author: 小红

1. OpenMP（以下简称 omp）

主要内容取自 http://parallel.zhangjikai.com/openmp.html

（1）OpenMP 的执行模式

omp 编程模型以线程为基础，执行模式采用 fork-join 模式，如下图。

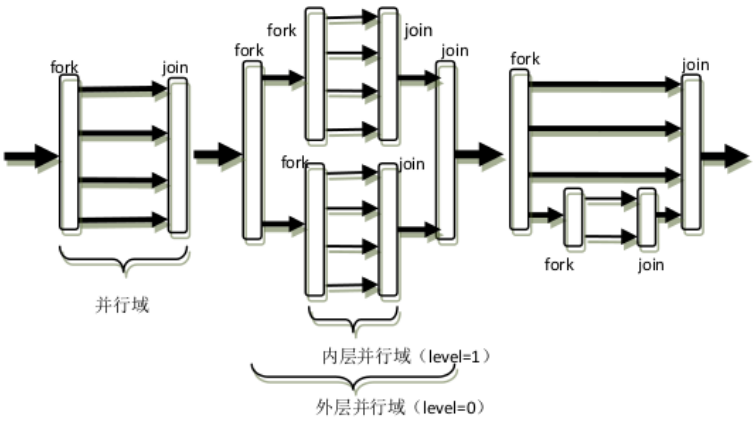


图 2.1 fork-join 并行执行模型

其中 fork 创建新线程或者唤醒已有的线程, join 将多个线程合并.

在程序执行的时候，只有主线程在运行，当遇到需要并行计算的区域，会派生出线程来并行执行，在并行执行的时候，主线程和派生线程共同工作，在并行代码结束后，派生线程退出或者挂起，不再工作，控制流程回到单独的线程中下.

（2）directive

所有的编译制导指令都是以#pragma omp 开始，后面跟具体的功能指令(directive)或者命令. 一般格式如下所示:

```
#pragma omp <directive> [clause [[,] clause]...]

{

// …… 要并行的语句块

}
```

<directive>有多种选择，最常见的是 parallel，会启动并行。

并行域结构

directive 选择 parallel。这样 omp 会创建线程组并发执行任务，但是并不会负责线程之间的任务分发，并行域结束之后会有一个隐式的屏障同步该区域内所有线程。

任务分担结构

主要为了给线程分配不同的任务，要放在并行域里面。

①directive 选择 for：

用于把下面的 for 循环的任务分配给不同线程。比如下面的 for 循环是打印 i=1~n 的整数，前面 parallel 出了 4 个线程，那么接下来 omp 会为这 4 个线程分配不同的 i 执行里面的语句，每个线程的输出不会相同，即不会共享相同的 i，同时 4 个线程一起能完成 1~n 的任务。

下面是一个使用示例:

```
void parallel_for() {
    int n = 9;
    int i = 0;
    #pragma omp parallel shared(n) private(i)
    {
        #pragma omp for
        for(i = 0; i < n; i++) {
            printf("Thread %d executes loop iteration %d\n", omp_get_thread_num(),i);
        }
    }
}
```

下面是程序执行结果

```
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

※omp_get_thread_num()返回当前线程号。

②directive 选择 sections

sections 可以为不同线程分配不同任务，底下有若干个 section 块。如果线程比 section 多，那么多余线程会处于空闲状态（idle），如果线程比 section 少，那么一个线程会执行多个 section。总之每个 section 只会被 1 个线程执行。

```
void parallel_section() {
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                (void)funcA();
            }

            #pragma omp section
            {
                (void)funcB();
            }
        }
    }
}
```

比如上面的图，只会有一个线程调用 funcA，只会有一个线程调用 funcB。

③directive 选择 single

single 用于指定某段代码只能被一个线程执行，而且在不指定 nowait 的情况下，别的线程会等待这段代码执行完（同步），才会一起执行之后的语句。

（3）clause 子句

① shared(list)

这个子句指定哪些数据在线程间共享。

比如你写：#pragma omp parallel shared(a,b) 那么下面的代码段共享 a 和 b 两个变量，这两个变量不一定是基本数据类型，可以是 struct/class/数组/等等。

omp 默认不维护它们的同步和互斥。

② private(list)

这个子句指定哪些数据私有，也就是说每个线程具有变量的私有副本，线程 1 怎么改都不影响线程 2 这个变量的值。

比如你写：

```
int i=0,a=3,n=10;

#pragma omp parallel for private(i,a) // parallel 和 for 能一起写

for(i = 0;i < n;++i){

   a = i+1;

}
```

那么每个线程的 a 最终都会从 3 加到 13.别的线程 a=i+1 不会影响自己的 a.

③ **lastprivate(list)**

用这个词修饰的变量会保存自己退出并行域前最后的取值，（给主线程接收）。

比如主线程 0 和线程 1、2、3 用 omp for 跑一个 for 循环，每次 a++。

```
In for: thread 3 has a value of a = 7 for i = 6
In for: thread 3 has a value of a = 8 for i = 7
In for: thread 2 has a value of a = 5 for i = 4
In for: thread 2 has a value of a = 6 for i = 5
In for: thread 1 has a value of a = 3 for i = 2
In for: thread 0 has a value of a = 1 for i = 0
In for: thread 0 has a value of a = 2 for i = 1
In for: thread 1 has a value of a = 4 for i = 3

