Future Generation Computer Systems 36 (2014) 19–30

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Master–worker model for MapReduce paradigm on the TILE64 many-core platform

Xuan-Yi Lin, Yeh-Ching Chung [*]

Department of Computer Science, National Tsing Hua University, 101 Section 2, Kuang Fu Road, Hsinchu City 30013, Taiwan

HIGHLIGHTS

- We model two shared memory master–worker programming schemes for TILE64.
- We apply proposed schemes to MapReduce paradigm.
- Analysis shows that the worker share is superior to the master share scheme.

ARTICLE INFO

Article history:
Received 3 January 2013
Received in revised form
1 May 2013
Accepted 7 May 2013
Available online 16 May 2013

Keywords:
Many-core
Master–worker
MapReduce
Shared memory
TILE64

ABSTRACT

MapReduce is a popular programming paradigm for processing big data. It uses the master–worker model, which is widely used on distributed and loosely coupled systems such as clusters, to solve large problems with task parallelism. With the ubiquity of many-core architectures in recent years and foreseeable future, the many-core platform will be one of the main computing platforms to execute MapReduce programs. Therefore, it is essential to optimize MapReduce programs on many-core platforms. Optimizations of parallel programs for a many-core platform are viewed as a multifaceted problem, where both system and architectural factors should be taken into account. In this paper, we look into the problem by constructing a master–worker model for MapReduce paradigm on the TILE64 many-core platform. We investigate *master share* and *worker share* schemes for implementation of a MapReduce library on the TILE64. The theoretical analysis shows that the *worker share* scheme is inherently better for implementation of MapReduce library on the TILE64 many-core platform.

1. Introduction

In recent years, the industry undergoes a transition from single core processors to the integration of multiple cores to produce multi-core and many-core processors due to power envelope restrictions [1]. While the trend of processor manufacturing is to increase the number of cores rather than clock frequency [2,3], software developers can no longer rely on the so called “free lunch” [4] that automatically makes existing programs run faster on processors clocked at higher frequencies.

In order to make performance of a program scaling well with the number of available cores on a multi-core or many-core platform, existing software need to be modified or re-written from ground up [5,6]. Efforts involving parallelization of an application are twofold, known as *design* and *implementation*. The former is about finding concurrency in a given application and to derive algorithms and program structures to make it run faster, while the latter is about utilization of available programming resources on the designated parallel platform to realize the designed algorithm and structure. The available programming resources include programming language, programming paradigm, and API (application programming interface), among others. Due to the flexibility of available options, there may be possible several implementations for a single design on a platform. Performance and scalability characteristics of completed applications may vary with different implementations. Thus, it is important to set guidelines for developers to follow in order to produce better programs on a given platform.

TILE64 is a family of general purpose many-core processors designed and manufactured by Tilera [7]. Fig. 1 shows the architecture overview of a TILE64 processor. A TILE64 processor contains a two-dimensional array of 64 identical processor cores interconnected via multiple on-chip mesh networks named iMesh. The iMesh is designed to be scalable to large number of cores while maintaining low-latency communication between tiles. Tilera provides a set of proprietary APIs called *iLib* for programmers to write application programs. The iLib provides both *shared memory* and *message passing* primitives for implementation of inter-process communication. The availability of different and varied implementation options adds both flexibility and complexity in building parallel programs on this platform.

[*] Corresponding author. Tel.: +886 35742971.
E-mail addresses: xylin@cs.nthu.edu.tw (X.-Y. Lin), ychung@cs.nthu.edu.tw (Y.-C. Chung).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.05.001

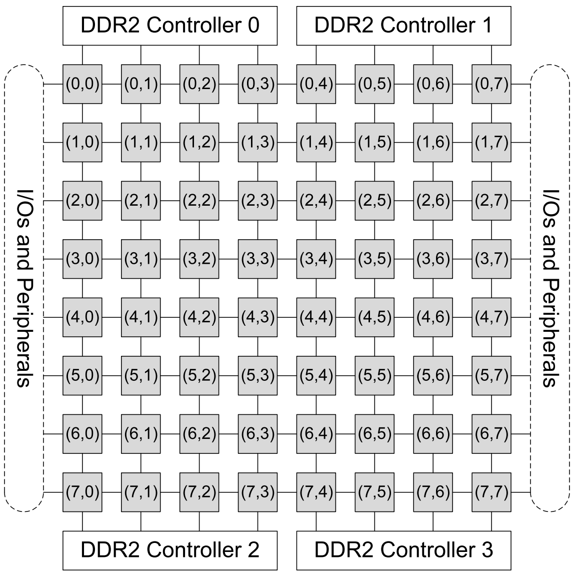


Fig. 1. TILE64 processor architecture overview.

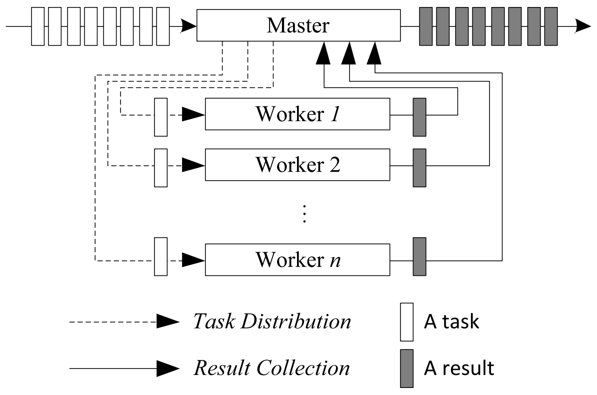


Fig. 2. A master–worker model.

The master–worker model has been successfully used in many research areas. It is often adopted when there is a need to dynamically balance workloads among available processors [8,9], especially in large distributed computing environments such as clusters [10], grids [11], clouds [12] and even on petascale resources [13]. In addition to applications in distributed computing environments, with the recent availability of multi-core and many-core processors, the master–worker model can also be adopted in smaller-scaled systems [14]. Fig. 2 shows a generic master–worker model, which consists of two main parts, *task distribution* and *result collection*. In the task distribution part, the master generates a set of tasks and distributes them to the workers. The master can be seen as a producer and the workers can be seen as consumers. Notwithstanding, in the result collection part, the master collects computation results generated by the workers. Thus, the workers can be seen as producers and the master can be seen as a consumer.

The MapReduce [15] paradigm has been successfully practiced on cluster systems for large scale distributed problems and the process of big data. It utilizes the master–worker model to schedule and dispatch computational tasks over a large set of distributed computers. In addition to the proprietary in-house implementation by Google Inc., there are also open source MapReduce implementations such as Hadoop [16], which is written in Java, and Phoenix [17,18], which is written in C. The Hadoop implementation is primarily deployed in distributed and loosely coupled environments. The Phoenix implementation is developed mainly for shared-memory architectures such as multi-core and SMP systems. The MapReduce paradigm can be adopted in many different application domains such as scientific computing, artificial intelligence, enterprise computing and image processing.

Although there are large amount of papers that discuss the applications of the master–worker model on a number of systems or platforms, only a few papers are related to the applications of master–worker model on many-core platforms. In addition to that, although there are MapReduce implementations that target multi-core shared-memory systems, it is not yet fully investigated the scalability of the implementations on a many-core platform with on-chip interconnection networks such as TILE64. With the ubiquity of many-core architectures in recent years and foreseeable future, the many-core platform will be one of the main computing platforms to execute MapReduce programs. It is important to explore the problem of mapping traditional models onto many-core platforms.

In this paper, we study how to develop a scalable and high performance MapReduce library similar to that of Phoenix on the TILE64 many-core platform. The management of the communications between master and worker processes is the key to the success of such development. We propose two shared memory schemes, *master share* and *worker share*, to implement the shared memory communication between master and worker processes. We model and compare these two schemes and conclude that the *worker share* scheme is superior to the *master share* scheme on the TILE64 many-core platform.

The rest of this paper is organized as follows. Section 2 provides background knowledge of TILE64 and the approach of carrying out shared memory communication between two processes on the TILE64. In Section 3, a master–worker MapReduce system is described and the master share and worker share schemes are introduced. Theoretical analysis is carried out in Section 4. Concluding remarks and future work are given in Section 5.

2. Preliminaries

2.1. TILE64 processor

The TILE64 processor is a many-core processor featured as an array of 64 identical processor cores (each referred to as a *tile*) interconnected via the on-chip two-dimensional mesh network. The TILE64 is fully programmable using standard ANSI C under Linux environment, including a set of proprietary APIs called *iLib*. The *iLib* library supports two communication mechanisms, shared memory and distributed memory, for processes running on different cores to communicate with each other.

The TILE64 platform has an on-chip network named iMesh to interconnect all 64 processor cores. All inter-process communications in a multi-process program will be translated into underlying network traffic, which is fully transparent to programmers. As a process is executing load/store instructions, it does not necessarily having the knowledge of the overheads on the underlying network traffic. Thus, when multiple processes are concurrently accessing memory devices, the generated network traffic can sometimes overwhelm the network, causing traffic congestions and routing delays, which will directly affect program performance. The inter-process communication should generate as little network traffic as possible such that the overall network performance on this many-core platform would not be pushed down.

