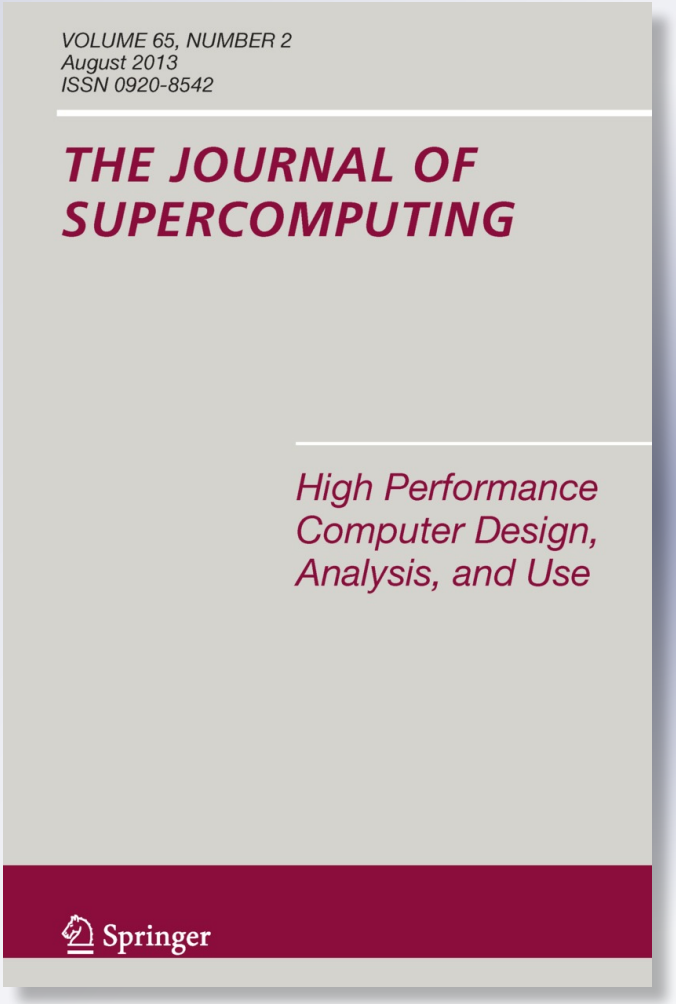
Efficient programming paradigm for video streaming processing on TILE64 platform

Xuan-Yi Lin, Kuan-Chou Lai, Kuan-Ching Li & Yeh-Ching Chung

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542
Volume 65
Number 2

J Supercomput (2013) 65:823-847
DOI 10.1007/s11227-012-0867-6



Springer

J Supercomput (2013) 65:823–847
DOI 10.1007/s11227-012-0867-6

Efficient programming paradigm for video streaming processing on TILE64 platform

Xuan-Yi Lin · Kuan-Chou Lai · Kuan-Ching Li · Yeh-Ching Chung

Published online: 24 January 2013

Abstract Advances at an unprecedented rate in computer hardware and networking technologies have made the many-core computing affordable and readily available in a matter of few years. Nonetheless, it incurs challenges to programmers to build scalable parallel software. Optimizations of parallel programs for a many-core platform are viewed as a multifaceted problem, where system and architectural factors should be taken into account. In this paper, we tackle this problem by implementing parallel programs with different available programming paradigms and evaluate application behaviors on TILE64 many-core platform. That is, we investigate a hybrid *producer-write* plus *consumer-read* shared memory programming paradigm for the implementation of master–worker video decoder and encoder in the referred many-core platform. Experimental results show that the proposed implementation has achieved competitive performance speedup, scaling well with the number of available cores and up to four times of performance improvement over other implementations on the decoding of sample 1080P video.

X.-Y. Lin · Y.-C. Chung (✉)
Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan, R.O.C.
e-mail: ychung@cs.nthu.edu.tw

X.-Y. Lin
e-mail: xylin@cs.nthu.edu.tw

K.-C. Lai
Department of Computer Science, National Taichung University of Education, Taichung 40306, Taiwan, R.O.C.
e-mail: kclai@mail.ntcu.edu.tw

K.-C. Li
Department of Computer Science and Information Engineering, Providence University, Taichung 43301, Taiwan, R.O.C.
e-mail: kuancli@pu.edu.tw

Keywords Many-core · Master–worker · Producer–consumer · Shared memory · Message passing · TILE64

1 Introduction

In recent years, the industry underwent a transition from single-core processors to multi-core or many-core processors due to thermal and 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 rewritten 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, API (application programming interface), among others.

Due to the flexibility of options available, there may exist 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. The purpose of this paper is to discuss and demonstrate how programming paradigm correlates with issues in performance and scalability of software implementations on a many-core platform.

TILE64 is a family of general purpose many-core processors designed and manufactured by Tilera [7]. Figure 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. Each individual core in a TILE64 processor is referred to as a *tile*. The iMesh is designed to be scalable to a large number of cores while maintaining low-latency communication between tiles. Tilera provides programmers with a set of proprietary APIs called *iLib* to write application programs. The iLib API 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.

Video streaming applications are the applications to process video streams. Examples of such applications are video encoders [8], video decoders [9], video transcoders [10], object detection and tracking [11], among others widely available. These applications are both computation intensive and data intensive in nature, which makes them intrinsically suitable for parallelization.

The master–worker model is very useful in parallel programming. It is often adopted when there is a need to dynamically balance workloads among available processors [12, 13]. Figure 2 shows a generic master–worker model, which consists of

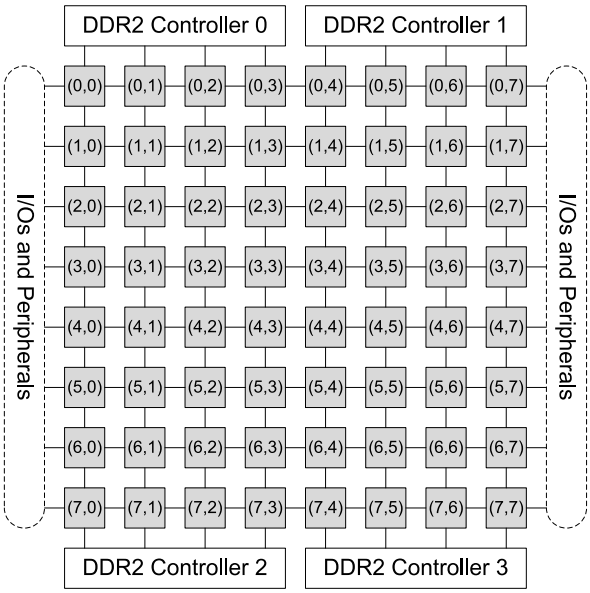


Fig. 1 TILE64 processor architecture overview

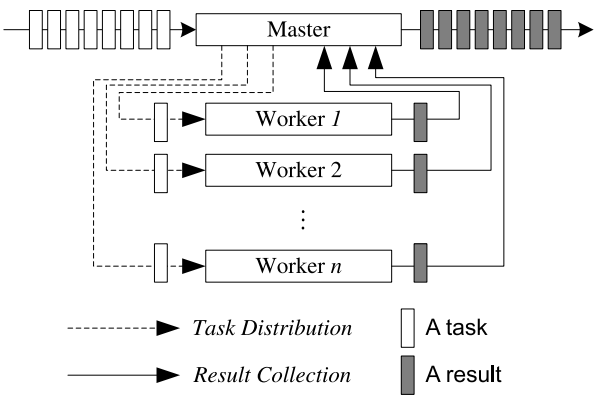


Fig. 2 A master–worker model

two main parts, *task distribution* and *result collection*. In the task distribution part, the master generates a set of tasks and distributes them next 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.

