Sieve of Eratosthenes

Complexity: $\Theta(n \ln \ln n)$
Pseudo-code

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
3. Parallelizing the Algorithm

3.a Mark all multiples of $k$ between $k^2$ and $n$

⇒

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

3.b Find smallest unmarked number $> k$

⇒

Min-reduction (to find smallest unmarked number $> k$)
Broadcast (to get result to all tasks)
Block Decomposition Method #1

Let \( r = n \mod p \)

If \( r = 0 \), all blocks have same size

Else

- First \( r \) blocks have size \( n/p \)
- Remaining \( p-r \) blocks have size \( n/p \)
Block Decomposition 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 $j$ \[ \left \lfloor \frac{p(j+1) - 1}{n} \right \rfloor \]

17 elements divided among 7 processes

17 elements divided among 5 processes

17 elements divided among 3 processes

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
## Comparing Block Decompositions

<table>
<thead>
<tr>
<th>Operation</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
Block Decomposition Macros

```c
#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))
```

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Local and Global Indices

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Looping with Local/Global Index

- Sequential program
  
  \[
  \text{for } (i = 0; i < n; i++) \{ \\
  \text{...} \\
  \}
  \]

- Parallel program
  
  \[
  \text{size} = \text{BLOCK\_SIZE} (id, p, n); \\
  \text{for } (i = 0; i < \text{size}; i++) \{ \\
  \text{\(gi\)} = i + \text{BLOCK\_LOW} (id, p, n); \\
  \}
  \]

\[\text{Index } i \text{ on this process...}\]

\[\text{...takes place of sequential program’s index } gi\]
Parallel Algorithm

1. Create list of unmarked natural numbers 2, 3, …, n
2. \( k = 2 \)  
   Each process creates its share of list
   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) \( k = \) 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

\[
\chi \left( n \ln \ln n \right) / p + \left( \sqrt{n} / \ln \sqrt{n} \right) \lambda \left\lfloor \log p \right\rfloor
\]
Task/Channel Graph
#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: %10.6f\n", elapsed_time);
}
MPI_Finalize ();
return 0;
Optimizations

• 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
Loop Reorganization

3-99: multiples of 3

3-99: multiples of 5

3-99: multiples of 7

(a)

3-17: multiples of 3

19-33: multiples of 3, 5

35-49: multiples of 3, 5, 7

51-65: multiples of 3, 5, 7

67-81: multiples of 3, 5, 7

83-97: multiples of 3, 5, 7

99: multiples of 3, 5, 7

(b)

Figure Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
User Defined Datatypes

- Methods for creating data types
  
  
  MPI_Type_contiguous();
  MPI_Type_vector();
  MPI_Type_indexed();
  MPI_Type_struct();
  MPI_Pack();
  MPI_Unpack();

- MPI datatypes defined similar to modern programming languages (C, C++, F90)
- Allows communication and I/O operations to use the same datatypes as rest of program
- Makes expressing the partitioning of datasets easier
Some datatype terminology

- Every MPI datatype has a few characteristics
  - type signature
    - list of the basic datatypes (in order) that make up the derived type
  - type map
    - basic datatypes
    - lower bound of each type
    - extent of the type (size + buffering)
- Some of this information is available about MPI datatypes through:
  - MPI_Get_extent
  - MPI_Get_size
Contiguous Array

• Creates an array of counts elements:

  MPI_Type_contiguous(int count,
                         MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
Strided Vector

• Constructs a cyclic set of elements

```c
MPI_Type_vector(int count,
    int blocklength,
    int stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

• Stride specified in number of elements
• Stride can be specified in bytes
  ```c
  MPI_Type_hvector();
  ```
• Stride counts from **start** of block
Indexed Vector

• Allows an irregular pattern of elements
  MPI_Type_indexed(int count,
                   int *array_of_blocklengths,
                   int *array_of_displacements,
                   MPI_Datatype oldtype,
                   MPI_Datatype *newtype);

• Displacements specified in number of elements
  – Displacements can be specified in bytes
    MPI_Type_hindexed();

• A shortcut if all blocks are the same length:
  MPI_Type_create_indexed_block();
Structured Records

- Allows different types to be combined
  ```c
  MPI_Typestruct(int count,
    int *array_of_blocklengths,
  MPI_Aint *array_of_displacements,
  MPI_Datatype *array_of_types,
  MPI_Datatype *newtype);
  ```
- Blocklengths specified in number of elements
- Displacements specified in bytes
Committing types

- In order for a user-defined derived datatype to be used as an argument to other MPI calls, the type must be “committed”.

\[
\text{MPI\_Type\_commit}(\text{type});
\]
\[
\text{MPI\_Type\_free}(\text{type});
\]

- Use commit after calling the type constructor, but before using the type anywhere else
- Call free after the type is no longer in use (no one actually does this, but it makes computer scientists happy...)
Pack and Unpack

- Packs sparse structures into contiguous memory

```c
MPI_Pack(void* inbuf, int incount,
    MPI_Datatype datatype,
    void *outbuf, int outsize,
    int *position, MPI_Comm comm);

