License

CS61C : Machine Structures

Lecture #41
Intra-Machine Parallelism and Threaded Programming

2008-5-7

TA Matt Johnson

inst.eecs.berkeley.edu/~cs61c-tm

Nvidia's Compute Unified Device Architecture

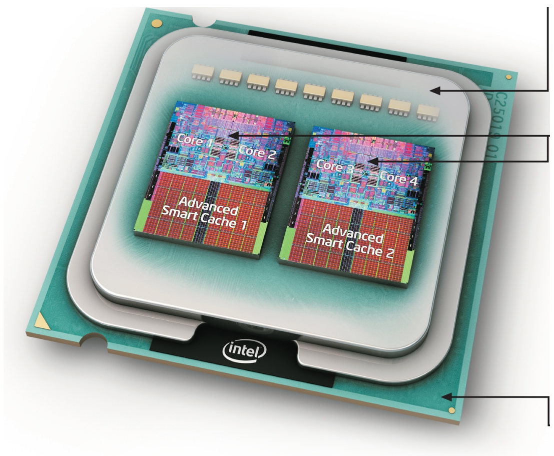
Nvidia's CUDA system for C was developed for the massive parallelism on their GPUs, but it's proving to be a useful API for general intra-machine parallel programming challenges.

http://www.geek.com/nvidia-is-shaking-up-the-parallel-programming-world/

http://hardware.slashdot.org/hardware/08/05/03/0440256.shtml

Review: Multicore everywhere!

- Multicore processors are taking over, *many*core is coming
- The processor is the "new transistor"
- This is a "sea change" for HW designers and especially for programmers
- Berkeley has world-leading research! (RAD Lab, Par Lab, etc.)



Outline for Today

- Motivation and definitions
- Synchronization constructs and PThread syntax
- Multithreading example: domain decomposition
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

How can we harness (many | multi)core?

- Is it good enough to just have multiple programs running simultaneously?

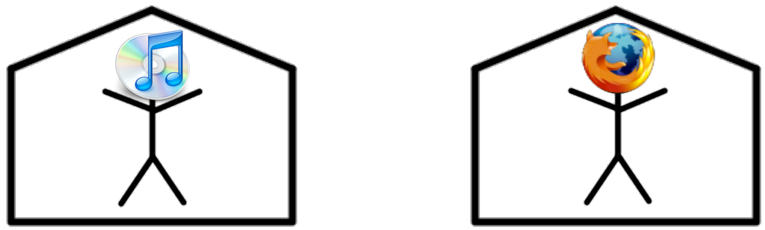
- We want per-program performance gains!

Crysis, Crytek 2007

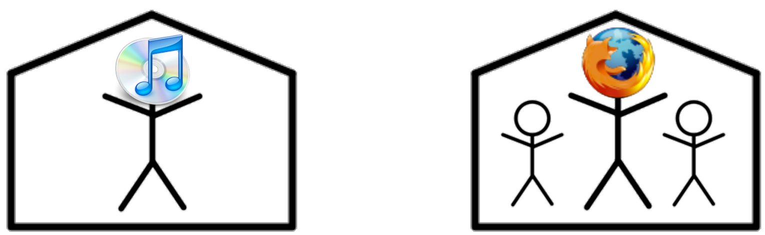
- The leading solution: *threads*

Definitions: threads v.s. processes

- A *process* is a "program" with its own address space.
  - A process has at least one thread!



