CSC367 Parallel computing

Lecture 8: Parallel Architectures and Parallel Algorithm Design Continued!

Thanks to Scinet, NSF XSEDE, James Demmel

Parallel Algorithm Design: Outline

- Tasks: Decomposition, Task Dependency, Granularity, Interaction, Mapping, Balance
- Decomposition techniques
- Mapping techniques to reduce parallelism overhead
- Parallel algorithm models
- Parallel program performance model

Parallel algorithm models

Model = 1 decomposition type + 1 mapping type + strategies to minimize interactions

Commonly used parallel algorithm models:

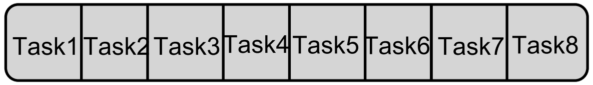
- Data parallel model
- Work pool model
- Master slave model

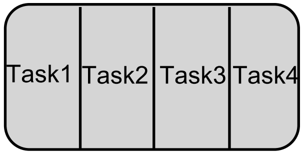
Data parallel model

- Decomposition: typically static and uniform data partitioning
- Mapping: static (mostly)
- Same operations on different data items, aka data parallelism
- Possible optimization strategies (depending on the problem and paradigm):
    - Choose a locality-preserving decomposition
    - Overlap computation with interaction
- This model scales really well with problem size (by adding more processes)

Work pool model

- Tasks are taken by processes from a common pool of work
- Decomposition: highly depends on the problem (data, recursive, etc.)
  - Can be statically available at start, or dynamically create more tasks during execution
- Mapping: dynamic
  - Any task can be performed by any process
- Possible strategies for reducing interactions:
  - Adjust granularity: tradeoff between load imbalance and overhead of accessing work pool





Master slave model

- Commonly used in distributed parallel architectures (more on this later)
- A master process generates work and allocates to worker (slave) processes
  - Could involve several masters, or a hierarchy of master processes
- Decomposition: highly depends on the problem (data, recursive)
  - Might be static if tasks are easy to break down a priori, or dynamic
- Mapping: Often dynamic
  - Any worker can be assigned any of the tasks
- Possible strategies for reducing interactions:
  - Choose granularity carefully so that master does not become a bottleneck
  - Overlap computation on workers with master generating further tasks

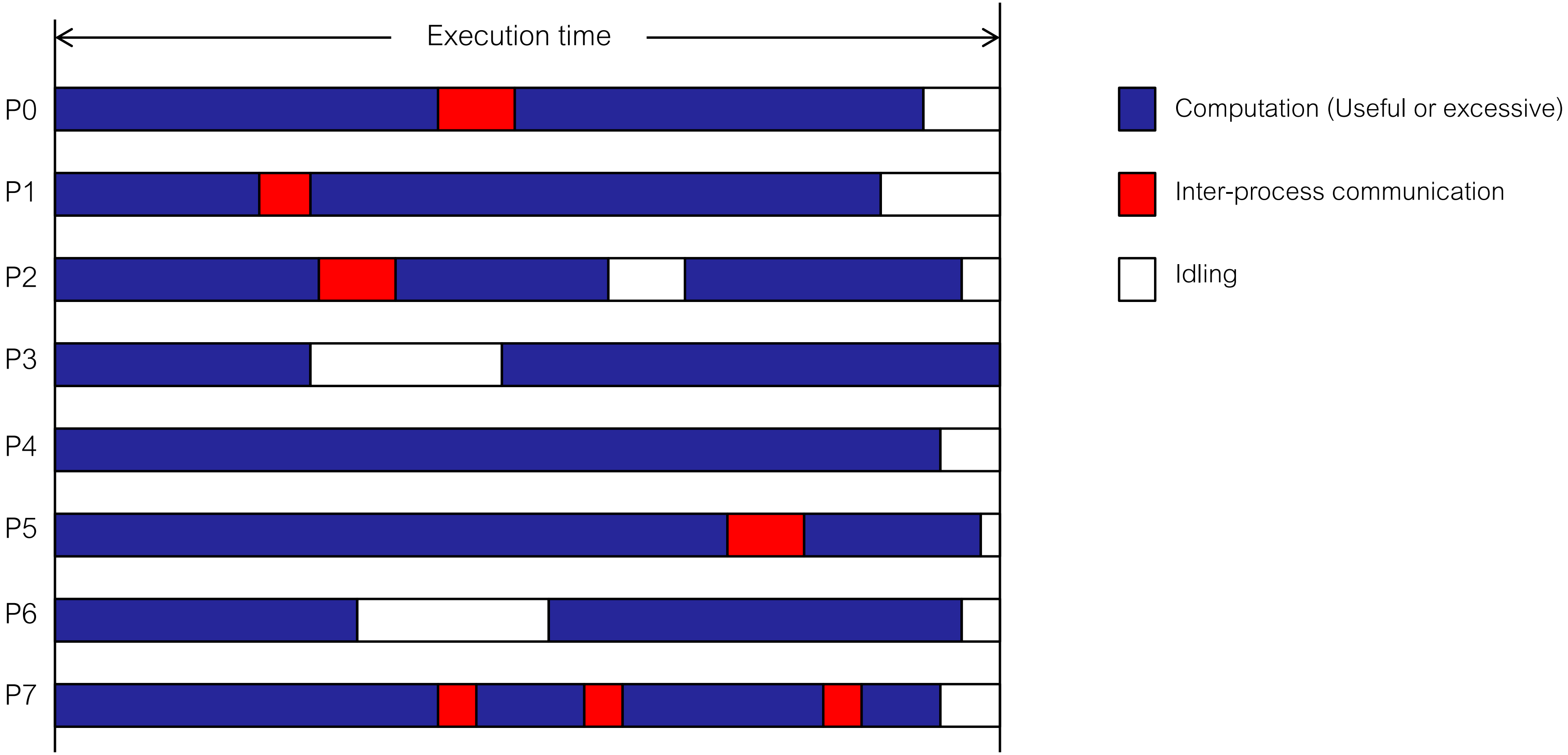
Parallel Algorithm Design: Outline

- Tasks: Decomposition, Task Dependency, Granularity, Interaction, Mapping, Balance
- Decomposition techniques
- Mapping techniques to reduce parallelism overhead
- Parallel algorithm models
- Parallel program performance model

Sources of overhead in parallel programs

- Parallel execution over N processes rarely results in N-fold performance gain
- Overheads of **inter-process communication**, **idling**, and/or **excess computation***

  *sometimes used to reduce communication
- e.g., some random application profile:

