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

February 14, 2017

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

目录

1 研究背景和动机

这一节主要介绍研究的背景和研究的动机

1.1 多核的普及与挑战

随着多核机器的广泛普及，在可预见的未来，一个芯片上的 CPU 核数可以达到上百个，甚至千个 [1]。然而，如何简单并高效地利用多核资源依然是一个挑战，因为并行编程依然是很困难的问题。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手动的管理线程间的同步，负载的均衡，任务分派和调度等问题，这要求程序员充分理解底层的硬件特性，这无形中给程序员增加了很多负担，也让并行编程变得容易出错，且复杂。为了让程序员避开这些设计的细节问题，一种可选的方案是依赖于运行时系统，让它管理所有这些并发的细节。

2004 年 Google 提出了针对集群的 MapReduce 编程模型，极大的简化了并行编程 [?]，它以一种简单高效的方式执行大数据集的计算。通过使用 MapReduce 模型，程序员不需要考虑复杂的底层设计，只需要提供两个函数的实现:*map()* 和 *reduce()*; *map()* 函数处理输入数据集，并产生一系列的 key-value 对； *reduce* 函数用于将具有相同 key 的多个 value 归并，得道最终的结果。

1.2 Phoenix

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。之后，为了让 Phoenix 能够获得更好的 scalability，Yoo 等人对 Phoenix 进行较大的改进和优化，并提出了 Phoenix2 [?](在本文中我们将其简称为 Phoenix)。其他的多核 MapReduce 库如 Metis [5], Tiled-mapreduce [?]，MRPhi [8] 等，都从各个角度对 Phoenix 进行优化。

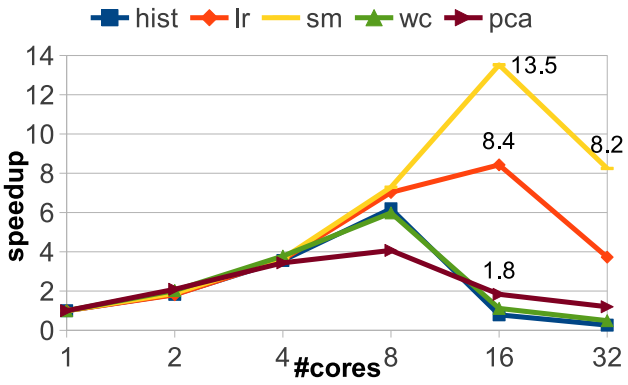


图 1: Speedup of workloads with Phoenix

Phoenix 利用基于共享内存的 Pthreads 库实现并发，运行时系统中的每个 worker 会与一个 Pthreads 线程进行绑定。理想情况下，向系统中增加更多的线程和核数，可以带来性能的提升，从而降低总的执行时间。然而，实现的结果显示，Phoenix 并不具有可伸缩性（实验的环境和配置细节将会在第??节详细讲述）。如图1所示，各个 MapReduce 应用程序在 Phoenix 上的运行时间。从实验结

果可以看出，尽管 Phoenix 能够让应用程序并发的执行，但是当核数超过 16 时，所有的应用程序的性能都开始变差。事实上，当核数超过 8 使，大部分的应用程序的性能已经开始下降。由此我们可以得出结论：多核环境下的 Phoenix 并不具有较好的 scalability.

2 Phoenix 的详细分析

这一小节的将详细的分析 Phoenix 存在的不足，并分析造成这些不足的根本原因。

2.1 Phoenix 的内部实现

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 [5]。Phoenix 的中间结构是一个全局的二维数组（如图7），因此，多个 map worker 可以同时向其中添加数据，而多个 reduce worker 也可以同时从中读取数据。为了避免多个 map worker 和 reduce worker 同时读写共享的中间结构，导致多线程间的竞争带来的数据不一致性，Phoenix 采用两种策略减少竞争：

