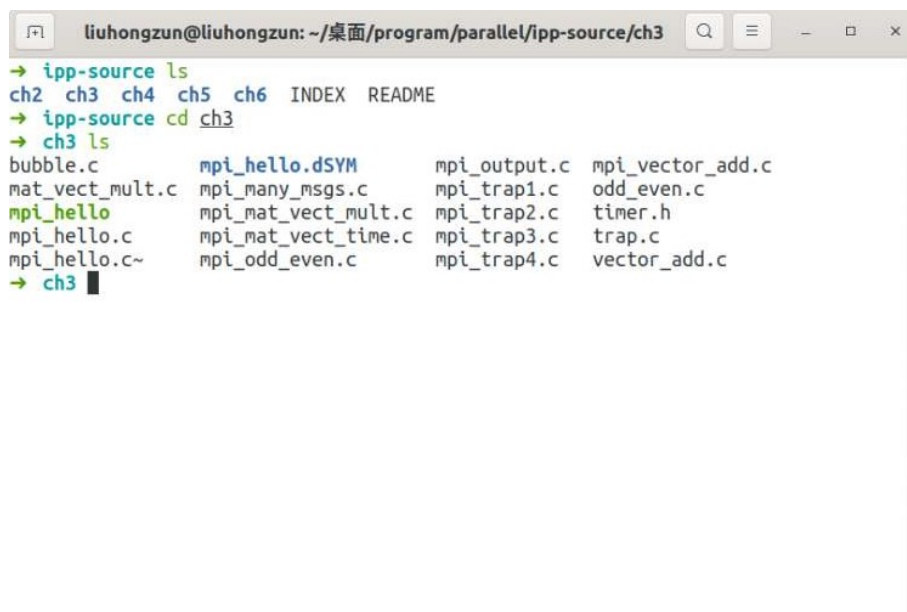


高性能计算导论: Homework 1

刘泓尊 2018011446 计 84

1. 命令行截图如下:



```
liuhongzun@liuhongzun: ~/桌面/program/parallel/ipp-source/ch3
→ ipp-source ls
ch2 ch3 ch4 ch5 ch6 INDEX README
→ ipp-source cd ch3
→ ch3 ls
bubble.c      mpi_hello.dSYM      mpi_output.c  mpi_vector_add.c
mat_vect_mult.c  mpi_many_msgs.c     mpi_trap1.c   odd_even.c
mpi_hello      mpi_mat_vect_mult.c  mpi_trap2.c   timer.h
mpi_hello.c     mpi_mat_vect_time.c  mpi_trap3.c   trap.c
mpi_hello.c~    mpi_odd_even.c       mpi_trap4.c   vector_add.c
→ ch3
```

2. 当 $n\%p == 0$ 时, 每个核可以实现均分, 即每个核的计算区间长度 n/p . 这时,

$$my_fisrt_i = (n/p) * i, my_last_i = (n/p) * (i + 1), \quad 0 \leq i \leq p, \quad n\%p == 0$$

当 $n\%p \neq 0$ 时, 可以先对每个核分配 n/p 个元素, 之后将剩下的 $n\%p$ 个元素分到前 $n\%p$ 个核, 这时我们可以推导出

$$my_fisrt_i = \begin{cases} (n/p + 1) \cdot i & 0 \leq i < n\%p \\ n/p \cdot i + n\%p & n\%p \leq i < p \end{cases} \quad (1)$$

由于循环代码采用‘左开右闭’区间, 所以 last 可以直接将 i 换为 $i+1$ 即可

$$my_last_i = \begin{cases} (n/p + 1) \cdot (i + 1) & 0 \leq i < n\%p \\ n/p \cdot (i + 1) + n\%p & n\%p \leq i < p \end{cases} \quad (2)$$

3. 显然, 由于每个核接收一个数据做一次加法, 所以每个核接收的次数就等于加法的次数。

a. 最初的全局总和:

0 号核需要对所有核的数据进行加总, 所以需要接收其他 $p - 1$ 个核的数据, 接受和加法次数为 $p - 1$.

b. 树形结构求全局总和:

由树形结构可知, 0 号核总接收次数等于树高, 而满二叉树的树高是 $\log(p)$, 所以 0 号核

接收和加法的次数是 $\log(p)$.

对不同核数的接收或加法次数汇总如下表:

p	0 号核接收或加法的次数	
	最初做法 ($p - 1$)	树形结构 ($\log(p)$)
2	1	1
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
256	255	8
512	511	9
1024	1023	10

4. 我认为树形结构第二阶段是“数据并行”和“任务并行”兼备的例子。

a. 一方面，在树形结构的非叶子的同层之间，该层对应的每个核都进行了数据的接受、求和、发送，它们所做的任务的相同的，所以可以看做“数据并行”。

b. 另一方面，树形结构的叶子节点中，有一半的核只进行了数据发送，并没有数据接收，而另一半的核有数据的接受、求和以及发送，它们执行的任务不同；不同的核执行计算的次数也不相同:0 号核执行求和的次数最多，为 $\log(p)$ ，而其他核执行求和的次数都比 0 号核少，有的甚至只有发送，没有求和。所以也可以看做“任务并行”。

5. 我曾用 C++ 实现过词嵌入算法 **Word2vec**，在该算法的流程中，首先我需要读取一个 2G 的大规模语料，尽管 c++ 读取文件的速度相当可观，但是我依然采用了多线程技术 (*pthread*) 加快了这一过程，具体操作是将文件按大小均为 p 个部分，为了使得分割尽可能均匀，我采用了第 2 题所示的分割方法。我的电脑有 12 个核心，根据我的测试，读取 2G 的语料库时 8 线程比单线程快出 6 倍以上；其次，在训练 word2vec 的神经网络时，需要按照语料中的句子 (或称“窗口”)，我也采用多线程的方式，按照上述读取文件的方法，将文件中的句子并行地进行处理与训练。8 线程的测试速度可以达到单线程的 5-6 倍。当然，这其中加了很多互斥锁。由于读取文件和训练网络的工作是相同的，所以这是“数据并行”的方式。

我曾用 C++ 实现过图分析中的社群发现算法 **FastUnfolding**，可以看做对图中节点做聚类。在该算法中，需要对图中节点遍历而不断修改社群，使得图的“模块度”不断上升。在其中，遍历节点的顺序对实验结果有明显影响，不好的遍历顺序会使得结果陷入“局部最优解”。所以我采用随机遍历的方式，即先将节点进行一遍 *shuffle*，之后再遍历。为了得到全局最优解，我将任务分配到多个线程，每个线程生成不同的随机排列，分别寻找最优解，之后将结果返回给主线程，主线程选择最优的一个解。在这里，多个线程的训练任务可以看做“数据并行”。多个线程将结果发送给主线程，主线程选择最优解的过程可以看做“任务并行”，与课上讲的例子类似。

高性能计算导论: Homework 2

刘泓尊 2018011446 计 84

1. a.

Operation	Time Use/ns
Fetch operands	2
Compare exponentst	1
Shift one operand	1
Add	1
Normalize result	1
Round result	1
Store result	2
Total	9

需要 9ns 时间完成一次加法。

b. 需要

$$1000 \times 9 = 9000ns$$

c. 流水线操作:

Time/ns	Fetch	Compare	Shift	Add	Normalize	Round	Store
0-1	1						
1-2	1						
2-3	2	1					
3-4	2		1				
4-5	3	2		1			
5-6	3		2		1		
6-7	4	3		2		1	
7-8	4		3		2		1
8-9	5	4		3		2	1
9-10	5		4		3		2
10-11	6	5		4		3	2
11-12	6		5		4		3

从上表中可以看到, 稳定情况下每对加法会新增加 2ns 的计算开销。n 对浮点数加法会有

$$2(n - 1) + 9 = 2n + 7(ns)$$

所以 1000 对浮点运算会花费 2007ns 时间。

d.

如果认为判断缓存缺失可以瞬间完成的话,

- 一级缓存缺失但二级缓存命中会花费 5ns
- 二级缓存缺失但主存命中需要花费 50ns

对于流水线而言, Fetch 操作会耗费更长的时间, 会阻塞后续流水线的工作, 延长整个工作完成的时间。

2. a. 以 $MAX = 8$ 为例, 当缓存变大即缓存行的个数变多时, 方式 1 读取矩阵元素每行仍会发生 2 次 miss, 也就是说, 方式 1(按行读取) 的每行 miss 次数与缓存行个数无关, 而与缓存行的大小有关;

方式 2 读取时, 如果缓存行个数 $= 8$ 个, 那么读取完第一列之后, 矩阵 8 行的前 4 个元素都会位于缓存中, 这时前 4 列、后 4 列各会发生 8 次 miss, 总共 16 次 miss, 显著减少了 miss 次数;

当矩阵大小变大时, 两种方式 miss 的次数都会增加。

- b. 如果 $MAX = 8$, 按照方式 1 读取矩阵元素时, 每一行会产生 2 次 miss。矩阵总共 8 行, 因此会发生 $2 \times 8 = 16$ 次 miss。

按照方式 2 读取矩阵元素, 由于只有 4 个缓存行, 所以读每列时都会发生 miss, 总共会发生 $8 \times 8 = 64$ 次 miss。

3. 最多页数

$$Pages = 2^{VirtualPageNumber} = 2^{20} = 1048576$$

4. a.

发送信息和运算都是同步进行的, 所以总共需要的时间为

$$t = t_{send} + t_{op} = 10^{-9} \times 10^9(p-1) + \frac{10^{12}}{p}/10^6 = p-1 + \frac{10^6}{p} = 1999s$$

b.

$$t = t_{send} + t_{op} = 10^{-3} \times 10^9(p-1) + \frac{10^{12}}{p}/10^6 = 10^6(p-1) + \frac{10^6}{p} = (999 \times 10^6 + 10^3)s \approx 31.68year$$

5. 设 p 为处理器个数

a. 对于环 (ring), 交换器个数 $n = p$, 链路个数 $e = p$

b. 对于二维环面网格 (toroidal mesh), 交换器个数 $n = p$, 链路个数 $e = 2p$

c. 对于全相连网络 (fully connected network), 交换器个数 $n = p$, 链路个数 $e = \frac{p(p-1)}{2}$

d. 对于超立方体 (hypercube), 交换器个数为 $n = p = 2^d$, 其中维度为 d 。下面对链路个数 (边数) 进行归纳:

从 d 维超立方体构造 $d+1$ 维超立方体, 其边数为

$$e_{d+1} = 2e_d + n = 2e_d + 2^d$$

递归基础为 $e_1 = 1$ 解得

$$e_d = d2^{d-1} = \frac{1}{2}p \log p$$

e. 对于交叉开关矩阵 (Crossbar), 其交换器个数为 $n = p^2$, 网络共 p 行, 每行 $p-1$ 个节点, 除去第一列和最后一列外, 每个节点均有 2 个出边。故链路数

$$e = 2p^2 - 2p = 2p(p-1)$$

f. 对于 Omega Network, 网络共有 $\log p$ 级, 每级 $p/2$ 个节点, 所以 2×2 交换器个数 $n = \frac{1}{2}p \log p$, 除去最后一列外, 每个交换器都有 2 个出边, 所以链路数

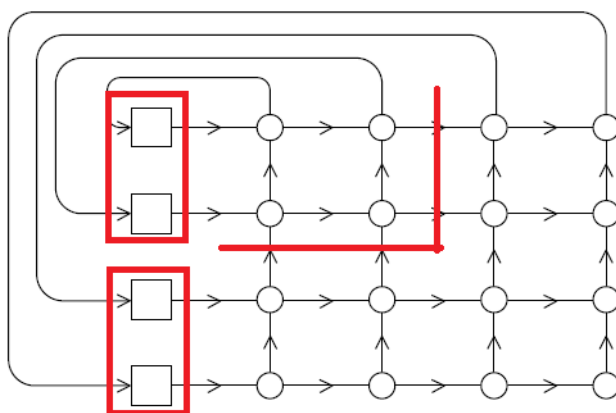
$$e = 2 \cdot \frac{1}{2}p \log p - p = p(\log p - 1)$$

上述结果汇总如下:

Distributed-memory interconnects	Switches	Links
Ring	p	p
Toroidal Mesh	p	$2p$
Fully Connected Network	p	$\frac{1}{2}p(p-1)$
HyperCube	p	$\frac{1}{2}p \log p$
Crossbar	p^2	$2p(p-1)$
Omega Network	$\frac{1}{2}p \log p$	$p(\log p - 1)$

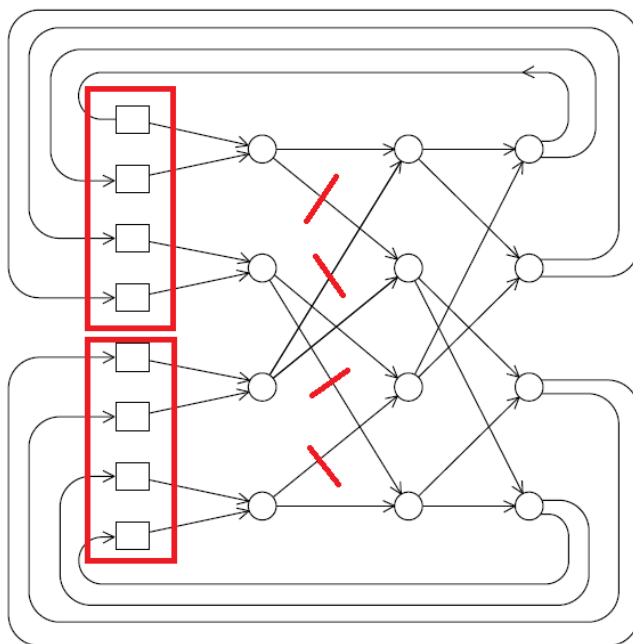
6. 对于 p 个处理器的交叉开关矩阵，将其上下侧均分为 2 部分，每部分 $p/2$ 个处理器，那么将矩阵的连接如图切断，下侧处理器和上侧处理器无法实现互联，再减少切断个数将不能实现完全分割。所以交叉开关矩阵的等分宽度为 p 。

即 8×8 交叉开关矩阵的等分宽度为 8



对于 $p = 8$ 的 Omega Network，将其均分为上下两组，从图中标注处进行切断，可以实现上下两组之间无法通信。

所以对于 8×8 的 Omega Network，等分宽度为 4



7. a. $y = 5$

由于采用监听缓存一致性协议，0 号核修改 x 的值之后在总线上进行广播，1 号核于是将变量 x 所在的缓存行标记为“无效”。1 号核取出 x 的值时，发生“读缺失”，1 号核在总线上广播。此时 0 号核检查到自己的缓存中有 x 变量，因此 0 号核将 x 写回内存，并将相应缓存行标记为“Valid”。1 号核随即再从内存中读出 x 。因此 1 号核执行 $y = x$ 得到 $y = 5$ 。

b. $y = 5$

采用基于目录的协议时，目录记录了缓存行在哪些核的 cache 里。0 号核修改 x 后，缓存控制器会使得 x 所在缓存行标记为“无效”。1 号核读取 x 的时候，发生“读缺失”，1 号核采用和“监听一致性协议”相同的方法，在总线上广播使得 0 号核将 x 的值写回内存，并将相应缓存行标记为“Valid”。之后 1 号核再从内存中读取。因此 1 号核执行 $y = x$ 得到 $y = 5$ 。

8. 解:

$T_s = n, T_p = \frac{n}{p} + \log p$, 所以

$$\frac{T_s}{T_p} = \frac{n}{\frac{n}{p} + \log p} = \frac{np}{n + p \log p}$$

$$E = \frac{T_s}{T_p} / p = \frac{n}{n + p \log p}$$

当 p 变为原来的 k 倍，即 $p' = kp$ 时，为保证效率不变

$$\frac{n'}{n' + kp \log(kp)} = \frac{n}{n + p \log p}$$

即

$$\frac{n'}{n} = \frac{kp \log(kp)}{p \log p} = \frac{k \log k}{\log p} + k > 1$$

所以 n 扩大为原来的 $\frac{k \log k}{\log p} + k$ 倍。

当 p 从 8 扩大到 12 时， $k = 2$ ，代入得:

$$\frac{n'}{n} = \frac{2 \log(2)}{\log(8)} + 2 = \frac{8}{3}$$

故， n 扩大为原来的 $8/3$ 倍。

由于可以保证效率不变，所以该程序是“可拓展的”。

p 每增加 1 倍， n 将扩大 $\frac{k \log k}{\log p} + k \sim k \log k$ 倍，所以为保证效率不变导致的规模增加是比较大的，超过了线性，所以不是“弱可拓展”的。

高性能计算导论: Homework 3

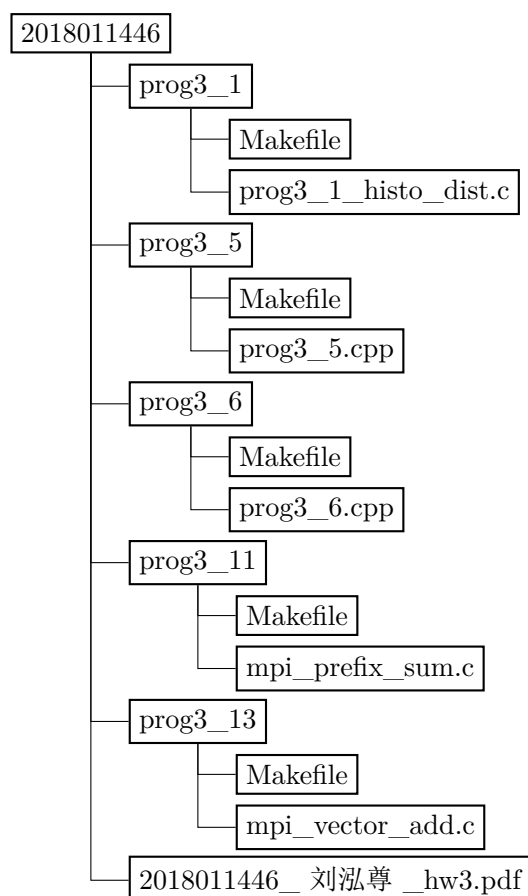
刘泓尊 2018011446 计 84

2020 年 3 月 18 日

目录

1 File Structure	2
2 Exercise 3.13	2
2.1 运行方式	2
2.2 代码思路	2
2.3 测试结果	3
3 Exercise 3.11(d)	4
3.1 运行方式	4
3.2 代码思路	4
3.3 测试结果	4
4 Exercise 3.1	5
4.1 运行方式	5
4.2 代码思路	5
4.3 测试结果	6
5 Exercise 3.5	7
5.1 运行方式	7
5.2 代码思路	7
5.3 测试结果	8
5.4 性能评估	8
6 Exercise 3.6	11
6.1 运行方式	11
6.2 代码思路	11
6.3 测试结果	12
6.4 性能评估	13

