

Week 10: Performance of Parallel Programs

COMP2129

COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Outline

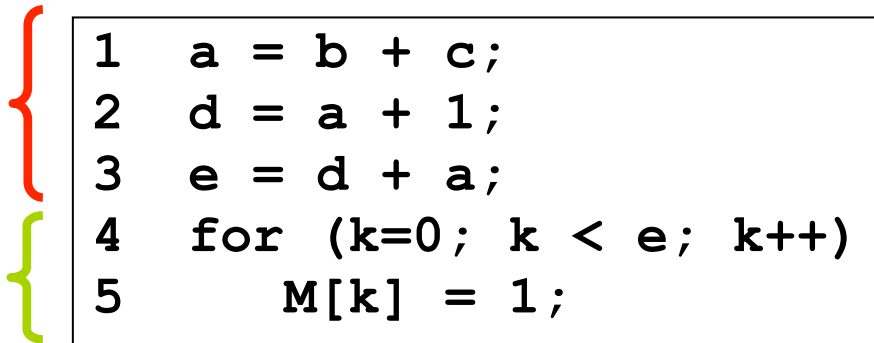
- Amdahl's Law
- Load Balancing
- Measuring Performance
- Sources of Performance Loss
 - Example Case-study
- Profiling
- Reasoning about Performance

Limits to Performance Scalability

- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

Sequential part: cannot be parallelized because of data dependencies.

Parallel part: no data dependence, the different loop iterations (Line 5) can be executed in parallel.



```
1  a = b + c;  
2  d = a + 1;  
3  e = d + a;  
4  for (k=0; k < e; k++)  
5      M[k] = 1;
```

Limits to Performance Scalability

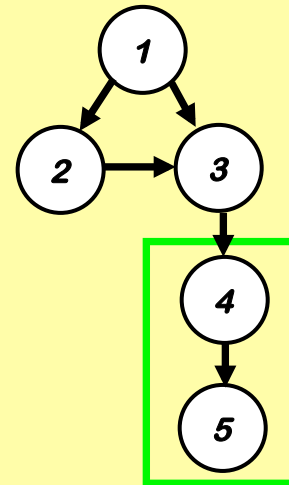
- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

Sequential part: cannot be parallelized because of data dependencies.

Parallel part: no data dependence, the different loop iterations (Line 5) can be executed in parallel.

```
1  a = b + c;  
2  d = a + 1;  
3  e = d + a;  
4  for (k=0; k < e; k++)  
5      M[k] = 1;
```

Dependencies:



Limits to Performance Scalability

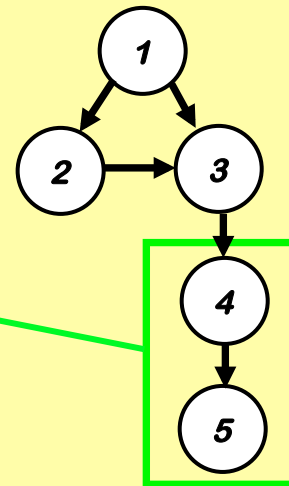
- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

Sequential part: cannot be parallelized because of data dependencies.

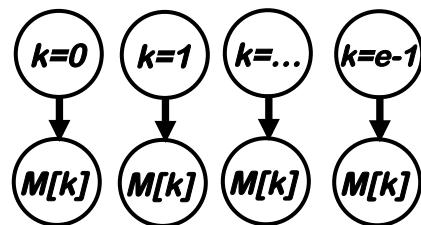
Parallel part: no data dependence, the different loop iterations (Line 5) can be executed in parallel.

```
1  a = b + c;  
2  d = a + 1;  
3  e = d + a;  
4  for (k=0; k < e; k++)  
5      M[k] = 1;
```

Dependencies:



Unroll loop:

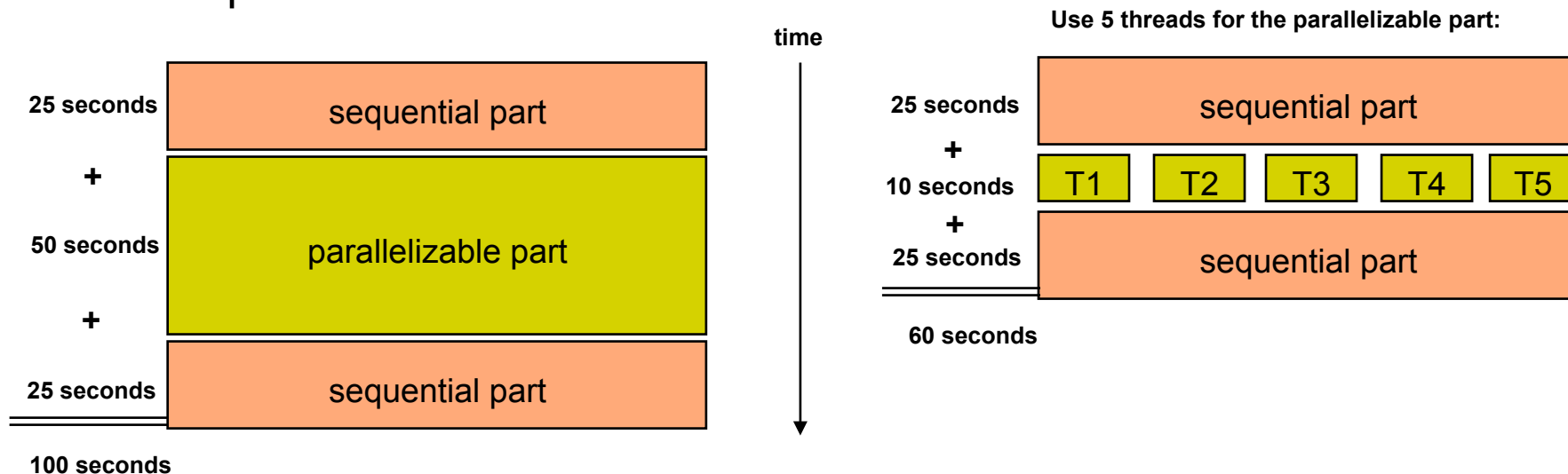


Amdahl's Law

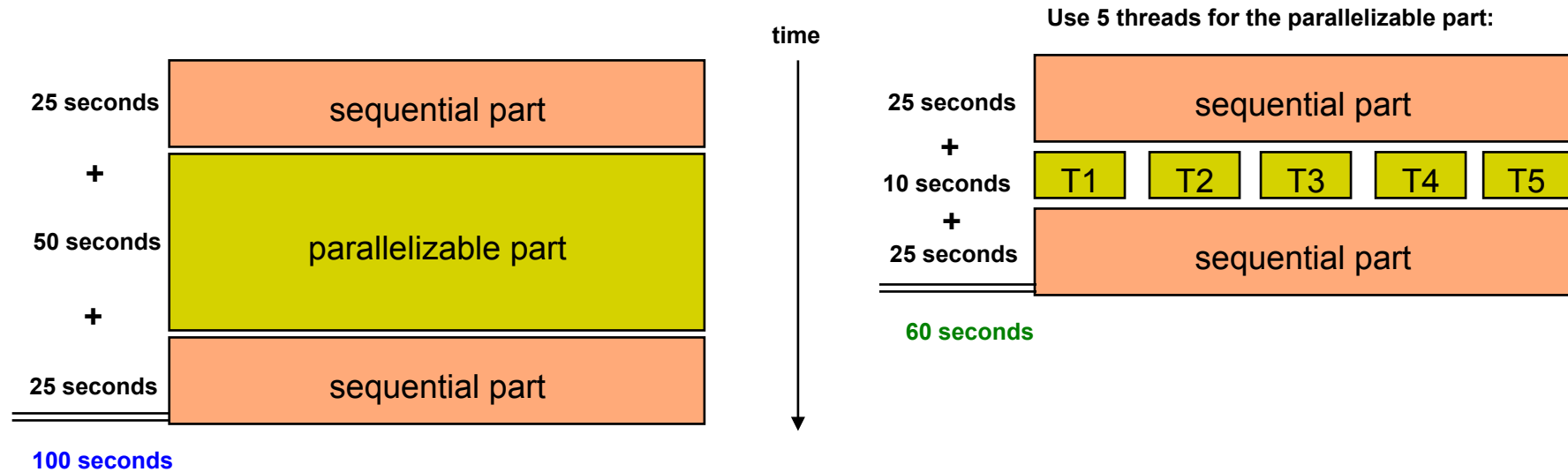
In 1967, Gene Amdahl, then IBM computer mainframe architect, stated that

“The performance improvement to be gained from some faster mode of execution is limited by the fraction of the time that the faster mode can be used.”

