View-Oriented Update Protocol with Integrated Diff for View-based Consistency

Z. Huang†, M. Purvis‡, 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 View-Oriented Update Protocol with Integrated Diff for efficient implementation of a View-based Consistency model which supports a novel View-Oriented Parallel Programming style based on Distributed Shared Memory. View-Oriented Parallel Programming requires the programmer to divide the shared data into views according to the nature of the parallel algorithm and its memory access pattern. The advantage of this programming style is that it offers the potential for the underlying Distributed Shared Memory system to optimize consistency maintenance. The View-Oriented Update Protocol with Integrated Diff is proposed to exploit this performance potential. This protocol is compared with a traditional diff-based protocol and an existing home-based protocol. Experimental results demonstrate the performance of the proposed protocol is significantly better than the diff-based protocol and the home-based protocol.

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. However, programs using DSM are normally not as efficient as those using the Message Passing Interface (MPI) [11, 4]. The reason is that message passing is part of the design of a MPI program and the programmer can finely tune the performance of the program by reducing the unnecessary message passing. As we know, message passing is a significant cost for applications running on distributed systems, which is also true for DSM programs. Since consistency maintenance for DSM deals with the consistency of the whole shared memory space [5], there are many unnecessary messages passed in DSM programs compared with MPI programs. In addition, the programmer cannot help reduce those messages when designing the DSM programs.

Traditionally DSM programs are required to be data race free (DRF) using system provided synchronization primitives such as *lock_acquire*, *lock_release*, and *barrier*. If a DSM program has no data race through using those primitives, it is called a *properly-labelled* program [3]. However, properly-labelled DRF programs do not facilitate optimization such as data selection [10] in consistency maintenance in DSM. Since DRF oriented programming focuses on mutual exclusion and synchronization rather than data allocation, there is no opportunity in those programs for expert programmers to interact with the DSM system in terms of performance tuning. As a matter of fact, it is the optimal data allocation which can improve the performance of DSM applications.

To help DSM optimize its performance as well as to allow programmers to participate in performance tuning of DSM programs, we have proposed a novel View Oriented Parallel Programming (VOPP) style [6] for DSM applications. The VOPP programming style allows programmers to participate in performance optimization of programs through wise partitioning of shared data objects into views. The focus of VOPP is shifted more towards shared data (i.e. data partitioning and allocation), rather than synchronization and mutual exclusion. A View-based Consistency (VC) model [6] is proposed to maintain the consistency of views in VOPP programs. From our experience the VOPP programs are normally more efficient than the traditional DSM programs [7]. However, compared with MPI programs VOPP programs are still slower. To make VOPP programs run as efficiently as their MPI counterparts we propose a View-Oriented Update Protocol with Integrated

Regular paper submitted to DSM05. Please contact Zhiyi Huang for correspondence: email hzy@cs.otago.ac.nz; fax +64-3-4798529.

Diff (VOUPID) in this paper, which can optimally implement the consistency maintenance of the VC model.

The rest of this paper is organised as follows. Section 2 describes the VOPP programming style and the VC model. Section 3 presents the View-Oriented Update Protocol with Integrated Diff (VOUPID) for efficient consistency maintenance of the VC model. Section 4 compares the VOUPID protocol with related work. Section 5 presents and evaluates the performance of the VOUPID protocol based on several applications. Finally, our future work is suggested in Section 6.

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 or algorithm, but are explicitly indicated through primitives such as *acquire_view* and *release_view*. *Acquire_view* means acquiring (maybe exclusive) access to a view, while *release_view* means having finished the access. By using these primitives, the focus of the programming is on accessing shared objects (views) rather than synchronization and mutual exclusion.

The programmer should divide the shared data 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 or algorithm. Once defined initially, they must be kept unchanged throughout the whole program. The view primitives must be used when a view is accessed, no matter if there is any data race or not in the parallel program.

Before a processor accesses any data objects in a view, *acquire_view* must be called; after it finishes the access to the view, *release_view* must be called. For example, suppose multiple processors share a variable A which alone is defined as a view (which is numbered as view 1). 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);
```

For the situation of read-only access, the view primitives *acquire_Rview* and *release_Rview* are provided. *Acquire_Rview* requests read-only access to a view.

