Tutorial: Assignment 3, Optimization

Natalie Perna
pernanm@mcmaster.ca

Department of Computing and Software
McMaster University

February 23-24, 2016

Outline

Assignment

Assignment

Review specification and code.

Compiler

Compiler

If your native C compiler doesn't support OpenMP by default (if the provided code fails to compile with `make`), check this list of compatible compilers and versions:
`http://openmp.org/wp/openmp-compilers/`.
On OS X, this may mean:

```
brew install homebrew/versions/gcc49
```

then:

```
make CC=gcc-4.9
```

Makefiles

A Note about makefiles

Assignment 3 has specific instructions about how your makefile should work. It may be a little tricky if you are not yet comfortable customizing makefiles. Make sure to leave yourself time to write the makefile or request help, if needed. Marks will be deducted otherwise.

Performance Optimization

Profiling

The **best** way to improve serial performance is through profiling to identify bottlenecks/hotspots.
Very small improvements in "hotspots" have disproportionately (enormous) benefits.
Knowing where your hotspots are is essential to optimizing serial code.
"Make the common case fast and the rare case correct."

Profilers

Time based sampling method:

- usually easiest to execute and understand
- small overhead to collect statistical data about work performed by application
- "where" in the code is executing at each sampling interval

Try `gprof` on Linux or Instruments on OS X (comes packaged with Xcode).

Experiment

Make small changes, then test.
Tests should be a *minimum* of 10 seconds, though preferably longer, to ensure your change made an improvement. Repeat control and test at least *twice*.

Conditionals

Branching is expensive.

Consider: `x = 0;`

Conditionals

Branching is expensive.

Consider: `x = 0;`

Can we save time by not setting `x` if it's already zero?

```
if (x != 0)x = 0;
```

Conditionals

Branching is expensive.

Consider: `x = 0;`

Can we save time by not setting `x` if it's already zero?

```
if (x != 0)x = 0;
```

No, the conditional testing whether `x == 0` takes as much time as just setting it would have.

Function inlining

Function calls are pretty cheap, but if small functions are called frequently, macros/inlining can reduce overhead.
Warning: Inlining "everything" with increase memory usage and can do more harm than good. Inline according to the results of your profiler, and always test.

Function inlining

Reduce the amount of branching and number of conditional checks.

```
// OLD
for (i = 0; i < 100; i++)
{
    do_stuff(i);
}
```

Function inlining

Reduce the amount of branching and number of conditional checks.

```
// OLD
for (i = 0; i < 100; i++)
{
    do_stuff(i);
}

// NEW
for (i = 0; i < 100; )
{
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    ...
}
```

Again, unroll where there are bottlenecks, not everywhere. Always test.

Loop Jamming

Be on the lookout for these types of opportunities.

```
// Initialize 2D array to 0's with 1's along diagonal
// OLD
for (i = 0; i < MAX; i++)
  for (j = 0; j < MAX; j++)
      a[i][j] = 0.0;
for (i = 0; i < MAX; i++)
  a[i][i] = 1.0;

// NEW
for (i = 0; i < MAX; i++)
{
  for (j = 0; j < MAX; j++)
      a[i][j] = 0.0;
  a[i][i] = 1.0;
}
```

Again, unroll where there are bottlenecks, not everywhere. Always test.

Loop Inversion

With some loops, direction doesn't matter, and comparisons to 0 are often extra-cheap thanks to special machine instructions.

```
// OLD
for (i = 1; i <= MAX; i++)
{
   ...
}

// NEW
i = MAX+1;
while (--i)
{
   ...
}
```

Be careful that you don't do this at the cost of caching benefits! Test.

Variable Scope

Minimize variable scope wherever you can!* Not only is it good coding practice, it leaves room for the compiler to make smarter optimizations for you.
Global and static variables can be surprisingly costly, especially when referred to in bottleneck loops.
*Exception: Pointers for which memory is allocated often benefit from scope just large enough that they need not be deallocated and reallocated with each loop iteration.

For more detail...

`http://leto.net/docs/C-optimization.php`
This is a brief overview, there are *many* ways to improve performance of C code. Research!

- Reduce the need stack usage for variables and function parameters (favouring register usage).
- Pass by reference, not value
- Does your function need a return value?
- Shift operations over integer arithmetic!
- In which contexts is + or += better?