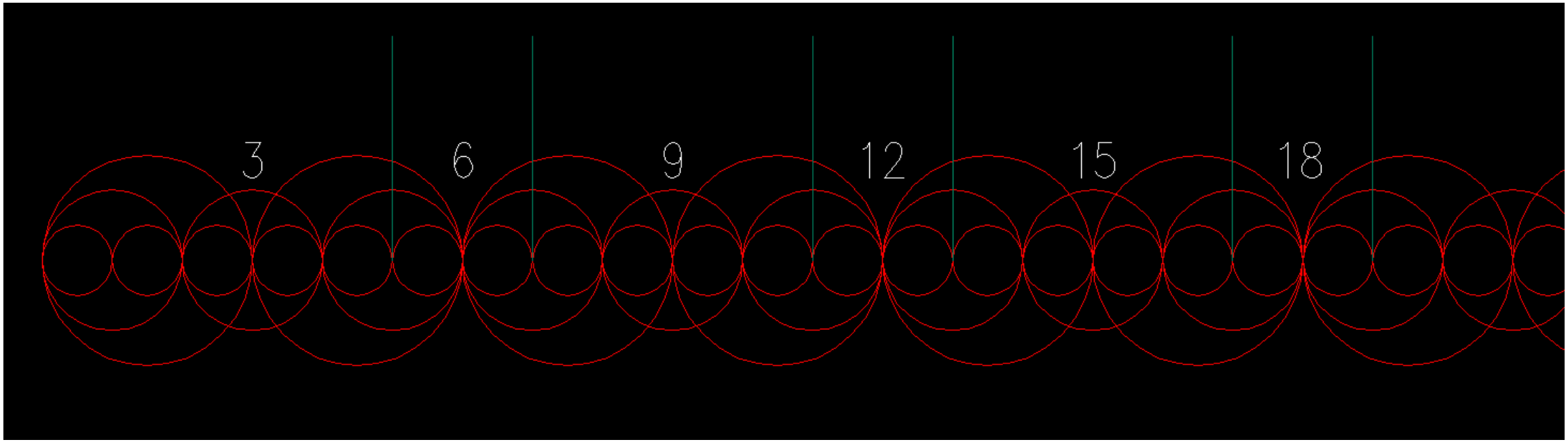
Concurrent Prime Number Generation

David Casente, Ellis Saupe

Sieve of Eratosthenes



Single threaded

```
//single threaded sieve function
for(i = 2; i < N; i++)
{
    if(numbers[i] == 1)
    {
        int j;
        for(j = 1; i * j < N; j++)
        {
            numbers[i * j] = 0;
        }
    }
}
```

- Advantages
  - Simple
  - Testing if integer x is prime is O(1) after generation
- Disadvantages
  - Memory intensive
  - Generation could be faster

Using a Bounded Buffer

- Advantages
  - Uses less memory
  - Unlimited generation
- Disadvantages
  - No O(1) isPrime testing
  - Needs concurrency, more complex

Multiple Threads / Processes to run sieve

- Should generate faster

- Multiple “Producers”
  - Evaluate multiples of primes
- “Consumer”
  - Parses array for primes

Current Sieve (Producer)

```
for(i = 0; i < count; i++)
{
    //get local versions of the minimums to reduce time in critical section.
    sem_wait(&minMutex);
    int rMin = readMin;
    int wMin = writeMin;
    sem_post(&minMutex);

    //get local version of the next multiple.
    sem_wait(&primeMutex);
    int x = next[i];
    sem_post(&primeMutex);

    //Testing if x's position in the bounded buffer is available for the sieve
    //ie it's doesn't loop all the way back around into the section of the array the consumer is processing
    //improve boolean logic later, this should work but it's not very cleanly written
    bool valid;
    if(rMin % N > wMin % N)
    {
        valid = (x % N > rMin % N && x - rMin < N) || (x % N < wMin && x - wMin < N);
    }
    else
    {
        valid = x % N > rMin % N && x % N < wMin % N && x - rMin < N;
    }

    if(valid)
    {
        //0 index in the buffer.
        buffer[next[i] % N] = 0;

        sem_wait(&primeMutex);
        //increment next
        next[i] += primes[i];
        sem_post(&primeMutex);
```

Current Consumer

```
int i = 2;
while(true)
{
    //get a local value of readMin.  Spends less time in critical section, reduces waiting.
    sem_wait(&minMutex);
    int rMin = readMin;
    sem_post(&minMutex);

    while(i < rMin)
    {
        //found a prime #
        if(buffer[i % N] == 1)
        {
            sem_wait(&primeMutex);
            //push new prime to the prime and next arrays.
            primes[count] = i;
            next[count] = 2 * i;
            count++;
            sem_post(&primeMutex);
        }

        //update writeMin and readMin
        sem_wait(&minMutex);
        rMin = readMin;
        writeMin = i;
        sem_post(&minMutex);

        //reset cell in buffer as it moves through so the sieve thread can work.
        buffer[i % N] = 1;
        i++;
    }
}
```