Performance Comparison between VOPP and MPI

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

View-Oriented Parallel Programming is based on Distributed Shared Memory which is friendly and easy for programmers to use. It requires the programmer to divide shared data into views according to the memory access pattern of the parallel algorithm. One of the advantages of this programming style is that it offers the performance potential for the underlying Distributed Shared Memory system to optimize consistency maintenance. Also it allows the programmer to participate in performance optimization of a program through wise partitioning of the shared data into views. In this paper, we compare the performance of View-Oriented Parallel Programming against Message Passing Interface. Our experimental results demonstrate a performance gap between View-Oriented Parallel Programming and Message Passing Interface. The contributing overheads behind the performance gap are discussed and analyzed, which sheds much light on further performance improvement of View-Oriented Parallel Programming.
Key Words: Distributed Shared Memory, View-based Consistency, View-Oriented Parallel Programming, Cluster Computing, Message Passing Interface

1 Introduction

View-Oriented Parallel Programming (VOPP) [4, 5, 6] is a programming style based on Distributed Shared Memory (DSM). A 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, traditional DSM programs are far from efficient compared with those using Message Passing Interface (MPI) [10]. 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 unnecessary message passing. As for DSM, since consistency maintenance [7] deals with the consistency of the whole shared memory space, there are many unnecessary messages incurred in DSM systems. As we know, message passing is a significant cost for applications running on distributed systems. Even worse, the programmer cannot help reduce those messages when designing a DSM program.

To help DSM optimize its performance as well as to allow programmers to participate in performance tuning such as optimization of data allocation, we have proposed the novel VOPP programming style for DSM applications [4]. VOPP programs perform significantly better than traditional DSM programs [6]. The performance gain is two-fold. First, the programmer is able to participate in performance tuning with the VOPP primitives such as *acquire_view* [6]. Second, consistency maintenance for views can be optimized using the VOUPID protocol [5].

Even though VOPP programs are much more efficient than traditional DSM programs, they are not as efficient as MPI programs when the number of processors becomes larger. Our ultimate goal is to make VOPP programs as efficient as their MPI counterparts. In this paper, we are going to compare VOPP and MPI in terms of performance and to investigate the reasons behind the performance gap.

The rest of this paper is organised as follows. Section 2 briefly describes the VOPP programming style and its underlying View-based Consistency model. Section 3 demonstrates the performance gap between VOPP and MPI. The contributing overheads behind the gap are analyzed and discussed. Finally, our future work on VOPP is suggested in Section 4.

2 VOPP and View-based Consistency

A *view* is a concept used to maintain consistency in distributed shared memory. It consists of data objects that require consistency maintenance as a whole body. Views can be either defined in programs using the primitive *alloc_view* with size of the view as its argument, or a view can be detected automatically by the system as long as the programmer uses the primitives such as *acquire_view* and *release_view* whenever a view is accessed. *Acquire_view* means acquiring exclusive access to a view, while *release_view* means having finished the access. However, *acquire_view* cannot be called in a nested style. For read-only accesses, *acquire_Rview* and *release_Rview* are provided, which can be called in a nested style. By using these primitives, the focus of the programming is on accessing shared

This work was done while the first author was working as an invited professor at EPFL, Switzerland.

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 defined using *alloc_view*. Once decided, 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. Interested readers may refer to [6] and [4] for more details about VOPP.

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 called VOUPID has been proposed in [5] based on this simplicity.

To maintain the consistency of views in VOPP programs, a View-based Consistency (VC) model has been proposed [4]. 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. When a view is acquired, consistency maintenance is restricted to the view.

The Sequential Consistency (SC) [9] of the VOPP programs can be guaranteed by the VC model, which has been proved in [4].

3 Performance comparison

The applications we use for performance comparison include Integer Sort (IS), Gauss, Successive Over-Relaxation (SOR), 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 [4]. 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. 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 neural network converges.

The following figures show the speedups of the applications coded with VOPP and MPI respectively. The speedups of TreadMarks [1], which is a state-of-the-art DSM system, are shown as a base of reference. All performance tests are carried out on our cluster computer called Godzilla. The cluster consists of 33 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. Our DSM system, VODCA, is used to run the VOPP programs, and MPICH [3] is used to run the MPI programs.

Figure 1 shows that the performance of VOPP is very close to and comparable to MPI [1]. Figure 2, Figure 3, and Figure 4 show bigger performance gaps between VOPP and MPI, especially when the number of processors is larger.

