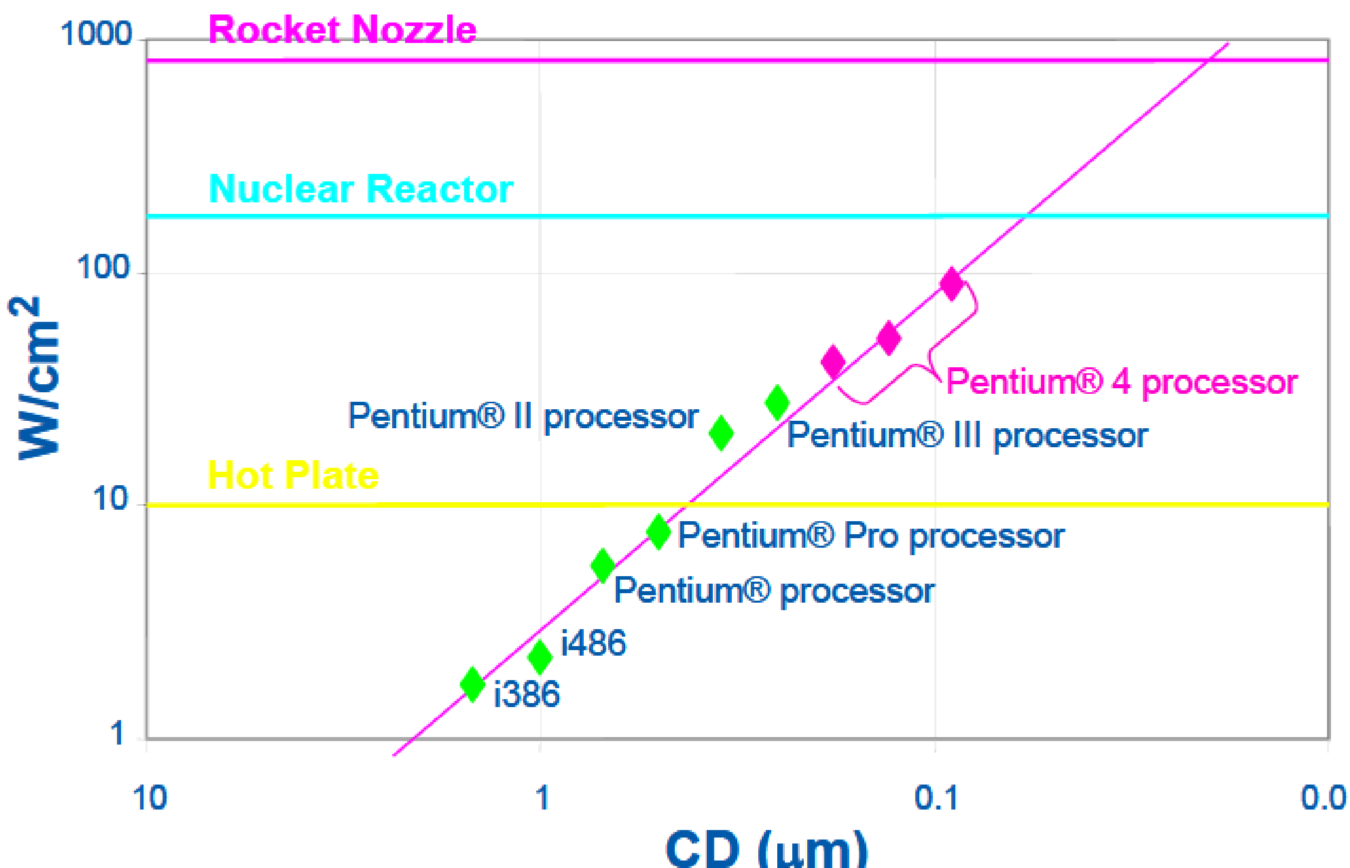
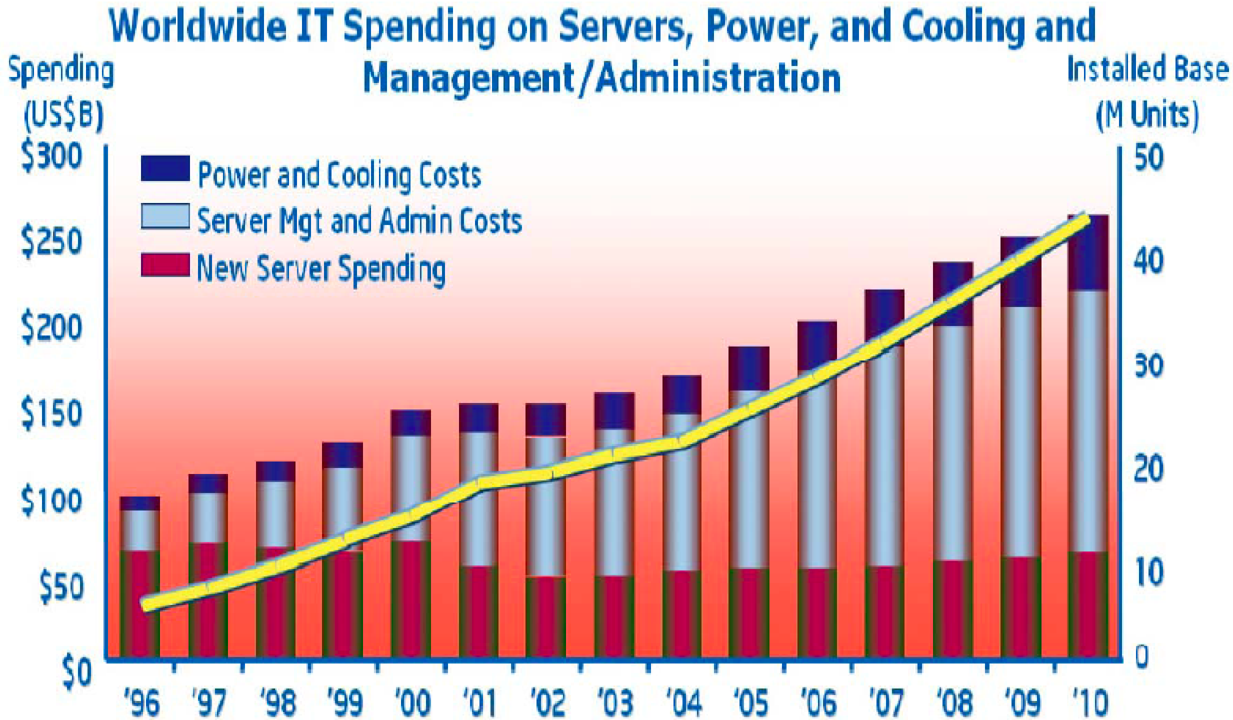
Multithreaded Programming in C++

