Int J Parallel Prog
DOI 10.1007/s10766-015-0390-5

DMR: A Deterministic MapReduce for Multicore Systems

Yu Zhang[1] · **Huifang Cao**[1]

Received: 30 March 2015 / Accepted: 20 May 2015

Abstract MapReduce has been shown promising to harness the multicore platform. Existing MapReduce libraries on multicore are written with shared-memory Pthreads, which introduce pervasive nondeterminism and might produce nondeterministic results if user-provided map or reduce functions are sensitive to the input order. We propose DMR, a deterministic MapReduce library, to ensure deterministic program behaviors no matter whether map/reduce function is sensitive to the input order. DMR adopts a round-robin scheduling of map tasks and a partitioned scheduling of reduce tasks to ensure deterministic scheduling. DMR is written with a deterministic message passing multithreaded model (DetMP) to provide Phoenix-like API, thus Phoenix workloads can be built and run on DMR with no or little change. Evaluation results by testing seven Phoenix workloads show that DMR only runs worse than Phoenix on an iterative MapReduce application kmeans, outperforms Phoenix between 1.42X and 3.33X faster on pca and word_count, and scales better than Phoenix on 3 of the rest 4 workloads.

Keywords MapReduce · Multicore · Deterministic parallelism

1 Introduction

The prevalence of multicore chips has raised the question of how applications can fully utilize multicore resources with ease-to-use programming models. MapReduce [1], a

✉ Yu Zhang
yuzhang@ustc.edu.cn

Huifang Cao
caobenzhi0915@gmail.com

[1] University of Science and Technology of China, Hefei, China

Published online: 06 October 2015

simple programming model originally designed for clusters, has been first shown promising to harness the multicore platform by Phoenix [2].

Existing MapReduce libraries on multicore [2–7] use threads to spawn parallel map or reduce tasks, and rely on shared memory to handle inter-task communications. They usually use a shared matrix of buckets to store intermediate data produced by map phase and consumed by reduce phase. To avoid contentions between map and reduce phases, most of them [2–5] enforce barrier synchronization between the two phases. This phase barrier causes the execution time of a MapReduce job is determined by the slowest worker in each phase, and requires the MapReduce runtime to keep all intermediate data through map phase. To improve resource utilization, Tiled-MapReduce [6] partitions a large job into a number of small subjobs, and uses a software pipeline to create parallelism among adjacent subjobs, however, it still keeps phase barrier inside the subjob. MRPhi [7] uses partition queues to pipeline map and reduce phases, where each queue is produced by multiple mappers and consumed by a single reducer.

No matter whether the phase barrier is used, MapReduce libraries on multicore usually use shared queues or a shared counter to dispatch tasks among mappers and/or among reducers. This causes that a mapper or a reducer may receive nondeterministic input in different runs of a MapReduce job. When user-defined map and/or reduce functions are *nondeterministic* [1], or the reduce function is not *commutative* and sensitive to the input order [8,9], the MapReduce job using such a library may produce nondeterministic program behavior, making it hard to debug and error-prone. An empirical study of MapReduce programs in production cluster has shown that non-commutative reducers are pervasive: 58% of the reducers examined are non-commutative [9].

To remedy these problems, this paper presents deterministic-MapReduce (DMR) written with a deterministic message passing multithreaded model—DetMP [10]. DetMP provides memory (including heap) isolation among threads by default, and only allows threads to communicate via deterministic channels declared in advance. The specific contributions of this work are:

- We present a round-robin scheduling of map tasks and a partitioned scheduling of reduce tasks to ensure each reducer receives deterministic input sequence in different runs. We abandon the phase barrier by using deterministic channels to pipeline map and reduce phases.
- We write DMR with DetMP to provide Phoenix-like API, thus Phoenix workloads can be built and run on DMR with no or little change. We show that DetMP simplifies produce-consume programming, but faces challenge of handling global variables and/or heap objects shared across phases in some MapReduce programs. We give solutions to the latter, which need add few new function calls into these programs to tell DMR to do special handling.
- DetMP is built atop our deterministic producer-consumer virtual memory emulated in Linux user space [11]. We evaluate DMR versus Phoenix 2.0 [3] across 7 Phoenix workloads on a 32-core machine. Evaluation results show that DMR only runs worse than Phoenix on an iterative MapReduce application kmeans, outperforms Phoenix between 1.42× and 3.33× faster on pca and word_count, and scales better than Phoenix on 3 of the rest 4 workloads.

