VODCA: View-Oriented, Distributed, Cluster-based Approach to parallel computing

Z. Huang† W. Chen‡ M. Purvis† W. Zheng‡
†Departments of Computer & Information Sciences
University of Otago, Dunedin, New Zealand
Email:hzy@cs.otago.ac.nz
‡Department of Computer Science
Tsinghua University, Beijing, China
Email:cwg@tsinghua.edu.cn

Abstract

This paper presents a high-performance Distributed Shared Memory system called VODCA, which supports a novel View-Oriented Parallel Programming on cluster computers. One advantage of View-Oriented Parallel Programming is that it allows the programmer to participate in performance optimization through wise partitioning of the shared data into views. Another advantage of this programming style is that it enables the underlying Distributed Shared Memory system to optimize consistency maintenance. VODCA implements a View-based Consistency model and uses an efficient View-Oriented Update Protocol with Integrated Diff to maintain consistency of a view. Important implementation details of VODCA are described in this paper. Experimental results demonstrate that VODCA performs very well and its performance is comparable with MPI (Message Passing Interface) systems.

Key Words: Distributed Shared Memory, VODCA, View-based Consistency, View-Oriented Parallel Programming, Parallel Computing, Cluster Computing, Message Passing Interface

1 Introduction

Distributed Shared Memory (DSM) is a convenient programming platform for parallel computing on cluster computers. It enables computing processes to communicate with a virtual shared memory, instead of using message passing. As we know, programming platforms based on shared memory greatly relieve the complexity of parallel programming. Unfortunately, the performance of currently available DSM systems such as TreadMarks [1] is far from satisfactory [6, 7], which renders them unusable, especially on cluster computers.

In order to provide a high performance DSM system on cluster computers, we have implemented a free GPL licensed software–VODCA. VODCA stands for View-Oriented, Distributed, Cluster-based Approach to parallel computing. It implements a View-based Consistency model [4] and supports a novel View-Oriented Parallel Programming (VOPP) style [6]. Consistency maintenance in VODCA is *view-oriented*, i.e., it is achieved based on the unit of a view (part of the shared memory), not for the whole memory space. Unlike traditional DSM systems, there is no centralized consistency maintenance (e.g. at barriers) for the whole memory space in VODCA. That is, in VODCA no consistency maintenance is done at barriers by a barrier manager. Instead, consistency maintenance in VODCA is *distributed* to individual view acquisitions, which significantly improves the performance of consistency maintenance [6]. Finally, VODCA is *cluster-based*, i.e., it is aimed at providing an efficient DSM system on cluster computers. Its performance was expected to be comparable with that of Message Passing Interface (MPI) systems, which proves to be true with our experimental results.

The rest of this paper is organised as follows. Section 2 illustrates the VOPP programming interface on VODCA with some VOPP programs. Section 3 describes the key techniques in implementation of VODCA. Section 4 compares our work with other related work. Section 5 presents the performance results of several applications. Finally, our future work on VODCA 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 *Properties of Views*

- A view is a set of data objects in shared memory. Suppose M is the set of total data objects 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 $\forall V_i \forall V_j, V_i \cap V_j = \phi$
- Views in shared memory should cover all data objects in shared memory. Suppose there are n views $V_1, ..., V_n$ in total in shared memory and M is the set of total data objects in shared memory, then

$$\sum_{i=1}^{n} V_i = M$$

- Once created, a view must not be changed (Note that views can be created and destroyed on the fly during execution of a program).

There are a number of requirements for VOPP programmers.

- The programmer should divide the shared data into a number of views according to the data sharing pattern of the parallel algorithm.
- Shared data should be divided in a way so that each view should consists 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).

