DMR1.0 设计与实现

俞玉芬[1]　张昱[2]

中国科学技术大学 计算机科学与技术学院

December 25, 2016

[1]yufeny@mail.ustc.edu.cn
[2]clarazhang@gmail.com

目录

1 Scalable of multicores

现有的对多核 scalability 的研究工作总结

《The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors》[?] 接口的设计限制了 scalability，例如 Posix 中 open 系统调用：open 一个 file，并返回一个 descriptor，但标准的 posix 要求 descriptor 的值是当前进程可用的最小值。"This forces the kernel to coordinate file descriptor allocation ,even when many threads are opening files in parallel" 因为多个线程共享进程的 descriptor，当多个线程同时需要分配的时候，就需要对最小值进行竞争 (内核代码？)，从而影响 scalability。但如果随机找一个可用的值，会让任务变得简单。

2 dmr-1.0 设计概述

dmr-1.0 是对原来的最初版本的 DMR 进行改写，论文的关键点：DMR 较好的 scalability。整体的思路：

- 观察和分析 Phoenix scalability 不好的原因
- 基于新的 Producer-Consumer 模型搭建 DMR.
- 优化性能：buffer 的重新设计与实现

3 研究动机

随着多核机器的广泛普及, 如何简单有效地利用多核资源已成为一个十分重要的课题。现有的多核上的并行编程模式, 包括共享内存多线程 pthread 和消息传递 MPI, 它们需要程序员自己管理线程之间的通信、任务分派和调度，这无形中给程序员增加了很多负担，也让并行编程变的复杂。

2004 年 Google 提出了 MapReduce 编程模型，极大的简化了并行编程。受到 Google 的 MapReduce 编程思想的启发，耶鲁大学的 Range 等人将 MapReduce 编程模型移植到多核环境，他们编写了一套针对多核的 MapReduce 库，并通过实验证明，应用程序使用 MapReduce 编程模型在性能上可以与 phtread 编写的程序相媲美，而且它延续了 MapReduce 的优点，就是它的编程接口简单。用户可以不用熟悉多核环境，也不需要了解任务的分割和并行执行等问题，只需要提供简单的 map 函数和 reduce 函数，就能够最大化的利用多核资源。

多核环境下的 MapReduce 库 phoenix 提出之后，学术界对 phoenix 提出了很多的改进和优化，例如：Phoenix2, Metis，phoenix++, MALK。我们通过实验发现，phoenix 的可扩展性比较差，具体表现为：应用程序使用 phoenix 库时，在 1-8 核环境下，运行时间随着核数的增多而下降；但是 8 核以上，运行时间不降低，反而上升，特别是在 32 核环境下，应用程序的运行时间甚至比 1 核情况的时间都长。这种不稳定性意味着，8 核以上的系统，很难通过使用 phoenix 达到理想的性能效果。为了改进这个问题，我们重新编写了一套 MapReduce 库，它给用户提供的接口与 phoenix 一致，因此 phoenix 的应用程序几乎不需要修改就可以使用我们的 MapReduce 库。

4 研究背景

4.1 相关研究

(简单介绍 MapReduce 模型，以及相关研究)

4.2 Phoenix 局限性分析

虽然 Phoenix 为程序员提供了简单编程的手段，它却存在一定的局限性。 如图1的数据结果

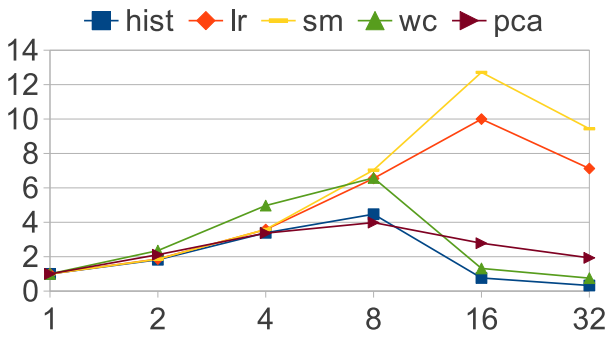

图 1: Phoenix 不开启 Combiner 情况下的 Speedup

(speedup 计算方法为: 高核下的运行时间/单核下的运行时间)，我们可以得出这样的结论：随着核数的增多，4 核以下，Phoenix 的性能越来越好；超过 4 核，Phoenix 的性能却越来越差，特别是 hist, wc, pca。

Phoenix 的上述特性意味着，针对低核的机器，它能够充分利用其资源，对于高核处理器，Phoenix 并不能充分利用资源。Phoenix 不适合在 8 核以上的机器上运行。然而，多核机器是一个趋势，一个 CPU 甚至达到上百的 core [?], 随着多核机器上核数的不断增多，Phoenix 便不具有实际的可用性。

制约 Phoenix 性能的主要因素有以下几点：

1. Phoenix 基于 Posix 线程库实现，开启的线程越多，需要共享的内核资源越多，会因为竞争这些资源以及共享的数据结构导致过高的 spinlock. 在 32 核情况下，hist 的 spinlock 占用 71.25%。后续章节会详细分析 Phoenix 的 spinlock 问题。
2. 由于中间结构的局限，Phoenix 中 Map 阶段与 Reduce 阶段中间存在一个严格的 barrier，降低了 map 和 reduce 线程并发的程度。此外，通常情况下，map 是 computation-intensivex 的，reduce 是 memory-intensive 的 [?]，barrier 会导致资源的利用率比较低。

DMR 将针对上述的局限性进行改进和优化，即避免 Spinlock 以获得较好的 scalability，同时打破原来的 barrier，增加 Map 和 Reduce 阶段并发的程度，从而获得更好的性能。

5 总体设计

这一部分，我们首先详细分析 Phoenix 较差 scalability 的根本原因，然后我们提出可行的解决方案，即构造一种具有较好 scalability 的 thread model，并详细解释这种新的线程模型的详细设计方案，最后通过实验证明它的 scalability.

