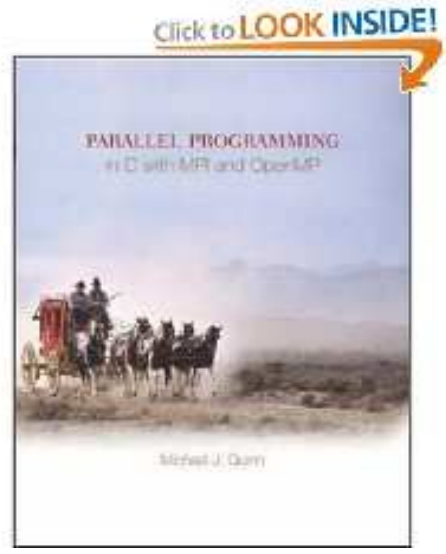
Two cases of parallelization

Overview

- Chapters 5 & 6 from *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**



  - The sieve of Eratosthenes: *finding prime numbers*
  - Floyd's algorithm: *the all-pairs shortest path problem*
- Understanding the design of parallel algorithms
- Use of simple but important MPI commands

The sieve of Eratosthenes

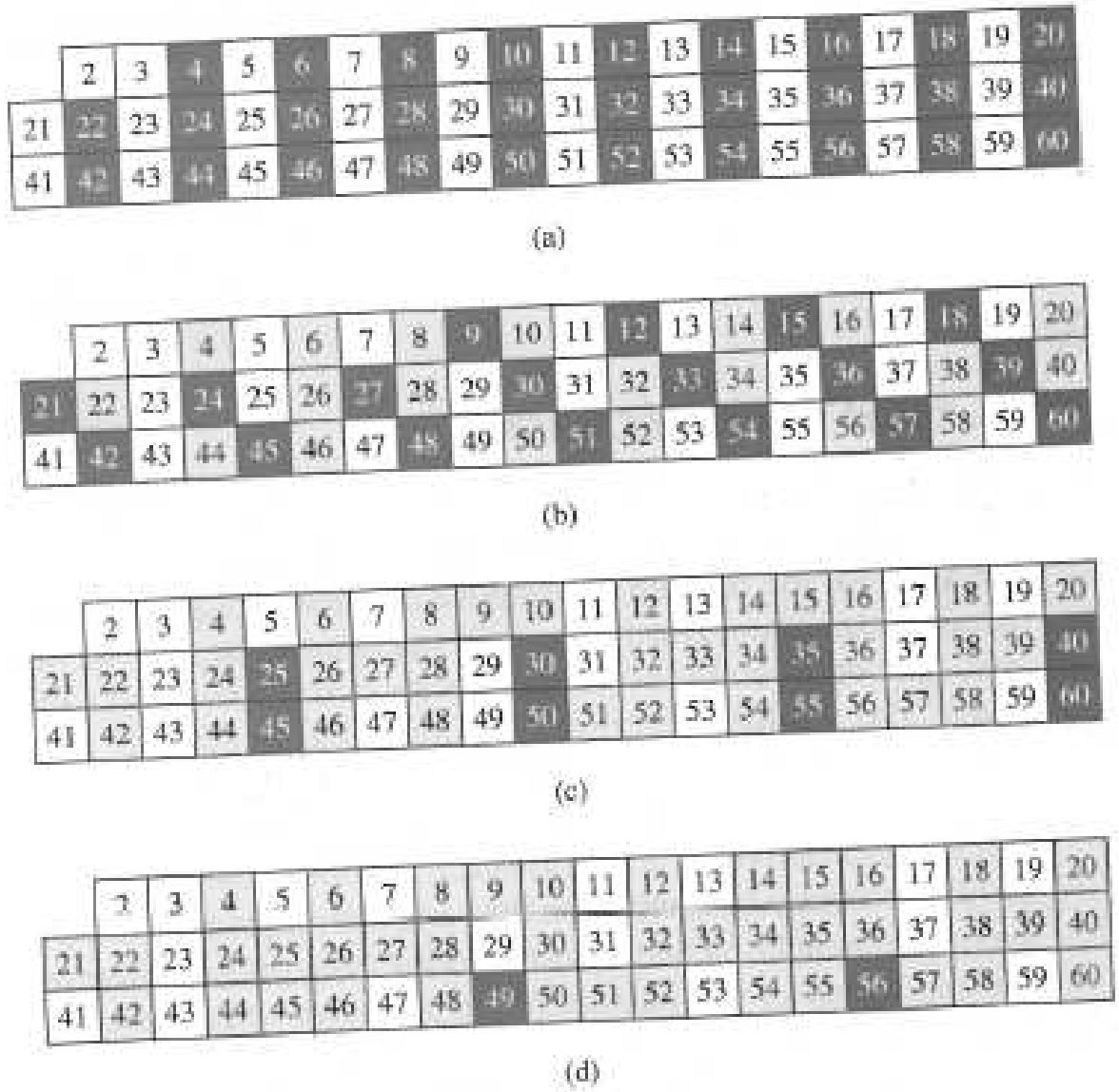
- Definition of a prime number, divisible only by 1 and itself
  - Examples: $2, 3, 5, 7, 11, \ldots$
- Pseudocode for the sieve of Eratosthenes:
  1. Create a list of natural numbers $2, 3, 4, \ldots, n$, none is marked.
  2. Set k to 2, the first unmarked number on the list
  3. Repeat