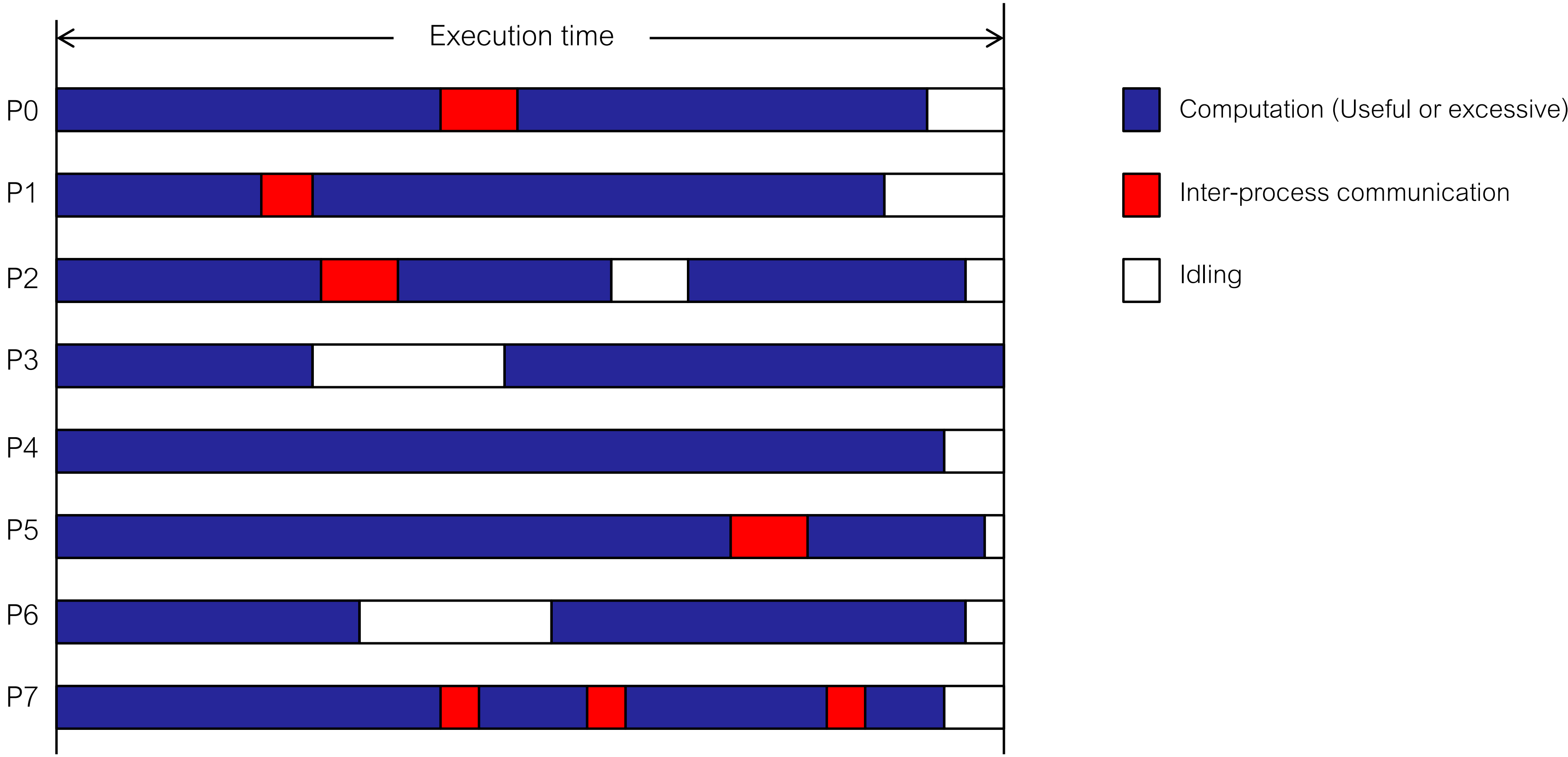


Performance metrics for parallel programs

- Execution time
- Speedup
- Efficiency
- Example performance plots
- Ways to fool the masses with performance results!

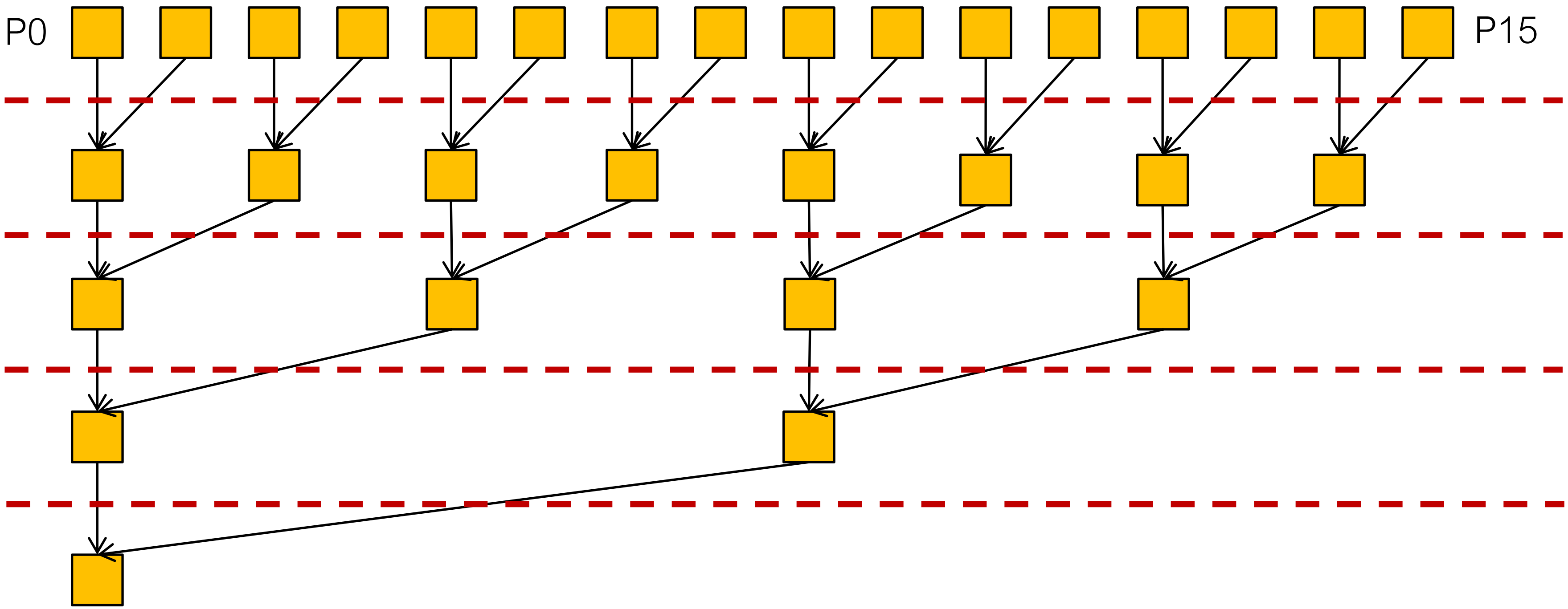
Execution time

- Execution time
  - Time elapsed from start of parallel computation and time when last process finishes
  - Determined by slowest process


Execution time
P0
P1
P2
P3
P4
P5
P6
P7
Computation (Useful or excessive)
Inter-process communication
Idling

Speedup

- Performance gain of the parallel algorithm over the sequential version
  - S = Ts (time of the serial execution) / Tp (time to execute on p processors)
- Example: sum of elements of an array, each process handles one element



- Tp = ϴ(log n) => S = ϴ(n / log n)

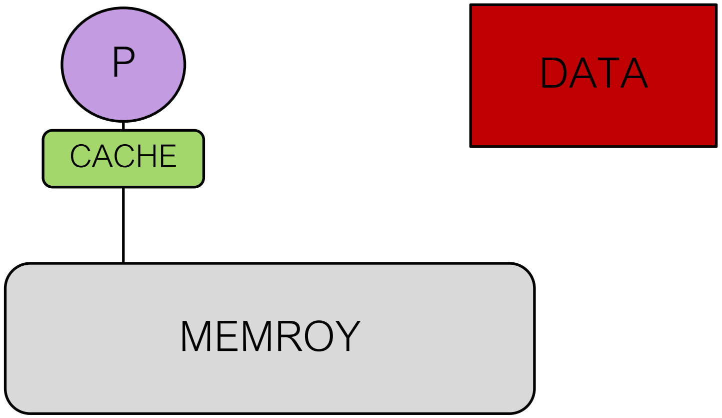
Speedup considerations

- Speedup is calculated relative to the best serial implementation of the algorithm. First improve your serial implementation, use that as the baseline, and optimize the improved serial implementation on p processors.