In order to implement a master–worker based video streaming application on TILE64, the efficiency of handling data communications between master and workers is crucial. Since the TILE64 provides both shared memory and message passing communication primitives, the communication of a master–worker based video streaming application can be implemented by shared memory or message passing. Therefore, efficient management of the communications between master and worker processes is required to develop a high performance application with efficiency on this platform.

This study explores two programming paradigms, *consumer-read programming* (CRP) and *producer-write programming* (PWP) for implementation of shared memory communication between master and worker processes. We have implemented master–worker video stream encoders and decoders with different communication schemes to compare their performance and scalability. Experimental results show that task distribution is best implemented with producer-write programming, while result collection is best when implemented with consumer-read programming.

The rest of this paper is organized as follows. Section 2 provides background knowledge of TILE64 and the approach of carrying out shared memory and message passing communication between two processes on the TILE64. In Sect. 3, a master–worker stream processing system is described. Section 4 introduces the *consumer-read programming*, the *producer-write programming*, and variations of shared memory implementations of a master–worker stream processing system. In Sect. 5, we implement parallel Motion JPEG decoder and encoder with different programming paradigms and compare performance of the implementations. Finally, concluding remarks and future scope of this work are given in Sect. 6.

2 Background

2.1 TILE64 processor

In this study, we apply TILE64 as an example of the many-core architecture. The TILE64 processor is a many-core processor with 64 cores featured as an array of 64 identical processor cores (each referred to as a *tile*) interconnected via 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 then be translated to underlying network traffic, which is fully transparent to programmers. As a process is executing load/store instructions, it is not necessary to have the knowledge of 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, and will directly affect program performance. The inter-process communication should generate as little network traffic as possible, so the overall network performance on this many-core platform would not be pushed down.

A previous study [14] suggests that programmers implement applications in such a way that 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 also researches discussing scalability issues on many-core processors featuring on-chip networks or multiple memory controllers [15, 16]. In our previous work [17], 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.

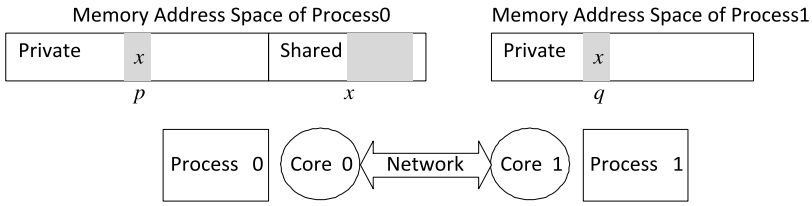


Fig. 3 Sharing of an integer on TILE64 between two processes

In this study, a *tile* runs one process at any given time. A process bound to a *tile* at the initialization period will keep running on the same *tile* to its end of life.

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 that shares information calls the *malloc_shared*() function to get an address pointing to a block of shared memory. Then, the process notifies other processes on the location of shared memory by sending them messages containing this address.

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

- There are two cores, each of which 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 the 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 back and forth is also valid.

2.3 Message passing communication on TILE64

Message passing communication uses point-to-point communication to copy sequences of data items directly from one process to another without going through the global shared memory. This type of communication requires one process to send

pieces of data to another process explicitly, which explicitly receives them. Message passing communication can simplify programs with clear producer–consumer relationships, since there is no need to synchronize objects in shared memory. A process intending to send content of a buffer can call *ilib_msg_send()*, the receiving process must also call *ilib_msg_receive()* to receive the data.

2.4 Related work

The master–worker model has been successfully used in many research areas. It is often used in large distributed computing environments such as clusters [18], grids [19], clouds [20] and even on petascale resources [21]. 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 [22]. Parallel software libraries such as Charm++ [23] and MapReduce [24] also utilize the master–worker paradigm. From the literature we can see that the master–worker model is very flexible and can be used on a wide variety of applications and platforms/systems.

There are some examples by using the master–worker model for video streaming applications. In Ref. [10], the parallelization of the H.261 video coding based on the master–worker model on the IBM SP2 multiprocessor system is discussed, and the master–worker model is utilized to implement a real-time video player that operates on a tiled-display system consisting of multiple PCs to provide a very large and high resolution display in [25]. Master–worker model implemented as a hardware-accelerated system-on-chip solution for MPEG-4 decoding is presented in [9], and the master–worker model used for live video and audio media multicasting in a wireless ad hoc network environment is presented in [26]. A distributed high performance video processing architecture based on a master–worker model in the cloud computing environment is presented in [27], and a parallel video face recognition system is built upon a group of personal computer using the master–worker model in [28].

Although there is a lot of literature that discusses the application of the master–worker model on different systems or platforms, only a few are related to the application of the master–worker model on many-core platforms. We believe that this is due to availability of such systems in academic and research communities and its wider spread. Since the number of cores in commercial processors will keep increasing in the foreseeable future, it is important and necessary to discuss the problem of mapping traditional models onto multi- and many-core platforms.

3 Master–worker stream processing

In general, video stream processing applications handle video data stream. Some examples of such applications are video encoders, decoders, and transcoders. Given a data stream to be processed by a video stream processing application, we assume that the stream can be divided into n sequential segments that can be independently processed and outputted. The input data stream can be represented as a set of sequenced data items, f_{i_1} to f_{i_n}, and the output data stream is represented as f_{o_1} to f_{o_n}. Assume

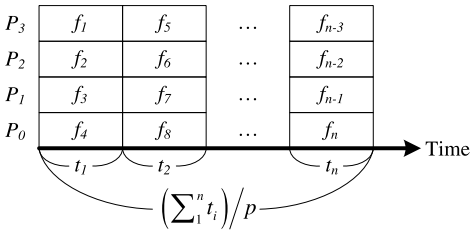


Fig. 4 Perfect task scheduling of stream processing on 4 processors

that the application is run on a processor, each segment takes time t_n to be processed from input format to output format. The time needed to process all segments in the stream is

$$\sum_{1}^{n} t_i \tag{1}$$

The ideal case of processing such data stream 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. Thus, the time needed to process all segments becomes

$$\frac{\sum_1^n t_i}{p} \tag{2}$$

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.

One way to speed up data stream processing on multiple processors is to use a master–worker scheme as underlying parallelization structure. 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.

Figure 5 illustrates a master–worker streaming system that consists of one master process and 4 worker processes. The master process reads in the input stream and divides the input stream into smaller chunks of data that can be independently processed. After initializing, all worker processes in this system keep monitoring the

