并行 Dijkstra 算法

一、 应用场景

近年来，云计算、大数据给互联网带来了巨大变革。但随着万物互联的飞速发展及广泛应用，网络边缘的设备数量急剧增加，数据规模不断扩大。传统的云计算虽然有高效的计算平台，但存在延迟过高、带宽不足、可用性、能耗及隐私问题。同时由于物联网实时计算、资源约束以及数据安全保护的需求，边缘式大数据处理应运而生，它以边缘计算模型为核心，面向网络边缘设备所产生的海量数据进行计算。边缘计算能够将原有云计算中心的部分或者全部计算任务迁移到数据源附近，极大缓解网络带宽和数据中心的压力，增强服务的响应能力，并能保证处理的实时性和数据的安全性。

美国韦恩州立大学的施巍松等人将边缘计算定义为 :边缘计算是指在网络边缘执行的一种新型计算模式。边缘计算中边缘的下行数据表示云服务，上行数据表示万物互联服务，而边缘计算的边缘是指从数据到云计算中心路径之间的任意计算和网络资源。

在边缘计算场景中，总的架构为云-边-端三层架构，如图 1 所示。在边缘层的边缘服务器 A 可能需要与在同一网络区域（灰色）的边缘服务器 B 进行通信，但是他们之间互不相连，所以需要通过其他服务器进行消息转发。整个网络区域不同服务器之前的通信代价不同，如何在 A、B 之间选择代价最小的通信路径？此问题映射成一个有权图的单源最短路径问题，典型的算法是 Dijkstra 算法。另外，如果这是一个大的网路区域，可能存在多个网络节点，边缘服务器的计算任务量可能会很大，于是可以通过其周围的几个边缘节点（灰色层中虚线所示）进行并行计算，分担其计算量。

