Performance of Parallel Programs

Outline

- Amdahl's Law
- Load Balancing
- Measuring Performance
- Sources of Performance Loss
    - Example Case-study
- Thread pool
- Reasoning about Performance
- A closer look at memory

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;
```

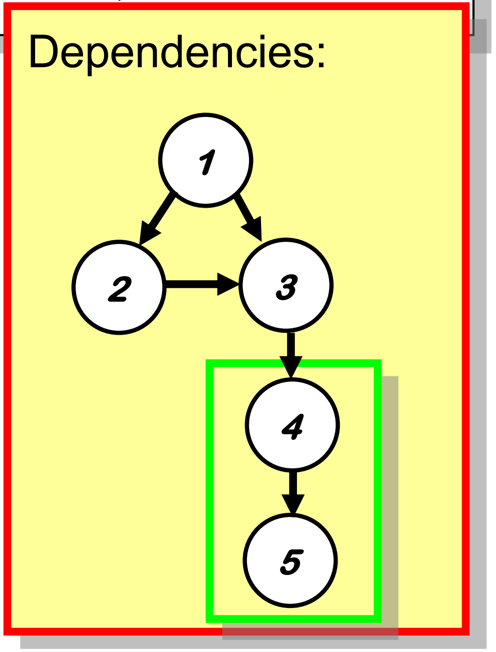
Limits to Performance Scalability

- 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;
```



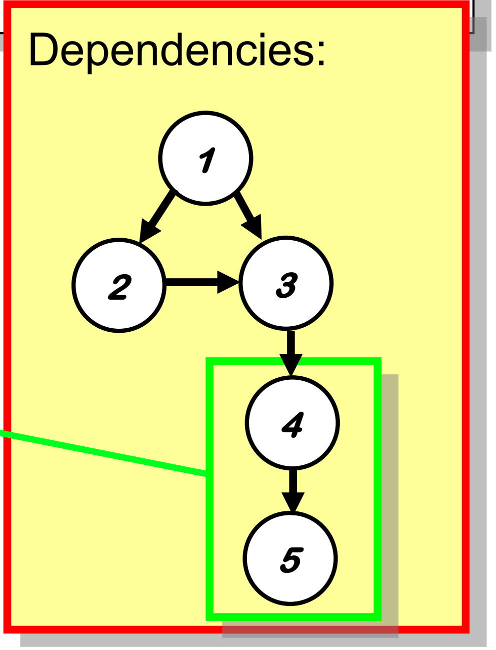
Limits to Performance Scalability

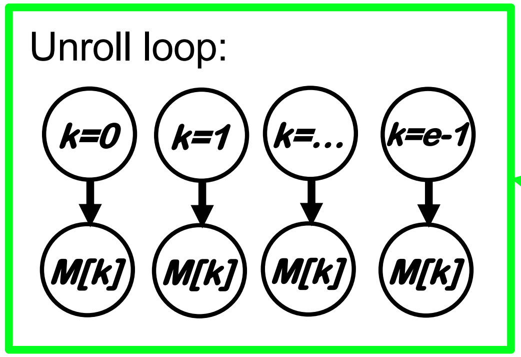
- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