Power Density vs. Critical Dimension



Source: G. Taylor, "Energy Efficient Circuit Design and the Future of Power Delivery" EPEPS'09



Rate of Server Management and Power/Cooling Cost Increase

Source: IDC

PThreads

Thanks to LLNL for their tutorial from which these slides are derived

http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html

What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads.
  - Standardization required for portable multi-threaded programming
  - For Unix, this interface specified by the IEEE POSIX 1003.1c standard (1995).
    - Implementations of this standard are called POSIX threads, or Pthreads.
    - Most hardware vendors now offer Pthreads in addition to their proprietary API's
    - Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library
  - Multiple drafts before standardization -- this leads to problems

Posix Threads - 3 kinds

- "Real" POSIX threads, based on the IEEE POSIX 1003.1c-1995 (also known as the ISO/IEC 9945-1:1996) standard, part of the ANSI/IEEE 1003.1, 1996 edition, standard.

  IEEE Std 1003.1, 2004 Edition.
- POSIX implementations are, not surprisingly, the emerging standard on Unix systems. POSIX threads are usually referred to as Pthreads.
- DCE threads are based on draft 4 (an early draft) of the POSIX threads standard (which was originally named 1003.4a, and became 1003.1c upon standardization).
- Unix International (UI) threads, also known as Solaris threads, are based on the Unix International threads standard (a close relative of the POSIX standard).

What are threads used for?

- Tasks that may be suitable for threading include tasks that
  - Block for potentially long waits (Tera MTA/HEP & I/O)
  - **Use many CPU cycles**
  - Must respond to asynchronous events
  - Are of lesser or greater importance than other tasks
  - **Are able to be performed in parallel with other tasks**
- Note that numerical computing is only part of what parallelism is used for

Three classes of Pthreads routines

- ***Thread management:*** creating, detaching, and joining threads, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
- ***Mutexes:*** Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- ***Condition variables:*** The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Creating threads

- **int pthread_create** (thread, attr, start_routine, arg)
- This routine creates a new thread and makes it executable. Typically, threads are first created from within main() inside a single process.
- the return value is true if a thread is successfully created, a negative error code otherwise.
- Question: After a thread has been created, how do you know when it will be scheduled to run by the operating system...especially on an SMP/ multicore machine? **You don't!**

Creating threads

- **int pthread_create** (thread, attr, start_routine, arg)
  - Once created, threads are peers, and may create other threads
  - The pthread_create subroutine returns the new thread ID via the *thread* argument. This ID should be checked to ensure that the thread was successfully created
  - The *attr* parameter is used to set thread attributes. Can be an object, or NULL for the default values
  - *start_routine* is the C routine that the thread will execute once it is created. A single argument may be passed to *start_routine* via *arg* as a void pointer.
  - The maximum number of threads that may be created by a process is implementation dependent -- creating too many causes performance problems

Terminating threads

1. The thread returns to its starting routine (the "main" routine for the initial thread)
2. The thread makes a call to the pthread_exit subroutine
3. The thread is *canceled* by another thread via the pthread_cancel routine
   - Same problems exist with data consistency as with, e.g., terminating Java threads: what if a constructor in the terminated thread is half completed? what if a destructor is not yet run?
4. The entire process is terminated because of a call to either the exec or exit APIs.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
  printf(”\n%d: Hello World!\n", threadid);
  pthread_exit(NULL);
}
int main (int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0; t < NUM_THREADS; t++){
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

pthread_exit(void *status)

- `pthread_exit()` routine is called *by a thread* after a thread has completed its work and is no longer required to exist
- If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute.
  - Otherwise, they will be automatically terminated when `main()` finishes
- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may *join* the calling thread
- Cleanup
  - `pthread_exit()` routine does not close files
  - Recommended to use `pthread_exit()` to exit from all threads...especially `main()`.

Passing arguments to a thread

- Thread startup is non-deterministic
- It is implementation dependent
- If we do not know when a thread will start, how do we pass data to the thread knowing it will have the right value at startup time?
  – Don't pass data as arguments that can be changed by another thread
  – In general, use a separate instance of a data structure for each thread.