A previous study [19] suggests that programmers can implement applications in a way where producer processes always write data directly into memory addresses shared by consumer processes to avoid unnecessary cache coherent traffics on the memory network. In the literature, there are some discussions of

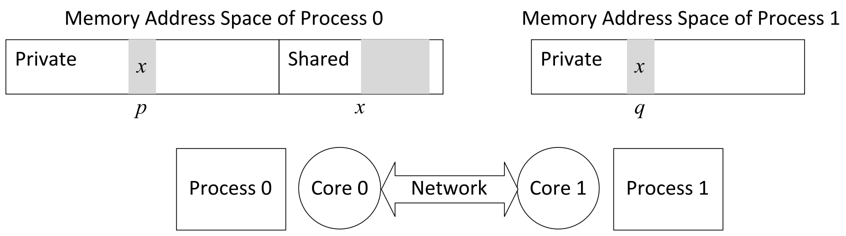

Memory Address Space of Process 0
Private
x
Shared
p
x
Memory Address Space of Process 1
Private
x
q
Process 0
Core 0
Network
Core 1
Process 1

Fig. 3. Sharing of an integer on TILE64 between two processes.

scalability issue on many-core processors featuring on-chip networks or multiple memory controllers [20,21]. In our previous work [22], we have shown that it is necessary to consider the memory hierarchy and on-chip networks in order to develop high performance applications on the TILE64 platform.

2.2. Shared memory communication on TILE64

In TILE64, shared memory communication allows each process in a parallel application to load/store values from/to a globally visible region of memory. Concurrent accesses to shared objects must be synchronized with mutex (mutual exclusion) locks to prevent inconsistent states.

Both the Linux and *iLib* programming environments provide tools for allocating and synchronizing accesses to the shared memory. Linux allows programs to allocate and synchronize using the standard Unix shared memory and pthreads APIs, while *iLib* supports a special function for shared memory allocation, *malloc_shared*(), as well as an implementation of a pthreads-style mutex lock. To use *iLib* to implement shared memory mechanisms in a program, the process which shares information can call the *malloc_shared*() function to get an address pointing to a block of shared memory. Then the process notifies other processes the location of shared memory by sending them messages containing this address.

Fig. 3 shows an example on the use of *iLib* to create an integer object shared between 2 processes. The initialization steps are as follows:

- There are two cores, each executes one process;
- Process 0 allocates a region of memory to hold one integer using *malloc_shared*();
- The *malloc_shared*() function returns a value x, which is the address of the shared integer. The value of x is stored in an integer pointer p in process 0;
- Process 0 sends content of p to process 1;
- Process 1 stores this address with integer pointer q.

After above initialization sequence, both processes 0 and 1 will be able to load from and store to this shared integer in the same way as normal variables. Any update to $^*q$ made by process 1 can be seen by process 0 using $^*p$, and vice-versa.

3. Master–worker model for MapReduce paradigm

Given an input dataset to be processed by a MapReduce program, we assume that the input dataset can be divided into n tasks that can be independently processed and outputted. The input dataset can be represented as a set of input data fi_1 to fi_n, and the output dataset is represented as fo_1 to fo_n. Assume that the application is run on a processor, each task i takes time t_i to be processed from input format to output format. The time to process all segments is:

$$\sum_{i=1}^{n} t_i. \quad (1)$$

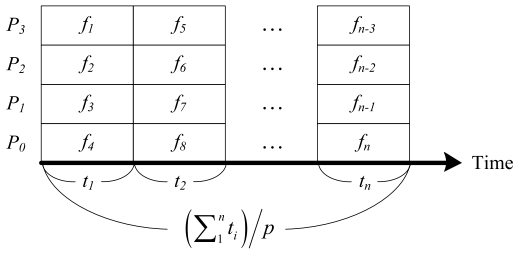


Fig. 4. Perfect task scheduling on 4 processors.

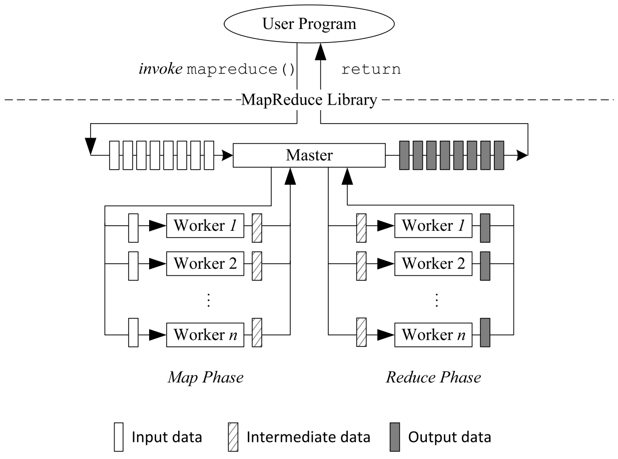


Fig. 5. Execution overview of master–worker MapReduce library.

The ideal case of processing such dataset using p processors would be similar to the one shown in Fig. 4. In such ideal case, $t_1 = t_2 = \cdots = t_n$ and n is an exact multiple of p. So the time needed to process all segments becomes:

$$\frac{\sum_{i=1}^{n} t_i}{p}. \quad (2)$$

This leads to a perfect speedup of p. In reality, it may take variable amount of time to process different data segments, and n is commonly not an exact multiple of p.

A master–worker system consists of a master process managing a set of worker processes. The master process distributes tasks to a set of subordinate worker processes and later collects computed results. There are two task pools in a master–worker system, the *pool of pending tasks* and the *pool of completed tasks*. Once a worker finishes a task, the worker process fills the result to the pool of completed tasks. The master process then fetches results from the pool of completed tasks and outputs the results.

Fig. 5 illustrates the execution overview of master–worker MapReduce library. At the beginning, a user program sets up essential information and invoke mapreduce(). In the map phase, all workers take split parts of the input data to compute according to user defined map() function to generate key-value pairs stored as intermediate data. Then in the reduce phase, all workers compute final results by running user defined reduce() function over the intermediate data.

During the progress of task distribution, a master process is considered a producer process and worker processes are considered consumer processes. Meanwhile, one-to-many communication is raised. On the other hand, in the progress of result collection, worker processes are considered producer processes and a master process is considered a consumer process. Meanwhile, many-to-one communication is raised.

The total time to process all tasks can be derived as:

$$t_{\text{total}} = t_{\text{read}} + t_{\text{fill}} + t_{\text{drain}} + t_{\text{write}} + t_{\text{comp}} + t_{\text{sync}} + t_{\text{idle}}. \quad (3)$$

Since time spent by workers are essentially overlapped with time spent by the master, so the total time only counts time spent by the master. Following is a list of detailed description of components in (3):

- t_{read}: time master spent reading input data from input to memory;
- t_{fill}: time master spent storing all pending tasks into pool of pending tasks;
- t_{drain}: time master spent loading all pending tasks from pool of completed tasks;
- t_{write}: time master spent writing output data from memory to output;
- t_{comp}: time master spent on computation such as decomposing input data and composing output data;
- t_{sync}: time master spent waiting for mutex locks to gain access to shared objects;
- t_{idle}: time master spent idling.

Of all the above 7 components, t_{read}, t_{write} and t_{comp} can be seen as constants for a given input dataset, that is, these three timing values are not affected by system configuration variables such as number of workers, size of task pools and how inter-process communications are carried out.

To look into more detail of the performance characteristics we further derive:

$$t_{\text{fill}} = \frac{S_{\text{input}}}{\omega_{\text{master}\to\text{pending}}} \quad (4)$$

where S_{input} is the total size of input data, and $\omega_{\text{master}\to\text{pending}}$ is the average throughput for master to store data into the pool of pending tasks, and

$$t_{\text{drain}} = \frac{S_{\text{output}}}{\omega_{\text{master}\leftarrow\text{completed}}} \quad (5)$$

where S_{output} is the total size of output data and $\omega_{\text{master}\leftarrow\text{completed}}$ is the average throughput for master to load data from the pool of completed tasks. From (4) and (5) we know that by increasing $\omega_{\text{master}\to\text{pending}}$ and $\omega_{\text{master}\leftarrow\text{completed}}$, t_{fill} and t_{drain} can be shortened.

As for the synchronization time t_{sync}, it can be seen as a function of two variables:

$$t_{\text{sync}} = \mathbb{F}(p, q) \quad (6)$$

where p is the number of shared objects in the system and q is the number of participating processes wishing to access the shared objects. Usually the t_{sync} will grow rapidly with the increment of p and q.