Out for: thread 0 has a value of a = 8 for i = 2
```

虽然主线程只是让 a 变成了 2，但是主线程会接收 a 最后的取值，即线程 3 导致的 a=8.

④ **firstprivate(list)**

用于给 private 变量提供初始值，初始值和前面定义的同名变量的值相同。比如前面有 a = {1,2,3,4}，那么后面你 firstprivate(a)就会继承这些值。

⑤ **nowait**

使用这个子句后的线程在执行完代码段时不会等待别的线程执行完就会继续跑下面的代码。

⑥ **schedule(kind[,chunk_size])** // 任务调度模式，**只用于循环结构**

kind 有 static / dynamic / guided / auto / runtime

- schedule(static, chunk_size)：静态调度。

  如果循环是 1~n，线程有 t 个，chunk_size 不指定。那么静态调度会为每个线程分配 n/t 或 n/t+1 次连续的迭代计算（均匀分配循环）。

  如果指定了 chunk_size，那么每次为线程分配 chunk_size 次迭代计算，如果第一轮没有分配完，则循环进行下一轮分配。

- schedule(dynamic, chunk_size)：动态调度

  只要线程空闲就分配任务给它。

- scheduled(guided, chunk_size)：启发式自调度

  开始时每个线程会分配到较大的迭代块，之后分配到的迭代块的大小会逐渐递减.

- runtime：运行时调度

  并不算真正的调度方式，在运行时同时环境变量 OMP_SCHEDULE 来确定调度类型，最终的调度类型仍为上面的 3 种调度方式之一．在 bash 下可以使用下面的方式设置：

  ```
  export OMP_SCHEDULE="static"
  ```

- auto：自动

  选择权交给编译器。

（4）同步结构

也属于编译制导指令，目的是为了实现同步。

① **#pragma omp barrier**

单纯的一个路障，后面不跟着语句，在 parallel 语句段内发挥作用：所有线程在此等待其他线程完成任务，线程到齐后继续执行下面的任务。for、single 等等自己就隐式地含有路障。

② **#pragma omp ordered** // 只用于循环结构

允许在并行域中以串行的顺序执行一段代码，如果我们在并行域中想按照顺序打印被不同的线程计算的数据，就可以使用这个子句

③ critical // 临界区

（5）更多重要的子句

① **reduction(operator:list)** //用于循环

如果利用循环，将某项计算的所有结果进行求和(或者减、乘等其他操作)得出一个数值，这在并行计算中十分常见，通常将其称为规约。

```
/*
* 开启并行计算，reduction(+:sum)制定sum变量在并行计算中进行求和和归约操作
* private(x)：指定每个线程都有自己的x变量的私有副本
*/
#pragma omp parallel for reduction(+:sum) private(x)
for (i=0; i < steps; i++) {
    x = (i+0.5)*step;
    sum += 4.0 / (1.0+x*x);
}
```

比如求π的这段语句，使用 reduction(+:sum)后，无需再对 sum 进行保护，sum 得到的即为 0~steps 的和。注意，reduction 的操作类似于 private(sum)和对所有线程的 sum 求和的操作的融合，也就是说，每个线程计算 sum 都是独立的，在代码段结束后，所有的 sum 副本会合并到主线程的 sum。

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

以上为 reduction 支持的运算符。

（6）一些函数操作

① omp_get_num_procs()：返回可用的处理器数目；

② omp_get_num_threads()：返回当前并行区域的活动线程个数（外部调用则只会有一个主线程，返回 1）；

③ omp_get_thread_num()：返回（执行这个语句的）线程号；

④ omp_set_num_threads(int num)：设置接下来的并行区域内要创建的线程个数为 num 个。

⑤ omp_in_parallel()：返回 1：当前处于并行状态，返回 0：没有处于并行。

⑥ omp_get_max_threads()：返回最大线程数量，与 set_num_threads 无关。

⑦ omp_set_dynamic(0 或非 0)：设置是否动态调整并行区域的线程数，参数为 0：禁用，为非 0，自动调整。

⑧ omp_get_dynamic()：返回是否动态调整并行区域的线程数。

⑨ omp_get_wtime()：返回当前时间戳，两个时间戳相减得到以秒为单位的时间。

※用 gcc 编译 omp 程序：gcc -fopenmp xxx.c -o xxx

2. MPI

※ 编译 mpicc -o xxx xxx.c

※ 两种运行方式

```
mpiexec -n <num_processes> ./<program_name>：用num_processes个进程启动program_name程序
mpirun -f hostfile -np 10 ./pi 用hostfile里面写的hadoop集群，分配共10个进程，来执行MPI并行计算任务。
```

（1）6 个基本函数

```
// 初始化

MPI_Init(int *argc, char ***argv)`

// MPI 结束调用
MPI_Finalize(void)

/**
 * @brief 获得进程的标识号
 * 进程标保存到 rank 里
 */
MPI_Comm_rank(MPI_Comm comm, int *rank)

/**
 * @brief 获得当前通信域中进程的个数
 * 进程数量保存到 size 里
 */
MPI_Comm_size(MPI_Comm comm, int *size)`

/**
 * MPI_Send
 * @brief 发送消息
 * @param buf 发送缓冲区的起始位置
 * @param count 发送的数据个数
 * @param datatype 发送数据的数据类型
 * @param dest 目标进程标号
 * @param tag 消息标志
 * @param comm 通信域
 *
*/
MPI_Send(void * buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)

/**
```

```
 * MPI_Recv
 * @brief 接收信息
 * @param buf 接受缓冲区地址
 * @param count 多少个 datatype 的数据
 * @param datatype 接受的数据类型
 * @param source 发送数据的进程号
 * @param tag 消息标志
 * @param comm 通信域
 * @param status 返回状态
*/
MPI_Recv(void * buf, int count, MPI_Datatype data, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

