

CHAPTER 4

消息传递程序设计

1



学习目标

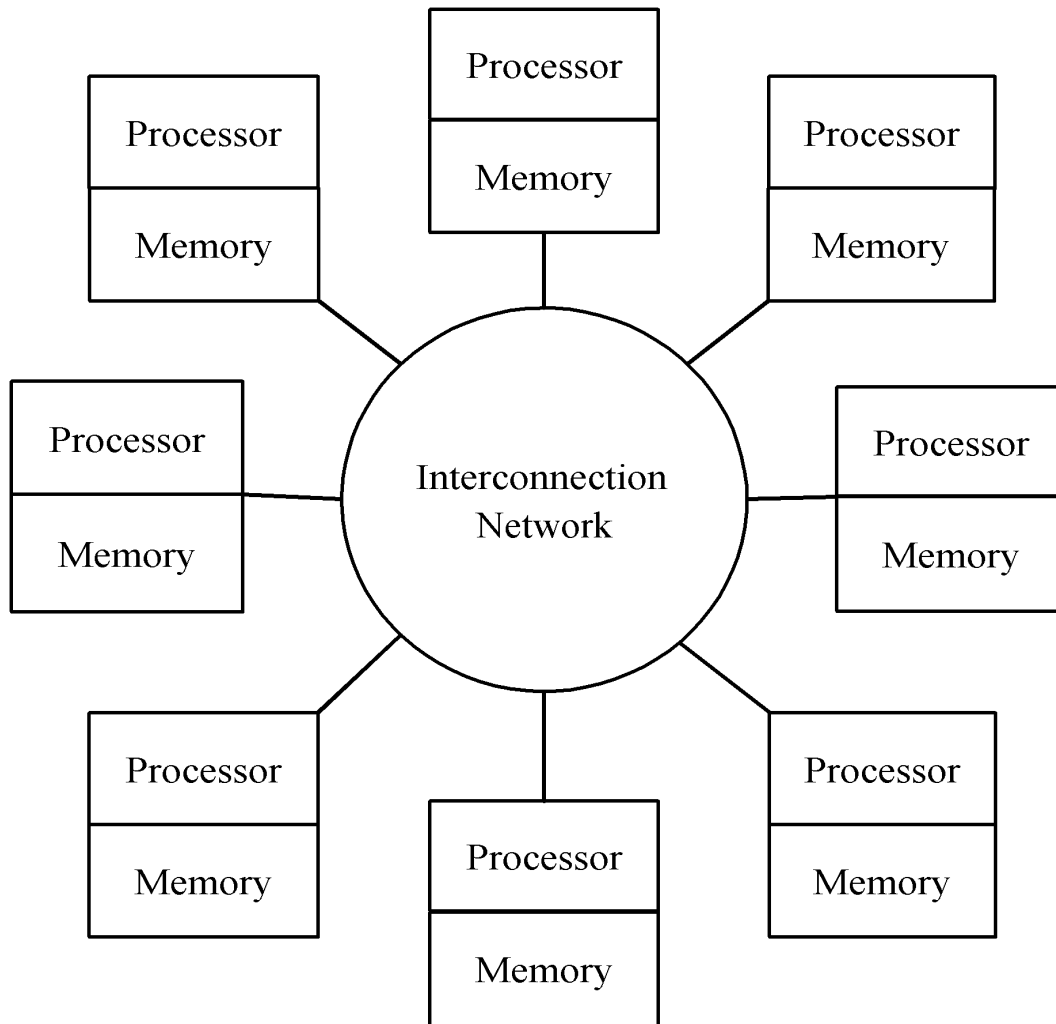
- 了解MPI程序如何执行
- 熟悉基本的MPI功能



OUTLINE

- 消息传递模型
- 消息传递接口 (MPI)
- 编码MPI程序
- 编译MPI程序
- 运行MPI程序
- 对MPI程序进行基准测试

消息传递模式



进程

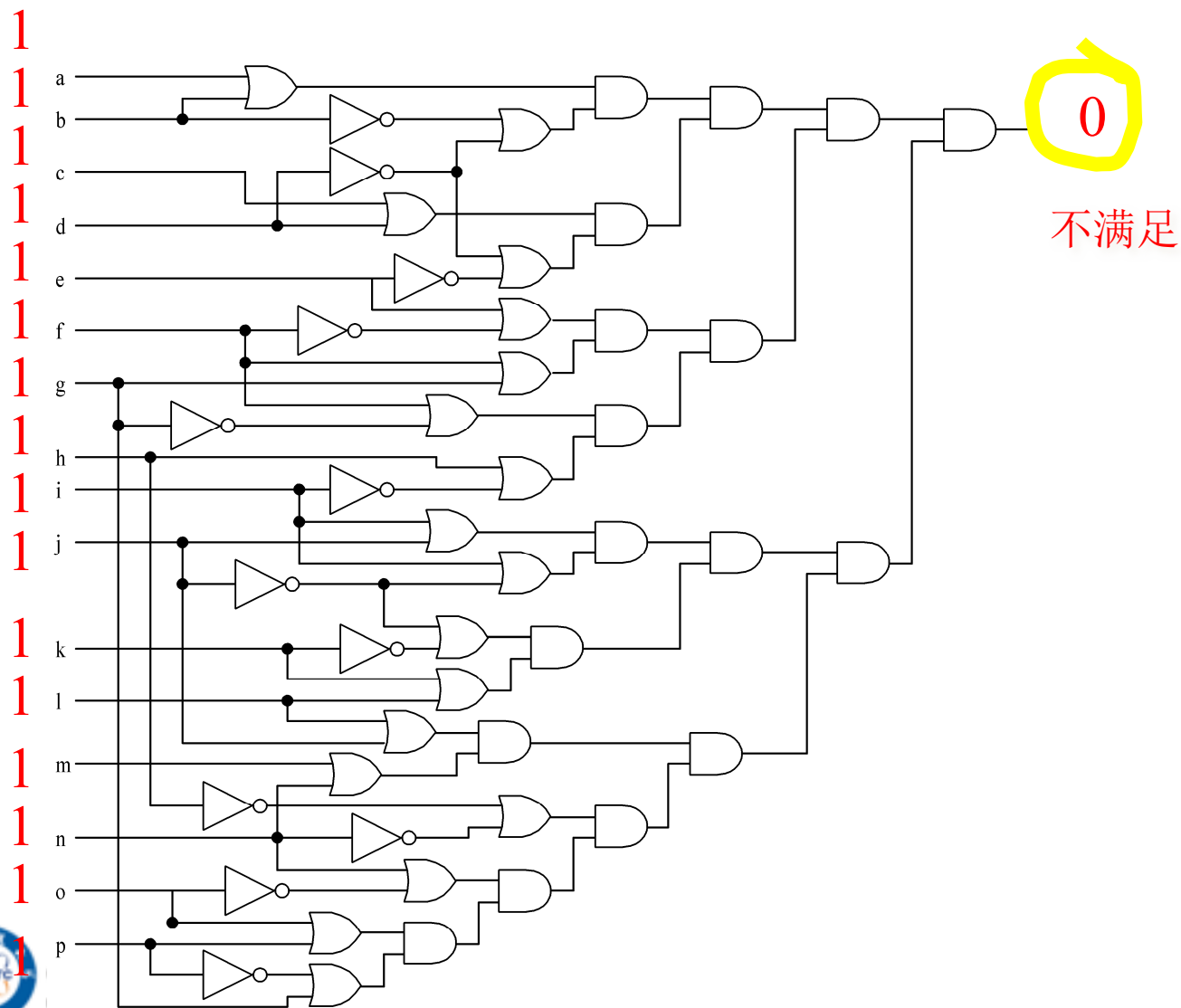
- 数量在启动时被指定
- 在整个程序执行过程中保持不变
- 所有执行相同的程序
- 每个进程都有唯一的ID号
- 交替进行计算和通信

消息传递接口（MPI）

- 1980年代末：供应商专有系统
- 1989：橡树岭国家实验室开发了并行虚拟机（PVM）
- 1992：MPI标准开始制定
- 1994：MPI标准的1.0版发布
- 1997：MPI标准的2.0版发布
- 当前：MPI是事实上的**消息传递接口**工业标准