The master idle time t_{idle} will come into place when both of the following conditions are true: (a) pool of pending tasks is full, and (b) pool of completed tasks is empty. The occurrence rate of condition (a) is decided by pool size, $\omega_{\text{master}\to\text{pending}}$ and $\omega_{(\text{worker}\leftarrow\text{pending})\text{aggregated}}$, where the latter represents aggregated throughput for all workers to load data from the pool of pending tasks. Similarly, the occurrence rate of condition (b) is decided by pool size, $\omega_{\text{master}\leftarrow\text{completed}}$ and $\omega_{(\text{worker}\to\text{completed})\text{aggregated}}$, where the latter represents aggregated throughput for all workers to store data to the pool of completed tasks. Ideally, the t_{idle} can be eliminated altogether with properly configured pool size and maintaining:

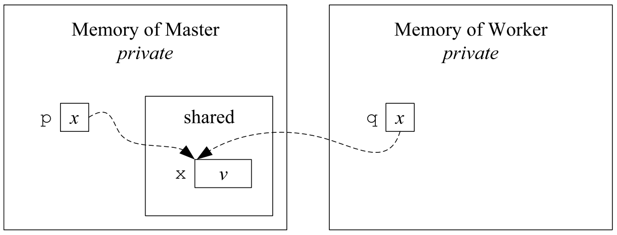


Fig. 6. Illustration of master share.

$$\begin{cases} \omega_{(\text{worker}\leftarrow\text{pending})\text{aggregated}} > \omega_{\text{master}\to\text{pending}} \\ \text{and} \\ \omega_{(\text{worker}\to\text{completed})\text{aggregated}} > \omega_{\text{master}\leftarrow\text{completed}}. \end{cases} \quad (7)$$

The throughput ω values in (7) will be affected by number of processes in the system and how the data communications are carried out between processes.

3.1. Shared memory schemes

Two shared memory schemes: *master share* and *worker share* are introduced as follows. Communication between two processes using shared memory mechanisms can be achieved by allowing a process to allocate a block of shared memory and then exchange the address of shared memory between processes, that means all participating processes in the data communication are able to directly load value from or store value to the specified shared memory addresses.

3.1.1. Master share

In the master share scheme, master process and worker process exchange data by using shared memory space allocated by the master process. Fig. 6 depicts the initialization of master share, where master process allocates a region of shared memory to accommodate shared objects. Master process then notifies worker process the location of shared memory, such that both master and worker can access the shared memory region processes.

By utilizing the master share scheme, because the task pool and result pool are memory buffers created and shared by the master process, the memory buffer will be *homed* to the *tile* running the master process. It means the overheads for memory load and store will be minimal to the master process. Thus the memory bandwidth $\omega_{\text{master}\to\text{pending}}$ and $\omega_{\text{master}\leftarrow\text{completed}}$ in (7) will be relatively higher. However, from the perspective of worker process, because the shared memory is not homed to the tile running the worker process, the memory overheads become higher. Also the $\omega_{(\text{worker}\leftarrow\text{pending})\text{aggregated}}$ and $\omega_{(\text{worker}\to\text{completed})\text{aggregated}}$ in (7) will be confined by the memory network bandwidth to the master tile, which causes performance bottleneck here when the number of worker processes increases.

3.1.2. Worker share

In the worker share scheme, the worker process allocates a region of shared memory buffer for data sharing with master process as depicted in Fig. 7. In Fig. 7, the worker process allocates a region of shared memory to accommodate shared objects. Similarly to above discussion, the worker process then notifies master process the location of shared memory, so both the worker

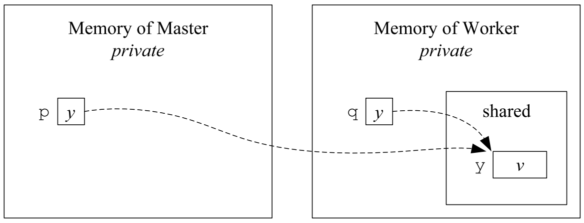


Fig. 7. Illustration of worker share.

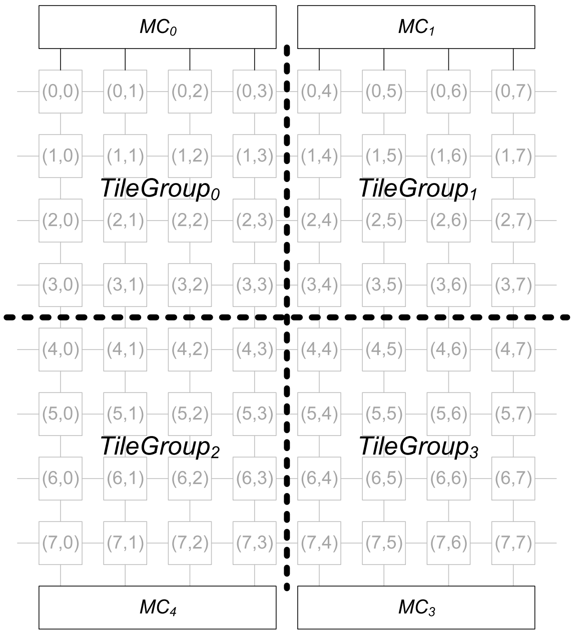


Fig. 8. Memory controller and tile grouping on TILE64.

and master process will have read and write access to this shared memory region.

By utilizing the worker share scheme, the aggregated bandwidth $\omega_{(\text{worker}\leftarrow\text{pending})\text{aggregated}}$ and $\omega_{(\text{worker}\rightarrow\text{completed})\text{aggregated}}$ will be increased since accesses to memory address locations will be distributed across all worker tiles, generating a more distributed memory network traffic to improve overall MapReduce performance.

4. Theoretical analysis

We derive performance characteristics of the master share and worker share in this section.

4.1. Shared memory access on TILE64

A TILE64 processor contains 64 cores, each referred as a *tile*. Tiles are identified by their coordinate in the 8 by 8 mesh. To denote tiles, we use the notation tile(m, n), where $0 \le m < 8$ and $0 \le n < 8$. Fig. 8 shows a TILE64 processor, it has four memory controllers located at four corners of the chip. The 64 tiles in a TILE64 processor are divided into four groups. Each group contains 16 tiles and every tile in a group shares the same memory controller. Private memory and shared memory within a process will be allocated first to the group memory controllers. For example, if a process running on tile(2, 2) allocates a block of shared memory, this block of shared memory will be homed to tile(2, 2) and allocated to MC_1.

Assuming memory is not cached, so every load and store operation will go directly to the associated memory controllers. For example, assuming the case that tile(0, 0) allocates a block of shared

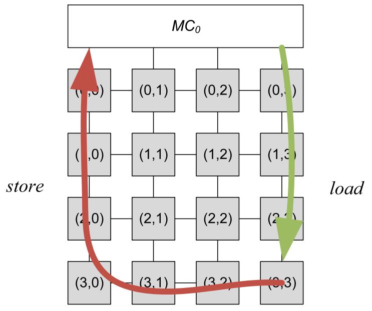


Fig. 9. Routing path for load and store operations from tile(3, 3) to memory shared by tile(0, 0).

memory and shares this memory block with tile(3, 3). Load and store accesses to memory addresses will be translated into network traffics in the on chip mesh network. On mesh-based networks, dimension-order routing such as XY routing is commonly used. In XY routing, messages sent from a source tile(m, n) to destination tile(p, q) will first be routed along the X dimension to tile(m, q), then routed along the Y dimension to tile(p, q). This routing algorithm guarantees that not only shortest paths from any source to destination are selected but also deadlock-free.

Fig. 9 shows routing path for load and store operations originated from tile(3, 3) to shared memory block created by tile(0, 0). For load operations, because the actual data resides in MC_0, network messages of the data will be routed from MC_0 to tile(3, 3) through a shortest path, which is

$$MC_0 \rightarrow \text{tile}(0,3) \rightarrow \text{tile}(1,3) \rightarrow \text{tile}(2,3) \rightarrow \text{tile}(3,3).$$

Network messages in this load operation travels through 5 switches and 4 intermediate wires. For store operations, because the shared memory is allocated and managed by tile(0, 0) and store operations involves updating values, network messages generated by store operations performed by tile(3, 3) will be routed through shortest path from tile(3, 3) towards tile(0, 0) to MC_0, which is

$$\text{tile}(3,3) \rightarrow \text{tile}(3,2) \rightarrow \text{tile}(3,1) \rightarrow \text{tile}(3,0)$$
$$\rightarrow \text{tile}(2,0) \rightarrow \text{tile}(1,0) \rightarrow \text{tile}(0,0) \rightarrow MC_0.$$

4.2. Sequential MapReduce performance

The time required for a single tile to perform MapReduce operation over a given input dataset can be calculated as:

$$t_{\text{mapreduce}} = t_{\text{map}} + t_{\text{reduce}} \tag{8}$$

where t_{map} and t_{reduce} are time for completing all map and reduce tasks, respectively. Furthermore, the time for map tasks is

$$t_{\text{map}} = t_{\text{read\_input}} + t_{\text{comp\_map}} + t_{\text{write\_itrm}} \tag{9}$$

where $t_{\text{read\_input}}$ represents memory access time required to load all data from input, $t_{\text{comp\_map}}$ is the computation time spent on the map function, $t_{\text{write\_itrm}}$ is the time spent on storing intermediate data into the intermediate buffer. The time for reduce tasks is

$$t_{\text{reduce}} = t_{\text{read\_itrm}} + t_{\text{comp\_reduce}} + t_{\text{write\_output}} \tag{10}$$

where $t_{\text{read\_itrm}}$ is memory access time required to load all intermediate data from intermediate buffer, $t_{\text{comp\_map}}$ is the computation time spent on the reduce function, $t_{\text{write\_itrm}}$ is the time spent on storing final results into the output buffer.

