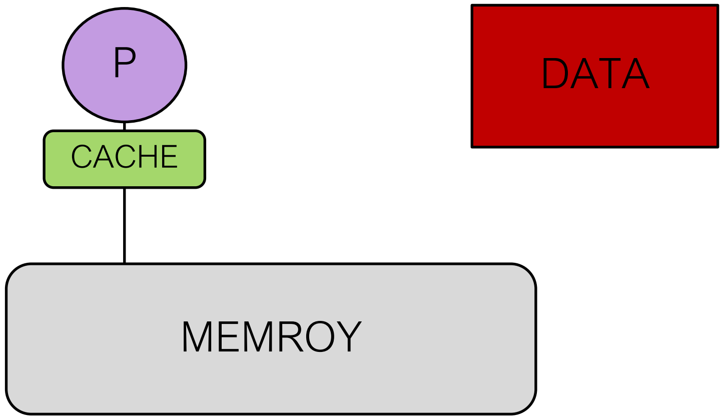
Speedup considerations

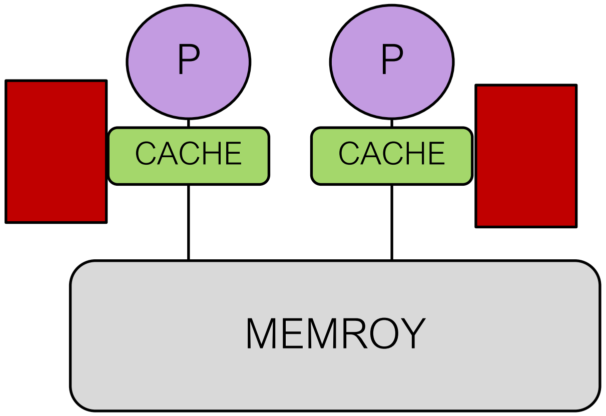
- If $S > p$ => superlinear speedup
  - Wait, if all processes spend less than Ts/p, why not use 1 process to run the whole thing then?
- Superlinear speedup can only happen if sequential algorithm is at a disadvantage compared to parallel version
  - Data too large to fit into 1 processor's cache => data accesses are slower for serial algorithm



If the program needs to stream the data n times, the data does not fit in the cache so the data has to be moved between the memory and caches n times!

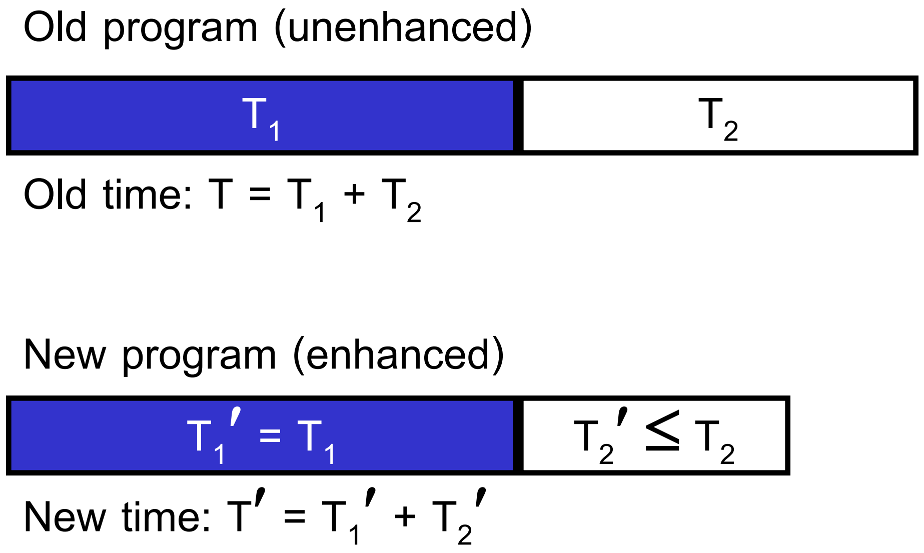
Speedup considerations

- If $S > p$ => superlinear speedup
  - Wait, if all processes spend less than Ts/p, why not use 1 process to run the whole thing then?
- Superlinear speedup can only happen if sequential algorithm is at a disadvantage compared to parallel version
  - Data too large to fit into 1 processor's cache => data accesses are slower for serial algorithm



If the half of the data fits in one of the L1 caches and work can be divided between the processors, then the data only gets loaded once into the caches from memory!

Speedup



T_1 = time that can NOT be enhanced.

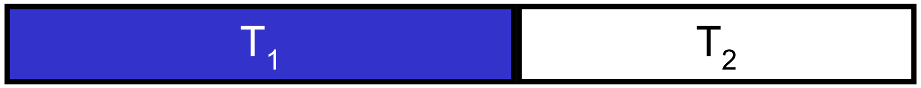
T_2 = time that can be enhanced.

T_2' = time after the enhancement.

Speedup: $S_{overall} = T / T'$

Amdahl's law

- Suppose only part of an application is parallel



- Amdahl's law

If $T = T_1 + T_2 = 1$

  - T_1 = fraction of work done sequentially (Amdahl fraction), so (T_2 =1 - T_1) is fraction parallelizable
  - p = number of processors

$$\text{Speedup}(P) = T / T'$$
$$\le 1/(T_1 + (1 - T_1)/p)$$
$$\le 1/T_1$$

- Even if the parallel part speeds up perfectly performance is limited by the sequential part

Efficiency

- The fraction of time when processes doing useful work

  $E = S / p$

- What is the ideal efficiency?
- What are the range of values for E?
- What is the efficiency of calculating the sum of n array elements on n processes?

