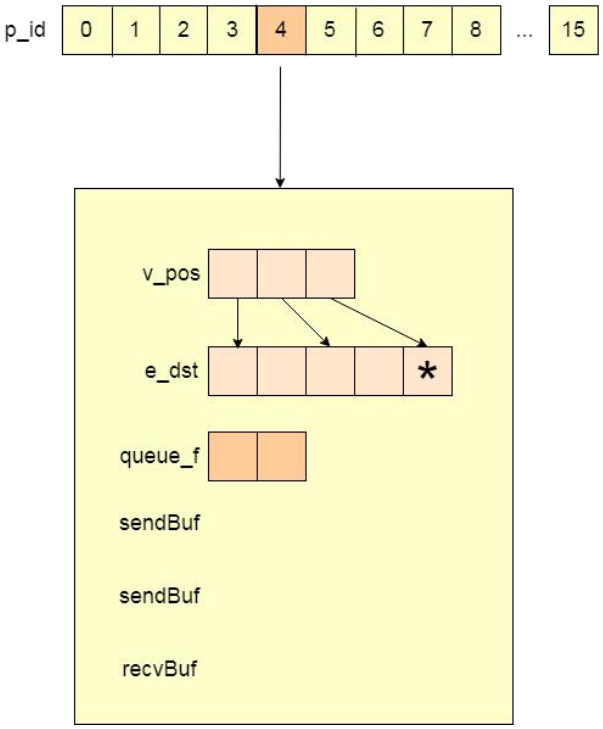
并行 BFS 算法

1. 一维划分数据结构

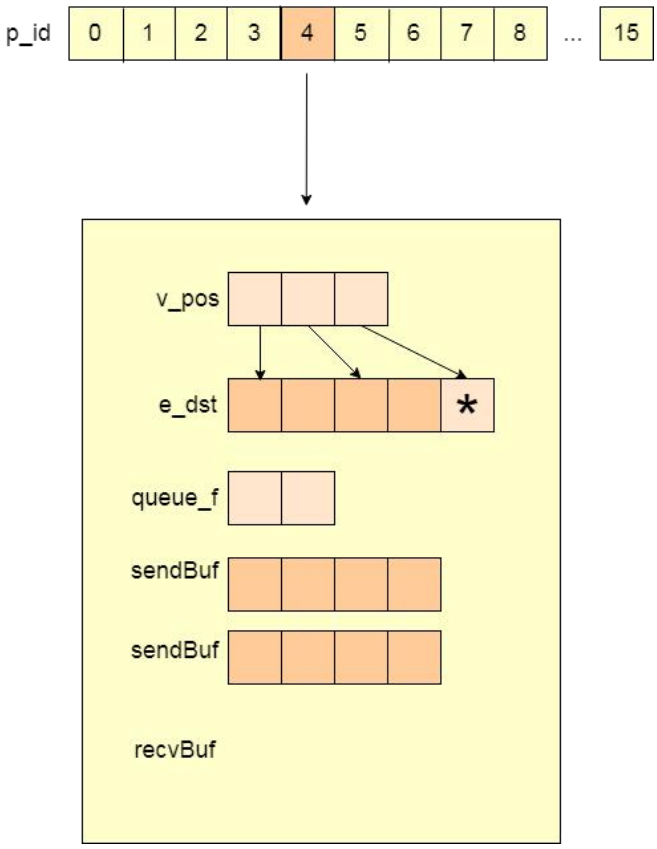
我们首先采用的是一维划分数据结构，即让每个处理器拥有 n/p 个顶点和从这些顶点出来的邻边。初始时每个进程拥有该进程负责处理的 v_pos 节点数组和 e_dst 邻边数组。