No explicit definitions of views are needed in a program. The data objects in a view are decided by the programmer in the algorithm. The programmer must allocate an identifier to each view in the algorithm. When a view is accessed in the program, view primitives must be used with the view identifier as their argument. The association between data objects and a view identifier is automatically detected by the underlying system. This association will not be changed until the view is destroyed. When we say a view is destroyed, it means its association with data objects is removed and its identifier can be used for new views. When we say a view is created, it means its association with data objects is established.

The following view primitives in C are provided by VODCA to support VOPP:

- *void Vdc_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.
- *void Vdc_release_view(int view_id)*: release the specified view.
- *void Vdc_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.
- *void Vdc_release_Rview(int view_id)*: finish read-only access to the specified view.

In current version of VODCA, a process can write only one view at a time (in order that VODCA will be able to detect modifications for only one view), but it can read multiple views at the same time by using nested calls of *Vdc_acquire_Rview*. That is, *Vdc_acquire_view*s cannot be nested but *Vdc_acquire_Rview*s can be nested.

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.
- *char *Vdc_malloc(unsigned size)*: allocate shared memory.
- *void Vdc_free(char *ptr)*: free shared memory.

To illustrate the use of VODCA C interface, the following parallel sum problem is used, whose memory access pattern is very typical in parallel programming. In this problem, every process has its local array and needs to add[1] it to a shared array. We divide the shared array into *Vdc_nprocs* equally-sized views, and each of them is allocated an identifier in the range 0 ... *Vdc_nprocs-1*. In each outer loop, every process works on a different view and adds its corresponding local array elements to the view. In the first outer loop, process 0 works on view 0, process 1 works on view 1,and process *Vdc_nprocs-1* works on view *Vdc_nprocs-1*; in the second outer loop, process 0 works on view 1, process 1 works on view 2,and process *Vdc_nprocs-1* works on view 0; and etc. Finally the master process (process 0) calculates the sum of the shared array, which equals to the sum of all local arrays. The VOPP program for this parallel sum algorithm is as below.

[1]It could be a more complicated operation, but we use addition in order to simplify the example.

```
int *shared_array, *local_array, a_size;

main(int argc, char **argv)
{
int i, j, s, e;
long sum;

initialise a_size;

Vdc_startup(argc, argv);

if(Vdc_proc_id == 0) {
  shared_array =
       Vdc_malloc(a_size*sizeof(int));
  initialise shared_array;
  }

local_array=malloc(a_size*sizeof(int));
initialise local_array;

for (i = 0; i < Vdc_nprocs; i++) {
  s=(i+Vdc_proc_id)%Vdc_nprocs
                    *a_size/Vdc_nprocs;
  e=((i+Vdc_proc_id)%Vdc_nprocs+1)
                    *a_size/Vdc_nprocs;

  Vdc_acquire_view((i + Vdc_proc_id)
                          %Vdc_nprocs);
  for (j=s;j < e;j++)
      shared_array[j]+=local_array[j];
  Vdc_release_view((i + Vdc_proc_id)
                          %Vdc_nprocs);
  }

Vdc_barrier(0);

if(Vdc_proc_id==0){
  for(j=0;j<Vdc_nprocs;j++)
                  Vdc_acquire_Rview(j);
  for (i = a_size-1; i > 0; i--)
     sum += shared_array[i];
  for(j=0;j<Vdc_nprocs;j++)
                  Vdc_release_Rview(j);
  }

}
```

Another VOPP example is using task queue for parallel computing. 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 task queue is a shared array which is allocated a view identifier 0. The code pieces of the program are as below.