     (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.

     Until $k^2 > n$
  4. The unmarked numbers are primes.

Finding primes smaller than 60



(a)

(b)

(c)

(d)

Source of parallelism

- Marking multiples of k between k^2 and n can be done concurrently by many processes, each responsible for a “segment” of the list
- Finding the new value of k may need info exchange among the processes

Block data decomposition

- An array is divided into p contiguous blocks of roughly equal size
- If the array length n is not divisible by p
  - make sure that the difference between the largest block and the smallest block is 1
- Which process is responsible for which part should be easily found out
  - mapping between local index and global index

Advantages of block decomposition

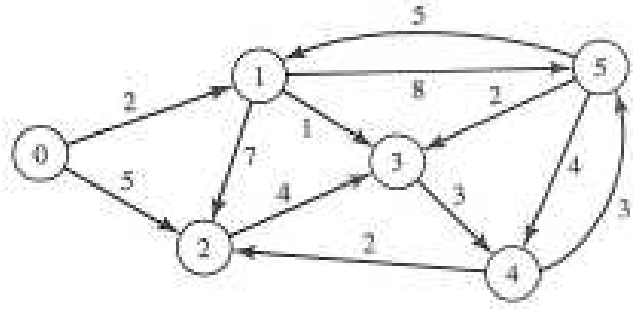
- Recall: the largest prime to sieve integers up to n is $\lfloor\sqrt{n}\rfloor$
- If $n/p > \sqrt{n}$, it is always process 0 that decides the new value of k
  - no need to compare different findings from different processes
- The actual work on a process:
  - find the first multiple of k and mark that cell (call it j)
  - then mark $j+k$, $j+2k$, and so on

A parallel 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
- Process 0 determines the new k value and broadcast it to all other processes (`MPI_Bcast`)
- There can be several improvements as suggested in Chap. 5.9

Finding shortest paths

- Starting point: a graph of vertices and weighted edges



- The edges may have directions
  - if there's path from vertex i to j, there may not be path from vertex j to i
  - path length from vertex i to j may be different than path length from vertex j to i
- Objective: finding the shortest path between every pair of vertices
- Application: table of driving distances between citie pairs

Adjacency matrix

- There are n vertices
- The path length from vertex i to vertex j is stored as $a[i, j]$
- An $n \times n$ adjacency matrix a keeps the entire path length info

	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0

- If $a[i, j]$ is ∞, it means there is no path from vertex i to vertex j

Example of all-pairs shortest path

For the adjacency matrix given on the previous slide, the solution of the all-pairs shortest path is as follows:

	0	1	2	3	4	5
0	0	2	5	3	6	9
1	∞	0	6	1	4	7
2	∞	15	0	4	7	10
3	∞	11	5	0	3	6
4	∞	8	2	5	0	3
5	∞	5	6	2	4	0

Table of shortest path lengths

Floyd's algorithm

Input: n — number of vertices
a — adjacency matrix
Output: Transformed a that contains the shortest path lengths

for $k \leftarrow 0$ to $n-1$
    for $i \leftarrow 0$ to $n-1$
        for $j \leftarrow 0$ to $n-1$
            $a[i,j] \leftarrow \min(a[i,j],\ a[i,k]+a[k,j])$
        endfor
    endfor
endfor