1 File Structure



2 Exercise 3.13

2.1 运行方式

./prog3_13 下放有 makefile, 在该目录下执行 “**make**” 可以得到可执行程序; 之后执行 “**make run p=...**” 可以运行程序。其中 p 的值可以任选, 如 “**make run p=10**”; 运行程序后, 会有提示进一步输入 n; 执行 “**make clean**” 可以删除生成的.o 文件和可执行程序。

在本程序中, 为了测试更普适的条件, 我将两个向量设为程序**随机生成**, 而不是从 stdin 读入。

2.2 代码思路

为了实现 n 可以不被 p 整除的情况, 我使用 MPI_Scatterv() 和 MPI_Gatherv() 函数实现向量元素的分发与收集。这两个函数需要显式指定每个进程得到的元素个数 *count* 和相对原向量的偏移量 *disp*。

为了实现数据比较均匀的分配, 我使用作业 1 中一个题目的做法, 先为每个进程分配 n/p 个元素, 之后再将 $n - (n/p) \cdot p$ 个元素给前 $n - (n/p) \cdot p$ 个进程各一个。并以此为依据构建 *disp* 偏移量。

具体代码如下:

Allocate_count_disp()

```

1 int overflow = n - (n / comm_sz) * comm_sz;
2 int base = n / comm_sz;
3 for(i = 0; i < comm_sz; i++){
4     (*count)[i] = base;
5 }
6 for(i = 0; i < overflow; i++){
7     (*count)[i] += 1;
8 }
9 for((*disp)[0] = 0, i = 1; i < comm_sz; i++){
10     (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
11 }

```

Read_vector()

```

1 MPI_Scatterv(a, count, disp, MPI_DOUBLE, local_a, count[my_rank],
    MPI_DOUBLE, 0, comm);

```

Print_vector()

```

1 MPI_Gatherv(local_b, count[my_rank], MPI_DOUBLE, b, count, disp,
    MPI_DOUBLE, 0, comm);

```

2.3 测试结果

下例给出了 $p = 4, n = 15$ 时的运行结果。可以看到，程序实现了在 n 不被 p 整除的情况下的向量加法。二范数误差 $error < 1e^{-6}$ 。

```

1 [2018011446@bootstraper prog3_13]$ make run p=4
2 srun -n 4 -l ./vectoradd
3 0: What's the order of the vectors?
4 15
5 0: x is
6 0: 0.887164 0.060926 0.839742 -0.539429 -0.333756 -0.310390 0.007394
   -0.574869 0.205154 0.839252 -0.208210 0.947841 0.851419 0.031242
   0.545745
7 0: y is
8 0: -0.343533 -0.527309 -0.899562 0.181500 0.743957 0.918453 0.086991
   -0.257361 -0.569818 -0.679560 0.243786 -0.304110 0.199846
   -0.652993 0.659348
9 0: The sum is
10 0: 0.543631 -0.466383 -0.059820 -0.357929 0.410201 0.608063 0.094386
   -0.832231 -0.364665 0.159691 0.035576 0.643730 1.051265 -0.621751
   1.205093

```

3 Exercise 3.11(d)

3.1 运行方式

./prog3_11 下放有 makefile, 在该目录下执行 “**make**” 可以得到可执行程序; 之后执行 “**make run p=...**” 可以运行程序。其中 p 的值可以任选, 如 “**make run p=10**” (默认 p=4); 执行 “**make clean**” 可以删除生成的.o 文件和可执行程序;

为了便于检验程序正确性, 在 debug 模式下可以输出每个进程产生的数组元素。即执行 “**make debug**” 产生调试模式下的可执行程序, 再执行 “**make run p=...**” 可以观察输出。

运行程序后, 每个进程会随机生成 10 个元素的数组, 数值范围 $[0, 10)$ ¹, 最终在主进程中输出总共 $10p$ 个元素数组的前缀和。

3.2 代码思路

首先在每个进程内计算数组的前缀和, 保存在 local_prefix_sum[] 中。

```
1 //compute local prefix sum of local_a
2 local_prefix_sum[0] = local_a[0];
3 for(i = 1; i < LOCAL_N; i++){
4     local_prefix_sum[i] = local_prefix_sum[i-1] + local_a[i];
5 }
```

之后使用 MPI_Scan() 函数得到在该进程编号之前的所有进程数组的和 (pred)。

```
1 //Perform MPI_Scan() get pred_sum, (local sum == local_prefix_sum[
    LOCAL_N])
2 MPI_Scan(&local_prefix_sum[LOCAL_N-1], &pred, 1, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);
3 pred -= local_prefix_sum[LOCAL_N-1];
```

之后为每个进程内的 local_prefix_sum[] 都加上 pred, 此时的 local_prefix_sum[] 便是在全部进程的大数组中的前缀和。

```
1 //add pred sum
2 for(i = 0; i < LOCAL_N; i++){
3     local_prefix_sum[i] += pred;
4 }
```

最后调用 MPI_Gather() 收集到主进程输出。

3.3 测试结果

下列样例执行了 “make run p=4” 命令, 程序在每个线程中生成 10 个元素, 共 40 个元素。

在 debug 模式下可以看到, 程序正确地输出了前缀和。

¹进程内随机数的生成加了 pid 的偏移 srand(pid+time(NULL));

debug 模式

```
1 [2018011446@bootstraper prog3_11]$ make run
2 srun -n 4 -l ./prefixsum
3 0: Process 0, vec:
4 0: 4 7 0 8 3 1 0 9 8 6
5 1: Process 1, vec:
6 1: 5 1 7 4 3 1 3 4 0 3
7 2: Process 2, vec:
8 2: 6 9 8 7 3 0 2 9 4 0
9 3: Process 3, vec:
10 3: 7 6 2 4 5 1 3 7 7 9
11 0: Prefix sum:
12 0: 4 11 11 19 22 23 23 32 40 46 51 52 59 63 66 67 70 74 74 77 83 92 100
    107 110 110 112 121 125 125 132 138 140 144 149 150 153 160 167 176
```

release 模式

```
1 [2018011446@bootstraper prog3_11]$ make run p=4
2 srun -n 4 -l ./prefixsum
3 0: Prefix sum:
4 0: 7 12 20 20 24 32 40 42 49 54 56 56 60 63 63 66 71 77 78 80 82 82 87
    96 101 109 114 115 122 126 126 127 132 140 147 150 150 152 158 160
```

4 Exercise 3.1

4.1 运行方式

./prog3_1 下放有 makefile, 在该目录下执行 “**make**” 可以得到可执行程序; 之后执行 “**make run p=...**” 可以运行程序。其中 p 的值可以任选, 如 “**make run p=10**” (默认 $p=4$); 执行 “**make clean**” 可以删除生成的.o 文件和可执行程序;

运行程序后, 根据提示分别输入桶的个数 $bins$ 、最小值 min 和最大值 max , 数据个数 n 。注意, n 应能被 p 整除。

4.2 代码思路

代码完善了 Find_bins() 和 Which_bin() 函数。

Which_bin(): 考虑到桶的个数不是很多 (通常 < 1000), 所以采用线性查找其实可以很快的得到结果。将 $data$ 与桶的边界依次比较即可。如果桶的数量 > 10000 , 可以将线性查找改为二分查找, 以达到更快的搜索速度。

Which_bin()

```
1 int Which_bin(float data, float bin_maxes[], int bin_count, float
   min_meas) {
2     if(min_meas <= data && data <= bin_maxes[0]){
```

```

3     return 0;
4 } else {//bin_maxes[] bounds are [left, right);
5     int i;
6     for(i = 1; i < bin_count; i++){
7         if (bin_maxes[i-1] <= data && data < bin_maxes[i]){
8             return i;
9         }
10    }
11    return bin_count - 1;
12 }
13 } /* Which_bin */

```

Find_bins(): 对当前进程的每个元素 *data*, 找到 *data* 所属的桶之后, 将 *loc_bin_cts[]* 对应的桶编号位置 +1 即可。在本进程所有元素都统计完毕后, 使用 *MPI_Reduce()* 函数将所有的 *loc_bin_cts[]* 进行加总, 存入 *bin_counts[]* 中。

Find_bins()

```

1 void Find_bins(int bin_counts[], float local_data[], int loc_bin_cts[],
  int local_data_count, float bin_maxes[], int bin_count, float
  min_meas, MPI_Comm comm){
2     /* Use a for loop to find bins, the statement in the loop can be:
3     bin = Which_bin(local_data[i], bin_maxes, bin_count, min_meas);
4     Then, calculate the global sum using collective communication.
5     */
6     int i;
7     for(i = 0; i < local_data_count; i++){
8         loc_bin_cts[Which_bin(local_data[i], bin_maxes, bin_count,
9             min_meas)] ++;
10    }
11    MPI_Reduce(loc_bin_cts, bin_counts, bin_count, MPI_INT, MPI_SUM, 0,
        comm);
12 } /* Find_bins */

```

4.3 测试结果

下列样例给出了执行 “make run p=4” 且 $n = 100, bin = 5, min = 0, max = 100$ 的运行结果。

$p = 4, n = 100, bin = 5, min = 0, max = 100$

```

1 [2018011446@bootstraper prog3_1]$ make run p=4
2 srun -n 4 -l ./histodist
3 0: Enter the number of bins
4 5
5 0: Enter the minimum measurement
6 0

```

```

7 0: Enter the maximum measurement
8 100
9 0: Enter the number of data
10 100
11 0: 0.000-20.000: XXXXXXXXXXXXXXXX
12 0: 20.000-40.000: XXXXXXXXXXXXXXXX
13 0: 40.000-60.000: XXXXXXXXXXXXXXXX
14 0: 60.000-80.000: XXXXXXXXXXXXXXXX
15 0: 80.000-100.000: XXXXXXXXXXXXXXXX

```

5 Exercise 3.5

5.1 运行方式

./prog3_5 下放有 makefile, 在该目录下执行 “**make**” 可以得到可执行程序; 之后执行 “**make run p=... n=...**” 可以运行程序。其中 p 的值可以任选, 如 “**make run p=10 n=1000**” (默认 $p=4$, $n=12$); 执行 “**make clean**” 可以删除生成的.o 文件和可执行程序; 注意: n 应能被 p 整除。

运行程序后, 程序会随机生成一个 $n \times n$ 的 *double* 矩阵和 n 维随机向量, 分别采用按列分块的并行和串行策略计算 $y = Ax$, 统计并输出串行/并行运行时间、加速比、二范数误差等结果。

5.2 代码思路

run_col():

为了使得数据可以按列分块进行数据分发, 本函数创建了派生数据类型 `col_t`, 表示按列分块的矩阵。

将矩阵 *matrix* 按列分块为 p 个 $n \times \frac{n}{p}$ 的竖长形矩阵 $A_{i,(n \times local_n)}$, 使用 `MPI_Scatter()` 函数分配到每个进程。

将 *vector* 均分为 p 个 $\frac{n}{p}$ 维的向量 *loc_vec*, 使用 `MPI_Scatter()` 函数分配到每个进程; 之后在进程内计算结果 $loc_res = A_{i,(n \times local_n)} \cdot loc_vec$, 是一个 n 维向量。

最后使用 `MPI_Reduce()` 函数加总各个进程的结果即可。

创建派生数据类型

```

1 int local_n = n / comm_sz;
2 MPI_Datatype vec_t, col_t;
3 MPI_Type_vector(n, local_n, n, MPI_DOUBLE, &vec_t);
4 MPI_Type_create_resized(vec_t, 0, local_n * sizeof(double), &col_t);

```

数据分发与收集

```

1 //Scatter data
2 MPI_Scatter(matrix, 1, col_t, loc_A, n * local_n, MPI_DOUBLE, 0, comm);

```

```

3 MPI_Scatter(vector, local_n, MPI_DOUBLE, loc_vec, local_n, MPI_DOUBLE,
    0, comm);
4 //Reduce data
5 MPI_Reduce(loc_res, result, n, MPI_DOUBLE, MPI_SUM, 0, comm);

```

计算

```

1 for(int i = 0; i < n; i++){
2     loc_res[i] = 0.0;
3     for(int j = 0; j < local_n; j++){
4         loc_res[i] += loc_A[i * local_n + j] * loc_vec[j];
5     }
6 }

```

5.3 测试结果

下列样例运行 “make run p=4 n=100” 得到如下结果, 可以看到, 本例并行算法的二范数误差 $error < 1e^{-10}$, 达到了预期要求:

$p = 4, n = 100$

```

1 [2018011446@bootstraper prog3_5]$ make run p=4 n=100
2 srun -n 4 -l ./matvectmul 100
3 0: error(2 norm):                0.00000000000001238903
4 0: time (Serial)                  0.000023s
5 0: time (with MPI_Scatter/MPI_Reduce) 0.000660s
6 0: time (without MPI_Scatter/MPI_Reduce) 0.000005s
7 0: time (Scatter&Reduce)          0.000655s
8 0: Speedup ratio(with MPI_Scatter/MPI_Reduce) 0.0345
9 0: Speedup ratio(without MPI_Scatter/MPI_Reduce) 4.5651

```

5.4 性能评估

下表展示了不同进程数 p 和矩阵大小 n 的性能测试结果: 运行时间及加速比的统计表与统计图如下:

$n \backslash p$	1	2	4	8	10	16	20
2000	0.017	0.021	0.020	0.017	0.006	0.017	0.051
5000	0.090	0.115	0.117	0.099	0.094	#N/A	0.092
10000	0.348	0.443	0.461	0.476	0.467	0.451	0.049
16000	0.820	1.001	1.044	1.037	1.041	1.747	1.077
20000	1.389	1.778	1.841	1.849	1.813	3.960	2.171
30000	3.287	4.087	4.193	4.106	4.058	3.982	4.421

表 1: Total time (s)

$n \backslash p$	1	2	4	8	10	16	20
2000	0.005	0.003	0.001	0.001	0.0008	0.0003	0.0002
5000	0.028	0.014	0.007	0.003	0.003	#N/A	0.002
10000	0.102	0.051	0.028	0.015	0.012	0.008	0.024
16000	0.255	0.116	0.065	0.034	0.028	0.032	0.027
20000	0.415	0.206	0.117	0.091	0.051	0.071	0.050
30000	1.034	0.516	0.262	0.134	0.114	0.081	0.110

表 2: Compute time (s)

$n \backslash p$	1	2	4	8	10	16	20
2000	0.011	0.018	0.018	0.016	0.018	0.016	0.051
5000	0.063	0.101	0.11	0.095	0.09	#N/A	0.090
10000	0.245	0.392	0.433	0.461	0.454	0.443	0.471
16000	0.564	0.886	0.979	1.004	1.012	1.716	1.050
20000	0.974	1.572	1.724	1.758	1.762	3.889	2.121
30000	2.253	3.569	3.931	3.973	3.944	3.901	4.303

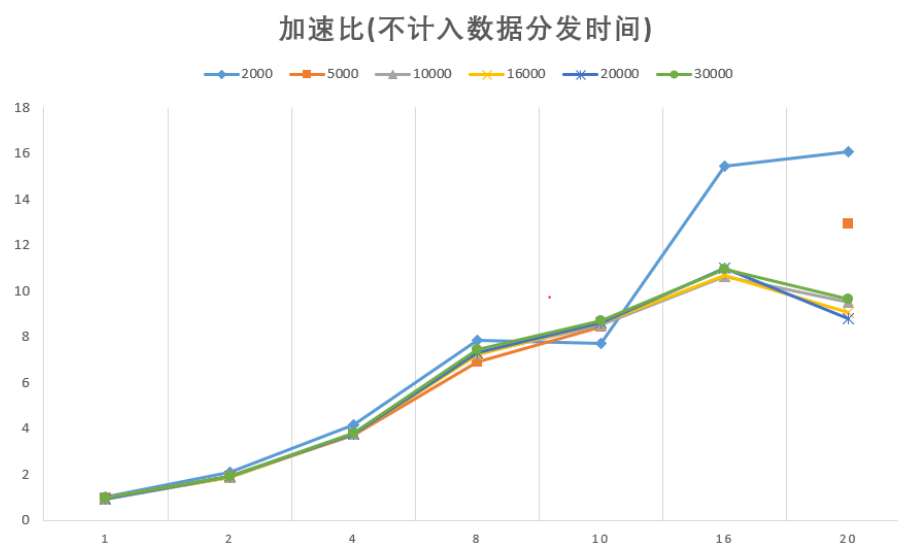
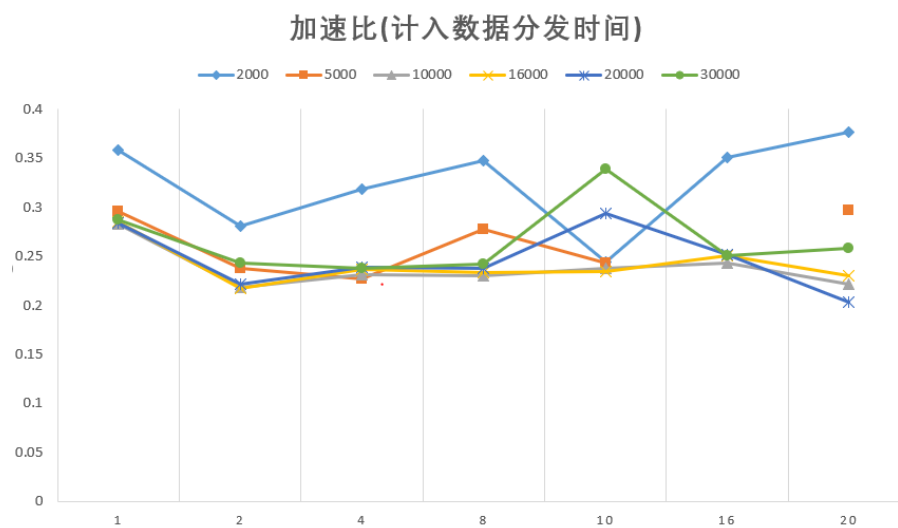
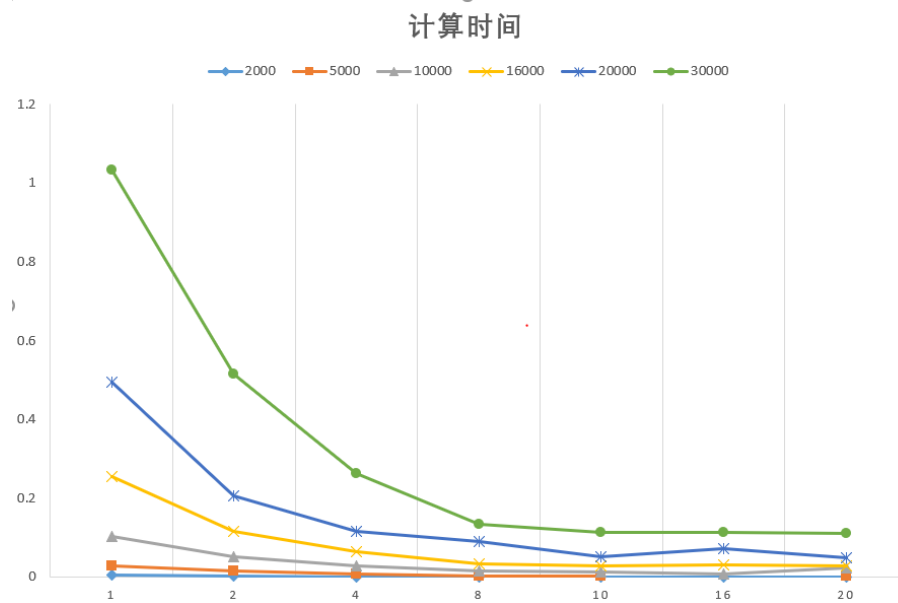
表 3: Communication time (s)

