并行计算 II 2023 春季第二次作业：MPI 优化 Attention 算法

1900017702 朱越

0. 概要

本作业针对Attention算法使用MPI进行优化，在四节点各4核下实现了约**640**倍加速，最终用时约**0.13s**（不计入转置时间，如果计入则为0.15s）。

分析和实验过程见第3部分，最终策略见第4部分。

1. Attention算法

$$Attention(\mathbf{Q},\mathbf{K},\mathbf{V})=\mathbf{softmax}(\frac{\mathbf{QK^T}}{\sqrt{\mathbf{d_k}}})\mathbf{V}$$

2. 串行计算

见attention.cpp，核心部分如下：

```
Matrix *kt = transpose(k);
Matrix *qk = matmul(q, kt);
Matrix *qk_s = scale(qk, 1.0 / sqrt(k->col));
Matrix *qk_s_s = softmax(qk_s);
Matrix *qkv = matmul(qk_s_s, v);
```

3. 并行计算

1. 矩阵乘法
   - 公式
     $$(a_{ij})_{m\times n}(b_{ij})_{n\times p}=(\sum_{k=1}^{n}a_{ik}b_{kj})_{m\times p}$$
   - 2*2分块
     $$\begin{pmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix}\begin{pmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{pmatrix}=\begin{pmatrix}A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22}\\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22}\end{pmatrix}$$
   - 时间复杂度
     由分块得递推公式：$T(n)=8*T(\frac{n}{2})+\Theta(n^2)$
     由主定理得，时间复杂度：$O(n^3)$。
   - （补充：存在一种Strassen算法可以把子矩阵的乘法降低到7次，此时时间复杂度可以降到$O(n^{log_27})$。但实现起来比较复杂而且在非递归情况下复杂度降低得不多，因此本文忽略了这种情况）
2. 分块乘法

> 分块乘法还可以扩展到任意的矩阵划分方式，只要左矩阵的按列划分和右矩阵的按行划分方式相等即可。在分块乘法中，观察各结果块的计算，发现彼此之间没有任何依赖，这给并行计算带来了可能性。接下来以最基础的两个形式分析可并行性：

   - 空间划分：按左矩阵行或右矩阵列划分，分别计算各个空间

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} A_1B_1 & A_1B_2 \\ A_2B_1 & A_2B_2 \end{pmatrix}$$

- 加数划分：按左矩阵列和右矩阵行划分，分别计算各个加数

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} A_1B_1 + A_2B_2 \end{pmatrix}$$

3. 算子耦合

> 对单个矩阵乘法而言，理论上可以使用任意的矩阵划分方式进行并行化。为了保证结果一致性，需要各控制流在并行计算后作结果同步：分别计算两块空间后，各个控制流归并；分别计算两个加数后，各个控制流聚合。总体通信量和负载量相同，效果区别不大。
>
> 对矩阵算子的流水线而言，情况有所不同。由于前后结果是互相依赖的，最保守的方法是每个算子都保证结果一致性。然而，有一些情况下我们无需保证中间过程的结果一致性——前一个算子的并行划分输出正好满足后一个算子的并行划分输入，就能节省一次同步，满足"**通信极小**"的目标。由此，导出两点推论：
>
> 1. 如果后一个算子是按行/列并行划分任务的，则前一个算子尽可能选择按对应行/列并行输出结果的；反之亦然。
> 2. 前一个算子尽量不选择加数划分，因为这种划分方式没有缩小后续问题规模，为了保证后续算子总任务量不变必然要进行同步规约。

- Attention算法运用上述结论：
  1. softmax函数是按行划分，按行输出的，要求前算子按行输出，后算子按行输入。因此，第一步和最后一步矩阵乘法都应该是按行划分的。
  2. softmax函数是非线性的，如果选择加数划分，softmax函数不可以在两个加数上分别计算并聚合。因此不使用加数划分方式。
  3. 最终正确性检查的是各进程Allreduce的矩阵各元素之和，因此最后无需同步计算结果。
- 因此得出本文的并行策略：

$$Attention(\mathbf{Q_{row}}, \mathbf{K}, \mathbf{V}) = \mathbf{softmax}(\frac{\mathbf{Q_{row}}\mathbf{K^T}}{\sqrt{\mathbf{d_k}}})\mathbf{V}$$

