View-Oriented Parallel Programming and View-based Consistency

Z. Huang†, M. Purvis‡, and P. Werstein†

†Department of Computer Science
‡Department of Information Science
University of Otago
Dunedin, New Zealand
Email:hzy@cs.otago.ac.nz, mpurvis@infoscience.otago.ac.nz, werstein@cs.otago.ac.nz

Abstract. This paper proposes a novel View-Oriented Parallel Programming style for parallel programming on cluster computers. View-Oriented Parallel Programming is based on Distributed Shared Memory. It requires the programmer to divide the shared memory into views according to the nature of the parallel algorithm and its memory access pattern. The advantage of this programming style is that it can help the Distributed Shared Memory system optimise consistency maintenance. Also it allows the programmer to participate in performance optimization of a program through wise partitioning of the shared memory into views. The View-based Consistency model and its implementation, which supports View-Oriented Parallel Programming, is discussed as well in this paper. Finally some preliminary experimental results are shown to demonstrate the performance gain of View-Oriented Parallel Programming.

1 Introduction

A Distributed Shared Memory (DSM) system can provide application programmers the illusion of shared memory on top of message-passing distributed systems, which facilitates the task of parallel programming in distributed systems. Distributed Shared Memory has become an active area of research in parallel and distributed computing with the goals of making DSM systems more convenient to program and more efficient to implement [1–9].

The consistency model of a DSM system specifies ordering constraints on concurrent memory accesses by multiple processors, and hence has fundamental impact on DSM systems' programming convenience and implementation efficiency. The Sequential Consistency (SC) model [10] has been recognized as the most natural and user-friendly DSM consistency model. The SC model guarantees that *"the result of any execution is the same as if the operations of all the processors were executed in some (global) sequential order, and the operations of each individual processor appear in this sequence in the order specified by its (own) program"* [10](p690). This means that in an SC-based DSM system, memory accesses from different processors may be interleaved in any sequential

order that is consistent with each processor's order of memory accesses, and the orders of memory accesses observed by different processors are the same. One way to strictly implement the SC model is to ensure all memory modifications be totally ordered and memory modifications generated and executed at one processor be propagated to and executed in that order at other processors instantaneously. This implementation is correct but it suffers from serious performance problems [11].

In practice, not all parallel applications require each processor to see all memory modifications made by other processors, let alone to see them in order. Many parallel applications regulate their accesses to shared data by synchronization, so not all valid interleavings of their memory accesses are relevant to their real executions. Therefore, it is not necessary for the DSM system to force a processor to propagate **all** its modifications to **every** other processor (with a copy of the shared data) at **every** memory modification time. Under certain conditions, the DSM system can select the *time*, the *processor*, and the *data* for propagating shared memory modifications in order to improve the performance while still appearing to be sequentially consistent [12]. For example, consider a DSM system with four processors P_1, P_2, P_3, and P_4, where P_1, P_2, and P_3 share a data object x, and P_1 and P_4 share a data object y, as shown in Fig. 1. The data object v is shared among processors at a later time not shown in this scenario.

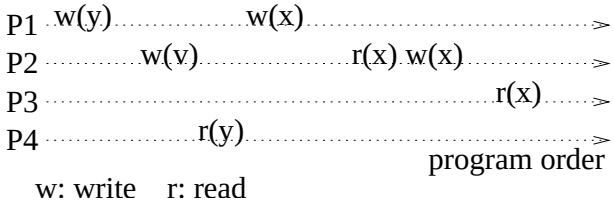

Fig. 1. A scenario of a DSM program

Suppose all memory accesses to shared data objects are serialized among competing processors by means of synchronization operations to avoid data races. Under these circumstances, the following three basic techniques can be used for optimisation of memory consistency maintenance [12].

- Time selection: Modifications on a shared data object by one processor are propagated to other processors *only at the time* when the data object is to be read by them. For example, modifications on x by P_1 may be propagated outward only at the time when either P_2 or P_3 is about to read x.
- Processor selection: Modifications on a shared data object are propagated from one processor to *only one other processor* which is the next one in sequence to read the shared data object. For example, modifications on x by P_1 may be propagated to P_2 (but not to P_3) if P_2 is the next one in sequence to read x.