电路可满足性

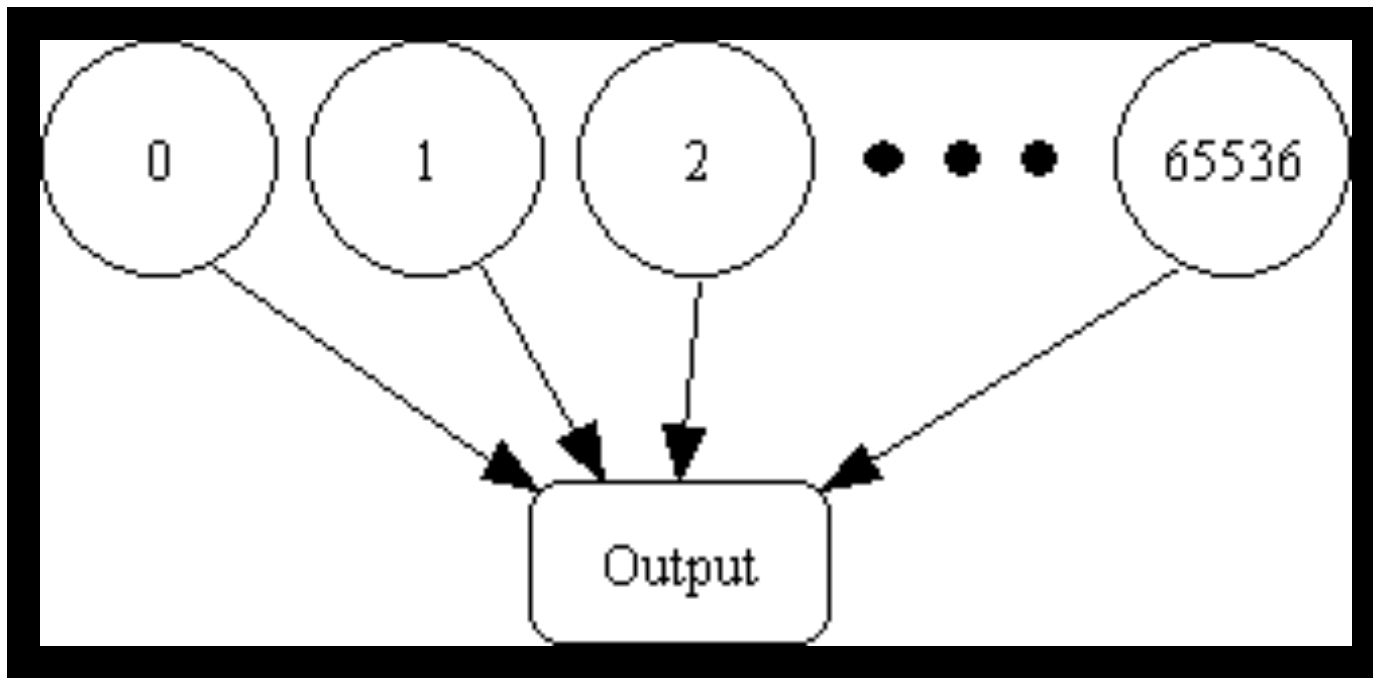


解决方法

- **电路可满足性是NP-完全的**
 - NP-完全问题：没有已知的算法可以在多项式时间内解决
- **我们寻找所有的解决方案，通过穷举式搜索**
- **16 个输入 \Rightarrow 65,536 个组合来测试**



划分和通信



- 尴尬的并行:任务之间没有通道



聚合和映射

■ 并行算法的特性

- ◆ 固定的任务数
- ◆ 任务之间没有通信
- ◆ 每个任务需要的时间是可变的

■ 咨询映射策略决策树

- ◆ 以循环的方式将任务映射到处理器上



循环（交错）分配

- 假设有 p 个进程
- 每个进程得到每一个分配的第 p 个工作
- Example: 有 5 个进程和 12 个任务
 - ◆ $P_0: 0, 5, 10$
 - ◆ $P_1: 1, 6, 11$
 - ◆ $P_2: 2, 7$
 - ◆ $P_3: 3, 8$
 - ◆ $P_4: 4, 9$