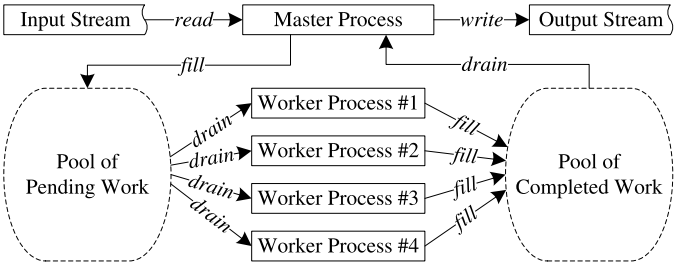


Fig. 5 A master–worker streaming system

Algorithm 1 Master-worker stream processor

```
Input    : A data stream I consists of a sequence of data items of datatype α.
Output   : A data stream O consists of a sequence of data items of datatype β.
Data Types: α, β
Buffers: pendingPool ⟵ ∅, completedPool⟵ ∅
Functions: fill(buffer, item), drain(buffer)

Process master:                                              /* one instance */
Private Data: F_α ⟵ ∅, F_β ⟵ ∅
begin
    initialize (pendingPool):
    initialize (completedPool):
    repeat
        /* task distribution                                                 */
        if F_α = ∅ then F_α ⟵ fetchNextItem(I) if F_α ≠ ∅ then
            result ⟵ fill(pendingPool, F_α)
            if result = true then F_α ⟵ ∅
        /* result collection                                                 */
        F_β ⟵ drain(completedPool)
        if F_β ≠ ∅ then
            output(O, F_β)
            F_β ⟵ ∅
    until program termination

Process worker:                                          /* multiple instances */
Private Data: F_α ⟵ ∅, F_β ⟵ ∅
begin
    initialize (pendingPool):
    initialize (completedPool):
    repeat
        F_α ⟵ drain(pendingPool)
        if F_α ≠ ∅ then
            F_β ⟵ processItem(F_α)
            fill(completedPool, F_β)
            F_α ⟵ ∅
            F_β ⟵ ∅
    until program termination
```

pool of pending tasks and check whether there are present workloads. If not empty, any available worker can fetch (drain) a task from the pool and start to process it. Once such an execution is completed, it fills the pool of completed tasks with the results of current task. Since completed tasks arrive in arbitrary order, master process keeps an output sequence counter. The counter is used to select the next completed task form the pool with correct sequence number to be output to the output stream. Algorithm 1 shows the pseudo code of a master–worker stream processor.

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

$$t_{\text{total}} = t_{\text{read}} + t_{\text{fill}} + t_{\text{drain}} + t_{\text{write}} + t_{\text{comp}} + t_{\text{sync}} + t_{\text{idle}} \tag{3}$$

Since the time spent by workers essentially overlaps with the time spent by the master, the total time takes into consideration solely the time spent by the master. A list containing detailed description of components in (3) follows:

- t_{read}: time master spent reading input data from input stream 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 stream;
- 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 stream; that is, these three timing values are not affected by system configuration variables such as number of workers, size of task pools and how interprocess communications are carried out.

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

$$t_{\text{fill}} = \frac{S_{\text{input}}}{\omega_{\text{master}\rightarrow\text{pending}}}, \tag{4}$$

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

$$t_{\text{drain}} = \frac{S_{\text{output}}}{\omega_{\text{master}\leftarrow\text{completed}}}, \tag{5}$$

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 $\omega_{\text{master}\rightarrow\text{pending}}$ 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:

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

where p is the number of shared objects in the system and q the number of participating processes intending 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, $\omega_{\text{master}\rightarrow\text{pending}}$ 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 $\omega_{(\text{worker}\rightarrow\text{completed})\text{aggregated}}$, 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 the condition:

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

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, which will be further explored in Sect. 4.

4 Programming paradigms for TILE64 platform

Two shared memory programming paradigms, the *consumer-read programming* (CRP) and the *producer-write programming* (PWP), are introduced as follows. Communication between two processes by 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, which means that all participating processes in the data communication are able to directly load value from or store value to the specified shared memory addresses.

4.1 Consumer-read programming (CRP)

When a producer process sends data to a consumer process, it writes the data into memory address shared by the producer process itself. The consumer process then reads the data from this shared address. The term *consumer-read* implies the action: "consumer reads data from producer shared memory."

Figure 6 depicts the initialization of CRP, where producer process allocates a region of shared memory to accommodate shared objects. Producer process then notifies consumer process on the location of shared memory, so that producer checks and fills the shared memory if it is not full. Consumer keeps checking the content in the shared memory and consumes it if the shared memory is not empty.

4.2 Producer-write programming (PWP)

When a producer process sends data to a consumer process, it writes the data into the memory address shared by the consumer process. The term *producer-write* implies the action: "producer writes data to consumer shared memory."

In Fig. 7, a consumer process allocates a region of shared memory to accommodate shared objects. Similarly to the above discussion, the consumer process then notifies a producer process on the location of shared memory, and the producer checks and fills the shared memory when it is empty. The consumer keeps checking the content in the shared memory and consumes it when the content is valid.

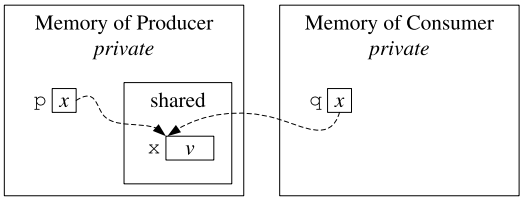


Fig. 6 Illustration of CRP

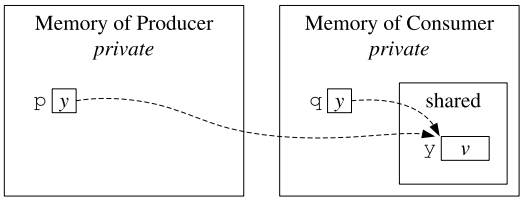


Fig. 7 Illustration of PWP

4.3 Implementation of master–worker system

There are multiple ways of using *iLib* shared memory primitives to implement a master–worker stream processing system. The major difference is on the implementation of two functions, *drain*() and *fill*(). Depending on the shared memory programming paradigm used, the two pools of tasks can reside in memory addresses shared by either master process or worker processes. The shared memory implementation algorithms are given in Algorithms 2, 3, 4 and 5.

(1) CRP + CRP: Both pools are implemented by using CRP. The pool of pending tasks resides in memory shared by master process. And all of the worker shared memory combined forms the pool of completed tasks,
(2) CRP + PWP: The pool of pending tasks is implemented by using CRP and the pool of completed tasks is implemented by using PWP. Both pools of pending tasks and completed tasks reside in memory shared by master process,
(3) PWP + CRP: The pool of pending tasks is implemented by using PWP and the pool of completed tasks is implemented by using CRP. Both pools of pending tasks and completed tasks are actually shared memory blocks distributed among all workers processes,
(4) PWP + PWP: Both pools are implemented by using PWP. And all of the worker shared memory combined forms the pool of pending tasks, and the pool of completed task resides in memory shared by master process.
(5) MP: There is no shared pool for tasks and results. In task distribution, the master sends works to workers directly by using iLib message passing API *ilib_msg_send()*. In result collection, the workers use *ilib_msg_send()* to send results to the master.

5 Experimental results

We use the master–worker structure to make parallel Motion JPEG decoders and encoders by modifying source codes from the *MJPEG Tools* [29], as in Sect. 3. Later, as discussed in Sect. 4.3, there are 5 implementation options: CRP + CRP (R + R), CRP + PWP (R + W), PWP + CRP (W + R), PWP + PWP (W + W) and MP. Therefore, we have 5 implementations for decoder and another 5 implementations for encoder. One fact to note here is that the PWP + PWP implementation can be considered as an implementation of the remote store programming (RSP) as described in [14]. Therefore, the performance of PWP + PWP can be seen as performance obtained by implementing the master–worker decoder/encoder using a programming paradigm (RSP) described in an existing literature.