Efficiency

- The fraction of time when processes doing useful work

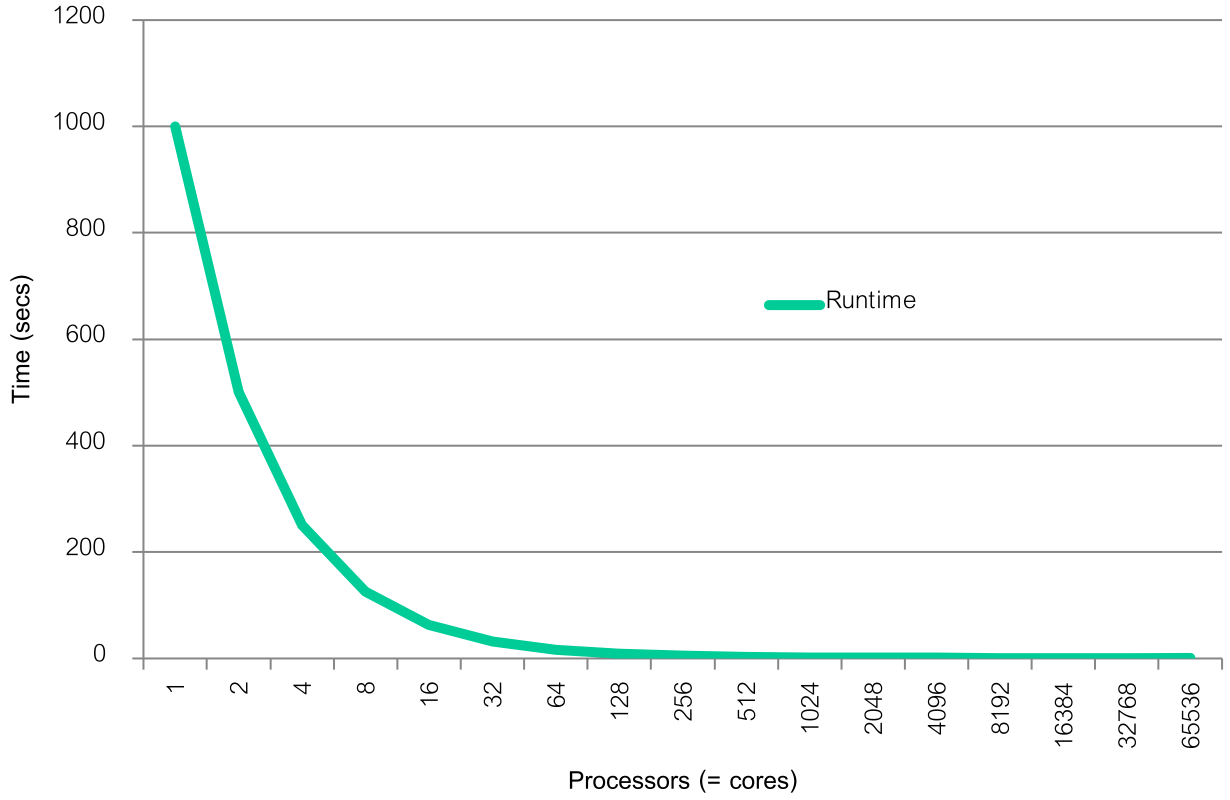
  $E = S / p$

- What is the ideal efficiency? 1 (sometimes shown as 100%)
- What are the range of values for E? 0 to 1
- What is the efficiency of calculating the sum of n array elements on n processes?

