CS61A Lecture 42

Amir Kamil

UC Berkeley

April 29, 2013
Announcements

☐ HW13 due Wednesday

☐ Scheme project due tonight!!!

☐ Scheme contest deadline extended to Friday
MapReduce Execution Model

Python Example of a MapReduce Application
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs.
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs
• Read from *standard input* and write to *standard output*!
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

- Read from *standard input* and write to *standard output*!

Mapper
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

- Read from *standard input* and write to *standard output*!

**Mapper**

```python
def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)
```
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

• Read from *standard input* and write to *standard output*!

**Mapper**

```python
#!/usr/bin/env python3

import sys
from ucb import main
from mapreduce import emit

def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)
```

The *mapper* and *reducer* are both self-contained Python programs

- Read from *standard input* and write to *standard output*!

```python
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit

def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)
```

Tell Unix: this is Python
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

• Read from *standard input* and write to *standard output*!

**Mapper**

```
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit

def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)
```

Tell Unix: this is Python

The `emit` function outputs a key and value as a line of text to standard output
The \textit{mapper} and \textit{reducer} are both self-contained Python programs
\begin{itemize}
\item Read from \textit{standard input} and write to \textit{standard output}!
\end{itemize}

\textbf{Mapper}

\begin{verbatim}
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit

def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)

for line in sys.stdin:
    emit_vowels(line)
\end{verbatim}

\textit{Tell Unix: this is Python}

\textit{The emit function outputs a key and value as a line of text to standard output}
The \textit{mapper} and \textit{reducer} are both self-contained Python programs

- Read from \textit{standard input} and write to \textit{standard output}!

```python
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit

def emit_vowels(line):
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)

for line in sys.stdin:
    emit_vowels(line)
```

\textbf{Mapper inputs are lines of text provided to standard input}
Python Example of a MapReduce Application

The mapper and reducer are both self-contained Python programs
• Read from standard input and write to standard output!

Reducer
The *mapper* and *reducer* are both self-contained Python programs

* Read from *standard input* and write to *standard output*!

```python
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit, group_values_by_key
```

Reducer
Python Example of a MapReduce Application

The *mapper* and *reducer* are both self-contained Python programs

- Read from *standard input* and write to *standard output*!

Reducer

```
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit, group_values_by_key
```

Takes and returns iterators
The *mapper* and *reducer* are both self-contained Python programs.

- Read from *standard input* and write to *standard output*!

### Reducer

```python
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit, group_values_by_key
```

- **Input:** lines of text representing key-value pairs, grouped by key
- **Output:** Iterator over (key, value_iterator) pairs that give all values for each key
Python Example of a MapReduce Application

The mapper and reducer are both self-contained Python programs

- Read from standard input and write to standard output!

Reducer

```python
#!/usr/bin/env python3
import sys
from ucb import main
from mapreduce import emit, group_values_by_key

for key, value_iterator in group_values_by_key(sys.stdin):
    emit(key, sum(value_iterator))
```

Takes and returns iterators

Input: lines of text representing key-value pairs, grouped by key

Output: Iterator over (key, value_iterator) pairs that give all values for each key
Parallel Computation Patterns

Not all problems can be solved efficiently using functional programming
Parallel Computation Patterns

Not all problems can be solved efficiently using functional programming

The Berkeley View project has identified 13 common computational patterns in engineering and science:
Parallel Computation Patterns

Not all problems can be solved efficiently using functional programming

The Berkeley View project has identified 13 common computational patterns in engineering and science:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine
Not all problems can be solved efficiently using functional programming.

The Berkeley View project has identified 13 common computational patterns in engineering and science:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

MapReduce is only one of these patterns.

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine
Parallel Computation Patterns

Not all problems can be solved efficiently using functional programming

The Berkeley View project has identified 13 common computational patterns in engineering and science:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

MapReduce is only one of these patterns

The rest require shared mutable state

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine
Parallelism in Python
Parallelism in Python

Python provides two mechanisms for parallelism:
Parallelism in Python

Python provides two mechanisms for parallelism:

*Threads* execute in the same interpreter, sharing all data
Parallelism in Python

Python provides two mechanisms for parallelism:

*Threads* execute in the same interpreter, sharing all data

- However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
Parallelism in Python

Python provides two mechanisms for parallelism:

*Threads* execute in the same interpreter, sharing all data

- However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
- Operations external to the interpreter, such as file and network I/O, may execute concurrently
Parallelism in Python

Python provides two mechanisms for parallelism:

* **Threads** execute in the same interpreter, sharing all data
  * However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
  * Operations external to the interpreter, such as file and network I/O, may execute concurrently

* **Processes** execute in separate interpreters, generally not sharing data
Parallelism in Python

Python provides two mechanisms for parallelism:

* **Threads** execute in the same interpreter, sharing all data
  * However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
  * Operations external to the interpreter, such as file and network I/O, may execute concurrently

* **Processes** execute in separate interpreters, generally not sharing data
  * Shared state can be communicated explicitly between processes
Parallelism in Python

Python provides two mechanisms for parallelism:

*Threads* execute in the same interpreter, sharing all data
- However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
- Operations external to the interpreter, such as file and network I/O, may execute concurrently

*Processes* execute in separate interpreters, generally not sharing data
- Shared state can be communicated explicitly between processes
- Since processes run in separate interpreters, they can be executed in parallel as the underlying hardware and software allow
Parallelism in Python

Python provides two mechanisms for parallelism:

* **Threads** execute in the same interpreter, sharing all data
  - However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
  - Operations external to the interpreter, such as file and network I/O, may execute concurrently

* **Processes** execute in separate interpreters, generally not sharing data
  - Shared state can be communicated explicitly between processes
  - Since processes run in separate interpreters, they can be executed in parallel as the underlying hardware and software allow

The concepts of threads and processes exist in other systems as well
Threads
The `threading` module contains classes that enable threads to be created and synchronized.
Threads

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:
The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread
```
The `threading` module contains classes that enable threads to be created and synchronized

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
```

```python
```
Threads

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
```

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
```

Function that the new thread should run
The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
```

Function that the new thread should run.

Arguments to that function.
The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
```

Function that the new thread should run

Arguments to that function
Threads

The **threading** module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
```

- **Function that the new thread should run**
- **Arguments to that function**
- **Start the other thread**
The **threading** module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()

thread_say_hello()
```

- **Function that the new thread should run**
- **Start the other thread**
- **Arguments to that function**
Threads

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    Function that the new thread should run

Arguments to that function
```

Start the other thread
Threads

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello()
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello()
    print('hello from', current_thread().name)
```

Function that the new thread should run

Start the other thread

Arguments to that function
The *threading* module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    print('hello from', current_thread().name)

>>> thread_hello()
```

Function that the new thread should run

Start the other thread

Arguments to that function
Threads

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    print('hello from', current_thread().name)

>>> thread_hello()
hello from Thread-1
```
The **threading** module contains classes that enable threads to be created and synchronized.

Here is a “hello world” example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    print('hello from', current_thread().name)

>>> thread_hello()
hello from Thread-1
hello from MainThread
```
Threads

The `threading` module contains classes that enable threads to be created and synchronized.

Here is a "hello world" example with two threads:

```python
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    print('hello from', current_thread().name)

>>> thread_hello()
hello from Thread-1
hello from MainThread
```

Function that the new thread should run

Start the other thread

Arguments to that function

Print output is not synchronized, so can appear in any order
The `multiprocessing` module contains classes that enable processes to be created and synchronized.

Here is a “hello world” example with two processes:

```python
from multiprocessing import Process, current_process

def process_hello():
    other = Process(target=process_say_hello, args=())
    other.start()
    process_say_hello()

def process_say_hello():
    print('hello from', current_process().name)

>>> process_hello()
hello from MainProcess
>>> hello from Process-1
```

Function that the new process should run

Start the other process

Arguments to that function

Print output is not synchronized, so can appear in any order
The Problem with Shared State
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.

Here is an example with two threads that concurrently update a counter:
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread
```

The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
```

The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread
counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()  # Wait until other thread completes
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()  # Wait until other thread completes
print('count is now', counter[0])
```
The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs.

Here is an example with two threads that concurrently update a counter:

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
iincrement()
other.join()  # Wait until other thread completes

print('count is now', counter[0])
```

What is the value of `counter[0]` at the end?
The Problem with Shared State

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

What is the value of `counter[0]` at the end?
The Problem with Shared State

```python
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

What is the value of `counter[0]` at the end?

Only the most basic operations in CPython are *atomic*, meaning that they have the effect of occurring instantaneously.
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])

What is the value of counter[0] at the end?

Only the most basic operations in CPython are *atomic*, meaning that they have the effect of occurring instantaneously

The counter increment is three basic operations: read the old value, add 1 to it, write the new value
The Problem with Shared State
The Problem with Shared State

We can see what happens if a switch occurs at the wrong time by trying to force one in CPython:
We can see what happens if a switch occurs at the wrong time by trying to force one in CPython:

```python
from threading import Thread

counter = [0]

def increment():
    count = counter[0]
    counter[0] = count + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```
The Problem with Shared State

We can see what happens if a switch occurs at the wrong time by trying to force one in CPython:

```python
from threading import Thread
from time import sleep

counter = [0]

def increment():
    count = counter[0]

    counter[0] = count + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```
The Problem with Shared State

We can see what happens if a switch occurs at the wrong time by trying to force one in CPython:

```python
from threading import Thread
from time import sleep

counter = [0]

def increment():
    count = counter[0]
    sleep(0)
    counter[0] = count + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```
The Problem with Shared State

We can see what happens if a switch occurs at the wrong time by trying to force one in CPython:

```python
from threading import Thread
from time import sleep

counter = [0]

def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```
def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1
The Problem with Shared State

def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:
The Problem with Shared State

def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0                     Thread 1
The Problem with Shared State

def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0
read `counter[0]`: 0

Thread 1
The Problem with Shared State

def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1

Given a switch at the sleep call, here is a possible sequence of operations on each thread:

Thread 0
read counter[0]: 0

Thread 1
read counter[0]: 0
The Problem with Shared State

def increment():
    count = counter[0]
    sleep(0)
    counter[0] = count + 1

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0
read counter[0]: 0
calculate 0 + 1: 1

Thread 1
read counter[0]: 0
The Problem with Shared State

def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0
read counter[0]: 0
calculate 0 + 1: 1
write 1 -> counter[0]

Thread 1
read counter[0]: 0
The Problem with Shared State

```python
def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1
```

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0
read counter[0]: 0
calculate 0 + 1: 1
write 1 -> counter[0]

Thread 1
read counter[0]: 0
calculate 0 + 1: 1
The Problem with Shared State

```python
def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1
```

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0
read `counter[0]`: 0

calculate `0 + 1`: 1
write `1 -> counter[0]`

Thread 1
read `counter[0]`: 0

calculate `0 + 1`: 1
write `1 -> counter[0]`
The Problem with Shared State

```python
def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1
```

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

Thread 0
- read `counter[0]`: 0
- calculate `0 + 1`: 1
- write `1 -> counter[0]`

Thread 1
- read `counter[0]`: 0
- calculate `0 + 1`: 1
- write `1 -> counter[0]`

The counter ends up with a value of 1, even though it was incremented twice!
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a *race condition*.
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a *race condition*

Race conditions are difficult to debug, since they may only occur very rarely
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a race condition.

Race conditions are difficult to debug, since they may only occur very rarely.

Access to shared data in the presence of mutation must be synchronized in order to prevent access by other threads while a thread is mutating the data.
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a *race condition*.

Race conditions are difficult to debug, since they may only occur very rarely.

Access to shared data in the presence of mutation must be *synchronized* in order to prevent access by other threads while a thread is mutating the data.

Managing shared state is a key challenge in parallel computing.
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a *race condition*.

Race conditions are difficult to debug, since they may only occur very rarely.

Access to shared data in the presence of mutation must be *synchronized* in order to prevent access by other threads while a thread is mutating the data.

Managing shared state is a key challenge in parallel computing:

- Under-synchronization doesn’t protect against race conditions and other parallel bugs.
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a race condition

Race conditions are difficult to debug, since they may only occur very rarely

Access to shared data in the presence of mutation must be synchronized in order to prevent access by other threads while a thread is mutating the data

Managing shared state is a key challenge in parallel computing

- Under-synchronization doesn’t protect against race conditions and other parallel bugs
- Over-synchronization prevents non-conflicting accesses from occurring in parallel, reducing a program’s efficiency
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a race condition.

Race conditions are difficult to debug, since they may only occur very rarely.

Access to shared data in the presence of mutation must be synchronized in order to prevent access by other threads while a thread is mutating the data.

Managing shared state is a key challenge in parallel computing:

• Under-synchronization doesn’t protect against race conditions and other parallel bugs.
• Over-synchronization prevents non-conflicting accesses from occurring in parallel, reducing a program’s efficiency.
• Incorrect synchronization may result in deadlock, where different threads indefinitely wait for each other in a circular dependency.
Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a *race condition*. 

Race conditions are difficult to debug, since they may only occur very rarely.

Access to shared data in the presence of mutation must be *synchronized* in order to prevent access by other threads while a thread is mutating the data.

Managing shared state is a key challenge in parallel computing:

- Under-synchronization doesn’t protect against race conditions and other parallel bugs.
- Over-synchronization prevents non-conflicting accesses from occurring in parallel, reducing a program’s efficiency.
- Incorrect synchronization may result in *deadlock*, where different threads indefinitely wait for each other in a circular dependency.

We will see some basic tools for managing shared state.
Synchronized Data Structures
Synchronized Data Structures

Some data structures guarantee synchronization, so that their operations are atomic.
Synchronized Data Structures

Some data structures guarantee synchronization, so that their operations are atomic

```python
from queue import Queue

queue = Queue()

def increment():
    count = queue.get()
    sleep(0)
    queue.put(count + 1)

other = Thread(target=increment, args=())
other.start()

queue.put(0)
increment()
other.join()

print('count is now', queue.get())
```
Synchronized Data Structures

Some data structures guarantee synchronization, so that their operations are atomic

```python
from queue import Queue

queue = Queue()

def increment():
    count = queue.get()
sleep(0)
    queue.put(count + 1)

other = Thread(target=increment, args=())
other.start()
queue.put(0)
increment()
other.join()
print('count is now', queue.get())
```
Synchronized Data Structures

Some data structures guarantee synchronization, so that their operations are atomic.

```python
from queue import Queue

queue = Queue()

def increment():
    count = queue.get()
    sleep(0)
    queue.put(count + 1)

other = Thread(target=increment, args=())
other.start()
queue.put(0)
increment()
other.join()
print('count is now', queue.get())
```
Some data structures guarantee synchronization, so that their operations are atomic

```python
from queue import Queue  # Synchronized FIFO queue

queue = Queue()

def increment():
    count = queue.get()  # Waits until an item is available
    sleep(0)
    queue.put(count + 1)

other = Thread(target=increment, args=())
other.start()

queue.put(0)  # Add initial value of 0
increment()
other.join()

print('count is now', queue.get())
```