Algorithm 2 CRP task distribution

```
initialize (pendingPool):
begin
    switch role of the calling process do
        case master
            addr ⟵ mallocShared(α, size)
            broadcast addr to all worker
            pendingPool ⟵ addr
            break
        case worker
            receive addr broadcasted by master
            pendingPool ⟵ addr
            break

fill(pendingPool, F):                                   /* by master */
begin
    if notFull (pendingPool) then
        lock(pIn)
        pendingPool[pIn] ⟵ F
        pIn ⟵ (pIn + 1) modulo size
        unlock(pIn)
        return true
    else
        return false

drain(pendingPool):                                     /* by workers */
begin
    if notEmpty (pendingPool) then
        lock(pOut)
        Fα ⟵ pendingPool[pOut]
        if CRP Result Collection is used then
            lock(OutputIndexQueue)
            enqueue myID to OutputIndexQueue
            unlock(OutputIndexQueue)
        pOut ⟵ (pOut + 1) modulo size
        unlock(pOut)
        return Fα
    else
        return ∅
```

We executed the implemented decoders/encoders on a TILExpress-20G card, which is a TILE64 development platform featured with a TILE64 processor running at 700 MHz and 4 GB of DDR2-800 memory. The advantage of TILE64 is that it delivers better performance per watt comparing to other multi-core platforms, for the TILE64 processor at maximum draws only 22 W power.

Each of the decoders/encoders decodes/encodes 4 video files of different resolutions. Table 1 lists the video test files, which are placed in a RAM file system. Since tiles located in the last row are reserved for system use and not available for users to run programs on the TILE64 hardware platform, the maximum number of tiles we used is 56 (8 columns by 7 rows.)

We measured both decoder and encoder performance from using 2 tiles (1 master process with 1 worker process) to 56 tiles (1 master process with 55 worker processes), to obtain a total of 2200 sets of timing data. Table 2 shows the number of performance data sets collected from among different configurations.

Algorithm 3 PWP task distribution

```
initialize (pendingPool):
begin
    case master
        for i ← 0 to numOfWorkers−1 do
            receive addr from worker_i
            pendingPool[i] ⟵ addr
        break
    case worker
        addr ⟵ mallocShared(α, 1)
        send addr to master
        pendingPool ⟵ addr
        lock(AvailableWorkers)
        append myPID to AvailableWorkers
        unlock(AvailableWorkers)
        break

fill (pendingPool, F):                                   /* by master */
begin
    if AvailableWorkers ≠ ∅ then
        lock(AvailableWorkers)
        remove x from AvailableWorkers
        if CRP Result Collection is used then
            lock(OutputIndexQueue)
            enqueue x to OutputIndexQueue
            unlock(OutputIndexQueue)
        unlock(AvailableWorkers)
        pendingPool[x] ⟵ F
        return true
    else
        return false

drain (pendingPool):                                     /* by workers */
begin
    if pendingPool ≠ ∅ then
        F_α ⟵ pendingPool
        return F_α
    else
        return ∅
```

Table 1 Motion JPEG test files used

File name	Format	Resolution	Frames
Deadline	CIF	352 × 288	1374
City	4CIF	704 × 576	600
Stockholm	720P	1280 × 720	604
Factory	1080P	1920 × 1088	1339

Figures 8 to 15 show the performance of decoders and encoders in different testing cases. These data are obtained by recording time spent on main decoding/encoding loop in the decoder/encoder. Since parallel versions contain at least one master process and one worker process, the minimum number of cores required to run these parallel decoders/encoders is 2. In the case that a parallel decoder/encoder is run-

Algorithm 4 CRP result collection

```
initialize (completedPool):
switch role of the calling process do
    case master
        for i ← 0 to numOfWorkers−1 do
            receive addr from worker_i
            completedPool[i] ⟵ addr
        break
    case worker
        addr ⟵ mallocShared(β, 1)
        send addr to master
        completedPool ⟵ addr
        break

fill (completedPool, F):                                   /* by workers */
begin
    completedPool ⟵ F
    wait until get confirm message from master

drain (completedPool):                                     /* by master */
begin
    if OutputIndexQueue ≠ ∅ then
        lock(OutputIndexQueue)
        dequeue x from OutputIndexQueue
        unlock(OutputIndexQueue)
        F_β ⟵ completedPool[x]
        send confirm message to worker_i
        return F_β
    else
        return ∅
```

Table 2 Performance data sets obtained

Tasks	Programming paradigms				
	R + R	R + W	W + R	W + W	MP
Decode CIF	55	55	55	55	55
Decode 4CIF	55	55	55	55	55
Decode 720P	55	55	55	55	55
Decode 1080P	55	55	55	55	55
Encode CIF	55	55	55	55	55
Encode 4CIF	55	55	55	55	55
Encode 720P	55	55	55	55	55
Encode 1080P	55	55	55	55	55

ning using 2 tiles, only the tile that executes the worker process is responsible for the decoding/encoding job.

For reference, performance of video encoding on similar platform discussed in existing literature shows that a speedup of 23.9 can be obtained when encoding 720P video clips using 56 cores [30].

Algorithm 5 PWP result collection

```
initialize (completedPool):
switch role of the calling process do
    case master
        addr ⟵ mallocShared(β, size)
        broadcast addr to all worker
        completedPool ⟵ addr
        break
    case worker
        receive addr broadcasted by master
        completedPool ⟵ addr
        break

fill (completedPool, F):                                   /* by workers */
begin
    if notFull (completedPool) then
        lock(cIn)
        currentPosition ⟵ cIn
        cIn ⟵ (cIn + 1) modulo size
        unlock(cIn)
        completedPool[currentPosition] ⟵ F
        return true
    else
        return false

drain (completedPool):                                     /* by master */
begin
    if notEmpty (completedPool) then
        F_β ⟵ completedPool[cOut]
        cOut ⟵ (cOut + 1) modulo size
        return F_β
    else
        return ∅
```

Table 3 Decoder performance—max frame rate

Video size	Programming paradigms				
	R + R	R + W	W + R	W + W	MP
CIF	1357.74	790.76	1617.44	1069.95	715.57
4CIF	462.63	209.72	518.88	267.21	204.34
720P	194.82	88.94	206.71	117.56	86.83
1080P	168.96	48.10	179.98	54.33	51.51

5.1 Decoder performance

Figures 8, 9, 10, 11 show the decoder performance for 4 different video frame sizes. The experimental results show that the decoder implemented with PWP + CRP outperforms all other versions as discussed in Sect. 4.3. And all of the shared memory-based implementations outperform the message-passing (MP) implementation. Table 3 shows the maximum obtained FPS from the decoders.

CIF video decoding: Fig. 8 shows the performance of CIF video decoding. The fastest one is the W + R implementation, which runs almost twice as fast as the slowest implementation, R + W. All implementations do not scale beyond 30 tiles when