- “Faster mode of execution” here means [program parallelization](#).
- The potential speedup is defined by the fraction of the code that can be parallelized.

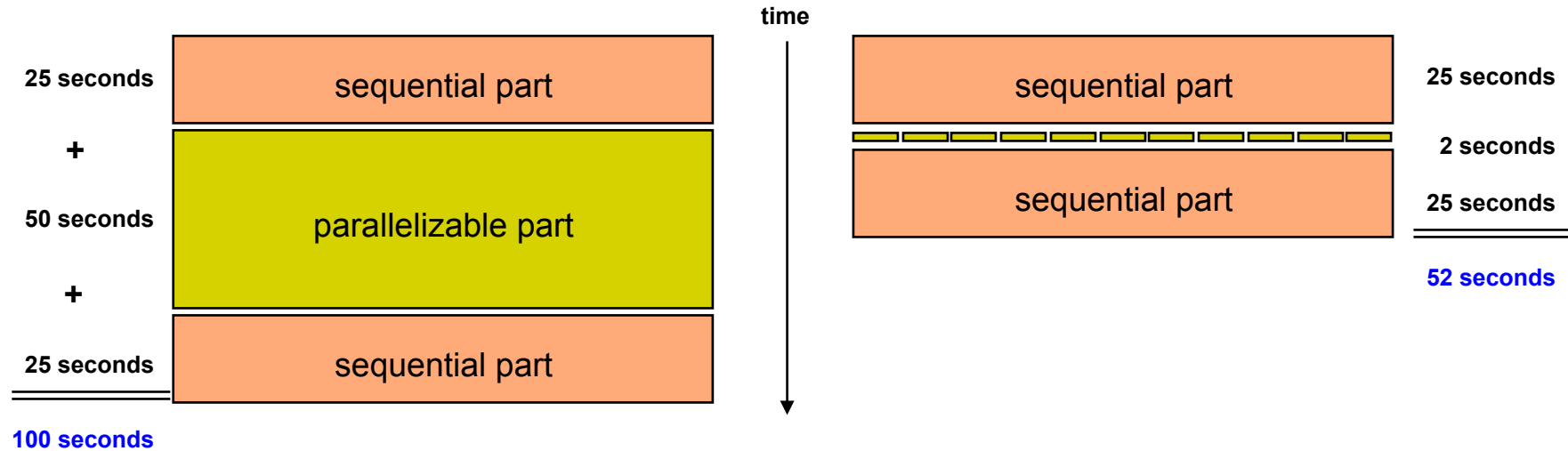


Amdahl's Law (cont.)



- Speedup = $\text{old running time} / \text{new running time}$
= 100 seconds / 60 seconds
= 1.67
- The Parallel version is 1.67 times faster than the sequential version.

Amdahl's Law (cont.)



- We may use more threads executing in parallel for the **parallelizable part**, but the **sequential part** will remain the same.
 - **The sequential part of a program limits the speedup that we can achieve!**
- Even if we *theoretically* could reduce the parallelizable part to 0 seconds, the best possible speedup in this example would be
$$\text{speedup} = 100 \text{ seconds} / 50 \text{ seconds} = 2.$$

Amdahl's Law (cont.)

- p = fraction of work that can be parallelized.
- n = the number of threads executing in parallel.

$$new_running_time = (1-p) * old_running_time + \frac{p * old_running_time}{n}$$

$$Speedup = \frac{old_running_time}{new_running_time} = \frac{1}{(1-p) + \frac{p}{n}}$$

- Observation: if the number of threads goes to infinity ($n \rightarrow \infty$), the speedup becomes $\frac{1}{1-p}$.
- Parallel programming pays off for programs which have a **large parallelizable part**.

Amdahl's Law (Examples)

- p = fraction of work that can be parallelized.
- n = the number of threads executing in parallel.

$$\text{Speedup} = \frac{\text{old_running_time}}{\text{new_running_time}} = \frac{1}{(1 - p) + \frac{p}{n}}$$

- Example 1: $p=0$, an embarrassingly sequential program.

$$\text{speedup} = \frac{1}{1 + \frac{0}{n}} = 1 \text{ (no speedup!)}$$

- Example 2: $p=1$, an embarrassingly parallel program.

$$\text{speedup} = \frac{1}{0 + \frac{1}{n}} = n \text{ (the number of processors gives the speedup!)}$$

- Example 3: $p=0.75$, $n = 8$

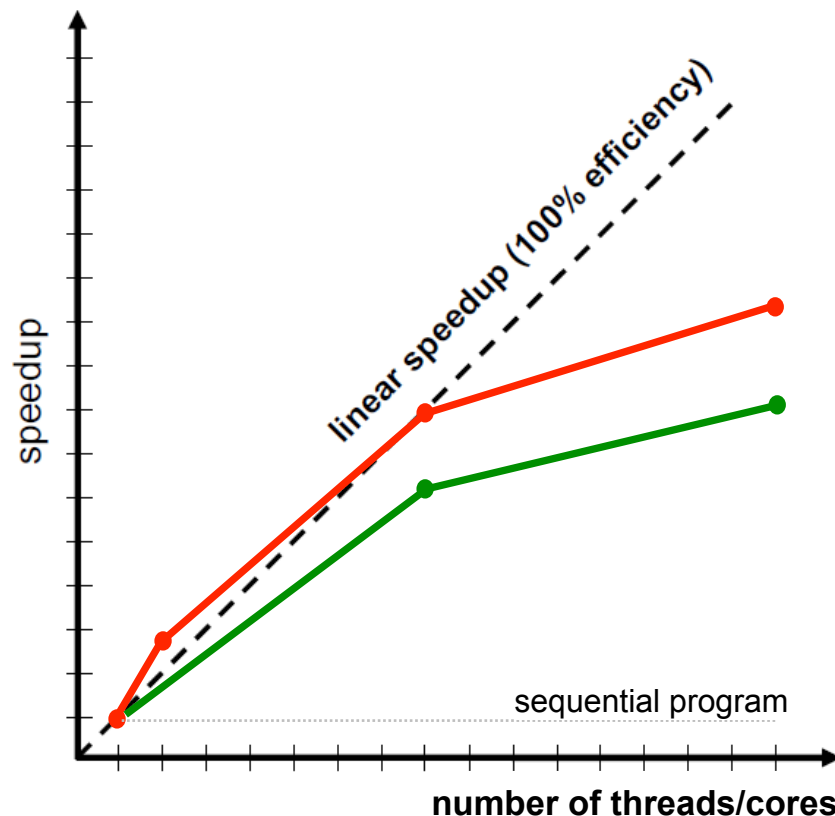
$$\text{speedup} = \frac{1}{0.25 + \frac{0.75}{8}} = 2.91$$

- Example 4: $p=0.75$, $n = \infty$

$$\text{speedup} = \frac{1}{0.25 + \frac{0.75}{\infty}} = 4 \text{ (theoretical upper bound if 25\% of the code cannot be parallelized)}$$

Performance Scalability

- On the previous slides, we assumed that performance directly scales with the number n of threads/cores used for the parallelizable part p .
 - E.g., if we double the number of threads/cores, we expect the execution time to drop in half. This is called **linear speed-up**.
 - However, linear speedup is an “ideal case” that is often not achievable.



