A Algotirhm List

A.1 Pipeline Shuffle in Daemon-Agent Framework

Algorithm 1: Pipeline Shuffle - Daemon Side

Input: Data area pointer n, c, u, Computer Device com_dev

```
1  while In Iteration do
2      Block_Recv(agent, msg);
3      if msg = "ExchangeFinished" then
4          Rotate(n → c → u → n);
5          Send(agent, "RotateFinished");
6          if c contains contents to compute then
7              com_dev.Load(*c);
8              com_dev.Compute();
9              *c ← com_dev.data;
10             Send(agent, "ComputeFinished");
11         else
12             Send(agent, "ComputeAllFinished");
13             End_Iteration();
```

Algorithm 2: Pipeline Shuffle - Agent Side

Input: Data area pointer n, c, u, Upper system interface USI

```
1  *n ← USI.Download();
2  Send(daemon, "ExchangeFinished");
3  while In Iteration do
4      Block_Recv(daemon, msg);
5      if msg = "RotateFinished" then
6          upload, download = new Thread();
7          for Thread upload do
8              USI.Upload(*u);
9          for Thread download do
10             *n ← USI.Download();
11     else if msg = "ComputeFinished" then
12         if upload.isTerminated()
             download.isTerminated() then
13             Send(daemon, "ExchangeFinished");
14     else if msg = "ComputeAllFinished" then
15         if upload.isTerminated()
             download.isTerminated() then
16             End_Iteration();
```

The pipeline shuffle contains 2 parts of processing scenarios, which are held by daemon side and agent side. In one process cycle, agent uses 2 threads to separately fetch updated data from block u (Line 7-8 in Algorithm 2) and put new wait-for-compute data into block n (Line 9-10 in Algorithm 2), while daemon uses data in block c for computation (Line 6-10 in Algorithm 1). Since that this 3 processes involve different separate data buffers, there is no any I/O conflict among this 3 processes, and they can be executed concurrently. There are several message send/recv functions which are used in both daemon and agent side processes for sequence control (Line 4 in Algorithm 2, Line 2 in Algorithm 1, etc). When one process cycle finished (which means data transfers and computation are all finished), agent sends message (Line 13 in Algorithm 2) to daemon, and daemon shuffles 3 pointers to point to different data buffer: block n points to original block u, block c points to original block n, and block u points to original block c (Line 4 in Algorithm 1), and notifies agent to begin next cycle process (Line 5 in Algorithm 1). In this way, we have: 1) block u which agent uses to fetch data contains data in previous block c, which holds data taken part in computation in previous cycle, 2) block c contains data in previous block n put by agent which needs to be computed in this cycle, and 3) block n contains data in previous block u which has been uploaded, and the data buffer itself can be reused directly by new downloaded data, with no need of memory reallocation.

Also, some judgements (Line 11-13 in Algorithm 1, Line 14-16 in Algorithm 2, etc) are used to judge if cycle continues, and also help to manage the end conditions of current iteration, in order to help system to switch to other phase in overall process as well as keep the global correctness.

A.2 Lazy Uploading

Algorithm 3: Lazy Uploading

Input: Updated Dataset s, Global query queue gqq, Global data queue gdq

```
1  s_q ← s.GetQueriedEntity();
2  Send(gqq, s_q);
3  Wait for other agents;
4  s_u ← Find(gqq, s.GetUpdatedEntity());
5  Send(gdq, s_u);
6  Wait for other agents and upper system
     synchronization;
7  s.Update(Fetch(gdq, s_q));
```

After daemon finished its computation task in current iteration and all data are transferred back to agent with the help of pipeline shuffle mechanism, agent first generate a list called s_q which records the data entities that are needed for next iteration, and push it to global query queue (Line 1-3 in Algorithm 3). Also, agent searchs in global query queue and compared with updated entities it holds. Only those updated entities which appears in global query queue are be used to

construct a sub-dataset s_u, and this is updated to global data queue for upper system synchronization (Line 4-6 in Algorithm 3). Finally, agent collects needed synchronized data from global data queue and updates local cache (Line 7 in Algorithm 3), in order to prepare daemon fetch at local cache in next iteration.

