Front. Comput. Sci. China 2009, 3(3): 290–301
DOI 10.1007/s11704-009-0028-0

REVIEW ARTICLE

Four styles of parallel and net programming

Zhiwei XU (✉)[1], Yongqiang HE (✉)[1, 2], Wei LIN[1, 2], Li ZHA[1]

1 Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

2 Graduate School of the Chinese Academy of Sciences, Beijing 100039, China

Abstract This paper reviews the programming landscape for parallel and network computing systems, focusing on four styles of concurrent programming models, and example languages/libraries. The four styles correspond to four scales of the targeted systems. At the smallest coprocessor scale, Single Instruction Multiple Thread (SIMT) and Compute Unified Device Architecture (CUDA) are considered. Transactional memory is discussed at the multicore or process scale. The MapReduce style is examined at the datacenter scale. At the Internet scale, Grid Service Markup Language (GSML) is reviewed, which intends to integrate resources distributed across multiple datacenters.

The four styles are concerned with and emphasize different issues, which are needed by systems at different scales. This paper discusses issues related to efficiency, ease of use, and expressiveness.

Keywords concurrent programming, CUDA, SIMT, transactional memory, MapReduce, GSML

1 Introduction

In the past ten years, many new concurrent programming models, constructs, languages, and libraries have been proposed and implemented, some of which have gained wide spread influences. This blooming in concurrent programming, i.e., parallel and net programming, is driven by rapid advances in both concurrent applications and concurrent systems.

This paper reviews the landscape for concurrent programming by discussing four representative styles. They correspond to four scales of the targeted systems, ranging from an accelerator chip to Internet scale. We will focus on programming styles that are relatively new and will not discuss traditional styles such as Message Passing Interface (MPI), Open Multi-Processing (OpenMP), and Universal Product Code (UPC). We will also exclude languages for supercomputing such as X^{10} and Chapel.

Figure 1 illustrates the landscape of concurrent programming. We can observe at least four distinct scales or levels. We can better understand Fig. 1 through an example. Suppose a user wants to render an animation movie of 60 minutes, with $30\times60\times60$=108 000 frames, and the user needs to do it ten times to get the final film. Supposing rendering one frame needs 20 seconds on a single Central Processing Unit (CPU), the computation of rendering the entire animation movie will need 250 days. However, the user can use computing resources on the net, called clouds or grids, to speed up the computation to finish in half a day.

Concurrent programming at four levels (scales) is involved in the above example:

- Internet scale. The user needs to write a concurrent program to execute the entire set of animation jobs on multiple datacenters on the Internet. We will use GSML to discuss Internet-scale concurrent programming.
- Datacenter scale. Once a job is dispatched to a datacenter, a multiple-process program is executed, where each process is responsible for executing a chunk of the job. We will discuss one particular style, the MapReduce model, as an example of datacenter-scale concurrent programming.
- Process scale. Even to support a single process executing on a multicore CPU, we need to consider many programming issues [12]. We will focus on the synchronization issue and discuss one new construct called transactional memory.
- Accelerator scale. At the smallest scale, within one process where a chunk of the rendering job for animation is executed, some operations can be accelerated by

Received January 20, 2009; accepted March 5, 2009

E-mail: zxu@ict.ac.cn; heyongqiang@software.ict.ac.cn

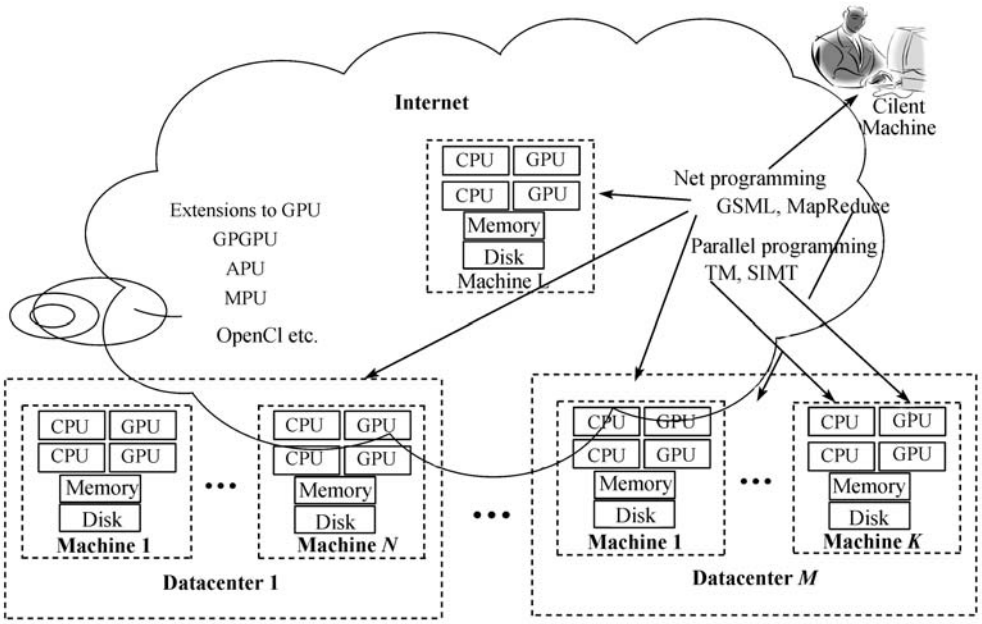

Fig. 1 Landscape of concurrent programming

special-purpose hardware called a coprocessor or accelerator, such as a Graphics Processing Unit (GPU). We will discuss a programming model called SIMT and its implementation architecture called CUDA.

2 Accelerator programming: SIMT and CUDA

We start with the lowest level (the smallest scale): programming for accelerators, which are special-purpose or multiple purpose hardware units designed for speeding up specific sets of applications. A prominent example is a GPU. This paper discusses in detail a representative programming model SIMT and its implementation CUDA on one family of GPUs.

