并行计算 II 2023 春季第一次作业：OpenMP 优化 Bellman-Ford 算法

0. 概要

本次作业针对Bellman-Ford算法使用OpenMP进行并行优化加速，最终加速了约27倍，最终源代码见persistent_T.cpp，编译脚本使用Makefile，运行脚本使用run.slurm，其中一次的输出结果可以参考job_211795_cu06.out。
具体的过程见第4部分，最终的策略见第5部分。

1. Bellman-Ford 算法

单源最短路径，以0点为起点，通过邻接矩阵更新距离向量
输入：邻接矩阵w
输出：距离向量d
算法：
    初始化：d[0]=0
    状态转移：d[x] = min(d[x], d[y]+w[y][x])
    终止条件：在某次迭代后没有发生更新，或已迭代了n-1轮。
        此时需要多迭代一轮，判定负权。

2. 串行计算

见serial.cpp

3. 并行计算

- 基本分析

> 状态转移： dist[v] = min(dist[v], dist[u]+mat[u][v])
> 基于上述状态转移过程，各状态可以独立计算。但必须注意：
>
> 1. 最外层循环即迭代轮数不可作并行，因为这是顺序依赖的，而两层内层循环则可以**选择性地作并行**。如何设置并行区？可以在每轮迭代时开辟并行区，也可以使用持久化的并行区。
> 2. 算法的一个终止条件是基于最短路径长度不超过n，这要求我们必须保证遍历过路径长为n的所有路径，对变量之间限定了一定的依赖关系。如果是完全独立地计算，可能某一点在计算了n步后其他点都还未开始，这将导致不能给出正确的结果。因此，在每轮迭代后需要作**栅栏同步**，此外，对has_change的共享也需要同步。
> 3. 对串行计算而言，每轮迭代更新的内层两重循环——工作划分并无显著区别。而对并行计算而言，**工作划分顺序是有意义的**。
> 4. 恰当**控制缓存**可以减少访存部分的代价，以在并行加速之上额外提升运行速度。

4. 因素比较

- 流程

```
# 记得在编译时添加选项 -fopenmp
make
# 记得在脚本中指定偏好节点
sbatch run.slurm
```

- 串行计算

  见serial.cpp。平均时间约0.24s

- 核心数和线程数

  1. 单核

     > 平均时间约0.3s，比串行计算还慢。因为在一个核心上多线程计算实际上还是串行执行的，还需要额外调度的时间。

  2. 多核

```
# 登录节点
sinfo --Node --long
scontrol show node
# 计算节点
lscpu
```

     > 在登录节点使用sinfo命令和scontrol命令可以查看各节点粗略信息。
     > 在计算节点使用slurm脚本获取该节点详细信息
     >
     > 线程数最好与硬件核心数匹配，这样能最大限度地并行化，考虑包括：
     >
     > 1. 是否支持超线程？若是，则一个核心可以运行多个线程以增加并行度（但达不到多核的并行度）。数院集群使用的CPU是E5-2650，双路12核单线程。
     > 2. 一个节点的核心是单路还是多路？如果是多路，需要权衡访存一致性和硬件资源的优劣：各路之间不共享缓存但共享内存，对同一片数据并行操作会增加延迟。而增加路数可以使计算资源加倍。本问题是访存密集型的，对访存一致性要求比较高，故只使用一路12核CPU。
     > 3. 虽然每个节点的CPU数基本都是24核，但在提交批处理脚本时不允许申请这么多个核心。实验中，申请12核的性能不太稳定，这是由于默认申请的节点cu01还有其他事务共享计算资源而发生调度，所以在性能下降。因此，后续在作业脚本命令中用"-w cuXX"指定节点cuXX 。