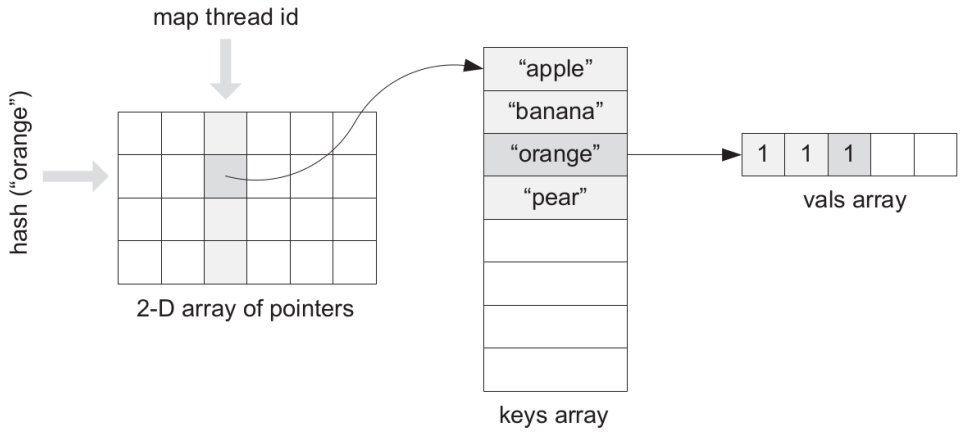


图 2: phoenix intermediate is a gobal array

- 对全局的二维数组进行划分：每个 map 对应二维数据的每一行，每个 reduce 对应每一列，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 之间加入屏障 barrier，即 Map 阶段结束后，才能开始 Reduce 阶段。从而可以避免 map 和 reduce 对同一区域的竞争。

通过采用以上两种策略，可以有效减少应用程序中锁的使用，从而简化程序，如实验结果显示 pthread_mutex 和 pthread_cond 的开销相当小。但是这些策略却限制了 Phoenix 的性能，接下来的章节中将会详细的解释。

2.2 scalability 的缺陷

Phoenix 中存在大量的堆对象的操作，因此 Phoenix 的性能和 scalability 受到内存分配器的影响。glibc 中自带的内存分配器 ptmalloc 对多核的可伸缩性较差 [2]，相比之下，jemalloc [3] 具有较好的可伸缩性。为了对比两种分配器对性能的影响，我们将 Phoenix 分别与 ptmalloc 和 jemalloc 进

行编译，记为 Phoenix-jemalloc 和 Phoenix-ptmalloc，并将应用程序在不同分配器下的运行时间进行对比。通过对比实验，结果如图3所示，对大部分应用程序，在核数较高的情况下，Phoenix-jemalloc 比 Phoenix-ptmalloc 具有更好的性能。

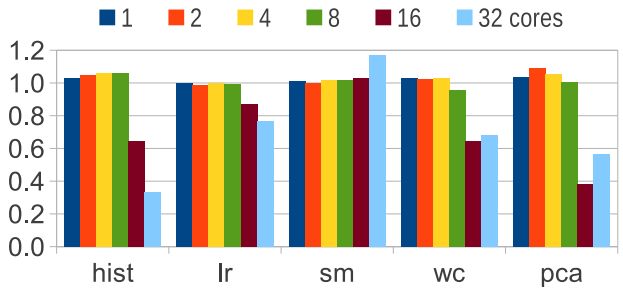


图 3: Execution time of Phoenix-jemalloc to Phoenix-ptmalloc

图1展示的是 Phoenix-ptmalloc 的 speedup。从图中我们可以看出，随着核数的增多，性能越来越差。这里我们将查看 Phoenix 使用性能较好的内存分配器 jemalloc 后，各应用程序的 speedup，实验结果如图4所示。从图中可以看出，从 1 到 8 核情况下，随着核数的增多，各个应用程序的性能越来越好。但是，当核数操作 8 核时，wc 和 hist 的性能越来越差；当核数超过 16 核时，lr, sm 和 pca 的性能越来越差。从实验的结果我们可以得出，即使使用性能较好的 jemalloc 内存分配器，Phoenix 的 scalability 也相当的差，当核数超过 16 时，基于 Phoenix-jemalloc 的应用程序的性能都会变差。