Passing data to a thread (a simple integer)

```
int *taskids[NUM_THREADS];
for(t=0;t < NUM_THREADS;t++) {
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL,
                        PrintHello, (void *) taskids[t]);
   …
}
```

The key is that a unique location is passed

Living dangerously…

```
int rc, t;
for(t=0;t < NUM_THREADS;t++) {
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL,
                         PrintHello, (void *) &t);
   ...
}
```

Here the same address is passed to all threads -- the value read at that address depends on when the thread executes and reads it.

In general

- Unless you know something is read-only
  - Only good way to know what the value is when the thread starts is to have a separate copy of argument for each thread.
  - Complicated data structures may share data at a deeper level

Thread identifiers

- pthread_self ()
  - pthread_self() routine returns the unique, system assigned thread ID of the calling thread
- pthread_equal (thread1,thread2)
  - pthread_equal( ) routine compares two thread IDs.
    - 0 if different, non-zero if the same.
    - Note that for both of these routines, the thread identifier objects are ***opaque***
    - Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value.

- pthread_join (threadId, status)
- The pthread_join( ) subroutine blocks the calling thread until the specified threadId thread terminates
- The programmer is able to obtain the target thread's termination return status if specified through pthread_exit( ), in the status parameter
- It is impossible to join a detached thread (discussed next)

Detatched threads are not joinable

- pthread_attr_t attr;
- pthread_attr_init (&attr)
- Pthread_attr_setdetachstate(&attr, *detachstate*)
  - detatchstate is PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE
- Pthread_attr_getdetachstate(&attr, detatchstate)
- Pthread_attr_destroy (&attr)
- Pthread_detach (threadId)
- According to the Pthreads standard, **all** threads should default to joinable, but older implementations may not be compliant.

What does detach do?

- Says that you will never *join* with the thread
- Therefore, thread information does not need to be kept until the thread is joined, and can be freed immediately
- That makes detached threads more efficient, especially when many threads are being created and destroyed/ending.

Locks

- Unlike Java, locks are not associated with every object
- Locks must be individually created
- pthread_mutex_init (mutex, attr)
- pthread_mutex_destroy (mutex)
  - what if the mutex is locked? Undefined!
- pthread_mutexattr_init (attr)
  - recursive, debug (reports state changes to debug interface), etc.
- pthread_mutexattr_destroy (attr)

Using locks

- declare lock using pthread_mutex_t mymutex;
- **pthread_mutex_lock** (mutex)
  - Acquire lock if available
  - Otherwise wait until lock is available
- pthread_mutex_trylock (mutex)
  - Acquire lock if available
  - Otherwise return lock-busy error
- pthread_mutex_unlock (mutex)
  - Release the lock to be acquired by another pthread_mutex_lock or trylock call
  - Cannot make assumptions about which thread will be woken up
- See **http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html** for an example

Using conditional variables

- Enables functionality similar to Java’s `wait( )`/`notify( )` calls
- Prevents programmer from having to loop on a variable to poll if a condition is true.

Condition variable scenario

- **Main Thread**
  - Declare and initialize global data/variables which require synchronization (such as "count")
  - Declare and initialize a *condition variable* object
  - Declare and initialize an associated mutex
  - Create threads A and B to do work

Thread A

- Execute up to where some condition must be true (e.g. *count* = *some value*)
- Lock associated mutex and check (and set) value of a global variable (e.g., ***count***)
- Call `pthread_cond_wait(condition, mutex)`
  - performs a blocking wait for signal from Thread-B.
  - call to `pthread_cond_wait()` unlocks the associated mutex variable so Thread-B can use it.
  - Wake up on signal -- Mutex is automatically and atomically locked
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
  - Change the value of the global variable that Thread-A is waiting on
  - Check if the value of the global Thread-A *wait* variable fulfills the desired condition, signal Thread-A with *pthread_cond_signal(&count_threshold_cv);*
- Unlock mutex
- Continue

Summary

- Pthreads and Java threads give similar functionality
- Consistency model for Pthreads between synchronization and thread creation/destruction calls is up to the individual compiler

qstruct04.ecn.purdue.edu - ee462b30@qstruct04 - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

```
[(qstruct04) ~/ ] more /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           E5310  @ 1.60GHz
stepping        : 7
cpu MHz         : 1595.930
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm syscal
l nx lm pni monitor ds_cpl tm2 cx16 xtpr lahf_lm
bogomips        : 3194.39
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

--More--(0%)
```

A useful command/file -- be aware of the effects of *hyperthreading*

qstruct04.ecn.purdue.edu - ee462b30@qstruct04 - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

total of 8 processors (0 – 7)

```
processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           E5310  @ 1.60GHz
stepping        : 7
cpu MHz         : 1595.930
cache size      : 4096 KB
physical id     : 1
siblings        : 4
core id         : 7
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm syscal
l nx lm pni monitor ds_cpl tm2 cx16 xtpr lahf_lm
bogomips        : 3191.89
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

[(qstruct04) ~/ ]
```

total of 4 physical cores (0 – 3)