The rest of the paper is organized as follows. We introduce background in Sect. 2. Section 3 gives the details of DMR, and Sect. 4 evaluates DMR with comparison to Phoenix 2.0. We conclude this paper in Sect. 5.

2 Background

This section uses Phoenix as an example to illustrate the implementation and nondeterminism of MapReduce using Pthreads, and then introduces DetMP.

2.1 The Phoenix Implementation on Multicore

Phoenix [2,3] is an implementation of MapReduce on shared-memory multicore systems using Pthreads. It shows that MapReduce can help applications perform competitively with hand-crafted Pthreads applications on a multicore platform.

Figure 1a shows the workflow of Phoenix from input to output, which mainly goes through the *map*, *reduce* and *merge* phases. The Phoenix runtime launches multiple threads to execute the computation. Each mapper thread repeatedly gets a chunk from the input using user-provided *split* function, processes the chunk using user-provided *map* function to generate intermediate key/value pairs, which reside in the corresponding row of Intermediate Buffer. A mapper also invokes the *combine* function (if provided by users) to perform local reduction for optimization at the end of the map phase. After all map tasks complete, reducer threads begin to work. Each reducer repeatedly gets intermediate data from the corresponding column of Intermediate Buffer, and applies user-provided *reduce* function to a set of intermediate pairs with the same key. Finally results from multiple reducers are merged and output.

2.1.1 Nondeterminism in the Runtime

To achieve load balance, the Phoenix runtime assigns map and reduce tasks to workers dynamically, introducing uncertainty in rows produced by a mapper or columns consumed by a reducer. In the left of Fig. 1b, nondeterministic map task assignment causes the reducer does reduction on different sequence of values in the column of

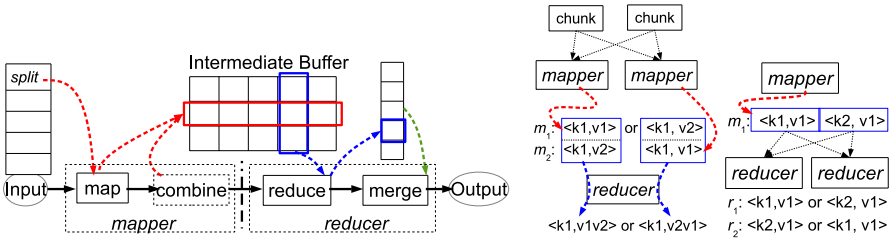

Fig. 1 The workflow and nondeterminism of the Phoenix library. **a** The workflow of Phoenix. **b** Nondeterministic examples

```
int m;
map(margs){
  ...
  if (...) m = true;
  ...
}
main() {
  m = true;
  map_reduce_init();
  ...initialize MR arguments
  while (m == true ) {
    m = false;
    map_reduce(mrargs);
    ...
  }...
  map_reduce_finalize();
}
```

(a) share global variable

```
map(margs){
  ...
  data = args-<data; ...
  data.out[...] = ...; ...
}
main() {
  ...
  data.out = (...)malloc(...);
  ...
  mrargs.task_data = &data;
  map_reduce (mrargs);
  map_reduce_finalize ();
  for(i = 0; i < ...; i++){
    ...= data.out[i]; ...
  }
  ...
}
```

(b) share heap object via pointer

```
// user-provided map and reduce
map(margs){
  ...
  v = (...)malloc(vsize);
  //set the content of key/value
  *k = ...;
  *v = ...;
  emit_intermediate(k, v, ksize);
}
reduce(k, vals){
  ...
  val_t *sum = (...)malloc(vsize);
  foreach v in vals {
    *sum += *v; free(v);
  }
  emit(k, sum);
}
```

(c) dynamic key/value objects

Fig. 2 Sharing modes used in some MapReduce applications

key k1, e.g., v1v2 for one run, or v2v1 for another. If the reduce function is sensitive to the order of its input column, it may generate nondeterministic result. In the right of Fig. 1b, even if the rows are deterministic, each reducer may process different columns in different runs and generate nondeterministic reduction result, leading to nondeterministic merge results depended on the order of all reduction results.