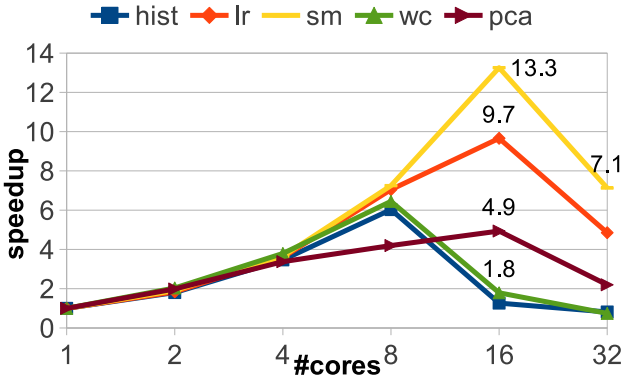


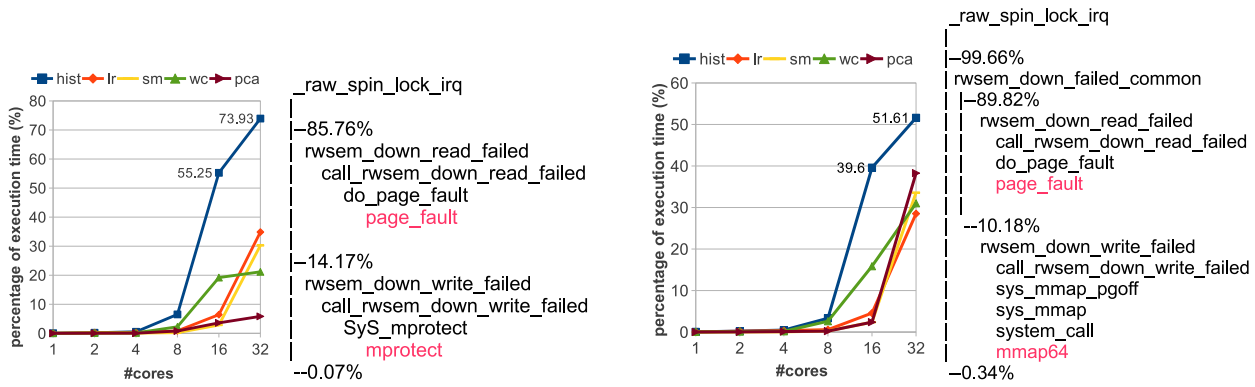
图 4: Speedup of Phoenix-jemalloc

为了分析 Phoenix 较差的 scalability，我们使用 Linux Perf [4] 收集热点函数的执行时间信息。结果显示，在核数较低的情况下，map 函数占用整个运行时间的比例最大，但是当增加更多核数时 (即超过 8 核时)，__ticket_spin_lock 函数将成为最热的函数。 __ticket_spin_lock 是 Linux 内核中的一种典型自旋锁，用于保护共享数据结构。锁的开销反映了 Linux 内核中的竞争情况，为了更加清楚分析这种自旋锁__ticket_spin_lock 对应用程序的影响，我们测试各个应用程序在不同核数情况下的竞争情况。

如图5给出了各个应用程序分别在 Phoenix-ptmalloc 和 Phoenix-jemalloc 情况下， __ticket_spin_lock 占用的时间比例。从实验结果中可以看出，当核数超过 8 时，__ticket_spin_lock 的开销占用将迅速上升。具体地，对于 hist 在 16 和 32 核情况下，__ticket_spin_lock 占用的总运行时间的比例分别为 39.6% 和 51.61%。实验结果证明，基于 Phoenix 运行的应用程序，当核数超过 8 时，将存在激烈的锁竞争。

由 Linux Perf 产生的函数调用栈显示，基于 Phoenx-ptmalloc 的 hist 中，__ticket_spin_lock 主要

来源于 page_fault 和 mprotect 系统调用，分别占用 85.76% 和 14.17%。而基于 Phoenix-jemalloc 的 hist 中，__ticket_spin_lock 主要来源于 page_fault 和 mmap 系统调用，分别占用 89.82% 和 10.18%。这两种不同的来源是由于不同分配器的原因。 __ticket_spin_lock 是由于多个线程对一个唯一的读写锁 mmap_sem 的竞争造成的。接下来的章节中，我们将详细的解释，为什么多个线程需要竞争一个唯一的 mmap_sem 信号量。



(a) Workloads with Phoenix-ptmalloc

(b) Workloads with Phoenix-jemalloc

图 5: The overhead cost by __ticket_spin_lock for workloads with Phoenix-ptmalloc or Phoenix-jemalloc

2.3 多线程地址空间的竞争

Phoenix 利用基于共享内存的 Pthreads 实现并行，基于 Phoenx 的应用程序中，所有的线程将共享同一进程的地址空间。在广泛使用的操作系统中，例如 Linux，进程的地址空间通常由多个线性区 (VMA) 组成，每个 VMA 代表一快连续的虚拟地址空间。Linux 中，进程的所有 VMA 通过链表链接起来，此外还会有一个红黑树来组织所有的 VMA。使用红黑树的目的是为了加快对某个特定 VMA 查找的速度，如图6所示。与红黑树对应的是一个读写信号量 mmap_sem，当多个线程需要同时修改和查找该红黑树时，首先需要获取 mmap_sem，以保证数据的一致性。