Once a data entities are marked as updated by daemon, it keeps updated until it is queried and sent to global data queue. With this modification, even earlier updates for the same data entity can be used for current global synchronization, and thus synchronization issues can be avoided.

With this technique, data entities are cached in local caches and never trigger copies until they are needed by other distributed partitions, which eliminates unnecessary data upload. Also, data exchanges happen at the beginning of each iteration use data at local cache, which means those data entities that have not been updated can be fetched locally and redundant data transfers related to upper system can be skipped.

B Calculation for Block Size b

In our work, we found the size of a block has a profound effect over the parallelism performance. We assume that there is a dataset which contains D entities need to be processed in the current iteration. Also, agent divides the dataset into s blocks evenly, $b = \frac{D}{s}$. Let $T_n(b_i)$, $T_c(b_i)$, $T_u(b_i)$ be the process time of block b_i in Thread.Download, Thread.Compute and Thread.Upload, respectively. T_n and T_u are proportional to the data block size. We can estimate pipeline processing time T_{total}.

$$\begin{aligned} T_{total} &= T_n(b) + \max(T_n(b), T_c(b)) \\ &+ \sum_{i=2}^{s-1} \max(T_n(b), T_c(b), T_u(b)) \\ &+ \max(T_c(b), T_u(b)) + T_u(b) \end{aligned}$$

T_c refers to the time cost of Thread.Compute, and consists of calling computation devices, loading data to devices, and computation. Their corresponding time costs are represented by T_{call}, T_{comp}, and T_{copy}, respectively. The operation of calling devices takes constant time, while computation and data copying time are related to data size. So, $T_c(b_i)$ can be modeled as follows.

$$T_c(b_i) = T_{call} + T_{comp}(b_i) + T_{copy}(b_i)$$

When s increases, block size b decreases, so do T_n and T_u. But the total execution time T_{total} can probably increase as s is larger while T_c is never lower than $Tcall$, and probably decrease because b is smaller. Experiments show the U-turn trend of T_{total} when s varies. Thus, s and b should be deliberately configured for achieving fine-tuned system performance. We will try to calculate the value of b in order to provide optimization suggestion to overall system.

The calcucation follows two assumptions: 1) The whole dataset has d entities, and is divided into s blocks evenly, which have the size $b = \frac{d}{s}$. 2) we assume both T_n, T_u are directly proportional to the block size; and 3) In T_c shown in Equation 2, T_{call} has a fixed number, while T_{comp} and T_{copy} are also directly proportional to b. Thus, Equation 2 can be further simplied as follows:

$$\begin{aligned} T_{total} &= k_1 b + \max(k_1 b, a + k_2 b) \\ &+ \sum_{i=2}^{s-1} \max(k_1 b, a + k_2 b, k_3 b) \\ &+ \max(a + k_2 b, k_3 b) + k_3 b \end{aligned} \tag{2}$$

Follow this equation, we have discussions in 3 situations.

B.1 $k_1 b$ is the maximum value

Obviously, this situation stands up only if k_1 is the maximum in the 3 parameters.

In this situation, we first give the range of b. Simply we have:

$$k_1 b \geq a + k_2 b \Rightarrow b \geq \frac{a}{k_1 - k_2}$$

Thus, Equation 2 can be transformed into:

$$\begin{aligned} T_{total} &= s k_1 b + \max(a + k_2 b, k_3 b) + k_3 b \\ &= k_1 d + \max(a + k_2 b, k_3 b) + k_3 b \end{aligned}$$