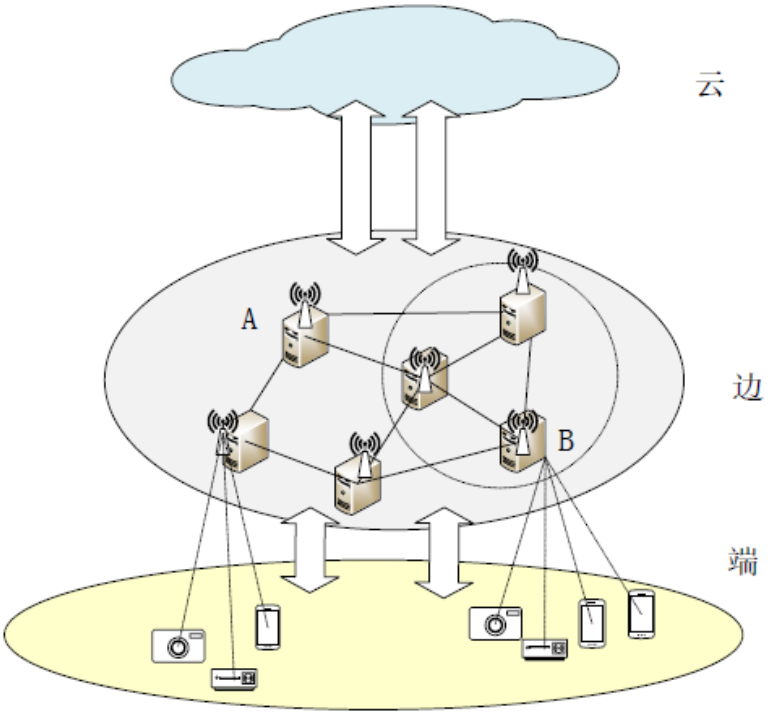

图 1 边缘计算架构

二、 并行思路

1. 串行 Dijkstra 算法

串行 Dijkstra 算法
输入：邻接矩阵
输出：单源节点 v0 到其他节点的最短路径值

```
1   const int   MAXINT =  ∞;
2   const int MAXNUM   ;                                    //节点数
3   int dist[MAXNUM]; prev[MAXNUM];                         //存放最短距离和前继
4   int A[MAXUNM][MAXNUM];                                  //邻接矩阵
5
6   void Dijkstra(int v0)                                   //从 v0 开始
7   {
8           bool S[MAXNUM];                                 // 判断是否已存入该点到 S 集合中
9           int n=MAXNUM;
10          for(int i=1; i<=n; ++i)
11          {
12                dist[i] = A[v0][i];
13                S[i] = false;                                     // 初始都未用过该点
14           }
15          dist[v0] = 0;
16          S[v0] = true;                                   //初始化
17          for(int i=2; i<=n; i++)   {
18                int u = v0;                               // 找出当前未使用的点 j 的 dist[j]最小值
19                for(int j=1; j<=n; ++j)
20                    if((!S[j]) && dist[j]<mindist) {
21                          u = j;                    // u 保存当前邻接点中距离最小的点的号码
22                          mindist = dist[j];
23                    }
24                S[u] = true;
25                for(int j=1; j<=n; j++)
26                     if((!S[j]) && A[u][j]<MAXINT) {
27                          if(dist[u] + A[u][j] < dist[j]) { //通过新加入的 u 找到离 v0 点更短的
28                               dist[j] = dist[u] + A[u][j];       //更新 dist
29                               prev[j] = u; }                          //记录前驱顶点
30                      }
31           }
32  }
```

串行的 dijkstra 算法有两层 N 次循环，串行的总复杂度为 $\mathrm{O}(N^2)$

2. 并行方式及复杂度分析

串行的 Dijkstra 算法有两层 N 次循环，第一层循环必须得按次序执行，不可并行。
那么考虑将第二层循环并行，第二层循环有两个 N 次的循环：

第一次循环是求 dist 数组的最小值，复杂度为 $O(N)$；

第二次循环是更新 dist 数组的值，复杂度也是 $O(N)$。

那么必须将这两次循环都并行才可以降低复杂度。

2.1 求 dist 数组最小值的并行方法

假设有 p 个处理器，N 个顶点。

1. 给每个处理器分配 N/p 个顶点，每个处理器求出局部的最小值，复杂度为 $O(\frac{N}{p})$。

2. 后一半的处理器将自己的最小值发送给第前 p/2 个处理器。前一半处理器接收到传来的值后，与自己局部的最小值比较，作为新值。继续循环，直到剩下一个处理器为止。
注意：若 p 为偶数，则进行一次合并后，剩 p/2 个处理器；
若 p 为奇数，则进行一次合并后，剩 p/2+1 个处理器。

并行后这一步复杂度为 $O(\frac{N}{p}+\log p)$

2.2 更新 dist 数组的并行方法

给每个处理器分配 N/p 个节点。设选取的 dist 最小的顶点为 u。

更新 dist 数组

for(j=0;j<mynum; j++)　　//仅计算自己部分的数据量更新 dist
{
if　dist[u]+w[u][j]<dist[j]
　then dist[j]= dist[u]+w[u][j];
}

并行后这一步的复杂度为 $O(\frac{N}{p})$

2.3 并行划分方式

设置一个单独的处理器用来的矩阵，采用行划分的并行方式，如图 2 所示。
有 P0-P4 五个处理器，有 12 个顶点，则对应一个 12*12 的对阵邻接矩阵。P0 用来读邻接矩阵，初始化其他处理器之后闲置；P1-P4 用来计算。
P1 负责计算 dist[0~2]
P2 负责计算 dist[3~5]
P3 负责计算 dist[6~8]
P4 负责计算 dist[9~11]
P1 需要矩阵 W[0~2][0~11]

P2 需要矩阵 W[3~5][0~11]
P3 需要矩阵 W[6~8][0~11]
P4 需要矩阵 W[9~11][0~11]

P1	0 1 2
P2	3 4 5
P3	6 7 8
P4	9 10 11

图 2 行划分

2.4 复杂度分析

并行的 dijkstra 算法复杂度为 $O(N*(\frac{N}{p}+\log p+\frac{N}{p}))=O(\frac{N^2}{p}+N\log p)$

$$E=\frac{S}{p}=\frac{T_s}{pT_p}=\frac{\Theta(N^2)}{p\Theta(\frac{N^2}{p}+N\log p)}=\frac{N}{N+p\log p}$$