qstruct04.ecn.purdue.edu - ee462b30@qstruct04 - SSH Secure Shell

File Edit View Window Help

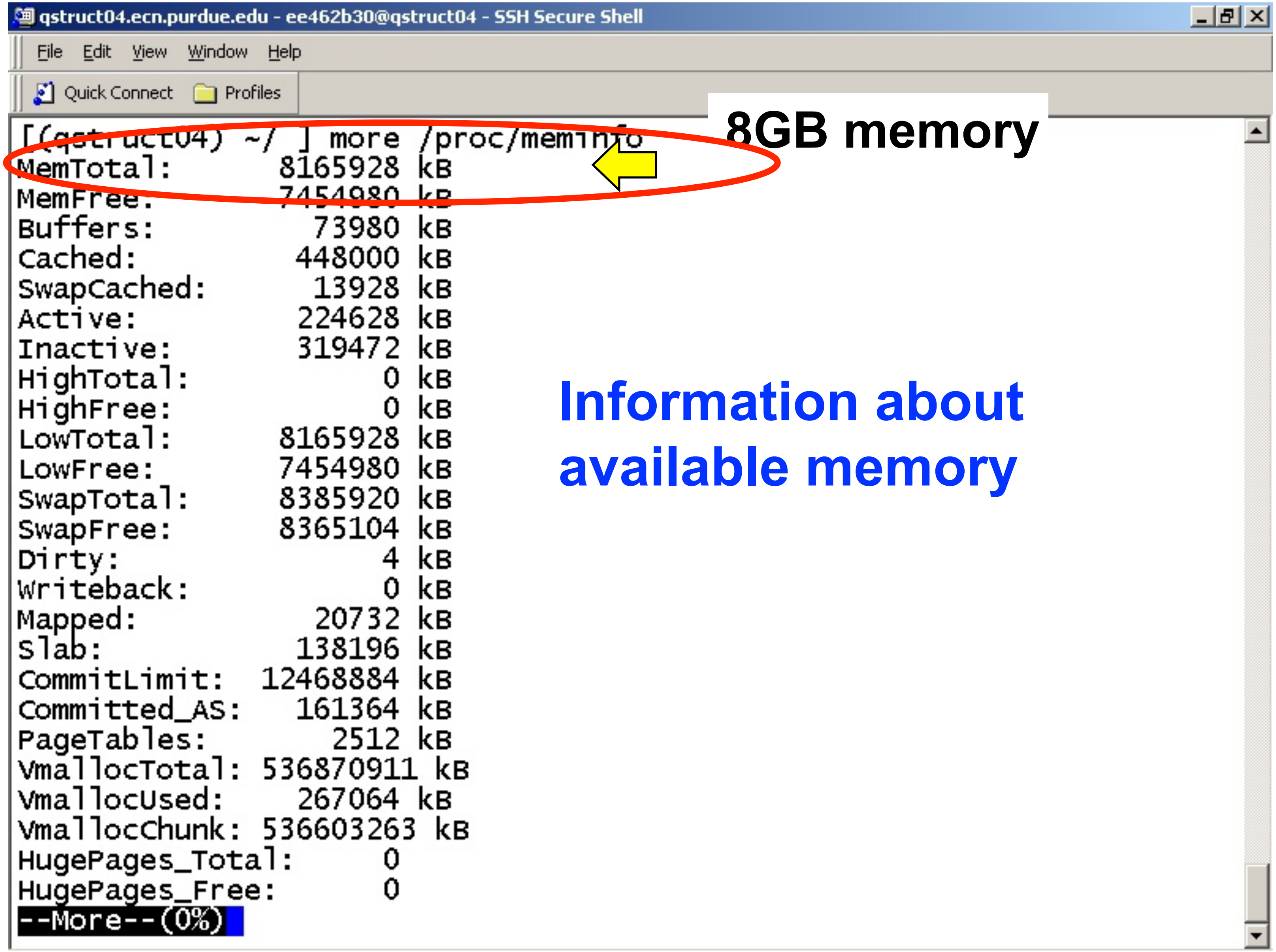
Quick Connect Profiles

total of 8 processors (0 – 7)

```
processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           E5310  @ 1.60GHz
stepping        : 7
cpu MHz         : 1595.930
cache size      : 4096 KB
physical id     : 1
siblings        : 4
core id         : 7
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm syscal
l nx lm pni monitor ds_cpl tm2 cx16 xtpr lahf_lm
bogomips        : 3191.89
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

[(qstruct04) ~/ ]
```

siblings equal number cores, no hyperthreading



Information about available memory

ECE 462
Object-Oriented Programming using C++ and Java

Parallel Program Performance

Performance Measurement

ts1 = current time;

execute the program without creating threads;

ts2 = current time;