Notice that a and k_i are all positive, both $\max(a + k_2 b, k_3 b)$ and $k_3 b$ increases when b increases. Thus, when $b = \frac{a}{k_1 - k_2}$, we have the minimum value of T_{total}:

$$\begin{aligned} T_{total} &= k_1 d + \max(a + \frac{a k_2}{k_1 - k_2}, \frac{a k_3}{k_1 - k_2}) + \frac{a k_3}{k_1 - k_2} \\ &= k_1 d + \max(\frac{a k_1}{k_1 - k_2}, \frac{a k_3}{k_1 - k_2}) + \frac{a k_3}{k_1 - k_2} \\ &= k_1 d + \frac{(k_1 + k_3) a}{k_1 - k_2} \end{aligned} \tag{3}$$

B.2 $(a + k_2 b)$ is the maximum value

It is a bit complicated to discuss this situation. First we have this equation (Notice that $s = \frac{d}{b}$):

$$\begin{aligned} T_{total} &= k_1 b + s(a + k_2 b) + k_3 b \\ &= (k_1 + k_3) b + k_2 d + \frac{a d}{b} \end{aligned} \tag{4}$$

In this equation, we can have the minimum T_{total} when $b = \sqrt{\frac{a d}{k_1 + k_3}}$ if it can be. Following this, we discuss T_{total} in 3 parts by classifing k_2.

k_2 is the minimum one. In this situation, both $(k_1 - k_2)$ and $(k_3 - k_2)$ are positive. Thus, we have:

$$a + k_2 b \geq \max(k_1, k_3) * b \Rightarrow b \leq \min(\frac{a}{k_1 - k_2}, \frac{a}{k_3 - k_2})$$

Assume that $k_1 \geq k_3$, we have $b \leq \frac{a}{k_1-k_2}$. Thus, we have the minimum T_{total} by using Equation 4:

$$T_{total} = \begin{cases} k_2 d + 2\sqrt{(k_1+k_3)ad}, & \frac{a}{k_1-k_2} \geq \sqrt{\frac{ad}{k_1+k_3}} \\ \frac{a(k_1+k_3)}{k_1-k_2} + k_2 d + (k_1-k_2)d, & \frac{a}{k_1-k_2} < \sqrt{\frac{ad}{k_1+k_3}} \end{cases}$$

Also, we have the minimum T_{total} when $k_3 \geq k_1$:

$$T_{total} = \begin{cases} k_2 d + 2\sqrt{(k_1+k_3)ad}, & \frac{a}{k_3-k_2} \geq \sqrt{\frac{ad}{k_1+k_3}} \\ \frac{a(k_1+k_3)}{k_3-k_2} + k_2 d + (k_3-k_2)d, & \frac{a}{k_3-k_2} < \sqrt{\frac{ad}{k_1+k_3}} \end{cases}$$

k_2 is the middle one. In this situation, we should notice the change of relationship in inequality when performing calculation.

For simplicity, we assume $k_3 \leq k_2 \leq k_1$. In this situation, $(k_1 - k_2)$ is positive, while $(k_3 - k_2)$ is negative. Thus, we can have $(b > 0)$:

$$\frac{a}{k_3-k_2} \leq b \leq \frac{a}{k_1-k_2} \Rightarrow b \leq \frac{a}{k_1-k_2} \tag{5}$$

Following this, we have the minimum value of T_{total} by using Equation 4:

$$T_{total} = \begin{cases} \frac{a(k_1+k_3)}{k_1-k_2} + k_2 d + (k_1-k_2)d, & \frac{a}{k_1-k_2} < \sqrt{\frac{ad}{k_1+k_3}} \\ k_2 d + 2\sqrt{(k_1+k_3)ad}, & other \end{cases} \tag{6}$$

Also we have the minimum value of T_{total} when $k_3 \geq k_2 \geq k_1$:

$$T_{total} = \begin{cases} \frac{a(k_1+k_3)}{k_3-k_2} + k_2 d + (k_3-k_2)d, & \frac{a}{k_3-k_2} < \sqrt{\frac{ad}{k_1+k_3}} \\ k_2 d + 2\sqrt{(k_1+k_3)ad}, & other \end{cases} \tag{7}$$

k_2 is the maximum one. In this situation, both $k_1 - k_2$ and $k_3 - k_2$ are negative. Since $b > 0$, b is also greater than $\frac{a}{k_1-k_2}$ and $\frac{a}{k_3-k_2}$. Thus, we simply have the minimum T_{total} when $b = \sqrt{\frac{ad}{k_1+k_3}}$:

$$T_{total} = k_2 d + 2\sqrt{(k_1+k_3)ad}$$

B.3 $k_3 b$ is the maximum value

Obviously, this situation stands up only if k_3 is the maximum in the 3 parameters.

