Revisit of View-Oriented Parallel Programming

Z. Huang
Department of Computer Science
University of Otago
Dunedin, New Zealand
Email:hzy@cs.otago.ac.nz

W. Chen
Department of Computer Science
Tsinghua University
Beijing, China
Email:cwg@tsinghua.edu.cn

Abstract

Traditional parallel programming styles have many problems which hinder the development of parallel applications. The message passing style can be too complex for many programmers. While shared memory based parallel programming is relatively easy, it requires programmers to guarantee there is no data race in programs by using mutually exclusive locks. Data race conditions are generally difficult to debug and difficult to prevent as well. The View-Oriented Parallel Programming (VOPP) is a novel shared-memory-based programming style. It removes the burden of guaranteeing data race free from the programmers. With the VOPP approach, shared data objects in a parallel program are divided into views according to the memory access pattern of the parallel algorithm. Data race is not an issue in VOPP, since mutual exclusion is automatically done by the underlying system when a view is accessed. The programmer only needs to synchronize the access of views using synchronization primitives like barriers. By removing data races of view access, VOPP makes it easier to code and less difficult to debug programs. It provides potential performance advantages on multi-core systems as well as cluster computers. It will also provide useful information for efficient implementation of transactional memory.

Key Words: View-Oriented Parallel Programming, Cluster computer, Multi-core system, Message Passing Interface, Parallel Computing, Data Race, Deadlock, Transactional Memory

1 Introduction

With the advent of Sun's UltraSPARC T1, we are moving towards a new age of parallel processing. UltraSPARC T1 (also known as Niagara) is a multi-core system which consists of 8 cores and can support running up to 32 parallel processes/threads. That means, inside the single chip, there can be 32 processes running in parallel. With conservative estimation, we expect hundreds of cores (processors) built into a single chip. Therefore, with multi-core technology, we effectively have a powerful parallel computer built inside one chip.

What will happen in the near future is that every desktop computer will be a parallel computer with lots of computing power. The problem for us is how to utilize the power. The task eventually falls on the shoulders of application programmers. The programmers should be able to make their applications to run in parallel on multi-core systems. Parallel programming on multi-core systems is important for utilizing the available computing power.

However, parallel programming is regarded as difficult and error prone. Sound parallel programming methodology is needed to ease the task of parallel programming. Traditionally, there are two camps in parallel programming methodologies. One is based on message passing such as Message Passing Interface (MPI), and the other is based on shared memory which is used for communications between computing entities such as processes. Parallel programming with message passing is notoriously difficult and complex, especially when there are hundreds of processes communicating with messages. Using shared memory for communications between processes is natural and straightforward for programmers, but the following problems hinder parallel programming using shared memory.

First, data race condition is difficult to debug. Data race means there are concurrent accesses to the same memory location and at least one of them is write access. If it happens in an execution of a parallel application, we say that the application has a data race condition and it may not be correctly executed due to the data race. Since a parallel execution is normally not repeatable, it is difficult to find the cause of a data race. Second, deadlock makes debugging more complex. Deadlock is a situation where multiple processes/threads wait for each other due to competing for locks. To prevent data race conditions, mutually exclusive primitives such as locks are used. Improper use of locks can result in deadlocks. Also mutual exclusion has complicated the mental model of parallel programming, since the programmer not only needs to consider what data to access, but also to consider how to access them (exclusively or non-exclusively). Third, parallel applications are normally not portable. There is no popular standard for shared memory based programming. Many languages and systems are proposed, such as OpenMP, HPF, Linda, Pthread, and etc. To have a standard API for shared memory based programming similar to MPI is very important for portability of parallel applications using shared memory. The API should be both efficient and convenient for parallel programming on different architectures and platforms such as SMPs, multi-core systems, and cluster computers.

This paper will address the above issues based on our previous novel View-Oriented Parallel Programming (VOPP). Compared to previous work [8], the views are further classified in this paper, the API for VOPP are refined and enhanced, and more examples are used in this paper to illustrate the versatile VOPP for a variety of applications. The rest of this paper is organised as follows. Section 2 describes the key concepts and primitives for VOPP. Section 3 illustrates VOPP with a few typical examples. Section 4 compares our work with other related work. Section 5 discusses the advantages of VOPP for parallel programming on various parallel computer architectures. Finally, our future work on VOPP is suggested in Section 6.

2 View-Oriented Parallel Programming (VOPP)

VOPP is a novel parallel programming style [6] based on the concept of view.

