# Parallel Programming
## in C with MPI and OpenMP

Michael J. Quinn

# Chapter 5

## The Sieve of Eratosthenes

# Chapter Objectives

- 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

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

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 3(a) Parallel

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 3(b) Parallel

Find smallest unmarked number > $k$

$\Rightarrow$

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

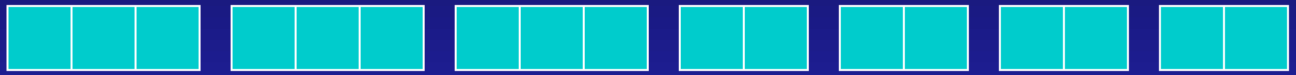
- Want to balance workload when $n$ not a multiple of $p$
- Each process gets either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ elements
- Seek simple expressions
  - Find low, high indices given an owner
  - Find owner given an index

# Method #1

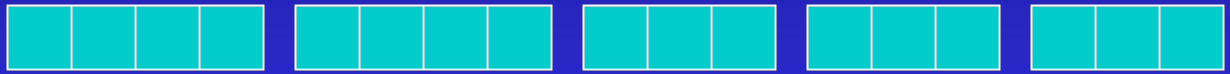
- Let $r = n \bmod p$
- If $r = 0$, all blocks have same size
- Else
  - First $r$ blocks have size $\lceil n/p \rceil$
  - Remaining $p$-$r$ blocks have size $\lfloor n/p \rfloor$

# Examples

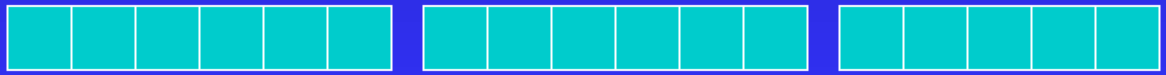
17 elements divided among 7 processes



17 elements divided among 5 processes



17 elements divided among 3 processes



# Method #1 Calculations

- First element controlled by process *i*

$$i\lfloor n/p \rfloor + \min(i, r)$$

- Last element controlled by process *i*

$$(i+1)\lfloor n/p \rfloor + \min(i+1, r) - 1$$

- Process controlling element *j*

$$\min(\lfloor j/(\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j-r)/\lfloor n/p \rfloor \rfloor)$$

# Method #2

- Scatters larger blocks among processes
- First element controlled by process *i*

$$\lfloor in/p \rfloor$$

- Last element controlled by process *i*

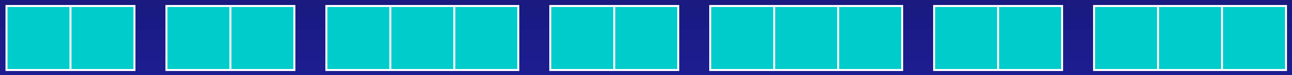
$$\lfloor (i+1)n/p \rfloor - 1$$

- Process controlling element *j*

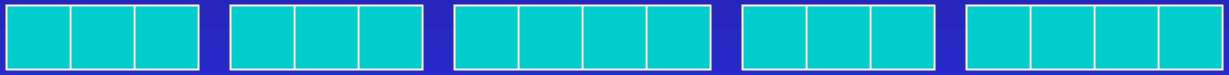
$$\lfloor (p(j+1)-1)/n \rfloor$$

# Examples

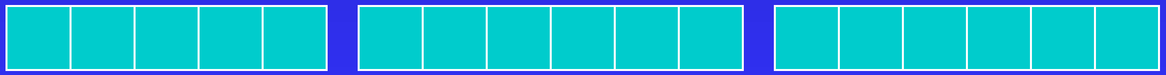
17 elements divided among 7 processes



17 elements divided among 5 processes



17 elements divided among 3 processes



# Comparing Methods

Our choice

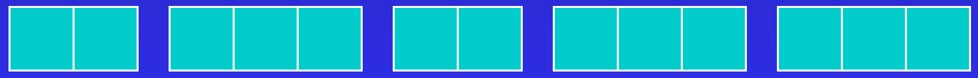
| **Operations** | **Method 1** | **Method 2** |
| --- | --- | --- |
| Low index | 4 | 2 |
| High index | 6 | 4 |
| Owner | 7 | 4 |