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

struct task *task_queue;
task_queue=Vdc_malloc(qsize);

/* task producer */
struct task t;
unsigned v;

v = get_unique_vid();
Vdc_acquire_view(v);
t.task_data=Vdc_malloc(tsize);
create_task(t);
t.view_id=v;
Vdc_release_view(v);
Vdc_acquire_view(0);
enqueue(task_queue, t);
Vdc_release_view(0);


/* task consumer */
struct task t;
unsigned v;

Vdc_acquire_view(0);
dequeue(task_queue, t);
Vdc_release_view(0);
v=t.view_id;
Vdc_acquire_view(v);
consume_task(t);
Vdc_release_view(v);
```

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; and the larger a view is, the more amount of data transmitted in the view acquisition. Besides maximizing parallelism, programmers should finely tune the program to reduce both the number of view acquisitions and the size of views.

VOPP does not place any 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 programmer does not need to specify which data objects are in which view in the program, as long as the same data objects are accessed whenever the view is acquired.

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 view is acquired using *Vdc_acquire_view*. In this way, the bug of ”data race” is removed from VOPP programs.

3 Implementation

VODCA is implemented entirely as a user-space library on top of Linux. Modifications to the Linux kernel are not necessary. Programs written in C are compiled and linked with the VODCA library. VODCA is portable on modern Unix systems, with some modifications to a small portion of system dependent code such as the SIGSEGV signal handler. Currently, we have tested VODCA on Linux/i386 and Linux/Itanium.

VODCA implements the View-based Consistency (VC) model [4]. When a view needs to be updated in VODCA, an efficient consistency protocol called VOUPID (View-Oriented Update Protocol with Integrated Diff) [5] is used.

VODCA is developed as an open source software under GPL. For historical reasons, some of its design ideas, e.g., using SIGIO signal for receiving data from sockets, are the same as in TreadMarks, but the code lines are re-written. The authors would like to thank TreadMarks group for their excellent work on the design of TreadMarks.

3.1 View-based Consistency (VC)

The consistency condition for the VC model is stated below.

Definition 2 *Consistency Condition for View-based Consistency*

- Before a process P_i is allowed to access a view by calling *Vdc_acquire_view* or *Vdc_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* process 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 barriers have nothing to do with consistency maintenance. All consistency maintenance are distributed to view primitives. Removing consistency maintenance from barriers has significantly relieved the bottle-neck problem in the implementation of barriers [6]. This bottle-neck problem has seriously affected the performance of many TreadMarks applications such as IS, especially when the number of processes is more than 8.

When a view is acquired, consistency maintenance is restricted to the view. In this way consistency maintenance is decentralized among the view primitives, rather than centralized in barriers. In our current implementation of VODCA, a process can only modify one view between *Vdc_acquire_view* and *Vdc_release_view*. Therefore, the data objects modified between *Vdc_acquire_view* and *Vdc_release_view* are counted into that view, and thus only those data objects are updated when the view is acquired later.