Definition 1 *Definition of View*

- A view is a set of memory units (bytes or pages) in shared memory. Suppose M is the set of total units in shared memory and V_i is a view, then $\forall V_i, V_i \subseteq M$.
- Views do not overlap with each other. Suppose there are two different views V_i and V_j, $i \neq j$, then $\forall V_i \forall V_j, V_i \cap V_j = \phi$

Definition 2 *Properties of View*

- Views are created and destroyed dynamically.
- Each view has a unique view identifier.
- Before a view is accessed (read or written), it must be acquired (with view primitives); after the access of a view, it must be released (with view primitives).
- Multiple views can be merged together when necessary.

So far we have identified the following classes of views.

Definition 3 *Classes of Views*

- Single-Writer view (SWV): only one process is allowed to acquire the view for write purpose at any particular time; multiple processes can acquire the view for read-only purpose at the same time. This class contains the following two special subclasses:
  - Consumable View (CV): This class of views is similar to a pipe. A writer (producer) produces the view while multiple readers (consumers) consumes the view. The producer and the consumers are synchronized with each other. Before a view is produced, the consumers should wait to consume the view; before the view is consumed by all consumers, the producer should wait to produce another version of the view.
  - Atomic View (AV): This class of views is accessed with atomic operations such as *read_view* and *write_view*.
- Multiple-Writer view (MWV): multiple processes can acquire the view for write purpose at the same time but they must write on different locations of the view.
- Automatically Detected View (ADV): The memory units of the view are not defined initially, but are automatically detected by the system along the course of parallel execution.

Definition 4 *Consistency of View*

- When a view is acquired by a process P_i, all previous *write* accesses to the view must *be performed with respect to* P_i according to their causal order.
- After a view is acquired by a process P_i, P_i's local copy of the view will not be affected by other processes until it acquires the view next time.

A write access to a unit of a view is said to *be performed with respect to* process P_i at a time point when a subsequent read access to that unit by P_i returns the value set by the write access. The above consistency is in conformity with the View-based Consistency (VC) proposed in [4].

There are a number of requirements for VOPP programmers.

- The programmer should use a number of views to store data objects according to the data sharing pattern of the parallel algorithm.
- Each view should consist of data objects that are always processed as an atomic set in the program.

- When any data object of a view is accessed, view primitives must be used (see below).

The following view primitives are needed to manipulate and access views:

- *int alloc_view(int *view_id, int size, int flag)*: create the specified view by allocating a memory space with *size* and specifying its class with *flag*. The return value indicates if there is any error or not. If *view_id* is null, the function will allocate an identifier to the view and return the identifier as its return value.
- *void *view_brk(int size)*: extend the view with size *size*, then return the pointer to the start of the extended area.
- *int free_view(int view_id)*: destroy the specified view and free the memory space of the view.
- *void *acquire_view(int view_id)*: acquire exclusive write access to the specified view; the calling process is blocked if the view is held by another process. The address of the view is returned if the acquire is successful.
- *void release_view(int view_id, int nr)*: release the specified view. If the view is consumable, the number of consumers the producer waits is *nr*.
- *void *acquire_Rview(int view_id)*: acquire read-only access to the specified view; the calling process gets an up-to-date version of the specified view. The address of the view is returned if the acquire is successful.
- *void release_Rview(int view_id)*: finish read-only access to the specified view.
- *int read_view(int view_id, void *buf, int *size, int offset)*: atomic read operation on views.
- *int write_view(int view_id, void *buf, int size, int offset)*: atomic write operation on views.
- *void enqueue_view(int view_id)*: enqueue a view identifier into the system queue.
- *int dequeue_view()*: dequeue a view identifier from the system queue.

VOPP allows programmers to participate in performance optimization through wise partitioning of shared data into views. The rule of thumb for VOPP overhead is that, the more view acquisitions, the more messages incurred in the network or the system bus; and the larger a view is, the more amount of data transmitted in the network at the view acquisition and the more coarse grain of parallelism there is possibly. The programmers will be able to finely tune the program by reducing both the number of view acquisitions and the size of views.

3 Examples of VOPP

In this section, we will use some examples to illustrate VOPP. These examples use a C interface provided by VODCA [8], a system supporting VOPP. VODCA provides the above view primitives with a prefix *Vdc_*, though current version of VODCA only implements a subset of those view primitives. In addition, VODCA also provides the following C interface.