Assuming no operations for “floor” function

# Pop Quiz

- 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

# Block Decomposition Macros

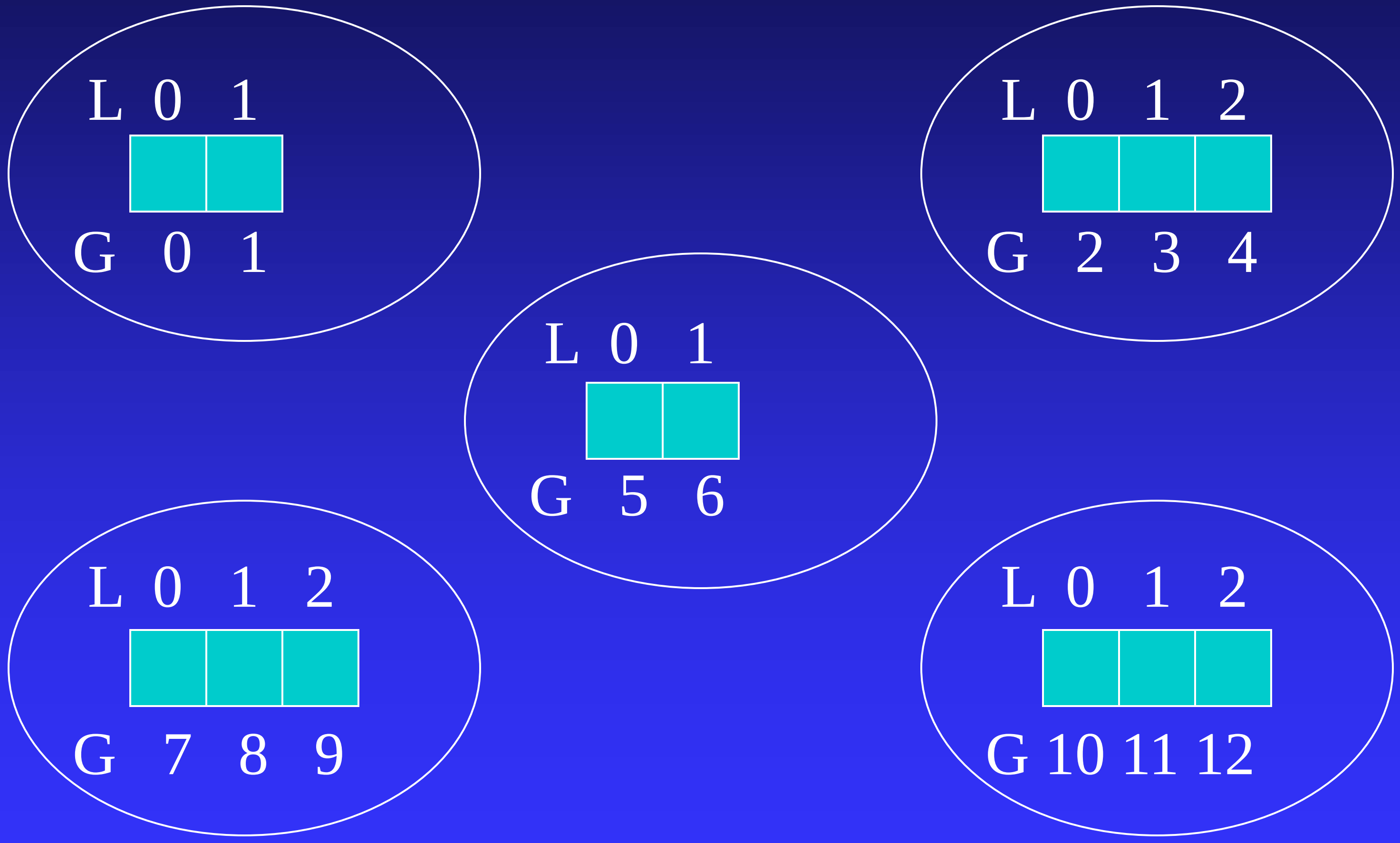
```
#define BLOCK_LOW(id,p,n)  ((i)*(n)/(p))

#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



# 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);
}
```

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
  
      if $j \bmod k = 0$ then mark $j$ (it is not a prime)

