Piccolo:使用分区表创建快速，分布式程序

摘要

Piccolo 是一个新型的以数据为中心的编程模型，可用于在数据中心编写并行内存应用程序。不同于现有的数据流模型，Piccolo 允许在不同机器上并行运行计算，并通过一个键-值表接口更改状态。Piccolo 使高效执行应用程序成为可能。尤其是，应用程序可以创建本地政策，来利用本地的共享状态访问，和使用用户自定义的累加函数来自动解析 Piccolo 运行时的写-写冲突。

使用 Piccolo，我们已经实现了几个问题域的应用程序，包括 PageRank 算法，k-均值聚类和分布爬虫。使用 100 个 Amazon EC2 实例和一个 12 台机器集群所做的实验显示，在很多问题上，Piccolo 比现有的数据流模型更快，还能提供相似的容错保障和方便的编程接口。

1、介绍

随着数据中心和云端平台的可用性不断增加，来自不同问题域的程序员面临的问题是：如何编写可跨多节点运行的并行应用程序。这些应用程序涉及到：机器学习问题（k-均值聚类、神经网络训练），图表算法（PageRank），科学计算等。许多这些应用程序大量访问和变异共享着内存的居间状态。

在计算机之间并行内存计算是很困难的。由于整个计算被分割为运行在不同计算机上的多线程，就需要在这些计算机中协调这些线程并共享中间结果。例如，要计算网页 p 的 PageRank 值，一个线程需要访问网页 p 的链接页面的 PageRank 值，这可能调用运行在不同计算机内存中的线程。传统的并行内存应用程序，已经使用如 MPI 的消息传递原语来建立。对许多用户来说，这种消息传递提供的以通信为中心的模式过于贴近底层抽象，相较于数据的位置和得到数据的方式，用户更关注数据本身和数据传输。

以数据为中心的编程模型，让用户通过一个简单的接口来访问数据，而没有明确的访问机制，这已经在阐述许多算法时被证实为一个方便、流行的机制。MapReduce 和 Dryad 提供的数据流编程模型并没有显示任何全局共享状态。尽管数据流模型非常适合对磁盘上的数据进行批量处理，但它并不适用于内存计算：应用程序不能在线访问中间状态，并且经常不得不通过加入多个数据流来模拟共享内存访问。分布式共享内存和元组空间允许分享并行内存状态。然而，简单的内存（或元组）模型让程序员难以在分布式环境中对应用程序性能进行更好的优化。

本文介绍了 Piccolo，一个为编写计算机之间的并行内存应用程序而开发的以数据为中心的编程模型。在 Piccolo 中，程序员围绕一系列应用内核函数组织计算，每一个内核在多个计算节点上并发执行多重实例。内核实例使用一套内存表共享并行，变异状态，这些内存表的输入位于不同计算节点的内存中。内核实例使用 get 和 put 原语，单独通过键-值表接口共享状态。底层的 Piccolo 运行时，通过发送消息来读取和修改存储在远程节点内存中的表的输入项。

通过显示全局共享状态，Piccolo 编程模型表现出一些引人注目的特点。第一，它允许自然高效的实现需要共享中间状态的应用程序，如：k-均值计算，n-体模型，PageRank 算法等。第二，Piccolo 使得需要立即访问的在线应用程序修改共享状态成为可能。例如：一个分布爬虫可以通过正在进行的网络爬行的状态更新，快速检测到新页面。

Piccolo 借用了现有的以数据为中心系统的思想，来保证应用程序的高效实施。Piccolo 强制在独立的键-值对上进行原子操作，并且使用用户定义的累加函数自动合并相同键的并发更新（类似于 MapReduce 中的 Reduce 函数）。这两种技术的结合，消除了大多数应用程序所需的细粒度应用级分布。Piccolo 允许应用程序开发本地访问以实现状态共享。用户通过定义分区函数，控制计算机之间的表输入分区方式。基于用户的本地政策，它底层的运行可以安排一个内核实例在所需的表分区处存储，从而减少昂贵的远程表访问。

我们已经建立了一个可运行的系统，它由一台主机（用于协调）和几个工作进程（用于存储内存表分区和执行内核）组成。运行时，使用一个简单的工作窃取启发式方法，来动态负载均衡工作进程中内核程序的执行。Piccolo 提供了一个全局的检查点/恢复机制来修复机器故障。运行时使用 Chandy-Lamport 快照算法，对于运行状态不暂停的活动计算，周期性的生成一致性快照。如果机器出现故障，Piccolo 将使用最近的快照状态，重新开始来恢复计算。

本文的其他内容安排如下。第二部分描述了 Piccolo 编程模型，接着是对 Piccolo 运行时的设计（第三部分）。在第四部分，介绍了使用 Piccolo 建立的一套应用程序。第五部分讨论了 Piccolo 原型的实现。第六部分显示了 Piccolo 的性能评价。在第七部分介绍了相关工作。

2、编程模型

Piccolo 的编程环境表现为一个现存语言库（当前实现了对 C++和 Python 的支持），并且不需要改变底层操作系统或编译器。这一部分描述了编程模型依据什么构建应用程序（2.1 节），通过键-值表共享居间状态（2.2 节），优化本地访问（2.3 节），和从故障中修复（2.4 节）。最后，通过展示怎样在 Piccolo 顶层实现 PageRank 算法作为这一部分的结束。

2.1 程序结构

使用 Piccolo 编写的应用程序由控制函数（Control）和内核函数（Kernel），控制函数在单个计算机上执行，内核函数同时在多台计算机上执行。控制函数创建共享表，启动内核函数的多重实例，并且执行全局同步。内核函数由循式程序组成，它在同步执行的内核示例中通过读取和写入表来共享状态。默认状态下，控制函数在一个单独线程内执行，单独线程是在执行每个内核实例时创建的。而且，程序员可以根据需要自由在控制函数和内核函数中创建附加的应用线程。

内核调用：程序员可以使用 Run 函数来启动一个特定数字（m）的内核实例，并在不同的计算机上执行所指定的内核函数。每一个内核实例都有对应的标识符 0…m-1，可以使用 my_instance 函数重新找回。

内核同步：程序员使用控制函数来调用一个全局障碍，以等待所有的已启动内核完成。当前，Piccolo 不支持在并行的内核实例上进行成对同步算法。因为 Piccolo 共享表接口，使得很多细粒度锁定操作没有进行的必要，这一点使全局障碍非常高效。这个全面的应用程序结构，也就是通过核心函数跨越一个或多个全局障碍启动内核，这使人回想到 CUDA 模型，它同样明确的摒弃对成对线程同步的支持。

2.2 表接口和语义

并行内核实例的跨机器居间状态共享是通过基于内存的键值-来实现的。表输入跨越所有节点传播，并且每个键-值对都位于内存的单一节点上。每个表都与明确的键-值类型相联系，这些键、值可以任意由用户声明串行化类型。如图表 1 所示，这个键值接口提供了统一的访问模型，来判断底层的表输入是存储在本地还是其他计算机上。该表的应用程序界面包括 get，put 等标准操作，以及 Piccolo 的 update，flush，get_itertator 等特定函数。只有控制函数可以创建表；控制函数和内核函数都可以调用表操作。

```
Table<Key, Value>:
  clear()
  contains(Key)
  get(Key)
  put(Key, Value)

  # updates the existing entry via
  # user-defined accumulation.
  update(Key, Value)

  # Commit any buffered updates/puts
  flush()

  # Return an iterator on a table partition
  get_iterator(Partition)
```