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;
```



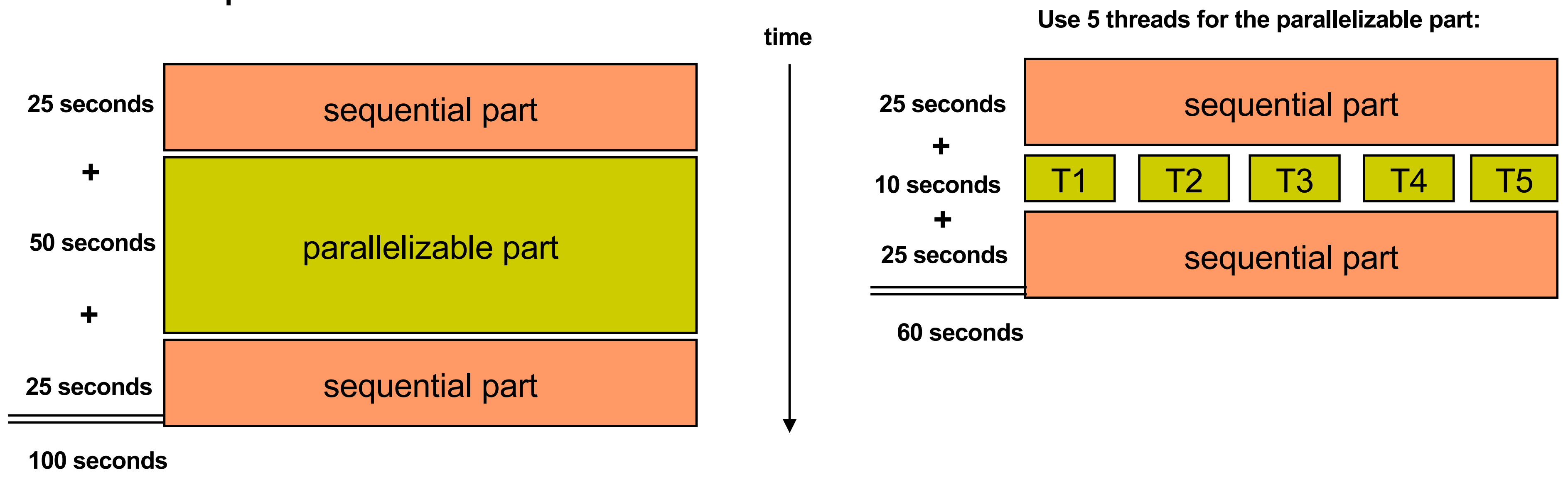


Amdahl's Law

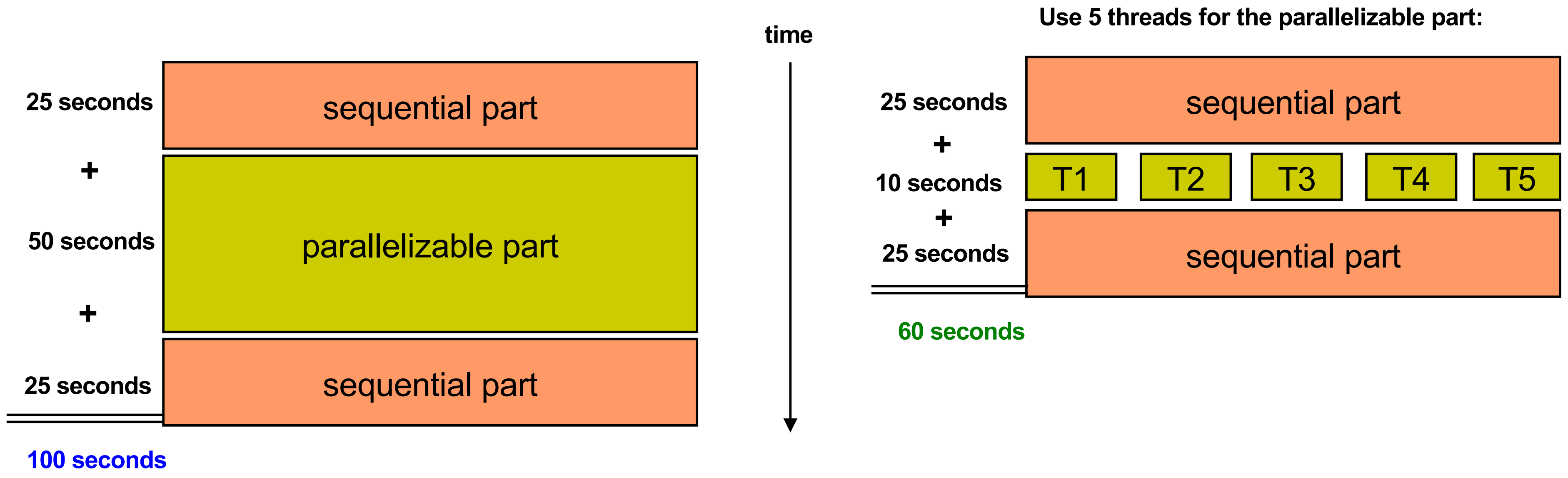
In 1967, Gene Amdahl, then IBM computer mainframe architect, stated that

> *"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.

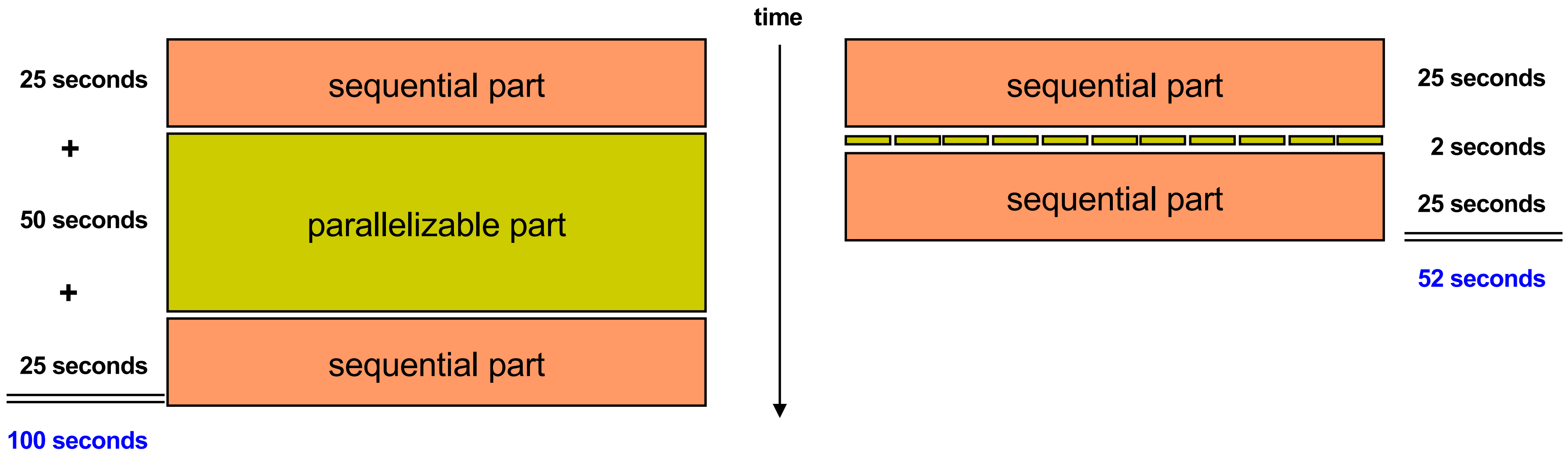


Amdahl's Law (cont.)



- Speedup = old running time / 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

speedup = 100 seconds / 50 seconds = 2.

Amdahl's Law (cont.)

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

$$\textit{new\_running\_time} = (1-p) * \text{old\_running\_time} + \frac{p * \textit{old\_running\_time}}{n}$$

$$\textit{Speedup} = \frac{\textit{old\_running\_time}}{\textit{new\_running\_time}} = \frac{1}{(1 - p) + \frac{p}{n}}$$

- Observation: if the number of threads goes to infinity ($n \to \infty$), 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.

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1 - p) + \frac{p}{n}}$$

- 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

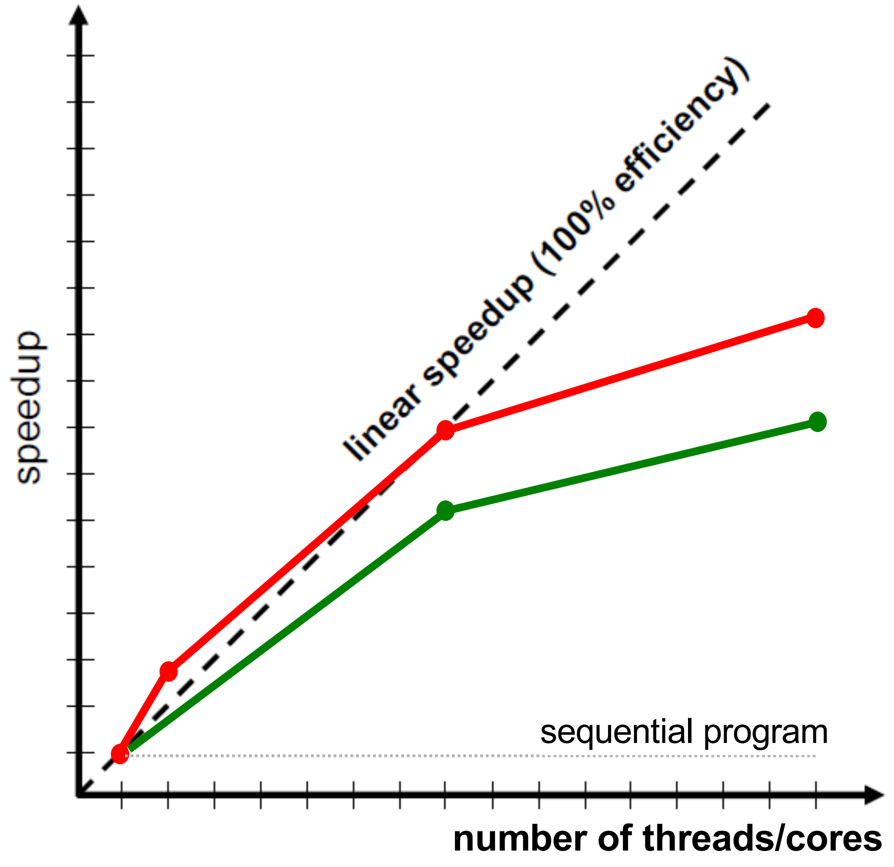
$$\text{speedup} = \frac{1}{0.25 + \frac{0.75}{10}} = 2.91$$

- Example 4: p=0.75, n = ∞

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

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.



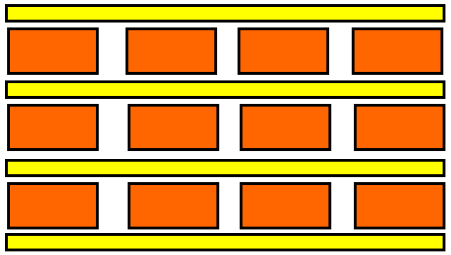
- The speedup graph shows that the curves level off as we increase the number of cores.
  - Result of keeping the problem size constant → the amount of work per thread decreases → overhead for thread creation also becomes more significant.
  - Other reasons possible also (memory hierarchy, highly-contended 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

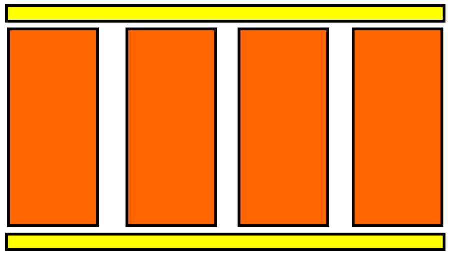
- Amdahl’s Law ✓
- Load Balancing
- Measuring Performance
- Sources of Performance Loss
  - Example Case-study
- Thread pool
- Reasoning about Performance
- A closer look at memory

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.

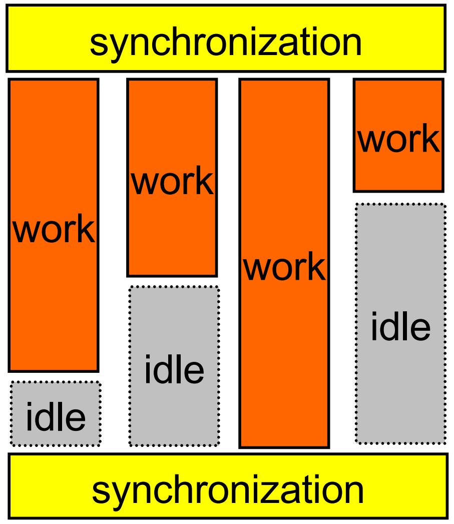


- Coarse-grained Parallelism
  - Large amount of computational work between communication / synchronization stages.
  - High computation to communication ratio.
  - Low communication / synchronization overhead.
  - More opportunity for performance improvement.
  - Harder to load-balance effectively.



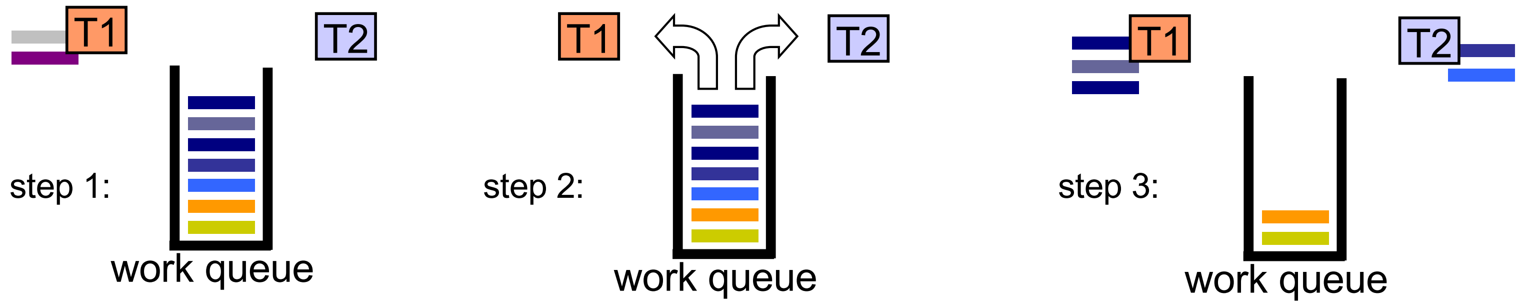
Load Balancing Problem

- 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.



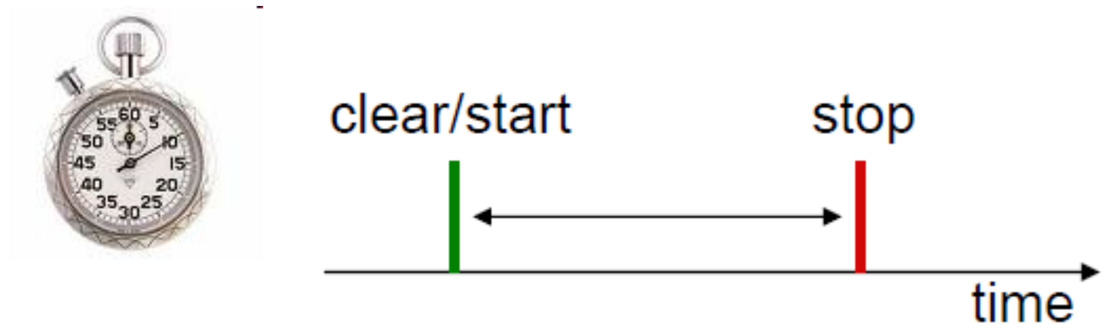
- With flexible allocations, load balance can be solved late in the design programming cycle.

Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance
- Sources of Performance Loss
  - Example Case-study
- Thread pool
- Reasoning about Performance
- A closer look at memory

Measuring Performance

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



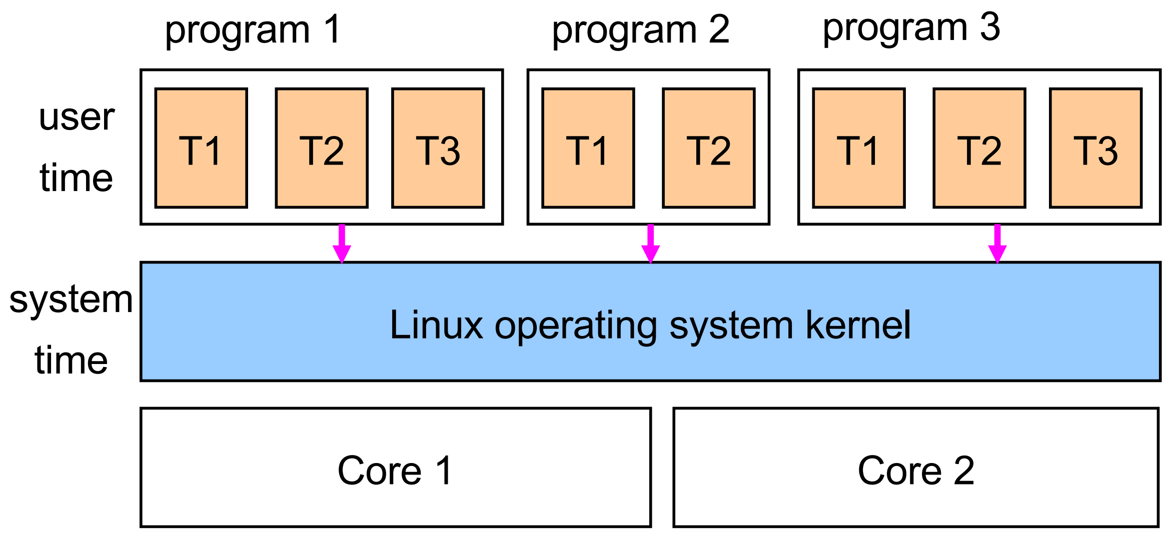
```
gcc foo.c
time ./a.out
real      0m4.159s
user      0m0.008s
sys       0m0.026s
```

elapsed wallclock time

time executed in user mode

time executed in kernel mode
(see next slide)

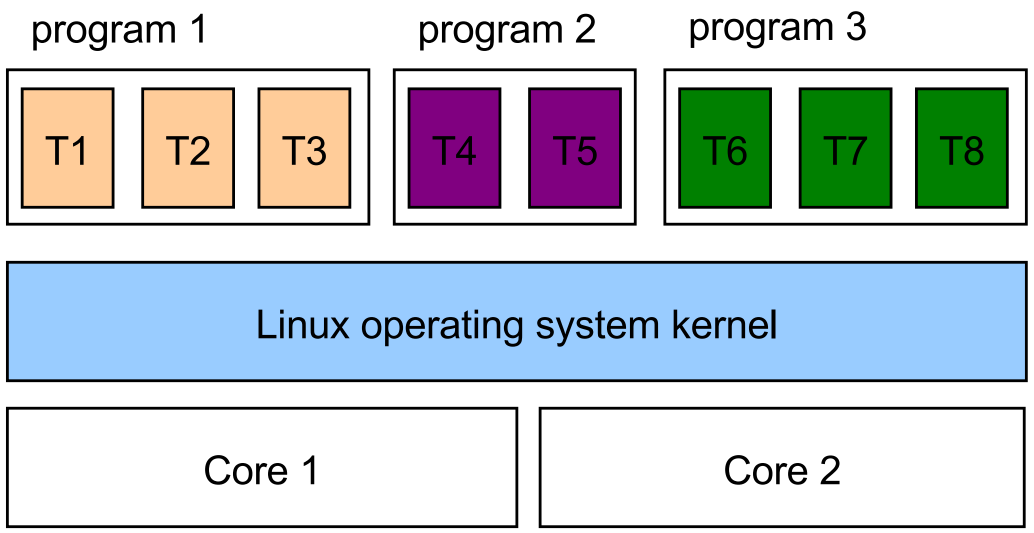
Difference between user and system time



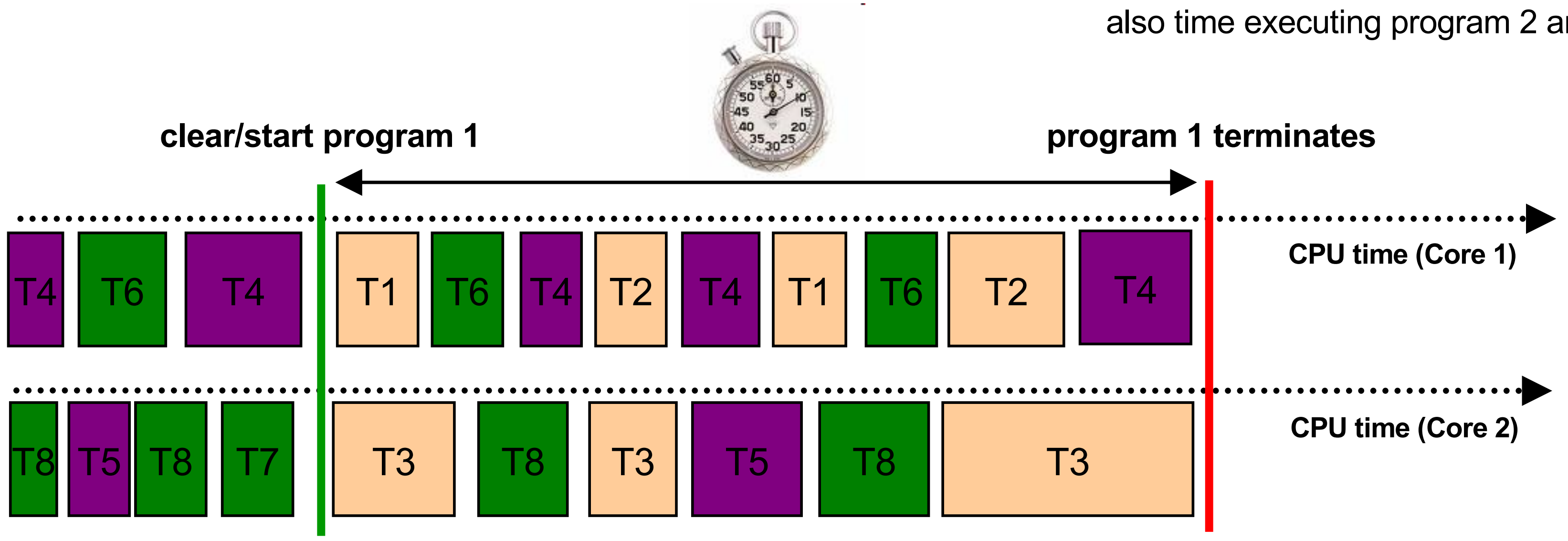
```
#include<stdio.h>

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 ≠ wallclock time ?