- Maximum achievable speedup is called linear speedup
  - Each process takes p times less than Ts => S = p

Speedup considerations

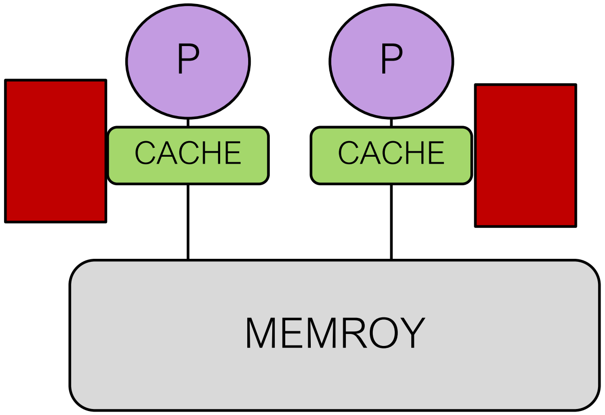
- If S > p => superlinear speedup
  - Wait, if all processes spend less than Ts/p, why not use 1 process to run the whole thing then?
- Superlinear speedup can only happen if sequential algorithm is at a disadvantage compared to parallel version
  - Data too large to fit into 1 processor's cache => data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

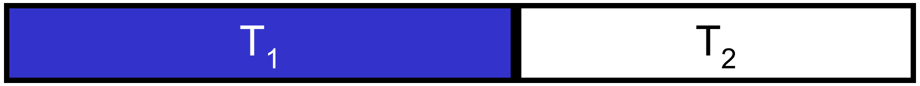
- If S > p => superlinear speedup
  - Wait, if all processes spend less than Ts/p, why not use 1 process to run the whole thing then?
- Superlinear speedup can only happen if sequential algorithm is at a disadvantage compared to parallel version
  - Data too large to fit into 1 processor's cache => data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

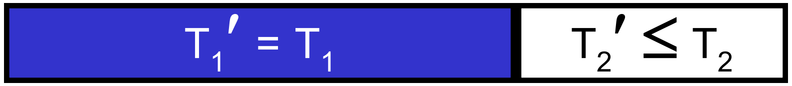
Old program (unenhanced)



T_1 T_2

Old time: $T = T_1 + T_2$

New program (enhanced)



$T_1' = T_1$ $T_2' \leq T_2$

New time: $T' = T_1' + T_2'$

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

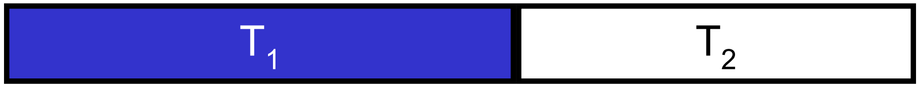
T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

T_2' = time after the enhancement.

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

$$\text{Speedup}(P) = T / T'$$

$$\leq 1/(T_1 + (1 - T_1)/p)$$

$$\leq 1/T_1$$

- 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%)

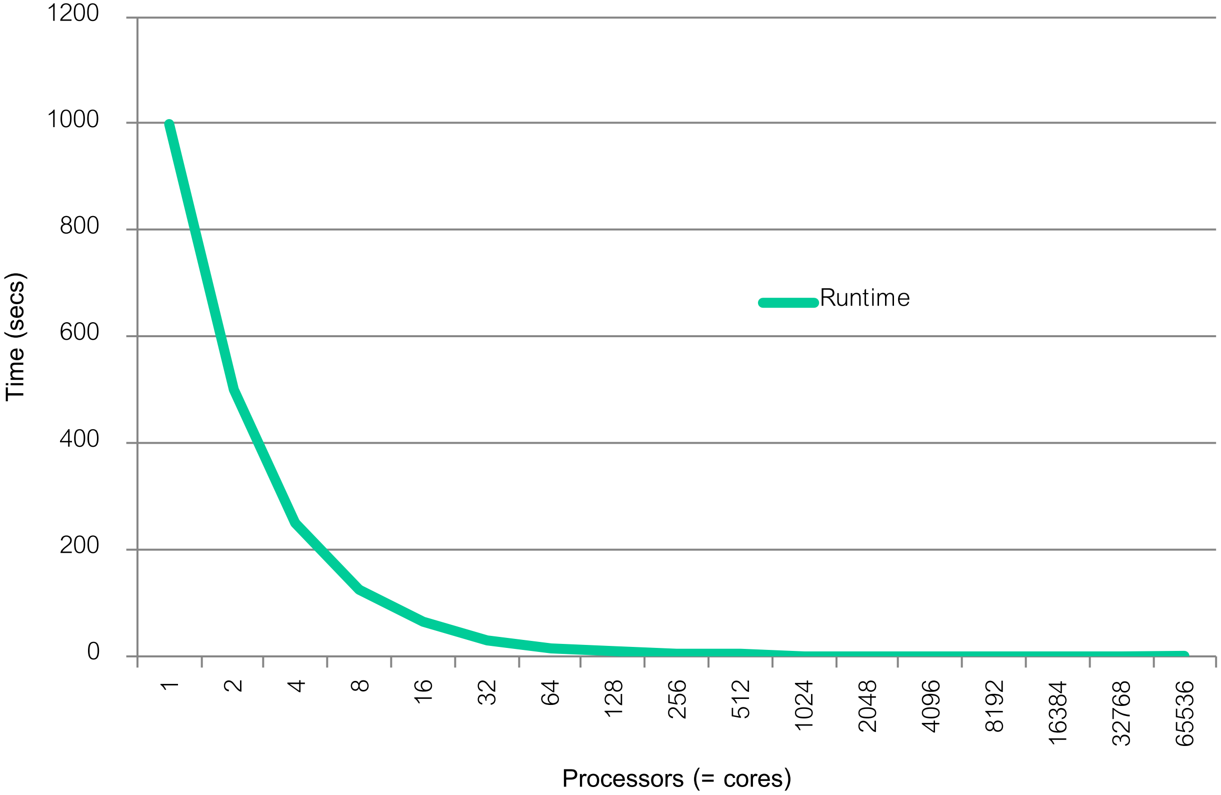
- What are the range of values for E? 0 to 1

- What is the efficiency of calculating the sum of n array elements on n processes?