A processor can only write one view at a time in VOPP (in order that the DSM system will be able to detect modifications for only one view), but it can read multiple views at the same time. That is, *acquire_view*s cannot be nested but *acquire_Rview*s can be nested. A processor can read multiple views at the same time by using nested *acquire_Rview* primitives. For example, suppose a processor needs to read arrays A and B, and puts their additions into array 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);
for(i=0; i<a_size; i++)
        C[i] = A[i] + B[i];
release_Rview(1);
release_Rview(2);
release_view(3);
```

To compare and contrast traditional 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. In each outer loop, every processor adds an arranged portion of its local array into the corresponding location of the shared array in parallel with other processors. The processors are synchronized by a barrier after each outer loop. Finally the master processor (processor 0) calculates the sum of the shared array, which equals to the sum of all local arrays. The traditional DSM program is similar to the code below.

```
for (i = 0; i < nprocs; i++) {
  s=(i+proc_id)%nprocs*a_size/nprocs;
  e=((i+proc_id)%nprocs+1)*a_size/nprocs;
  for (j=s;j < e;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];
   }
```

For the same problem, VOPP style offers the following code pattern.

```
for (i = 0; i < nprocs; i++) {
  s=(i+proc_id)%nprocs*a_size/nprocs;
  e=((i+proc_id)%nprocs+1)*a_size/nprocs;

  acquire_view((i + proc_id)%nprocs);
  for (j=s;j < e;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, the shared array with size a_size is partitioned into $nprocs$ views, where $nprocs$ is the number of processors. Similar to the traditional DSM program, every processor adds an arranged portion of its local array into the right view of the shared array in parallel with other processors in every outer loop. The primitives $acquire\_view$ and $release\_view$ are added into the code to get access to the views. Finally processor 0 reads all $nprocs$ views with $acquire\_Rview$ and $release\_Rview$ to calculate the sum.

Inserting the view primitives is not an extra burden to the programmer; 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 in the VOPP program the barrier is called outside the outer *for* loop and the number of barriers is effectively reduced. The reason is that the barrier is originally used for mutual exclusion between loops but is not needed in the VOPP program because view primitives automatically achieve the exclusive access to views. This advantage enables programmers to optimise VOPP programs by reducing barriers, since barriers tend to be more time-consuming than the view primitives, which was demonstrated in our experimental results [6, 7]

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. The task queue is defined as view 0, and each task is defined as a separate view. 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 identifier. Below is the VOPP code.

```
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);
```

In a VOPP program, there is no global view that includes every data object in the shared memory. Barriers in VOPP are only used for synchronisation but have nothing to do with consistency maintenance for DSM. In traditional DSM programs, every processor can have a global view of the shared memory after each barrier. To keep this convenience, we provide a primitive *merge_views* in VOPP to merge views into a global view, so the programmer will be able to redefine the views after *merge_views*. The price paid for this convenience, of course, is the DSM efficiency.

In summary, VOPP has the following features:

- The VOPP style allows programmers to participate in performance optimization of programs through wise partitioning of shared objects (i.e. data allocation) into views and wise use of view primitives. The focus of VOPP is shifted more towards shared data (e.g. data partitioning and allocation), rather than synchronization and mutual exclusion.
- VOPP does not place any extra burden on programmers since the partitioning of shared objects is an implicit task in parallel programming. VOPP just makes the task explicit, which renders parallel programming less error-prone in handling shared data.
- VOPP offers a large 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. An optimal consistency maintenance protocol is going to be proposed in this paper based on this simplicity.

To maintain the consistency of views in VOPP programs, a View-based Consistency (VC) model has been proposed [6, 5]. In the VC model, a view is updated when a processor calls *acquire_view* or *acquire_Rview* to access the view. Since a processor will modify only one view between *acquire_view* and *release_view*, which should be guaranteed by the programmer, we are certain that the data objects modified between *acquire_view* and *release_view* belong to that view and thus we only update those data objects when

the view is accessed later. More formally, the consistency condition for the VC model is stated below.

Definition 1 *Consistency Condition for View-based Consistency*

- Before a processor P_i is allowed to access a view by calling *acquire_view* or *acquire_Rview*, 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.