程序设计

- 程序将考虑16个布尔输入的所有65,536种组合
- 组合以循环的方式分配给进程
- 进程检查其每个组合
- 如果它发现了一个可满足的组合，打印输出

包含的文件

```
#include <mpi.h>
```

■ MPI header file

```
#include <stdio.h>
```

■ 标准I/O头文件

局部变量

```
int main (int argc, char *argv[]) {  
    int i;  
    int id; /* Process rank */  
    int p;  /* Number of processes */  
    void check_circuit (int, int);  
}
```

- 包括argc和argv：需要它们来初始化MPI
- 为运行该程序的每个进程提供一份每个变量的副本



初始化MPI

```
MPI_Init (&argc, &argv);
```

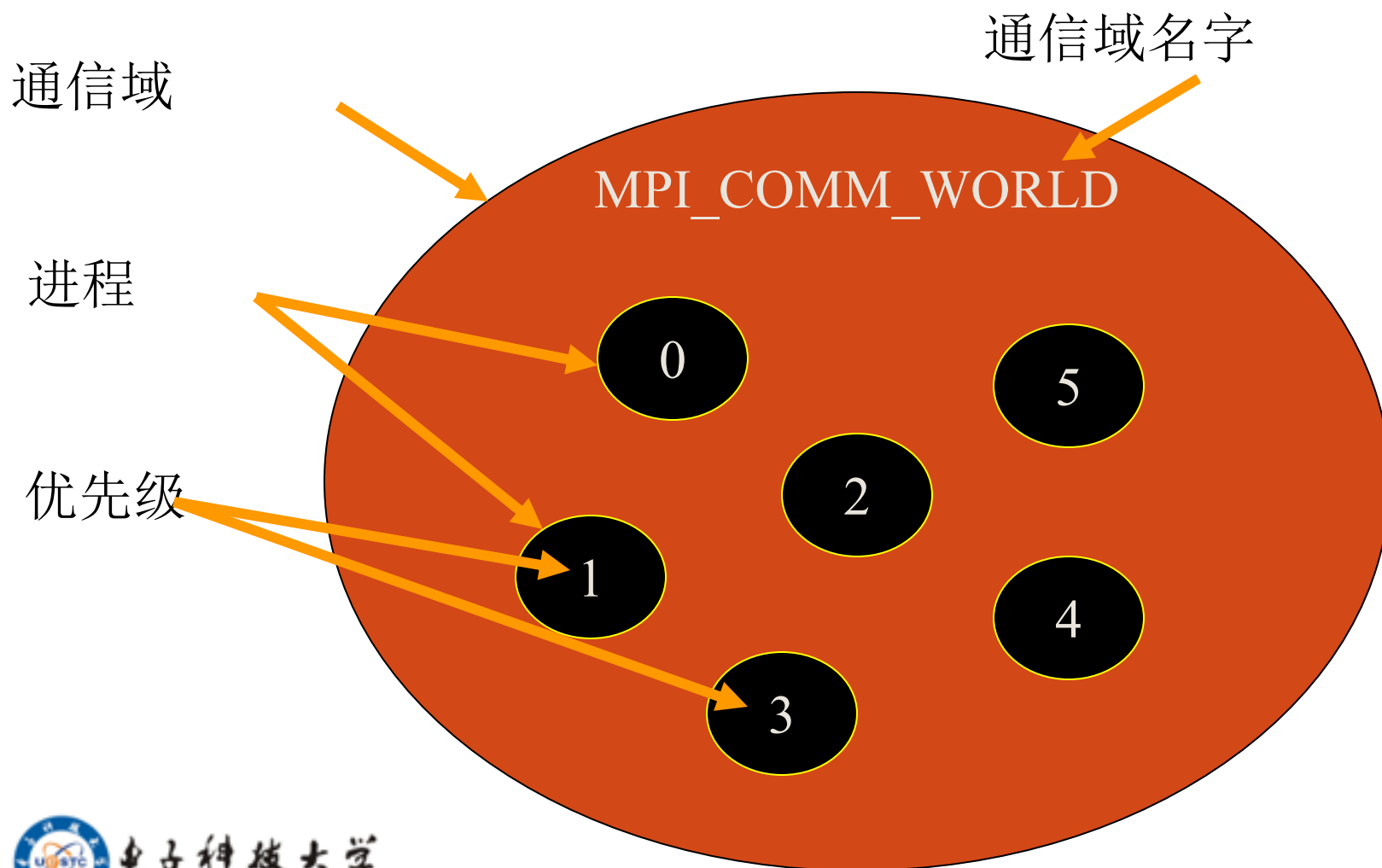
- 每个进程调用的第一个MPI函数
- 不一定是第一个可执行的语句
- 允许系统进行任何必要的设置



