A first application
Performance measurement and characterization
Today’s lecture

• Programming with threads
• Assignment #1
• Performance measurement and characterization
Announcements

• Room

• Quiz on Friday 1/18, 15 minutes, closed book, no notes

• A1 assigned on Monday, due 1/25

• Lectures slides
Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- A spawned thread executes asynchronously until it completes
Hello world with pthreads

#include <pthread.h>  …
void *Hello(void *arg) {
    sleep(1);
    ThreadArgs* args = (ThreadArgs*) arg;
    int TID = args->tid;
    cout << "Hello from thread " << TID << endl;
    pthread_exit(NULL);  return NULL;
}  //struct ThreadArgs { int tid; int _NT; }; 
int main(int argc, char *argv[ ]){
    pthread_t* threads = new pthread_t[NT];
    ThreadArgs* args = new ThreadArgs[NT];
    for(int t=0;t<NT;t++){
        args[t].tid = t;  args[t]._NT = NT;
        assert(!pthread_create(&threads[t], NULL, Hello, &args[t] ));
    }
    for(int t=0;t<NT;t++)
        assert(!pthread_join(threads[t], &val));
    pthread_exit(NULL);
}
A first application

- Sum a list of integers
  for i = 0:N-1
  sum = sum + x[i];
- Partition the array x[] into intervals, assign each to a unique thread
- Each thread sweeps over a reduced problem, compute in less time
Parallelizing the computation

- for i = 0:N-1
  sum = sum + x[i]
- Partition the array x[] into intervals, assign each to a unique thread
- Each thread sums a reduced problem, computes in less time
- Root thread sums the individual contributions
Main code

struct ThreadArgs {int tid; int64_t localSum; int64_t n; int NT; };  
pthread_t* threads = new pthread_t[nThreads];  
ThreadArgs* args = new ThreadArgs[nThreads];  
for(int t=0; t<nThreads; t++) {
   args[t].tid = t;                      args[t].n = N;  
   args[t].localSum = 0;    args[t].NT = nThreads;  
   assert(!pthread_create(&threads[t], NULL, sum, &args[t]));
}
for(int t=0; t<nThreads; t++) {
   void *value;
   pthread_join( threads[t], &value );
}
int64_t global_sum = 0;
for(int t=0; t<nThreads; t++)
   global_sum += args[t].localSum;

Thread function

```c
// assert(!pthread_create(&threads[t], NULL, sum, &args[t]));
void *sum(void *arg){
    ThreadArgs* args = (ThreadArgs*) arg;
    int TID = args->tid; int N = args->n;
    int i0 = TID*(N/(args->NT)),  i1 = i0 + (N/(args->NT));

    int64_t sum = 0;
    for (int i=i0;  i<i1;  i++)
        sum += x[i];

    args->localSum = sum;

    pthread_exit(NULL);
    return NULL;
}
```
Results

• ./sum 1 1000000000
  1.30 seconds

• ./sum 2 10^9
  0.79 seconds  [speedup = 1.64]

• ./sum 4 10^9
  0.69 seconds  [incremental speedup = 1.14]

• ./sum 8 10^9
  0.68 seconds  [incremental speedup = 1.01]
Using a more expensive kernel

- for (int i=i0; i<i1; i++)
  sum += \( \sin(x[i]) \);  
- >/sumSine 1 \(10^8\) 
  6.50 seconds  
- >/sumSine 2 \(10^8\) 
  3.27 seconds  [speedup = 1.99]  
- >/sumSine 4 \(10^8\) 
  1.63 seconds  [incremental speedup = 2.0]  
- >/sumSine 8 \(10^8\) 
  0.82 seconds  [incremental speedup = 1.99]
2nd application: testing for partiality

• Given a list of numbers, which are prime?
  primes <# threads> 2 17 31 3415501328329

• Code in $PUB/Examples/Threads/Primes/Pthreads/$
Today’s lecture

• A first program with threads

• Performance measurement and characterization
Measures of Performance

• Why do we measure performance?
• How do we report it?
  ‣ Completion time
  ‣ Processor time product
    Completion time × # processors
  ‣ Throughput: amount of work that can be accomplished in a given amount of time
  ‣ Relative performance: given a reference architecture or implementation
    AKA Speedup
Parallel Speedup and Efficiency

• How much of an improvement did our parallel algorithm obtain over the serial algorithm?