MPI_Unpack(void* inbuf, int insize,
    int *position, void *outbuf,
    int outcount,
    MPI_Datatype datatype,
    MPI_Comm comm);
```
Dealing with Groups

- A group is a set of tasks
- Groups are used to construct communicators
- Group accessors:
  
  ```
  int MPI_Group_size(MPI_Group group, int *size)
  int MPI_Group_rank(MPI_Group group, int *rank)
  int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
  int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
  ```
Creating Groups

• Group constructors:

  int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
  int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
  int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
  int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
  int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
  int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
  int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
  int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
Destroying Groups

- Group destructors:
  ```c
  int MPI_Group_free(MPI_Group *group)
  ```
Dealing with Communicators

- MPI collective operations deal with all the processes in a communicator
- MPI_COMM_WORLD by default contains every task in your MPI job
- Other communicators can be defined to allow collective operations on a subset of the tasks
- Communicator Accessors:
  - int MPI_Comm_size(MPI_Comm comm, int *size)
    - Returns the size of the group in comm
  - int MPI_Comm_rank(MPI_Comm comm, int *rank)
    - Returns the rank of the caller in that communicator
  - int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
    - Returns if two communicators are the same, similar(same tasks but different ranks), or different
Creating Communicators

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
- Creates an exact copy of the communicator

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
- Creates a new communicator with the contents of group
  - Group must be a subset of Comm

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
- Creates a communicator for each distinct value of color, ranked by key
Destroying Communicators

int MPI_Comm_free(MPI_Comm comm)

- Destroys the named communicator
Topologies and Communicators

• MPI allows processes to be grouped in logical topologies
• Topologies can aid the programmer
  – Convenient naming methods for processes in a group
  – Naming can match communication patterns
  – A standard mechanism for representing common algorithmic concepts (i.e. 2D grids)
• Topologies can aid the runtime environment
  – Better mappings of MPI tasks to hardware nodes
  – Not really useful in a simple cluster environment....
Cartesian Topologies

```c
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
    int *dims, int *periods, int reorder, MPI_Comm
    *comm_cart)
```

- `comm_old` - input communicator
- `ndims` - # of dimensions in cartesian grid
- `dims` - integer array of size `ndims` specifying the number of processes in each dimension
- `periods` - true/false specifying whether each dimension is periodic (wraps around)
- `reorder` - ranks may be reordered or not
- `comm_cart` - new communicator containing new topology.
**MPI_DIMS_CREATE**

- A helper function for specifying a likely dimension decomposition.

  ```c
  int MPI_Dims_create(int nnodes, int ndims, int *dims)
  ```

  - `nnodes` - total nodes in grid
  - `ndims` - number of dimensions
  - `dims` - array returned with dimensions

- **Example:**

  ```c
  MPI_Dims_create(6,2,dims)
  ```

  - will return (3,2) in dims

  ```c
  MPI_Dims_create(6,3,dims)
  ```

  - will return (3,2,1) in dims

- **No rounding or ceiling function provided**
Cartesian Inquiry Functions

- `MPI_Cartdim_get` will return the number of dimensions in a Cartesian structure
  
  ```c
  int MPI_Cartdim_get(MPI_Comm comm, int *ndims);
  ```

- `MPI_Cart_get` provides information on an existing topology
  - Arguments roughly mirror the create call
    ```c
    int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords);
    ```
  - Maxdims keeps a given communicator from overflowing your arguments
Cartesian Translator Functions

- Task IDs in a Cartesian coordinate system correspond to ranks in a "normal" communicator.
  - point-to-point communication routines (send/receive) rely on ranks
    
    ```c
    int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
    int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
    ```

- Coords - cartesian coordinates
- rank - ranks
Cartesian Shift function

int MPI_Cart_Shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

- direction - coordinate dimension of shift
- disp - displacement (can be positive or negative)
- rank_source and rank_dest are return values
  - Use that source and dest to call MPI_Sendrecv
Cartesian Shift Example

```c
MPI_Comm ICOMM;
MPI_Status status;
int NY, srank, rrank;
int idims[2] = {4, 4};
int periods[2] = {1, 1};
void *plane1, *plane2;

MPI_Cart_create(MPI_COMM_WORLD, 2, idims, periods,
                0, &ICOMM);

MPI_Cart_shift(ICOMM, 0, 1, &rrank, &srank);
MPI_Sendrecv(plane1, NY, MPI_DOUBLE, srank,
             plane2, NY, MPI_DOUBLE, rrank,
             ICOMM, &status);

MPI_Cart_shift(ICOMM, 1, -1, &rrank, &srank);
MPI_Sendrecv(plane1, NY, MPI_DOUBLE, srank,
             plane2, NY, MPI_DOUBLE, rrank,
             ICOMM, &status);
```
Graph Topologies

```c
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph)
```

- `nnodes` - number of nodes
- `index` - the number of edges adjacent to all nodes 0 .. i
- `edges` - adjacent nodes for each node
- `reorder` - allow node ranks to be reordered

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i`, `i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[j]`, for `0 <= j <= index[0]-1` and the list of neighbors of node `i`, `i>0`, is stored in `edges[j]`, `index[i-1]+1 <= j <= index[i]-1`.
Graph Example

```
MPI_Comm newcomm;
int nnodes = 7;
int index[] = {2, 5, 6, 7, 10, 11, 12};
int edges[] = {1, 4, 0, 2, 3, 1, 1, 0, 5, 6, 4, 4};
MPI_Graph_create(MPI_COMM_WORLD, nnodes, index,
    edges, 0, &newcomm);
```
Graph Inquiry Functions

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
• Provides info needed to size index and edges

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)
• Get index and edges

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
• Get number of neighbors

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
• Get neighbors