如果想要达到代价最优，则 $E=\Theta(1)$，那么 $\frac{N}{p\log p}=\Theta(1)$，可以达到代价最优

当 $N>p\log p$，主要时间花费在求 dist 数组最小值，算法复杂度 $O(\frac{N^2}{p})$；

当 $N<p\log p$，主要时间花费在更新 dist 数组，算法复杂度 $O(N\log p)$；

总的来说比串行复杂度优化很多。

三、MPI 原理

1.1 MPI 基础

MPI(Message Passing Interface)是消息传递编程模型的标准，即信息传递接口，是用于跨节点通讯的基础软件环境。它提供让相关进程之间进行通信，同步等操作的 API。从编程的角度看，MPI 是一个库，可以被 C、C++、FORTRAN 绑定。

一个 MPI 程序包含若干个进程。每个 mpi 进程都运行一份相同的代码，进程的行为由通讯域(communication world)和该通讯域下的 id(rank id)所决定。MPI 的编程方式，是“一处代码，多处执行”。

1.2 MPI 基本接口

MPI_INIT：初始化。

MPI_FINALIZE：结束。

MPI_COMM_SIZE：总进程数。

MPI_COMM_RANK：当前进程标号。

MPI_SEND：发送消息。

MPI_RECV：接受消息。

1.3 MPI 常用命令

mpd & 启动本机的 mpi 守护进程

mpdboot: 启动集群 mpd 守护进程，在运行前必须开启每个节点上的 mpd 守护进程。

mpdtrace: 查看集群 mpd 守护进程。

mpdexit： 杀死指定节点 mpd 守护进程。

mpicc: MPI 程序编译命令。

mpiexec: MPI 程序运行命令，运行前必须开启 mpd 守护进程。

mpirun: MPI 程序快速执行命令，运行前不必运行 mpdboot 开启守护进程。

四、算法改进

4.1 所有处理器都参与计算

在之前的算法中，0 号处理器在读完邻接矩阵后就处于闲置状态。改进后，0 号处理器读完数据后也将参与计算中，不再闲置。

主要是在算法改进前，0 号处理器并不会初始化自己的 dist 和 bdist 矩阵，并且在执行主算法时会将要处理的顶点数置 0。改进后，0 号处理器会与其他处理器同时初始化自己的 dist 和 bdist，最后执行主算法。

4.2 负载均衡

4.2.1 未改进前

比如顶点数为 100，处理器个数为 4，且编号依次为 0，1，2，3。由于 0 号进程不分配顶点，则按原先的方法 ep=34（ep = nodenum/(group_size-1);）;分配的顶点数依次为 34，34，32，显然保证负载最均衡的分配应该为 34，33，33。

4.2.2 改进后

改进后的分配方法

ep = nodenum/group_size;
mod=nodenum%group_size;
if(my_rank<mod) mynum=ep+1; //保证任意两个处理器之间的任务数最多差 1
else mynum=ep;

改进后的算法可以保证任意两个处理器之间的任务数最多差 1。计算全局节点编号的方法如下：比如 i 号进程中的局部编号 j 的节点，全局编号 id 由如下方式计算：

改进后的计算全局顶点方法

```
If(i<mod){
        id=(ep+1)*i+j; }              //多分配了一个
    else
    {
        id=(ep+1)*mod+ep*(i-mod)+j;      //序号比 mod 小的进程顶点数都要 ep+1
    }
```

4.3 支持有向图的计算

之前的按行划分只适用于对称矩阵，在未改进前的算法中，num + W(j,index) < dist[j]实际上应该为 num + W(index，j) < dist[j]。在改进算法中，方法就是采用按列块划分，如图 3 所示，

P1 负责计算 dist[0~2]
P2 负责计算 dist[3~5]
P3 负责计算 dist[6~8]
P4 负责计算 dist[9~11]
P1 需要矩阵 W [0~11] [0~2]
P2 需要矩阵 W [0~11] [3~5]
P3 需要矩阵 W [0~11] [6~8]
P4 需要矩阵 W [0~11] [9~11]
这样发送的时候需要逐行发送，每行又分成 p 段发送。

P1	P2	P3	P4
0 1 2	3 4 5	6 7 8	9 10 11

图 3 列划分

五、实验设置

1. 实验环境

金山云：

2. 实验数据

本次实验测试了 100,1000 个节点，邻接矩阵为对称矩阵，权重为（0-100）随机生成（1-7.txt 和 10000-10007.txt）.还测试了节点数分别为 100 和 1000 的非对称矩阵（a.txt 和 1000a.txt）,权重仍然为（0-100）随机生成。