However, we need to point out two emerging trends. First, people are starting to use GPUs to accelerate general purpose applications [19, 21, 26, 27]. GPU vendors also try to provide a more general and flexible parallel programming model and an underlying architecture to supply friendly Application Programming Interfaces (APIs), flexible memory access patterns, and many arithmetic and logic units. A new term, General-Purpose Graphics Processing Unit (GPGPU), is created to represent such research efforts. Second, the industry and academic community has established a standard called Open Computing Language (OpenCL) [13] to support parallel programming on different accelerators in a hardware platform independent way.

2.1 GPU architecture and SIMT execution

We will use NVIDIA's CUDA [18] as an example and discuss the main issues for parallel programming at the accelerator scale. The baseline architecture of a NVIDIA's GPU is partitioned into a number of Streaming Multiprocessors (SMs), and there are up to 30 SMs in NVIDIA T10 GPU [5]. Each SM has eight Scalar Processing (SP) cores. The eight cores share one instruction unit. They execute the same instruction stream of the same application kernel but different threads at a time. This execution style is called *Single Instruction Multiple Thread* (SIMT). The SIMT model allows a programmer to write thread-level parallel code for independent, scalar threads, and data-parallel code for coordinated threads.

CUDA is well suited for using fine-grain threads and is able to active thousands of threads at the same time with zero-overhead thread scheduling. CUDA tries to generate and maintain thousands of threads in flight, in contrast to the use of large caches to hide memory latencies in CPU designs.

The SIMT programming model on CUDA is realized by using American National Standards Institute (ANSI) C extended with a small set of abstractions for expressing parallelism and memory locality. As a whole, the language is easy to use.

2.2 Relation between host and device

A GPU (device) operates as a coprocessor to the main CPU (host) to accelerate the data-parallel compute-intensive phases of an application. The phases of little data parallelism are usually implemented in the host and use host C compiler to compile into an object file. On the other hand, the parts with rich data parallelism are isolated into functions, also

```
__constant__ float c_gx[][] = constant Gaussian window;
void Main(const float* I,  int nrows, int ncols, float * Ires) {
① allocate memory space on the device
cudaMalloc(&d_Iin, size);      cudaMalloc (&d_Iout, size);
② transfer data from host to device:
cudaMemCpy (d_Iin, I, size, cudaMemcpyHostToDevice);
③ execution configuration setup(2D here):
dim3 dimBlock (BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid (ncols / dimBlock.x, nrows / dimBlock.y);
④ kernel call kernelOne<<< exe config >>> ( args… ):
Gauss_kernel <<< dimGrid, dimBlock >>> (d_Iin, nrows,
                                        ncols, d_Iout);
⑤ transfer results from device to host:
cudaMemcpy (Ires, d_Iout, size, cudaMemcpyDeviceToHost);
cudaFree (d_Iin);
cudaFree (d_Iout); }
```

Fig. 2 Typical host side code

called *kernels*, and are executed on the device using multiple threads. These kernel functions are compiled by the NVIDIA CUDA C compiler and the kernel GPU object code generator.

Both the host and the device maintain their own Dynamic Random Access Memory (DRAM). The host code transfers data to and from the GPU's device memory using API calls that utilize the device's Direct Memory Access (DMA) engines. In details, the host code uses "cudaMalloc", "cudaFree," and "cudaMemcpy" library calls to allocate/free a linear memory in device memory and to transfer data between the host and the device.

We use as an example a Gaussian smooth operation in image processing. As Fig. 2 illustrates, steps ①, ②, and ⑤ correspond to the memory allocation, input data transfer, and result transfer, respectively. A kernel function is initiated running on the device by performing a function call from the host, as shown in step ④.

2.3 Thread hierarchy and management

In order to provide a natural way to represent the computation on the elements of a 1D (1 dimensional) vector, 2D matrix, or a 3D field, the threads are organized into a grid of *thread blocks*, which is a two-level hierarchy with multiple dimensions, as Fig. 3 illustrates. A block is a set of tightly coupled threads, where each thread is identified by a thread ID. The grid is a set of loosely coupled blocks with similar size and dimension, where each block is identified by a block ID. A kernel function represents a phase of parallel execution. It is first formed as a grid, where there maybe 3D, 2D, or 1D blocks. Each block is identified through using 3-, 2-, or 1- component coordinates to generate the block ID.

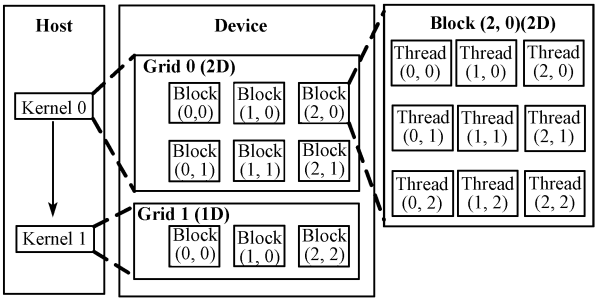

Fig. 3 CUDA thread hierarchy

Each block is only assigned to one SM. In the lower level of block, the threads are also explicitly organized as an up to 3D array. Threads also have unique coordinates and an upper limited number due to resource limitation (no more than 512 threads per block). For instance, in a 2D block of size (Dx, Dy), the thread ID of a thread with index (x, y) is (x + y Dx). These IDs can form an offset to the input pointer to help with complex addressing for the private input data.

Worker management in CUDA is done implicitly. This means that programmers do not manage the creations, scheduling, and destructions of threads. However, workload partition and worker mapping in CUDA is done explicitly by programmers. They need to specify the dimensions of the grid and of the block required to process a certain kernel. In detail, they define the threads to be run in parallel by using "Gauss_kernel⟨⟨⟨dimGrid,dimBlock⟩⟩⟩ (arguments)." as specified by step ④ in Fig. 2, where 1) Gauss_kernel is the kernel name; 2) dimGrid and dimBlock are the dimensions of the grid and of the block, respectively; and 3) the arguments is the passing parameters for the kernel function.

2.4 Memory hierarchy

There are five memory spaces in the GPU. The most important two are the "global" and the "shared" spaces. The former is the global shared off-chip DRAM of the device, and the shared memory is the high-speed scratched-pad memory on chip, 16-KB in each SM. It is shared by threads in the same block and supports 16 simultaneously accesses if there is no bank confliction. In Fig. 4, each thread only loads one element of d_Iin into the s_I, which resides in the shared memory. After all of the pixel matrices are loaded into it, the s_I is shared among the threads in one block and is used by the following separate computations.