2.1.2 Nondeterminism Introduced by the Application

Some MapReduce applications share global variables and heap objects with the MapReduce runtime, which may cause threads launched by the runtime get access to them and update them in an arbitrary order. From the code of Phoenix workloads, we sum up several sharing modes. As shown in Fig. 2a, b, a global variable (e.g., variable modified in workload kmeans) or heap objects (e.g., those pointed by mm_data.output in matrix_multiply) might be updated by map function called by multiple mappers in an arbitrary order, respectively. For the well-written Phoenix workloads, whose map/reduce functions are not sensitive to the order of their input, such uncertainty cannot produce nondeterministic results. However, we argue that such sharing modes may encourage the programmer to use pointers to heap objects or global variables to communicate between tasks, thus the nondeterministic MapReduce runtime cannot anticipate any cases appearing in applications and cannot ensure deterministic behavior and correctness.

2.2 The DetMP Programming Model

DetMP is a deterministic message passing multithreaded model, which isolates memory among threads by default, and restricts threads to communicate via deterministic channels. Each DetMP thread has its own sub-heap for dynamic memory allocation, but inherits memory state (including heap) except channels from its parent (if exists) when started, and then does modifications in its own copy.

Table 1 Main functions in the DetMP API

Function	Description
int thread_alloc(int gid)	Allocate a child thread of global ID and return its internal ID
int thread_start(int tid, void *(*fn)(void*), void *args)	Start a child thread with specified internal ID to run fn(args)
void thread_join(int tid)	Wait for a child thread with specified internal ID to terminate
int chan_alloc(int nino)	Allocate a channel and return its ID
int chan_setprod(int cino, int tid, bool ascons)	Set the given child thread be the producer of the channel
int chan_setcons(int cino, int tid)	Set the given child thread be the consumer of the channel
size_t chan_send/chan_sendLast(int cino, void *buf, size_t sz)	Send a message stored in buf of sz bytes via the channel
size_t chan_recv(int cino, void *buf)	Receive a message from the channel and save it in buf

2.2.1 The DetMP API

Table 1 lists main functions in the DetMP API to manipulate threads and channels. The important programming abstraction in DetMP is the *deterministic channel*, which allows a single producer thread and any number of consumer threads to communicate deterministically, and frees the programmer from managing a FIFO buffer and its concurrency control using low-level synchronization such as locks or semaphores. To make deterministic write-read communication among a set of threads, a DetMP thread can create a channel by invoking chan_alloc, transfer the single produce permission to a child via chan_setprod call, and copy the consume permission to a child by invoking chan_setcons. After setting up producer-consumer relationship by invoking parent-to-child chan_setprod and chan_setcons functions, the relative threads can asynchronously send or receive each message in production order by simply invoking chan_send or chan_recv, respectively. It is the responsibility of the DetMP runtime to ensure deterministic parallelism at the message granularity.

2.2.2 The DetMP Runtime Atop the SPMC Memory

The DetMP programming model is initially proposed to explore programming models supported by our single-producer/multi-consumer (SPMC) virtual memory [10]. The SPMC memory is a race-free virtual memory foundation for deterministic write-read sharing at page granularity by distinguishing a single *producer* mapping and any number of *consumer* mappings to a page frame. It ensures any consumer successfully reads an SPMC page only after the producer explicitly makes the page visible by invoking setfixed primitive. We first extended the Determinator microkernel with the SPMC memory by using a eager page mapping mechanism [10], which lowers the utilization rate of page frames and only supports small programs. To generalize the SPMC memory, we proposed a lazy-tree-mapping mechanism to manage the SPMC memory, which allocates page frames lazily "on demand", and enables extend primi-

tive to remap a finite virtual address range to lazily-generated page frames, effectively representing an infinite stream. To support more realistic applications, we retrofited the SPMC memory into Linux, and implemented it entirely in Linux user space for quick prototype [11].