六、实验结果及分析

本次实验中，实验变量为顶点数分别为 100 和 1000，处理器数量分别为 4 和 8。对比算法分别为串行算法，并行算法，改进后的并行算法。其中实验数据一共分为 8 组随机生成的对称邻接矩阵。对于改进后的并行算法，单独设置一组有向图的矩阵。在实验结果中，我们分别列出了每个处理器的运行时间和最大的串行时间。下表 1.1，表 2.1，表 3.1 分别是顶点数=100，处理器=4（如果为并行方法）的各种方法对于每个数据集的处理时间，单位为秒，下表 1.2，表 2.2，表 3.2 分别是顶点数=1000，处理器=4（如果为并行方法）的各种方法对于每个数据集的处理时间，单位为秒。

表 1.1 顶点数=100，串行

数据	1.txt	2.txt	3.txt	4.txt	5.txt	6.txt	7.txt	8.txt	a.txt
时间	4.57	5.38	5.1	6.13	8.84	9.45	6.95	6.71	6.26

表 1.2 顶点数=1000，串行

数据	10000.txt	10001.txt	10002.txt	10003.txt	10004.txt	10005.txt	10006.txt	10007.txt	1000a.txt
时间	7.67	11.06	6.83	7.07	6.25	9.14	8.55	6.37	6.42

表 2.1 顶点数=100，处理器=4，并行

n=4	P1	P2	P3	P4	最大值
1.txt	0	3.47	3.47	3.47	3.47
2.txt	0	3.51	3.51	3.51	3.51
3.txt	0	3.44	3.44	3.44	3.44
4.txt	0	4.65	4.65	4.65	4.65
5.txt	0.03	3.74	3.75	3.73	3.75
6.txt	0	3.6	3.6	3.6	3.6
7.txt	7.28	7.28	0.06	7.2	7.28
8.txt	0	5.29	5.29	5.29	5.29

表 2.2 顶点数=1000，处理器=4，并行

n=4	P1	P2	P3	P4	最大值
10000.txt	0.17	7.11	7.2	7.2	7.2
10001.txt	0.18	8.37	8.38	8.36	8.38
10002.txt	0.16	6.92	6.93	6.92	6.93
10003.txt	0.17	11.06	11.07	11.08	11.08
10004.txt	0.16	7.41	7.41	7.42	7.42
10005.txt	0.16	10.19	10.2	10.19	10.2
10006.txt	0.16	6.93	6.94	6.93	6.94
10007.txt	0.16	6.83	6.84	6.84	6.84

表 3.1 顶点数=100，处理器=4，改进后并行算法

Enhance n=4	P1	P2	P3	P4	最大值
1.txt	2.461049	2.461048	2.461048	2.46105	2.46105
2.txt	2.821248	2.821249	2.821249	2.821249	2.821249
3.txt	3.925588	3.925586	3.925587	3.925587	3.925588
4.txt	4.000548	4.000549	4.000549	4.000548	4.000549
5.txt	5.605405	5.605407	5.605405	5.605405	5.605407
6.txt	3.400182	3.400182	3.400183	3.400182	3.400183
7.txt	2.940016	2.940015	2.940015	2.939979	2.940016
8.txt	3.843334	3.843332	3.843333	3.843328	3.843334
a.txt(有向图)	2.799736	2.799737	2.799738	2.799736	2.799738

表 3.2 顶点数=1000，处理器=4，改进后并行算法

Enhance n=4	P1	P2	P3	P4	最大值
10000.txt	4.36631	4.366319	4.366315	4.366313	4.366319
10001.txt	6.681018	6.681019	6.681019	6.68101	6.681019
10002.txt	4.567613	4.567624	4.567623	4.567621	4.567624
10003.txt	5.377331	5.377337	5.377342	5.377342	5.377342
10004.txt	4.891336	4.891345	4.891347	4.891347	4.891347
10005.txt	4.096704	4.096706	4.096706	4.096701	4.096706
10006.txt	4.579387	4.579386	4.579375	4.579386	4.579387
10007.txt	4.512378	4.512381	4.512397	4.512397	4.512397
1000a.txt	5.854842	5.854851	5.854851	5.854852	5.854852