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

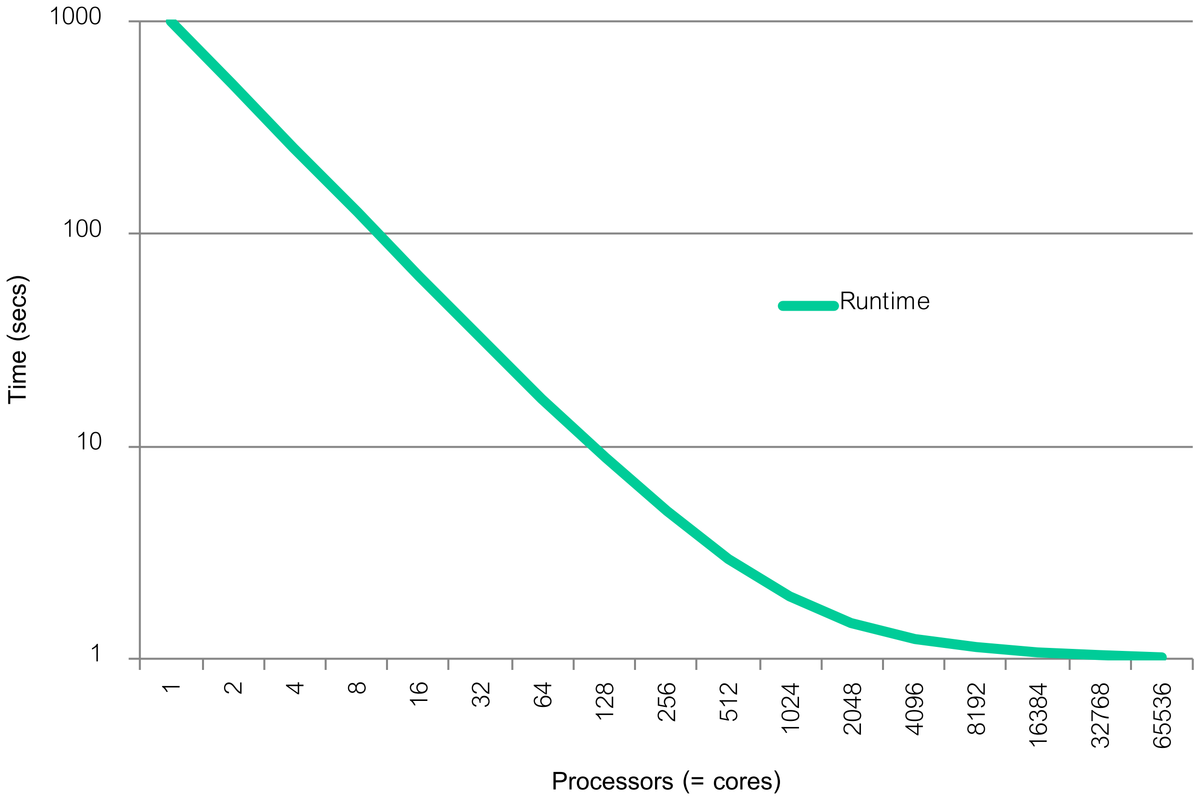
  $E = \Theta(1 / \log n)$

Reporting running time



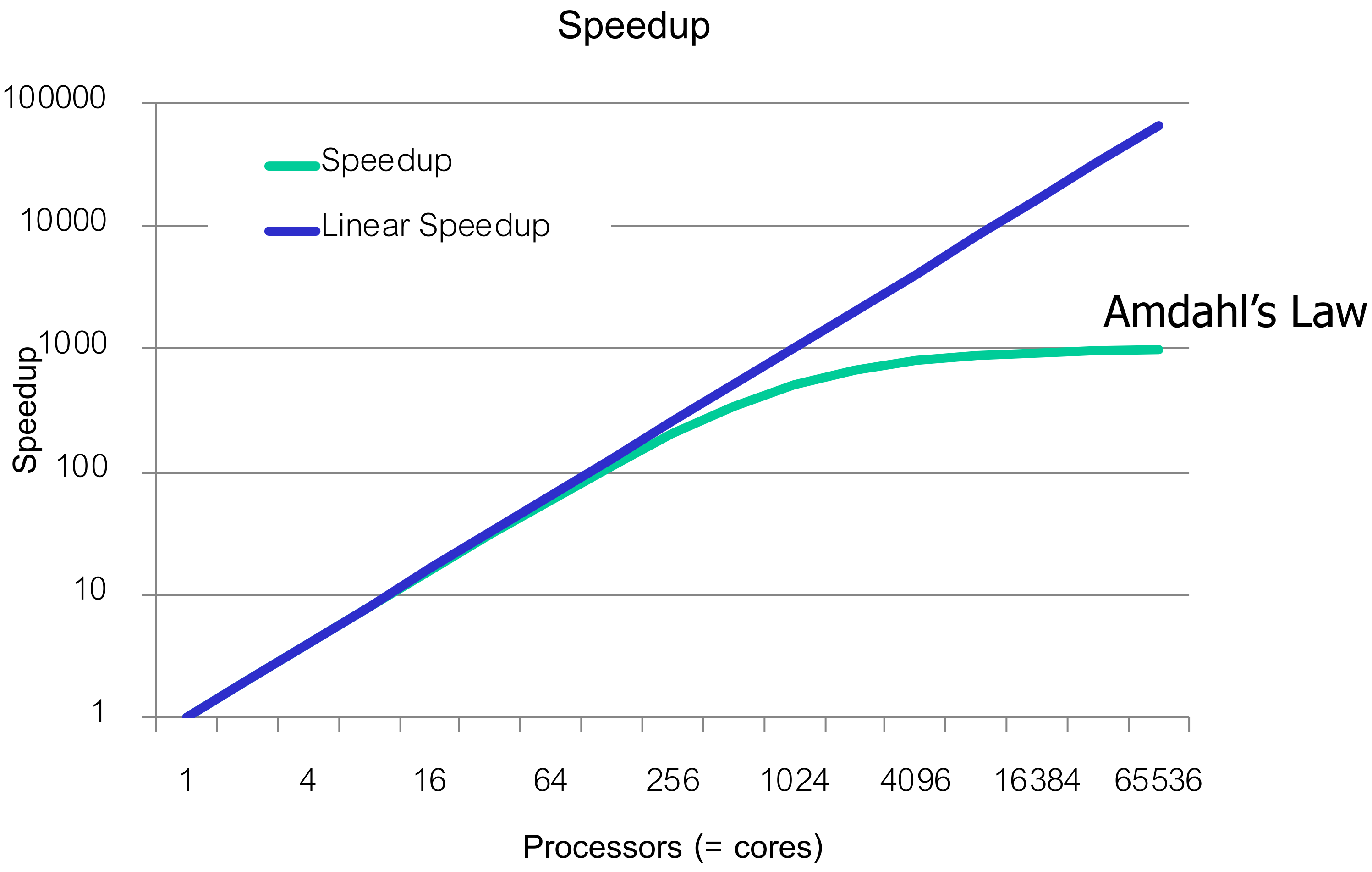
Hard to see performance gains from parallelism after 32 processors!