In this paper, the DetMP runtime used by DMR is based on the implementation in Linux user space. A DetMP thread is emulated using a single-threaded Linux process. The page tables as well as physical pages used by SPMC memory are both kept in mmap shared memory. Each channel wraps an SPMC region of limited address range (e.g., 512KB or less, at least 8KB) to pass unlimited messages based on the remapping mechanism provided by extend primitive. To enforce determinism at user-defined message granularity and achieve high memory utilization rate, we adopt *lazy-fix* policy to enable a page contains several messages, and make a page visible only when it is filled or when the producer invokes chan_sendLast.

2.2.3 Challenges in Implementing DMR Using DetMP

Since DetMP provides memory isolation among threads by default, it cannot support sharing modes like Fig. 2a, b, but make each mapper modify global variables or heap objects in its own copy. Moreover, as shown in Fig. 2c, applications such as pca, linear_regression and kmeans dynamically allocate and set key or value objects of user-defined size in map or reduce function, then read and free them in the next phase function. It is easy for Phoenix to share these objects via their pointers among threads since all threads share the heap. However, DetMP provides per-thread isolated sub-heaps, which causes a thread cannot get access to an object dynamically allocated by another concurrent thread. We need address challenge caused by the three sharing modes shown in Fig. 2 when using DetMP to write DMR.

3 The DMR Runtime

This section describes the details of DMR written with DetMP to provide API like Phoenix 2.0 [3], including its deterministic execution flow, and how to address the challenge caused by sharing modes on DetMP.

3.1 The Execution Flow of DMR

We introduce key design issues and considerations first, then the execution flow.

3.1.1 Determinism

The key to ensuring determinism in the MapReduce runtime is how to dispatch tasks to each mapper or reducer deterministically. To make each mapper process deterministic tasks (i.e., chunks), DMR splits the input into a number of chunks and inserts them to each mapper's task queue in a round-robin fashion. Thus each mapper can get deterministic chunks from its own queue, and invoke map function to emit key/value pairs into its local hash table. To make each reducer process deterministic tasks, DMR

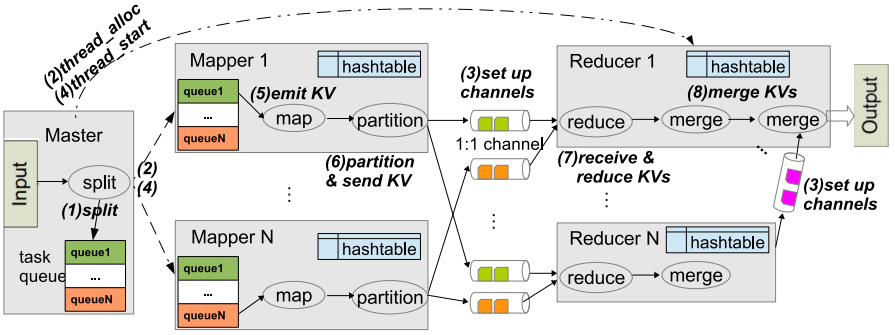

Fig. 3 Deterministic execution flow of DMR

introduces the *partition* phase in mapper, which partitions each mapper's hash table according to the same principle and sends partition i to reducer i. Each reducer further adopts a round-robin fashion to fetch partitions from all mappers, performs reduction on them and saves the result in its own hash table. Both round-robin scheduling and partitioned scheduling are deterministic, ensuring the determinism of DMR.

3.1.2 Pipelined Execution for Map and Reduce Phases

Unlike Phoenix, DMR breaks the phase barrier by pipelining the computation-intensive map and memory-intensive reduce to improve the hardware resource utilization. It introduces DetMP channels between each pair of mappers and reducers to send partitions of intermediate data from a mapper to a reducer, thus the reducer can perform reduction on received data before all mappers complete.

3.1.3 In-Mapper Combiner

To reduce the amount of communication between mappers and reducers, a mapper can combine its produced key/value pairs with the same key, and insert or update corresponding entries in its local hash table.

3.1.4 Application-Oriented Scheduler

A MapReduce job may declare reduce function as NULL. For such an application, Phoenix still performs reduction on the intermediate data using a default reduce function, which traverses key/value pairs. This approach is inefficient and may spoil the order of pairs generated by mappers. Unlike Phoenix, for such applications, DMR does not start the reduce phase and directly starts the merge phase after the map phase.