  $E = \Theta(n / \log n) \; / \; n$

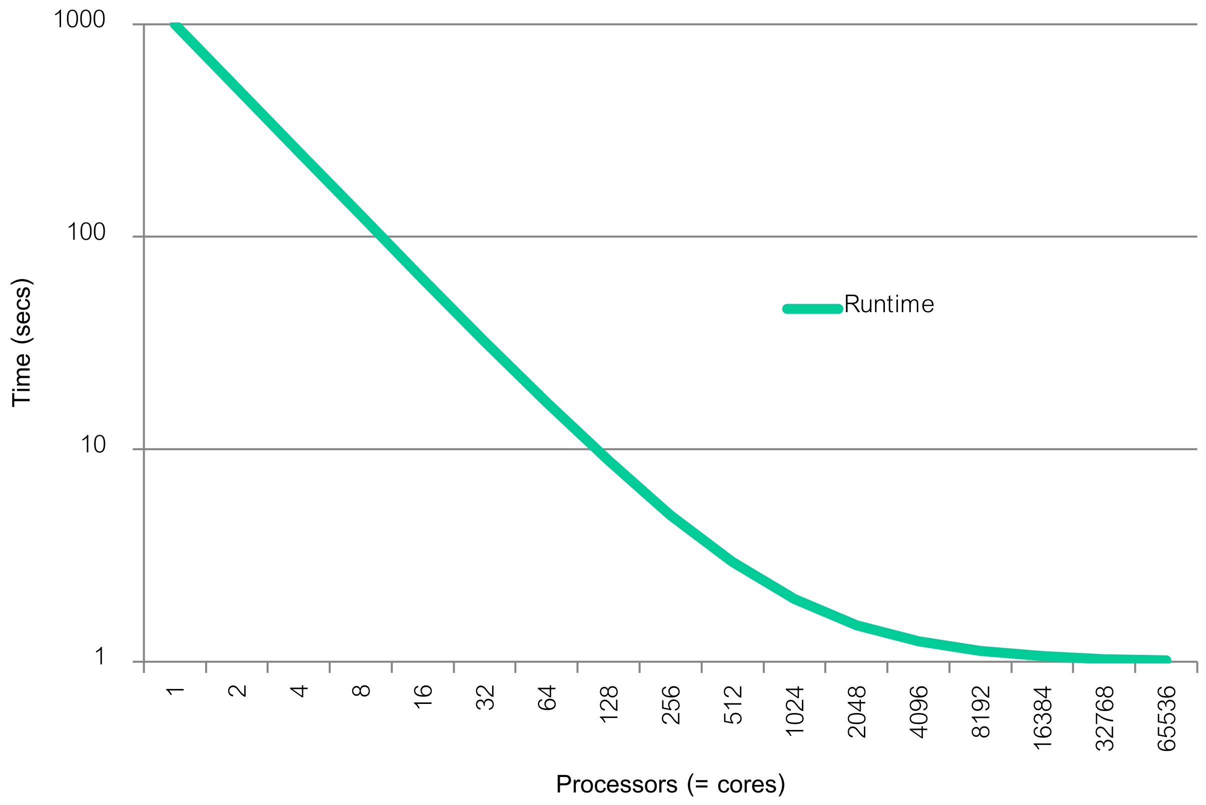
  $E = \Theta(1 / \log n)$

Reporting running time



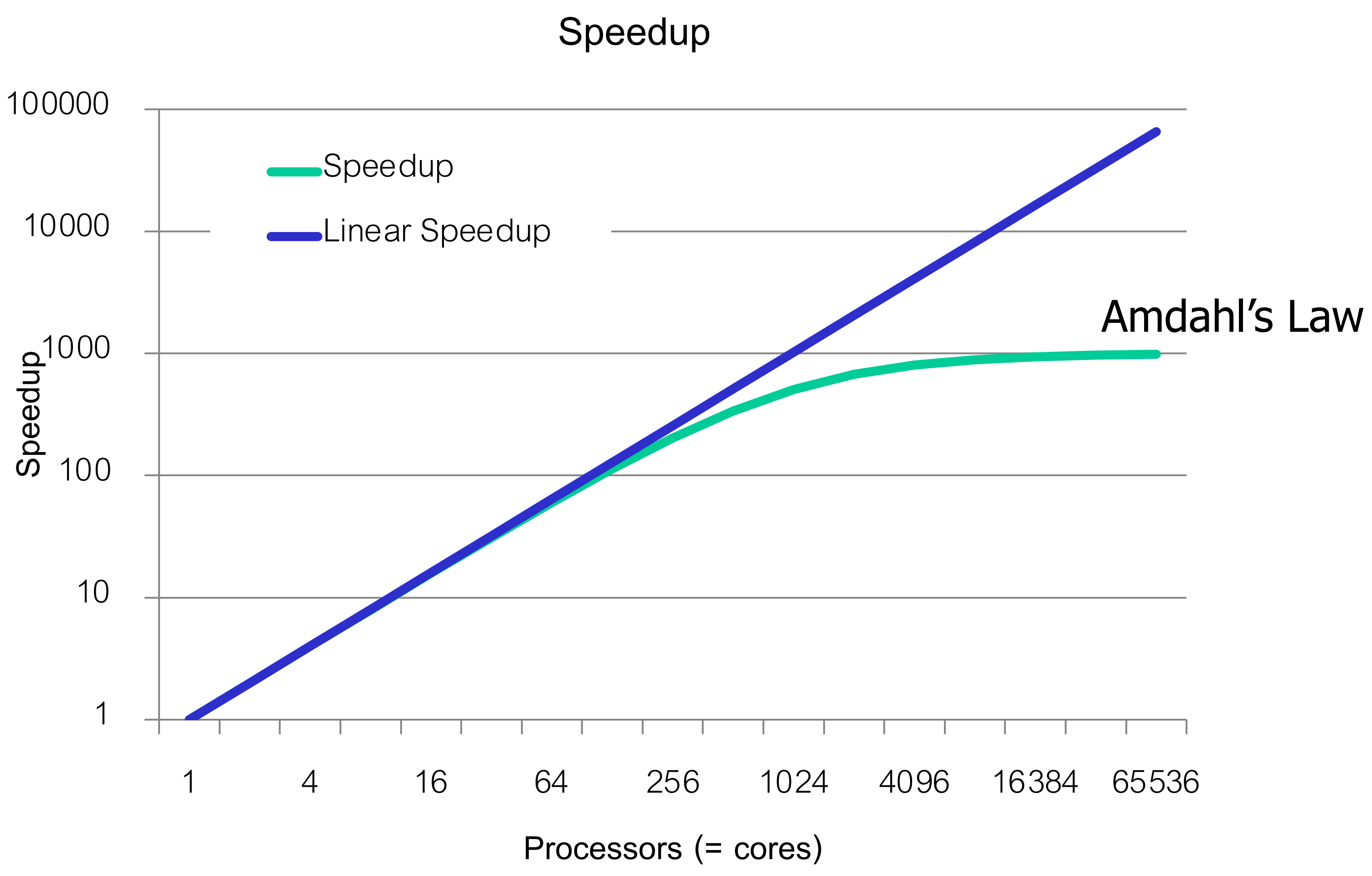
Hard to see performance gains from parallelism after 32 processors!

Reporting running time

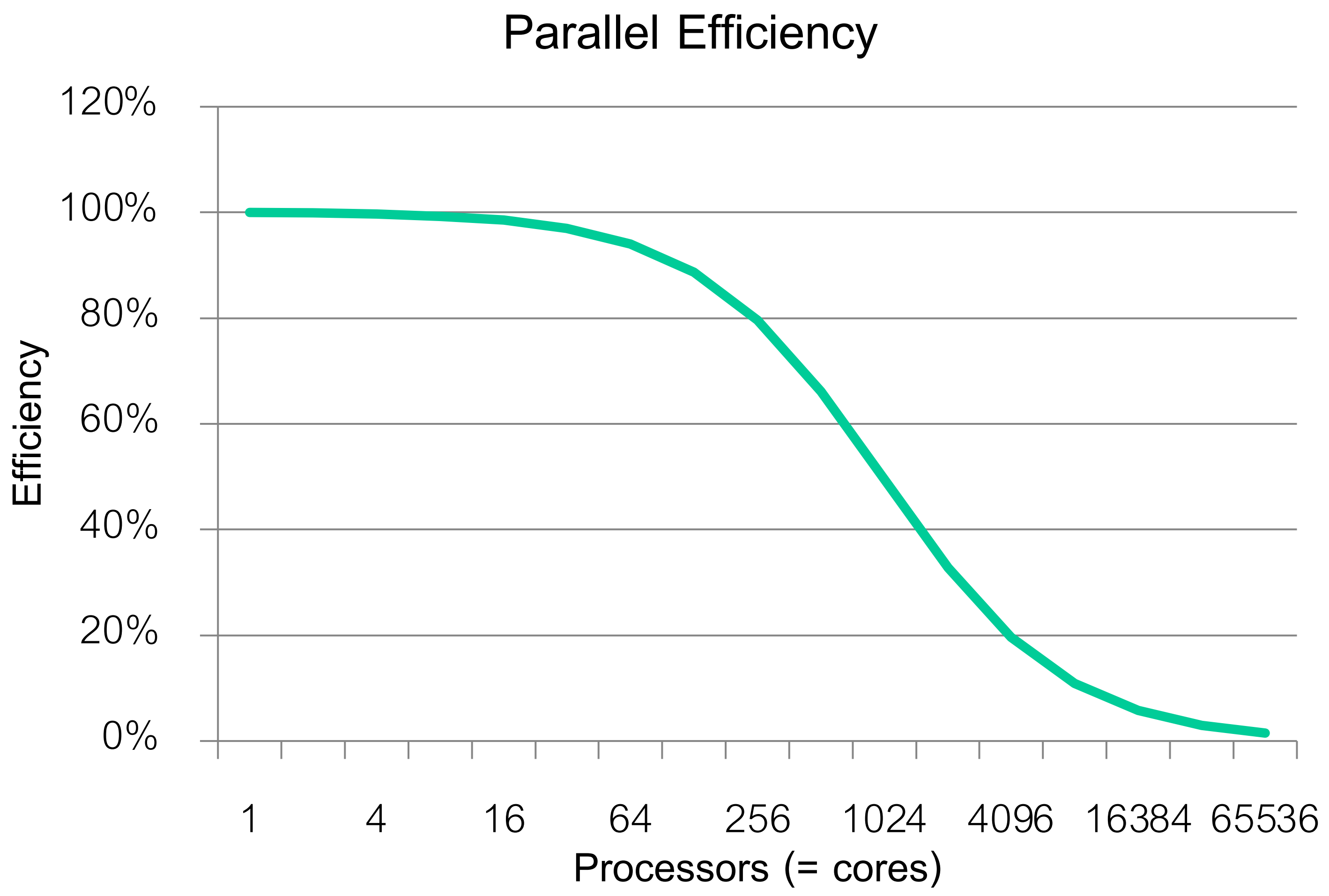


Lets take the y axis (running time) to log scale: A bit better!