The throughput for map tasks can be defined as:

$$\varphi_{\text{map}} = \frac{S_{\text{input}}}{t_{\text{map}}} = \frac{S_{\text{input}}}{t_{\text{read\_input}} + t_{\text{comp\_map}} + t_{\text{write\_itrm}}}, \tag{11}$$

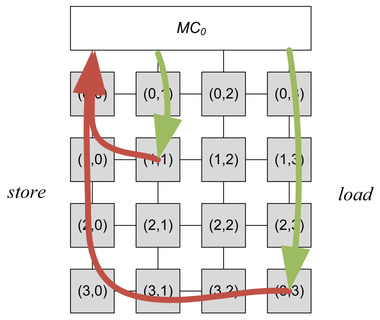


Fig. 10. Message routing for memory accesses from different tiles.

and the throughput for reduce tasks can be defined as:

$$\varphi_{\text{reduce}} = \frac{S_{\text{itrm}}}{t_{\text{reduce}}} = \frac{S_{\text{itrm}}}{t_{\text{read\_itrm}} + t_{\text{comp\_reduce}} + t_{\text{write\_output}}}. \tag{12}$$

If there is only a single tile running MapReduce with all the other tiles on a TILE64 idling, the memory access times, $t_{\text{read\_input}}$, $t_{\text{write\_itrm}}$, $t_{\text{read\_itrm}}$ and $t_{\text{write\_output}}$ in (9) and (10) will not be practically affected by the tile which the MapReduce is executed on. This is because the tile-to-tile network latency is designed to be very low. But when there are a lot of tiles using the network, contentions might occur when the on-chip network becomes very busy, resulting in increased latency for memory access.

4.3. Parallel MapReduce performance

With the increased number of worker processes participating in parallel MapReduce operation, the memory access times, $t_{\text{read\_input}}$, $t_{\text{write\_itrm}}$, $t_{\text{read\_itrm}}$ and $t_{\text{write\_output}}$ in (9) and (10) for worker processes varies with the physical locations of tiles. A worker process running on a tile further from a memory controller will have higher memory access latency under high networker traffic due to network contention.

Fig. 10 shows an example of message routing on the TILE64 for two tiles, tile(1, 1) and tile(3, 3), which share memory buffer allocated by tile(0, 0). As discussed in Section 4.1, store operations issued by tile(1, 1) will be routed through the path $MC_0 \rightarrow$ tile(0, 3) $\rightarrow$ tile(1, 3) $\rightarrow$ tile(2, 3) $\rightarrow$ tile(3, 3), which overlaps with the path for store operations issued by tile(3, 3) on the network link between tile(0, 0) and tile(1, 0).

Assume that the switch in a tile routes messages from each port with equal priority, when the link of an output port is fully utilized, messages received from all other input ports will share the output bandwidth equally. For example, Fig. 11 shows the scenario where 4 tiles, tile(m, n), tile(m, n−1), tile(m, n+1) and tile(m+1, n), are all sending messages to tile(m −1, n) through the on-chip switch of tile(m, n). Under such situation, if the maximum bandwidth for the link from switch to tile(m−1, n) is λ, then the average message rate from each source would be λ/4.

4.3.1. Master share

If master process is running on tile(0, 0), memory bandwidth for store operations of worker process on tile(m, n) will be

$$\frac{\lambda}{3^{m+1} \times 2^{n}}, \tag{13}$$

and the memory bandwidth for load operation is

$$\begin{cases} \dfrac{\lambda}{2^{m+1}} & \text{if } m \le 2 \\ \dfrac{\lambda}{3^{m+1} \times 2^{n-3}} & \text{if } m \ge 3, \end{cases} \tag{14}$$

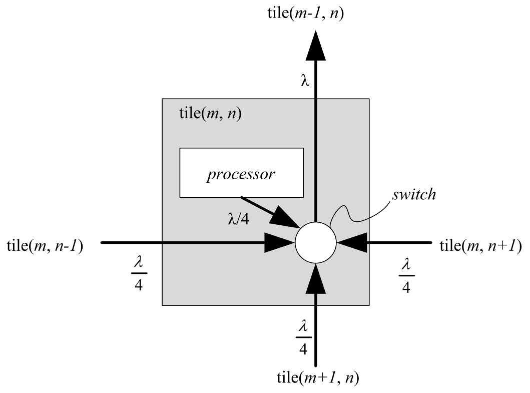


Fig. 11. Network contention and bandwidth sharing within a tile.

and the aggregated map throughput for w workers is

$$\sum_{i=0}^{w} \varphi_{\text{map}}(i). \tag{15}$$

The variable w and i in the second part of (15) represent the number of worker processes and worker process id, respectively. Thus we have

$$\begin{aligned} \varphi_{\text{map}}(i) &= \varphi_{\text{map}} \times \tau \\ \frac{S_{\text{input}}}{t_{\text{map}}(i)} &= \frac{S_{\text{input}}}{t_{\text{map}}} \times \tau \\ t_{\text{map}}(i) &= \frac{t_{\text{map}}}{\tau} \\ \tau &= \frac{t_{\text{map}}}{t_{\text{map}}(i)} \\ &= \frac{t_{\text{read\_input}} + t_{\text{comp\_map}} + t_{\text{write\_itrm}}}{p \times t_{\text{read\_input}} + t_{\text{comp\_map}} + q \times t_{\text{write\_itrm}}} \end{aligned} \tag{16}$$

where

$$\begin{aligned} p &= \begin{cases} 2^{\left\lfloor \frac{i}{8} \right\rfloor + 1} & \text{if } i\%8 \le 2 \\ 3^{\left\lfloor \frac{i}{8} \right\rfloor + 1} \times 2^{(i\%8)-3} & \text{if } i\%8 \ge 3, \end{cases} \\ q &= 3^{\left\lfloor \frac{i}{8} \right\rfloor + 1} \times 2^{i\%8}. \end{aligned} \tag{17}$$

Similarly, aggregated reduce throughput for w worker is:

$$\sum_{i=0}^{w} \varphi_{\text{reduce}}(i) \tag{18}$$

and

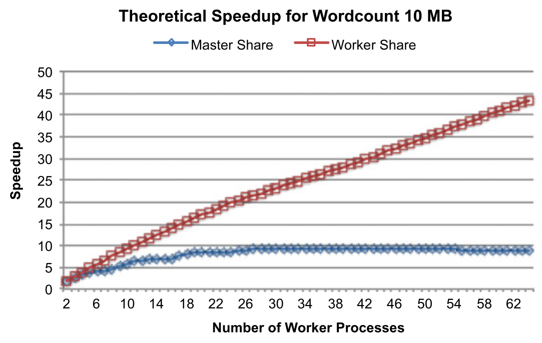
$$\varphi_{\text{reduce}}(i) = \varphi_{\text{reduce}} \times \tau, \tag{19}$$

$$\begin{aligned} \tau &= \frac{t_{\text{reduce}}}{t_{\text{reduce}}(i)} \\ &= \frac{t_{\text{read\_itrm}} + t_{\text{comp\_map}} + t_{\text{write\_output}}}{\frac{t_{\text{read\_itrm}}}{p} + t_{\text{comp\_map}} + \frac{t_{\text{write\_output}}}{q}} \end{aligned} \tag{20}$$

where p and q are derived from (17).