From the above condition we know, in VOPP programs barriers are only used for synchronisation and have nothing to do with consistency maintenance for DSM. When a view is acquired, consistency maintenance is restricted to the view. In this way the amount of data traffic for DSM consistency in the cluster network can be reduced and the VC model can be implemented optimally as what will be proposed in Section 3. The Sequential Consistency (SC) [9] of VOPP programs can also be guaranteed by the VC model, which has been proved in [6].

3 View-Oriented Update Protocol with Integrated Diff

In View-based Consistency, when a view is acquired we only update the view with previous modifications made on the view. A version number is maintained for each view so that when a view is acquired by a processor we can decide if the view in the processor should be updated or not according to the version of the view of the processor and the latest version of the view. The last processor that releases a view should always have a copy of the latest view. If a view is modified by a processor the latest version number of the view is increased by one. In this section we propose a View-Oriented Update Protocol with Integrated Diff (VOUPID) to efficiently update a view of a processor when the version number of the view of the processor is smaller than the latest version number of the view.

3.1 Diff accumulating problem

In TreadMarks [1] a multiple writer protocol [2] is used to implement the DSM consistency of the LRC model [8]. In the protocol, *diffs* are 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 page fault occurs. Then the page fault handler creates and stores a *twin* of the page and makes the page both readable and writable. When the modifications on the page are later needed by another processor, the current version of the page is compared with the twin in order to create a *diff*, which can then be used to update the copies of the page in other processors. Based on the diff scheme, multiple processors can write on different parts of the same page concurrently and consistency of the page can be maintained by applying the corresponding diffs.

Our VC model can be implemented based on the above diff-based scheme. When a view is released, diffs are created for all pages modified between the $acquire\_view$ and the $release\_view$. When a view is acquired, pages are invalidated according to the consistency information (i.e. write notices as in TreadMarks). When an invalidated page is accessed later, a page fault occurs. The page fault handler will request the corresponding diffs in order to make the page up-to-date.

However, there is a diff accumulation problem in the above diff-based protocol. Along the course of execution of a DSM program, diffs can be accumulating and occupying lots of memory space and CPU time. In order to update a copy of a page in a processor, numerous diffs generated by other processors have to be passed to the processor and then applied to the copy of the page. To explain the problem, Figure 1 shows the execution of a typical VOPP program.

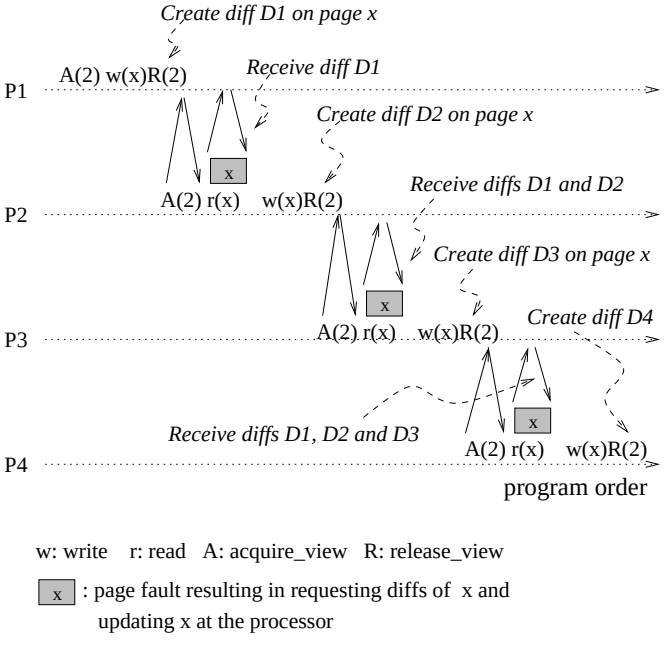

Figure 1: Diff accumulation problem

In Figure 1 each processor accesses page x in turn by acquiring view 2. Every time the view is released by a processor, a new diff is created for the modifications done by the processor. Every time the view is acquired, consistency information (such as write notices) is piggy-backed on the

view granting message and page x which is previously modified by other processors is invalidated. When page x is accessed, a page fault occurs which results in requesting the diffs of page x. When the diffs are received by a processor, they are applied to the copy of page x in the processor. A processor has to get all diffs of page x previously created by other processors in order to make the page up-to-date. In Figure 1 when the page fault on page x occurs P_4 receives the diffs created by P_1, P_2 and P_3 and applies them one by one to the page. If the number of processors increases in the figure, the number of diffs created for page x will be accumulating proportionally. If there are more pages modified in the figure, the diff accumulating problem will be more severe and the number of page faults will increase proportionally.

