并行计算复习————第四篇 并行计算软件支撑：并行编程_JCGuo的专栏-CSDN博客_并行计算编程软件

并行计算复习

第四篇 并行计算软件支撑：并行编程

Ch13 并行程序设计基础

13.1并行语言构造方法

- 库例程：MPI、Pthreads
- 扩展串行语言：Fortran90
- 加编译注释构造：OpenMP

13.2并行性问题

可利用SPMD来伪造MPMD

需要运行MPMD：parbegin S1 S2 S3 parend
可以改造成SPMD：

```
for i = 1 to 3 par-do
    if i == 1 then S1
    else if i == 2 then S2
    else if i == 3 then S3
endfor
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

那么并行扩展至需要支持SPMD即可

13.3交互/通信

（1）交互类型

- 通信：进程间传数（共享变量、消息传递、参数传递［父进程传给子进程］）
- 同步：进程间相互等待或继续执行的操作（原子同步、控制同步［同步障、临界区］、数据同步［锁、condition、监控程序和事件］）
- 聚合：将分进程所计算的的结果整合起来（规约、扫描）

（2）交互方式

- 同步交互：所有参与者全部到达后继续执行
- 异步交互：任意进程到达后不必等待其他进程即可继续执行

（3）交互模式

按编译时是否能确定交互模式可分为静态的、动态的

按多少发送者和接收者：

- 一对一：点到点通信
- 一对多：广播、散播
- 多对一：收集、规约
- 多对多：全交换、扫描、置换、移位

13.4并行编程风范

- 相并行：BSP模式，程序由一组超级步组成，步内各自并行计算，步间通信同步
- 主从并行：主进程串行执行并且协调任务，子进程计算任务，需要划分设计并结合相并行
- 分治并行：父进程把负载分割并指派给子进程，难以平衡负载
- 流水线并行：进程划分成流水线，依次依赖，数据开始流动
- 工作池并行：进程从工作池中取任务执行

13.5并行程序设计模型

（1）隐式并行

用串行语言编程，编译器货操作系统自动转化成并行代码

特点：语义简单、可以执行好、易调试易验证、but效率低

（2）数据并行

SIMD模型，包括数据选路和局部计算，特点：但现场、松散同步、常用聚合操作

数据并行计算π：

```
long i,j,t,N=100000;
double local[N], temp[N], pi, w;
w = 1.0/N;

forall (i = 0; i < N; i++) {
    local[i] = (i + 0.5) * w;
    temp[i] = 4.0 / (1.0 + local[i] * local[i]);
}

pi = reduce(temp, +);
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

（3）消息传递

MPP、COW自然模型，MPI广泛应用：多线程异步并行、地址空间分开、常用SPMD形式编码

MPI消息传递计算π：

```
#define N 100000
main(){
    double local=0.0,pi,w,temp=0.0;
    long i,taskid,numtask;

    w=1.0/N;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    MPI_Comm_Size(MPI_COMM_WORLD,&numtask);

    for (i = taskid; i < N; i=i + numtask){
        temp = (i+0.5)*w;
        local = 4.0/(1.0+temp*temp)+local;
    }

    MPI_Reduce(&local,&pi,1,MPI_Double,MPI_MAX,0, MPI_COMM_WORLD);

    if (taskid == 0) printf(“pi is %f \n”,pi*w);

    MPI_Finalize() ;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

（4）共享变量

PVP、SMP、DSM自然模型，多线程异步，显示同步而隐式通信

OpenMP使用共享变量计算π：

```
#define N 100000
main(){
    double local,pi=0.0,w;
    long i;

    w=1.0/N;

    #pragma parallel
    #pragma shared(pi, w)
    #pragma local(i, local)
    {
        #pragma parallel for (i = 0; i < N; i++)
        {
            local = (i + 0.5) * w;
            local = 4.0 / (1.0 + local * local);
        }
        #pragma critical
        {
            pi = pi + local
        }
    }
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

Ch14 共享存储系统并行编程——OpenMP编程

14.1 OpenMP编程风格

1.OpenMP编程模型

OpenMP是FORK-JOIN模型，主线程串行执行，直到编译制导并行域出现

2.并行域

并行域格式：

```
#pragma omp parallel [if (scalar_expression) | private(list) | shared(list) |
default(shared|none) | firstprivate(list) | reduction(operat)]
     1
```

