Chapter 5

The Sieve of Eratosthenes

Background

• The Greek mathematician Eratosthenes (Er’a.tas’.the.nez’, 276-194 BC) wanted to find a way of generating the prime numbers up to some number n
  – No formula generates these primes
  – However, he devised a method that has become known as the sieve of Eratosthenes
Outline to the Solution

- The sequential algorithm
- Sources of parallelism
- Data decomposition options
- Parallel algorithm development, analysis

Sieve of Eratosthenes
Sequential Algorithm in Pseudocode

1. Create a list of unmarked natural numbers
   \[ 2, 3, \ldots, n \]
2. \( k \leftarrow 2 \)
3. Repeat
   (a) Mark all multiples of \( k \) between \( k^2 \) and \( n \)
   (b) Let \( k \leftarrow \) smallest unmarked
        number > \( k \)
     until \( k^2 > n \)
4. The unmarked numbers are primes
The algorithm is not practical for large values of $n$
Algorithm complexity is $\Theta(n \ln \ln n)$
The number of digits of $n$ is logarithmic to size of $n$
Who cares?
- Finding prime numbers is important for encryption algorithms
- A modified form of sieve is an important research tool in number theory

Data Structure Used For Sequential Algorithm

- Assume a Boolean array of $n$ elements
- Array indices are 0 through $n$-2 and they represent the numbers 2, 3, ..., $n$
- The boolean value at index $i$ represents whether or not the number $i+2$ is marked
  - Indices that are marked represent composite numbers (i.e., not prime)
- Initially, all numbers are unmarked
One Method to Parallelize

• Because the focus of the algorithm is the marking of elements in an array, domain decomposition makes sense
• Domain decomposition
  – Divide data into n-1 pieces
  – Associate computational steps with data
• One primitive task per array element
  – These will be agglomerated into larger groups of elements

Parallelizing Algorithm Step 3(a)

• Recall Step 3(a):
  3 a) Mark all multiples of k between $k^2$ and $n$

• The following straightforward modification allows this to be computed in parallel:
  for all $j$ where $k^2 \leq j \leq n$ do
    if $j \mod k == 0$ then
      mark $j$ (i.e., it is not a prime)
    endif
  endfor

• Each $j$ above represents a primitive task
Parallelizing Algorithm Step 3(b)

• Recall Step 3(b):
  3 b) Find smallest unmarked number > k

• Parallelizing requires two steps:
  – Min-reduction (to find smallest unmarked number > k)
  – Broadcast (to get result to all tasks)
• Plus – remember these are in a repeat-until loop which loops until \(k^2 > n\)

Good News – Bad News

• We have found lots of parallelism to exploit
• That is the good news!
• Look back at the last slide – there are a lot of reduction and broadcast operations
• That is the bad news!
• As usual, we will try to agglomerate the primitive tasks into more substantial tasks and hopefully improve the situation
• We will see that we end up with an algorithm that requires less computation and less communication than the original algorithm
Agglomeration Goals

• We want to:
  – Consolidate tasks
  – Reduce communication cost
  – Balance computations among processors
• We often call the result of partitioning, agglomeration, and mapping the data, decomposition or just the decomposition

Data Decomposition Options

1. Interleaved (cyclic)
   – Different PEs handle the below sets of integers, where p is the number of PEs:
     • $P_0$ handles $2, 2+p, 2+2p, ...$
     • $P_1$ handles $3, 3+p, 3+2p, ...$
     • $P_2$ handles $4, 4+p, 4+2p, ...$
   – It’s easy to determine the owner or handler of each number:
     • The number $i$ is handled by process $(i-2) \mod p$
Data Decomposition Options

1. Interleaved (cyclic) - continued
   - But, this scheme leads to a load imbalance for this problem
   - If we are using two processes, process 0 marks the 2-multiples among even no’s while process 1 marks 2- multiples among odd no’s
     - Process 0 marks \( \left\lfloor \frac{(n-1)}{2} \right\rfloor \) elements & process 1 marks none
   - On the other hand, for four processes, process 2 is marking multiples of 4 which is duplicating process 0’s work
   - Moreover, finding the next prime still requires a reduction/broadcast operation so nothing is saved there

Data Decomposition Options

2. Block
   - Array \([1,n]\) is divided into \(p\) contiguous blocks of roughly the same size for each PE
   - We want to balance the loads with minimum differences between the processes
   - It is not desirable to have some processes doing no work at all
   - We’ll tolerate the added complication to determine owner when \(n\) not a multiple of \(p\)
Block Decomposition Options