Reporting running time

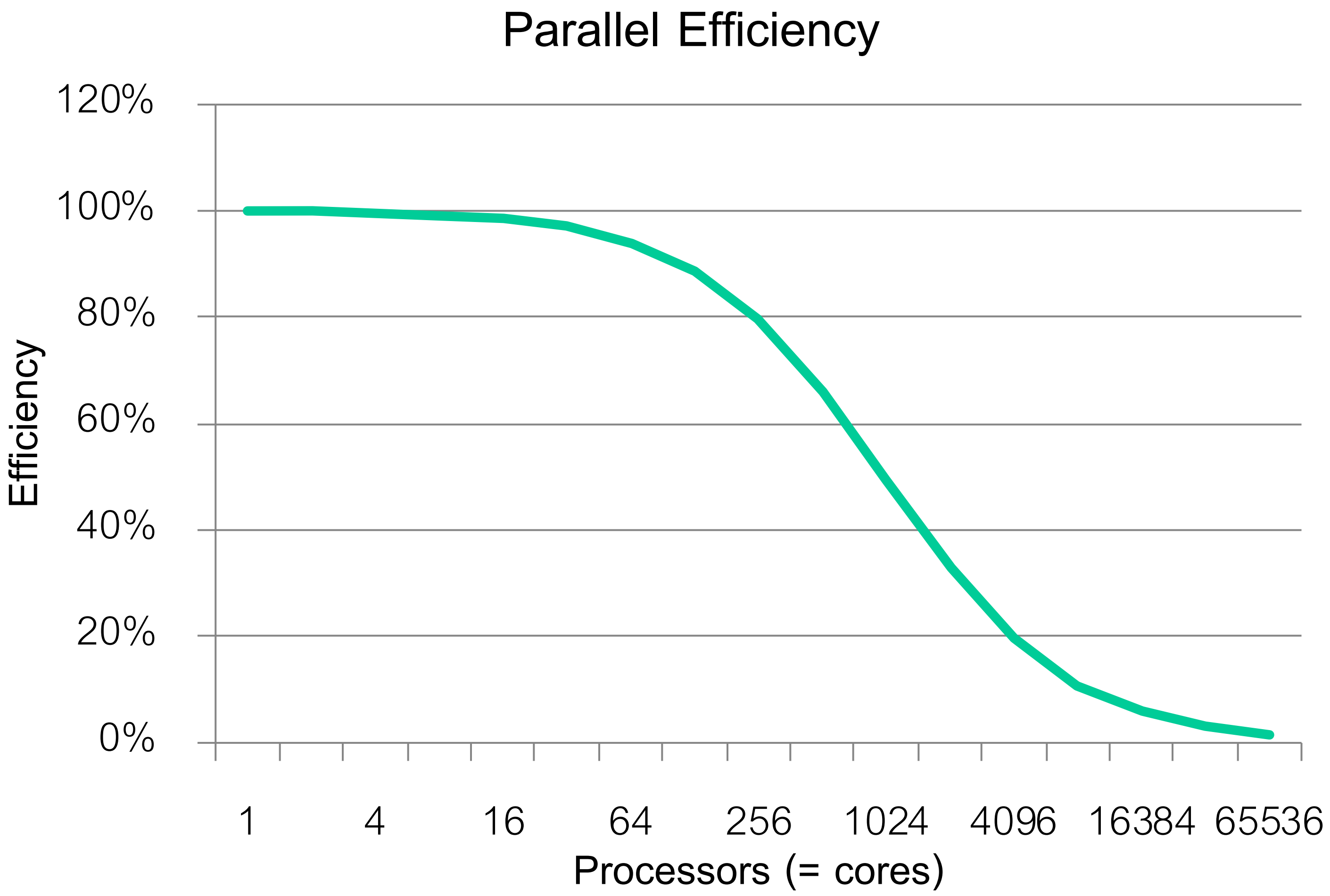


Lets take the y axis (running time) to log scale: A bit better!

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!

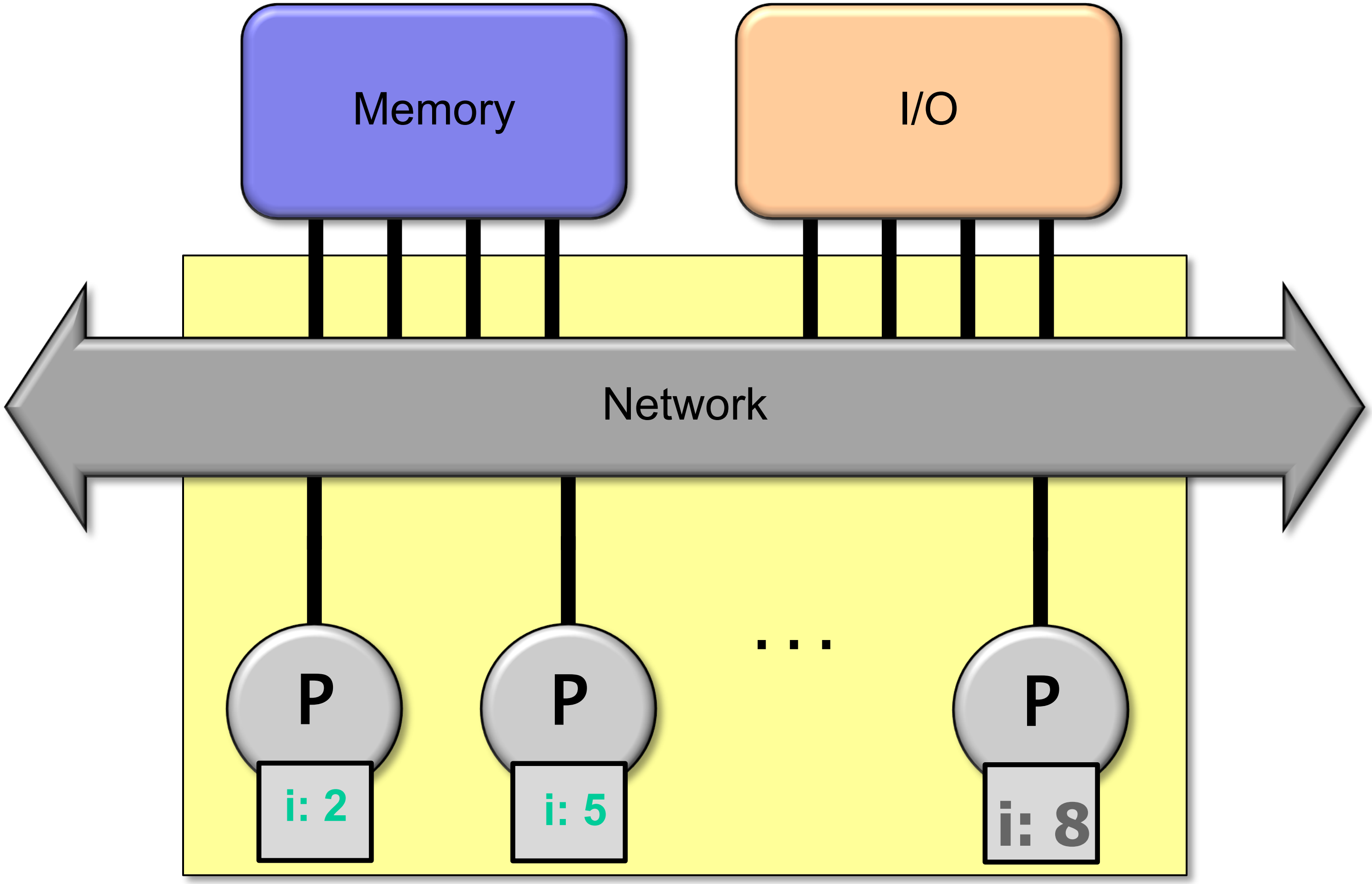
Next up ...

- 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

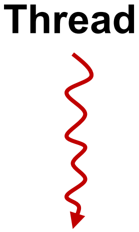
- Programming model is made up of the languages and libraries that create an abstract view of the machine: Pthreads!

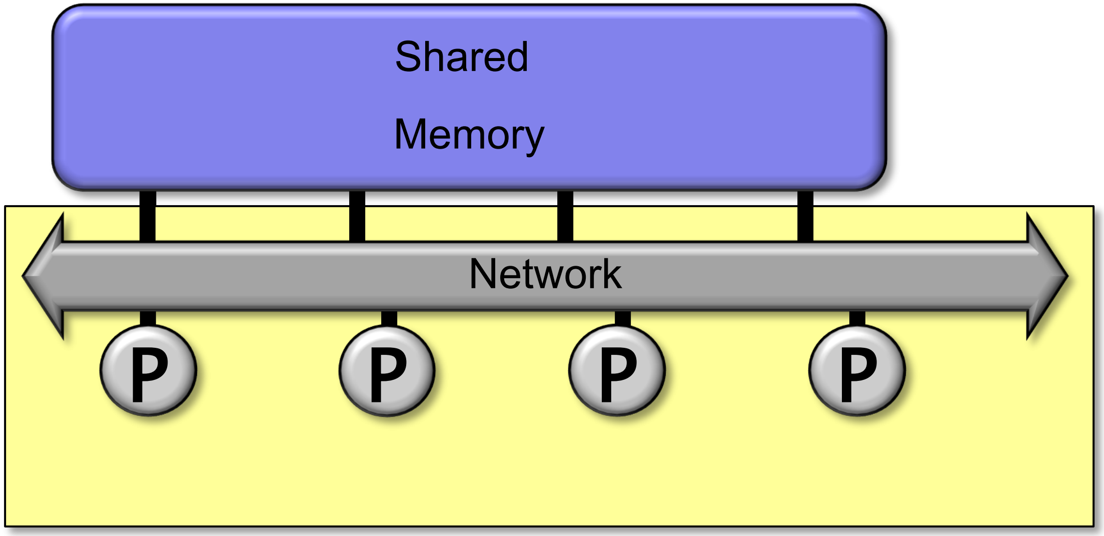
The programming model enables us to identify