Figure 1: Shared Table Interface

图表 1 共享表接口

用户定义累加：多重内核实例可以对相同键发布并行 update。为解决这样的写-写冲突，程序员可以将用户定义累加函数和每个表联系起来。Piccolo 通过在运行时合并相同键的共同更新来执行累加器。如果程序员希望结果独立于更新次序，那么累加器必须是交替且组合的函数。

Piccolo 提供了一套标准累加函数：求和，乘，最小值/最大值。要定义一个累加器，用户可创建四种函数：初始化（Initialize）是为一个新创建的键将累加器初始化，累加（Accumlate）是单独的更新操作合并的结果，合并（Merge）是将多个累加器在相同键上的内容结合起来，视图（View）是返回当前累加器的状态，它反映了迄今为止所有更新的累加。累加器函数没有全局状态访问，除非相应的表输入被更新。

用户控制表分区：Piccolo 使用了用户定义的分区函数（Partition function）将主要空间分区。表分区是表达用户程序本地偏好的核心原语。程序员在创建表时指定分区编号（p）。一个表的 p 分区是由整数 0…p-1 来命名的。内核函数可以使用 get_iterator 函数来扫描所给表分区中的全部输入（详见图表 1）。

Piccolo 没有给程序员透露哪个节点存储表分区，但是保证所有给定分区的表输入存储在相同计算机上。尽管运行时的目标是将计算机上的表分区负载均衡分配，但程序员有责任保证一台单独计算机的可用内存能够存储最大的表分区。这通常可以通过指定使分区的数量远远大于计算机数量来实现。

表语义：从应用程序的角度来看，所有的表操作包括单独的键-值对都是原子的。写操作（如：update，put）被指定在其他计算机上可以缓冲以避免阻塞内核执行。在处理缓冲的远程写入时，Piccolo 提供了以下保障：

- 所有内核实例在相同键上发布的操作，都按它们的发布顺序请求执行。不同内核实例在相同键上发布的操作，按总顺序请求执行。
- 对于一个成功的 flush 操作，所有缓冲写入将由调用者的内核实例完成，它们将被分配至相应的远程位置，并且将会由任意内核实例的随后 gets 反应作为回应。
- 为实现一个全局障碍，所有内核实例将会完成，并且它们的全部写入将会被申请执行。

2.3 表达本地偏好

当写入远程表输入时，可以在本地节点缓冲，这种沟通的潜在因素涉及到引人注意的远程表输入不能被有效隐藏。然而，获得好的应用表现关键，是利用本地访问而最少获取远程 gets。通过分区来组织内核和共享状态的计算，Piccolo 为程序员提供了一个简单的方式来表述本地政策。这样的政策允许底层 Piccolo 在运行时，在一台存储最多所需数据的计算机上执行内核实例，从而最少读取远程数据。

Piccolo 支持两种本地政策：（1）将内核执行与表分区放置在同一位置。（2）将不同表的分区放置在同一位置。在启动一些内核时，程序员可以再 Run 函数中指定一个表内容，来阐述他对于内核执行与相关表放置位置的偏好。程序员通常根据指定表分区的编号，启动相同编号的内核实例。运行时的排序是指在执行第 i 个内核实例的计算机上存储指定表的第 i 个分区。为优化内核使其可以从更多表中读取，程序员需要使用 GroupTables(T1,T2,…)函数重置多个表。运行时分配第 i 分区的表 T1,T2,…存储在同一计算机上。这样得到的结果是，通过将内核执行与一张表放在一起，程序员可以避免为内核从多重表的相同分区内远程读取。

2.4 用户辅助检查与恢复

Piccolo 通过一个全局检查/恢复机制来处理机器故障。这个机制当前还不是完全自动执行的，Piccolo 保存了所有共享表状态的一致性全局快照，但是需要用户保存用于恢复所在位置核心与控制函数执行的额外信息。这个设计保证了合理的交换使用。在实践中，程序

所需要的检查点用户信息相对较少。另一方面，这种设计避免了因为自动执行检查点的C/C++执行程序，而带来的架空与复杂性。

根据编写应用程序的经验，设置了两个检查点应用程序接口：一个是同步的（CpBarrier），一个是异步的（CpPeriodic）。两个函数都调用了一些控制函数。同步检查点非常适合于反复迭代应用（如：PageRank），它可以通过全局障碍在多重循环中分别启动内核，并会在每几个循环中请求保存中间状态。另一方面，有着长时间运行内核的应用程序（如：分布式爬虫）需要使用异步检查点来周期性保存它的状态。

CpBarrier 使用了一系列表和用户数据字典的内容来作为一部分检查点的保存。典型的用户数据包括控制线程中的一些迭代器的值。例如在 PageRank 中，程序员想要记录一些 PageRank 迭代计算的编号作为全局检查点的一部分。CpBarrier 表现为一个全局障碍，并且保证检查点状态在障碍处与执行状态等价。

CpPeriodic 使用了一系列表的内容，一个时间间隔作为检查点周期，以及一个内核反馈函数 CheckpointCallback。这个反馈在节点被检查所分配的表分区状态之后，立刻调用这个节点上所有活动的内核。反馈函数为程序员提供了一条途径，来保存运行内核实例时需要存储的必要数据。通常是在一个处理内核实例的分区上放置迭代器。在存储时，Piccolo 重新装载所有节点的表状态，并且通过在检查点上保存的字典来调用内核实例。

2.5 综合举例：PageRank

以怎样用 Piccolo 实现 PageRank 排名作为具体举例。PageRank 算法需要输入一个松散的网络图表并且计算每一页的排名值。计算发生在多重迭代中：页面 i 的排名值在第 k 次迭代中是将规范性排名和引入的相邻页面的现存迭代加总。例如：（公式未翻译。）

```
tuple PageID(site, page)
const PropagationFactor = 0.85

def PRKernel(Table(PageID,double) curr,
             Table(PageID,double) next,
             Table(PageID,[PageID]) graph_partition):
  for page, outlinks in
      graph.get_iterator(my_instance()):
    rank = curr[page]
    update = PropagationFactor * rank / len(outlinks)
    for target in outlinks:
      next.update(target, update)

def PageRank(Config conf):
  graph = Table(PageID,[PageID]).init("/dfs/graph")
  curr = Table(PageID, double).init(
        graph.numPartitions(),
        SumAccumulator, SitePartitioner)

  next = Table(PageID, double).init(
        graph.numPartitions(),
        SumAccumulator, SitePartitioner)

  GroupTables(curr, next, graph)

  if conf.restore():
    last_iter = curr.restore_from_checkpoint()
  else: last_iter = 0

  # run 50 iterations
  for i in range(last_iter, 50):
    Run(PRKernel,
        instances=curr_pr.numPartitions(),
        locality=LOC_REQUIRED(curr),
        args=(curr, next, graph))

    # checkpoint every 5 iterations, storing the
    # current iteration alongside checkpoint data
    if i % 5 == 0:
      CpBarrier(tables=curr,
                {iteration=i})
    else: Barrier()

    # the values accumulated into 'next' become the
    # source values for the next iteration
    swap(curr,next)
```

图表 2 PagePank 的实施

完整的 PageRank 在 Piccolo 中的实施展示在图表 2 中。输入的网页图表表现为一套输出链接，对每一个页面都有 page→target 指向链接。图表从一个并行文件系统下载到共享内存表（graph）中。那些对于内存来说太大的图表，Piccolo 同样支持使用一个只读 DiskTable 接口读取来自磁盘的数据流。