Example Speedup Plot



Example Efficiency Plot



Carefully choose and report your serial/baseline

See *David Bailey's Twelve Ways to Fool the Masses*. Below are examples of how to fool the masses when reporting results from your parallel program:

1. Use 64-bit for baseline/serial and 32-bit for parallel numbers:
   - Correct approach: Use the same precision for both the parallel implementation and the serial/baseline: This type of "cheating" in speedup reports often happens in GPU parallel programming, where single-precision is faster than double-precision computing.
2. Use a bad algorithm for the baseline:
   - Correct approach: Always optimize the serial algorithm first and use it as the baseline for speedups.
3. Use a bad implementation for the baseline:
   - Correct approach: While optimizing the parallel code if you realize you could have optimized the serial version better, go back and optimize the serial code and use that as baseline.
4. Don't report running times at all:
   - Correct approach: Report running times as well as speedup.

Shared Memory Architectures and Their Parallel Programming Models!

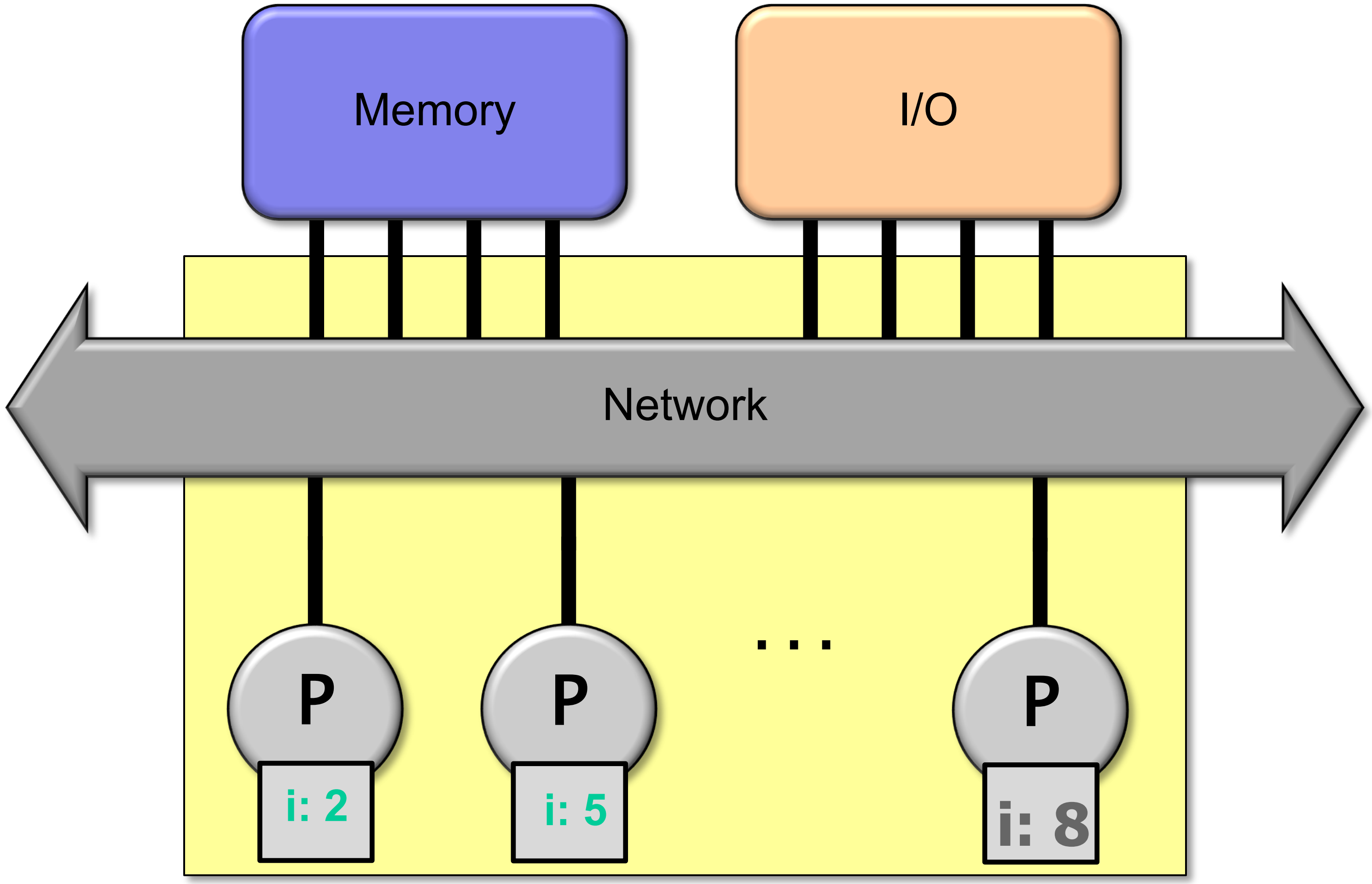
Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Shared Memory Architecture



Chip Multiprocessor (CMP)

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Parallel Programming Models

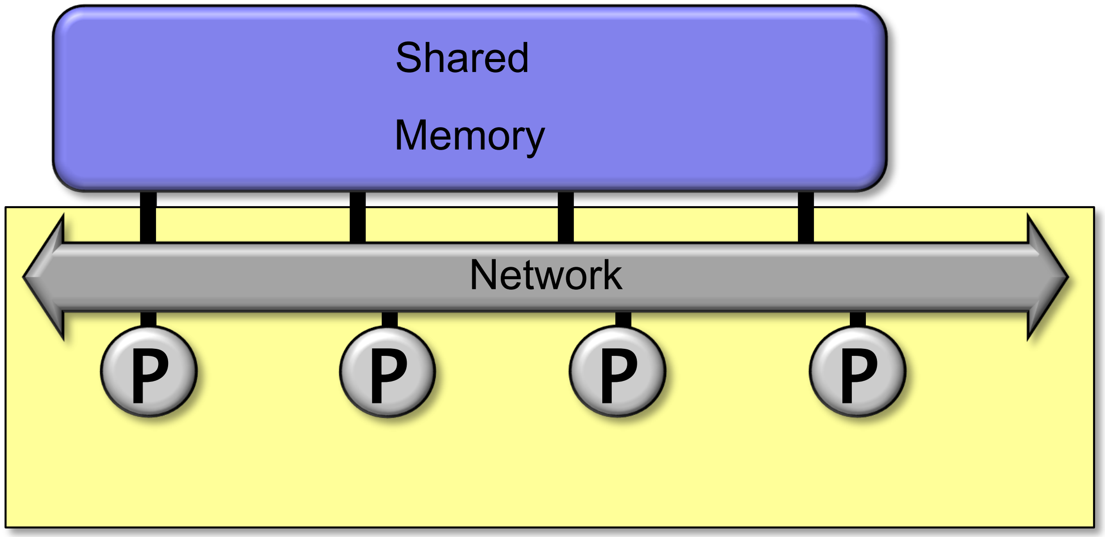
- Programming model is made up of the languages and libraries that create an abstract view of the machine: Pthreads!

The programming model enables us to identify

- Control
  - How is parallelism created?
  - What orderings exist between operations?
- Data:
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?