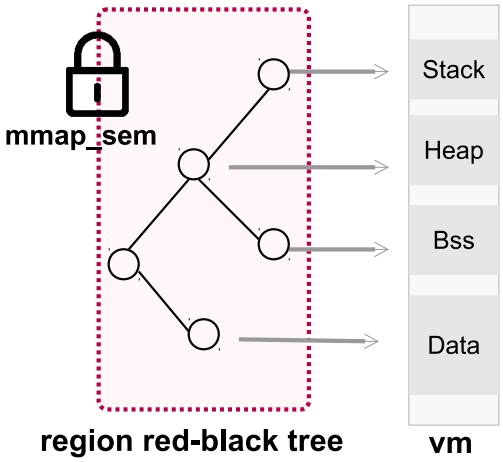


图 6: Address space is protected by mmap_sem

操作系统提供给用户的系统调用：mmap,munmap 和 mprotect 会修改红黑树（这些操作我们统称为内存映射操作）。具体表现为：mmap 会创建一个新的线性区 vma，并将其插入到红黑中； munmap 会

移除一块线性区，并调整红黑树；mprotect 则会修改红黑树中某个节点 (vma) 的访问权限域。除了这几种更新操作外，当进程访问一个非法的虚拟地址空间时，便会触发一个 page-fault，page-fault 的 handler 首先会查找地址空间（即查询红黑树），确定该地址是否为合法的地址，之后再做相应的处理。总之，多个线程共享一个进程的地址空间时，多个线程并发的产生，mmap,munmap 和 mprotect，或者 page-faults 时，它们会并发的访问或修改同一个红黑树，这需要内核有一定的同步手段，保证多个线程并发执行时的正确性。

为了保证进程地址空间数据的一致性，Linux 使用了一个读写信号量 mmap_sem 信号量, 由于同步（如图6所示），这是一个典型的读写信号量，它的特点是，同一时刻可以有多个读者同时持有锁，但只能有一个写着持有锁，且读者和写者不能同时获得锁。Linux 内核中，一个线程如果需要调用内存映射操作，那么它首先需要以写模式获取 mmap_sem 信号量，一旦获取后，便阻碍其他线程进行内存映射操作或 page-fault 的 handle 执行。相应地，当发生 page-fault 的时候，handler 函数的执行路径是，首先需要以读模式获取 mmap_sem 信号量。如果多个并发的内存映射操作和 page-fault 只能串行的执行。

2.4 Phoenix 的内部实现

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 [5]。

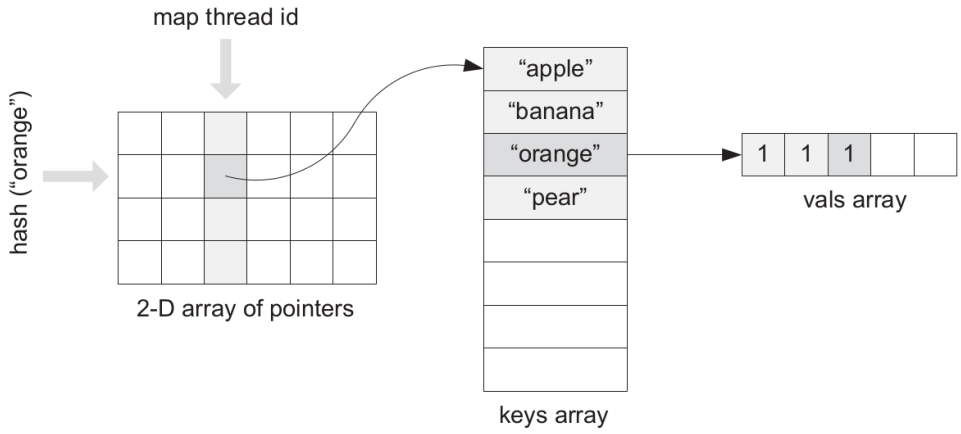


图 7: phoenix intermediate is a gobal array

Phoenix 中间结构是一个全局的二维数组（如图7），map worker 向其中增加数据，而 reduce worker 将读取数据，为了避免多个 map worker 和 reduce worker 对共享的中间结构竞争，保证数据的一致性，Phoenix 采用两种策略：

- 对全局的二维数组进行划分：每个 map 对应二维数据的每一行，每个 reduce 对应每一列，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。