中间排名值保存在两个表中：curr 表用于保存在当前迭代中需要读取的排名值，next 表保存被写的排名值。控制函数（PageRank）反复启动第 p 个 PRKernel 内核实例，p 代表在 graph 中的表分区编号（与 curr 表和 next 表中的编号相同）。核心实例 i 扫描了 graph 中第 i 分区的所有页面。对每一个 page→target 指向链接，核心实例都读取了 curr 表中的排名值，并且为下一次迭代不断更新 next 表中增加的 target 排名值。

因为这个程序不断地并行更新 next 表中的相同键，它将 sum 累加器与 next 表联系起来，他可以通过 PageRank 算法正确的而结合请求的更新。每一轮全部的计算进程都是用全

局障碍在 PRKernel 中调取。

为实现本地优化，程序将 graph，curr，next 表放在一起为组，并通过 PRKernel 在 curr 表中的执行来阐述重新定位偏好。作为结果，没有内核实例需要执行任何远程调用。另外，程序需要分区函数 SitePartitioner，在相同域的相同分区分配 URL。由于相同域的页面需要不时链接其他页面，这样的分区显著减少了远程更新的数量。

检查点/恢复是直接实现的：控制线程使用同步检查点保存 next 表中的每五次迭代，并且下载了最新的检查点表来从故障中恢复。

3、系统设计

这一部分描述了在使用高速以太网连接的大机器集群上执行 Piccolo 程序的运行设计。

3.1 总体概述

Piccolo 的执行环境由一个 Master 处理和许多 Worker 处理组成，每个执行都尽可能在不同机器上。图 3 描绘了在执行 Piccolo 程序时，Master 处理和 Worker 处理们的全部交互。如图 3 所示，Master 自身执行用户控制线程，并安排核心实例在 worker 处理上执行。另外，master 处理决定怎样将表分区分配给 worker 处理。每一个 worker 处理负责在它本身的内存中存储所分配的表分区，并处理与这个表分区相联系的表操作。仅使用一个 master 处理并未带来表达瓶颈：master 将当前的分区分配情况通知所有的 worker，这样 worker 就不需要查询 master 从而避免带来表达冲突表操作。

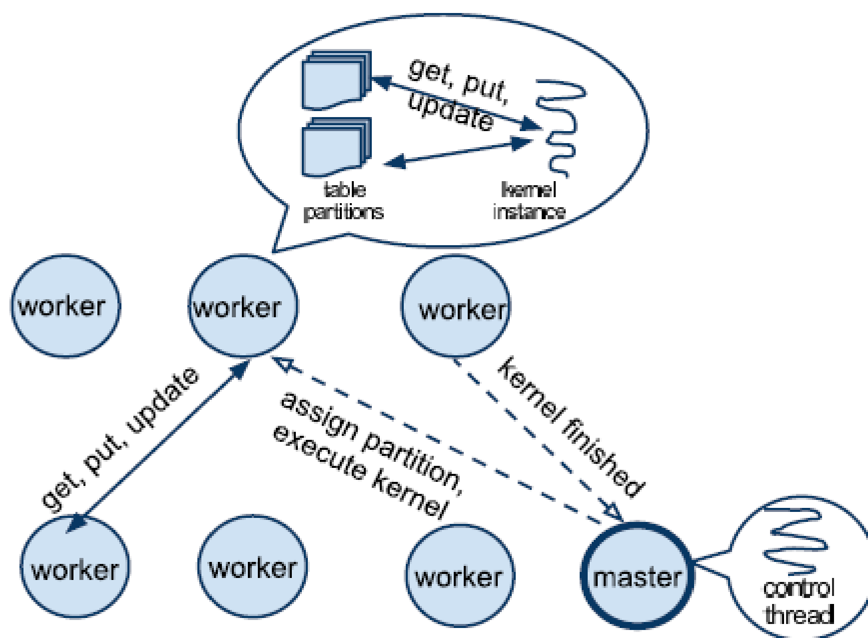

Figure 3: The interactions between master and workers in executing a Piccolo program.

图表 3 执行 Piccolo 程序时 master 和 worker 的相互影响

Master 处理通过调用控制线程中的输入函数，来开始 Piccolo 程序的执行。在每个表创

建的 API 请求上，master 决定分区分配。Master 将分区分配告知所有 worker，每一个 worker 初始化它分配到的一组分区，使其在开始时为空。每一个 run API 请求上执行 m 个内核实例，master 将为每一个内核实例准备一个任务。Master 基于本地偏好安排这些任务在 worker 上执行。每个 worker 一次运行一个内核实例并在任务完成时通知 master。Master 指示每一个任务完成的 worker 在可用情况下继续执行另一个任务。当遭遇全局障碍时，master 将阻挡控制线程直到所有活动的任务完成。

在核心执行时，一个 worker 为远程 worker 定位缓冲 update 操作，使用用户定义累加器将它们结合起来并在一个短的休息之后将它们传给远程 worker。为控制一份 get 或 put 操作，worker 在将更新操作发往远程 worker 前清楚相同键上的累加更新。每一个用户在他们的接收端请求操作（包括累加更新）。Piccolo 并未表现缓冲，但是对预取支持限制格式：在每一个 get_iterator API 请求之后，worker 在当前的迭代值之上预取一部分表输入。

基于以上基本设计产生的两个主要难题是：第一，怎样在一个负载均衡改革中分配任务以全局障碍的总体等待时间？这对于在每一个迭代计算中引起全局障碍的迭代应用尤其重要。第二，如何呈现出有效的检查点和存储表状态。在这一部分剩下的内容中，将详细描述 piccolo 怎样解决这两个难题。

3.2 负载均衡任务调度

没有负载均衡任务的基本调度运行如下。在创建表时，master 为空的存储表使用一个简单的循环分配，分配表分区到所有的 worker。对于从分布式文件下载到的表，master 选择一个分配，使得在大略保证 worker 间的分区数量平衡时尽量降低内部追踪转移。Master 通过特定的本地偏好调度 m 个任务，也就是，它将任务 i 分配给存储 i 分区的 worker 执行。

最初的调度可能并不理想。因为各种各样的硬件配置或者不同变量大小的计算输入，worker 会使用不同数量的时间来完成任务分配，导致负载不均衡和不理想的计算机利用。因此在运行时，最初的调度需要执行负载均衡内核。

Piccolo 的调度权限有两个约束限制：第一，没有运行的任务应该被终止。一个运行的内核实例会修改共享表状态，重新执行终端内核实例需要从保存的检查点执行昂贵的存储操作。因此，一旦一个内核实例启动，让任务完全执行完毕好过半路终止它进行重新调度。相比之下，mapreduce 系统并没有这个约束，它的 reducer 在所有的 mapper 结束之后才执行集合。第二个约束来自于信誉用户本地偏好的需要。特别的，如果一个内核实例从一个 worker 移动到另一个，与它同一位置的表分区也必须转移到那个 worker 上。