The other spaces are the "constant", "local," and "texture" memory. The local memory is the default definition for most variables in a kernel function, while a small number of variables will first choose registers. If there are not enough registers, variables will be spilled into the local memory. Both of the constant and texture memory have a cache on-

```
__shared__ float s_I[BLOCK_SIZE][BLOCK_SIZE];
//load pixel value from global memory to shared memory
s_I(ty,tx) = (float)(d_Iin(ty,3*tx)/3 +
                d _Iin(ty,3*tx+1)/3 + d_Iin(ty,3*tx+2)/3);
//synchronize to make sure the pixel matrices are loaded
__syncthreads();
//do the gaussian smooth
if(tx>gausswidth && tx<BLOCK_SIZE-gausswidth &&
  ty>gaussheight && ty<BLOCK_SIZE-gaussheight)
{
  for(int m=0;m<gaussheight;m++)
     {
          for(int n=0;n<gausswidth;n++)
       {
         a+=s_I(ty+m-gap, tx+n-gap) * c_gx(m,n);
       }
  }
}
//write the result back to global memory
d_Iout(ty,tx) = a;
```

Fig. 4 Kernel function of Gaussian smooth for image executed on device side

chip and read-only, so that they can be more quickly accessed. Programmers can conveniently manage the data layout. For example, they can declare a variable in shared memory or constant memory with the prefix of "_shared_" or "_constant_".

It should be noticed that, although this kind of memory hierarchy can offer us a large optimization space, it becomes much more complex to tune for best performance.

2.5 Synchronization and communication

The threads within a block are allowed to synchronize with each other via a "_syncthreads" library call, which is equal to a barrier. There is no direct support for synchronization among threads from different blocks in the same grid. Therefore, CUDA recommends that thread blocks be independent. This restriction on the dependencies between blocks of a kernel provides scalability. However, it also implies that when developing an application for CUDA, a main consideration for the programmer is to decompose parallel work into separate kernels to avoid the needs to do global communication or synchronization among blocks.

2.6 Acceleration performance

In the Gaussian smooth example, the computation for each pixel only depends on the associating input pixels. Therefore, it is convenient to partition the image into small blocks according to the size of shared memory and make all the blocks independent of each other. When all the pixels within a block are available in the shared memory, the block's thread can finish the computation of corresponding pixel without any global memory access, which means a high compute-to-memory ratio. The processing of each pixel is the same as others, following the SIMT execution model. Table 1 shows nice speedup compared.

Tables 1 Speedups in Gaussian Smooth

image size	CPU (ms)	GPU (ms)	Speedup
200 × 200	3.094	0.152	20.36
400 × 400	12.677	0.52	24.38
600 × 600	24.662	1.06	23.27
800 × 800	44.05	1.878	23.46
1000 × 1000	69.016	3.099	22.27

Note: CPU: Intel Core2 6300@1.86GHz; GPU: NVIDIA GeForce 8800-GT@1.5GHz.

Many other applications, both academic researches and industrial products, including linear algebra, bioinformatics, machine learning, etc., have been accelerated using CUDA. Readers can refer to [5] for more details. A performance of 206 Gflops has been observed for linear algebra applications on a GeForce8800GTX GPU that has a 346-Gflops peak speed.

3 Process synchronization: transactional memory

Today processors tend to integrate more and more cores in a single chip. The resulting multicore chip allows high parallelism in a single chip by letting multiple threads of a process executing concurrently. However, parallelism is not free [12]. Few programs today are well written to exploit the multicore parallelism advantages.

In this section, we will focus on the transactional memory construct that is proposed to enhance parallelism within a process by addressing the share memory synchronization problem.

We will first introduce transactional memory, including why we need transactional memory and what transactional memory can provide. Then, we will center on some issues. For efficiency, issues like optimistic or pessimistic executions, granularity, and conflict resolution policy are considered. For expressiveness and ease of use, issues like conditional coordination and composability of transactions are considered.

3.1 Introduction to transactional memory

Since multiple cores can read from and write to the same memory location concurrently, we need a mechanism to coordinate their accesses. Though synchronization primitives like locks, monitors, and others from traditional multithread

programming can be borrowed, these mechanisms either do not scale well in the multicore systems, or they are difficult to use by programmers.

Locking plays a central role for protecting a shared data structure from concurrent operations in traditional multi-thread programming. However, locking requires mutual exclusion whether it is needed or not. When applying locks, programmers need to carefully code to avoid deadlocks and unlock misses. In general, locks can cause nine kinds of problems, such as priority inversion, convoying, and deadlock [12].

In addition to the above problems, locking primitives also have other disadvantages. For example, atomic primitives such as CompareAndSwap operate on only one word at a time, resulting in complex algorithms.

• transactional memory

Transactional memory was proposed to offer efficient and easy to use synchronization primitives with a lock-free memory access solution [9]. Transaction mechanism has long been studied in the database community, and the values it delivers are Atomicity, Consistency, Isolation and Durability (ACID). One difference between transactional memory and transaction in database is that transactional memory only considers Atomicity, Consistency and Isolation (ACI). With transactional memory, programmers can write programs without considering synchronization with other execution bodies. The transaction mechanism guarantees concurrent accesses from different execution threads to be isolated with each other. A transaction produces the same result as if there were no other transactions currently running. Atomicity guarantees that actions made in a transaction either all become effective or none. All intermediate states before a transaction is committed are invisible to other transactions.

Figure 5 shows a transactional memory example, which transfers money from one account to another account. The atomic code body from step ① to step ④ in the example encloses a transaction. Step ① starts the transaction, and step ④ commits it. In the case where either step ② and step ③ fail, the money transfer transaction will abort.

