Project 2: Parallelizing the Sieve of Eratosthenes

1 Introduction to Sieve of Eratosthenes

The Sieve of Eratosthenes is an technique for computing all primes up to a given limit. The basic algorithm is simple. Start with an array of N booleans and then mark all multiples of 2 and greater for every number $i \in 2, 3, ...$ as false. As this iteration proceeds, if a number is determined to be composite, then all of its multiples will already be marked. Once this iteration is complete, then any entry in the array that remains true is prime. The pseudo-code of the algorithm is given by:

```
bool primes[2..N] = true
for i in {2..N}:
  if primes[i]:
    j = i+i
    while j<N:
       primes[j] = false
       j = j + i
for i in {2..N}:
  if primes[i]:
    print i
```

1.1 First Phase of Optimization

The previous implementation, while simple, is not particularly optimal. There are several straightforward optimizations that we can make. First the outer `i` loop does not need to loop all the way to N, but rather only to $\sqrt{N}$ as any composite number would be formed from at least two factors, the upper bound of the smallest would be given when the composite is a perfect square of a prime number. Second, we note that 2 is the only even prime, so we can eliminate even primes from the search and start searching from 3. For this we create an array of primes where index 0 represents 3, index 1 represents 4, and index i represents $(i + 1) * 2 + 1$. Using this technique we can reduce the amount of memory by a factor of 2. This optimization can given by the pseudo code:

```
bool primes[0..N/2] = true
i = 0
while 4*i*(i+3)+9 < N: # note (p^2<N)
  if primes[i]:
    p = (i+1)*2+1
    j = 3*p  # skip to next odd prime
    while j<=N:
       primes[(j-1)/2-1] = false
```

```
        j = j + 2*p #skip to next odd prime
print 2
for i in {0..N/2}:
  if primes[i]:
    p = (i+1)*2+1
    print p
```

1.2 Second Phase Optimization: Segmenting

One shortcoming of the sieve is that it requires a great deal of memory when solving for large numbers of primes. However, the algorithm can be segmented where primes can be found in blocks and once the primes are identified the booleans for that specific segment can be discarded. This optimization can help with cache performance as memory accessed in segments that can be small enough to fit into cache.

In this approach we create a list of primes (denoted `plist`) which we insert primes into as we solve each successive segments. A block size that checks a segment of primes of 512*1024 primes at a time is found to provide the best performance. In this algorithm we perform the first segment is performed just as the original algorithm is performed. This process adds enough primes to `plist` that the subsequent 274 billion primes can be computed (i.e. all primes up to $(512 * 1024)^2$). Then the remaining primes are computed using a loop over the following segments. Each segment allocates bools for its 512*1024 primes and subsequently marks the composite numbers and then inserts the newly found primes within the segment onto `plist`. The implementation of this version of the sieve can be found in `sieveo2.cc`.

1.3 Third Phase Optimization: Bit Storage

In the previous steps if this algorithm we were using arrays of booleans to mark the status of a number as prime or composite. Usually the boolean type is represented internally as an integer which means that many bits are wasted when storing an array of booleans. In this optimization we create an abstract data type, denoted `bitArray`, which uses bits to store these boolean values dramatically compressing the storage requirements improving speed. This provides additional speed improvement due to the reduced memory bandwidth requirements of the computation. In addition, we note that using `push_back()` to insert primes into the list of primes causes the vector to be periodically reallocated and data to be copied. We can use the prime counting function to estimate the size of the `plist` data structure to save this extra cost. Both of these optimizations are implemented in the program `sieveo3.cc`.

1.4 Fourth Phase Optimization: OpenMP parallelization

The next phase for optimization is to explore parallelization. For this optimization we will use thread based shared address based parallelism that is provided by OpenMP directives.

This particular implementation starts with the optimized version described in the previous sections. The approach recognizes that after the first segment is computed we have enough primes to compute all of the remaining segments. This means that they can be computed in any order, and thus can also be computed in parallel. The approach utilized is a static allocation of segments to parallel threads. We create the threads using the directive

```
#pragma omp parallel
```

Because each thread is implementing a separate segment, the boolean array needs to be allocated for each thread. There is also a problem in coordinating writing primes into the `plist` array. This is done by counting the number of primes that will be inserted into plist by each thread. The offsets for each thread is computed by performing a barrier to synchronize the threads using

```
#pragma omp barrier
```

Then one processor performs the prefix sum operation to determine where each thread is inserting its results into the `plist` array. Once the prefix sum is computed, then each thread can insert their primes into the larger list in parallel. Thus, while the approach is not as parallel as it could be (for example the first segment computation and the prefix sums are not parallelized), it is able to achieve a parallelism with a modest serial fraction. The program can be run with various numbers of threads to exploit multi-core architectures. The number of threads utilized is set by the environment variable `OMP_NUM_THREADS`. For example, running this using all 20 cores of the shadow cluster can be performed with the command:

```
OMP_NUM_THREADS=20 ./sieveomp
```

1.5 Performance Results

The running times reveal the benefits of the optimizations described. On the shadow cluster the execution times are documented in table 1. It can be seen here that algorithmic improvements can account for significant performance improvements where serial algorithm improvements account for a speedup of nearly a factor of seven while the parallel shared memory OpenMP implementation provides an additional speedup of nearly a factor of nine.

	sieve	sieveo1	sieveo2	sieveo3	sieveomp
Time	305 s	87 s	64 s	44 s	5 s

Table 1: Run-times on a single shadow node

2 Project 2 : The Assignment

Your job is to see if you can beat the best performance for the sieve using distributed memory methods. For the distributed memory solution the primes will be stored in a distributed array (that is the primes will not be gathered to processor zero, but instead remain distributed across processors). The example codes provide a count of the number of primes found along with a checksum which is the bitwise exclusive or operation of all prime numbers. Note, for this computation we are using 64 bit unsigned integers to represent the primes as we are computing primes larger than can fit in a 32 bit integer representation. Use the total count of primes and the checksum to compare your implementation of the sieve. For this project you are to pick one of the serial sieve implementations and develop a distributed memory parallel version using MPI. Try to beat the 5 second time achieved by the OpenMP implementation. Note: It is not necessary to beat this time for the grading of the project, but keep it as a challenge for yourself.

2.1 Graduate Student Exercise

Those students taking the course for graduate credit will need to also implement a hybrid OpenMP/MPI implementation using the openMP implementation provided. This will be included in addition to the exclusively distributed memory implementation.

3 References

- Wikipedia has a good section on the Sieve of Eratosthenes.

  `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

- MPI API documentation in addition to text:

  `https://www.mpich.org/static/docs/v3.2/`

- OpenMP documentation in addition to text:

  `https://www.openmp.org/spec-html/5.1/openmp.html`