通过采用这两种策略，可以有效减少应用程序中锁的使用，从而简化程序，如实验结果显示 pthread_mutex 和 pthread_cond

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。

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8的数据结果

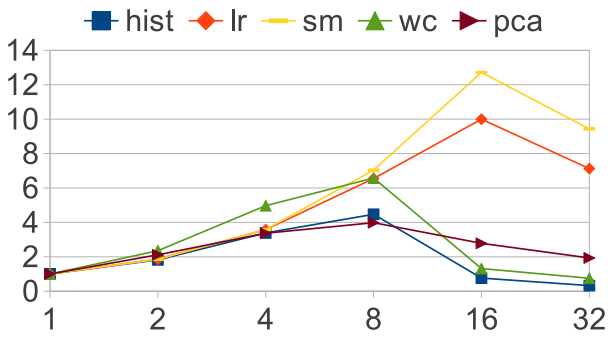


图 8: Phoenix 不开启 Combiner 情况下的 Speedup

(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 [6], 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 [7]，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。

总结：

3 总体设计

3.1 新的编程模型

基于 produce-consume 模型，以及隔离的地址空间的需求，我们不再使用 pthread 实现 SMR，而是重新实现了一个线程模型。这个线程模型与传统的 pthread 线程不同：(1) 其中每个线程都拥有自己独立的地址空间，线程之间相互隔离，不需要共享主进程的地址空间; (2) 线程之间有一个共享的 channel，用于线程间的通信。每个 map worker 和 reduce worker 就是这样一个线程，并且我们在初始化的时候，让每个 map worker 和 reduce worker 之间拥有一个共享的 channel，用于传递 map 产生的中间结果。

3.1.1 Phoenix 较差 scalability 的分析

必须结合应用程序的特点来说明问题： 是什么导致 speedup 下降（pagefault），为什么 hist, wc, sm 的 pagefault 会比较多，为什么 pca, sm 的 pagefault 会比较少？ 我们能够做的改进是不是只对那种 pagefault 比较读的应用程序有效果呢？

3.1.2 线程模型

Linux 操作系统中，使用一个 lock 来保护地址空间的目的是……

为了避免 produce-consumer 模型如图每个 map 线程的地址空间中有

如图8的数据显示，8 核以上，Phoenix 处理相同数据集的时间时间越来越长。我们试图通过性能工具 Perf []，深入分析影响 scalability 的主要原因。Perf 的实验结果显示，16 核与 32 核情况下，hist 中占用时间最多的函数是 ticket_spin_lock，分别为 40.5% 和 71.25%，即应用程序运行中的绝大部分的时间用于 spinlock，而未做时间的运算。通过分析 perf record 记录的函数调用栈，我们发现 ticket_spin_lock 是源于 page-fault。内核中 page-fault 函数需要操作 raw_spin_lock，该 spin_lock 是产生 ticket_spin_lock 的主要原因。

由于 Phoenix 采用 Pthread 线程模型，多个线程之间需要共享很多内核态数据结构，特别地，当多个线程并发执行时，会引起很多的 page-faults。Linux 内核源码显示，当发生 page-faults 时，线程首先需要获取进程的 mm_sem 信号量——该信号量属于一个进程私有的信号量，用于保护进程的映射表，以及 pagetable_lock 用于保护一个进程的页表。当进程的多个线程并发的访问 mm_sme 信号量时，将会导致线程对信号量的竞争。实际上，信号量是睡眠锁，等待该信号量的线程会被加入到等待

队列的末尾，然后进入睡眠状态，直到信号量来时，才会被唤醒继续运行。随着核数的增多，多个线程对 mm_sem 信号量的竞争会愈加激烈。实验结果9显示，Phoenix 中 spinlock 的开销所占的比例越来越大，严重影响系统的性能，以至于其 scalability 较差。

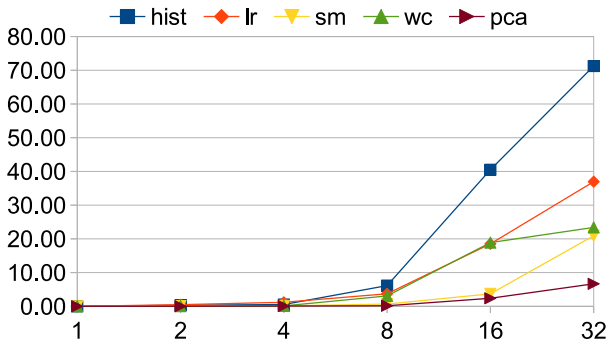


图 9: Phoenix 运行过程中 ticket_spin_lock 所占百分比

DMR 将不再采用 Pthreads 线程模型，而是使用进程。即每个 map worker 或 reduce worker 都是一个独立的进程。由于进程拥有自己独立的地址空间，拥有自己私有的 mm_sem，由于不需要与其他进程共享信号量，因此不存在锁的等待问题。这可以避免上述多个线程竞争读写信号量导致的 scalability 问题。然而，采用进程会带来一定的开销和不便，主要表现在两方面：(1) 相比线程，创建进程的开销比较大。(2) 线程是基于共享内存的，因此线程间的数据的共享较简单，而进程之间的共享需要一定的手段，DMR 中我们采用内存映射和隐式 queue 的方式，用于 map worker 和 reduce worker 之间数据的传递。进程的创建以及内存映射的环境构建，都在 DMR 的初始化部分完成。实验部分我们会详细分析 DMR 初始化的开销问题。