The write accesses to data objects of a view are detected by the system. VODCA uses the *mprotect* system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a SIGSEGV signal (i.e., page fault). The SIGSEGV signal handler examines the exception stack to determine whether the access is a read or write, and responds according to the type of access. Except at initial stage when pages are set *empty* in all processes (except process 0), DSM pages are always readable in VODCA. After a *Vdc_acquire_view* call is finished, all DSM pages are write protected. A write access to a write protected page will cause a SIGSEGV signal whose handler creates a twin copy of the page and makes the page writable. When a *Vdc_release_view* is called, a diff is created for each of those modified pages through comparing the page and its twin copy. The diffs of the modified pages consist in the modifications of the corresponding view. After a *Vdc_release_view* is finished, all DSM pages are write protected again.

We use a modifying session to represent modifications of a view. A *modifying session* represents the modifications (i.e. diffs) made on a number of pages between *Vdc_acquire_view* and *Vdc_release_view*. It is a data structure containing a list of page diffs. We make a modifying session whenever a process finishes updating a view by calling *Vdc_release_view*. When a modifying session is created, it is associated with the related view whose id number is the argument of the *Vdc_release_view* that causes the creation of the session. Note that when a session is created, diffs of the modified pages are created immediately, instead of being delayed as in TreadMarks. In this way we can avoid mixing irrelevant modifications into the modifying session and thus reduce the size of the session.

A version number is maintained for each view. Once a modifying session is created, the version number of the related view is increased by one. When a view is acquired by a process we can decide if the view in the process should be updated or not according to the version of the view of the process and the latest version of the view.

3.2 The VOUPID protocol

During execution of a parallel program, with more and more modifying sessions created for a view, more and more diffs are created for the same page. This phenomenon is called diff accumulating problem. To solve this problem, VODCA uses the VOUPID protocol to merge diffs.

VOUPID is an update protocol for view consistency which integrates all the diffs of the same page into a single diff and then updates the page with the single integrated diff. The diffs of the same page are thus merged together if they are for the same view. In the protocol, a single integrated diff is maintained for each page of a view. When a modifying session is created, the newly created diff of a page is merged into the single integrated diff if it exists; otherwise, the newly created diff becomes the single integrated diff of the page. Similar to an update protocol, when a view is acquired, the single integrated diffs of the view are piggy-backed on the view granting message and then applied immediately to the corresponding pages of the view. In this way, VOUPID reduces the number of messages and the amount of diff data. Moreover, it completely removes those page faults that request diffs. For details of the VOUPID protocol, refer to [5].

3.3 Exponential backoff timer in VODCA