– Data selection: Processors propagate to each other *only those shared data objects* that are really shared among them. For example, P_1, P_2, and P_3 may propagate to each other only data object x (not y and v), and P_1 and P_4 propagate to each other only data object y (not x).

To improve the performance of the strict SC model, a number of Relaxed Sequential Consistency (RSC) models have been proposed [13–17, 9], which perform one or more of the above three selection techniques. RSC models can be also called conditional Sequential Consistency models because they guarantee Sequential Consistency for some class of programs that satisfy the conditions imposed by the models. These models take advantage of the synchronizations in data race free (DRF) programs and relax the constraints on modification propagation and execution. That means modifications generated and executed by a processor may not be propagated to and executed at other processors immediately. Most RSC models can guarantee Sequential Consistency for DRF programs that are *properly labelled* [14] (i.e., explicit primitives, provided by the system, should be used for synchronization in the programs).

However, properly-labelled DRF programs do not facilitate data selection in consistency models. There has been some effort exploring data selection in consistency models. Examples are Entry Consistency (EC) [16], Scope Consistency (ScC) [17], and View-based Consistency (VC) [9]. Either they have to resort to extra annotations, or they cannot guarantee the SC correctness of some properly-labelled DRF programs. For example, EC requires data objects to be associated with locks and barriers and ScC requires extra scopes to be defined, while VC cannot guarantee the SC correctness of some properly-labelled DRF programs [18]. Those extra annotations are inconvenient and error-prone for programmers. To facilitate the implementation of data selection in consistency models with the SC correctness intact, we propose a novel parallel programming style for DSM, called View-Oriented Parallel Programming (VOPP). This programming style can facilitate data selection in consistency maintenance. Sequential Consistency can be guaranteed for the VOPP programs with the presence of data selection in consistency maintenance.

The rest of this paper is organised as follows. Section 2 presents the VOPP programming style and some program examples. Section 3 presents the VC model being associated with VOPP and its correctness. Section 4 discusses implementation issues of the VC model. Section 5 compares VOPP with related work. Section 6 presents and evaluates the preliminary performance results. Finally, our future work on VOPP is suggested in Section 7.

2 View-Oriented Parallel Programming (VOPP)

A *view* is a concept used to maintain consistency in distributed shared memory. A view consists of data objects that require consistency maintenance as a whole body. Views are defined implicitly by the programmer in his/her mind, but are explicitly indicated through primitives such as *acquire_view* and *release_view*. *Ac-*

quire_view means acquiring exclusive access to a view, while *release_view* means having finished the access.

The programmer should divide the shared memory into views according to the nature of the parallel algorithm and its memory access pattern. Views must not overlap each other. The views are decided in the programmer's mind and must be kept unchanged throughout the whole program. A view must be accessed by processors through using *acquire_view* and *release_view*, no matter if there is a data race or not in the parallel program. Before a processor accesses any objects in a view, *acquire_view* must be called; after it finishes operations on the view, *release_view* must be called. For example, suppose multiple processors share a variable A which alone is defined as a view, and every time a processor accesses the variable, it needs to increment it by one. The code in VOPP is as below.

```
acquire_view(1);
A = A + 1;
release_view(1);
```

A processor usually can only get exclusive write access to one view at a time in VOPP. However, VOPP allows a processor to get access to multiple views at the same time using nested primitives, provided there is at most one view to write (in order that the DSM system will be able to detect modifications for only one view). The primitives for acquiring read-only access to views are *acquire_Rview* and *release_Rview*. For example, suppose a processor needs to read arrays A and B, and puts their addition into C, and A, B and C are defined as different views numbered 1, 2, and 3 respectively, a VOPP program can be coded as below.

```
acquire_view(3);
acquire_Rview(2);
acquire_Rview(1);
C = A + B;
release_Rview(1);
release_Rview(2);
release_view(3);
```

To compare and contrast the normal DSM programs and VOPP programs, the following parallel sum problem is used, which is very typical in parallel programming. In this problem, every processor has its local array and needs to add it to a shared array. The shared array with size *a_size* is divided into *nprocs* views, where *nprocs* is the number of processors. Finally the master processor calculates the sum of the shared array. The normal DSM program is as below.

```
for (i = 0; i < nprocs; i++) {
  j=(i+proc_id)%nprocs*a_size/nprocs;
  k=((i+proc_id)%nprocs+1)*a_size/nprocs;
  for (;j < k;j++)
      shared_array[j] += local_array[j];
```

```
  barrier(0);
  }

if(proc_id==0){
   for (i = a_size-1; i > 0; i--)
      sum += shared_array[i];
   }
```

