Lecture 15—Compiler Optimizations

ECE 459: Programming for Performance

March 5, 2013

Previous Lecture

- Lock Granularity
- What "Reentrant" means
- Inlining
- Leveraging Abstractions (C++)

About That Midterm

Haven't looked at it yet; marking today.

Probably should've had only 5 short-answer questions and omitted 4(b).

The final will be better for you.

Part I

Compiler Optimizations

Introduction

We'll be looking at compiler optimizations.

Mostly related to performance.
Better gcc than you:

- For you, probably a waste of time
- Plus: optimizations make your code less readable.

Compilers have a host of optimization options.
We'll look at gcc.

Compiler Optimization

"Optimization" is a bit of a misnomer:
compilers don't generate "optimal" code.

Compilers do generate better code, though.

The program you wrote is too slow.
Contract of a compiler:
produce a program with same behaviour, but faster.

gcc Optimization Levels

`-O1` (`-O`)

- Reduce code size and execution time.
- No optimizations that increase compiliation time.

`-O2`

- All optimizations except space vs. speed tradeoffs.

`-O3`

- All optimizations.

`-O0` (default)

- Fastest compilation time, debugging works as expected.

Disregard Standards, Acquire Speedup

`-Ofast`

- All `-O3` optimizations and non-standards compliant optimizations, particularly `-ffast-math`. (Like `-fast` on Solaris.)

This turns off exact implementations of IEEE or ISO rules/specifications for math functions.

Generally, if you don't care about the exact result, you can use this for a speedup

Constant Folding

```
i = 1024 * 1024
```

Why do later something you can do now?

The compiler will not emit code that does the multiplication at runtime.
It will simply use the computed value

```
i = 1048576
```

- Enabled at all optimization levels.

Common Subexpression Elimination

`-fgcse`

- Perform a global[1] common subexpression elimination pass.
- (Also performs global constant and copy propagation.)
- Enabled at `-O2`, `-O3`.

Example:

```
a = b * c + g;
d = b * c * d;
```

Instead of computing `b * c` twice, we compute it once, and reuse the value in each expression

[1] across basic blocks

Constant Propagation

Moves constant values from definition to use.

- Valid if there's no intervening redefinition.

Example:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

with constant propagation would produce:

```
int x = 14;
int y = 0;
return 0;
```

Copy Propagation

Replaces direct assignments with their values.
Usually runs after common subexpression elimination.

Example:

```
y = x
z = 3 + y
```

with copy propagation would produce:

```
z = 3 + x
```

Dead Code Elimination

`-fdce`

- Remove any code guaranteed not to execute.
- Enabled at all optimization levels.

Example:

```
if (0) {
    z = 3 + x;
}
```

would not be included in the final executable

Loop Unrolling

`-funroll-loops`

- Unroll loops, using a fixed number of iterations.

Example:

```
for (int i = 0; i < 4; ++i)
    f(i)
```

would be transformed to:

```
f(0)
f(1)
f(2)
f(3)
```

(Also useful for SIMD).

Loop Interchange

`-floop-interchange`
Enhances locality; big wins possible.

Example: in C the following:

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        a[j][i] = a[j][i] * c
```

would be transformed to this:

```
for (int j = 0; j < M; ++j)
    for (int i = 0; i < N; ++i)
        a[j][i] = a[j][i] * c
```

since C is **row-major** (meaning a[1][1] is beside a[1][2]).
(Other possibility is **column-major**.)

Loop Fusion

Example:

```
for (int i = 0; i < 100; ++i)
    a[i] = 4

for (int i = 0; i < 100; ++i)
    b[i] = 7
```

would be transformed to this:

```
for (int i = 0; i < 100; ++i) {
    a[i] = 4
    b[i] = 7
}
```

Trade-off between data locality and loop overhead;
the inverse is also an optimization, called **loop fission**.

Loop-Invariant Code Motion

- Moves invariants out of a loop.
- Also called loop hoisting.

Example:

```
for (int i = 0; i < 100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```

would be transformed to this:

```
s = x * y;
for (int i = 0; i < 100; ++i) {
    a[i] = s * i;
}
```

Reduces the amount of work we have to do for each iteration of the loop.

Devirtualization

`-fdevirtualize`

- Attempt to convert virtual function calls to direct calls.
- Enabled with `-O2`, `-O3`.

Virtual functions impose overhead and impede other optimizations.

C++: must read the object's RTTI (run-time type information) and branch to the correct function.

Devirtualization (example)

Example:

```
class A {
    virtual void m();
};

class B : public A {
    virtual void m();
}

int main(int argc, char *argv[]) {

    std::unique_ptr<A> t(new B);
    t.m();
}
```

Devirtualization could eliminate RTTI access;

instead just call to B's `m`.

Needs call graph, which can be tricky to compute (more soon).

Scalar Replacement of Aggregates

`-fipa-sra`

- Replace references by values.
- Enabled at `-O2` and `-O3`.

Example:

```
{
    std::unique_ptr<Fruit> a(new Apple);
    std::cout << color(a) << std::endl;
}
```

could be optimized to:

```
std::cout << "Red" << std::endl;
```

if the compiler knew what `color` does.

Aliasing and Pointer Analysis

We've seen `restrict`: tell compiler that variables don't alias.

Pointer analysis automatically tracks the variables in your program to determine whether or not they alias.

If they don't alias, we can reason about side effects, reorder accesses, and do other types of optimizations.

Call Graph

- A directed graph showing relationships between functions.

Relatively simple to compute in C
(function pointers complicate things).
Hard for virtual function calls (C++/Java).

Virtual calls require pointer analysis to disambiguate.

Importance of Call Graphs

The call graph can enable optimization of the following:

```
int n;

int f() { /* opaque */ }

int main() {
    n = 5;
    f();
    printf("%d\n", n);
}
```

We could propagate the constant value 5 in `main()`, as long as `f()` does not write to `n`.

Tail Recursion Elimination

`-foptimize-sibling-calls`

- Optimize sibling and tail recursive calls.
- Enabled at `-O2` and `-O3`.

Example:

```
int bar(int N) {
    if (A(N))
        return B(N);
    else
        return bar(N);
}
```

Compiler can just replace the call to `bar` by a `goto`.
Avoids function call overhead and reduces call stack use.

Branch Predictions

gcc attempts to guess the probability of each branch to best order the code.
(for an `if`, fall-through is most efficient, why?)

Use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program.

Example (in the Linux kernel):

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Architecture-Specific

Two common ones: `-march` and `-mtune`
(`-march` implies `-mtune`).

- enable specific instructions that not all CPUs support (SSE4.2, etc.)
- **Example:** `-march=corei7`
- Good to use on your local machine, not so much for shipped code.

Part II

Profile-Guided Optimization

Moving Beyond Purely-Static Approaches

So far: purely-static (compile-time) optimizations.

Static approach has limitations:

- how aggressively to inline?
- which branch to predict?

Dynamic information answers these questions better.

Profile-Guided Optimization Workflow

1. Compile the program;
   produce an instrumented binary.
2. Run the instrumented binary:
   on a representative test suite;
   get run-time profiling information.
3. Compile final version, with the profiles you've collected.

Price of Performance from Profile-Guided Optimization

(–) Complicates the compilation process.
(+) Produces faster code (if inputs are good).

How much faster? 5–25%, per Microsoft.

Solaris, Microsoft VC++, and GNU gcc all support profile-guided optimization (to some extent).

Another Option

Just-in-time compilers (like Java Virtual Machines) automatically do profile-guided optimizations:

- compile code on-the-fly anyway;
- always collecting measurements for the current run.

What's Profiling Data Good For?

Synergistic optimizations:

- Inlining: inlining rarely-used code is a net lose;
  inlining often-executed code is a win,
  enables more optimizations.

  Everyone does this.

- Improving Cache Locality: make commonly-called functions, basic blocks adjacent;
  improves branch prediction (below) and with `switch`.

More Uses of Profiling Data

- Branch Prediction: a common architecture assumption—
  forward branches not taken, backwards branches taken.
  can make these assumptions more true with profiles;
  for a forward branch, make sure common case is fall-through.

- Virtual Call Prediction: inline the most common target
  (guarded by a conditional).

Part III

Summary

Summary

Today: Got a feel for what the optimization levels do.

We saw examples of compiler optimizations. Classes:

- Scalar optimizations.
  (e.g. common subexpression elimination,
  constant propagation, copy propagation).
- Redundant code optimizations.
  (e.g. dead code elimination).
- Loop optimizations.
  (e.g. loop interchange, loop fusion, loop fission,
  software pipelining).
- Alias and pointer analysis; call graphs.
  (uses: devirtualization, inlining, tail recursion elimination).

Summary II

Full list of gcc options:
http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
More optimization examples:
http://www.digitalmars.com/ctg/ctgOptimizer.html

Also, profile-guided optimization:
see what actually happens at runtime, optimize for that.

Lecture 16—Midterm Solutions, Profiling

ECE 459: Programming for Performance

March 7, 2013

Previous Lecture

- Compiler Optimizations (Laundry List)
  - Scalar opts, redundant code opts
  - Loop opts, alias and pointer opts, call graphs
- Profile-Guided Optimizations

Today

- Midterm Solutions

Part I

Midterm Solutions

Midterm Status

- Out to Morteza for marking.
- I will mark 4(b) myself and decide what to do; more about it later.

Question 1: Short-Answer

1. single vs master: single, doesn't have to wait for a particular thread to become free.
2. main reasons: context switches and cache misses.
3. I'd expect the 9-thread server to be faster, because responding to requests ought to be I/O-bound.
4. start a new thread (or push to a thread pool) for each of the list elements; can't parallelize list traversal.
5. `volatile` ensures that code reads from memory or writes to memory on each read/write operation; does not optimize them away. Does not prevent re-ordering, impose memory barriers, or protect against races.

Question 1: Short-Answer

6. oops, this was a doozy. Sorry!
   First, weak consistency: run `r2 = x` from T2 first,
   then `x = 1; r1 = y` from T1, and finally `y = 1`.
   Get `x = 1, r1 = 0, r2 = 0, y = 1`.

   Sequential consistency: at end, always have `x = 1, y = 1`.

   In T1, if you reach `r1 = y`, then you had to execute `x = 1`.
   Maybe you've executed `y = 1` already, maybe you haven't.
   If you have, then `r1 = 1`; eventually get `r1 = 1, r2 = 0`.
   If you haven't, then `r1 = 0, r2 = 1` eventually.
   In neither case is `r1 = r2 = 0` possible.

Question 1: Short-Answer

7. No, a race condition is two concurrent accesses to a shared variable, one of which is a write. If you're making a shared variable into a private variable, there's no way to make a new concurrent access to a shared variable—you're not creating a new shared variable.
8. Yes, the behaviour might change. I expect something a bit more detailed, but basically you can put a variable write of a shared variable in a parallel section; changing it to private will change the possible outputs.
9. Simple:

```
int * x = malloc(sizeof(int));
foo(x, x);
```

   It was fine to pass in the address of an `int` as well.
10. Some possibilities: the outer loop has a loop-carried dependence, or maybe it only iterates twice and you'd like to extract more parallelism.

