最大独立集问题

问题描述

给定图 G = <V,E>，独立集是指图 G 顶点集的一个子集，其中的任意两顶点都不邻接。最大独立集是指含顶点最多的独立集。

问题说明

编写一个多线程程序，找出给定图形的最大独立集。应用程序的命令行应包含输入文件和输出文件的名称。

输入文件说明：文件中应包含若干行，用来表示图形内的边。节点用从"A"到"Z"的两个大写字母表示。输入文件的每一行包含四个大写字母，第一个和第二个字母表示一个节点，第三个和第四个字母表示另一个节点，这两个节点代表图形内一条边的两个端点。文件末尾 (EOF) 标记说明图形输入结束。

输出：列出您编写的应用程序找到的最大独立集内的顶点名称，每行列出一个顶点。

命令行示例：mis.exe graph.txt graphout.txt

输入文件示例：graph.txt

==========

CCDD
JJCC
GGHH
HHBB
BBCC
DDBB
IICC
FFAA
AABB
EEAA
EEFF
AACC
BBGG
JJII
DDAA

输出文件示例：graphout.txt

==========
FF
GG
JJ
DD

计时：将根据应用程序的总执行时间来计分。

串行算法

假定用 MIS 保存最大独立集
首先用 VList 保存图中所有的顶点
找到图中第一个顶点 V1，假设顶点 V1 有三个邻接点 Va,Vb,Vc，那么我们有两种选择：
方案一，保留顶点 V1，删去 VList 中 Va,Vb,Vc 三个邻接顶点，开始处理第二个顶点 V2...
方案二，保留 V1 的三个邻接顶点，删除 VList 中的 V1，开始处理第二个顶点 V2...
我们接着开始处理第二个顶点 V2：如果 V2 还在 VList 中，说明 V2 到目前为止与 VList 中所有其它顶点都不邻接。和 V1 一样，我们还是有上述两种选择...
递归处理图中所有顶点，迭代到最后一个顶点时，VList 就是一个独立集。与 MIS 比较，在 MIS 中保存最大独立集。
下面是查找独立集串行版本的伪代码

```
// 查找独立集串行版本
VList g_Max; //保存最大独立集
void serial_do_mis(iterator begin, iterator end, VList vec) //从图中的顶点 begin 到 end 查找独立集,vec 保存目前为止所有互不邻接的顶点
{
        if(begin == end && g_Max < vec){ //如果已经迭代到最后一点，vec 就是一个独立集。
                g_Max = vec;
                return;
        }

        if(vec.contain(begin)) //如果 vec 含有 begin 点，则我们有两个选择
        {
                //方案一，删除当前点，递归处理下一点
                VList vec1 = vec;
                vec1.remove(begin);
                serial_do_mis(begin+1, end, vec1);

                //方案二，删除所有邻点，递归处理下一点
                VList vec2 = vec;
                for each(vectex v in begin.adjacency) vec2.remove(v);
                serial_do_mz(begin+1, end, vec2);
        }
}
```

优化

我们可以考虑把顶点集保存到一个 bitset 中，这样，象 `for each(vectex v in begin.adjacency) vec2.remove(v);`这样的操作就可以变成 `vec2 &= (~begin.adjacency)`。
如果 begin 的所有邻点都已被删除，那么就可以在本次递归中处理后面的顶点，减小递归深度。
使用串行版本处理我自己的一个测试用例所需时间是 2.1s 左右（Intel Core2 Duo CPU T7100）

并行算法

对于递归方案，一个简单的并行化方法是使用 Intel Threading Build Block（简称 TBB）的任务编程。
全局数据：最大独立集 g_Max 作为临界区要用读写锁保护。

