# CS475: The prime sieve of Erastosthenes in OpenMP

Sanjay Rajopadhye, Wim Bohm
Colorado State University

# Primes problem

- Find all the prime numbers up to a given number *n*
- Sieve of Erastosthenes

  Have an array of prime candidates

  discover a prime, remove all multiples
- Strategy

  Start with a sequential algorithm and systematically parallelize it taking locality into account
  - Very OpenMP

# Algorithm

Create an array of numbers 2 … n,
    none of which is "marked"
        Invariant: the smallest unmarked number is a prime
k ← 2  /* k is the "next" prime number */
repeat
    Mark off all multiples of k as non-primes
    Set k to the next unmarked number
        Invariant: which must be a prime
        Why does the invariant work?
 until "done"

# Pseudo code

```
for (i=1; i<=n; i++) marked[i] = 0;
marked[0] = marked[1] = 1;
k = index = 2;
while (k<=n) {
    for (i=3; i<=n; i++) if (i%k == 0) marked[i]=1;
    while (marked[++index]) ; // do nothing
    //now index has the first unmarked number:
    //    the next prime
    k = index;
}
```

# Analysis & Improvement

- Where does the program spend its time?

  complexity?
- How to improve?
  - If $x=a*b$ is a composite number, then at least one of $a$ or $b$ is less than (or equal to) $\sqrt{x}$ (algorithmic improvement)
  - *So the upper bound of the* `for (i=3; i<=n; i++)` *loop can be tightened to?*
  - Do we need to start at i=3?

Take k = 7 (i.e. we have sieved with 2, 3 and 5)

has 2*7 been marked? has 3*7 been marked?

what is the first unmarked multiple of 7?

in general, where can i start?

# Better loop bounds

```
for (i=0; i<=n; i++) marked[i] = 0;
k = index = 2;
marked[0] = marked[1] = 1;
while (k*k<=n) { // stop at sqrt(n)
  for (i=k*k; i<=n; i++) if (i%k == 0) marked[i]=1;
    while (marked[++index]) ; // do nothing
    //now index has the first unmarked number:
    //    the next prime
    k = index;
}
```

**start at k*k see previous slide**

**can you improve the step?**
**can you avoid the conditional?**

# Sequential Algorithm

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 |
| 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 |

**How can we save space?**

we don't need the evens
make 2 a special case, save half the space

# Efficient sequential code

- Step wise improve the original sequential program Sieve 1
  - do the order of magnitude improvements, going to $\sqrt{n}$ only, starting from k*k
  - save space by only storing odds (pre-sieving evens)
    - **what about the step now?**
    - can we do this again, pre-sieving multiples of 3?
      - **what are the gains, the complications?**
- compare to original
- measure the running time for large values of $n$
  - find an appropriate range

# First easy parallelization

```
for (i=1; i<=n; i++) marked[i] = 0; // parallelize
k = index = 2;
marked[0] = marked[1] = 1;
while (k*k<=n) {
    for (i=k*k; i<=n; i+= k) marked[i]=1; // parallelize
    while (marked[++index]) ;
    k = index;
}
```

# Analysis

- The speedup flattens
  - What memory bandwidth does the program accomplish?

- In the parallel loop, data elements of a very large array ***marked*** are accessed once, with zero reuse
  - worse then PA1, because we access memory with a large stride, and write one byte of the whole cache line we have read

  .. and then we go through the whole array again for the next prime

- We need to rewrite (transform) the program so that a smaller, controllable chunk of memory gets reused (re-accessed) multiple times. HOW?

# Program transformation

- In the current program we go through the whole ***marked*** array for one prime.

```
for (i=k*k; i<=n; i+= k) marked[i]=1;
```

- This inner loop can be parallelized because the loop bodies are independent and the index ranges can be pre-computed. Then we find the next prime and repeat.
- What if we grab a block of ***marked*** and apply **all necessary primes** to it? Now we can adjust the **blocksize** and so adjust the memory access pattern.
- Which are the necessary primes, i.e. which primes do we sieve with and apply to block [ start .. finish ] ?

# Blocking the sieve

Preamble: In an array primes[] store primes up to sqrt(n), say there are numprimes of them

Elements of the marked array up to index sqrt(n) have been marked
So we can start **blocking** at that index (call it blockStart):
instead of going all the way to n with one prime at the time,
we sieve with all primes one block of size BLKSIZE at the time.

```
for (ii=blockStart; ii<=n; ii+=BLKSIZE)  // parallelize
  for (j=0; j<=numprimes; j++)
    for (i=start; i<=min(start+BLKSIZE, n); i+= primes[j])
          marked[i]=1;
```

We have changed the order of computation, **is it legal?**

Yes, as long as sieving with prime[j] starts with the **proper** next multiple of cP = prime[j]

What is the value of start?

