具有性能保证的图连接问题的Pregel算法

Da Yan#[1], James Cheng#[2], Kai Xing*[3], Yi Lu#[4], Wilfred Ng*[5], Yingyi Bu†[6]
#香港中文大学计算机科学和工程系
{1yanda, 2jcheng, 4ylu}@cse.cuhk.edu.hk
*香港科技大学计算机科学与工程系
{3kxing, 5wilfred}@cse.ust.hk
†加州大学欧文分校计算机科学系
6yingyib@ics.uci.edu

ABSTRACT

现实生活中的图往往是巨大的，如网络图和各种社交网络。这些庞大的图通常在分布式站点中存储和处理。在本文中，我们研究了采用谷歌Pregel的图算法，这是一个用于云端图处理的以顶点为中心的迭代框架。我们首先确定了一套高效Pregel算法的理想属性，如线性空间、每次迭代的通信和计算成本，以及对数的迭代次数。我们将这样的算法定义为*实用Pregel算法*（PPA）。然后，我们提出了计算*连接部件*（CCs）、*双连接部件*（BCCs）和*强连接部件*（SCCs）的PPA。计算BCC和SCC的PPA使用许多基础图问题的PPA作为构建模块，这些问题本身就很有意义。在大型真实图上进行的大量实验验证了我们算法的效率。

1. 简介

在线社交网络、移动通信网络和语义网络服务的普及，激发了人们对在大规模现实世界图上进行高效和有效分析的兴趣。为了处理这样的大图，谷歌的Pregel[12]提出了以顶点为中心的计算范式，该范式允许专业人员在设计分布式图算法时自然地像顶点一样思考。类似Pregel的系统运行在一个无共享的分布式计算基础设施上，可以很容易地部署在一个低成本的商品PC集群上。该系统还免除了程序员处理故障恢复的负担，这对在云环境中运行的程序很重要。

Pregel被证明比MapReduce模型更适用于迭代图的编译[8, 12, 13]。然而，虽然以前关于Pregel算法的工作[13]很好地展示了Pregel如何被用来解决一些图问题，但他们缺乏对成本复杂性的正式分析。

在本文中，我们研究了三个基本的图连通性问题，并提出了Pregel算法作为解决方案，该算法具有每...

Proceedings of the VLDB Endowment, Vol. 7, No. 14

的保证。我们所研究的问题是*连接在一起的。这些问题包括：组件*（**CCs**）、*双连接组件*（**BCCs**）和*强连接组件*（**SCCs**）。这些问题在现实生活中有着大量的应用，它们的解决方案是解决许多其他图问题的基本构件。例如，计算电信网络的BCCs可以帮助检测网络设计中的薄弱环节，而几乎所有的可达性指数都需要SCC的计算作为预处理步骤[4]。

为了避免特别的算法设计并确保良好的性能，我们定义了一类Pregel算法，它满足一组严格但实用的各种性能指标的约束：*(1)线型空间使用，(2)每迭代的线性计算成本*[1]*，(3)每迭代的线性通信成本，以及(4)最对数的迭代次数。*我们称这样的算法为**实用普雷格尔算法**（**PPA**）。最近，有人为MapReduce模型提出了一套类似但更严格的约束条件[19]。与我们要求的迭代次数对数相比，他们的工作要求迭代次数恒定，这对大多数图问题来说限制太大。事实上，在*共享内存*PRAM模型[22]下，即使是列表排序（即对仅由一条简单路径组成的有向图中的顶点进行排序），也需要使用$O(\log n)$时间，其中n是顶点的数量。这个约束也适用于许多其他的基本图形问题，如连接部件和生成树[18]。

为BCCs和SCCs这样的问题设计Pregel算法是具有挑战性的。虽然有简单的顺序算法来计算BCCs和SCCs，基于深度优先搜索（DFS），DFS是-完整的[15]，因此它不能应用于设计计算BCCs和SCCs的并行算法。

我们应用PPA的原理来开发满足严格性能保证的Pregel算法。特别是，为了计算BCCs和SCCs，我们开发了一套有用的构件，这些构件是基本图问题的PPA，如*广度优先搜索、列表排名、生成树、欧拉游和前/后序遍历*。作为基本的图问题，它们的PPA解决方案也可以应用于本文所考虑的BCC和SCC之外的许多其他图问题。

我们使用具有多达数亿个顶点和数十亿条边的大型现实世界图来评估我们的Pregel算法的性能。我们的结果验证了我们的算法在计算大型图的CCs、BCCs和SCCs时是有效的。

本文的其余部分组织如下。第2节回顾了Pregel和相关工作。我们在第3节中定义了PPA。第4-6节讨论了CCs、BCCs和SCCs的算法。然后，我们报告了

[1]Pregel算法以*迭代*（或*超级步骤*）的方式进行。

第7节为实验结果，第8节为结论。

2. 相关的工作

Pregel[12]。Pregel是基于批量同步parallel（BSP）模型设计的。它将顶点分配给集群中的不同机器，其中每个*顶点都*与它的邻接列表（即v的邻居集合）相关联。Pregel中的程序实现了一个用户定义的*compute*()函数，并以迭代方式进行（称为*超级步骤*）。在每个超级步骤中，程序为每个活动顶点调用*compute*()。*compute*()函数为一个顶点v执行用户指定的任务，比如处理v的传入信息（在前一个超级步骤中发送），向其他顶点发送信息（在下一个超级步骤中接收），以及让v投票停止。如果一个停止的顶点在随后的超级步骤中收到消息，它就会被重新激活。当所有顶点都投票决定停止，并且在下一个超级步骤中没有悬而未决的消息时，程序就会终止。Pregel对超级步骤进行了编号，以便用户在实现算法逻辑时可以使用当前的超级步骤编号。因此，一个Pregel算法可以按*compute*()函数。因此，一个Pregel算法可以按在不同的超级步骤中形成不同的操作，通过分支在当前的超阶数。

Pregel允许用户实现一个*combined*()函数，该函数指定如何将从机器$_{Mi}$发送到机器$_{Mj}$中的同一顶点v*的*消息合并。这些消息被合并成一个单一的消息，然后从$_{Mi}$发送到$_{Mj}$中的v。只有在对消息进行换元和同元操作时，才会应用Combiner。例如，在Pregel的PageRank算法[12]中，来自机器$_{Mi}$的要发送给机器$_{Mj}$中同一目标顶点的消息可以被合并成一个等于它们之和的单一消息，因为目标顶点只对消息之和感兴趣。Pregel还支持聚合器，这对全局通信很有用。每个顶点都可以在超级步骤的*compute*()中向聚合器提供一个值。系统对这些值进行聚合，并在下一个超级步骤中把聚合的结果提供给所有顶点。

Pregel算法。除了本文和Google关于Pregel的原创论文[12]，我们只知道另外两篇研究Pregel算法的论文，[13]和[17]。然而，这些算法是在尽力而为的基础上进行的，没有对其复杂性进行任何正式的分析。具体来说，[13]旨在证明Pregel模型可以被用来解决社会网络分析中的许多图问题，而[17]则侧重于克服由直接的Pregel实现引起的一些性能瓶颈的优化技术。

GraphLab[11]和PowerGraph[10]。GraphLab[11]是另一个以顶点为中心的分布式图计算系统，但它采用了与Pregel不同的设计。GraphLab同时支持同步和异步执行。然而，异步执行没有超步数的概念，因此不能支持在不同的超步下分支到不同操作的算法，如4.2节中的S-V算法。此外，由于GraphLab主要是为异步执行而设计的，它的同步模式并不像Pregel那样富有表现力。例如，由于GraphLab只允许一个顶点访问其相邻的顶点和边的状态，所以它不能支持顶点需要与非相邻的人交流的算法。GraphLab的另一个限制是它不支持图的变异。

GraphLab 2.2，即PowerGraph[10]，按边而不是按顶点来划分图形，以解决高度顶点造成的不平衡工作量。然而，应该使用更复杂的以边为中心的聚集-应用-散射（GAS）计算模型，这损害了用户友好性。

PRAM。PRAM模型假定有许多处理器和一个共享存储器。PRAM算法已经被提出用于计算CC[18]、BCC[20]和SCC[2, 3, 9]。然而，PRAM模型并不适合建立在无共享架构上的云环境。此外，与Pregel和MapReduce不同，PRAM算法没有容错能力。然而，许多PRAM算法的思想可以应用于设计高效的Pregel算法，我们将在后面的章节中展示。

3. 实用的Pregel算法

现在我们定义一些常用的符号，并介绍实用普雷格尔算法的概念。

符号。给定一个图$G = (V, E)$，我们用n表示顶点V*的*数量，用m*表示边*E的数量，我们还用δ表示G的*直径*。对于无向图，我们用$\Gamma(v)$表示顶点v的*邻居*集合，用$d(v)=|\Gamma(v)|$表示v的*度*。对于一个有向图，我们用$\Gamma(v)$和$\Gamma(v)$表示v的*内邻*和*外邻*集合，而v的*入度*和*出度*。$d_{in}(v) = |\Gamma_{in}(v)|$ and $d_{out}(v) = |\Gamma_{out}(v)|$，分别。

如果一个Pregel算法满足以下约束条件，则被称为**平衡实用Pregel算法（BPPA）。**

1. *线性空间使用：*每个顶点v使用$O(d(v))$（或$O(d_{in}(v)+d_{out}(v))$）空间的存储。
2. *线性计算成本：计算*()的时间复杂性每个顶点v的函数是$O(d(v))$（或$O(d_{in}(v)+d_{out}(v))$）。
3. *线性通信成本：*在每个超级步骤中，每个顶点v发送/接收的信息大小为$O(d(v))$（或$O(d_{in}(v)+d_{out}(v))$）。
4. *最多对数轮数：*该算法在$O(\log n)$个超级步骤后终止。

约束条件1-3提供了良好的负载平衡和每个超级步骤的线性成本，而约束条件4控制了总运行时间。我们将在后面的章节中看到，一些满足约束条件1-3的算法需要$O(\delta)$轮。由于许多大型真实图的直径较小，特别是由于小世界现象导致的社交网络，我们考虑需要$O(\delta)$轮的算法也满足约束条件4，如果δ对数n的输入图。