3.1.5 Execution Flow of DMR

Figure 3 gives the execution flow of DMR when an application provides the reduce function. Assume there are N available CPU cores, the DMR master, which is a DetMP

thread, initializes and destroys the DMR environment at the beginning and end of a map_reduce call, respectively. At initial stage, the master creates N task queues for N mappers, splits the input into chunks at user-specified granularity by default, then pushes them into N queues in a round-robin fashion. If the number of chunks is less than N, DMR would divide the input evenly to make each mapper have data to process.

After that, the master allocates threads by invoking thread_alloc for N mappers and N reducers (if the reduce function exists). It then creates and sets up N^2 channels for communication between each pair of mappers and reducers (if the reduce function exists), $(N-1)$ channels for binary-merge communication between reducers (or mappers if the reduce is NULL), as well as one channel to send final result from the first worker to the master. All channels have one producer and one consumer, which not only ensure determinism, but also obtain good locality. After setting up channels, the master starts all workers by invoking thread_start to inherit memory state except channels from the master and run.

Each mapper processes chunks from its corresponding task queue inherited from the master, and emits key/value pairs into its local hash table. Each mapper-local hash table is partitioned into N parts. Once the number of pairs in the hash table exceeds a specified threshold, the mapper sends partition i to reducer i via the corresponding channel. Each reducer receives data from N channels in a round-robin fashion, and might be blocked if it is attempting to access a channel which data is not ready. After a reducer receives a round of data from N mappers, it performs reduction on them by invoking the reduce function for each key, while mappers still run in parallel. If a reducer completes the reduction, it continues to receive subsequent data from mappers in the next round. A mapper would send a special message to notify all reducers when it has no more data to send. Results from all reducers (or mappers if no reduce function) are merged into the first worker's buffer in $log_2(N/2)$ steps by means of $(N-1)$ channels, and the final result is sorted and sent to the master.

3.2 Addressing the Challenge Caused by Sharing Modes

Sharing modes mentioned in Sect. 2 are not supported by DetMP, since DetMP gives each thread isolated memory by default, including global variables in data segment and dynamically-allocated objects in heap. For the cases in Fig. 2a, b on DetMP, the value of m or data.out in main() cannot be changed by the map_reduce call, even if some child thread spawned in execution of the map_reduce call modified it. For the case in Fig. 2c, although the reduce function can obtain pointers sent by a mapper, it cannot get access to the values written by the mapper indirectly through the pointers, since memory objects pointed by the pointers in the reducer are not the same as those in the mapper. Next we discuss our solution to address the challenges caused by these sharing modes.

3.2.1 Sharing Global Variables Between the Master and Workers

The case in Fig. 2a is abstracted from the kmeans workload, where m corresponds to modified in kmeans. We observe that each mapper updates (if it does) the variable

Table 2 Functions added by DMR and their calls added into some Phoenix workloads

Optional functions added by DMR	
`void set_copyargs(char flags, size_t sz)`	Tell DMR the characteristics of pointers whose contents need be sent via a channel, as well as the content size. flags might be KEY_COPY, VALUE_COPY, MAP_COPY and/or REDUCE_COPY, which are joint by '|'. The former two specify whether the key or value object is a pointer, and the latter specify in which phase the object is allocated
`void set_shareargs(char*addr, size_t sz)`	Tell DMR the starting address and size of a global variable to be shared

Modifications in Phoenix Workloads	
Application	Function calls added in the application code
linear_regression	`set_copyargs(MAP_REDUCE_COPY|VALUE_COPY, sizeof(long long));`
pca	`set_copyargs(KEY_COPY|MAP_COPY, sizeof(pca_cov_loc_t));`
kmeans	`set_copyargs(VALUE_COPY|REDUCE_COPY, sizeof(int)*dim);`
	`set_shareargs(& modified, sizeof(int));`

to the same value, so the final value of modified isn't affected by the update order of mappers. To make the master obtain the updated value of the global variable, DMR adds set_shareargs function into the API, as shown in Table 2. A MapReduce application can invoke it before invoking map_reduce to tell DMR which area in the data segment need be specially handled in execution of map_reduce call. DMR internally would check whether all threads update the area to the same value or not, and if exist difference, DMR adopts the *last-writer-wins* protocol according to the order that the threads were created.