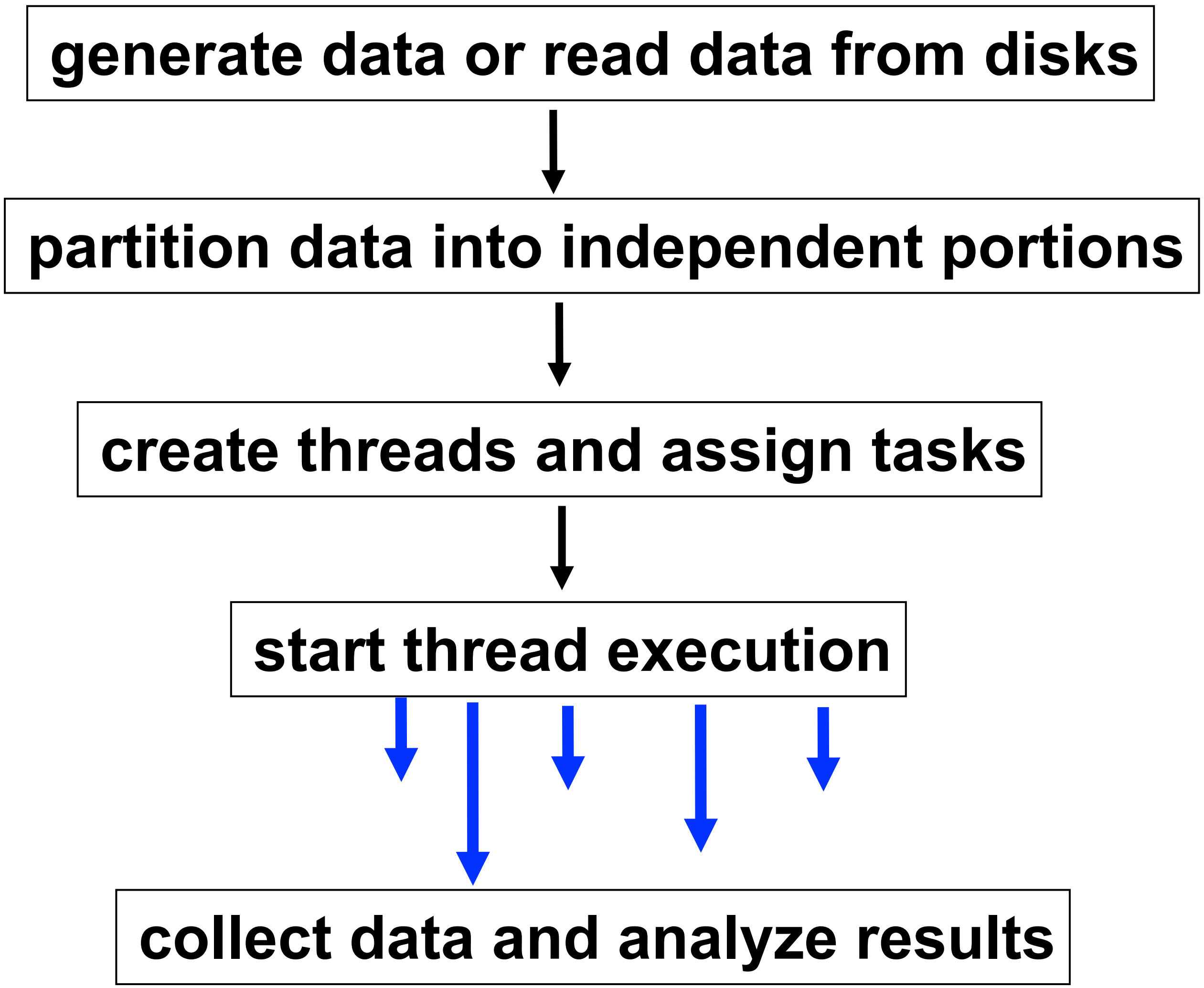
tp1 = current time;

execute the program with multiple threads

tp2 = current time;

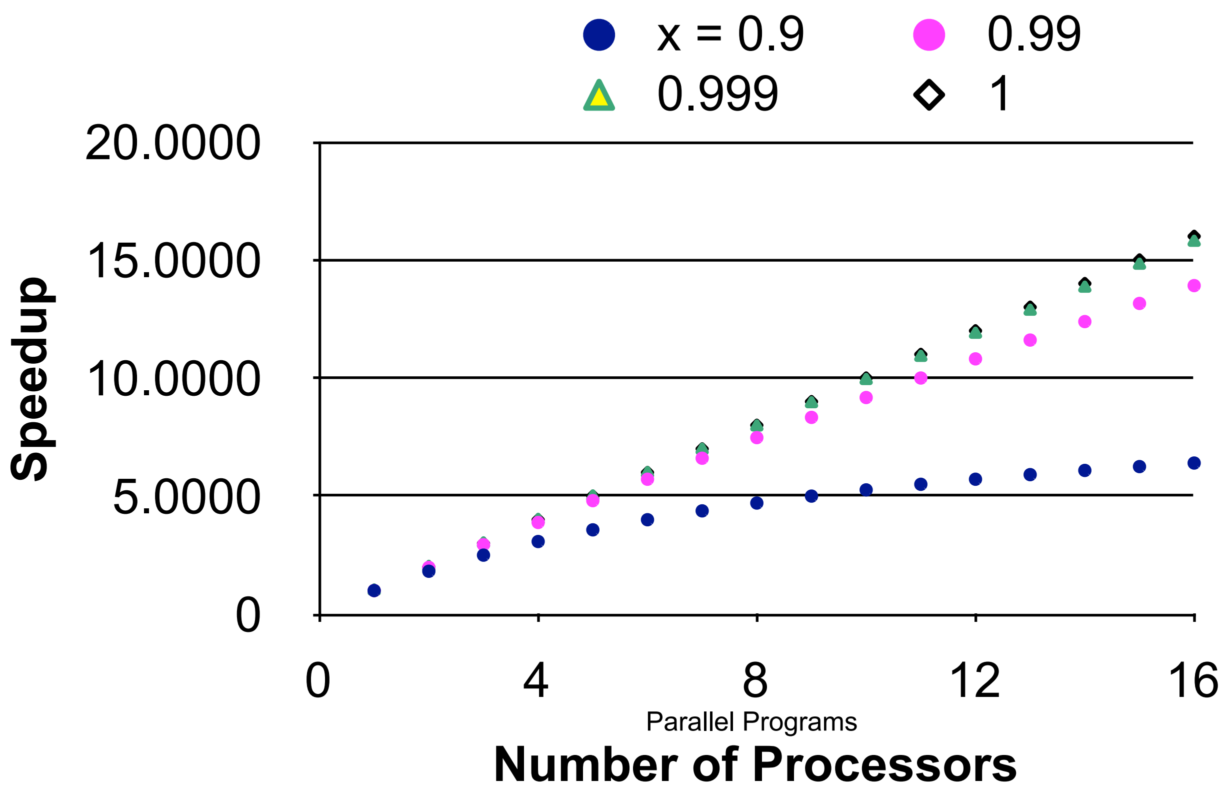
$$\text{improvement} = \frac{ts2 - ts1}{tp2 - tp1}$$