对于某些问题，BPPA的每顶点要求可能过于严格，我们只能实现*整体的线性空间使用*、*计算和通信成本*（仍为$O(\log n)$轮）。我们把满足这些约束的Pregel算法简单地称为**实用Pregel算法（PPA）**。

动机。我们定义BPPA和PPA是为了描述一组能在实践中有效运行的Pregel算法的特征。除了本文提出的算法外，其他Pregel算法，例如Pregel论文[12]中的四个演示算法，也具有这些特征（我们可以证明它们是BPPA）。PageRank（恒定的超级步骤），单源最短路径（$O(\delta)$超级步骤），双点匹配（$O(\log n)$超级步骤），以及半聚类（恒定的超级步骤）。然而，这些现有的Pregel algorithms是在尽力而为的基础上设计的，没有正式的性能要求需要满足或设计规则需要遵循。例如，虽然文献[13]中开发的用于二维码估计的Pregel算法是一个$O(\delta)$超步骤的BPPA，但文献[13]中用于三角形计数和聚类系数计算的Pregel算法不是一个PPA。具体来说，在三角形计数算法的超级步骤1中，

一个顶点为每一对邻居发送一个消息。

导致需要缓冲和发送的信息数量达到四次方。有了PPA/BPPA的概念，三角形计数算法的用户在应用该算法时就可以意识到可扩展性的限制。PPAs/BPPAs的要求也可以作为希望开发高效Pregel算法的程序员/研究人员的指导原则，他们可以将现有的PPAs作为他们算法的构建模块。

在接下来的三节中，我们介绍了三个有趣的图连接问题的PPAs/BPPAs。我们还展示了一些基本图问题的PPAs/BPPAs如何被用作构建模块，以开发更复杂的PPA/BPPA来解决其他图问题，如计算BCCs。

4. 连接的组件

在本节中，我们介绍了两种Pregel算法，用于计算

CCs。第4.1节介绍了一个需要$O(\delta)$个超级步骤的BPPA。这种算法在许多现实世界中的图上都能很好地工作，并且有一个小的
直径。然而，它在拉长直径的图上可能非常慢。
如空间网络，对其而言，$\delta \approx O(\mathrm{n})$。我们提出了一个
第4.2节中的$O(\log n)$-superstep PPA来处理这种图。

4.1 Hash-Min PPA

在提出用于计算CC的$O(\delta)$-超步BPPA之前。
我们首先定义一些图的符号。我们假设图G中的所有顶点都有一个唯一的ID。为了讨论的方便，我们简单地用v来指代顶点v的ID，因此，表达式$u <$
v意味着u的顶点ID比v的小。我们将G中一个（强）连接组件的*颜色定义*为该组件中所有顶点中最小的顶点。一个*顶点*的颜色，用$color(v)$表示，被定义为包含v的组件的颜色，因此，G中所有颜色相同的顶点构成一个组件。

最近提出了一种MapReduce算法，称为Hash-Min，用于计算CC[14]。该算法的思路是将每个顶点v *到*目前为止看到的最小顶点（ID）广义化，用$min(v)$表示；当这个过程收敛时，$min(v)= color(v)$，适用于所有的v。

在超级步骤1中，每个顶点v初始化$min(v)$为集合中最小的顶点$(\{\} \cup v$
$\Gamma(v))$，将$min(v)$发送给v的所有邻居，并投票决定停止。在随后的每个超级步骤中，一个顶点v从传入的信息中获得最小的顶点，用u表示。如果$u<v$，v设置$min(v)=u$，并将$min(v)$发送给所有邻居。最后，v投票决定停止。

我们证明该算法是一个BPPA，具体如下。对于任何CC，最小顶点的ID到达CC中的所有顶点最多需要δ个超级步骤，在每个超级步骤中，每个顶点v最多需要$O(d(v))$时间来计算$min(v)$，并使用$O(1)$空间发送/接收$O(d(v))$消息。

[14]中还提出了另外三种计算CC的MapReduce算法。然而，它们要求每个顶点都要保持一个集合，这个集合的大小可以和它的CC的大小一样大，并且要把整个集合发送给一些顶点。因此，由于高度倾斜的通信和计算以及过高的空间成本，它们不能被转化为高效的Pregel实现。

4.2 S-V PPA

我们的贡献。我们提出了一个基于S-V算法[18]的$O(\log n)$-superstep
PPA。我们注意到，最先进的分布式算法在某些类型的图上只能达到$O(\log n)$的迭代指数[14]，而[14]也声称，并发写的要求使得S-V算法难以被翻译成MapReduce（类似于Pregel）。我们表明，将S-V算法直接翻译成Pregel是不正确的，并且

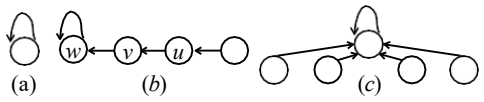


图1：Shiloach-Vishkin算法的森林结构

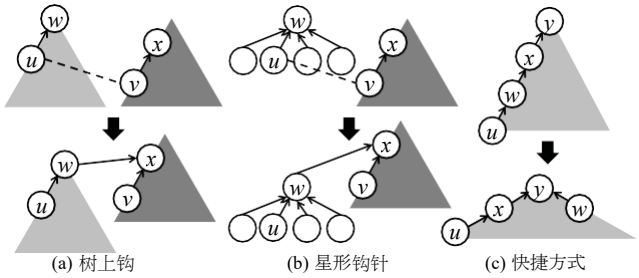


图2：树形挂钩、星形挂钩和快捷方式

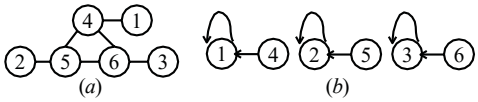


图3：改用星形钩的说明

然后改变算法逻辑，得到一个用于CC计算的$O(\log n)$超步骤的PPA。

算法概述。在S-V算法中，每个顶点u*都*有一个指针$D[u]$。最初，$D[u] = u$，形成一个自循环，如图1（a）所示。在整个算法中，顶点被一个森林化，使得森林中每棵树的所有顶点都属于同一个CC。树的定义在这里放宽了一些，允许树根w有一个自循环（见图1（b）和1（c）），即$D[w]=w$；而树中任何其他顶点v的$D[v]$都指向v的父。

S-V算法分轮进行，在每一轮中，指针按三个步骤更新（如图2所示）。(1)*树钩*：对于每条边(u, v)，如果u的父级$w = D[u]$是树根，则将w钩为v的父级$D[v]$的孩子（即。(2)*星形钩住*：对于每条边（u, v），如果u在一个星形中（星形的例子见图1(c)），像步骤(1)那样将星形钩到v的树上；(3)*捷径*：对于每个顶点v，通过将v钩到v的父辈，即设置$D[v]=D[v]$，使顶点v及其后裔更靠近树根。当每个顶点都在一个星中时，该算法就结束了。

只有当$D[v]<$
$D[u]$时，我们才会在步骤(1)中进行树的勾连，因此如果u的树由于边（u, v）而被勾连到v*的树*，那么边（v, u）就不会再将v的树勾连到u的树上。

对于星形钩子来说，"$D[v]<$
$D[u]$"这个条件是不需要的，因为如果u的树被钩到v的树上，v的树就不可能是一个星形（例如，在图2（b）中，在钩子之后，u*距离*x*有*两跳）。然而，在Pregel的计算模型中，步骤(2)是不能进行的。考虑图3(a)所示的图形，假设我们在步骤(1)之后马上得到图3(b)中的三个星星。如果不需要单向条件，将$D[D[u]]$设为$D[v]$会使$D[1]=2$，$D[2]=3$，$D[3]=1$，通过边（4，5），（5，6）和（6，4），从而形成一个循环，违反了算法所要求的树形构造。在PRAM模型中不存在这样的问题，因为$D[u]$和$D[v]$的值在每次写操作后都会立即更新

，而在Pregel中，这些值是上一个超级步骤中收到的。

为了解决这个问题，我们研究了Pregel执行中的算法逻辑，发现如果我们在挂钩过程中设置$D[v_a] \leftarrow v_b$时总是要求$v_a < v_b$，我们可以证明指针值是单调减少的，因此当算法终止时，对于任何顶点v，$D[v] = color(v)$。基于这个结果，我们改变了算法，要求"$D[v] < D[u]$"为星形勾选。然后，S-V算法可以转化为Pregel算法，通过改变信息来更新指针$D[.]$，遵循图2中的三个步骤，直到每个顶点v都在一个星中。由于篇幅有限，我们在技术报告[24]的附录A中介绍了该算法的细节。

基于S-V的Pregel算法是一个$O(\log n)$-superstep PPA，这可以被证明为原始的S-V算法计算CCs在$O(\log n)$轮中[18]，其中每一轮都是在一个恒定数量的超级步骤。然而，该算法不是BPPA，因为一个*顶点*可能成为超过$d(v)$顶点的父顶点，因此在一个超级步骤中接收/发送超过$d(v)$的消息，尽管每个超级步骤中的总消息数总是以$O(n)$为界。

S-V算法也可以被扩展，得到一个用于计算生成树的$O(\log n)$-超级步骤PPA，其细节可以在[24]的附录A中找到。

5. 双连接的组件

我们用于计算BCC的PPA是基于[20]中的PRAM算法的思想，但我们做出了以下新的贡献。

我们的贡献。据我们所知，在Pregel中计算BCCs的问题以前从未被研究过。因此，重要的是要证明对于这个问题存在一个具有强大性能保证的Pregel算法，我们将通过提出一个计算BCC的PPA来建立这个算法。其次，现有的关于设计Pregel算法的研究[14, 13]往往忽略了丰富的PRAM算法。我们的计算BCC的PPA证明了PRAM算法的一些想法可以应用于设计Pregel算法。第三，虽然主要思想是基于[20]，但我们的PPA用于BCC计算的design是不简单的。具体来说，我们的BCC算法是由一些构件组成的，为了确保我们的最终算法是一个PPA，我们为每个构件设计了一个PPA。最后，在我们的BCC算法中使用的这些构件本身就是基本的图操作，对许多其他分布式图算法的设计很有用。因此，我们对它们进行了更深入的研究，并为它们中的每一个精心设计了一个PPA，这通常比现有的PRAM算法要简单得多。