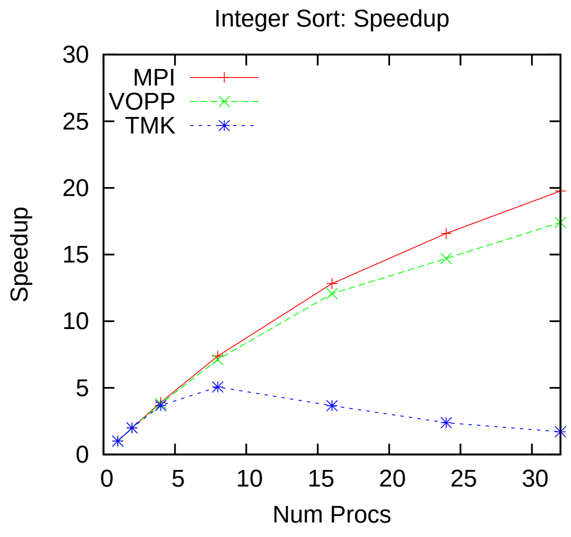

Figure 1: Speedup of IS on VOPP and MPI

There are three overhead sources contributing to the performance gap: barriers, delayed data transmission, and consistency maintenance.

3.1 Barriers

When comparing the VOPP programs with the MPI programs, we find there are more barriers in the VOPP programs than in the MPI programs. There are almost no barriers in our MPI programs, while there are many barriers

[1]In the figures, TMK refers to TreadMarks.

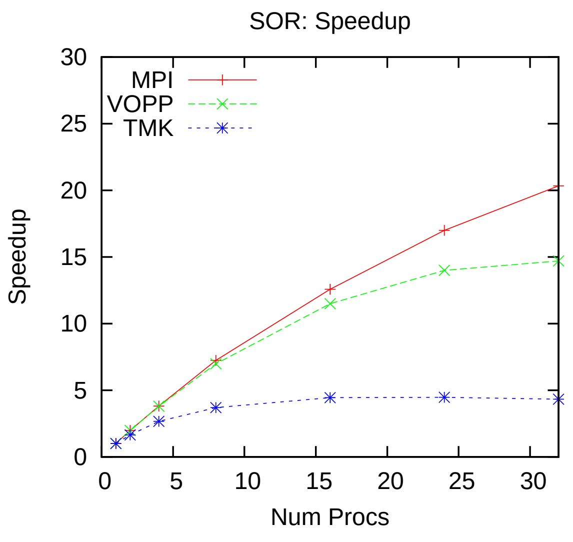

Figure 2: Speedup of SOR on VOPP and MPI

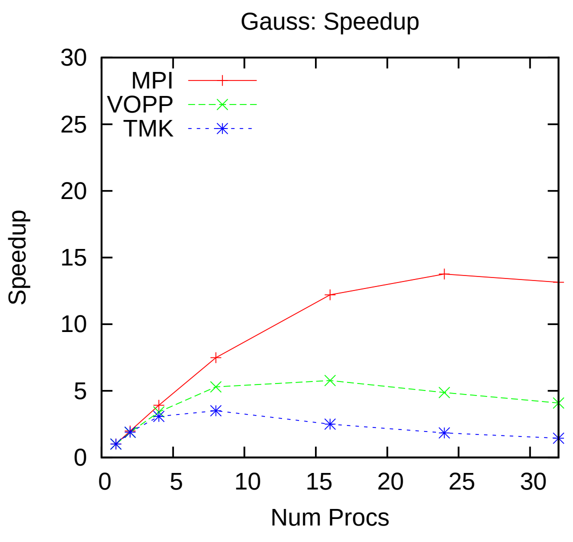

Figure 3: Speedup of Gauss on VOPP and MPI

called in our VOPP programs. In IS 122 barriers are called; SOR calls 102 barriers; Gauss calls 4098 barriers; and NN calls 473 barriers. Accordingly the performance gaps for IS and SOR are smaller, while for Gauss and NN they are larger.

As we know, barriers incur many messages, especially when the number of processors becomes larger. Moreover, barriers cause synchronization among all processors and thus every processor often waits for the slowest processor. This significantly slows down parallel processing when barriers are frequently called.

To explain why there have to be more barriers in VOPP programs, we use a typical producer-consumer problem as an example. Suppose processor 0 is the producer of variable x, and all processors are consumers of the variable. The MPI code is as below.