Question 2: Zeroing a Register

Source: `http://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/`

(2a): the `mov` requires a constant value embedded in the machine language, which is going to be more bytes, so imposes more pressure on the instruction cache;
all else being equal (it is), `xor` is therefore faster.

(2b): first write down a set of reads and writes for each instruction.

#	W set	R set
1	`eax`	`eax`
2	`ebx`	`eax`
3	`eax`	`eax`
4	`eax`	`eax, ecx`

Now, read off that set for each pair of instructions; no OOO possible.

- 1–2: RAW on `eax`
- 1–3: RAW on `eax`, WAR on `eax`, WAW on `eax`
- 1–4: RAW on `eax`, WAR on `eax`, WAW on `eax`
- 2–3: WAR on `eax`
- 2–4: WAR on `eax`
- 3–4: RAW on `eax`, WAR on `eax`, WAW on `eax`

Question 2: Zeroing a Register

(2c): behind the scenes, register renaming is going on. We'll assume that we can rename the instructions here.

```
add eax, 1
mov ebx, eax
xor edx, edx
add edx, ecx
```

#	W set	R set
1	`eax`	`eax`
2	`ebx`	`eax`
3	`edx`	`edx`
4	`edx`	`ecx, edx`

Changed dependencies:

- 1–3: all gone
- 2–3: all gone
- 1–4: all gone
- 2–4: all gone
- 3–4: RAW on `edx`, WAR on `edx`, WAW on `edx`

Now we can run (1, 3) or (2, 3) out of order.
Still deps between (1, 2) and (3, 4), preventing (1, 4) or (2, 4).

Question 3: Dynamic Scheduling using Pthreads

Once again, this was more complicated than I thought. Sorry.

First, let's start by writing something that handles a chunk.

```
#define CHUNK_SIZE 50

typedef struct chunk { int i; int j; } chunk;

void handle_chunk(chunk * c) {
  int i = c->i, j = c->j;
  int count = 0;
  for (; j < 200 && count < CHUNK_SIZE; ++j)
    data[i][j] = calc(i+j);
  for (i++; i < 200 && count < CHUNK_SIZE; ++i) {
    for (j = 0; j < 200 && count < CHUNK_SIZE; ++j) {
      data[i][j] = calc(i + j);
    }
  }
}
```

Question 3: Dynamic Scheduling using Pthreads

Now let's implement code to pull off the queue.

```
pthread_mutex_t wq_lock = PTHREAD_MUTEX_INITIALIZER;

chunk * extract_work() {
  chunk * c;

  pthread_mutex_lock(&wq_lock);
  if (!work_queue.is_empty())
    c = work_queue.pop();
  pthread_mutex_unlock(&wq_lock);
  return c;
}
```

Each thread just needs to pull off the queue.

```
void * worker_thread_main(void * data) {
  while (1) {
    chunk * c = extract_work();
    if (!c) pthread_exit();
    handle_chunk(c);
  }
}
```

Question 3: Dynamic Scheduling using Pthreads

Finally, we populate the queue and start our threads.

```
queue work_queue;

int main() {
  for (int i = 0; i < 200; ++i) {
    for (int j = 0; j < 200; j += CHUNK_SIZE) {
      // relied on 200 % CHUNK_SIZE == 0
      chunk * c = malloc(sizeof(chunk));
      c->i = i; c->j = j;
      work_queue.enqueue(c);
    }
  }

  pthread_t threads[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_create(&threads[i], NULL,
                   worker_thread_main, NULL);
  }
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(&threads[i], NULL);
  }
}
```

Question 4a: Race Conditions

The race occurs because one thread is potentially reading from, say, `pflag[2]` while the other thread is writing to `pflag[2]`.

You can fix the race condition by wrapping accesses to `pflag[i]` with a lock. The easiest solution is to use a single global lock.

This race is benign because it causes, at worst, extra work; if the function reads `pflag[i]` being 1 when it should be 0, it'll still check `v % i == 0`.

Question 4b: Memory Barriers

I was happy to see that I didn't actually say that these were `pthread` locks, so maybe they don't have memory barriers.

I'll probably make this question a bonus mark, since it relies on extra information which I didn't give you.

For more information: `erdani.com/publications/DDJ_Jul_Aug_2004_revised.pdf`

(4b (i)) The idea is that the `new` operation is actually three steps: allocate; initialize; set the pointer.
Thread A might allocate and set the pointer (due to reordering).
Thread B could return the un-initialized object.

Question 4b: Memory Barriers

Part (ii) is actually really easy.
Just don't double-check the lock.
Protect the singleton with a single lock and be done with it.

Lecture 17—Midterm Results, A3 Discussion, Profiling

ECE 459: Programming for Performance

March 12, 2013

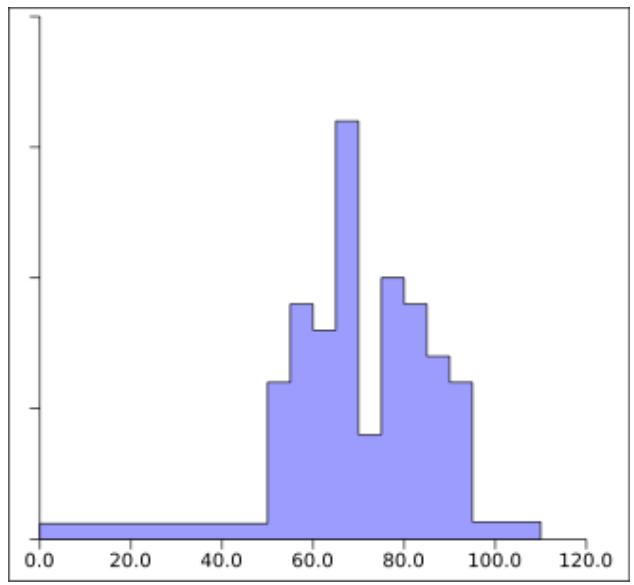
Previous Lecture

Midterm Solutions

Midterm Results

Note: I noticed that Morteza sometimes forgot to give points for Q1 (iii). Double-check your exam.

Raw mean: 69.96%
Raw median: 69.4%



Scaling

We can see two clumps:
"A": 75+
"B": 60–70

Here's the formula I used:

$$P_s = 5 + \frac{100}{90} M_r.$$

P_s is your midterm mark, in percent.
M_r is the raw mark.

Part I

About Assignment 3

Assignment Problem: Beier-Neely image morphing

> *"an image processing technique typically used as an animation tool for the metamorphosis of one image to another."*

Thaddeus Beier and Shawn Neely. "Feature-Based Image Metamorphosis". SIGGRAPH 1992, pp. 35–42.

Assignment Problem: Beier-Neely image morphing

> *"an image processing technique typically used as an animation tool for the metamorphosis of one image to another."*

Thaddeus Beier and Shawn Neely. "Feature-Based Image Metamorphosis". SIGGRAPH 1992, pp. 35–42.

Assignment Problem: Beier-Neely image morphing

> *"an image processing technique typically used as an animation tool for the metamorphosis of one image to another."*

Thaddeus Beier and Shawn Neely. "Feature-Based Image Metamorphosis". SIGGRAPH 1992, pp. 35–42.

Introduction

Image morphing:

computes intermediate images between source and dest.

More complicated than cross-dissolving between pictures

(interpolating pixel colours).

Morphing = warping + cross-dissolving.

Domain Discussion Omitted

Sorry, I had no time to learn and explain how code works.

We'll see it again on Thursday.

Algorithms

All of the C++ built-in algorithms work with anything using the standard container interface.

- `max_element`—returns an iterator with the largest element; you can use your own comparator function.
- `min_element`—same as above, but the smallest element.
- `sort`—sorts a container; you can use your own comparator function.
- `upper_bound`—returns an iterator to the first element greater than the value; only works on a **sorted** container (if the default comparator isn't used, you have to use the same one used to sort the container).
- `random_shuffle`—does `n` random swaps of the elements in the container

Some Hurdles

If you've worked with C++ before, you probably know the awful compiler messages and pages of template expansions.

- You can use `clang` if you have a compiler error, and let it show you the error instead `-std=c++11`
- For the profiler messages, it might get pretty bad. Look for one of the main functions, or if it's a weird name, look where it's called from.
- You can use Google Perf Tools to help break it down—more fine grained.

Example Profiler Function Output

```
[32] std::_Hashtable<unsigned int, std::pair<unsigned
     int const, std::unordered_map<unsigned int,
    double, std::hash<unsigned int>, std::equal_to<
    unsigned int>, std::allocator<std::pair<unsigned
    int const, double> > > >, std::allocator<std::
    pair<unsigned int const, std::unordered_map<
    unsigned int, double, std::hash<unsigned int>,
    std::equal_to<unsigned int>, std::allocator<std::
    pair<unsigned int const, double> > > > >, std::
    _Select1st<std::pair<unsigned int const, std::
    unordered_map<unsigned int, double, std::hash<
    unsigned int>, std::equal_to<unsigned int>, std::
    allocator<std::pair<unsigned int const, double> >
     > > >, std::equal_to<unsigned int>, std::hash<
    unsigned int>, std::__detail::_Mod_range_hashing,
     std::__detail::_Default_ranged_hash, std::
    __detail::_Prime_rehash_policy, false, false,
    true>::clear()
```

is actually `distance_map.clear()` (from last year's assignment), which is automatically called by the destructor.

Things You Can Do

Well, it's a basic implementation, so there should be a lot you can do.

- You can introduce threads, using pthreads (or C++11 threads, if you're feeling lucky), OpenMP, or whatever you want.
- Play around with compiler options.
- Use better algorithms or data strutures
  (maybe Qt is inefficient!)
- The list goes on and on.

Things You Need to Do

Profile!

- Keep your inputs constant between all profiling results so they're comparable.
- Baseline profile with no changes.
- You will pick your two best performance changes to add to the report.
  - You will include a profiling report before the change and just after the change (and only that change!)
  - More specific instructions in the handout.
- There may or may not be overlap between the baseline and the baseline for each change.
- My recommendation: use your initial baseline as the "before" for your first change, and the "after" of the first change for the baseline of your second change.
- Whatever you choose, it should be convincing.

Things To Notice

Your program will be run on `ece459-1` (or equivalent), with a 10 second limit.

We will have some type of leaderboard, so the earlier you have some type of submission, the better.

A Word

This assignment should be **enjoyable**.

Part II

Profiling

Introduction to Profiling

So far we've been looking at small problems.

Must **profile** to see what takes time in a large program.

Two main outputs:

- flat;
- call-graph.

Two main data gathering methods:

- statistical;
- instrumentation.

Profiler Outputs

Flat Profiler:

- Only computes the average time in a particular function.
- Does not include other (useful) information, like callees.

Call-graph Profiler:

- Computes call times.
- Reports frequency of function calls.
- Gives a call graph: who called what function?

Data Gathering Methods

Statistical:
Mostly, take samples of the system state, that is:

- every 2ns, check the system state.
- will cause some slowdown, but not much.

Instrumentation:
Add additional instructions at specified program points:

- can do this at compile time or run time (expensive);
- can instrument either manually or automatically;
- like conditional breakpoints.

Guide to Profiling

When writing large software projects:

- First, write clear and concise code.
  Don't do any premature optimizations—focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Things to Look For

Good signs:

- Time is spent in the right part of the system.
- Most time should not be spent handling errors; in non-critical code; or in exceptional cases.
- Time is not unnecessarily spent in the operating system.