5.1 BCC及其PRAM算法

双连通部分（BCC）。无向图G的BCC是G的一个最大的子图，在移除一个任意的顶点后仍然保持连接。我们用图4所示的图来说明BCC的概念，其中虚线构成一个BCC，实线构成另一个。让R成为G的边的集合上的等价关系，使得$e_1 R e_2$ iff $e_1 = e_2$或者e_1和e_2一起出现在某个简单的循环中，那么R就定义了G的BCCs。

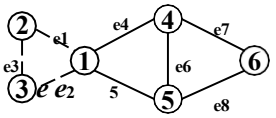
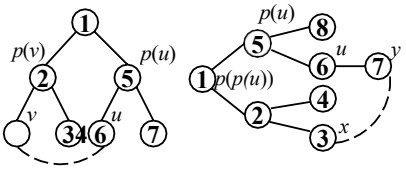


图4：BCC图示



(a) 案例1和2　　(b) 案例3

图5：G的构造图示*

含有（4，5）和（1，2）。如果一个顶点属于一个以上的BCC，例如图4中的顶点1，则被称为*衔接点*。移除一个衔接点会断开包含该衔接点的连接组件。

Tarjan-Vishkin的PRAM算法。如果我们构建一个新的图G'，其顶点对应于G的边，并且G'中存在一条边（e_1, e_2），如果$e_1 R e_2$，那么G'的CC对应于BCCs

然而，G'中的边的数量可以是m的超线性。Tarjan-Vishkin（T-V）的PRAM算法[20]构建了一个简明图G^*，包含G'中的一个小的边的子集，其大小被$O(m)$所限制。他们证明，只要计算G的CCs*，就可以得到G的BCCs。

由于我们用于计算BCC的PPA也是基于这个想法，我们首先在下面介绍G的定义*。假设$G=(V, E)$是连接的，T是G的生成树。是有根的，T中的每个顶点u都被分配了一个预设的编号$pre(u)$。让$p(u)$为T中顶点u的父本。我们构建$G^* = (V^*, E^*)$如下。首先，我们设置$V^* = E$。然后，我们向E^*添加一条边（e_1, e_2），其中$e_1, e_2 \in E$, iff e_1和e_2满足以下三种情况之一。

- 情况1：$e_1 = (p(u), u)$是T中的一条树边，$e_2 = (v, u)$是一条非树边（即e_2不在T中），$pre(v) < pre(u)$。
- 情况2：$e_1 = (p(u), u)$和$e_2 = (p(v), v)$是T中的两条树边，u和v在T中没有上升-下降的关系，并且$(u, v) \in E$。
- 案例3：$e_1 = (p(u), u)$和$e_2 = (p(p(u)), p(u))$是T中的两条树边，$(x, y) \in E$ s.t. x是T中$p(u)$的非子嗣，y是T中u的子嗣。

图5说明了这三种情况，其中实线是T中的树边，虚线是(G')中的非树边。顶点用它们的前序号来标示。案例1由图5(a)中的$e_1=(5, 6)$和$e_2=(3, 6)$表示。注意$e_1 R e_2$，因为e_1和e_2在一个简单的循环中，1, 2, 3, 6, 5, 1。情况2也如图5（a）所示，$e_1=(5, 6)$，$e_2=(2, 3)$，而且e_1和e_2也在同一个简单循环中。案例3如图5（b）所示，$e_1=((5, 6)$，$e_2=)(1, 5)$，其中e_1和e_2处于简单循环1，2，3，7，6，5，1。

由于情况1（和情况2），G的每个非树边（u，v）最多引入一条边到G^*，由于情况3，每个树边（$p(u)$，u）最多引入一条边。因此，$|E^*| = O(m)$。

5.2 计算BCC的PPA

我们计算BCC的PPA也是基于Tarjan-Vishkin算法的思想，即构建简明图G^*，然后计算G的CC*，得到G的BCC。在不失一般性的情况下，我们假设G是连接的，因为BCC计算在

不同的CC是独立的，可以并行化。为了构建G^*
，我们首先在第5.2.1-
5.2.4节中提出了一组构件，然后在第5.2.5节中把所有东西
放在一起，得到了
我们用于计算BCC的最终PPA。

5.2.1 生成树的计算

为了构建G^*，我们首先需要一个G的生成树，表示为T.我们提出了一个用于生成树编译的$O(\delta)$超步骤BPPA。拨款情况如下。

该算法执行*广度优先搜索*（BFS），并从一个源顶点开始计算非加权图G上的生成树。

s.G中的每个顶点v都保持两个字段，即v的父代，用$p(v)$表示；以及v与s的最短路径距离（或BFS级别），用$dist(v)$表示。最初，只有s是活跃的，$p(s)=null$并且在超级步骤1中，$dist(s)=0$，对于所有其他v，$dist(v)=\infty$。

s向其所有邻居发送s、$dist(s)$，并投票决定是否停止。在随后的每个超级步骤中，如果一个顶点v收到任何消息，它首先检查v是否曾经被访问过（即$dist(v)$是否$<\infty$）：如果不是，它就用收到的任意消息u、$dist(u)$更新$dist(v)=dist(u)+1$和$p(v)=u$，并将v、$dist(v)$发给v的所有邻居。最后，v投票决定停止。

当算法终止时，我们得到一条树边（$p(v), v$）。从每个顶点$v=s$出发，构成一棵生成树，其根在s.很容易看出，该算法是一个$O(\delta)$超步骤的BPPA。在G是断开的情况下，我们首先使用第4.1节中描述的算法计算每个顶点v的*颜色*（v），然后选择$s=color(s)$的顶点s作为每个CC的源。由于，多源BFS是以并行方式进行的，所以总体的超级步骤的数量仍然是$O(\delta)$。对于处理大直径δ的图，如第4.2节所述，S-V算法可以被扩展为一个$O(\log n)$-superstep PPA来计算生成树T。

5.2.2 预先订购的编号

考虑第5.1节中构建G的边的三种情况*。在案例1中，我们需要生成树T中每个顶点v的前序数（即$pre(v)$）。我们提出了一个$O(\log n)$的超级步骤BPPA来计算所有顶点的预序数在T中。我们还提出了一个用于计算后序数的对称BPPA。

为了计算前序编号，我们首先计算生成树T的欧拉游。欧拉游是树的一种表示方法，在许多并行图算法中很有用。树被看作是一个有向图，其中每条树边（u，v）被看作是两条有向边（u，v）和（v，u），而树的欧拉游只是有向图的一个欧拉回路，也就是说，一个正好访问每条边一次，并在它开始的同一顶点结束的路径。

我们提出了一个$O(\log n)$-superstep BPPA来计算树T的Euler之旅，如下所示。

用于计算欧拉游的BPPA。假设每个顶点v的邻居是按照他们的ID排序的，这对于图的邻接列表表示法来说是很常见的。对于一个顶点v，让$first(v)$和$last(v)$是v在排序后的第一个和最后一个邻居；对于v的每个邻居u，如果$u=last(v)$，让$_{nextv}(u)$是v在排序后的邻接列表中挨着u的邻居。我们进一步定义$_{nextv}(last(v)) = first(v)$。考虑图6中的例子。对于顶点4的邻接列表，我们有$first$（4）=0，$last$（4）=6，$_{next4}$（0）=5，$_{next4}$（5）=6和$_{next4}(6)=0$。

如果我们把$_{nextv}(u) = w$翻译成指定(u, v)旁边的边是(v, w)，我们就会得到一个树的欧拉游。再参考图6的例子，其中给出了一个以顶点0为起点和终点的欧拉游。(2, 1)的下一条边是(1, 3)，b-导致$_{next1}(2)=3$，而(6, 4)的下一条边是(4, 0)，因为

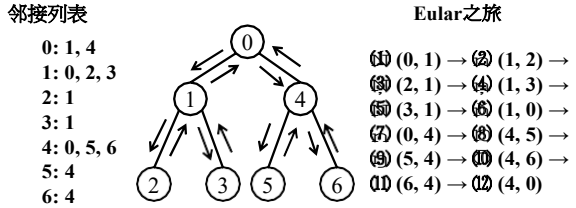


图6：欧拉旅行

$_{next4}(6)=0$。事实上，从任何顶点v和任何邻居开始v的u，（（v，$x = _{nextv}$（u）），（x，$y = _{nextx}$（v）），（y，$_{nexty}$（x））。

.., （u，v））定义了一个欧拉游。

我们提出了一个2个超级步骤的BPPA来构建欧拉游，如下所示。在超级步骤1中，每个顶点v向每个邻居u发送消息$u, _{nextv}(u)$；在超级步骤2中，每个顶点u接收每个邻居v发送的消息$u, _{nextv}(u)$，并在u的邻接列表中与v一起存储$_{nextv}(u)$。当算法结束时，对于每个顶点u和每个邻居v来说，(u, v)的下一条边被获得为$(v, _{nextv}(u))$。

该算法需要恒定数量的超级步骤，在每个超级步骤中，每个顶点v发送/接收$O(d(v))$消息（每个消息使用$O(1)$空间）。通过将$_{nextv}(.)$实现为与v相关的哈希表，我们可以在给定u的$O(1)$预期时间内获得$_{nextv}(u)$。

在得到T的欧拉游，也就是一个边的循环之后，我们在某个边上将其断开，得到一个边的列表。然后，我们使用列表排位操作，从列表中计算出T中顶点的前序和后序编号。由于我们的前序和后序编号的BPPA是基于列表排位的，我们首先介绍了列表排位的概念，并在下面提出了一个用于列表排位的$O(\log n)$-超步骤BPPA。

列表排名的BPPA。考虑一个有n个对象的 链接列表，其中每个对象v都与一个值$val(v)$和一个指向其前任$pred(v)$的链接相关。位于头部的对象v有$pred(v) = null$。对于每个对象v，让我们定义$sum(v)$为从v到头部的所有对象的值的总和，这些对象都是沿着predecessor链接的。*列表排名*问题为每个对象v计算$sum(v)$。如果$val(v) = 1$对于每个$v$in，那么$sum(v)$只是v在列表中的排名，即v前面的对象数量加1。