3.2.2 Sharing Heap Objects Between the Master and Workers

The case in Fig. 2b is abstracted from the matrix_multiply workload, where data.out corresponds to mm_data.output. For heap objects allocated by the master and modified by workers, in order to let the master obtain the modified values, we enhance DetMP with a *twin-and-diff* mechanism [12] on the master's sub-heap. The extended DetMP runtime takes a snapshot of the master's sub-heap before starting any worker, creates a copy (*twin*) of the modified page modified by one or more workers, and performs diffs when the master joins workers. If multiple writers update the same address, DMR will update the master's same address in the last-writer-wins protocol mentioned above.

3.2.3 Sharing Dynamically-Allocated Heap Objects Between Workers

The case in Fig. 2c appears in three Phoenix workloads. To make the next phase worker get access to a pointer's content set by the previous phase worker, DMR internally sends the content via the channel, instead of just sending the pointer. DMR need find such a pointer and send its content, where the content size depends on applications. Thus DMR adds set_copyargs function, as shown in Table 2, and an application can invoke it to provide information about the characteristics of pointers before map_reduce call.

4 Evaluation

We perform our evaluation on a 32-core Intel 4× Xeon E7-4820 system with 128GB RAM, running Ubuntu 12.04.

Methodology. We evaluate the performance and scalability of DMR versus Phoenix 2.0 [3] across Phoenix workloads with the largest input set. Since the performance is significantly affected by the hash table size and chunk size, we keep the same settings for these parameters in DMR and Phoenix to make fair comparison. What's more, since Phoenix is heap allocation intensive, we build it with jemalloc-3.5.0 [13], a memory allocator for multicore systems, which is more efficient than that provided by glibc. All workloads were built as 64-bit executables with gcc -O3. We logically disable CPUs using Linux's CPU hotplug mechanism, which allows to disable or enable individual CPUs by writing "0" (or "1") to a special pseudo file (/sys/devices/system/cpu/cpuN/online), and the number of workers in the map or reduce phase equals to the number of CPUs enabled. Each workload is executed ten times, and the lowest and highest execution times for each workload are discarded, so each result is the average of the remaining eight runs.

Compatibility. All Phoenix workloads are built and run with DMR with no or little change. Table 2 lists all modifications on Phoenix workloads to fit for DMR. We see that it is easy to make Phoenix programs compatible with DMR.

4.1 Determinism

The determinism of DMR is ensured by the following points: (1) The splitted chunks are dispatched to each mapper in a round-robin fashion, which ensures each mapper processes the same chunks of the same order in different runs. (2) Intermediate data are shuffled to reducers deterministically. (3) Mappers and reducers communicate with deterministic channels. Both 2 and 3 ensure each reducer processes the same intermediate data in different runs. (4) DMR internally checks if there exists data race when multiple workers update application-defined heap objects and global variables concurrently, and then updates them into the master's space in a deterministic order. We also experimentally verify DMR's ability to ensure determinism by collecting the result of each reducer for many runs. For each reducer, DMR reports the same result all the time.

4.2 Performance and Scalability

4.2.1 Performance

We compare the performance of DMR to Phoenix. Figure 4a presents the results graphically (normalized to Phoenix). DMR matches or outperforms Phoenix on 6 out of 7 workloads, but runs worse than Phoenix only on kmeans. For pca and word_count, DMR outperforms Phoenix between 1.42× and 3.33× faster. For linear_regression and histogram, DMR performs on par with Phoenix at 1–8 CPU cores, but scales better on 16 and 32 cores. DMR ensures determinism but still results in good performance for several reasons:

(1) DetMP emulates thread in Linux process, which isolates updates in separate processes and can improve performance by eliminating false sharing.
(2) A DetMP channel can buffer unlimited messages by invoking the SPMC extend primitive, thus its producer cannot be blocked.
(3) The application-oriented scheduler makes applications with null reduce function run more efficiently than Phoenix. For example, pca and the second job of word_count declare reduce function as NULL. Results at 1 CPU core show that such a scheduler reduces 42.8, 29.8 % runtime for pca, word_count, respectively.
(4) The pipeline execution for phases and in-mapper combiner can significantly improve performance for applications with a large number of duplicate keys such as word_count. Running word_count at 1 CPU core with 100MB dataset, the in-mapper combiner can decrease $n = 14863618$ pairs that need be sent to reducers before optimization, occupying 82.84 % of total pairs in map phase. The decreased number for N CPU cores equals to n/N. Using combiner and pipeline, reducer in word_count on DMR takes only 3 % execution time of that on Phoenix.

kmeans is an iterative application, e.g., including 82 map_reduce calls at 1 CPU core. The reason of worse performance on kmeans is that DMR currently does not efficiently support iterative MapReduce applications, but simply initializes and destroys all DMR environment at each map_reduce call, including the DetMP thread creation and join. This can be solved in the future by extending DetMP with a thread pool support, which allows dynamically binding tasks and data to threads on the premise of ensuring determinism.

4.2.2 Scalability

Figure 4b presents the scalability of DMR and Phoenix, marked with suffixes -D and -P, respectively. DMR scales better than Phoenix on histogram (Hist), linear_regression (LR), and string_match(SM). DMR matches the scalability of Phoenix on matrix_multiply(MM). DMR scales worse than Phoenix on pca (PCA) and word_count (WC), but performs faster than Phoenix on them.

We next analyze the reason why Phoenix scales worse on three workloads. For histogram, we find that mappers concurrently read global shared variables, e.g., *red_keys[256]*, *blue_keys[256]*, *green_keys[256]*. This causes Linux suffers from contention in down_read_trylock due to memory management. We further

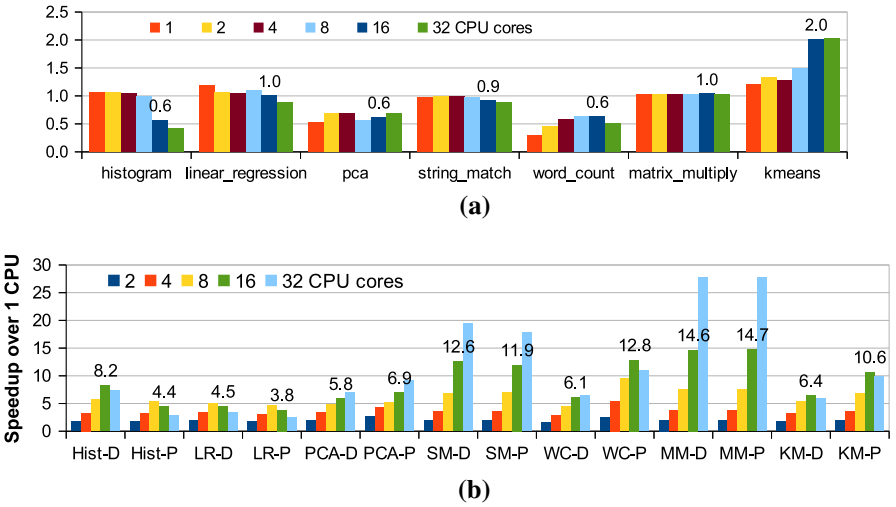

Fig. 4 Performance and scalability of DMR versus Phoenix. **a** Execution time of DMR relative to Phoenix. **b** Speedup over single-CPU performance on various workloads

use Linux Perf to run histogram with Phoenix. Results show that overhead of down_read_trylock increases heavily as the number of CPU cores increases, and it even becomes the hottest function, occupying 16.22, 25.79 % of the total runtime at 16, 32 CPU cores, respectively. However, DetMP emulates thread with process, which makes each mapper run in its own address space, avoiding such contention in Linux.

For linear_regression and string_match, Phoenix suffers from lock contention in ticket_spin_lock due to the lock used by reducers to preempt shared tasks. Perf results of linear_regression with Phoenix show that ticket_spin_lock takes only 9.17 % of total runtime at 8 CPU cores, but grows rapidly to 38.02, 42 % at 16, 32 cores, respectively. However, DMR does not encounter such lock contention since intermediate data produced by a mapper are partitioned to reducers.