Many implementations of transactional memory have been released, either entirely in software or entirely in hardware or in combination of the two. We will not overview all such work. For more details of these systems, readers can refer to [10, 15, 17, 22, 24]. Overview and comparison of hardware-based, software-based, and hybrid transactional memory implementations can be found in Ref. [14]. In the following, we will look at some common issues transactional memory needs to address for efficiency, programmability, and expressiveness.

```
// transfer money from one account to another account
Void Transfer(Account from, Account to, int amount){
① begin a transaction:
 atomic {
        ② draw money from one account, may fail
          int withdraw = from.pop (amount);
        ③ store money to another account
          to.add (withdraw);
    ④ commit transaction
    }
}//end_transfer
```

Fig. 5 A transactional memory example

3.2 Issues of transactional memory

3.2.1 Optimistic or pessimistic executions

In optimistic execution, a transaction makes data modifications as if there were no others concurrently modifying them. It checks for conflicts with other transactions when it finishes and wants to commit its changes. If there are conflicting transactions, it either rolls back or commits, causing other involved transactions to roll back. In the pessimistic way, a transaction always checks for conflicts as it progresses.

Which strategy is better depends on workloads. Although the optimistic strategy can save conflict validation cost, it may cause the entire transaction to roll back when conflicts occur. The pessimistic strategy can save time when rolling back since it detects a conflict immediately, but it dose not perform well in workloads where conflicts seldom occur. Currently, most transactional memory implementations adopt the optimistic execution strategy.

3.2.2 Granularity

The granularity problem can be defined as the memory granularity that a transactional memory implementation intents to protect. Currently, there are three kinds of granularities, object granularity, cache-line granularity, and word granularity. Though word granularity can avoid unnecessary guard if two transactions concurrently access different words of an object, it makes the implementation of word granularity much more complicated than object granularity. The current status is that hardware-based transactional memory systems generally protect memory at word or cache-line granularity; most software-based transactional memory systems are implemented with object granularity. However, WSTM is an example of software transactional memory with word granularity [7].

3.2.3 Conflict resolution

While concurrent read-read operations are always good, con-

current transactions with read-write or write-write operations to the same object may conflict with each other. If there is no conflict, all transactions can successfully commit. When a conflict occurs, only one of the involved transactions can successfully commit, while all the others abort and roll back.

An appropriate strategy for detecting and resolving conflicts can significantly influence the overall performance. For example, for read-intensive workloads, an implementation can be optimized as follows. The implementation can record each object a transaction reads and maintain version numbers for these objects. Conflict detection can be made simply by checking the version numbers of an individual transaction.

When a conflict is detected, there must be ways to find out which transaction should be committed and which ones should roll back. Several papers have studied conflict resolution policies [6, 23]. A backoff technique and priority are used in the implementation of conflict resolution policy called Polka [14]. This backoff technique is very similar to that used in Ethernet but combines a priority concept. The more objects a transaction already accessed, the higher priority it has. When a transaction wants to access an object and finds a conflict with another transaction with higher priority, the lower priority transaction backs off.

3.2.4 Conditional coordination

It seems that the transactional memory mechanism can well replace locks of traditional multithread programming. Besides lock, there are other important coordination mechanisms, including semaphore and conditional variable. With conditional coordination, an execution is pending waiting for some requirement to be satisfied. Conditional coordination primitives are usually used together with locks. An execution first holds a lock, and if a condition is satisfied, it progresses. If the condition is not satisfied, the execution releases its lock and waits for condition notification, which enables the execution to restart by retrying and testing the condition.

The Haskell Transactional Memory [8] introduces the "orElse" and "retry" primitives for addressing this kind of conditional coordination problems. Executing a retry statement will abort the surrounding transaction and wait for executing when the value of the memory location it previously accessed changes.

The orElse statement gives a transaction of multiple choices. The pseudocode in Fig. 6, modified from [11], shows an example of orElse. The example transfers money from either account src1 or account src2 to account dest. The transaction first executes step ②, transferring money

```
// transfer money from one of two source accounts to
// another account
Void Transfer(Account src1, Account src2, Account dest,
              int amount ){
① begin a transaction:
atomic {
      ② draw money from src1, may fail by calling retry
        int withdraw = src1.pop(amount);
        to.add(withdraw);
  } orElse{
      ③ draw money from src2, may fail by calling retry
        int withdraw = src2.pop(amount);
        to.add(withdraw);
  }
}//end_transfer
```

Fig. 6 An example of conditional coordination of money transfer

from account src1 to dest. If it succeeds, then the transaction commits. If it fails, the transaction will execute statements starting from step ③, which tries to transfer money from account src2 to dest. If the transfer from account src2 to dest succeeds, the transaction commits. If transfers from src1 to dest and from src2 to dest both fail, the transaction will re-execute from step ① later.

3.2.5 Composable and nested transaction

Another disadvantage of the lock mechanism that we have not mentioned yet is its lack of composability. In the example shown in Fig. 5, if the logic is implemented with locks, the from.pop operation owns its private lock, and the to.add operation also has its own lock, and if they are composed together to the transfer example, there needs a large lock outside. Holding the outside large lock, we still need to hold the private locks of from.pop and to.add. If there exists another concurrent task that acquires from.pop () and to.add () in a reverse order, a deadlock may occur.

Transactional memory allows the operations to be composed simply. As Fig. 5 shows, from.pop () and to.add () are individual transactions, and they can be combined together into the transfer transaction. This then becomes a nested transaction example.

Research on the efficiency and semantic issues of nested transactions is still on going. For example, what should happen if an inner transaction aborts? Should it cause the most outside transaction as a whole to be rolled back? Which transaction should back off?

4 Datacenter programming: MapReduce

"The data center is the computer" [20]. Current programs

for data centers can utilize thousands of machines in the whole data center. Concerns of these programs include parallelizing a computation among thousands of computers inside a datacenter, partitioning data and feeding input into each computation unit, scheduling computation tasks, balancing workload, etc. Engineers in Google abstracted their computation needs and offered two primitives, map and reduce [4].

MapReduce is popularized mainly for its ease of use, simple parallelism, built-in ability for fault tolerance, and nonstop system scalability. The MapReduce model has gained more and more attention from both academia and open source communities. Moreover, several open source implementations have been released, among which the most popular one is Hadoop MapReduce.

We first examine the MapReduce abstraction in the first subsection to demonstrate its expressiveness and ease of use. For efficiency, we discuss scheduling, locality, and partition functions. Finally, we will discuss the performance results and limitations of MapReduce.

