

# CHAPTER 7

## The Sieve of Eratosthenes (埃拉托斯特尼筛法)



# 学习目标

- 块分配方案的分析
- 函数 **MPI\_Bcast**
- 性能增强



# 大纲

- 串行算法
- 并行算法设计与分析
- **MPI**程序设计
- 优化

# 素数的筛选

- 2到N之间有多少个素数？



# 顺序算法

	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	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

复杂度:  $\Theta(n \ln \ln n)$



# 伪代码

1. 创建无标记的自然数列表  $2, 3, \dots, n$
2.  $k \leftarrow 2$
3. 重复进行
  - (a) 标记  $k^2$  到  $n$  之间  $k$  的所有倍数
  - (b) 将最小的未标记的且大于  $k$  的数赋值给  $k$  ,直到  $k^2 > n$
4. 未标记的数就是素数



# 并行性的来源

- 领域划分
  - 将数据分成几块
  - 将计算步骤与数据联系起来
- 每个数组元素有一个原始任务



# 将3(a)并行化

标记出 $k^2$ 到 $n$ 之间所有 $k$ 的倍数

⇒

```
for all  $j$  where  $k^2 \leq j \leq n$  do
  if  $j \bmod k = 0$  then
    mark  $j$  (it is not a prime)
  endif
endfor
```

1. 创建无标记的自然数列表 2,  
3, ...,  $n$

2.  $k \leftarrow 2$

3. 重复进行

(a) 标记 $k^2$  到 $n$ 之间 $k$ 的所有  
倍数

(b) 将最小的未标记的且  
大于 $k$ 的数赋值给 $k$  ,  
直到 $k^2 > n$

4. 未标记的数就是素数





# 将3(B)并行化

找到最小的大于 $k$ 的未被标记的数

⇒

Min-reduction (为了找到最小的大于 $k$ 的未被标记的数)

Broadcast (为了得到所有结果)



# 聚合的目标

- 整合任务
- 降低通信成本
- 平衡各进程的计算



# 数据划分方式

- 交错式（循环式）划分
  - 容易确定每个索引的 "所有者"
  - 导致这个问题的负载不平衡
- 块划分
  - 平衡负载
  - 容易标记倍数
  - 如果 $n$ 不是 $p$ 的倍数，要确定所有者就比较复杂了



# 块划分

- 当 $n$ 不是 $p$ 的倍数时，想要平衡工作负荷
- 每个进程得到 $\lceil n/p \rceil$  or  $\lfloor n/p \rfloor$  个元素
- 寻求简单的表达方式
  - 给出一个所有者，寻找低、高指数
  - 给出一个所有者，寻找低、高指数

# METHOD #1

- 让  $r = n \bmod p$
- 如果  $r = 0$ , 所有块都有相同的大小
- 否则
  - 浅  $r$  个块的大小为  $\lceil n/p \rceil$
  - 剩下  $p-r$  个块的大小为  $\lfloor n/p \rfloor$



# EXAMPLES

17个元素分为7个过程



17个元素分为5个过程



17个元素分为3个过程



# METHOD #1 CALCULATIONS

- 进程*i*控制的第一个元素

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

- 进程*i*控制的最后一个元素

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

- 控制*j*的进程

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



# METHOD #2

- 在进程中分散较大的数据块
- 由进程*i*控制的第一个元素  
 $\lfloor in / p \rfloor$
- *i*进程*i*控制的最后一个元素  
 $\lfloor (i+1)n / p \rfloor - 1$
- 控制元素*j*的过程  
 $\lfloor p(j+1) - 1 / n \rfloor$