The VOPP program has the following code pattern.

```
for (i = 0; i < nprocs; i++) {
  j=(i+proc_id)%nprocs*a_size/nprocs;
  acquire_view((i + proc_id)%nprocs);
  k=((i+proc_id)%nprocs+1)*a_size/nprocs;
  for (;j < k;j++)
      shared_array[j] += local_array[j];
  release_view((i + proc_id)%nprocs);
  }

barrier(0);

if(proc_id==0){
  for(j=0;j<nprocs;j++)acquire_Rview(j);
  for (i = a_size-1; i > 0; i--)
     sum += shared_array[i];
  for(j=0;j<nprocs;j++)release_Rview(j);
  }
```

In the VOPP program, *acquire_view* and *release_view* primitives are added, while the normal DSM program only uses barriers. These primitives do not add much complexity to the VOPP program. On the contrary, they make the programmer feel more clear about which part of the shared array a processor needs to access. However, these primitives generate messages in DSM systems. The more primitives are used, the more messages have to be passed in DSM systems. By comparing the above two programs, it seems the VOPP program will generate more messages. But if we look more closely at the two programs, we can find the VOPP program has reduced the number of barriers since it uses view primitives to achieve exclusive access and thus does not need barriers in the first *for* loop. This advantage enables programmers to optimise VOPP programs by reducing barriers, since barriers tend to be more time-consuming than the view primitives (which will be demonstrated in our experimental results).

Read-only access to views can be explicitly declared with *acquire_Rview* and *release_Rview* in VOPP. Programmers can use them to improve the performance of VOPP programs, since multiple read-only accesses to the same view can be granted simultaneously, so that the waiting time for acquiring access to read-only views is very small. Programmers can use them to replace barriers and

read/write view primitives (*acquire_view* and *release_view*) wherever possible to optimise VOPP programs.

The VOPP style allows programmers to participate in performance optimization of programs through wise partitioning of shared objects into views and wise use of view primitives. VOPP does not place an extra burden on programmers since the partitioning of shared objects is an implicit task in parallel programming and VOPP just makes the task explicit. In this way, parallel programming is less error-prone in terms of handling shared objects.

More importantly, VOPP offers a huge potential for efficient implementations of DSM systems. When a view primitive such as *acquire_view* is called, only the data objects associated with the related view need to be updated. Therefore some optimal consistency maintenance protocol can be designed based on this simplicity, and data selection can be achieved straightforwardly.

VOPP requires that a view be defined initially and not changed throughout the program. In this way, it has placed some restriction on programming and thus has brought some inconvenience to the programmer. To offer some flexibility to the programmer, we provide some primitives such as *merge_views* to merge views into a global view as done in TreadMarks' barriers and/or to redefine views at some stage of a program. The price paid for this flexibility, of course, is the DSM efficiency.

To demonstrate more about the features of VOPP, we provide the following VOPP program for a task-queue based parallel algorithm. In the algorithm every processor can access the task queue to either enqueue a new task or dequeue a task. Before a processor enqueues a new task it generates a new view for the new task with *acquire_view*(-1) which will return a system-chosen view id. The VOPP code is as below.

```
V = acquire_view(-1);
create_task(T);
release_view(V);
T.view_id = V;
acquire_view(0);
enqueue(task_queue, T);
release_view(0);
```

When a processor dequeues a new task, the VOPP code is shown below. V and T are local variables, and T is a structure with a pointer element pointing to a shared task.

```
acquire_view(0);
dequeue(task_queue, T);
release_view(0);
V = T.view_id;
acquire_view(V);
consume_task(T);
release_view(V);
```

3 View-based Consistency

A processor will modify only one view between *acquire_view* and *release_view*, which should be guaranteed by the programmer. Therefore, we can detect modified data objects for each view in order to achieve view consistency.

Consistency maintenance in views requires updating data objects of a view when a processor calls *acquire_view*. More precisely, the following consistency condition is given for the View-based Consistency (VC) model that supports the VOPP programs. Any implementation of VC should satisfy the condition.