5.1 新的线程模型

5.1.1 Phoenix 较差 scalability 的分析

Phoenix 中启动的 map 或 reduce 线程数与系统的核数相同，从 Phoenix 的 speedup 图 (??) 中，我们可以看出，当超过一定的限度，增加核数，并不能为 Phoenix 的应用程序带来 scalability 的

提升，即随着核数的增多，处理应用程序所需要的时间就越长，speedup 的曲线证明了 Phoenix 的 scalability 非常差。

为了深入探究较差 scalability 的根本原因，我们利用 Linux perf 工具来收集程序的热点函数信息（主要看热点函数占用总运行时间的百分比），通过热点函数的分析，查看高核情况下，占用时间最多的运行函数，如表??显示，在 16 和 32 核情况下，各个应用程序中占用时间最多的函数。从表中可以看出，对一 hist, lr, wc, sw，在 32 核情况下 ticket_spin_lock 的开销非常大，针对 ticket_spin_lock，我们测试各应用程序的占用情况，如图??所示，从图中可以看出，随着核数的增多，各应用程序中的 ticket_spin_lock 的占用的开销越来越大，特别地，在 16 核和 32 核情况下，histgram 中 ticket_spin_lock 占用的开销最大，分别为 71.25% 和 40.15%. 这表明，16 核以上，hist, lr, wc, sm 的运行时间主要用于锁的竞争和等待，而没有做实际的计算。并且随着核数的增多，这种竞争越激烈，导致 Phoenix 的 scalability 较差。事实上，从前面对 Phoenix 的分析我们知道，Phoenix 中采取了划分和 barrier 两种策略，以避免多个 map 和 reduce 对同一个 matrix 竞争，然而实验的结果却显示在 8 核以上，依然有很激烈的竞争。函数的调用图显示，ticket_spin_lock 几乎全部是由 pagefault 引起的。

为了保证进程能够在 O(lgn) 的时间读写查找 mmap 的内存区域，linux 采用一个红黑树来组织多个 mmap 区域，当一个新的区域插入，或者删除一个不用的区域时，需要重现调整红黑树的结构，以保证红黑树的特性，为了保证一致性和正确性，整个红黑树结构通过一个 lock 来保护的。进程地址管理结构 mm_struct 中，有一个 mmap_sem 信号量，任意需要访问和修改 mmap 区域的线程或进程，首先需要获得 mmap_sem 信号量，才能能够读写虚拟地址空间。由于线程没有自己的地址空间，需要共享进程地址空间，因此当多个线程都需要读写内存区域时，便会竞争 mmap_sem.

Phoenix 的应用程序通过 mmap 系统调用来读取输入数据，当 map worker 调用 map 函数处理输入函数时，会产生大量的 pagefault. pagefault 内核路径中，首先需要获取 mm_sem 信号量，然后才能修改 mmap 区域。因此当多个 map worker 同时产生 pagefault 时，多个编程便会竞争 mmap_sem，这个信号量是一个睡眠锁，如果信号量不能满足，它便进入对应的等待队列中睡眠等待，竞争的线程越多，等待的时间便越长，从而浪费多核资源，造成整个多核系统的性能下降。

5.1.2 可伸缩的线程模型

Phoenix 中采取一定的策略避免多线程对共享数据的竞争，然而在高核情况下，由于内核中的 pagefault 的可扩展性较差，导致激烈的锁的竞争。为了让 pagefault 具有更好的可伸缩性，可以采用一些方式如 []...。但是，我们不想改变操作系统的内部实现，我们仅仅希望设计一个简单的可伸缩的线程模型，以支持 mapreduce 处理。事实上，如果使用进程来实现 map worker 或 reduce worker，便可以可以避免多个线程对共享地址空间的读写信号量的竞争。但进程之间数据的共享却是一个比较复杂的问题。

为此，我们设计了一个新的 scalable 线程模型 Sthread, 它基于 C 库实现。与传统的 Pthread 线程不同，Sthread 线程都用有自己独立的地址空间（即在内核中拥有一个私有的 mm_struct 结构），而不需要与其他的线程共享地址空间，从而可以有效避免 mmap_sem 读写信号量的等待。另一方面，我们知道基于共享地址空间的多个线程间，可以直接与其他的线程通信，一旦线程地址空间隔离，通信将变的不那么容易。为此，Sthread 为线程提供了一个 channel，即线程之间可以共享 channel，用于数据的共享，从而避免地址空间隔离导致的数据共享不便。

表??展示了 Sthread 用于管理线程和通道的主要接口。初始化情况下，主线程调用`thread_alloc`分配 map 和 reduce 线程，并且调用`chan_alloc`为每对 map worker 和 reduce worker 创建 channel，

用于 map 和 reduce 之间的通信。之后，主线程便可以调用thread_start启动线程。SMR 中，所有的 channel 都有一个 producer 和 consumer，构成了一个 producer-consumer 模型。master 调用chan_setprod和chan_setcons 分别将 map worker 设置成 producer 和 reduce worker 设置成 consumer(将在下一章详细解释).

相比传统的 Pthread 模型，Sthread 可以减少多线程对内核数据结构的竞争，但 Sthread 线程的创建，channel 的分配和设置，都为系统带来了额外的开销，我们将在第五章详细测试并分析这些额外的开销。

5.1.3 无边界的 channel

一旦设置好 channel 的生产者和消费者关系后，map workers(producer) 便可以调用chan_send将消息发送到 channel, reduce workers(consumer) 通过调用chan_recv从 channel 中接受数据。通常情况下，当 channel 的通信 buffer 时，producer 需要等待，直到 channel 中的数据被 consumer 取走，但这种停止等待限制了系统的性能。为此，我们设计了一个无边界的 channel，即当 channel 的 buffer 满时，producer 无需等待，可以继续向 channel 中发送数据。

无边界的 channel 的实现，依赖于底层的 extend 机制，它允许将 channel buffer 区域重新映射到一块新的物理地址，并且不影响 consumer 对旧的物理的读取。channel buffer 对应 producer 和 consumer 地址空间中一块区域，我们称之为CHAN区域，有一个 pagetable（ptab）用于管理该区域的物理地址映射。