# Parallel Algorithm Development

1. Create list of unmarked natural numbers 2, 3, …, n

Each process creates its share of list

2. $k \leftarrow 2$

Each process does this

3. Repeat

Each process marks its share of list

(a) Mark all multiples of $k$ between $k^2$ and $n$

(b) $k \leftarrow$ smallest unmarked number $> k$

Process 0 only

(c) Process 0 broadcasts $k$ to rest of processes

until $k^2 > m$

4. The unmarked numbers are primes

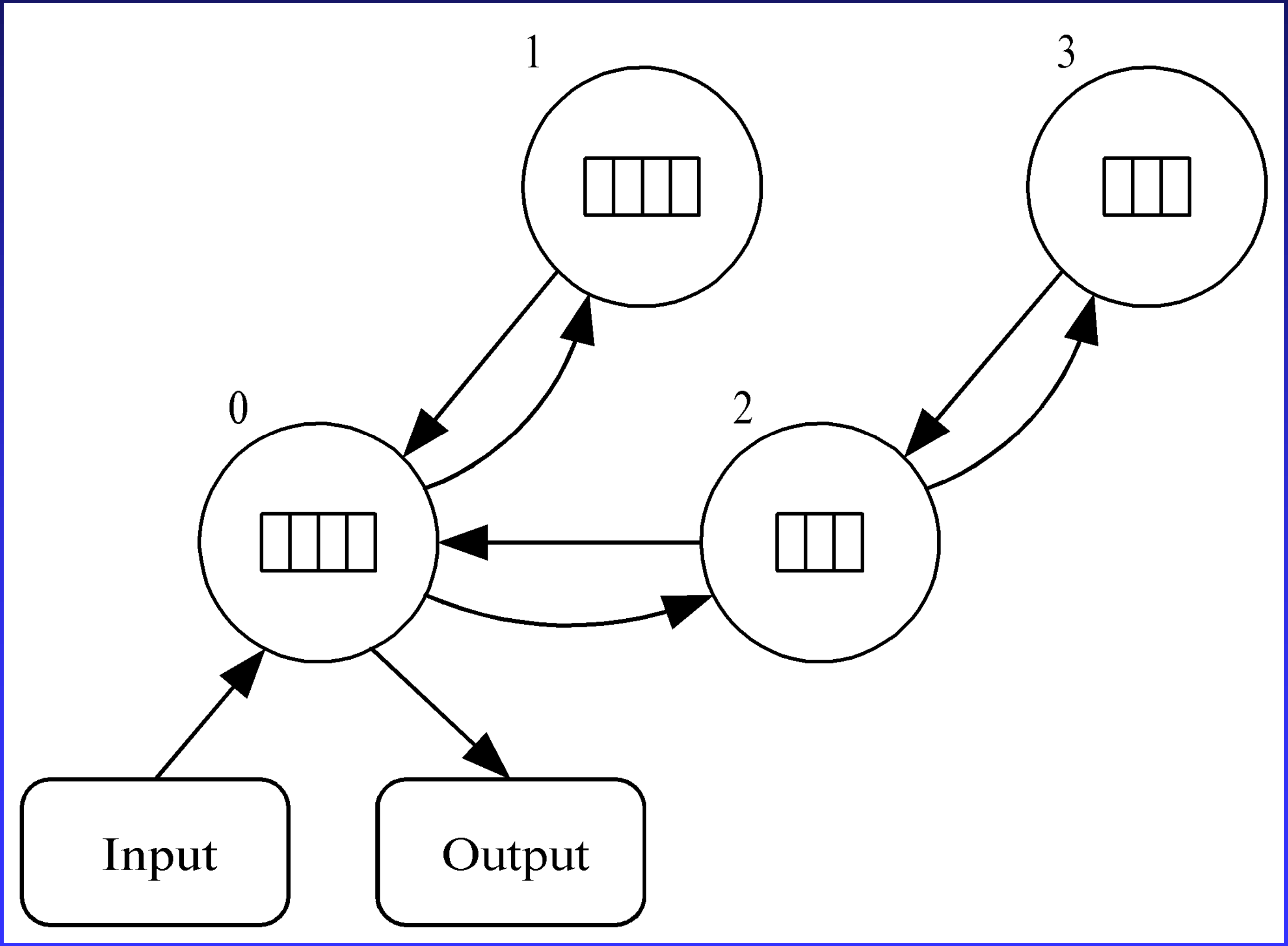
5. Reduction to determine number of primes

# 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);
```

# Task/Channel Graph



# Analysis

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

$$\chi n \ln \ln n / p + (\sqrt{n} / \ln \sqrt{n}) \lambda \lceil \log p \rceil$$

# Code (1/4)

```
#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);
}
```

# Code (2/4)

```
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);
}
```

# Code (3/4)

```
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);
```

# Code (4/4)

```
   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: %10.6f\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