使用任务截取实现负载均衡：piccolo 使用了一个简单的模式来呈现负载均衡：master 观察不同 worker 的进程，并指示一个已完成所分配全部任务的 worker(w_{idle})从剩余任务最多的 worker(w_{busy})截取还未执行的任务 i。我们将这种启发式调度先应用于大型任务。执行启发式调度时，master 通过在相应表分区中的键的数量，估算每一个任务的输入大小。Master 从所有的 worker 在每一个全局障碍和表下载时间处收集分区大小的信息。Master 指导每一个 worker 按照现有任务大小的递减顺序执行所分配的任务。另外，空闲的

worker(w_{idle})总是从忙碌的 worker(w_{busy})剩余的任务中截取最大的任务执行。

表分区迁移：因为用户的本地偏好，在执行所截取的任务 i 之前，空闲的 worker(w_{idle})需要从忙碌的 worker(w_{busy})处转移相应的表分区 i。因为表迁移发生在其他活动的任务发送操作到分区 i 时，piccolo 必须注意不能丢失，重排或复制任何 worker 上给定键的操作，以保护表语义。Piccolo 使用多阶段迁移流程从而不需要暂停任何活动的任务。

Master 协调分区 i 从 worker w_a 迁移到 worker w_b 的过程，它分两个步骤进行。在第一步，master 将 M_1 的消息发送给所有 worker，以标明 i 的新的所在地。收到 M_1 之后，所有的 worker 清除它们关于任务 i 在 w_a 的缓冲操作，并开始关于任务 i 在 w_b 发送后续请求。在收到 M_1 之后，w_a 暂停更新任务 i，并开始向前接收来自于其他 worker 关于任务 i 在 $w_a w_b$ 的请求，接着将 i 的暂停状态转移给 w_b。在这一阶段中，worker w_b 缓冲所有从 w_a 或其他 worker 处来的关于 i 的请求，但并不处理它们。

在 master 接受所有 worker 第一阶段完成的回复之后，它将 M_2 发给 $w_a w_b$ 完成迁移。在接收到 M_2 之后，w_a 清除所有关于任务 i 在 w_b 的挂起操作并抛弃分区 i 的暂停状态。w_b 首先按顺序掌控从 w_a 收到的缓冲操作，接着重新获得分区 i 上的正常操作。

可以看到的是，迁移过程并没有阻塞更新操作并且为许多内核实例减少了延迟开销。正常的检查点/修复机制被用于克服在迁移过程中可能导致的错误。

3.3 容错

Piccolo 使用用户辅助检查点和存储来克服在程序执行过程中 master 和 worker 的错误。Piccolo 在运行时保存了一个程序状态的检查点（包含表和其他用户状态）在分布式文件系统上，在从错误中恢复时，使用最新完成的检查点。

检查点：Piccolo 需要较小的开销来存储一致的全局检查点。为保证一致性，Piccolo 必须为程序状态决定全局快照。为减少开销，运行时必须面向活动的内核实例或控制线程来执行检查点。

Piccolo 使用 Chandy-Lamport(CL)分布式快照算法来呈现检查点。为保存一个 CL 快照，每一个过程记录自身的状态，两个过程在一个交流通道汇合合作保存通道状态。在 piccolo 中，通道状态可以仅用内核通过表相互单独交流时的表修改信息来高效的捕捉。

为启动一个检查点，master 选择一个新的检查点记录数字（E）并且发送启动检查点信息 $Start_E$ 到所有的 worker。在接收到启动消息之后，worker w 立刻取得所负责表分区的当前状态快照，并缓冲未来表操作（除去正在使用的）。一旦快照内的表分区被写入稳定存储，w 发送标记消息 $M_{E,W}$ 到所有其他的 worker。Worker w 接着进入记录状态，它将所有的缓冲操作记录到一个重放文件。w 如果收到所有其他 worker 发来的标记 $(M_{E,w'}, \forall w' \neq w)$，它将重放记录写入稳定内存并发送 $Fin_{E,w}$ 到 master，master 在接收到 $Fin_{E,w}$ 之后，从全部 worker 处考察检查点的完成情况。

对于异步检查点，master 使用一个计时器来周期性的启动检查点。为使记录的用户数据与记录的表状态一致，每一个 worker 自动获取表状态的快照，并调用检查点反馈函数来

保存当前运行的内核实例的任意附加用户状态。同步检查点提供语义使检查点状态与全局障碍之后的状态立刻等同。因此，对于同步检查点，每一个 worker 在将检查点标记 $M_{E,W}$ 发送给所有其他 worker 前等待它完成所分配的全部任务。另外，master 尽在其从所有 worker 收到 $Fin_{E,w}$ 之后在控制线程内保存用户数据。有一个协议来决定何时启动同步检查点。如果 master 过早启动检查点，例如：在 worker 仍然有剩余任务时重放文件变得有冗余。另一方面，如果 master 延迟了检查点直到所有 worker 完成，它错过了使用检查点覆盖内核计算的机会。Piccolo 用一个启发式算法来平衡这个协定：只要有一个 worker 完成了其分配的任务，master 就启动同步检查点。

为简化这个设计，master 只有在有活动的表迁移时才启动检查点，反之亦然。

恢复：检测到任意 worker 的错误之后，master 重置所有的 worker 状态并从最后完成的全局检查点中恢复计算。Piccolo 并不检查 master 的内部状态——如果 master 重新启动，恢复照常进行，然而，替换的 master 可以在恢复期间，任意选择新的分区分配和进行任务调度。

4、更多应用

除 PageRank 之外，Piccolo 实现了四种应用：分布式网页爬虫，K-均值计算，n-体模型，矩阵乘法。这一部分简要说明了 Piccolo 编程模型如何高效实施这些应用。

4.1 分布式网页爬虫

除 PageRank 之类的迭代算法外，Piccolo 可用于在多台计算机之间实现分布式和协同细粒度任务的应用。为证明这个应用，我们实施了一个分布式网页爬虫。基本的爬虫操作比较简单：从几个最初的网址开始，爬虫重复下载一个页面并将它们分块以发现新的可抓取网址。一个实践性爬虫必须同时满足其他重要的约束：（1）尊重每个网站的 robots.txt 文件。（2）不要铺天盖地的涌向某网站设定上限以固定利率抓取。（3）避免重复抓取同样的网址。

实施时用到了三个相关的表：

- url_table 表为每一个网址存储了爬虫状态 To Fetch，Fetching，Blacklisted，Done。对每一个网址 p 在 To Fetch 状态，爬虫抓取相关的网页并设定 p 的状态为 Fetching。当爬虫将 p 完成分块并提取出衍生链接后，设定 p 的状态为 Done。
- Politeness 表跟踪每个网站最后下载这个页面的时间。
- Robots 表为每个网站存储处理的 robots 文件。

```
#local variables kept by each kernel instance
fetch_pool = Queue()
crawl_output = OutputLog('./crawl.data')

def FetcherThread():
  while 1:
    url = fetch_pool.get()
    txt = download_url(url)
    crawl_output.add(url, txt)

    for l in get_links(txt):
      url_table.update(l, ShouldFetch)
    url_table.update(url, Done)


def CrawlKernel(Table(URL,CrawlState) url_table):
  for i in range(20)
    t = FetcherThread()
    t.start()

  while 1:
    for url, status in url_table.my_partition :
      if status == ShouldFetch
        #omit checking domain in robots table
        #omit checking domain in politeness table
        url_table.update(url, Fetching)
        fetch_pool.add(url)
```

Figure 4: Snippet of the crawler implementation.