VODCA implements inter-process communication using the Berkeley sockets interface. It uses UDP/IP as the transport protocol. Since UDP/IP does not guarantee reliable delivery, VODCA uses a request/response model to insure request arrival. When a request is sent, a response is expected no matter how simple the response is. If no response arrives within a certain amount of time, the original request is retransmitted. A timer is set when a request is sent and a response is expected.

Since modern computer networks are very reliable, messages rarely get lost. In most situations that a timer expires, the request is either for a held view or for an incomplete barrier. That means, the request arrives at the destination but the reply is withheld by the requestee waiting for some conditions. In those situations, it is useless to retransmit the request in a constant period as in TreadMarks. Rather the retransmissions simply increase the network traffic and the requestee's workload, since receiving a message involves interrupts, context switches, and the execution of several layers of networking software and signal handler.

VODCA adopts an exponential backoff timer for retransmitting possibly lost messages. The value of the timer is first set as one second. Every time the timer expires, the request is retransmitted, but the value of the timer is doubled when the timer is reset. In this way, the number of retransmissions is significantly reduced when the requestee withholds the reply for a long time.

3.4 View ownership

For each view, a process is allocated as its manager. A view manager keeps track of the ownership of the view. A view acquiring request is first sent to the manager which forwards the request to the right process. Multiple processes acquiring the same view with *Vdc_acquire_view* are formed as a queue, called ownership queue, according to their requests' arriving order at the manager. A new acquiring request is forwarded by the manager to the last process in the queue. When a process finishes a call to *Vdc_acquire_view*, the ownership of the view is transferred to the process. The owner of a view has the up-to-date version of the view.

However, when a process calls *Vdc_acquire_Rview*, the ownership of the view is not transferred to that process. The read-only request is simply forwarded by the view manager to the last process in the ownership queue. That process will transmit the diffs of the view to the requester after it releases the view.

4 Comparison with other related work

The idea of restricting the scope of consistency in the VC model is not new. There were some related work on restricting the scope of consistency, e.g. Entry Consistency (EC) [2] and Scope Consistency (ScC) [8]. Similar to VC, both EC and ScC were aiming at reducing the cost of consistency maintenance through restricting the scope of consistency. 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 explicitly associate data objects with locks and barriers in programs, while the VOPP programmer just uses the view primitives to annotate the code section that accesses a view. The data objects in a view are detected and associated with the view automatically, instead of being specified by the programmer as in EC programs. Since the association is achieved dynamically, views can contain dynamically allocated memory space, which cannot be specified statically in the programs of Entry Consistency. Therefore, VOPP is more flexible than the EC programming style.