  经过实验，在核心数和线程数均为12时，速度最快。

- 并行划分

1. 工作分配

   在一轮迭代中：对于两层循环，在并行划分时可以选择针对内层或者外层作分配。

```
for (int v = tid; v < n; v += nt)   // 分配
{
    // 从内层提取的复用片段
    for (int u = 0; u < n; u ++)    // 完全遍历
    {
        dist[v] = min(dist[v], dist[u]+mat[u][v]);
    }
}
```

   > 如果两层循环是直接嵌套的，那么分配方式区别不大。
   > 不过，如果内层循环（工作集）中存在复用的代码段，则可以提取到外层一次性完成，以减少重复执行次数。
   > 只要缓存能存得下， 就尽可能地使用更大的工作集，以充分节约重复操作。

   因此，应该在外层作分配，而使内层完全循环。

2. 遍历方式

   > 状态转移： dist[v] = min(dist[v], dist[u]+mat[u][v])

   1. 终点优先

      > 每个线程一次对一个v遍历所有u。
      > 好处是每个线程每次独立维护自己的可变变量dist[v]。

   2. 途径点优先（X）

      > 每个线程一次对一个u遍历所有v。
      > 好处是每个线程每次重复访问同一个只读变量dist[u]。
      > 这里产生了一个问题：各线程共同修改dist[v]，如果不加锁会有数据竞争，如何保证结果正确性？
      > 事实上，dist[v]发生数据竞争时，有可能最终写入的不是最优值。但是只进行n轮迭代的要求是在每一轮都得出路径长度至多为i的最优路径，所以得到次优解违反了状态转移方程的条件，结果可能**不正确**。

   3. 循环展开

      > 在一轮迭代中，同时对终点和途径点作并行，最简单的方法就是使用collapse从句。
      > 好处是更高的视角实现更平均的循环分配（负载更均衡），但问题是这样就不支持前述的数据复用了。

   经过实验，循环展开的速度更快，但由于这样不能在两层循环间加入优化。为了更精细化调节，使用终点优先方式。

3. 调度策略（？）

> 如果一层循环遍历从1到n需要作并行任务划分，有两种常见静态调度策略：round-robin和range。
> 理论上说，使用range策略可以让每个线程连续访问一块内存，能有效利用空间局部性（缓存线）。
> 但是实际上的测试结果：round-robin效果更好。
> **没有思考出一个合理的解释**

结论：round-robin的效果更好。

充分尝试各种划分-遍历策略，发现最优的情况可以把并行算法加速至约**4-5倍**。

- 临时或持久的并行区

> 最简单的做法是每轮迭代都开辟一个并行区，并行区内划分循环，但每轮都需要初始化并行区，这可能引入额外的初始化开销。
> 可以尝试使用持久化的并行区，对内层循环作任务分配，在不同轮次间作一次栅栏同步即可。

经过实验，持久化的并行区效率更高。

- 缓存

> 鉴于一个线程经常访问同一个变量，因此可以对重复访问的部分作缓存优化。虽然主动缓存也需要额外开销，但在访问次数较多时还是能够加速。
> 可能复用的变量包括：has_change, dist[v], dist[u], mat[u][v]。
> has_change缓存：每个线程可以维护一个私有变量local_has_change，最后再作规约，以减小维护共享变量一致性的代价。
> mat[u][v]批量缓存：可以转置矩阵，故不论是按u还是按v遍历都可以做缓存。
> dist[u]缓存：这要求任务按u进行划分，根据前述讨论，这种划分影响了结果的正确性，不能使用。
> dist[v]缓存：这要求任务按v进行划分，每个线程独立维护各自的v，以保证数据安全性。缓存减少了对更外层存储器(L3cache)写的次数，但由于是条件写，次数可能不多。
> dist批量缓存：二者均不成立。对dist[v]的批量缓存利用率不高，因为每个dist[v]未必发生修改；对dist[u]的批量缓存会解除原有“非规则的”数据依赖，使每轮迭代更新量减少，而对只读空间，memcpy拷贝带来的时间节约并不多。
> 此外，由于first touch原则，在分配共享内存后，还应该在各线程上分别赋初值，以抵消内存的延迟分配。
> 实验发现，对has_change，mat[u][v]和dist[v]作缓存都可以有效加速。