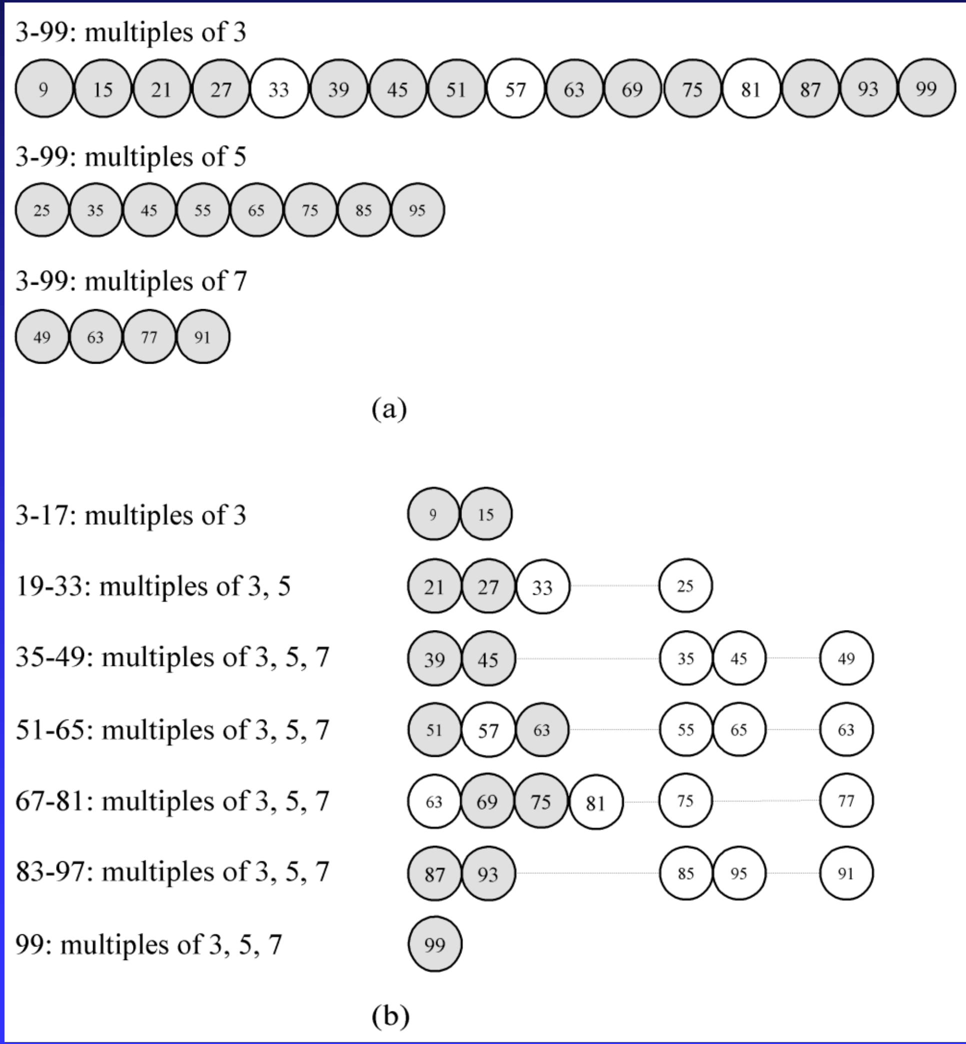
# Execution Times (sec)

| Processors | Predicted | Actual (sec) |
|---|---|---|
| 1 | 24.900 | 24.900 |
| 2 | 12.721 | 13.011 |
| 3 | 8.843 | 9.039 |
| 4 | 6.768 | 7.055 |
| 5 | 5.794 | 5.993 |
| 6 | 4.964 | 5.159 |
| 7 | 4.371 | 4.687 |
| 8 | 3.927 | 4.222 |

# 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




Lower

Cache hit rate

Higher

# Comparing 4 Versions

| *Procs* | *Sieve 1* | *Sieve 2* | *Sieve 3* | *Sieve 4* |
|---|---|---|---|---|
| 1 | 24.900 | 12.237 | 12.466 | 2.543 |
| 2 | 12.721 | 6.609 | 6.378 | 1.330 |
| 3 | 8.843 | 5.019 | 4.272 | 0.901 |
| 4 | 6.768 | 4.072 | 3.201 | 0.679 |
| 5 | 5.794 | 3.652 | 2.559 | 0.543 |
| 6 | 4.964 | 3.270 | 2.127 | 0.456 |
| 7 | 4.371 | 3.059 | 1.820 | 0.391 |
| 8 | 3.927 | 2.856 | 1.585 | 0.342 |

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