Definition 1. *Condition for View-based Consistency*

- Before a processor P_i is allowed to access a view by calling *acquire_view*, all previous *write* accesses to data objects of the view must *be performed with respect to* P_i according to their causal order.

A write access to a data object is said to *be performed with respect to* processor P_i at a time point when a subsequent read access to that object by P_i returns the value set by the write access.

In VOPP, barriers are only used for synchronisation and have nothing to do with consistency maintenance for DSM.

Since a processor will modify only one view between *acquire_view* and *release_view*, which should be guaranteed by the programmer, we can detect modified data objects for each view and use them later to maintain the consistency of the view.

SC correctness

Processors are synchronised to modify the same view, one after another, but may modify different views concurrently in any view-oriented parallel program. Based on this observation, for any parallel execution of a view-oriented parallel program, we can produce a global sequential order of the modifications on views. In this sequential order, the modifications on the same view are ordered in the same way as the synchronised order of the parallel execution, and the modifications on different views are put in program order if they are executed sequentially in the program; otherwise they are parallel and put in any order. Parallel modifications on different views can be executed in any order, which will not affect the execution result. Obviously, according to the consistency condition for VC, the parallel execution result of the program under VC is the same as the above sequential execution of the modifications. Therefore, a global sequential order has been found to match the parallel execution result under VC. According to the definition of the SC model, VC can guarantee Sequential Consistency for view-oriented parallel (VOPP) programs.

In this way, VC achieves time selection (at the time of calling *acquire_view*), processor selection (by passing an updated view to the next processor waiting to access the view), and data selection (by updating only the data objects of the view).

Any implementation of the VC model should conform with the above consistency condition. There are two important technical issues in the implementation: view detection and view consistency. View detection means identifying all the data objects (particularly modified objects) of a view. View consistency means updating all the modified data objects of a view before a processor gets (exclusive) access to the view using *acquire_view*.

Correctness and accuracy are two important issues in view detection. A correct view should include all data objects that are previously modified while the view is exclusively accessed. An accurate view should include those and only those data objects. The correctness of view detection must be satisfied in a VC implementation, while inaccuracy of view detection may only affect its performance (e.g., propagation of irrelevant modifications of data objects).

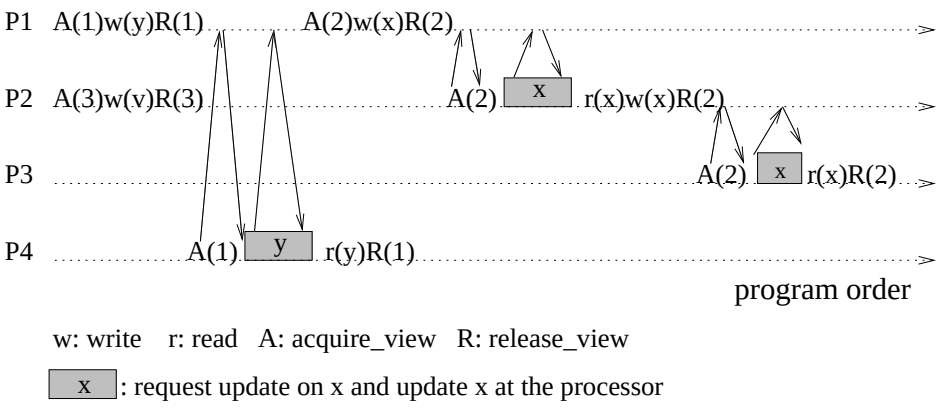

Fig. 2. View-based Consistency in action

Fig. 2 shows how the VC model works. In Fig. 2, there are two views numbered 1 and 2. View 1 includes y when P_4 enters the view, and view 2 includes x when P_2 and P_3 enter the view. When P_4 enters view 1, only the modification of y is propagated to the processor to update the view. For P_2 and P_3, only the modification on x is propagated to them to update view 2. The view acquisition and modification propagation are separate in the figure, but they can be combined in an implementation in order to improve DSM performance, i.e. the modifications can be piggy-backed on the *release_view* message.

In the following section, we will discuss our implementation of the VC model.