在列表排名中，对象是以任意的顺序给出的。我们可以简单地把它看作是一个由单一模拟路径组成的有向图。尽管简单，但列表排名是并行计算中的一个重要问题，因为它是许多其他并行算法的基础。

我们现在描述一下我们的列表排名的BPPA。最初，每个vertex v都指定$sum(v) = val(v)$。然后在每一轮中，每个vertex v做以下事情。如果$pred(v)=null$，v设置$sum(v)=sum(v)+sum(pred(v))$，$pred(v)=pred(pred(v))$；否则，$v$投票决定停止。if-分支在三个超级步骤中完成。(1)v向$u = pred(v)$发送消息，要求提供$sum(u)$和$pred(u)$的值；(2)u将要求的值发回给v；(3)v更新$sum(v)$和$pred(v)$。这个过程重复进行，直到每个顶点v的$pred(v)=null$，这时所有顶点都投票停止，我们的$sum(v)$就如愿以偿。

图7说明了该算法的工作方式。最初，对象v_1-v_5组成一个链接列表，$sum(v_i) = val(v_i) = 1$，$pred(v_i) = v_{i-1}$。在第一轮中，我们有$pred(v_5)=v_4$，所以我们设置$sum(v_5)=sum(v_5)+sum(v_4)=1+1=2$，$pred(v_5)=pred(v_4)=v_3$。我们可以用类似的方法验证其他顶点的状态。在第二轮中，我们有$pred(v_5)=v_3$，因此我们设置$sum(v_5)=sum(v_5)+sum(v_3)=2+2=4$，$pred(v_5)=pred(v_3)=v_1$。

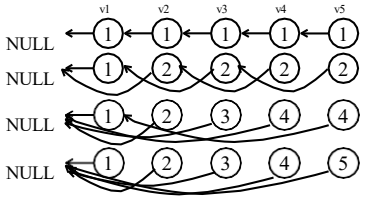


图7：用于列表排名的BPPA图示

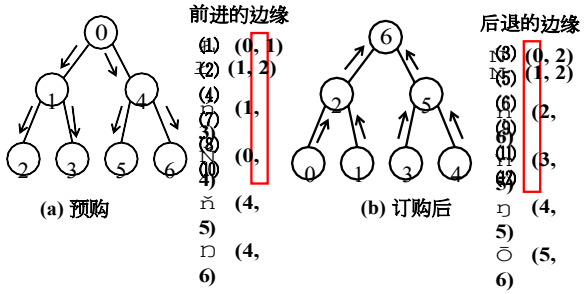


图8：订单前和订单后

$pred(v_5) = pred(v_1) = null$。我们可以通过归纳法证明，在第$i$轮，我们设定$sum(v_j) = \sum_{k=j-2^i+1}^{j} val(v_k)$和$pred(v_j) = v_{j-2^i}$。此外，每个对象$v_j$最多发送一个消息给$v_{j-2^i}$，并最多接收一个来自$v_{j+2^i}$的消息。该算法是一个BPPA，因为它在对数$n$轮中终止，并且每个对象每轮最多发送/接收一条信息。

让我们假设，我们已经得到了从s开始的生成树的欧拉游P，给出的是$= (s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, s)$。我们通过设置将循环分解成一个列表$pred(s, v_1) = null$。我们现在考虑如何使用列表排位法从列表中计算出前序和后序的数字。

前序和后序编号。树T的深度优先遍历会产生树中顶点的前序或后序编号，这些编号在许多应用中都很有用，比如决定两个树顶点的祖孙关系，我们将在第5.2.3节讨论。让$pre(v)$和$post(v)$分别为T中每个顶点v的前序和后序数。我们提出了一个BPPA，用于从树T的欧拉游中计算前序和后序数。

我们通过对每条边e的处理来制定一个列表排名的问题$\in P$作为一个顶点，并设定$val(e)=1$。在获得每个e的$sum(e)\in P$后，我们使用两个超级步骤BPPA将其中的边标记为前向/后向边：在超级步骤1中，每个顶点$e = (u, v)$将$sum(e)$发送给$e^t = (v, u)$；在超级步骤2中，每个顶点$e^t = (v, u)$从$e = (u, v)$接收$sum(e)$，如果$sum(e^t) < sum(e)$，则将e^t自己设为前向边，否则为后向边。在图6中，边（1，2）是一条前向边，因为其等级（即2）是小于（2，1）（即3），而边（4，0）是一条后向边，因为其等级（即12）大于（0，4）（即7）。

为了计算$pre(v)$，我们运行第二轮列表排名，对每条前向边$e$in设置$val(e)=1$，P对每条后向边e^t)=0。然后，对每条前向边$e = (u, v)$，我们得到顶点v的$pre(v) = sum(e)$。我们对树设置$pre(s) = 0$

例如，图8(a)显示了前向边(u, v)的顺序，其P中顶点已经被标上了预序号。显然，（u，v）

我们可以使用Pregel中的聚合器很容易地计算出n，每个顶点提供1的值。（在G是一个森林的更一般的情况下，聚合器计算每个树/组件的顶点数量）。例如，图8(b)显示了后向边(v, u)的顺序为，顶点被重新P标记为后向序号。显然，（v，u）的等级给出了$post$（v）。

该算法正确地计算了所有顶点v的$pre(v)/post(v)$，因为树上的每个顶点v（根s除外）都正好有一个由前向/后向边$(u, v)/(v, u)$定义的父母u。最后，BPPA的证明直接来自于以下事实：两个欧拉旅行和列表排名可以通过BPPA来计算。

5.2.3 祖先-后裔查询

在案例2中，为了构建G的边*，我们需要决定两个顶点u和v在生成树T中是否有祖先-后裔关系。

让$pre(v)$为v的前序数，$nd(v)$为v在树上的后代数。我们表明，如果$pre(v)$和$nd(v)$对树中的每个顶点v都是可用的，那么祖先-后代的查询可以在$O(1)$时间内得到回答。给定u和v，按照预排序的定义，我们有：u是一个
如果$pre(u) \le pre(v) < pre(u) + nd(u)$，则为$v$的祖先。对于顶点1在图8(a)中，我们有$pre(1) = 1$和$nd(1) = 3$。因此，任何$1 \le pre(v) < 1+3$的顶点$v$（即顶点1、2和3）是一个的等级给出了$pre$（$v$）。

为了计算$post(v)$，我们通过对每个前向边e设置$val(e)=0$，对每个后向边e^t设置$val(e^t)=1$来运行列表排名。然后，对于每条后向边$e^t = (v, u)$，我们得到顶点v的$post(v) = sum(e^t)$。我们为树根s设置$post(s) = n$ 1。其中n是树中顶点的数量。如果n不知道。

顶点1的后代。

我们在第5.2.2节中介绍了一个计算$pre(v)$的BPPA。现在我们表明，$nd(v)$也可以通过同样的过程得到。对于每条前向边$e = (u, v)$，我们设置$nd(v) = sum(e^t) - sum(e) + 1$，其中$e^t$是后向边（$v$, u）。例如，对于图8(a)中的顶点1，我们计算$nd(1) = sum(1, 0) - sum(0, 1) + 1 = 3 - 1 + 1 = 3$。

5.2.4 案例3 条件检查

有了前面几个小节中提出的PPA/BPPA，第5.1节中的案例1和案例2现在可以通过与邻居交换信息，在恒定数量的超级步骤中得到检查。然而，案例3的处理要复杂得多，因为涉及到直接邻居以外的顶点。

我们现在开发一个处理案例3的PPA，如下所示。我们首先需要为每个顶点v计算另外两个字段，这些字段被递归地计算，如下所示。

> $min(v)$: (1)$pre(v)$, (2)$min(u)$的最小值，为所有的v的子女u，以及(3)$pre(w)$为所有非树的边（v，w）。
>
> $max(v)$：(1)$pre(v)$, (2)$max(u)$的最大值，对所有的v的子女u，以及(3)$pre(w)$为所有非树的边（v，w）。

设$desc(v)$为v的后代（包括v本身），$\Gamma_{desc}(v)$为通过非树边与$desc(v)$中的任何顶点相连的顶点集。直观地说，$min(v)$（或$max(v)$）是$desc(v) \cup \Gamma_{desc}(v)$中最小（或最大）的前序数。

在情况3中，(x, y)存在，如果$x \in \Gamma_{desc}(u) - desc(v)$。由于$x$不是$p(u)$的后代，要么$pre(x) < pre(p(u))$，这意味着$min(u) < pre(p(u))$；要么$pre(x) \geq pre(p(u)) + nd(p(u))$这意味着$max(u) \geq pre(p(u)) + nd(p(u))$。综上所述，如果$min(u) < pre(p(u))$或$max(u) \geq pre(p(u)) + nd(p(u))$，情况3对$u$成立。

当每个顶点v的$pre(v)$、$nd(v)$、$min(v)$和$max(v)$都可用时，所有这三种情况都可以用$O(1)$次超级步骤来处理。我们现在展示如何通过PPA在$O(\log n)$个超级步骤中为每个v计算$min(v)$（计算$max(v)$是对称的）。

为了便于表述，我们只用v来表示$pre(v)$。我们进一步定义*局部*(v)为v和所有的

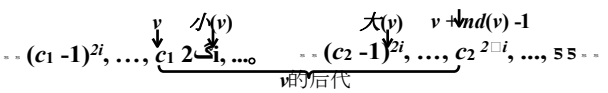


图9：计算*min*(*v*)的插图

通过一条非树边与*v*相连的邻居。注意，*min*(*v*)是只是*v*的所有后代*u*中*local*(*u*)的最小值。

我们在*O*(log *n*)轮中计算*min*(*v*)。在*第i*轮开始时，每个顶点*v* = *c* 2^*i*（*c*是一个自然数）保持一个*global*(*c*2^*i*) = min{*local*(*u*) : *c* 2^*i* ≤ *u* < (*c*+1)2^*i*}。然后在第(*i*+1)轮，对于每个顶点*v* = *c* 2^*i*+1，我们可以简单地更新*global*(*v*) = min{*global*(*v*), *global*(*v* + 2^*i*)}，即合并两个长度为2的连续段的结果。这里，每一轮可以通过一个三超步的PPA完成。(1)每个*v*向(*v*+2^*i*)请求*global*(*v*+2^*i*)；(2)(*v*+2^*i*)通过向*v*发送*global*(v+2^*i*)来重新响应；(3)*v*收到*global*(*v*+2^*i*)来更新*global*(*v*)。最初，对于每个*v*来说，*global*(*v*)=*local*(*v*)，*local*(*v*)可以通过类似的方式计算，即从每个通过非树边连接到*v*的邻居*u*那里请求*pre*(*u*)。