Following the discussion in Equation 3, we simply have the conclusion that T_{total} has the minimum value $k_3 d + \frac{(k_1+k_3)a}{k_3-k_2}$ when $b = \frac{a}{k_3-k_2}$.

B.4 Discussion

Following the previous discussion, and the order of k_1, k_2 and k_3, we have 3 possible routes to calculate b.

k_1 is the maximum one: If $\frac{a}{k_1-k_2} \geq \sqrt{\frac{ad}{k_1+k_3}}$, $b = \sqrt{\frac{ad}{k_1+k_3}}$, and T_{total} have minimum value $k_2 d + 2\sqrt{(k_1+k_3)ad}$. Otherwise, $b = \frac{a}{k_1-k_2}$, and T_{total} have minimum value $k_1 d + \frac{(k_1+k_3)a}{k_1-k_2}$.

k_2 is the maximum one: $b = \sqrt{\frac{ad}{k_1+k_3}}$, and T_{total} have minimum value $k_2 d + 2\sqrt{(k_1+k_3)ad}$.

k_3 is the maximum one: If $\frac{a}{k_3-k_2} \geq \sqrt{\frac{ad}{k_1+k_3}}$, $b = \sqrt{\frac{ad}{k_1+k_3}}$, and T_{total} have minimum value $k_2 d + 2\sqrt{(k_1+k_3)ad}$. Otherwise, $b = \frac{a}{k_3-k_2}$, and T_{total} have minimum value $k_1 d + \frac{(k_1+k_3)a}{k_3-k_2}$.

Finally, we have b:

$$b_{opt} = \begin{cases} \frac{a}{k_1-k_2}, & \left(k_1 > \max(k_2,k_3), \text{ and } \frac{a}{k_1-k_2} < \sqrt{\frac{ad}{k_1+k_3}}\right) \\ \frac{a}{k_3-k_2}, & \left(k_3 > \max(k_2,k_1), \text{ and } \frac{a}{k_3-k_2} < \sqrt{\frac{ad}{k_1+k_3}}\right) \\ \sqrt{\frac{ad}{k_1+k_3}}, & other \end{cases} \tag{8}$$

And corresponding T_{total}:

$$T_{total_{min}} = \begin{cases} \frac{a(k_1+k_3)}{k_1-k_2} + k_1 d, & \left(k_1 > \max(k_2,k_3), \text{ and } \frac{a}{k_1-k_2} < \sqrt{\frac{ad}{k_1+k_3}}\right) \\ \frac{a(k_1+k_3)}{k_3-k_2} + k_3 d, & \left(k_3 > \max(k_2,k_1), \text{ and } \frac{a}{k_3-k_2} < \sqrt{\frac{ad}{k_1+k_3}}\right) \\ k_2 d + 2\sqrt{(k_1+k_3)ad}, & other \end{cases} \tag{9}$$

In real situation, since both s and b must be integer, we first calculate $s_{opt} = \frac{d}{b_{opt}}$, and choose 2 values s_{min} and $s_{max} = s_{min} + 1$, which make $s_{min} < s_{opt} \leq s_{max}$. After that, we calculate $b_1 = \lfloor s_{min} \rfloor$, $b_2 = \lceil s_{min} \rceil$, $b_3 = \lfloor s_{max} \rfloor$, and $b_4 = \lceil s_{max} \rceil$. Finally, we put this 4 values into Equation 2 and choose one which makes T_{total} be minimum.