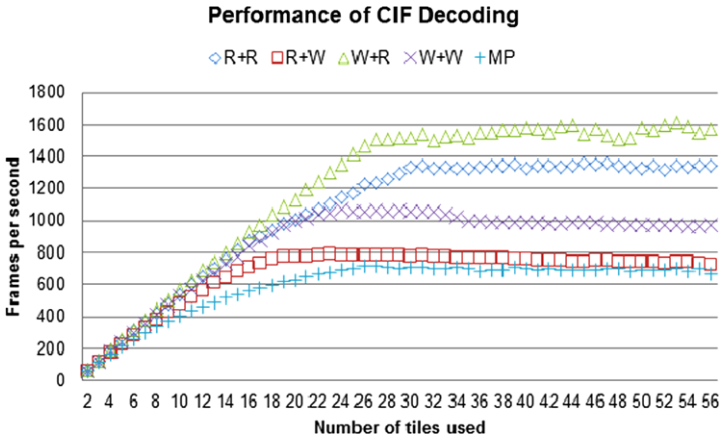


Fig. 8 Performance of CIF decoding

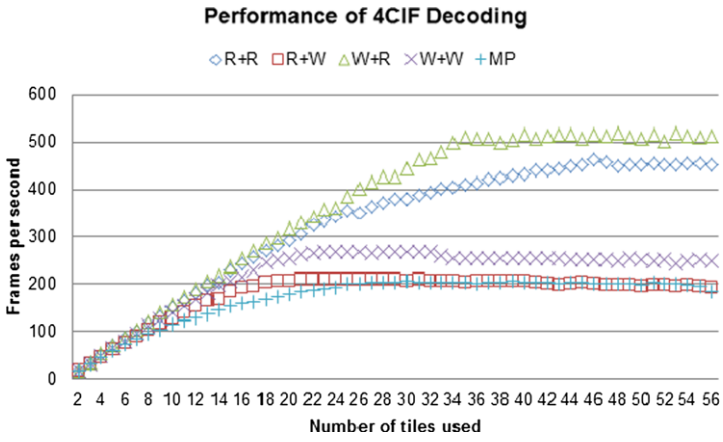


Fig. 9 Performance of 4CIF decoding

decoding CIF videos. The frame rates saturate at 31, 18, 27 and 23 tiles respectively for R + R, R + W, W + R and W + W.

QCIF video decoding: Fig. 9 shows the performance of QCIF video decoding. The fastest implementation is still W + R, which now runs 2.47 times faster than the slowest one. The frame rates saturate at 46, 18, 35 and 21 tiles for R + R, R + W, W + R and W + W, respectively.

720P video decoding: Fig. 10 shows the performance of 720P video decoding. The W + R implementation runs 2.32 times faster than the R + W implementation. The frame rates saturate at 45, 19, 39 and 25 tiles for R + R, R + W, W + R and W + W, respectively.

1080P video decoding: Fig. 11 shows the performance of 1080P video decoding. The W + R implementation runs 3.74 times faster than the R + W implementation.

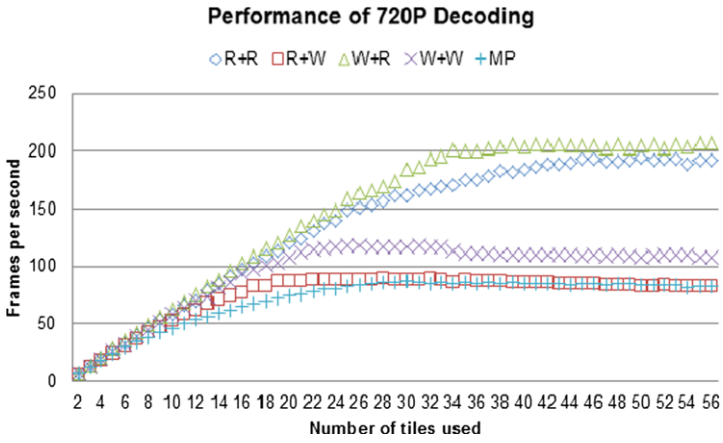


Fig. 10 Performance of 720P decoding

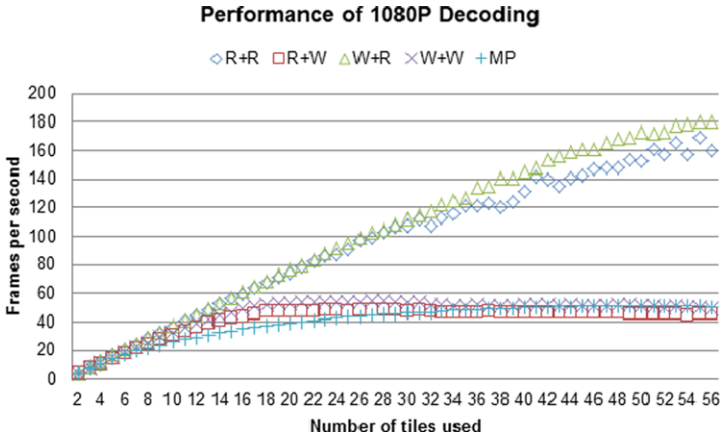


Fig. 11 Performance of 1080P decoding

The frame rates saturate at 56, 19, 55 and 19 tiles for R + R, R + W, W + R and W + W, respectively.

From the above experimental results, it can be observed that the implementations can be separated into two groups by their scalability when the given video frame size is increased. Based on CRP, the R + R and W + R decoders have similar performance behavior. On the other hand, the R + W and W + W decoders, which are based on PWP, have similar performance behavior. When decoding the 1080P video, the two implementations based on CRP result collection achieve 168.96 and 179.98 FPS decoding rates while the other two implementations based on PWP result collection only achieve 48.10 and 54.33 FPS decoding rates. As for scalability, the two implementations based on CRP result collection keep scalable with the available number of tiles but the two decoders based on PWP result collection cannot be scalable beyond 19 tiles when decoding the 1080P video.

To further investigate the reasons behind decoder scalability issues, we looked into program statistical data and have the following findings.

Smaller frame size videos: When the decoders are decoding videos of smaller frame sizes such as CIF, the bottleneck that limits the scalability of R + R and W + R implementations is the time spend by master process on I/O. That is, when the number of worker processes passes the saturation point, the worker processes consume workloads at a rate faster than the master process prepares the workloads. Hence the decoding frame rate will not increase with the number of tiles beyond the saturation point. However, this is not the case for R + W and W + W implementations. The bottleneck that limits the scalability of R + W and W + W implementations is the time spent by worker processes filling the pool of completed tasks. Since the pool of completed tasks is shared by the master process, when there is larger number of workers trying to write results to the pool, the iMesh network becomes congested towards the tile that runs the master process. This is the reason why implementations based on PWP result collection scale poorly.

Larger frame size videos: When the decoders are decoding videos of larger frame sizes, the saturation points of R + R and W + R implementations increase with frame sizes. For the case of 1080P decoding, the frame rate of the implementations based on CRP result collection scales almost linearly with the number of available tiles throughout the experiment. On the other hand, performance of implementations based on PWP result collection does not show improvement with different-sized workloads. R + W and W + W implementations both do not scale well beyond 20 tiles no matter what frame size the given video stream encompasses.

5.2 Encoder performance

Figures 12, 13, 14, 15 show the encoder performance for 4 different video frame sizes. The experimental results show that the encoders implemented with PWP + CRP and PWP + PWP exhibit very similar performance. Table 4 shows the maximum obtained FPS from the encoders.

CIF video encoding: Fig. 12 shows the performance of CIF video encoding. The fastest implementations are W + R and W + W, which have nearly similar performance. The fastest implementation (W + R) runs 31.7 % faster than the slowest one (R + R). The encoding frame rates saturate at 50 and 33 tiles for R + R and R + W, respectively, and 34 tiles for both W + R and W + W.

QCIF video encoding: Fig. 13 shows the performance of QCIF video encoding. The fastest implementation (W + W) runs 35.6 % faster than the slowest one (R + R).