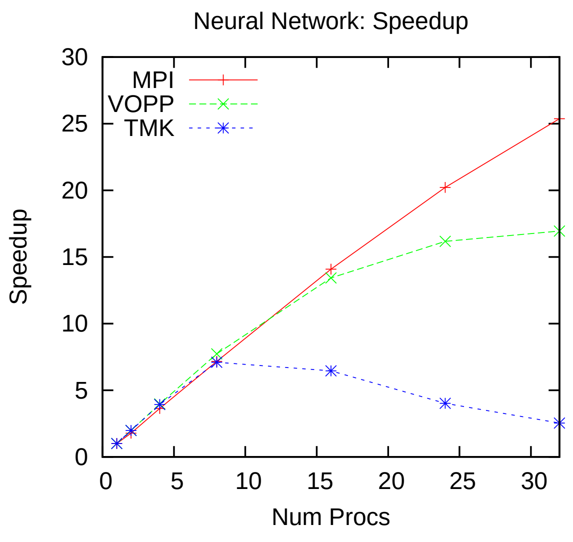

Figure 4: Speedup of NN on VOPP and MPI

```
if(proc_id == 0){
         produce(x);
         send x to all other processors;
}

if(proc_id != 0)
        receive x from processor 0;
consume(x);
```

For the same problem, the VOPP code is as below.

```
if(proc_id == 0){
         acquire_view(1);
         produce(x);
         release_view(1);
}

barrier(0);

acquire_Rview(1);
consume(x);
release_Rview(1);
```

In the VOPP code, the barrier has to be used to make sure the processors consume the variable x after it is produced. This situation is very typical for shared memory based programs. In contrast, in the MPI code, there is no need to use a barrier for synchronization, since the *receive* primitive is synchronized with the *send* primitive and is always finished after the *send* primitive.

To make it even worse, when the VOPP code is executed in a loop as below, another barrier has to be added to make sure the consumers will not overtake the producer in the next loop. As for the MPI code, there is no problem for it to be executed in a loop, as the consumers will never overtake the *send* primitive.

```
for(i=0;i<N;i++){
    if(proc_id == 0){
            acquire_view(1);
            produce(x);
            release_view(1);
        }

        barrier(0);

        acquire_Rview(1);
        consume(x);
        release_Rview(1);

        barrier(1);
}
```

The above situation occurs in our VOPP programs with Gauss and NN. To verify that overhead of barriers is a significant contributor behind the performance gap, we intentionally make the number of barriers in VOPP programs the same as that in MPI programs. We remove some barriers from the VOPP programs and add some barriers to the MPI programs. The performance of those VOPP and MPI programs for SOR and Gauss is shown in Figure 5 and Figure 6.

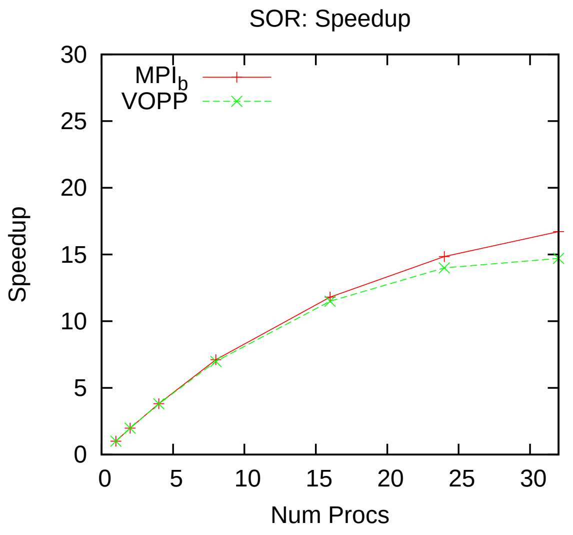

Figure 5: VOPP and MPI with the same number of barriers (SOR)

In Figure 5 the curve *MPI_b* represents the MPI program with unnecessary barriers. For SOR, we find the performance gap is much smaller between VOPP and MPI if the same number of barriers is added to the MPI program. For 32 processors, the speedup gap is reduced from about 6 to 2.

In Figure 6 the curve *MPI_b* represents the MPI program with unnecessary barriers, and the curve *VOPP_b* represents the VOPP program with less barriers (The execution results for the VOPP program are wrong, of course). Both of them have the same number of barriers. Under this situation, the performance gap for Gauss becomes very small. For 32 processors, the speedup gap is reduced from 9 to less than 2.

From the above figures we can conclude that the overhead of barriers is a significant overhead for VOPP programs. The programmer should first avoid unnecessary barriers in programs. As system developers, we should reduce the overhead of barrier implementation to make barriers more efficient.

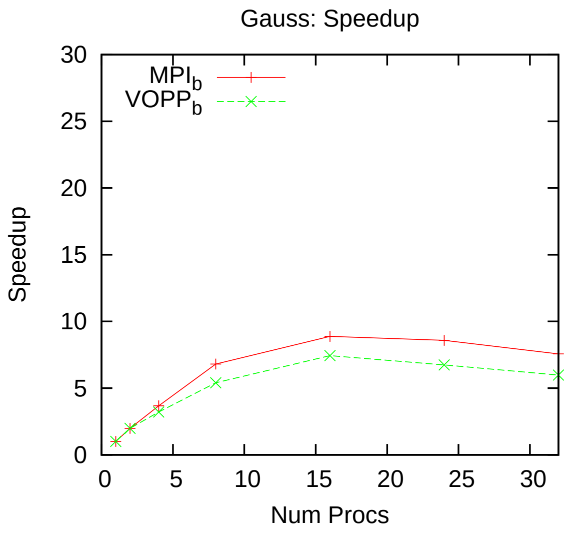

Figure 6: VOPP and MPI with the same number of barriers (Gauss)