$n \backslash p$	1	2	4	8	10	16	20
2000	0.3581	0.2806	0.3183	0.3476	0.2446	0.3506	0.3767
5000	0.296	0.2372	0.2273	0.2776	0.2428	#N/A	0.297
10000	0.2823	0.2179	0.2317	0.2303	0.2375	0.2426	0.2214
16000	0.2848	0.2175	0.2365	0.2331	0.2346	0.2501	0.2301
20000	0.2837	0.2216	0.2392	0.238	0.2937	0.2519	0.2036
30000	0.2866	0.2432	0.2375	0.2422	0.3391	0.2510	0.2576

表 4: Speedup Ratio(考虑数据分发)

	1	2	4	8	10	16	20
2000	1.0415	2.0932	4.1795	7.8404	7.7315	15.4281	16.0935
5000	0.9676	1.9125	3.7297	6.9176	8.4419	#N/A	12.9149
10000	0.956	1.8704	3.7659	7.2619	8.5391	10.6438	9.5133
16000	0.9656	1.8797	3.7799	7.2342	8.6007	10.6960	9.0774
20000	0.953	1.9125	3.7644	7.3127	8.597	10.9949	8.7999
30000	0.9668	1.9237	3.8066	7.4312	8.6962	10.9626	9.6352

表 5: Speedup Ratio(不考虑数据分发)



6 Exercise 3.6

6.1 运行方式

./prog3_6 下放有 makefile, 在该目录下执行 “**make**” 可以得到可执行程序; 之后执行 “**make run p=... n=...**” 可以运行程序。其中 p 的值可以任选, 如 “**make run p=4 n=2000**” (默认 $p=16, n=100$); 执行 “**make clean**” 可以删除生成的.o 文件和可执行程序;

注意: p 应为完全平方数, 且 n 应能被 \sqrt{p} 整除。

运行程序后, 程序会随机生成一个 $n \times n$ 的 *double* 矩阵和 n 维随机向量, 分别采用的矩阵按块并行和串行策略计算 $y = Ax$, 统计并输出串行/并行运行时间、加速比、二范数误差等结果。

6.2 代码思路

`run_submat()`:

为了使得数据可以按分块矩阵进行数据分发, 本函数使用 `MPI_Cart_create()` 定义了新的具有笛卡尔拓扑结构的通信子 *comm*, 并使用 `MPI_Cart_sub()` 提取出某位置的通信子 *row_comm*, *col_comm*, *diag_comm*, 分别负责分块矩阵的行、列和对角线的通信。

定义新通信子

```
1 // Create a communicator of 2-dim Cart
2 MPI_Cart_create(MPI_COMM_WORLD, d, dims, periods, reorder, &comm);
3 // Build a communicator for each process row
4 free_coords[0] = 0, free_coords[1] = 1;
5 MPI_Cart_sub(comm, free_coords, &row_comm);
6 // Build a communicator for each process col
7 free_coords[0] = 1, free_coords[1] = 0;
8 MPI_Cart_sub(comm, free_coords, &col_comm);
9 // Get the group underlying comm
10 MPI_Comm_group(comm, &group);
11 // Create the new group
12 MPI_Group_incl(group, row_comm_sz, p_ranks, &diag_group);
13 // Create the communicator
14 MPI_Comm_create(comm, diag_group, &diag_comm);
```

之后, 我定义了新的派生数据类型 *submat_mpi_t*, 表示一个 $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ 的矩阵块。

定义派生数据类型

```
1 MPI_Datatype vect_mpi_t, submat_mpi_t;
2 MPI_Type_vector(loc_m, loc_n, n, MPI_DOUBLE, &vect_mpi_t);
3 MPI_Type_create_resized(vect_mpi_t, 0, loc_n * sizeof(double), &
    submat_mpi_t);
```

进行数据分发：使用 `MPI_Scatterv()` 函数将 *matrix* 中的元素按矩阵块分给各个进程。使用 `MPI_Scatter()` 函数将 *vector* 的数据分给对角进程 (diag) 中的 `loc_x[]` ($\frac{n}{\sqrt{p}}$ 维)，对角进程在列内使用 `MPI_Bcast()` 将分块向量分给同一列的进程的 `loc_x[]`。运算结束后，使用 `MPI_Reduce()` 函数将每行的结果加总得到 `loc_y[]`，保存在对角进程中。最后使用 `MPI_Gather()` 函数将对角进程的结果收集到主进程。

数据分发与收集

```

1 // Scatter data of matrix
2 MPI_Scatterv(matrix, counts, disp, submat_mpi_t, loc_A, loc_m * loc_n,
   MPI_DOUBLE, 0, comm);
3 // Scatter data of vector
4 MPI_Scatter(vector, loc_n, MPI_DOUBLE, loc_x, loc_n, MPI_DOUBLE, 0,
   diag_comm);
5 MPI_Bcast(loc_x, loc_n, MPI_DOUBLE, coords[1], col_comm);
6 // add up the partial sums in my process row and store the result on
   the diagonal
7 MPI_Reduce(sub_y, loc_y, loc_m, MPI_DOUBLE, MPI_SUM, coords[0],
   row_comm);
8 //gather data of results on the diagonal
9 MPI_Gather(loc_y, loc_n, MPI_DOUBLE, parallel_res, loc_n, MPI_DOUBLE,
   0, diag_comm);

```

计算

```

1 //matrix-vector multiply
2 for(int i = 0; i < loc_m; i++){
3   sub_y[i] = 0.0;
4   for(int j = 0; j < loc_n; j++){
5     sub_y[i] += loc_A[i * loc_n + j] * loc_x[j];
6   }
7 }

```

6.3 测试结果

下列样例运行 “make run p=4 n=100” 得到如下结果，可以看到，本例并行算法的二范数误差 $error < 1e^{-10}$ ，达到了预期要求：

$p = 4, n = 100$

```

1 [2018011446@bootstraper prog3_6]$ make run p=4 n=100
2 srun -n 4 -l ./matvectmul 100
3 0: error(2 norm):                                0.00000000000000941541
4 0: time (Serial)                                0.000021s
5 0: time (with MPI_Scatter/MPI_Reduce)           0.000435s
6 0: time (without MPI_Scatter/MPI_Reduce)         0.000006s
7 0: time (Scatter&Reduce)                        0.000429s
8 0: Speedup ratio(with MPI_Scatter/MPI_Reduce) 0.0487

```

6.4 性能评估

下表展示了不同进程数 p 和矩阵大小 n 的性能测试结果：
运行时间及加速比的统计表与统计图如下：

$n \setminus p$	1	4	16	25
2000	0.005/1.0439	0.002/2.3784	0.013/0.4618	0.168/0.0268
8000	0.073/0.9644	0.039/1.7993	0.190/0.3601	1.274/0.0311
12000	0.164/0.9612	0.068/1.7961	0.437/0.3506	2.234/0.0314
16000	0.264/0.9616	0.126/1.8334	0.768/0.3571	5.006/0.0315
24000	0.662/0.9636	0.188/2.2858	2.548/0.3496	8.891/0.0281
36000	1.485/0.9649	#N/A	#N/A	#N/A

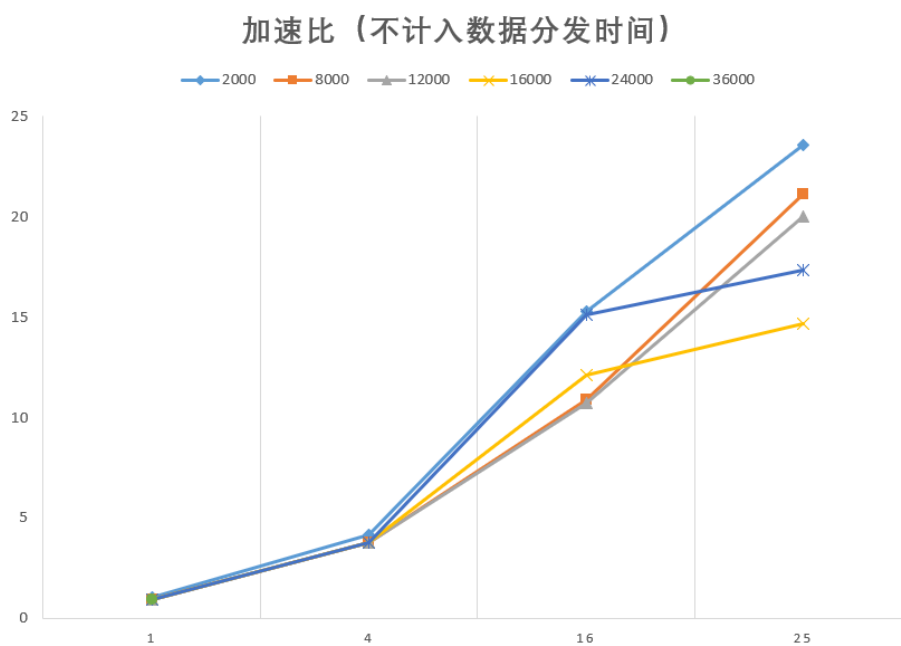
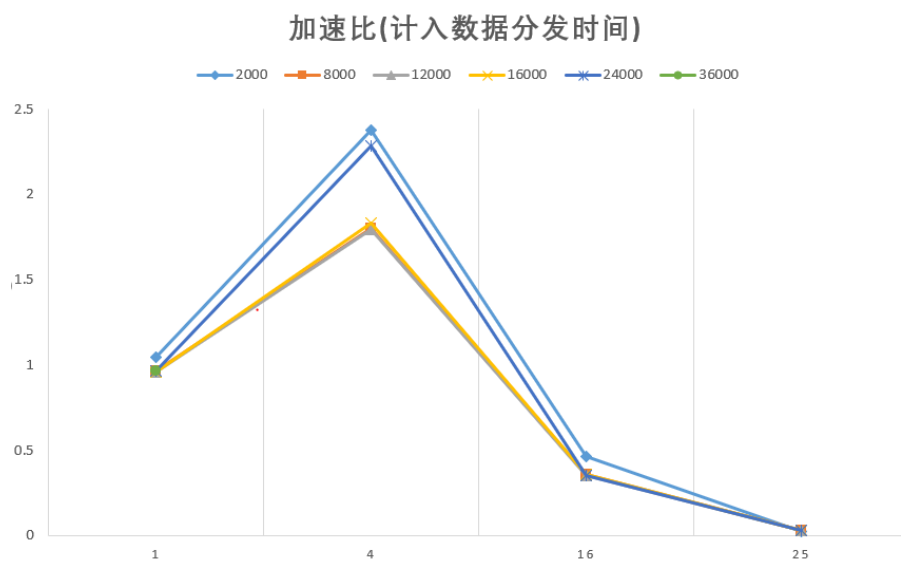
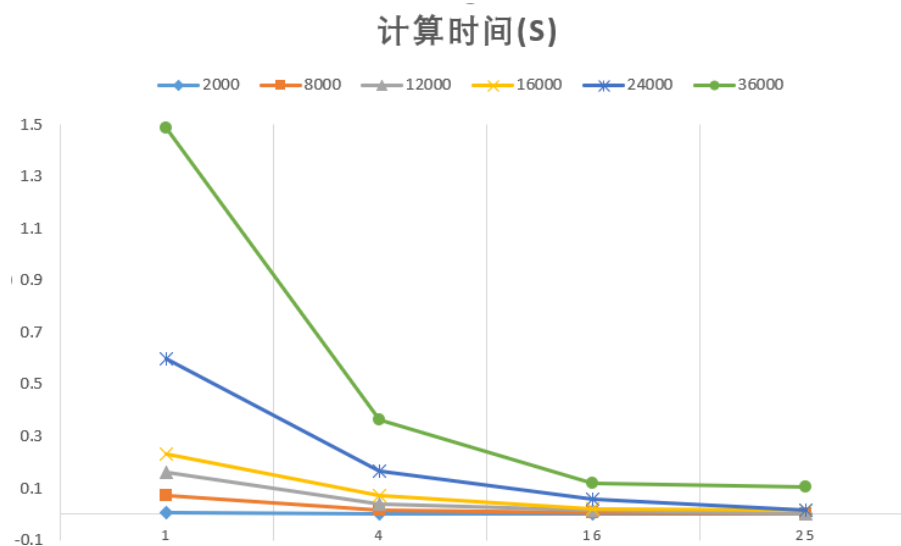
表 6: Total time(s) & Speedup Ratio(考虑数据分发)

$n \setminus p$	1	4	16	25
2000	0.005/1.0619	0.001/4.1508	0.0004/15.2852	0.0002/23.5762
8000	0.072/0.9665	0.018/3.7956	0.006/10.8752	0.001/21.1074
12000	0.163/0.9625	0.041/3.7985	0.014/10.7188	0.003/19.9999
16000	0.233/0.963	0.072/3.7988	0.022/12.1364	0.014/14.6549
24000	0.661/0.9641	0.315/3.7989	0.056/5.1244	0.015/17.3379
36000	1.484/0.9656	#N/A	#N/A	#N/A

表 7: Compute time(s) & Speedup Ratio(不考虑数据分发)

$n \setminus p$	1	4	16	25
2000	0.0001	0.001	0.128	0.168
8000	0.0001	0.201	0.183	1.272
12000	0.0002	0.028	0.423	2.230
16000	0.0002	0.054	0.745	4.992
24000	0.003	0.075	2.491	8.877
36000	0.001	#N/A	#N/A	#N/A

表 8: Communication time(s)



高性能计算导论:hw4

刘泓尊 2018011446 计 84

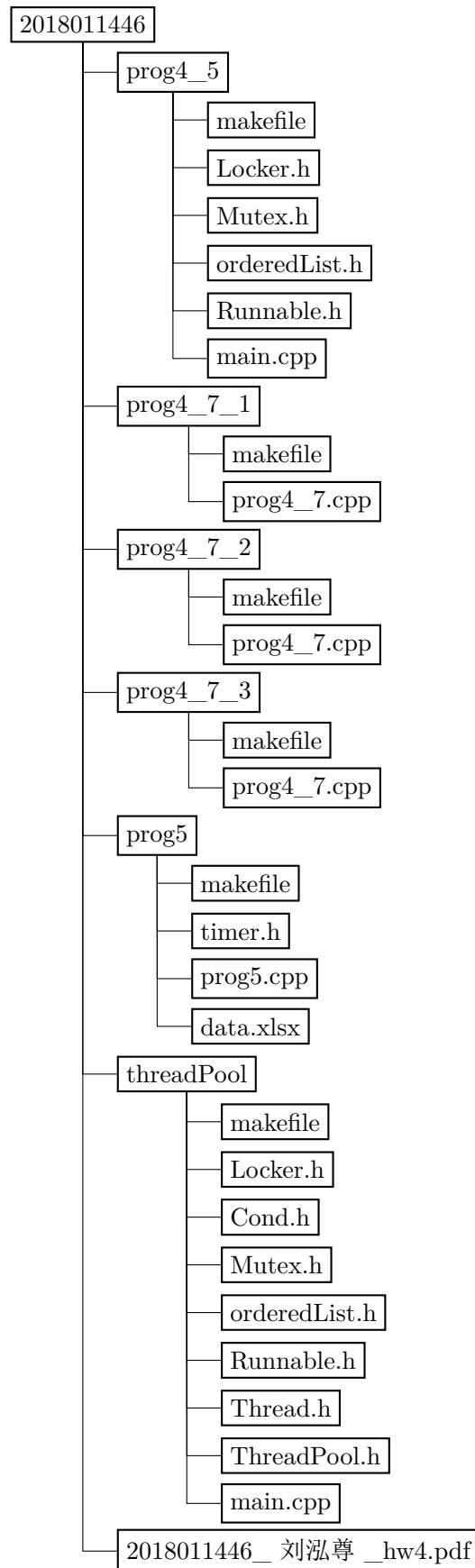
2020 年 4 月 10 日

目录

1 File Structure	2
2 Exercise 4.7	3
2.1 一个生产者 + 一个消费者	3
2.2 偶数生产者 + 奇数消费者	4
2.3 每个线程既是生产者又是消费者	5
3 Exercise 4.11	7
3.1 两个 Delete 同时执行	7
3.2 一个 Insert 和一个 Delete 同时执行	7
3.3 一个 Member 和一个 Delete 同时执行	8
3.4 两个 Insert 同时执行	8
3.5 一个 Insert 和一个 Member 同时执行	8
4 Exercise 4.12	8
5 Programming Assignment 4.5	9
5.1 线程安全的有序链表类	9
5.2 线程池的实现	10
5.3 实验结果	12
6 Pthread Programming 5	13
6.1 派生线程计算 fib(n-1)	13
6.1.1 普通递归 (并行)	13
6.1.2 记忆化递归 (并行)	14
6.1.3 串行部分的优化	15
6.2 运行结果	16
A 对线程 (Thread) 和线程池 (ThreadPool) 的封装	18
A.1 对互斥量 (Mutex) 与信号量 (Cond) 以及 Locker 的封装	18
A.2 对 Thread 的封装	19
A.3 对线程池 (ThreadPool) 的封装	21
A.4 重写 Programming Assignment 4.5	22

本次作业已上传至集群./2018011446/hw4 文件夹下。

1 File Structure



2 Exercise 4.7

2.1 一个生产者 + 一个消费者

因为只有两个线程，且每个线程责任清晰，所以我将生产者和消费者分别放在一个函数中，让结构更清晰。

生产者首先获得互斥量，然后发送信息到缓冲区，随后将标志位 (ok) 置为 1; 消费者采用忙等待 (busy-waiting) 机制，检测到标志位为 1 后，读取消息缓冲区中的内容，此过程中也采用互斥锁。(实际上，本例中“消费者”加锁的意义不大，除非编译器或 CPU 对指令进行了重排，也就是“内存序”改变的问题。)

此部分代码在 ./prog4_7_1 下

共享变量

```
1 const int MAX_LENGTH = 100;
2 pthread_mutex_t mu; //互斥量
3 bool ok; //标志位
4 char msg[MAX_LENGTH]; //消息缓冲区
```