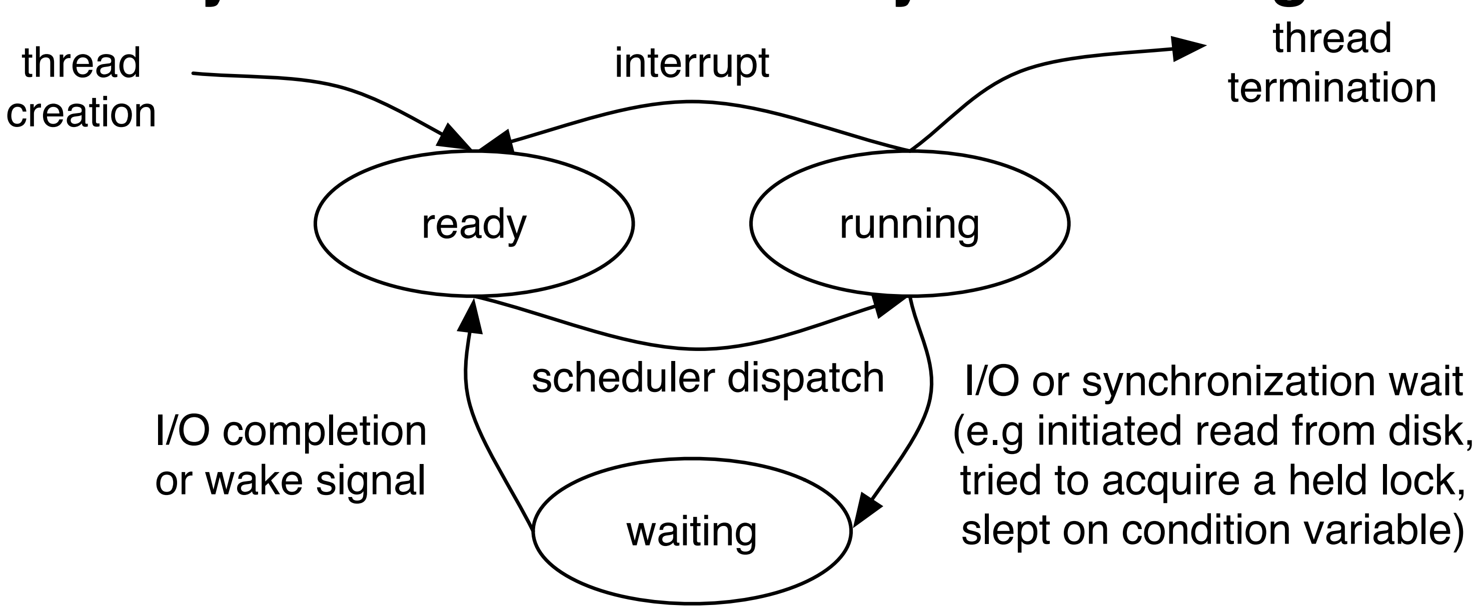
- A *thread of execution* is an independent sequential computational task with its own control flow, stack, registers, etc.
  - There can be many threads in the same process sharing the same address space



  - There are several APIs for threads in several languages. We will cover the PThread API in C.

How are threads *scheduled?*

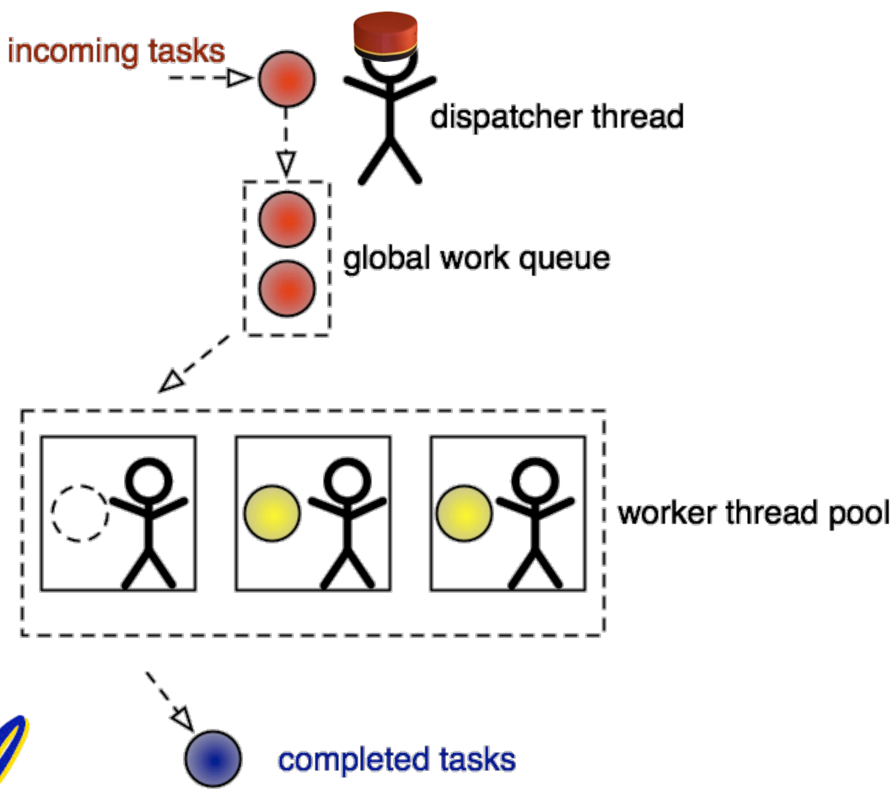
- **Threads/processes are run sequentially on one core or simultaneously on multiple cores**
  - **The operating system schedules threads and processes by moving them between states**
  - **# threads running = # logical cores on CPU**
  - **Many threads can be "ready" or "waiting"**