`gprof` introduction

Statistical profiler, plus some instrumentation for calls.

Runs completely in user-space.

Only requires a compiler.

gprof usage

Use the `-pg` flag with `gcc` when compiling and linking.

Run your program as you normally would.

- Your program will now create a `gmon.out` file.

Use gprof to interpret the results: `gprof <executable>`.

gprof example

A program with 100 million calls to two math functions.

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}
```

- Looking at the code, we have no idea what takes longer.
- Probably would guess floating point math taking longer.
- (Overall, silly example.)

Example (Integer Math)

```
int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

Example (Float Math)

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Flat Profile

When we run the program and look at the profile, we see:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- One function per line.
- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.

Flat Profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- **calls:** number of times this function was called
- **self ns/call:** just self nanoseconds / calls
- **total ns/call:** average time for function execution, including any other calls the function makes

Call Graph Example (1)

After the flat profile gives you a feel for which functions are costly, you can get a better story from the call graph.

```
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000      int_math [2]
                0.43    6.04 100000000/100000000      float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000      main [1]
[2]     53.2    0.58    7.13 100000000          int_math [2]
                2.44    3.13 100000000/100000000      int_math_helper [4]
                1.56    0.00 100000000/300000000      int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000      main [1]
[3]     44.7    0.43    6.04 100000000          float_math [3]
                1.65    2.93 100000000/100000000      float_math_helper [6]
                1.47    0.00 100000000/300000000      float_power [7]
```

Reading the Call Graph

The line with the index is the current function being looked at **(primary line)**.

- Lines above are functions which called this function.
- Lines below are functions which were called by this function (children).

Primary Line

- **time:** total percentage of time spent in this function and its children
- **self:** same as in flat profile
- **children:** time spent in all calls made by the function
  - should be equal to self + children of all functions below

Reading Callers from Call Graph

Callers (functions above the primary line)

- **self:** time spent in primary function, when called from current function.
- **children:** time spent in primary function's children, when called from current function.
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function.

Reading Callees from Call Graph

Callees (functions below the primary line)

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - self + children is an estimate of time spent in current function when called from primary function.
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function.

Call Graph Example (2)

```
index % time    self  children    called     name

                2.44    3.13 100000000/100000000      int_math [2]
[4]     38.4    2.44    3.13 100000000          int_math_helper [4]
                3.13    0.00 200000000/300000000      int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000      int_math [2]
                3.13    0.00 200000000/300000000      int_math_helper [4]
[5]     32.4    4.69    0.00 300000000          int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000      float_math [3]
[6]     31.6    1.65    2.93 100000000          float_math_helper [6]
                2.93    0.00 200000000/300000000      float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000      float_math [3]
                2.93    0.00 200000000/300000000      float_math_helper [6]
[7]     30.3    4.40    0.00 300000000          float_power [7]
```

We can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc.

This example isn't too exciting; we could simplify the math.

Part III

gperftools

Introduction to gperftools

Google Performance Tools include:

- CPU profiler.
- Heap profiler.
- Heap checker.
- Faster `malloc`.

We'll mostly use the CPU profiler:

- purely statistical sampling;
- no recompilation; at most linking; and
- built-in visual output.

Google Perf Tools profiler usage

You can use the profiler without any recompilation.

- Not recommended—worse data.

```
LD_PRELOAD="/usr/lib/libprofiler.so" \
CPUPROFILE=test.prof ./test
```

The other option is to link to the profiler:

- `-lprofiler`

Both options read the `CPUPROFILE` environment variable:

- states the location to write the profile data.

Other Usage

You can use the profiling library directly as well:

- `#include <gperftools/profiler.h>`

Bracket code you want profiled with:

- `ProfilerStart()`
- `ProfilerEnd()`

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable.

- **Default value:** 100

pprof Usage

Like gprof, it will analyze profiling results.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output dot, ps, pdf or gif instead of gv.

Text Output

Similar to the flat profile in `gprof`

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Text Output Explained

Columns, from left to right:

Number of checks (samples) in this function.
Percentage of checks in this function.

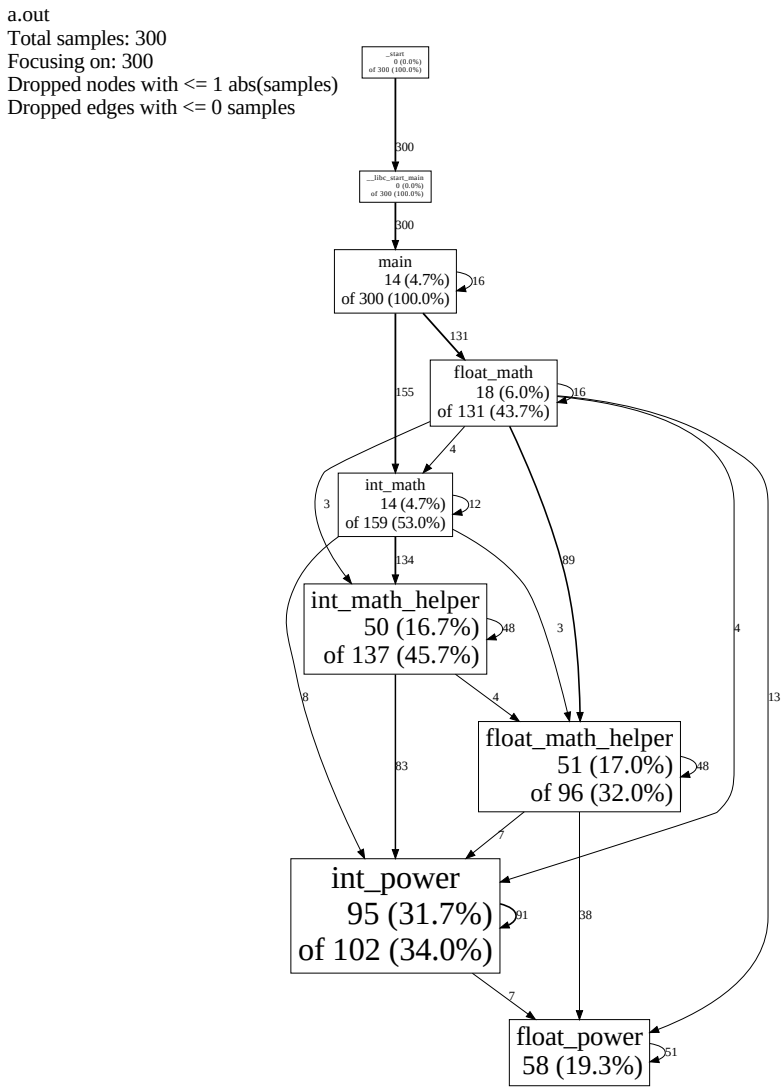
- Same as **time** in `gprof`.

Percentage of checks in the functions printed so far.

- Equivalent to **cumulative** (but in %).

Number of checks in this function and its callees.
Percentage of checks in this function and its callees.
Function name.

Graphical Output



Graphical Output Explained

Output was too small to read on the slide.

- Shows the same numbers as the text output.
- Directed edges denote function calls.
- Note: number of samples in callees =
  number in "this function + callees" -
  number in "this function".
- **Example:**
  `float_math_helper`, 51 (local) of 96 (cumulative)
  96 - 51 = 45 (callees)
  - callee `int_power` = 7 (bogus)
  - callee `float_power` = 38
  - callees total = 45

Things You May Notice

Call graph is not exact.

- In fact, it shows many bogus relations which clearly don't exist.
- For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `-focus` flag to look at individual functions.

Summary

- Talked about midterm and A3 (more on Thursday).
- Saw how to use `gprof`
  (one option for Assignment 3).
- Profile early and often.
- Make sure your profiling shows what you expect.

- We'll see other profilers we can use as well:
  - OProfile
  - Valgrind
  - AMD CodeAnalyst
  - Perf

Lecture 18—More A3, More Profiling Tools

ECE 459: Programming for Performance

March 14, 2013

Part I

More A3 Discussion

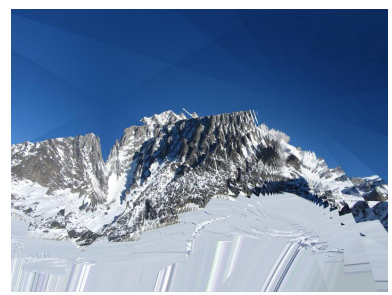
Morphing = Warping + Cross-Dissolving

$\Downarrow$ warp

warp $\Downarrow$

dissolve $\Longrightarrow$

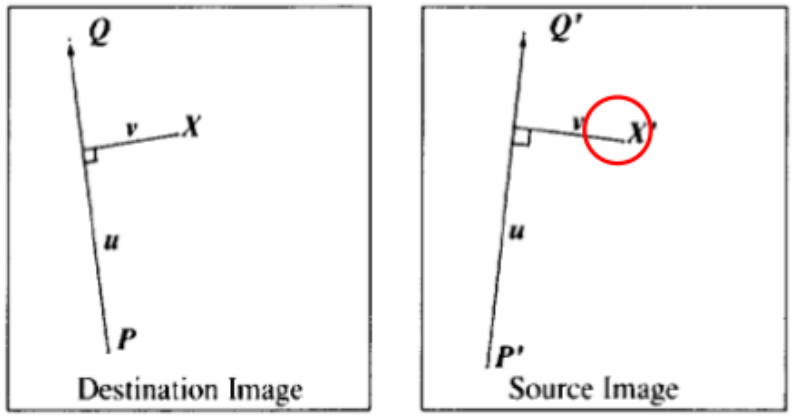
$\Longleftarrow$ dissolve



Warping Examples: Single-Line Warp

Next few figures are from the original paper.

Another reference:
`http://www.cs.unc.edu/~lazebnik/research/fall08/qi_mo.pdf`.



$$u = \frac{(X-P)\cdot(Q-P)}{\|Q-P\|^2} \quad (1)$$

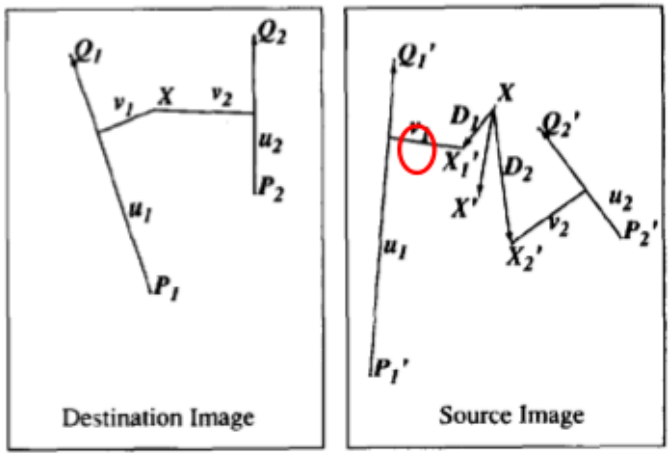
$$v = \frac{(X-P)\cdot Perpendicular(Q-P)}{\|Q-P\|} \quad (2)$$

$$X' = P' + u\cdot(Q'-P') + \frac{v\cdot Perpendicular(Q'-P')}{\|Q'-P'\|} \quad (3)$$

For each point in the destination, use the corresponding transformed point in the source.