3.1.3 通道的特征

3.2 SMR 的 dataflow

这一节将介绍新的 mapreduce 系统 ***SMR*** 的运行时，相比已有的工作，***SMR*** 所做的改进工作

3.3 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。

3.3.1 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 [8] 为了使 Map 和 Reduce 并行执行，便是采用这种 Producer-Consumer 模型，具体如图10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

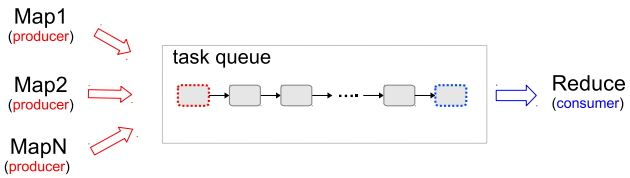


图 10: producer-consumer model in MRPhi

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 [8] 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。

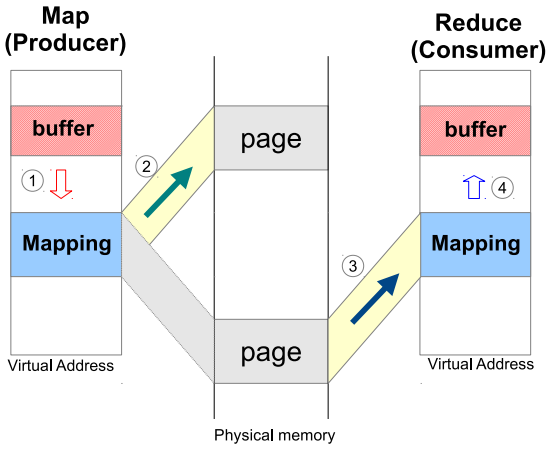


图 11: producer-consumer model in DMR

如图11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。

3.3.2 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 [5]。

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12。

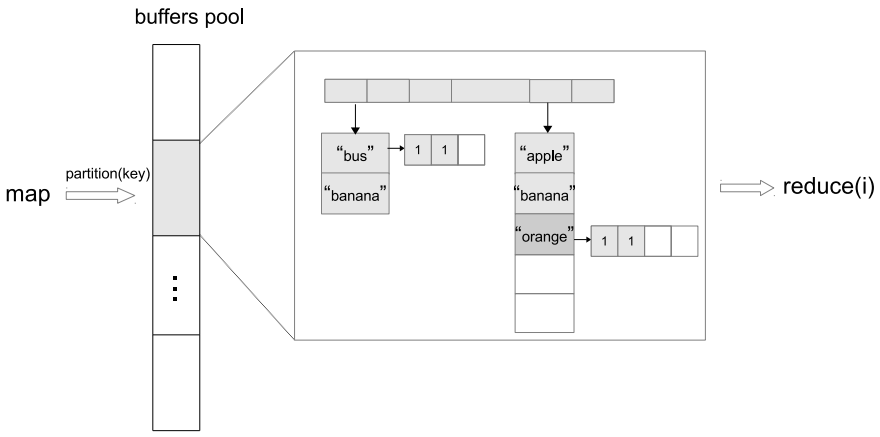


图 12: local buffer pool for each map worker

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13显示，左图表明，随着核数的增多，性能较平稳；右图为 DMR 与 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