We do some experiments to show the performance of block size selection mechanism. Figure 15 shows the performance when choosing different block sizes. For both LP and PageRank algorithms, we use the first iteration as the testing data. For SSSP algorithm, we use 6-th iteration as the testing data, since the computation workload is the maximum during the whole execution. We can find that when block size increases, iteration time cost first decreases, and then increases. We also give our estimated results following the analysis here for the

3 different algorithms. It shows that the time cost of our estimated block sizes are close to that of optimal block sizes, demonstrating good accuracy of our estimation model.

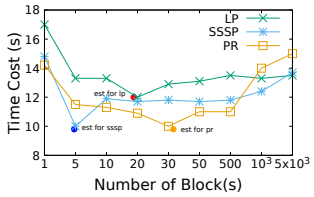

Figure 15. Block Size Selection

C Code Example

In this section, we show an example about graph algorithm implementation using our APIs. In particular, we compare the implementation difference between conventional distributed systems and our proposed middlewares, denoted by CPU- and GPU-based implementation, respectively[4].

The codes for CPU- and GPU-based implementation are shown in Sample Codes 1 and 2. The differs of both side of codes are highlighted in different colors, red for CPU-based code and green for GPU-based code. From the two samples, we can see the implementations are quite similar, and only lines 17-28 are different. For CPU-based implementation, this part is for invoking CPU computation. For GPU-implementation, this part is for transferring data to GPU and executing corresponding kernel functions. This way, the distributed graph algorithms for conventional distributed systems can be replanted to our middleware with small changes. More, algorithm engineers can access kernel function implementation for further optimization.

D Proofs for Workload Balancing

Proof. **(Proofs for Lemma 3.1)** First, if every d_j meets the condition, we have:

$$F(d_1, ..., d_m) = \max_{j=1}^{m}\{c_j \cdot \frac{\frac{1}{c_j}}{\sum_{j=1}^{m}\frac{1}{c_j}}D\} = \frac{D}{\sum_{j=1}^{m}\frac{1}{c_j}}$$

Second, we prove that for any possible d_j, we have $F \geq \frac{D}{\sum_{j=1}^{m}\frac{1}{c_j}}$.

We prove this assertion by contradiction. We assume it holds that $F = max_{j=1}^{m}(c_j d_j) < \frac{D}{\sum_{j=1}^{m}\frac{1}{c_j}}$. Then for every d_j, we have:

$$c_j d_j < \frac{D}{\sum_{j=1}^{m}\frac{1}{c_j}} \Rightarrow d_j < \frac{\frac{1}{c_j}}{\sum_{j=1}^{m}\frac{1}{c_j}}D \Rightarrow D = \sum_{j=1}^{m} d_j < \frac{\sum_{j=1}^{m}\frac{1}{c_j}}{\sum_{j=1}^{m}\frac{1}{c_j}}D = D$$

Here, contradiction occurs. Thus, function F reaches the minimum value $\frac{D}{\sum_{j=1}^{m}\frac{1}{c_j}}$, if and only if for all d_j, $d_j = \frac{\frac{1}{c_j}}{\sum_{j=1}^{m}\frac{1}{c_j}}D$. The lemma is hence proved. □

Proof. **(Proofs for Lemma 3.2)** Let $\frac{1}{c_*}$ be $max_{j\leq m}\frac{1}{c_j}$. Since $\frac{1}{c_*} \leq f$, we have:

$$\frac{d_*}{f} \leq c_* d_* \leq F' = \max_{j=1}^{m}(c_j d_j)$$

To make $F' = \frac{d_*}{f}$, all other $c_j d_j$ must be not greater than $\frac{d_*}{f}$. With the condition to make $\frac{1}{c_j}$ be as small as possible, we have:

$$\frac{1}{c_j} = min\{\frac{1}{c_j}, \text{where } c_j d_j \leq \frac{d_*}{f}\} = \frac{f d_j}{d_*}$$