- The speedup graph shows that the curves level off as we increase the number of cores.
 - Result of keeping the problem size constant
→ the amount of work per thread decreases
→ overhead for thread creation also becomes more significant.
 - Other reasons possible also (memory hierarchy, highly-contended lock, ...).
- Efficiency = $\frac{Speedup}{n}$
 - Ideal efficiency of 1 indicates linear speedup.
 - Efficiency > 1 indicates super-linear speedup.
- Super linear speedups are possible due to registers and caches.

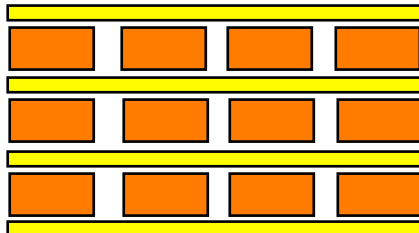
Outline

- Amdahl's Law ✓
- Load Balancing
- Measuring Performance
- Sources of Performance Loss
 - Example Case-study
- Profiling
- Reasoning about Performance

Granularity of parallelism = frequency of Interaction between threads

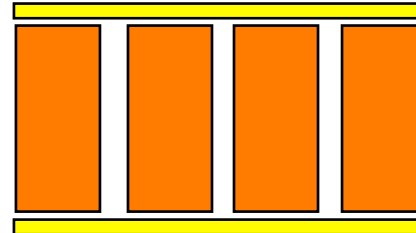
■ Fine-grained Parallelism

- Small amount of computational work between communication / synchronization stages.
- Low computation to communication ratio.
- High communication / synchronization overhead.
- Less opportunity for performance improvement.



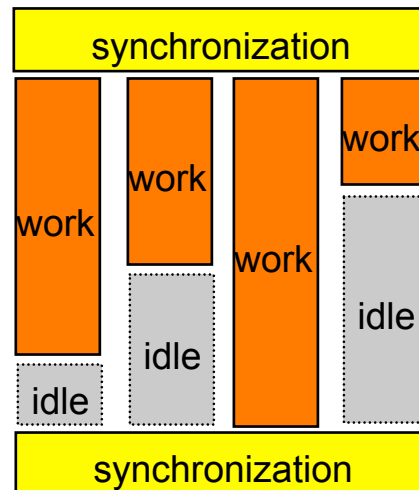
■ Coarse-grained Parallelism

- Large amount of computational work between communication / synchronization stages.
- High computation to communication ratio.
- Low communication / synchronization overhead.
- More opportunity for performance improvement.
- Harder to load-balance effectively.



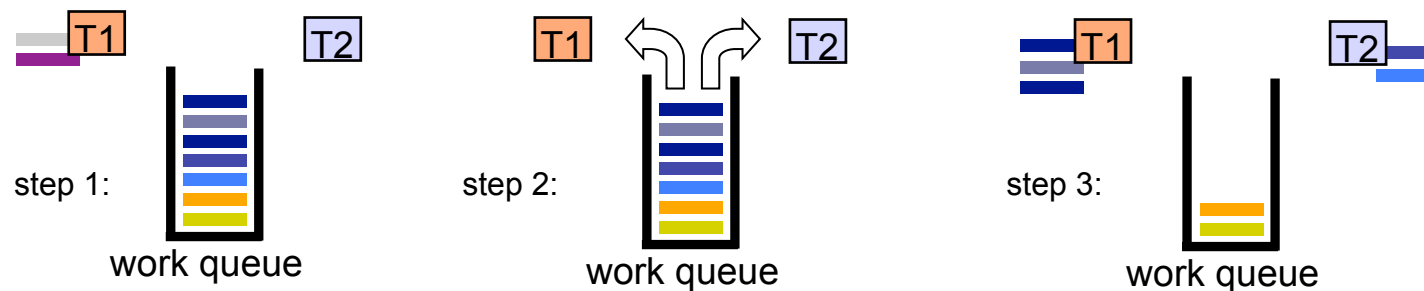
Load Balancing Problem

- Threads that finish early have to wait for the thread with the largest amount of work to complete.
 - This leads to idle times and lowers processor utilization.



Load Balancing

- **Load imbalance**: work not evenly assigned to cores.
 - Underutilizes parallelism!
- The assignment of *work*, not *data*, is key.
- **Static assignment** = assignment at program writing time
 - cannot be changed at run-time.
 - More prone to imbalance.
- **Dynamic assignment** = assignment at run-time.
 - Quantum of work must be large enough to amortize overhead.
 - Example: Threads fetch work from a work queue.



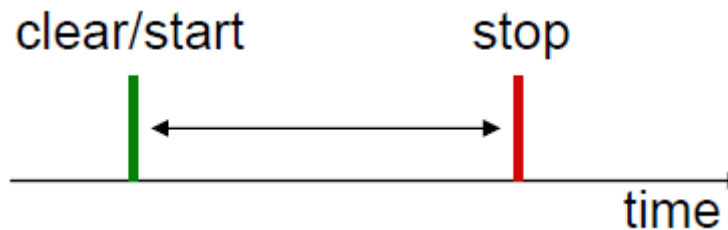
- With flexible allocations, load balance can be solved late in the design programming cycle.

Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance
- Sources of Performance Loss
 - Example Case-study
- Profiling
- Reasoning about Performance

Measuring Performance

- Execution time ... what's time?
- We can measure the time from start to termination of our program.
 - Also called **wallclock time**.



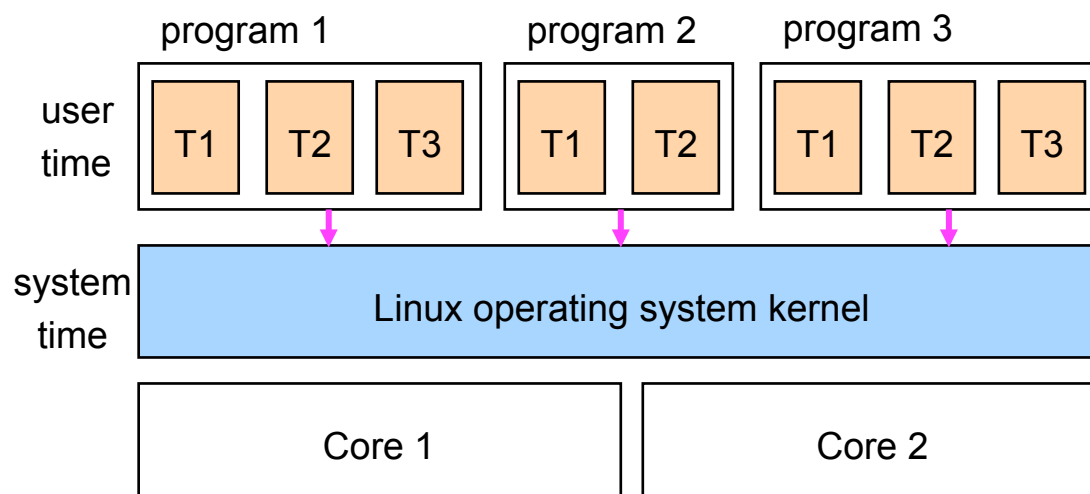
```
gcc foo.c
time ./a.out
real    0m4.159s
user    0m0.008s
sys     0m0.026s
```

elapsed wallclock time

time executed in user mode

time executed in kernel mode
(see next slide)