① 消息传递（阻塞通信）

主要是 MPI_Send 和 MPI_Recv。

MPI_Datatype 是 MPI 自己定义的一些基本数据类型。

MPI预定义数据类型	对应的C数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int (optional)
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int

一部分展示

传递信息时数据类型要相同。

发送消息必须指定发送对象的进程标号 dest 和消息标识 tag。接收消息除了可以指定接受的对象 source 和消息标志 tag，也可以设定为 MPI_ANY_SOURCE 和 MPI_ANY_TAG 来表示接受任意进程发给本进程的消息。

通信域 comm 一般用 MPI_COMM_WORLD 就行了。

tag 用 99 就行了。

（2）其他常用函数

① 获得当前时间

```
double MPI_Wtime(void)
```

② 获得机器名字

```
int MPI_Get_processor_name(char* name, int* result_len)
```

name:当前进程所在机器名字；result_len:名字长度

③ MPI 版本

```
int MPI_Get_version(int *version, int *subversion)
```

④ 判断 MPI_Init 是否执行，唯一一个可以在 MPI_Init 之前调用的函数

```
int MPI_Initialized(int *flag)
```

flag:是否已调用

⑤ 使通信域 Comm 中的所有进程退出

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

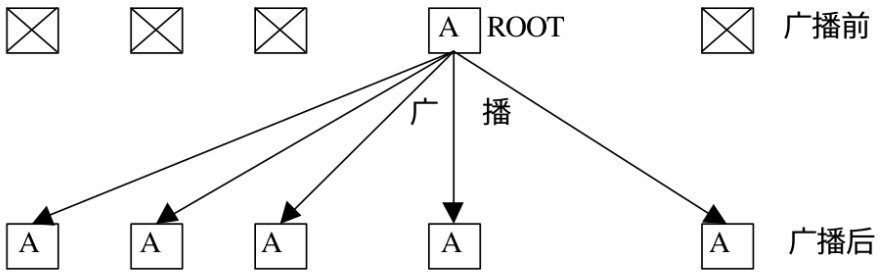
（3）数据广播

即 MPI_Bcast

```
MPI_Bcast(
    void* data,             // 缓冲区的起始地址
    int count,              // 数据的个数
    MPI_Datatype datatype,  // 数据类型
    int root,               // 广播数据的根进程标识
    MPI_Comm communicator   // 通信域
)
```

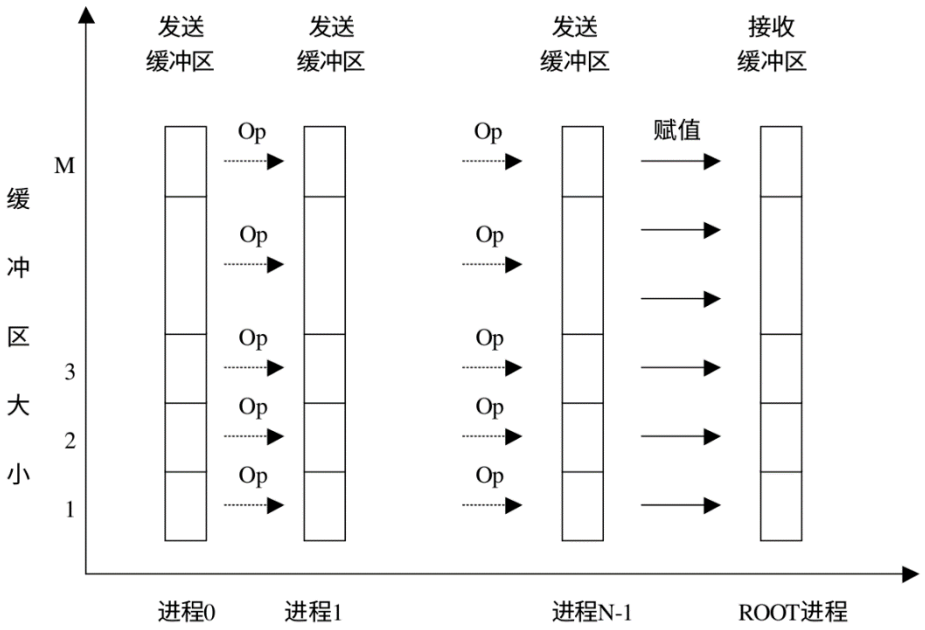
功能类似于根进程对每个别的进程都调用 MPI_Send 传送数据，每个别的进程都用 MPI_Recv 接收根进程的数据。

但是！在使用 MPI_Bcast 的时候，我们要注意不需要使用 MPI_Recv 接受，而是在每个进程里都调用 MPI_Bcast。



（4）归约

即 MPI_Reduce，用来将组内每个进程输入缓冲区中的数据按给定的操作 op 进行预案算，然后将结果返回到序号为 root 的接收缓冲区中。操作 op 始终被认为是可以结合的，并且所有 MPI 定义的操作被认为是可交换的。



MPI_Reduce 的内容：

```
int MPI_Reduce(
    void * sendbuf,           // 发送缓冲区的起始地址
    void * recvbuf,           // 接收缓冲区的起始地址
    int count,                // 发送/接收 消息的个数
    MPI_Datatype datatype,    // 发送消息的数据类型
    MPI_Op op,                // 规约操作符
    int root,                 // 根进程序列号
    MPI_Comm comm             // 通信域
);
```

归约操作符如下：

操作	含义
MPI_MAX	最大值
MPI_MIN	最小值
MPI_SUM	求和
MPI_PROD	求积
MPI_LAND	逻辑与
MPI_BAND	按位与
MPI_LOR	逻辑或
MPI_BOR	按位或
MPI_LXOR	逻辑异或
MPI_BXOR	按位异或
MPI_MAXLOC	最大值且相应位置
MPI_MINLOC	最小值且相应位置