Warping Examples: Multiple-Line Warp

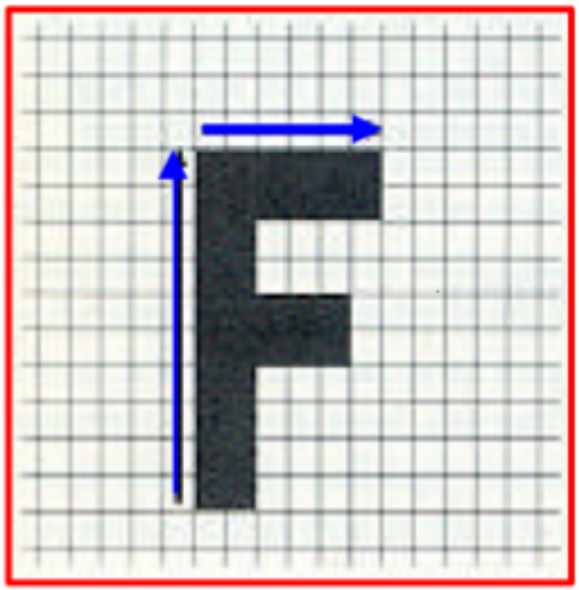
$$D_i = X_i' - X_i$$

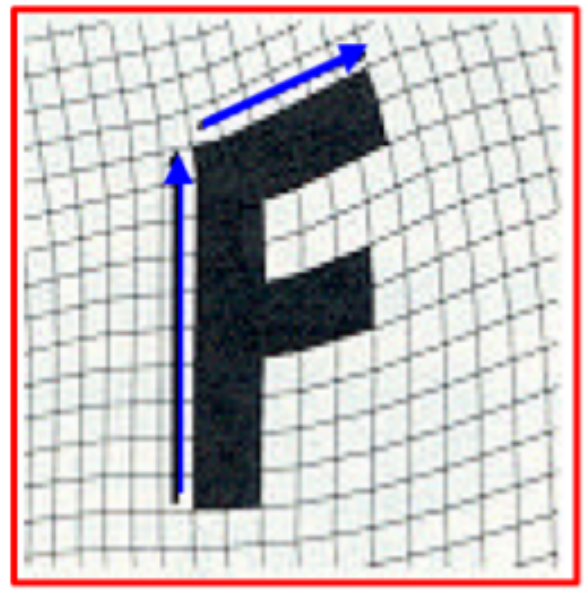


$$weight = \left(\frac{length^p}{(a + dist)} \right)^b$$

Generalize the definition of "corresponding point" for multiple lines.

Warped Image





Parameters: a, b, p

The warping algorithm takes three parameters:

- a: smoothness of warping
  (a near 0 means that lines go to lines)
- b: how relative strength of lines falls off with distance.
  large means every pixel only affected by nearest line;
  0 means each pixel affected by all lines equally.
  suggested range: $[0.5, 2]$
- p: relationship between length of line and strength.
  0 means all lines have same weight;
  1 means longer lines have greater relative weight;
  suggested range: $[0, 1]$.

Algorithm Pseudocode

for each pixel X in the destination:
    DSUM $\leftarrow$ (0, 0)
    weightsum $\leftarrow$ 0
    for each line P_iQ_i:
        calculate u, v based on P_iQ_i
        calculate X_i' based on u, v and $P_i'Q_i'$
        calculate displacement $D_i = X_i' - X_i$ for this line
        dist $\leftarrow$ shortest distance from X to P_iQ_i
        weight $\leftarrow$ (length^p / (a+dist))^b
        DSUM $+=$ $D_i \times$ weight
        weightsum $+=$ weight
    X' = X + DSUM / weightsum
    destinationImage(X) $\leftarrow$ sourceImage(X')

Introduction

- The code is more or less a direct translation of the high level functions.
- The language should not be the main hurdle, if there's anything you don't understand about the provided code, feel free to talk to me.
- Code uses standard library functions plus Qt types.

Built-in Data Structures

- QPoint
- QVector2D
- QImage

This is a fairly straightforward computation.

Functions

All of the code that you need to deal with is in `model.cpp`.

For your purposes, this file is called from `test_harness.cpp`; it is also called from `window.cpp`, the interactive front-end.

`model` contains three methods:

- `prepStraightLine`: draws the lines and computes auxiliary lines;
- `commonPrep`: draws auxiliary lines;
- `morph`: carries out the actual morphing algorithm.

I refactored `model` out of the initial `window.cpp`, and may have made some mistakes. If you find them, fix them.

Notes

I plan to add some more test cases and tweak the parameters shortly.

You'll probably want to refactor `morph()` to get useful profiling information from it.

You can remove code if you can prove it useless (using tests and text, which you should have anyway.)

Part II

System profiling: oprofile, perf, DTrace, WAIT

Introduction: oprofile

`http://oprofile.sourceforge.net`

Sampling-based tool.

Uses CPU performance counters.

Tracks currently-running function;
records profiling data for every application run.

Can work system-wide (across processes).

Technology: Linux Kernel Performance Events
(formerly a Linux kernel module).

Setting up oprofile

Must run as root to use system-wide, otherwise can use per-process.

```
% sudo opcontrol \
      --vmlinux=/usr/src/linux-3.2.7-1-ARCH/vmlinux
% echo 0 | sudo tee /proc/sys/kernel/nmi_watchdog
% sudo opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

Per-process:

```
[plam@lynch nm-morph]$ operf ./test_harness
operf: Profiler started

Profiling done.
```

oprofile Usage (1)

Pass your executable to opreport.

```
% sudo opreport -l ./test
CPU: Intel Core/i7, speed 1595.78 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not
halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        symbol name
7550     26.0749  int_math_helper
5982     20.6596  int_power
5859     20.2348  float_power
3605     12.4504  float_math
3198     11.0447  int_math
2601      8.9829  float_math_helper
160       0.5526  main
```

If you have debug symbols (-g) you could use:

```
% sudo opannotate --source \
--output-dir=/path/to/annotated-source /path/to/mybinary
```

oprofile Usage (2)

Use opreport by itself for a whole-system view.
You can also reset and stop the profiling.

```
% sudo opcontrol --reset
Signalling daemon... done
% sudo opcontrol --stop
Stopping profiling.
```

Perf: Introduction

`https://perf.wiki.kernel.org/index.php/Tutorial`

Interface to Linux kernel built-in sampling-based profiling.
Per-process, per-CPU, or system-wide.
Can even report the cost of each line of code.

Perf: Usage Example

On the Assignment 3 code:

```
[plam@lynch nm-morph]$ perf stat ./test_harness

 Performance counter stats for './test_harness':

       6562.501429 task-clock                #    0.997 CPUs utilized
               666 context-switches          #    0.101 K/sec
                 0 cpu-migrations            #    0.000 K/sec
             3,791 page-faults               #    0.578 K/sec
    24,874,267,078 cycles                    #    3.790 GHz                      [83.32%]
    12,565,457,337 stalled-cycles-frontend   #   50.52% frontend cycles idle     [83.31%]
     5,874,853,028 stalled-cycles-backend    #   23.62% backend  cycles idle     [66.63%]
    33,787,408,650 instructions              #    1.36  insns per cycle
                                             #    0.37  stalled cycles per insn [83.32%]
     5,271,501,213 branches                  #  803.276 M/sec                    [83.38%]
       155,568,356 branch-misses             #    2.95% of all branches          [83.36%]

       6.580225847 seconds time elapsed
```

Perf: Source-level Analysis

perf can tell you which instructions are taking time, or which lines of code.

Compile with `-ggdb` to enable source code viewing.

```
% perf record ./test_harness
% perf annotate
```

`perf annotate` is interactive. Play around with it.

DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

(Typical instrumentation can have a slowdown of 100x (Valgrind).)
Design goals:

1. No overhead when not in use;
2. Guarantee safety—must not crash
   (strict limits on expressiveness of probes).

DTrace: Operation

How does DTrace achieve 0 overhead?

- only when activated, dynamically rewrites code by placing a branch to instrumentation code.

Uninstrumented: runs as if nothing changed.

Most instrumentation: at function entry or exit points. You can also instrument kernel functions, locking, instrument-based on other events.

Can express sampling as instrumentation-based events also.

DTrace Example

You write this:

```
syscall::read:entry {
    self->t = timestamp;
}

syscall::read:return
/self->t/ {
    printf("%d/%d spent %d nsecs in read\n"
           pid, tid, timestamp - self->t);
}
```

`t` is a thread-local variable.
This code prints how long each call to `read` takes, along with context.

To ensure safety, DTrace limits what you write; e.g. no loops.

- (Hence, no infinite loops!)

Other Tools

AMD CodeAnalyst—based on oprofile; leverages AMD processor features.

WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.

Other Tools

AMD CodeAnalyst—based on oprofile.

WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.

WAIT: Introduction

Built for production environments.

Specialized for profiling JVMs, uses JVM hooks to analyze idle time.

Sampling-based analysis; infrequent samples (1–2 per minute!)

WAIT: Operation

At each sample: records each thread's state,

- call stack;
- participation in system locks.

Enables WAIT to compute a "wait state" (using expert-written rules):
what the process is currently doing or waiting on, e.g.

- disk;
- GC;
- network;
- blocked;
- etc.

WAIT: Workflow

You:

- run your application;
- collect data (using a script or manually); and
- upload the data to the server.

Server provides a report.

- You fix the performance problems.

Report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

Paper presents 6 case studies where WAIT identified performance problems: deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Other Profiling Tools

Profiling: Not limited to C/C++, or even code.

You can profile Python using `cProfile`; standard profiling technology.

Google's Page Speed Tool: profiling for web pages—how can you make your page faster?

- reducing number of DNS lookups;
- leveraging browser caching;
- combining images;
- plus, traditional JavaScript profiling.

Summary

- More Assignment 3 discussion
- System profiling tools:
  oprofile, DTrace, WAIT, perf

Future Lectures

- OpenMPI
- OpenCL
- Hadoop MapReduce
- Software Transactional Memory
- Early Phase Termination, Big Data

If there's anything else you want to see in this course, let me know.

Lecture 19—Reduced-resource computation, STM, Languages for HPC

ECE 459: Programming for Performance

March 19, 2013

Part I

Reduced-Resource Computation

Trading Accuracy for Performance

Consider Monte Carlo integration.
It illustrates a general tradeoff: accuracy vs performance.
You'll also implement this tradeoff manually in A4
(with domain knowledge).

Martin Rinard generalized the accuracy vs performance tradeoff with:

- early phase termination [OOPSLA07]
- loop perforation [CSAIL TR 2009]

Early Phase Termination

We've seen barriers before.
No thread may proceed past a barrier until all of the threads reach the barrier.

This may slow down the program: maybe one of the threads is horribly slow.

Solution: kill the slowest thread.

Early Phase Termination: Objection

"Oh no, that's going to change the meaning of the program!"

Early Phase Termination: When is it OK anyway?

OK, so we don't want to be completely crazy.

Instead:

- develop a statistical model of the program behaviour.
- only kill tasks that don't introduce unacceptable distortions.

When we run the program:

get the output, plus a confidence interval.

Early Phase Termination: Two Examples

Monte Carlo simulators:
Raytracers:

- already picking points randomly.

In both cases: spawn a lot of threads.

Could wait for all threads to complete;
or just compensate for missing data points,
assuming they look like points you did compute.

Early Phase Termination: Another Justification

In scientific computations:

- using points that were measured (subject to error);
- computing using machine numbers (also with error).

Computers are only providing simulations, not ground truth.
Actual question: is the simulation is good enough?

Loop Perforation

Like early-phase termination, but for sequential programs:
throw away data that's not actually useful.