图表 4 爬虫程序执行片段

爬虫为 m 台计算机产生 m 个内核实例，实施时使用 Python 完成，以利用 Python 的网络关联库。图 4 显示了简化的爬虫内核（忽略处理 robots.txt 和设定每个网站下载率上限等细节）。每个内核扫描本地 url_table 分区以发现 ToFetch 网站并用一个帮助线程池处理它们。在三个表都由 SitePartitioner 函数分区并与相联系之后，内核实例可以在下载网址之前高效的检查 politeness 表的信息和 robots 表的输入。实施时使用最大的累加器按照 Done > Blacklisted >Fetching > ToFetch 顺序，来解决在 url_table 表内相同网址上的写-写冲突。这使这个简单并优雅的操作得以展示在表 4 中，内核重新发现一个已抓取的网址 p，能够请求更新 p 的状态到 To Fetch，并持续进行直到 P 的正确状态。

一致的全局检查点对于爬虫的修复非常重要。如果没有全局检查点，修复的爬虫会找到一个页面 p 处于 Done 状态但不会在 url_table 表中看到任何 p 的衍生链接，这可能导致那些链接永远不会被爬虫抓取到。实施中异步检查点每 10 分钟执行一次，这样爬虫在处理一个节点错误时不会花费超过 10 分钟。从最新的检查点修复可能导致一些页面被爬虫多次抓取（一些在最近的检查点中丢失），但是检查点机制保证了没有页面会遗漏。

4.2 并行计算

k-均值算法，k-均值算法是一个迭代算法，通过在一个多维空间内将数据分成 n 组指向 k 个集群实现。Piccolo 在实施中将数据指向分配中心和中心在共享表中的位置存储起来。

每一个内核实例为下一次迭代运算使用总和累加器，执行一个数据指向的子集，来为数据指向和更新中心位置计算新中心的分配。

n-体算法，这个应用程序模拟在许多离散的时间步长中一系列微粒的动力性。实施 n-体模拟时，倾向使用短距离，这个距离大于所假设的两个微粒不相互影响的阈值范围（r）。在每个时间步长中，内核实例执行粒子们的子集，它更新粒子的速度和基于当前速度的位置以及在阈值范围以外的其他粒子的位置。实施中使用分区函数将空间分成立方体，使一个内核实例最大程度表现本地读取从而检索到阈值范围以外的粒子。

矩阵乘法。计算 C=AB，A 和 B 为两个大矩阵，是一个在数值线性代数中常见的基本运算。输入和输出矩阵可以被分割成 m*m 块存储在三个表中。运算时将 A，B，C 三个表联系起来。每一个内核实例通过计算 $C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$ 执行一个表分区。

5、实施

Piccolo 是用 C++实现的，它提供了 C++和 Python 应用程序接口，允许用户使用这两种语言编写内核和控制函数。Piccolo 使用 SWIG 工具构建 Python 接口。实施中利用了一些现有的库，如：用于通信的 OpenMPI，google 用于对象序列化的协议缓冲以及用于压缩磁盘上表的 LZO。

所有的并行运算（PageRank，k-均值算法，n-体模型和矩阵乘法）都使用 piccolo 的 C++应用接口实现。分布式爬虫使用 Python 应用接口实现。

6、评价

测试第四部分描述的应用程序在 piccolo 上的表现时。一些应用程序，如 PageRank 和 k-均值算法，同样可以用现有的一些数据流模型实现，我们比较了这些应用程序在 piccolo 和 hadoop 上的运行表现。

运行结果中 piccolo 的亮点如下：

- Piccolo 更快。PageRank 和 k-均值算法分别比在 Hadoop 上执行快 11 倍和 4 倍。已发布的 DryadLinq 中提到，对 900M 页面图片进行 PageRank 迭代算法用时 69 秒，而 Piccolo 对 1B 的页面图片进行迭代时，在 EC2 上用时 70 秒，而只占用了 1/5 的 CPU 核心。
- Piccolo 规模大。在所有测试的应用程序中，随着 worker 数量的增长，计算时间几乎呈线性减少。PageRank 在 EC2 上执行的 100 个实例同样证明了大规模。
- Piccolo 能够帮助如爬虫这样的非常规应用实现好的并行表现。实验中的爬虫，尽管使用 Python 实现，却几乎覆盖了所在集群的全部网络带宽。

6.1 测试安装

大多数实验是由本地集群的 12 台计算机完成的：6 台机器分别有 1 个四核英特尔至强 X3360(2.83GHz)处理器，4GB 内存，另外 6 台分别有 2 个四核英特尔至强 E5520(2.27GHz）处理器，8GB 内存。所有计算机通过商品千兆以太网转换连接。EC2 实验中包含 100 个大型实例，每个占用 7.5G 内存，和两个虚拟核心，每个虚拟核心相当于 2007 年代单独核心

的 2.5G 英特尔至强处理器。在全部的实验中，为每个核心创建了一个 worker 处理器，并强制它使用那个核心。

Application	Default input size	Maximum input size
PageRank	100M pages	1B pages
k-means	25M points, 100 clusters	1B points, 100 clusters
n-body	100K points	10M points
Matrix Multiply	edge size = 2500	edge size = 6000

Figure 5: Application input sizes

图表 5 应用程序输入大小

在进行规模实验时，改变了不同应用程序的输入大小，表 5 显示了默认和最大输入大小。基于在英国 100M 页面的网络图片统计为 PageRank 产生了网络链接图。特别的，我们抽取了每个站点中页面数量的分布和站点内外链接的比例。通过抽取网站大小分布知道达到所需的页面数量，生成一个任意大小的网络图；网站中每个页面的输出链接基于站点内外链接的比例产生。对于其他的应用，我们使用随机生成的输入。

6.2 规模表现

表 6 显示了应用程序在默认输入大小下，随着 worker 数量（N）从 8 增加到 64 时的加速情况。所有的应用程序为计算密集型，并随着 N 的增加，表现出良好的加速情况。理想情况下，所有的应用程序（除去 PageRank）已经完美的平衡了表分区并应该实现了线性加速。然而，在 N=8 时为获得合理的运行时间，我们选择了一个相对小的默认输入大小。因此，当 N 增加到 64 时，Piccolo 的开销相对于应用计算本身不再是微不足道（如：k-均值在 N=64 时，完成每次迭代需要 1.4 秒），比理想的加速状态下降了 20%。PageRank 的表分区不再平衡并且任务截取在此规模下变得更加重要（见 6.5）。

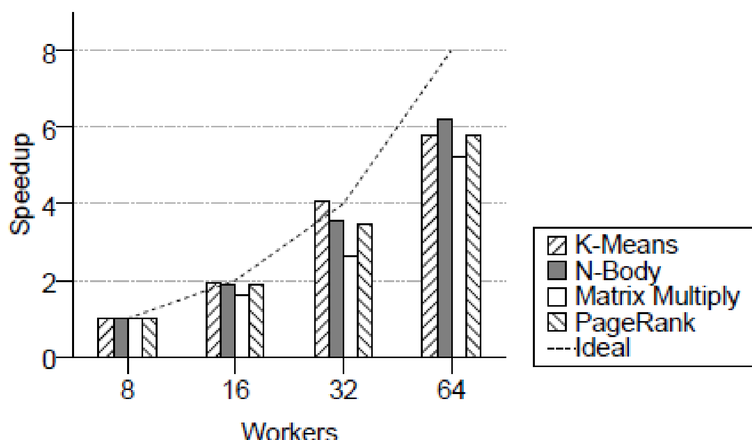

Figure 6: Scaling performance (fixed default input size)