4.3.2. Worker share

Although worker share is harder than master share to implement, if worker share is properly implemented, every worker allocates blocks of shared memory buffers for storage of input data, intermediate data and output data. In such way, a worker will use



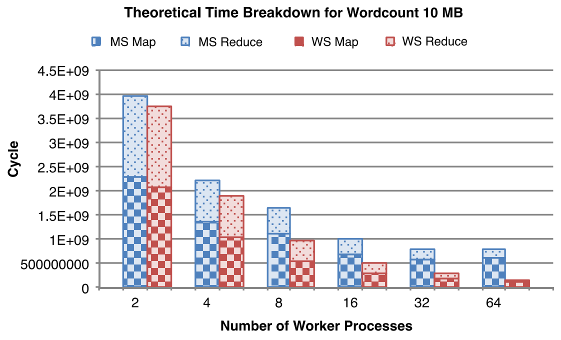


Fig. 12. Theoretical performance of Word Count for smaller problem size.

memory spaces allocated by itself, so memory bandwidth for both store and load operations of worker process on tile(m, n) will be

$$\begin{cases} \dfrac{\lambda}{2^{m+1}} & \text{if } 0 \le m \le 3 \\ \dfrac{\lambda}{2^{8-m}} & \text{if } 4 \le m \le 7. \end{cases} \tag{21}$$

Thus to calculate $\varphi_{\text{map}}(i)$ in (16) and $\varphi_{\text{reduce}}(i)$ in (19), the p and q values will be

$$p = q = \begin{cases} 2^{\lfloor \frac{i}{8} \rfloor + 1} & \text{if } 0 \le \left\lfloor \dfrac{i}{8} \right\rfloor \le 3 \\ 2^{8 - \lfloor \frac{i}{8} \rfloor} & \text{if } 4 \le \left\lfloor \dfrac{i}{8} \right\rfloor \le 7. \end{cases} \tag{22}$$

4.4. Theoretical performance

To derive theoretical performance of the master share and worker share implementations we first cross-compile the Phoenix [17] MapReduce implementation onto TILE64 platform. Then we profile benchmarks that are included with Phoenix using single tile to obtain timing and data size information for Eqs. (11) and (12). Thus we will be able to calculate $\sum_{i=0}^{w} \varphi_{\text{map}}(i)$ and $\sum_{i=0}^{w} \varphi_{\text{reduce}}(i)$ by going through Eqs. (13)–(22).

4.4.1. Benchmark applications

There are 8 benchmarks included with the Phoenix MapReduce library. These benchmarks represent key computations from various application domains. Word Count, Reverse Index and String Match are for enterprise computing. Matrix Multiply is for scientific computing. KMeans, PCA and Linear Regression are for artificial intelligence. Histogram is for image processing. Following are brief introductions to each benchmark application.

Word Count: The input of Word Count is a text file. It determines frequency of words in the input file. In the Map stage, workers process different sections of the input files and return intermediate data that consist of a word (key) and a value of 1 if the word is found. In the Reduce stage, workers add up the values for each word (key) to obtain occurrence frequency for each word in the input file. A 10 MB text file is used as input of smaller problem size and a 100 MB text file is used as input of larger problem size.

Histogram: The input of Histogram is a bitmap image file. It analyzes the input image to compute the frequency of occurrence of a value in the 0–255 range for the RGB components of all pixels. Similar to that of Word Count, in the Map stage, workers process different portions of the image to parse the image and insert the frequency of components occurrences into the intermediate data buffer array. In the Reduce stage, workers sum up these occurrence numbers across all portions. A 100 MB bitmap image file is used as input of smaller size and a 400 MB bitmap image file is used as input of larger problem size.

Reverse Index: The input is a set of HTML files. This application extracts all hyperlinks in the files and generates an index from each unique hyperlinks to its associated file names. In the Map stage, workers parse disjoint subsets of the input HTML files to find hyperlinks. If a hyperlink is found, the worker outputs an intermediate pair with the link as the key and the file name as the value. In the Reduce stage, all files referencing the same link are combined into a single linked-list. The smaller problem set contains a HTML file set of around 250 KB, the larger problem set contains a HTML file set of around 1 GB.

String Match: It processes two files: “encrypt” and “keys”. The “encrypt” file contains a set of encrypted words and the “keys” file contains a list of plain text words. This application encrypts all words in the “keys” file in order to find which plain text words are used to generate the “keys” file. In the Map stage, workers process different portions of the “keys” file and return the plain text word as key and a flag indicating whether the plain text word is a match as value. There is no actual computation task in the Reduce stage so the Reduce task is just an identity function. The size of smaller and larger input “keys” files are 50 and 500 MB, respectively.

PCA: This application performs a portion of the Principal Component Analysis algorithm in order to find the mean vector and the covariance matrix of a set of data points. The data is presented in a matrix as a collection of column vectors. The algorithm uses two MapReduce iterations, first computes the mean for a set of rows and second computes a few elements in the required covariance matrix. The Reduce task is the identity in both iterations. The input matrix size is 100×100 for smaller problem set and 1000×1000 for larger problem set.

KMeans: This application utilizes KMeans iterative clustering algorithm to group a set of input data points into clusters. The MapReduce function is executed iteratively until the algorithm converges. Workers in the Map stage process subsets of the data points to find the distance between each point and each mean to assign the point to the closest cluster. In the Reduce stage, workers gather all points with the same cluster-id and calculate their mean vector. The input size is 100K and 500K data points for smaller and larger problem sets.

Linear Regression: This application computes the line that best-fits a given set of coordinates in an input file. In the Map stage, workers process different portions of the input file to compute summary statistics. In the Reduce stage, the statistics are computed across the entire dataset to finally determine the best-fit line. The smaller problem set is a 50 MB file and larger problem set is a 500 MB file containing coordinates.

Matrix Multiply: In the Map stage, workers compute subsets of rows of the output matrix and returns the (x, y) location of each element as the key and the result of the computation as the value. The Reduce task is just the identity function. The smaller problem set is two 300×300 matrices and larger problem set is two 600×600 matrices.

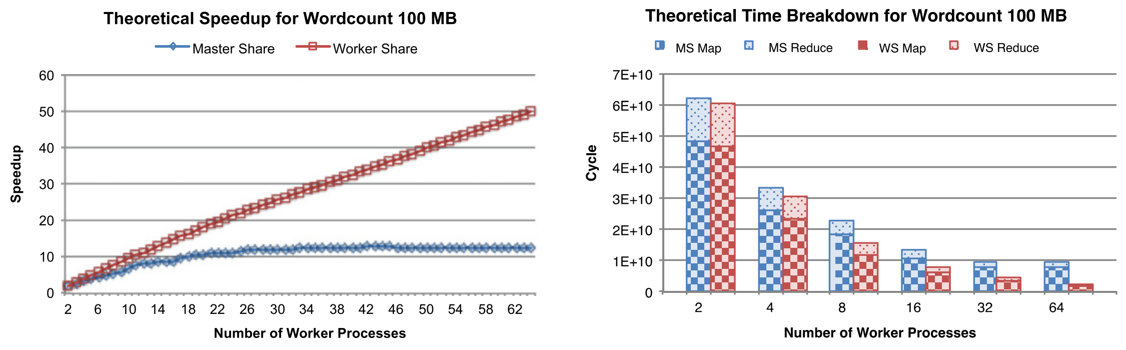


Fig. 13. Theoretical performance of Word Count for larger problem size.

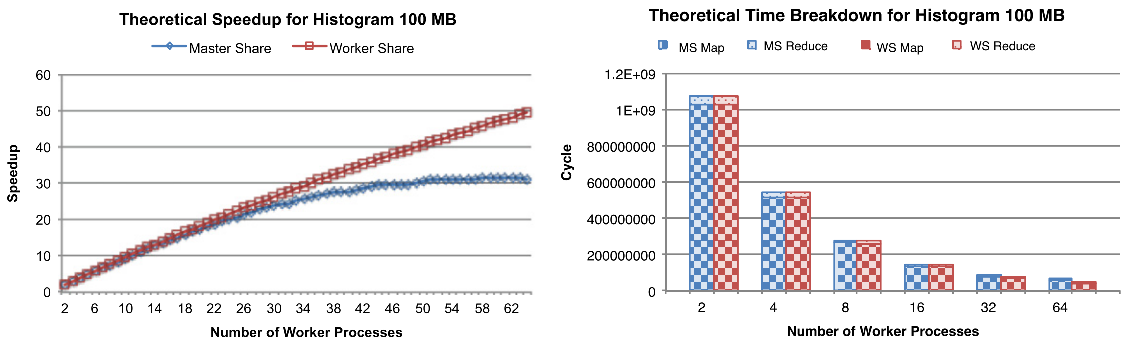


Fig. 14. Theoretical performance of Histogram for smaller problem size.

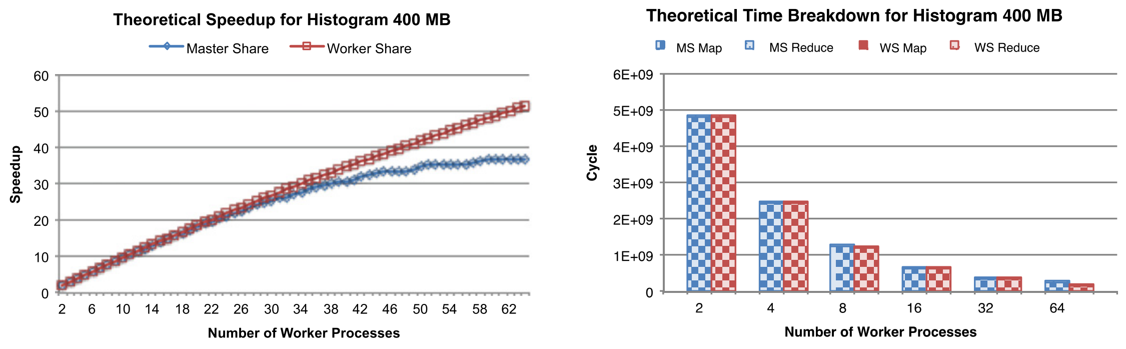


Fig. 15. Theoretical performance of Histogram for larger problem size.