3. MPI 和 OpenMP 实践：计算 π

（1）MPI

PI 的计算公式可以通过下面的公式计算出来：

$$f(x) = \frac{4}{1+x^2}$$

$$\pi \approx \frac{1}{N} \times \sum_{i=1}^{N} f(\frac{i-0.5}{N})$$

使用 MPI 并行的思路是每个进程计算一部分 N 值，计算完成之后通过 `MPI_Reduce` 将结果收集起来，下面是实现代码：

```
#include "mpi.h"

#include <stdio.h>
#include <math.h>
#include<unistd.h>
/**
 * @brief 计算π
 */

int main(argc, argv)
int argc;
char *argv[];
{
    /**
     * @param done 控制程序是否继续执行
     * @param n 区间数，用户输入的用于计算π的区间数量
     * @param myid 当前进程的id
     * @param numprocs 进程总数
     * @param i 循环计数器
     * @param namelen 处理器名字的长度
     * @param processor_name 存储处理器名字的数组
     * @param PI25DT pi 的值，定义为常数
     */
```

```
    int done = 0, n, myid, numprocs, i, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc, &argv); // 指示系统完成所有初始化，在任何 MPI 函数前使用，除了 MPI_Initialized
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);         // 获取进程总数
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);             // 获取当前进程 id
    MPI_Get_processor_name(processor_name, &namelen); // 获取处理器名字
    printf("进程总数：%d\n 当前进程 id：%d\n 处理器名字：%s\n", numprocs, myid, processor_name);
    sleep(1);
    n = 0;
    while (!done) // 准备完成下一次计算
    {
        if (myid == 0)
        { // 只有进程为 0 的进程才能输入
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); // 区间数广播给其他进程

        printf("n = %d, Rank %d / %d @ %s\n", n, myid+1, numprocs, processor_name); // 每个进程输出自己的进程 ID、总进程数和处理器名字

        if (n == 0)
            break;

        // 每个进程根据接收到的区间数计算一部分 π 的值
        h = 1.0 / (double)n;
        sum = 0.0;
```

```
        for (i = myid + 1; i <= n; i += numprocs)
        {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x * x);
        }
        mypi = h * sum;
        // 归约操作，各部分 pi 值相加得到最终的 pi 值
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                   MPI_COMM_WORLD);

        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n",
                   pi, fabs(pi - PI25DT));
    }
    // 终止
    MPI_Finalize();
    return 0;
}
```

每个进程的计算区间大概如下图：

0	1	2	……	X-2	X-1	0	1	2	……	X-2	X-1	0	1	2	……

里面标的是进程号，表示这个区间由哪个进程负责。

过程是：初始化→获取当前进程号→0 号进程获取区间数 n→0 号进程把 n 广播给其他进程→每个进程计算自己负责的区间，获得 mypi→把 mypi 归约到 0 号进程的变量 pi 中，即为答案。

（2）OpenMP

```
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 128

static long steps = 1000000000; // 计算π的步数
double step; // 每一步的大小

int main (int argc, const char *argv[]) {

    int i,j;
    double x;
    double pi, sum = 0.0;
    double start, delta;
    double delta_serial;
    step = 1.0/(double) steps; // 初始化步长
    // Compute parallel compute times for 1-MAX_THREADS
    // 计算有j个线程时，迭代1e9次计算pi的时间
    printf("计算结果\t线程数\t时间(s)\t加速比\t效率\n"); // 本输出方式方便在终端查看
    // printf("计算结果,线程数,时间(s),加速比,效率\n"); // 本输出方式用于构造csv文件来方便画图
    for (j=1; j<= MAX_THREADS; j++) {
        omp_set_num_threads(j);

        sum = 0.0;
        double start = omp_get_wtime();

        /*
```

```
        * 开启并行计算，reduction(+:sum)制定 sum 变量在并行计算中进行求和和归约操作
        * private(x)：指定每个线程都有自己的 x 变量的私有副本
        */
        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0; i < steps; i++) {
            x = (i+0.5)*step;
            sum += 4.0 / (1.0+x*x);
        }

        // Out of the parallel region, finialize computation
        pi = step * sum;
        delta = omp_get_wtime() - start;
        if(j == 1){
            delta_serial = delta;
        }
        double speedup = delta_serial / delta;
        double efficiency = speedup / j;
        printf("%.16g\t%d\t%.4g\t%.4g\t%.4g\n", pi, j, delta, speedup, efficiency); // 本输出方式方便在终端查看
        // printf("%.24g,%d,%.4g,%.4g,%.4g\n", pi, j, delta, speedup, efficiency); // 本输出方式用于构造 csv 文件来方便画图
    }
}
```

基本过程是：omp_set_num_threads(j)设置准备用 j 个线程跑代码→omp_get_wtime()获取当前时间→开启并行以及 for 循环以及求和归约，令每个线程拥有变量 x 副本→计算π，途中 sum+=…进行归约→进一步计算，获取最终结果π，再计算当前时间，时间相减获取运行时间。