线程数静态设定方法（线程号0～n-1）：

- 调用运行库omp_set_num_threads(numOfThreads)
- 设置环境变量setenv OMP_NUM_THREADS numOfThreads

3.常用运行库

在include头文件”omp.h”后可以调用OpenMP运行库：

- int omp_get_thread_num(void)获取并行域中当前线程的线程号
- int omp_get_num_threads(void)获取当前并行域使用的线程数

14.2 共享任务结构

OpenMP有三种典型的共享任务结构：

- for：在线程组中共享一个循环的多次迭代（数据并行模式）
- sections：把任务划分成离散段，每段由一个线程执行（功能并行模式）
- single：指定串行执行

注意：共享任务结构入口处没有同步障，但出口处有一个隐含的同步障

（1）for

编译制导的for语句指令紧跟它的循环语句由线程组并行执行，语法格式为：

```
#pragma omp for [schedule(type[, chunk]) | private(list) | shared(list) |
reduction(operation:list) | nowait]
     1
```

schedule指定划分方式，chunk指定size，若未指定则尽量平均分配

for编译制导语句必须在OpenMP并行域中，以向量加法举例：

```
#pragma omp parallel shared(a, b, c) private(i)
{
#pragma omp for schedule(dynamic, CHUNKSIZE)
for (i = 0; i < N; i++)
c[i] = a[i] + b[i];
}
```

（2）sections & single

sections编译制导语句可以作为任务划分方式，single编译制导语句用于标记非线程安全语句的串行化，略

（3）parallel for

与编译制导for的区别在于编译制导parallel for表明一个包含单独for语句的并行域，因此它不必在并行域中

上面的向量加法可以改写成编译制导parallel for的形式：

```
#pragma omp parallel for shared(a, b, c) private(i) schedule(dynamic, CHUNKSIZE)
for (i = 0; i < N; i++)
c[i] = a[i] + b[i];
```

14.3 同步结构

OpenMP提供很多同步控制编译制导语句，注意它们必须在并行域中才能使用

（1）master

master编译制导语句制定代码段只有主线程执行：

```
#pragma omp master
     1
```

（2）critical

critical编译制导语句指定代码段为临界区，仅有一个线程能够进入临界区，其他线程被阻塞

```
#pragma omp critical
     1
```

（3）barrier

barrier编译制导语句显式地声明一个同步障，所有线程均到达同步障后菜继续执行

```
#pragma omp barrier
     1
```

（4）atomatic

atomatic编译制导语句指定代码段为原子执行，表示该操作必须由一个线程原子执行，它只能作用于自增语句

```
#pragma omp atomatic
     1
```

14.4 数据域属性

OpenMP是共享存储模型，因此我们需要指定并行域中出现的变量是共享还是私有，大多数变量默认是共享的

（1）private

private子句表示变量是线程私有的（必须在这些变量声明后使用），那么这条编译制导语句会为每个线程复制一个私有副本，一个线程对private变量操作对其他线程是不可见的

最常用的private变量就是for循环里的循环控制变量i

（2）shared

shared子句指定变量是所有线程共享的，变量访问正确性由程序员保证

（3）reduction

reduction子句指定变量是规约的，并行段初始为所有线程创建一个副本，并行段结束时根据指定的operation对私有副本进行规约，最后存在改变量的全局值中

14.5 编程实例

举出一个用并行规约求π的OpenMP简例：

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
int i;
double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
for (i=0;i<num_steps; i++)
{
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17

Ch15 分布存储系统并行编程——MPI编程

MPI是一种消息传递接口的标准，适用于分布式存储系统的编程模型

与OpenMP不同的是，MPI是多进程的并行模式，运行时需要在外部指定开启进程数，并且是用SPMD的编程风格去模拟MPMD的编程风格（用进程号区别），不会FORK-JOIN而是通过消息传递同步

15.1 MPI基本函数

（1）启动函数

完成MPI启动，进入MPI并行环境：

```
int MPI_Init(int *argc, char **argv)
     1
```

直接把main函数的命令行传入参数argc和argv传入MPI_Init即可

（2）结束函数

完成MPI结束，退出MPI并行环境：

```
int MPI_Finalize(void)
     1
```

（3）获取进程编号

获取MPI进程编号来区别不同的进程：

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
     1
```

用取址运算&rank传入一个指针指定rank结果写入的地址

MPI_Comm是MPI通信域，通信域相当于消息传递范围，一般使用全局通信域MPI_COMM_WORLD

（4）获取总进程数

获取总进程数来，一般用于划分任务比例：