To make it even worse, if a page is widely modified a diff of the page is almost the same size as the page. When the amount of diffs is large, more messages have to be used to transfer them since the maximum transfer unit of messages is limited. Therefore, when the diffs are accumulating the number of messages and the amount of data traffic in the cluster network increase significantly. Many applications have demonstrated this problem in our experiments.

3.2 Diff merging algorithm

The idea of the VOUPID protocol is to integrate all the diffs of each page into a single diff and then update the page with the single integrated diff. The diffs of a page are merged based on the diff format.

A diff is compressed using run-length encoding. It consists of independent items each of which represents a range of continuous bytes in a page. Each item has the format *<length, offset, byte, byte,...>*, where *length* is the number of bytes in the range, *offset* tells from where to apply the following bytes in the page. When a diff is applied to a page, the bytes are simply copied to overwrite the corresponding bytes in the page.

According to the above diff format, we propose a diff merging algorithm to merge two diffs into one. Suppose there are two diffs D_1 and D_2 for the same page, where D_2 is more recently created. The diff merging algorithm can merge them into a new diff D_3. In the algorithm an item (range of continuous bytes) is removed sequentially from each of D_1 and D_2. The two items are compared and merged together if their byte ranges overlap each other. Assume the ranges of two items $< L_1, O_1, byte, byte, ... >$ and $< L_2, O_2, byte, byte, ... >$ (from D_1 and D_2 respectively) overlap each other. To merge the two items together, a new item $< L_3, O_3, byte, byte, ... >$ is created, where O_3 is the minimum of O_1 and O_2, L_3 is $L_1 + L_2$ minus the length of the overlapped part, the bytes are copied accordingly from the two items for the non-overlapped parts, but the bytes at the overlapped part are copied from the corresponding part in $< L_2, O_2, byte, byte, ... >$. For example, suppose there are two items $< 8, 8, 1, 2, 3, 4, 5, 6, 7, 8 >$ and $< 8, 12, 9, 10, 11, 12, 13, 14, 15, 16 >$ from D_1 and D_2 respectively. After the merging, the new item is $< 12, 8, 1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 15, 16 >$.

The diff merging algorithm is described as below. In the algorithm D_1 and D_2 (where D_2 is more recently created) are the input, and D_3 is the output. I_1, I_2 and I_3 are variables for the items in the diffs, where I_1 is initialized as the first item in D_1 and I_2 is initialized as the first item in D_2. The algorithm repeatedly executes the following steps until all items in D_1 and D_2 are processed.

1. If the range of I_1 does not overlap with the range of I_2, then
   - if the offset of I_1 is smaller than the offset of I_2, copy I_1 into D_3 and assign the next item from D_1 to I_1, go to step 1.
   - if the offset of I_1 is larger than the offset of I_2, copy I_2 into D_3 and assign the next item from D_2 to I_2, go to step 1.
2. If the range of I_1 does overlap with the range of I_2, the two items are merged into I_3. Suppose $I_1 =< L_1, O_1, byte, byte, ... >$ and $I_2 =< L_2, O_2, byte, byte, ... >$.
   - If $O_2 + L_2$ is greater than or equal to $O_1 + L_1$, assign I_3 to I_2 and assign the next item from D_1 to I_1. Go to step 1.
   - If $O_2 + L_2$ is smaller than $O_1 + L_1$, assign I_3 to I_1 and assign the next item from D_2 to I_2. Go to step 1.

After the above algorithm is finished, D_1 and D_2 are merged into D_3. If a page is updated by applying D_3 to it, the result is the same as applying D_1 and D_2 sequentially to the page.

The advantages of diff merging are obvious. First, diff merging can reduce the number of diffs as well as the amount of memory space used for diffs since most diffs of the same page overlap each other and the merged diff only keeps the most up-to-date diffs. Second, less CPU time is consumed by diff applying. The CPU time for diff applying is proportional to the total size of the applied diffs.