4 Implementation

We have implemented the VC model based on TreadMarks [7], which is a page-based DSM system. In TreadMarks, a *diff* is used to represent modifications on a page. Initially a page is write-protected. When a write-protected page is

first modified by a processor, a *twin* of the page is created and stored in the system space. When the modifications on the page are needed, a comparison of the *twin* and the current version of the page is done to create a *diff*, which can then be used to update copies of the page in other processors. We use this same mechanism in our view detection. We associate all modifications (in the form of diffs) made between *acquire_view* and *release_view* with the related view.

We have also implemented the VC model based on the home-based protocol [19], in which every page has a home node (processor) and the diffs of a page are passed to its home to maintain an up-to-date copy (called home page), and thus to update a page is to fetch the home page from the home node. This home-based implementation uses pages to represent data objects, which is a coarse-grained solution to data selection.

4.1 View detection

In consistency maintenance we only need to know which are the modified data objects and then update them. Likewise, to maintain the consistency of a view, we only need to update the modified data objects in the view. Therefore, we are not interested in the unchanged pages of a view and thus only the modified parts of pages (in the form of diffs) are recorded for the related view in our implementation of view detection.

In our implementation, view detection is achieved at run time. We use the concept of interval in TreadMarks to represent view modifications. An interval is a data structure which represents the modifications (e.g. diffs) made on a number of pages between *acquire_view* and *release_view*. We make an interval whenever a processor finishes updating a view by calling *release_view*. Note that when an interval is created, we make the diffs of related pages immediately. The diffs can be piggy-backed on the *release_view* message when another processor acquires access to the view, which is another potential optimisation for our VC implementation. When we implement VC with the home-based protocol, the diffs are passed to respective home nodes when an interval is created.

When an interval is created, it is associated with the related view whose ID number is the argument of the *release_view* causing the creation of the interval. This interval represents a modification of data objects in the view and will be used to update the view later when a processor accesses the view with *acquire_view*. Once an interval is created, the related view's version number is increased by one.

4.2 View consistency

To achieve view consistency is to make a view up to date. When a view is to be accessed by a processor calling *acquire_view*, view consistency must be achieved for the view. Write notices are created according to the intervals associated with the related view and the current version number of the view of the calling processor. Those write notices are passed to the calling processor to invalidate the related pages. When an invalid page is accessed, a page fault will cause

the processor to fetch the diffs (or to fetch the home page in the home-based consistency protocol). In this way view consistency is achieved.

View consistency is an area where we can improve the performance of the DSM system based on our VC model. In TreadMarks diffs are used to update pages, while home pages are used to update pages in the home-based protocol. In our experiments we found, when diffs are accumulating, more data and messages are passed through the network in TreadMarks than in the home-based protocol, since the cost of updating a page in the home-based protocol is constant and involves only one transmission of a page. Therefore, if we can merge accumulating diffs, transmission of diffs can be reduced.

5 Comparison with related work

VOPP is different from the programming style of Entry Consistency in terms of the association between data objects and views (or locks). Entry Consistency [16] requires the programmer to explicitly associate data objects with locks and barriers in programs, while VOPP only requires the programmer to implicitly associate data objects with views (in the programmer's mind). The actual association is achieved in view detection in the implementation of the VC model. VOPP is more flexible than the programming style of Entry Consistency.

VOPP is also different from the programming style of Scope Consistency in terms of the definition of view and scope. Views in VOPP are non-overlapped and constant throughout a program, while scopes in ScC can be overlapped and are merged into a global scope at barriers.

VOPP is more convenient and easier for programmers than the message-passing programming style such as MPI or PVM, since it is still based on the concept of shared memory (except the shared memory is divided into multiple non-overlapped views). Moreover, VOPP provides experienced programmers an approach to fine-tune the performance of their programs by carefully dividing the shared memory into views. Partitioning of shared memory into views becomes part of the design of a parallel algorithm in VOPP. This approach offers the potential for programmers to make VOPP programs perform as well as MPI programs.

