并行计算复习资料 by 1611 MHY （不保证各题答案的正确性！）

第〇章 Activity1：

1. 谈谈你所知道的并行计算与云计算的区别?

并行计算是一台计算机，配备有多处理机，多处理机之间进行合同协作计算，最终结果由一台计算机处理。云计算指计算机通过网络发送计算命令给服务器，让服务器执行计算任务并将结果返还给发送命令的计算机。并行计算是由单个用户完成的，云计算是没有用户参与，交给网络另一端的服务器完成的。云计算由并行计算发展而来，是并行计算的商业实现。

2. 并行程序的描述应如何？与串行程序有什么不同?

瞎扯吧……

3. 如何并行地尽快求解 n 个元素的最大值或排序?

最大值：平衡树设计技术，两两比较求出最大值；

排序：均匀划分技术，PSRS 排序。

第一章 Activity2：

1. 再论 O(1)时间并行求解 n 个元素最大值算法，修改算法能否省去二维 B 数组? M 数组呢?

不可省去二维 B 数组，但 M 数组可以。因为 M 数组可以用 B 数组的对角元代替。

2. CMP、SMP 和 Cluster 在处理器、操作系统、并行编程上有什么不同？其他还有什么更多的区别?

见第一章 PPT 2-右上。

第二章 Activity 3：

1. 画出 8 个节点的超立方一维组织方式。
2. 如何设计一个 2×2×2 个处理器的度数均衡互连网络(节点度数均衡是指度数至多差 1)，用最少的连边数获得度数小于等于 3 的互连网络并使网络直径最小?

这不难，自己画吧…

第二章 Activity 12：

1. 试给出环上收集(all-to-one)的 CT 选路算法，并画出 8 个节点环上的选路步骤示意图(收集到节点 0)，以及推导环上的通信时间 。**（2019 春考试题）**

做一到多的逆操作即可。时间与一到多完全相同。

第三章 Activity4：文献阅读，略。

第四章 Activity5：文献阅读，略。

第五章 Activity6：文献阅读，略。

第六章 Activity10:

1. 序列 x1, x2, … , xn 前缀和串行算法的直接并行化可实现吗？又如何应用策略 2 和 3 进行并行化?

前缀和串行算法的直接并行化是不可实现的。因为串行程序是按照下标依次向前计算的，且每一步都要用到之前步的内容。实现并行化可以借助平衡二叉树。详见书 P199 算法 7.9.

（2019 春考试题：写出并行成本最优的前缀和并行算法。）提示：使用 n/logn 个处理器，先算内部前缀和。对这些处理器负责的元素做局和，使用书上给的二叉树算法继续计算，再转换回来。感谢 gjx 同学考前提供的此算法。

2. 上面策略 2 和 3 的并行算法，分别是在什么并行计算模型上实现的算法?

见算法描述。上面都写了。

第七章 Activity11:

1. 在 PRAM-CREW 模型上，用 n 个处理器在 O(1)时间内求出数组 A[1..n]={0,…,0,1,…,1}，最先为 1 值的下

标。写出并行伪代码。

```
for all Pi par-do
    if (A[i] == 0&& A[i+1] == 1) then
        return i
```

2. A 是一个大小为 n 的布尔数组，欲求出最小的下标 i 且 A[i]为真，试设计一个常数时间的 PRAM-CRCW 并行算法。如果使用 PRAM-CREW 模型，运行时间如何？

同 1，各处理器检查 A[i]是否为真，且 i 是否<结果。如果是，则写入。其运行时间为常数。

但如果换成 PRAM-CREW，不能并发写，则需要花时间处理冲突，运行时间为 O(n)。

第八章 无

第九章 Activity12:

1. 如果 $p(n)=n^2$，矩阵相乘的简单分块(Simple)、Cannon alg.和 Fox alg.并行算法时间各是多少?

简单乘法：PPT-33；Cannon：PPT-42；Fox：PPT-47。代入即可。

2. 在超立方结构上，为什么 Simple alg.比 Cannon alg.要快？而 Cannon alg.要比 Fox alg.快?