VOPP is also different from the programming style of Scope Consistency (ScC) [8] in terms of the definition of the concepts of view and scope. Once determined by the programmer, 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. As we discussed and demonstrated in [6], consistency maintenance in barriers suffers from serious bottleneck problem, which affects the performance of traditional DSM programs as well as the ScC programs.

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.

Since ScC only optimizes consistency of local scopes, the performance of programs without lock or scope primitives, such as IS, SOR and Gauss in our benchmark applications, cannot be improved by ScC. In contrast, in VOPP view primitives are always used to access views. Therefore, the VOPP version of those benchmark applications can run more efficiently than their ScC counterparts.

In addition, the diff integration in the VOUPID protocol can only be done when views are not modified in a nested style. If views or scopes are modified in a nested style, which is allowed in ScC programs and other traditional DSM programs, diffs from different views may mix with each other and diff integration can result in incorrect memory consistency. For example, suppose there are four processors P_1, P_2, P_3, and P_4, each of them modifies either x or y or both, where x and y belong to view 1 and view 2 respectively, but are in the same page. As shown in Figure 1, P_1 modifies both x and y in a nested style, so the diffs for view 1 and view 2 have to be mixed with each other. After P_1 finishes accessing x and y, the diff created by P_1 is D_1: $<(x,1),(y,2)>$, which contains the written values of both x and y. Then P_2 accesses and modifies view 1, and create diff D_2: $<(x,3)>$; P_3 accesses and modifies view 2, and create diff D_3: $<(y,4)>$. Finally P_4 accesses both view 1 and 2 after the views are updated using D_1, D_2 and D_3. However, if diff integration is applied in this example, D_1 and D_2 will be merged at P_2 as DM_1: $<(x,3),(y,2)>$, and D_1 and D_3 will be merged at P_3 as DM_2: $<(x,1),(y,4)>$. When P_4 accesses view 1 and 2, DM_1 and DM_2 will be used to update the views. No matter in which order to apply DM_1 and DM_2 at P_4, it will not be able to read both the up-to-date value (3) of x and the up-to-date value (4) of y.

Note that when the *acquire_view*s and *release_view*s in Figure 1 are replaced with $open\_scope$ and $close\_scope$ (as

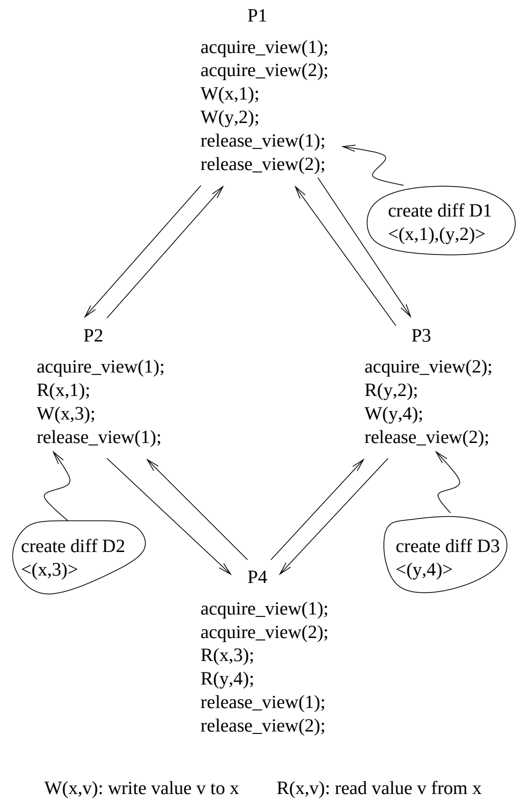

Figure 1: A failure scenario for diff integration

in ScC programs) or $acquire\_lock$ and $release\_lock$ (as in TreadMarks) respectively, diff integration will result in the same incorrect memory consistency. Therefore, ScC and TreadMarks cannot adopt the VOUPID protocol to improve the performance of DSM systems.

Note that, if nested writable view acquisitions need to be supported, a variant of VOUPID can be developed to cope with the special case, though the protocol will become more complex and less efficient.

Diff integration can only be done in a centralized way at barriers for traditional DSM programs because of the above problem. For example, TreadMarks integrates diffs at barriers using garbage collection, and Brazos [10] integrates diffs from local scopes into the diffs of the global scope at barriers. In contrast, diff integration in VODCA is achieved in a timely, distributed way, which is more efficient.

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 the consistency of the shared memory is maintained according to views). 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 instead of sending and receiving 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 the view.