- *VDC_NPROCS*: the maximum number of parallel processes supported by VODCA.
- *VDC_NVIEWS*: the number of view identifiers available for use.
- *VDC_NPAGES*: the number of pages in the shared memory.
- *Vdc_nprocs*: the actual number of parallel processes in an execution.
- *Vdc_proc_id*: the process id, an integer ranging from 0 to *Vdc_nprocs-1*.
- *void Vdc_startup(int argc, char **argv)*: initialise VODCA and start remote processes.
- *void Vdc_exit(int status)*: terminate the calling process.
- *void Vdc_barrier(unsigned id)*: block the calling process until every other process arrives at the barrier.

The following VOPP examples are based on typical problems in parallel programming.

3.1 Sum problem

In this problem, every process has its local array and needs to sum up all local arrays. We create a view for the subtotal of each local array in every process. Finally the master process (process 0) will sum up all subtotals.

```
int *local_array, a_size;
long *subtotal;

main(int argc, char **argv)
{
int i, j;
long sum=0;

initialise a_size;

Vdc_startup(argc, argv);

local_array=malloc(a_size*sizeof(int));
initialise local_array;
Vdc_alloc_view(&Vdc_proc_id, \
                 sizeof(long), SWV);

subtotal=Vdc_acquire_view(Vdc_proc_id);
*subtotal = 0;
for (i=0;i<a_size;i++)
   *subtotal+=local_array[i];
Vdc_release_view(Vdc_proc_id, 0);

Vdc_barrier(0);

if(Vdc_proc_id==0){
  for(j=0;j<Vdc_nprocs;j++){
     subtotal = Vdc_acquire_Rview(j);
     sum += *subtotal;
     Vdc_release_Rview(j);
  }
  printf("The total sum is %l\n", sum);
}

}
```

We may also use a multiple-writer view (MWV) to code the above problem in the following way. An array is allocated as a multiple-writer view and all processes simultaneously store their subtotals into their respective elements. Finally the master process (process 0) sums up all subtotals with the MWV.

```
if(Vdc_proc_id==0)
  Vdc_alloc_view(&Vdc_proc_id, \
       Vdc_nprocs*sizeof(long), MWV);

Vdc_barrier(0);

subtotal=Vdc_acquire_view(0);
subtotal[Vdc_proc_id] = 0;
for (i=0;i<a_size;i++)
  subtotal[Vdc_proc_id]+=local_array[i];
Vdc_release_view(0, 0);

Vdc_barrier(0);

if(Vdc_proc_id==0){
  subtotal = Vdc_acquire_Rview(0);
  for(j=0;j<Vdc_nprocs;j++)
     sum += subtotal[j];
  Vdc_release_Rview(0);
  printf("The total sum is %l\n", sum);
}
```

3.2 Producer/consumer problem

Suppose there is a piece of shared data produced by process 0 and consumed by *Vdc_nprocs* processes. We can use a consumable view to contain the shared data as below.

```
if(Vdc_proc_id==0)
  Vdc_alloc_view(&Vdc_proc_id, \
                size, SWV|CV);

Vdc_barrier(0);

while(condition){
  if(Vdc_proc_id==0){
    vp=Vdc_acquire_view(0);
    produce(vp);
    Vdc_release_view(0, Vdc_nprocs);
  }

  vp=Vdc_acquire_Rview(0);
  consume(vp);
  Vdc_release_Rview(0);
}
```

3.3 Task queue problem

Using task queue for parallel computing is common. The shared data in each task is regarded as a different view and a unique identifier is allocated to the view when the task is created. The VOPP code pieces of the program are as below.

```
struct task {
    char state;
    char *task_data;
}

/* task producer */
struct task *t;
int vid;

vid = Vdc_alloc_view(NULL, \
        sizeof(struct task), SWV);
t=Vdc_acquire_view(vid);
t->task_data=Vdc_view_brk(data_size);
create_task(t);
Vdc_release_view(vid);
Vdc_enqueue_view(vid);


/* task consumer */
struct task *t;
unsigned vid;
```

```
vid=Vdc_dequeue_view();
t=Vdc_acquire_view(vid);
consume_task(t);
Vdc_release_view(vid);
```

From the above examples, we see that VOPP does not place extra burden on programmers since the partitioning of shared data is an implicit task in parallel programming. VOPP just makes the task explicit by adding view primitives, which renders parallel programming less error-prone in handling shared data.