初始情况下，系统不会为CHAN区域 (默认情况下为 512K) 分配实际的物理页，而是将其ptab中的pte指向一个 codetAnchor1 对应的物理地址，Anchor1 是一个物理页，被 producer 和 consumer 共享。当 producer 需要发送一页的数据时（即向 CHAN 区域中写数据时），会产生一个 pagefault，系统分配一个实际的物理页page，并让 anchor 中对应的 pte 记录该 page 的地址。Producer 可以通过 ptab 中 pte 找到该物理地址，并将需要发送的数据 copy 到该物理页，完成数据的发送。consumer 接受数据时，更具读取的位置 address，得到 ptab 中对应的 pte，得到 anchor 中对应的 anch_pte，anch_pte 中记录了对应的物理 page 对应的位置，从中读取数据，读取结束后，会释放对应的物理页 page。

与传统的 channel 不同，当 CHAN 区域被写满时，producer 不需要等待 consumer 将数据取走，而是通过 extend 机制，将 CHAN 区域映射到一个新的 anchor2 物理页，之后将重复上述过程，为 producer 分配物理页并发送数据。anchor1 中有一个特殊的 pte 将两个 anchor 链接起来。此时 Consumer 可以继续读取旧的物理页 page1，当 page1 读取完毕后，通过 anchor 中的一个特殊的 pte，找到 anchor2 以及其指向的数据，继续读取 page2 的数据。

接下来的章节中，我们将介绍基于 Sthread 实现的 SMR，并详细解释 Sthread 的 producer-consumer 模型，以及无边界的 channel buffer 是如何被使用。

5.2 SMR 的 dataflow

5.3 流水线并行

如上问所述，影响 Phoenix 性能的关键因素是 barrier 的存在，以及 Posix 线程库较差的 scalability。DMR 基于一种新的 Producer-Consumer 模型，打破 barrier，且不再使用线程库，从而提高处理的效率和 scalability。本节阐述新的 Producer-Consumer 的设计原理，DMR 的流水并行，以及地址空间的隔离。

多核下的 MapReduce 模型中，Map 阶段产生的 key-value，都存放于一个共享的中间结构，之前的很多研究都显示，影响多核 MapReduce 系统性能的关键因素是对该中间结构的操作效率 [?]。

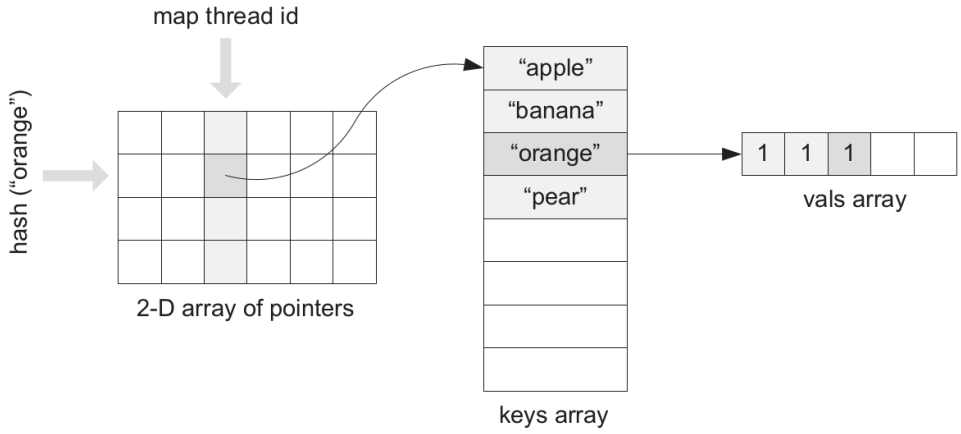

图 2: phoenix intermediate is a gobal array

Phoenix 中间结构是一个全局的二维数组（如图2），为了避免多个 map worker 和 reduce worker 对共享的中间结构竞争，Phoenix 采用两种策略：

- 对全局的二维数组进行划分：每个 map 操作每一行，每个 reduce 操作每一列。即 map 和 reduce 都拥有自己独立的读写区域。可有效避免多个 map 或多个 reduce 之间读写同一区域的竞争。
- 为了避免 map 和 reduce 的交织，Phoenix 在 map 和 reduce 之间加入 barrier，即 Map 阶段结束后，才能开始 Reduce 阶段。可以避免 map 和 reduce 对同一区域的竞争。

Map 阶段通常会进行大量的计算，Reduce 阶段则是需要大量的内存访问。如果像 Phoenix 一样，在 Map 和 Reduce 之间加入 barrier，即等到所有的 Map 阶段结束，才开始 reduce 计算，就会存在两个问题：

- 不利于 CPU 的利用率，Map 阶段会集中使用 CPU，而 Reduce 阶段需要大量的内存访问，CPU 资源被浪费。
- reduce 阶段开始时间，由最慢的 map worker 决定；当某个 map worker 非常慢，便会影响整体的性能

DMR 设计实现中，首先打破 Phoenix 的 barrier, 不再使用共享的中间结构，DMR 中的每个 map worker 都拥有属于自己的私有 buffer(buffer 的设计细节见下一节)，一旦 buffer 中的 key-value 达到一定的阈值便发送给相应的 reduce worker, reduce 收到 key-value 后，不等 map worker 结束便开始 reduce 工作，即 Map 和 Reduce 阶段并发的粒度变小，这既能充分利用资源，又能提高计算的速度。

特别地，当我们采用 array buffer，并且在 map 阶段不开启 combiner 时，Map 阶段不需要对 key-value 排序，即无需 key 的查找和插入，以及 memmove 等操作。map worker 产生 key-value 后，只需简单地将其追加到 buffer 中即可，这减轻的 Map 阶段的工作量。key-value 的排序工作由 Reduce 承担，reduce worker 对收到的 key-value 进行有序插入。虽然整个过程的工作量并未减少，但是 Reduce 的排序工作与 Map 阶段是并发执行的，从而整个过程的时间变小，提高运行的效率。

此外，为了防止数据的倾斜，即大量的 key-value 被发送到一个 reduce worker，在 Reduce 阶段，我们添加了局部的 reduce 过程（即 combiner），一旦某个 Reduce 收到某个 key-value 的数量超过预先设定的值，便会触发 combiner，从而避免过多的内存分配带来的开销，防止内存溢出。