Based on diagram from Silberschatz, Galvin, and Gagne

Side: threading without multicore?

- Is threading useful without multicore?
  - Yes, because of I/O blocking!
- Canonical web server example:




```
global workQueue;

dispatcher() {
  createThreadPool();
  while(true) {
    task = receiveTask();
    if (task != NULL) {
      workQueue.add(task);
      workQueue.wake();
    }
  }
}

worker() {
  while(true) {
    task = workQueue.get();
    doWorkWithIO(task);
  }
}
```

Outline for Today

- Motivation and definitions
- Synchronization constructs and PThread syntax
- Multithreading example: domain decomposition
- Speedup issues
    - Overhead
    - Caches
    - Amdahl's Law

How can we make threads cooperate?

- If task can be completely decoupled into independent sub-tasks, cooperation required is minimal
  - Starting and stopping communication
- Trouble when they need to share data!
- Race conditions:

Scenario 1

Thread A	readX		incX		writeX	
Thread B		readX		incX		writeX

time -->

Scenario 2

Thread A	readX	incX	writeX			
Thread B				readX	incX	writeX

time -->

- We need to force some serialization
  - Synchronization constructs do that!

Lock / mutex semantics

- **A *lock* (mutual exclusion, mutex) guards a *critical section* in code so that only one thread at a time runs its corresponding section**
  - ***acquire* a lock before entering crit. section**
  - ***releases* the lock when exiting crit. section**
  - **Threads share locks, one per section to synchronize**
- **If a thread tries to acquire an in-use lock, that thread is put to sleep**
  - **When the lock is released, the thread wakes up *with the lock*! (blocking call)**

Lock / mutex syntax example in PThreads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int x;
```

```
threadA() {
  int temp = foo(x);
  pthread_mutex_lock(&lock);
  x = bar(x) + temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```

```
threadB() {
  int temp = foo(9000);
  pthread_mutex_lock(&lock);
  baz(x) + bar(x);
  x *= temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```

Thread A	readX	…	acquireLock => SLEEP				WAKE w/ LOCK	…	releaseLock
Thread B	…	acquireLock	readX	readX	writeX	releaseLock	…		

time -->

- **But locks don't solve everything…**
  - **Problem: potential deadlock!**