```
for (i = 0; i < n; ++i) sum += numbers[i];
```

```
for (i = 0; i < n; i += 2) sum += numbers[i];
sum *= 2;
```

This gives a speedup of ~ 2 if `numbers[]` is nice.

Works for video encoding: can't observe difference.

Applications of Reduced Resource Computation

Loop perforation works for:

- evaluating forces on water molecules (summing numbers);
- Monte-Carlo simulation of swaption pricing;
- video encoding.

More on the video encoding example:
Changing loop increments from 4 to 8 gives:

- speedup of 1.67;
- signal-to-noise ratio decrease of 0.87%;
- bitrate increase of 18.47%;
- visually indistinguishable results.

Applications of Reduced Resource Computation

Loop perforation works for:

- evaluating forces on water molecules (summing numbers);
- Monte-Carlo simulation of swaption pricing;
- video encoding.

More on the video encoding example:
Changing loop increments from 4 to 8 gives:

- speedup of 1.67;
- signal-to-noise ratio decrease of 0.87%;
- bitrate increase of 18.47%;
- visually indistinguishable results.

Video Encoding Skeleton Code

```
min = DBL_MAX;
index = 0;
for (i = 0; i < m; i++) {
  sum = 0;
  for (j = 0; j < n; j++) sum += numbers[i][j];
  if (min < sum) {
    min = sum;
    index = i;
  }
}
```

The optimization changes the loop increments and then compensates.

Part II

Software Transactional Memory

STM: Introduction

Instead of programming with locks, we have transactions on memory.

- Analogous to database transactions

An old idea; recently saw some renewed interest.

A series of memory operations either all succeed; or
all fail (and get rolled back), and are later retried.

STM: Benefits

Simple programming model: need not worry about lock granularity or deadlocks.

Just group lines of code that should logically be one operation in an `atomic` block!

It is the responsibility of the implementer to ensure the code operates as an atomic transaction.

STM: Implementing a Motivating Example

```
transfer_funds(Account* sender, Account* receiver,
               double amount) {
  atomic {
    sender->funds -= amount;
    receiver->funds += amount;
  }
}
```

[Note: bank transfers aren't actually atomic!]

With locks we have two main options:

- Lock everything to do with modifying accounts
  (slow; may forget to use lock).
- Have a lock for every account
  (deadlocks; may forget to use lock).

With STM, we do not have to worry about remembering to acquire locks, or about deadlocks.

STM: Drawbacks

Rollback is key to STM.

But, some things cannot be rolled back.

(write to the screen, send packet over network)

Nested transactions.

What if an inner transaction succeeds, yet the transaction aborts?

Limited transaction size:

Most implementations (especially all-hardware) have a limited transaction size.

Basic STM Implementation (Software)

In all atomic blocks, record all reads/writes to a log.

At the end of the block, running thread verifies that no other threads have modified any values read.

If validation is successful, changes are **committed**. Otherwise, the block is **aborted** and re-executed.

Note: Hardware implementations exist too.

Basic STM Implementation Issues

Since you don't protect against dataraces (just rollback), a datarace may trigger a fatal error in your program.

```
atomic {
  x++;
  y++;
}
```

```
atomic {
  if (x != y)
    while (true) { }
}
```

In this silly example, assume initially `x = y`. You may think the code will not go into an infinite loop, but it can.

STM Implementations

Note: Typically STM performance is no worse than twice as slow as fine-grained locks.

- Toward.Boost.STM (C++)
- SXM (Microsoft, C#)
- Built-in to the language (Clojure, Haskell)
- AtomJava (Java)
- Durus (Python)

STM Summary

Software Transactional Memory provides a more natural approach to parallel programming:

no need to deal with locks and their associated problems.

Currently slow,

but a lot of research is going into improving it. (futile?)

Operates by either completing an atomic block,
or retrying (by rolling back) until it successfully completes.

Part III

High-Performance Languages

Introduction

DARPA began a supercomputing challenge in 2002.

Purpose:

create multi petaflop systems (floating point operations).

Three notable programming language proposals:

- X10 (IBM) [looks like Java]
- Chapel (Cray) [looks like Fortran/math]
- Fortress (Sun/Oracle)

Machine Model

We've used multicore machines and will talk about clusters (MPI, MapReduce).

These languages are targeted somewhere in the middle:

- thousands of cores and massive memory bandwidth;
- Partitioned Global Address Space (PGAS) memory model:
  - each process has a view of the global memory
  - memory is distributed across the nodes, but processors know what global memory is local.

Parallelism

These languages require you to specify the parallelism structure:

- Fortress evaluates loops and arguments in parallel by default;
- Others use an explicit construct, e.g. `forall`, `async`.

In terms of addressing the PGAS memory:

- Fortress divides memory into locations, which belong to regions (in a hierarchy; the closer, the better for communication).
- Similarly, places (X11) and locales (Chapel).

These languages make it easier to control the locality of data structures and to have distributed (fast) data.

X10 Example

```
import x10.io.Console;
import x10.util.Random;

class MontyPi {
  public static def main(args:Array[String](1)) {
    val N = Int.parse(args(0));
    val result=GlobalRef[Cell[Double]](new Cell[Double](0));
    finish for (p in Place.places()) at (p) async {
      val r = new Random();
      var myResult:Double = 0;
      for (1..(N/Place.MAX_PLACES)) {
        val x = r.nextDouble();
        val y = r.nextDouble();
        if (x*x + y*y <= 1) myResult++;
      }
      val ans = myResult;
      at (result) atomic result()() += ans;
    }
    val pi = 4*(result()())/N;
  }
}
```

X10 Example: explained

This solves Assignment 1, but distributes the computation.

Could replace `for (p in Place.places())` by
`for (1..P)` (where P is a number) for a parallel solution.
(Also, remove `GlobalRef`).

`async`: creates a new child activity,
which executes the statements.
`finish`: waits for all child `asyncs` to finish.
`at`: performs the statement at the place specified;
here, the processor holding the result increments its value.

HPC Language Summary

- Three notable supercomputing languages: X10, Chapel, and Fortress (end-of-life).
- Partitioned Global Adress Space memory model: allows distributed memory with explicit locality.
- Parallel programming aspect to the languages are very similar to everything else we've seen in the course.

Part IV

Overall Summary

Today's Lecture

Three topics:

- Reduced-Resource Computation.
- High-Performance Languages.
- Software Transactional Memory.

ECE students: Good luck on your FYDP!

Lecture 20—About OpenCL

ECE 459: Programming for Performance

March 26, 2013

Introduction

OpenCL: coding on a heterogeneous architecture.

- No longer just programming the CPU;
  will also leverage the GPU.

OpenCL = Open Computing Language.
Usable on both NVIDIA and AMD GPUs.

SIMT

Another term you may see vendors using:

- **S**ingle **I**nstruction, **M**ultiple **T**hreads.
- Runs on a vector of data.
- Similar to SIMD instructions (e.g. SSE).
  However, the vector is spread out over the GPU.

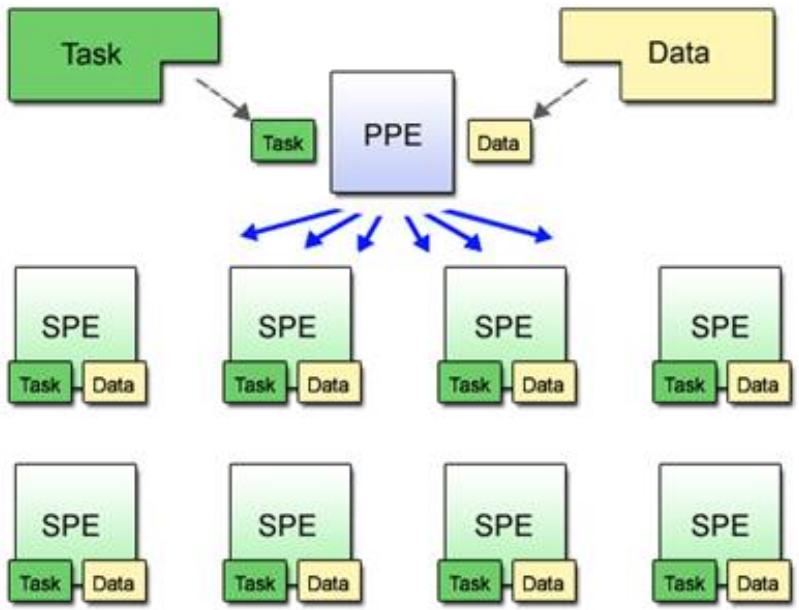
Other Heterogeneous Programming Examples

- PlayStation 3 Cell
- CUDA

(PS3) Cell Overview

Cell consists of:

- a PowerPC core; and
- 8 SIMD co-processors.



(from the Linux Cell documentation)

CUDA Overview

Compute Unified Device Architecture:
NVIDIA's architecture for processing on GPUs.

"C for CUDA" predates OpenCL,
NVIDIA supports it first and foremost.

- May be faster than OpenCL on NVIDIA hardware.
- API allows you to use (most) C++ features in CUDA; OpenCL has more restrictions.

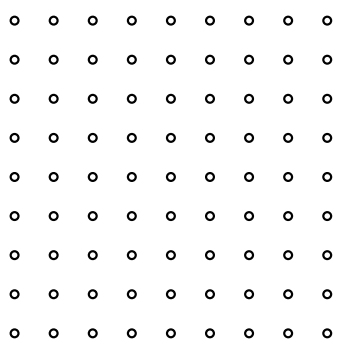
GPU Programming Model

The abstract model is simple:

- Write the code for the parallel computation (*kernel*) separately from main code.
- Transfer the data to the GPU co-processor (or execute it on the CPU).
- Wait ...
- Transfer the results back.

Data Parallelism

- Key idea: evaluate a function (or *kernel*) over a set of points (data).



Another example of data parallelism.

- Another name for the set of points: *index space*.
- Each point corresponds to a **work-item**.

Note: OpenCL also supports *task parallelism* (using different kernels), but documentation is sparse.

Work-Items

Work-item: the fundamental unit of work in OpenCL.
Stored in an *n*-dimensional grid (ND-Range); 2D above.

OpenCL spawns a bunch of threads to handle work-items.
When executing, the range is divided into **work-groups**, which execute on the same compute unit.

The set of compute units (or cores) is called something different depending on the manufacturer.

- NVIDIA - *warp*
- AMD/ATI - *wavefront*

Work-Items: Three more details

One thread per work item, each with a different thread ID.

You can say how to divide the ND-Range into work-groups, or the system can do it for you.

Scheduler assigns work-items to warps/wavefronts until no more left.

Shared Memory

There are many different types of memory available to you:

- private memory: available to a single work-item;
- local memory (aka "shared memory"): shared between work-items belonging to the same work-group; like a user-managed cache;
- global memory: shared between all work-items as well as the host;
- constant memory: resides on the GPU and cached. Does not change.

There is also host memory (normal memory); usually contains app data.

Example Kernel

Here's some traditional code to evaluate $C_i = A_i B_i$:

```
void traditional_mul(int n,
                     const float *a,
                     const float *b,
                     float *c) {
  int i;
  for (i = 0; i < n; i++) c[i] = a[i] * b[i];
}
```

And as a kernel:

```
kernel void opencl_mul(global const float *a,
                       global const float *b,
                       global float *c) {
  int id = get_global_id(0);  // dimension 0
  c[id] = a[id] * b[id];
}
```

Restrictions when writing kernels in OpenCL

It's mostly C, but:

- No function pointers.
- No bit-fields.
- No variable-length arrays.
- No recursion.
- No standard headers.

OpenCL's Additions to C in Kernels

In kernels, you can also use:

- Work-items.
- Work-groups.
- Vectors.
- Synchronization.
- Declarations of memory type.
- Kernel-specific library.

Branches in kernels

Kernels contain code, which can contain branches (if statements). Hence, computation from each work-item can branch arbitrarily. The hardware will execute *all* branches that any thread in a warp executes—can be slow!

In other words: an `if` statement will cause each thread to execute both branches; we keep only the result of the taken branch.

A loop will cause the workgroup to wait for the maximum number of iterations of the loop in any work-item.

Note: when you set up work-groups, best to arrange for all work-items in a workgroup to execute the same branches.

Synchronization

Different workgroups execute independently.
You can only put barriers and memory fences between work-items in the same workgroup.

OpenCL supports:

- Memory fences (load and store).
- Barriers.
- `volatile` (beware!)

Summary

Brief overview of OpenCL and its programming model.

Many concepts are similar to plain parallel programming (more structure).

Lecture 21—Programming with OpenCL

ECE 459: Programming for Performance

March 26, 2013

Introduction

Today, we'll see how to program with OpenCL.

- We're using OpenCL 1.1.
- There is a lot of initialization and querying.
- When you compile your program, include `-lOpenCL`.

You can find the official documentation here:

`http://www.khronos.org/opencl/`

More specifically:

`http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/`

Let's just dive into an example.

First, reminders

- All data belongs to an **NDRange**.
- The range can be divided into **work-groups**. (in software)
- The work-groups run on **wavefronts/warps**. (in hardware)
- Each wavefront/warp executes **work-items**.

All branches in a wavefront/warp should execute the same path.

If an iteration of a loop takes `t`:
when one work-item executes 100 iterations,
the total time to complete the wavefront/warp is `100t`.

Part I

Simple Example

Simple Example (1)

```
#include <CL/cl.h>
#include <stdio.h>

#define NWITEMS 512

// A simple memset kernel
const char *source =
"__kernel void memset( __global uint *dst )                    \n"
"{                                                             \n"
"    dst[get_global_id(0)] = get_global_id(0);                 \n"
"}                                                             \n";

int main(int argc, char ** argv)
{
    // 1. Get a platform.
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, NULL);
```

Explanation (1)

Include the OpenCL header.

Request a platform (also known as a host).

A platform contains *compute devices*:

- GPUs or CPUs.

Simple Example (2)

```
    // 2. Find a gpu device.
    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
                   1,
                   &device,
                   NULL);

    // 3. Create a context and command queue on that device.
    cl_context context = clCreateContext(NULL,
                                         1,
                                         &device,
                                         NULL, NULL, NULL);
    cl_command_queue queue = clCreateCommandQueue(context,
                                                  device,
                                                  0, NULL);
```

Explanation (2)

Request a GPU device.

Request a OpenCL context (representing all of OpenCL's state).

Create a command-queue:
get OpenCL to do work by telling it to run a kernel in a queue.

Simple Example (3)

```
// 4. Perform runtime source compilation, and obtain
//    kernel entry point.
cl_program program = clCreateProgramWithSource(context,
                                               1,
                                               &source,
                                               NULL,
                                               NULL);
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "memset",
                                  NULL);

// 5. Create a data buffer.
cl_mem buffer = clCreateBuffer(context,
                               CL_MEM_WRITE_ONLY,
                               NWITEMS * sizeof(cl_uint),
                               NULL, NULL);
```

Explanation (3)

We create an OpenCL "program" (runs on the compute unit):

- kernels;
- functions; and
- declarations.

In this case, we create a kernel called `memset` from `source`.
OpenCL may also create programs from binaries
(may be in intermediate representation).

Next, we need a *data buffer* (enables inter-device communication).

This program does not have any input,
so we don't put anything into the buffer (just declare its size).

Simple Example (4)

```
// 6. Launch the kernel. Let OpenCL pick the local work
//    size.
size_t global_work_size = NWITEMS;
clSetKernelArg(kernel,0, sizeof(buffer), (void*)&buffer);
clEnqueueNDRangeKernel(queue,
                       kernel,
                       1,    // dimensions
                       NULL, // initial offsets
                       &global_work_size, // number of
                                          // work-items
                       NULL, // work-items per work-group
                       0, NULL, NULL);  // events
clFinish(queue);

// 7. Look at the results via synchronous buffer map.
cl_uint *ptr;
ptr = (cl_uint *)clEnqueueMapBuffer(queue, buffer,
                                    CL_TRUE, CL_MAP_READ,
                                    0, NWITEMS *
                                    sizeof(cl_uint),
                                    0, NULL, NULL, NULL);
```

Explanation (4)

Set kernel arguments to `buffer`.

We launch the kernel, enqueuing the 1-dimensional index space starting at 0.

We specify that the index space has `NWITEMS` elements; and not to subdivide the program into work-groups.

There is also an event interface, which we do not use.

We copy the results back; call is blocking (`CL_TRUE`);
hence we don't need an explicit `clFinish()` call.

We specify that we want to read the results back into buffer.

Simple Example (5)

```
    int i;
    for(i=0; i < NWITEMS; i++)
        printf("%d %d\n", i, ptr[i]);
    return 0;
}
```

- The program simply prints `0 0`, `1 1`, ..., `511 511`.
- Note: I didn't clean up or include error handling for any of the OpenCL functions.

Part II

Another Example

C++ Bindings

If we use the C++ bindings, we'll get automatic resource release and exceptions.

- C++ likes to use the RAII style (resource allocation is initialization).

Change the header to `CL/cl.hpp` and define `__CL_ENABLE_EXCEPTIONS`.

We'd also like to store our kernel in a file instead of a string.

The C API is not so nice to work with.

Vector Addition Kernel

Let's write a kernel that adds two vectors and stores the result.
This kernel will go in the file `vector_add_kernel.cl`.

```
__kernel void vector_add(__global const int *A,
                         __global const int *B,
                         __global int *C) {

    // Get the index of the current element to be processed
    int i = get_global_id(0);

    // Do the operation
    C[i] = A[i] + B[i];
}
```

Other possible qualifiers: `local`, `constant` and `private`.

Vector Addition (1)

```
#define __CL_ENABLE_EXCEPTIONS

#include <CL/cl.hpp>

#include <iostream>
#include <fstream>
#include <string>
#include <utility>
#include <vector>

int main() {
    // Create the two input vectors
    const int LIST_SIZE = 1000;
    int *A = new int[LIST_SIZE];
    int *B = new int[LIST_SIZE];
    for(int i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }
```

Vector Addition (2)

```
    try {
        // Get available platforms
        std::vector<cl::Platform> platforms;
        cl::Platform::get(&platforms);

        // Select the default platform and create a context
        // using this platform and the GPU
        cl_context_properties cps[3] = {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[0])(),
            0
        };
        cl::Context context(CL_DEVICE_TYPE_GPU, cps);

        // Get a list of devices on this platform
        std::vector<cl::Device> devices =
            context.getInfo<CL_CONTEXT_DEVICES>();

        // Create a command queue and use the first device
        cl::CommandQueue queue = cl::CommandQueue(context,
            devices[0]);
```

Explanation (2)

You can define `__NO_STD_VECTOR` and use `cl::vector` (same with strings).

You can enable profiling by adding `CL_QUEUE_PROFILING_ENABLE` as 3rd argument to `queue`.

Vector Addition (3)

```
        // Read source file
        std::ifstream sourceFile("vector_add_kernel.cl");
        std::string sourceCode(
            std::istreambuf_iterator<char>(sourceFile),
            (std::istreambuf_iterator<char>())
        );
        cl::Program::Sources source(
            1,
            std::make_pair(sourceCode.c_str(),
                           sourceCode.length()+1)
        );

        // Make program of the source code in the context
        cl::Program program = cl::Program(context, source);

        // Build program for these specific devices
        program.build(devices);

        // Make kernel
        cl::Kernel kernel(program, "vector_add");
```

Vector Addition (4)

```
// Create memory buffers
cl::Buffer bufferA = cl::Buffer(
    context,
    CL_MEM_READ_ONLY,
    LIST_SIZE * sizeof(int)
);
cl::Buffer bufferB = cl::Buffer(
    context,
    CL_MEM_READ_ONLY,
    LIST_SIZE * sizeof(int)
);
cl::Buffer bufferC = cl::Buffer(
    context,
    CL_MEM_WRITE_ONLY,
    LIST_SIZE * sizeof(int)
);
```

Vector Addition (5)

```
        // Copy lists A and B to the memory buffers
        queue.enqueueWriteBuffer(
            bufferA,
            CL_TRUE,
            0,
            LIST_SIZE * sizeof(int),
            A
        );
        queue.enqueueWriteBuffer(
            bufferB,
            CL_TRUE,
            0,
            LIST_SIZE * sizeof(int),
            B
        );

        // Set arguments to kernel
        kernel.setArg(0, bufferA);
        kernel.setArg(1, bufferB);
        kernel.setArg(2, bufferC);
```

Explanation (5)

`enqueue*Buffer` arguments:

- *buffer*
- `cl_ bool` *blocking_write*
- `::size_t` *offset*
- `::size_t` *size*
- `const void *` *ptr*

Vector Addition (6)

```
        // Run the kernel on specific ND range
        cl::NDRange global(LIST_SIZE);
        cl::NDRange local(1);
        queue.enqueueNDRangeKernel(
            kernel,
            cl::NullRange,
            global,
            local
        );

        // Read buffer C into a local list
        int* C = new int[LIST_SIZE];
        queue.enqueueReadBuffer(
             bufferC,
             CL_TRUE,
             0,
             LIST_SIZE * sizeof(int),
             C
        );
```

Vector Addition (7)