DMR 中 map worker 和 reduce worker 之所以能够进行流水并行执行，是因为它们基于一个 Producer-Consumer model，在这个 model 中，每个 map worker 都拥有一个私有的 buffer，map woker 是这个 buffer 的生产者，reduce 是 buffer 的消费者，当 buffer 中的数据达到一个阈值时，reduce 便可以取到 buffer 中的数据开始工作，而不需要等到所有 map 结束。

5.3.1 produce-consume 模型

如上所述，map worker 和 reducer worker 的流水执行，需要底层 Producer-Consumer 模型的支持，本节将详细描述该模型的设计原理。

通常 producer-consumer 模型中，producer 和 consumer 之间有一个 queue，producer 向 queue 中添加任务，consumer 从 queue 中读取任务。MRPhi [?] 为了使 Map 和 Reduce 并行执行，便是采用这种 Producer-Consumer 模型，具体如图3所示。在这个模型中，每个 reduce worker 对应一个 queue，map worker 产生的数据会追加到 queue 中，这是一个多对一的 produce-consume 模型， 虽

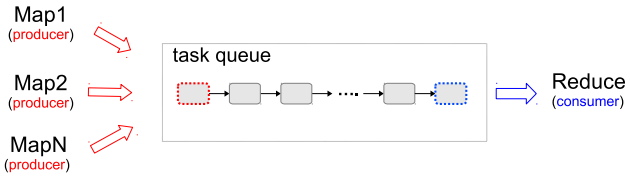

图 3: producer-consumer model in MRPhi

然 MRPhi 利用该 produce-consume 模型实现的 Map 和 Reduce 阶段的并发执行，但是它存在以下两个问题：

- map worker 将 task 插入到 queue 中之前，需要竞争 queue 的 lock。并且线程数越多时，等待 lock 的开销会越大。由于多个 map worker 同时向 task queue 中插入任务，需要一定的同步机制保证插入的正确性，这会造成一定的等待开销。
- queue 的管理问题，虽然 MRPhi [?] 中并没有提到，具体的 queue 是如何管理的，但 queue 的管理不外乎两种：(1) 采用固定分配的方式：预先分配一块固定大小的空间，之后重复利用，但当 queue 满，map woker 需要停止等待，直到 reduce 将 queue 中的数据取走。(2) 采用动态分配：虽然 map worker 不需要等待，但是会存在大量的动态内存分配和回收的开销。

DMR 中使用的 producer-consumer 模型则不同，其中 map worker 和 reduce worker 使用一种一对一的隐式 queue，每个 map worker 只需将 task 插入到专属的 queue 中即可。因此，多个 map worker 不需要竞争 queue，从而可有效减少锁的开销。

如图4所示，综上所述，DMR 中 producer-consumer 模型的关键不同之处有两点：其一，producer 和 consumer 之间是一对一的隐式 queue，因此 producer 之间无须竞争，可以避免锁的竞争带来的开销。其二，producer 和 consumer 之间的隐式 queue，不采用显示的操作，而是采用一种 mapping 的方式，即一旦 buffer 中的数据满，producer 便会触发一个 send 操作，底层的实现中，会将 buffer 对应的物理内存加入到隐式 queue 中，buffer 重新映射到一块新的物理内存，producer 只需将 buffer 设置为空，便可继续对 buffer 进行读写。这种方式的优点在于其快速高效。

DMR 中的具体操作为，map worker 向 buffer 中写入数据，当 buffer 满时触发 send 操作，之后将 buffer 标志为空，便可继续向 buffer 中写数据。reduce worker 以轮循的方式读取各个隐式 queue 中的数据，从而，map worker 和 reduce worker 可以并发的进行工作，且不需要复杂的同步手段。

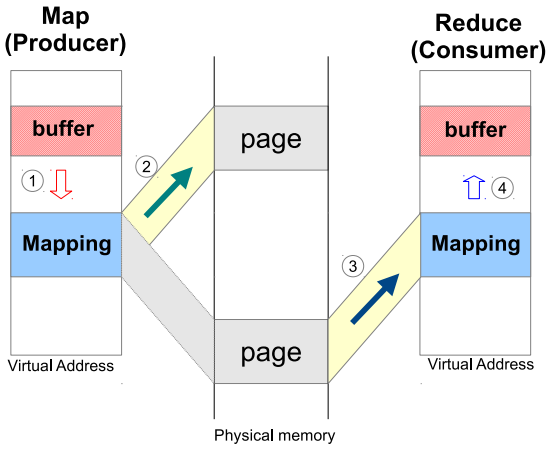

图 4: producer-consumer model in DMR

5.3.2 buffer 的设计与实现

多核下的 MapReduce 模型中，Map 阶段产生的 key-value，都存放于一个共享的中间结构，之前的很多研究都显示，影响多核 MapReduce 系统性能的关键因素是对该中间结构的操作效率 [?]。

Phoenix 中 map 插入 key-value 的过程为：map 根据 hash(key) 确定 key 在二维数组中的位置，具有相同 hash 值的多个 key 存于一个 array 中，通过折半查找，确定 key 在 array 中的位置，最后插入。多个 map 产生的相同 key 会存于同一列。reduce 会对二维数组中的每一列中 key 的多个 value 进行归并，得到最终结果。

从 Phoenix 处理中间数据的流程可以看出，Phoenix 中 key-value 需要保持有序，便于 Reduce 阶段的归并。Map 阶段需要完成 map 计算以及 key 的查找、插入，Reduce 阶段只需要简单的进行归并。

通过分析 Phoenix 中间结构，以及中间结构设计存在的局限性，DMR 试图从两个角度对 Phoenix 的中间结构进行改进：

- 改进 Phoenix 中的公有 buffer，每个 map 线程都拥有自己私有的 buffer，这样可以避免 Map 和 Reduce 之间的 barrier，增加并行的程度。
- 针对 DMR 本身设计的特点，优化 buffer 的内部实现，不再使用 hash 表的方式组织 key，而是采用 array 表。

以下将详细解释私有 buffer 的设计与优化。

为了避免 Map 和 Reduce 之间的 barrier，DMR 将 Phoenix 中共享的中间结构拆分，每个 map worker 拥有一个 local buffer pool, buffer 池中有多个 buffer，分别用于存放特定 reduce 的数据如图5。