图表 6 规模表现（设定默认输入大小）

我们同样评估随着输入大小的增长的程序规模，通过适应输入大小随着 N 的增加来保持每个 worker 固定计算量。在 PageRank 和 k-均值算法中随着 N 的增加我们线性增加输入规模。在矩阵乘法中，边际大小以 $O(N^{1/3})$增长。我们并没有显示在 n-体算法中的结果，因为很难随着输入规模增加保证每个 worker 的计算量固定。在这些实验中，理想规模的常规运行时间作为输入量随着 N 的增长而增长。如表 7 所示，全部应用程序获得的规模在理想数量的 20% 以内。

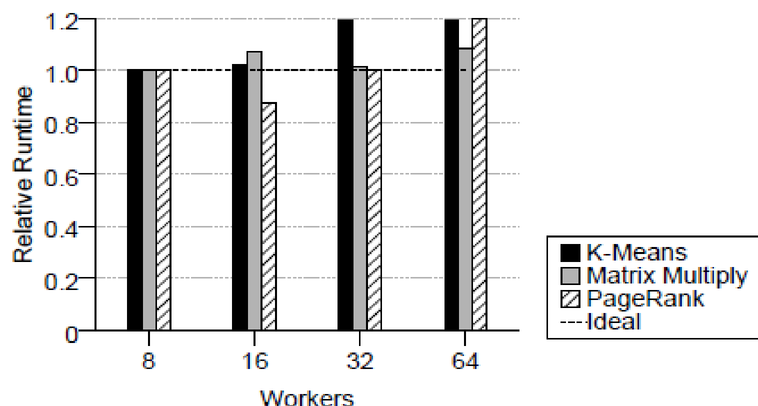

Figure 7: Scaling input size.

图表 7 规模输入大小

6.3 EC2

研究 Piccolo 怎样将大批计算机使用 100 个 EC2 实例规模化。表 8 显示了 PageRank 和 k-均值计算在 EC2 上随着输入量随 N 值增加的规模增长。比较令人惊奇的是，在 EC2 上的规模化比我们在本地实验平台上进行小实验时获得了更好的结果。本地实验平台的 CPU 表现出相当的变异性，影响了规模化。在更多的研究之后，我们相信这种变异性的来源很可能是动态 CPU 的频率调整。

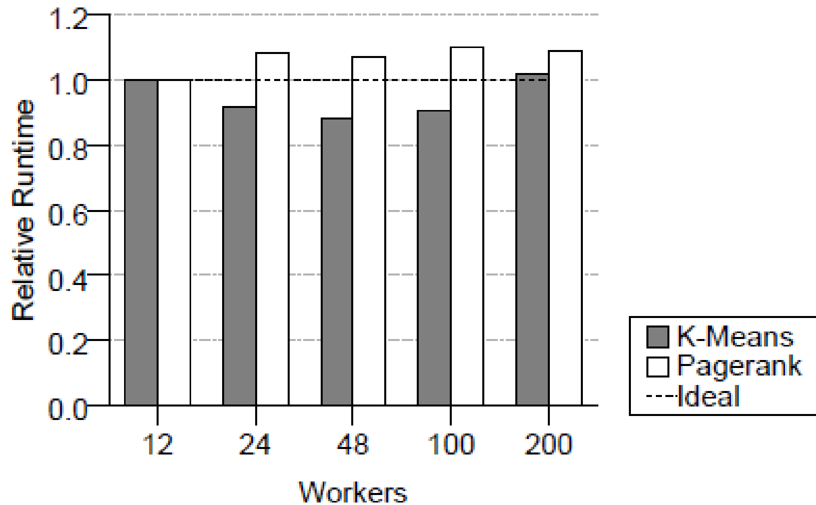

Figure 8: Scaling input size on EC2.

图表 8 EC2 上的规模输入大小

在 N=200 时，PageRank 在 70 秒内完成了 1B 页面的链接表。相似数量的图（900M 页面）在本地的实验台上，使用更少的 worker（N=64）获得了可以相媲美的表现（80 秒），这要归功于本地实验台的高性能核心。

6.4 与其他框架的比较

与 hadoop 比较：将在 Hadoop 上实施 PageRank 和 k-均值算法的表现与两者在 piccolo 上实施的表现做对比。剩下的应用，包括分布式网络爬虫，n-体模型和矩阵乘法，在 Hadoop 数据流模型中并没有直接的实施情况。

在 Hadoop 上实施 PageRank，如同 Piccolo 一样，我们按照网站将输入链接表分区。在运行过程中，每一个 map 任务将正在操作的分区表本地化。Mapper 将图和 PageRank 分数输入合成，使用一个合成器将分块结果汇总。在 hadoop 上执行的 k-均值算法是高度优化的。每一个 mapper 通过 hadoop 的文件系统（HDFS）从当前的迭代中抓取全部的 100 个质心，在输入流中计算集群分配的每一点。作为结果，reducer 只需汇总每个 mapper 的一个更新来生成新的质心。

我们做了大量的努力来优化 PageRank 和 k-均值算法在 Hadoop 上的表现，包括改变 hadoop 本身。优化包括使用原始存储比较，使用初始类型来避免 java 的装箱和拆箱开销，禁止日志校验，提高 Hadoop 的整合实施等。表 9 显示了运行状态下，Piccolo 和 Hadoop 使用默认的输入量时的状态。Piccolo 在两个基准上的表现都优于 Hadoop（在 N=64 时，PageRank 是 11 倍，k-均值是 4 倍）。由于我们已经优化了 k-均值的实施，其在 hadoop 和 Piccolo 上的表现差异已经比较小；但 PageRank 的框架并不允许相似的优化。

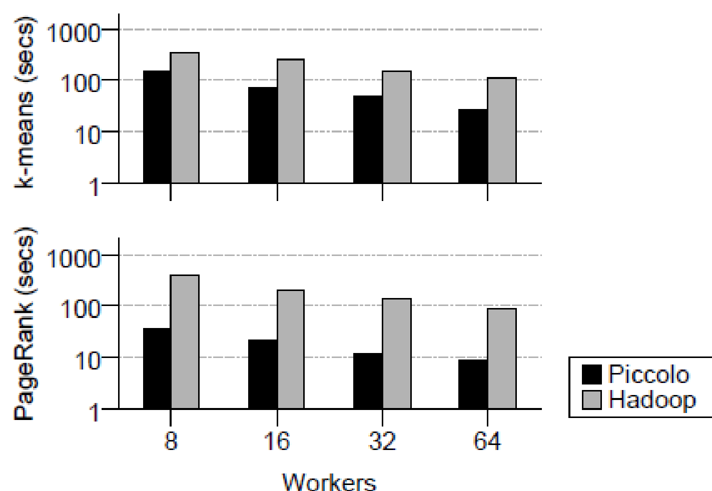

Figure 9: Per-iteration running time of PageRank and k-means in Hadoop and Piccolo (fixed default input size).

