CS 194 Parallel Programming

Creating and Using Threads

Katherine Yelick
yelick@cs.berkeley.edu
http://www.cs.berkeley.edu/~yelick/cs194f07
Parallel Programming Models

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

- **Control**
  - How is parallelism created?
  - How is ordering enforced?

- **Data**
  - Can data be shared or is it all private?
  - How is shared data accessed or private data communicated?

- **Synchronization**
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
Simple Example

• Consider applying a function \texttt{square} to the elements of an array \texttt{A} and then computing its sum:

\begin{align*}
A &= \text{array of all data} \\
\text{Asqr} &= \text{map(square, } A) \\
s &= \text{sum(Asqr)}
\end{align*}

• Stop and discuss:
  • What can be done in parallel?
  • How long would it take (in big-O) if we have an unlimited number of processors?
Problem Decomposition

In designing a parallel algorithm
- Decompose the problem into smaller tasks
- Determine which can be done in parallel with each other, and which must be done in some order
- Conceptualize a decomposition as a task dependency graph:
  - A directed graph with
  - Nodes corresponding to tasks
  - Edges indicating dependencies, that the result of one task is required for processing the next.
- A given problem may be decomposed into tasks in many different ways.
- Tasks may be of same, different, or indeterminate sizes.
Example: Multiplying a Matrix with a Vector

\[ \begin{align*}
\text{for } i &= 1 \text{ to } n \\
\text{for } j &= 1 \text{ to } n \\
y[i] &= y[i] + A[i,j] \times x[j];
\end{align*} \]

**Dependencies:** Each output element of \( y \) depends on one row of \( A \) and all of \( x \).

**Task graph:** Since each output is independent, our task graph can have \( n \) nodes and no dependence edges.

**Observations:** All tasks are of the same size in terms of number of operations.

**Question:** Could we have more tasks? Fewer?

8/30/2007

Slide derived from: Grama, Karypis, Kumar and Gupta
Example: Database Query Processing

Consider the execution of the query:

MODEL = "CIVIC" AND YEAR = 2001 AND
(COLOR = "GREEN" OR COLOR = "WHITE")

on the following database:

<table>
<thead>
<tr>
<th>ID#</th>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Dealer</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>4523</td>
<td>Civic</td>
<td>2002</td>
<td>Blue</td>
<td>MN</td>
<td>$18,000</td>
</tr>
<tr>
<td>3476</td>
<td>Corolla</td>
<td>1999</td>
<td>White</td>
<td>IL</td>
<td>$15,000</td>
</tr>
<tr>
<td>7623</td>
<td>Camry</td>
<td>2001</td>
<td>Green</td>
<td>NY</td>
<td>$21,000</td>
</tr>
<tr>
<td>9834</td>
<td>Prius</td>
<td>2001</td>
<td>Green</td>
<td>CA</td>
<td>$18,000</td>
</tr>
<tr>
<td>6734</td>
<td>Civic</td>
<td>2001</td>
<td>White</td>
<td>OR</td>
<td>$17,000</td>
</tr>
<tr>
<td>5342</td>
<td>Altima</td>
<td>2001</td>
<td>Green</td>
<td>FL</td>
<td>$19,000</td>
</tr>
<tr>
<td>3845</td>
<td>Maxima</td>
<td>2001</td>
<td>Blue</td>
<td>NY</td>
<td>$22,000</td>
</tr>
<tr>
<td>8354</td>
<td>Accord</td>
<td>2000</td>
<td>Green</td>
<td>VT</td>
<td>$18,000</td>
</tr>
<tr>
<td>4395</td>
<td>Civic</td>
<td>2001</td>
<td>Red</td>
<td>CA</td>
<td>$17,000</td>
</tr>
<tr>
<td>7352</td>
<td>Civic</td>
<td>2002</td>
<td>Red</td>
<td>WA</td>
<td>$18,000</td>
</tr>
</tbody>
</table>

Slide derived from: Grama, Karypis, Kumar and Gupta
Example: Database Query Processing

One decomposition creates tasks that generate an intermediate table of entries that satisfy a particular clause.
Example: Database Query Processing

Here is a different decomposition

<table>
<thead>
<tr>
<th>ID#</th>
<th>Model</th>
<th>Year</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>4523</td>
<td>Civic</td>
<td>2001</td>
<td>White</td>
</tr>
<tr>
<td>6734</td>
<td>Civic</td>
<td>2001</td>
<td>White</td>
</tr>
<tr>
<td>4395</td>
<td>Civic</td>
<td>2001</td>
<td>Green</td>
</tr>
<tr>
<td>7352</td>
<td>Civic</td>
<td>2001</td>
<td>Green</td>
</tr>
</tbody>
</table>

Choice of decomposition will affect parallel performance.

8/30/2007

Slide derived from: Grama, Karypis, Kumar and Gupta
Granularity of Task Decompositions

- Granularity: number of tasks into which a problem is decomposed. Rough terminology
  - Fine-grained: large number of small tasks
  - Coarse-grained: smaller number of larger tasks

A coarse grained version of dense matrix-vector product.