3.3 The VOUPID protocol

Using the diff merging algorithm, the VOUPID protocol maintains a single integrated diff for each page of a view. Since the VOPP style requires that writable views must not be acquired in a nested manner, it is guaranteed that modifications on different views are not mixed during execution of

any VOPP-style programs. Also processors modify a view one after another in a synchronized way. Therefore, it is possible to maintain a single integrated diff for each page of a view and then to update the view with those single diffs in the implementation of the VC model. Note that a page may belong to multiple views because of false sharing, in which case the page will have a single diff for each involved view. Since views are non-overlapping, the diffs of the same page for different views are irrelevant.

The VOUPID protocol for the optimal implementation of the VC model is described as below. According to the condition of View-based Consistency, VOUPID only updates the pages involved in the view when a view is acquired. Pages (of a view) that were previously modified are very likely to be accessed after the view is acquired. Thus, instead of invalidating pages, VOUPID piggy-backs the single diffs of those pages on the view granting message and eagerly updates the pages by applying the diffs. In this way VOUPID reduces the number of messages and avoids page faults resulting from invalidation of pages. When a diff of a page is created at view releasing time, the diff is merged with the present single diff of the page to form the new single integrated diff of the page.

More specifically, the following tasks are done in VOUPID when $acquire\_view$ or $acquire\_Rview$ is called.

- Send out the message of view request to the view manager and wait for the view granting message.
- When view granting message is received, the piggy-backed diffs are applied to the corresponding pages to update the pages in the view.
- In case of $acquire\_Rview$, if a page has a twin due to being previously modified, when the diff of the page is applied to the page it has to be applied to the twin as well, in order to correctly acquire the modifications of the page later at $release\_view$ for a writable view.
- Make write-protected all pages with no twin so that any page to be modified can be detected and its twin can be created.

When $release\_view$ or $release\_Rview$ is called the following tasks are done in VOUPID.

- In case of $release\_view$, create a diff for each page that is modified during the current access of the view.
- In case of $release\_view$, for each modified page merge the newly created diff and the present single diff into a new single integrated diff, which should be put into the diff list of the view.
- If there is a view requester waiting for accessing the view, send to the requester the view granting message along with the single diffs from the diff list of the view; otherwise leave this task to the view request handler which processes view requests at the background.

Figure 2 shows an example explaining how VOUPID works.

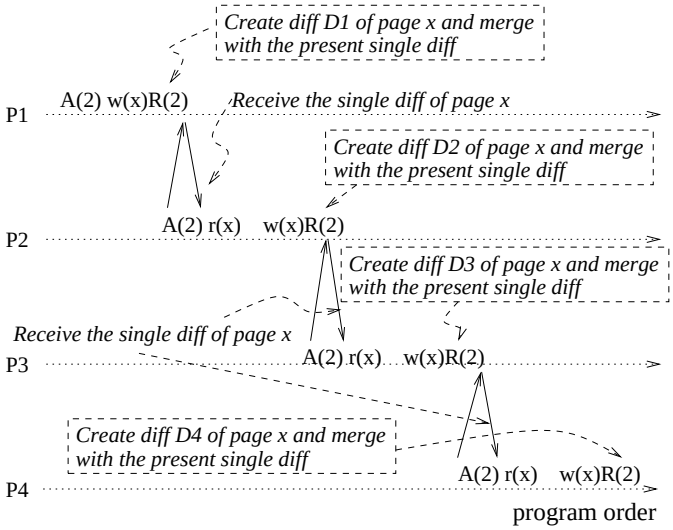

Figure 2: The VOUPID protocol in action

The program in Figure 2 is the same as the one in Figure 1. In Figure 2 every time view 2 is released, a new diff is created and merged into the present single diff of page x. Every time the view is acquired, the single diff of page x is piggy-backed on the view granting message and then applied to the page. Since page x is already updated after the view is acquired, there is no page fault for requesting diffs when the page is accessed. In this way, the number of messages and the amount of diffs are significantly reduced in VOUPID, especially when the number of processors and the number of pages involved in a view is large.

In summary, compared with the original diff scheme VOUPID has reduced the diff requests and the amount of diffs. The extra overhead is diff integration, but it is normally faster than page fault handling, especially when the ranges of the diffs are overlapping.

4 Comparison with the home-based protocol

The home-based protocol [12] allocates a processor (home) for each page. The home processor of a page keeps an up-to-date copy of the page (home page). Every time a page is modified, its diff is created and sent to the home of the page in order to update the home page. When a processor needs

to update its copy of a page, it requests the home page from the home of the page.