给定一个顶点*v*，*v*在*T*中的后代是*v*, *v* +1, {. ., *v* + *nd*(*v*) 1}。在*第*i轮开始时，我们定义*little*(*v*)（分别为*big*(*v*)）为第一个（分别为最后一个）是2的倍数的后裔。图9说明了*little*(*v*)和*big*(*v*)的概念。

我们为每一轮保持以下不变性。

$$min(v) = \min\{local(u) : u \in [v, little(v)) \cup [big(v), v + nd(v) - 1]\}。 \quad (1)$$

显然，当*little*(*v*)时，*min*(*v*)的正确值被计算出来。= *big*(*v*)，当*i*从0到log₂ *n*时，*i*的某个值必须发生。我们对*第i*轮中的每个*v*进行以下操作，这样可以保持方程（1）给出的不变性。

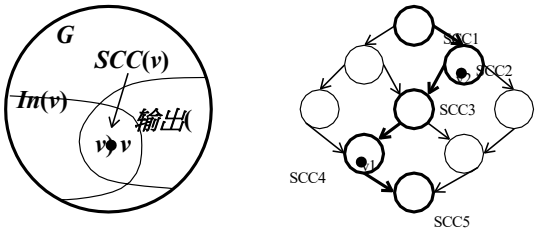
- 如果*little*(*v*) < *big*(*v*) 并且*little*(*v*) 不是2的倍数 *i*+1，设*min*(*v*) = min{*min*(*v*), *global*(*little*(*v*))}，然后设置*little*(*v*)= *little*(*v*)+2^*i*。
- 如果*little*(*v*)< *i*+1 *big*(*v*)，并且*big*(*v*)不是2的倍数，设*min*(*v*)=min{*min*(*v*), *global*(*big*(*v*) - 2^*i*)}，然后设*big*(*v*)= *big*(*v*) - 2^*i*。

如果*little*(*v*)（或*big*(*v*)）是2的倍数，我们就不更新*i*+1，这样在第(*i* + 1)轮中，它就与*c* 2^*i*+1保持一致。我们通过类似于更新*global*(*v*)的Pregel操作来更新*little*(*v*), *big*(*v*), 和*min*(*v*)。

5.2.5 构建块的整合

再次参考第5.1节，当计算*G** 的CC时，我们只需要考虑*G**中对应于*T*中的树边的那些顶点。这是因为*G**的其他顶点对应于案例1的非树边*e*₂，因此以后可以分配给相应*e*₁的CC。

现在我们将所有的构件整合到一个用于计算BCC的PPA中。该算法由一连串的PPA任务组成：(1)*HashMin*：使用第4.1节的BPPA计算所有*v* ∈ *V*的*color*(*v*)；(2)*BFS*或*S-V*：使用第5.2节的BPPA计算*G*的生成森林。来源为 *s* ∈ *V*，*color*(*s*) = *s*；或者，我们可以使用第4.2节的S-V算法获得生成林，用*S-V* al-gorithm for CC computation; (3)*EulerTour*：使用第5.2.2节的BPPA从生成林中构造Euler游；(4)*ListRank1*：将每个Euler游分解为一个列表，并对其进行排序。



(a) *In*(*v*), *Out*(*v*) & *SCC*(*v*)　　(b) DAG路径

图10：SCC计算中的概念

(5)*ListRank2*：使用边的前向/后向标记，用列表排名法计算每个*V* 的 *pre*(*v*)和*nd*(*v*)（见第5.2.2节）；(6)*MinMax*：使用第5节的PPA计算每个*V* 的*min*(*v*)和*max*(*v*)。2.4；(7)*AuxGraph*：使用*pre*(*v*)、*nd*(*v*)、*min*(*v*)和*max*(*v*)信息构造*G*；这是一个恒定超级步骤的BPPA，因为所有三种边缘构造的情况都可以在恒定数量的超级步骤中检查。

(8)*HashMin2*或*S-V2*：使用*HashMin*计算*G*的CC*。但只考虑树形边缘；或者，我们可以使用*S-V*

(9)*Case1Mark*：使用Case1来决定*G*中非树边的BCCs*。

这个算法是一个PPA，因为其每个任务都是一个BPPA/PPA。

6. 强连接组件

在这一节中，我们提出了两种新的Pregel算法，从有向图*G* = (*V*, *E*)中计算*出强连接*组件（**SCC**）。设SCC(*v*)为包含*v*的SCC，设*Out*(*v*)(和*In*(*v*))为*G*中能从*v*到达的顶点集合(分别为能到达*v*的顶点)。*SCC*(*v*) = *Out*(*v*) ∩*In*(*v*)（见图10(a)）。他们计算出

这个过程在$G[Out(v)\ SCC(v)]$、$G[In(v)\ SCC(v)]$和$G[V\ (Out(v)\ In(v))]$。

其中$G[X]$表示由顶点集X诱导的G的子图。正确性由以下属性保证：任何剩余的SCC必须在这些子图中。

我们的贡献。将PRAM算法转化为Pregel算法是很困难的。因此，我们设计了两个基于标签传播的Pregel算法。第一种算法传播每个顶点迄今所见的最小顶点（ID），而第二种算法传播多个源顶点以加快SCC的计算。我们还注意到Pregel[17]中一个非常新的计算SCC的算法，它与我们的第一个算法有着类似的想法。然而，他们的算法只执行了一轮标签传播，然后由主机器进行串行计算，这被称为FCS（Finishing Computations Serially）。对于处理大型图来说，经过一轮计算后的剩余图有可能仍然太大，无法装入单台机器的内存。与此相反，我们引入了图的去构成，这使得我们可以运行多轮的标签预处理。此外，我们在第6.1节末尾描述的优化2执行了类似于[17]中FCS的处理，但它是完全分布式的（通过利用递归图分解的特性），而不是在主机器上计算。最后，我们的第二个SCC算法所使用的想法是新的，它克服了我们第一个算法的一个弱点。

在描述我们的SCC算法之前，我们首先介绍一个用于图分解的BPPA，它被用在我们的算法中。

图分解。给定V的一个分区，用V_1, V_2, ..., V_k表示，我们用两个超级步骤将G分解为$G[V_1], G[V_2], ..., G[V_k]$（假设每个顶点$v$包含一个标签$i$ in-
告知$v \in V_i$)：(1)每个顶点通知其所有内邻和
邻居关于其标签i*的信息*；(2)每个顶点检查进入的
信息，删除标签与自己的标签不同的顶点之间的边，并投票决定是否停止。

6.1 最小标签算法

我们首先描述*最小标签*传播的Pregel操作，其中每个顶点v保持两个标签min_f（v）和min_b（v）。

前向最小标签传播。(1)每个顶点v初始化min_f (v) = v，将min_f
(v)传播给所有v的外邻，并投票停止；(2)当一个顶点v收到一组来自其
in-neighbors $I_{in}(v)$，让min^*为收到的最小信息，那么如果$min^* < min_f(v)$，v更新min_f (v) = min^*，prop-
将min_f
(v)发送给所有v的外邻；v最后投票决定停止。我们重复步骤（2），直到所有顶点都投票决定停止。

后向最小标签传播。这个操作是在前向最小标签传播之后进行的。不同之处在于：(1)最初，只有满足v = min_f (v)的顶点是活跃的，min_b (v) = v，而对于其他顶点u，min_b (u) = ∞；(2)每个活跃顶点v向所有v的近邻传播min_b (v)。

这两种操作都是具有$O(\delta)$超级步骤的BPPAs。经过前向和后向的最小标签传播，每个顶点v*都*有一个标签对（min_f
（v），min_b
(v））。这种标签具有如下性质（证明可在[24]的附录B中找到）。

判断力：1.*让*$V_{(i,j)} = \{v \mid v \in V \wedge (min_f(v), min_b(v)) = (i, j)\}$*。那么，(i)任何SCC都是某个*$V_{(i,j)}$ *的子集，(ii)*$V_{(i,i)}$ *是一个具有颜色*i*的SCC。*

最小标签算法重复以下操作：（1）前向最小标签传播；（2）后向最小标签传播。
(3)一个聚合器收集标签对(i, j)，并给每个$V_{(i,j)}$*身*
分；然后进行图的分解，重新移动跨越不同$G[V_{ID}]$的
边；最后，我们把每个Vtex v用标签（i，i）来表示它的SCC被发现。

在每一步中，只有未标记的顶点是活跃的，因此，一旦它的SCC被确定，顶点就不会参与以后的回合。

该算法的每一轮都会完善前一轮的顶点划分。由于所有的三个步骤都是BPPA，所以每一轮最小标签算法都是一个BPPA。一旦所有顶点都被标记，该算法就终止了。

该算法的正确性直接来自于定理1。我们现在分析一下最小标签算法所需的回合数。给定一个图G，如果我们把每个SCC收缩为一个超级顶点，我们得到一个DAG，其中一条边从一个超级顶点（代表一个SCC，scc_i）指向另一个超级顶点（代表另一个SCC，scc_j），只要有一条边从scc_i的某个顶点指向scc_j的某个顶点。假设是DAG中最长的路径长度，那么我们有以下约束（证明可以在[24]的附录B中找到）。

定理1.*最小标签算法的运行时间最多为*L*轮。*

对于SCC_4 -SCC_5 中的任何顶点v，min_f (v) = v_1。由于v_1被选为向后传播的源，对于SCC_1 -SCC_4 中的任何顶点v，min_b (v) = v_1。因此，任何顶点v $\in$ SCC_4 有标签对（v_1，v_1）并被标记。现在考虑scc_4之前的子路径，即scc_1、scc_2、scc_3，并假设v_2 ($v_2 \in$
scc_2是G中第二小的顶点，那么类似的推理表明
即SCC_2，可以发现$V_{(v_1,v_2)}$。这样一来，P很快就被打破了

上述约束是非常宽松的，通常情况下，在一个回合中，每个DAG路径有一个以上的SCC被标记。我们用图来说明。
图10(b)，有一条DAG路径$P=(scc_1, scc_2, scc_3, scc_4, scc_5)$，而$v_1$是$G$中最小的顶点，显然。

分为许多子路径，每个子路径都可以并行处理，因此在实践中需要的轮次可以远远少于.