图表 9 PageRank 和 k-均值在 Hadoop 和 Piccolo 上每次迭代的运行时间（固定默认输入大小）

由于 Hadoop 使用 Java 实现而 Piccolo 使用 C++实现，我们期望看到两者在表现上的差异，但重要性先后顺序的不同又带来了令人吃惊的结果。简要描述 RageRank 在 Hadoop 上的实施以发现有用的特征。降低实施速度的主要原因有：（1）分类键在 map 语句中（2）序列化和非序列化数据流（3）读取和写入 HDFS。在 PageRank 的基准中建分类单独就占用了将近 50%的运行时间，序列化占用了另外的 15%。相反，在 Piccolo 中，与（1）相关的需求已经被消除，与（2）（3）相关的开销被大量减少。PageRank 的排名值被存储在内存中，并且可用于迭代而无需在分布式文件系统中序列化。另外，多数输出链接指向同一网站的其他页面，核心实例直接停止表现许多本地存储的表数据的更新，从而避免了序列化的完全更新。

与 MPI 比较：将矩阵乘法在 Piccolo 上实施的表现与在基于 MPI 的第三方软件上的表现相比较。MPI 版本使用加农算法处理块状的矩阵乘法，并采用特定的 MPI 通信原语来处理数据广播，同时发送和接收的数据。在 piccolo 上，实施原始的块状相乘算法，使用分布表来控制矩阵状态的通信。由于 Piccolo 依赖 MPI 原语进行通信，我们并不期望在表现上有优势，而对产生花销的大小有更多兴趣。

表 10 显示了 Piccolo 在运行时低于 MPI 10%。我们吃惊的看到在有更多 worker 的试验中，Piccolo 表现出了优于 MPI 版本的实施情况。经检查，我们发现这是由于集群中计算机的轻微性能差异造成的；由于在实施中 MPI 比 Piccolo 多了许多同步点，它被强制等待速度慢的节点赶上。

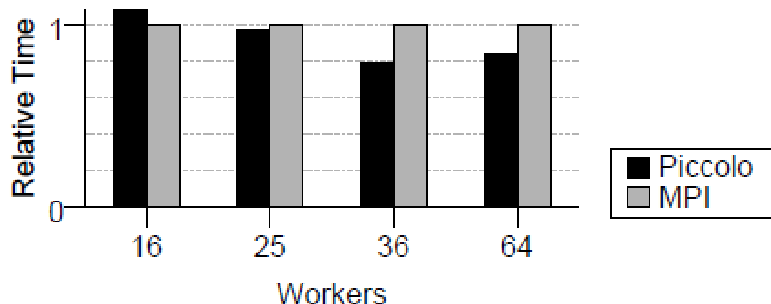

Figure 10: Runtime of matrix multiply, scaled relative to MPI.

图表 10 运行矩阵相乘，相对 MPI 的规模增长

6.5 任务截取和慢速计算机

PageRank 的基准为测试任务截取的影响提供了一个好的基础，因为网络图片分区大小变化较大：900M 页面图片的最大分区是最小分区的 5 倍。使用相同的基准，我们测试当一个 worker 比其他操作慢时，会产生怎样的性能改变。在一个核上运行一个密集型运算程序，这使得相较于其他 worker，这个 worker 仅有 50%的 CPU 运行时间。

测试的结果显示在表 11 中，在所有计算机运转正常的情况下，任务截取减少了约 10%的运行时间，这种提高归结于输入分区大小的不平衡——在运行不执行任务截取时，计算会等待更长时间让 worker 处理更多的数据抓取。

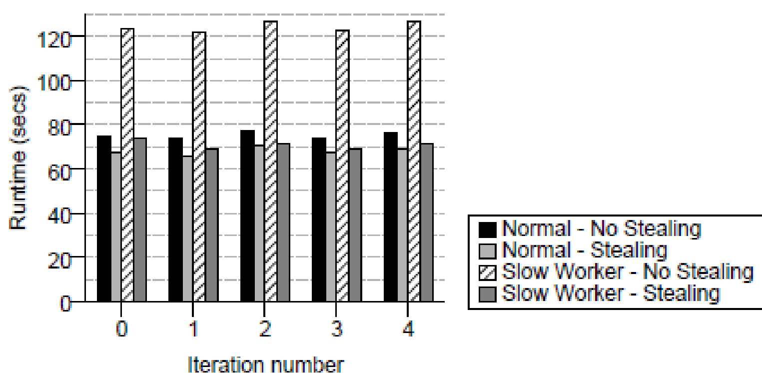

Figure 11: Effect of Work Stealing and Slow Workers

图表 11 任务截取的影响和慢速 worker

慢速 worker 在计算上的影响更富有动态性。当任务截取禁用时，运行时间几乎两倍于正常的运算，因为每次迭代必须等待最慢的 worker 完成所有分配的任务。启用任务截取增加了这种环境的动态性——计算时间减少到只超过无慢速例子的 5%.

6.6 检查点

使用 PageRank，k-均值和 n-体问题来评估检查点的花费。相较于其他问题，PageRank 有更大的表需要被检查，这让检查点/修复的性能测试更具有需求性。在试验中，每个 worker

在本地磁盘写入它的检查点表分区。表 12 显示了在运行时，当没有检查点的时候，检查点会自动关联。在本地同步检查点战略中，Master 仅在所有 worker 完成检查点之后启动检查点。在优化战略中，只要有一个 worker 结束 Master 就会开始检查点。如图所示，优化后的检查点策略的花费几乎可以忽略（2%左右），同时，优化的检查点较早启动，会使 PageRank 的大型检查点减少花费。

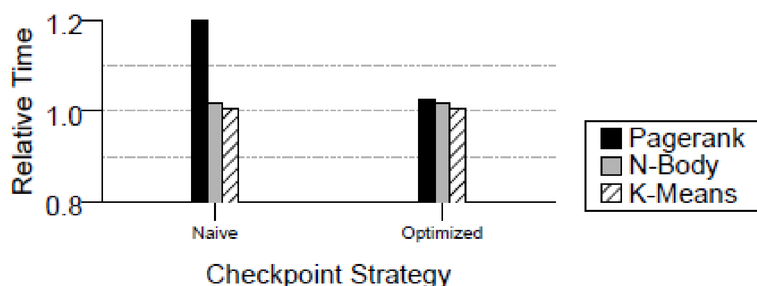

Figure 12: Checkpoint overhead. Per-iteration runtime is scaled relative to without checkpointing.

图表 12 检查点消耗。每次迭代运行时间在没有检查点的情况下的相对规模

全局检查点的限制和恢复：Piccolo 的原始全局错误恢复机制会带来扩展性的问题。随着集群的扩大，错误出现的频率更高；这导致了更加频繁的检查和恢复，从而更多占用全局运算时间。当没有更多的计算机资源去直接测试 Piccolo 在上千台计算机上的性能时，就根据预期的计算机运行时间估算 Piccolo 检查点机制的规模限制。

假设一个有 16GB 内存和 4 个磁盘驱动器的计算机集群。检测检查点和恢复机制在这样一台机器上运行的“最坏情况”所花费的时间——对于占用所有可用内存的表状态进行计算。我们估测 Piccolo 的计算时间能够根据计算机数量和故障率灵活高效利用（而不是花费在检查点或恢复状态上），在我们的模型中，假设计算机故障将根据故障率和集群中的计算机数量定义一个固定的时间间隔到达。虽然这是对真实错误情况的简化，但是最坏的情况下的恢复机制并提供了一个有用的下限。根据我们的模型预期的效率，如表 13 所示。为保证数据中心按我们熟悉的方式良好运转，平均的计算机运行时间为一年左右。这些数据中心，全球的检查点机制可以有效地扩展到几千台计算机。

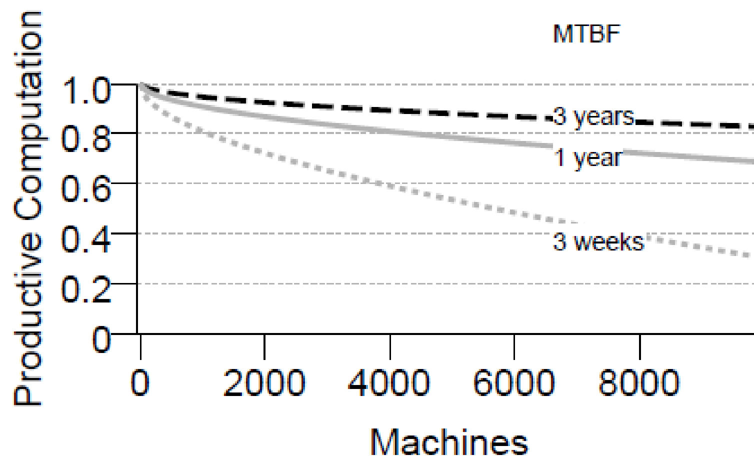

Figure 13: Expected scaling for large clusters.

