# Lecture 29 - Performance Tweaks

## ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

March 28, 2012

# Introduction

- We'll look into the improvements people made in Assignment 3
- Improvements we could make to Assignment 4
- Some other examples of doing less work

# Baseline Performance

- `distance` took the majority of the time, followed by `crossover` then `selection`

- The hash table's `at` function took a silly amount of time, followed by `find` in crossover

# Hashtables vs. Arrays/Vectors

- You shouldn't assume a hashtable is always fast for every data type
- There is a lot of overhead to hash an `int` to another number then access that number indirectly, in this case a simple 2 dimensional array/vector is much better
- We lose some generality since our indexs can no longer be more complex things like `strings`, if needed though, we could convert between numbered indexes and strings
- This alone should account for about a 2.7x speedup

# Change 1

```
 double GATSP::distance(const tour_container& tour)
 {
-    auto i = tour.begin();
-    double distance = distances[first_index][*i];
-    while (i != tour.end()) {
-        auto& distances_i = distances.at(*i);
-        ++i;
-        if (i != tour.end()) {
-            distance += distances_i.at(*i);
-        }
-        else {
-            distance += distances_i.at(first_index);
-        }
+    tour_container::size_type n = tour.size();
+    double distance = distances[first_index][tour[0]];
+    for (tour_container::size_type i = 0; i < (n-1); ++i) {
+        distance += distances[tour[i]][tour[i + 1]];
     }
+    distance += distances[tour[n-1]][first_index];
     return distance;
 }
```

# Removing the find

- The `find` in crossover checks that the element from the second parent was already added in the child, we can remove this
- Instead of going and searching the sub-tour each time, make a lookup array which has `true` for every index in the sub-tour and `false` otherwise
- Replace the find with an access to this array
- This is about a 1.55 speedup over the last change (overall 4.19x)

# Change 2

```
+     std::vector<bool> lookup(a.tour.size(), false);
+     for (auto i = child_copy_begin; i < child_copy_end;
           ++i) {
+         lookup[*i] = true;
+     }
+
      auto i = child.tour.begin();
      for (auto& index : b.tour) {
          /* Search to see if this index is already included
             as part of the copy (from a) */
-         auto result = std::find(child_copy_begin,
                                  child_copy_end, index);
-         if (result == (child.tour.begin() + offset_end)) {
+         if (lookup[index] == false) {
```

# Reordering the `sort`

- At the beginning of `iteration` the population already has valid distance values, it could also be sorted at this point
- Replace all of the `min_element` or `max_element` with `population.back()` or `population.front()`
- We can also replace checking if the fitness sum is going to be zero, by checking if the largest element is zero

# Change 3

```
     /* Find the maximum distance, at this point the
        metadata for each individual should be its
        distance */
-    double distance_max = std::max_element(population.beg..
+    double distance_max = population.back().metadata.dis...
```

```
     /* Normalize the fitness values */
-    double fitness_sum = std::accumulate(population.beg...
-    if (fitness_sum != 0.0) {
-        for (auto& individual : population) {
-            individual.metadata.normalized_fitness = ind...
+    if (population.front().metadata.fitness != 0.0) {
+        double fitness_sum = std::accumulate(population.b..
+        for (auto& individual : population) {
+            individual.metadata.normalized_fitness = ind...
+        }
```

Other places too...

# Merging in Selections

- Get rid of the selection container all together and just do it all in a single step

- Do the preprocessing
- `for(... i = 0; i < kPopulationSize; ++i)`
  - Pick the two individuals
  - Crossover
  - Randomly mutuate

- This also seperates out sequential code from obviously parallel code!

# Adding OpenMP

- Just add the following around the previous loop
- `#pragma omp parallel for shared(new_population)`
- Make `new_population` with `kPopulationSize` default constructed elements so each thread can update its own index without a critical section

# Optimize Uniform Selections

- It so happens with the input and these genetic operations, the population gets homogeneous
- All of the fitness values are equal to 0, therefore there is an equally likely chance to select them
- Checking for them is easy too! You just see if the largest element (which is at the front) is 0

# Change 4

```
        if (uniform_selection) {
            /* Pick the first individual */
            first = population->begin();
            std::advance(first,
                         rand_r(&seed) % kPopulationSize);

            /* Pick the second individual */
            second = population->begin();
            std::advance(second,
                         rand_r(&seed) % kPopulationSize);
        }
```

- Also, change the calls from `rand` to `rand_r` and use a **threadprivate** seed set to some initial value

# Minor Changes

- Compiler flags add much speedup in this case, but they're free and easy
- Use `-Ofast` and `-D_GLIBCXX_PARALLEL` (which doesn't seem to parallelize the sort, which is what I wanted
- Inline `distance`, `crossover` and `mutate`
- Use an `unsigned short` instead of an `unsigned int` for the index type

# Some Fixes

- `cl_float4` won't let you access `x` and it was suggested to try `s0`, which didn't work either
- Add `-U__STRICT_ANSI__` to `CXXFLAGS` (already fixed in the provided tarball)
- The platform may be messed up on `ece459-1` since there are two of them, should be fixed shortly

# Introduction

- You're computing the forces (or accelerations since m $= 1$) of all points in a space
- The forces for points far away are small and are costly to compute
- We can approximate these other points or disregard them (depending on how important the speedup/accuracy trade-off is for you)

# Binning

- Real solutions would probably use a specialized data structure like and octree but an easier way to do it is to seperate the points into bins
- In this case our space is $1000^3$, we can use 1000 bins of size $100^3$
- Compute the centre of mass for each bin, which we can do in parallel
- You also want to keep track of which points are in which bin

# Computing Forces

- Go over every bin
- Use all of the points in adjacent bins to compute the forces on each point in the bin
- Use the centres of mass to compute the forces with all other bins
- Ignore bins greater than a set number of bins away

# Introduction

- So, previously we traded off accuracy for performance, we can generalize this a bit

- Martin Rinard summarized these types of improvements [Rinard et al., 2010]:
  - Early phase termination [Rinard, 2007]
  - Loop perforation [Hoffmann et al., 2009]

# Early Phase Termination

- Recall barriers, we have to wait for every thread to reach the barrier, even if one is horribly slow
- Well, let's kill the slowest thread (this may change the meaning of the program)
  - We could develop a statistical model of the program behaviour and only kill tasks that don't introduce unacceptable distortions
  - You could output a confidence interval as well

# Loop Perforation

- Same idea to sequential programs, just throw away some data if it's not that useful (in a general manner)

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

changed to

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

- Given an appropriately distributed set of numbers, you would get a speedup of 2
- The paper does detail this for video encoding, giving indistinguishable results

# Summary

- A bunch of optimizations you could make to a real problem like in assignment 3
- Most performance boosts require a combination of knowledge:
  - Algorithms
  - Data structures
  - Hardware
  - Domain knowledge
- Approximation algorithms are a good way to get some speedup if you don't mind the trade-off

# References I

Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., and Rinard, M. (2009).
Using code perforation to improve performance, reduce energy consumption, and respond to failures.
Technical Report MIT-CSAIL-TR-2009-042, MIT CSAIL, Cambridge, MA.

Rinard, M. (2007).
Using early phase termination to eliminate load imbalances at barrier synchronization points.
In Proceedings of OOPSLA 2007, pages 369–386, Montreal, Quebec, Canada.

# References II

Rinard, M., Hoffmann, H., Misailovic, S., and Sidiroglou, S. (2010).
Patterns and statistical analysis for understanding reduced resource computing.
In Proceedings of Onward! 2010, pages 806–821, Reno/Tahoe, NV, USA. ACM.