The focus of VOPP is shifted more towards data management (e.g. data partitioning and sharing), instead of mutual exclusion and data race as in traditional shared memory based parallel programming. Mutual exclusion is automatically achieved by VODCA system when a process acquires a normal view rather than MWV. Except multiple-writer views (MWVs) and automatically detected views (ADVs) which are for experienced parallel programmers, the memory area of a view is protected by the system so that no other processes will be able to ”touch” the area. In this way, the bug of ”data race” is removed.

To avoid deadlock, we are going to provide some wrapper functions to acquire multiple views together in the same order. The programmers can use these wrapper functions when they are not sure if there is a deadlock because of acquiring multiple views.

4 Comparison with other related work

The idea of combining data with mutual exclusion is not new. There were some related work such as Entry Consistency (EC) [2] and Scope Consistency (ScC) [9]. However, their programming interfaces are very different from VOPP.

VOPP is different from the programming style of Entry Consistency in terms of the association between data objects and views (or locks). Entry Consistency [2] requires the programmer to associate manually data objects with locks and barriers in programs, while the VOPP programmer simply creates views with *alloc_view* primitive, which is just as easy as memory allocation.

VOPP is also different from the programming style of Scope Consistency (ScC) [9] Programs based on ScC are extended from the traditional DSM programs, i.e., lock primitives are normally used in those programs while scope primitives such as *open_scope* are used only when required by memory consistency. Therefore, the programming model provided in ScC is a mixture. The programmer has to think of mutual exclusion when lock primitives are used, but has to think of memory consistency when scope primitives are used. This blended programming model simply confuses programmers. However, in contrast to the traditional DSM programs, the focus of VOPP is shifted towards shared data (views) rather than mutual exclusion. Programmers only think of shared data (views) when view primitives are used, while mutual exclusion and view consistency are left to the underlying system.

VOPP is very different from MPI. From programming point of view, VOPP is more convenient and easier for programmers than MPI, since VOPP is still based on the concept of shared memory (except that view primitives are used whenever shared memory is accessed). In addition, VOPP provides experienced programmers an opportunity to finely-tune the performance of their programs by carefully dividing the shared memory into views.

Since partitioning of shared data into views becomes part of the design of a parallel algorithm in VOPP, VOPP offers the potential to make VOPP programs perform as well as MPI programs. The reason is that a VOPP program can be finely tuned so that its underlying message passing behavior can match that of its MPI counterpart. That is, if there is a finely-tuned MPI program, we can make a VOPP program whose underlying message passing behavior is similar to that of the MPI program. The VOPP program can imitate the MPI program in a way that wherever there is data transfer between processors in the MPI program, the VOPP program allocates a shared view for the data and uses view acquisition to get the data. In this way, the overhead of message passing in VOPP can be almost the same as that in MPI program, since the cost of view acquisition in VODCA is almost the same as that of sending and receiving a block of data in MPI.

Though the message passing behavior of VOPP programs can be made similar to that of MPI programs, the programming interface provided in VOPP is very much different from MPI. MPI programmers have to know where a block of data is located, while location of a view is transparent to VOPP programmers. VOPP programmers only need to worry about which view to acquire, but not the location of data as in MPI programs.

5 Advantages of VOPP

VOPP is useful for a wide range of parallel programmers, from novice to veteran, from traditional shared memory programmers to MPI programmers. For novice programmers, a few single-writer views (SWVs) can be used for communications between processes. Though the performance may not be good initially, but at least the programs can be easily made working since the programmers don’t need to worry about the issues of mutual exclusion, data race, and deadlock. For veteran programmers, data objects can be carefully partitioned into more views in order to enable fine-grained parallelism.

For traditional shared memory programmers, a single multiple-writer view (MWV) can be used like the example in Section 3.1, as long as the programmers guarantee there is no data race when the view is accessed in parallel. If the programmers are clear about their data partitions, automatically detected views (ADVs) can be used to easily adapt the traditional programs (e.g. TreadMarks [1] programs) into VOPP programs.

For MPI programmers, a view can be regarded as a message buffer with an identifier. It has transparent location and is shared by all processes. It is identified by an identifier instead of location. This essential difference of a view from a message buffer has removed the mental burden of keeping track of locations of data among processes in MPI programs. Apart from this difference, MPI programmers can simply treat a view as a message and map any MPI program into a VOPP program.