4.1 A high-level programming abstraction

MapReduce provides programmers with two functions, Map and Reduce, which can be combined to express complex data processing tasks. A typical MapReduce computation can be split into three stages, map, shuffle, and reduce. At the map stage, the input file's data for Map function is formatted to (key, value) pairs, and for each pair, the map function produces some intermediate key value pairs. At the shuffle stage, the intermediate key value pairs produced by maps are partitioned, and each partition is routed to a certain reducer for further processing. At the reduce stage, the input fed into reduce function is a key and a corresponding list of values. The reduce function takes a key, iterates through the key's corresponding values, performs some aggregation operations, and outputs results.

A runtime dataflow of MapReduce is illustrated in Fig. 7. In Fig. 7, an input file is logically partitioned into multiple splits, and each map instance, called map task, is responsible for processing one split at a time. One map task will produce at most one intermediate key-value local file, called partition, for each reduce instance. After map tasks are finished, reducers are informed of the intermediate data locations for them.

The main concerns of the MapReduce programming model is its expression ability and the simplicity of its interfaces provided for programmers.

4.1.1 Expressiveness

The wide adoption of MapReduce inside and outside Google shows its ability for expressing computations over large datasets. Multiple papers have studied the MapReduce model's applicability and especially provided comparison with Structured Query Language (SQL). References [2, 3] show that MapReduce can provide some functionalities that SQL provides. One query statement written in SQL can be translated into MapReduce jobs and can run concurrently inside a datacenter. The MapReduce is especially fit for aggregation operations, such as summarize, average, group, order, etc.

4.1.2 Interface simplicity

We use as an example of a statistical application (PV), which computes page views for various internet sites. We have implemented this application both in traditional approaches in single-machine environment and in MapReduce approach. The pseudocode of Fig. 8 shows the sequential single machine implementation, and the pseudocode of Fig. 9 shows the MapReduce approach.

As Fig. 8 shows, the sequential single machine implementation consists of two external merge sorts and a sequential scan and accumulation computation. The whole program is composed of 465 lines of Java code. Coding and debugging this simple program costs six person hours. The MapRe-

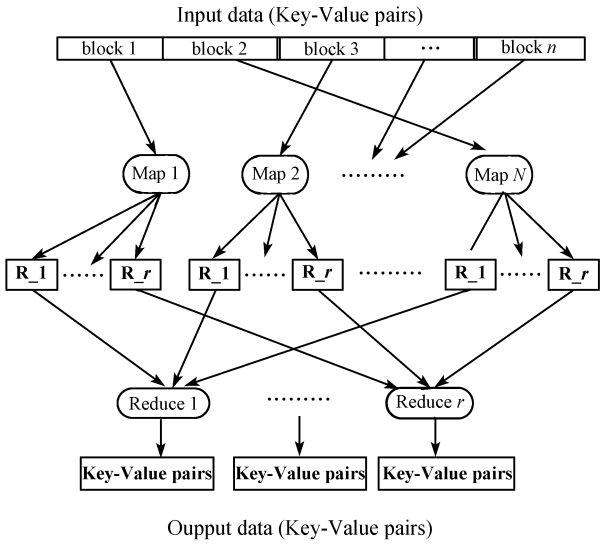

Fig. 7 A typical dataflow of MapReduce

```
// sequential single machine implementation
int main() {
  external_merge_sort_according_sitename(input file);
  summarize_page_view_for_each_site ();
  external_merge_sort_according_site_pv ();
}
```

Fig. 8 Single machine implementation of PV statistic

```
//Map-Reduce implementation consisting two jobs
//first job, accumulating page view for each site

Map (Key k, Value v, OutputCollector out)
{ //key: site name, value: neglected
   out.collect (k, 1); //key: site_name, value: 1
}
Reduce (Key k, Iterator values_iterator, OutputCollector out) {
     int count=0;
     while (values_iterator.hasNext ())
     {
      Values v=values_iterator.next ();
      count =count+v.getValue ();
     }
     out.collect (k, count);
}

// second job, sorting according the page view count
// map-reduce neglected since they do nothing just
// emitting key-values again and handle the key value
// pair to the map-reduce system for sorting
// Partition does range partition by pv of each site

Partition (Key k, Value v, int number_of_reducers)
{
   int boundary=Long.Max_Value/ number_of_reducers;
   return Math.floor(v/boundary);
}
```

Fig. 9 MapReduce implementation of PV statistic

duce approach, shown in Fig. 9, consists of 200 lines of Java code, where only 74 lines of code are written from scratch. The MapReduce approach costs about two person hours for coding and debugging.

4.2 MapReduce runtime system

MapReduce programmers specify the map and reduce functions. The runtime system takes care of the execution life cycle of the MapReduce job, including parallelizing the computations, partitioning data, feeding input data to computation units, scheduling, load balance, fault tolerance, and dynamic scalability. User specified map and reduce functions are automatically parallelized by the runtime system. As Fig. 7 shows, the runtime system splits input files into units that are concurrently processed by map tasks. Moreover, reduce tasks are also concurrently running on multiple nodes each processing one part of the whole job.

The MapReduce runtime system treats the whole datacenter as a computer. There are many factors to be concerned about.

4.2.1 Scheduling

Scheduling in MapReduce can be defined as deciding the time to run a job and finding the appropriate subset of computers for the job. Given a task, a processing unit of a job, scheduling is to decide which computer should run this task. While much work on scheduling has been done in traditional computer fields, the scheduling problem has not been widely studied in MapReduce runtime system. Google's original MapReduce paper [4] reveals little about their scheduler implementation. Researchers can refer to the open source MapReduce implementation, Hadoop MapReduce, which ships with three schedulers and can be easily plugged in with different scheduler implementations.

Issues considered by schedulers in MapReduce runtime have much difference with traditional operating system schedulers.