4. Hadoop

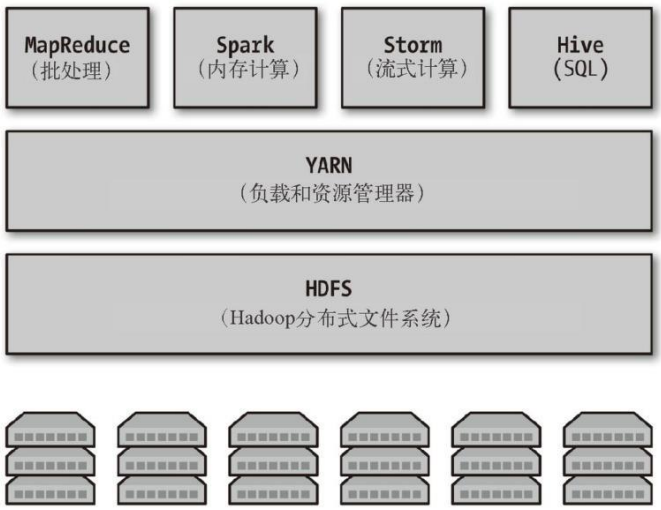
（1）概念和介绍

Hadoop 由两个主要组件组成：**HDFS 和 YARN**。

HDFS(DFS)是 Hadoop 的分布式文件系统，负责管理存储在集群中磁盘上的数据；**YARN 则是集群资源管理器**，将计算资源（worker 节点上的处理能力和内存）分配给希望执行分布式计算的应用程序。

往上有个重要模块：

MapReduce：一个帮助计划对数据运行并行计算的框架。该 Map 任务会提取输入数据，转换成能采用键值对形式对其进行计算的数据集。Reduce 任务会使用 Map 任务的输出来对输出进行汇总，并提供所需的结果。



hadoop 架构栈

HDFS 和 YARN 由几个守护进程实现，一直在集群节点上接收输入并通过网络输出，每个进程在自己的 JVM 上运行，有 OS 管理。集群中有下面几种节点。

① master 节点。通常是用户访问集群的入口点，为 worker 节点提供协调服务。

② worker 节点。用于从 master 节点接受任务，一般用于存储或检索数据、运行特定应用程序。

③ NameNode（master 服务）。用于存储文件系统的目录树、文件元数据和集群中每个文件的位置。如果客户端想访问 HDFS，必须先通过从 NameNode 请求信息来查找相应的存储节点。

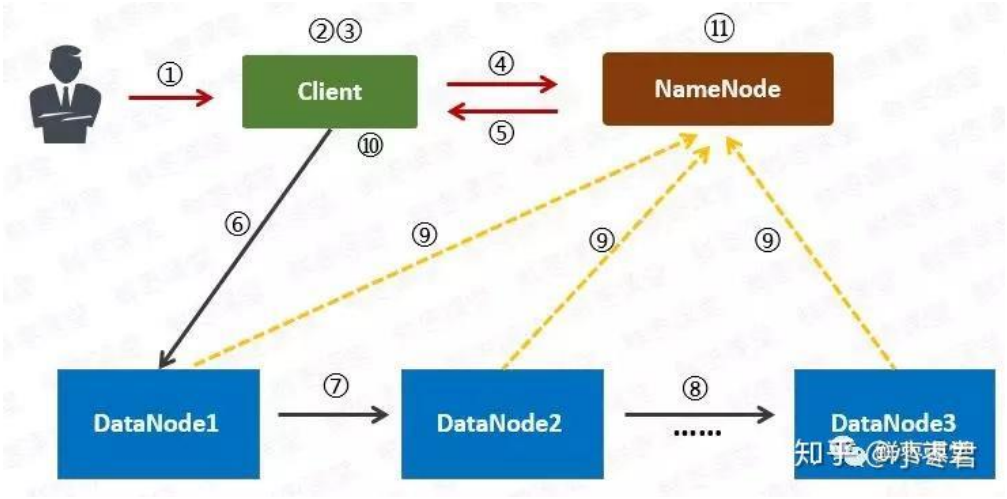
④ DataNode（worker 服务）。用于存储和管理本地磁盘上的 HDFS 块，将各个数据存储的健康状况和状态报告给 NameNode。

当从 HDFS 访问数据时，客户端应用程序必须先向 NameNode 发出请求，以在磁盘上定位数据。NameNode 将回复一个存储数据的 DataNode 列表，客户端必须直接从 DataNode 请求每个数据块。