Structure of Multithread Programs



Amdahl's Law

- If a program has x fraction of parallel code, 1-x sequential code
- the speedup of using n threads (no sync) is $$\frac{1}{1-x+\frac{x}{n}}$$
- if x = 0.9, as n $\rightarrow \infty$, speedup = 10



Multi-Threaded = High Performance?

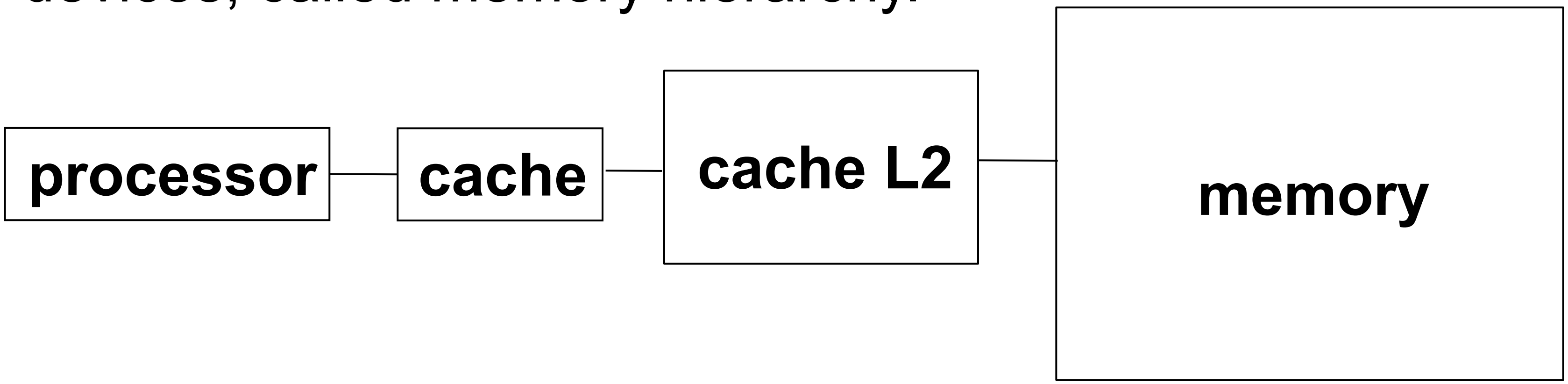
- multi-thread $\neq$ high performance (faster)
- If a program is IO-bound, multi-thread (or multi-core) may not help.
- Finding sufficient parallelism (make x closer to 1) can be difficulty.
- Reduce the sequential parts as much as possible
  - read data from multiple, parallel sources
  - partition data as they arrive
  - create long-living threads and reuse them
  - reduce competitions of mutex keys

Superlinear Speedup

- Sometimes, a multithread program's performance exceeds the number of processors.

$$\frac{\text{execution time of single thread}}{\text{execution time of multiple thread}} > \text{number of processors}$$

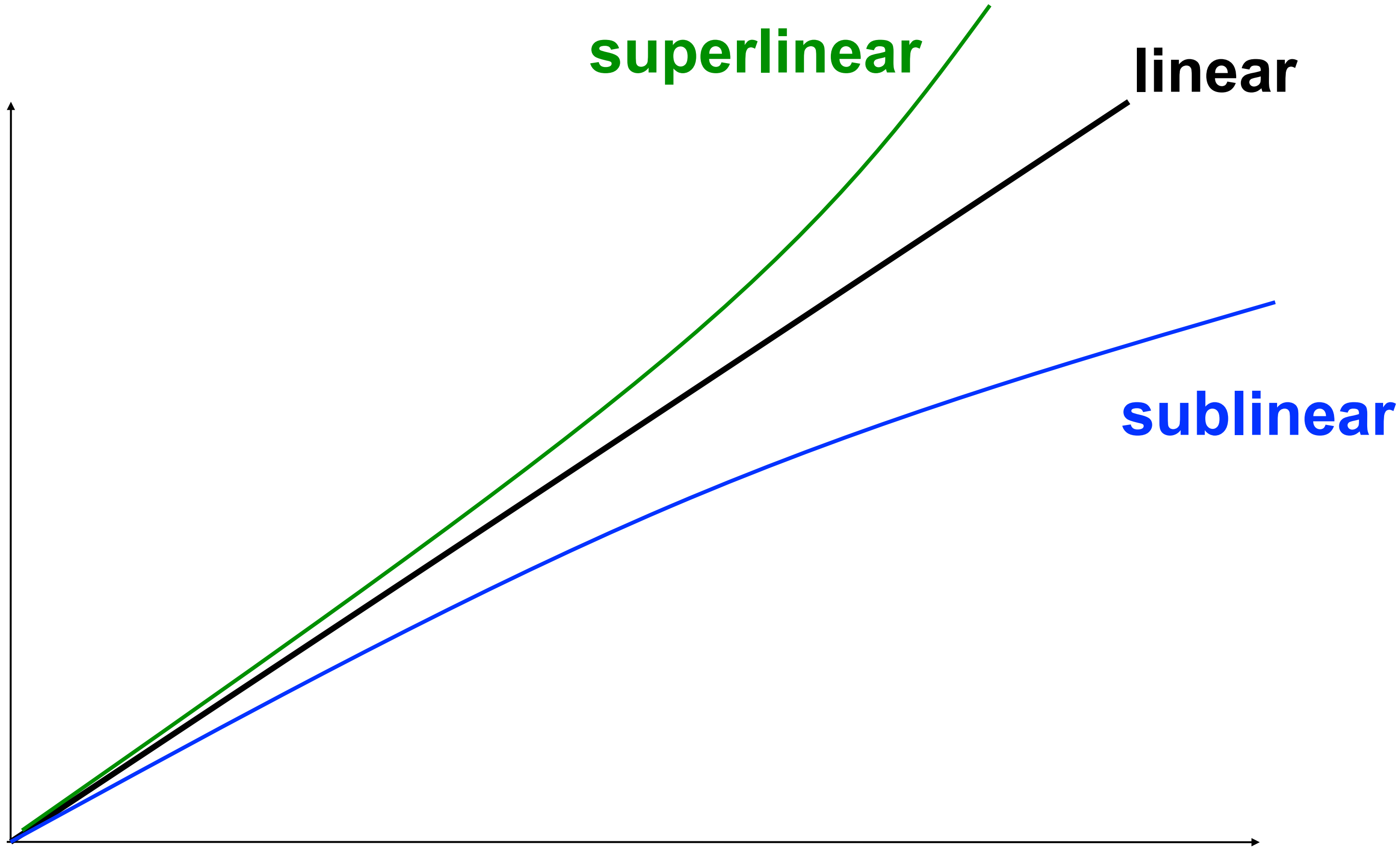
- All modern processors are built upon a series of storage devices, called memory hierarchy.