- 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

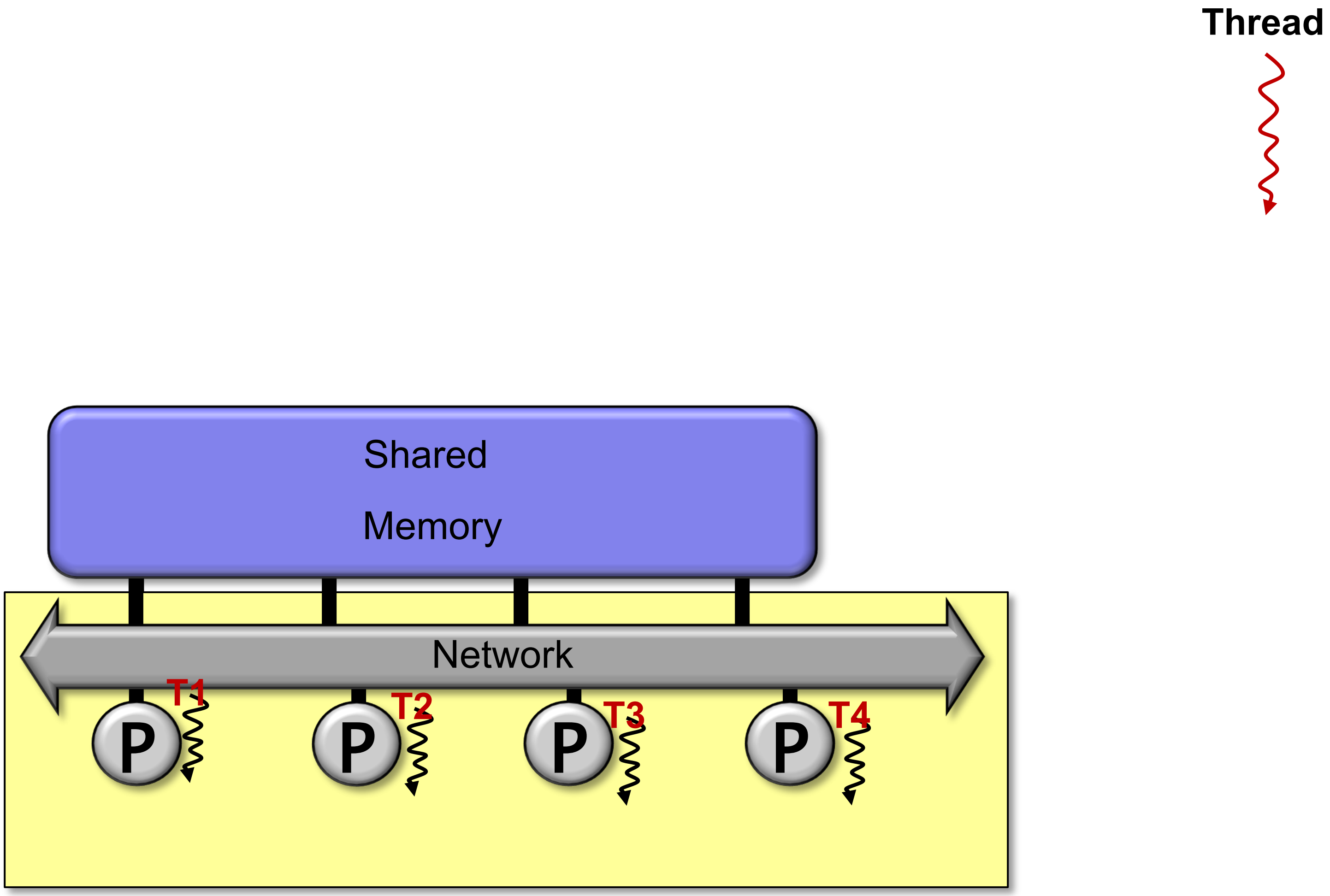
Program is a collection of threads of control, can be created mid-execution.





Programming Model: Shared Memory

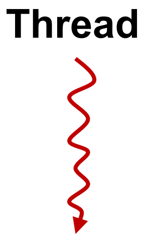
Program is a collection of threads of control, can be created mid-execution.

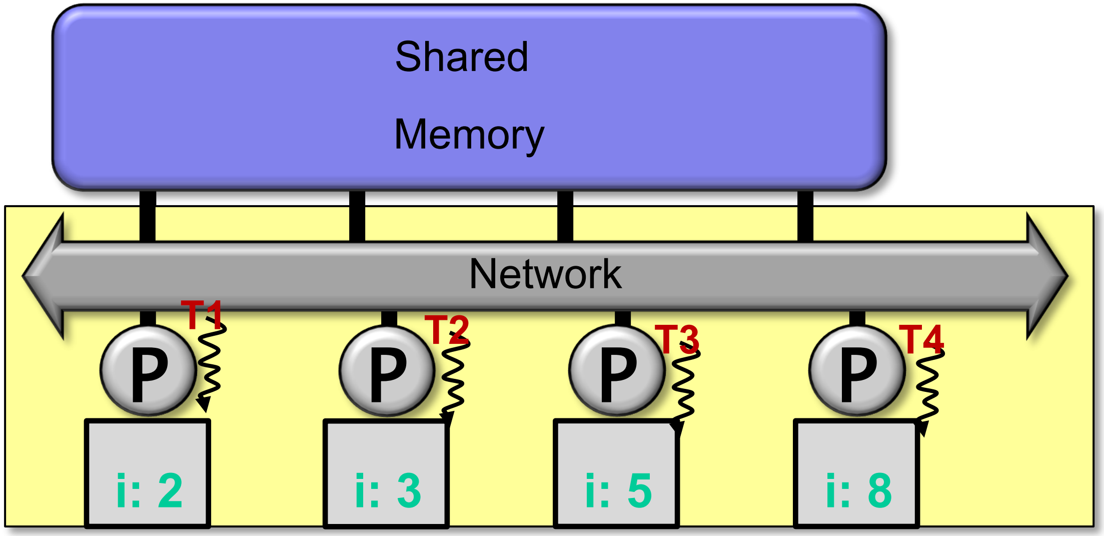


Programming Model: Shared Memory

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.





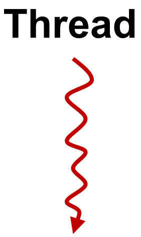
Programming Model: Shared Memory

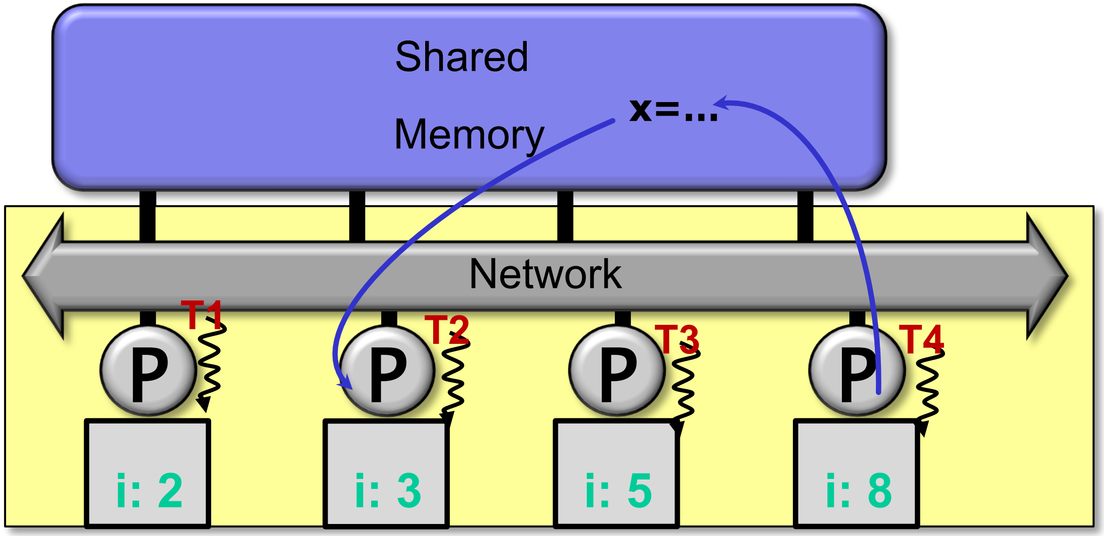
Program is a collection of threads of control, can be created mid-execution.