还有其他的，比如 Secondary NameNode、ResourceManager、ApplicationMaster 等，分别负责集群的一部分任务。

（2）交互流程

① 写入流程


②③
⑪
①
Client
④
NameNode
⑤
⑩
⑥
⑨
⑨
⑨
DataNode1
⑦
DataNode2
⑧
.....
DataNode3

1 用户向Client（客户机）提出请求。例如，需要写入200MB的数据。

2 Client制定计划：将数据按照64MB为块，进行切割；所有的块都保存三份。

3 Client将大文件切分成块（block）。

4 针对第一个块，Client告诉NameNode（主控节点），请帮助我，将64MB的块复制三份。

5 NameNode告诉Client三个DataNode（数据节点）的地址，并且将它们根据到Client的距离，进行了排序。

6 Client把数据和清单发给第一个DataNode。

7 第一个DataNode将数据复制给第二个DataNode。

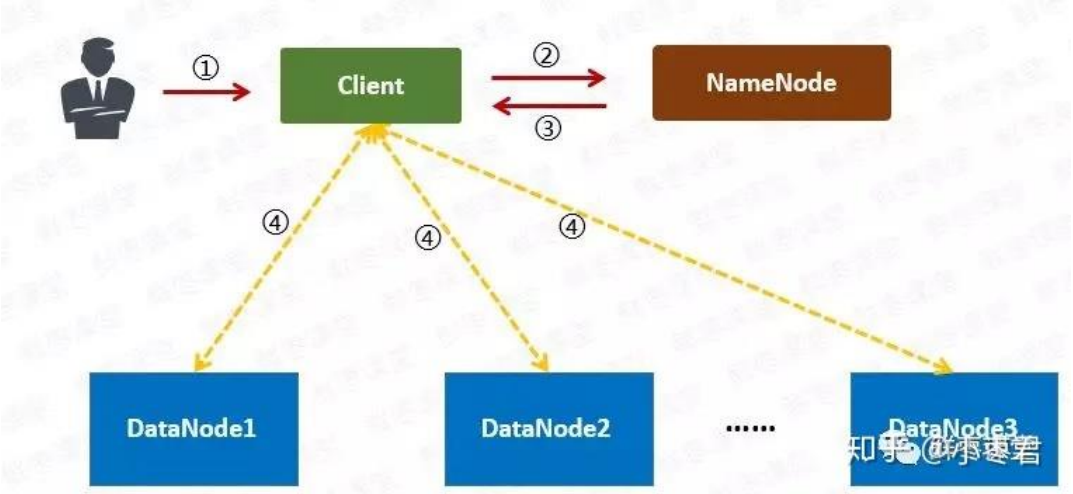
8 第二个DataNode将数据复制给第三个DataNode。

9 如果某一个块的所有数据都已写入，就会向NameNode反馈已完成。

10 对第二个Block，也进行相同的操作。

11 所有Block都完成后，关闭文件。NameNode会将数据持久化到磁盘上。

② 读取流程



1 用户向Client提出读取请求。

2 Client向NameNode请求这个文件的所有信息。

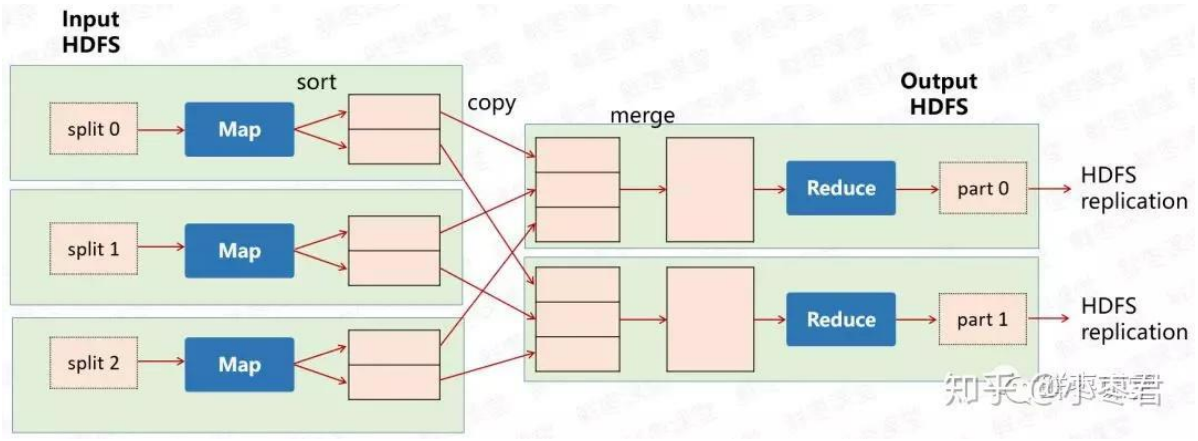
3 NameNode将给Client这个文件的块列表，以及存储各个块的数据节点清单（按照和客户端的距离排序）。

4 Client从距离最近的数据节点下载所需的块。

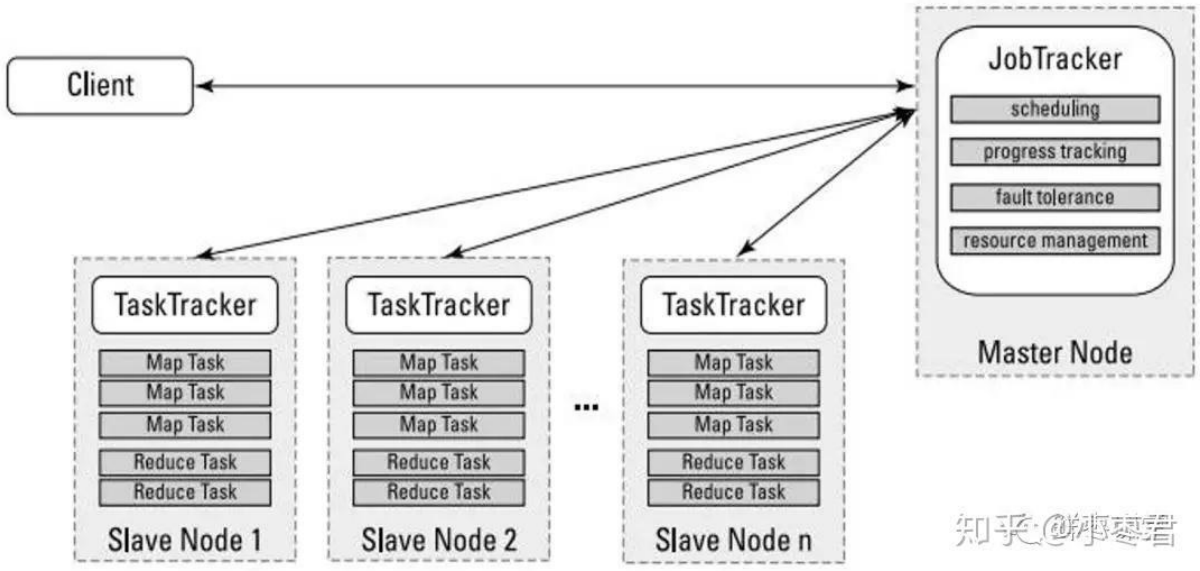
（注意：以上只是简化的描述，实际过程会更加复杂。）

（3）MapReduce

当你向MapReduce框架提交一个计算作业时，它会首先把计算作业拆分成若干个**Map任务**，然后分配到不同的节点上去执行，每一个Map任务处理输入数据中的一部分，当Map任务完成后，它会生成一些中间文件，这些中间文件将会作为**Reduce任务**的输入数据。Reduce任务的主要目标就是把前面若干个Map的输出汇总到一起并输出。