The home-based protocol can avoid diff accumulating problem by integrating diffs of a page into the home page. For those applications with diff accumulation, the home-based protocol is significantly better than the original diff scheme. However, compared with VOUPID, it incurs more messages for requesting home pages. Figure 3 gives an example to explain how the home-based protocol works in our VC model.

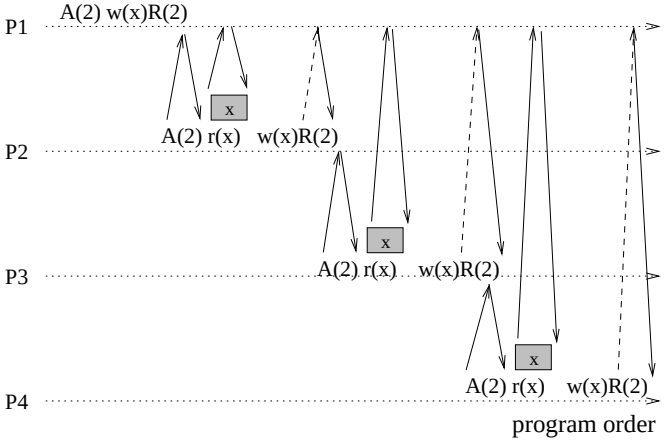

Figure 3: The home-based protocol in action

In Figure 3, every time view 2 is released a diff of page x is created and sent to the home of the page (the home is assumed to be P_1 in the figure). Every time view 2 is acquired, the consistency information of the view is piggy-backed on the view granting message. The consistency information is generated according to the version information of the view. In the figure, the consistency information invalidates page x in P_2, P_3 and P_4. When page x is accessed by any of those processors, a page fault occurs which brings the home page of x to the processor.

By comparing Figure 3 with Figure 2, we can see the home-based protocol incurs more messages than the VOUPID protocol. Each page fault incurs two messages in the home-based protocol. In addition, each modified page incurs two messages for updating the home page. On the other hand, the page faults are reduced in VOUPID by presending the single diffs and the diff requests are reduced accordingly. For example, if view 2 involves two pages, the number of messages will increase by 12 in Figure 3 (in each of P_2, P_3 and P_4 there will be two extra messages for the extra page fault and two extra messages for the extra home updating), while in Figure 2 the number of messages stays the same. Also the home-based protocol requests a whole page from the home once a page needs to be updated, while VOUPID only needs a single diff to update a page. Since a diff is normally smaller than a page and would not be larger than a page in the worst case, the amount of data transferred in VOUPID is smaller. Compared with the home-based protocol, the extra overhead for VOUPID is again diff integration. Overall, VOUPID is more efficient than the home-based protocol, especially when a view involves more pages.

5 Experimental evaluation

In this section, we present our experimental results of several applications running on the following three DSM implementations: VC_d, VC_h and VC_{VOU}.

- VC_d is our implementation of VC based on the diff-based protocol which uses multiple diffs to represent modifications of a page.
- VC_h is our implementation of VC based on the home-based protocol.
- VC_{VOU} is our implementation of VC based on the VOUPID protocol.

All tests are carried out on a cluster 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 is 4 KB.

The applications used in our tests include Integer Sort (IS), Gauss, Successive Over-Relaxation (SOR), Binary Tree (BT), and Neural network (NN). *IS* 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 memory access pattern is very similar to the pattern of our sum example in Section 2. *Gauss* implements the Gauss Elimination algorithm in parallel. Multiple processors process a matrix following the Gaussian Elimination steps. *SOR* uses a simple iterative relaxation algorithm. The input is a two-dimensional grid. During each iteration, every matrix element is updated to a function of the values of neighboring elements. *BT* generates a fixed-depth binary tree. In the algorithm, multiple processors get unexpanded nodes from a task queue. If a processor finds an unexpanded node, it expands the node and creates new unexpanded nodes which are put into the task queue. The algorithm terminates when all nodes in the fixed-depth binary tree are expanded. *NN* trains a back-propagation neural network in parallel using a training data set. After each epoch, the errors of the weights

are gathered from each processor and the weights of the neural network are adjusted before the next epoch. The training is repeated until the the neural network converges.