Table 4 Encoder performance—max frame rate

Video size	Programming paradigms				
	R + R	R + W	W + R	W + W	MP
CIF	782.05	819.24	*1029.98*	1019.09	603.27
4CIF	199.21	217.20	268.00	*270.20*	163.41
720P	90.18	92.34	117.60	*119.18*	70.47
1080P	41.83	45.74	54.72	*54.79*	33.65

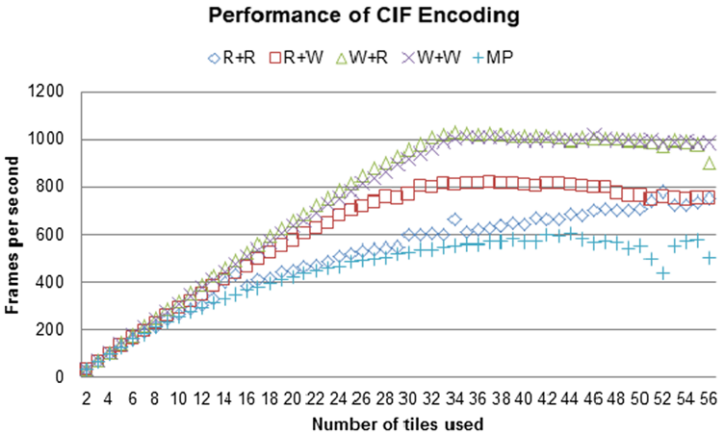


Fig. 12 Performance of CIF encoding

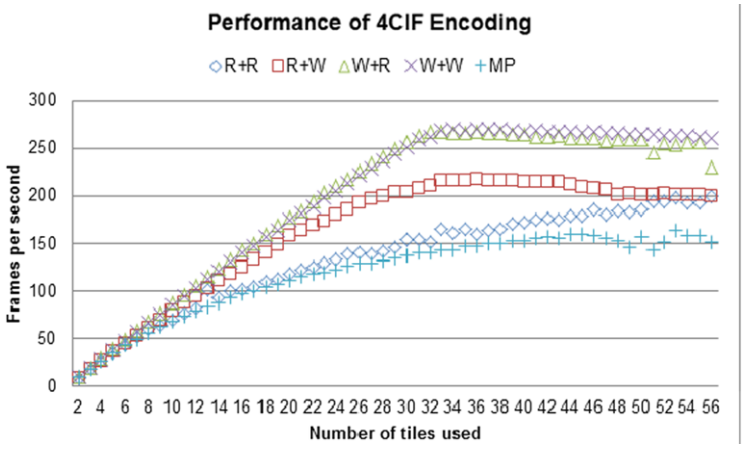


Fig. 13 Performance of 4CIF encoding

The frame rates saturate at 50 and 34 tiles for R + R and R + W, respectively, and 32 tiles for both W + R and W + W.

720P video encoding: Fig. 14 shows the performance of 720P video encoding. The W + R implementation runs 2.32 times faster than the R + W implementation. The frame rates saturate at 50 and 35 tiles for R + R and R + W, respectively, and 36 tiles for both W + R and W + W.

1080P video encoding: Fig. 15 shows the performance of 1080P video encoding. The W + R implementation runs 3.74 times faster than the R + W implementation. The frame rates saturate at 52 and 37 tiles for R + R and R + W, respectively, and 35 tiles for both W + R and W + W.

From the performance results obtained from encoders, it can be observed that the implementations can be separated into three types. The W + R and W + W encoders perform very similarly and are the fastest implementations. The R + R encoder scales

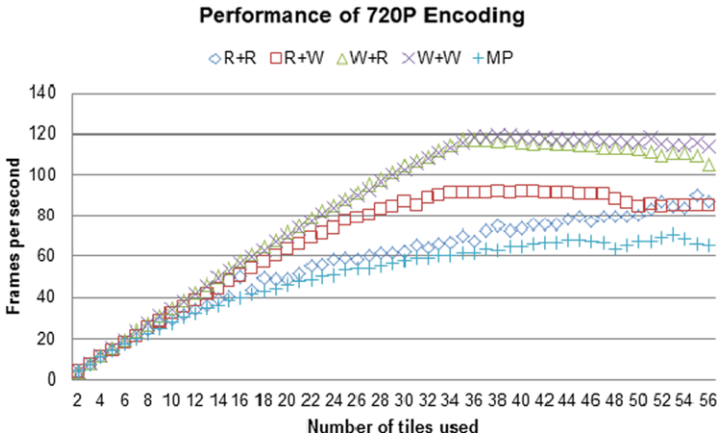


Fig. 14 Performance of 720P encoding

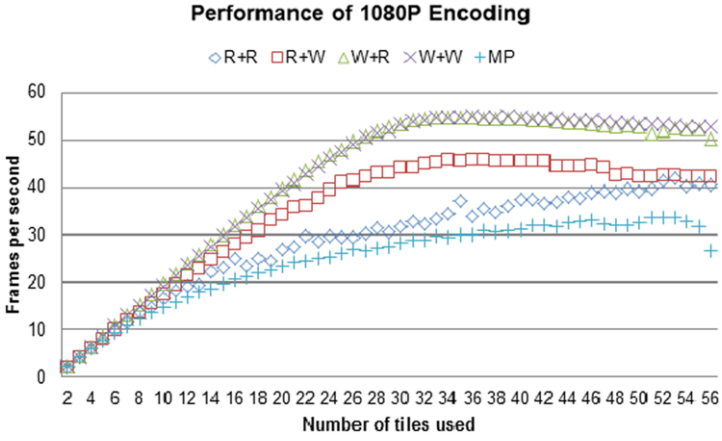


Fig. 15 Performance of 1080P encoding

almost linearly towards the maximum number of tiles. The R + W implementation runs faster but saturates earlier than R + R implementation. The scalability of the encoders is similar from small frame size to large frame size.

Experimental data show that the scalability bottleneck for W + R and W + W is the time spent by the master process filling the pool of pending tasks, which is distributed among all worker processes. The scalability bottleneck for R + R and R + W is the time spent by worker processes draining the pool of pending tasks, which is shared by the master process.

5.3 Summary

As shown in both decoder and encoder performance results, the scalability bottleneck for decoder applications is related to the shared access to the pool of completed

results and the scalability bottleneck for encoder application is related to the shared access to the pool of pending tasks. This is since the size of a completed task is far larger than the size of a pending task in a decoder application. On the other hand, in an encoder application, the size of a pending task is far larger than the size of a completed task. Therefore it is important to carefully arrange the shared memory access. The best implementation that suits both decoder and encoder is the PWP + CRP combination.

For decoder implementations, the reason why the decoding CIF, 4CIF and 720P videos does not scale beyond a certain point is the task granularity. When the number of worker processes increases beyond a certain number, the bottleneck is at the master process, since video frame sizes are not big enough for a worker process to spend enough time decoding it. When there is one master and a few workers, the input rate and output rate of the master process will be faster than decoding speed combined by all the workers. As for 1080P decoding, the decoding speed is slow, and the size of decoded frames is very large, so when using efficient implementations such as R + R or W + R, the combined throughput made by the workers has not reached the I/O limitation of the master, and these two implementations keep scaling throughout the spectrum of number of workers. As for all 4 cases of encoder implementations, the raw video is large and the input rate of master process is not high enough to supply tasks for more than 36 worker processes to saturate their combined encoding capability. Figure 16 shows the speedup and efficiency chart of the decoders and Fig. 17 those of the encoders. Based on the experimental results, they make the discussion more convincing and proving.

6 Conclusions and future work

New generations of multi-core and many-core processors bring higher performance within the same or lower power envelope. This advantage comes tied with the price of abstraction to application design and programming. Therefore, this study explores the programming paradigm for master–worker stream processing on the TILE64 many-core platform, in which a master–worker structure for stream processing and two shared memory programming paradigms—*consumer-read programming* and *producer-write programming*—are proposed. Experimental results show that a single programming paradigm does not guarantee good performances for both encoder and decoder implementations. For instance, although the PWP + PWP implementation performs best for the encoder, it only runs at about 25 % of the performance of W + R when decoding a 1080P video file. On the other hand, W + R performs quite well for both decoder and encoder in its hybrid implementation; we therefore conclude that the CRP suits best the implementation of the result collection part in a master–worker Motion JPEG decoder/encoder while PWP performs best in the task distribution part.

