Finding Primes

- Analysis of block allocation schemes
- Function MPI_Bcast
- Performance enhancements
Outline

- Sequential algorithm
- Sources of parallelism
- Data decomposition options
- Parallel algorithm development, analysis
- MPI program
- Benchmarking
- Optimizations
Sequential Algorithm

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

Complexity: $\Theta(n \ln \ln n)$
Pseudocode

1. Create list of unmarked natural numbers 2, 3, …, n
2. \( k \leftarrow 2 \)
3. Repeat
   (a) Mark all multiples of \( k \) between \( k^2 \) and \( n \)
   (b) \( k \leftarrow \) smallest unmarked number > \( k \)
   until \( k^2 > n \)
4. The unmarked numbers are primes
Sources of Parallelism

- **Domain decomposition**
  - Divide data into pieces
  - Associate computational steps with data
- **One primitive task per array element**
Making it Parallel (3a)

Mark all multiples of $k$ between $k^2$ and $n$

\[ \Rightarrow \]

for all $j$ where $k^2 \leq j \leq n$ do
  if $j \mod k = 0$ then
    mark $j$ (it is not a prime)
  endif
endfor
Making it Parallel (3b)

Find smallest unmarked number > \( k \)

⇒

Min-reduction (to find smallest unmarked number > \( k \))

Broadcast (to get result to all tasks)
Agglomeration Goals

- Consolidate tasks
- Reduce communication cost
- Balance computations among processes
Data Decomposition Options

- Interleaved (cyclic)
  - Easy to determine “owner” of each index
  - Leads to load imbalance *for this problem*

- Block
  - Balances loads
  - More complicated to determine owner if \( n \) not a multiple of \( p \)
Block Decomposition Options

- Want to balance workload when $n$ not a multiple of $p$
- Each process gets either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ elements
- Seek simple expressions
  - Find low, high indices given an owner
  - Find owner 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 $\lfloor n/p \rfloor$
Examples

17 elements divided among 7 processes

17 elements divided among 5 processes

17 elements divided among 3 processes
Method #1 Calculations

- **First element controlled by process** $i$
  \[ i\left\lfloor \frac{n}{p} \right\rfloor + \min(i, r) \]

- **Last element controlled by process** $i$
  \[ (i + 1)\left\lfloor \frac{n}{p} \right\rfloor + \min(i + 1, r) - 1 \]

- **Process controlling element** $j$
  \[ \min\left(\left\lfloor \frac{j}{\left\lfloor \frac{n}{p} \right\rfloor + 1} \right\rfloor, \left\lfloor \frac{j - r}{\left\lfloor \frac{n}{p} \right\rfloor} \right\rfloor\right) \]
Method #2

- Scatters larger blocks among processes
- First element controlled by process $i$
  \[
  \left\lfloor \frac{in}{p} \right\rfloor
  \]
- Last element controlled by process $i$
  \[
  \left\lfloor \frac{(i+1)n}{p} \right\rfloor - 1
  \]
- Process controlling element $j$
  \[
  \left\lfloor \frac{p(j+1) - 1}{n} \right\rfloor
  \]
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

Our choice
Pop Quiz

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

\[
\begin{align*}
13(0)/5 &= 0 & 13(2)/5 &= 5 & 13(4)/5 &= 10 \\
13(1)/5 &= 2 & 13(3)/5 &= 7
\end{align*}
\]
Block Decomposition Macros

#define BLOCK_LOW(id,p,n)   ((i)*(n)/(p))

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

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

#define BLOCK_OWNER(index,p,n) \  
  (((p)*(index)+1)-1)/(n))
Local vs. Global Indices

- Local Indices:
  - L 0 1
  - G 0 1
  - L 0 1
  - G 5 6
  - L 0 1 2
  - G 7 8 9

- Global Indices:
  - G 2 3 4
  - G 10 11 12
Looping over Elements

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

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

  Index $i$ on this process…

  …takes place of sequential program’s index $gi$
Decomposition Affects Implementation

- Largest prime used to sieve is $\sqrt{n}$
- First process has $\lfloor n/p \rfloor$ elements
- It has all sieving primes if $p < \sqrt{n}$
- First process always broadcasts next sieving prime
- No reduction step needed
Fast Marking

- Block decomposition allows same marking as sequential algorithm:

\[ j, j + k, j + 2k, j + 3k, \ldots \]

instead of

for all \( j \) in block

\[ \text{if } j \mod k = 0 \text{ then mark } j \text{ (it is not a prime)} \]
Parallel Algorithm Development

1. Create list of unmarked natural numbers 2, 3, …, n
2. $k \leftarrow 2$
3. Repeat
   (a) Mark all multiples of $k$ between $k^2$ and $n$
   (b) $k \leftarrow$ smallest unmarked number $> k$
   (c) Process 0 broadcasts $k$ to rest of processes
4. The unmarked numbers are primes
5. Reduction to determine number of primes
Function MPI_Bcast

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 (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
Analysis

