并行计算 实验二

PB19071501 李平治

I. 问题描述

输入一个带非负边权的图、一个源节点，求该节点到图中所有节点的最短路径长度，参考现有的并行单源点最短路径算法。

II. 算法设计

i. 问题分析

SSSP问题：给定图$G=(V,E)$，$|V|=n$，$|E|=m$，以及一个源结点s，所有节点v到达s的最短路径距离$dist(v,s)$，若不可达，记为$dist(v,s)=\infty$。

该问题最重要的算法是Dijkstra[1]算法，其主要思想是松弛操作(relax)，即从源顶点开始，不断更新边来松弛源顶点到其他顶点的距离。但Dijkstra对松弛顺序的高度依赖使得它作为一个纯顺序方法，难以被并行化。

```
function Dijkstra(Graph, source):
    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

Meyer等人提出的Delta-stepping[2]算法，基本上是一种Dijkstra算法的近似桶实现。它维护一个桶数组(大小取决于最大边长和参数Δ)，每个桶中储存节点集合，并行地对桶中节点进行原子性松弛。

然而，对桶的维护消耗了大量同步时间。Zhang等人提出的桶融合技术[3]，基于融合处理相同桶的连续轮次的思想，大大减小了同步开销。

ii.算法描述

线性Delta-stepping算法：

Algorithm 1 Pseudocode of the Δ-stepping algorithm.

```
function Δ-Stepping(V,E,c,s,Δ):

  for each vertex v in V:
    heavy[v] ← {(v,w) ∈ E : c(v,w) > Δ}
    light[v] ← {(v,w) ∈ E : c(v,w) <= Δ}
    tent[v] ← ∞
  end for
  relax(s,0)
  i ← 0

  while B ≠ Ø:
    S ← Ø
    while B[i] ≠ Ø:
      Req ← {(w,tent(v)+c(v,w)) : v ∈ B[i] and
                                  (v,w) ∈ light[v]}
      S ← S ∪ B[i]
      B[i] ← Ø
      for each (w,d) ∈ Req: relax(w,d)
    end while
    Req ← {(w,tent(v)+c(v,w)) : v ∈ S and
                                (v,w) ∈ heavy[v]}
    for each (w,d) ∈ Req: relax(w,d)
    i ← i+1
  end while
  return tent[]
end function
```

Algorithm 2 Pseudocode of the auxiliary relax function.

```
function relax(w,d):
  if d<tent[w]
    tent[w] ← d
    B[⌊tent[w]/Δ⌋] ← B[⌊tent[w]/Δ⌋] \ {w}
    B[⌊d/Δ⌋] ← B[⌊d/Δ⌋] ∪ {w}
  end if
end function
```

带有bucket fusion技术的并行delta-stepping算法:

```
Dist = {∞, . . . , ∞}                                          ▷ Length |V| array
procedure SSSP WITH Δ-STEPPING(Graph G, Δ, startV)
    B = new ThreadLocalBuckets(Dist, Δ, startV);
    for threadID : threads do
        B.append(new LocalBucket());
    Dist[startV] = 0
    while ¬empty B do
        minBucket = B.getMinBucket()
        parallel for threadID : threads do
            for src : minBucket.getVertices(threadID) do
                for e : G.getOutEdge[src] do
                    Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
                    B[threadID].updateBucket(e.dst, ⌊Dist[e.dst]/Δ⌋)
            while B[threadID].currentLocalBucket() is not empty  do
                currentLocalBucket = B[threadID].currentLocalBucket()
                if currentLocalBucket.size() < threshold then
                    for src : currentLocalBucket do
                        for e : G.getOutEdge[src] do
                            Dist[e.dst] = min(Dist[e.dst], Dist[src] + e.weight)
                            B[threadID].updateBucket(e.dst, ⌊Dist[e.dst]/Δ⌋)
                else break
```

Figure 7. Δ-stepping for single-source shortest paths with the eager bucket update approach and the bucket fusion optimization.

III. 实验评测

i.实验配置

本次实验在服务器上运行，相关配置如下:

- **24v**CPUs (Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz)
- MemTotal: 188GB
- Linux version 3.10.0-1160.el7.x86_64
- **g++** (conda-forge gcc 11.2.0-16) 11.2.0
- 并行库 **intel-openmp** 2022.0.1

图数据使用随机生成的全连通图（$|E| = \frac{|V|(|V|+1)}{2}$），顶点数$|V| = 10000$，边权重采用均匀分布产生

ii.实验结果

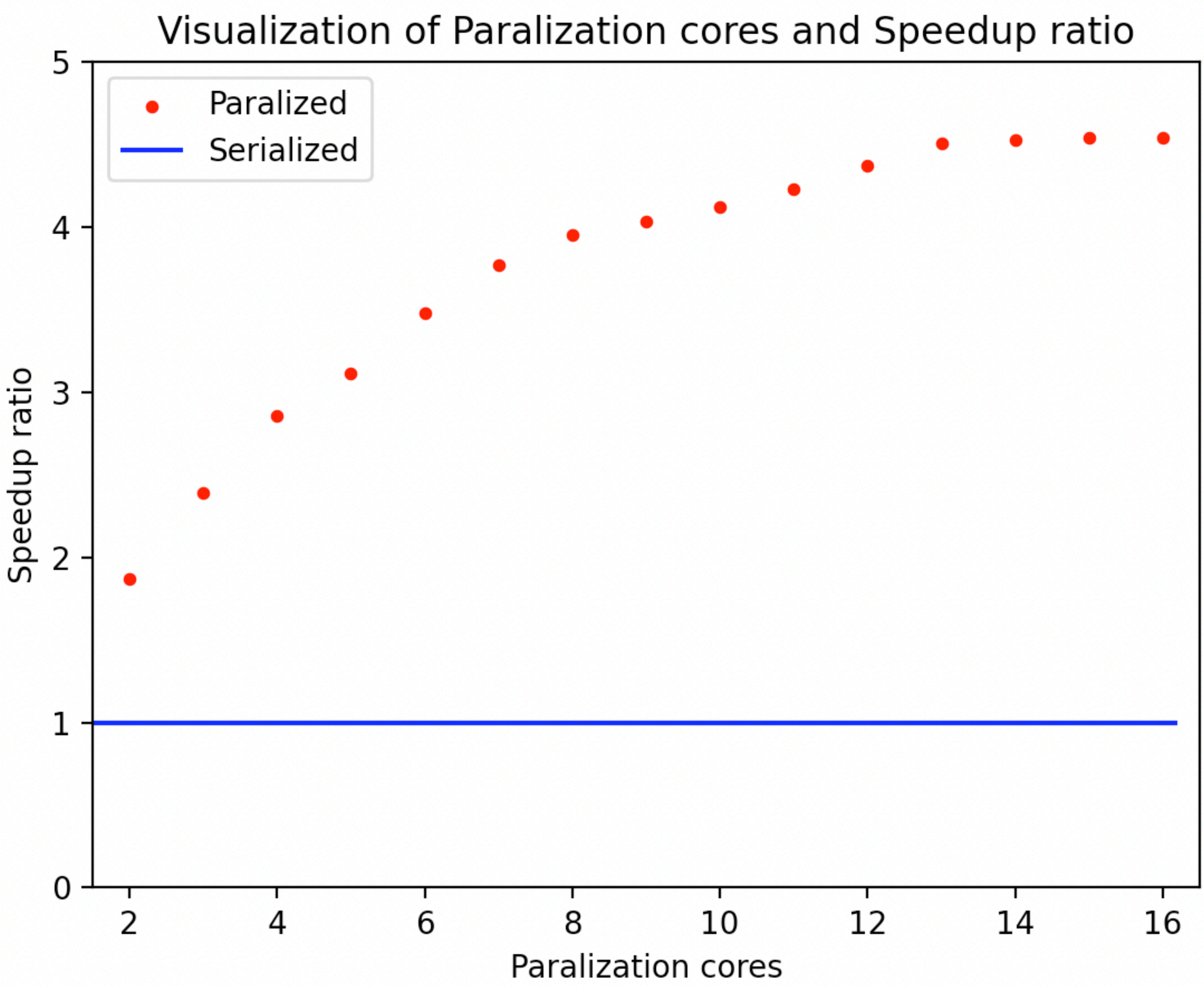
a. 正确性验证

将三种算法(Dijkstra, 线性delta-stepping, 并行桶融合delta-stepping)相互对比输出结果，均相同，说明结果正确

b.加速比分析

选择处理器核数2-16的并行delta- stepping和Dijkstra、串行delta- stepping进行分析:

算法/并行处理器核数	程序运行时间/s	加速比
Dijkstra	1.59	None
串行Delta-stepping	4.656	None
2	2.490	1.87
3	1.948	2.39
4	1.628	2.86
5	1.497	3.11
6	1.338	3.48
7	1.235	3.77
8	1.178	3.95
9	1.155	4.03
10	1.130	4.12
11	1.101	4.23
12	1.066	4.37
13	1.033	4.51
14	1.029	4.52
15	1.026	4.54
16	1.026	4.54



可以看出，尽管没有取得线性加速比，在并行数不高的情况下获得了非常好的结果。

在并行数$N>4$之后，并行delta-stepping算法运行时间开始低于Dijkstra算法，推测原因为Dijkstra算法局部性非常好，在gcc optimize(3)和没有通信消耗的加成下，比低并行数的delta-stepping算法要快很多

在并行数$N\geq 8$之后，加速比提升逐渐趋于平缓，收益不高，因此可以选取$N=8$作为最佳参数

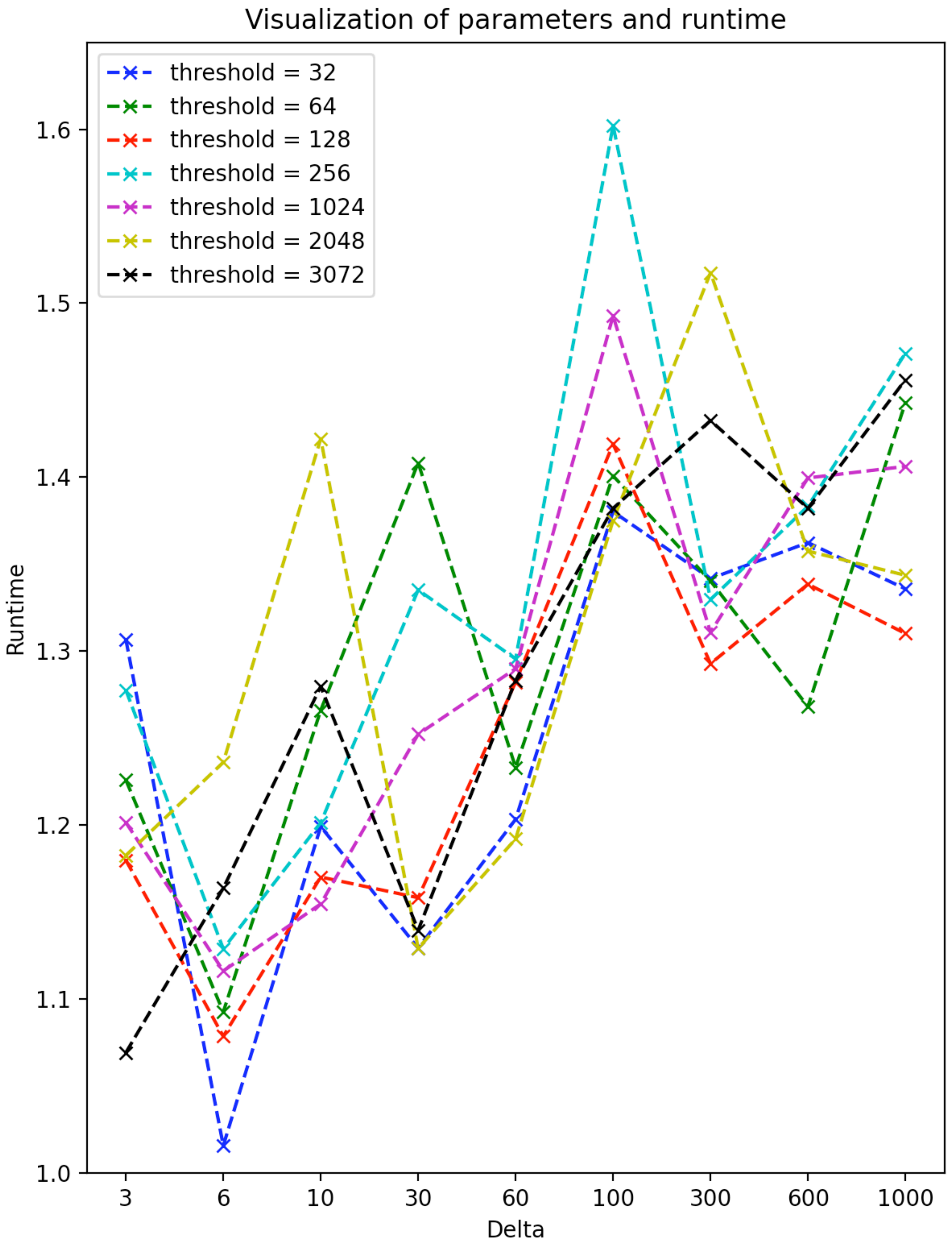
c.性能建模

Zhang et al.[3] 的论文中指出，参数Δ选择与图大小密切相关，在小图上的最佳值小于大图上的最佳值；此外，桶融合中的`THRESHOLD`的选取对于性能也至关重要，其直接影响到了负载均衡。

因此，本人选取以下参数组进行搜索实验，试图给出本次试验中图数据（10000结点全连接）的最佳参数组，并找出这些参数对性能的影响效果。