算法描述如下：

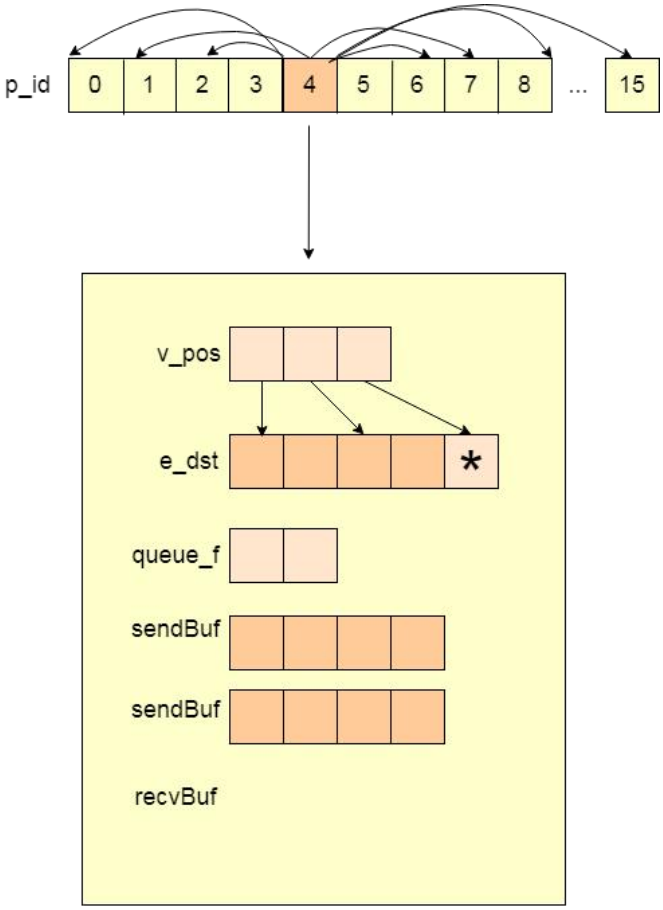
（1）每个进程队列中存储有将要遍历的点，准备开始遍历。如果这是第一轮迭代，则将起始点 s 加入其拥有者进程所在的队列中。



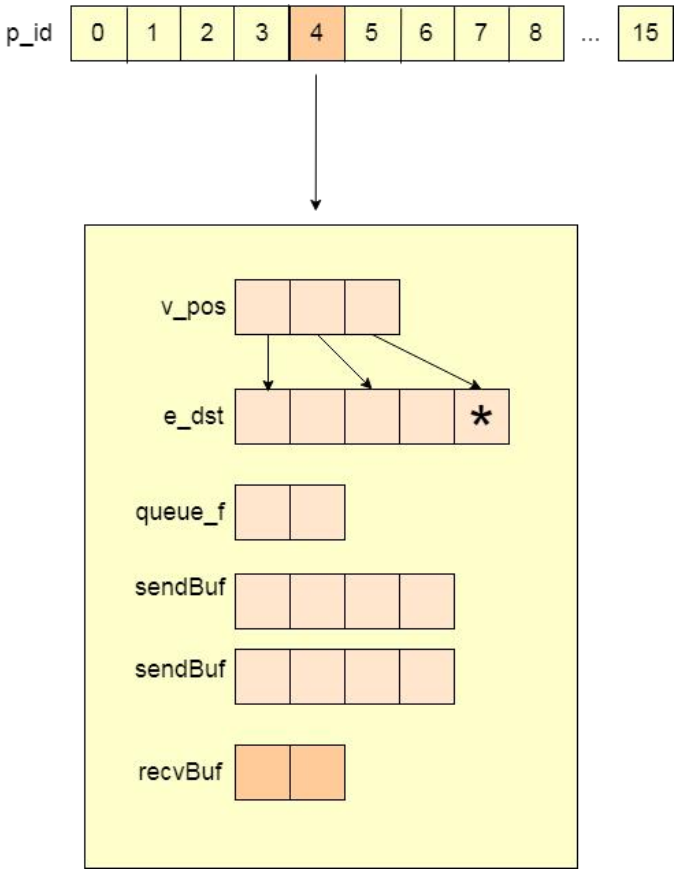
（2）每个进程各自遍历其队列中的节点，并根据 v_pos 数组找到邻边节点在 e_dst 中的偏移，通过遍历 e_dst 数组中的指定部分，找到下一步新加入的节点，将新加入的节点 v 存入缓冲区 $sendBuf$ 中，将新加入节点的前驱节点也就是当前队列节点 u 存入缓冲区 $sendpre$ 中。