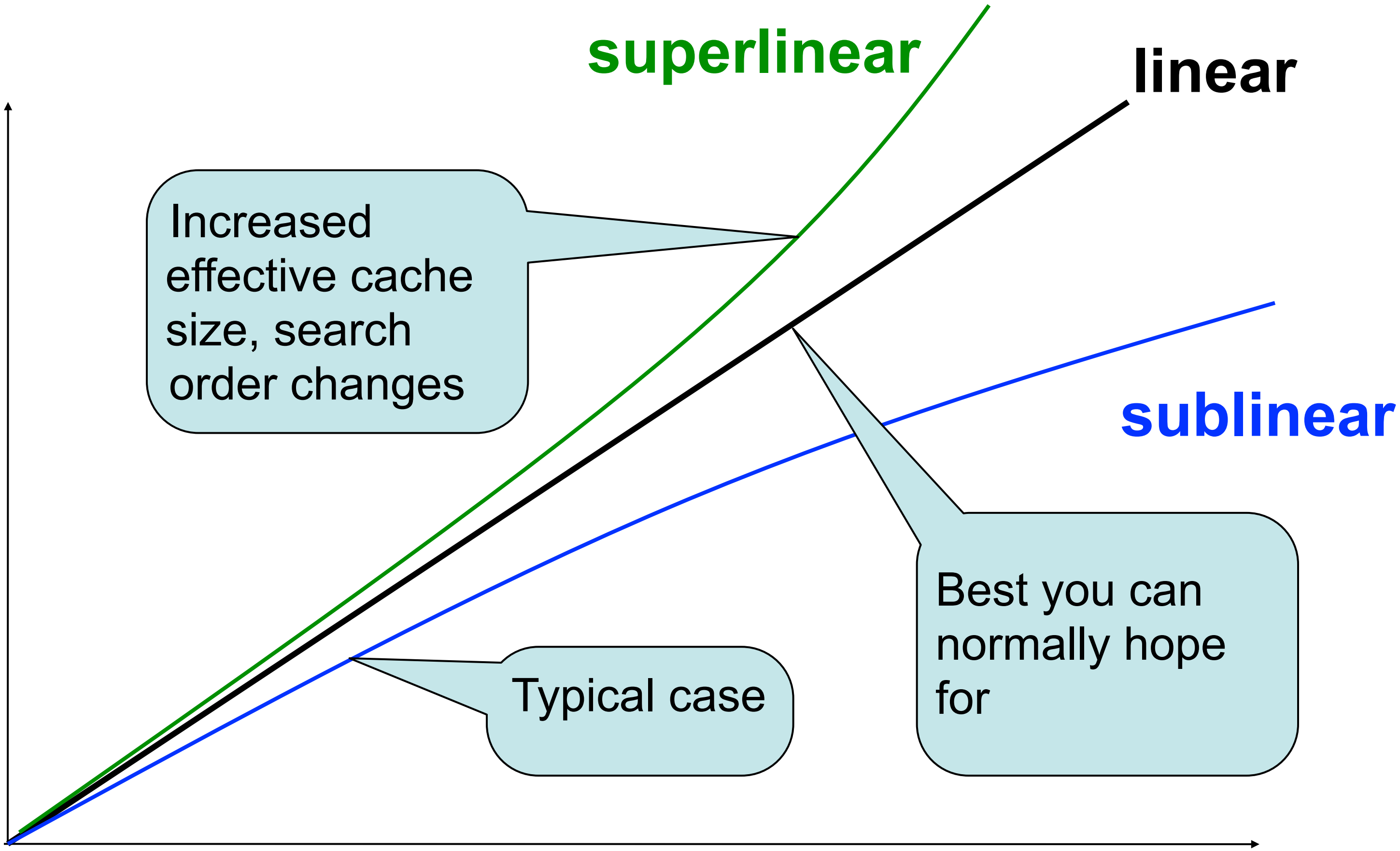
Memory Hierarchy

- Smaller and faster memory (cache) is installed closer to the processor. When data cannot fit into L1 cache, the data are stored in L2 cache, then memory.
- Multiple processors can have larger L1 cache collectively to accommodate more data for faster access. As a result, the program is faster.
- Multi-thread programs can also cause frequent data contention and trigger expensive (i.e. slow) consistency checking code. When this happens, the program can be much slower.