DMR 中 map 和 reduce 之间数据流动的过程为：map 产生 key-value，根据 partition(key) 确定 buffers pool 中的某个 buffer，当该 buffer 中的 key-value 达到预先设定的阈值时，map 触发物理内存的重映射，reduce 开始读取该 buffer 中的数据，开始 reduce 的工作。

buffer 的内部实现，采用类似 Phoenix 的方式，即 hash 表的方式进行组织。在 map 阶段开启 combiner，以减少内存和动态内存分配带来的开销。采用 hash 表组织的优势在于：key 的查找长度比较短。

改进后的实验结果如图6显示，左图表明，随着核数的增多，性能较平稳；右图为 DMR 与

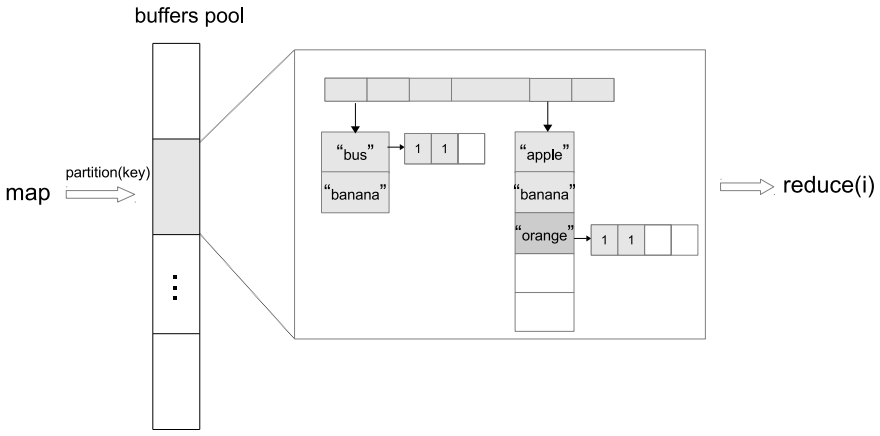

图 5: local buffer pool for each map worker

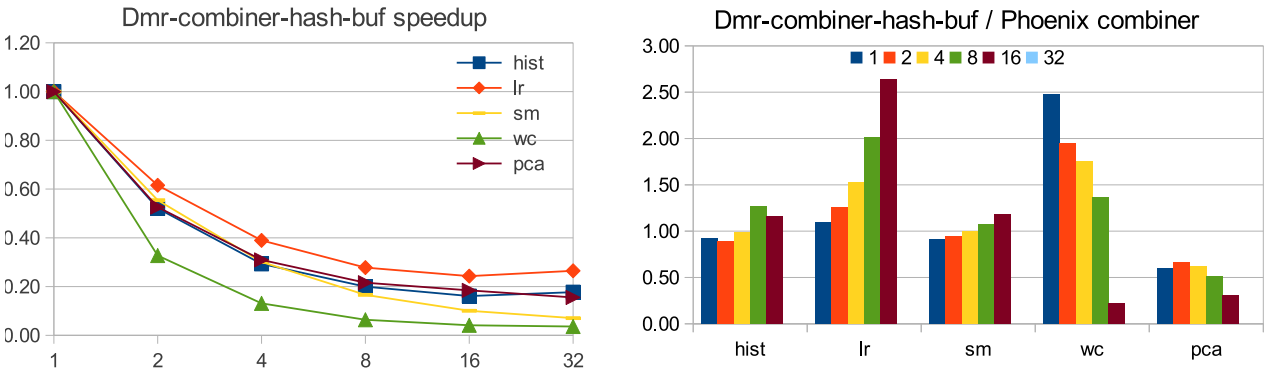

图 6: dmr compare to phoenix

Phoenix 的对比结果，其中 hist, pca, sm，DMR 具有较好的性能，但结果并不理想，为此，我们需要分析 DMR-hash 实现的局限，充分利用 DMR 处理流程的特点，以改进性能。

DMR-hash 的 buffer 中，每个 buffer 是 hash 表的方式实现，其中存放的 key-value 数据的分布是离散的，然而 Producer-Consumer 模型要求 key-value 存放于一块连续的物理空间中。因此，hash buffer 在出发重新映射之前首先需将分散的 key-value 汇集到一块连续的空间中，通常是数组，我们称这个过程为 group 阶段。通过实验的数据发现，group 的开销相当大。此外，hash buffer 的方式，需要大量的动态内存分配。

针对 DMR 中 Map 和 Reduce 阶段并行这一特点，我们试图改进 buffer 的 hash 实现。它不再采用原来 Phoenix 中的 hash 表的组织方式，而是采用更简单的 array，map worker 产生的 key-value 只需追加到 array 中即可，无需排序。同时 reduce 阶段也可以开启 combiner. 通过 array 实现会带来以下优势：

- 初始化 array buffer 时，运行时为其分配一块固定大小的空间，该预分配的空间可重复利用，可减少动态内存分配带来的开销。
- 相较于 hash buffer，array buffer 是一块连续的空间，因此可以避免 group 带来的开销。
- 由于 L1 cache 的大小有限，group 阶段在遍历时，会更大的可能性访问不在 L1 cache 中的 key-value; 而 array buffer 不需要 group 阶段，因此会更少的 cache miss. 如图7左图所示。从而 array buffer 可有效提高性能