从上表 1.1，表 2.1，表 3.1 可看出，并行算法的运行时间平均时间小于串行时间。特别地，可看到未改进前的并行算法中总有一个处理器的运行时间基本为 0，因为在此算法中，此处理器分配完数据后，就处于闲置状态，改进后的并行算法，运行时间在三种方法中最小，因为一个归结于其的平均分配，再加上多了一个处理器进行计算，故并行运行时间减少。

从表 1.1 和表 1.2 可看出，随着顶点数的增加，串行的时间增加的较为明显。从表 2.1 和表 2.2 看出，顶点数的增加无异于增加了处理器间的通信开销和计算量，从实验结果看此时的并行算法不如串行算法。而从表 3.1 和表 3.2，改进后的并行算法表现出得性能就很好了，由于处理器间分配均匀且所有处理器都参与算法，所以时间还是三者之间最佳。

下表 4，表 5 分别是顶点数=100，处理器=8 的并行方法对于每个数据集的处理时间。

表 4.1 顶点数=100，处理器=8，并行

n=8	P1	P2	P3	P4	P5	P6	P7	P8	最大值
1.txt	10.99	10.74	10.58	7.04	10.53	10.53	10.54	10.52	10.99
2.txt	6.61	9.3	9.21	8.83	9.12	8.94	9.07	8.99	9.3
3.txt	7	10.79	10.77	11.35	11.01	10.67	11.04	10.62	11.35
4.txt	9.5	9.33	9.22	6.45	9.27	9.64	9.25	9.58	9.64
5.txt	10.36	10.7	6.11	10.29	10.41	10.03	10.25	10.32	10.7
6.txt	6.91	10.2	10.15	10.06	10.28	10.27	10.08	10.13	10.28
7.txt	6.49	8.4	8.06	8.31	8.21	8.6	8.04	8.2	8.6
8.txt	7.76	10.27	10.7	10.48	10.71	10.69	10.31	10.5	10.71

表 4.2 顶点数=1000，处理器=8，并行

n=8	P1	P2	P3	P4	P5	P6	P7	P8	最大值
10000.txt	72	71	71	72	68	72	71	71	71
10001.txt	66	67	66	66	63	66	66	67	65.875
10002.txt	68	68	67	68	68	64	67	68	67.25
10003.txt	72	77	77	77	77	76	76	77	76.125
10004.txt	64	64	64	65	64	64	64	62	63.875
10005.txt	69	69	68	69	66	69	69	69	68.5
10006.txt	67	67	67	67	67	67	66	63	66.375
10007.txt	66	65	66	66	63	66	65	65	65.25

表 5.1 顶点数=100，处理器=8，改进后的并行方法

Enhance n=8	P1	P2	P3	P4	P5	P6	P7	P8	最大值
1.txt	14.48884	14.495821	14.48496	14.487982	14.495807	14.498812	14.478999	14.478974	14.49881
2.txt	13.828907	13.836891	13.822923	13.822931	13.833908	13.843833	13.823956	13.843903	13.8439
3.txt	14.556942	14.573899	14.545933	14.574887	14.56792	14.557974	14.567863	14.577816	14.57782
4.txt	15.200865	15.200996	15.210795	15.190953	15.20489	15.194939	15.219897	15.199806	15.2199
5.txt	16.601999	16.621816	16.612843	16.602933	16.613891	16.623814	16.617984	16.621864	16.62381
6.txt	15.079901	15.087881	15.087876	15.069956	15.068974	15.057932	15.069968	15.089858	15.08986
7.txt	14.455845	14.453982	14.445988	14.455902	14.455879	14.465844	14.433963	14.455863	14.46584
8.txt	14.591001	14.6019	14.591991	14.591946	14.601924	14.601868	14.601913	14.591997	14.60192
a.txt(有向图)	15.315831	15.319864	15.306034	15.323824	15.315864	15.315724	15.304896	15.305991	15.32382