As further plans to the development of this research, we plan to evaluate CRP and PWP with variable buffer sizes as experiments in this paper were conducted using fixed-sized task pools and result pools. We believe that it will be interesting to discuss

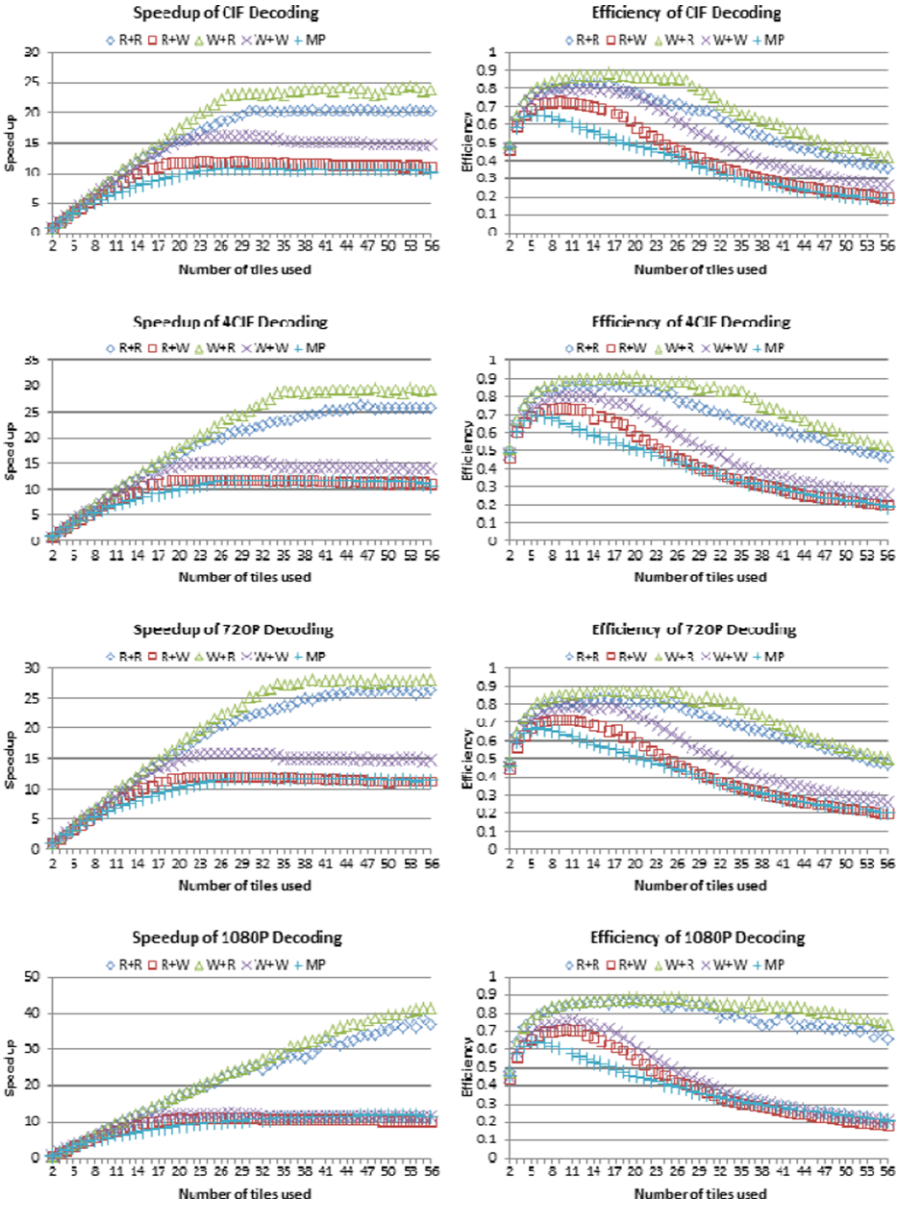


Fig. 16 Speedup and efficiency chart of the decoders

the correlation between memory usage and number of workers. Also we would like to further explore this topic by applying CRP and PWP onto more complex paradigms such as hierarchical master–worker structures. We would also like to explore the possibilities of applying CRP and PWP to wider areas of programming structures and applications.

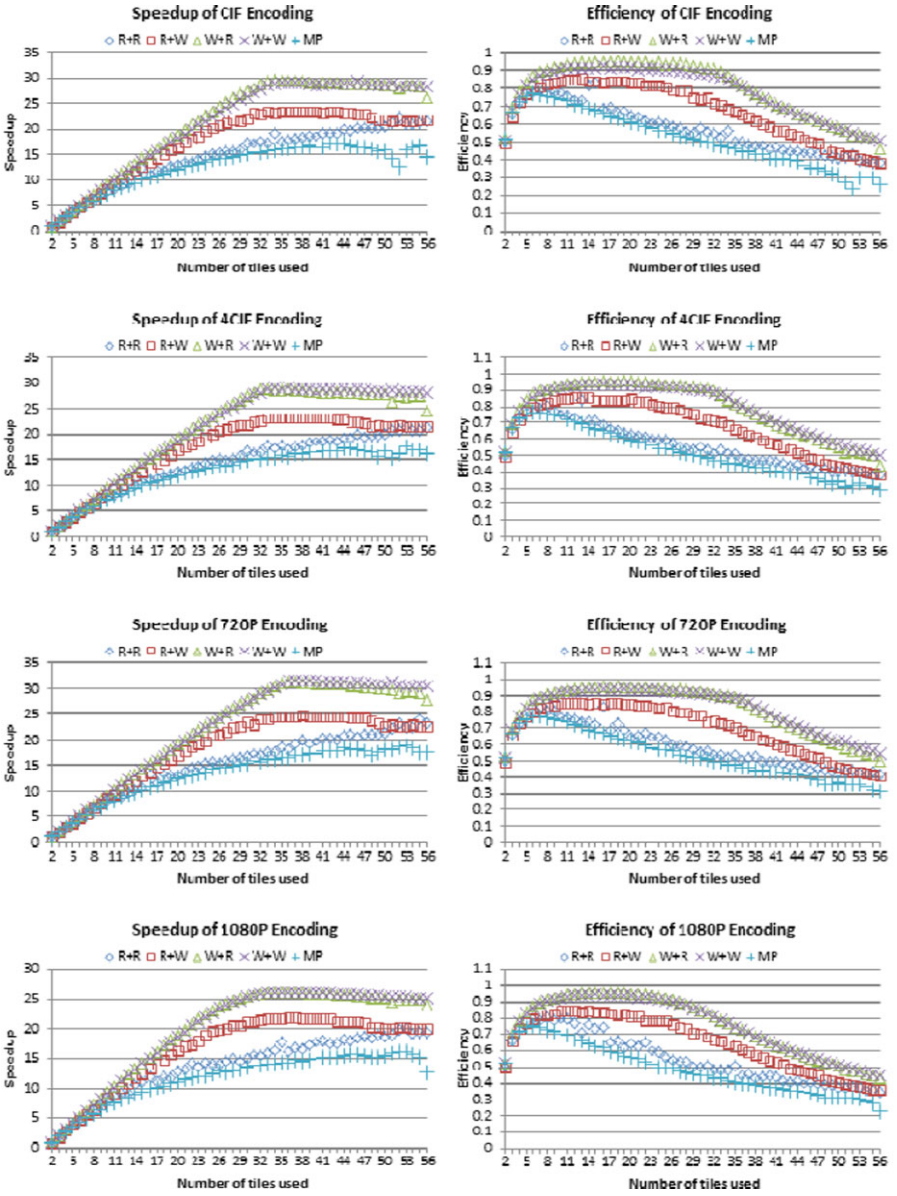


Fig. 17 Speedup and efficiency charts of the encoders

References

1. Borkar S (2007) Thousand core chips: a technology perspective. In: Proceedings of the 44th design automation conf (DAC 07), pp 746–749. doi:10.1145/1278480.1278667
2. Parkhurst J, Darringer J, Grundmann B (2006) From single core to multi-core: preparing for a new exponential. In: Proceedings of the IEEE/ACM int conf computer-aided design (ICCAD 06), pp 67–72. doi:10.1145/1233501.1233516