通信

- **通讯域：不透明的对象，为进程提供消息传递的环境**
- **MPI_COMM_WORLD**
 - ◆ 默认的通信域
 - ◆ 包括所有进程
- **有可能创建新的通信域**
 - ◆ 将在第8章中这样做

通信域(COMMUNICATOR)



确定进程的数量

```
MPI_Comm_size (MPI_COMM_WORLD, &p);
```

- 第一个参数是通信域
- 通过第二个参数返回的进程数



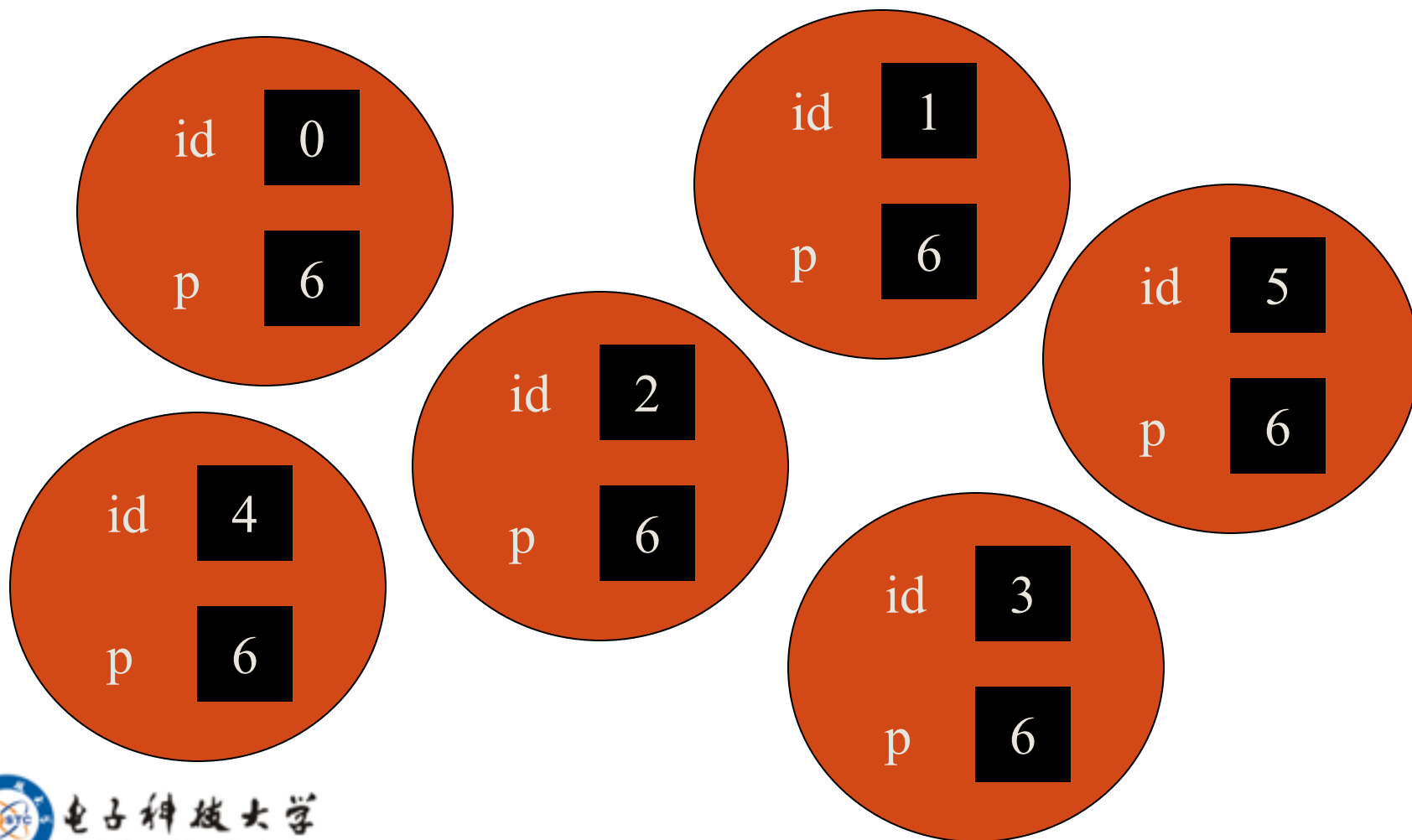
确定进程编号

```
MPI_Comm_rank (MPI_COMM_WORLD, &id);
```