Difference between user and system time

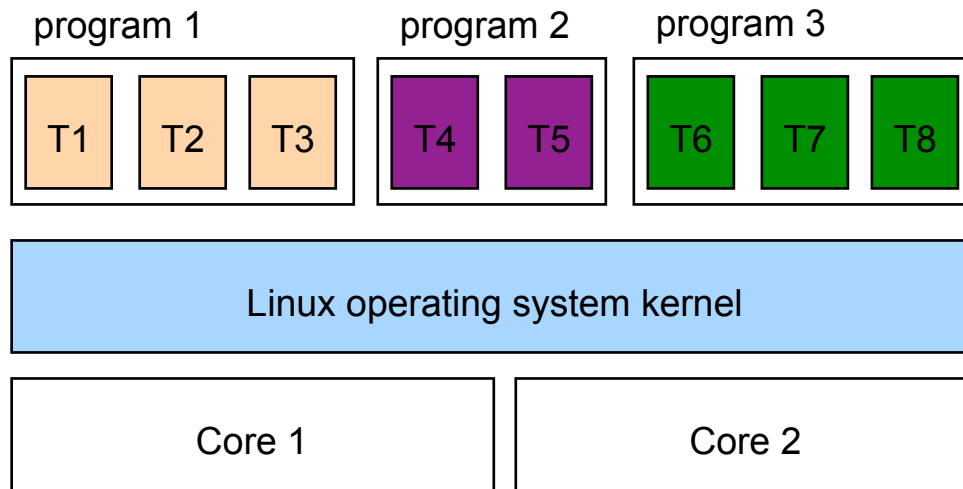


```
#include<stdio.h>

int main() {
    int i, *j, k;
    FILE f = fopen(...);
    i = 255;
    j = malloc(255*sizeof(int));
    for (k=0;k<i;k++)
        *(j+k) = 0;
    ...
}
```

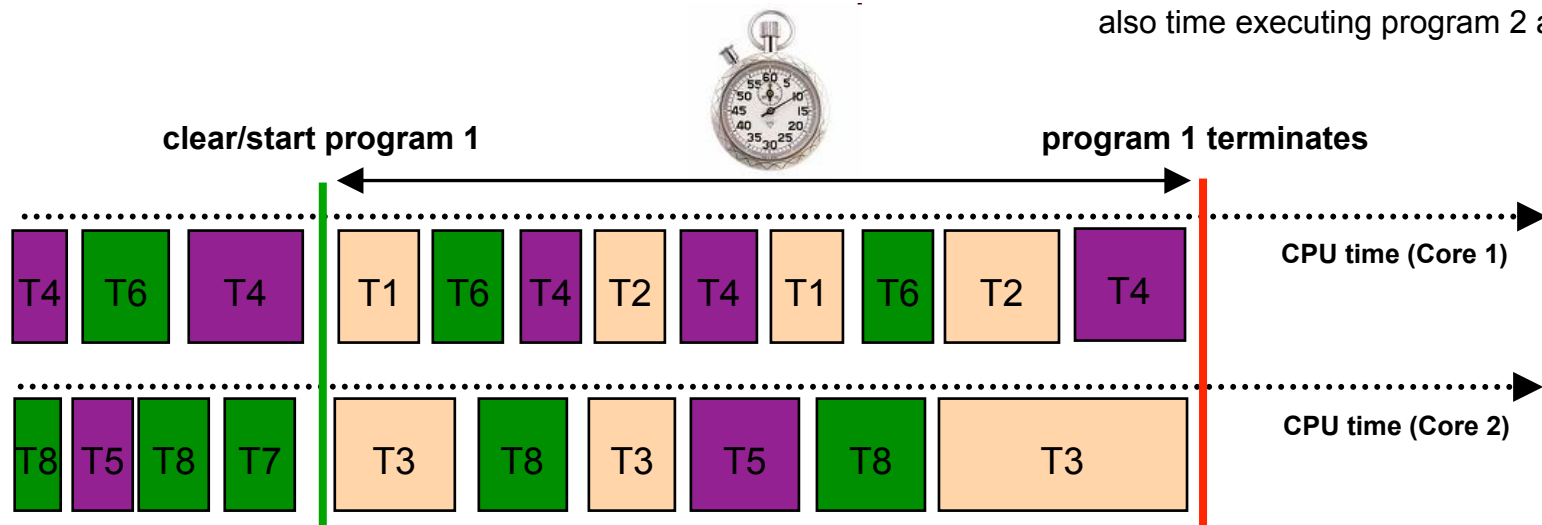
- A program requests services from the operating system (file I/O, dynamic memory, ...) via calls to the operating system kernel.
 - Called system calls (↓).
 - Depicted in **blue** in the above code example.
 - The time spent in system calls is counted as **system time**.
- The rest of the code is executed outside of the system kernel.
 - Counted as **user time**.
 - Depicted in **orange** in the above example.
- Question: why is user time + system time \neq wallclock time ?

Wanted: an unloaded machine



Principal problem with wallclock time:

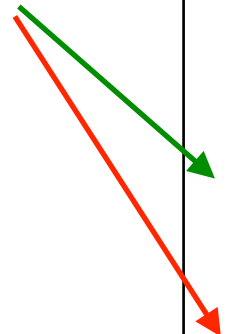
- Different programs share the available CPUs/cores.
- The Linux kernel schedules the threads of all programs on those CPUs/cores.
- Every thread gets to run for a little while (its **timeslice**), then another thread gets to run.
- Programs get in the way of each other. If we are interested in the execution time of program 1, the measured wallclock time contains also time executing program 2 and 3.



Measuring Performance

- On an unloaded machine, wallclock time gives us a crude program performance indication.
- Run the program several times
 - Difference between cold start and warm start
 - Linux file system caches
 - instruction caches, data caches
- Wallclock time does not tell us in which parts of the program we spend time.
- `gettimeofday()` allows us to measure wallclock time for **specific** parts of the program.
- We will study more elaborate performance measurement methods in a dedicated lecture.

```
int main() {  
    double s=0,s2=0; int i,j;  
    for (j = 0; j < T; j++) {  
        for (i = 0; i < N; i++) {  
            b[i] = 0;  
        }  
        cleara(a);  
        memset(a,0,sizeof(a));  
        for (i = 0; i < N; i++) {  
            s += a[i] * b[i];  
            s2 += a[i] * a[i] + b[i] * b[i];  
        }  
    }  
    printf("s %f s2 %f\n",s,s2);  
}
```



record start time

record stop time

Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss
 - Example Case-study
- Profiling
- Reasoning about Performance

Justin Sheehy. **There is no now.** Problems with simultaneity in distributed systems.

ACM queue Vol. 13, issue 3. March 2015.

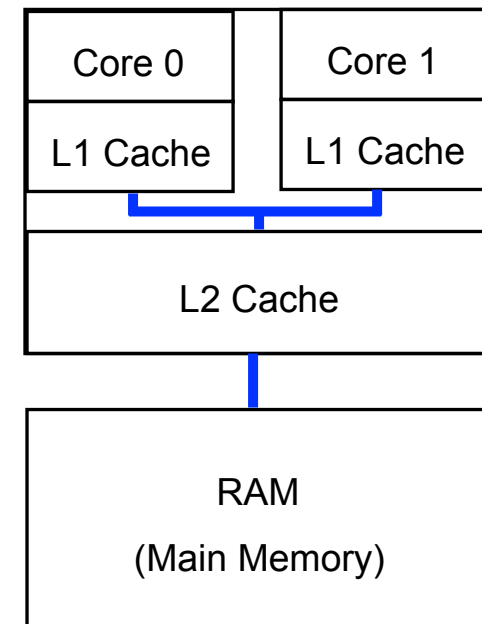
Let's consider a simple problem

- Sum up the numbers in `array[]`
 - Array has `length` values.
- Sequential solution:

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

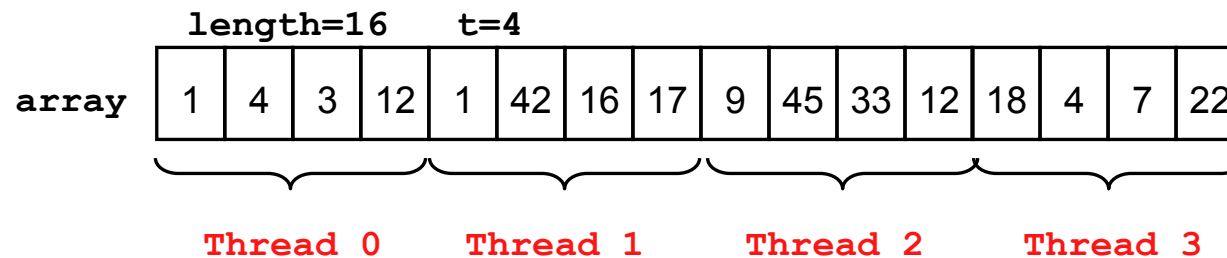
Write a parallel program

- We need to know something about the machine...
- Let's use a multicore architecture!
- Main memory access is extremely slow compared to the processing speed of today's CPUs.
 - A cache is a small but fast memory that is used to store copies of data from the main memory.
 - Holds copies of data from the most recently used locations in main memory.
 - Memory hierarchy:
 - registers→L1 cache→L2 cache→main memory.
 - Cache miss: CPU wants to access data which is not in the cache.
 - Need to access data in level below, eventually in main memory (slow!).