（3）通过 MPI 的多对多通信，对于每个进程，都将新加入的节点发送给拥有者进程存储在*recvBuf* 中。



（4）每个进程按照其前驱节点所在进程的顺序遍历 $recvBuf$ 数组，如果节点没访问过，则将其加入暂存队列 $queue\_g$ 。然后检查有无新节点加入，如果有新节点加入，则将 $queue\_f$ 和 $queue\_g$ 交换，返回第一步继续迭代；如果没有新的节点加入，则退出循环。



2. 二维划分数据结构

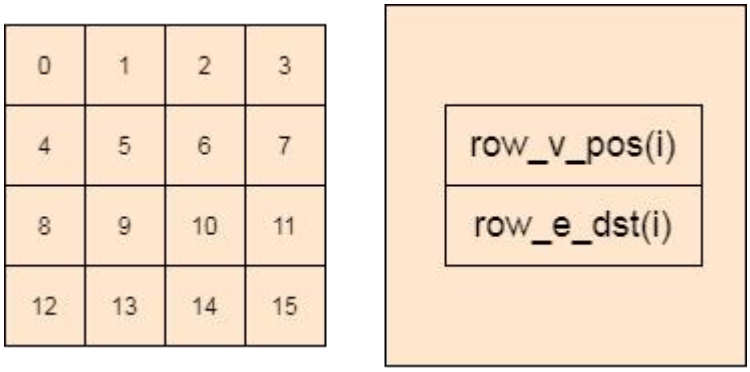
根据图的一维划分算法，我们得到启发，提出把图进行二维划分来实现并行广度优先搜索方法，具体思想如下：

我们把 p 个处理器组织成一个逻辑上的二维处理器阵列，共 p_r 行， p_c 列（ $p = p_r \cdot p_c$ ），把每个处理器按其所再的行和列索引标记成 $P(i, j)$ 。我们将二维处理器阵列划分为 p_r 个行通信域和 p_c 个列通信域。

根据我们的初始化算法，初始时所有进程存储有 v_pos 节点数组和 e_dst 邻边数组。

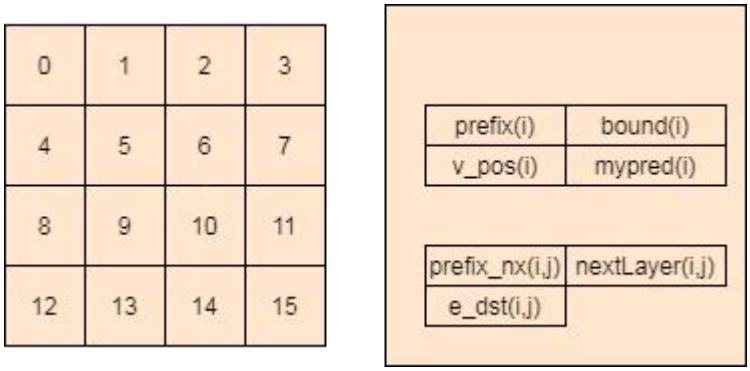
算法描述如下：

（1）初始阶段，每一个行通信域内的所有进程的 v_pos 和 e_dst 进行 *AllGather* 操作，让行通信域内每个进程都存储有该行通信域内的所有进程所拥有的 v_pos 和 e_dst 之集合 row_v_pos ， row_e_dst 。