The straggler problem [4] is a specific issue MapReduce scheduler should consider, in which a job cannot be finished due to some subtasks performing extremely poorly. How to define the boundary and detect a straggle is what a specific scheduler should take care of. One specific technique schedulers in MapReduce can use for speeding up jobs and for trying to overcome potential component failures is the speculative execution technique. Speculation is heavily used to overcome the straggler problem in MapReduce runtime. The main issues of the speculative execution include how to determine a straggle and when to launch a speculation task. With poor design, the speculative technique may lead the datacenter to perform even worse. LATE [28] has studied the straggler problem and speculative execution technique in heterogeneous environments.

Execution isolation is another problem for a share datacenter, in which jobs are submitted by different departments. Even in a private cluster, where jobs are submitted by a single department, people want to make jobs isolated with each other. The main demand for this is for the resource competition and security reasons. While execution isolation is widely studied in traditional operating system fields, little work has been done on MapReduce runtime system.

4.2.2 Locality

One key design principle of MapReduce runtime system that should be emphasized is copying program toward data rather than moving data to where computation locates. In a large datacenter, moving program to data is cheaper than transferring massive data sets across the datacenter [1]. The data locality is benefited from the underlying distributed file system. A large file is partitioned sequentially into n blocks, and each block is stored in one machine. As Fig. 7 illustrated, each map task takes a file split as input. If the map task is collocated with its input split in the same machine, then the computation avoids unneeded copying operations across the datacenter. For the reason of load balance, not every task can be dispatched into where data locates, but it can be sent to machines with minimum distance from the source data. The distance between two machines can be defined as the least number of switches needed to walk through from one to

the other.

4.2.3 Partition function

Another important primitive we have not mentioned is the partition function. In the second MapReduce job of the page view example, we have implemented a simple range partition function. The partition function is called by map functions and controls the partitioning of the intermediate map outputs' keys. Each intermediate key-value pair emitted by the map function is allocated to a specific reducer by applying the partition function. The total number of partitions is the same as the number of reduce instances, and each partition of a map task's outputs is fed into a certain reduce task. Typically, MapReduce uses a hash of the key for partition function. The main difficulty partition implementation faces is how to uniformly distribute the intermediate keys or finding other ways to avoid load imbalance among reducers.

4.3 Performance study and limitations of MapReduce

We have experimented with the PV application both with the single machine sequential program and with the MapReduce program. In the experiment, the input size is 3.76 GB, and has about 35 million records. With the single machine implementation, the PV program finishes in about 100 seconds. We have experimented in the MapReduce approach in small clusters configured with one, two, four, and six slaves. While the MapReduce approach performs almost linearly scalable in clusters configured with one and two slaves, it does not perform better than the first approach. The performance penalty may be caused by copying data through MapReduce server processes, rather than reading the disk directly.

In MapReduce, each task is independent and has no knowledge of the others. Without message passing between tasks, it is difficult for collaborative computation to be expressed with MapReduce.

Another limitation of MapReduce is that it is especially fit for sequential read-only processing. We have implemented a hierarchical clustering algorithm with no special optimization. The performance is extremely poor, mainly because that the application needs to modify the data often, and the subsequent reads need the modified data.

5 Net programming: GSML

While MapReduce is an abstract programming model for writing programs for datacenters, GSML [16, 25], short for Grid Service Markup Language, raises abstract programming model into a higher level, which is intended for writing programs across datacenters. GSML is a programming language designed for integrating ready-to-use components together into a single business application, which can be run on grids and clouds.

GSML, based on a programming paradigm called BEAP [16], is composed of a resource part and a resource interaction part. With the resource part, GSML proposes a new resource primitive, funnel. Resource interaction part is defined as asynchronous occurrences of, and response to, events. In this section, we will first give a GSML example for demonstrating its functionalities and then overview the funnel part and interaction part.

First, we show an application example in Section 5.1 to demonstrate GSML's ease of use and efficiency. Then, for the expressiveness, the funnel and event set mechanisms are examined.

5.1 Povray-G example

Povray-G is a real application, which intends to speed up the rendering of complex three-dimensional graphics with computation powers from multiple heterogeneous grid sites. For a full comparison, we implement the Povray-G grid application both with traditional toolset plus grid middleware and with GSML. We use the povray tool program for implementing the rendering logic.

5.1.1 Traditional toolset plus middleware

The structure of the Povray-G implementation in a traditional way is shown in Fig. 10. User interacts with web User Interface (UI), and the application logic is running behind a web server. The web server transforms user's action and interacts with multiple heterogeneous grid sites.