**The first odd multiple of cP >= cP*cP in the current block [ ii .. ii+BLKSIZE-1 ]**

# Finding odd multiple of p in [lo..hi]

- Cases:
  - hi < p*p
  - lo < p*p <= hi
  - lo >= p*p
    - Now locate first odd multiple of p
    - in CONSTANT time
    - (Some of you write a loop ☹ )

# Blocked Sieve n=100, BLKSIZE=30

```
 1  3  5  7  9

11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

41 43 45 47 49 51 53 55 57 59 61 63 65 67 69

71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

1: Pre compute primes in block <= sqrt(n)

– **3** **5** **7** –

11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

41 43 45 47 49 51 53 55 57 59 61 63 65 67 69

71 73 75 77 79 81 83 85 87 89 91 93 95 97 99

2: Sieve block 1 with 3 (start = 15)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | | **5** | | **7** | – | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | | – | 23 | 25 | | – | 29 | 31 | | – | 35 | 37 | | – |
| 41 | 43 | 45 | 47 | 49 | 51 | 53 | 55 | 57 | 59 | 61 | 63 | 65 | 67 | 69 | | | | |
| 71 | 73 | 75 | 77 | 79 | 81 | 83 | 85 | 87 | 89 | 91 | 93 | 95 | 97 | 99 | | | | |

We do NOT sieve block 1 with 7, why not?

3: Sieve block 1 with 5 (start = 25)

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | 3 | 5 | 7 | – | | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – |
| 41 | 43 | 45 | 47 | 49 | 51 | 53 | 55 | 57 | 59 | 61 | 63 | 65 | 67 | 69 |
| 71 | 73 | 75 | 77 | 79 | 81 | 83 | 85 | 87 | 89 | 91 | 93 | 95 | 97 | 99 |

4: Sieve block 2 with 3 (start = 45)

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | | **5** | **7** | – | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – |
| 41 | 43 | – | 47 | 49 | – | 53 | 55 | – | 59 | 61 | – | 65 | 67 | – |
| 71 | 73 | 75 | 77 | 79 | 81 | 83 | 85 | 87 | 89 | 91 | 93 | 95 | 97 | 99 |

5: Sieve block 2 with 5 (start = 45)

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | **5** | **7** | – | | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – |
| 41 | 43 | – | 47 | 49 | – | 53 | – | – | 59 | 61 | – | – | 67 | – |
| 71 | 73 | 75 | 77 | 79 | 81 | 83 | 85 | 87 | 89 | 91 | 93 | 95 | 97 | 99 |

6: Sieve block 2 with 7 (start = 49)

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | | **5** | **7** | – | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – |
| 41 | 43 | – | 47 | – | – | 53 | – | – | 59 | 61 | – | – | 67 | – |
| 71 | 73 | 75 | 77 | 79 | 81 | 83 | 85 | 87 | 89 | 91 | 93 | 95 | 97 | 99 |

7: Sieve block 3 with 3 (start = 75)

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | **5** | **7** | – | | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – |
| 41 | 43 | – | 47 | – | – | 53 | – | – | 59 | 61 | – | – | 67 | – |
| 71 | 73 | – | 77 | 79 | – | 83 | 85 | – | 89 | 91 | – | 95 | 97 | – |

8: Sieve block 3 with 5 (start = 75)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | **5** | **7** | – | | | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – | |
| 41 | 43 | – | 47 | – | – | 53 | – | – | 59 | 61 | – | – | 67 | – | |
| 71 | 73 | – | 77 | 79 | – | 83 | – | – | 89 | 91 | – | – | 97 | – | |

9: Sieve block 3 with 7 (start = 77)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | **3** | **5** | **7** | – | | | | | | | | | | | |
| 11 | 13 | – | 17 | 19 | – | 23 | – | – | 29 | 31 | – | – | 37 | – | |
| 41 | 43 | – | 47 | – | – | 53 | – | – | 59 | 61 | – | – | 67 | – | |
| 71 | 73 | – | – | 79 | – | 83 | – | – | 89 | – | – | – | 97 | – | |

# Interleaved data decomposition

Book also discusses interleaved allocation:

- threads sieve the whole block with interleaved primes
- use thread_num (start) and num_threads (step) to pick the next prime
- BUT

  load imbalance

  locality problems