在我们的实现中，我们进一步进行了两项优化。

优化1：去除微不足道的SCC。如果一个顶点v的in-degree或out-degree为0，那么v本身就构成了一个琐碎的SCC，可以直接标记以避免无用的标签传播。我们在每一轮的最小标签传播之前标记琐碎的SCCs。

我们现在描述一个Pregel算法来标记所有内度为0的顶点。最初，每个内度为0的顶点v标记自己，将自己发送给所有外邻，并投票停止。在随后的步骤中，每个顶点v从传入信息中出现的邻居中删除其内联，并检查其内联度是否为0，如果是，该顶点标记自己并将自己发送给所有外邻居。最后，该顶点投票决定停止。我们也在每个超级步骤中以对称的方式标记外度为0的顶点。

该算法在实践中只需要少量的超级步骤（如第7节的实验所示），因为现实世界的图（如社交网络）有一个密集的核心，而有限数量的琐碎的SCCs只存在于图的稀疏的边界区域。此外，在每个超级步骤中，许多内度/外度为零的顶点被平行标记为琐碎的SCC。另一方面，移除琐碎的SCC可以防止它们参与最小标签传播，如果一些琐碎的SCC顶点有一个小的ID，这可能会降低算法的有效性。

优化2：提前终止。我们不需要运行算法，直到所有顶点都被标记为发现的SCC的一个顶点。如果v_{ID}小于阈值τ，我们也会标记一个顶点v

$G[v_{ID}]$，这样子图$G[v_{ID}]$中的所有顶点在以后的几轮中仍然是无源的。这里，v_{ID}是使用步骤（3）的聚合器得到的。一旦所有顶点都被标记为SCC/子图顶点，我们就停止。然后，我们使用一轮MapReduce将子图分配给不同的机器，直接从每个子图$G[v_{ID}]$计算SCC。由于每个子图都很小，它的SCCs可以在一台机器上使用高效的主内存算法进行计算，不需要机器间的通信。

6.2 多标签算法

现实世界的图通常有一个巨大的SCC，它包含了大部分的顶点，最好能尽早找到这个巨大的SCC（例如，在第一轮），这样我们就可以通过应用上面提到的优化2提前终止。然而，最小标签算法可能无法在第一轮找到巨型SCC。例如，假设巨型SCC是scc_{max}，那么如果有一个顶点v/scc_{max}的ID小于scc_{max}中的所有顶点，并且如果v链接到scc_{max}中的一个顶点，那么scc_{max}就不能在第一轮被最小标签算法找到。另一方面，本小节要介绍的多标签算法几乎总能在第一轮找到巨型SCC。

多标签算法旨在通过平行传播k个源顶点来加快SCC发现和图的分解，而不是像最小标签算法和现有算法[9, 3, 2, 17]那样只随机挑选一个源。

在这个算法中，每个顶点v保持两个标签集$Src_f(u)$和$Src_b(u)$。该算法与最小标签算法类似。

除了最小标签传播操作被替换为
下面描述的*k-label*传播操作。

前向*k-*标签传播。假设当前的顶点分区为V_1，V_2，...，$V_{\mathcal{L}}$。(1)在超级步骤1中，一个聚合器从每个子图$G[V_i]$中运行地选择k个顶点样本。(2)在超级步骤2中，每个源u初始化$Src_f(u) = u$，并向其所有的外邻居propagates label u，而每个非源顶点v初始化$Src_f(v) = \{\}$。最后，顶点投票决定停止。(3)在随后的超级步骤中，如果一个顶点v从一个内邻收到一个标签u $Src_f(v)$，它就更新$Src_f(v) = Src_f(v)\ u$，并将u传播给所有外邻，然后投票停止。

后向K标签的传播是对称的。与最小标签算法不同的是，后向传播是在前向传播之后进行的，在多标签算法中，我们并行地进行前向和后向传播。

k*标签*传播操作也是一个BPPA，有$O(\delta)$个步骤，当它终止时，每个顶点v得到一个标签对（$Src_f(v)$, $Src_b(v)$）。这个标签有以下属性（证明可以在[24]的附录B中找到）。

悖论2。*设$V_{(s_f,s_b)} = \{v \in V : (Src_f(v), Src_b(v)) = (S_f, S_b)\}$。那么，(i)任何SCC都是某个$V_{(s_f,s_b)}$的子集，(ii)如果$S_f \cap S_b \neq \emptyset$，则$V_{(s_f,s_b)}$是一个SCC。*

我们现在分析一下所需的回合数。在任何一轮中，我们有$\mathcal{L}$个子图，因此大约有$\mathcal{L}k$个源顶点。由于我们不知道$\mathcal{L}$，我们只给出一个非常松散的分析，假设$\mathcal{L}$=1（即只有k个来源）。此外，我们假设ver-
钉子被标记只是因为它们形成了一个SCC，而在实践中，第6.1节的优化2也被应用于标记SCC的顶点。
足够小的子图。
假设我们可以将$(1-\theta)n$个顶点标记为SCC顶点，在那么，在i轮之后，该图有
θ^i n个顶点，并且在$O(\log_{1/\theta} n)$轮的图形是足够的
小，以允许有效的单机SCC计算。我们现在研究θ和k之间的关系。

假设G中存在c个SCC：scc_1，scc_2，....，scc_c。设n_i为scc_i中的顶点数量，$p_i = n_i/n$。我们分析在一轮之后，预期有多少顶点被标记。注意，如果x个采样的源顶点属于同一个SCC，那么我们实际上浪费了$x-1$个样本。我们的目标是要证明这种浪费是有限的。

我们定义一个随机变量X，指的是被标记的ver-
tices的数量。我们还为每个SCC scc_i定义了一个指标变量X_i，如下所示。如果至少有一个样本属于scc_i，$X_i=1$，否则$X_i=0$。让s_j为第j个样本。我们有

$$E[X_i] = Pr\{X_i = 1\} = 1 - \prod_{j=1}^{k} Pr\{s_j \notin SCC_i\}$$

$$= 1 - \prod_{j=1}^{k} (1 - p_i) = 1 - (1 - p_i)^k。$$

请注意，$X = \sum_{i=1}^{c} n_i - X_i$。根据线性的
预期，我们有

$$\sum_{i=1}^{c} \quad \sum_{i=1}^{c} \quad k$$

数据	类型	|V|	|E|
BTC	不定向的	164,732,473	772,822,094
LJ-UG	不定向的	10,690,276	224,614,770
脸书	不定向的	59,216,214	185,044,032
美国	不定向的	23,947,347	58,333,344
欧元	不定向的	18,029,721	44,826,904
推特	定向	52,579,682	1,963,263,821
LJ-DG	定向	4,847,571	68,993,773
博克	定向	1,632,803	30,622,564
Flickr	定向	2,302,925	33,140,017
专利	定向	3,774,768	16,518,948

图11：数据集

换句话说，$\theta = \sum_{i=1}^{c} p_i(1 - p_i)^k$。由于剩余未标记的顶点数量为$\theta n$，我们希望$\theta$越小越好。事实上，如果SCC的大小是有偏差的，那么θ就很小。这是因为如果有一个非常大的SCC，它的一些顶点很可能被采样为源顶点，因此许多顶点会被标记为SCC的一个顶点。

最坏的情况发生在所有的SCC都是同等大小的时候，即对所有的i来说$p_i = 1/c$，在这种情况下$\theta = (1 - 1/c)^k$。由于$(1 - 1/c) < 1$，θ随k而减少，但减少的速度取决于c。例如，当c=1000时，为了得到θ=0.9，我们需要设置k=100。然而，我们注意到，现实世界的图很少有所有的SCC都有类似的大小，而且这个分析是非常松散的。在实践中，即使是非常大的c，k也可以小得多。

我们现在提出一个定理，将上述讨论正式化（证明可以在[24]的附录B中找到）。

THEOREM 2. *If $p_i < \frac{2}{k+1}$ for all i, then $\theta \leq (1 - 1/c)^k$.*

否则，$\theta = \sum_{i=1}^{c} p_i(1-p_i)^k < 1 - 1/k$.

最后，我们强调，定理2是非常宽松的：当存在一个p_i远大于$\frac{2}{k+1}$的SCC scc_i时，θ是非常宽松的。小于
$1 - 1/k$.