```
delta = [3, 6, 10, 30, 60, 100, 300, 600, 1000]
threshold = [32, 64, 128, 256, 1024, 2048, 3072]
```

实验结果如下：



网格搜索结果表明，本次实验图数据的最佳Delta和Threshold参数均集中在较小区间中，并且Delta参数对性能影响显著大于Threshold参数的影响。

推测原因是图数据为全连接图，并且边权重由均匀分布产生，几乎不存在负载均衡问题，因此性能对Threshold参数不敏感。

IV. 结论

本次实验考量了不同并行数下的并行算法效果，同时对比测量了并行算法参数选择的实际效果。结果证明并行算法存在边界收益衰弱的表现，以及参数选择与性能表现与数据高度相关。

参考

[1] Dijkstra, E. W., A note on two problems in connexion with graphs, Numerische Mathematik. 1: 269–27

[2] U. Meyer and P. Sanders, Delta-stepping: a parallelizable shortest path algorithm, Journal of Algorithms 49(1):114-152

[3] Zhang et al., Optimizing Ordered Graph Algorithms with GraphIt

附录

本次实验的代码都与本文档一同打包提交，其中`dijkstra.cpp`为Dijkstra串行算法(作对比用)，`delta_stepping.cpp`为$\Delta - Stepping$并行算法(作对比用)，`delta_stepping_parallel.cpp`为$\Delta - Stepping$并行算法，`graph_generator.py`为图数据生成脚本，`grid_search.py`为参数网格搜索脚本。