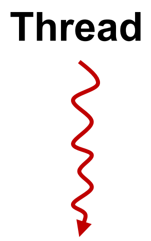
Programming Model: Shared Memory

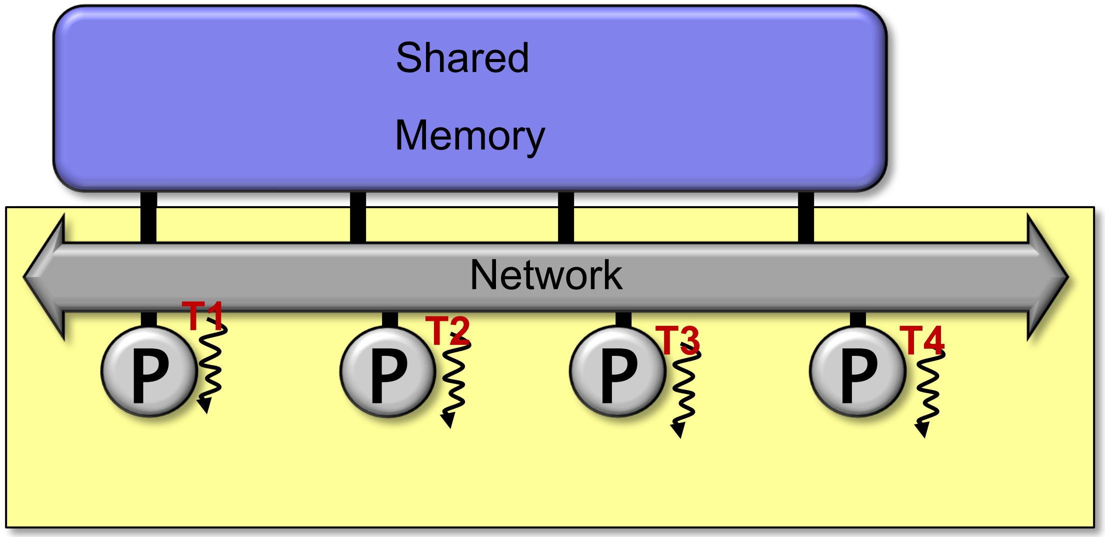
Program is a collection of threads of control, can be created mid-execution.



Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

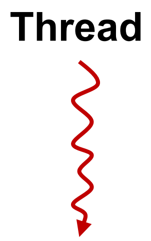


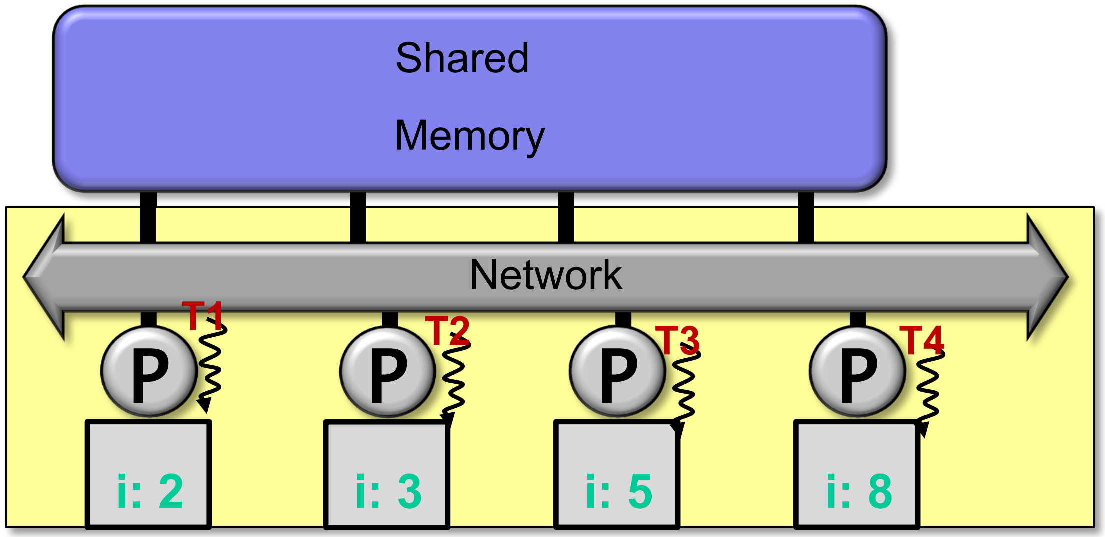


Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

Each thread has a set of private variables, e.g., local stack variables.





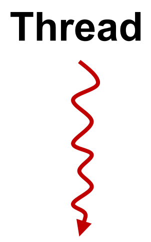
Programming Model: Shared Memory

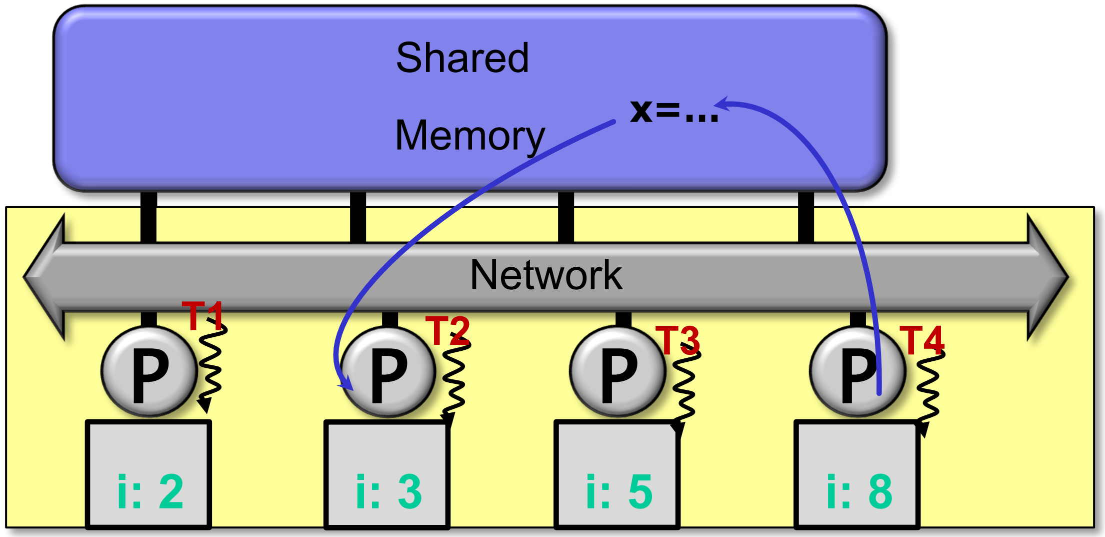
Program is a collection of threads of control, can be created mid-execution.

Each thread has a set of private variables, e.g., local stack variables.

Also a set of shared variables, e.g., static variables.

- Threads communicate implicitly by writing and reading shared variables.