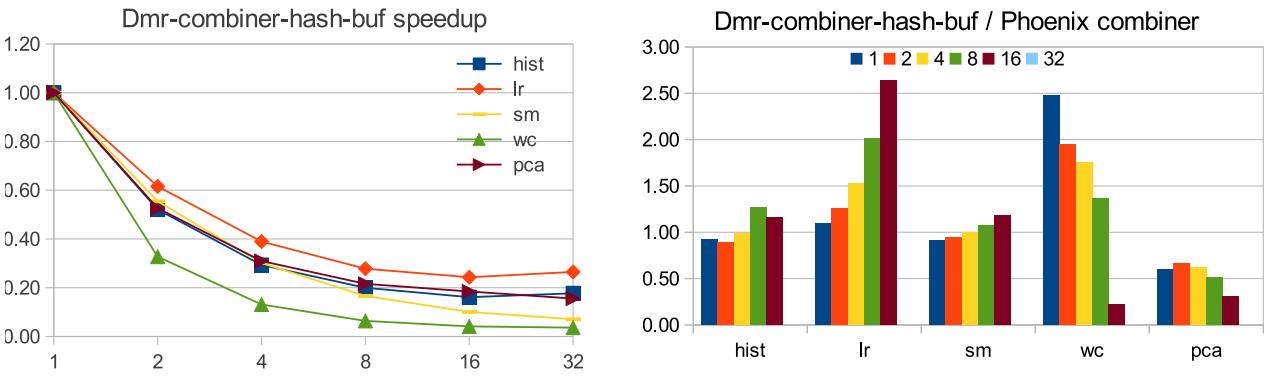


图 13: dmr compare to phoenix

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14左图所示。从而 array buffer 可有效提高性能

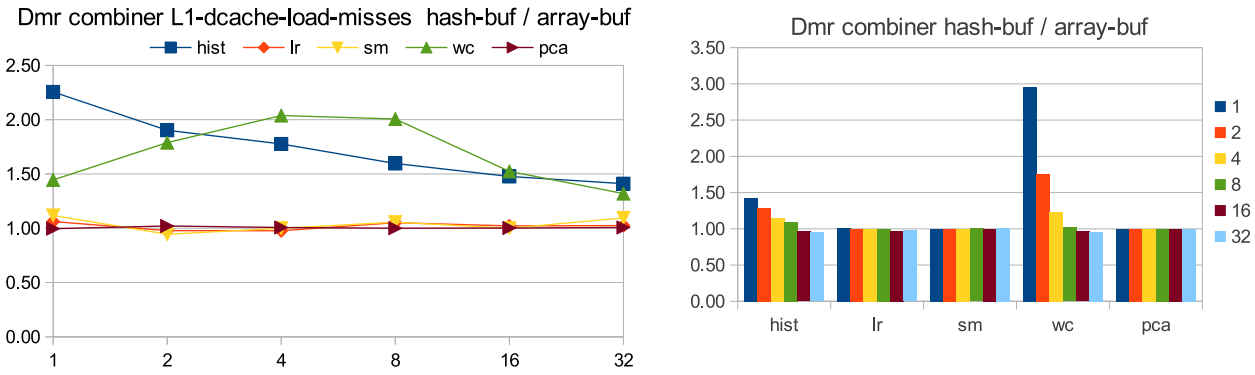


图 14: hash compare to array

优化后 DMR 与 Phoenix 对比的整体性能如图15

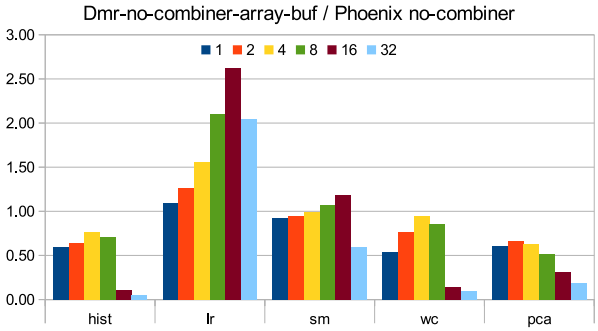


图 15: dmr compare to phoenix

4 实验结果与分析

4.1 Phoenix 较差 scalability 的实验结果

实验结果显示8，8 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9，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16。

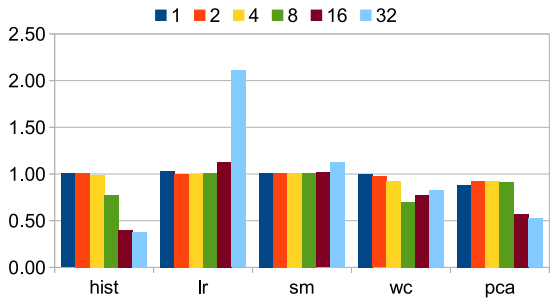


图 16: Phoenix-jemalloc / Phoenix-glibc no combiner

然而，即使 Phoenix-jemalloc 具有较好的性能，其 scalability 也是较差的，如图17， 我们可以看出，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%。