Wanted: an unloaded machine

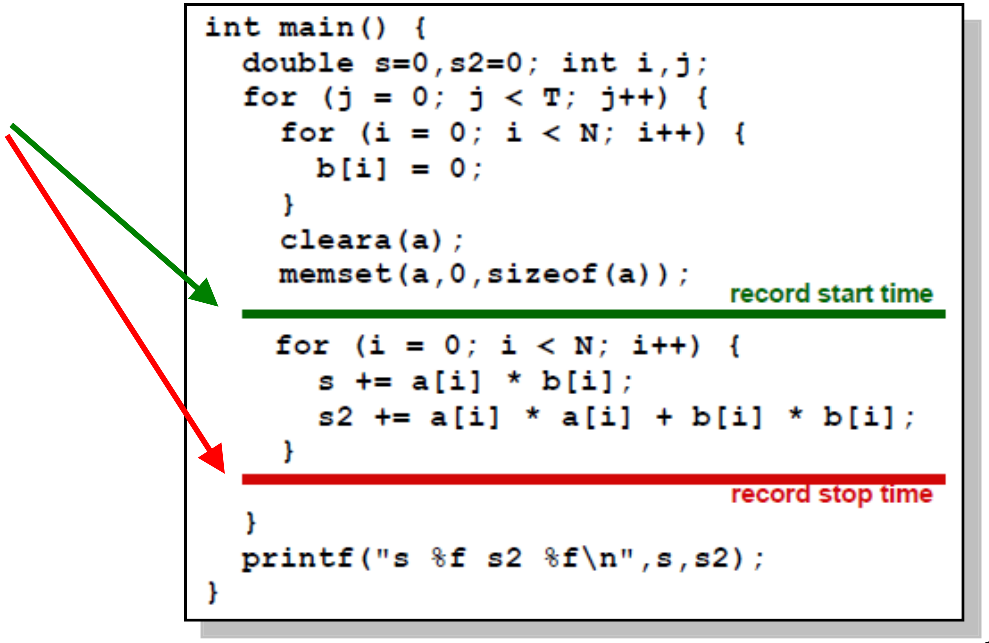


- Principal problem with wallclock time:
  - Different programs share the available CPUs/cores.
  - The Linux kernel schedules the threads of all programs on those CPUs/cores.
  - Every thread gets to run for a little while (its **timeslice**), then another thread gets to run.
  - Programs get in the 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.
- A bit better
  - clock()



```
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);
}
```

Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss
  - Example Case-study
- Thread pool
- Reasoning about Performance
- A closer look at memory

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.

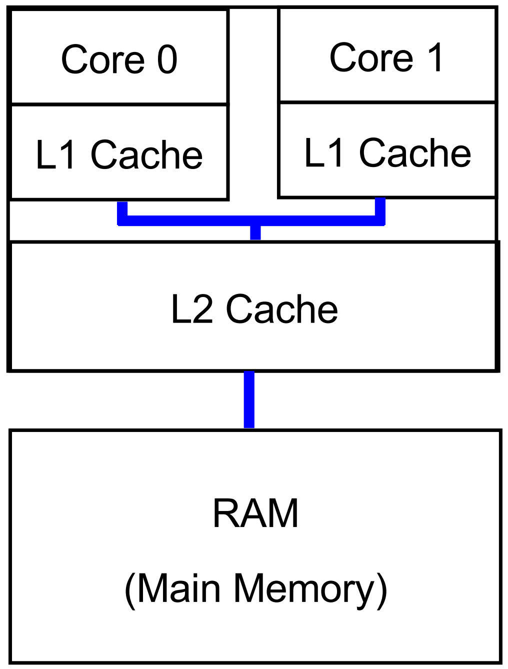
- Sequential solution:

```
int compute_sum(int* array, size_t length)
{
   int sum = 0;

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

Write a parallel program

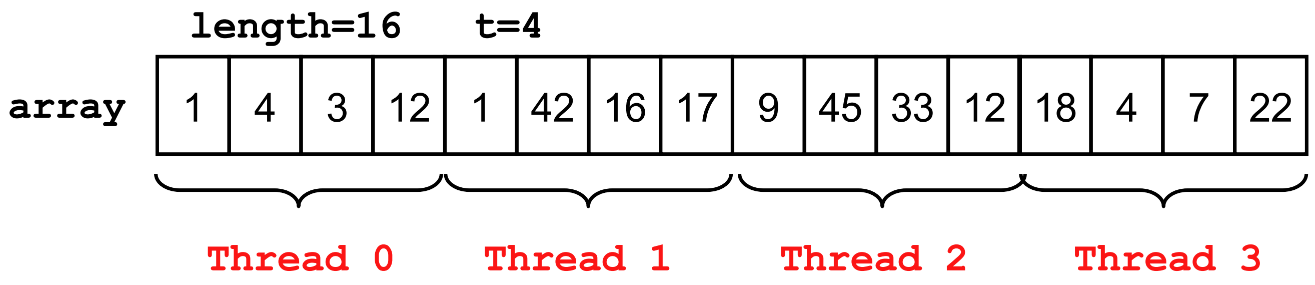
- We need to know something about the machine...
- Let's use a multicore architecture!

- 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.

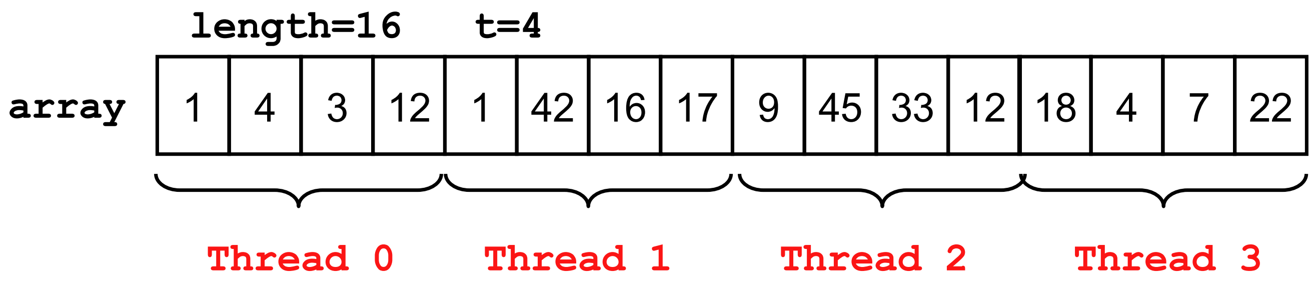


```
int length_per_thread = length / t;
int = start = id * length_per_thread;
sum = 0;

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.

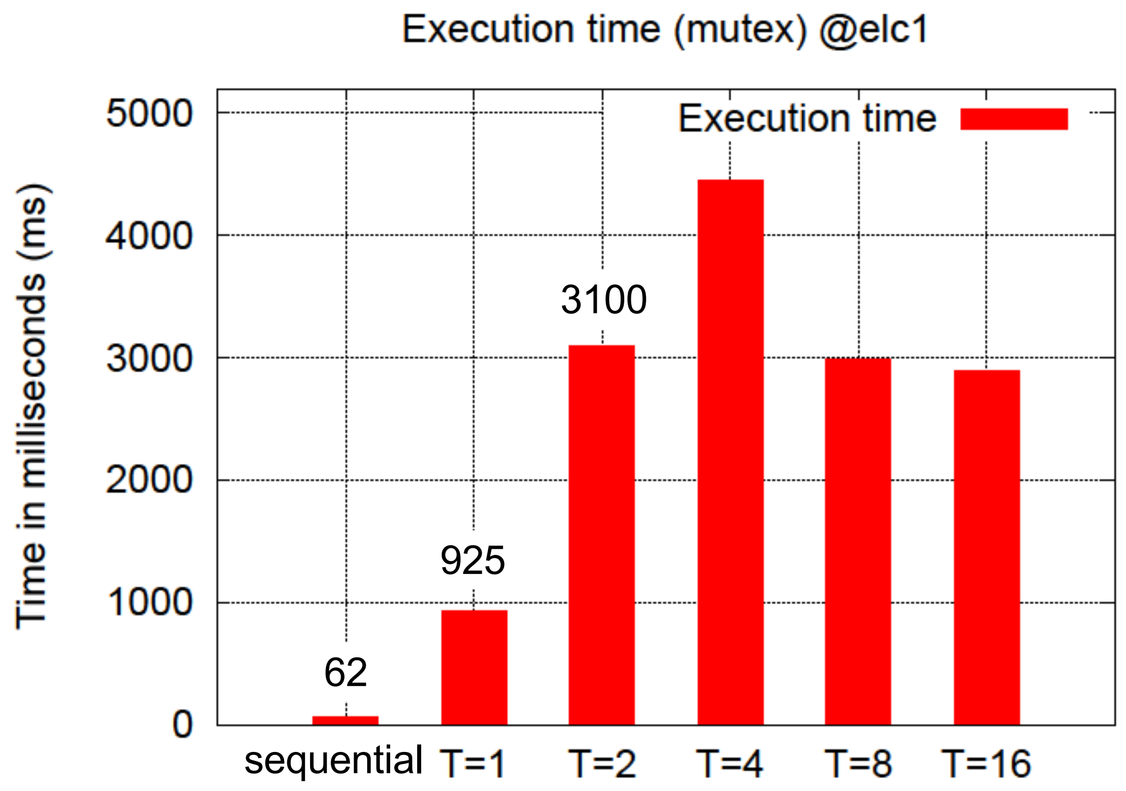


```
int length_per_thread = length / t;
int start = id * length_per_thread;
sum = 0;
pthread_mutex_t m;

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.

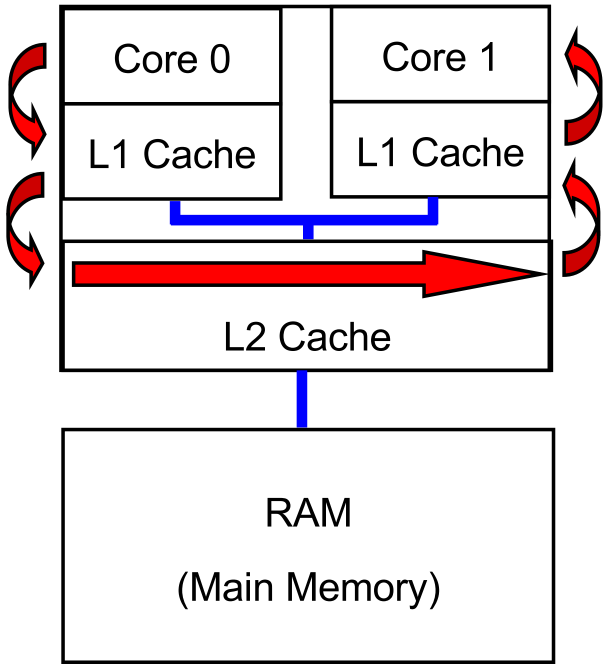


array of 16 million

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);
}
```

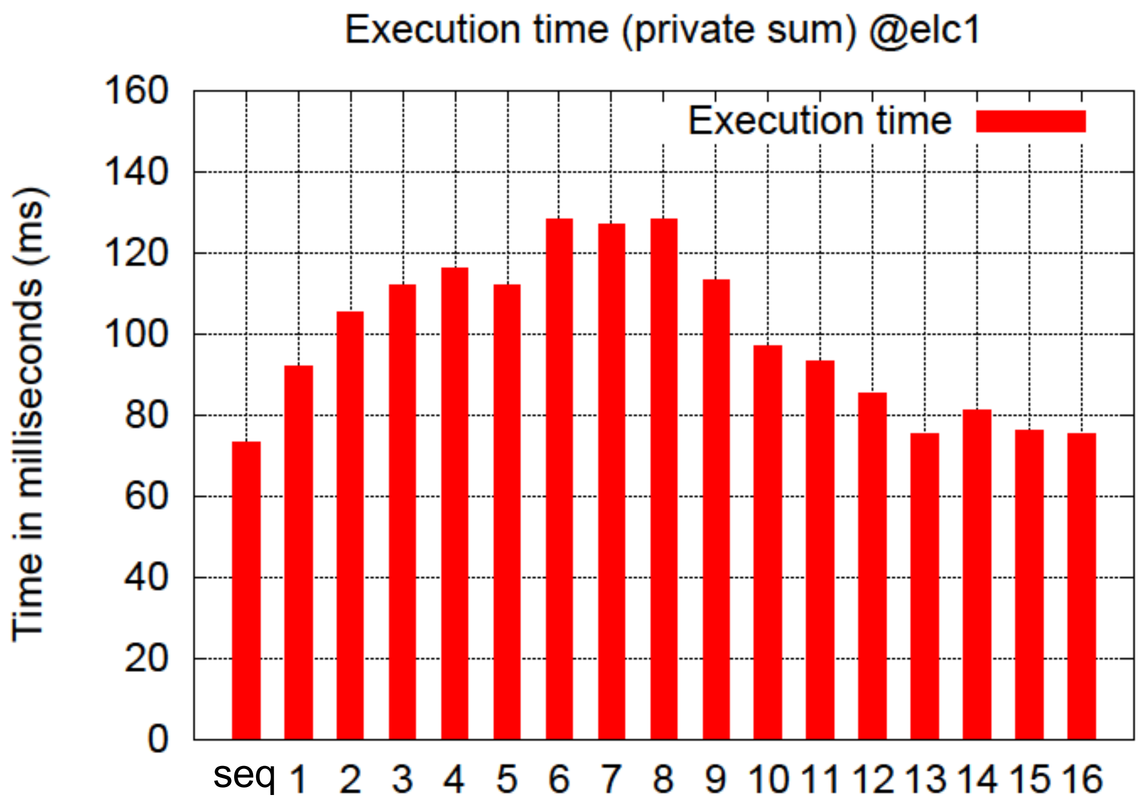
Accumulate into private sum variable

- 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!