8/30/2007

Slide derived from: Grama, Karypis, Kumar and Gupta
Degree of Concurrency

• The degree of concurrency of a task graph is the number of tasks that can be executed in parallel.
  • May vary over the execution, so we can talk about the maximum or average
  • The degree of concurrency increases as the decomposition becomes finer in granularity.

• A directed path in a task graph represents a sequence of tasks that must be processed one after the other.

• The critical path is the longest such path.

• These graphs are normally weighted by the cost of each task (node), and the path lengths are the sum of weights
Limits on Parallel Performance

• Parallel time can be made smaller by making decomposition finer.
• There is an inherent bound on how fine the granularity of a computation can be.
  • For example, in the case of multiplying a dense matrix with a vector, there can be no more than \((n^2)\) concurrent tasks.
• In addition, there may be communication overhead between tasks.
Shared Memory Programming

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

```
y = ..s ...
```

```
i: 2
```
```
i: 5
```
```
i: 8
```

```
P0
P1
Pn
```

```
Shared memory
```

```
Private memory
```
Shared Memory “Code” for Computing a Sum

```java
static int s = 0;
```

Thread 1
for i = 0, n/2-1
s = s + sqr(A[i])

Thread 2
for i = n/2, n-1
s = s + sqr(A[i])

- Problem is a race condition on variable s in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously
Shared Memory Code for Computing a Sum

A \[3\ 5\] \(f = \text{square}\)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>compute (f([A[i]])) and put in reg0</td>
<td>9 (f([A[i]])) and put in reg0</td>
</tr>
<tr>
<td>reg1 = s</td>
<td>reg1 = s</td>
</tr>
<tr>
<td>reg1 = reg1 + reg0</td>
<td>reg1 = reg1 + reg0</td>
</tr>
<tr>
<td>s = reg1</td>
<td>s = reg1</td>
</tr>
</tbody>
</table>

\[\text{static int } s = 0;\}

- Assume \(A = [3, 5]\), \(f\) is the square function, and \(s = 0\) initially
- For this program to work, \(s\) should be 34 at the end
  - but it may be 34, 9, or 25
- The \textit{atomic} operations are reads and writes
  - Never see \(\frac{1}{2}\) of one number, but no \(+=\) operation is not atomic
  - All computations happen in (private) registers
**Corrected Code for Computing a Sum**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>local_s1 = 0</code>&lt;br&gt;<code>for i = 0, n/2-1</code>&lt;br&gt;<code>local_s1 = local_s1 + sqr(A[i])</code>&lt;br&gt;<code>lock(lk);</code>&lt;br&gt;<code>s = s + local_s1</code>&lt;br&gt;<code>unlock(lk);</code></td>
<td><code>local_s2 = 0</code>&lt;br&gt;<code>for i = n/2, n-1</code>&lt;br&gt;<code>local_s2 = local_s2 + sqr(A[i])</code>&lt;br&gt;<code>lock(lk);</code>&lt;br&gt;<code>s = s + local_s2</code>&lt;br&gt;<code>unlock(lk);</code></td>
</tr>
</tbody>
</table>

- Since addition is associative, it’s OK to rearrange order
  - Right?
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
Parallel Programming with Threads
Overview of POSIX Threads

- POSIX: *Portable Operating System Interface for UNIX*
  - 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:

```c
int pthread_create(pthread_t *,
const pthread_attr_t *,
void * (*)(void *),
void *);
```

Example call:

```c
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
- **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

```c
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
See Millennium/NERSC docs for paths/modules

Stop, run code, and discuss
```
Loop Level Parallelism

• Many application have parallelism in loops
  • With threads:
    ... A [n];
    for (int i = 0; i < n; i++)
      ... pthread_create (square, ...,
                      &A[i]);

• Problem:
  • Overhead of thread creation is nontrivial
  • Square is not enough work for a separate thread (much less time to
    square a number than to create a thread)

• Unlike original example, this would overwrite A[i]; how
  would you do this if you wanted a separate result array?
Shared Data and Threads

• Variables declared outside of main are shared
• Object allocated on the heap may be shared (if pointer is passed)
• Variables on the stack are private: passing pointer to these around to other threads can cause problems

• Often done by creating a large “thread data” struct
  • Passed into all threads as argument
  • Simple example:
    ```
    char *message = "Hello World!\n";
    
    pthread_create( &thread1, 
    NULL, 
    (void*)&print_fun, 
    (void*) message);
    ```
(Details: Setting Attribute Values)

• Once an initialized attribute object exists, changes can be made. For example:
  • To change the stack size for a thread to 8192 (before calling pthread_create), do this:
    • pthread_attr_setstacksize(&my_attributes, (size_t)8192);
  • To get the stack size, do this:
    • size_t my_stack_size;
      pthread_attr_getstacksize(&my_attributes, &my_stack_size);

• Other attributes:
  • Detached state – set if no other thread will use pthread_join to wait for this thread (improves efficiency)
  • Guard size – use to protect against stack overflow
  • Inherit scheduling attributes (from creating thread) – or not
  • Scheduling parameter(s) – in particular, thread priority
  • Scheduling policy – FIFO or Round Robin
  • Contention scope – with what threads does this thread compete for a CPU
  • Stack address – explicitly dictate where the stack is located
  • Lazy stack allocation – allocate on demand (lazy) or all at once, “up front”
Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - SPMD “Single Program Multiple Data”
- Simple use of barriers -- all threads hit the same one
  
  ```
  work_on_my_problem();
  barrier;
  get_data_from_others();
  barrier;
  ```
- More complicated -- barriers on branches (or loops)
  
  ```
  if (tid % 2 == 0) {
    work1();
    barrier
  } else { barrier }
  ```
Creating and Initializing a Barrier

• To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```c
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, 3);
```

