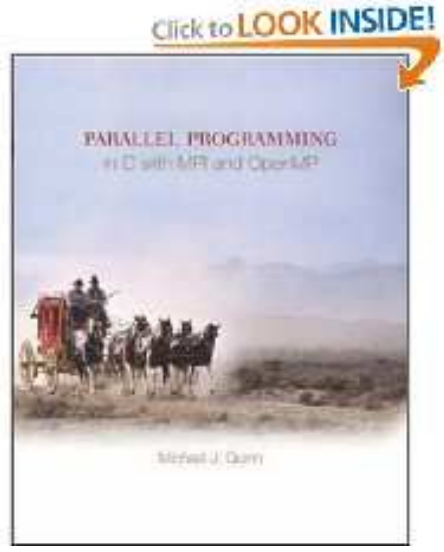
# The Sieve of Eratosthenes

# Overview

- Chapter 5 from *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**

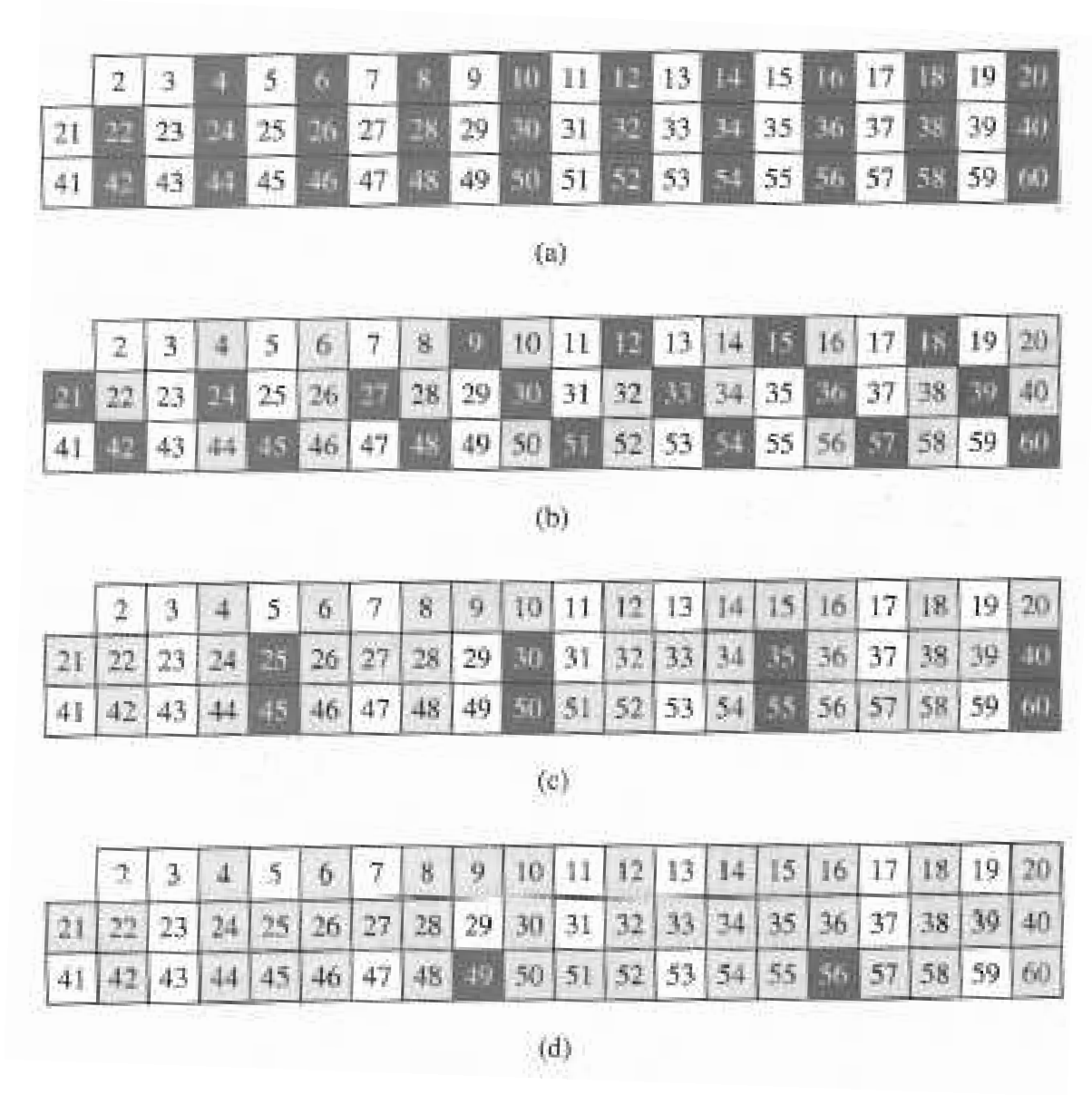



- The Sieve of Eratosthenes: *finding prime numbers*

# Introduction

- Definition of a prime number: divisible only by 1 and itself
  - Examples: $2, 3, 5, 7, 11, 13, 17, 19 \ldots$
- Pseudocode for the Sieve of Eratosthenes:
  1. Create a list of natural numbers $2, 3, 4, \ldots, n$, none is marked.
  2. Set $k$ to 2, the first unmarked number on the list.
  3. Repeat until $k^2 > n$:
     (a) Mark all multiples of $k$ between $k^2$ and $n$.
     (b) Find the smallest number greater than $k$ that is unmarked. Set $k$ to this new value.
  4. The unmarked numbers are primes.