Another difference between VOPP and MPI is that barriers are badly needed in VOPP programs, while MPI programs normally do not need barriers. This difference has significant impact on the performance gap between VOPP and MPI, which has been discussed in [7].

5 Performance comparison

In this section, we present the performance results of several applications coded with MPI, VOPP and the traditional DSM programming style. The MPI applications are run on MPICH [3], the VOPP programs are run on VODCA, and the traditional DSM applications are run on TreadMarks [1].

The performance tests are carried out on an Itanium cluster connected by InfiniBand network. The cluster consists of 128 nodes running Linux kernel 2.4.21, 16 nodes of which are used for performance evaluation. We run two processes on each node, since each node has two 1.3 GHz Itanium 2 processors with 4 Gbytes of memory. The page size of the virtual memory is 16 KB. The C compiler used is GCC 3.2.3. All programs are compiled with "-O2" option.

We have tested VODCA, MPI, and TreadMarks on the cluster with Integer Sort (IS), Gauss, Successive Over-Relaxation (SOR), and Neural network (NN). Figures 2, 3, 4, and 5 have shown the speedups of the applications on the three systems.

The figures show that the performance of VODCA is comparable with that of MPI. For some applications, VODCA performs even better than MPI. Since InfiniBand is a fast network with low latency, the barriers running on

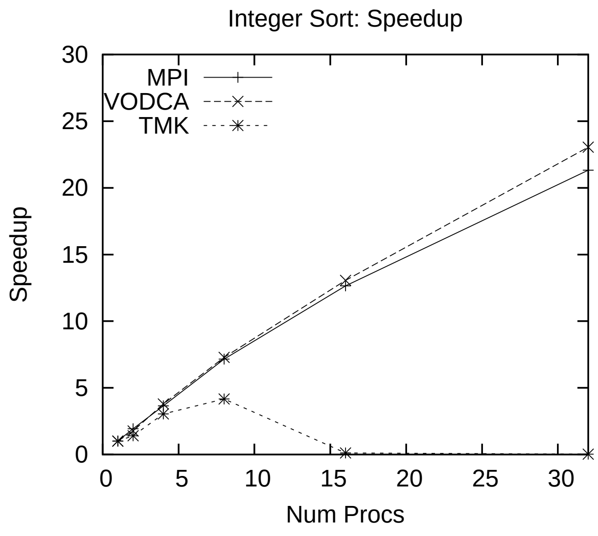

Figure 2: Speedup curves of IS

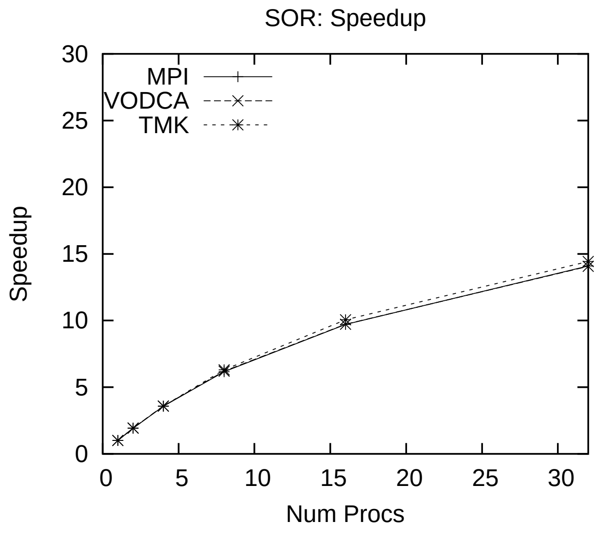

Figure 3: Speedup curves of SOR

such a network is much faster than on Ethernet (refer to [7] for results on fast Ethernet). Faster barriers can improve the performance of both VODCA and TreadMarks, which explains why TreadMarks performs better for Gauss and SOR on InfiniBand. For applications like Gauss, there is still a small performance gap between VODCA and MPI due to thousands of barriers used in Gauss. Detailed reasons for this performance gap has been discussed in [7].

6 Conclusions and future work

Our experimental results demonstrate that VODCA is a high performance DSM system. VODCA provides a convenient programming interface for parallel computing on

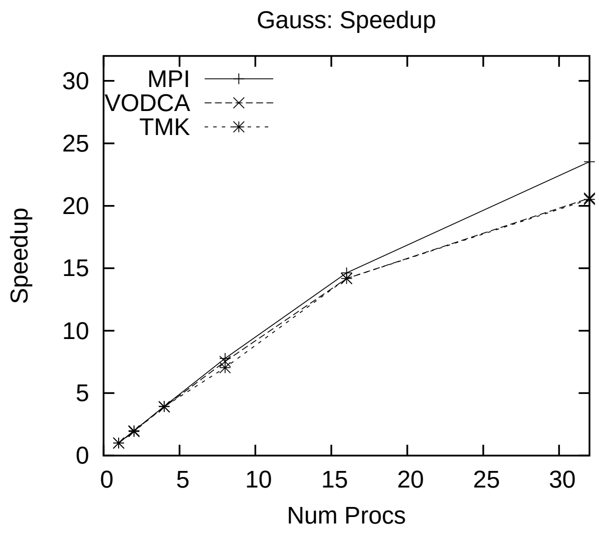

Figure 4: Speedup curves of Gauss

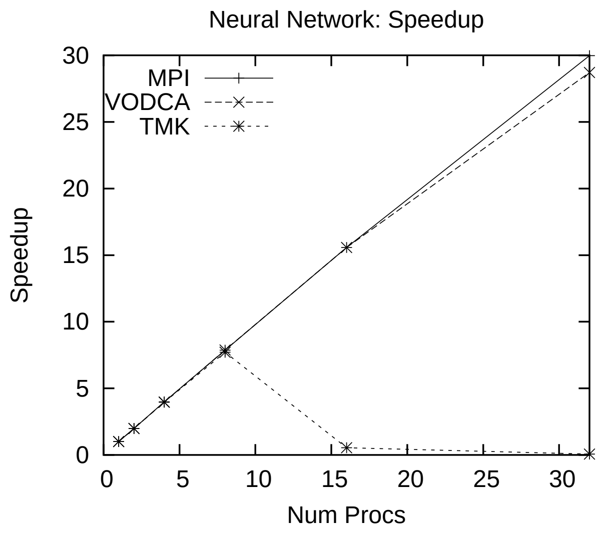

Figure 5: Speedup curves of NN

cluster computers and its performance is comparable with that of MPI systems. It is encouraging to know that, with a high bandwidth, low latency network such as InfiniBand, VODCA can perform better than MPI systems for some applications.

We will use more applications to investigate the performance gap between VODCA and MPI systems, especially when the number of processors is large, e.g., up to 64 or more processors. Techniques for fast barriers need to be investigated when the number of processors is very large, e.g., the order of hundreds or thousands. In order to make VODCA a convenient platform for parallel computing, a view-based debugger for VOPP programmers is needed in the near future. For long-duration applications, a fault-tolerant system is needed in VODCA to restore execution when some computing node fails.

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] 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)

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

[10] Speight, E.: Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors. Ph.D. Thesis (Rice University) (1997)