```
int MPI_Comm_size(MPI_Comm comm, int *size)
     1
```

和rank一样，用取址运算&size传入一个指针指定size结果写入的地址

（5）发送消息函数

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
     1
```

该函数把起始地址为buf的count个datatype类型的数据发送给目标进程，int是目标进程进程号，tag是这个消息的标签，comm指定消息传播的通信域

（6）接收消息函数

```
int MPI_Receive(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
     1
```

该函数从comm域中原进程source处接收标记是tag的消息，把该消息存储到起始地址为buf的count个datatype类型的数据缓冲区中

从发送消息和接收消息的API可以看出消息长度是由程序员手动维护的，一定要保证同tag发送的消息长度是一致的，否则会引发的缓冲区溢出

15.2 MPI点对点通信

MPI_Send和MPI_Recieve都是点对点通信，MPI点对点通信支持多种通信模式和通信机制

（1）通信模式

通信模式包括了发送方和接收方的缓存管理和同步方式，MPI有以下四种通信方式：

- 标准通信模式：MPI决定是否对发送数据缓存，而不是用户决定（大多数）；发送操作不管接收操作是否启动都可以执行
- 缓冲通信模式：需要申请一块缓冲区，发送方不必管接收方是否启动都可执行
- 同步通信模式：同步发送直到接受方接受过程已经启动才能返回
- 就绪通信模式：发送方必须等到接收方开始接收后才能发送，否则出错

（2）通信机制

发送和接收包括阻塞和非阻塞两种通信机制，阻塞必须等待通信完成后才能返回，而非阻塞不必等待操作是否完成就能返回

15.3 MPI集群通信

集群通信是指一个进程组（同一个通信域中）中所有进程都要参加的全局通信操作，MPI提供一套完整的API来实现集群的通信、聚集和同步等工作

（1）广播通信

```
int MPI_Bcast(void *address, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
     1
```

进程号为root的进程给comm域内所有进程（包括自己）发送一条长度为count的datatype类型的消息，address既是root发送数据的地址，也是所有进程接收数据的地址

（2）收集通信

```
int MPI_Gather(void *sendAddress, int sendCount, MPI_Datatype sendDatatype, void *recvAddress,
int recvCount, MPI_Datatype recvDatatype, int root, MPI_Comm comm)
     1
```

进程号为root的进程从comm域内所有进程（包括自己）收集一条消息，并且把数据依次存放在recvAddress为起始地址的长为count * size的datatype类型的缓冲区中

这个API比较奇怪的是给出了两个count和type，实际上这两个值必须保持一致啊？

（3）散播通信

```
int MPI_Scatter(void *sendAddress, int sendCount, MPI_Datatype sendDatatype, void *recvAddress,
int recvCount, MPI_Datatype recvDatatype, int root, MPI_Comm comm)
     1
```

进程号为root的进程从comm域内所有进程（包括自己）发送一条消息，这些发送消息依次排列在sendAddress为起始地址的长为count * size的datatype类型的缓冲区中

Scatter是Gather的反操作，之前提到的疑问同样存在

（4）全局收集通信

```
int MPI_Allgather(void *sendAddress, int sendCount, MPI_Datatype sendDatatype, void *recvAddress,
int recvCount, MPI_Datatype recvDatatype, MPI_Comm comm)
     1
```

全局收集相当于每个进程都执行一次Gather，每个进程send一条消息，每个进程接受size条消息依次存放在接收缓冲区中，Allgather后所有进程接受缓冲区的内容一致

（5）全局交换通信

```
int MPI_Alltoall(void *sendAddress, int sendCount, MPI_Datatype sendDatatype, void *recvAddress,
int recvCount, MPI_Datatype recvDatatype, MPI_Comm comm)
     1
```

全局交换相当于每个进程都进行一次Scatter，进程按进程号依次排列所有进程Scatter的消息，每个进程像所有进程发送发送缓冲区中依次排列的size条消息，每个进程接收缓冲区中接收来自所有进程各一个的消息，共size条