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

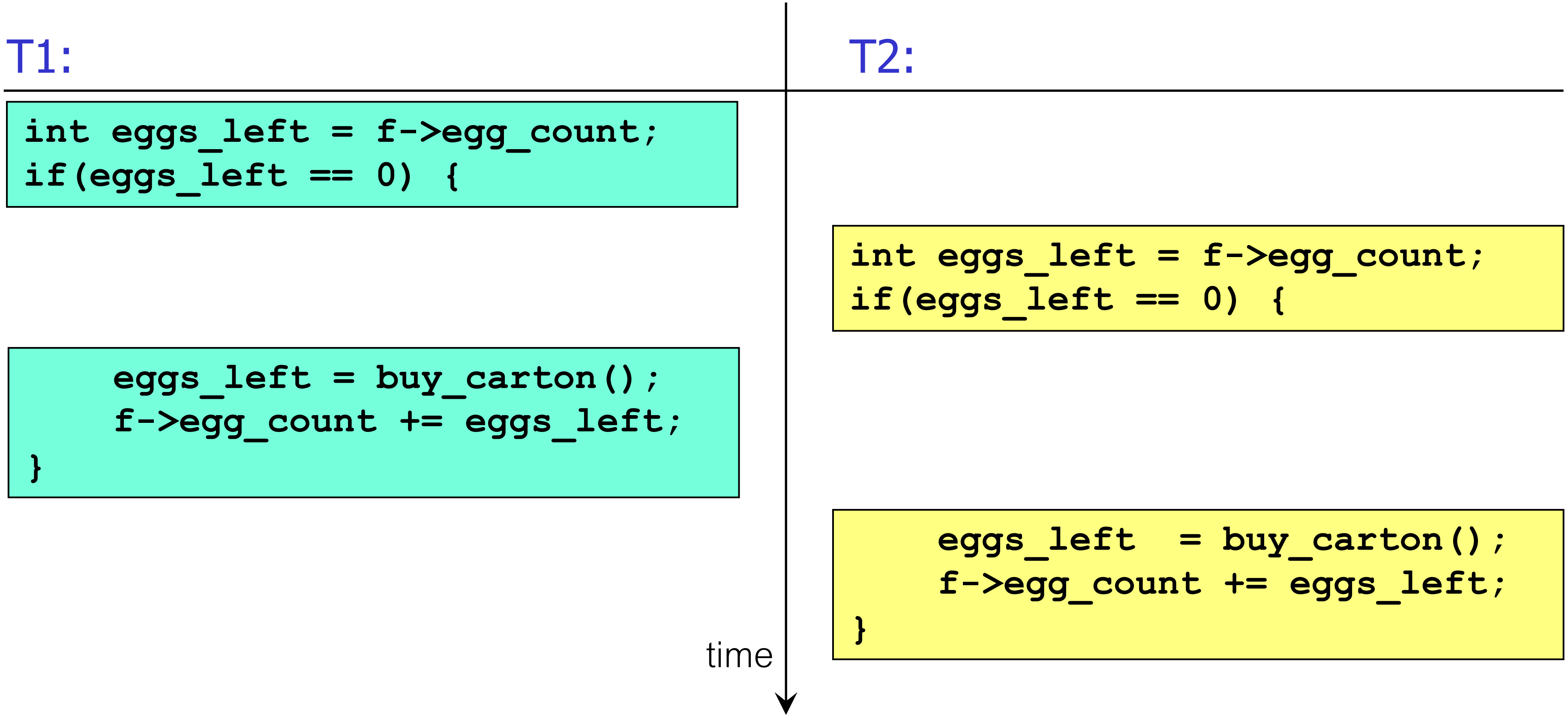
```
EggRun(fridge *f) {
    int eggs_left = f->egg_count;
    if(eggs_left == 0) {
        eggs_left = buy_carton();
        f->egg_count += eggs_left;
    }
}
```

```
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:

T2:

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;
}
```

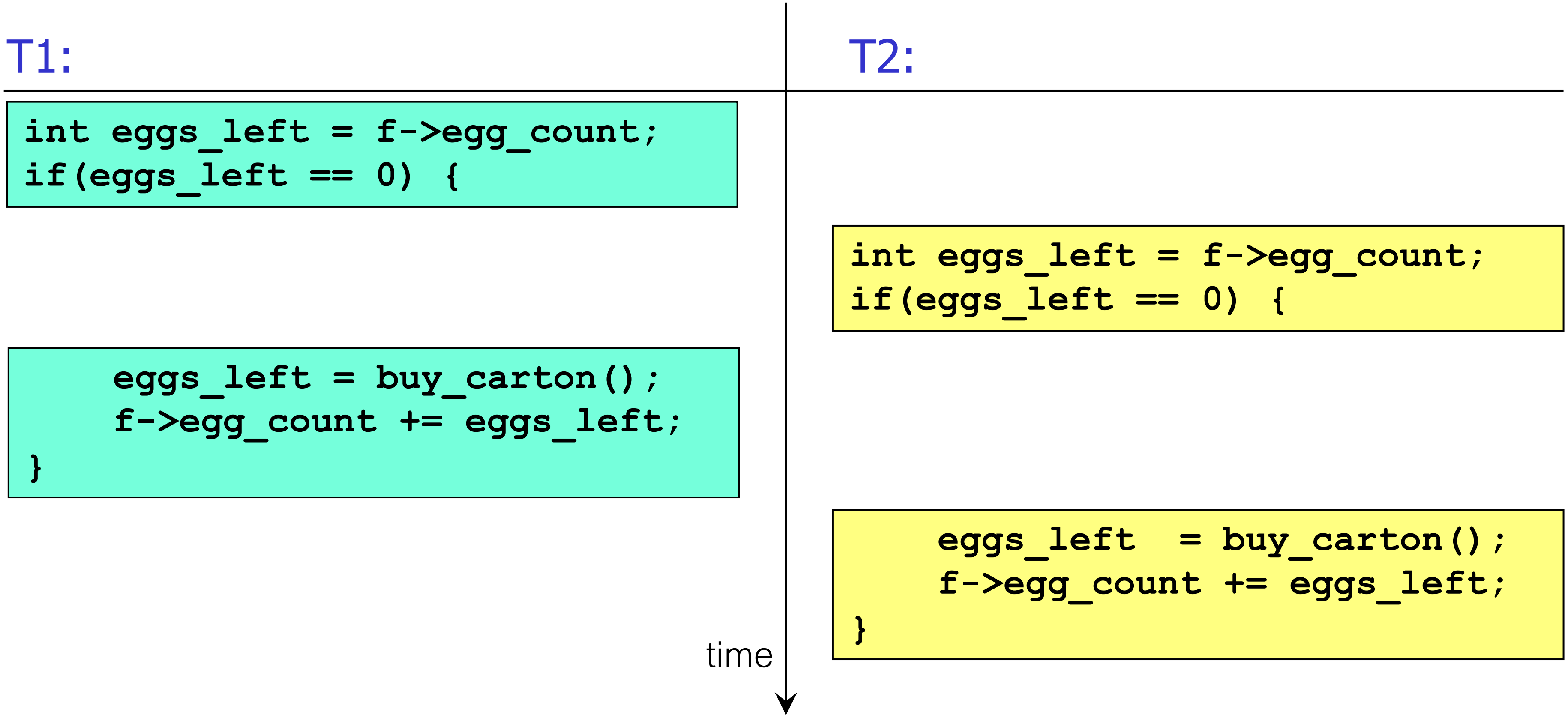
T2:

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

time

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) {
```

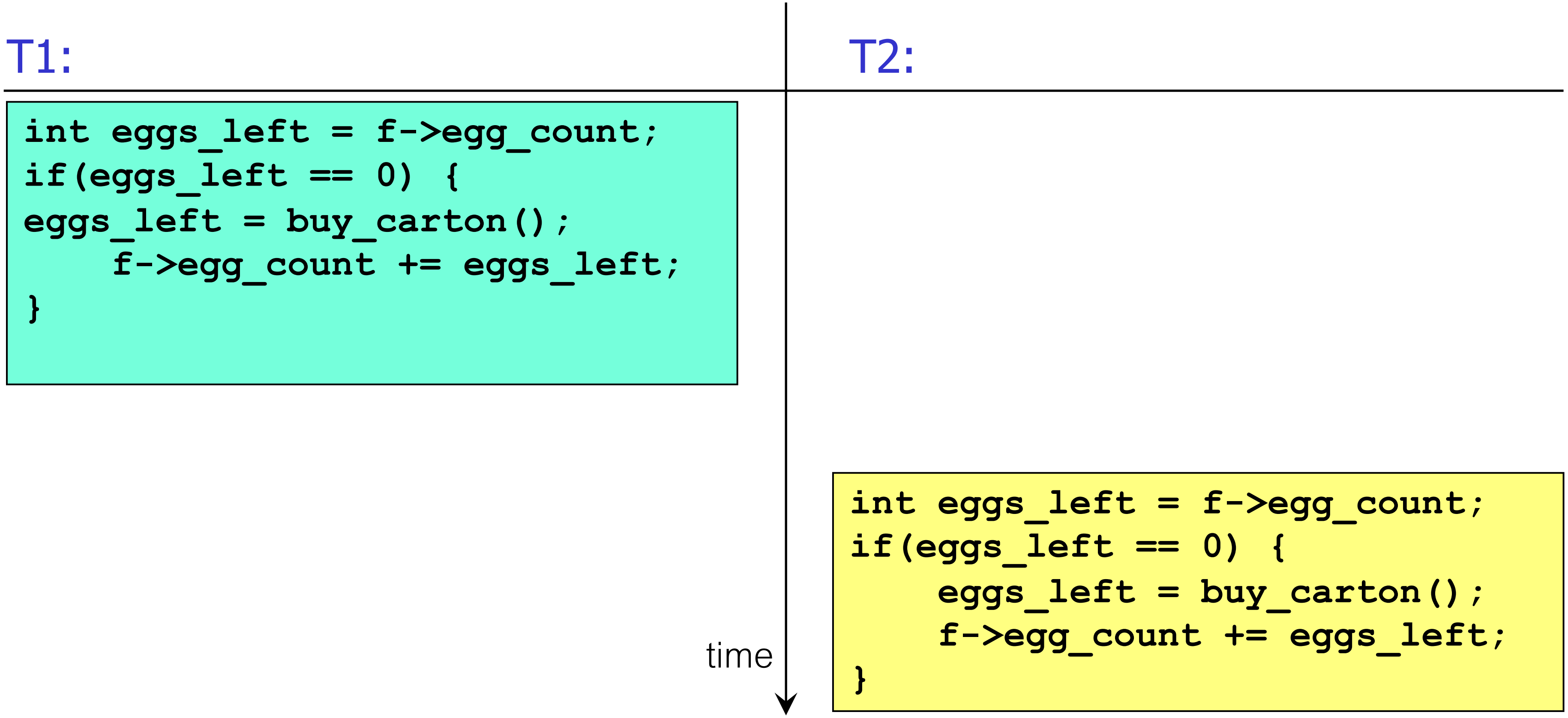
```
    eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

```
    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:

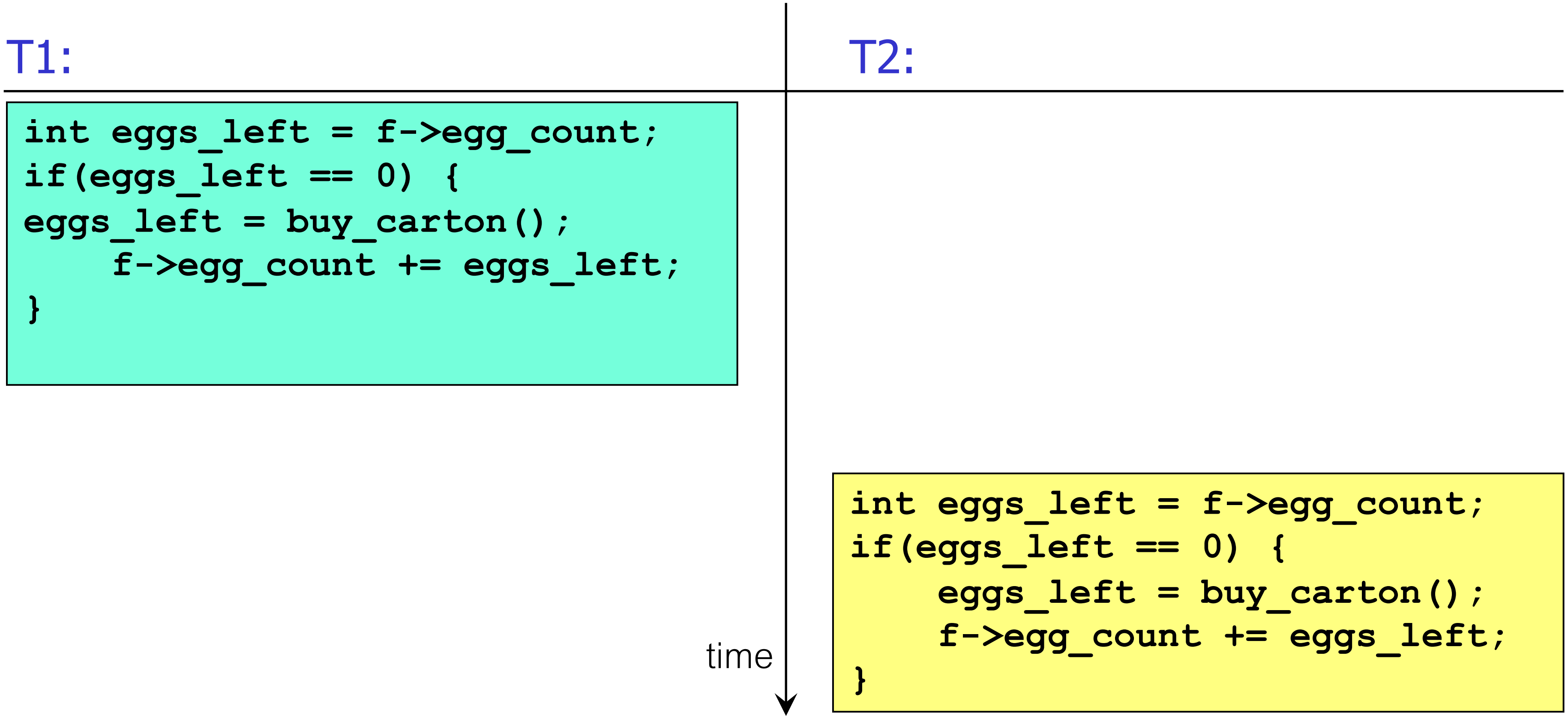
```
int eggs_left = f->egg_count;
if(eggs_left == 0) {
eggs_left = buy_carton();
    f->egg_count += eggs_left;
}
```

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:



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;
}
```

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