4.3 Others

4.3.1 Overheads on Handling Sharing Modes

As shown in Sect. 3.2, DMR adopts a twin-and-diff mechanism to handle heap objects sharing between the master and workers, which appears in matrix_multiply. Experimental results show that the twin-and-diff mechanism is less expensive and only leads to 0.05 s extra cost with input dataset of 2000×2000 matrix, totally 16MB data. For the case in Fig. 2c, which appears in linear_regression, pca and kmeans, DMR need copy the content of dynamically allocated key (or value) objects. Results show that the copy cost for linear_regression and kmeans is almost zero, while pca spends 0.2 s copying 64MB data at 1 CPU core, but decreases to 0.01 s at 32 CPU cores.

4.3.2 Load Balancing

As discussed in Sect. 3, DMR partitions the splitted chunks and intermediate data statically. Generally speaking, static partition may incur load imbalance. To see how it influences, we collect the workload of each map and reduce worker at 4 CPU cores. For all workloads used in this paper, experimental results show that the load of each mapper is balanced. For histogram and linear_regression, the default partition function cannot partition the pairs in balance, causing imbalance load for each reducer. This would be settled by providing a more suitable partition function. We further observe that there is little difference between the execution time of each reducer.

5 Conclusion

DMR is a deterministic replacement for Phoenix. Evaluation results show that DMR matches or outperforms Phoenix for 6 out of 7 workloads examined here, making DMR a safe and efficient alternative to Phoenix for many applications. However, the static partition policy used in DMR incur load imbalance, and total thread creation and destruction at each map_reduce call causes inefficiency for iterative applications. In the future, we will focus on extending DetMP with a thread pool support, which allows dynamically binding tasks and data to threads on the premise of ensuring determinism.

Acknowledgments This work was supported in part by the National Natural Science Foundation of China under Grant No. 61170018, 61229201, the National High Technology Research and Development 863 Program of China under Grant No. 2012AA010901.

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: 6th OSDI, pp. 10–10. USENIX Association, Berkeley, CA, USA (2004)
2. Ranger, C., Raghuraman, R., Penmetsa, A., et al.: Evaluating MapReduce for multi-core and multiprocessor systems. In: 13th HPCA, pp. 13–24. IEEE Computer Society, Washington, DC, USA (2007)
3. Yoo, R.M., Romano, A., Kozyrakis, C.: Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In: IISWC '09, pp. 198–207. IEEE Computer Society, Washington, DC, USA (2009)
4. Mao, Y., Morris, R., Kaashoek, M.F.: Optimizing MapReduce for multicore architectures. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical report (2010)
5. Talbot, J., Yoo, R.M., Kozyrakis, C.: Phoenix++: modular MapReduce for shared-memory systems. In: 2nd MapReduce, pp. 9–16, New York, NY, USA, ACM (2011)
6. Chen, R., Chen, H., Zang, B.: Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In: 19th PACT, New York, NY, USA, ACM. pp. 523–534 (2010)
7. Lu, M., Zhang, L., Huynh, H.P., et al.: Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. In: BigData Congress '2013, pp. 125–130 (October 2013)
8. Csallner, C., Fegaras, L., Li, C.: New ideas track: testing MapReduce-style programs. In: ESEC/FSE '11, pp. 504–507. ACM, New York, NY, USA (2011)
9. Xiao, T., Zhang, J., Zhou, H., et al.: Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs. In: 36th ICSE, pp. 44–53. ACM, New York, NY, USA (2014)
10. Zhang, Y., Ford, B.: A virtual memory foundation for scalable deterministic parallelism. In: 2nd APSys (July 2011)

11. Zhang, Y., Ford, B.: Lazy tree mapping: generalizing and scaling deterministic parallelism. In: 4th APSys (July 2013)
12. Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Implementation and performance of Munin. In: 2nd PPoPP (October 1991)
13. Evans, J.: A scalable concurrent malloc (3) implementation for FreeBSD. In: BSDCan Conference, Ottawa, Canada (2006)