简单并行分块乘法是使用多到多播送传输数据的。经过$\log\sqrt{p}$次播送，数据可以传输到所有处理器。而 Cannon 需要进行旋转，旋转次数共$\sqrt{p}-1$次，所需次数多，所以慢。

简单并行分块算法是少次大规模数据传输，Cannon 是多次小规模数据传输。

Fox 是先旋转后多到多播送，且每一次旋转都要多到多播送一次，而 Cannon 不需要多到多播送，所以 Fox 慢。

3. 如何将分布式 Cannon alg.伪代码改造为共享存储式算法？并与 Simple alg.比较不同之处。

Cannon:

```
for all Pij par-do {
    Cij = 0
    for k = 0 to sqrt(p) - 1 do
        Cij += Ai, (i + j + k) mod sqrt(p) * B(i + j + k) mod sqrt(p), j
}
Simple Alg. //PRAM-CREW
for all Pij par-do {
    Cij = 0
    for k = 0 to sqrt(p) - 1 do
        Cij += Ai, k * Bk, j
}
```

Cannon 算法需要+i+j，后取模。因为 Cannon 乘法的本质是旋转，需要加上初始旋转位置，然后取模。

第十章 Activity13

1. 对于上三角方程组求解问题，如果使用块行带状划分，其计算效果与前面的循环行带状相比如何?

在上三角方程组求解问题中，越靠下的地方运算负载越小（见第三层循环），越靠上的地方运算负载越大。块行带状划分会导致负载出现不均衡，最上面的一块运行得最慢，成为瓶颈。

2. 如果前面的求前缀和问题可以归为一阶递归方程，这里三对角方程求解属于二阶递归方程，请同学们调研 n 阶递归方程的并行求解方法。

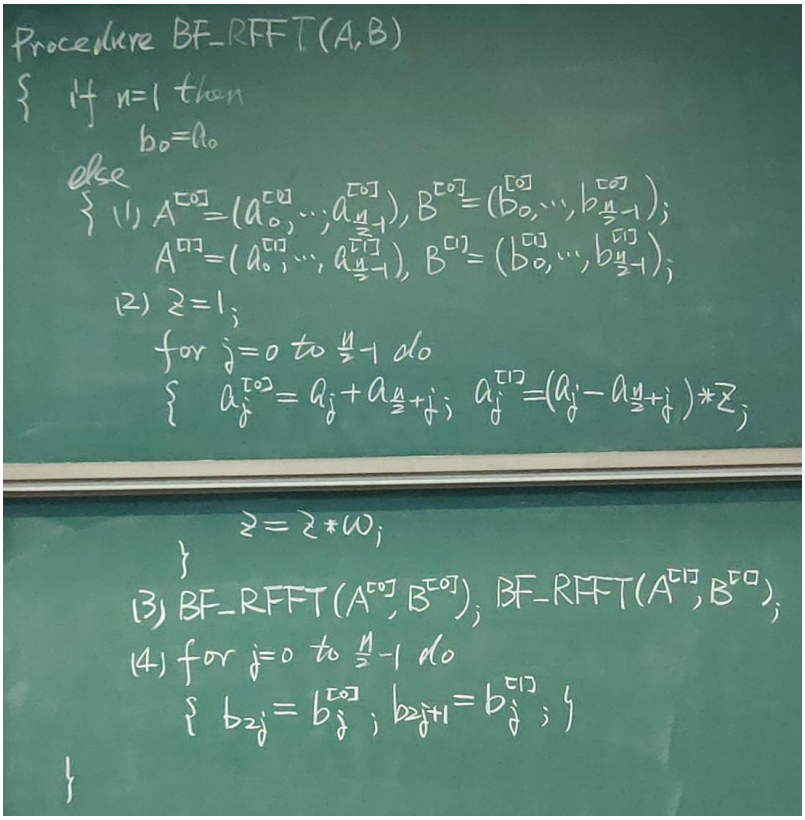
这是啥…?