• The second argument specifies an object attribute; using NULL yields the default attributes.

• To wait at a barrier, a process executes:

```c
pthread_barrier_wait(&b);
```

• This barrier could have been statically initialized by assigning an initial value created using the macro

```c
PTHREAD_BARRIER_INITIALIZER(3).
```

Note: barrier is not in all pthreads implementations, but we’ll provide something you can use when it isn’t.
Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

- threads are working mostly independently
- need to access common data structure
  
  ```c
  lock *l = alloc_and_init();  /* shared */
  acquire(l);
  access data
  release(l);
  ```

- Java and other languages have lexically scoped synchronization
  - similar to cobegin/coend vs. fork and join tradeoff
- Semaphores give guarantees on “fairness” in getting the lock, but the same idea of mutual exclusion
- Locks only affect processors using them:
  - pair-wise synchronization
Mutexes in POSIX Threads

• To create a mutex:
  
  ```c
  #include <pthread.h>
  pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_init(&amutex, NULL);
  ```

• To use it:
  ```c
  int pthread_mutex_lock(amutex);
  int pthread_mutex_unlock(amutex);
  ```

• To deallocate a mutex
  ```c
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```

• Multiple mutexes may be held, but can lead to deadlock:
  ```c
  thread1                  thread2
  lock(a)                  lock(b)
  lock(b)                  lock(a)
  ```
Shared Memory Programming

Several other thread libraries besides PTHREADS
• E.g., Solaris threads are very similar
• Other older libraries P4, Parmacs, etc.
• OpenMP can also be used for shared memory parallel programmer
  • [http://www.openMP.org](http://www.openMP.org)
  • Easier to use, i.e., just mark a loop as parallel
  • But not available everywhere
  • And performance is harder to control
Summary of Programming with Threads

• POSIX Threads are based on OS features
  • Can be used from multiple languages (need appropriate header)
  • Familiar language for most of program
  • Ability to shared data is convenient

• Pitfalls
  • Data race bugs are very nasty to find because they can be intermittent
  • Deadlocks are usually easier, but can also be intermittent

• Researchers look at transactional memory an alternative
• OpenMP is commonly used today as an alternative
Monte Carlo Example
Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} \pi r^2 = \pi/4$
- Randomly throw darts at $x,y$ positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - $\pi = 4 \times$ ratio

\[ r = 1 \]
Pi in C

- Independent estimates of pi:

```c
main(int argc, char **argv) {
    int i, hits, trials = 0;
    double pi;

    if (argc != 2) trials = 1000000;
    else trials = atoi(argv[1]);

    srand(0);  // see hw tutorial

    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
}
```
Helper Code for Pi in UPC

• Required includes:

  #include <stdio.h>
  #include <math.h>
  #include <upc.h>

• Function to throw dart and calculate where it hits:

  int hit(){
    int const rand_max = 0xFFFFFFFF;
    double x = ((double) rand()) / RAND_MAX;
    double y = ((double) rand()) / RAND_MAX;
    if ((x*x + y*y) <= 1.0) {
      return(1);
    } else {
      return(0);
    }
  }
Parallelism in PI

• Stop and discuss

• What are some possible parallel task decompositions?
Administrivia

- See web page for lecture slides and some reading assignments
- Please fill out the course survey
- Homework 0 due Friday
- Homework 1 handed out by Friday (online)
- Pick up your NERSC user agreement forms from Brian
  - Return them to Brian when they’re filled out (list UC Berkeley as your institution and Prof. Yelick as PI)
  - If you didn’t get one, you can download and fax yourself
    http://www.nersc.gov/nusers/accounts/usage.php
Extra Slides
Simple Example

• Shared memory strategy:
  • small number \( p << n = \text{size}(A) \) processors
  • attached to single memory

• Parallel Decomposition:
  • Each evaluation and each partial sum is a task.

• Assign \( n/p \) numbers to each of \( p \) procs
  • Each computes independent “private” results and partial sum.
  • Collect the \( p \) partial sums and compute a global sum.

Two Classes of Data:

• Logically Shared
  • The original \( n \) numbers, the global sum.

• Logically Private
  • The individual function evaluations.
  • What about the individual partial sums?