Divide into several parts...

- Solution for several threads: divide the array into several parts.



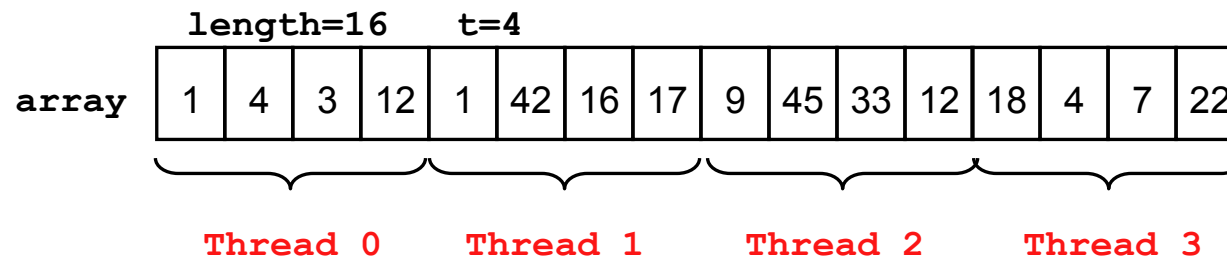
```
int length_per_thread = length / t;
int start = id * length_per_thread;
sum = 0;

for(i=start; i<start+length_per_thread; i++){
    sum = sum + array[i];
}
```

What's wrong?

Divide into several parts...

- Solution for several threads: divide the array into several parts.



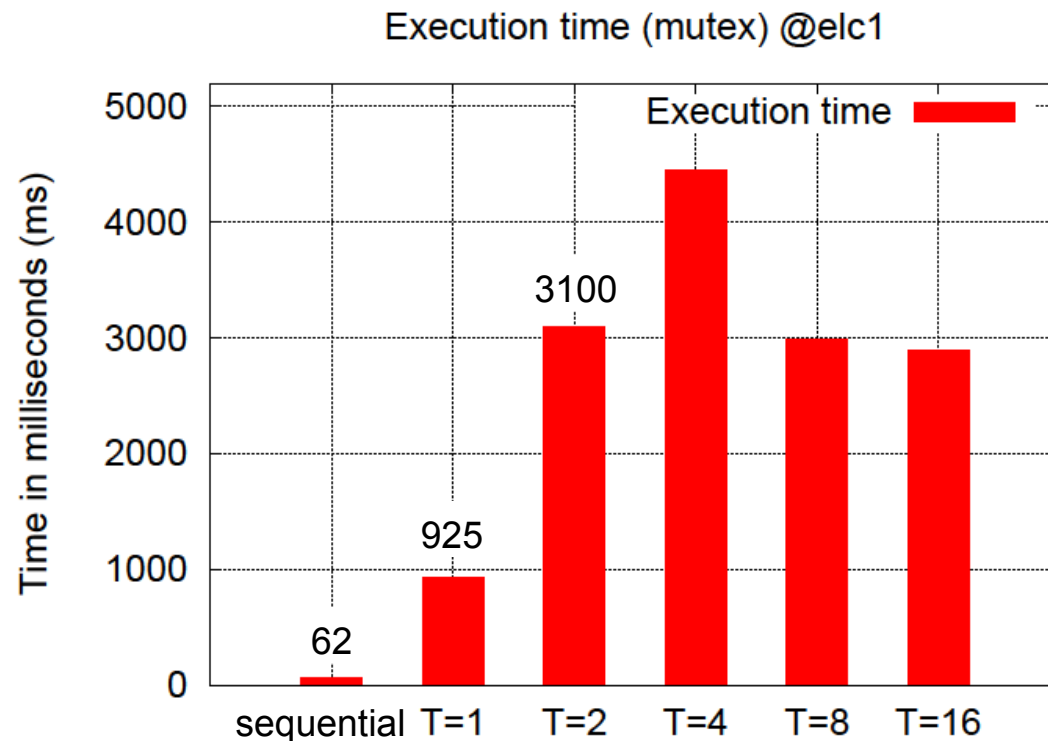
```
int length_per_thread = length / t;
int start = id * length_per_thread;
sum = 0;
pthread_mutex_t m;

for(i=start; i<start+length_per_thread; i++){
    pthread_mutex_lock(&m);
    sum = sum + array[i];
    pthread_mutex_unlock(&m);
}
```

Sum is a shared variable. Need to ensure mutual exclusion when accessing sum from different threads.

The correct program is slow...

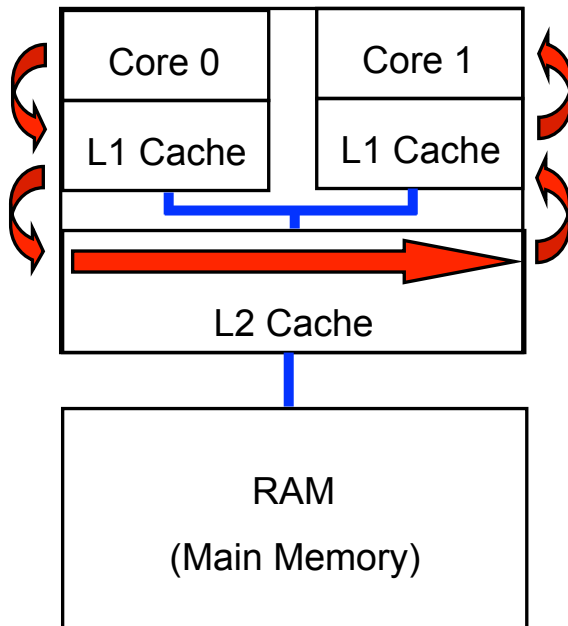
- We are mainly acquiring/releasing the mutex.
 - lock()/unlock induces huge overhead!
- Threads serialize at the mutex.
 - Have to wait for each other.
- Memory traffic for sum and mutex variable.



array of 16 million
unsigned chars

Closer look: motion of **sum** and **m**

- The variables **sum** and **m** need to be moved across the memory hierarchy because both threads access them.
 - Takes time.
- Lock contention at mutex **m**.
 - Overhead for lock/unlock operations.



```
int length_per_thread = length / t;
int start = id * length_per_thread;
unsigned long long sum = 0;
pthread_mutex_t m;

for(i=start; i<start+length_per_thread; i++){
    pthread_mutex_lock(&m);
    sum = sum + array[i];
    pthread_mutex_unlock(&m);
}
```

Accumulate into private sum variable

- Each thread adds into its own memory (variable `private_sum[]`).
- Combine sums at the end.