We implement the web UI part with Hypertext Markup

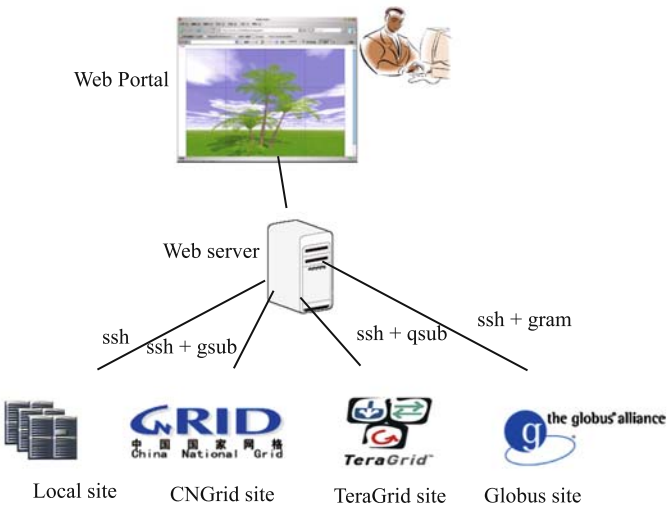

Fig. 10 Povray-G implementation without GSML

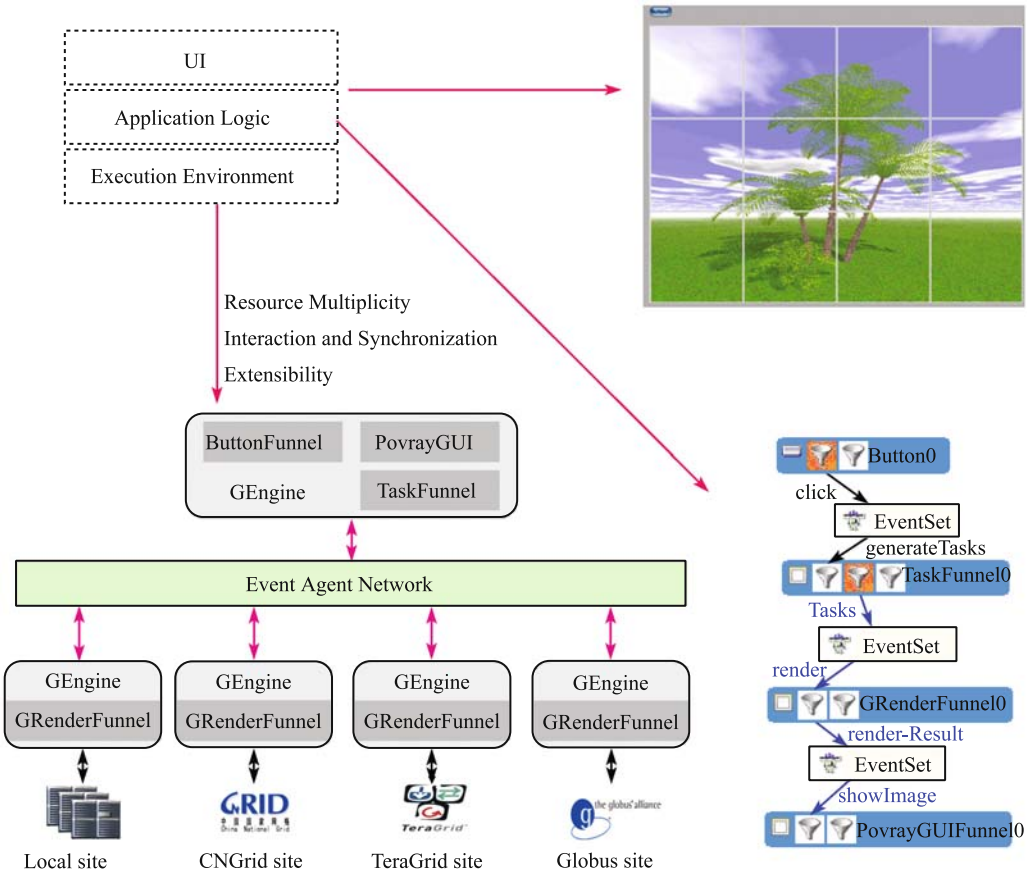

Fig. 11 Structure of Povray-G GSML implementation

Language (HTML) plus Cascading Style Sheet (CSS) and the application logic with Java Server Pages (JSP). The implementation includes web layout, interaction with interface modules, task split modules, and parallel task submission and execution module. Moreover, each subtask consists of uploading configuration files, splitting graphics, submitting rendering jobs, downloading results, and displaying them.

5.1.2 GSML approach

Structure of the Povray-G implementation with GSML is shown in Fig. 11. We implement the same UI as the previous traditional approach. To reduce the burden of programming application logic and UI, the whole program is decomposed into several basic functional components. These include ButtonFunnel, TaskFunnel, GRenderFunnel, and PovrayGUIFunnel. Basic funnels like ButtonFunnel can be shipped directly with the GSML workshop. Complex funnels, like GRenderFunnel and PovrayGUIFunnel, can be assembled quickly with basic funnels. We will give more detailed description of funnels in Section 5.2.

5.1.3 Comparison

The cost of implementing Povray-G in traditional approach is listed in Table 2. The cost listed does not include the gird sites' deploy time and the interface modules' implementation time. It costs about 615 person minutes and involves multiple languages and tools. In contrast to the traditional approach, the GSML approach only costs 68 person minutes for implemention, as shown in Table 3, and this can all be done with editing EXtensible Markup Language (XML) plain text. However, the cost of GSML approach does not include the cost of implementing the basic funnels. It is acceptable since basic funnels are common with all applications and shipped with the GSML workshop.

In terms of performance, we have experimented one rendering task five times with each implementation. GSML approach performs about 5.52% worse than traditional ap-

Tables 2 Cost of traditional Povray-G implementation

	lines of code	time (min)
user interface	74 (html, css, js)	155
business logic	121(Java, JSP)	45
others	133 (js, Java)	125
resource access	379(Java)	290
total	712	615

Tables 3 Cost of GSML implementation of Povray-G

	lines of code	time (min)
user interface	42 (XML)	25
business logic	366 (XML)	25
others	0	0
resource access	172 (XML)	18
total	580(XML)	68

proach. This is mainly because every rendering subtask involves more than 10 interactions. Each interaction includes event dispatches and responses through the event network, increasing the response time by the Round-Trip Time.

5.2 Resource virtualization and interaction

5.2.1 Funnel

In order to integrate resources sitting in distributed sites into a single program, there must be a mechanism to represent underlying heterogeneous resources and operate them in a uniform way. Funnel is introduced in GSML to address resource representation problem. Funnel is a high-level grid resource virtualization. It encapsulates the detailed resource implementation and interfaces and provides a uniform event accessing mechanism, which allows it to be invoked by events. Funnels can be written by composing other funnels [16, 25]. Once written, it can be reused anywhere anytime.

Each funnel can expose some states or properties, which can be edited to support funnel customization. In addition to the states or properties described above, a funnel is also defined with detailed event descriptions. Funnels interact with each other by emitting and responding to events. Each funnel is defined by a set of input events and output events. Input events are those that a funnel is interested in and may respond to. Output events are those a funnel can generate, which are used for invoking other funnels or notifying state changes. An output event is dependent on an input event if and only if this input event may cause the funnel to generate the output event.

5.2.2 Event set

With funnels, a business application can be built by specifying how funnels are connected [25]. Since funnels accept events and emit events, funnels can be connected into a graph, in a manner where some funnels' input events are some others' output events. While it seems that a whole program can be simply made by connecting funnels with events, two issues must be addressed, interface compatibility and synchronization [16]. To address these problems, GSML introduces event sets.

Since funnels can be developed independently with each other, it is difficult to connect them together. For example, a web service invocation funnel outputs XML data, while its subsequent funnel only accepts HTML document. Transformation is then needed to adapt the output of one funnel to the form its subsequent funnel accepts. Event set mechanism does this transformation responsibility with little user help [16, 25].