3.2 Delayed data transmission

In the above producer-consumer example, data transmission is delayed in the VOPP code due to the use of the lazy release consistency [8] for consistency maintenance of views. To explain the situation, we use Figure 7 and Figure 8 to describe the difference in message transmission between MPI and VOPP. We assume the processor P_0 is the producer of x, and P_1 and P_2 are the consumers of x.

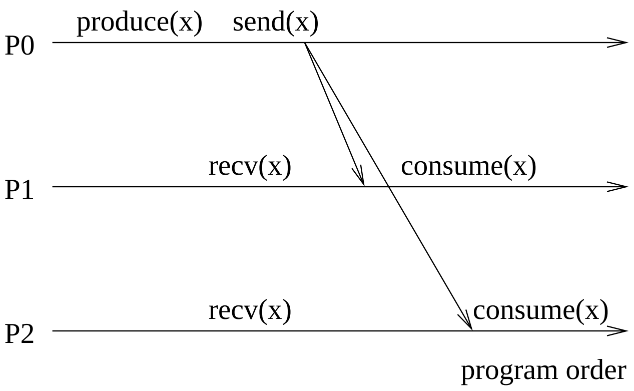

Figure 7: Message transmission in MPI code

Figure 7 shows the message transmission in the MPI code. While P_0 is producing x, P_1 and P_2 are blocked by *recv* waiting for x. Once x is received, P_1 and P_2 consume it immediately.

Figure 8 shows the message transmission in the VOPP code is more complex. While P_0 is producing x, P_1 and P_2 are waiting at barrier 1. After P_0 has produced x and

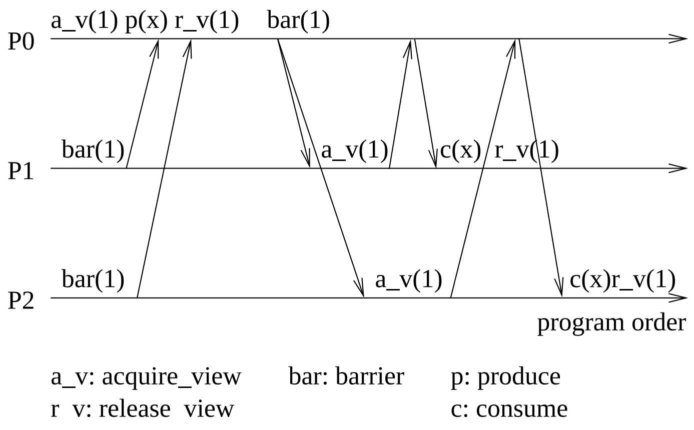

Figure 8: Message transmission in VOPP code

reaches barrier 1, it sends messages to P_1 and P_2 which then carry on the execution. In $acquire\_view$, a message is sent to request the modifications of the view. When receiving the modifications, P_1 and P_2 update the view (x) and then consume x. From the figure, we know the consuming of x is delayed compared with Figure 7. The delayed time is one round-trip time (RTT) plus modification processing time such as view updating. If the view is large, it takes more time to process the modifications. Our NN application suffers this delay very much since the views in the application are large.

3.3 Consistency maintenance

Consistency maintenance is another extra overhead compared with MPI. When a view is acquired, normally there are 3 messages involved. The request message is first sent to the view manager, which then forwards it to the view holder. The view holder eventually sends reply to the view requester. In MPI program, for the similar situation only two messages are needed to serve the purpose. Consistency maintenance also includes modification detection and view updating. However, these overheads are minor compared with barrier overhead and data transmission delay.

4 Conclusions and future work

The above experimental results demonstrate the performance gap between VOPP and MPI. We have discussed the three sources of overhead in VOPP contributing to the performance gap: barriers, data transmission delay, and consistency maintenance.

We are considering to reduce some of the overheads such as barriers and data transmission delay. To reduce the overhead of barriers, we need to investigate distributed algorithms for efficient implementation of barriers. We are also considering to replace barriers with some light-weight synchronization primitives such as signaling.

To remove the data transmission delay, we may use the eager release consistency [2], but there will be other overheads related to the eager release consistency. We will investigate the overheads of the eager release consistency for consistency maintenance of views.

Our ultimate goal is to make shared memory parallel programs as efficient as message-passing parallel programs.

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] 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.

[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. 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), IEEE Computer Society (2005)

[7] 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)

[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] 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