表 5.1 顶点数=1000，处理器=8，改进后的并行方法

Enhance n=8	P1	P2	P3	P4	P5	P6	P7	P8	最大值
10000.txt	127.160145	127.176912	127.179812	127.167032	127.180854	127.161149	127.186828	127.169969	127.1868

1000 1.txt	127.91 5956	127.92 8897	127.92 5965	127.92 6008	127.91 6005	127.92 5916	127.91 5984	127.92 5912	127. 9289
1000 2.txt	125.26 9971	125.26 2996	125.25 9991	125.28 086	125.28 0881	125.28 3973	125.26 9943	125.26 3133	125. 284
1000 3.txt	123.37 7953	123.40 5851	123.39 9752	123.39 9916	123.40 9815	123.41 6797	123.39 8972	123.39 6008	123. 4168
1000 4.txt	120.04 9805	120.05 0034	120.03 9041	120.03 0944	120.04 8001	120.05 9854	120.06 9712	120.04 9812	120. 0697
1000 5.txt	122.13 4997	122.12 4148	122.14 5854	122.13 6108	122.14 6012	122.14 5858	122.14 5933	122.13 5976	122. 146
1000 6.txt	123.00 9975	123.01 985	123.01 9901	123.01 9835	123.04 6854	123.01 9946	123.01 4992	123.01 6996	123. 0469
1000 7.txt	129.23 2758	129.24 005	129.24 0051	129.24 0033	129.23 2823	129.23 2861	129.23 0196	129.22 9899	129. 2401
1000 a.txt	158.09 2343	158.09 2359	158.09 2633	158.09 373	158.09 5827	158.08 9417	158.09 1602	158.08 1338	158. 0958

从表 4.1、表.5.1 可看出，当处理器增加时，顶点数不变，通信开销相对占比增加，所以并行算法相对串行时的时间增加。其中，改进后的算法时间最多，这是由于为了保证每个处理器分得的顶点数均衡，增加了计算全局顶点数的开销，相比未改进的并行算法，时间增加的更多。

表 4.1 和表 4.2 相比，表 5.1 和表 5.2 相比，随着顶点数增加，时间几乎呈指数级上升。也说明了顶点数的增加使处理器间的通信开销剧增，其中改进后的算法相对于未改进的并行方法增加的更多，也间接证明了计算全局顶点数时的开销很大。

总结分析：

表 6 是整个实验的总表，表中并行方式的数据采用所有处理器中的最大值代表运行时间。总体来讲，1)并行算法的运行时间平均时间小于串行时间，其中改进后的并行算法，运行时间在三种方法中最小；2)当处理器增加时，顶点数不变，通信开销相对占比增加，所以并行算法的时间开销增加，甚至大于串行时间；3)当顶点数增加的时候，三种方式的处理时间均增加，处理器数目较小的时候并行效果较好，当处理器数据增加，并行时间呈指数级增加；4）对于有向图来讲，也符合上述三种变化。

表 6 并行 Dijkstra 算法总表

节点数	数据	串行	n=4	n=8	提升 n=4	提升 n=8
100	1.txt	4.57	3.47	10.99	2.46	14.5
	2.txt	5.38	3.51	9.3	2.82	13.84
	3.txt	5.1	3.44	11.35	3.93	14.58
	4.txt	6.13	4.65	9.64	4	15.22
	5.txt	8.84	3.75	10.7	5.61	16.62
	6.txt	9.45	3.6	10.28	3.4	15.09
	7.txt	6.95	7.28	8.6	2.94	14.47
	8.txt	6.71	5.29	10.71	3.84	14.6
	a.txt	6.26			2.8	15.32

1000	10000.txt	7.67	7.2	71	4.37	127.19
	10001.txt	11.06	8.38	65.88	6.68	127.93
	10002.txt	6.83	6.93	67.25	4.57	125.28
	10003.txt	7.07	11.08	76.13	5.38	123.42
	10004.txt	6.25	7.42	63.88	4.89	120.07
	10005.txt	9.14	10.2	68.5	4.1	122.15
	10006.txt	8.55	6.94	66.38	4.58	123.05
	10007.txt	6.37	6.84	65.25	4.51	129.24
	1000a.txt	6.42			158.1	5.85

七、展望

求局部最小值时，可用堆的思想，可以将该部分复杂度将低为 $O(\log(\frac{N}{p}))$。