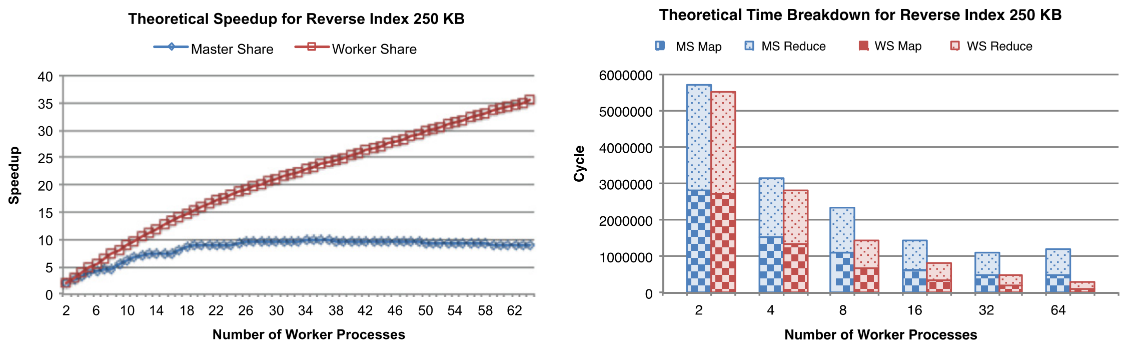


Fig. 16. Theoretical performance of Reverse Index for smaller problem size.

4.4.2. Analysis results

Following are the theoretical Master–worker MapReduce on TILE64 for different benchmarks with different problem size.

The theoretical performance for the 8 MapReduce benchmarks, Word Count, Histogram, Reverse Index, String Match, PCA, KMeans, Linear Regression, and Matrix Multiply from Phoenix are shown in Figs. 12–27. From the theoretical results, we can see that for all cases, the worker share scheme is superior to the master share scheme in terms of both scalability and performance. Different benchmarks have different characteristics as shown in the figures.

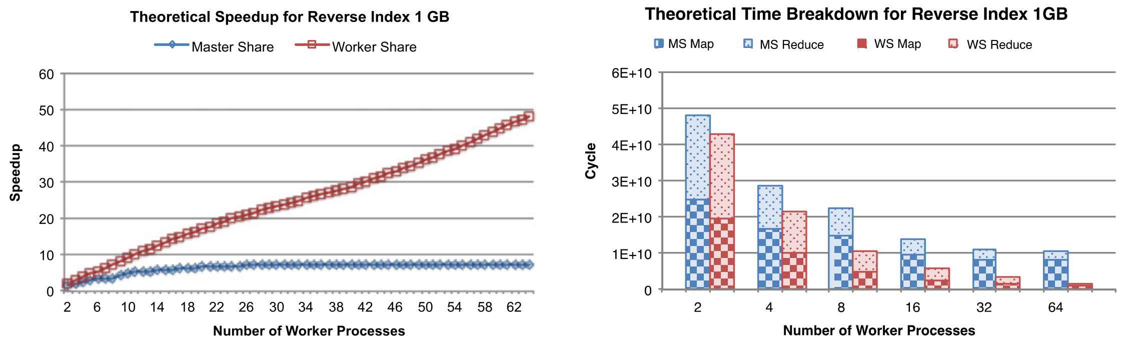


Fig. 17. Theoretical performance of Reverse Index for larger problem size.

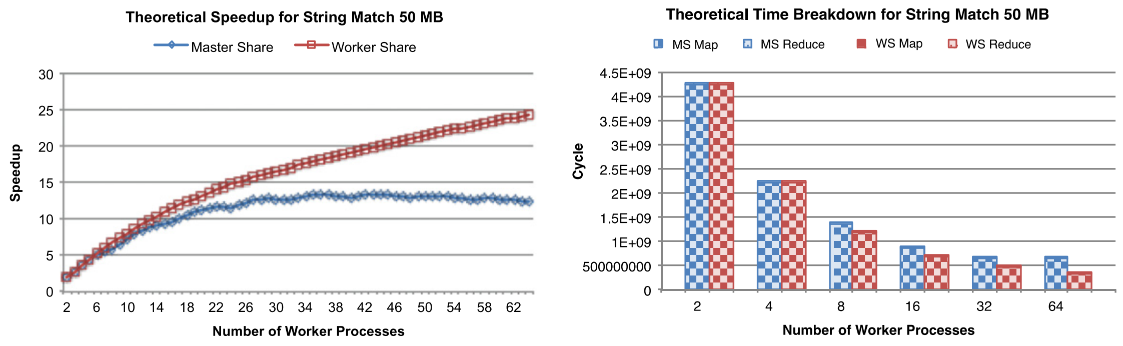


Fig. 18. Theoretical performance of String Match for smaller problem size.

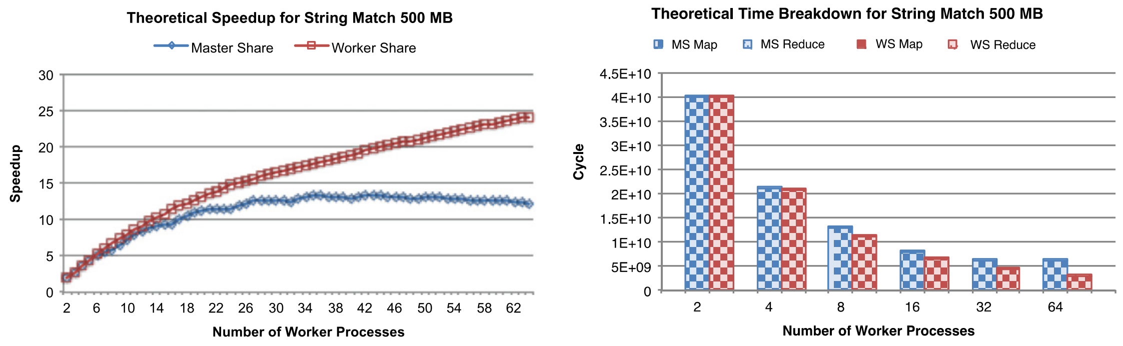


Fig. 19. Theoretical performance of String Match for larger problem size.

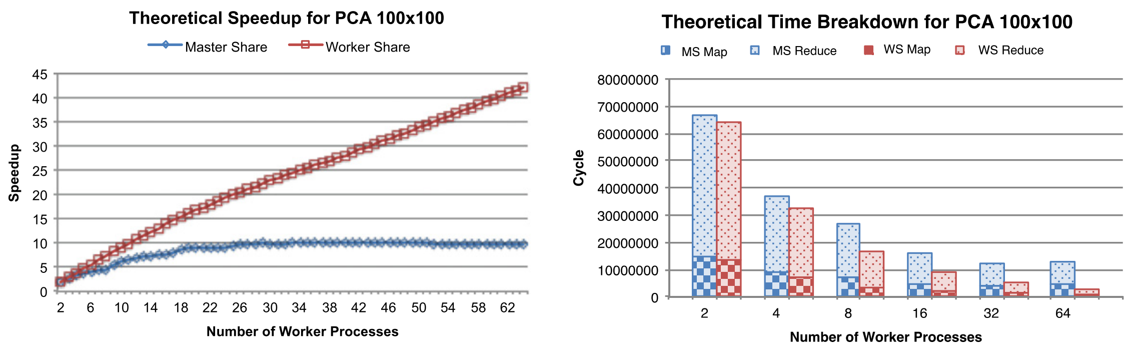


Fig. 20. Theoretical performance of PCA for smaller problem size.

Most benchmarks spend the majority of total execution time in the Map stage. It can also be observed that change of problem size might change time proportion of Map to Reduce. With the increasing number of worker processes, both Map and Reduce time can be shortened, this also varies by benchmark and problem size.