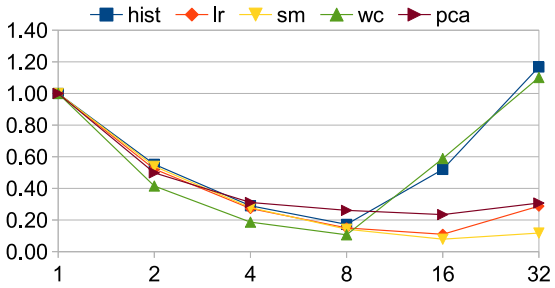


图 17: Phoenix jemalloc no combine speedup

将具有较好性能的 Phoenix-jemalloc 与 DMR 进行对比，实验结果如图18所示， 可以看出，除

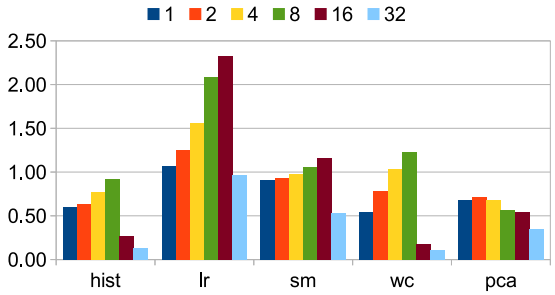


图 18: DMR-array / Phoenix-jemalloc no combine

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。

4.2 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，

4.3 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。

如图19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20)，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

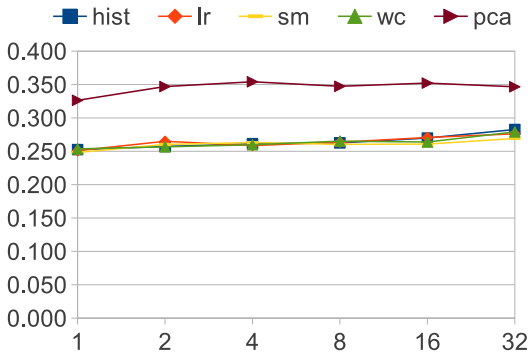


图 19: DMR environment initialize time(seconds)

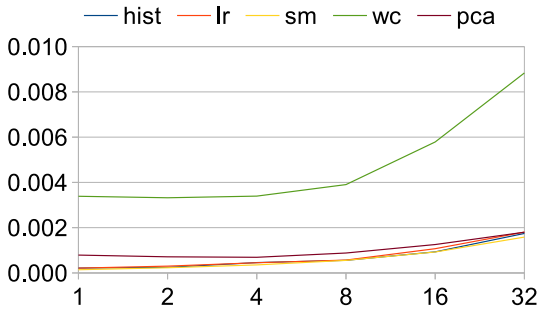


图 20: Phoenix environment initialize time(seconds)

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15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21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。

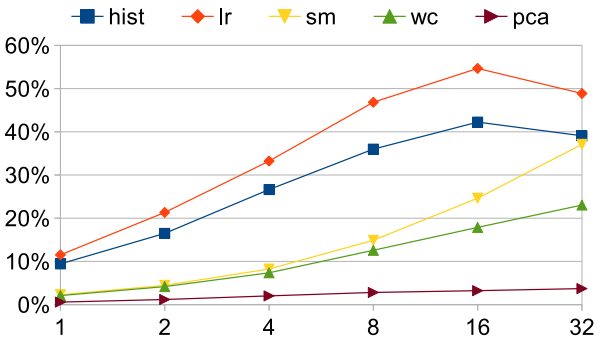


图 21: 环境初始化时间所占时间的百分比

4.4 benchmarks

4.4.1 SMR 中使用 benchmarks

应用程序的特点如图1

- 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	

4.4.2 已有工作中的 benchmark 调研

4.5 环境参数

4.5.1 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

4.5.2 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
```

5 附加

5.1 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

5.1.1 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

5.1.2 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

5.1.3 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

5.2 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)

5.2.1 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

5.2.2 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

5.2.3 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);
```

参考

[1] Shekhar Borkar. Thousand core chips: A technology perspective. In *44th Annual Design Automation Conference (DAC)*, pages 746–749, New York, NY, USA, 2007. ACM.

[2] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries, May 2006. http://www.malloc.de/en/index.html.

[3] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*, Ottawa, Canada, 2006.

[4] Arnaldo Carvalho de Melo. Performance counters on linux. In *Linux Plumbers Conference*, 2009.

[5] Yandong Mao, Robert Morris, and M. Frans Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[6] Shekhar Borkar. Thousand core chipsa technology perspective. pages 746–749, 2007.

[7] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *2nd International Workshop on MapReduce and Its Applications (MapReduce)*, pages 9–16, New York, NY, USA, 2011. ACM.

[8] Mian Lu, Lei Zhang, Huynh Phung Huynh, et al. Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. In *IEEE International Conference on Big Data*, pages 125–130, October 2013.