```
threadA() {
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
}
```

```
threadB() {
  pthread_mutex_lock(&lock2);
  pthread_mutex_lock(&lock1);
}
```

Condition variable semantics

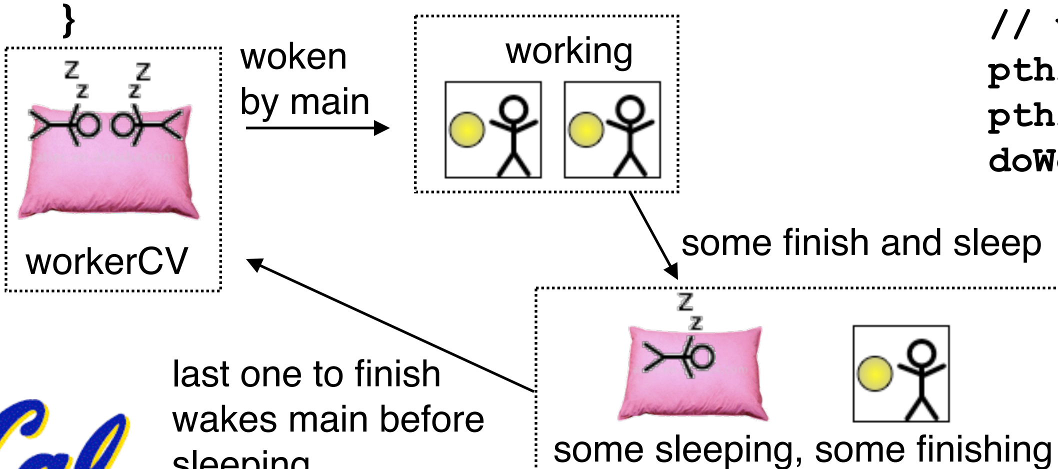
- *A condition variable* (CV) is an object that threads can sleep on and be woken from
  - *Wait* or *sleep* on a CV
  - *Signal* a thread sleeping on a CV to wake
  - *Broadcast* all threads sleeping on a CV to wake
  - I like to think of them as thread pillows…
- *Always* associated with a lock!
  - Acquire a lock before touching a CV
  - Sleeping on a CV releases the lock in the thread's sleep
  - If a thread wakes from a CV it will have the lock
- Multiple CVs often share the same lock

Condition variable example in PThreads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mainCV = PTHREAD_COND_INITIALIZER;
pthread_cond_t workerCV = PTHREAD_COND_INITIALIZER;
int A[1000];
int num_workers_waiting  = 0;

mainThread() {
  pthread_mutex_lock(&lock);
  // set up workers so they sleep on workerCV
  loadImageData(&A);
  while(true) {
    pthread_cond_broadcast(&workerCV);
    pthread_cond_wait(&mainCV,&lock);
    // A has been processed by workers!
    displayOnScreen(A);
  }
}
```

```
workerThreads()  {
while(true){
    pthread_mutex_lock(&lock);
    num_workers_waiting += 1;
    // if we are the last ones here…
    if(num_workers_waiting == NUM_THREADS){
       num_workers_waiting = 0;
       pthread_cond_signal(&mainCV);
    }
    // wait for main to wake us up
    pthread_cond_wait(&workerCV, &lock);
    pthread_mutex_unlock(&lock);
    doWork(mySection(A));}}
```



Creating and destroying PThreads

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5
pthread_t threads[NUM_THREADS];

int main(void) {
  for(int ii = 0; ii < NUM_THREADS; ii+=1) {
    (void) pthread_create(&threads[ii], NULL, threadFunc, (void *) ii);
  }

  for(int ii = 0; ii < NUM_THREADS; ii+=1) {
    pthread_join(threads[ii],NULL); // blocks until thread ii has exited
  }

  return 0;
}

void *threadFunc(void *id) {
  printf("Hi from thread %d!\n",(int) id);
  pthread_exit(NULL);
}
```

To compile against the PThread library, use gcc's -lpthread flag!

Side: OpenMP is a common alternative!

- PThreads aren't the only game in town
- OpenMP can automatically parallelize loops and do other cool, less-manual stuff!

```
#define N 100000
int main(int argc, char *argv[]){
   int i, a[N];
   #pragma omp parallel for
   for (i=0;i<N;i++)
      a[i]= 2*i;
   return 0;
}
```

Outline for Today

- Motivation and definitions
- Synchronization constructs and PThread syntax
- **Multithreading example: domain decomposition**
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

Domain decomposition demo (1)

- ***Domain decomposition*** **refers to solving a problem in a data-parallel way**
  - **If processing elements of a big array can be done independently, divide the array into sections (domains) and assign one thread to each!**
  - **(Common data parallelism in Scheme?)**
- **Remember the shader from Casey's lecture?**
  - ***Thanks for the demo, Casey!***

Domain decomposition demo (2)