3. Karam L, AlKamal I, Gatherer A, Frantz G, Anderson D, Evans B (2009) Trends in multicore DSP platforms. IEEE Signal Process Mag 26(6):38–49. doi:10.1109/MSP.2009.934113
4. Sutter H (2005) The free lunch is over: a fundamental turn toward concurrency in software. Dr Dobb's J 30(3):202–210
5. Chen G, Li F, Son SW, Kandemir M (2008) Application mapping for chip multiprocessors. In: Proceedings of the 45th design automation conf (DAC 08), pp 620–625. doi:10.1145/1391469.1391628
6. Tan G, Sun N, Gao GR (2007) A parallel dynamic programming algorithm on a multi-core architecture. In: Proceedings of the 19th ACM symp parallel algorithms and architectures (SPAA 07), vol 07, pp 135–144. doi:10.1145/1248377.1248399
7. Bell S, Edwards B, Amann J, Conlin R, Joyce K, Leung V, MacKay J, Reif M, Liewei B, Brown J, Mattina M, Chyi-Chang M, Ramey C, Wentzlaff D, Anderson W, Berger E, Fairbanks N, Khan D, Montenegro F, Stickney J, Zook J (2008) TILE64 processor: a 64-core SoC with mesh interconnect. In: Proceedings of the IEEE intl solid-state circuits conf (ISSCC 08), pp 88–598. doi:10.1109/ISSCC.2008.4523070
8. Chen S, Chen S, Gu H, Chen H, Yin Y, Chen X, Sun S, Liu S, Wang Y (2010) Mapping of H.264/AVC encoder on a hierarchical chip multicore DSP platform. In: Proceedings of the 12th IEEE int conf high performance computing and communications (HPCC 10), pp 465–470. doi:10.1109/HPCC.2010.82
9. Boutellier J, Jaaskelainen P, Silven O (2007) Run-time scheduled hardware acceleration of MPEG-4 video decoding. In: Proceedings of the 2007 int symp system-on-chip, pp 1–4
10. Yung NHC, Leung K-K (2001) Spatial and temporal data parallelization of the H.261 video coding algorithm. IEEE Trans Circuits Syst Video Technol 11(1):91–104
11. Rodriguez-Fernandez D, Vilarino DL, Pardo XM (2009) A pixel-parallel moving object segmentation and tracking algorithm for video surveillance applications. In: Proceedings of the 6th int symp image and signal processing and analysis (ISPA 09), pp 614–619
12. Berthold J, Dieterle M, Loogen R, Priebe S (2008) Hierarchical master–worker skeletons. In: Proceedings of the 10th int conf practical aspects of declarative languages (PADL 08). Lecture notes in computer science, pp 248–264
13. Benoit A, Marchal L, Pineau JF, Robert Y, Vivien F (2010) Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. IEEE Trans Comput 59(2):202–217. doi:10.1109/TC.2009.117
14. Hoffmann H, Wentzlaff D, Agarwal A (2010) Remote store programming. In: Patt Y, Foglia P, Duesterwald E, Faraboschi P, Martorell X (eds) High performance embedded architectures and compilers. Lecture notes in computer science, vol 5952. Springer, Berlin, pp 3–17. doi:10.1007/978-3-642-11515-8_3
15. Awasthi M, Nellans DW, Sudan K, Balasubramonian R, Davis A (2010) 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), pp 319–330. doi:10.1145/1854273.1854314
16. Abts D, Jerger NDE, Kim J, Gibson D, Lipasti MH (2009) Achieving predictable performance through better memory controller placement in many-core CMPs. In: Proceedings of the 36th int symp computer architecture (ISCA 09), pp 451–461. doi:10.1145/1555754.1555810
17. Lin X-Y, Huang C-Y, Yang P-M, Lung T-W, Tseng S-Y, Chung Y-C (2011) Parallelization of motion JPEG decoder on TILE64 many-core platform. In: Hsu C-H, Malyshkin V (eds) Methods and tools of parallel programming multicomputers. Lecture notes in computer science, vol 6083. Springer, Berlin, pp 59–68. doi:10.1007/978-3-642-14822-4_7
18. Jackson JD, Hatcher PJ (2011) 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), pp 219–224
19. Goux JP, Kulkarni S, Linderoth J, Yoder M (2000) An enabling framework for master–worker applications on the computational grid. In: Proceedings of the 9th int symp high-performance distributed computing (HDPC 00), pp 43–50
20. Fujimoto RM, Malik AW, Park A (2010) Parallel and distributed simulation in the cloud. SCS M&S Mag 1(3):1–10
21. Rynge M, Callaghan S, Deelman E, Juve G, Mehta G, Vahi K, Maechling PJ (2012) 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), pp 1–8. doi:10.1145/2335755.2335846

22. Blagojevic F, Nikolopoulos DS, Stamatakis A, Antonopoulos CD (2007) Dynamic multigrain parallelization on the cell broadband engine. In: Proceedings of the 12th ACM SIGPLAN symp principles and practice of parallel programming, pp 90–100. doi:10.1145/1229428.1229445
23. Zheng G, Meneses E, Bhatelé A, Kalé LV (2010) Hierarchical load balancing for Charm++ applications on large supercomputers. In: Proceedings of the 39th int conf parallel processing workshops (ICPPW 10), pp 436–444. doi:10.1109/ICPPW.2010.65
24. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113. doi:10.1145/1327452.1327492
25. Giseok C, Jeongsoo Y, Jeonghoon C, Jongho N (2007) Design and implementation of a real-time video player on tiled-display system. In: Proceedings of the 7th IEEE int conf computer and information technology (CIT 07), pp 621–626
26. Nunome T, Tasaka S (2004) Application-level QoS assessment of continuous media multicasting in a wireless ad hoc network. In: Proceedings of the 2004 IEEE int conf communications, pp 2047–2053
27. Pereira R, Azambuja M, Breitman K, Endler M (2010) An architecture for distributed high performance video processing in the cloud. In: Proceedings of the 3rd IEEE int conf cloud computing (CLOUD 10), pp 482–489
28. Ali U, Bilal M (2006) Video based parallel face recognition using Gabor filter on homogeneous distributed systems. In: Proceedings of the 2006 IEEE int conf engineering of intelligent systems, pp 1–5
29. MJPEG Tools. http://mjpeg.sourceforge.net
30. Wang Z, Liang L, Yang G, Zhang X, Sun J, Zhao D, Gao W (2011) A novel macro-block group based AVS coding scheme for many-core processor. J Signal Process Syst 65(1):129–145. doi:10.1007/s11265-010-0543-0