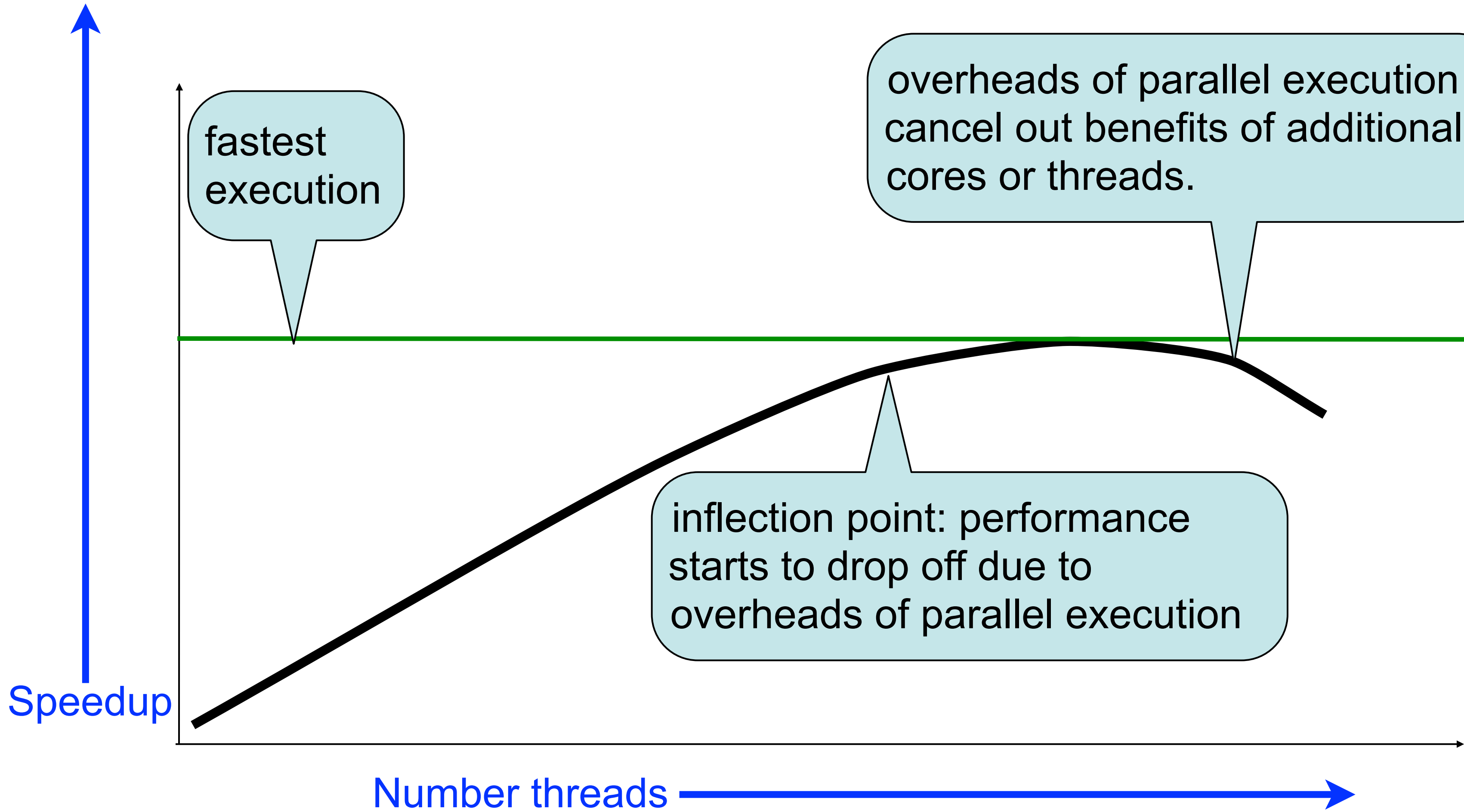
Superlinear or Sublinear



Superlinear or Sublinear



What often happens



Efficiency

- Efficiency is speedup/#processors
- Superlinear speedup has an efficiency > 1, linear has efficiency of 1, sub-linear speedup has efficiency < 1
- Efficiency is a measure of how well the processors are being utilized
  - Low efficiency says tune how parallelism is exploited
  - High efficiency says better performance must come by reducing work (e.g., a better algorithm) or more processors
  - Better (minimal work) algorithms sometimes are harder to parallelize

When to Use Multi-Threads?

- To meet parallelism requirement: multiple users access their bank accounts on-line.
- To provide quick response to a user: graphical user interface responds to mouse click, even though computation is still being performed
- To achieve faster computation: simulate an airplane's wings
- To provide fault tolerance: whether different threads (with different algorithms) reach the same answer
- .... many more ...