5.1 Integer Sort (IS)

The problem size of *IS* in our experiment is ($2^{25} \times 2^{15}$, 40). Table 1 shows the statistics of IS running on 32 processors.

	VC_d	VC_h	VC_{VOU}
Time (Sec.)	158.2	26.6	23.6
Data (GByte)	1.03	0.595	0.344
Num. Msg	627,862	481,305	324,762

Table 1: Statistics of IS on 32 processors

In the table, $Time$ is the running time of the application; $Data$ is the total amount of data transferred; and $Num.Msg$ is the total number of messages. VC_d demonstrates serious diff accumulation problem in IS. From the statistics, we find the amount of data transferred in VC_d is about twice of that in VC_h and three times of that in VC_{VOU}. Even though there is no diff accumulation problem in VC_h, the amount of data transferred in VC_h is larger than that in VC_{VOU} since a home page in the home-based protocol is normally larger than a single integrated diff in the VOUPID protocol. Table 1 shows the number of messages and the amount of data transferred in VC_{VOU} are greatly reduced compared with VC_d and are significantly less than VC_h, which is consistent with our comparison between the diff-based protocol, the home-based protocol, and the VOUPID protocol. Not surprisingly, VC_{VOU} is about seven times faster than VC_d and significantly faster than VC_h.

5.2 Gauss

The matrix size of *Gauss* is 2048×2048 and the number of iterations is 1024 in our tests. The diff accumulation problem is not serious in $Gauss$. The shared data between processors is much smaller than a page, so using diffs to represent modifications is more efficient than using pages. Table 2 shows VC_h transfers seven times more data than VC_d. Though the number of messages in VC_d is more than that in VC_h, VC_d is still three times faster than VC_h. VC_{VOU} is significantly distinguished among the three implementations in terms of time, data traffic, and number of messages.

	VC_d	VC_h	VC_{VOU}
Time (Sec.)	16.6	48.5	11.7
Data (MByte)	27.4	200.3	24.7
Num. Msg	232,574	219,353	171,238

Table 2: Statistics of Gauss on 32 processors

5.3 Successive Over-Relaxation (SOR)

SOR processes a matrix with size 4000×4000 and the number of iterations is 50 in our tests. Similar to *Gauss*, *SOR* does not have serious diff accumulation problem, and the shared data in *SOR* between processors is smaller than a page. Therefore the diff-based protocol is more efficient than the home-based protocol in this application. Table 3 shows data traffic in VC_h is nine times of that in VC_d and the number of messages in VC_h is significantly larger than that in VC_d. Again VC_{VOU} performs the best among the three implementations in terms of time, data traffic, and number of messages.

	VC_d	VC_h	VC_{VOU}
Time (Sec.)	7.18	7.93	5.61
Data (MByte)	6.29	56.37	5.72
Num. Msg	69,160	81,043	44,368

Table 3: Statistics of SOR on 32 processors

The above three applications demonstrate that, if there is a serious diff accumulation problem in an application the home-based protocol performs better than the diff-based protocol; otherwise the diff-based protocol performs better. However, the VOUPID protocol is superior to both the diff-based protocol and the home-based protocol, no matter if there is a diff accumulation problem or not.

5.4 Binary Tree (BT)

BT generates a binary tree with a depth 9 in our tests. It uses a task queue to keep all those unexpanded nodes. The memory access pattern is very similar to the task queue example described in Section 2. Each processor repeatedly acquires the task queue to get an unexpanded node. The number of times to access the task queue is not very stable and is different every time the application is run, but the range for that number is very stable for any particular implementation. In Table 4, $Num.Acquires$, which is the number of view primitives called in the application, is taken from a typical execution of the application. From the table, we find the number of view primitives is significantly larger when *BT* is running on VC_{VOU} than when running on VC_d or VC_h. The reason is that VC_{VOU} is more efficient and the processors have more time to repeatedly check the task queue which may be empty. From the row $AcquireTime$

in the table, we find the average time taken for view primitives in VC_{VOU} is much smaller than that in VC_d or VC_h. Therefore, even though the data traffic and the number of messages are larger in VC_{VOU} due to the large number of view primitives called during execution, VC_{VOU} performs significantly better than VC_d and VC_h.