The philosophy behind VOPP is that independence and isolation are better than sharing in parallel computing. We encourage more independence/isolation than sharing in VOPP. When there is a sharing, the programmer should be reminded of the cost. In VOPP, every time there is a sharing of data, a view has to be created by the programmer. Every time a shared data object is accessed, view primitives have to be used. In this way, sharing is discouraged and the programmer is reminded to carefully budget the amount of sharing. In the same line as the above philosophy, we prefer process to thread in implementation of VOPP since threads have lots of unnecessary sharing which expose programs to potential problems like data race and deadlock. By using processes in implementation of VOPP, we can reduce the sharing of data among processes to the minimal and the sharing of data in VOPP programs can only be achieved through views. The overhead difference between creating a process and creating a thread has significantly been reduced now with the techniques of light-weight process (LWP) and copy-on-write (COW). Overall, we believe the above philosophy will lead to an easy and enjoyable programming style which is acceptable to most parallel programmers.

VOPP has already demonstrated its performance advantage on cluster computers [6, 8, 5, 7, 4]. Moreover, VOPP has potential performance advantage on SMP or multi-core systems. Since memory address and size of a view is known, a view can be pre-fetched into CPU cache when a view is acquired. This information cannot be obtained in traditional parallel programs.

Also VOPP can be efficiently implemented based on Remote Direct Memory Access (RDMA) [10]. RDMA provides read and write services directly to applications and enables data to be transferred from memory of remote computers directly into user buffers without intermediate data copies. It also enables a kernel bypass implementation. Since views can be easily mapped to RDMA memory buffers, VOPP can be implemented efficiently based on RDMA and thus provides a convenient interface for applications on top of complicated RDMA interface.

Additionally VOPP will have performance advantage on systems with transactional memory. With transactional memory, mutual exclusive mechanisms such as locks are not needed when shared data are accessed. Access conflicts are detected and resolved by transactional memory. However, with shared memory partitioned into views, we can disable the conflict-resolving mechanism of the transactional memory if the same view is not acquired by two or more processors simultaneously. Since the conflict-resolving mechanism is costly, disabling the mechanism will produce performance advantage.

VOPP enables more effective debugging. Since shared data are divided into views, debugging VOPP programs based on views is very natural. Views can be monitored while a program is executed. Views are the only shared data among processes. They are easy to be tracked down with the view primitives. In this way, degugging a parallel program becomes simpler and more effective.

6 Future work

More applications are needed to evaluate the convenience and performance of VOPP. Techniques for fast barriers need to be investigated when the number of processors is very large, e.g., the order of hundreds or thousands. An implementation of VOPP based on RDMA needs to be investigated. In order to make an effective parallel programming environment for VOPP, a view-based debugger is needed in the near future.

References

[1] Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared memory computing on networks of workstations. IEEE Computer 29 (1996) 18–28

[2] Bershad, B.N., Zekauskas, M.J.: Midway: Shared memory parallel programming with Entry Consistency for distributed memory multiprocessors. CMU Technical Report (CMU-CS-91-170) Carnegie-Mellon University (1991)

[3] Gropp, W., Lusk, E., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22 (1996) 789–828

[4] Huang, Z., Purvis M., and Werstein P.: View-Oriented Parallel Programming and View-based Consistency.

In: Proc. of the Fifth International Conference on Parallel and Distributed Computing, Applications and Technologies (LNCS 3320) (2004) 505-518, Singapore.

[5] Huang Z., Purvis M., and Werstein P.: View Oriented Update Protocol with Integrated Diff for View-based Consistency. DSM Workshop 2005, In: Proc. of the IEEE/ACM Symposium on Cluster Computing and Grid 2005 (CCGrid05), IEEE Computer Society (2005)

[6] Huang Z., Purvis M., and Werstein P.: Performance Evaluation of View-Oriented Parallel Programming. In: Proc. of the 2005 International Conference on Parallel Processing (ICPP05), pp.251-258, IEEE Computer Society (2005)

[7] Huang Z., Purvis M., and Werstein P.: Performance Comparison between VOPP and MPI. In: Proc. of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies, pp.343-347, IEEE Computer Society (2005)

[8] Huang Z., Chen W., Purvis M., and Zheng W.: VODCA: View-Oriented, Distributed, Cluster-based Approach to parallel computing, DSM Workshop 2006, In: Proc. of the IEEE/ACM Symposium on Cluster Computing and Grid 2006 (CCGrid06), IEEE Computer Society (2006)

[9] Iftode, L., Singh, J.P., Li, K.: Scope Consistency: A bridge between Release Consistency and Entry Consistency. In: Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (1996)

[10] Recio R., Culley P., Garcia D., and Hilland J.: An RDMA Protocol Specification (Version 1.0). Technical report, RDMA Consortium, October 2002.