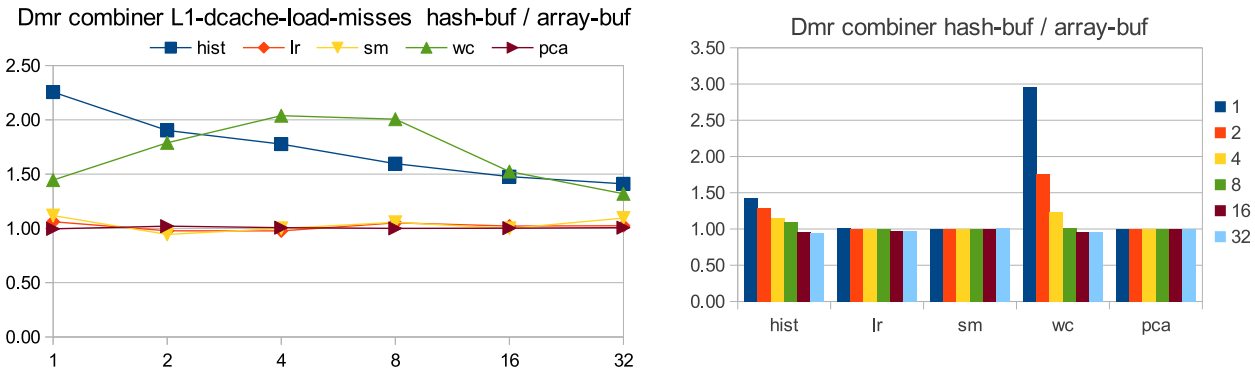

图 7: hash compare to array

优化后 DMR 与 Phoenix 对比的整体性能如图8

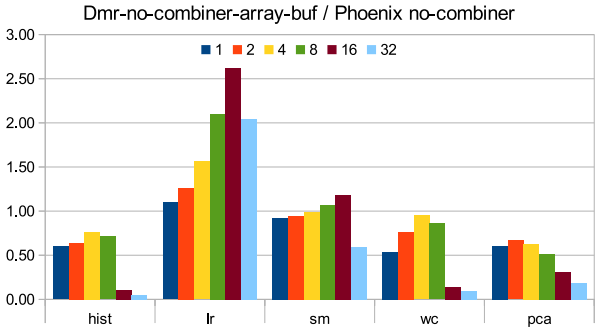

图 8: dmr compare to phoenix

6 实验结果与分析

6.1 Phoenix 较差 scalability 的实验结果

实验结果显示1，8 核以上，随着核数的增多，Phoenix 的性能越来越差，其中 hist 表现的最为明显，通过详细分析发现，hist 中有几个被频繁访问的全局变量分别为：red_keys[256],blue_keys[256],green_keys[256] 多个 map 线程会并发地访问这些全局变量，（这会导致多个线程竞争 Linux 中保护进程描述副的读写信号量，这个读写信号量是用来保证同一个进程创建的多个线程串行地访问同一个进程的描述符。）(需要继续补充和验证的) 通过收集 perf record 的数据??，结果显示，随着核数的增多，ticket_spin_lock 的开销越来越大, 并且在 16 和 32 核下, 它们是最耗时的函数, 分别占整个程序开销的 40.50% 和 71.25%。虽然 lr, sm, wc, pca 中没有频繁访问的全局变量，但是 Phoenix 是基于 Pthread 多线程编程的，随着核数的增多，多个线程需要竞争内核态锁和信号量，导致 ticket_spin_lock 过高。

此外，我们发现使用不同的内存分配器也会影响应用程序的性能. 通过对比 Phoenix-jemalloc 和 Phoenix-glibc 的时间，可以发现，hist, wc, pca 在高核下 Phoenix-jemalloc 性能较好。因此，在高核数环境下,Phoenix 使用相对稳定的 jemalloc，具有更好的性能9。

然而，即使 Phoenix-jemalloc 具有较好的性能，其 scalability 也是较差的，如图10， 我们可以

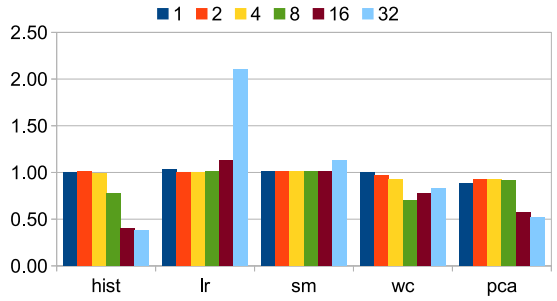

图 9: Phoenix-jemalloc / Phoenix-glibc no combiner

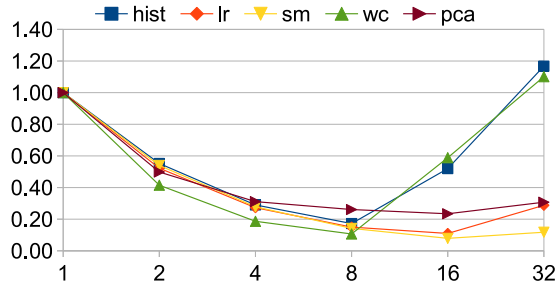

图 10: Phoenix jemalloc no combine speedup

看出，wc,hist 在 8 核以上，性能越来越差；通过 perf record 的详细分析发现，相比 Phoenix-glibc，Phoenix-jemalloc 的 ticket_spin_lock 明显降低，但随着核数的增多，该部分的开销也是越来越大，特别地，在 16 核与 32 核情况下，wc 的 ticket_spin_lock 的开销分别为 24.04% 和 39.91%。

将具有较好性能的 Phoenix-jemalloc 与 DMR 进行对比，实验结果如图11所示， 可以看出，除

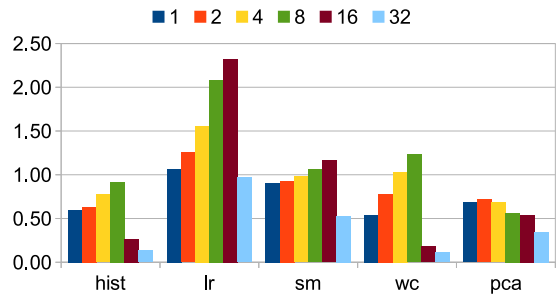

图 11: DMR-array / Phoenix-jemalloc no combine

了 lr，DMR 表现的性能都较好，特别地，在 16 和 32 核情况下，DMR 的性能远优于 Phoenix-jemalloc。结合 DMR 和 Phoenix-jemalloc 的 scalability，我们可以推测，在高核情况下，核数越多，DMR 的优势会更加明显。

6.2 piepeline 带来的性能优势

开启单线程情况下，可以看出 pipeline 带来的性能优势，

6.3 DMR 环境初始化的开销分析

DMR 采用新的 Producer-Consumer 模型，以及地址空间隔离的进程，它为 DMR 带来好的性能的同时，也存在一部分额外的开销，主要表现在环境初始化部分。环境初始化的时间主要是指：从应用程序调用 mapreduce 库开始，到 Map 阶段开始前的时间。