- 第一个参数是通信域
- 通过第二个参数返回的处理优先级（范围为0 , 1 , ... , p-1)



自动变量的复制



外部变量呢？

```
int total;
```

```
int main (int argc, char *argv[]) {  
    int i;  
    int id;  
    int p;  
    ...  
}
```

■ **total**变量存储在哪里？

循环分配计算任务

```
for (i = id; i < 65536; i += p)
    check_circuit (id, i);
```

- 并行是在函数check_circuit之外
- 它可以是一个普通的、顺序的函数



MPI调用结束

```
MPI_Finalize();
```

- 在所有其他MPI库调用之后调用
- 允许系统释放MPI资源



```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    int id;
    int p;
    void check_circuit (int, int);

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    for (i = id; i < 65536; i += p)
        check_circuit (id, i);

    printf ("Process %d is done\n", id);
    fflush (stdout);
    MPI_Finalize();
    return 0;
}
```

在每一个printf()之后用fflush()




```

/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i)) ? 1:0)

void check_circuit (int id, int z) {
    int v[16];          /* Each element is a bit of z */
    int i;

    for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
        && (!v[3] || !v[4]) && (v[4] || !v[5])
        && (v[5] || !v[6]) && (v[5] || v[6])
        && (v[6] || !v[15]) && (v[7] || !v[8])
        && (!v[7] || !v[13]) && (v[8] || v[9])
        && (v[8] || !v[9]) && (!v[9] || !v[10])
        && (v[9] || v[11]) && (v[10] || v[11])
        && (v[12] || v[13]) && (v[13] || !v[14])
        && (v[14] || v[15])) {
        printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
            v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
            v[10],v[11],v[12],v[13],v[14],v[15]);
        fflush (stdout);
    }
}

```



编译MPI程序

```
mpicc -O -o foo foo.c
```

- **mpicc**: 编译和连接C+MPI程序的脚本
- **Flags**: 与C语言编译器含义相同
 - ◆ **-O** — 优化
 - ◆ **-o <file>** — 放置可执行文件的地方

运行MPI程序

■ `mpirun -np <p> <exec> <arg1> ...`

- ◆ `-np <p>` — 进程的数量
- ◆ `<exec>` — 可执行程序
- ◆ `<arg1> ...` — 命令行参数



指定主机处理器

- 主目录中的`.mpi-machines`文件按照使用顺序列出了主机处理器

- `.mpi_machines`文件内容示例

`band01 . cs . ppu . edu`

`band02 . cs . ppu . edu`

`band03 . cs . ppu . edu`

`band04 . cs . ppu . edu`



在1个CPU上执行

```
% mpirun -np 1 sat  
0) 10101111110011001  
0) 01101111110011001  
0) 11101111110011001  
0) 10101111111011001  
0) 01101111111011001  
0) 11101111111011001  
0) 101011111110111001  
0) 011011111110111001  
0) 111011111110111001  
Process 0 is done
```