• We want to balance the workload when \( n \) is not a multiple of \( p \)
• Each process gets either \( \lceil n/p \rceil \) or \( \lfloor n/p \rfloor \) elements
• We seek simple expressions as we must be able to find:
  – Low & high indices in block for each PE
  – The owner of a given an index

Method #1

• Let \( r = n \mod p \)
• If \( r == 0 \), all blocks have same size
• Else
  – First \( r \) blocks have size \( \lceil n/p \rceil \)
  – Remaining \( p-r \) blocks have size \( \lfloor n/p \rfloor \)
• Example: \( p = 8 \) and \( n = 45 \)
  Observe that \( r = 45 \mod 8 = 5 \)
  So first 5 blocks have size \( \lceil 45/8 \rceil = 6 \) and
  the \( p-r = 8-5 = 3 \) others have size \( \lfloor 45/8 \rfloor = 5 \).
  We’ve divided 45 items into 8 blocks as follows:
  5 blocks of 6 items, then 3 blocks of 5 items
Examples

17 elements divided among 7 processes

17 elements divided among 5 processes

17 elements divided among 3 processes

Method #1 Calculations

- Let \( r = n \mod p \)
- The first element controlled by process \( i \) is
  \[ i \left\lfloor \frac{n}{p} \right\rfloor + \min(i, r) \]
- Example: The first element controlled by process 1 is \( 1 \times 3 + \min(1, 2) = 4 \) in below example:

17 elements divided among 5 processes
Method #2

- Scatters larger blocks among processes
  - Not all given to PEs with lowest indices
- First element controlled by process $i$ is
  \[ \left\lfloor \frac{in}{p} \right\rfloor \]
- Last element controlled by process $I$ is
  \[ \left\lfloor \frac{(i+1)n}{p} \right\rfloor - 1 \]
- Process controlling element $j$ is
  \[ \left\lfloor \frac{(p(j+1) - 1)}{n} \right\rfloor \]

Example: 17 tasks, 5 processes
- First element controlled by process $i$ will be
  \[ \left\lfloor \frac{in}{p} \right\rfloor = \left\lfloor \frac{17i}{5} \right\rfloor \]

<table>
<thead>
<tr>
<th>$P_i$</th>
<th>1$^{st}$ of $P_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
</tbody>
</table>

17 elements divided among 5 processes
Method #2

- Example: Find process controlling element 7 when 17 elements are divided among 5 processes.
  
  - Recall formula: \[
  \left\lfloor \frac{\left( p \left( \frac{j}{j+1} \right) - 1 \right)}{n} \right\rfloor
  \]

  17 elements divided among 5 processes
  
  The process controlling element 7 is \( \left\lfloor \frac{(5*8-1)/17} \right\rfloor = \left\lfloor \frac{39/17} \right\rfloor = 2 \)

  Note this involves only 1 division.

Some Examples

17 elements divided among 7 processes

17 elements divided among 5 processes

17 elements divided among 3 processes
Comparing Methods

<table>
<thead>
<tr>
<th>Operations</th>
<th>Method 1</th>
<th>Method 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low index</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>High index</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Owner</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Assuming no operations for "floor" function

Another Example

• Illustrate how block decomposition method #2 would divide 13 elements among 5 processes.

  13(0)/5 = 0  13(2)/5 = 5  13(4)/5 = 10

  13(1)/5 = 2  13(3)/5 = 7
Macros in C

- A macro (in any language) is an in-line routine that is expanded at compile time.
- Function-like macros can take arguments, just like true functions.
- To define a macro that uses arguments, you insert parameters between a pair of parentheses in the macro definition.
- The parameters must be valid C identifiers, separated by commas and optionally whitespace.
- Typically macro functions are written in all caps.

Short if-then-else in C

The construct in C of

\[ \text{logical} \ ? \ \text{if-part} : \ \text{then-part} \]

For example,

\[ a = (x < y) \ ? \ 3 \ : \ 4; \]

is equivalent to

\[ \text{if } x < y \text{ then } a = 3 \text{ else } a = 4; \]
Example of a C Macro

• `#define MIN(X, Y) ((X) < (Y) ? (X) : (Y))`

• This macro is invoked (i.e. expanded) at compile time by strict text substitution:
  • `x = MIN(a, b);` \(\Rightarrow\) `x = ((a) < (b) ? (a) : (b));`
  • `y = MIN(1, 2);` \(\Rightarrow\) `y = ((1) < (2) ? (1) : (2));`
  • `z = MIN(a + 28, *p);` \(\Rightarrow\) `z = ((a + 28) < (*p) ? (a + 28) : (*p));`

Define Block Decomposition Macros

`#define BLOCK_LOW(id,p,n) ((id)*(n)/(p))`
Given id, p, and n, this expands to the lowest index controlled by process id.