Slide Source: Demmel

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Overview of POSIX Threads

- POSIX: ***P**ortable **O**perating **S**ystem **I**nterface*
    - Interface to Operating System utilities
- PThreads: The POSIX threading interface
    - System calls to create and synchronize threads
    - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
    - Creating parallelism
    - Synchronizing
    - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *, const pthread_attr_t *,  void * (*)(void *), void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute; &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attributes: minimum stack size, priority
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

“Simple” Threading Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Synchronization

- Threads interact in a multiprogrammed system
  - To share resources (such as shared data)
  - To coordinate their execution
- Arbitrary interleaving of thread executions can have unexpected consequences
  - We need a way to restrict the possible interleavings of executions
  - Scheduling is invisible to the application => cannot know when we lose control of the CPU and another thread/process runs
- Synchronization is the mechanism that gives us this control

Motivating Example

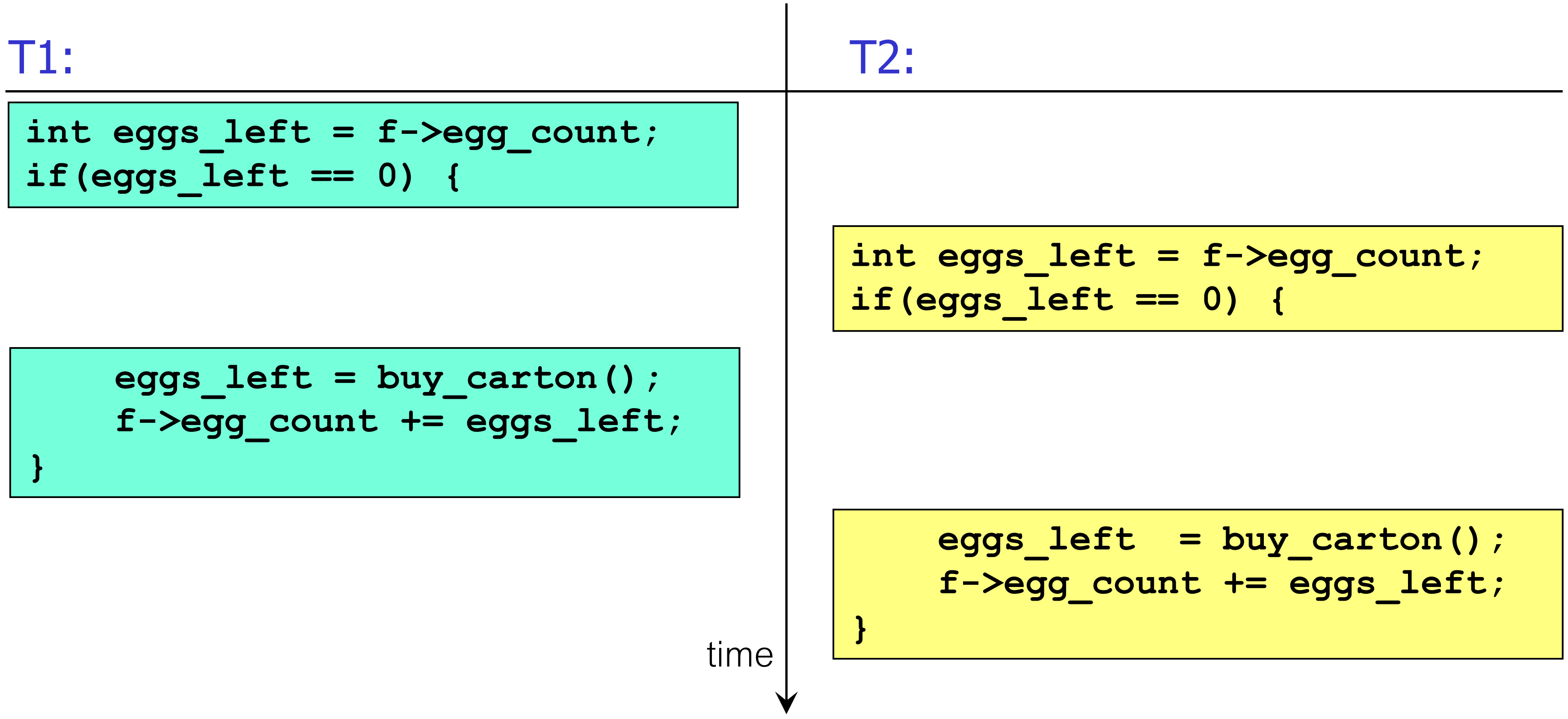
```
EggRun(fridge *f) {
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
        eggs_left = buy_carton();
        f->egg_count += eggs_left;
    }
}
```

```
EggRun(fridge *f) {
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
        eggs_left = buy_carton();
        f->egg_count += eggs_left;
    }
}
```

- Separate threads, which may run concurrently; eggs_left is local to each thread while the f->egg_count is shared
- Assume fridge has no eggs initially
- Think about potential schedules for these two threads

Interleaved Schedules

- The execution of the two threads can be interleaved:



T1:
```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
```

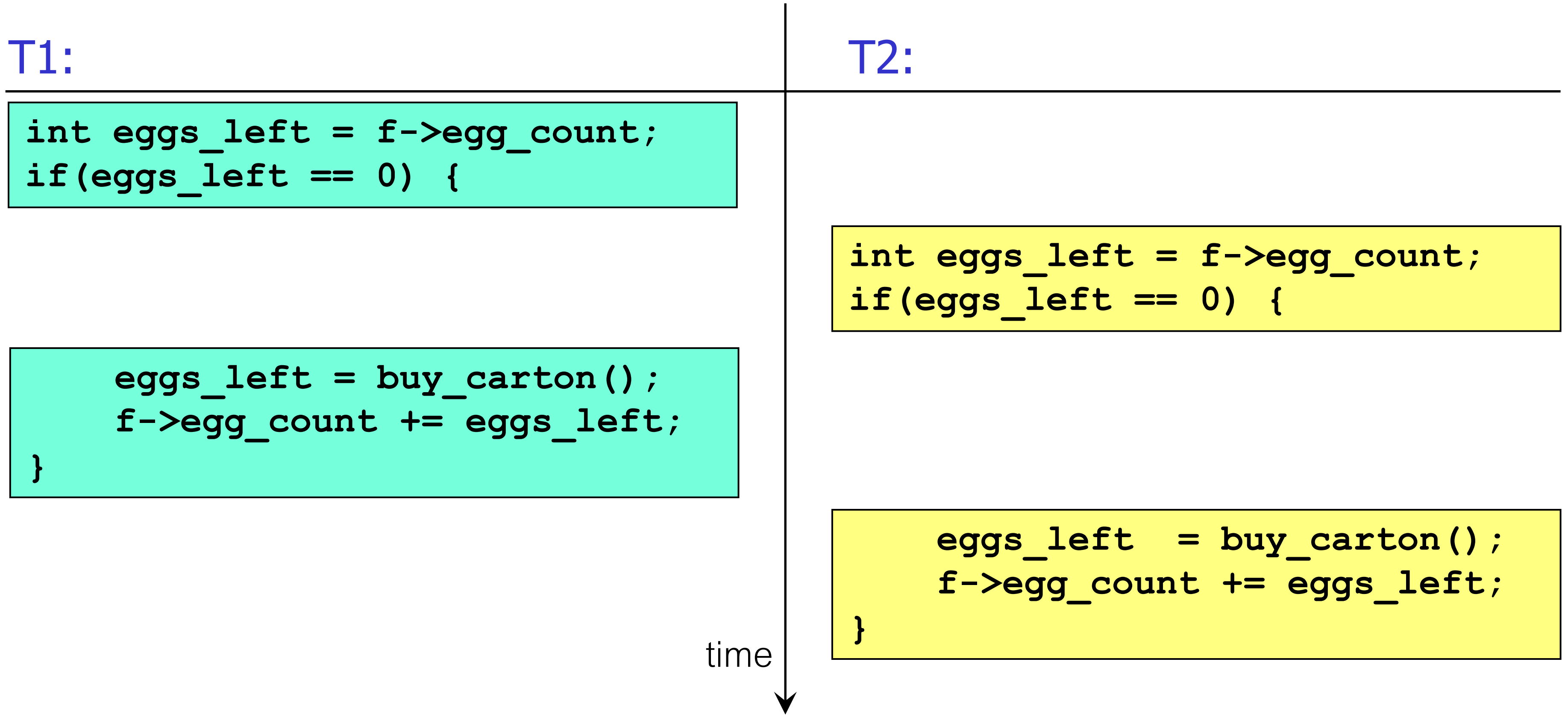
T2:
```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
```