```
        for(int i = 0; i < LIST_SIZE; i ++) {
            std::cout << A[i] << " + " << B[i] << " = "
                      << C[i] << std::endl;
        }
    } catch(cl::Error error) {
        std::cout << error.what() << "(" << error.err()
                  << ")" << std::endl;
    }

    return 0;
}
```

This program just prints all the additions (equalling 1000).

Other Improvements

The host memory is still unreleased.

- With the same number of lines, we could use the C++11 `unique_ptr`, which would free the memory for us.

You can use a vector instead of an array,
and use `&v[0]` instead of `<type>*`.

- Valid as long as the vector is not resized.

Summary

Went through real OpenCL examples.
Have the reference card for the AP.

Saw a C++ template for setting up OpenCL.

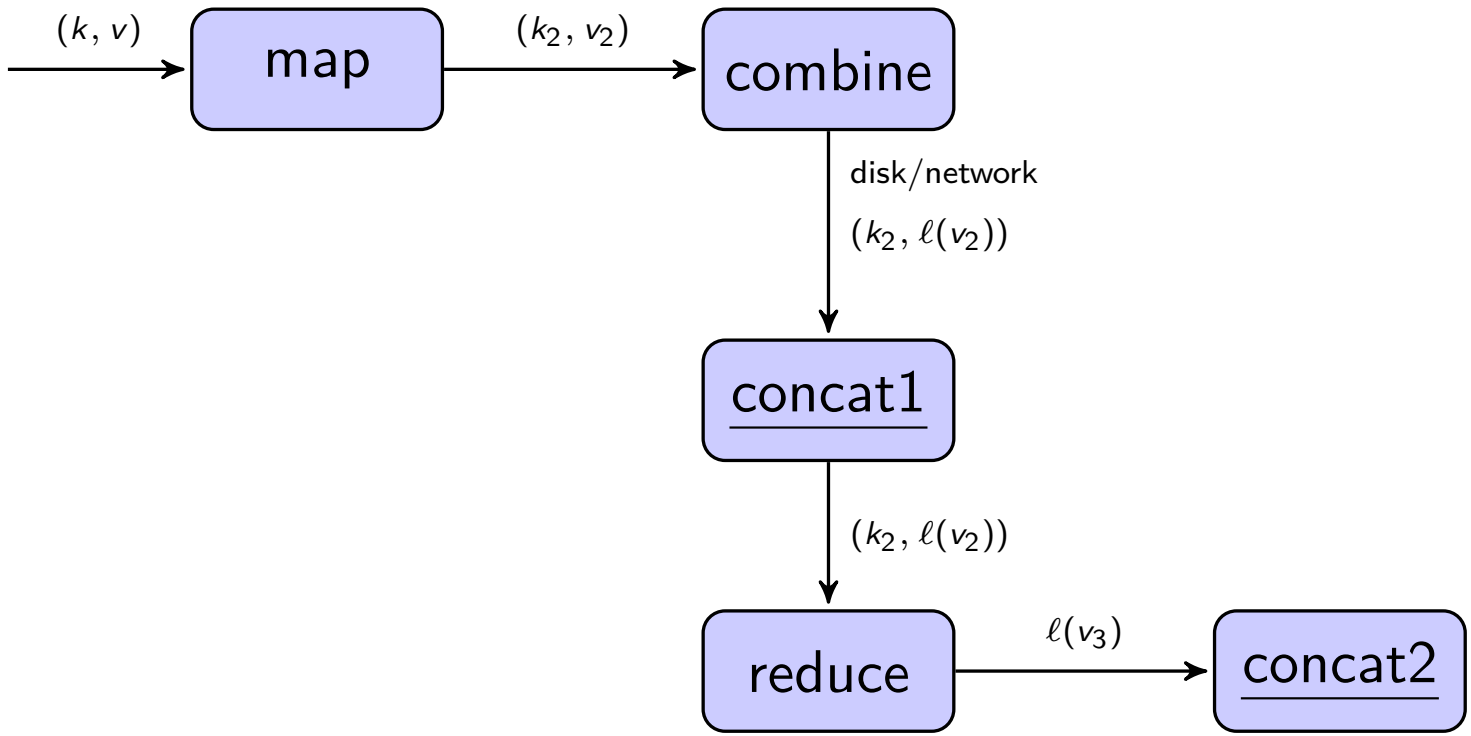
Aside: if you're serious about programming in C++, check out **Effective C++** by Scott Meyers (slightly dated with C++11, but it still has some good stuff)

Lecture 22—MapReduce/Hadoop

ECE 459: Programming for Performance

March 28, 2013

MapReduce


(k, v)
map
(k_2, v_2)
combine
disk/network
$(k_2, \ell(v_2))$
concat1
$(k_2, \ell(v_2))$
reduce
$\ell(v_3)$
concat2

Introduction: MapReduce

Framework introduced by Google for large problems.
Consists of two functional operations: map and reduce.

```
>>> map(lambda x: x*x, [1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]
```

- **map**: applies a function to an iterable data-set.

```
>>> reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
15
```

- **reduce**: applies a function to an iterable data-set cumulatively.
  ((((1+2)+3)+4)+5) in this example.

MapReduce Intuition

In functional languages, the functions are "pure" (no side-effects). Since they are pure, and hence independent, it's always safe to parallelize them.

Note: functional languages, like Haskell, have their own parallel frameworks, which allow easy parallelization.

Many problems can be represented as a map operation followed by a reduce (for example, Assignment 1).

Hadoop

Apache Hadoop is a framework which implements MapReduce.

- The most widely used open source framework,
  used by Amazon's EC2 (elastic compute cloud).
- Allows work to be distributed across many different nodes
  (or re-tried if a node goes down).
- Includes HDFS (Hadoop distributed file system):
  distributes data across nodes and provides failure handling.
  (You can also use Amazon's S3 storage service).

Map (massive parallelism)

You split the input file into multiple pieces.

The pieces are then processed as (key, value) pairs.

Your **Mapper** function uses these (key, value) pairs and outputs another set of (key, value) pairs.

Reduce (not as parallel)

Collects the input files from the previous map
(which may be on different nodes, needing copying).

Merge-sorts the files,
so that the key-value pairs for a given key are contiguous.

Reads the file sequentially and
splits the values into lists of values with the same key.

Passes this data, consisting of keys and lists of values,
to your **reduce** method (in parallel), & concatenates results.

Combine (optional)

This step may be run right after map and before reduce.

Takes advantage of the fact that elements produced by the map operation are still available in memory.

Every so many elements, you can use your combine operation to take (key, value) outputs of the map and create new (key, value) inputs of the same types.

WordCount Example

Say we want to count the number of occurrences of words in some files.

Consider, for example, the following files:

```
Hello World Bye World
```

```
Hello Hadoop Goodbye Hadoop
```

We want the following output:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

WordCount: Example Operations

Mapper

- Split the input file into strings, representing words.
- For each word, output the following (key, value) pair: **(word, 1)**

Reduce

- Sum all values for each word (key) and output: **(word, sum)**

Combine

- We could do the reduce step for in-memory values while doing map.

  Note: here, the output of map and input/output of reduce are the same, but they don't have to be.

WordCount: Running the Example (File 1)

```
Hello World Bye World
```

After map:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
```

After combine:

```
(Hello, 1)
(Bye, 1)
(World, 2)
```

WordCount: Running the Example (File 2)

```
Hello Hadoop Goodbye Hadoop
```

After map:

```
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

After combine:

```
(Hello, 1)
(Goodbye, 1)
(Hadoop, 2)
```

WordCount: Running the Example (Reduce)

After concatenation, sorting, and creating lists of values:

```
(Bye, [1])
(Goodbye, [1])
(Hadoop, [2])
(Hello, [1, 1])
(World, [2])
```

After the reduce, we get what we want:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

WordCount Example: C++ Code (1)

- APIs for Java/Python also exist.

```
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue()," ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};
```

WordCount Example C++ Code (2)

```
class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(),
                 HadoopUtils::toString(sum));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(
    HadoopPipes::TemplateFactory<WordCountMap,
                                 WordCountReduce>());
}
```

Other Examples

- Distributed Grep.
- Count of URL Access Frequency.
- Reverse Web-Link Graph.
- Term-Vector per Host.
- Inverted Index:
  - **Map:** parses each document, and emits a sequence of (word, document ID) pairs.
  - **Reduce:** accepts all pairs for a given word, sorts the corresponding document IDs, and emits a (word, list(document ID)) pair.
  - **Output:** all of the output pairs from reducing form a simple inverted index.

Other Notes

Hive builds on top of Hadoop and allows you to use an SQL-like language to query outputs on HDFS; or you can provide custom mappers/reducers to get more information.

The cloud is a great way to start a new project: you can add or remove nodes easily as your problem changes size. (Hadoop or MPI are good examples).

References:

`http://wiki.apache.org/hadoop/`

`http://code.google.com/edu/parallel/mapreduce-tutorial.html`

Summary

MapReduce is an excellent framework
for dealing with massive data-sets.

Hadoop is a common implementation you can use
(on most cloud computing services, even!)

You just need 2 functions (optionally 3):
mapper, reducer and combiner.

Just remember:
output of the mapper/combiner is input to the reducer.

Lecture 23—Clusters & Message Passing Interface (MPI); The Cloud

ECE 459: Programming for Performance

April 2, 2013

Part I

Clusters

Switching Gears

So far, we've seen how to make things fast on one computer:

- threads;
- compiler optimizations;
- GPUs.

To get a lot of bandwidth, though, you need lots of computers,
(if you're lucky and the problem allows!)

Today: programming for performance using multiple computers.

Key Idea: Explicit Communication

Mostly we've seen shared-memory systems;
complication: must manage contention.

Last week, GPU programming: explicitly copy data.

Message-passing: yet another paradigm.

What is MPI?

Message Passing Interface:
A language-independent communation protocol
for parallel computers.

- Use it to run the same code on a number of nodes
  (different hardware threads; or servers in a cluster).
- Provides explicit message passing between nodes.
- Is the dominant model for high performance computing
  (de-facto standard).

High Level View of MPI

MPI is a type of SPMD (single process, multiple data).

Idea: have multiple instances of the same program,
all working on different data.

The program could be running on the same machine,
or a cluster of machines.

MPI facilitates communication of data between processes.

MPI Functions

```
// Initialize MPI
int MPI_Init(int *argc, char **argv)

// Determine number of processes within a communicator
int MPI_Comm_size(MPI_Comm comm, int *size)

// Determine processor rank within a communicator
int MPI_Comm_rank(MPI_Comm comm, int *rank)

// Exit MPI (must be called last by all processors)
int MPI_Finalize()

// Send a message
int MPI_Send (void *buf,int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)

// Receive a message
int MPI_Recv (void *buf,int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

MPI Function Notes

- `MPI_Comm`: a communicator,
  often `MPI_COMM_WORLD` for global channel.
- `MPI_Datatype`: just an enum, e.g. `MPI_FLOAT_INT`, etc.

- `dest`/`source`: "rank" of the process (in a communicator) to send a message to/receive a message from;
  you may use `MPI_ANY_SOURCE` in `MPI_Recv`.

- Both `MPI_Send` and `MPI_Recv` are blocking calls—
  see `man MPI_Send` or `man MPI_Recv` for more details.
- The `tag` allows you to organize your messages,
  so you can filter all but a specific tag.