图表 13 大型集群上所期望的规模

6.7 分布式爬虫

我们使用不同数量的 worker 来评估分布式爬虫的实施情况。初始化种子集有 1000 个网址在网址表中。在实验运行 30 分钟结束后，测量抓取的网页量和下载的字节数。图 14 显示了爬虫的网页随着 N 的增加，从 1 到 64 兆字节/秒的下载吞吐量。爬虫大部分的 CPU 时间花费在解析 HTML 和网址的 Python 代码上。因此，其吞吐量的规模随着 N 线性变化，在 N = 32 时，爬虫下载的吞吐量峰值在 10MB / s，这是由我们的实验运行的 100-Mbps 上行传输互联网所限制的。高度优化的爬虫在单服务器上实现时，可以维持下载速度超过 100Mbps。然而，尽管使用 Python 搭建，Piccolo 基于爬虫仍可能扩展到更高的下载速度。

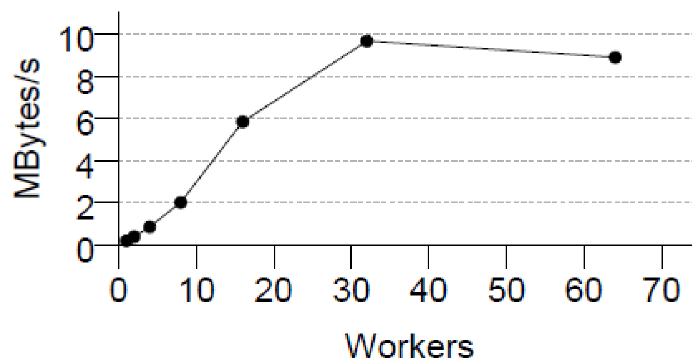

Figure 14: Crawler throughput

图表 14 爬虫程序的吞吐量

7、相关研究

面向通信的模型：如 MPI 和并行虚拟机（PVM）等通信原语，被用于构建分布式程序已流行多年。MPI 和 PVM 提供广泛的通讯机制，包括单播和广播，以及支持在分布式环境

中创建和管理远程进程。一直有研究开发 MPI 的实验功能，如优化集体操作，通过虚拟机的容错，和利用混合检查点及日志进行修复。MPI 已被用于建立非常高性能的应用——它对明确沟通的支持允许很大的灵活性，可在高速计算环境中编写应用程序时利用各种各样的网络拓扑结构。这种灵活性有一个复杂形式的成本——用户必须明确管理 worker 之间的沟通和同步状态，这在试图保持高效和正确执行时会变得困难。

大容量同步并行（BSP）是一个高层的面向沟通的模型。在这种模型下，线程在本地内存运行不同的处理器，使用消息互相沟通，并执行全局障碍同步。BSP 的实施通常使用 MPI。最近，BSP 模型已适应在 Pregel 框架下并行处理大图。

分布式共享内存：面向通信模型的编程复杂性在分布式共享内存（DSM）系统领域掀起了一阵研究热潮。大多数 DSM 系统，旨在提供透明内存访问，这将导致使用 DSM 编写的程序引发许多细粒度同步事件和远程内存读取。虽然最初很有希望，但 DSM 的研究已经脱离了网络延迟比例转而到本地 CPU 性能的大小，这使得基本的远程访问和同步花费过高。

并行全局地址空间（PGAS）是一个实现分布式共享地址空间的语言扩展。这些扩展尝试改善 DSM 的延迟问题，使用户表达通过一个特定的线程的一部分共享内存的密切关系，从而减少了远程内存引用频率。他们保持面向 DSM 的通用低水平（平面内存）接口。因此，为 PGAS 系统编写的应用程序在操作非原语数据类型时，仍然需要细粒度同步，或者将几个值加总（例如，计算多个写入的内存位置总和）。

元组空间，正如 Linda 和最近的 JaveSpaces 等协调语言所暴露给用户的一样，一个全局元组空间可以从所有分区线程访问。虽然元组空间为读取和编写元组提供了原子原语，但并不倾向于接受高频率的访问。正因为如此，既不支持本地优化，也没有写-写冲突的解决方案。

MapReduce 和数据流模型：近年来，MapReduce 的已成为并行数据处理的流行编程模型。有很多研究受 MapReduce 启发产生，从推广 MapReduce 以支持合并操作，改进的 MapReduce 的流水线性能，到在 MapReduce 的顶层建设高层次的语言（如 DryadLINQ，Hive，pig 和 Sawzall）。FlumeJava 提供了一个抽象集合和并行处理原语可优化和编译 MapReduce 操作顺序。

MapReduce 和 Dryad 的编程模型都是流处理实例或数据流模型。由于 MapReduce 的的普及，程序员开始使用它建立如 PageRank 的内存迭代应用程序，尽管数据流模型不是天然适合这些应用程序。Spark 建议增加分布只读内存中的缓存以提高基于 MapReduce 的迭代计算性能。

单机共享存储模型：许多编程模型可在单机上并行执行。在此设置中，在计算核心之间有共享物理内存支持的低延迟内存访问和计算线程之间快速同步，这在分布式环境中是不能实现的。虽然也有普及的流/数据流模型，但许多单机并行模型是基于共享内存的。GPU 平台，有 CUDA 技术和 OpenCL。对于多核 CPU，Cilk 和最近的英特尔线程构建模块，提供了对低开销线程的创建和优秀的任务调度支持。OpenMP 是一种流行在科学计算社区

的共享存储模型：它允许用户针对并行执行的代码段，并提供同步和原语缩减。最近，一直在努力支持跨机器群集 OpenMP 程序。然而，基于分布式共享内存的软件，在实施过程中与 DSM 和 PGAS 系统一样遭受相同的限制。

分布式数据结构：分布式数据结构目标是提供一个灵活和可扩展的数据存储或缓存接口。这样的例子有：DDS，Memcached，和最近提出的 Ram-云，以及许多基于分布式哈希表的键值存储。这些系统不追求提供计算模型，而注重松耦合分布式应用程序，如 Web 服务。

8、结论

并行内存应用程序需要将访问与共享的中间状态驻留在不同的计算机上。Piccolo 提供了一个编程模型，支持可变共享，通过一个键/值表接口分布式内存状态。Piccolo 有助于应用程序实现高性能，可通过优化本地访问共享状态和运行时使用应用程序指定的积累函数自动解决写-写冲突实现。