在2个CPU上执行

```
% mpirun -np 2 sat
0) 01101111110011001
0) 01101111111011001
0) 01101111110111001
1) 10101111110011001
1) 11101111110011001
1) 10101111111011001
1) 11101111111011001
1) 10101111110111001
1) 11101111110111001
Process 0 is done
Process 1 is done
```



在3个CPU上执行

```
% mpirun -np 3 sat
0) 01101111110011001
0) 11101111111011001
2) 10101111110011001
1) 11101111110011001
1) 10101111111011001
1) 01101111110111001
0) 10101111110111001
2) 01101111111011001
2) 11101111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```



输出

- 输出顺序只部分反映了并行计算机内部输出事件的顺序
- 如果进程A打印两条信息，第一条信息会出现在第二条之前
- 如果进程A在进程B之前调用printf，不能保证进程A的信息会出现在进程B的信息之前



优化程序

- 我们希望找到解决方案的总数
- 将sum-reduction 放入程序中
- Reduction 是一种集体通信



改进

- **修改函数check_circuit**
 - ◆ 如果电路可以满足输入组合，则返回1
 - ◆ 否则返回0
- **每个进程保留它所发现的可满足电路的本地计数**
- **在for循环之后进行还原**



新的声明和代码

```
int count;    /* Local sum */  
int global_count; /* Global sum */  
int check_circuit (int, int);  
  
count = 0;  
for (i = id; i < 65536; i += p)  
    count += check_circuit (id, i);
```



MPI_REDUCE()的原型

```
int MPI_Reduce (  
    void          *operand,  
                /* addr of 1st reduction element */  
    void          *result,  
                /* addr of 1st reduction result */  
    int           count,  
                /* reductions to perform */  
    MPI_Datatype  type,  
                /* type of elements */  
    MPI_Op        operator,  
                /* reduction operator */  
    int           root,  
                /* process getting result(s) */  
    MPI_Comm      comm  
                /* communicator */  
)
```



MPI_DATATYPE选项

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT



MPI_OP选项

- MPI_BAND
- MPI_BOR
- MPI_BXOR
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM



我们对MPI_REDUCE()的调用

```
MPI_Reduce (&count,  
            &global_count,  
            1,  
            MPI_INT,  
            MPI_SUM,  
            0,  
            MPI_COMM_WORLD);
```

只有进程0会
得到结果

```
if (!id) printf ("There are %d different solutions\n",  
                global_count);
```



执行第二个计划

```
% mpirun -np 3 seq2
0) 01101111110011001
0) 11101111111011001
1) 11101111110011001
1) 10101111111011001
2) 10101111110011001
2) 01101111111011001
2) 11101111110111001
1) 01101111110111001
0) 10101111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```



程序性能测定

- `MPI_Barrier` — barrier 同步化
- `MPI_Wtick` — 定时器分辨率
- `MPI_Wtime` — 当前时间

WALL-CLOCK时间VS CPU时间

■ CPU时间（或处理时间）

- ◆ 是指CPU用于处理指令的时间.
- ◆ 用户时间 + 系统时间

■ Wall-clock time (or wall time)

- ◆ 从一个任务开始到完成的时间流逝.
- ◆ CPU时间，I/O时间，以及通信通道的延迟

■ `/usr/bin/time`



基准代码

```
double elapsed_time;  
...  
MPI_Init (&argc, &argv);  
MPI_Barrier (MPI_COMM_WORLD);  
elapsed_time = - MPI_Wtime();  
...  
MPI_Reduce (...);  
elapsed_time += MPI_Wtime();
```