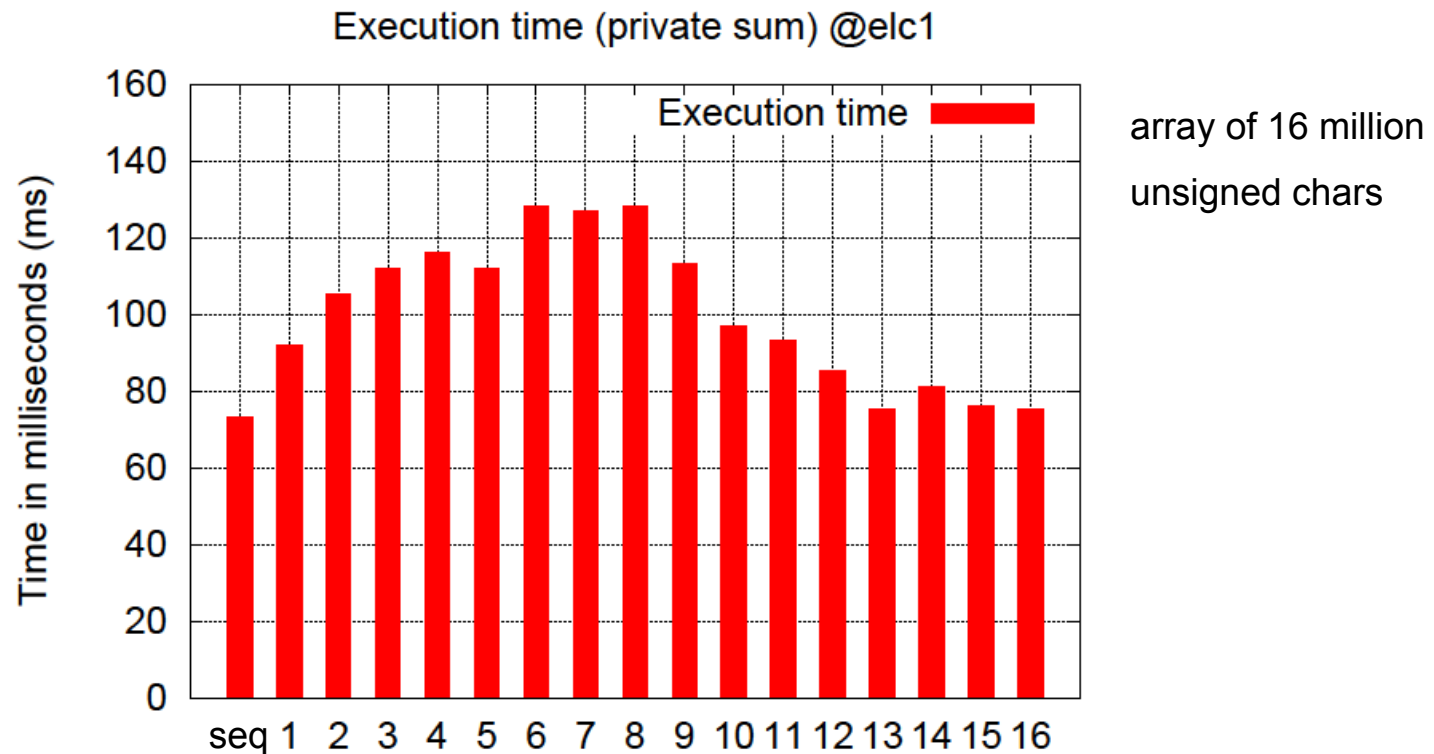
```
#define t 4
int length_per_thread = length / t;
int start = id * length_per_thread;
unsigned long long private_sum[4]={0, ...};
unsigned long long sum = 0;
pthread_mutex_t m;

for(i=start; i<start+length_per_thread; i++){
    private_sum[id] = private_sum[id] + array[i];
}

pthread_mutex_lock(&m);
    sum = sum + private_sum[id];
pthread_mutex_unlock(&m);
```

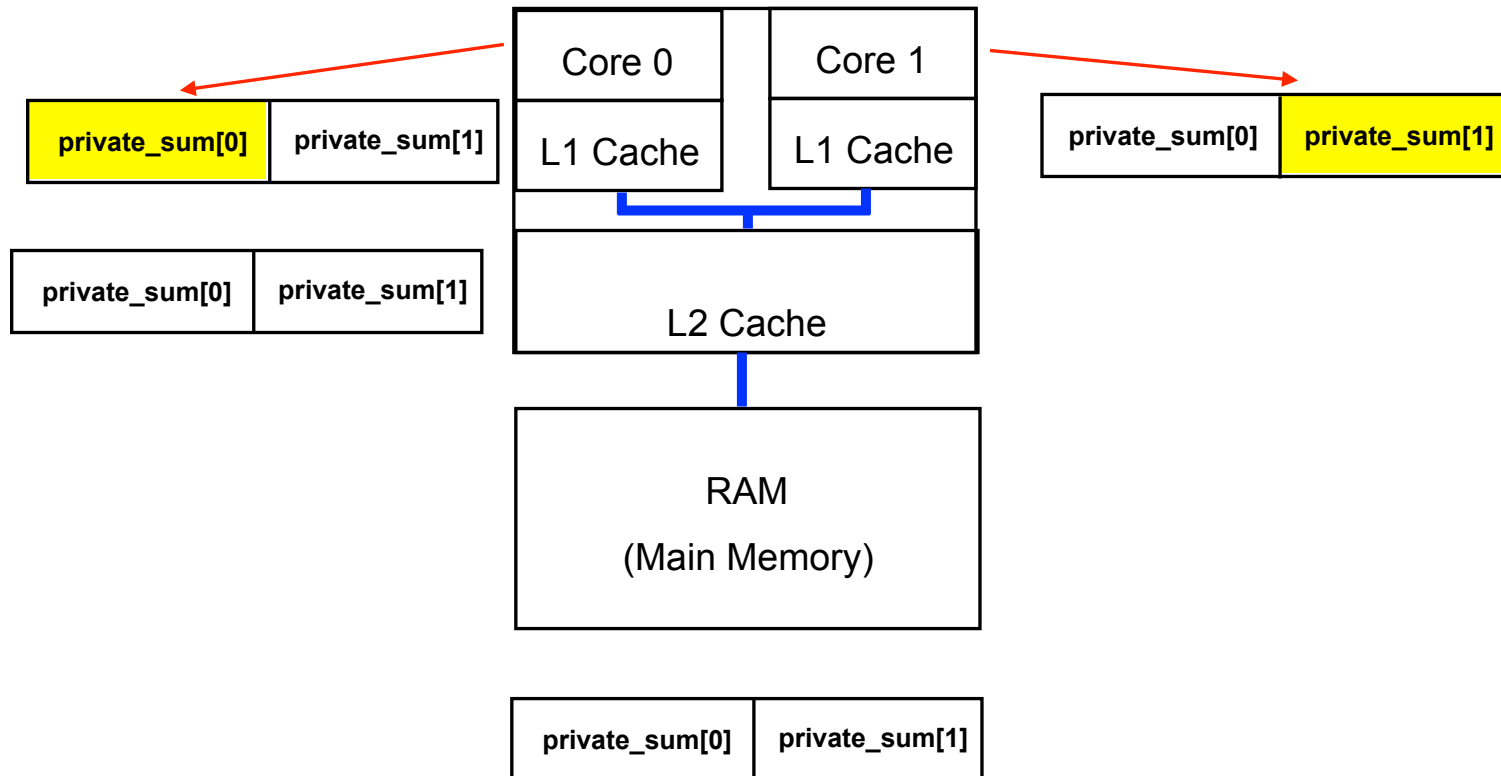
Keeping up, but not gaining

- 1-thread version almost as good as sequential
 - apart from overhead for creating threads
- It gets worse with 2 or more threads.
 - With the 9th thread, we introduce a new cache line and things get slightly better.
- However, still not faster than the sequential version!



False sharing

- Private sum variable \neq private cache-line.