4. （补充：形式化定义）

> 前述的"空间划分"和"加数划分"是我口头的称呼，随后我感觉到对于这种规律应该存在某种形式化的定义，因此我找到了在分布式机器学习训练系统中参考信息：

1. 数据并行：SIMD，要求数据具有一定性质
   1. 数据内并行：数据可以划分为多个维度，各维度之间的计算图独立
   2. 数据间并行：多个批次样本独立计算，结果可聚合（或近似聚合）
2. 算子并行：MISD，要求算子具有一定性质
   1. 算子内并行：算子可以划分为多个分量，各分量之间的计算图独立
   2. 算子间并行：多个顺序依赖算子可流水线化（同时把数据也按批次划分）

- 在Attention算法中，虽然是大规模矩阵运算，但由于它特殊的按行独立性质，本文对其按行划分作数据内并行。

References

- 算子级并行和流水线并行——MindSpore
- 数据并行和模型并行——OPEN MLSYS
- Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

5. 节点与核心，进程与线程

- 节点与核心

> 在集群上限制使用总核心数一定（只能申请16个核心作为计算资源）时，任务是集中在少量节点的更多核心上更快，还是更多节点的少数核心上更快？对于Attention问题的并行计算而言，在按照前一节所述方式进行任务划分后，计时部分内不涉及彼此之间的数据传输，因此在理论上并无区别。
>
> 但在实验中，单节点16核的平均表现不如四节点4核的平均表现，并且表现出性能抖动。这可能是因为单个机器的频率存在上限，多个核心并行工作时达到峰值，计算能力受限，而在多个节点上则快速且稳定。

- 进程与线程

> 同理，在计时部分中各控制流间不存在数据传输，理论上并无区别。
> 但在实验中，由于我没有花精力针对多线程编程作优化，仅在for循环部分作了并行化，这样每次循环之前都要初始化并行区，产生了额外的代价，肯定是不如用进程作并行的。

- 结论：使用四节点4核心的配置，启动16个进程作并行，每个进程都是单线程工作的。因为进程间彼此无关，加速比约16倍。

6. 编译选项

> 编译器可以帮我们优化一些公共子表达式等情况。然而编译优化和并行计算部分的耦合比较复杂，在深度优化后往往不是简单的乘性关系或加性关系。另一方面，最终结果必然包含编译优化选项，因此，为了取得最终效果最好的组合，任何深度优化都应该在开启编译优化的情况下进行。因此，只讨论最初的两个编译优化情况。以下是在方案和非最终并行方案下观察-Ofast选项的运行结果摘要，可以看到，随着代码的优化，编译优化的效果（加速比）还变得更好了。

```
basic
Correct! Time: 73.7376849651

basic -Ofast
Correct! Time: 21.7690162659

try without Opt
Correct! Time: 4.8341712952

try
Correct! Time: 0.7029755116
```

- 结论：使用-Ofast编译优化选项，可以加速约3-7倍。

> 6.26补充：使用自动向量化编译选项-march=native，可以使总时间进一步缩小到0.13s（不计入转置时间）。

7. 缓存优化

> 在Attention算法中，主要计算代价集中在矩阵乘法上，时间复杂度$O(n^3)$，其余部分都是$O(n^2)$的，在计算规模较大时所占比例很小。因此，接下来的优化仅针对矩阵乘法。

- 存储方式：matmul_T

> 对于二维矩阵的访存，按行遍历空间局部性好，而按列遍历空间局部性差。而矩阵乘法又需要计算一行乘以一列。因此，可以把右矩阵预先作转置，这样就变成两个矩阵都按行访问，对缓存线友好。此外，题目允许改变矩阵的存储方式，因此提前转置可以不计入算法运行时间中（测试表明作一次转置耗时约0.02s）。

- 遍历顺序：j-i-k

  > 在已经改变过存储方式后，对于两个乘子矩阵而言，都是按相同行下标k连续访问的，因此最内层循环应该遍历k，以最大限度利用空间局部性。
  >
  > 现讨论外两层循环的i和j。在两个乘子部分，i和j是地位相等的，只有在结果部分，i对应行而j对应列。直觉上来说，现遍历i再遍历j会对结果矩阵C[i][j]有空间局部性，但在实际尝试中发现似乎不是这样。不论是否开启编译优化，都是按列优先的方式访问C（即外j内i）更快。估计是写操作比较特殊，无法充分利用缓存线的缘故（可能使用非写或写穿策略，导致频繁IO的性能下降）。