为了完成上面的任务，还需要引入两个角色：JobTracker 和 TaskTracker。



JobTracker 用于调度和管理其它的 TaskTracker。JobTracker 可以运行于集群中任一台计算机上。TaskTracker 负责执行任务，必须运行于 DataNode 上。

① client：用户编写的 MapReduce 程序通过 Client 提交到 JobTracker 端

用户可通过 Client 提供的一些接口查看作业运行状态。

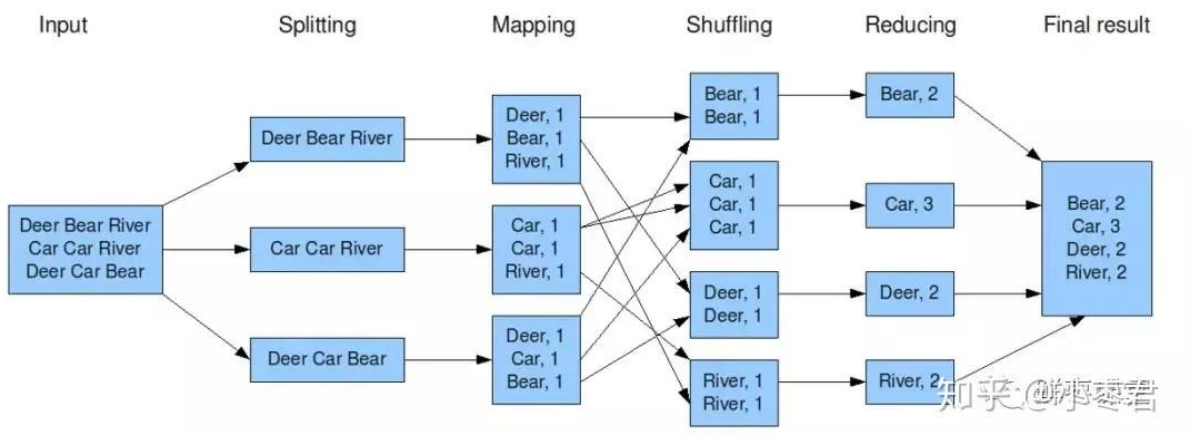
② JobTracker：对 TaskTracker 和 Job 进行资源监控和作业调度，当任务有问题就转移到其他节点进行，这些信息会告诉 TaskScheduler。

③ TaskTracker：会周期性地通过“心跳”将本节点上资源的使用情况和任务的运行进度汇报给 JobTracker，同时接收 JobTracker 发送过来的命令并执行相应的操作（如启动新任务、杀死任务等） TaskTracker 使用“slot”等量划分本节点上的资源量（CPU、内存等）。一个 Task 获取到一个 slot 后才有机会运

行，而 Hadoop 调度器的作用就是将各个 TaskTracker 上的空闲 slot 分配给 Task 使用。slot 分为 Map slot 和 Reduce slot 两种，分别供 MapTask 和 Reduce Task 使用。