Some observations

- Floyd's algorithm is an exhaustive and incremental approach
- The entries of the a-matrix are updated n rounds
- $a[i,j]$ is compared with all n possibilities,
  that is, against $a[i,k] + a[k,j]$, for $0 \leq k \leq n-1$
- n^3 of comparisons in total

Source of parallelism

- During the k'th iteration, the work is (in C syntax)

```
for (i=0; i<n; i++)
  for (j=0; j<n, j++)
    a[i][j] = MIN( a[i][j], a[i][k]+a[k][j] );
```

- Can all the entries in a be updated concurrently?
- Yes, because the k'th column and the k'th row remain the same during the k'th iteration!
  - Note that `a[i][k]=MIN(a[i][k],a[i][k]+a[k][j])` will be the same as `a[i][k]`
  - Note that `a[k][j]=MIN(a[k][j],a[k][k]+a[k][j])` will be the same as `a[k][j]`

Work division

- Let one MPI process be responsible for a piece of the a matrix
- Memory storage of a is accordingly divided
- The division can in principle be arbitrary, as long as the number of all $a[i,j]$ entries is divided evenly
- However, a row-wise block data division is very convenient
  - 2D arrays in C are row-major
  - easy to send/receive an entire row of a
- We therefore choose to assign one MPI process with a number of consecutive rows of a

Communication pattern

- Recall that in the k'th iteration:
$$a[i,j] \leftarrow \min(a[i,j],\ a[i,k] + a[k,j])$$
- Since the data of a is divided rowwise, so $a[i,k]$ is also in the memory of the MPI process that owns $a[i,j]$
- However, $a[k,j]$ is probably in another MPI process's memory
- Communication is therefore needed!
- Before the k'th iteration, the MPI process that has the k'th row of the a matrix should broadcast this row to everyone else

Recap: creating 2D arrays in C

To create a 2D array with m rows and n columns:

```
int **B, *Bstorage, i;
...
Bstorage=(int*)malloc(m*n*sizeof(int));
B=(int**)malloc(m*sizeof(int*));
for (i=0; i<m; i++)
  B[i] = &Bstorage[i*n];
```

The underlying storage is contiguous, making it possible to send and receive an entire 2D array.

Global index vs. local index

- Suppose a matrix (2D array) is divided into row-wise blocks and distributed among p processors
- Each processor only allocates storage for its assigned row block
- We need to know: a local row corresponds to which global row?
- Mapping: local index $\rightarrow$ global index
- On processor number `proc_id`
  `global_index=BLOCK_LOW(proc_id, p, n)+local_index`

Main work of parallel Floyd's algorithm

```
void compute_shortest_paths (int id, int p, dtype **a, int n)
{
   int  i, j, k;
   int  offset;   /* Local index of broadcast row */
   int  root;     /* Process controlling row to be bcast */
   int* tmp;      /* Holds the broadcast row */
   tmp = (dtype *) malloc (n * sizeof(dtype));
   for (k = 0; k < n; k++) {
      root = BLOCK_OWNER(k,p,n);
      if (root == id) {
         offset = k - BLOCK_LOW(id,p,n);
         for (j = 0; j < n; j++)
            tmp[j] = a[offset][j];
      }
      MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
      for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
         for (j = 0; j < n; j++)
            a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
   }
   free (tmp);
}
```

Matrix input

- Recall that each MPI process only stores a part of the a matrix
- When reading a from a file, we can
  - let only process $p-1$ do the input
  - once the number of rows needed by process i are read in, they are sent from process $p-1$ to process i using `MPI_Send`
  - process i must issue a matching `MPI_Recv`
- The above simple strategy is not parallel
- Parallel I/O can be done using MPI-2 comands

Matrix output

- Let only process 0 do the output
- Each process needs to send its part of a to process 0
- To avoid many processes sending its entire subdata to process 0 at the same time
  - Process 0 communicates with the other processes in turn
  - Each process waits for a “hint” (a short message) from process 0 before sending its data (a large message)

Exercises

- 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 10^9. Study the speedup by using different numbers of MPI processes.
- Implement the complete Floyd's algorithm and try it on a large enough adjacency matrix (with randomly chosen path lengths).