Phoenix 环境初始化主要包括：全局变量的构建和初始化，map 任务队列的初始化，thread pool 的创建和初始化（包括构造 thread pool 需要的控制结构、线程的创建、启动线程）；DMR 环境初始化中全局变量和 map 任务队列的初始化与 Phoenix 相当，主要差别在于 producer-consumer 模型的搭建，它包括进程的创建、producer 和 consumer 角色的设置、map worker 和 reduce worker 之间一对一的隐式 queue 的构建，实验结果显示该部分的开销远大于 Phoenix 中 thread pool 的创建。

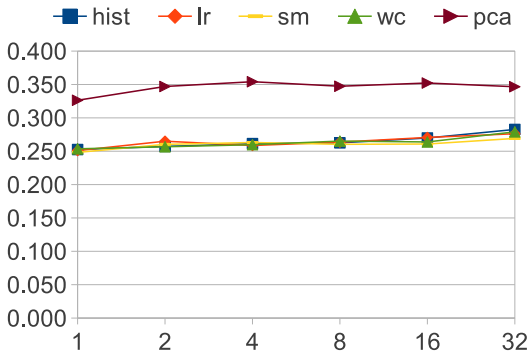

图 12: DMR environment initialize time(seconds)

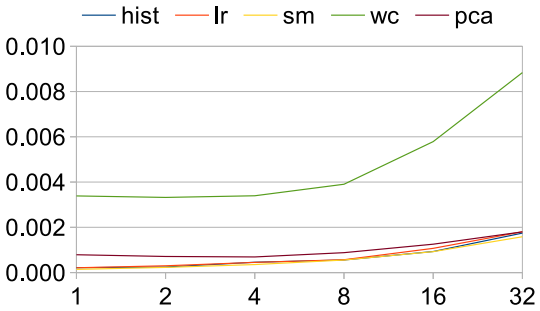

图 13: Phoenix environment initialize time(seconds)

如图12所示，DMR 初始化时间为 0.25s ～ 0.35s，Phoenix 初始化的时间为 0.0002s ～ 0.0020s(如图13)，相比之下，DMR 初始化花费时间远大于 Phoenix。此外，从数据中我们可以看出，对不同的应用，不同的核数，DMR 的初始化时间基本稳定，由此我们可以得出：针对数据集较大，运行时间较长的应用程序，初始化时间所占的比例就越小，DMR 的性能就会越好。

图8给出了 DMR 的性能，其中 hist, wc, pca 具有较好的性能，sm 持平，lr 的性能较差，图14 给出各应用程序初始化时间的占总运行时间的百分比，可以看出 lr 的初始化时间所占的百分比相当大，且高于其他应用程序, 这使得 lr 在 DMR 上执行的效率比 Phoenix 差。

6.4 benchmarks

应用程序的特点如图1

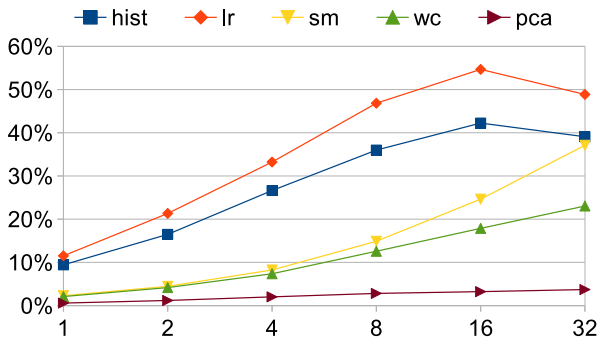

图 14: 环境初始化时间所占时间的百分比

- AppName: 应用程序的名称
- Dataset: 输入数据集
- Total key/value Pair Num: 给定的数据集总共会产生的键值对的数目
- Total Key Num: 给定的数据集中总共会产生 key 不同的键值对的数目

表 1: 应用程序信息

AppName	Dataset	Total key/value Pair Num	Total Key Num	
word_count(WC)	100MB	17941890	123748	
histogram(Hist)	1.4GB	761903	692	
linear regression(LR)	500MB	2590	5	
pca(PCA)	4000*4000	8002000	8002000	
matrix multiply(MM)	2000*2000	0	0	
kmeans(KM)	-d 3 -c 1000 -p 100000 -s 1000	340300000	1000	
string match(SM)	500MB	0	0	

6.5 环境参数

6.5.1 编译选项

1. 定义一个编译参数 xxx，编译时开启的方式如下：

```
make xxx=1
```

多个参数可组合

```
make xxx=1 yyy=1
```

2.DMR 中的编译参数如下：

- combiner: 定义该参数，则开启 combiner；默认情况下关闭
- array: 定义该参数，则采用数组实现的 buffer；默认情况下使用 hash 表实现的 buffer
- nosort: 定义该参数，在 hash 表实现的 buffer 中 key 数组无需排序，并采用普通查找法；默认情况下排序，采用折半查找
- ncores: 可以定义不同核数的版本；默认为 32 核

3. 不同应用程序的编译选项

针对五个应用程序，DMR 相比 phoenix 具有较好性能的策略如表2所示

表 2: 不同应用程序最佳策略选择

application	dmr strategy
histogram	no combiner array buffer
linear_regression	combiner hash buffer
string_match	no combiner hash buffer
word_count	no combiner array buffer
pca	combiner hash buffer

6.5.2 DMR 需要的 dlinux 运行参数

当应用程序的数据集较大时，需要改变 dlinux 中的参数，才能正确运行

- 调整改变 EMEM 的大小:smpc.h

```
#define EMEM_NPAGE  (1<<22) // 8G->16G
```

- 调整 channel 和元数据的大小：/libspmc/inc/spmc_chan.h

```
#define CHAN_MAXSHIFT   17  // channel size: 512KB ->128K
#define SPMC_METASIZE   (1<<23) // channel metadata size: 4MB->8MB
```

- 调整 heap 的小小：spmc_malloc.c

```
#define SPMC_HEAP_SIZE      (((uint64_t)PAGE_SIZE << 22)) // 4MB->16MB
```