T1:
```
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

T2:
```
    eggs_left  = buy_carton();
    f->egg_count += eggs_left;
}
```

Interleaved Schedules

- The execution of the two threads can be interleaved:



T1:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
```

T2:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
```

T1:

```
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

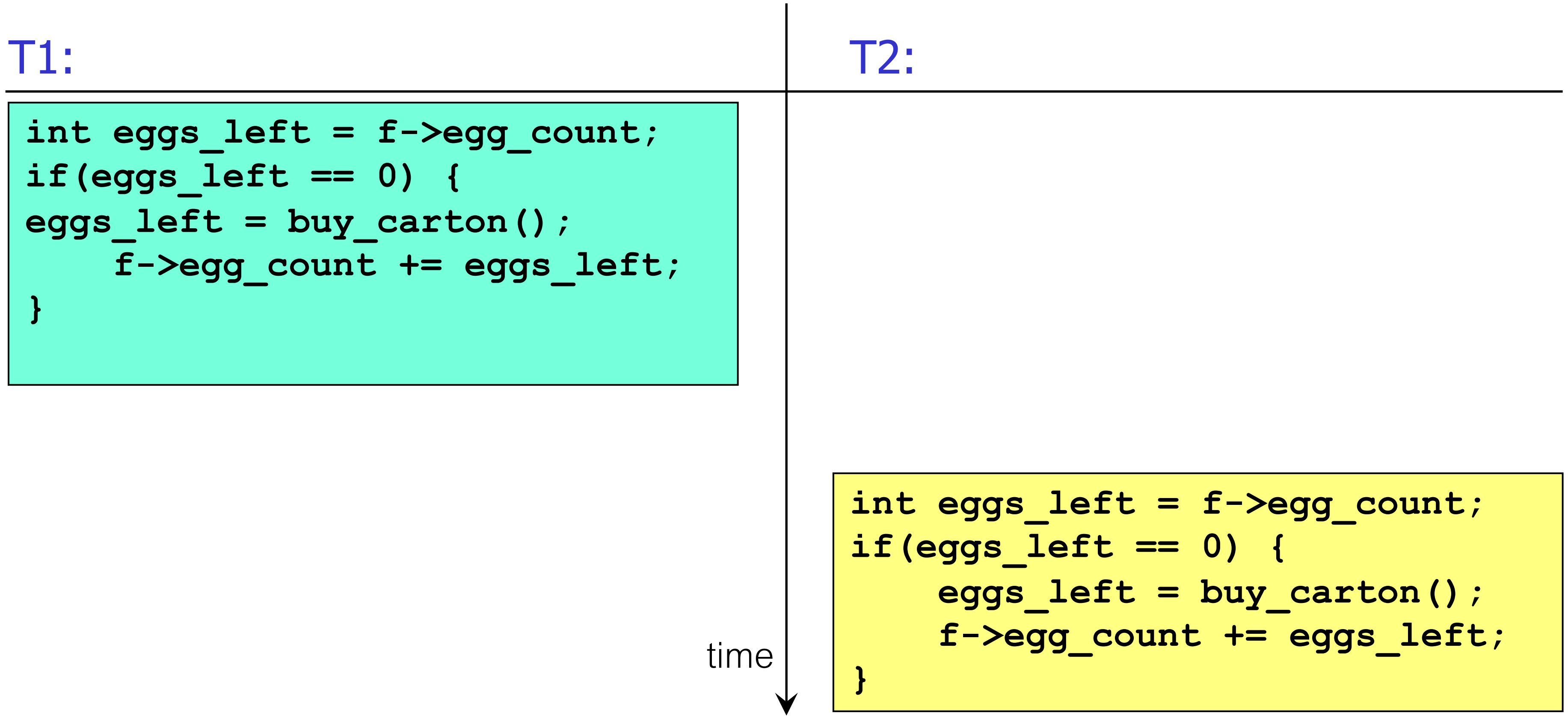
T2:

```
    eggs_left  = buy_carton();
    f->egg_count += eggs_left;
}
```

We end up buying **two** cartons of eggs

Interleaved Schedules

- The execution of the two threads can be interleaved:



T1:

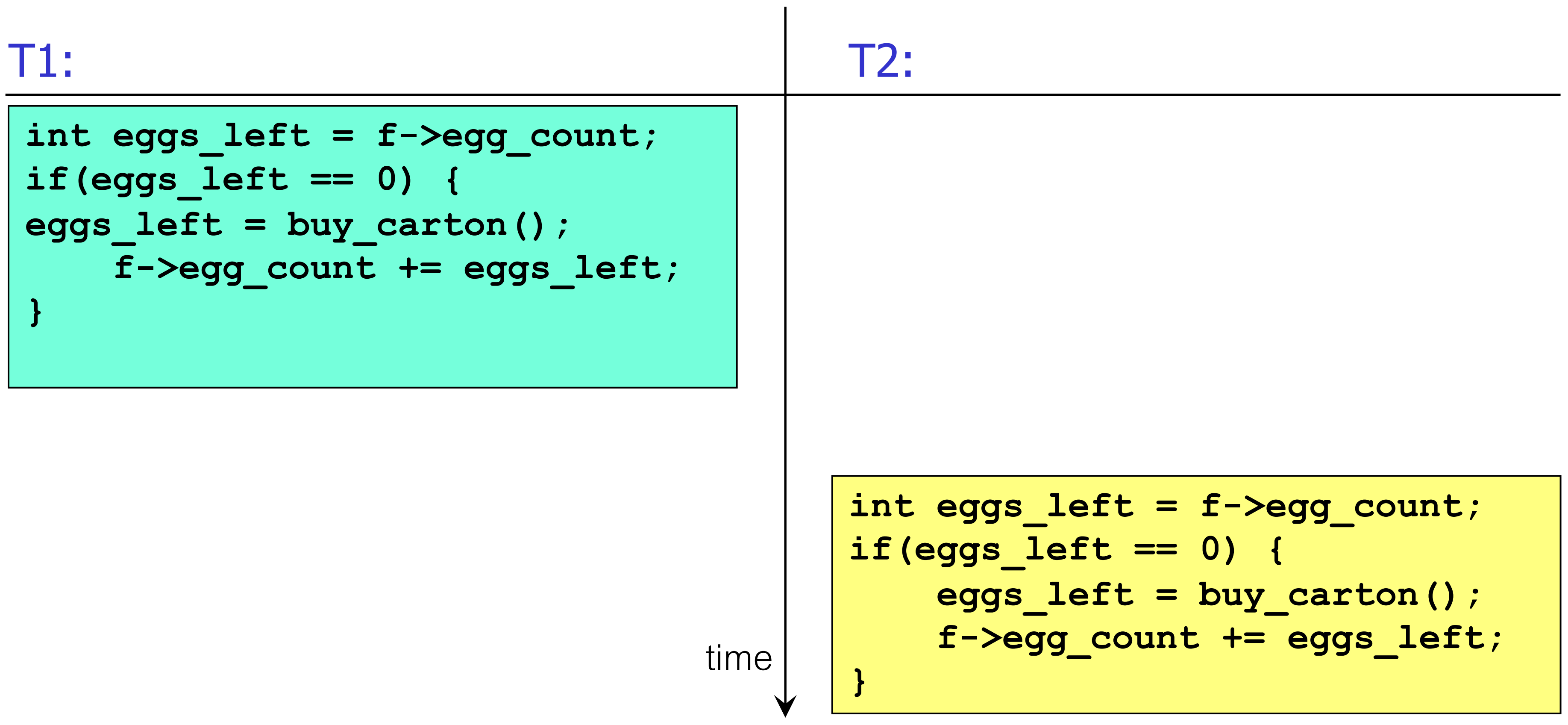
```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

T2:

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

Interleaved Schedules

- The execution of the two threads can be interleaved:



```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

We end up buying **one** carton of eggs

Race conditions and synchronization

- What happens when 2 or more concurrent threads manipulate a *shared resource* (e.g., a piece of data) without any synchronization?
  - The outcome depends on the order in which accesses take place!
  - This is called a ***race condition***
- We need to ensure that only one thread at a time can manipulate the shared resource
  - So that we can reason about correct program behavior

=> We need ***synchronization***

How do we handle this?

- How about whoever gets to check first, locks the fridge and takes the sole key, for the duration of the entire grocery run?
    - Nobody else can unlock the shared resource until the key owner unlocks it

Mutual Exclusion

- Given:
  - A set of *n* threads, T_0, T_1, …, T_{n-1}
  - A set of resources shared between threads
  - A segment of code which accesses the shared resources, called the *critical section, CS*
- We want to ensure that:
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another can enter

Mutex locks

- Typically associated to a resource, to ensure one access at a time, to that resource
- Ensure mutual exclusion to a critical section
- For Mutexes, a thread go to sleep when they see the lock is busy.

Next up ...

- Using locks for synchronization
- Common mistakes, potential correctness problems
- Coarse-grained vs. fine-grained locking
- Deadlocks

POSIX mutex API

- Pthreads library has builtin mutexes
    - You've seen these in the labs already
- Basic API:
    - pthread_mutex_t mutex;
    - pthread_mutex_init(pthread_mutex_t *mutex);
    - pthread_mutex_lock(pthread_mutex_t *mutex);
    - pthread_mutex_unlock(pthread_mutex_t *mutex);

Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {
    int egg_count;
    double milk_qty;
    pthread_mutex_t lock;
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {
    pthread_mutex_lock(f->lock);
    if(f->egg_count > 0) {
        f->egg_count --;
    }
    else {
        printf("Plan B: cereal\n");
    }
    pthread_mutex_unlock(f->lock);
}
```

```
EggRun(fridge *f) {
    pthread_mutex_lock(f->lock);
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
       eggs_left = buy_carton();
       f->egg_count += eggs_left;
    }
    pthread_mutex_unlock(f->lock);

    printf("Eggs refilled: %d remaining!", f->egg_count);
}
```

Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {
    int egg_count;
    double milk_qty;
    pthread_mutex_t lock;
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {
    pthread_mutex_lock(f->lock);
    if(f->egg_count > 0) {
        f->egg_count --;
    }
    else {
        printf("Plan B: cereal\n");
    }
    pthread_mutex_unlock(f->lock);
}
```

```
EggRun(fridge *f) {
    pthread_mutex_lock(f->lock);
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
       eggs_left = buy_carton();
       f->egg_count += eggs_left;
    }
    pthread_mutex_unlock(f->lock);

    printf("Eggs refilled: %d remaining!", f->egg_count);
}
```

Problem?

Potential correctness problems

- Both reads and writes to shared data must be locked, if a concurrent write is possible

```
typedef struct {
    int egg_count;
    double milk_qty;
    pthread_mutex_t lock;
} fridge;
```

```
EatEggOrDieTrying(fridge *f) {
    pthread_mutex_lock(f->lock);
    if(f->egg_count > 0) {
        f->egg_count --;
    }
    else {
        printf("Plan B: cereal\n");
    }
    pthread_mutex_unlock(f->lock);
}
```

```
EggRun(fridge *f) {
    pthread_mutex_lock(f->lock);
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
       eggs_left = buy_carton();
       f->egg_count += eggs_left;
    }
    pthread_mutex_unlock(f->lock);

    printf("Eggs refilled: %d remaining!", f->egg_count);
}
```

No lock around printf in the yellow box so possible bogus output of

```
Eggs refilled:
 0 remaining!
```

Potential correctness problems

- Careful about losing track of a lock without unlocking
  - e.g., what happens here:

```
bool CanEatEggs(fridge *f) {
   pthread_mutex_lock(f->lock);
   int eggs_left = f->egg_count;
   if(eggs_left == 0){
       printf("Oh no!\n");
       return false;
   }
   printf("Yummy, eggs!\n");
   pthread_mutex_unlock(f->lock);
   return true;
}
```

- If a thread never releases a lock, all other waiting threads are stuck
  - Such concurrency bugs are called deadlocks! (more on this later...)