3. 对于稀疏矩阵采用压缩编码，思考前面的矩阵乘向量、矩阵乘并行算法有什么影响？包括 GPU 算法。

（以下为瞎扯）这样不方便分块。如果每一次取值都要遍历一遍压缩编码矩阵，那么时间复杂度会提高；如果提前把压缩编码矩阵翻译成正常矩阵，那么空间复杂度又会提升。所以**只能为特定的压缩编码存储方式设计特定的并行算法予以计算。**

第十一章 Activity14

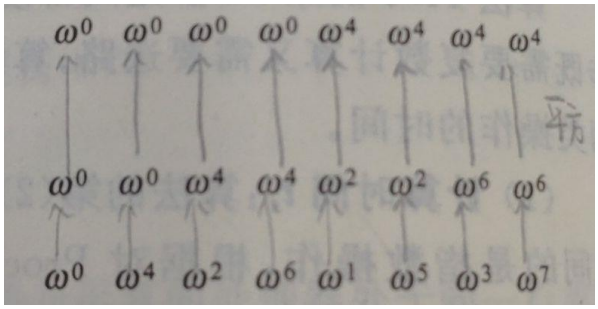
1. 证明串行 FFT 分治递归算法的正确性。
   这可咋整啊.jpg
2. 写出串行 FFT 蝶式递归算法的伪代码。



3. SIMD-BF 上的 FFT 算法在蝶形互连的分布式结构上 ω 权因子计算，除了各个节点独立计算方法之外，还有什么计算方法？该方法与前者的区别在哪里？ **（2019 春考试题）**

$$\omega_n^k = cos\frac{2k\pi}{n} + \mathrm{isin}\frac{2k\pi}{n}$$

所以可以每个节点在用的时候现场计算一个。
PPT 上给的方法是上层利用下层的计算 ω。第一层得自己算，然后上层利用下层的平方。



如果每个处理器知道自己的位置的话，前者是 O(1)的方法。如果不知道，就和后者一样，是 O(logn)。

4. SIMD-BF 上的 FFT 算法除了适应于蝶形互连的分布式结构，还有什么样的结构适合？
   我认为也适用于超立方连接。SIMD-BF 上的 FFT 计算方法类似于流水线。可以让不同的行使用同样的 n 个处理器进行计算。其连接方式恰好是超立方连接。

第十三章 无

第十四章 无

第十五章 Activity8：调研问题：

1. 没有提问，略。
2. MPI 标准或各种实现系统中有没有进程分配中的 CPU 亲缘性设置方法或函数?