④ Task：Task 分为 Map Task 和 Reduce Task 两种，均由 TaskTracker 启动。

4. Hadoop 实践：WordCount



上面是统计词频的任务。

（1）过程描述

① Hadoop 将输入数据切成若干片，并将每个 split（分割）交给一个 map task（Map 任务）处理。

② Mapping 之后，相当于得出这个 task 里面，每个词以及它出现的次数。

③ shuffle（拖移）将相同的词放在一起，并对他们进行排序，分成若干片。

④ 根据这些分片进行 reduce（归约）

⑤ 统计出 reduce task 的结果，输出到文件。

整个过程描述起来就像是一个老师有 100 份试卷要阅卷。他找来 5 个帮手，扔给每个帮手 20 份试卷。帮手各自阅卷。最后，帮手们将成绩汇总给老师。

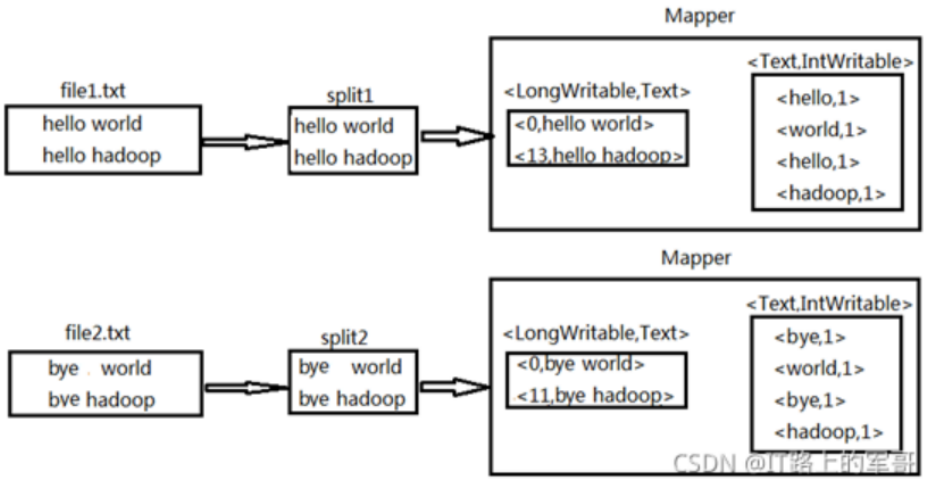
（2）进一步理解

表1 map函数和rudece函数

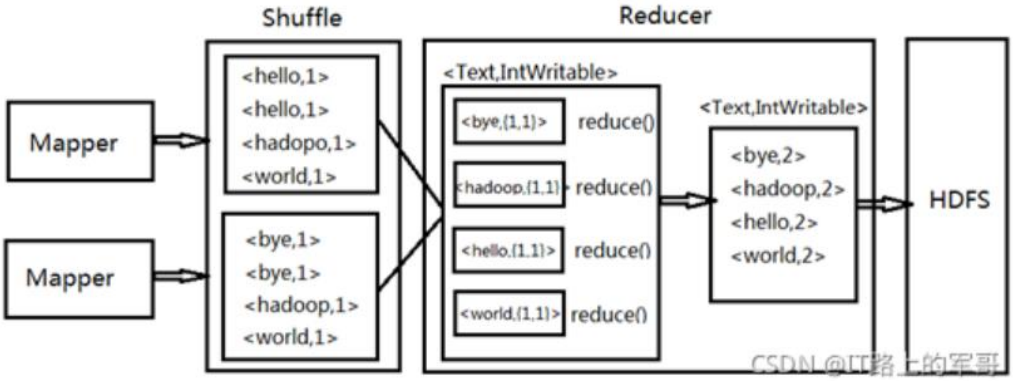
函数	输入	输出	说明
map	**<k1,v1>** <0,helle world> <12,hello hadoop>	**List<k2,v2>** <hello,1> <world,1> <hello,1> <hhadoop,1>	将获取到的数据集进一步解析成<key,value>,通过Map函数计算生成中间结果，进过shuffle处理后作为reduce的输入
reduce	**<k2,List(v2)>** <hadoop,1> <hello,{1,1}> <world,1>	**<k3,v3>** <hadoop,1> <hello,2> <world,1>	reduce得到map输出的中间结果，合并计算将最终结果输出HDFS，其中List(v2)，指同一k2的value

工作流程是Input从HDFS里面并行读取文本中的内容，经过MapReduce模型，最终把分析出来的结果用Output封装，持久化到HDFS中。

1.Mapper工作过程：



2.Reducer工作过程：




```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
```

```java
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

/*
* 将输入文本数据切分为单词，并为每个单词输出一个键值对，
* 其中键是单词本身，值是 1。这是 WordCount 程序的 Map 阶段的核心逻辑。
* 在 Reduce 阶段，这些键值对将被合并，以统计每个单词的总出现次数。
* */
public class WordCount {

  public static class TokenizerMapper

    // 输入键类型 Object、输入值类型 Text、输出键类型 Text、输出值类型
IntWritable
       extends Mapper<Object, Text, Text, IntWritable>{
//    one: 初始值为 1，将在 Mapper 中为每个单词输出一个计数值为 1 的键值对
    private final static IntWritable one = new IntWritable(1);
//    word: 保存每个被切分出的单词
    private Text word = new Text();
//    map 会被输入文件的每一行调用一次
//    @param Object key: 输入键，通常为文件中位置的偏移量
//    @param Text value: 输入值，文件中的一行文本
//    @param Context context: 写入 Mapper 输出的上下文
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
//      使用 StringTokenizer 将文本行切分成单词，toString 将 Text->字符串
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
//        将当前单词设置到 Text 类型的变量 word 中
        word.set(itr.nextToken());
//        通过 context 对象将键值对输出到 Mapper 的上下文，对每个单词，这里输出的
```

```
键是单词，值=1
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer // 实现了 Reducer
      extends Reducer<Text,IntWritable,Text,IntWritable> {
    // result: 存储每个单词的计数
    private IntWritable result = new IntWritable();

    /*
    * reduce 方法： 统计每个单词的总出现次数
    * @param Text key: 单词（输入键）
    * @param Iterable<IntWritable> values: 与该单词有关的所有值的迭代器
    * @param Context context: 写入 Reducer 输出的上下文
    * */
    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0; // 存储该单词的总计数
      for (IntWritable val : values) { // 遍历与当前单词关联的所有值，这些值是来自 Map 阶段的输出。
        sum += val.get(); // 将每个值加到总计数上
      }
      result.set(sum); // 将计算出的总计数设置到 IntWritable 类型的变量 result 中。
      context.write(key, result); // 通过 context 对象将键值对输出到 Reducer 的上下文。对于每个单词，这里输出的键是单词，值是该单词的总计数。
    }
  }

/*
```

```
* main: 负责配置和提交 MapReduce 作业
* */
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count"); // 创建作业对象，名称为 word count
    job.setJarByClass(WordCount.class); // 指定作业的 JAR 文件（即包含 WordCount 类的 jar 文件）
    job.setMapperClass(TokenizerMapper.class); // 设置作业的 Mapper 类
//    设置作业的 Combiner 类，Combiner 类似于本地的 Reducer，用于在 Map 阶段进行本地合并，减少数据传输到 Reducer 阶段的量。
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);  // 设置作业的 Reducer 类
    job.setOutputKeyClass(Text.class); // 设置作业的输出键类型
    job.setOutputValueClass(IntWritable.class); // 设置作业输出值类型
    FileInputFormat.addInputPath(job, new Path(args[0])); // 设置输入路径，从命令行参数获取
    FileOutputFormat.setOutputPath(job, new Path(args[1])); // 设置输出路径，从命令行参数获取
    System.exit(job.waitForCompletion(true) ? 0 : 1); // 提交作业并等待完成
//    waitForCompletion(true) 会阻塞程序直到作业执行完成。System.exit 会终止整个 Java 进程，返回作业的执行状态（0 表示成功，1 表示失败）
  }
}
```