	VC_d	VC_h	VC_{VOU}
Time (Sec.)	45.66	29.56	17.95
Data (MByte)	6.39	11.23	11.76
Num. Msg	8850	11,545	83,743
Num. Acquires	1536	1800	28,214
Acquire Time (usec.)	711,630	340,434	8750

Table 4: Statistics of BT on 32 processors

5.5 Neural Network (NN)

The size of the neural network in *NN* is $9 \times 40 \times 1$ and the number of epochs taken for the training is 235. *NN* is an application which has a very serious diff accumulation problem, especially when the number of processors is large. From Table 5 we find the data traffic in VC_d is more than ten times of that in VC_{VOU} and VC_d is seven times slower than VC_{VOU}. VC_h performs much better than VC_d, but takes twice the time as VC_{VOU}.

	VC_d	VC_h	VC_{VOU}
Time (Sec.)	302.89	83.74	42.64
Data (MByte)	1420.6	436.6	122.3
Num. Msg	343,658	334,481	165,042

Table 5: Statistics of NN on 32 processors

Table 6 presents the speedups of VC_d, VC_h and VC_{VOU} with a varying number of processors. The table shows the speedups of *NN* are significantly improved by VC_{VOU}. The speedup starts to drop in VC_d when the number of processors is 16, and the speedup in VC_h starts to drop when the number of processors is 24.

	2-p	4-p	8-p	16-p	24-p	32-p
VC_d	1.97	3.79	6.18	5.58	3.51	2.22
VC_h	1.97	3.79	6.64	9.01	8.73	7.81
VC_{VOU}	1.99	3.97	7.73	13.43	16.17	16.95
MPI	1.78	3.64	7.17	14.08	20.22	25.38

Table 6: Speedup of NN on VC_d, VC_h, VC_{VOU} and MPI

To compare the performance of VOPP programs with MPI programs, we run the equivalent MPI version of *NN* on MPICH [4]. The speedups of the MPI version of *NN* is also shown in Table 6. The performance of VC_{VOU} is comparable with that of the MPI version on up to 16 processors. On more than 16 processors, the speedup of *NN* running with VC_{VOU} still keeps growing, though it is not as good as the MPI program. We will investigate the reason behind the performance difference between the VOPP program and the MPI program running on larger number of processors in the future.

6 Conclusions

The VOUPID protocol is very efficient for implementation of the VC model. Compared with the diff-based protocol and the home-based protocol, the VOUPID protocol is significantly better in terms of performance. The amount of data traffic and the number of messages are greatly reduced in VOUPID, especially when there is a serious diff accumulation problem in the diff-based protocol. VOUPID is an optimal protocol for supporting VOPP programs and makes their performance comparable with MPI programs, though MPI programs may still perform better when the number of processors is large. We will investigate the reasons behind the performance difference between VOPP programs and MPI programs and will develop more efficient implementation techniques for the VC model. Our ultimate goal is to make shared memory parallel programs as efficient as message-passing parallel programs on cluster computers.

Acknowledgments

The authors would like to thank Mark Pethick who kindly provided his neural network application.

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] Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Techniques for reducing consistency-related information in distributed shared memory systems. ACM Transactions on Computer Systems 13 (1995) 205–243

[3] Gharachorloo, K., Lenoski, D., and 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.

[4] Gropp, W., Lusk, E., Skjellum, A.: A high-performance, portable implementation of the MPI

message passing interface standard. Parallel Computing 22 (1996) 789–828

[5] 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

[6] Huang, Z., Purvis M., and Werstein P.: View-Oriented Parallel Programming and View-based Consistency. to appear In: Proc. of the Fifth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT04) (2004) Singapore.

[7] Huang, Z., Purvis M., and Werstein P.: View-Oriented Parallel Programming on Cluster Computers. submitted to the 34th International Conference on Parallel Processing (ICPP05) (2005) Norway.

[8] Keleher, P.: Lazy Release Consistency for distributed shared memory. Ph.D. Thesis (Rice Univ) (1995)

[9] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers 28 (1979) 690–691

[10] 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

[11] Werstein, P., Pethick, M., Huang, Z.: A Performance Comparison of DSM, PVM, and MPI. In: Proc. of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT03), IEEE Press, (2003) 476–482

[12] 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