Consumer

```
1 //消费者
2 void* consumer_func(void* rank){
3     long my_rank = (long)rank;
4     while(1){
5         pthread_mutex_lock(&mu); //加锁
6         if(ok){
7             printf("message: %s\n", msg);
8             pthread_mutex_unlock(&mu);
9             break;
10        }
11        pthread_mutex_unlock(&mu);
12    }
13    return nullptr;
14 }
```

Producer

```
1 //生产者
2 void* producer_func(void* rank){
3     long my_rank = (long)rank;
4     pthread_mutex_lock(&mu); //加锁
5     sprintf(msg, "Hello from thread %ld!", my_rank);
6     ok = 1;
7     pthread_mutex_unlock(&mu);
8     return nullptr;
```

```
9 }
```

main

```
1 pthread_t consumer, producer;
2 pthread_mutex_init(&mu, nullptr);
3
4 pthread_create(&consumer, nullptr, ::consumer_func, (void*)0);
5 pthread_create(&producer, nullptr, ::producer_func, (void*)1);
6 pthread_join(consumer, nullptr);
7 pthread_join(producer, nullptr);
8
9 pthread_mutex_destroy(&mu);
```

运行结果 执行 make run 之后可以运行程序，可以看到程序实现了“生产者-消费者”同步。

main

```
1 [2018011446@bootstraper prog4_7_1]$ make run
2 ./prod_cons1
3 message in thread 0: Hello from thread 1!
```

2.2 偶数生产者 + 奇数消费者

我将生产者和消费者写在同一函数中，便于区分奇数与偶数。共享变量设置与第一部分相同。

对于每一个线程，采用忙等待 (busy-waiting) 机制，首先获得互斥量：如果是奇数线程，检测标志位 (ok) 是否为 1, $ok == 1$ 则读取缓冲区中的内容，并将标志位 (ok) 置为 0，以便后续线程写入缓冲区；如果是偶数线程，在标志位为 0 的情况下，将消息写入缓冲区，并将标志位置为 1。消费者和生产者完成操作后均要释放锁。

代码放在 ./prog4_7_2 下

Producer&Consumer

```
1 void* run(void* rank){
2     long my_rank = (long)rank;
3     while(1){
4         pthread_mutex_lock(&mu); // 加锁
5         if(my_rank % 2 == 1){ // 奇数消费者
6             if(ok){
7                 printf("message received by %ld: %s\n", my_rank, msg);
8                 ok = 0;
9                 pthread_mutex_unlock(&mu);
10                break;
11            }
12        } else { // 偶数生产者
```



```

13         if(!ok){
14             sprintf(msg, "Hello from thread %ld", my_rank);
15             ok = 1;
16             pthread_mutex_unlock(&mu);
17             break;
18         }
19     }
20     pthread_mutex_unlock(&mu);
21 }
22 return nullptr;
23 }

```

main

```

1 pthread_mutex_init(&mu, nullptr);
2 pthread_t* threads = new pthread_t[threadnum];
3 for(size_t i = 0; i < threadnum; i++){
4     pthread_create(&threads[i], nullptr, ::run, (void*)i);
5 }
6 for(size_t i = 0; i < threadnum; i++){
7     pthread_join(threads[i], nullptr);
8 }
9 pthread_mutex_destroy(&mu);

```

运行结果 执行 make run n=10 后得到的结果如下 (n 是线程个数, n 必须为偶数!):

main

```

1 [2018011446@bootstraper prog4_7_2]$ make run n=10
2 ./prod_cons2 10
3 create 10 threads...
4 message received by 3: Hello from thread 0
5 message received by 1: Hello from thread 4
6 message received by 7: Hello from thread 2
7 message received by 9: Hello from thread 6
8 message received by 5: Hello from thread 8

```

2.3 每个线程既是生产者又是消费者

采用共享变量 *recv* 标记应接收消息的消费者。

依然采用忙等待机制, 每个线程初始时不设置角色。线程获得互斥锁后, 当发现标志位 (*ok*) 为 0 时, 成为生产者, 发送消息并将标志位置为 1, 指定消息接收方 (*recv*) 为 $(my_rank + 1) \% num_threads$ 。当有线程发现标志位为 1 且自己是接受方 (*recv*) 时, 成为消费者, 读取缓冲区并将标志位置 0。

代码放在./prog4_7_3 下

共享变量

```
1  const int MAX_LENGTH = 100;
2  bool ok;//标志位
3  char msg[MAX_LENGTH];//消息缓冲区
4  pthread_mutex_t mu;//互斥量
5  int recv;//下一个消费者的编号
6  int num;//线程个数
```

Producer&Consumer

```
1  void* run(void* rank){
2      long my_rank = (long)rank;
3      bool is_recv = false, is_send = false;
4      while(!is_send || !is_recv){
5          pthread_mutex_lock(&mu);
6          if(ok){
7              if(!is_recv && my_rank == recv){
8                  printf("receiver %ld: %s\n", my_rank, msg);
9                  ok = false;
10                 is_recv = true;
11             }
12             } else if (!is_send){
13                 sprintf(msg, "Hello from thread %ld", my_rank);
14                 ok = true;
15                 is_send = true;
16                 recv = (my_rank + 1) % num;
17             }
18             pthread_mutex_unlock(&mu);
19         }
20         return nullptr;
21     }
```

main

```
1  pthread_mutex_init(&mu, nullptr);
2  pthread_t* threads = new pthread_t[num];
3  for(size_t i = 0; i < num; i++){
4      pthread_create(&threads[i], nullptr, ::run, (void*)i);
5  }
6  for(size_t i = 0; i < num; i++){
7      pthread_join(threads[i], nullptr);
8  }
9  pthread_mutex_destroy(&mu);
```

运行结果 执行 make run n=10 的结果如下 (此题 n 任意):

```
main
1 [2018011446@bootstraper prog4_7_3]$ make run
2 ./prod_cons3 10
3 created 10 threads
4 receiver 1: Hello from thread 0
5 receiver 2: Hello from thread 1
6 receiver 9: Hello from thread 8
7 receiver 5: Hello from thread 4
8 receiver 8: Hello from thread 7
9 receiver 7: Hello from thread 6
10 receiver 0: Hello from thread 9
11 receiver 4: Hello from thread 3
12 receiver 3: Hello from thread 2
13 receiver 6: Hello from thread 5
```

3 Exercise 4.11

注: 下面所说的 *prev* 不代表是双向链表, 只是代表位置关系在 “前”

3.1 两个 Delete 同时执行

如果两个线程试图删除同一个节点, 执行顺序可能如下:

- Thread1 找到节点 N
- Thread2 找到节点 N
- Thread1 将 $N \rightarrow prev \rightarrow next$ 赋值为 $N \rightarrow next$
- Thread1 释放节点 N
- Thread2 将 $N \rightarrow prev \rightarrow next$ 赋值为 $N \rightarrow next$ 。此时访问了已经释放的内存, 发生段错误。

如果两个线程释放前后相邻的节点, 可能的执行顺序如下:

- Thread1 找到节点 N1
- Thread2 找到节点 N2, 且 N2 紧邻在 N1 后面
- Thread1 将 $N1 \rightarrow prev \rightarrow next$ 赋值为 $N1 \rightarrow next$
- Thread2 将 $N2 \rightarrow prev \rightarrow next$ (即 $N1 \rightarrow next$) 赋值为 $N2 \rightarrow next$ 。

此时 $N1 \rightarrow prev$ 的后继为 N2, 而不是 $N2 \rightarrow next$ 。

- Thread1 删除节点 N1
- Thread2 删除节点 N2

此时链表发生断裂。

3.2 一个 Insert 和一个 Delete 同时执行

插入的线程恰好插入在待删除点的前面:

- Thread1 找到待删除节点 N1
- Thread2 创建新节点 N2, 找到插入位置, 在 N1 的前面

- Thread2 将 N2 的后继标记为 N1
- Thread1 删除 N1 节点

此时发生链表**断裂**, 节点 N2 之后的部分将丢失。

3.3 一个 Member 和一个 Delete 同时执行

Member 和 Delete 是同一节点时:

- Thread1 调用 Member 访问节点 N1, 发现 N1 存在
- Thread2 删除 N1
- Thread1 函数返回报告 N1 存在, 实际上却已经被删除.

Member 的节点在 Delete 之后:

- Thread1 查找 (Member) 节点 N1, 当前正在节点 N2(在 N1 之前)
- Thread2 删除节点 N2
- Thread1 调用 $cur = N1 \rightarrow next$, 访问已经释放的内存, 发生**段错误**。

3.4 两个 Insert 同时执行

两个线程 insert 在同一位置:

- Thread1 找到插入位置的前一个节点 N1
- Thread2 找到插入位置的前一个节点 N1
- Thread1 插入: $N1 \rightarrow next = newNode1$
- Thread2 插入: $N1 \rightarrow next = newNode2$

此时相当于只成功插入了一个节点, 而不是两个。

3.5 一个 Insert 和一个 Member 同时执行

Insert 线程插入节点恰好是 Member 查询的.

- Thread1 查找节点 N1, 未找到
- Thread2 插入节点 N1
- Thread1 调用返回, 报告 N1 不存在, 而实际上已经在链表中。

4 Exercise 4.12

不安全

- 两个线程删除同一节点时:

第一阶段: 由于读写锁可以同时读, 所以两个线程都将找到该节点.

第二节点: 线程 1 加“写锁”, 删除该节点, 释放锁; 之后线程 2 获得“写锁”再删除该节点。出现**重复释放**的问题。

- 两个线程同时插入时, 也会与进行上例相同的第一阶段, 因此在第二阶段可能在同一位置插入两个节点。

两种情况的**本质**都是: 在两次加锁的中间存在时间差, 这段时间内, 链表的状态可能发生改变。因此不安全。

5 Programming Assignment 4.5

代码在./prog4_5 下

5.1 线程安全的有序链表类

为了便于管理锁的获取与释放，我简单封装了互斥量与信号量，并实现了 Locker 来自动管理锁的释放，见附录 A。之后我实现了线程安全的有序链表 orderedList，为了简洁性，内部数据结构使用 STL，代码如下：

orderedList.h

```
1  template <class T>
2  class orderedList{
3      list<T> alist;
4      ReadWriteMutex rwmu;
5  public:
6      void push(T t){
7          WriteLocker locker(&rwmu);//写锁
8          alist.insert(std::upper_bound(alist.begin(), alist.end(), t), t);
9      }
10     bool pop(T t){
11         WriteLocker locker(&rwmu);//写锁
12         auto iter = std::find(alist.begin(), alist.end(), t);
13         if(iter != alist.end()){
14             alist.erase(iter);
15             return true;
16         } else {
17             return false;
18         }
19     }
20     bool find(T t){
21         ReadLocker locker(&rwmu);//读锁
22         auto iter = std::find(alist.begin(), alist.end(), t);
23         if(iter != alist.end()){
24             return true;
25         } else {
26             return false;
27         }
28     }
29     string toString(){
30         string res = "head";
31         ReadLocker locker(&rwmu);//读锁
32         for(auto iter: alist)
33             res += "->" + std::to_string(iter);
34         res += "\n";
35         return res;
```

```
36     }
37 };
```

5.2 线程池的实现

线程池的实现主要是填充了题给的框架。先使用 `pthread_create()` 创建给定数量的线程。然后生成一定数量的任务，压入任务队列，同时用信号量唤醒线程。当任务产生完毕的时候，设置标志位 `finished = true`，之后进行广播唤醒所有线程。

main.cpp: main()

```
1  //Create Thread Pool
2  pthread_t* thread_handles = new pthread_t[thread_num];
3
4  /* Initialize mutexes and conditional variables */
5  pthread_mutex_init(&mu, nullptr);
6  pthread_cond_init(&cond, nullptr);
7  finished = false;
8
9  /* Start threads */
10 for(int i = 0; i < thread_num; i++){
11     pthread_create(&thread_handles[i], nullptr, ::runtask, nullptr);
12 }
13
14 /* Generate Tasks */
15 for(int i = 0; i < task_num; i++){
16     pthread_mutex_lock(&mu);
17     taskQueue.push( Task( rand() % total, rand() % task_num) );//加入随机任务
18     pthread_cond_signal(&cond);
19     pthread_mutex_unlock(&mu);
20     serialRun(t);//串行程序进行同样的操作
21 }
22
23 /* Wait fot threads to compelete */
24 finished = true;
25 pthread_cond_broadcast(&cond);
26 for(int i = 0; i < thread_num; i++){
27     pthread_join(thread_handles[i], nullptr);
28 }
29
30 /* Destroy mutex and conditional variables */
31 pthread_cond_destroy(&cond);
32 pthread_mutex_destroy(&mu);
33 delete[] thread_handles;
34
35 /* check for parallel and serial */
```

```

36 std::cout << "\nparallel:\n" << mylist.toString();
37 std::cout << "\nserial:\n" << serialList.toString();

```

线程执行的函数 runTask(), 使用类似忙等待的机制, 一直循环直至有信号唤醒该线程, 之后该线程执行随机操作的任务。具体代码如下:

```

runTask()
1 //线程函数
2 void* runtask(void* args){
3     long rank = long(arg);
4     while(1){
5         pthread_mutex_lock(&mu);
6         while(taskQueue.empty() && !finished){//线程等待被唤醒
7             while( pthread_cond_wait(&cond, &mu) != 0)
8                 ;
9         }
10        if(taskQueue.empty()){
11            pthread_mutex_unlock(&mu);
12            break;
13        }
14        Task t = taskQueue.front();
15        taskQueue.pop();
16        pthread_mutex_unlock(&mu);
17        run(t, rank);//执行对链表的随机操作
18    }
19    return nullptr;
20 }

```

为了减少耦合, 对链表的随机操作写在 run() 函数中, 比例为“插入: 删除: 查询: 打印 = 4:2:8:1”。为了便于显示多线程效果, 每次执行完该函数后 sleep 1s, 并在每次操作后输出线程编号和任务执行结果。代码如下:

```

run()
1 //定义操作比例
2 constexpr int findRate = 8;
3 constexpr int deleteRate = 2;
4 constexpr int insertRate = 4;
5 constexpr int printRate = 1;
6 constexpr int total = findRate + deleteRate + insertRate + printRate;
7 //并行操作, 随机调用
8 void run(const Task& task, const long& rank){
9     if (task.op < findRate) {find for 8/15 [0, 7)}
10        if(mylist.find(task.value)){
11            printf("threadid %lld, find value %d success\n", rank, task.
                value);

```

```

12     } else {
13         printf("threadid %lld, find value %d failed\n", rank, task.value
14             );
15     }
16 } else if (findRate <= task.op && task.op < findRate + deleteRate) {//
17     delete for 2/15 [7, 9)
18     if(mylist.pop(task.value)){
19         printf("threadid %lld, delete value %d success\n", rank, task.
20             value);
21     } else {
22         printf("threadid %lld, delete value %d failed\n", rank, task.
23             value);
24     }
25 } else if (findRate + deleteRate <= task.op && task.op < total -
26     printRate){//insert for 4/15 [9, 14)
27     mylist.push(task.value);
28     printf("threadid %lld, inserted value %d\n", rank, task.value);
29 } else { // print for 1/15 [14, 15)
30     printf("threadid %lld, print: %s", rank, mylist.toString().c_str());
31 }
32 sleep(1);//sleep for 1 sec
33 }

```

5.3 实验结果

执行“make run t=4 n=20”后的输出结果如下，可以看到任务可以分配给 2 个线程。由于每次执行后 sleep(1)，加上线程个数的限制，形成了周期性输出的表现，每隔约 1s 则较快地输出 2 个。从多次测试的结果来看，并程序是正确的。

输出结果

```

1 [2018011446@bootstraper prog4_5]$ make run t=4 n=20
2 ./main 4 20
3 threadid 3, inserted value 0
4 threadid 2, inserted value 10
5 threadid 1, delete value 10 failed
6 threadid 0, inserted value 1
7 threadid 3, inserted value 9
8 threadid 2, inserted value 14
9 threadid 1, inserted value 2
10 threadid 0, inserted value 9
11 threadid 3, find value 16 failed
12 head->0->1->2->9->9->10->14
13 threadid 1, inserted value 0
14 threadid 0, find value 1 success
15 threadid 3, find value 10 success

```



```

16 threadid 2, inserted value 1
17 threadid 1, inserted value 8
18 threadid 0, delete value 12 failed
19 threadid 3, inserted value 13
20 threadid 2, find value 11 failed
21 threadid 0, find value 15 failed
22 threadid 1, delete value 13 success

```

在运行结束后，会输出并行结果与串行结果得到的链表，经过多次验证，并程序的实现是正确的。

正确性检验示例

```

1 parallel:
2 head->0->0->1->1->2->8->9->9->10->14
3 serial:
4 head->0->0->1->1->2->8->9->9->10->14

```

为了后续编程方便，我还对线程 (Thread) 和线程池 (ThreadPool) 进行了封装，并重写了本题，极大地做到了代码的简化，具体内容请看[附录 A](#)。

6 Pthread Programming 5

代码在 ./prog5 下

6.1 派生线程计算 fib(n-1)

我实现了两个版本的 fib 函数，其中一个只是普通递归求值，复杂度是 $O(2^n)$ ；第二个版本使用记忆化方法，将复杂度为 $O(n)$ 。

由于创建进程会占用大量时间，在串行递归时，我设置在 $fib(n), n < 15$ 的时候使用串行函数计算、返回，而不再创建新线程。实际上，在求解较大 n 时 (比如 $n > 40$)，由于搜索树过深且计算节点的创建是指数级增长的，所以到达 $n = 15$ 的时候也不会再创建新线程。因此，此做法可以认为是合理的。

此外，我将 fib(n) 的最大 n 设为 60，因为 $n > 60$ 的串行版本将消耗大量的计算时间。

6.1.1 普通递归 (并行)

基于串行版本的递归做了修改。核心是设置一个共享变量 cur_thread_num 计算当前线程数，当当前线程数 < 最大线程数的时候，创建一个新线程计算 fib(n-2)，本线程计算 fib(n-1)。这里考虑到了均衡，因为 fib(n-1) 的计算时间较大，所以将计算时间短的 fib(n-2) 交给新线程去做，以平衡创建线程的开销。

fib_para(n)

```

1 void* fib_para(void* n){

```

```

2     long my_n = (long)n;
3     if(my_n < 15){
4         return (void*)fib_serial(my_n);
5     }
6 #   ifdef DEBUG
7     printf("thread %d: fib(%ld)\n", (unsigned int)pthread_self(), my_n);
8 #   endif
9     bool can_create = false;
10    pthread_mutex_lock(&mu);
11    if(++cur_thread_num < total_thread){//判断是否可以派生新线程
12        can_create = true;
13    }
14    pthread_mutex_unlock(&mu);
15    if(can_create){
16        pthread_t new_thread;
17        long firstres, secondres;
18        //create a new thread to calculate fib(n-2)
19        pthread_create(&new_thread, nullptr, fib_para, (void*)(my_n-2));
20        //this thread to calculate fib(n-1)
21        firstres = (long)fib_para((void*)(my_n-1));
22        pthread_join(new_thread, (void**)&secondres);
23        pthread_mutex_lock(&mu);
24        cur_thread_num--;//修改共享变量
25        pthread_mutex_unlock(&mu);
26        return (void*)(firstres + secondres);
27    } else {
28        return (void*)((long)fib_para( (void*)(my_n-1) ) + (long)fib_para(
                (void*)(my_n-2) ) );
29    }
30 }