(2) 预处理阶段，各个行通信域构建要处理的节点数组 $v\_pos(i)$，单个行通信域内各进程分别构建各自所要处理的邻边数组 $e\_dst(i,j)$。

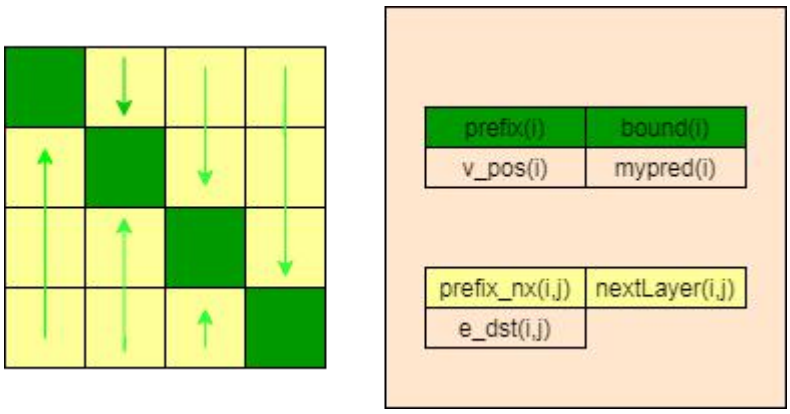
需要注意的是，同一个行通信域内进程的 $e\_dst(i,j)$ 数组不同，表示每个进程各自处理的的邻边。（划分规则为：邻边 (u,v) 的端点 v 所在的行通信域号为 k，则邻边 (u,v) 所在的列通信域号为 k），同一个行通信域内的进程的 $v\_pos(i)$ 数组是一样的，表示该行通信域内进程所拥有的节点，以及该节点所对应的邻边在本进程的 $e\_dst(i,j)$ 中的偏移。



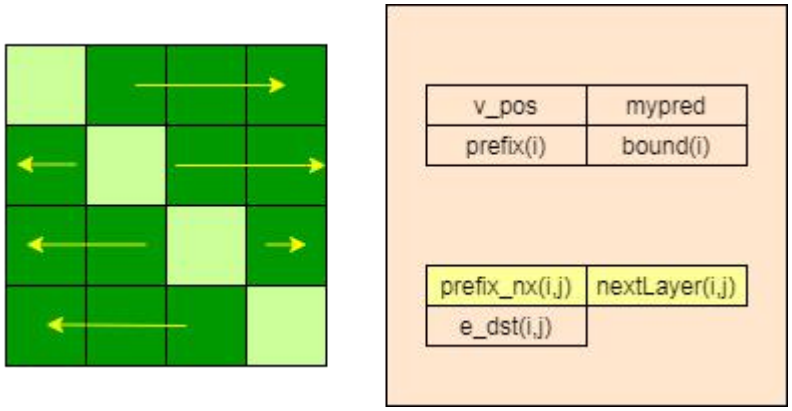
(3) 各行通信域中所有进程同时遍历 $bound(i)$ 队列，（算法开始时将遍历起始点 s 加入他所在的拥有者进程的 $bound(i)$ 队列），单个行通信域内所有进程的 $bound(i)$ 队列是一样的拷贝，它们共同进行遍历。

如果队列中节点 u 的邻边 (u,v) 的 v 端点在该进程自己的列通信域内，则将新

加入的节点加入到该进程的 $nextLayer(i,j)$ 数组中，将端点 u 保存在 $prefix\_nx(i,j)$ 数组中。最后，通过 *Gather* 操作将各个列通信域的所有 $prefix\_nx(i,j)$ 数组和 $nextLayer(i,j)$ 数组聚集为 $prefix(i)$ 和 $bound(i)$ 数组，存储在对角线进程中。（即行通信域号等于列通信域号的进程）



(4) 各行通信域将对角线进程中的 $prefix(i)$，$bound(i)$ 数组进行行广播，更新各行通信域内各进程的 $prefix\_nx(i,j)$ 与 $nextLayer(i,j)$。