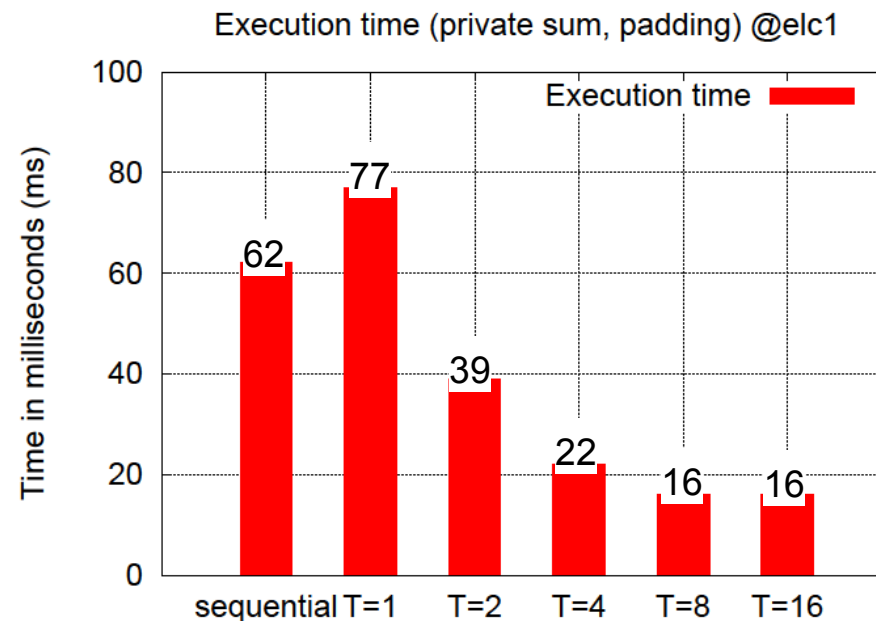


Force into different cache lines

- Padding the private sum variables forces them into separate cache lines and removes false sharing.

```
struct padded_int
{ unsigned long long value; // 8 bytes wide
  char padding [56]; //assuming a cache line is 64 bytes
} private_sum[MaxThreads];
```

- Two threads now almost twice as fast as one:



array of 16 million
unsigned chars

Summary: Performance Bottlenecks

- Huge sequential part of a program.
 - Amdahl's Law tells us that we cannot gain much in such a case.
- Highly contended lock
 - All threads need to queue up to enter their critical section.
- High communication overhead, little computation.
 - Remember: computation is our goal! Communication and synchronization is just overhead due to multiple threads.
 - Sharing of variables (mutexes, semaphores are also variables!)
 - `rand()` uses a seed variable (shared!) use `rand_r()` instead!
 - False sharing of variables in a cache line.
- Poor load balancing.
 - Some threads do all the work (are overworked!), other threads are idle. Processor resources are not well-utilized.
- Scalar code
 - Use vector units (SIMDization of scalar code) as much as possible.

Summary: Performance Improvements

- SIMDization of scalar code
 - using the vector units of processors (SSE, etc.)
- Data Locality
 - Main memory is slow, caches help, but only if...
 - threads have their “private” data they are working on (spacial locality).
 - threads access the same data over and over again (temporal locality).

Outline

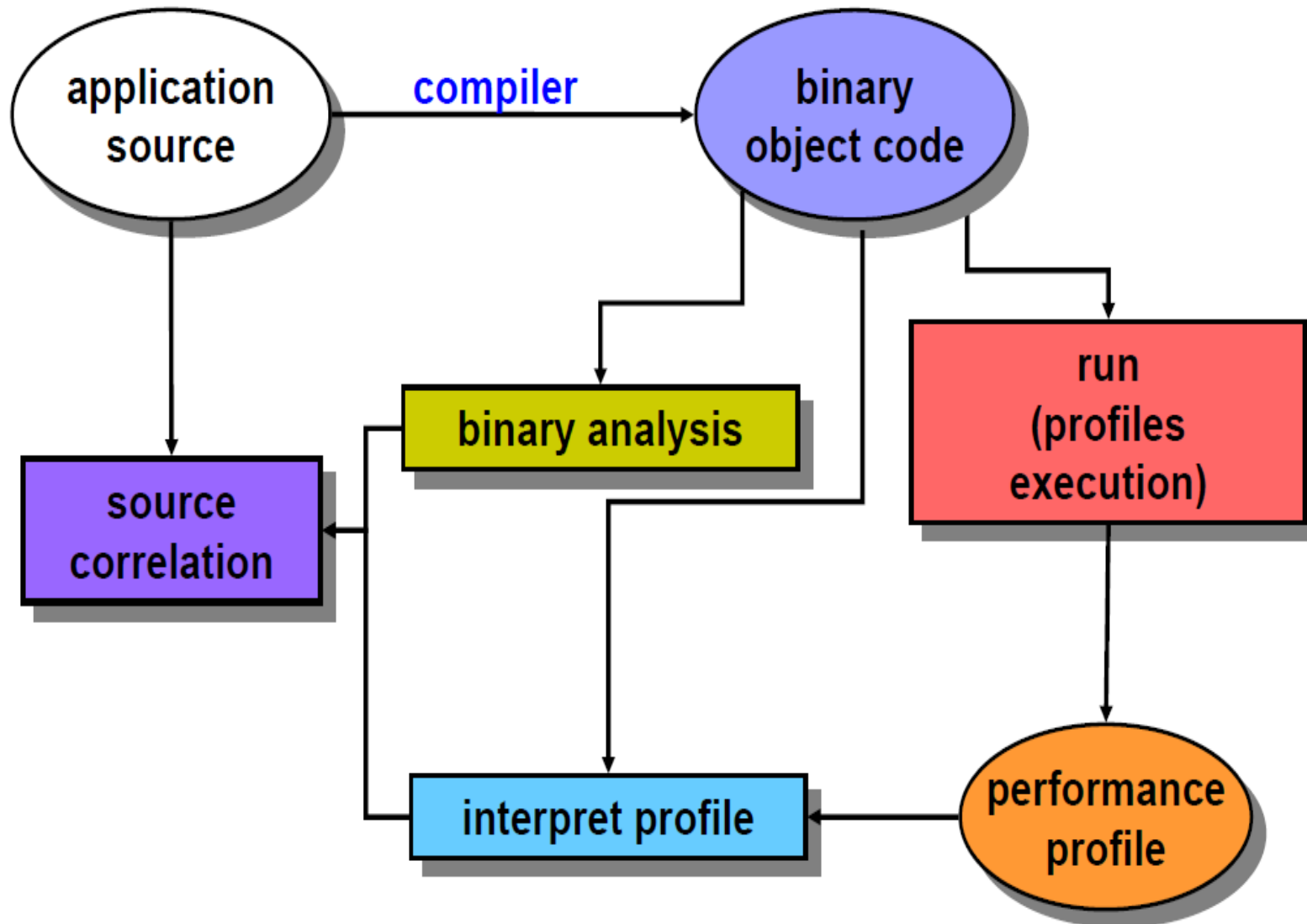
- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss ✓
 - Example Case-study ✓
- Profiling
- Reasoning about Performance