```

6.1.2 记忆化递归 (并行)

针对上述并行递归版本做了记忆化，改动的部分并不多，只是加了一个记忆数组 `dp[]`。

```

                                fib_para_fast(n)
1  long fib[50] = {0};//共享变量，记录dp状态
2  bool finished[50] = {0};//记录是否被计算
3  void* fib_para_dp(void* n){
4      long my_n = (long)n;
5      if(finished[my_n]){
6          return fib[my_n];
7      }
8      if(my_n < 15){
9          return (void*)fib_serial(my_n);

```

```

10     }
11 #   ifdef DEBUG
12     printf("thread %d: fib(%ld)\n", (unsigned int)pthread_self(), my_n);
13 #   endif
14     bool can_create = false;
15     pthread_mutex_lock(&mu);
16     if(++cur_thread_num < total_thread){
17         can_create = true;
18     }
19     pthread_mutex_unlock(&mu);
20     if(can_create){
21         pthread_t new_thread;
22         long firstres, secondres;
23         //create a new thread to calculate fib(n-2)
24         pthread_create(&new_thread, nullptr, fib_para_dp, (void*)(my_n-2));
25         //this thread to calculate fib(n-1)
26         firstres = (long)fib_para_dp((void*)(my_n-1));
27         pthread_join(new_thread, (void**)&secondres);
28         pthread_mutex_lock(&mu);
29         cur_thread_num--;
30         fib[my_n] = firstres + secondres; //共享变量dp[]
31         finished[my_n] = true; //共享变量finished[]
32         pthread_mutex_unlock(&mu);
33         return (void*)(fib[my_n]);
34     } else {
35         long result = (long)fib_para_dp( (void*)(my_n-1) ) + (long)
            fib_para_dp( (void*)(my_n-2) );
36         pthread_mutex_lock(&mu);
37         fib[my_n] = result; //共享变量dp[]
38         finished[my_n] = true; //共享变量finished[]
39         pthread_mutex_unlock(&mu);
40         return (void*)( result );
41     }
42 }

```

6.1.3 串行部分的优化

实际上，计算 $\text{fib}(n)$ 可以不采用递归函数，使用循环迭代完全可以实现 $O(n)$ 的复杂度，并且只有 $O(1)$ 空间。此策略可以极大减少创建线程和递归调用的开销。其串行版本如下：

$\text{fib_serial_fast}(n)$

```

1 long fib_serial_fast(int n){
2     long a = 0, b = 1, c;
3     for(int i = 1; i < n; i++){
4         c = b; b = a + b; a = c;

```

```
5     }  
6     return b;  
7 }
```

将此优化加入到并行版本的 fib(n) 求解中，可以实现较大改进，性能测试见 6.2 部分。

6.2 运行结果

执行 make run 命令后，依次输入线程个数和 n，之后分别计算串行版本和以上三个版本的计算时间. 示例如下：

运行示例

```
1 [2018011446@bootstraper prog2]$ make run  
2 ./prog2  
3 Enter thread_num:  
4 8  
5 Enter n of fib(n):  
6 46  
7 para_normal: fib(n) = 1836311903, time: 1.353691s  
8 para_fast:   fib(n) = 1836311903, time: 0.210368s  
9 para_dp:     fib(n) = 1836311903, time: 0.000023s  
10 serial:     fib(n) = 1836311903, time: 2.867669s
```

从多次输出结果来看，并行函数的正确性是得到验证的。

下面对不同 threadnum 和 n 值的性能进行统计:

p\n	10	20	25	30	35	40	45	50	55
串行	0.000001	0.000036	0.000339	0.003685	0.034168	0.22032	1.840282	19.100321	218.093399
1	0.000199	0.000251	0.00052	0.003439	0.032831	0.205708	1.653113	17.722985	196.093399
2	0.000202	0.000257	0.000536	0.003444	0.032849	0.205942	1.653773	17.792085	195.970721
4	0.000224	0.000373	0.000452	0.001664	0.014966	0.115644	0.778252	7.663938	85.838119
8	0.000221	0.002438	0.003865	0.002151	0.014333	0.109306	0.754843	8.905228	93.74916
16	0.000224	0.006023	0.007535	0.006285	0.015639	0.106378	0.802445	9.339412	99.025345
24	0.000222	0.007819	0.011001	0.012939	0.017517	0.103107	0.779098	8.518038	89.329015

表 1: 计算时间/s(普通递归并行版本)

p\n	10	20	25	30	35	40	45	50	55
串行	0.000001	0.000036	0.000339	0.003685	0.034168	0.22032	1.840282	19.100321	218.093399
1	0.00008	0.000087	0.000062	0.000257	0.001684	0.008121	0.08914	0.989676	11.026279
2	0.000066	0.000073	0.000072	0.000311	0.001682	0.008167	0.089213	0.989144	10.566794
4	0.000053	0.000072	0.000072	0.00026	0.001909	0.011863	0.089168	1.033208	10.468103
8	0.000053	0.0000054	0.000106	0.00023	0.00119	0.012483	0.15095	1.055574	10.605881
16	0.000071	0.001384	0.000116	0.000251	0.00194	0.012322	0.154355	1.051279	10.544585
24	0.000054	0.005715	0.000112	0.000249	0.001911	0.013263	0.150649	1.076399	10.509891

表 2: 计算时间/s(优化线程内计算复杂度的并行版本)

p\n	10	20	25	30	35	40	45	50	55
串行	0.000001	0.000036	0.000339	0.003685	0.034168	0.22032	1.840282	19.100321	218.093399
1	0.00007	0.000034	0.000045	0.000044	0.000052	0.000033	0.000028	0.000083	0.000062
2	0.000058	0.000066	0.000045	0.000084	0.000051	0.000048	0.000036	0.000068	0.000079
4	0.000033	0.000072	0.000061	0.000055	0.000054	0.000027	0.000071	0.000097	0.000079
8	0.00004	0.000062	0.000055	0.000047	0.000034	0.000045	0.000053	0.000073	0.000091
16	0.000035	0.000043	0.000056	0.00004	0.000060	0.000047	0.000061	0.000076	0.000071
24	0.000031	0.004317	0.000049	0.000038	0.000063	0.00005	0.000068	0.000088	0.000080

表 3: 计算时间/s(记忆化递归的并行版本)

从上表可以看到, 普通递归并行版本计算时间约为串行版本的一半; 采用记忆化的并行版本实现了极大的优化, 几乎为瞬间完成, 这也验证了 $O(n)$ 与 $O(2^n)$ 的巨大差距; 采用优化后的递归并行版本也实现了 10-20 倍的速度提升。与串行版本相比, 三个版本均实现了较高的效率, 对并行版本的一点点改进直至获得百倍的加速也是很有成就感的。值得注意的是, 在单线程情况下的计算时间居然比串行版本还要快, 推测是编译器对并行版本做了更多的递归优化, 减少了递归调用的开销。

A 对线程 (Thread) 和线程池 (ThreadPool) 的封装

代码在./threadPool 下。

为了方便创建线程，免去频繁书写 pthread 函数的繁琐，我实现了 Thread 类、ThreadPool 类，并对锁 (mutex, rwlock) 和信号量 (cond) 进行了简单的封装，在此基础上重写了 Programming Assignment 4.5。

A.1 对互斥量 (Mutex) 与信号量 (Cond) 以及 Locker 的封装

在写程序过程中，我发现“释放锁”的过程比较繁琐，在每一次 return 或 break 语句前都需要手动释放。因此我实现了 Locker 类，Locker 对象在构造时加锁，析构时解锁，这样就可以利用作用域来管理锁的释放。同时对 pthread 的 mutex, rwlock 和 cond 进行了封装。代码如下：

Mutex.h

```
1 class Mutex{//互斥量
2     pthread_mutex_t m_mutex;
3 public:
4     Mutex(){ pthread_mutex_init(&m_mutex, nullptr); }
5     virtual ~Mutex(){ pthread_mutex_destroy(&m_mutex); }
6     void lock(){ pthread_mutex_lock(&m_mutex); }
7     void unlock(){ pthread_mutex_unlock(&m_mutex); }
8 };
9
10 class ReadWriteMutex{//读写锁
11     pthread_rwlock_t m_rwlock;
12 public:
13     ReadWriteMutex(){ pthread_rwlock_init(&m_rwlock, nullptr); }
14     virtual ~ReadWriteMutex(){ pthread_rwlock_destroy(&m_rwlock); }
15     void readlock(){ pthread_rwlock_rdlock(&m_rwlock); }
16     void writelock(){ pthread_rwlock_wrlock(&m_rwlock); }
17     void unlock(){ pthread_rwlock_unlock(&m_rwlock); }
18 };
```

Locker.h

```
1 class MutexLocker{
2     Mutex* m_mutex;
3 public:
4     MutexLocker(Mutex *mu): m_mutex(mu){ m_mutex->lock(); }
5     virtual ~MutexLocker() {m_mutex->unlock(); }
6 };
7
8 class ReadLocker{
9     ReadWriteMutex* m_rmutex;
10 public:
11     ReadLocker(ReadWriteMutex *mu): m_rmutex(mu){ m_rmutex->readlock(); }
```

```

12     virtual ~ReadLocker() { m_rmutex->unlock(); }
13 };
14
15 class WriteLocker{
16     ReadWriteMutex* m_wmutex;
17 public:
18     WriteLocker(ReadWriteMutex *mu): m_wmutex(mu){ m_wmutex->writelock(); }
19     virtual ~WriteLocker() { m_wmutex->unlock(); }
20 };

```

Cond.h

```

1 class Cond{
2     pthread_cond_t m_cond;
3 public:
4     Cond(){ pthread_cond_init(&m_cond, nullptr); }
5     virtual ~Cond(){ pthread_cond_destroy(&m_cond); }
6     int wait(Mutex *mu){return pthread_cond_wait(&m_cond, &mu->getpmutex())
7         ;}
8     void notify_one(){ pthread_cond_signal(&m_cond); }
9     void notify_all(){ pthread_cond_broadcast(&m_cond); }
10 };

```

A.2 对 Thread 的封装

仿照 c++ 标准的的线程库，我的 Thread 类继承 Runnable. 创建线程时，只需要继承 Thread 类重写 run() 方法即可。run() 方法负责在新线程中执行，只有 run() 方法内的变量是线程独有的。具体代码请见./Threadpool

Runnable.h

```

1 //Runnable can be shared by multi-threads
2 class Runnable{
3     static int taskNumber;
4     int taskId;
5 public:
6     Runnable(){ taskId = taskNumber++; }
7     virtual void run() = 0;
8 };

```

Thread.h

```

1 //Thread can't be shared by multi-threads
2 class Thread: public Runnable{
3     pthread_t m_thread;
4     int m_state;//pthread_create()返回的线程状态
5     bool is_running;//标记线程是否正在执行

```

```

6      int thread_id;//线程id
7      void* m_args;//静态func()函数的参数
8      Runnable* runnable;//指向Runnable创建的线程,配合线程池使用
9      static int thread_tot;//线程总数
10     static void *func(void *args);//配合pthread_create()成为回调函数
11 public:
12     Thread(): m_state(0), m_args(nullptr), is_running(false), runnable(
        nullptr){
13         thread_id = thread_tot++;
14     }
15     Thread(Runnable* r): m_state(0), m_args(nullptr){
16         is_running = false;
17         thread_id = thread_tot++;
18         runnable = r;
19     }
20     virtual ~Thread(){
21         m_state = 0;
22         m_args = nullptr;
23     }
24     virtual void run(){
25         if(runnable){ runnable->run(); }
26     };
27     void start(){
28         m_state = pthread_create(&m_thread, nullptr, Thread::func, this);
29         is_running = true;
30     }
31     void join(){
32         if(is_running){
33             pthread_join(m_thread, nullptr);
34             is_running = false;
35         }
36     }
37     int threadid()const{ return thread_id; }
38 };
39 int Thread::thread_tot = 0;
40 void* Thread::func(void* args){
41     Thread* p = static_cast<Thread*>(args);
42     p->run();
43     return nullptr;
44 }

```


A.3 对线程池 (ThreadPool) 的封装

仿照 PA4.5 采用信号量实现线程调度的思路，我对线程池进行了封装。之后调用时，只需要继承 Runnable 类重写 run() 函数，之后调用 ThreadPool 类的 addTask() 函数压入任务队列即可。

ThreadPool.h

```
1 class ThreadPool{
2     list<Runnable*> taskq; //任务队列，为了方便管理，使用list而不是queue
3     unsigned thread_num; //线程个数
4     Mutex mu; //互斥量
5     Cond cond; //信号量
6     pthread_t* threads;
7     std::atomic_bool finished; //标记所有任务是否完成
8 public:
9     ThreadPool(unsigned _n): thread_num(_n), finished(false),
10        threads(new pthread_t[thread_num]){
11        for(size_t i = 0; i < thread_num; i++){ //一次性创建线程
12            pthread_create(&threads[i], nullptr, ThreadPool::task, this);
13        }
14    }
15    virtual ~ThreadPool(){
16        finished = true;
17        cond.notify_all(); //没有任务就广播唤醒所有线程
18        for(size_t i = 0; i < thread_num; i++){
19            pthread_join(threads[i], nullptr);
20        }
21        for(auto iter: taskq){ //销毁任务
22            delete iter;
23        }
24    }
25    void addTask(Runnable* t){ //将任务压入队列
26        MutexLocker locker(&mu);
27        taskq.push_back(t);
28        cond.notify_one(); //唤醒线程执行该任务
29    }
30    static void* task(void* args); //作为回调函数
31 };
32
33 void* ThreadPool::task(void* args){ //由子线程负责执行
34     ThreadPool *thp = static_cast<ThreadPool*>(args);
35     while(1){
36         Runnable* t;
37         { //条件等待，实现任务的分配调度
38             MutexLocker locker(&thp->mu);
39             while(thp->taskq.empty() && !thp->finished){
```

```

40         while( thp->cond.wait(&thp->mu) != 0 )
41             ;
42     }
43     if(thp->taskq.empty()){
44         break;
45     }
46     t = thp->taskq.front();
47     thp->taskq.pop_front();
48 }
49 t->run();
50 }
51 return nullptr;
52 }

```

A.4 重写 Programming Assignment 4.5

经过上述封装之后，我重新编写了 4.5 的代码，实现了代码耦合度的减少和逻辑的清晰。首先是由 listTest 类继承 Runnable 类重写 run() 方法，实现对 4.5 实现的有序链表的随机操作。

listTest

```

1  orderedList<int> mylist;//共享变量
2
3  class listTest: public Runnable{
4      int op, value;
5  public:
6      listTest() = default;
7      listTest(int _op, int _val): op(_op), value(_val){}
8      void run(){
9          if (op < findbound) {//find for 80%
10             if(mylist.find(value)){
11                 printf("threadid %u, find value %d success\n", (unsigned int)
12                     pthread_self(), value);
13             } else {
14                 printf("threadid %u, find value %d failed\n", (unsigned int)
15                     pthread_self(), value);
16             }
17         } else if (findbound <= op && op < insertbound){//delete for 10%
18             if(mylist.pop(value)){
19                 printf("threadid %u, delete value %d success\n", (unsigned
20                     int)pthread_self(), value);
21             } else {
22                 printf("threadid %u, delete value %d failed\n", (unsigned
23                     int)pthread_self(), value);
24             }
25         } else {//insert for 10%

```

```

22         mylist.push(value);
23         printf("threadid %u, inserted value %d\n", (unsigned int)
                pthread_self(), value);
24     }
25 #     ifdef DEBUG
26         std::cout << mylist.toString();
27 #     endif
28     }
29 };

```

其次在主线程中开辟线程池，不断创建任务提交给线程池，由线程池实现调度。

main.cpp

```

1  ThreadPool pool(thread_num); //开辟线程池
2  for(size_t i = 0; i < task_num; i++){ //提交任务
3      pool.addTask( new listTest( rand() % 100, rand() % 500 ) );
4  }

```

可以看到，通过对 pthread 库进行良好的封装，可以极大方便后续编程任务的实现，降低了开发难度。

高性能计算导论: hw5

刘泓尊 2018011446 计 84

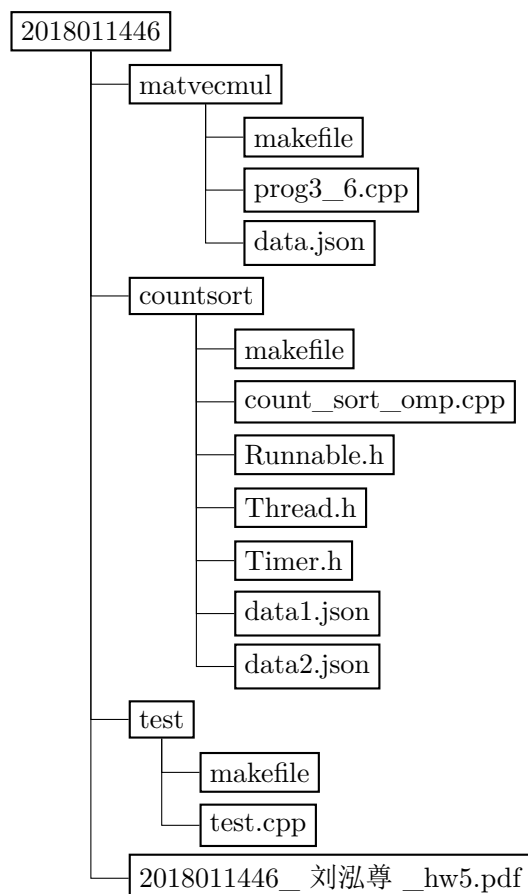
2020 年 4 月 19 日

目录

1 File Structure	2
2 Exercise 5.4	2
3 Exercise 5.5	2
4 Programming Assignment 1	2
4.1 运行方式	2
4.2 测试结果	3
4.3 和作业 3 中的比较	5
5 Programming Assignment 2	6
5.1 运行方式	6
5.2 哪些变量是私有的, 哪些变量是共享的	6
5.3 外循环是否存在循环依赖	6
5.4 是否能并行化对 memcpy 的调用	7
5.5 程序说明	7
5.5.1 串行 $O(n^2)$ 计数排序的优化	7
5.5.2 $O(n^2)$ 版本的并行化	8
5.5.3 $O(n + k)$ 版本的并行化	8
5.5.4 最后 4 路归并的计数排序	9
5.6 性能测试	10
5.7 不同调度策略和块大小的对比	13