`#define BLOCK_HIGH(id,p,n) \
(BLOCK_LOW((id)+1,p,n)-1)`
Given id, p, and n, this expands to the highest index controlled by process id.
Define Block Decomposition Macros

#define BLOCK_SIZE(id,p,n) \
(BLOCK_LOW((id)+1)- \
BLOCK_LOW(id))

Given id, p, and n this expands to the size of the block controlled by id.

#define BLOCK_OWNER(index,p,n) \
(((p)*(index)+1)-1)/(n))

Given index, p, and n this expands to the process id that controls the given index.

Local vs. Global Indices

Note: We need to distinguish between these
Example: Looping over Elements

- **Sequential program**
  
  ```
  for (i = 0; i < n; i++) {
    ...
  }
  ```

- **Parallel program**
  
  ```
  size = BLOCK_SIZE (id,p,n);
  for (i = 0; i < size; i++) {
    gi = i + BLOCK_LOW(id,p,n);
  }
  ```

  ...takes place of sequential program’s index. Think of this as the global index.

---

Fast Marking

- **Block decomposition allows for the same marking as the sequential algorithm**, but it is sped up:
  - We don't check each array element to see if it is a multiple of k (which requires n/p modulo operations within each block for each prime)

- **Instead within each block**
  - Find the first multiple of k, say cell $j$
  - Then mark the cells $j$, $j + k$, $j + 2k$, $j + 3k$, ...

- **This allows a loop similar to the one in the sequential program**
  - Requires about $(n/p)k$ assignment statements
Decomposition Affects Implementation

- Largest prime used by sieve algorithm is bounded by $\sqrt{n}$
- First process has $\lfloor n/p \rfloor$ elements
  - If $n/p > \sqrt{n}$, then the first process controls all primes through $\sqrt{n}$
  - Normally $n$ is much larger than $p$, so this will be the case
- Consequently, in this case, the first process can broadcast the next sieving prime and no reduction step is needed
  - Example: $17/3 = 5.666\ldots > \sqrt{17}$ and 2, 3, 5 are in 1st block:
    
    | Process 0 | Process 1 | Process 2 |
    |-----------|-----------|-----------|
    | [ ]       | [ ]       | [ ]       |

Convert the Sequential Algorithm to a Parallel Algorithm

1. Create list of unmarked natural numbers 2, 3, …, $n$
2. $k \leftarrow 2$
3. Repeat
   - Each process creates its share of list
   - Each process does this
   - Each process marks its share of list
     - (a) Mark all multiples of $k$ between $k^2$ and $n$
     - (b) $k \leftarrow$ smallest unmarked number $\geq k$
     - Process 0 only
     - (c) Process 0 broadcasts $k$ to rest of processes
   until $k^2 > n$
4. The unmarked numbers are primes
5. Reduction to determine number of primes found
Function MPI_Bcast

```c
int MPI_Bcast (  
    void *buffer, /* Addr of 1st element */  
    int count,    /* # elements to broadcast */  
    MPI_Datatype datatype, /* Type of elements */  
    int root,     /* ID of root process */  
    MPI_Comm comm) /* Communicator */  
) /* MPI_Bcast */
```

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

Task/Channel Graph for 4 Processors

Red are I/O channels

Straight lines – channels used in the reduction step

Curved lines – channels used in broadcast step
Task/Channel Model Added Assumption

- The analysis of algorithms typically performed assumes that this model supports the concurrent transmission of messages from multiple tasks, as long as:
  - they use different channels
  - no two active channels have the same source or destination
- This is claimed to be a reasonable assumption
  - based on current commercial systems
  - for some clusters
- This is not a reasonable assumption for networks of workstations connected by hub or any communications systems supporting only one message at a time
- This assumption is not reasonable for many communication-intensive applications