Sometimes synchronization among events is required by applications. As an example, consider the situation where money is to be transferred from account A to B. Before money is transferred from account A to account B, the validation of user information and checking of user balances must be synchronized. To this end, GSML introduces synchronization barriers in the event set mechanism. A synchronization barrier guaranties that some funnel cannot start until a set of events have occured.

6 Conclusion

This paper examines four representative styles of concurrent programming, for computations at accelerator, process, datacenter, and Internet scales. We use SIMT/CUDA, transactional memory, MapReduce, and GSML to illustrate these styles.

The four styles have different emphases and limitations on the application scope and the three issues of efficiency, expressiveness, and programmability.

At the Internet scale, GSML focuses on the mash up of Internet resources and emphasizes programmability. We give an example to show that GSML can significantly reduce Internet application development time from 615 person minutes to 68 person minutes.

At the datacenter scale, MapReduce is a functional programming style construct, which focuses on massive data processing using multiple servers within a datacenter. It emphasizes programmability and efficiency. We give a page view statistic example to show that MapReduce lowers the development cost from six person hours to two person hours, while achieving almost linearly performance scalability. We use another hierarchical clustering example to show that MapReduce performs poorly on applications with interleaving reads and writes.

At the process level, transactional memory is concerned with synchronization within one memory address space and intends to provide simple programmability by avoiding complex traditional primitives such as lock, semaphore, and conditional variable. Currently, there are open source implementations of software transactional memory, which perform 2–7 times worse than lock-based systems. Hardware transactional memory has research prototypes.

At the accelerator level, SIMT/CUDA focuses on small-

grain arithmetic-intensive computations with good locality. It emphasizes efficiency. We show by an example that using CUDA on a GPU can outperform a CPU by 20 times. Usually, accelerator scale programming needs to expose to the user many details of the special hardware. The SIMT model and the CUDA library carefully design a uniform interface to enhance programmability.

Acknowledgements This research was supported partly by the National Basic Research (973) Program of China (2005CB321807, 2005CB321600), the Hi-Tech Research and Development (863) Program of China (2006AA01A106, 2006AA01Z121, 2006AA04Z158), and National Natural Science Foundation of China(Grant No. 60603004).

References

1. Bryant R. Data-intensive supercomputing: the case for DISC. Technical Report CMU-CS-07-128, Carnegie Mellon University, 2007
2. Chaiken R, Jenkins B, Larson P, et al. SCOPE: easy and efficient parallel processing of massive data sets. In: International Conference of Very Large Data Bases (VLDB). VLDB Endowment, 2008, 1265–1276
3. Cooper B, Ramakrishnan R, et al. PNUTS: Yahoo!'s hosted data serving platform. In: International Conference of Very Large Data Bases (VLDB). VLDB Endowment, 2008, 1277–1278
4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation. USENIX Association, 2004, 137-150
5. Garland M, Grand S, Nickolls J, et al. Parallel computing experiences with CUDA. IEEE Micro, 2008, 28(4): 13–27
6. Guerraoui R, Herlihy M, Pochon B. Polymorphic contention management. In: Proceedings of the 19th International Symposium on Distributed Computing. New York: Springer Verlag, 2005, 303–323
7. Harris T, Fraser T. Language support for lightweight transactions. In: Proceedings of the 8th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. New York: ACM Press, 2003, 388–402
8. Harris T, Marlow S, Peyton-Jones S, et al. Composable memory transactions. In: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2005, 48–60
9. Herlihy M, Eliot J, Moss B. Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. New York: ACM Press, 1993, 289–300
10. Herlihy M, Luchangco V, Moir M, Scherer III W N. Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing. New York: ACM Press, 2003, 92–101
11. Herlihy M, Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008
12. Hwang K. Xu Z. Scalable parallel computing: technology, architecture, programming. McGraw-Hill Science/Engineering/Math, 1998
13. Khronos Group. The OpenCL specification, version 1.0. 12/08/2008
14. Larus J, Kozyrakis C. Transactional memory. New York: Communication of the ACM, 2008, 51(7): 80–88
15. Larus R, Rajwar R. Transactional memory. Morgan & Claypool, 2006
16. Liu X, Radenac Y, Ban atre J, et al. A chemical interpretation of GSML programs. In: 7th International Conference on Grid and Cooperative Computing(GCC2008). 2008, 459–466
17. Minh C, Trautmann M, Chung J, et al. An effective hybrid transactional memory system with strong isolation guarantees. In: Proceedings of the 34th International Symposium on Computer Architecture. New York: ACM Press, 2007, 69–80
18. NVIDIA. Corp. NVIDIA CUDA programming guide, version 2.0. 06/07/2008
19. Owens J, Luebke D, et al. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum. 2007, 26(1): 80–113
20. Patterson D. The data center is the computer. New York: Communication of the ACM, 2008, 51(1): 105–105
21. Ryoo S, Rodrigues C, Baghsorkhi S, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2008, 73–82
22. Saha B, Adl-Tabatabai R, Jacobson Q. Architectural support for software transactional memory. In: Proceedings of the 39th International Symposium on Microarchitecture. 2006, 185–196
23. Scherer III W N, Scott M L. Advanced contention management for dynamic software transactional memory. In: Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. New York: ACM Press, 2005, 240–248
24. Shavit N, Touitou D. Software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing. New York: ACM Press, 1995, 204–213
25. Shu C, Yu H, Liu H. Beap: an end-user agile programming paradigm for business applications. Journal of Computer Science and Technology, 2006, 21(4): 609–619
26. Tarditi D, Puri S, et al. Accelerator: using data parallelism to program GPUs for general-purpose uses. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. 2006, 325–335
27. Volkov V, Demmel J. Benchmarking GPUs to tune dense linear algebra. In: Proceedings of Conference on Supercomputing. IEEE Press, 2008
28. Zaharia M, Konwinski A, Joseph A, et al. Improving MapReduce performance in heterogeneous environments. In: 8th Symposium on Operating Systems Design and Implementation. USENIX Association, 2008