7 附加

7.1 twin-and-diff

DMR 中有两个应用程序需要使用到 Twin-and-diff 机制

1.matrix_multiply: 该应用程序的 map() 函数，处理输入数据，并将结果直接存放于最终结果 data.output 中，data.output 指向主线程 master 的 heap 区域中一块空间；如果 master 与 slaver 采用共享堆，多个线程会共享该区域，mapreduce 计算结束之后，data.output 便会收集各个线程的局部结果。

```
    data->output[x_loc*data->matrix_len + i] = value;//data

```

但 DMR 中，由于采用进程模拟线程，各线程间是地址隔离的，heap 是私有的，因此主线程 master 的 data.output 无法收集到各个 slaver 的处理结果

Twin-diff

7.1.1 详细设计方案

1.scope 的范围保证 page 对齐，未填满的部分，0 填充；未保护的区域从下一个页开始分配，这样做的目的是：避免多余的保护；如果不需要保护的对象落入保护区，可能造成不必要的开销，甚至影响结果的正确性

2. 多次迭代，scope 是全局变量，master 在进行下次的 mapreduce 时，并不会清空 scope 的内容，保护的范围依然是第一次调用之前保护的范围如果重新设置，错误，而且 heap allocator 可能会使用之前释放的 heap 空间

3. 对于 application 中没有 heap 共享对象时，无需进行 protect，无需进行 twin-and-diff，application 不会触发 malloc,spmc_s.heap=NULL;

因此在 twin_dif.c 中，需要判断 scope==NULL 的情况，即无需进行 twin-and-diff

7.1.2 TODO

父进程中给出保护范围 ¡start, end¿，通常是整个 master 的 heap 子进程在保护区域的时候，不仅仅需要保护父进程的保护区域，同时还需要保护其祖先传递给它的地址范围这两个范围线性地址空间，可能不连续，那么孙子进程就需要保护多块地址空间

7.1.3 bug 记录

snapshot_init() 备份的应该是产生 pagefault 的页，而不是从 fva 开始的页，否则比对的时候两者不一致，这将导致结果错误

7.2 插桩收集 DMR 时间信息说明

为了详细分析 DMR 执行过程中各阶段的时间开销，我们对 DMR 进行插桩

代码位置: git 库 dmr-1.0branch

commit: 26b7c9a7193d9e4de8e0679ef291b37238dbf788

源文件: src/dmr_stat.c, src/inc/dmr_stat.h, inc/dmr_wrapfuns.h(配置文件)

7.2.1 重要的全局变量和数据结构

1. 结构体

```
typedef struct dmr_stat {
#ifdef DMR_TIMING
    int tid;
    double runtime;
    double tstart[CALLLAST];
    // time and counts of syscall/lib function calls
    double calltime[CALLLAST];
    uint64_t ncalls[CALLLAST];
#endif
} dmr_stat_t;
```

dmr_stat_t 中 tstart[id] 记录每次运行 id 函数时的开始时间，calltime[id] 记录 id 的总运行时间，ncalls[id] 记录 id 被调用的次数

2. 全局变量

```
dmr_stat_t *timeinfo = NULL;
dmr_stat_t *dmr_s_tstat = NULL;
```

timeinfo 用于存放所有线程的时间信息，在 map,reduce 中，默认情况下 map 线程数等于 reduce 线程数，另加 master 线程，因此初始化时，设置 (MAX_THREADS * 2 + 1) 个，并使用 MAP_SHARE 的 mmap，保证多个线程都可以都可以修改该变量。

dmr_s_tstat：每个线程都拥有一个该全局变量，用于指向 timeinfo[TID] 的位置，每个线程在运行前，调用 init_

```
  void init_timeinfo()
{
#ifdef DMR_TIMING
    int threads = MAX_THREADS * 2 + 1;
    timeinfo = (dmr_stat_t *)mmap(NULL,
                    sizeof(dmr_stat_t) * threads,
                    PROT_READ|PROT_WRITE,
                    MAP_SHARED|MAP_ANONYMOUS,
                    -1, 0);
    /* for master */
    init_dmr_tstat(0);
    //printf("dmr_s_tstat=%p\n", dmr_s_tstat);
#endif
}
```

注意以下几点：

1. spmc_set_copyargs() 是在 map_reduce() 之前被调用，为保证其中的 DMR_MALLOC() 收集时间信息时，不出现指针为空的 segment fault，需要提前调用 init_timeinfo 设置 timeinfo
2. 为了收集 master 的完整时间信息，需要在 init_timeinfo() 中调用 init_dmr_tstat(0)，因为在 spmcenv() 调用之前，无法通过 init_dmr_tstat(TID) 来设置
3. 最后在结束 env_fini() 中 timeinfo=NULL，因为有些 pca 和 Kmeans 是多次 map_reduce() 的计算

7.2.2 内存时间的收集

1.DMR 中涉及动态内存的操作:malloc, calloc, realloc, free，使用宏定义来替换，方便收集不用类型的操作使用的 malloc 的时间开销

```
#define DMR_MALLOC(a, b) mem_malloc(a, b)
#define DMR_FREE(a, b) mem_free(a, b)
#define DMR_CALLOC(a, b, c) mem_calloc(a, (size_t)b, (size_t)c)
#define DMR_REALLOC(a, b, c) mem_realloc(a, (void *)b, (size_t)c)
```

2. 配置文件中

```
  mmxx(TIME, KV, mem_kv, 9) //define MEM_KV
```

mmxx() 用于定义一个 MEM_KV 标识符 (枚举类型)，在应用程序中 DMR_MALLOC(MEM_KV, sizeof(int) * len) 来统计

3. 插桩

```
void *mem_malloc (ID_t type, size_t size)
{
    MEM_BEGIN(type);//memory time
    void *temp = malloc (size);
```

```
    assert(temp);
    MEM_END(type);//memory time

    return temp;
}
```

使用 MEM_BEGIN()，MEM_END() 进行插桩收集数据

7.2.3 时间的收集

收集时间的信息，最终会存入到 dmr_tstat.csv 文件中 1. 配置文件中

```
fnxx(TIME, map, map, 1)
```

2. 插桩

```
        BEGIN_TIMING(map);
        /* do map()*/
        map_worker_do_task(env, tid);
        END_TIMING(map);
```