7. 实验评价

我们评估了我们的算法在大型真实世界图上的性能。我们在一个由16台机器组成的集群上运行所有实验，每台机器有24个处理器（两个英特尔至强E5-2620
CPU）和48GB内存。一台机器被用作主机器，只运行一个工作进程（或简单地说，工作者），而其他15台机器作为从机器，每台机器运行10个工作者。集群中任何一对节点之间的连接是1Gbps。

我们所有的算法都是在Pregel+[2]
，它是Pregel的一个开源实现，尽管任何类似Pregel的系统都可以用来实现我们的算法。我们注意到，我们没有在Pregel+中使用任何优化技术，也就是说，我们使用了
由于我们的目的是测试我们的算法在类似于Pregel的一般系统中的表现，所以只介绍了Pregel的基本特征。本文讨论的所有算法的源代码都可以找到
在http://www.cse.cuhk.edu.hk/pregelplus/download.html。

数据集。我们使用了10个真实世界的图数据集，这些数据集在图11中列出：(1)*BTC*[3]：一个从BTC中转换的语义图。2009年三国挑战赛的RDF数据集[5]；(2)*LJ-UG*[4]
：一个由中国人组成的网络。
LiveJournal用户和他们的团体成员；(3)*Facebook*[5]
：Facebook社交网络的友谊网络；(4)*USA*：[6]

$$E[X] = \sum_{i=1}^{c} n_i - E[X_i] = \sum_{i=1}^{c} [n_i - n_i (1 - p_i)]$$
$$= n - \sum_{i=1}^{c} n_i (1 - p_i)^k = n - n \sum_{i=1}^{c} p_i (1 - p_i)^k .$$

[2] http://www.cse.cuhk.edu.hk/pregelplus

[3] http://km.aifb.kit.edu/projects/btc-2009/

[4] http://konect.uni-koblenz.de/networks/livejournal-groupmemberships

[5] http://konect.uni-koblenz.de/networks/facebook-sg

[6] http://www.dis.uniroma1.it/challenge9/download.shtml

	BTC			美国		
	普瑞格+	图形实验室		普瑞格+	图形实验室	
		同步	异步		同步	异步
#超级步骤的数量	30	30	不适用	6262	6262	不适用
计算时间（秒）	32.24	83.1	155	1011	2982	627

图12：Pregel+和GraphLab运行Hash-Min

美国公路网；(5)*Euro*[7]：欧洲公路网；(6)*Twit-ter*[8]。基于2009年快照的Twitter who-follows-who网络；(7)*LJ-DG*[9]：LiveJournal博客社区的友谊网络；(8)*Pokec*[10]：Pokec社交网络的友谊网络；(9)*Flickr*[11]：Flickr social网络的友谊网络；(10)*Patent*[12]：美国专利引证网络。

7.1 与GraphLab的性能比较

如第2节所述，GraphLab[11]（和PowerGraph[10]）只允许一个顶点访问其相邻顶点和边的状态。因此，它不支持顶点需要与非邻居通信的算法，如第4.2节的S-V算法、第5.2.2节的列表排名算法和第5.2.4节的算法。因此，我们不能在GraphLab中实现我们的BCC算法。此外，我们也不能在GraphLab中实现我们的SCC算法，因为GraphLab不支持图的突变，而我们的SCC算法中的图分解操作涉及边的删除。

由于上述限制，我们只比较了GraphLab与Pregel+在Hash-Min算法上的性能。我们使用GraphLab 2.2，它包括PowerGraph[10]的所有功能，并同时运行GraphLab的异步和同步模式。GraphLab的同步模式可以模拟Pregel，但由于上述的限制，在其同步模式下也很难实现S-V、BCC和SCC算法。

图12报告了Pregel+和GraphLab在小直径*BTC*图（学位分布偏斜）和大直径*美国*公路网（其中所有顶点的学位都很小）上运行Hash-Min时的性能。结果显示，Pregel+在处理小直径*BTC*图时明显比GraphLab快。对于大直径的*美国*图，Pregel+几乎比同步GraphLab快3倍，但比异步GraphLab慢1.6倍（原因见下文）。

对于像*美国*这样的大直径图，异步执行比其同步模式要快。这是因为在异步执行中，对$min(v)$的更新对所有其他顶点都是立即可见的，而在同步执行中，更新只在下一个超级步骤中对其他顶点可见，导致收敛速度较慢。然而，对于像*BTC*这样的小直径图，同步执行只在30个超级步骤中收敛；因此，即使异步模式可以更快地收敛，其收益也不足以覆盖异步执行中的数据锁定/解锁的开销。

总的来说，Pregel+在具有倾斜度分布的小直径图上的表现要好得多，而Pregel+在大直径图上的表现也是合理的，这证明了

[7] http://www.dis.uniroma1.it/challenge9/download.shtml
[8] http://konect.uni-koblenz.de/networks/twitter_mpi
[9] http://snap.stanford.edu/data/soc-LiveJournal1.html
[10] http://snap.stanford.edu/data/soc-pokec.html
[11] http://konect.uni-koblenz.de/networks/flickr-growth
[12] http://snap.stanford.edu/data/cit-Patents.html

任务	BTC		LJ-UG		脸书	
	#的阶梯	汇编。时间	#的阶梯	汇编。时间	#的阶梯	汇编。时间
HashMin	30	32.24 s	18	11.85 s	16	37.10 s
BFS	31	20.56 s	19	8.61 s	17	10.14 s
S-V	86	449.97 s	58	142.24 s	72	337.02 s
欧拉之行	3	14.26 s	3	2.26 s	3	7.56 s
榜单排名1	49	544.71 s	53	97.98 s	57	408.79 s
列表等级2	49	541.86 s	53	98.59 s	57	411.17 s
最小最大值	46	35.88 s	50	8.54 s	54	20.42 s
AuxGraph	4	58.05 s	4	21.94 s	4	22.52 s
HashMin2	34	42.91 s	11	21.04 s	16	43.31 s
S-V2	72	443.16 s	58	138.59 s	86	385.86 s
Case1Mark	4	35.62 s	4	20.83 s	4	13.27 s
总时间 (CC by HashMin)	1326.09 s		291.64 s		974.28 s	
总时间	2123.51 s		530.97 s		1606.61 s	

图13：BTC、LJ-UG和Facebook上的CC/BCC性能

Pregel+是实现我们算法的一个很好的分布式图计算系统的选择。此外，上面讨论的GraphLab的局限性使得使用GraphLab实现某些类别的图算法很困难，这进一步证明了在我们的工作中采用Pregel+的合理性。

最后，我们要说明的是，本文的重点是类Pregel系统的实用算法，而不是系统本身，我们请读者参考我们的在线报告，对现有系统进行全面比较，包括GraphLab、Giraph[1]、GPS[16]和Pregel+：
`http://www.cse.cuhk.edu.hk/pregelplus/expTR.pdf`。

7.2 CC和BCC算法的性能

在第5节中，我们为一系列的基本图问题提出了PPAs/BPPAs。由于它们在PPA中被用作计算BCC的构件，我们也将它们的性能结果作为BCC计算的步骤进行报告。回顾第5.2.5节，BCC计算中的PPA任务序列包括。(1)-(2)*HashMin* + *BFS*，或者*S-V*；(3)*EulerTour*；(4)*ListRank1*；(5)*ListRank2*；(6)*MinMax*；(7)*AuxGraph*；(8)*HashMin2*或者*S-V2*；(9)*Case1Mark*。其中，任务(1)和(8)也报告了我们在第4节中描述的计算CC的两个PPA的性能。

我们在图13中报告了BCC计算在三个*小直径*图*BTC*、*LJ-UG*和*Facebook*上的每任务性能。由于图的直径小，Hash-Min在这些图上的效率比S-V高得多。例如，Hash-Min在*LJ-UG*上以18个超级步骤完成，只用了11.85秒。相比之下，S-V需要58个超级步骤和142.24秒。因此，结果证明，当图的直径较小时，使用Hash-Min计算CC是更有效的。我们还给出了我们的BCC算法的总运算时间，结果再次表明，在BCC计算中使用Hash-Min作为构建块，比使用S-V的总时间几乎短了一倍。

接下来，我们在图14中报告了BCC计算在*美国*和*欧洲两个大直径*道路网络上的单任务性能。由于图的直径很大，Hash-Min非常耗时。例如，它在*美国*需要1011.19秒和6262个步骤。相比之下，S-V只需要198个超级步骤和368.20秒。这再次显示了$O(\delta)$-超级步骤PPA（例如Hash-Min）和$O(\log n)$-超级步骤PPA之间的区别(例如，S-V）。类似的行为也被观察到，在G^*上的CC计算，HashMin2在*美国*使用了5437.72秒，而

任务	美国		欧元	
	# 步数	汇编。时间	# 步数	汇编。时间
HashMin	6262	1011.19 s	4896	692.39 s
BFS	6263	964.11 s	4897	639.78 s
S-V	198	368.20 s	212	340.19 s
欧拉之行	3	3.04 s	3	2.42 s
榜单排名1	55	203.05 s	55	165.89 s
列表等级2	55	197.78 s	55	160.49 s
最小最大值	52	16.65 s	52	12.77 s
AuxGraph	4	12.29 s	4	9.89 s
HashMin2	7365	5437.72 s	2836	2935.28 s
S-V2	226	536.69 s	184	414.20 s
Case1Mark	4	2.90 s	4	1.94 s
总时间 (CC by HashMin)	7848.73 s		4620.85 s	
总时间 (CC by S-V)	1340.60 s		1107.79 s	

图14：CC/BCC在美国和欧洲的表现

S-V2只用了526.69秒。总的来说，基于S-V的BCC算法比基于Hash-Min的BCC算法在*美国*要快5.85倍，在*欧洲*要快4.17倍。这表明了我们的S-V算法在处理大直径图方面的优势。可能有人会说，在Pregel中计算道路网络的CC并不重要，因为道路网络通常是连通的，而且不是非常大。然而，有些空间网络规模巨大，比如模拟地形的三角不规则网络（TIN），当我们要计算给定的特定海平面的岛屿时，CC的计算就很有用。另外，CC计算是我们的PPA中计算BCC的一个关键构件，找到空间网络的BCC对于分析其弱点非常重要。连接点。

7.3 SCC算法的性能

现在我们报告我们在有向图上计算SCC的最小标签和多标签算法的性能。

7.3.1 最小标签算法的性能

在描述结果之前，我们首先回顾一下我们的最小标签算法在每一轮中执行的任务顺序。(1)*Opt 1*：这项任务按照第6节的优化1所述，删除琐碎的SCC。1；(2)*MinLabel*：前向最小标签传播，然后是后向最小标签传播；(3)*GDecom*。这个任务使用一个聚合器来收集标签对（min_f *身份证* (u),*身* $min_b(u)$）并给每个标签对分配一个新的，根据u的（$min_f(u)$, $min_b(u)$）设置每个顶点u的，用$min_f(u) = min_b(u)$标记每个顶点u在一个SCC中，并使用第6节开头所述的算法进行图分解。回顾一下，如果一个子图的大小（由顶点数量决定）小于用户定义的阈值τ，我们就不对其进行分解。