本次作业已上传至集群./2018011446/hw5 文件夹下。

1 File Structure



2 Exercise 5.4

&&	1(true)
	0(false)
&	111...111
	0
^	0

3 Exercise 5.5

- a 计算完 4 次加法后，寄存器得到结果 1008，四舍五入后存入内存为 101×10^1 , 输出 **1010.0**
- b 线程 0 得到结果 4.0，线程 2 寄存器得到结果 1003，存入内存后为 100×10^1 , 规约后寄存器得到 1004，再次四舍五入存入内存为 100×10^1 , 输出 **1000.0**

4 Programming Assignment 1

4.1 运行方式

./matvectmul 文件夹下运行 “make run [p=...] [n=...] [t=..]” 可以运行程序，并对不同的进程数 p，线程数 t 和数据规模 n 进行设置。程序会输出二范数误差，总时间、通信时间、计算矩阵乘法时间和计算二范数时间，并给出相应的加速比和并行效率。正常的运行结果如下：

运行结果

```
1  srun -n 25 -l matvectmul 16000 4
2  00: error(2 norm):                0.0000000002357028461
3  00: time (Serial):                 0.281075s
4  00: time (Total: multiply+communication): 8.792906s
5  00: time (Matrix-vector Multiply): 0.014261s
6  00: time (Scatter&Reduce):         8.778645s
7  00: time (Calculate 2-norm):       0.000142s
8  00: Speedup ratio(with MPI_Scatter/MPI_Reduce): 0.0320
9  00: Speedup ratio(without MPI_Scatter/MPI_Reduce): 19.7092
10 00: Efficiency(with MPI_Scatter/MPI_Reduce): 0.0013
11 00: Efficiency(without MPI_Scatter/MPI_Reduce): 0.7884
```

4.2 测试结果

下一页展示了不同 n, p, th 值的运行时间 (总时间、通信时间、计算时间)、加速比 (总时间、计算时间)、并行效率 (总时间、计算时间) 统计图如下。(因为数据过多, 我在报告中只展示折线图, 您可以在 data.json 中查看实际的统计数据, 其中数据是三个维度的字典, 维度信息依次为: 进程数 p , 线程数 t , 数据规模 n 。($p \geq 25$ 时会跨机器运行, 运行时间大大增加, 与较小进程数不具备可比性。))

由于 MPI 通信部分无法并行, 所以不同线程数的通信时间是相同的, 这一点从统计图中也可以看出来。

在只考虑计算时间的并行效率上, 在进程数固定时, 线程数增多则效率下降, 这符合我们的预期。进程数增多也会导致并行效率的下降。在总并行效率的表现上, 进程内线程数的增多并不能显著增加并行效率, 因为 MPI 的通信时间占了 95% 以上的比重。

从图中可以看到, 提高此程序效率的关键在于 MPI 通信, 而不是计算部分。对于 MPI 通信时间的改进, 我也进行了多种尝试, 我发现通信的 90% 都用在 matrix 的 Scatter 上, 因为 Scatter 函数是线性进行数据分配的, 而且矩阵规模巨大, 所以 Scatter 的耗时很长。我将 Scatter 替换为了基于 MPI_Isend() 和 MPI_Irecv() 的点到点非阻塞通信, 通过“非阻塞通信 + 并行分发”的方法, 我将数据分发时间缩短到了原来的一半! 但是对于 $n > 10000$ 的规模, 点到点通信总是会引起程序崩溃。保险起见, 我最终还是使用了 Scatter。但这一尝试至少为我们提供了一种缩短通信时间的思路! 我也期待助教老师能为我提供更多的解决方案!

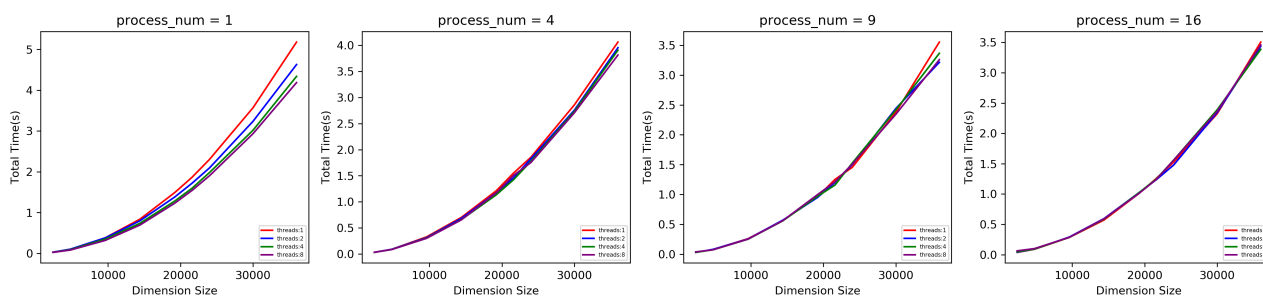


图 1: Total Time(s)

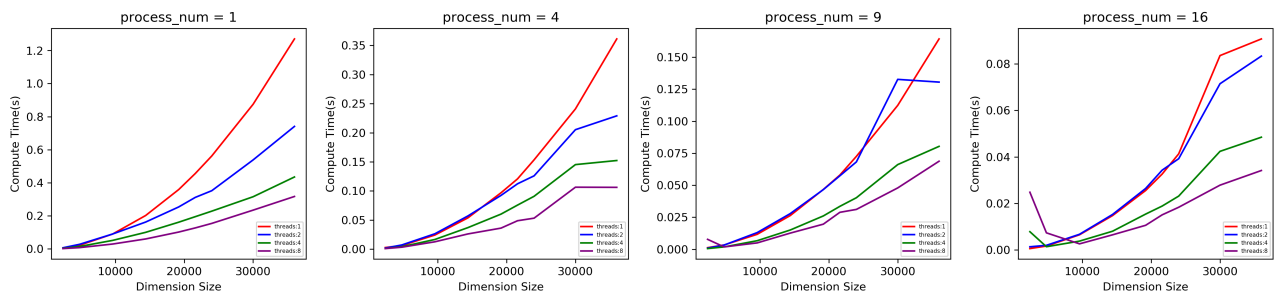


图 2: Compute Time(s)

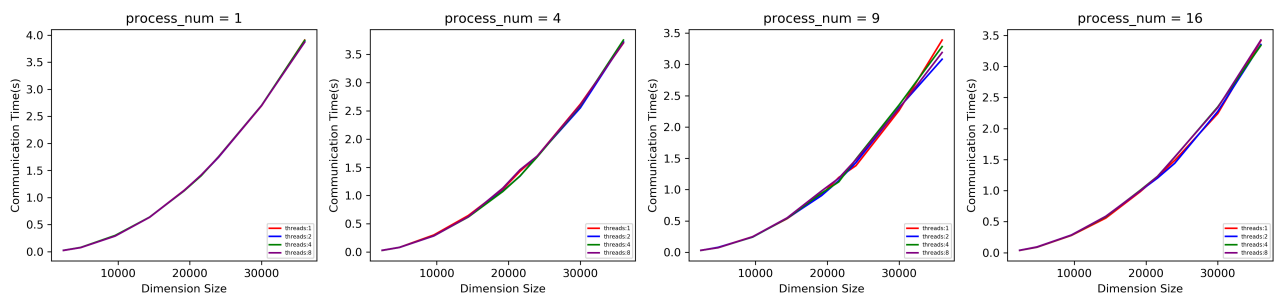


图 3: Communication Time(s)

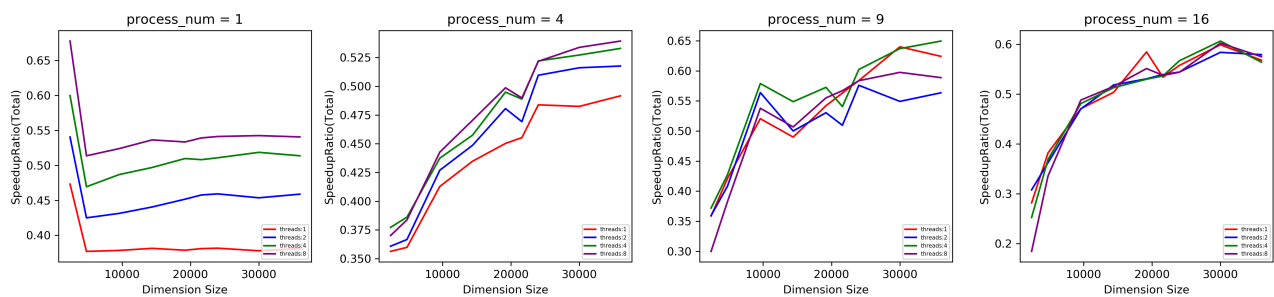


图 4: SpeedupRatio(Total)

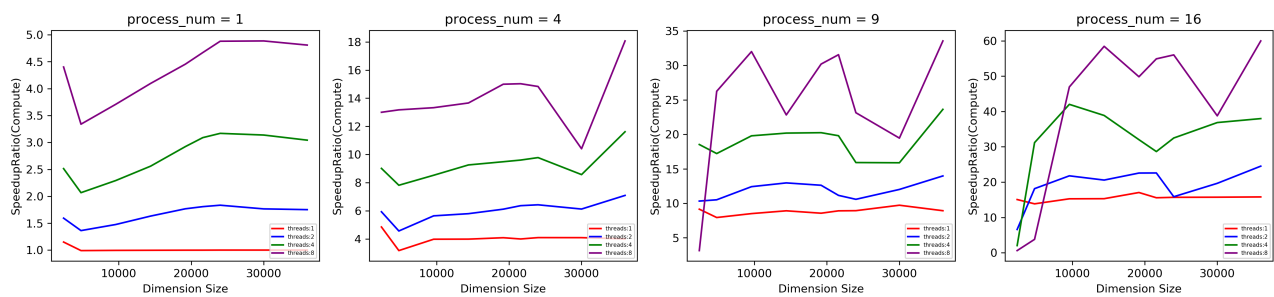


图 5: SpeedupRatio(Compute)

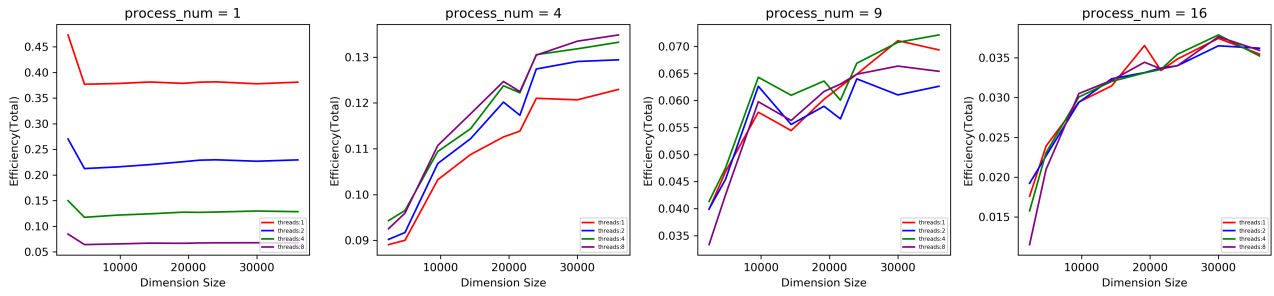


图 6: Efficiency(Total)

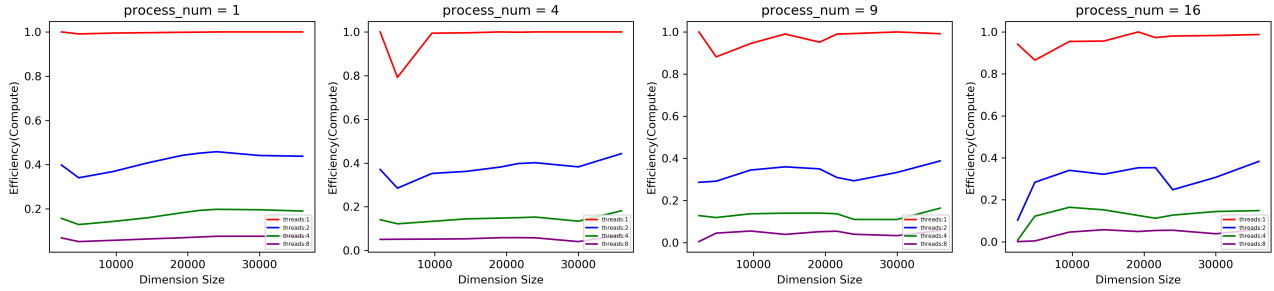


图 7: Efficiency(Compute)

4.3 和作业 3 中的比较

本次作业比作业 3 添加了 OpenMP 的多线程编程，具体做法是在每个线程中派生出若干个新线程，同时计算并行效率时应注意除以 $p \times th$ 。和作业三中的结果比较，主要是计算时间有了明显加速，平均加速比从 0.4 上升到了 0.6。但是程序热点依然在于 MPI 通信，占掉了 95% 以上的时间，并且通信部分使用 MPI_Scatter() 是无法并行的，但是计算时间有了明显加速，和线程个数呈正相关。这进一步印证了“线程”调度开销远小于“进程”通信开销的结论，因此，在存在较多数据共享的场合，使用“多线程编程”能达到更高的效率；但是在可靠性上，多进程因为数据隔离，实际上更加安全，并且适用于多机并行。实际应用应该综合两者的优点进行设计。此外我发现，由于核心数量的限制，在 p 较大时，增大线程数无法起到加速效果，因为此时多线程的任务实质上是在一个核心上进行的，这反而会拖慢效率。

$n \setminus p$	4	16	25
2000	0.013/0.4523	0.013/0.4618	0.168/0.0268
8000	0.184/0.3611	0.190/0.3601	1.274/0.0311
12000	0.422/0.3514	0.437/0.3506	2.234/0.0314
16000	0.781/0.3514	0.768/0.3571	5.006/0.0315
24000	2.556/0.3412	2.548/0.3496	8.891/0.0281

表 1: 作业 3 中 Total time(s) & Speedup Ratio(考虑数据分发)

我还发现，在不同线程数上，并行效率与数据规模存在相关性，比如在 $p = 4, t = 16$ 时，规模 $n = 20000$ 的情况获得了远超其他规模的并行效率。这说明为了达到更高的效率，应对不同的数据规模设计不同的线程数和进程数。

5 Programming Assignment 2

5.1 运行方式

“make”编译成功后，执行“make run [p=...] [n=...] [c=...] [max=...]”可以运行程序，其中 p 代表线程数， n 代表数据规模， c 代表块大小 (chunk_size)， max 代表生成的数据范围 (鉴于 $O(n+k)$ 计数排序的要求，推荐为 $max < 5000$)。最终程序会输出各种排序的运行时间、加速比及并行效率。一个可能的运行结果如下：

```
1 thread_num: 4, num: 50000, chunk_size:32, max_num:50
2 serial count:          0.321601s
3 parallel count:        0.311788s
4 parallel count+merge:  0.044520s
5 parallel count O(n+k): 0.020629s
6 qsort() time:          0.003546s
7
8 Speedup Ratio(count):  1.031474
9 Speedup Ratio(count+merge): 7.223718
10 Speedup Ratio(count O(n+k)): 15.589802
11
12 Efficiency(count):     0.257868
13 Efficiency(count+merge): 1.805929
14 Efficiency(count O(n+k)): 3.897450
15
16 parallel count <static>: 0.293447s
17 parallel count <dynamic>: 0.256954s
18 parallel count <guided>: 0.361447s
19
20 Speedup Ratio <static>: 1.095942
21 Speedup Ratio <dynamic>: 1.251590
22 Speedup Ratio <guided>: 0.889759
23
24 Efficiency <static>:    0.273986
25 Efficiency <dynamic>:   0.312897
26 Efficiency <guided>:    0.222440
```

5.2 哪些变量是私有的，哪些变量是共享的

private	i, j, count
shared	a, n, temp

5.3 外循环是否存在循环依赖

不存在循环依赖。a 和 n 只有读操作，对 temp 只有写操作，并且不同线程不会访问 temp 的同一元素，因此不存在。

5.4 是否能并行化对 memcpy 的调用

修改区间划分之后可以并行化 memcpy，因为 memcpy 的内部实现也是基于循环的，并且没有循环依赖。下面我给出实现代码（测试代码放在./test 文件夹下）：

并行化 memcpy

```
1  int interval = n / thread_count + 1, i;  
2  # pragma omp parallel for num_threads(thread_count)  
3  for(i = 0; i < n; i += interval){  
4      memcpy(a+i, temp+i, min(n-i, interval) * sizeof(int));  
5  }
```

下面我给出实测的不同规模 n 和线程数 $threads$ 的加速比数据，注意横坐标是对数的：

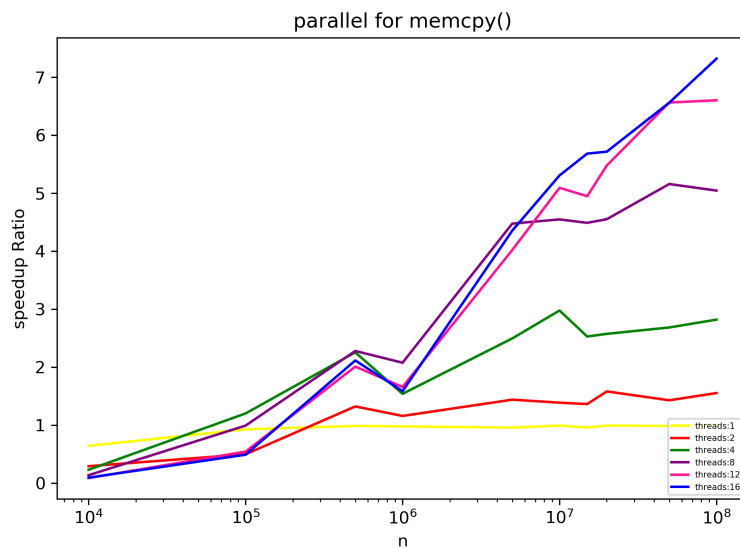


图 8: Speedup Ratio of OpenMP parallel for memcpy()

从数据中可以看到，openMP 对于 $n > 10^5$ 的数据规模会有比较明显的加速。在编写程序时可以采用加速后的 memcpy()，但是因为题给计数排序是 $O(n^2)$ 的，所以这一改进对整体性能优化不明显。

5.5 程序说明

为了更全面地了解各种优化的效果，我分别实现了“串行 $O(n^2)$ 计数排序”，“并行 $O(n^2)$ 计数排序”，“并行 $O(n+k)$ 计数排序”，“四分区间最终归并的计数排序”共 4 个版本，并进行了性能测试。编译选项选择“-O3 -march=native”

5.5.1 串行 $O(n^2)$ 计数排序的优化

我发现在作业要求中给出的计数排序是 $O(n^2)$ 的，并且在循环内的判断是十分低效的，*if-else* 分支预测错误会浪费很多时间。因此我将原程序修改成了如下代码，在 $n = 100000$ 的规模下，优化后的版本 (1.34s) 比原始版本 (28.34s) 快了 20 倍以上。具体思路是使用两段循环代替了之前的分支判断。