基于持久化的并行区，使用局部空间和临时变量可以额外加速大约**2倍**。

- 存储方式

> 默认的存储方式是邻接矩阵mat[u][v]。此外，还可以存储转置矩阵和邻接表。
> 转置矩阵matT[v][u]的应用场景是按批量按列访问时可以一次缓存一组。
> 邻接表table[u][index]的应用场景是当图较稀疏时，可以减少对空边的无效访问，但要额外保存v作为存储代价。在本问题中，尽管额外的空间代价不至于造成缓存满溢，但图是稠密的，预计使用邻接表的效果并不显著，不作为主要优化思路。

最终方案中选用的是按列访问，故使用转置矩阵。

- 编译选项

> 在C++代码中，无法显式控制缓存和寄存器的使用方式。前述缓存相关的讨论我只能以局部空间和临时变量的方式实现（显式指定保存到一个新的位置），以**期望**其进入缓存。而编译器层面则可以优化产生的汇编代码中寄存器的使用方式，给出更紧凑的流水线，以及其他汇编层面的优化。 实验发现，-O2选项即有显著优化，而-O3和-Ofast没有进一步优化。

使用编译优化可以再加速大约**3-4倍**（果然编译优化还是强大）。

- 负权回路检测

> 有必要检测负权回路时，已迭代了n-1轮，而检测本身只需要1轮迭代，加速的比例本身就不大。

因此，对负权回路检测部分，只使用简单的并行划分就足够了。

- 正确性检查

> check.cpp代码将输出output.txt与标准输出（串行结果）output_std.txt进行比对

在运行脚本中使用./check计算正确性，各策略结果均正确
为了快速验证正确性，只需在运行脚本run.slurm中修改target变量的值即可

- 时长比较

```
serial
Time(s): 0.242653
collapse
Time(s): 0.015215
PASSED
destination
Time(s): 0.019752
PASSED
cache
Time(s): 0.010194
PASSED

FINAL:
persistent_T
Time(s): 0.008852
PASSED
```

> 以上是执行一次脚本所反馈的结果（在job_211795_cu06.out中）：
> serial是串行方法，其他的都是并行方法，且作了编译优化。
> destination是固定终点遍历途径点的策略，collapse是将两层循环统筹分配的策略。
> cache引入了持久化和缓存机制，而perisitent_T则对缓存作了进一步优化。

5. 总结

- 最终方案
  直观上看，上述几种因素大致是**正交**的，所以最自然的方案是在每种因素上选择最优方案，组合起来即可。因此，最终的代码**persistent_T.cpp**使用了如下选项：

1. 线程数为12。在一轮迭代中，每个线程负责整个mat[u][v]矩阵中1/12行或列的遍历，即每轮迭代的任务负载N^2/12
2. 外层循环作划分，内层循环完全遍历。对应地，使用转置矩阵
3. 外层循环遍历v（终点），内层循环遍历u（途径点）
4. 调度策略为round-robin
5. 使用局部空间和临时变量“引导”缓存和寄存器的使用
6. 对共享内存并行赋初值以抵消内存延迟分配
7. 编译语句中使用-Ofast选项进行优化

```
g++ -std=c++11 -o final persistent_T.cpp  -Ofast -fopenmp
```

- 测试方法

```
make clean
make
sbatch run.slurm
```

- 优化效果
  在前一节的测试结果中，取serial和persistent_T的计时结果进行比较。

```
serial
Time(s): 0.242653

FINAL:
persistent_T
Time(s): 0.008852
```

可以算出最终加速了约**27倍**。大体上来说，各因素是正交的，即加速比可以作连乘（但有时某些因素优化过深，就可能对另一因素的优化不显著）。