MPI 中不能设置某处理器的父处理器或子处理器，但可以设置处理器间的拓扑结构。

实验一代码：OMP

```
void pi_parallel_section()
{
    int i, id;
    double x, pi, sum[NUM_THREADS];
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(NUM_THREADS);
//设置 2 线程
#pragma omp parallel private(i, id, x)
//并行域开始，每个线程(0 和 1)都会执行该代码
    {
        id = omp_get_thread_num();
        for (i = id, sum[id] = 0.0; i <
num_steps; i += NUM_THREADS)
        {
            x = (i + 0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for (i=0,pi= 0.0;i<NUM_THREADS;i++)
        pi += sum[i] * step;
}
```

```
void pi_parallel_task()
{
    int i;
    double pi, sum[NUM_THREADS];
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(NUM_THREADS);
//设置 2 线程
#pragma omp parallel  //并行域开始，每个
线程(0 和 1)都会执行该代码
    {
        double x;
        int id = omp_get_thread_num();
        sum[id] = 0;

#pragma omp for  //未指定 chunk，迭代平均
分配给各线程（0 和 1），连续划分
        for (i = 0; i < num_steps; i++)
        {
            x = (i + 0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for (i=0,pi= 0.0;i<NUM_THREADS;i++)
        pi += sum[i] * step;
}
```

```
void pi_parallel_partial()
{
    int i;
    double pi = 0.0,sum = 0.0,x = 0.0;
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(NUM_THREADS);
//设置 2 线程
#pragma omp parallel private(i, x, sum)
//该子句表示 x,sum 变量对于每个线程是私有的
    {
        int id = omp_get_thread_num();
        for (i=id,sum=0.0;i<num_steps;
i += NUM_THREADS)
        {
            x = (i + 0.5)*step;
            sum += 4.0 / (1.0 + x * x);
        }
    #pragma omp critical  //指定代码段在
同一时刻只能由一个线程进行执行
        pi += sum * step;
    }
}
```

```
void pi_parallel_reduction()
{
    int i;
    double pi = 0.0,sum = 0.0,x = 0.0;
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(NUM_THREADS);
//设置 2 线程
#pragma omp parallel for
reduction(+:sum) private(x) //每个线程保
留一份私有拷贝 sum，x 为线程私有，最后对线程
中所以 sum 进行+规约，并更新 sum 的全局值
    for (i = 1; i <= num_steps; i++)
    {
        x = (i - 0.5)*step;
        sum += 4.0 / (1.0 + x * x);
    }
    pi = sum * step;
    printf("%.8lf\n", pi);
}
```

```
void PSRS(int *_array, int _length, int p)
{
    int* privot = new int[p];
    int* sample = new int[p*p + 10];
    int* tmpstr = new int[_length + 10];

    omp_set_num_threads(p);  //设置线程数量
#pragma omp parallel shared(_array, privot, sample, tmpstr)
    {
        //均匀划分
        int id = omp_get_thread_num();
        int *head = _array + (id*_length / p);
        int len = (id + 1)*_length / p - (id*_length / p);
        //菊部排序
        qsort((int *)head, len, sizeof(int), cmp);
        //选取样本
        for (int i = 0; i < p; i++)
            sample[id * p + i] = *(head + i * len / p);
#pragma omp barrier
#pragma omp master
        {
            //样本排序
            qsort(sample, p*p, sizeof(int), cmp);
            //选择主元
            for (int i = 0; i < p - 1; i++)
                privot[i] = sample[(i + 1)*p - 1];
        }
#pragma omp barrier
        //主元划分
        int headpos = id * _length / p;
        int tailpos = (id + 1)*_length / p;

        for (int i = 0; i < p; i++)
            sample[id*p + i] = headpos;

        for (int i = headpos, j = 0; i < tailpos; i++){
            sample[id*p + j + 1] = i;
            for (; _array[i] > privot[j] && j < p - 1; j++)
                if (j < p - 2)
                    sample[id*p + j + 2] = sample[id*p + j + 1];
            if (j == p - 1)
                break;
        }
        sample[p*p] = _length;
#pragma omp barrier
        //全局交换
        volatile int movdest = 0, destbkp;
        for (int j = 0; j < p; j++)
            movdest += (sample[j*p + id] - sample[j*p]);
```

```
        destbkp = movdest;
        for (int i = 0; i < p; i++)
        {
            headpos = sample[i*p + id];
            tailpos = sample[i*p + id + 1];
            for (volatile int j = headpos; j < tailpos; j++, movdest++)
                tmpstr[movdest] = _array[j];
        }
        //归并排序
        qsort(tmpstr + destbkp, movdest - destbkp, sizeof(int), cmp);

#pragma omp barrier
        for (int i = id; i < _length; i += p)
            _array[i] = tmpstr[i];
    }
}
```