并行化第一版本

下面是依据串行版本改写的并行化版本

```
typedef tbb::spin_rw_mutex mutex_t; //读写锁类型

mutex_t g_veMIS_mut;   //存取 g_veMIS 锁
VList g_veMIS;   //最大独立集,其中的 tag 成员保存 adj 成员里的顶点数

struct do_mis_task : tbb::task{

        // begin - 迭代起始顶点
        // end - 迭代终止顶点
        // vec - 保存当前独立集
        do_mis_task(iterator begin,
                iterator end,
                const VList &vec)
                :m_vec(vec),m_begin(begin),m_end(end)
        {}

        task* execute(){
                for(;
                        m_begin!=m_end &&
                        (!m_vec.contain(begin) || !m_vec.contain(m_begin.adjacency));
                ++m_begin); //跳过一些无意义点(已删除或所有邻点已删除)
```

```
                    if(m_end == m_begin){ //处理完毕，保存最大独立集
                              mutex_t::scoped_lock lock(g_veMIS_mut);
                              if(m_vec > g_veMIS) g_veMIS = m_vec;
                              return NULL;
                    }

                    size_t nMISCount = 0;
                    { // 取g_veMIS当前顶点数
                              mutex_t::scoped_lock lock(g_veMIS_mut, false);
                              nMISCount = g_veMIS.count();
                    }

                    //不管哪种方案，都至少会去除一个顶点，先确定是否有机会“竞选”最大独立集
                    if(m_vec.count()-1 > nMISCount)
                    {
                              tbb::task_list tl;
                              set_ref_count(1);
                              // 方案一，删除当前顶点
                              VList vec = m_vec
                              vec.remove(m_begin);
                              tbb::task &t1 =
*new(allocate_child())do_mis_task(m_begin+1,m_end,vec);
                              tl.push_back(t1);
                              increment_ref_count();
                              // 方案二，删除当前顶点的邻接点
                              m_vec.remove(m_begin.adjacency);
                              if(m_vec.count() > nMISCount)
                              {
                                        tbb::task &t2 =
*new(allocate_child())do_mis_task(m_begin+1,m_end,m_vec);
                                        tl.push_back(t2);
                                        increment_ref_count();
                              }

                              spawn_and_wait_for_all(tl);
                    }
                    return NULL;
          }
private:
          VList m_vec;
          iterator m_begin,m_end;
};
```

第一版本执行时间是 1.6s，比串行版本快了约 30%

并行化第二版本

观察第一版本的任务结构，发现父任务在 spawn_and_wait_for_all 之后没有工作要做。这种情况下可以考虑重用父任务，这样可以去除不必要的 wait 及成员拷贝的开销，这就是 Recycling Continuation Passing 模式。

```
...
struct do_mis_task : tbb::task{
          ...
          task* execute(){
                    ...
                    if(m_vec.count()-1 > nMISCount)
                    {
                              tbb::task_list tl;
                              tbb::task &c = *new(allocate_continuation()) tbb::empty_task;

                              // 方案一，删除当前顶点
                              VList vec = m_vec
                              vec.remove(m_begin);
                              tbb::task &tl =
*new(c.allocate_child())do_mis_task(m_begin+1,m_end,vec);
                              tl.push_back(tl);
                              c.increment_ref_count();
                              // 方案二，删除当前顶点的邻接点
                              m_vec.remove(m_begin.adjacency);
                              if(m_vec.count() > nMISCount)
                              {
                                        m_begin++;
                                        recycle_as_child_of(c); // recycling
                                        c.increment_ref_count();
                              }
                              c.spawn(tl);
                    }
                    return NULL;
          }
          ...
};
```

第二版本执行时间提高到 1.3s，有约 20%的改进

并行化第三版本

方案一的执行机会比方案二多得多，所以把“recycling”应用到方案一效率会更好一点。empty_task 是空任务，方案二会产生很多这种空任务，浪费任务调度时间。于是我尝试把新任务设置为与当前任务同一级，然后重用当前任务。这样，相当于“横向”增加任务，即所

有任务都是根任务的直接子任务。

```
...
struct do_mis_task : tbb::task{
          ...
          task* execute(){
                    ...
                    if(m_vec.count()-1 > nMISCount)
                    {
                              //方案二，删除所有邻点
                              VList v2 = m_vec;
                              v2.remove(m_begin.adjacency);
                              if(v2.count() > nMISCount)
                              {
                                        task* p = parent();
                                        do_mis_task* ptask =
new(allocate_additional_child_of(*p)) do_mis_task(m_begin+1, m_end, v2);
                                        p->spawn(*ptask);
                              }

                              //方案一，删除当前点
                              m_vec.remove(m_begin);
                              m_begin++;
                              recycle_as_continuation();
                              return this;
                    }
                    return NULL;
          }
          ...
};
```

第三版本执行时间提高到 1.0s，比第二版本有约 23%的改进。

结论

使用迭代法查找最大独立性是一个比较快速有效的方法，利用 TBB 程序库可以简单快速地把串行的迭代法的递归转化为并行的任务处理。
在编写高性能程序时，首先要确保算法的高效性，然后再做多核优化。
下面是程序执行测试样本的对比数据

	串行版本	并行版本
Core2 T7100(2 core)	2.15s	1.08s
Xeon E5345(8 core)	3.94s	0.65s