（6）同步障

```
int Barrier(MPI_Comm comm)
     1
```

显式地设置一个同步障

（7）规约聚合

```
int MPI_Reduce(void *sendAddress, void *receiveAddress, int count, MPI_Datatype datatype, MPI_Op
op, int root, MPI_Comm comm)
     1
```

该API定义了一个规约聚合操作，Op是规约运算操作（MPI内置），规约后保存在receiveAddress地址中

（8）扫描聚合

```
int MPI_Scan(void *sendAddress, void *receiveAddress, int count, MPI_Datatype datatype, MPI_Op
op, int root, MPI_Comm comm)
     1
```

扫描聚合是一种特殊的规约聚合，它让所有进程都做一次规约，进程号为i的进程对进程号在它前面的所有进程（0,…,i）做一次规约

15.4 MPI编程实例

下面是一个MPI实现计算的程序：

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

#define N 100000

int main(int argc, char** argv) {
    double local = 0;
    double pi, w, temp;
    int i, rank, size;
    int tag = 1001;

    w = 1.0 / N;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    for (i = rank; i < N; i += size) {
        temp = (i + 0.5) * w;
        local += 4.0 / (1.0 + temp * temp);
    }

    if (rank == 0) {
        pi = local;
        MPI_Status status;
        for (i = 1; i < size; i++) {
            double recv = 0.0;
            MPI_Recv(&recv, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD, &status);
            pi += recv;
        }
        printf("pi = %lf\n", pi * w);

    } else {
        MPI_Send(&local, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    }

    MPI_Finalize();

    return 0;
}
```

- 1
- 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

更多的例子见我另一篇博文：用MPI_Send和MPI_Recv实现简单集群通信函数

Ch16 GPU体系结构及编程——CUDA编程

16.1 基本概念

- GPU：图形处理器
- GPGPU：通用计算图形处理器，可编程、功能和性能不断完善，逐渐演化成一个新型并行计算平台

16.2 GPU体系结构

见我另外一篇博文：CUDA编程入门：向量加法和矩阵乘法

16.3 其他要点

CUDA编程入门中没有讲清楚的几点：

（1）SM的组成

SM（Stream Multiprocessor流多处理器）由以下部件组成：

- 8个SP(scalar processor),主频为1.35GHZ,所有 SP受控同一个指令单元,同步执行
- 两个SFU(special function unit)
- 一个指令cache(I cache)
- 一个常数cache(C cache) 8KB
- 一个纹理cache(T cache) 6~8KB
- 一个多线程发射单元(MT issue)

- 一个16KB的shared memory,用于线程块内共享数据, 访存速度很快
- 8192个32位字大小的寄存器文件供共享

（2）全局存储器的coalesced memory access

全局存储器的访问延迟较大，通常把全局数据加载到Shared Memory上供Block内的线程共享访问，从而达到提升性能的目的

除此之外还有一种方法：coalesced memory access

coalesced memory access（合并访问控制）是让线程从Global Memory中一次性取出连续的多个相同类型的数据，来掩盖多次访问Global Memory的高延迟

CUDA中定义了一些类型int2、int4、float2、float4等类型，线程可以直接对这些类型进行读取、计算和存储，或者读取存储用这些类型，在操作时先转换成普通类型的数组进行计算，然后再转换成这些联合访问类型

此外还有其他方法优化：

- 尽量增加SM上线程数量，尽可能体改实际并发运行的warp个数
- block内线程个数应该是warp size的整数倍
- 避免在一个warp中有分支语句

（3）共享存储器的存储体冲突

Shared Memory分成16个存储体（Bank），每个Bank按连续4byte循环分配（串行），不同的Bank支持并发访问

Shared Memory的访问速度很快，如果不存在存储体冲突则访问速度和寄存器一样

存储体冲突是指连续访问同一个Bank，会产生延迟

（4）执行机制中的WARP

每个线程块Block又分成若干个包含32个线程的组，称为WARP，WARP物理上以SIMD方式并行

（5）同步问题

CUDA中有一下几种同步方式

- CPU和GPU间的同步：cudaThreadSynchronize()设置CPU和GPU的同步障，CPU若要使用GPU计算结果，则用cudaThreadSynchronize阻塞CPU操作直到GPU执行完毕
- 线程块内同步：__syncthreads()设置块内同步障
- 线程块间的同步：CUDA不支持块间同步