Motivation

• Traditional program development is typically focused on a single *thread* of processing.
  – That is, every operation performed by a typical program occurs in a set, one-by-one (sequential) order.
  – This is quite often used to enable a program’s functionality – an assignment of one value to a variable can be used in the next code line.
Motivation

• Accordingly, traditional computer hardware development has been focused on the speedup of a single thread of processing.
  – This would enable a typical program to get the maximum performance boost possible.
Motivation

• However, in recent years, there has been a substantial shift in the sort of improvements that are being made in hardware.
  – It has become increasingly harder to increase the speed with which individual operations execute on CPUs.
  – Instead, we now have “dual core” and “quad core” CPUs.
Motivation

- For much of computer history, the majority of processors had a single core.
  - One core is capable of processing a single thread of execution.
The Thread

• Think of a program as a fully orchestrated musical piece.
  – If a program were defined in this manner, each thread would be the sheet music for one instrument or vocalist.

• Alternatively, think of a program as a movie script.
  – Each thread would be the set of lines and actions to be taken by one actor.
The Thread

• In the case of music, there are plenty of songs done solo – by a single musician.
  – However, there are a lot of arrangements composed of multiple parts, which are designed to work together cooperatively.
Cores

• So, a multi-core processor is capable of simultaneously running multiple threads.

• The problem?
  – If a program is written as a single thread and run on a multi-core system, those additional cores will not be able to enhance the program.
Motivation

• These days, processors are gaining more cores, as opposed to more single-threaded speed.
  – Thus, it’s quite important to learn how to utilize these extra cores in order to make future programs more time-efficient.
Concurrency

- *Concurrency* refers to the idea and the study of systems which utilize multiple threads.
  - In some cases, there may even be multiple programs involved – a computer may have limited resources which must be shared.
Concurrency

• The part of concurrency which we will examine is the coding and analysis of methods within a single program, across multiple threads.
Concurrency

• First off, how can we set up more than one thread within a program?
  – The exact mechanisms sometimes vary by operating system.
  – However, we’re programming in C++!!
Thread Objects

- The relatively new C++11 standard provides some common threading objects similar to those in other languages.
  - The base thread object takes in either a function pointer or a functor as the core “main” method for the created thread.
Thread Objects

• As Java lacks function pointers, its Runnable interface exists to allow appropriate “functors.”
  – A single method called run() is specified in said interface, which is called when a thread is created.
  – In C++, we’d override operator() instead.
Thread Objects

- The Thread class of Java itself is designed to actually instantiate and run threads.
  - C++11 also has its own thread class.
  - Unlike Java’s implementation, a new C++11 thread *instantly* begins execution when object construction completes.
Thread Objects

- Sometimes, one thread’s operations may be a prerequisite for work in another thread, or the main thread.
  - For cases where one thread must wait on another, there is typically a join() operation defined.
  - If we wish to wait for thread t1 to complete, we would then call t1.join().
Thread Methods

- Languages typically have a `sleep()` method that can pause a thread for a set amount of time.
  - In C++, `sleep_for()`.
  - There is often a `yield()` method that pauses a thread to pause momentarily while allowing it to resume at the earliest convenience of the host OS.
  - This allows a thread to relinquish control and prioritize other threads’ operations.
Uses Of Concurrency

• There are many potential uses of concurrency within applications:
  – Decouple operations requested from a GUI from the GUI’s thread.
  – This would allow the application to remain responsive to the user, even while the application evaluates uncompleted requests.
  – Great example: printing!
Uses Of Concurrency

• There are many potential uses of concurrency within applications:
  – Allow unrelated portions of code to evaluate simultaneously.
  – Sometimes parts of programs can be identified that do not rely upon each other.
  – In such a case, why should those parts have to wait upon each other?
  – Example: a data server replying to queries from different users.
Uses Of Concurrency

- There are many potential uses of concurrency within applications:
  - Acceleration of basic algorithms.
  - While this may not be the case for all algorithms, for many it is possible to find ways in which they may be parallelized.
    - That is, subdivided into multiple threads for their evaluation.
  - Example: What if, on the first split of merge sort, each half went to a separate thread?
Uses of Concurrency

• If we parallelized a merge sort by sorting the first two obtained halves within different threads, we might get something like the following...
Example: Parallel Sort

Thread1: sort(first half)
Thread2: sort(second half)

Thread1.join()
Thread2.join()

//Cannot be parallelized.
merge(first half, second half)
Uses of Concurrency

• In the parallelized merge sort example, note that what we really did was to identify two regions which could be evaluated independently.
  – This is the general technique used for parallelizing code.
  – Why? As it turns out, the more interaction that is required between two threads, the harder concurrency gets.