Thus, the lemma is proved. □

[4]For ease of presentation, we show MSGApply() functions for SSSP-BF. The implementation of other functions, such as MSGGen() and MSGMerge(), and other algorithms, such as PageRank, are similar, and are omitted due to page limits. Please refer to our open source project for more details on the implementation.

```cpp
template <typename VertexValueType, typename
     MessageValueType>
int BellmanFord<VertexValueType, MessageValueType>::MSGApply
     (Graph<VertexValueType> &g, const std::vector<int> &
     initVSet, std::set<int> &activeVertice, const
     MessageSet<MessageValueType> &mSet)
{
    //******************Init******************
    activeVertice.clear();
    if(g.vCount <= 0) return 0;
    MessageValueType *mValues = new MessageValueType [g.
         vCount * this->numOfInitV];
    for(int i = 0; i < g.vCount * this->numOfInitV; i++)
        mValues[i] = (MessageValueType)INVALID_MASSAGE;
    for(int i = 0; i < mSet.mSet.size(); i++){
        auto &mv = mValues[mSet.mSet.at(i).dst * this->
             numOfInitV + g.vList.at(mSet.mSet.at(i).src).
             initVIndex];
        if(mv > mSet.mSet.at(i).value)
            mv = mSet.mSet.at(i).value;
    }
    //****************Init End*****************
    //******************CPU******************
    for(int i = 0; i < g.vCount; i++)
        vSet[i].isActive = false;
    for(int i = 0; i < g.vCount *
        numOfInitV; i++){
        if(g.verticesValue[i] > (VertexValueType)mValues[i])
            {
            g.verticesValue[i] = (VertexValueType)mValues[i
                ];
            if(!g.vList[i / numOfInitV].
                isActive)
                g.vList[i / numOfInitV].
                    isActive = true;
        }
    }
    //*****************CPU End*****************
    for(int i = 0; i < g.vCount; i++){
        if(g.vList.at(i).isActive)
            activeVertice.insert(i);
    }
    free(mValues);
    return activeVertice.size();
}
\label{lst:cpu}
```

Sample Code 1. CPU Implementation

```cpp
template <typename VertexValueType, typename
     MessageValueType>
int BellmanFord<VertexValueType, MessageValueType>::MSGApply
     (Graph<VertexValueType> &g, const std::vector<int> &
     initVSet, std::set<int> &activeVertice, const
     MessageSet<MessageValueType> &mSet)
{
    //******************Init******************
    activeVertice.clear();
    if(g.vCount <= 0) return 0;
    MessageValueType *mValues = new MessageValueType [g.
         vCount * this->numOfInitV];
    for(int i = 0; i < g.vCount * this->numOfInitV; i++)
        mValues[i] = (MessageValueType)INVALID_MASSAGE;
    for(int i = 0; i < mSet.mSet.size(); i++){
        auto &mv = mValues[mSet.mSet.at(i).dst * this->
             numOfInitV + g.vList.at(mSet.mSet.at(i).src).
             initVIndex];
        if(mv > mSet.mSet.at(i).value)
            mv = mSet.mSet.at(i).value;
    }
    //****************Init End*****************
    //******************GPU******************
    GPU_Mem_Copy_In(g.vList, vSet_d);
    for(auto offset = 0; mSet.mSet.length() >
        offset; offset += threadNum){
        int localMNum = mSet.mSet.length() -
            offset > threadNum ? threadNum :
            mSet.mSet.length() - offset;
        GPU_Memory_Copy_In(mSet.mSet[offset],
            msgSet_d, localMNum);
        MSGApply_kernel<<<1, threadNum>>>(mSet.
            mSet.length() - offset);
    }
    GPU_Mem_Copy_Out(vSet_d, g.vList);
    //*****************GPU End*****************
    for(int i = 0; i < g.vCount; i++){
        if(g.vList.at(i).isActive)
            activeVertice.insert(i);
    }
    free(mValues);
    return activeVertice.size();
}
\label{lst:gpu}
```

Sample Code 2. GPU Implementation