6 Preliminary experimental results

In this section, we present our preliminary experimental results based on two (currently sub-optimal) versions of the VC implementation. The first VC version (called VC_d) uses diffs to update views, while the second one (called VC_h) uses the home-based protocol to achieve view consistency. Their performance is compared with that of the LRC (Lazy Release Consistency) model implemented in TreadMarks [7]. We have tested our implementations on two clusters. One cluster, called Vodca, consists of 8 PCs running Linux 2.4, which are connected by a 100 Mbps Ethernet hub. Each of the PCs has a 800 MHz processor and 128 Mbytes of memory. The other cluster, called Godzilla, consists of 32 PCs

running Linux 2.4, which are connected by a N-way 100 Mbps Ethernet switch. Each of the PCs has a 350 MHz processor and 192 Mbytes of memory. The page size of the virtual memory for both clusters is 4 KB.

We chose one application, Integer Sort (IS), in the experiment because of the limited resources available for converting the applications to VOPP programs. *IS* (Integer Sort) is a benchmark application provided by TreadMarks research group. It ranks an unsorted sequence of N keys. The rank of a key in a sequence is the index value i that the key would have if the sequence of keys were sorted. All the keys are integers in the range [0, B_{max}] and the method used is bucket sort. The problem size in our experiment is ($2^{25} \times 2^{15}$, 40). The memory access pattern is very similar to the pattern of our sum example in Section 2. *IS* is an application that cannot demonstrate the performance advantage of our VC model since it does not need locks for synchronisation, but we will see how the VOPP program itself can improve its performance.

VC vs. *LRC* vs. VC_{vopp}

Table 1 shows the running time (in sec.) of IS on LRC and VC. VC_d is the VC implementation using diffs to update views, which is the same as the LRC implementation in TreadMarks. $VC_{d_{vopp}}$ shows the running time of an optimised VOPP program of *IS* which uses less barriers. The table shows the running time on the cluster Godzilla for 2 nodes, 4 nodes, 8 nodes and 16 nodes. The running time on one node is not displayed because it is abnormally large (around 4000 seconds) due to excessive memory usage on one node.

Table 1 shows that VC is generally more efficient than LRC, with a 21% performance gain on 16 nodes. The optimised VOPP program runs generally faster than the original one, with 8.5% performance gain on 16 nodes.

	2-node	4-node	8-node	16-node
LRC	143.8	78.4	56.7	68.7
VC_d	144.6	77.5	51.5	54.3
$VC_{d_{vopp}}$	144.3	77.4	49.8	49.7

Table 1. Running time of IS on LRC and VC

Table 2 shows the detailed statistics of IS on 16 nodes. In the table, Ba is the number of barriers called in the program, Ac is the number of lock/view acquire messages, Da is the total amount of data transmitted, Msg is the number of messages, and Rxm is the number of messages retransmitted. From the statistics we find the number of messages and the amount of data transmitted in VC are more than in LRC, which may not be the case in many other applications. Then why is the VC implementation faster than LRC? The reason is

two-fold. The barriers in LRC need to maintain consistency while those ones in VC do not. The consistency maintenance in barriers in LRC is normally time-consuming and centralised at one processor which can be a bottleneck, while consistency maintenance in VC is distributed among the processors through the view primitives.

	Ba	Ac	Da	Msg	Rxm
LRC	682	1	1.23G	123941	118
VC_d	682	20481	1.27G	180212	2
$VC_{d_{vopp}}$	122	20481	1.27G	163418	12

Table 2. Statistics of IS on 16 nodes of Godzilla

In addition, LRC has more message loss than VC according to the statistics. On 16 nodes of Godzilla, LRC has 118 retransmissions while VC only has 2 and 12 retransmissions. On 8 nodes of Vodca as shown in Table 3, the number of retransmissions in LRC is as high as 782. Nodes in Vodca are connected by a hub, so the bursty traffic at barriers causes more message loss. One message retransmission results in about 1 second waiting time. Therefore, the distribution of data traffic in VC can reduce message retransmissions and improve the performance of applications.

	Time	Ba	Ac	Da	Msg	Rxm
LRC	324.7	362	1	285M	44288	782
VC_d	256.8	122	5121	297M	53255	410

Table 3. Statistics of IS on 8 nodes of Vodca

Home-based vs. diff-based page consistency

Table 4 shows the running time of IS on the two different VC implementations: VC_d and VC_h. The performance of VC_h is significantly better than that of VC_d when the number of nodes becomes larger. On 16 nodes of Godzilla, VC_h performs 40.9% faster than VC_d!