```
void drawEllipse() {
    glBegin(GL_POINTS);
    for(int x = 0; x < viewport.w; x++) {
        for(int y = 0; y < viewport.h; y++) {
            float sX = sceneX(x);
            float sY = sceneY(y);
            if(inEllip(sX,sY)) {
                vec3 ellipPos    = getEllipPos(sX,sY);
                vec3 ellipNormal = getEllipNormal(ellipPos);
                vec3 ellipColor  = getEllipColor(ellipNormal,ellipPos);
                setPixel(x, y, ellipColor);
            }
        }
    }
    glEnd();
}

void setPixel(int x, int y, GLfloat r, GLfloat g, GLfloat b) {
          // openGL calls work via an internal state machine
          // what would you call this section?
          glColor3f(r, g, b);
          glVertex2f(x, y);
}
```

Domain decomposition demo (3)

- Demo shown here

Outline for Today

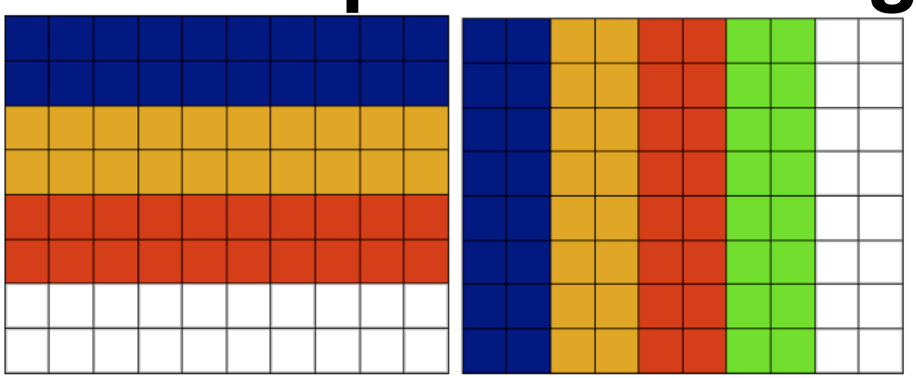
- Motivation and definitions
- Synchronization constructs and PThread syntax
- Multithreading example: domain decomposition
- Speedup issues
    - Overhead
    - Caches
    - Amdahl's Law

Speedup issues: overhead

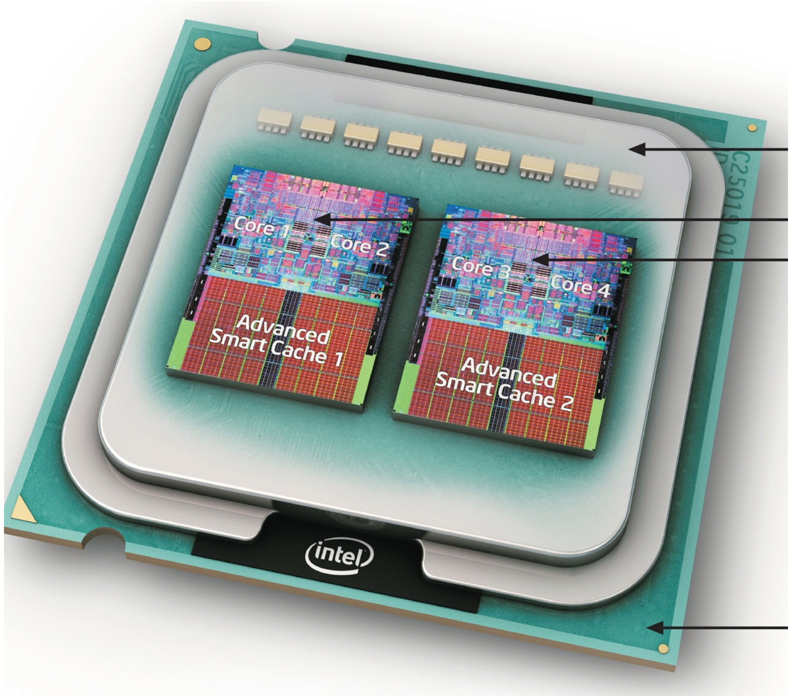
- In the demo, we saw (both relative to single threaded version):
  - 2 threads => ~50% performance boost!
  - 3 threads => ~10% performance boost!?
- More threads does not always mean better!
  - I only have two cores…
  - Threads can spend too much time *synchronizing* (e.g. waiting on locks and condition variables)
- Synchronization is a form of overhead
  - Also communication and creation/ deletion overhead

Speedup issues: caches

- Caches are often one of the largest considerations in performance
- For multicore, common to have independent L1 caches and shared L2 caches
- Can drive domain decomposition design

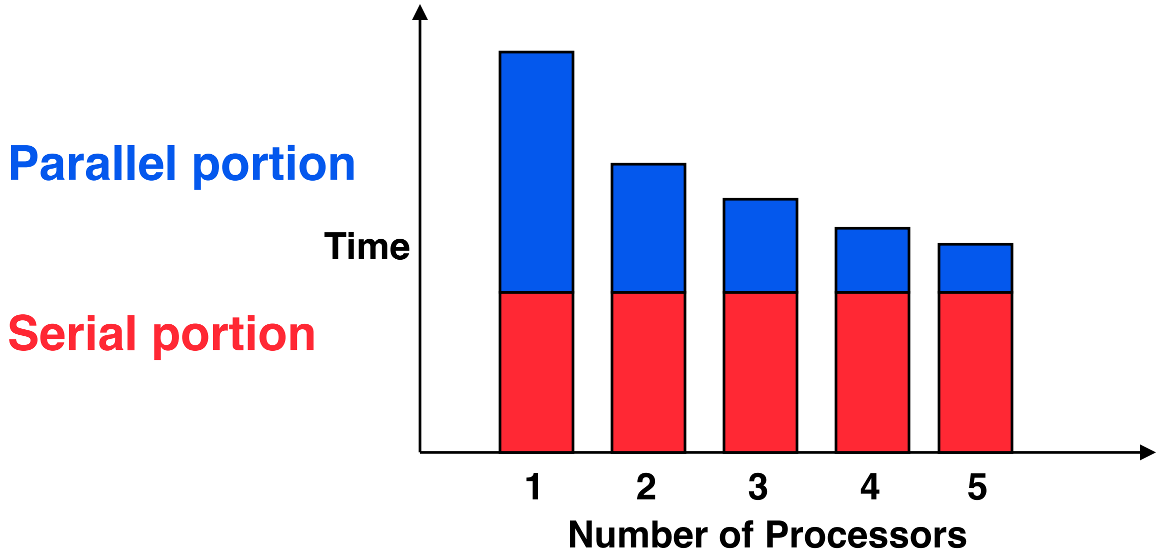


(a) "Horizontal" Decomposition (b) "Vertical" Decomposition