# EXAMPLES

17个元素分为7个过程



17个元素分为5个过程



17个元素分为3个过程



# COMPARE

Our choice

Operations	Method 1	Method 2
Low index	4	2
High index	6	4
Owner	7	4



# POP QUIZ

- 说明块分解方法2如何将**13**个元素划分给**5**个进程。

$$13(0)/5 = 0 \quad 13(2)/5 = 5 \quad 13(4)/5 = 10$$



$$13(1)/5 = 2 \quad 13(3)/5 = 7$$

# BLOCK DECOMPOSITION MACROS

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

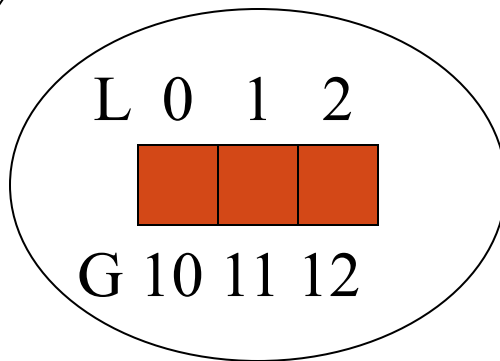
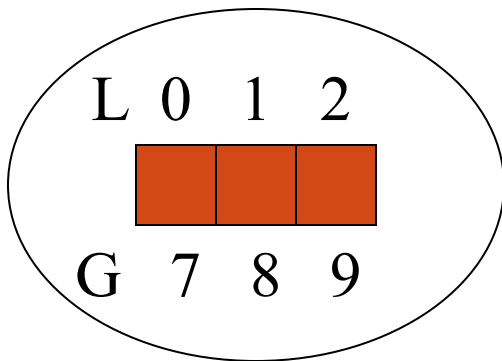
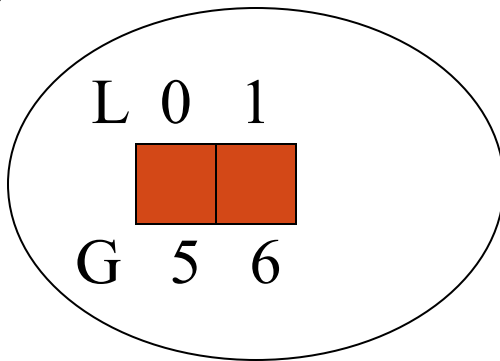
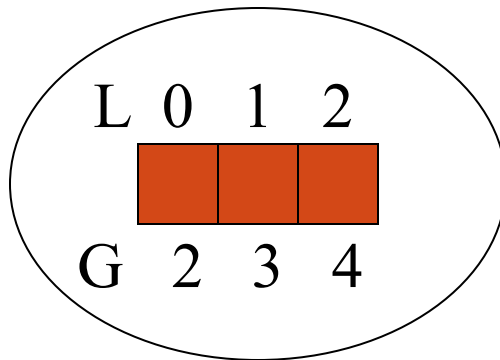
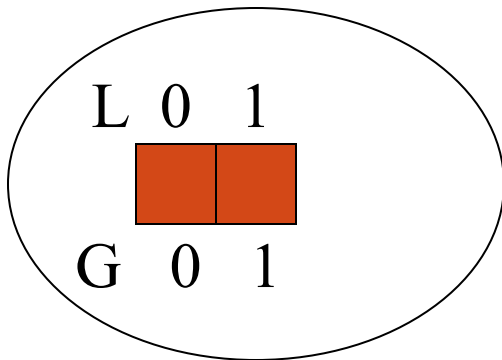
```
#define BLOCK_HIGH(id,p,n) \
    (BLOCK_LOW((id)+1,p,n)-1)
```

```
#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



# 循环使用元素

- 顺序程序