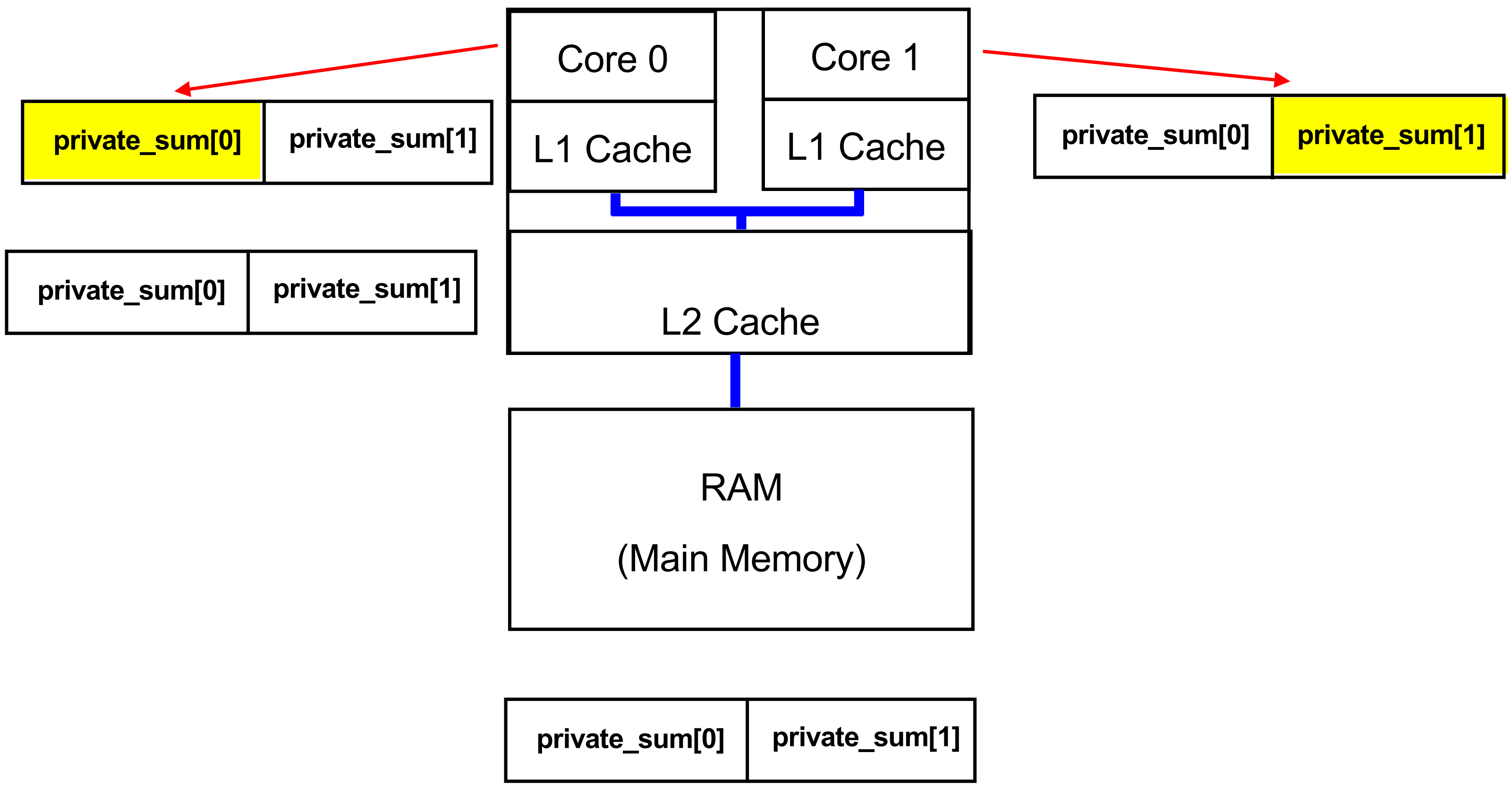


array of 16 million

unsigned chars

False sharing

- Private sum variable ≠ private cache-line.

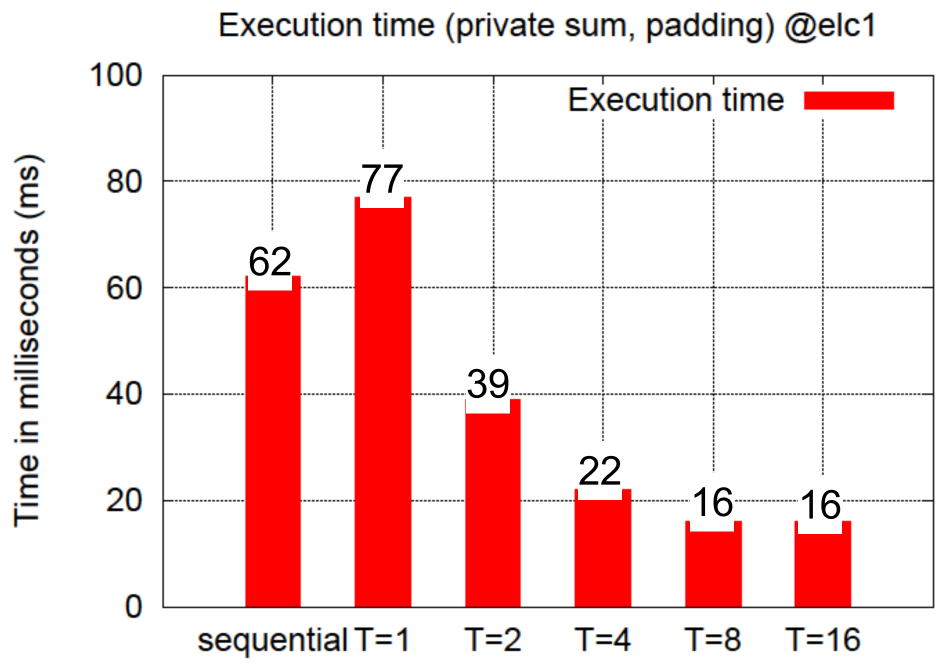


Force into different cache lines

- 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

/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size

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:

/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size

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 (spatial 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 ✓
- Thread pool
- Reasoning about Performance
- A closer look at memory

Implementing with threads

- Creating and joining threads takes time
  - Pre-create all threads at beginning?
- Task parallelism: e.g. threads for specific functions foo(), bar()
  - Will all tasks finish the same time?
  - Idle threads waiting on data for a specific task could be more useful
  - Tasks may have equal weight, but data can vary also
- Data parallelism
  - what assumptions about available threads have you made?
- What is the problem relationship with threaded programming
  - Independent or cooperative work?

Thread pool

- A thread pool is a collection of **N** threads that will perform a general computation task.

- Objectives
  - amortise time cost by load balancing smaller workloads
  - an opaque parallel programming construct to aid non-parallel programmer
    - Reliability of synchronisation/performance
    - For good software construction: definition of work (Object Oriented Design)
  - Optimisation possible with other parts of system by varying **N** threads (JVM)

Thread pool

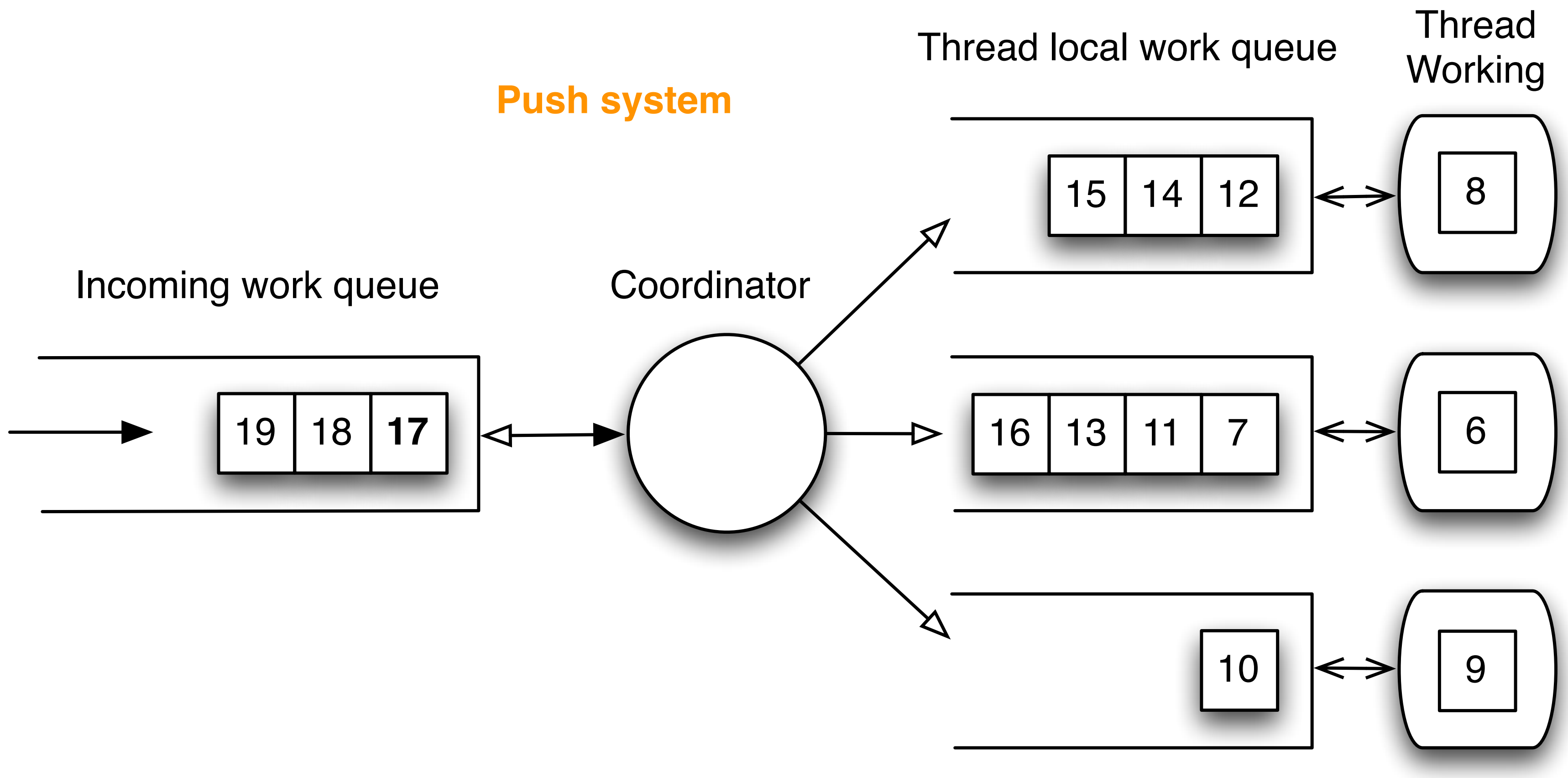
```
struct frame_info {
  uint8_t  magic[2];
  uint8_t  control;
   …
  uint32_t time_stamp;
  uint8_t  data[];
  uint8_t  stage; //0,1,2,3…
  pthread_mutex_t lock;
};
```

- Pthread already allows us to define a general computation task
- **void *func(void *arg);**
- Any function we define as this prototype can be passed around by using it's function pointer

```
void kinect_1stpass(struct frame_info *fm);

// wrap existing function
void *kinect_1stpass_for_pthreadpool(void *param) {
     // extract parameters
     struct frame_info *fm = (struct frame_info*) param;
     kinect_1stpass(fm);

     return NULL;
}
```

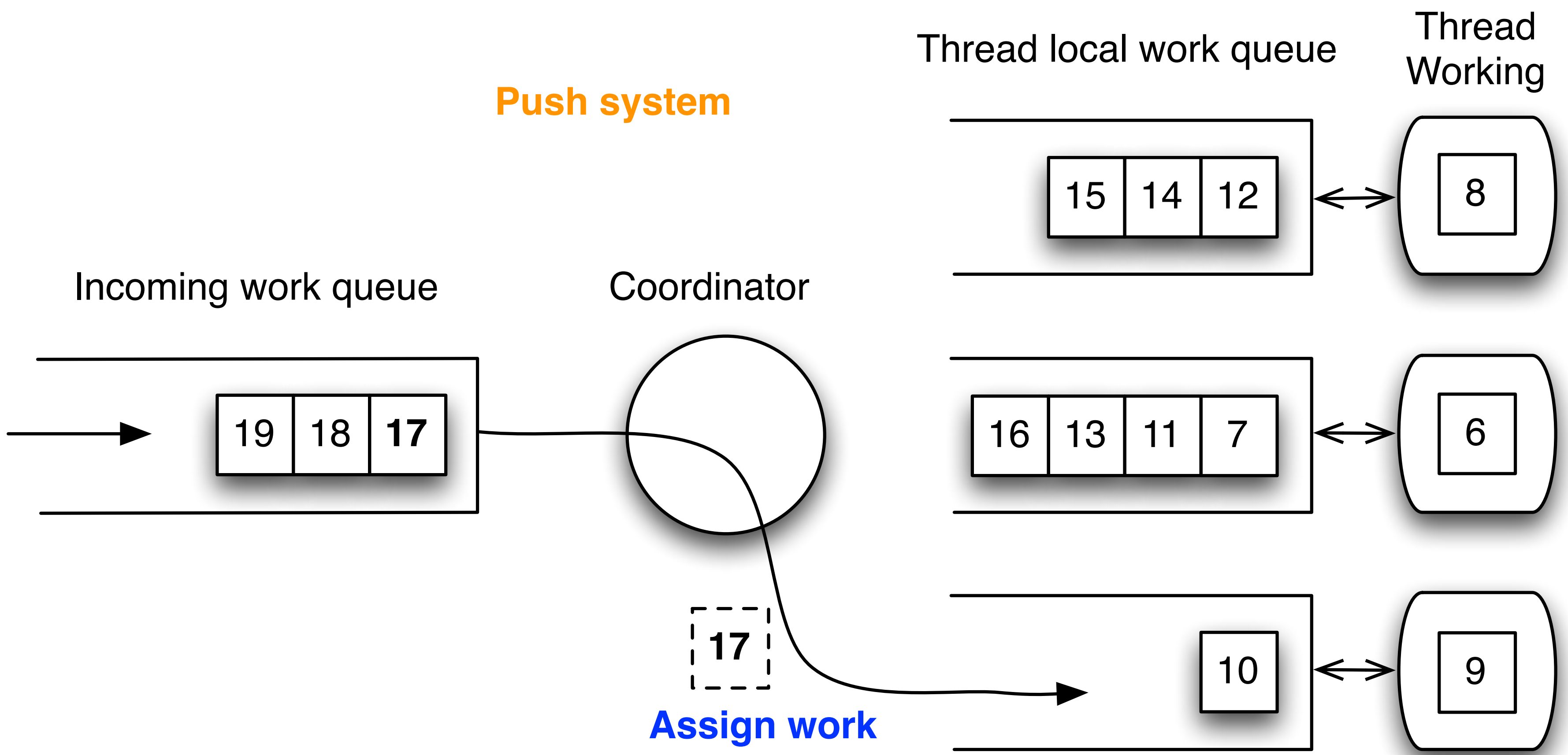
Thread pool

- How is the work distributed among N threads?