In Fig. 28, a theoretical maximum speedup for 64 workers is shown. From Fig. 28, we can see that for Word Count, Reverse Index, PCA and Matrix Multiply, the worker share scheme improves speedup for a large amount. This is due to higher aggregated memory bandwidth are demanded by these applications. On the other

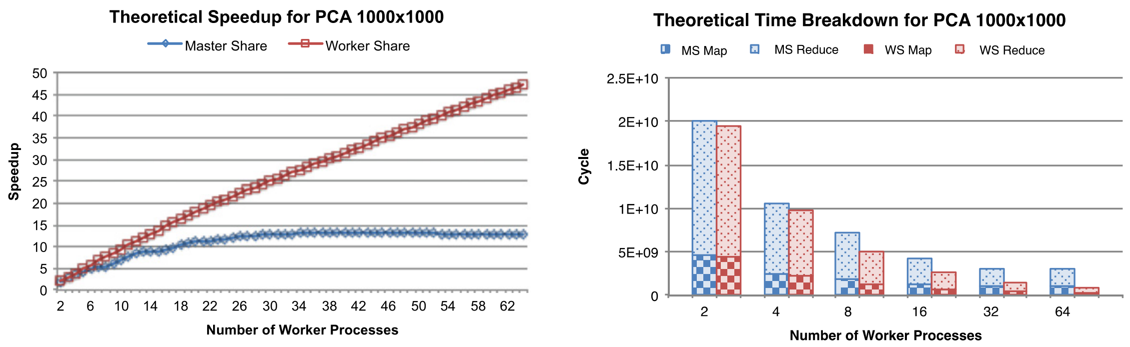


Fig. 21. Theoretical performance of PCA for larger problem size.

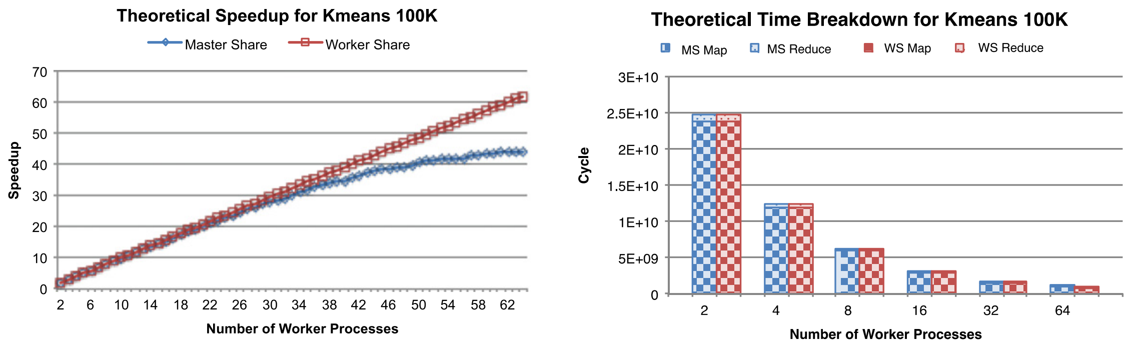


Fig. 22. Theoretical performance of KMeans for smaller problem size.

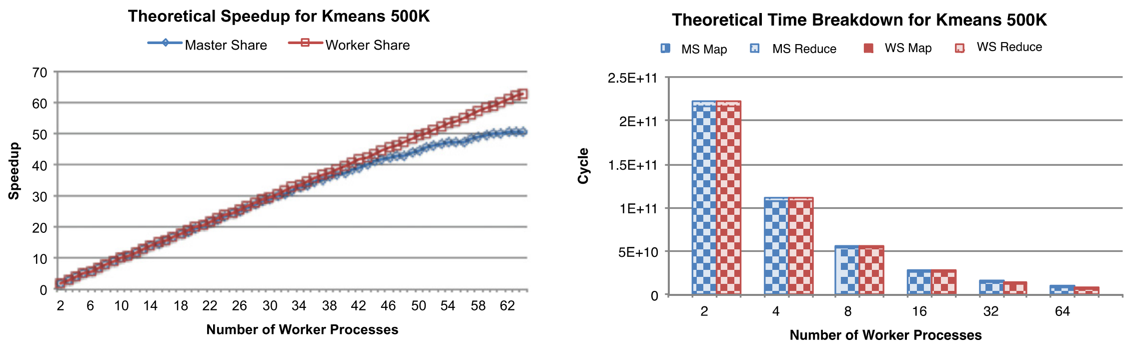


Fig. 23. Theoretical performance of KMeans for larger problem size.

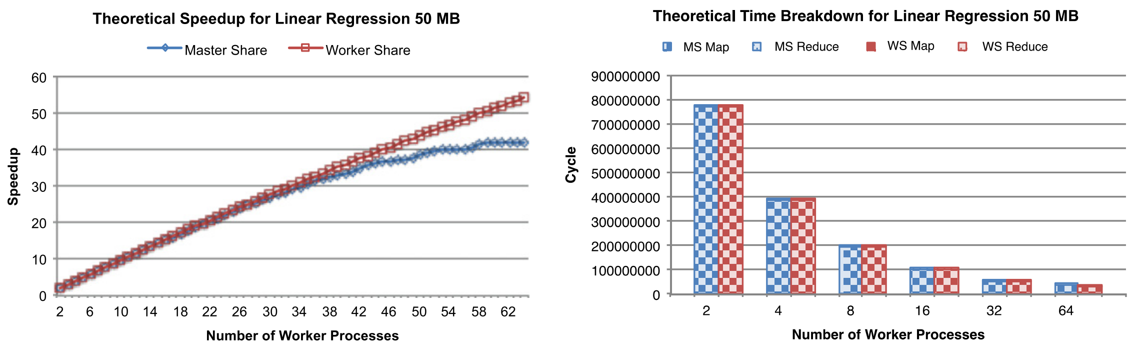
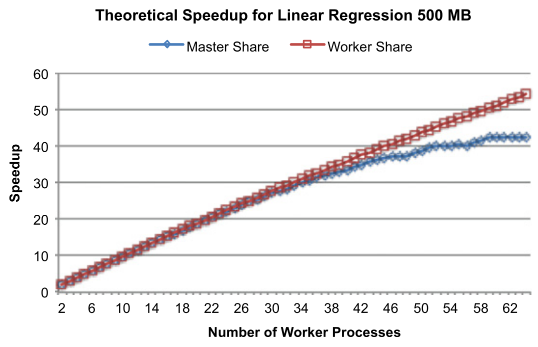


Fig. 24. Theoretical performance of Linear Regression for smaller problem size.

hand, the performance improvement of worker share over master share for Histogram, Kmeans and Linear Regression is relatively small. This is because these applications spend more time on computation than I/O, thus memory contention problem are less likely to happen in these applications.

5. Conclusion and future work

New generations of multi-core and many-core processors bring higher performance within same or lower power envelope. This advantage comes tied with the price of complication of application



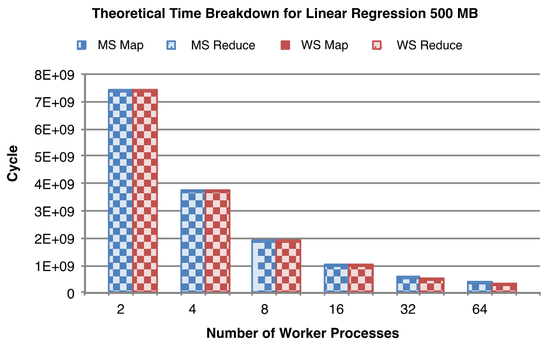
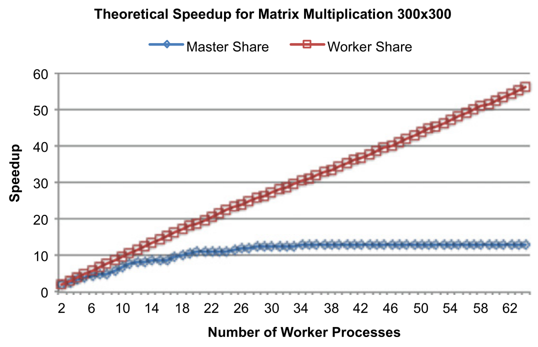


Fig. 25. Theoretical performance of Linear Regression for larger problem size.



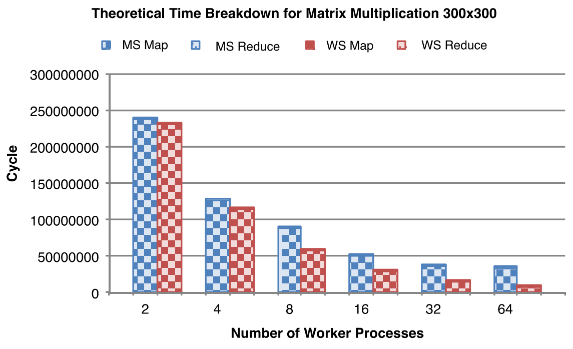
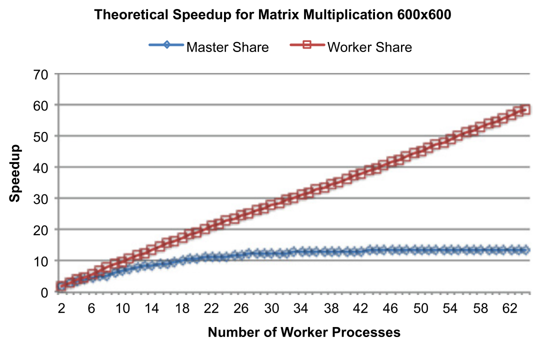


Fig. 26. Theoretical performance of Matrix Multiply for smaller problem size.