# Example: Finding primes smaller than 60



(a)

(b)

(c)

(d)

# Some remarks

- The complexity of the sequential algorithm: $\mathcal{O}(n \ln \ln n)$
- In a C implementation:
  - We can make use of a 1D array, `char *marked`
  - The length of `marked` is $n-1$, corresponding to $2, 3, \ldots, n$
  - To begin with, all elements in `marked` are given value 0
  - In the end, `marked[i]==0` means $i+2$ is a prime number

# Applying Foster's design methodology

- Step 1 (Partitioning): We can break array `marked` into $n-1$ elements, each being a primitive task
- Each task represents one natural number $j$ ($j$=`i+2`)
- In each iteration of the "Sieve of Eratosthenes", when $k$ is known:
  - Each task (with index `i`) checks for itself whether `mod(i+2,k)==0`
  - If so, $j = i + 2$ is a multiple of $k$, `marked[i]=1`
- Step 2 (Communication): Two communications are needed before each iteration
  - Finding a new value of $k$, by a reduction
  - Informing each primitive task of the chosen $k$, by a broadcast

# Agglomeration

- We want to merge the primitive tasks into a few large tasks
- Each large task is responsible for a group of natural numbers
- Option 1 (interleaved data decomposition):
  - process 0 is responsible for $2, 2+p, 2+2p, \ldots$
  - process 1 is responsible for $3, 3+p, 3+2p, \ldots$
  - and so on
- Option 2 (block data decomposition):
  - $2, 3, \ldots, n$ are divided into $p$ contiguous blocks of (roughly) equal size

# Interleaved data decomposition

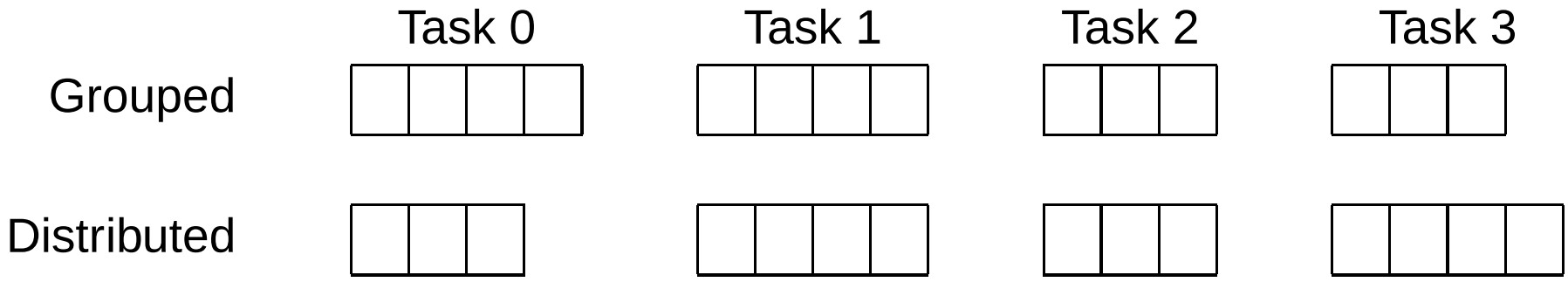
- Advantage: given a particular array index `i`, it's very easy to know which process owns that index (`mod(i,p)`)
- Disadvantage 1: significant load imbalance among the processes
  - For example, if $p = 2$, and both processes are marking multiples of 2
  - process 0 marks $\lceil (n-1)/2 \rceil$ elements
  - process 1 marks none
- Disadvantage 2: reduction is needed to find the next $k$ value

# Block data decomposition

- Very easy if the number of elements, $m$, is divisible by $p$
- When $m$ is not divisible by $p$, we want each processes to have either $\lceil m/p \rceil$ or $\lfloor m/p \rfloor$ elements
  - Method 1:
    - Calculate $r$ = $m$ mod $p$
    - The first $r$ processes get $\lceil m/p \rceil$ elements
    - The remaining processes get $\lfloor m/p \rfloor$ elements
  - Method 2:
    - The first element index assigned to process $i$ is $\lfloor (im)/p \rfloor$
    - The last element index assigned to process $i$ is $\lfloor ((i+1)m)/p \rfloor - 1$