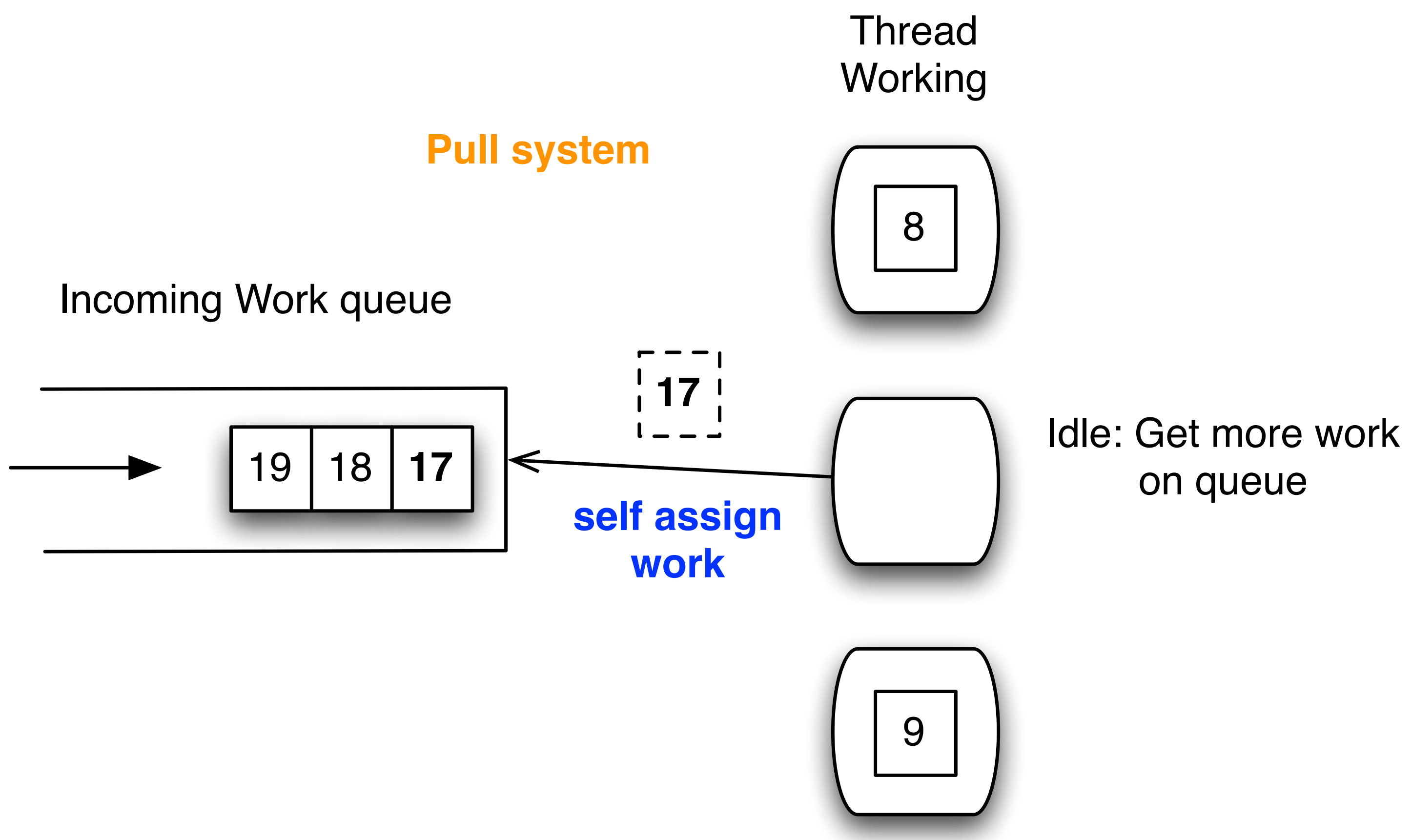
Thread pool: Push work to threads

- Check work load of each thread worker
    - Separate thread for coordinator?



Thread pool: Pull from work queue

- Don't need private thread queue
  - Consequences?



Thread pool: build your own

- A structure to define the general computation
- Functions for the User of the threadpool
  - Creating/Destroying
  - Blocking/non-blocking?
- User synchronisation
  - Thread pool is meant to disassociate work and thread but…
  - Results are still needed
  - Multiple work items stem from larger sets of tasks

```
// basic unit of work
struct _workitem {
        void *(*action)(void*);
        void *arg;
};
typedef struct _workitem workitem_t;
```

```
extern threadpool_t *
threadpool_init( int max_threads );

extern void
threadpool_q_workitem( threadpool_t *tp,
        void *(*action)(void*), void *arg );

extern void threadpool_barrier(
        threadpool_t *tp,
        workitem_t *witem, int witem_count );
```

Thread pool: build your own

- Structure to track each thread worker
- The general worker function
- Communication within the thread pool
  - How to synchronise getting work?
  - What to do after finished work?
- Fairness

```
// per thread data
struct _worker_data {
        int id;
        …
        struct threadpool_t *tp;
};
typedef struct _worker_data workerdata_t;
```

```
static void *luthreadpool_start(void *arg) {
        while (1) { // work forever
                // get workitem
                workitem_t *work = …

                // do the general computation
                (*work->action)(work->arg);

                // anything else?
                free(work);
        }
}
```

```
threadpool_init(…) {
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_attr_setschedpolicy(&attr, SCHED_RR); // man 7 sched

  for ( ; i < max_threads; ++i)
    pthread_create(&tp->threads[i], &attr, threadpool_start, wd);
}
```

Thread pool: design considerations

- Task size: meant to be small, but how small?
  - Experimentally, find the threshold on unloaded machine
  - Dynamically, do a small test in the program! (beware cold/hot cache)
- Overhead costs
  - additional function calls
  - reformatting parameters to be described with one address (pthread)
  - Inside threadpool there are delays which has nothing to do with task
- Which data structure would minimise contention for distributing work?
- < 200 lines of C code (using pthreads)

Outline

- Amdahl's Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss ✓
  - Example Case-study ✓
- Thread pool ✓
- Reasoning about Performance
- A closer look at memory

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 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 cost(re-computation) < cost (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

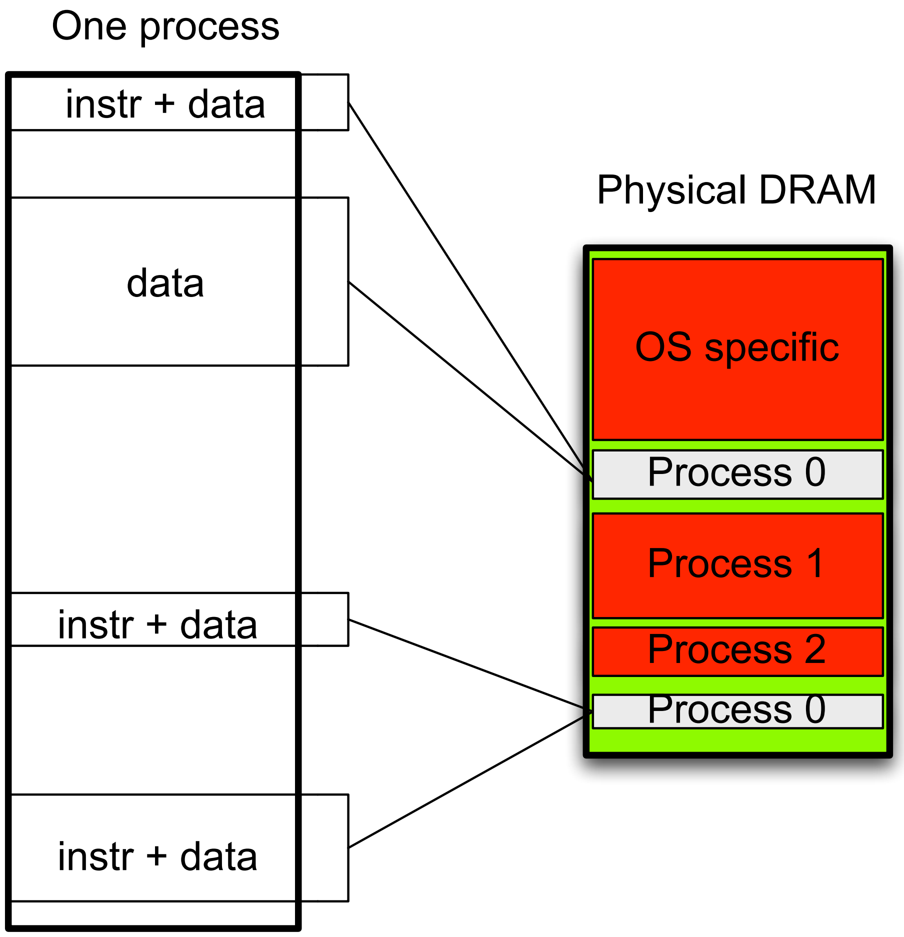
- Trade-off 3: Overhead versus Parallelism
  - Parallelize Overhead
    - Final accumulation of private_sums in earlier slides can become bottleneck with large number of threads →parallelize overhead itself!
    - We will discuss in subsequent lecture
  - Load-Balance versus Overhead
    - increased 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 ✓
- Thread pool ✓
- Reasoning about Performance ✓
- A closer look at memory

Memory

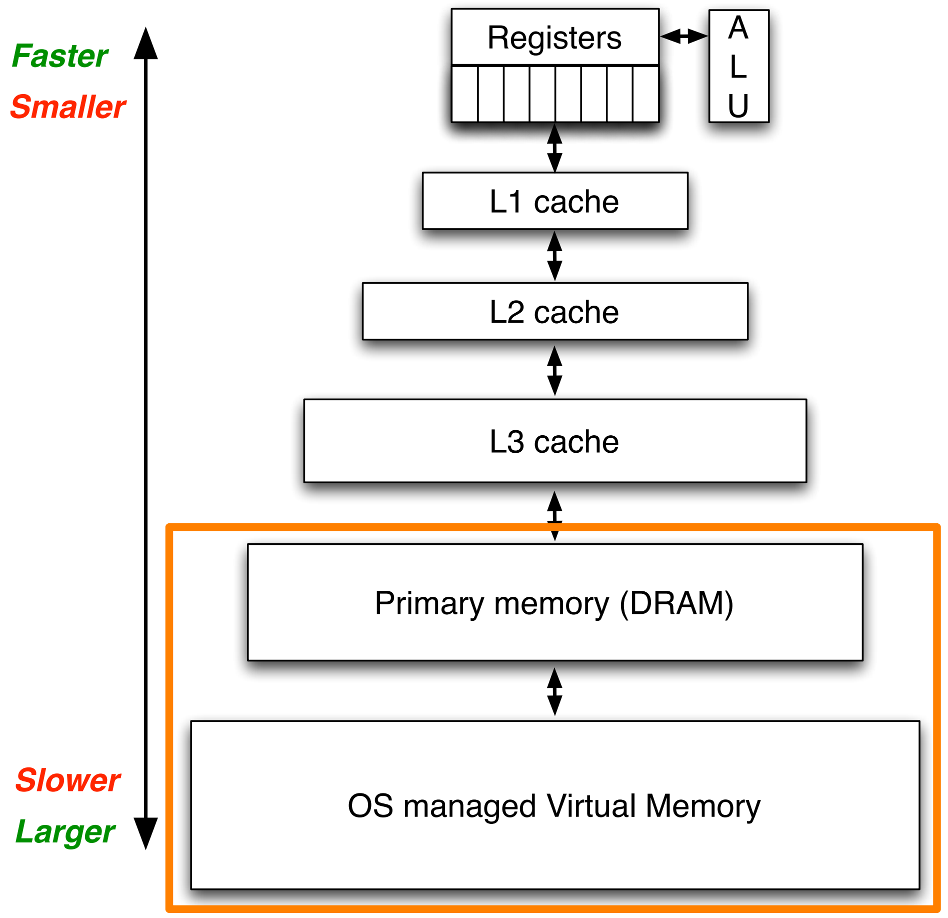
- Virtual memory of a process is mapped to physical memory.
- Physical memory is a mix: cache, RAM, disk, CD, tape, network…