- 延迟写：临时变量cc

  > 观察矩阵C中一个元素的计算过程：先初始化为0，再循环对内存或缓存中的对应位置作累加，需要对同一内存（或缓存）位置反复读写。如果我们使用一个临时变量cc来存储对应元素，可能使它保持在寄存器中，进而最大限度地利用它的空间局部性。即，先初始化临时变量cc，然后在cc上累加矩阵乘法的各项，最后把cc写到矩阵C的对应位置上。

- 循环展开：四路

  > 基于上述讨论，我们得出了计算矩阵乘法的三重循环每次计算一个元素的一项加数的最合理写法。然而三重循环可以作适当的修改，每次多计算几项，而在某一维度上循环次数减少。具体来说，就是在每次遍历k时计算C的连续四项元素:

```
double aik = a->data[i][k];
cc[0] += aik * b->data[j][k];
cc[1] += aik * b->data[j + 1][k];
cc[2] += aik * b->data[j + 2][k];
cc[3] += aik * b->data[j + 3][k];
```

  > 这会带来两个好处：

  1. 次要：合并多个循环节，减少循环跳转频率，增加流水线利用率。在本例中，这会将跳转频率减少到1/4。
  2. 主要：部分改变遍历顺序，进一步利用数据局部性。在本例中，可以一次计算矩阵4项，进一步利用C的空间局部性和A的时间局部性。

- 上述因素分别带来的性能提升忘记记录了。。。

8. 计时模块

- Surprise!

  > 在尝试调配负载的过程中，我发现这个计时模块有点疏漏：可以利用这个计时模块的非同步性把时间偷了。
  >
  > 计时方式假设了编程者是忠实作负载均衡的，各个进程用时基本相等，因此只计算了主进程在attention模块的时间。然而，经过有意的设计，主进程可以不安排负载，这样反馈时间就约等于零，即使其他进程仍处于工作之中。由于reduce部分作了集合通信而包含隐式同步，还能保证结果的正确性。

- 校正

  > 为了把计时部分变严格，应该对运行时间变量也作一次**MAX的Allreduce**，或者在计时部分的最后加上**显式的栅栏同步**。这两种方法分别对应两种理解方式：并行计算的是按最后一个进程计算完成为标志，还是按所有进程都知道计算完成为标志？

> 假设每个进程的运行时长是一个随机变量，那么理论上前者比修改之前期望有所增加，实际测得并不显著，即随机变量方差并不大。而后者比前者多一次Allreduce所耗时间，实际测得时间明显增加（在高度优化的情况下），即多节点间通信成本更大。

4. 最终策略

- 使用16个进程并行计算，这16个进程分布在四个节点的各四个核心上，每个进程单线程执行。运行时间作MAX的Allreduce
- 将Q矩阵按行进行划分，每个进程使用Q的一组行向量进行计算，最终得到结果的一组行向量（拼凑起来就是总体结果）
- 编译选项-Ofast和-march=native
- 使用转置后的矩阵乘法
- 遍历顺序j-i-k
- 类似写回的延迟写机制
- 四路循环展开

以下是运行一次脚本，可以看到加速比大约是640倍。

```
-----STANDARD-----
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698808
Correct! Time: 83.2920932770
-----CHECK-----
INPUT1
Matrix size: 8 x 5
Ans: 21.2349314776
Your answer: 21.2349314775
Correct! Time: 0.0000517368
INPUT2
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698837
Correct! Time: 0.1301429272

-----TEST TIME by input2 * 5-----
1
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698837
Correct! Time: 0.1291923523
2
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698836
Correct! Time: 0.1285407543
3
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698837
Correct! Time: 0.1285209656
4
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698837
Correct! Time: 0.1279060841
```

```
5
Matrix size: 2048 x 1024
Ans: 1048568.4615699096
Your answer: 1048568.4615698836
Correct! Time: 0.1292662621
```

5. 结果复现

- 文件说明
  - 原始代码：attention.cpp
  - 优化代码：attention_try.cpp
  - 编译脚本：Makefile
  - 运行脚本：run.slurm
  - 测试输入：input1.in & input2.in
  - 样例输出：job_*.out
  - 实验报告：README.md
- 编译：编译优化前后的程序

```
make clean
make
```

- 测试：正确性验证，并测定了优化前后的运行时长

```
sbatch run.slurm
```

> 注意：测试脚本run.slurm中指定了cu05,cu06,cu07,cu08四个节点，在实际测试中可以根据集群的实际负载情况选择空载节点