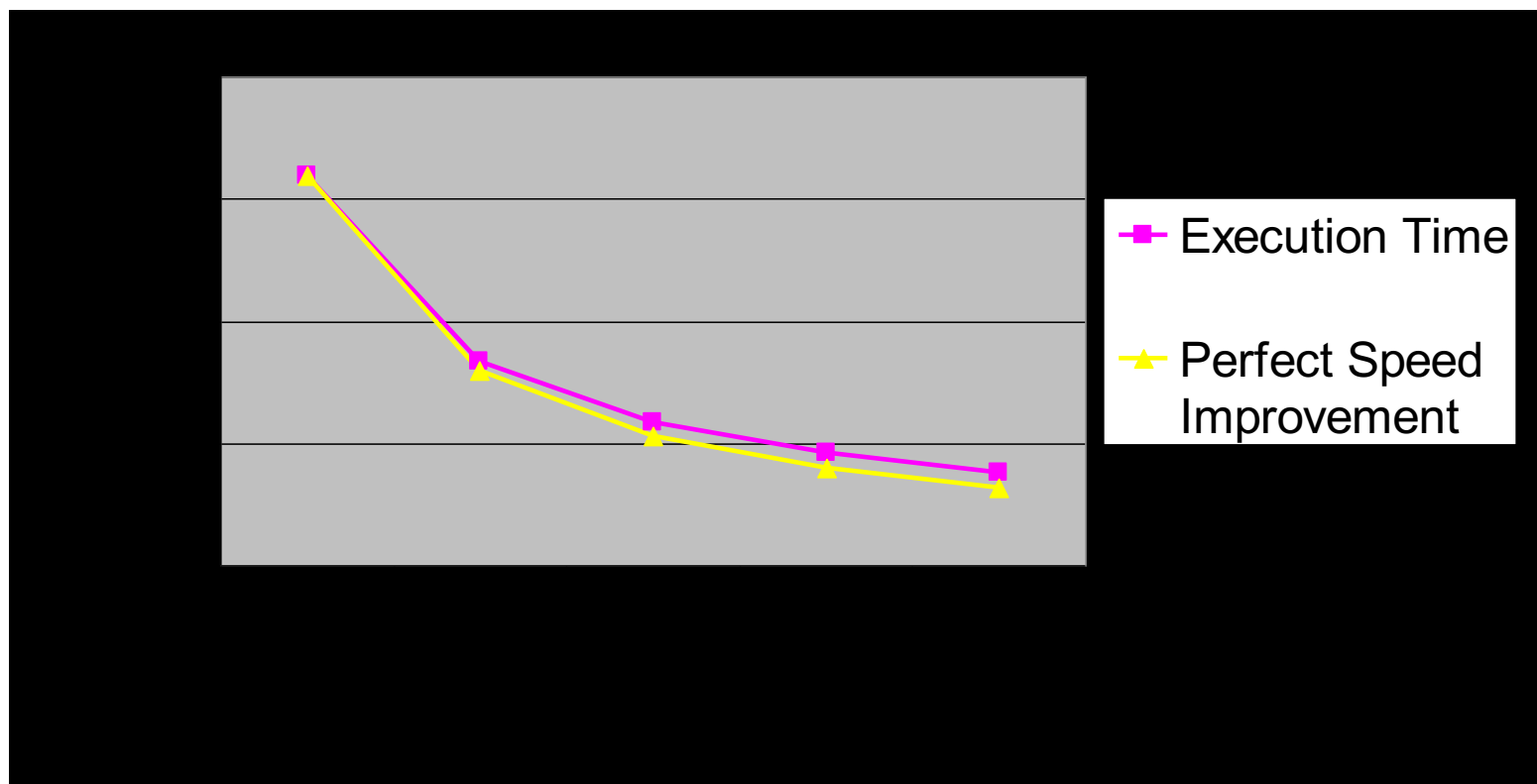


基准测试结果

Processors	Time (sec)
1	15.93
2	8.38
3	5.86
4	4.60
5	3.77



基准测试结果



总结 (1/2)

- 消息传递编程自然地遵循任务/通道模型
- 消息传递程序的可移植性
- MPI是最广泛采用的标准



总结 (2/2)

■引入MPI函数

- ◆MPI_Init
- ◆MPI_Comm_rank
- ◆MPI_Comm_size
- ◆MPI_Reduce
- ◆MPI_Finalize
- ◆MPI_Barrier
- ◆MPI_Wtime
- ◆MPI_Wtick