Speedup Issues: Amdahl's Law

- Applications can almost <u>never</u> be completely parallelized; some serial code remains



- s is serial fraction of program, P is # of processors
- Amdahl's law:

$$\text{Speedup}(P) = \text{Time}(1) / \text{Time}(P)$$

$$\leq 1 / (s + ((1-s) / P)), \text{ and as } P \to \infty$$

$$\leq 1/s$$

- Even if the parallel portion of your application speeds up perfectly, your performance may be limited by the sequential portion

Pseudo-PRS Quiz

- Super-linear speedup is possible
- Multicore is hard for architecture people, but pretty easy for software
- Multicore made it possible for Google to search the web

Pseudo-PRS Answers!

- Super-linear speedup is possible
  True: more cores means simply more cache accessible (e.g. L1), so some problems may see super-linear speedup
- Multicore is hard for architecture people, but pretty easy for software
  False: parallel processors put the burden of concurrency largely on the SW side
- Multicore made it possible for Google to search the web
  False: web search and other Google problems have huge amounts of data. The performance bottleneck becomes RAM amounts and speeds! (CPU-RAM gap)

Summary

- Threads can be *awake* and ready/running on a core or *asleep* for sync. (or blocking I/O)
- Use PThreads to thread C code and use your multicore processors to their full extent!
  - `pthread_create(), pthread_join(), pthread_exit()`
  - `pthread_mutex_t, pthread_mutex_lock(), pthread_mutex_unlock()`
  - `pthread_cond_t, pthread_cond_wait(), pthread_cond_signal(), pthread_cond_broadcast()`
- Domain decomposition is a common technique for multithreading programs
- Watch out for
  - Synchronization overhead
  - Cache issues (for sharing data, decomposing)
  - Amdahl's Law and algorithm parallelizability