• Define the parallel speedup, \( S_P = \frac{T_1}{T_P} \)

\[
S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on } P \text{ processors}}
\]

• \( T_1 \) is defined as the running time of the “best serial algorithm”

• In general: not the running time of the parallel algorithm on 1 processor

• **Definition**: Parallel efficiency \( E_P = \frac{S_P}{P} \)
What can go wrong with speedup?

• Not always an accurate way to compare different algorithms….
• .. or the same algorithm running on different machines
• We might be able to obtain a better running time even if we lower the speedup
• If our goal is performance, the bottom line is running time $T_p$
Superlinear speedup

- We have a *super-linear* speedup when
  \[ E_P > 1 \Rightarrow S_P > P \]

- Is it real?
  - Super-linear speedups are often an artifact of inappropriate measurement technique
  - Where there is a super-linear speedup, a better serial algorithm may be lurking
Scalability

- A computation is **scalable** if performance increases as a “nice function” of the number of processors, e.g. linearly.
- In practice scalability can be hard to achieve:
  - Serial sections: code that runs on only one processor
  - “Non-productive” work associated with parallel execution, e.g. synchronization
  - Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives
  
  ```
  for i=0:n-1  sum = sum + x[i]
  ```

![Graph showing speedup vs. number of processors]

![Diagram of a tree structure]
Serial Sections

• Limit scalability
• Let $f =$ the fraction of $T_1$ that runs serially
• $T_1 = f \times T_1 + (1-f) \times T_1$
• $T_p = f \times T_1 + (1-f) \times T_1 / P$

Thus $S_p = 1/[f + (1 - f)/p]$

• As $P \rightarrow \infty$, $S_p \rightarrow 1/f$

• This is known as Amdahl’s Law (1967)
Amdahl’s law (1967)

• A serial section limits scalability
• Let $f = \text{fraction of } T_1 \text{ that runs serially}$
• *Amdahl’s Law* (1967) : As $P \to \infty$, $S_P \to 1/f$
Weak scaling

• Is Amdahl’s law pessimistic?
• Observation: Amdahl’s law assumes that the workload ($W$) remains fixed
• But parallel computers are used to tackle more ambitious workloads
• If we increase $W$ with $P$ we have weak scaling
  \[ f \text{ often decreases with } W \]
• We can continue to enjoy speedups
  ‣ Gustafson’s law [1992]
    www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf
Isoefficiency

- Consequence of Gustafson’s observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The *isoefficiency* function specifies the growth of N in terms of P
- If N is linear in P, we have a scalable computation
- Problem: the amount of memory per core is shrinking
Challenges to measuring performance

• Reproducibility
  ‣ Transient system operating conditions
  ‣ Differing systems or program configuration

• Measurements are imprecise
  ‣ “Heisenberg uncertainty principle:” measurement technique may affect performance
  ‣ Overheads and inaccuracy

• Explain anomalous behavior, but ignore anomalies that are not significant
Complications

• Cost of measuring a full run is prohibitive
  ‣ Ignore startup code if you plan to run for a much longer time in production

• Transient behavior
  ‣ Repeat your measurements
  ‣ “Warm up” the code before collecting measurements
  ‣ Ignore outliers unless their behavior is important to you
  ‣ Average time, maximum time, minimum time?
Measurement collection

- Report the *best* timings
  - Repeat results $\times 3$ to 5 until at least 2 measures agree to within… $5\%$, $10\%$
  - Report the minimum time
- Also report outliers
- A scatter plot or error bar can be useful
Why do we take the minimum time?
Measurement errors are not distributed symmetrically
Timing collection

• Measures of time
  ► Elapsed, or “wall clock” time
  ► CPU time = system + user time
  ► Overhead, resolution, and quantization effects

• Measurement tools
  ► Can be platform dependent, especially library routines
  ► Unix `time` command does a reasonable job for long-running programs
  ► `gettimeofday()`
Enable others to reproduce your results

- Builds confidence within a community
- Report where you ran, software versions, processor, etc.
  - `uname -a`
    
    ```
    Linux ccom-bang-login.local 2.6.32-220.13.1.el6.x86_64 #1 SMP Tue Apr 17 23:56:34 BST 2012 x86_64 x86_64 x86_64 GNU/Linux
    ```
  - `gcc --version`
    
    ```
    gcc (GCC) 4.7.2
    ```
Fin