Count_sort_fast()

```

1  for(i = 0; i < n; i++){
2      count = 0;
3      for(j = 0; j < i; j++){//减少分支预测
4          count += (a[j] <= a[i]);
5      }
6      for(j = i; j < n; j++){
7          count += (a[j] < a[i]);
8      }
9      temp[count] = a[i];
10 }

```

5.5.2 $O(n^2)$ 版本的并行化

之后我对改进后的计数排序进行了并行化，正如前面几个小题分析的那样，外层循环可以并行化，因此复杂度可以从 $O(n^2)$ 下降到 $O(n^2/th)$ ，其中 th 是线程个数。openMP 的并行化十分简单，代码如下。

```

                                Count_sort_para()
1  # pragma omp parallel for num_threads(thread_num) private(i, j, count)
2      for(i = 0; i < n; i++){
3          count = 0;
4          for(j = 0; j < i; j++){
5              count += (a[j] <= a[i]);
6          }
7          for(j = i; j < n; j++){
8              count += (a[j] < a[i]);
9          }
10         temp[count] = a[i];
11     }

```

5.5.3 $O(n + k)$ 版本的并行化

实际上计数排序有复杂度 $O(n + k)$ 的实现，其中 k 是数据范围。当 $k < n \log n$ 的时候，计数排序的理论时间复杂度是小于基于比较的排序的。简单分析可以看到，因为 $O(n + k)$ 的计数排序涉及到对每个数字频率的统计，而不是直接算出它们的位置，之后还需要计算“前缀和”，所以存在着较强的依赖。并行化的方式需要特殊设计，而不是像 $O(n^2)$ 的并行那样直接。

我的思路是，将数据均分给各个线程之后，首先在各个线程内部设置 `local_count[]` 数组，用于统计本线程负责数据的频率统计，然后再对各个线程的 `local_count[]` 进行规约得到 `count[]`。之后在主线程内计算前缀和，之后各个线程再负责计算每个元素的位置。

如果线程数为 th ，串行版本的计数排序复杂度为 $O(2n+k)$ ，则并行后的奇数排序复杂度为 $O(2n/th+2k)$ ，但是考虑到线程调度的开销以及本地数组 `local_count[]` 创建销毁的开销，实际上并行版本的常数比较大。实测发现，4 个线程版本的速度仅仅是串行版本的 1.5 倍左右，而且这种优化只适用于 k 比较小的情况。当然，因为复杂度是线性的，依然比原始 $O(n^2)$ 的版本快了很多！

```

1  int length = n / thread_num + 1;
2  # pragma omp parallel for num_threads(thread_num)
3      for(int t = 0; t < thread_num; t++){//统计频率, O(n)
4          int* local_count = new int[k+1];
5          memset(local_count, 0, (k+1)*sizeof(int));
6          int begin = t * length;
7          int end = std::min(begin + length, n);
8          for(int i = begin; i < end; i++){//各线程统计线程内数组
9              local_count[a[i]] ++;
10         }
11         for(int i = 0; i < k; i++){//规约
12     #         pragma omp critical
13             count[i] += local_count[i];
14         }
15         delete[] local_count;
16     }
17     int i, j, index = 0, temp, loc;
18     for(i = 1; i < k; i++){
19         count[i] += count[i-1];
20     }
21 # pragma omp parallel for num_threads(thread_num) private(temp, loc)
22     for(i = n-1; i >= 0; i--){//统计结果O(n), 可以简单并行
23         temp = a[i];
24     #     pragma omp critical
25         loc = --count[temp];
26         result[loc] = temp;
27     }

```

5.5.4 最后 4 路归并的计数排序

实际上, 归并排序是十分修改成并行的。为了使得程序达到更好的表现, 我将数据划分为 4 部分, 交给 4 个线程做并行计数排序, 之后进行 4 路归并得到最终的结果。在底层排序采用 $O(n^2)$ 的计数排序的情况下, 复杂度优化到了

$$O(\frac{(n/4)^2}{th} + \frac{n}{2} + n) = O(\frac{n^2}{16th} + \frac{3n}{2})$$

归并部分采用线性归并, 代码如下:

```

1  if(can_sort){//4个线程分别进行计数排序
2      Count_sort_para(a + left, right - left);
3      finish_count ++;
4      return;
5  }
6  int* b = new int[right - left];//如果不需要排序, 对两个子数组进行线性归并

```

```

7  int mid = (left + right) >> 1;
8  for(int p = left, q = mid, i = 0; p < mid || q < right; ){
9      b[i++] = (q >= right || (p < mid && a[p] <= a[q])) ? a[p++] : a[q++];
10 }
11 memcpy(a + left, b, (right - left) * sizeof(int));
12 finish_count ++; //线程完成数++

```

归并部分的调度如下,其中 mergeThread() 是我基于 Pthread 封装的线程类,这也算是 openMP 和 Pthread 混合编程的一个尝试。显然, Pthread 更适用于细粒度地进行线程调度,赋予了开发人员更大的自由度,但是同时也增加了繁琐性。将 openMP 和 Pthread 混合使用,可以综合两者的优点,达到更好的效果。

Count_sort_para_merge()

```

1  int m2 = n >> 1; //区间划分
2  int m1 = m2 >> 1;
3  int m3 = (m2 + n) >> 1;
4  //先分配给4个线程进行计数排序
5  mergeThread* t1 = new mergeThread(a, 0, m1, 1);
6  mergeThread* t2 = new mergeThread(a, m1, m2, 1);
7  mergeThread* t3 = new mergeThread(a, m2, m3, 1);
8  mergeThread* t4 = new mergeThread(a, m3, n, 1);
9  finish_count = 0;
10 t1->start(); t2->start(); t3->start(); t4->start();
11 t1->join(); t2->join(); t3->join(); t4->join();
12 while(finish_count < 4) //busy-wait
13     ;
14 if(finish_count == 4){
15     finish_count = 0;
16     t1->set(0, m2, 0); //重新设置排序区间, reuse t1 and t2
17     t2->set(m2, n, 0);
18     t1->start(); t2->start(); t1->join(); t2->join(); //两个线程进行2路归并
19 }
20 while(finish_count < 2) //busy-wait
21     ;
22 if(finish_count == 2){
23     finish_count = 0;
24     t1->set(0, n, 0); //reuse t1, 负责最后的2路归并
25     t1->start(); t1->join();
26 }
27 while(finish_count < 1) //busy-wait
28     ;

```

5.6 性能测试

下一页展示了我对上面 4 种版本的计数排序的性能进行了测试,整数范围 0 – 49, 各版本计算用时、加速比、并行效率如下 (原始数据位于 ./data1.json 下, 数据量过大, 这里只给出统计图):

从数据中可以看到, `qsort()` 的效率最高, 但是实际上, 因为本题中整数范围 0-49, 所以快速排序中很多的区间划分不需要做任何的数据移动操作, 使得 `qsort` 获得了很大的性能改进。但是对于完全随机的 0-65535 的数据, `qsort` 的性能产生了明显下降, 速度会小于 $O(n+k)$ 的计数排序。

在 $O(n^2)$ 的版本中, 两者使用相同的算法, 在线程数比较小时 (比如 `threads=2`), 并行版本的速度并没有比串行版本快, 线程的调度开销比较大。此外, 因为使用了 “-O3 -march=native” 编译选项, 串行版本的 `CountSort()` 比 “-O2” 选项又提升了 3 倍左右, 所以在线程数较少的时候, 线程调度占了很大比重的时间, 导致并行效率不高。在线程数 >4 之后, 加速比开始超过 1, 并且数据规模越大, 加速效果越明显。这是因为计算时间在总时间中比重上升, 使得线程调度时间相对而言影响减少了。此外, 线程数的增加会导致并行效率的下降, 这一现象是符合预期的, 因为调度开销会增多。

此外可以看到, `CountSort+4 路归并` 的版本获得了更高的效率, 尽管算法依然是 $O(n^2)$ 的, 但是因为分成 4 段进行排序, 所以这种加速更加显著。加速比始终超过了 1, 并且并行效率也是随着数据规模不断上升的。我们也可以看到, `Pthread+OpenMP` 不仅为开发人员提供了灵活性, 还带来了更大的效率提升。

对于 $O(n+k)$ 复杂度的 `CountSort` 而言, 讨论其加速比似乎没有意义, 因为复杂度已经和 $O(n^2)$ 不在一个量级, “加速比” 只能再一次印证线性复杂度的高效! 但是我也发现了一个有趣的现象, 随着进程数的上升, $O(n+k)$ 的计数排序的并行效率有了显著的下降。分析之后我认为有两个原因: 1. 我们的数据规模对于 $O(n+k)$ 的算法没有充分体现其并行效率, 因为计算时间占比大大下降了, 线程的调度时间占了上风, 增大数据规模后, 计数排序的性能又有了提升。2. $O(n+k)$ 版本的 `CountSort` 存在很多不完全并行的情况, 诸如存在临界区、子线程需要创建 `local_count[]` 数组等, 这极大地增大了算法的常数。

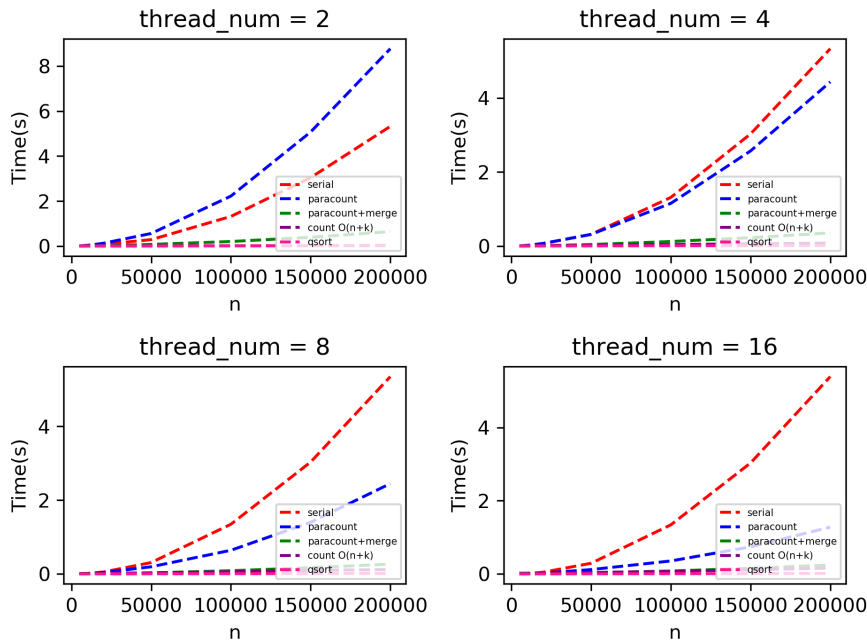


图 9: Time(s)

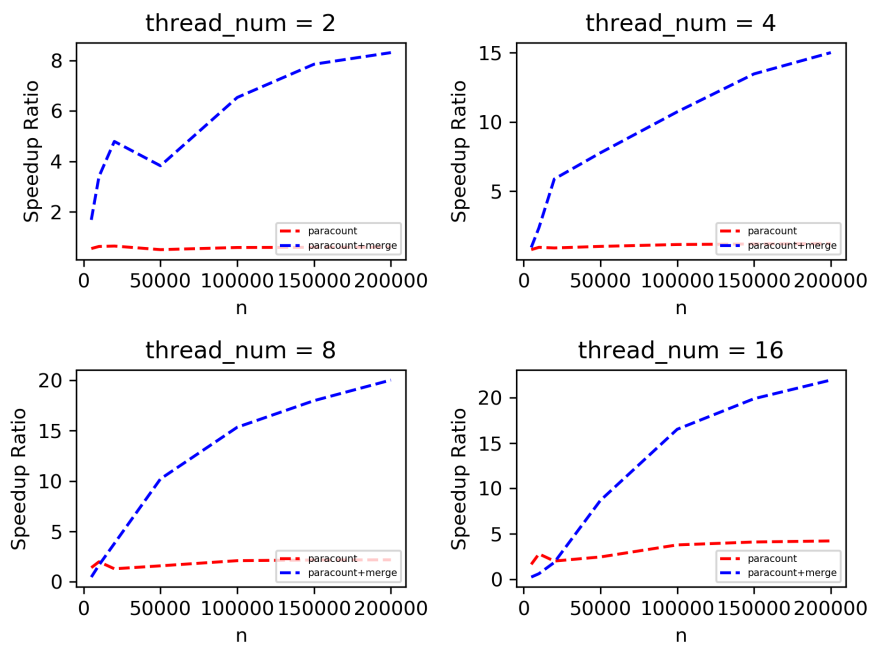


图 10: SpeedupRatio

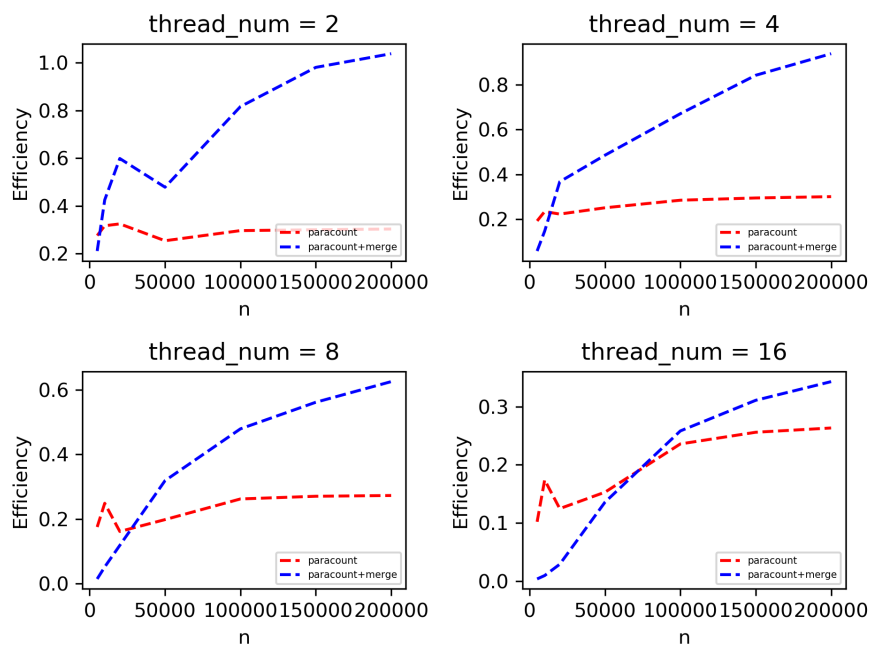


图 11: Efficiency

5.7 不同调度策略和块大小的对比

我对不同调度策略和 `chunk_size` 的运行效率进行了测试 (只对比了 $O(n^2)$ 的串行、并行版本), 对于不同的线程数 t 和 $n = 100000$ 的数据规模, 得到的结果如下 (原始数据位于 `./data2.json` 下, 数据量过大, 这里只给出统计图).

从数据中可以看到, `dynamic` 调度的加速比和并行效率是最高的, `static` 次之, `guided` 最差。这说明线程间的负载其实是不太均匀的。对于不同的 `chunksizes`, 2 个线程时, 在块大小为 150 左右时, 并行效率达到最高。8 或 16 个线程时, 块大小为 100 时并行效率达到最高。实际上, 块大小的变化对并行效率的影响并不大, 整体上曲线都十分“水平”。最优块大小和线程数之间存在相关性, 在实际工作中, 我们需要对不同的线程数设置不同的块大小, 以达到最优速度。最终我选择 `chunk_size = 100`。

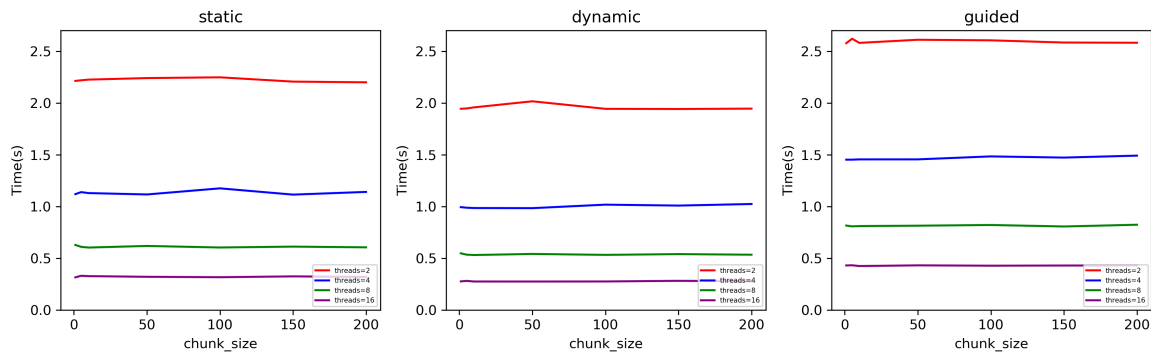


图 12: Time(s)

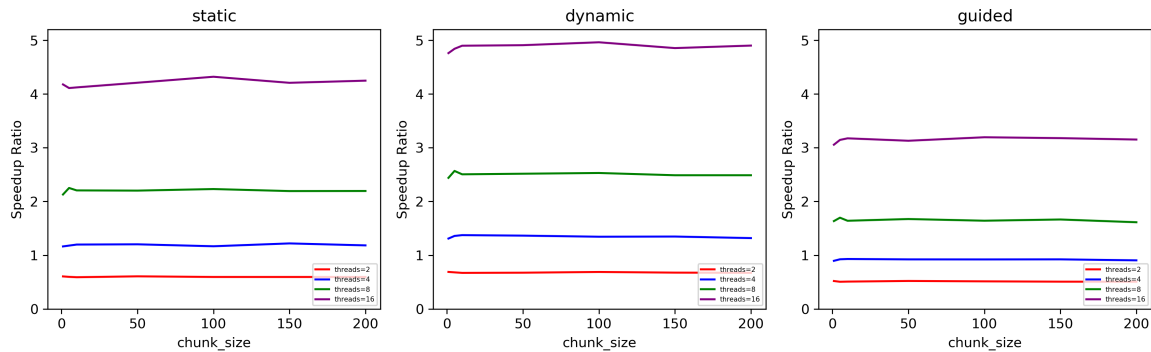


图 13: Speedup Ratio

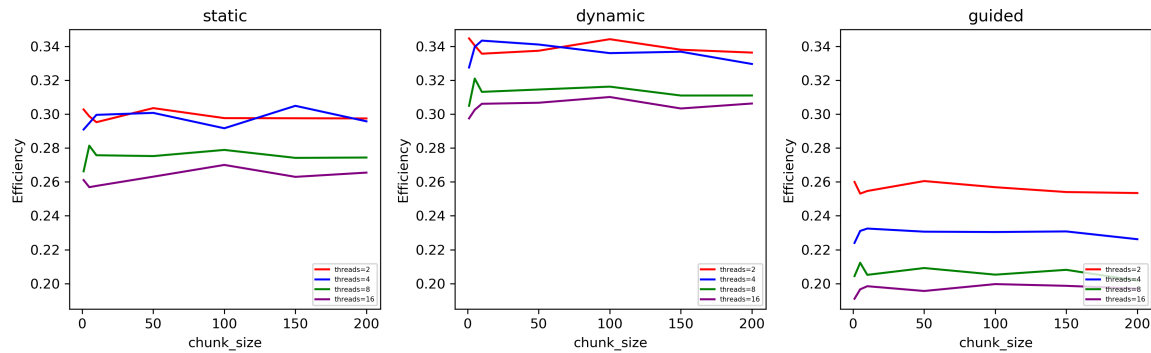


图 14: Efficiency

高性能计算导论: hw6

刘泓尊 2018011446 计 84

2020 年 5 月 13 日

目录

1 GEMM	2
1.1 代码实现	2
1.1.1 循环展开的版本	2
1.1.2 矩阵分块的版本	2
1.1.3 循环展开	3
1.1.4 内存对齐	3
1.2 性能测试与分析	4
1.2.1 对不同规模方阵和不同分块大小的测试	4
1.2.2 对 MemPitch 版本的测试	5
1.2.3 对狭长形矩阵的测试	6
1.2.4 对 double 的统计数据展示	6
1.3 总结	7

本次作业已上传至集群./2018011446/hw6 文件夹下。

1 GEMM

本次实验我实现了基于矩阵分块 (tile) 的 GEMM 程序, 包括 SGEMM 和 DGEMM, 并利用 Shared Memory 进行加速。功能为计算矩阵-矩阵乘法:

$$C_{m \times n} = \alpha A_{m \times k} \cdot B_{k \times n} + \beta C_{m \times n}$$

其中 $\alpha = 1.5, \beta = 0.5$.