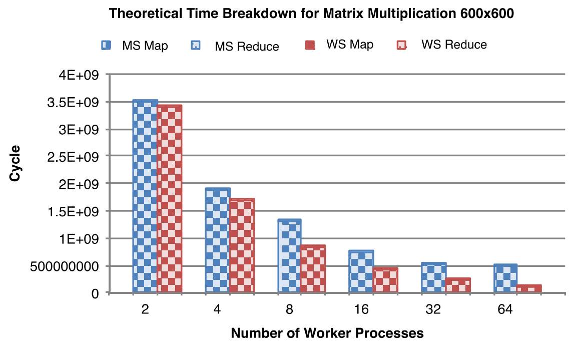


Fig. 27. Theoretical performance of Matrix Multiply for larger problem size.

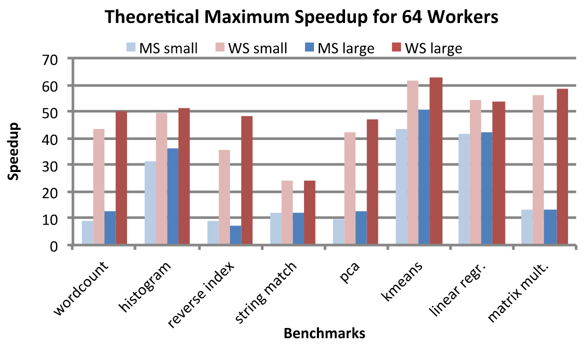


Fig. 28. Theoretical maximum speedup for 64 workers.

design and programming. Therefore, this study explores the shared memory programming schemes for master–worker MapReduce processing on TILE64 many-core platform. We model and compare two shared memory implementation schemes, *master share* and *worker share*. Analysis shows that the worker share scheme is superior to the master share scheme.

As further plans to the development of this research, we plan to implement a MapReduce library on the TILE64 platform by incorporating both master share and worker share schemes and run MapReduce benchmarks to verify that the experimental result does match theoretical analysis. Also we would like to further explore this topic by applying master share and worker share onto more complex paradigms such as hierarchical master–worker structures.

References

[1] S. Borkar, Thousand core chips: a technology perspective, in: Proceedings of the 44th Design Automation Conf., DAC 07, 2007, pp. 746–749. http://dx.doi.org/10.1145/1278480.1278667.

[2] J. Parkhurst, J. Darringer, B. Grundmann, From single core to multi-core: preparing for a new exponential, in: Proceedings of the IEEE/ACM Int. Conf. Computer-Aided Design, ICCAD 06, 2006, pp. 67–72. http://dx.doi.org/10.1145/1233501.1233516.

[3] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, B. Evans, Trends in multicore DSP platforms, IEEE Signal Process. Mag. 26 (6) (2009) 38–49. http://dx.doi.org/10.1109/MSP.2009.934113.

[4] H. Sutter, The free lunch is over: a fundamental turn toward concurrency in software, Dr Dobb's J. 30 (3) (2005) 202–210.
[5] G. Chen, F. Li, S.W. Son, M. Kandemir, Application mapping for chip multiprocessors, in: Proceedings of the 45th Design Automation Conf., DAC 08, 2008, pp. 620–625. http://dx.doi.org/10.1145/1391469.1391628.
[6] G. Tan, N. Sun, G.R. Gao, A parallel dynamic programming algorithm on a multi-core architecture, in: Proceedings of the 19th ACM Symp. Parallel Algorithms and Architectures, SPAA 07, 2007, pp. 135–144. http://dx.doi.org/10.1145/1248377.1248399.
[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, B. Liewei, J. Brown, M. Mattina, M. Chyi-Chang, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, TILE64 processor: a 64-core SoC with mesh interconnect, in: Proceedings of the IEEE Intl. Solid-State Circuits Conf., ISSCC 08, 2008, pp. 88–598. http://dx.doi.org/10.1109/ISSCC.2008.4523070.
[8] J. Berthold, M. Dieterle, R. Loogen, S. Priebe, Hierarchical master–worker skeletons, in: Proceedings of the 10th Int. Conf. Practical Aspects of Declarative Languages, PADL 08, in: LNCS, 2008, pp. 248–264.
[9] A. Benoit, L. Marchal, J.F. Pineau, Y. Robert, F. Vivien, Scheduling concurrent bag-of-tasks applications on heterogeneous platforms, IEEE Trans. Comput. 59 (2) (2010) 202–217. http://dx.doi.org/10.1109/TC.2009.117.
[10] J.D. Jackson, P.J. Hatcher, Efficient parallel execution of sequence similarity analysis via dynamic load balancing, in: Proceedings of the ISCA 3rd Int. Conf. Bioinformatics and Computational Biology, BICoB 11, 2011, pp. 219–224.
[11] J.P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, An enabling framework for master–worker applications on the computational grid, in: Proceedings of the 9th Int. Symp. High-Performance Distributed Computing, HDPC 00, 2000, pp. 43–50.
[12] R.M. Fujimoto, A.W. Malik, A. Park, Parallel and distributed simulation in the cloud, SCS M&S Magazine 1 (3) (2010) 1–10.
[13] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, P.J. Maechling, Enabling large-scale scientific workflows on petascale resources using MPI master/worker, in: Proceedings of the 1st Conf. Extreme Science and Engineering Discovery Environment, XSEDE 12, 2012, pp. 1–8. http://dx.doi.org/10.1145/2335755.2335846.
[14] F. Blagojevic, D.S. Nikolopoulos, A. Stamatakis, C.D. Antonopoulos, Dynamic multigrain parallelization on the cell broadband engine, in: Proceedings of the 12th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, 2007, pp. 90–100. http://dx.doi.org/10.1145/1229428.1229445.
[15] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. http://dx.doi.org/10.1145/1327452.1327492.
[16] A. Bialecki, M. Cafarella, D. Cutting, O. O'Malley, Hadoop: a framework for running applications on large clusters built of commodity hardware. Wiki at http://wiki.apache.org/hadoop/ 11, 2005.
[17] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for multi-core and multiprocessor systems, in: Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture. HPCA 2007, 2007, pp. 13–24. http://dx.doi.org/10.1109/HPCA.2007.346181.
[18] R. Chen, H. Chen, B. Zang, Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling, in: Paper Presented at the Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, Vienna, Austria, 2010.
[19] H. Hoffmann, D. Wentzlaff, A. Agarwal, Remote store programming, in: Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, X. Martorell (Eds.), High Performance Embedded Architectures and Compilers, in: LNCS, vol. 5952, Springer-Verlag, 2010, pp. 3–17. http://dx.doi.org/10.1007/978-3-642-11515-8_3.
[20] M. Awasthi, D.W. Nellans, K. Sudan, R. Balasubramonian, A. Davis, Handling the problems and opportunities posed by multiple on-chip memory controllers, in: Proceedings of the 19th Int. Conf. Parallel Architectures and Compilation Techniques, PACT 10, 2010, pp. 319–330. http://dx.doi.org/10.1145/1854273.1854314.
[21] D. Abts, N.D.E. Jerger, J. Kim, D. Gibson, M.H. Lipasti, Achieving predictable performance through better memory controller placement in many-core CMPs, in: Proceedings of the 36th Int. Symp. Computer Architecture, ISCA 09, 2009, pp. 451–461. http://dx.doi.org/10.1145/1555754.1555810.
[22] X.-Y. Lin, C.-Y. Huang, P.-M. Yang, T.-W. Lung, S.-Y. Tseng, Y.-C. Chung, Parallelization of motion JPEG decoder on TILE64 many-core platform, in: C.-H. Hsu, V. Malyshkin (Eds.), Methods and Tools of Parallel Programming Multicomputers, in: LNCS, vol. 6083, Springer-Verlag, 2011, pp. 59–68. http://dx.doi.org/10.1007/978-3-642-14822-4_7.

Xuan-Yi Lin received his B.S. degree in Computer Science and Information Engineering from Da-Yeh University in 2001, and the M.S. degree in Information Engineering and Computer Science from Feng Chia University in 2003. He is currently a Ph.D. student in the Department of Computer Science, National Tsing Hua University, Taiwan. His research interests include cluster systems and cloud computing and many-core platforms. He is a student member of the IEEE computer society and ACM.

Yeh-Ching Chung received a B.S. degree in Information Engineering from Chung Yuan Christian University in 1983, and the M.S. and Ph.D. degrees in Computer and Information Science from Syracuse University in 1988 and 1992, respectively. He joined the Department of Information Engineering at Feng Chia University as an associate professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the chairman of the department. In 2002, he joined the Department of Computer Science at National Tsing Hua University as a full professor. His research interests include parallel and distributed processing, cluster systems, grid computing, multi-core tool chain design, and multi-core embedded systems. He is a member of the IEEE computer society and ACM.