Profiling

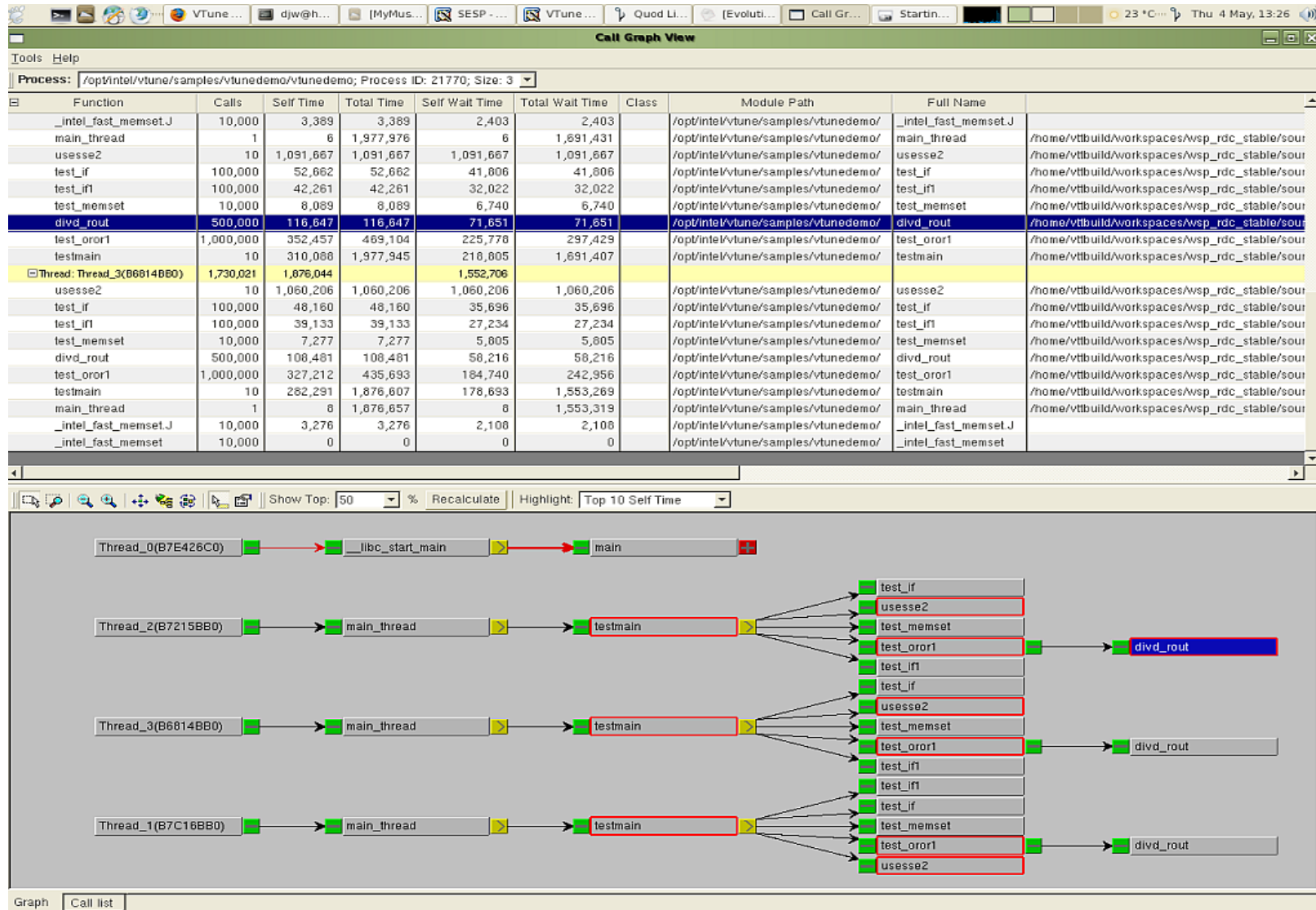
Profiling = Program Instrumentation + Execution

- CPUs provide performance counters to measure various events:
 - cache hits
 - cache misses
 - number of instructions executed
 - pipeline stalls
 - number of loads from memory
 - number of stores to memory
- And then there is the Heisenberg effect: measuring can perturb a program's execution time!
- GNU **gprof**
- Intel VTune
 - <http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm>
- OProfile
 - http://www.bsc.es/plantillaH.php?cat_id=464
 - <http://oprofile.sourceforge.net/>
- Many more!

Common Profiling Workflow



Sample Intel VTune Output



Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss ✓
 - Example Case-study ✓
- Profiling
- Reasoning about Performance

Sources of Performance Loss

Linear speedup with parallel processing frequently not attainable for following reasons:

1. Overhead which sequential computation does not have to pay
2. Non-parallelizable computation
3. Idle processors
4. Contention for resources

- All other sources are special cases of these four!

- In next slides, we consider those four sources of performance loss.

Loss 1: Overhead

- Any cost caused in parallel solution but not required by sequential solution is called **overhead**.
- Examples: creating/joining threads, synchronization, ...
- Overhead 1: Communication
 - Because sequential solution does not have to communicate with other threads, all communication is overhead.
 - Example: private sums have to be communicated back to main thread.
 - Cost for communication depends on hardware
 - Examples: shared-memory communication cheaper than sending messages over network, non-uniform memory access (NUMA) architectures.
- Overhead 2: Synchronization
 - Arises when one thread must wait for 'event' on another thread.
 - Example: thread must wait for another thread to free resource, e.g., mutex, in the sum computation example. Sequential solution does not require synchronization!

Loss 1: Overhead (cont.)

- Overhead 3: Computation

- Parallel computations almost always perform extra computations not needed in sequential solution.
- Examples:
 - threads in gray-scale conversion need to compute the part of the array which they should work on.
 - Initializations of counters also executed in each of the parallel threads

- Overhead 4: Memory

- Parallel computations frequently require more memory than sequential computations.
 - Example: padding in the parallel sum computation.
- Padding is a small memory overhead that actually improves performance.
- Applications with significant memory overhead will experience performance loss.

Loss 2: Non-parallelizable computations

- Non-parallelizable computations limit potential benefit from parallelism.
- Amdahl's law observes that if $1/S$ of a computation is inherently sequential, maximum speedup limited to a factor of S .
- Special case: **redundant computations**
 - N threads execute loop k times:

```
for(i=0; i<k; i++) { ... }
```
 - all N threads increment their private loop variable, test loop condition, ...
- Idle threads waiting for one thread to perform computation
 - Example: disk I/O

Loss 3: Idle Times

- Ideally, all processors/cores are working all the time.
- Thread might become idle because
 - 1) of lack of work
 - 2) it is waiting for external event, such as arrival of data from other thread
 - Example: producer-consumer relationship
 - 3) Load Imbalance
 - uneven distribution of work to processors/cores
 - Example: sequential computation running on single core, all other cores idle
 - 4) Memory-bound computations
 - Main memory very slow compared to CPU
 - Caches hide memory-latency, but applications with poor locality or large data-sets spend 100s of CPU-cycles waiting for data from main memory.

Loss 4: Contention for Resources

- Contention is degradation of system performance caused by competition for a shared resource.
 - Example: highly contended lock.
- Contention can degrade performance beyond the overhead observed from a 1-thread solution.
- Example: both true sharing and false sharing create excessive load on the memory bus (bus traffic)
 - Affects even threads that access other memory locations (and hence do not contend for shared data)!

Performance Trade-offs

- We have seen 4 sources of performance loss.
- Limiting one source of performance loss may increase another
- Important to understand the trade-offs between different sources of performance loss.

- Trade-off 1: Communication versus Computation
 - Often possible to reduce communication overhead by performing additional computations.
 - **Redundant computations**
 - Re-compute value locally (within a thread) instead of transmitting from other thread
 - works if $\text{cost}(\text{re-computation}) < \text{cost}(\text{transmission})$
 - Example: pseudo-random numbers
 - **Overlapping Communication and Computation**
 - Communication that is independent of computation can be performed while computation is going on (hiding communication behind computation)
 - Makes program more complicated (DMA-transfers, wait-for-completion, ...)

Performance Trade-offs (cont.)

- Trade-off 2: Memory versus Parallelism
 - Parallelism can often be increased at cost of increased memory usage
 - [Privatization](#) (e.g., private_sum variable)
 - [Padding](#)

- Trade-off 3: Overhead versus Parallelism
 - [Parallelize Overhead](#)
 - Final accumulation of private_sums on Slide #37 can become bottleneck with large number of threads → parallelize overhead itself!
 - We will discuss in subsequent lecture
 - [Load-Balance versus Overhead](#)
 - increased parallelism can improve load-balance
 - → over-decomposing problem in fine-grained work-units
 - [Granularity Trade-offs](#)
 - Increase granularity of interaction
 - Examples:
 - 1) each thread grabs several work items at once from work-queue
 - 2) send whole row/column of matrix instead of individual elements

Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss ✓
 - Example Case-study ✓
- Profiling
- Reasoning about Performance ✓