- Multiple processes share the same finite memory resources
  - The physical primary memory (DRAM/SRAM) is easily exhausted
- Whatever cannot fit is stored in secondary memory
  - OS does this management of virtual memory translation to physical memory (with some hardware help)

Memory

- Memory exists at different levels
  - based on the need to have instructions and the data ready to continue the program



- OS does most of this work in software*
- Memory moves to closest part of CPU for computation
- Stays closer with higher frequency access

*newer processors have paging mechanisms

Memory: machine level

- Deciding when memory moves up and down the hierarchy *above* primary memory is up to the cache protocol

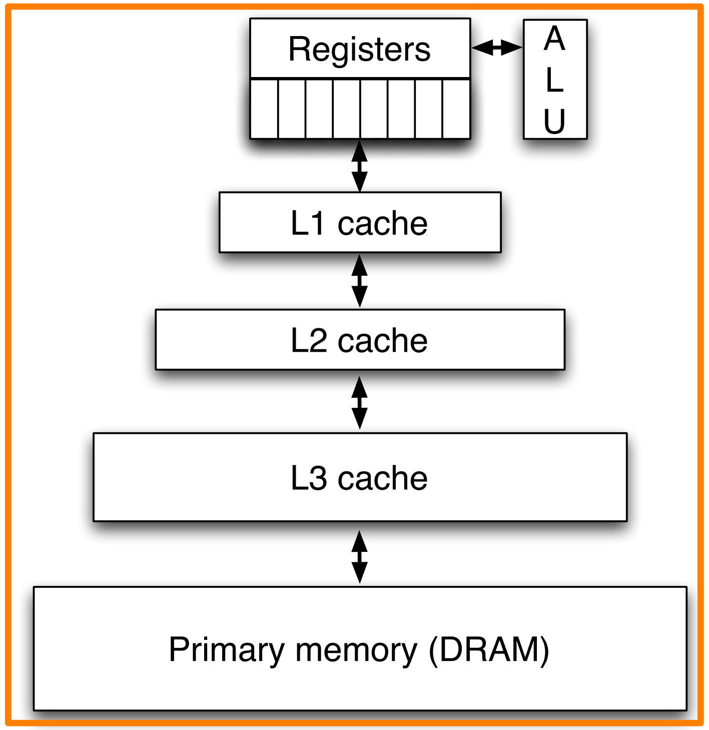
- Software is too slow to manage memory at this level (fine grained)

- Must be done in hardware

- Why?

  Software needs memory

  It is less useful to use more memory to solve memory shortage problems!



Memory: relative costs

- Data that is too large or less frequently used will have to move up/down the memory hierarchy
- Latency for cache hits and misses access times (Intel):

To Where	Cycles (P2 350MHz)	Cycles (Pentium M)	Cycles (i7 Haswell)	Cycles (i7 Skylake)
Register	~ 3	<= 1	0	0-1
L1 data	~ 6	~ 3	~ 4	~ 4
L2	~ 57	~ 14	~ 10	~ 12
L3			~ 40 - 75	~ 44
Main memory (DRAM)	~ 169 DDR	~ 240 DDR2	~ 200	

Memory: relative costs

- L2 cache miss costs are non-uniform

To Where	Intel Core^{TM} 2	i7 Haswell
L1 miss	~ 10	~ 6
L1 Data miss (from L2)	~ 20	~ 10
L2 Hit	~ 12	
L2 Miss	~ 165 desktop / ~ 300 server	~ 50 (from L3)

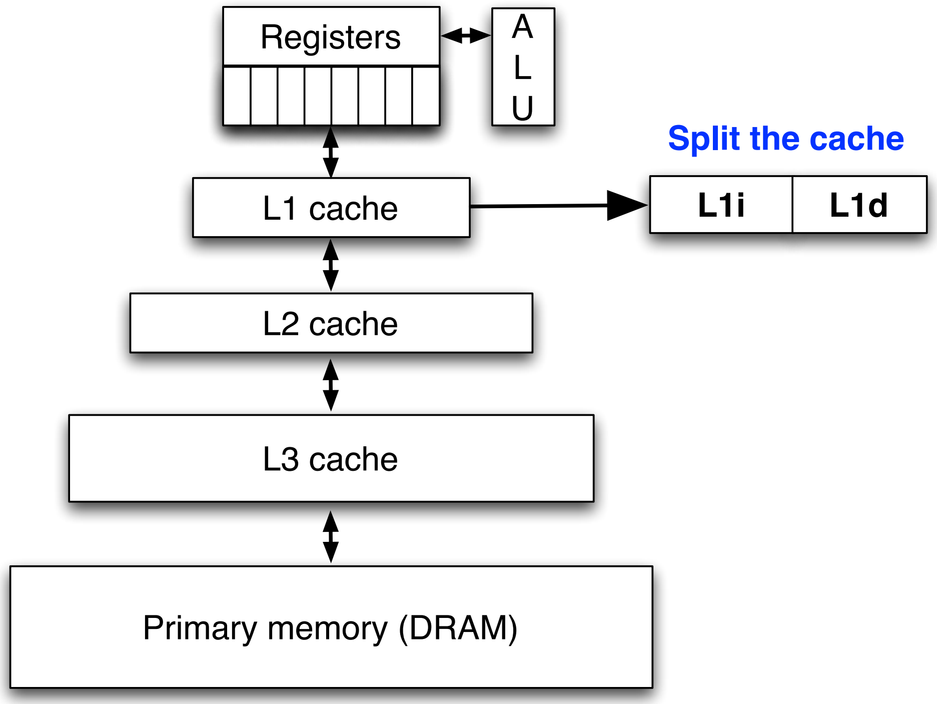
Memory: relative costs

- L3 cache becoming increasingly complex to measure performance

To Where	Intel Core^{TM} i7 and Intel Xeon 5500
L3 cache hit, line unshared	~ 40
L3 cache hit shared line in another core	~ 65
L3 cache hit, modified in another core	~ 75
Remote L3 CACHE	~ 100 - 300

Memory: L1 cache

- Instruction cache – is specifically for storing instructions that have been used and/or will follow
- If a pipeline stall occurs when fetching the next instruction this is catastrophic for performance
  - Intel has employed a separated L1 instruction cache in their CPUs since 1993



Memory: Instruction cache

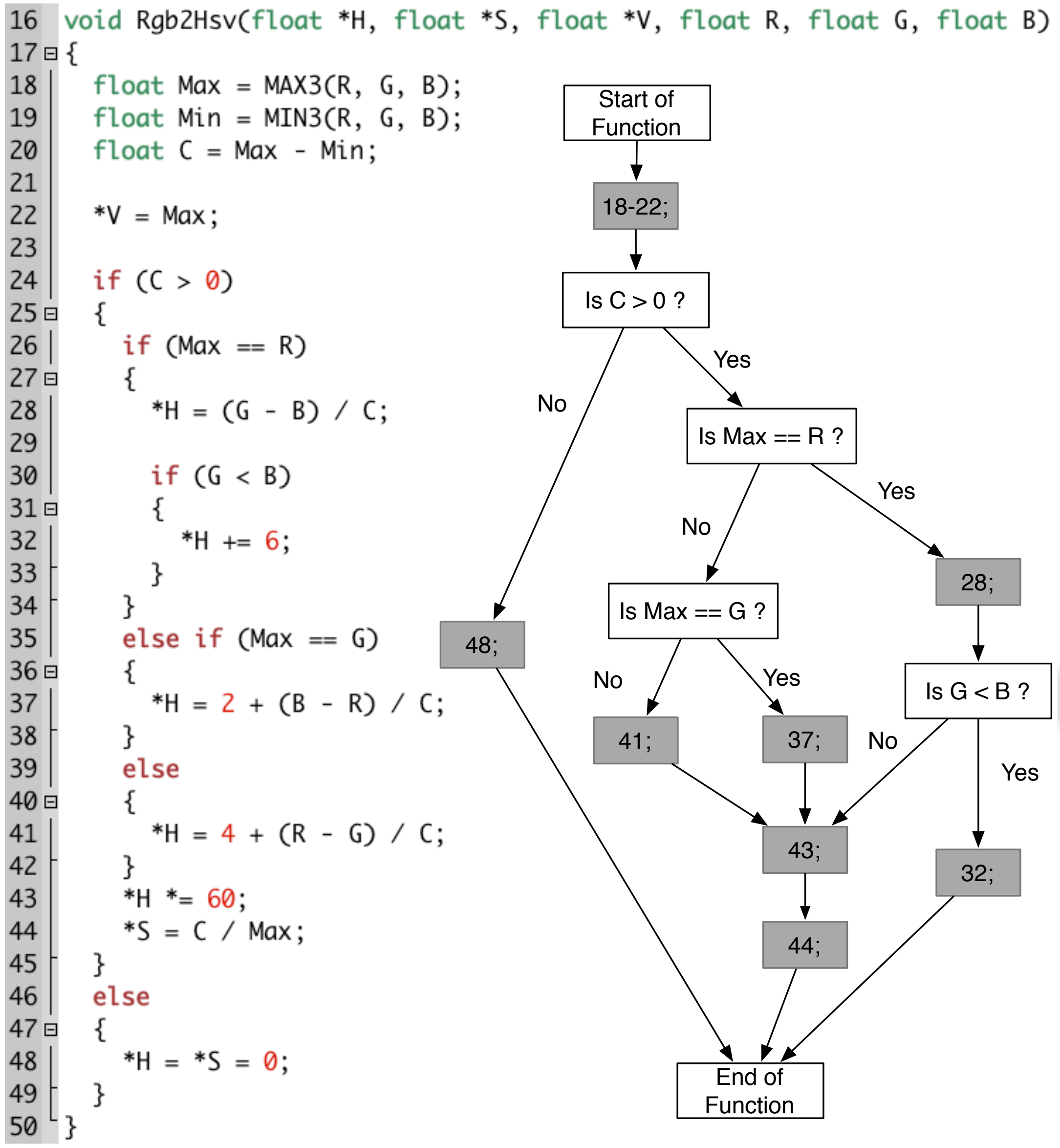
- Difference of instruction cache to data cache
  1. Amount of instructions to execute is proportional to the size of code
  2. The size of code is proportional to the complexity of problem
  3. Complexity of problem is fixed (SMC exception)

- Programmers design the data handling, but don't have to think about instructions
  - Compiler writers have good rules for optimising code generation

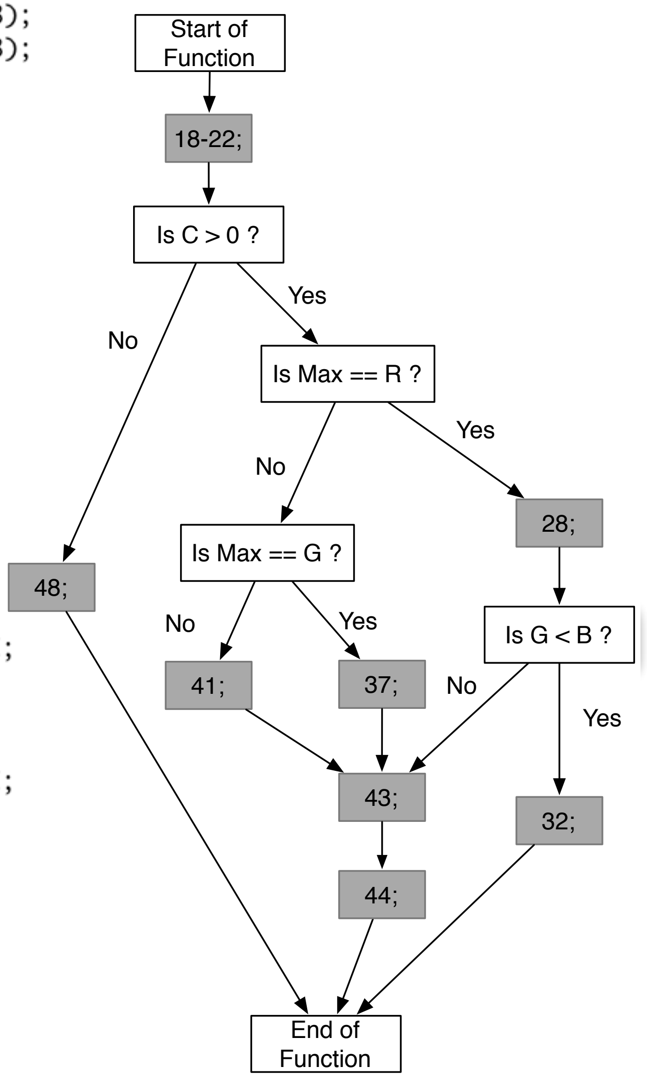
- Access pattern of instructions much more predictable than data
  - Good in both spatial and temporal locality
  - Don't need same hardware cache protocol as data, simpler is better!