Analysis

- \( \chi \) (i.e., chi) is time needed to mark a cell
- Sequential execution time: \( \sim \chi n \ln \ln n \)
- Number of broadcasts: \( \sim \sqrt{n} / \ln \sqrt{n} \)
- Broadcast time: \( \lambda \left\lceil \log p \right\rceil \) with \( \lambda \) latency
- Expected execution time:

\[
\frac{\chi n \ln \ln n}{p} + \left( \sqrt{n} / \ln \sqrt{n} \right) \lambda \left\lceil \log p \right\rceil
\]

This uses the fact that the number of primes between 2 and \( n \) is about \( n/\ln n \). So, a good approximation to the number of loop iterations is the term underlined above.
Code for Sieve of Eratosthenes

(Complete code starts on page 124)

```c
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include "MyMPI.h"
#define MIN(a,b) ((a)<(b)?(a):(b))

int main (int argc, char *argv[]) {
    /* Bunch of data declarations here */
    MPI_Init (&argc, &argv);
    /* Start timer here */
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();

    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
```
Capturing Command Line Values

Example: Invoking the UNIX compiler mpicc

```
mpicc -o myprog myprog.c
```

would result in the following values being passed to mpicc:

- `argc` 4 - number of tokens on command line – an int
- `argv[0]` mpicc each `argv[i]` is a char array
- `argv[1]` -o
- `argv[2]` myprog i.e., name for object file
- `argv[3]` myprog.c i.e., source file

Improvements

- Delete even integers
  - Cuts number of computations in half
  - Frees storage for larger values of $n$
  - Cuts the execution time almost in half
- Each process finds own sieving primes
  - Replicating computation of primes to $\sqrt{n}$
  - Eliminates about $\sqrt{n} / \ln \sqrt{n}$ broadcast steps
- Reorganize loops
  - As designed, the algorithm is marking widely dispersed elements of a very large array
  - Changing this can increase the cache hit rate
Reorganize Loops

Suppose cache has 4 lines of 4 bytes each. So

line 1 holds 3, 5, 7, 9
line 2 holds 11, 13, 15, 17 etc.

Then if we sieve all the multiples of one prime before doing the next one, all of the yellow numbers will be cache misses. Note: Multiples of 2 are already not included.

multiples of 3: 9 15 21 27 33 39 45 51 57 63 69 75 81 87 93 99
multiples of 5: 25 35 45 55 65 75 85 95
multiples of 7: 49 63 77 91

Reorganize Loops

Now use 8 bytes in two cache lines and sieve multiples of all primes for the first 8 bytes before going to the next 8 bytes. Again yellow numbers show cache misses:

3-17: Multiples of 3: 9 15
19-33: Multiples of 3, 5: 21 27 33 25
35-49: Multiples of 3, 4, 7: 39 45 35 45 49
51-65: Multiples of 3, 5, 7: 51 57 63 55 65 63
67-81: Multiples of 3, 5, 7: 63 69 75 81 75 77
83-97: Multiples of 3, 5, 7: 87 93 85 95 92
99: Multiples of 3, 5, 7: 99
Comparing (as shown in text)

Comparing 4 Versions

<table>
<thead>
<tr>
<th>Procs</th>
<th>Sieve 1</th>
<th>Sieve 2</th>
<th>Sieve 3</th>
<th>Sieve 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.900</td>
<td>12.237</td>
<td>12.466</td>
<td>2.543</td>
</tr>
<tr>
<td>3</td>
<td>8.843</td>
<td>5.019</td>
<td>4.272</td>
<td>0.901</td>
</tr>
<tr>
<td>4</td>
<td>6.768</td>
<td>4.072</td>
<td>3.267</td>
<td>0.379</td>
</tr>
<tr>
<td>5</td>
<td>5.794</td>
<td>3.652</td>
<td>2.559</td>
<td>0.543</td>
</tr>
<tr>
<td>6</td>
<td>4.964</td>
<td>3.270</td>
<td>2.127</td>
<td>0.456</td>
</tr>
<tr>
<td>7</td>
<td>4.371</td>
<td>3.059</td>
<td>1.820</td>
<td>0.391</td>
</tr>
<tr>
<td>8</td>
<td>3.927</td>
<td>2.856</td>
<td>1.585</td>
<td>0.342</td>
</tr>
</tbody>
</table>

Note: Graphical display of this chart in Fig. 5.10.
Parallel Speedup Metric
(Initial Overview)

• A measure of the increase in running time due to parallelism.

• Speedup = \( \frac{\text{sequential time}}{\text{parallel time}} \)
  – The sequential time is the worst case sequential running time
  – The parallel time is the worst case parallel running time.