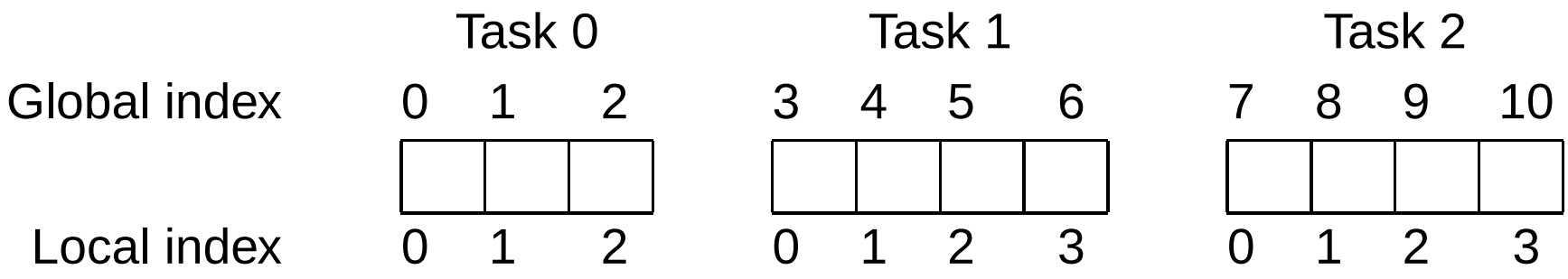
# Two examples of block data decomposition

$m = 14$, $p = 4$



# Mapping from local index to global index

- When we decompose an array into blocks, it is very important to distinguish between global and local indices
- We also need to remember the mapping



# Block decomposition is good here

- Note that the largest prime used for sieving is $\sqrt{n}$
- By block decomposition, process 0 is responsible for $2, 3, \ldots, n/p$
- If $n$ is very large, i.e., $\sqrt{n} > p$, then we have $n/p > \sqrt{n}$
  - This means that process 0 controls all the primes through $\sqrt{n}$
  - Process 0 can decide the new value of $k$, no need for reduction
- A second advantage of block decomposition:
  - Each process can find for itself the first multiple of $k$ and mark that cell (call it $i$), and then mark $i + k$, $i + 2k$ and so on
  - No need for modulo operations anymore!

# MPI_Bcast

```
int MPI_Bcast (
  void         *buffer,
  int           count,
  MPI_Datatype datatype,
  int           root,
  MPI_Comm      comm)
```

- Before each sieve iteration, process 0 still needs to tell all the other processes about the new value of $k$

```
MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

# Parallel sieve algorithm

- Every process creates its portion of the list
- Every process starts with setting $k = 2$
- During each $k$-iteration
  - every process finds the first multiple of $k$ in its portion
  - mark every $k$'th element after the first multiple of $k$
- Before the next iteration, process 0 determines the new $k$ value and broadcasts it to all the other processes (`MPI_Bcast`)

# Analysis of time usage

- Let $\chi$ denote the time needed to mark a particular cell
- Time usage for the sequential sieve algorithm is thus $\chi(n \ln \ln n)$
- For the parallel sieve algorithm, each iteration requires a broadcast
  - Time usage for a broadcast is $\lambda \lceil \log_2 p \rceil$
- Number of sieve iterations needed: $\sqrt{n} / \ln \sqrt{n}$
- Total time needed for the parallel sieve algorithm:

$$\frac{\chi(n \ln \ln n)}{p} + \left(\sqrt{n} / \ln \sqrt{n}\right) \lambda \lceil \log_2 p \rceil$$

# Improvement 1: Deleting even integers

- All even integers, except 2, are not prime numbers
- It's thus a waste to create data storage corresponding to $2, 4, 6, \ldots$
- Enough to only consider $3, 5, 7, 9, \ldots$
- We can save half the storage—memory footprint is halved
- Time can also be halved for the sequential algorithm
- Time usage of the parallel algorithm:

$$\frac{\chi(n \ln \ln n)}{2p} + \left(\sqrt{n}/\ln \sqrt{n}\right) \lambda\lceil \log_2 p \rceil$$

# Improvement 2: Eliminating broadcast

- If each process, beforehand, does an extra job to find all the primes between 3 and $\lfloor \sqrt{n} \rfloor$
- Then, each process knows what $k$ values to use in the parallel algorithm
- Thus, no need for `MPI_Bcast`!
- In other words, elimination of communication at the cost of extra work
- Time usage of the parallel algorithm:

$$\frac{\chi(n \ln \ln n)}{2p} + \chi \sqrt{n} \ln \ln \sqrt{n}$$

# Improvement 3: Reorganizing loops

- Observation: there are two nested loops involved in the sieve algorithm
  - Outer loop over the prime sieve $k$ between 3 and $\lfloor\sqrt{n}\rfloor$
  - Inner loop over each process's block share of integers $3, 5, \ldots, n$
- When $k$ is large, marking elements $i + k$, $i + 2k$, $i + 3k$ can span a large distance in the memory, therefore very little (or no) cache reuse
- To improve the cache hit rate, a remedy is to switch the order of the two nested for-loops
  - See Figure 5.9 in the textbook for an example

# Exercises

- Divide the indices $0, 1, 2, 3, \ldots, m-1$ evenly into $p$ blocks, where two consecutive blocks share exactly two overlapping indices.
- Implement the complete parallel algorithm for the Sieve of Eratosthenes, also making use of the improvements suggested in Chap. 5.9.1 and 5.9.2. Use the parallel program to find all primes that are smaller than $10^6$.