我们首先计算最大的图*Twitter*的SCC，其中我们只在一个顶点的SCC被确定时（即τ=0）标记它。图15报告了超级步骤的数量和每个任务的计算时间。最后一列 "最大尺寸"显示了最大的
如图15所示，最小标签算法只需要4轮就能计算出*Twitter*上的所有SCCs，这表明
这表明，在实践中，最小标签算法所需的时间远远少于定理1中给出的回合数。此外，由于图的直径较小，最小标签的推广操作只需要很少的超级步骤。例如，在第一轮中，前向传播只需要15个超级步骤，然后是14个超级步骤的后向传播。

圆的	任务	# 步数	汇编。时间	最大尺寸
1	**选择1**	18	9.41 s	238,986
	最小标签	15 + 14	75.80 s	
	GDecom	3	76.86 s	
2	**选择1**	5	0.81 s	22
	最小标签	36 + 75	18.73 s	
	GDecom	3	1.74 s	
3	**选择1**	3	0.46 s	2
	最小标签	6 + 5	2.02 s	
	GDecom	3	0.44 s	
4	**选择1**	1	0.21 s	0
	最小标签	3 + 3	0.96 s	
	GDecom	3	0.50 s	
共计			187.94 s	

图15：Twitter上的最小标签性能（τ=0）。

圆的	任务	# 步数	汇编。时间	最大尺寸
1	**选择1**	16	2.73 s	51,697
	最小标签	14 + 16	14.91 s	
	GDecom	3	7.24 s	
2	**选择1**	4	0.34 s	81
	最小标签	8 + 9	1.91 s	
	GDecom	3	0.72 s	
3	**选择1**	3	0.22 s	29
	最小标签	6 + 7	1.08 s	
	GDecom	3	0.40 s	
4	**选择1**	1	0.14 s	12
	最小标签	4 + 4	1.01 s	
	GDecom	3	0.32 s	
共计			31.02 s	

图16：LJ-DG的最小标签性能（τ = 0）。

传播。对于一个有近20亿条边的图来说，总的计算时间只有187.94秒，这是非常高效的。

请注意，在图15中的第二回合之后，最大的未标记子图的大小仅为22。因此，另一个选择是将这些小的子图分布到不同的机器上，使用MapReduce进行单机SCC计算。因此，我们也在*Twitter*上运行我们的最小标签算法，使用τ=50,000，这样，一旦一个子图包含的顶点少于50,000，就会被标记以避免进一步分解。所有的顶点在2轮之后都被标记了，其表现与图15中的相似。然后我们运行一个MapReduce作业来计算被标记子图的SCC，这需要199秒。

对于三个相对较小的图，*Pokec*、*Flickr*和*Patent*，τ=0的最小标签算法在不到4轮的时间内找到了所有的SCC，分别用了18.09、17.88和2.34秒。由于篇幅有限，详细的结果在附录C[24]中报告。然而，我们注意到，τ=0的min-label算法并不总是能够找到任意图的所有SCC。一个例子是*LJ-DG*数据集，其性能如图16所示（仅显示前4轮的结果）。事实上，在随后的三轮中，"最大尺寸"缓慢下降为11、10和9。相反，在*LJ-DG*上运行min-label算法，使用τ=50,000，只需要2轮就可以标记所有顶点，然后是一个MapReduce作业，在27秒内计算出标记子图的SCC。

7.3.2 多标签算法的性能

我们现在报告我们的多标签算法的性能。对于并行的前向和后向k-*label*传播，我们固定k = 10.我们还设置了τ=50，000，顶点少于50，000的子图不做进一步分解。多标签算法在*Twitter*和*LJ-DG*这两个大图上的表现是

圆的	任务	#步数	汇编。时间	最大尺寸
1	选择1	18	8.07 s	238,986
	多标签	17	423.85 s	
	GDecom	3	102.98 s	
2	选择1	5	0.81 s	206,319
	多标签	5	0.55 s	
	GDecom	3	0.41 s	
3	选择1	1	0.16 s	206,292
	多标签	6	0.94 s	
	GDecom	3	0.38 s	
绘图还原（MapReduce			181 s	

图17：Twitter上的多标签性能（τ = 50, 000）。

圆的	任务	#步数	汇编。时间	最大尺寸
1	选择1	16	2.66 s	51,697
	多标签	19	27.02 s	
	GDecom	3	6.30 s	
2	选择1	5	0.74 s	50,706
	多标签	6	0.71 s	
	GDecom	3	0.25 s	
3	选择1	1	0.13 s	50,629
	多标签	8	0.80 s	
	GDecom	3	0.36 s	
绘图还原（MapReduce			26 s	

图18：LJ-DG的多标签性能（τ = 50, 000）。

如图17和18所示。我们可以看到，尽管第一轮将最大的无标记子图大小限定为一个相对较小的数字，但 "最大尺寸"在后面的几轮中缓慢下降，我们无法承受运行到小于50, 000的情况。然而，子图足够小，可以分配给不同的机器，使用MapReduce进行本地SCC计算，最后一轮MapReduce的后处理对两个图来说都是高效的。我们对*Pokec*、*Flickr*和*Patent*等数据集也得到了类似的结果，结果见附录C[24]。

与最小标签算法不同的是，我们可以一直运行到终止，多标签算法在每一轮中最多可以找到k 个SCC，而且只有在有大的SCC时，在早期的几轮中才有效。然而，正如第6.2节开头所讨论的，多标签算法几乎总是在第一轮找到最大的SCC，这比最小标签算法更可取。因此，在只需要最大的SCC（也称为巨型SCC）的应用中，多标签算法将是一个更好的选择；在需要所有SCC的应用中，在第一轮运行多标签算法，然后在随后的几轮运行最小标签算法，可能是一个不错的选择。

8. 结论

我们提出了计算三个基本图连接问题的高效分布式算法，即CC、BCC和SCC。具体来说，我们定义了PPA的概念，以设计具有保证性能的Pregel算法，即每次迭代只需要线性空间、通信和计算，并且只需要$O(\log n)$或$O(\delta)$次计算。在大型真实世界图上的实验验证了我们的算法在无共享的并行计算平台上具有良好的性能。

对于未来的工作，我们计划为以块为中心的计算模型定义一类类似于PPA的算法[23]。我们还对开发高效的Pregel算法感兴趣，该算法用于枚举图的子结构，如三角形[7]、矩形[21]和最大悬臂[6]。

鸣谢。我们感谢审稿人给了我们许多建设性的意见，使我们的论文有了很大的改进。这项工作部分是在第一作者在香港科技大学时完成的。这项研究得到了上海交通大学8115048号拨款和香港科技大学FSGRF14EG31号拨款的部分支持。

9. 参考文献

[1] C.Avery.Giraph:大规模的图处理基础设施hadoop.*Hadoop峰会论文集。圣克拉拉*，2011年。

[2] J.Barnat, J. Chaloupka, and J. van de Pol.改进的SCC分解的分布式算法.*Electronic Notes in Theoretical Computer Science*, 198(1):63-77, 2008.

[3] J.Barnat and P. Moravec.寻找隐含给定图中的sccs的并行算法.In *Formal Methods:应用与技术》*，第316-330页。Springer, 2007.

[4] J.Cheng, S. Huang, H. Wu, and A. W.-C.Fu.Tf-label:一个用于大型图中可达性查询的拓扑折叠标签方案。在*SIGMOD会议上*，第193-204页，2013年。

[5] J.Cheng, Y. Ke, S. Chu, and C. Cheng.大图中距离查询的高效处理：一个顶点覆盖方法。在*SIGMOD会议上*，第457-468页，2012。

[6] J.Cheng, Y. Ke, A. W.-C.Fu, J. X. Yu, and L. Zhu.通过h*图寻找大规模网络中的最大集群.在*SIGMOD会议上*，第447-458页，2010年。

[7] S.Chu and J. Cheng.海量网络中的三角形列表及其应用.在*KDD*，第672-680页，2011年。

[8] M.Dayarathna and T. Suzumura.exedra初探：大型图分析工作流程的特定领域语言。在*WWW（配套卷）*中，第509-516页，2013年。

[9] L.K. Fleischer, B. Hendrickson, and A. Pınar.关于识别并行中的强连接组件。在*平行和分布式必理中*，第505-511页。Springer, 2000.

[10] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin.Powergraph:自然图上的分布式图形并行计算。在*OSDI*，第17-30页，2012。

[11] Y.Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein.分布式graphlab。云中机器学习的框架。*pvldb*, 5(8):716-727, 2012.

[12] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N.Leiser, and G. Czajkowski.Pregel：一个用于大规模图形处理的系统。在*SIGMOD会议上*，第135-146页，2010年。

[13] L.Quick, P. Wilkinson, and D. Hardcastle.使用类似pregel的大规模图形处理框架进行社会网络分析。在*ASONAM*，第457-463页，2012。

[14] V.Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma.在map-reduce中以对数轮次寻找连接组件。*icde*, 0:50-61, 2013.

[15] J. H. Reif.深度优先搜索是固有的顺序。*Information Processing Letters*, 20(5):229-234, 1985.

[16] S.Salihoglu and J. Widom.Gps：一个图形处理系统。在*SSDBM*，第22页，2013年。

[17] S.Salihoglu and J. Widom.Pregel-like系统上的图算法优化.*pvldb*，7（7）：577-588，2014。

[18] Y.Shiloach and U. Vishkin.一种o(log n)并行连接算法。*J. Algorithms*, 3(1):57-67, 1982.

[19] Y.Tao, W. Lin, and X. Xiao.最小化的mapreduce算法.在*SIGMOD会议*，第529-540页，2013年。

[20] R.E. Tarjan和U. Vishkin.一种高效的并行双连接性算法.*SIAM Journal on Computing*, 14(4):862-874, 1985.

[21] J.Wang, A. Fu, and J. Cheng.大型双胞胎图中的矩形计数.在*大数据大会上*，2014年。

[22] J.C. Wyllie.并行计算的复杂性。技术报告，康奈尔大学，1979

。
[23] D.Yan, J. Cheng, Y. Lu, and W. Ng.Blogel：现实世界图上分布式计算的以块为中心的框架。*pvldb*, 7(14), 2014.
[24] D.Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu.具有性能保证的图连接问题的Pregel算法.*在线附录, 2014. (www.cse.cuhk.edu.hk/pregelplus/ppaApp.pdf)*.