Memory: instruction access pattern

- Convert colour space
  - Red Green Blue -> Hue Saturation Value
- More branches make it harder to predict next instruction to load
- Poor utilisation of prefetch for instructions and memory
- Q: How many unique execution paths?

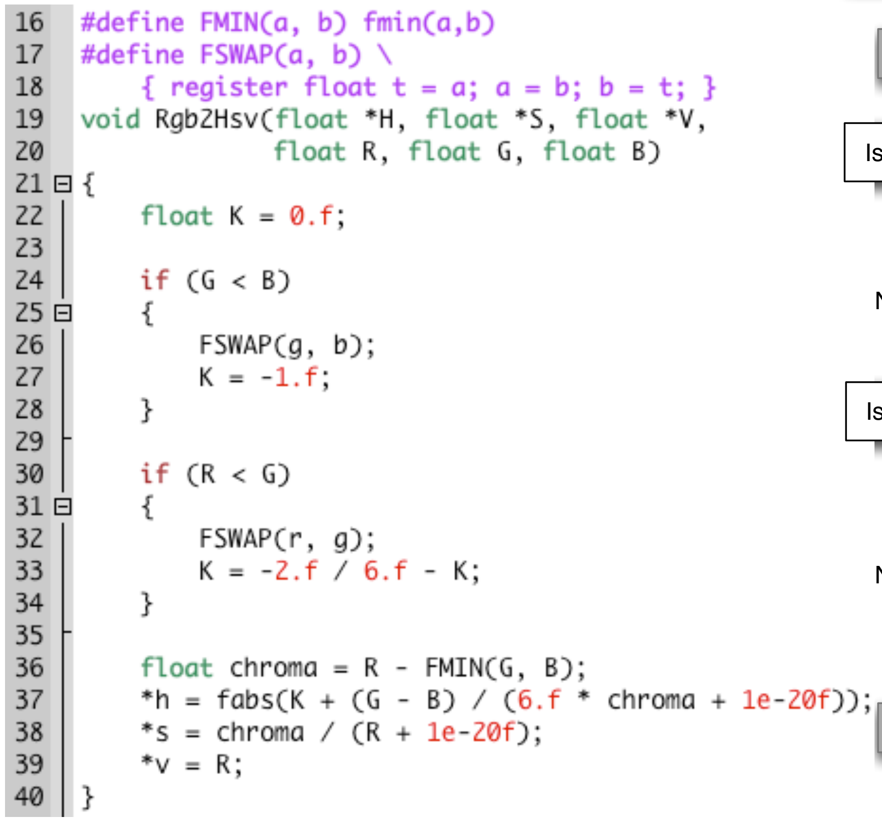


```
16 void Rgb2Hsv(float *H, float *S, float *V, float R, float G, float B)
17 {
18   float Max = MAX3(R, G, B);
19   float Min = MIN3(R, G, B);
20   float C = Max - Min;
21
22   *V = Max;
23
24   if (C > 0)
25   {
26     if (Max == R)
27     {
28       *H = (G - B) / C;
29
30       if (G < B)
31       {
32         *H += 6;
33       }
34     }
35     else if (Max == G)
36     {
37       *H = 2 + (B - R) / C;
38     }
39     else
40     {
41       *H = 4 + (R - G) / C;
42     }
43     *H *= 60;
44     *S = C / Max;
45   }
46   else
47   {
48     *H = *S = 0;
49   }
50 }
```

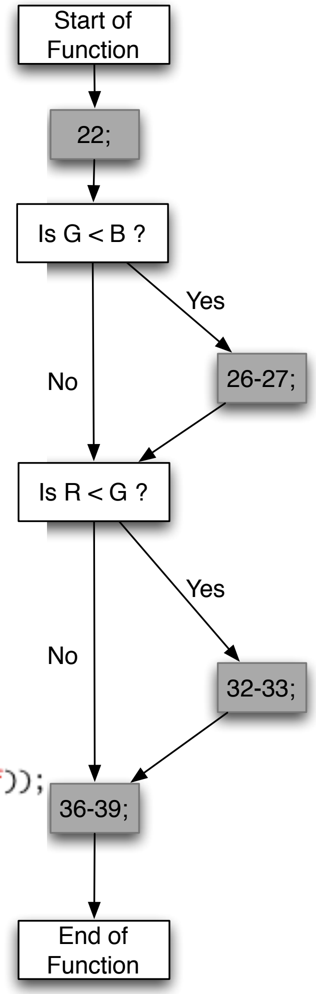


Memory: instruction access pattern

- Convert colour space
- Better version enjoys 10 - 30% increased performance!



```
16  #define FMIN(a, b) fmin(a,b)
17  #define FSWAP(a, b) \
18      { register float t = a; a = b; b = t; }
19  void Rgb2Hsv(float *H, float *S, float *V,
20               float R, float G, float B)
21  {
22      float K = 0.f;
23
24      if (G < B)
25      {
26          FSWAP(g, b);
27          K = -1.f;
28      }
29
30      if (R < G)
31      {
32          FSWAP(r, g);
33          K = -2.f / 6.f - K;
34      }
35
36      float chroma = R - FMIN(G, B);
37      *h = fabs(K + (G - B) / (6.f * chroma + 1e-20f));
38      *s = chroma / (R + 1e-20f);
39      *v = R;
40  }
```



Memory: data access pattern

- Sequential access

- Depends on what you want to do
  - E.g. Matrix vector multiply on CPU
- Depends on architecture
  - One will work better than another
- Depends on cost of duplication
  - maintain associative data structure
  - memory and computation overhead

- Known limits should be exploited
  - Cache line size (L1 and L2)
  - Cache capacity (keep shared data close)
  - SIMD instructions

```
// Array of structures
struct Vertex { float x, y, z; }
Vertex myvertices[1000];

// structure of arrays
struct VertexChunk
{ float x[1000], y[1000], z[1000]; }
VertexChunk myvertices;

// hybrid, SIMD exploit (4 at a time)
struct Vertex4
{ float x[4], y[4], z[4]; }
Vertex4 myvertices[250];
```

Memory: data access pattern

```
struct _node {
    // other things
    struct _node *next;
}
void list_op(struct _node *head)
{
    struct _node *p;
    p = head;
    while (p->p_next) {
        p = p->p_next;
    }
}
```

- Random access
- Linked list and tree like structures
  - Memory is not laid out sequentially
  - Pointer redirection can cause delayed

  access to data
- Cache miss: CPU can wait ~100 times longer than it takes to follow the pointer (if the cache has that address)
- Compiler designers try to predict the patterns of data access by code analysis, but if they can’t, this makes the purpose of cache for locality of data almost useless
- Just in time compilation can help (Java)
- Allocation of memory*

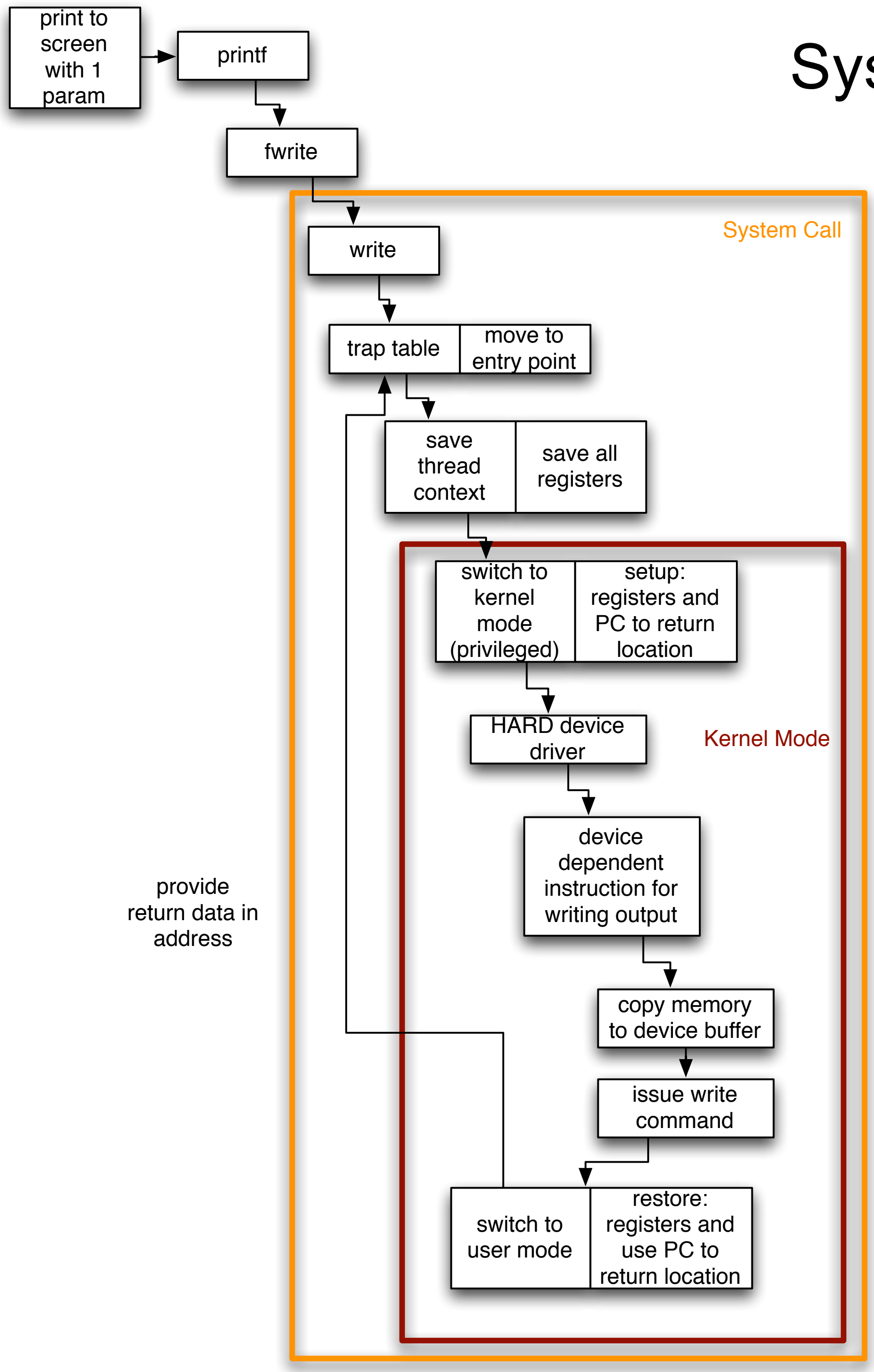
Memory: Exercise

```
void TransformVectors0( float *pDestVectors,
     float const (*pMatrix)[3],
     float const *pSourceVectors, int NumberOfVectors )
{
     int Counter, i, j;
     for ( Counter = 0; Counter < NumberOfVectors; Counter++) {
          for ( i = 0; i < 3; i++ ) {
               float Value = 0.0f;
               for ( j = 0; j < 3; j++ ) {
                    Value += pMatrix[i][j] * pSourceVectors[j];
               }
               *pDestVectors++ = Value;
          }
          pSourceVectors += 3;
     }
}
```

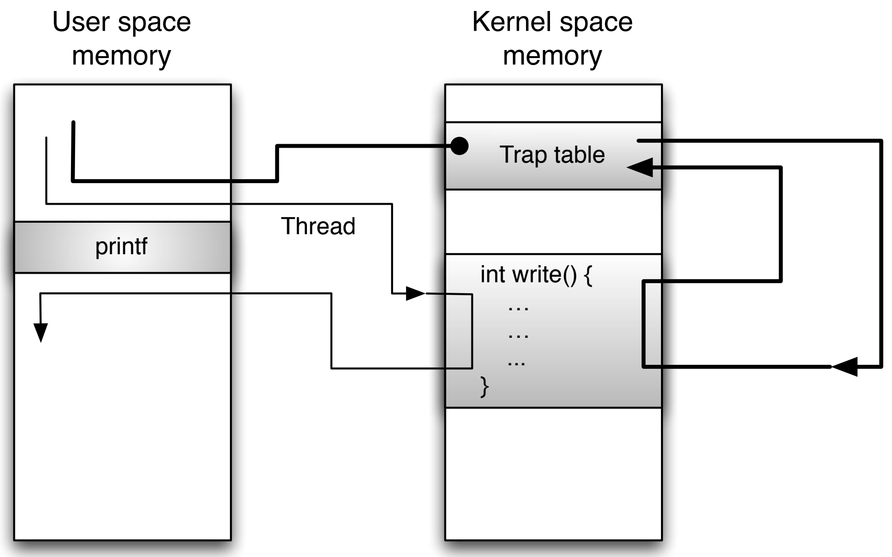
- Convert the above to improve performance
- More techniques for data access to consider
  - Chapter 5: Assigning work to processes statically (0321549422)
    - Block cyclic allocation

```
void TransformVectors0(
     float *pDestVectors, const float (*pMatrix)[3],
     const float *pSourceVectors, int NumberOfVectors )
{
     int Counter;
     float Value;
     float _Krr1;
     float _Krr2;
     for ( Counter = 0; Counter < NumberOfVectors; Counter++ ) {
          _Krr1 = pMatrix[0][0] * pSourceVectors[0];
          _Krr2 = pMatrix[1][0] * pSourceVectors[0];
          Value = pMatrix[2][0] * pSourceVectors[0];
          _Krr1 += pMatrix[0][1] * pSourceVectors[1];
          _Krr2 += pMatrix[1][1] * pSourceVectors[1];
          Value += pMatrix[2][1] * pSourceVectors[1];
          _Krr1 += pMatrix[0][2] * pSourceVectors[2];
          _Krr2 += pMatrix[1][2] * pSourceVectors[2];
          Value += pMatrix[2][2] * pSourceVectors[2];
          *pDestVectors++ = _Krr1;
          *pDestVectors++ = _Krr2;
          *pDestVectors++ = Value;
          pSourceVectors += 3;
     }
```

System call joy


print to screen with 1 param
printf
fwrite
System Call
write
trap table
move to entry point
save thread context
save all registers
switch to kernel mode (privileged)
setup: registers and PC to return location
HARD device driver
Kernel Mode
device dependent instruction for writing output
copy memory to device buffer
issue write command
switch to user mode
restore: registers and use PC to return location

provide return data in address



References

Dr. David Levinthal PhD. Intel Corp. Cycle Accounting Analysis on Intel® Core^{TM}2 Processors.