Hello, World in MPI

As with OpenCL kernels:

first, figure out what "current" process is supposed to compute.

```
// http://www.dartmouth.edu/~rc/classes/intro_mpi/
#include <stdio.h>
#include <mpi.h>

int main (int argc, char * argv[])
{
  int rank, size;

  /* start MPI */
  MPI_Init (&argc, &argv);
  /* get current process id */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  /* get number of processes */
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  printf("Hello world from process %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Longer MPI Example (from Wikipedia)

Here's a common example:

- The "master" (rank 0) process creates some strings and sends them to the worker processes.
- The worker processes modify their string and send it back to the master.

Example Code (1)

```
/*
  "Hello World" MPI Test Program
 */
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;
```

Example Code (2)

```
/* all MPI programs start with MPI_Init; all 'N'
 * processes exist thereafter
 */
MPI_Init(&argc,&argv);

/* find out how big the SPMD world is */
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

/* and this processes' rank is what? */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

/* At this point, all programs are running equivalently;
 * the rank distinguishes the roles of the programs in
 * the SPMD model, with rank 0 often used specially...
 */
```

Example Code (3)

```
if(myid == 0)
{
  printf("%d: We have %d processors\n", myid, numprocs);
  for(i=1; i<numprocs; i++)
  {
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
             MPI_COMM_WORLD);
  }
  for(i=1; i<numprocs; i++)
  {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
             MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
  }
}
```

Example Code (4)

```
    else
    {
      /* receive from rank 0: */
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG,
               MPI_COMM_WORLD, &stat);
      sprintf(idstr, "Processor %d ", myid);
      strncat(buff, idstr, BUFSIZE-1);
      strncat(buff, "reporting for duty", BUFSIZE-1);
      /* send to rank 0: */
      MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG,
               MPI_COMM_WORLD);
    }

    /* MPI Programs end with MPI Finalize; this is a weak
     * synchronization point.
     */
    MPI_Finalize();
    return 0;
}
```

Compiling with OpenMPI

```
// Wrappers for gcc (C/C++)
mpicc
mpicxx

// Compiler Flags
OMPI_MPICC_CFLAGS
OMPI_MPICXX_CXXFLAGS

// Linker Flags
OMPI_MPICC_LDFLAGS
OMPI_MPICXX_LDFLAGS
```

OpenMPI does not recommend you to set the flags yourself.
To see them, try:

```
# Show the flags necessary to compile MPI C applications
shell$ mpicc --showme:compile

# Show the flags necessary to link MPI C applications
shell$ mpicc --showme:link
```

Compiling and Running

```
mpirun -np <num_processors> <program>
mpiexec -np <num_processors> <program> # a synonym
```

Starts `num_processors` instances of the program using MPI.

```
jon@riker examples master % mpicc hello_mpi.c
jon@riker examples master % mpirun -np 8 a.out
0: We have 8 processors
0: Hello 1! Processor 1 reporting for duty
0: Hello 2! Processor 2 reporting for duty
0: Hello 3! Processor 3 reporting for duty
0: Hello 4! Processor 4 reporting for duty
0: Hello 5! Processor 5 reporting for duty
0: Hello 6! Processor 6 reporting for duty
0: Hello 7! Processor 7 reporting for duty
```

- By default, MPI uses the lowest-latency communication resource available; shared memory, in this case.

MPI Matrix Multiplication Example

Highlights of: `http://www.nccs.gov/wp-content/training/mpi-examples/C/matmul.c`.

To compute the matrix product AB:

1. Initialize MPI.
2. If the current process is the master task (task id 0):
   1. Initialize matrices.
   2. Send work to each worker task:
      row number (offset); number of rows; row contents from A; complete contents of matrix B.

      ```
      MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest,
               mtype, MPI_COMM_WORLD);
      ```
   3. Wait for results from all worker tasks (`MPI_Recv`).
   4. Print results.
3. For all other tasks:
   1. Receive offset, number of rows, partial matrix A, and complete matrix B, using `MPI_Recv`:

      ```
      MPI_Recv(&offset, 1, MPI_INT, MASTER,
               mtype, MPI_COMM_WORLD, &status);
      ```
   2. Do the computation.
   3. Send the results back to the sender.

Other Things MPI Can Do

We can use nodes on a network (by using a `hostfile`).

We can even use MPMD:

- *multiple processes, multiple data*

```
% mpirun -np 2 a.out : -np 2 b.out
```

This launches a single parallel application.

- All in the same `MPI_COMM_WORLD`; but
- Ranks 0 and 1 are instances of `a.out`, and
- Ranks 2 and 3 are instances of `b.out`.

You could also use the `-app` flag with an appfile instead of typing out everything.

Performance Considerations

Your bottleneck for performance here is message-passing.

Keep the communication to a minimum!

In general, the more machines/farther apart they are, the slower the communication.

Each step from multicore machines to GPU programming to MPI triggers an order-of-magnitude decrease in communication bandwidth and similar increase in latency.

Part II

Cloud Computing

Using a Cluster

Historically:

- find $$$;
- buy and maintain pile of expensive machines.

Not anymore!

We'll talk about Amazon's Elastic Compute Cloud (EC2) and principles behind it.

Evolution of servers

You want a server on the Internet.

- Once upon a time: had to get a physical machine hosted (e.g. in a rack).
  Or, live with inferior shared hosting.

- Virtualization: pay for part of a machine on that rack.
  A win: you're usually not maxing out a computer, and you'd be perfectly happy to share it with others, as long as there are good security guarantees. All users can get root access.

- Clouds enable you to add more machines on-demand.
  Instead of having just one virtual server, spin up dozens (or thousands) of server images when you need more compute capacity.
  Servers typically share persistent storage, also in the cloud.

Paying for Computes

Cloud computing:

- pay according to the number of machines, or instances, that you've started up.

Providers offer different instance sizes;
sizes vary according to the number of cores, local storage, and memory.

Some instances even have GPUs!

Launching Instances

Need more computes? Launch an instance!

Input: Virtual Machine image.

Mechanics: use a command-line or web-based tool.

New instance gets an IP address and is network-accessible.
You have full root access to that instance.

What to Launch?

Amazon provides public images:

- different Linux distributions;
- Windows Server; and
- OpenSolaris (maybe not anymore?).

You can build an image which contains software you want, including Hadoop and OpenMPI.

Cleanup

Presumably you don't want to pay forever for your instances.

When you're done with an instance:

- shut it down, stop paying for it.

All data on instance goes away.

Data Storage

To keep persistent results:

- mount a storage device, also on the cloud (e.g. Amazon Elastic Block Storage); or,
- connect to a database on a persistent server (e.g. Amazon SimpleDB or Relational Database Service); or,
- you can store files on the Web (e.g. Amazon S3).

Summary

- MPI is a powerful tool for highly parallel computing across multiple machines.
- MPI Programming is similar to a more powerful version of `fork/join`; but you have to manage communication more explicitly.
- Saw cloud computing basics.

Lecture 24—Massive Scalability; Course Summary

ECE 459: Programming for Performance

April 4, 2013

Part I

Massive Scalability

People worth following

- Fran Allen (IBM)
- Jeff Dean (Google)
- Keith Packard (Intel)
- Herb Sutter (Microsoft)

Some Thoughts from Jeff Dean

URL: `research.google.com/pubs/jeff.html`

Selections from:
"Software Engineering Advice for Building Large-Scale Distributed Systems".

On scaling:

- design for $\sim$ 10x, rewrite before 100x

Key concept for scaling:

- sharding, also known as partitioning.

Why Distribute?

Let's say that we want a copy of the Web.

20+ billion web pages $\times$ 20KB = 400+ TB.

$\sim$ 3 months to read the web.

$\sim$ 1000 HDs (in 2010) to store the web.

And that's without even processing the data!

Magic solution: distribute the problem!

1000 machines $\Rightarrow$ < 3 hours.

No Free Lunch.
Problems: need to deal with ...

- communication & coordination;
- recovering from machine failure;
- status reporting;
- debugging;
- optimization;
- locality

...and that, from scratch, for each problem!

Designing Systems as Services/Platforms

Steve Yegge on Google and Platforms:

`https://plus.google.com/112678702228711889851/posts/eVeouesvaVX`

[Bezos's] Big Mandate went something along these lines:
1) All teams will henceforth expose their data and functionality through service interfaces.
2) Teams must communicate with each other through these interfaces.
3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4) It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter. Bezos doesn't care.
5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6) Anyone who doesn't do this will be fired.
7) Thank you; have a nice day! [j/k]

Why Services?

Decouple different parts of a system.

“Fewer dependencies, clearly specified”.

“Easy to test new versions.”

“Small teams can work independently.”

How to Design Stuff

Talk to people!

Write down a [some] rough sketch[es],
chat at a whiteboard.

Design good interfaces. (This is hard.)

Using Back-of-the-Envelope Calculations I

"How long to generate image results page (30 thumbnails)?"

1. read serially, thumbnail 256K images on-the-fly:
   30 seeks $\times$ 10 ms/seek + 30 $\times$ 256K / 30MB/s = 560ms
2. issue reads in parallel:
   10 ms/seek + 256KB read / 30 MB/s = 18ms
   (plus variance: 30-60ms)

Using Back-of-the-Envelope Calculations II

"How long to quicksort 1GB of 2-byte numbers?"

Comparisons: lots of branch mispredicts.
log(2^{28}) passes over 2^{28} numbers = $\sim 2^{33}$ compares
$\sim 50\%$ mispredicts = 2^{32} mispredicts $\times$ 5 ns/mispredict = 21s.

Memory bandwidth: mostly sequential streaming.
2^{30} bytes $\times$ 28 passes = 28GB;
memory bandwidth $\sim 4GB/s$, so ~ 7 seconds.

Total: about 30 seconds to sort 1GB on 1 CPU.

Also, write microbenchmarks. Understand your building blocks.

Numbers Everyone Should Know

`http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

Part II

Course Summary

Key Concepts I: goals

- Bandwidth versus Latency.
- Concurrency versus Parallelism.
- More bandwidth through parallelism.
- Amdahl's Law and Gustafson's Law.

Key Concepts II: leveraging parallelism

- Features of modern hardware.
- Parallelism implementations: pthreads.
  - definition of a thread;
  - spawning threads.
- Problems with parallelism: race conditions.
  - manual solutions: mutexes, spinlocks, RW locks, semaphores, barriers.
  - lock granuarlity.
- Parallelization patterns; also SIMD.

Key Concepts III: inherently-sequential problems

- Barriers to parallelization: dependencies.
  - loop-carried, memory-carried;
  - RAW/WAR/WAW/RAR.
- Breaking dependencies with speculation.

Key Concepts IV: higher-level parallelization

- Automatic parallelization; when does it work?
- Language/library support through OpenMP.

Key Concepts V: hardware considerations

- Unwelcome surprises: memory models & reordering.
  - fences and barriers; atomic instructions.
  - cache coherency implementations.

Key Concepts VI: help from the compiler

- Three-address code.
- Compiler constructs: volatile, restrict.
- Inlining and other static optimizations.
- Profile-guided optimizations.

Key Concepts VII: profiling

- Profiling tools and techniques.
- Call graphs, performance counters from profilers.
- When your profiler lies to you!
- Query-based DTrace approach.

Key Concepts VIII: assorted topics

- Reduced-resource computing.
- Software transactions.

Key Concepts IX: beyond single-core CPU programming

- Languages for high-performance computing.
- GPU Programming (e.g. with OpenCL).
- Clusters: MapReduce, MPI.
- Clouds.
- Big Data.

Final Words

Thanks for coming all term at 8:30AM!

Good luck on the final & see you at convocation!