1.1 代码实现

1.1.1 循环展开的版本

在 baseline 版本的矩阵乘法中, 每个线程都要读矩阵 A 的一行和矩阵 B 的一列, 计算与访存比约为 1:1, 我先对 baseline 版本进行了循环展开, 作为一个小测试。代码如下:

```
1 int i = 0;
2 for(; i < k; i += stride) {
3     tmp += A[row_off + i] * B[col + i*n];
4     tmp += A[row_off + i + 1] * B[col + (i+1)*n];
5     tmp += A[row_off + i + 2] * B[col + (i+2)*n];
6     tmp += A[row_off + i + 3] * B[col + (i+3)*n];
7 }
8     i -= stride;
9 for(; i < k; i++){
10     tmp += A[row_off + i] * B[col + i*n];
11 }
```

实际测试发现, 循环展开之后性能没有明显提升, GFLOPS 仅仅提升了 5 个点左右, 因此我没有再使用此版本。

1.1.2 矩阵分块的版本

观察可以发现, 矩阵乘法中每个元素都被多个线程重复使用, 将元素存入 Shared Memory 供线程使用有望获得更好的性能, 因此我使用矩阵分块策略, 每一个 block 计算一小块方阵, 每个 thread 计算这一小块方阵的一个元素, 分块大小为 TILE_WIDTH. 当矩阵不能被方阵完全覆盖时, 方阵补零做 padding 即可, 每个线程将矩阵 A 和 B 的元素读入 Shared Memory. 代码如下:

gemm_kernel()

```
1 int block_col = threadIdx.x;
2 int block_row = threadIdx.y;
3 int col = blockIdx.x * blockDim.x + block_col;
4 int row = blockIdx.y * blockDim.y + block_row;
5 int i, j;
6 T v = 0;
7 __shared__ T shareM[TILE_WIDTH][TILE_WIDTH];
8 __shared__ T shareN[TILE_WIDTH][TILE_WIDTH];
9 for(i = 0; i < block_num; i++){
```

```

10     shareM[block_row][block_col] = ((i * TILE_WIDTH+block_col < K) && (row < M))
        ? A[row * K + (i * TILE_WIDTH+block_col)] : 0.;
11     shareN[block_row][block_col] = ((i * TILE_WIDTH+block_row < K) && (col < N))
        ? B[(i * TILE_WIDTH+block_row) * N + col] : 0.;
12     __syncthreads();
13     if( (col < N) && (row < M) ){
14         for(j = 0; j < TILE_WIDTH; j++){ //can unroll loop
15             v += shareM[block_row][j] * shareN[j][block_col];
16         }
17     }
18     __syncthreads();
19 }
20 if( (col < N) && (row < M) ) {
21     C[row * N + col] = alpha*v + beta * C[row * N + col];
22 }

```

在第二部分将对性能进行测试和分析。

1.1.3 循环展开

为了更大限度地激发指令级并行，我将上述代码中的循环部分展开，将步数为 `TILE_WIDTH` 的循环改为了 `TILE_WIDTH` 条顺序执行的指令，但实测发现没有明显的加速，GFLOPS 仅仅提升了 1. 左右。因为循环展开的部分十分简单，这里就不罗列代码了。

1.1.4 内存对齐

线程访问内存以段对齐的方式读取，遇到非 2 的幂次的数组，实际上也是要读取完整的 2 的幂次，因此我对矩阵 *A*, *B*, *C* 做了内存对齐，方法是使用 `cudaMallocPitch()` 函数。预处理部分如下：

MemoryPitch

```

1  T *tmp_A, *tmp_B, *tmp_C;
2  size_t pitch_a_device, pitch_b_device, pitch_c_device;
3  cudaMallocPitch((void**)&tmp_A, &pitch_a_device, sizeof(T)*K, M);
4  cudaMallocPitch((void**)&tmp_B, &pitch_b_device, sizeof(T)*N, K);
5  cudaMallocPitch((void**)&tmp_C, &pitch_c_device, sizeof(T)*N, M);
6  cudaMemcpy2D(tmp_A, pitch_a_device, A, sizeof(T)*K, sizeof(T)*K, M,
        cudaMemcpyDeviceToDevice);
7  cudaMemcpy2D(tmp_B, pitch_b_device, B, sizeof(T)*N, sizeof(T)*N, K,
        cudaMemcpyDeviceToDevice);
8  cudaMemcpy2D(tmp_C, pitch_c_device, C, sizeof(T)*N, sizeof(T)*N, M,
        cudaMemcpyDeviceToDevice);
9  checkCudaErrors(cudaDeviceSynchronize());

```

预处理之后调用内存对齐版本的核函数，寻址时不再使用 `A[row * M + col]` 而是 `A[row * A_pitch + col]`，可以更好的利用访存性能。在第二部分将对此优化的性能进行测试和分析。

1.2 性能测试与分析

分析部分的统计图均为针对 float 的统计数据，double 的数据放在报告最后。

1.2.1 对不同规模方阵和不同分块大小的测试

使用上述矩阵分块版本 (无内存对齐) 针对不同规模的方阵进行测试，得到的结果如下:

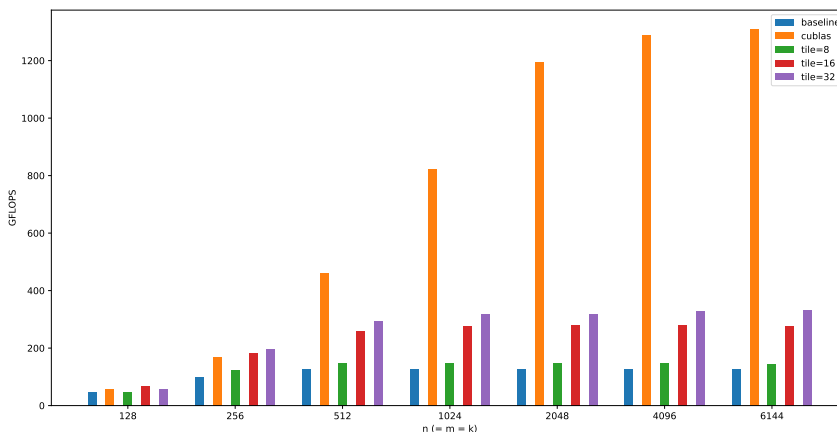


图 1: 矩阵分块版本 GEMM(with cublas)(float)

可以看到, cublas 非常快, 我的实现仅仅达到了 cublas 的 25% 左右。为了便于展示我的数据, 接下来的统计将不计入 cublas 的程序, 但我十分希望助教老师讲解作业的时候能介绍一下 cublas 的优化技巧!

我对不同矩阵规模 (N) 和不同分块大小 (TILE) 进行统计, 得到 GFLOPS 如下:

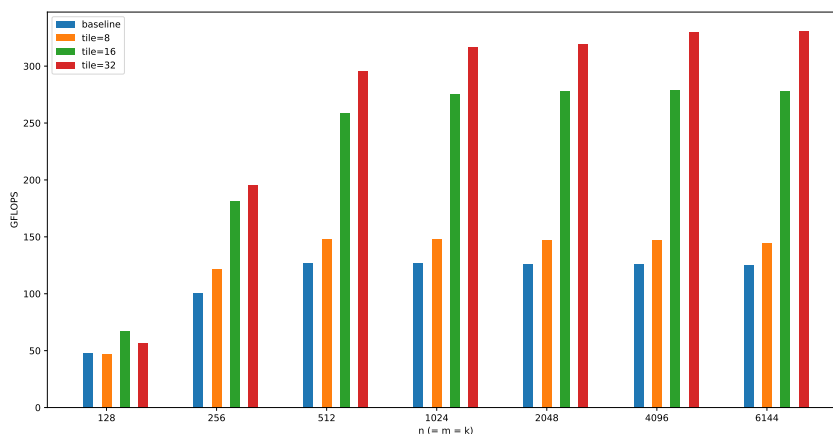


图 2: GEMM 性能对比 (不同矩阵规模和分块大小)(float)

从图中可以看到, baseline 版本和我的版本均在 $N=1024$ 左右达到性能峰值。不同分块大小对性能的影响很大, $TILE=8$ 的版本仅仅比 baseline 好一点; $TILE=16$ 的版本可以达到 baseline 的两倍以上; $TILE=32$ 的版本可以达到 baseline 的三倍左右。如果 Shared Memory 足够大, 进一步增大 $TILE_SIZE$ 可以取得更好的性能。

分块策略是矩阵乘法常用的优化策略。因为矩阵乘法中每个元素都被多个线程重复使用，分块乘法可以更好地利用局部性。此外，将元素存入 Shared Memory 供线程使用可以获得更好的性能，因为 Shared Memory 能够用来隐藏由于 latency 和 bandwidth 对性能的影响。

我进一步探究了 row-major 和 column-major 对性能的影响，我的程序涉及向 SM 读数据和存数据，因为在同一个 warp 中的 thread 的使用相邻的 `threadIdx.x` 来访问 SM，如果使用 column-major 来存取数据，每个 SM 的读写都会导致 bank-conflict，导致程序的访存性能降低。因此在读写 SM 的时候，应该尽量使用 row-major。

1.2.2 对 MemPitch 版本的测试

我对加了 MemPitch 的矩阵乘法做了测试，矩阵规模均选取不是 2 的幂次的数据，这样更方便展现 MemPitch 的效果。

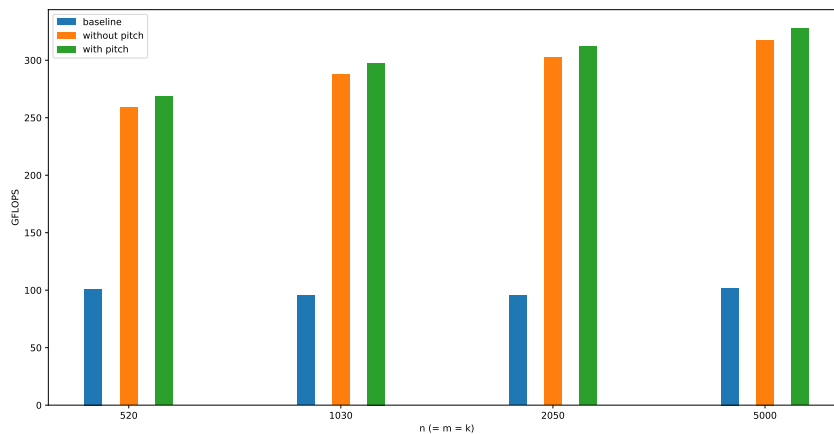


图 3: 使用 MemPitch 的 GEMM(float)

可以看到，使用 MemPitch 的版本性能有了略微的提升 (实际上因为 GFLOPS 单位很大，运算次数已经提升很多了)。对于规模不是 2 的 n 次幂的矩阵，使用内存对齐预处理可以达到更好的性能。

1.2.3 对狭长形矩阵的测试

为了更全面地展示性能，我对长方形矩阵做了性能测试，使用 `TILE_SIZE=32` 的分块策略。性能统计如下。

baseline 对于矩阵形状不太敏感，但是使用分块策略的矩阵乘法对矩阵形状较为敏感，虽然性能一直优于 baseline，但是对于特别“狭长”的矩阵，性能有了明显的下降。这是因为在固定的 `TILE_SIZE` 下，矩阵宽度的减小将减少 block 的数量 (分块策略的不灵活性也由此体现)，导致有效线程数减少，性能降低。

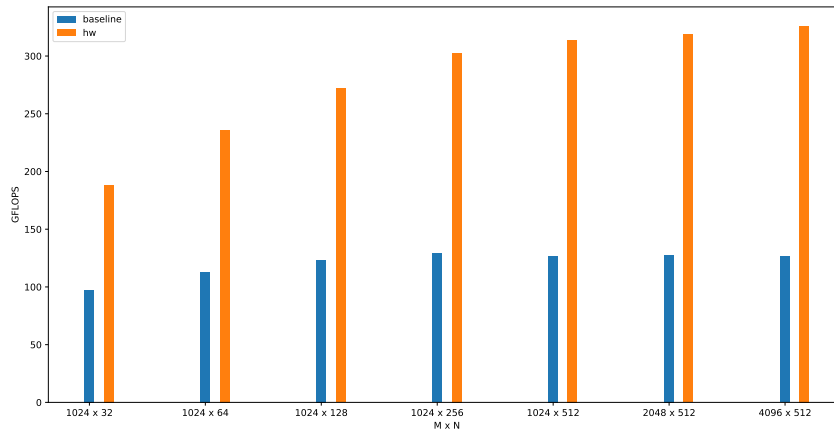


图 4: 长方形矩阵的性能测试 (float)

1.2.4 对 double 的统计数据展示

double 类型矩阵的结果与 float 类似，因此我放在最后统一展示。cublas 的性能有了明显下降，我的实现相对于 baseline 的改进基本和 float 类型类似。

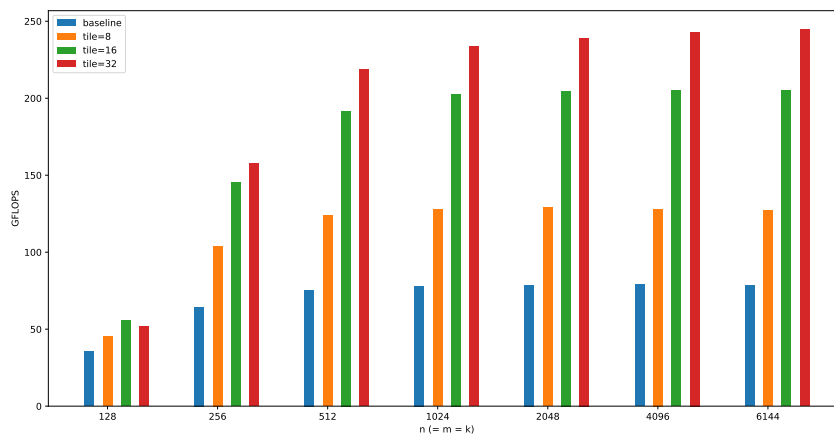


图 5: 不同规模矩阵和分块策略 (double)

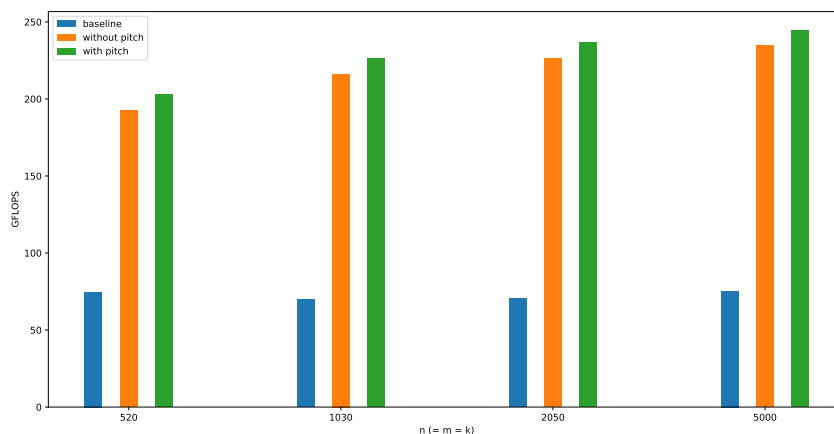


图 6: 使用 MemPitch 的 GEMM(double)

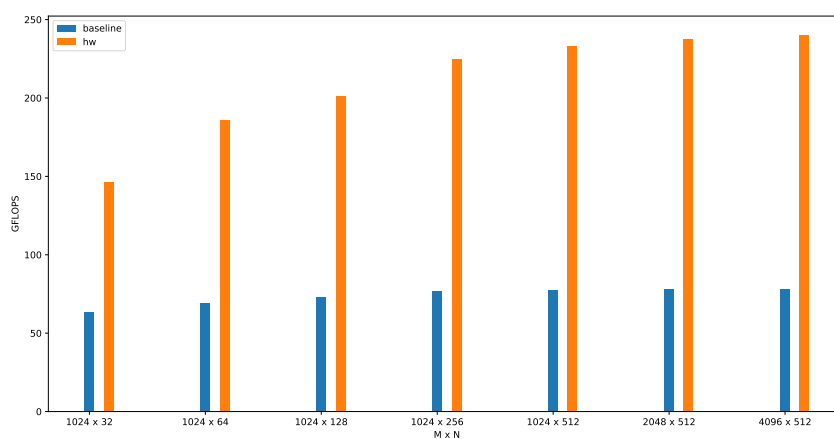


图 7: 长方形矩阵的性能测试 (double)

1.3 总结

本次作业我实现了 CUDA 版本的 GEMM，采用矩阵分块策略加速矩阵乘法，达到了 baseline 版本 3 倍的加速。使用了 Shared Memory 存取数据，并尽可能避免了 Bank Conflict。但是和 cublas 的实现依然存在很大的差距，仅仅有其 1/5 - 1/4 的性能。接下来可以进一步研究使用数据预取和指令级并行来进一步优化。感谢助教老师的悉心指导！