Dr. David Levinthal PhD. Intel Corp. Performance Analysis Guide for Intel® Core^{TM} i7 Processor and Intel Xeon 5500 Processors. 2009

Ulrich Drepper Red Hat, Inc. What Every Programmer Should Know About Memory. 2007

Chris Hecker Definition Six Inc. PowerPC Compilers: Still Not So Hot. Game Developer June/July 1996

Calvin Lin and Larry Snyder. Principles of Parallel Programming. ISBN 0321549422

Randal E. Bryant and David R. O`Hallaron. Computer Systems: A Programmer`s Perspective 3rd Edition.

AMD64 Architecture Programmer’s Manual Volume 2: System Programming December 2017. https://support.amd.com/TechDocs/24593.pdf

Outline

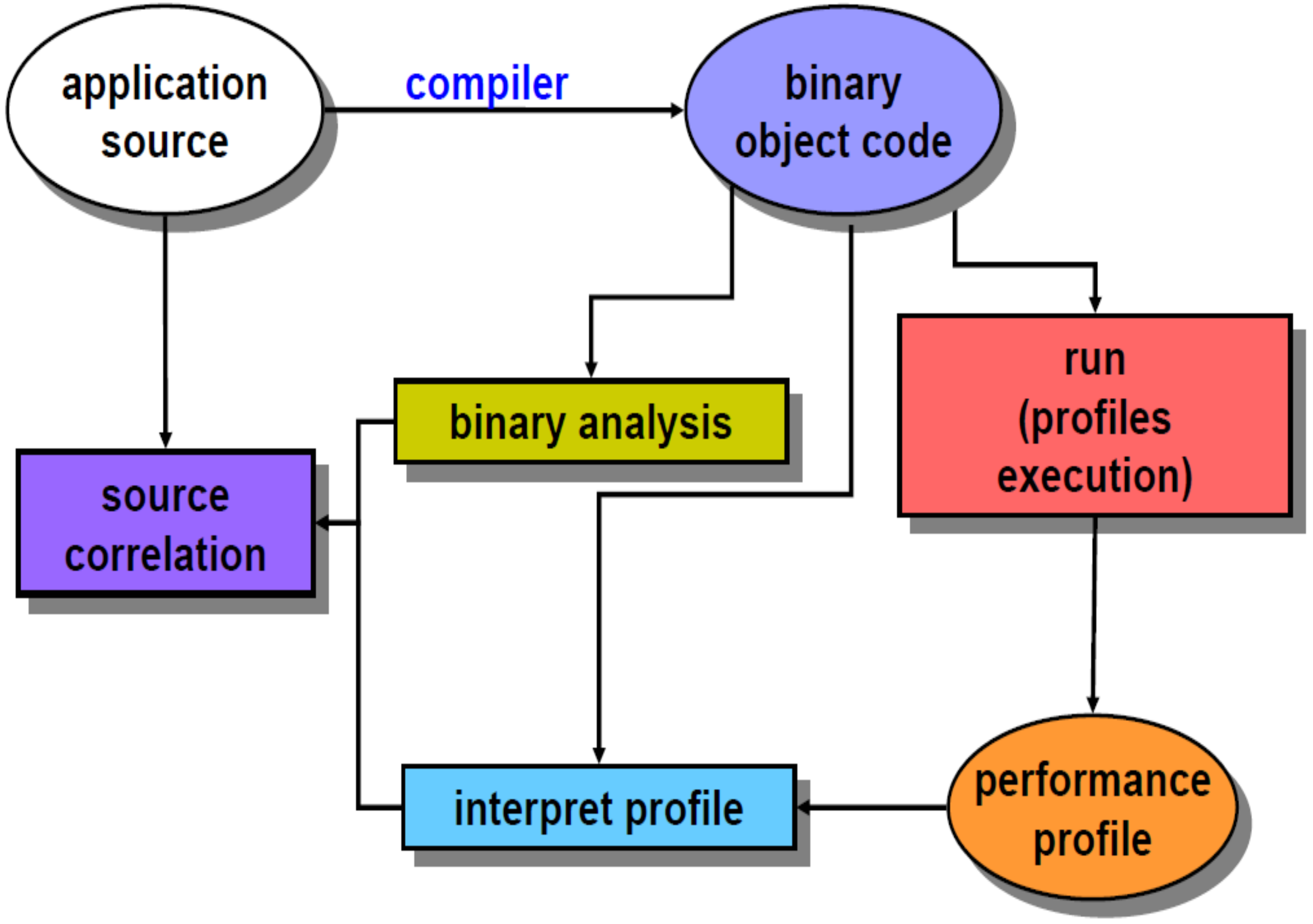
- Amdahl’s Law ✓
- Load Balancing ✓
- Measuring Performance ✓
- Sources of Performance Loss ✓
  - Example Case-study ✓
- Thread pool
- Reasoning about Performance ✓
- A closer look at memory ✓

Profiling

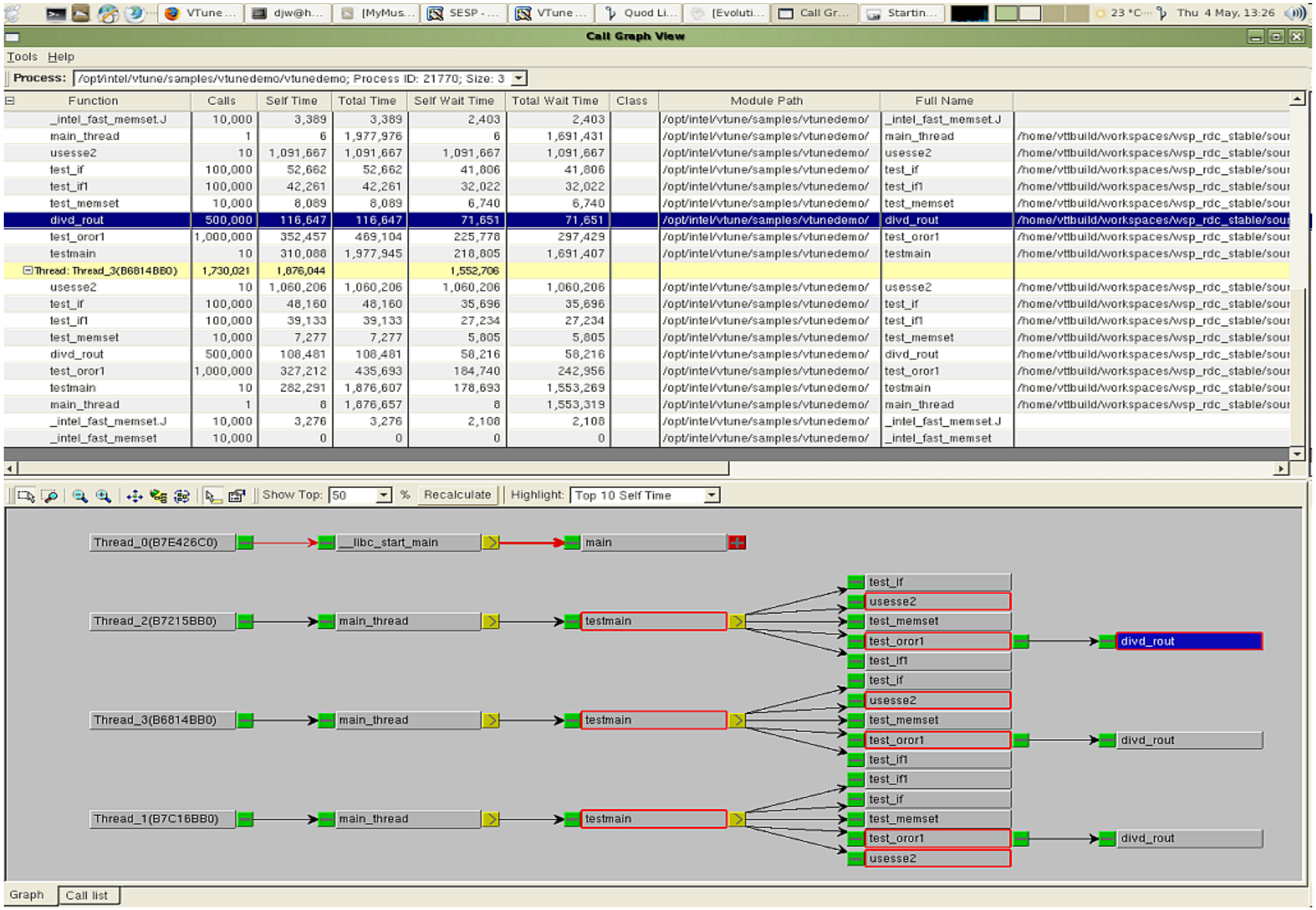
Profiling = Program Instrumentation + Execution

- 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**
- Install linux-perf → $ perf stat <binary>
- 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



Call Graph View

Process: /opt/intel/vtune/samples/vtunedemo/vtunedemo; Process ID: 21770; Size: 3

Function	Calls	Self Time	Total Time	Self Wait Time	Total Wait Time	Class	Module Path	Full Name	
_intel_fast_memset.J	10,000	3,389	3,389	2,403	2,403		/opt/intel/vtune/samples/vtunedemo/	_intel_fast_memset.J	
main_thread	1	6	1,977,976	6	1,691,431		/opt/intel/vtune/samples/vtunedemo/	main_thread	/home/vttbuild/workspaces/wsp_rdc_stable/sour
usesse2	10	1,091,667	1,091,667	1,091,667	1,091,667		/opt/intel/vtune/samples/vtunedemo/	usesse2	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_if	100,000	52,662	52,662	41,806	41,806		/opt/intel/vtune/samples/vtunedemo/	test_if	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_if1	100,000	42,261	42,261	32,022	32,022		/opt/intel/vtune/samples/vtunedemo/	test_if1	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_memset	10,000	8,089	8,089	6,740	6,740		/opt/intel/vtune/samples/vtunedemo/	test_memset	/home/vttbuild/workspaces/wsp_rdc_stable/sour
divd_rout	500,000	116,647	116,647	71,651	71,651		/opt/intel/vtune/samples/vtunedemo/	divd_rout	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_oror1	1,000,000	352,457	469,104	225,778	297,429		/opt/intel/vtune/samples/vtunedemo/	test_oror1	/home/vttbuild/workspaces/wsp_rdc_stable/sour
testmain	10	310,088	1,977,945	218,805	1,691,407		/opt/intel/vtune/samples/vtunedemo/	testmain	/home/vttbuild/workspaces/wsp_rdc_stable/sour
Thread: Thread_3(B6814BB0)	1,730,021	1,876,044		1,552,706					
usesse2	10	1,060,206	1,060,206	1,060,206	1,060,206		/opt/intel/vtune/samples/vtunedemo/	usesse2	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_if	100,000	48,160	48,160	35,696	35,696		/opt/intel/vtune/samples/vtunedemo/	test_if	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_if1	100,000	39,133	39,133	27,234	27,234		/opt/intel/vtune/samples/vtunedemo/	test_if1	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_memset	10,000	7,277	7,277	5,805	5,805		/opt/intel/vtune/samples/vtunedemo/	test_memset	/home/vttbuild/workspaces/wsp_rdc_stable/sour
divd_rout	500,000	108,481	108,481	58,216	58,216		/opt/intel/vtune/samples/vtunedemo/	divd_rout	/home/vttbuild/workspaces/wsp_rdc_stable/sour
test_oror1	1,000,000	327,212	435,693	184,740	242,956		/opt/intel/vtune/samples/vtunedemo/	test_oror1	/home/vttbuild/workspaces/wsp_rdc_stable/sour
testmain	10	282,291	1,876,607	178,693	1,553,269		/opt/intel/vtune/samples/vtunedemo/	testmain	/home/vttbuild/workspaces/wsp_rdc_stable/sour
main_thread	1	8	1,876,657	8	1,553,319		/opt/intel/vtune/samples/vtunedemo/	main_thread	/home/vttbuild/workspaces/wsp_rdc_stable/sour
_intel_fast_memset.J	10,000	3,276	3,276	2,108	2,108		/opt/intel/vtune/samples/vtunedemo/	_intel_fast_memset.J	
_intel_fast_memset	10,000	0	0	0	0		/opt/intel/vtune/samples/vtunedemo/	_intel_fast_memset	

Show Top: 50 % Highlight: Top 10 Self Time