The data (mainly diffs) transmitted in VC_d is significantly increased when the number of nodes is large. This diff accumulation problem affects the performance of diff-based implementations. The home-based implementation VC_h can resolve diff accumulation by using home pages and thus reduce the amount of data

	2-node	4-node	8-node	16-node
VC_d	144.6	77.5	51.5	54.3
VC_h	143.6	75.5	43.5	32.1

Table 4. Running time of IS on home-based and diff-based protocols

transmitted, as demonstrated in Table 5. The amount of data reduced in VC_h is 81.5% compared with VC_d.

	Time	Ba	Ac	Da	Msg	Rxm
VC_d	32.1	682	20481	1.27G	180212	2
VC_h	54.3	682	20481	0.235G	170727	0

Table 5. Statistics of IS on 16 nodes of Godzilla

From the above results we realize that VC can be further improved if accumulated diffs can be integrated into a single diff. The single diff should integrate all previous modifications just as the home page does in home-based protocol. Also we can piggy-back those diffs associated with a view on the *release_view* messages. This improvement is one of our objectives for the near future.

7 Conclusions

We have proposed a novel VOPP programming style for DSM parallel programs on cluster computers. Our preliminary results have demonstrated the performance advantage of VOPP. We will use more applications to demonstrate its performance gain and its programming features. We will also investigate efficient implementation techniques of the associated VC model, such as an update protocol based on diff integration in order to integrate accumulated diffs for the same view. Our ultimate goal is to make shared memory parallel programs as efficient as message-passing parallel programs such as MPI programs.

References

1. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Trans. on Computer Systems **7** (1989) 321–359
2. Fleisch, B., Katz, R.H.: Mirage: A coherent distributed shared memory design. In: Proc. of the 12th ACM Symposium on Operating Systems (1989) 211–223
3. Lenoski, D. et al: The Stanford DASH multiprocessor. IEEE Computer **25** (1992) 63–79

4. Dasgupta, P. et al: The design and implementation of the Clouds distributed operating system. Computing Systems Journal **3** (1990)
5. Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Implementation and performance of Munin. In: Proceedings of the 13th ACM Symposium on Operating Systems Principles (1991) 152–164
6. Bershad, B.N., Zekauskas, M.J., Sawsonm W.A.: The Midway distributed shared memory system. In: Proc. of IEEE COMPCON Conference, (1998) 528–537
7. 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
8. Huang, Z., Lei,W.-J., Sun, C., Sattar, A.: Heuristic diff acquiring in Lazy Release Consistency model. In: Proc. of 1997 Asian Computing Science Conference (ASIAN'97) (1997) 98–109
9. Huang, Z., Sun, C., Purvis, M., Cranefield, S.: View-based Consistency and its implementation. In: Proc. of the First IEEE/ACM Symposium on Cluster Computing and the Grid (2001) 74–81
10. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **28** (1979) 690–691
11. Tanenbaum, A.S.: Distributed Operating Systems. Prentice Hall (1995)
12. Sun, C., Huang, Z., Lei, W.-J., Sattar, A.: Towards transparent selective sequential consistency in distributed shared memory systems. In: Proc. of the 18th IEEE International Conference on Distributed Computing Systems, Amsterdam (1998) 572–581
13. Dubois, M., Scheurich, C., Briggs, F.A.: Memory access buffering in multiprocessors. In: Proc. of the 13th Annual International Symposium on Computer Architecture (1986) 434–442
14. Gharachorloo, K., Lenoski, D., Laudon, J.: Memory consistency and event ordering in scalable shared memory multiprocessors. In: Proc. of the 17th Annual International Symposium on Computer Architecture (1990) 15–26
15. Keleher, P.: Lazy Release Consistency for distributed shared memory. Ph.D. Thesis (Rice Univ) (1995)
16. 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)
17. 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)
18. Huang, Z., Sun, C., Cranefield, S., Purvis, M.: A View-based Consistency model based on transparent data selection in distributed shared memory. Technical Report (OUCS-2004-03) Dept of Computer Science, Univ. of Otago, (2004) (http://www.cs.otago.ac.nz/research/techreports.html)
19. Zhou, Y., Iftode, L., Li, K.: Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In Proc. of the Operating Systems Design and Implementation Symposium (1996) 75–88