```
for (i = 0; i < n; i++) {  
    ...  
}
```

- 并行程序

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

关于这个进程的索引i...

取代了顺序程序的索引 gi



# 划分影响到执行

- 用于筛分的最大素数是 $\sqrt{n}$
- 第一个进程有 $\lfloor n/p \rfloor$ 个元素
- **if  $p < \sqrt{n}$  则有所有的素数**
- 第一进程总是广播下一个筛分素数
- 不需要还原步骤



# 快速标记

- 块分解允许与顺序算法相同的打标:

**$j, j + k, j + 2k, j + 3k, \dots$**

**instead of**

**for all  $j$  in block**

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



# 并行算法开发

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

2.  $k \leftarrow 2$

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)  $k \leftarrow$  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



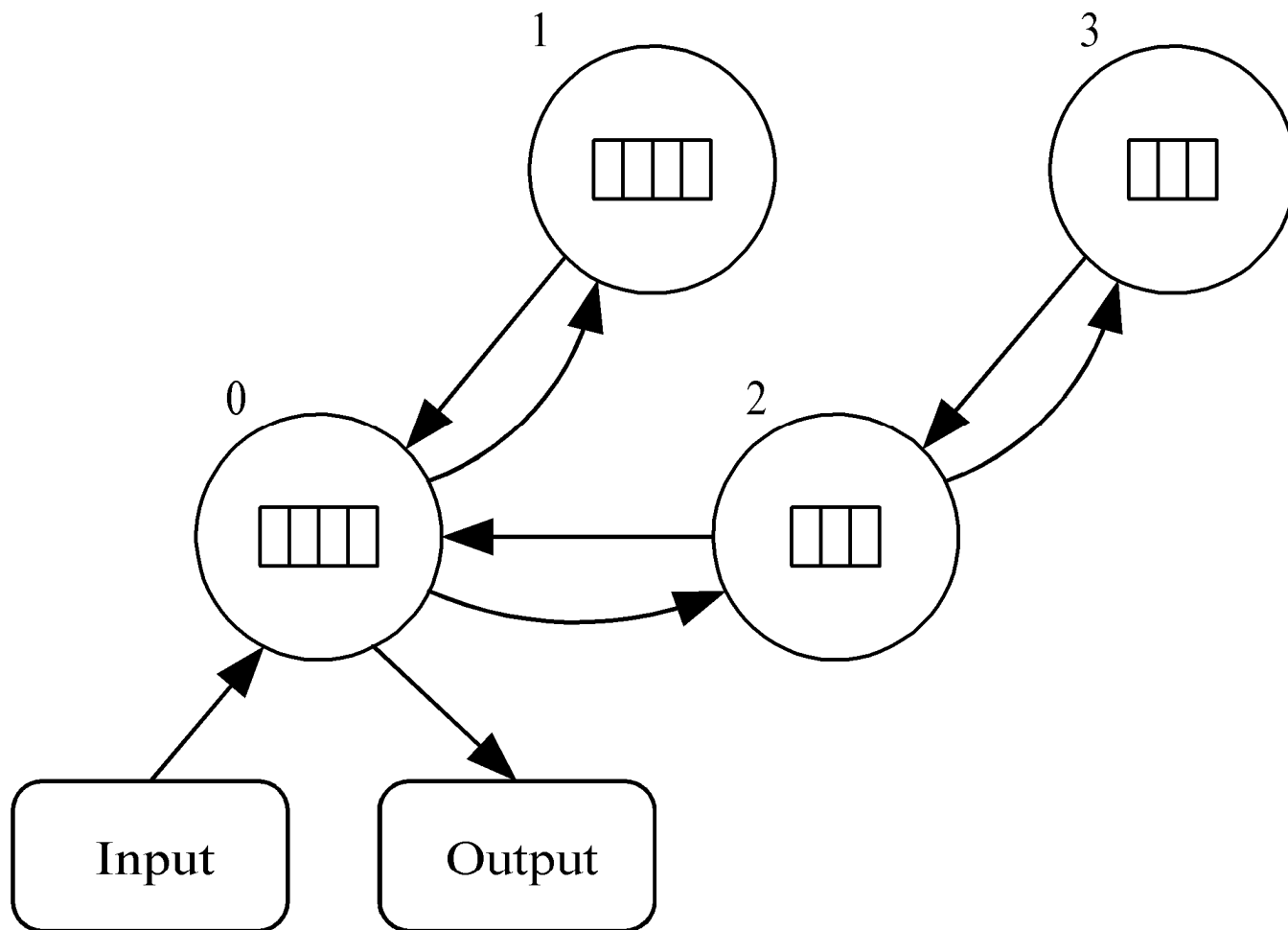
# 函数MPI\_BCAST

```
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);
```



# 通道图



# 分析

- $\chi$  是标记一个单元格所需的时间
- 顺序执行时间:  $\chi n \ln \ln n$
- 广播的数量:  $\sqrt{n} / \ln \sqrt{n}$
- 广播时间:  $\lambda \lceil \log p \rceil$
- 预期的执行时间:

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

# 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]); /*find the smallest unmarked*/
        prime = index + 2;
    }
    MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while (prime * prime <= n);
```

Local index



# 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;
}
```





# 基准测试

- 执行顺序算法
- 确定  $\chi = 85.47$  纳秒
- 执行一系列的广播
- 确定  $\lambda = 250 \mu\text{sec}$



# 执行时间（秒）

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



# 优化

- 删除偶数
  - 将计算次数减半
  - 为更大的 $n$ 值释放存储空间
- 每个进程找到自己的筛选素数
  - 将素数的计算复制到 $\sqrt{n}$
  - 消除了广播步骤
- 重新组织循环
  - 增加缓存命中率



# 重新组织循环

3-99: multiples of 3



3-99: multiples of 5



3-99: multiples of 7



Lower

3	5	7	9	11	13	15	17
19	21	23	25	27	29	31	33
35	37	39	41	43	45	47	49
51	53	55	57	59	61	63	65
67	69	71	73	75	77	79	81
83	85	87	89	91	93	95	97
99							

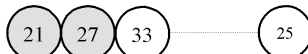
(a)

Cache hit rate

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



Higher

(b)



电子科技大学  
University of Electronic Science and Technology of China

假设高速缓存有4行，每行有4个字节。

# 比较4个版本

<i>Procs</i>	<i>Sieve 1</i>	<i>Sieve 2</i>	<i>Sieve 3</i>	<i>Sieve 4</i>
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



# 总结

- 埃拉托塞尼斯的筛子：并行设计使用领域划分
- 对比两个块状分布
  - 选择了公式更简单的一个
- 引入**MPI\_Bcast**
- 优化显示了最大化单处理器性能的重要性