- $\chi$ is time needed to mark a cell
- Sequential execution time: $\chi \cdot n \ln \ln n$
- Number of broadcasts: $\sqrt{n} / \ln \sqrt{n}$
- Broadcast time: $\lambda \left\lceil \log p \right\rceil$
- Expected execution time:

$$\chi n \ln \ln n / p + (\sqrt{n} / \ln \sqrt{n}) \lambda \left\lceil \log p \right\rceil$$
```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[])
{
    ...
    MPI_Init (&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    if (argc != 2) {
        if (!id) printf ("Command line: %s <m>\n", argv[0]);
        MPI_Finalize(); exit (1);
    }
}```
n = atoi(argv[1]);
low_value = 2 + BLOCK_LOW(id,p,n-1);
high_value = 2 + BLOCK_HIGH(id,p,n-1);
size = BLOCK_SIZE(id,p,n-1);
proc0_size = (n-1)/p;
if ((2 + proc0_size) < (int) sqrt((double) n)) {
    if (!id) printf ("Too many processes\n");
    MPI_Finalize();
    exit (1);
}

marked = (char *) malloc (size);
if (marked == NULL) {
    printf ("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit (1);
}
for (i = 0; i < size; i++) marked[i] = 0;
if (!id) index = 0;
prime = 2;
do {
    if (prime * prime > low_value)
        first = prime * prime - low_value;
    else {
        if (!(low_value % prime)) first = 0;
        else first = prime - (low_value % prime);
    }
    for (i = first; i < size; i += prime) marked[i] = 1;
    if (!id) {
        while (marked[++index]);
        prime = index + 2;
    }
} MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while (prime * prime <= n);
count = 0;
for (i = 0; i < size; i++)
    if (!marked[i]) count++;
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
if (!id) {
    printf "%d primes are less than or equal to %d\n",
        global_count, n);
    printf "Total elapsed time: %.6f\n", elapsed_time);
}
MPI_Finalize ();
return 0;
Benchmarking

- Execute sequential algorithm
- Determine $\chi = 85.47$ nanosec
- Execute series of broadcasts
- Determine $\lambda = 250$ $\mu$sec
<table>
<thead>
<tr>
<th>Processors</th>
<th>Predicted</th>
<th>Actual (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.900</td>
<td>24.900</td>
</tr>
<tr>
<td>2</td>
<td>12.721</td>
<td>13.011</td>
</tr>
<tr>
<td>3</td>
<td>8.843</td>
<td>9.039</td>
</tr>
<tr>
<td>4</td>
<td>6.768</td>
<td>7.055</td>
</tr>
<tr>
<td>5</td>
<td>5.794</td>
<td>5.993</td>
</tr>
<tr>
<td>6</td>
<td>4.964</td>
<td>5.159</td>
</tr>
<tr>
<td>7</td>
<td>4.371</td>
<td>4.687</td>
</tr>
<tr>
<td>8</td>
<td>3.927</td>
<td>4.222</td>
</tr>
</tbody>
</table>
Improvements

- Delete even integers
  - Cuts number of computations in half
  - Frees storage for larger values of $n$
- Each process finds own sieving primes
  - Replicating computation of primes to $\sqrt{n}$
  - Eliminates broadcast step
- Reorganize loops
  - Increases cache hit rate
Reorganize Loops

- The cache effect!

- Instead of marking for each prime,

- Mark with multiple primes using the same integers in the cache
Reorganize Loops

- 4 lines of 4 byte cache
- Marking with 3
  - 3 5 7 9
  - 11 13 15 17
  - 19 21 23 25
  - 27 29 31 33 hit!
- 35 37 39 41
- 43 45 47 49
- 51 53 55 57 hit!

- 4 lines of 4 byte cache
- Marking with 3,5
  - 3 5 7 9
  - 11 13 15 17
  - 19 21 23 25 hit!
  - 27 29 31 33 hit!
  - 35 37 39 41 hit!
  - 43 45 47 49 hit!
  - 51 53 55 57 hit!
Reorganize Loops

- 3-99: multiples of 3
  
  9 15 21 27 33 39 45 51 57 63 69 75 81 87 93 99

- 3-99: multiples of 5
  
  25 35 45 55 65 75 85 95

- 3-99: multiples of 7
  
  49 63 77 91

- 3-17: multiples of 3
  
  9 15

- 19-33: multiples of 3,5
  
  21 27 33 25

- 35-49: multiples of 3,5,7
  
  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
  
  69 75 81 75 77

- 83-97: multiples of 3,5,7
  
  87 93 85 95 91

- 99: multiples of 3,5,7
  
  99
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>3.201</td>
<td>0.679</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>

10-fold improvement

7-fold improvement
Summary

- Sieve of Eratosthenes: parallel design uses domain decomposition
- Compared two block distributions
  - Chose one with simpler formulas
- Introduced `MPI_Bcast`
- Optimizations reveal importance of maximizing single-processor performance
Homework

■ Find 20 primes whose string include your student number.
■ For example, if your student number is 2000-10101,
  ◆ the prime should be of the form X200010101Y
  ◆ Where X and Y could be any arbitrary numbers.
■ Please hand in your design report (due 9/28)
  ♦ Partitioning/communication/agglomeration/mapping
  ◆ As well as the final report (due 9/30)
■ If your student number is 8 digit, put 1 in front
  ◆ 97419-001 => 197419001