实验二代码：MPI

```
int main(int argc, char** argv){
    int group_size, my_rank;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size( MPI_COMM_WORLD, &group_size);
    n=2000; /* Broadcast n to all other nodes */
    MPI_Bcast(&n,1,MPI_LONG,0,MPI_COMM_WORLD); h = 1.0/(double) n; sum = 0.0;
    for (i = my_rank+1; i <= n; i += group_size)
         x = h*(i-0.5); sum = sum +4.0/(1.0+x*x);
    mypi = h*sum;
    /*Global sum */
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    if(my_rank==0)
    { /* Node 0 handles output */
        printf("pi is approximately : %.16lf\n",pi);
    }
     MPI_Finalize();
}
```

```
void PSRS(int *_array, int _length){
    int p, id;
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Bcast(_array, _length, MPI_INT, 0, MPI_COMM_WORLD);

    //均匀划分
    int div_shift = (id*_length / p);
    int div_len = (id + 1)*_length / p - (id*_length / p);

    //菊部排序
    int *my_arr = (int*)malloc(div_len*sizeof(int));
    memcpy(my_arr, _array + div_shift, div_len*sizeof(int));
```

```
qsort(my_arr, div_len, sizeof(int), cmp);

//选取样本
int* sample = (int*)malloc(p*sizeof(int));
for (int i = 0; i < p; i++)
    sample[i] = *(my_arr + i * div_len / p);
int *collectedsample = (int*)malloc(p*p*sizeof(int));
int *privot = (int*)malloc(p*sizeof(int));
MPI_Barrier(MPI_COMM_WORLD);
MPI_Gather(sample, p, MPI_INT, collectedsample, p, MPI_INT, 0, MPI_COMM_WORLD);

if (id == 0){
    //样本排序
    qsort(collectedsample, p*p, sizeof(int), cmp);
    //选择主元
    for (int i = 0; i < p - 1; i++)
        privot[i] = collectedsample[(i + 1)*p] ;
}
//广播主元
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(privot, p, MPI_INT, 0, MPI_COMM_WORLD);

int *send_len = (int*)malloc(p*sizeof(int));
int *send_shift = (int*)malloc(p*sizeof(int));
//主元划分
for (int i = 0; i < p; i++){
    send_len[i] = 0;
    send_shift[i] = 0;
}
for (int i = 0, j = 0; i < div_len; i++){
    send_shift[j + 1] = i;
    for (; my_arr[i] > privot[j] && j < p - 1; j++)
        if (j < p - 2)
            send_shift[j + 2] = send_shift[j + 1];
    send_len[j]++;
}
//取得全局长度
int *glbl_len = (int*)malloc(p*p*sizeof(int));
MPI_Allgather(send_len, p, MPI_INT, glbl_len, p, MPI_INT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
//
int *recv_len = (int*)malloc(p*sizeof(int));
int *recv_shift = (int*)malloc(p*sizeof(int));

for(int i = 0; i < p; i++)
    recv_len[i] = glbl_len[i*p + id];

recv_shift[0] = 0;
for(int i = 1; i < p; i++)
```

```
        recv_shift[i] = recv_shift[i - 1] + recv_len[i - 1];

    int merge_len = recv_shift[p - 1] + recv_len[p - 1];
    int* merge_str = (int*)malloc(merge_len*sizeof(int));
    MPI_Alltoallv(my_arr, send_len, send_shift, MPI_INT, merge_str, recv_len, recv_shift,
MPI_INT, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    //归并排序
    qsort(merge_str, merge_len, sizeof(int), cmp);
    MPI_Barrier(MPI_COMM_WORLD);

    //接收
    MPI_Gather(&merge_len, 1, MPI_INT, recv_len, 1, MPI_INT, 0, MPI_COMM_WORLD);
    recv_shift[0] = 0;
    for(int i = 1; i < p; i++)
        recv_shift[i] = recv_shift[i - 1] + recv_len[i - 1];
    MPI_Gatherv(merge_str, merge_len, MPI_INT, _array, recv_len, recv_shift, MPI_INT, 0,
MPI_COMM_WORLD);
}
```

实验三：GPU

```
__device__  int GetSubMatrix(int m, int index, int width){
    return  width * index * BLOCK_WIDTH + m * BLOCK_WIDTH;
}
__global__ void matrixMul(int* _A, int* _B, int* _C, unsigned int width){
    //shared memory
    __shared__ int A_shared[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ int B_shared[BLOCK_WIDTH][BLOCK_WIDTH];
    // 线程块 block
    int bx = blockIdx.x, by = blockIdx.y;
    // 线程在块中编号 thread
    int tx = threadIdx.x, ty = threadIdx.y;
    int pos = width * tx + ty;
    int C_value = 0;
    for (int i = 0; i < width / BLOCK_WIDTH; i++)
    {
        A_shared[tx][ty] = _A[GetSubMatrix(i, bx, width) + pos];
        B_shared[tx][ty] = _B[GetSubMatrix(by, i, width) + pos];
        __syncthreads();
        for (int k = 0; k < BLOCK_WIDTH; k++)
            C_value += A_shared[tx][k] * B_shared[k][ty];
        __syncthreads();
    }
    _C[GetSubMatrix(by, bx, width) + pos] = C_value;
}
```

Main 函数主要操作：

```
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);
    dim3 dimGrid(width / BLOCK_WIDTH, width / BLOCK_WIDTH);
    matrixMul <<< dimGrid, dimBlock >>> (A_D, B_D, C_D, width);
```