computation
    - Memory
  - **Non-parallelizable code**
  - **Contention**
  - **Idle time**
- **Performance trade-offs**
  - Communication vs. computation
  - Memory vs. parallelism
  - Overhead vs. parallelism