Mike Fleming & Braden Scothern
CS 4230
Final Project Report

Quadratic Sieve

Algorithm Explanation

In order to fully understand our parallel implementation limitations an understanding of the sequential algorithm is necessary. The quadratic sieve is a factoring algorithm that is highly parallelizable when implemented with that intention from the ground up. It requires that you find values to satisfy the following equations:

$$x^2 \equiv y^2 \ (mod\ n)$$
$$x \not\equiv \pm y \ (mod\ n)$$

which then allow you to use and to factor . The algorithm to do this can be represented with this pseudocode:

```
1. Int x_o = ⌊√n⌋
2. Int i = 1
3. Set residues = new Set()
4. while(true) {
5.       x_i = (x_0 + i)^2 % n
6.       Set factored_x_i = x_i.factor()
7.       Set new_residues = factored_x_i.copy()
8.
9.       for f in factored_x_i {
10.          for r in residues {
11.              Int fr = f×r
12.              if fr.isSquareMod(n) {
13.                  return gcd(n,fr)
14.              }
15.          }
16.      }
17.
18.      for r in new_residues {
19.          residues.add(r)
20.      }
21.
22.      i++
23.}
```

This is a fairly simple algorithm aside from the recursive nature of needing to factor elements to factor a large element. Typically the inner factoring is handled by the pollard-rho method once the numbers get small enough because it is more efficient in those cases.
The part of this that can be parallelized is the while(true) loop. If you compute the next intermediate value then the hard part is factoring it and then combining and comparing it to every other intermediate value's residues so far. Because this can be computationally expensive the set is often handled as a sparse matrix with periodic checks to see if a combination of values can create a square.

Our Approach

Because we found open source software that is roughly equivalent to the pseudocode used to explain the algorithm, our OpenMP implementation doesn't have any speedup. The portions that we have parallelized are the smaller factoring portions of the code. The other difficulty that came with this is the fact that all of the threads need a lot of private access to the global residue set after finding the factorization of the intermediate value.

Because the algorithm is only really useful or interesting for values larger than a 64 bit int can represent, we needed a BigInt library. Had we been able to implement this from the ground up we could have designed it to be safe with parallel operations and set up a more robust tracking mechanism for our intermediate factorings than just a set that must be looped over each iteration. We could have also designed it to be compatible with MPI which would be the ideal parallelization library for such a task.

A Better Approach

If you are using MPI to factor a large number you can easily distribute iterations through the while loop and have each processor find and factor a fixed amount of intermediate value residues. Once they have done this they should send these values to the other processes and then receive the other residues and combine them into the local processes' master residue set, or ideally sparse matrix.

Taking an MPI approach, you would be able to get significant speedup so long as the factoring of an intermediate value is the time consuming portion of the process. If the combining and finding a square is the most expensive part, then all speedup will be lost. Assuming that factoring is the difficult part, which it is in good implementations, then you can expect a speedup that grows linearly with the number of processes executing.

The theoretical speedup according to Amdhal's Law is 1/(.1 + (1/P)(.9)) assuming you spend 90% of your time factoring and 10% comparing results. If you are able to cut down the comparison portion even more, which is essential in practice for large pseudoprimes then you can approach a speedup of P. An example of this kind of speedup is when the algorithm was first used to factor the number 9686961375462206147714092225435588290575999112457431987469512093081629822514570835693147662288398962801339199055182994515781515 4 where they spent 6 months doing the small factoring portions and 3 days computing the square from a sparse matrix in order to find the 2 primes that form this number.