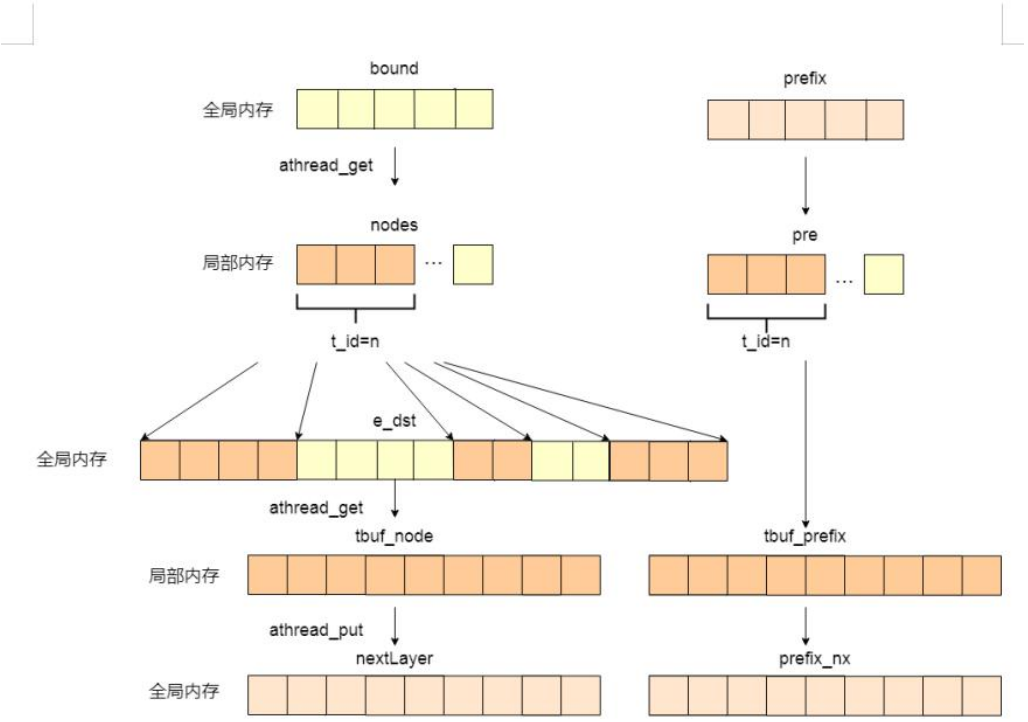


(4) 如果没有新入队的节点，迭代结束，否则返回第（3）步继续迭代。

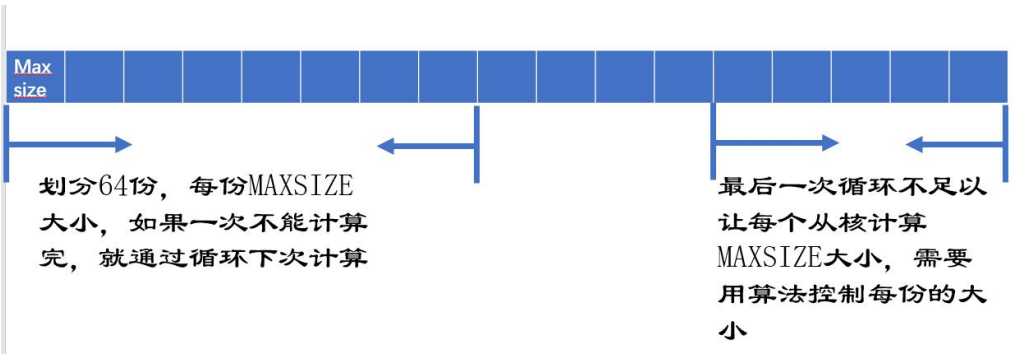
3. 从核二级缓存构造

对 $bound(i)$ 队列的遍历以及邻接边的遍历过程可以利用从核的多线程来分块完成，我们可以尝试将 $bound(i)$ 队列，$v\_pos(i)$ 数组，$e\_dst(i)$ 数组在从核 LDM 构建二级缓存，以减少对主核全局内存的访问。值得一提的是通过实验发现在从

核中对 $v\_pos(i)$ 数组的访问是离散的，不能直接构造从核缓存，我们尝试过在主核中先构造缓冲区先将 $v\_pos(i)$ 连续存储，然后再读入从核二级缓存，然而我们发现效果反而降低，我们权衡之后决定对 $v\_pos(i)$ 采用离散访问的方式，只对 $bound(i)$， $prefix(i)$， $e\_dst(i)$ 在从核中构建二级缓存。



4. 数据预处理的从核优化




```
int pr = local_e % CORE_NUM;
int length = ((local_e - offset) / CORE_NUM) + (my_id >= pr ? 0 : 1);
offset = offset + my_id * ((local_e - offset) / CORE_NUM) + (my_id >= pr ? pr : my_id);
```

最后一次的控制代码如上图所示