Common Parallel Programming Problems - Part 2

Thread Safe Functions and Libraries

- Example from earlier
  - Foo (1) Foo might add its arg to a shared variable
- Updating unprotected hidden state in a subroutine is poor programming practice
- Desirable that routines be *thread safe*
- Complete thread safety unrealistic
  - Every call would have to do locking
  - Huge performance impact
- Compromise: make thread safe when called concurrently on different objects but not when called on same object
- Implicit if objects do not share state
- Implementer must protect state otherwise

Thread-safe functions

- Interfaces should be defined so they can be implemented in a thread safe manner
  - Should not update hidden global state
    - With threads not clear whose state it would be
- Example: strtok which tokenizes a string
  - First call sets state of hidden parser
  - Successive threads advance parser
  - Hidden state makes unsafe
- Solutions
  - Put parser in thread local storage – but should not need threads
  - Make parser object an explicit argument – each thread would have local parser object and pass as argument

Thread Safe libraries

- Some libraries have thread safe and thread unsafe versions
- On Windows compiler option /MD uses thread safe versions
- On Linux some versions can be thread safe e.g. random number functions
- Need to read compiler, library and function documentation to check

Memory Issues

- Program performance now are often limited by memory speed
- Multicore can exacerbate this
- Conserving memory bandwidth
  - Pack data tightly
    - boolean arrays into values per bit not per byte
    - use shortest integer type that can hold values
    - when declaring C/C++ structures declare fields in descending size
      - Minimizes padding
    - Some compilers have #pragma pack that uses no padding
      - Sometimes counterproductive – causes misaligned loads/stores (slower)
  - Move it less frequently between cores

Working in Cache

- Moving data less frequently more subtle
- Two issues: between core and memory, between cores
- Between core and memory benefits sequential also
- *cache oblivious blocking*
  - recursively divides problem into smaller and smaller subproblems
  - Eventually fit in cache
  - E.g. FFTW – Fastest Fourier Transform in the West (Frigo 1997)
- reorder steps in code
  - Simplest just by interchanging loops

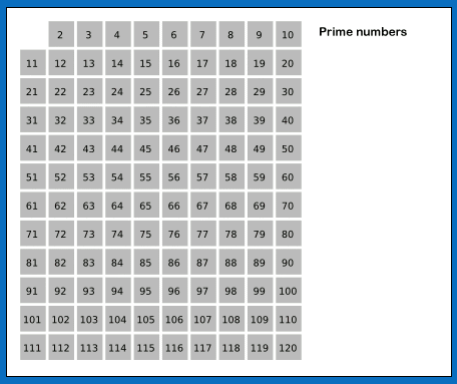
Sieve of Eratosthenes

- Can be used to demonstrate
- Enumerates the primes up to some number *n*
- Simple version had 2 nested loops
  - Outer finds primes
  - Inner strikes out composite (non-prime) numbers
  - Unfriendly to cache – inner loop over entire array *composite*
  - Code in *progs/Ahkter/Ch7/Figure_7.16*

Sieve of Eratosthenes - details

1. Write a list of numbers from 2 to the largest number you want to test for primality. Call this List A. (This is the list of squares on the left side of the picture.)
2. Write the number 2, the first prime number, in another list for primes found. Call this List B. (This is the list on the right side of the picture.)
3. Strike off 2 and all multiples of 2 from List A.
4. The first remaining number in the list is a prime number. Write this number into List B.
5. Strike off this number and all multiples of this number from List A. The crossing-off of multiples can be started at the square of the number, as lower multiples have already been crossed out in previous steps.
6. Repeat steps 4 and 5 until no more numbers are left in List A. Note that, once you reach a number greater than the square root of the highest number in List A, all the numbers remaining in List A are prime.

Sieve of Eratosthenes - illustration



	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	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Cache Friendly Sieve of Eratosthenes

- Represents sieve as small window into large array *composite*
- Window size is $\sqrt{n}$ bytes
- inner loop now suspended at end of window
- array *striker* stores indices of these suspended loops
  - Has element for each prime up to $\sqrt{n}$
  - Grows more slowly than *n*
  - Fits in *10^6* cache when n is up to *10^11*
- Restructuring introduces extra complexity and bookkeeping
  - Still significant performance improvement
  - Cache friendly version can be 5 times faster
- Code in *progs/Ahkter/Ch7/Figure_7.17*

Multi-core case

- Additional issue of transfer between cores
- Not explicit in code but implicit from patterns of read/write
- Two types of data dependencies
  - Read-write: a core writes a cache line, and then different core reads it
  - Write-write: a core writes a cache line, and then different core writes it
- Note 2 cores reading a cache line that is not written does not cause movement
  - Each core keeps own copy

Minimizing memory bus traffic

- Minimize core interaction through shared locations
  - Patterns that reduce lock contention do this
  - Having each thread work on local copy of data and merging at the end can be effective

Multi-threaded Cache Friendly Sieve of Eratosthenes

- fill array factor sequentially – operate on windows in parallel
- sequential part takes time O(√*n*) – so not significant
- Need to share some data
  - Array *factor* is read-only so can be shared
  - Array *composite* updated as primes found – but threads operate on different windows so only interference is at window boundaries that fall in a cache line
    - each thread can have private portion that holds only window
    - reduced space requirements from O(*n*) to O(√*n*)
      - Can count primes up to 10^11 on 32 not machine
  - *count* updated as primes found
    - could use atomic but better to give each thread private *count* – sum at end

Multi-threaded Cache Friendly Sieve of Eratosthenes

- array striker updated as window processed
  - Private copy for each thread
  - Introduces loop carried dependence between windows
    - for window initial value of *striker* is last from previous window
    - Break by computing initial values from scratch
- *base* is new in parallel version
  - Keeps track of start of window for which *striker* true
  - When base different from start of window, thread recomputes striker from scratch
    - Sets *striker[k]* to lowest multiple of *factor[k]* inside or after window
- Code in *progs/Ahkter/Ch7/Figure_7.19*
- In practice can reduce work by only looking for odd primes