Performance Evaluation of View-Oriented Parallel Programming

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 evaluates the performance of a novel View-Oriented Parallel Programming style for parallel programming on cluster computers. 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. Experimental results demonstrate a significant performance gain of the programs based on the View-Oriented Parallel Programming style.

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

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) [12]. 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 [3, 10, 8] deals with the consistency of the whole shared memory space, there are many unnecessary messages incurred in DSM systems. Furthermore, the programmer cannot help reduce those messages when designing a DSM program.

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 [8], which only updates part of the shared memory in consistency maintenance in DSM. Since DRF-oriented programming focuses on mutual exclusion and synchronization rather than data allocation, there is no communication channel 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 such as optimization of data allocation, we propose a novel View-Oriented Parallel Programming (VOPP) style for DSM applications.

The rest of this paper is organised as follows. Section 2 briefly describes the VOPP programming style. Section 3 discusses possible optimizations when a VOPP program is created or converted from an existing program. Section 4 compares VOPP with related work. Section 5 presents and evaluates the performance results of several applications. Finally, our future work on VOPP is suggested in Section 6.

2 View-Oriented Parallel Programming (VOPP)

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 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 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 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 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 [7, 6] for more details about VOPP and program examples.

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 has been proposed based on this simplicity [5].

To support VOPP, a View-based Consistency (VC) model has been proposed and efficiently implemented [7, 5]. In this 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. In this way, the amount of data traffic for DSM consistency in the cluster network can be reduced and thus the VC model can be implemented optimally [5]. The Sequential Consistency (SC) [11] of VOPP programs can also be guaranteed by the VC model, which has been proved in [7].

3 Optimizations in VOPP

Traditional DSM programs need to be converted into VOPP programs before being run on the VC model. The applications we have converted include Integer Sort (IS), Gauss, Successive Over-Relaxation (SOR), and Neural network (NN). These programs are used in our performance evaluation in Section 5.

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. 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 task of conversion includes identifying exclusive views and inserting view primitives. During the conversion, the applications have been optimized using the following tips.

3.1 Local buffer for infrequently-shared or read-only data

In traditional DSM programs, shared data is accessed directly from the shared memory, no matter how frequently it is shared. Even some read-only data is put into the shared memory. This generous use of shared memory may unnecessarily cause the false sharing effect [8], which results from two or more processors accessing different data objects in the same page of DSM and unnecessary memory consistency maintenance. For some applications such as Gauss, shared data is read in by individual processors and is accessed by the same processor until the end of the program when final result has to be printed out. When we convert Gauss into the VOPP program, local buffers are used to keep the infrequently-shared data during processing. The shared data is divided into views for individual processors and each processor reads its view into a local buffer. The processing on the shared data is carried out on the local buffers by the processors. After the processing is finished, the data in the local buffers is copied back into the shared memory using view primitives. The pseudo-code is shown below.

```
acquire_view(proc_id);
copy from the view to loc_buffer;
release_view(proc_id);

for(i=0;i<max_iterations;i++){
   process data in loc_buffer;
}
acquire_view(proc_id);
copy from loc_buffer to the view;
release_view(proc_id);

barrier(0);
if(proc_id==0){
  for(j=0;j<nprocs;j++)
              acquire_Rview(j);
  read and print data in all views;
  for(j=0;j<nprocs;j++)
               release_Rview(j);
}
barrier(0);
```

We also optimized the NN application in this manner. The application trains a back-propagation neural network in parallel. The training data set is read-only and initially read into the shared memory for programming convenience. We divide the training data set evenly into *nprocs* views, where *nprocs* is the number of processors. The views are copied into local buffers for later processing. The pseudo-code is shown below.

```
acquire_view(proc_id);
copy from the view to loc_buffer;
release_view(proc_id);

while (not trained){
   train the network with data
              in loc_buffer;
   barrier(0);
   change the weights of the network;
   check the errors from the target;
}
```

By using the local buffers, the applications can avoid false sharing effect. Of course, there is an overhead of copying between the shared memory and the local buffers. If the processing on the local buffers is sustained for relatively longer time, the overhead is negligible and there is a performance gain due to the removal of the false sharing effect.

Even though the above optimizations can be done in traditional DSM programs, they are enforced in VOPP. Since *acquire_view*s cannot be nested, infrequently-shared data have to be moved to local buffers so that frequently-shared data can be accessed at the same time.

3.2 Removal of barriers

In VOPP, barriers are only used for synchronization among processors and have nothing to do with access exclusion and consistency maintenance of DSM. The consistency maintenance and access exclusion are achieved automatically by the view primitives. In some traditional DSM programs, barriers are used for access exclusion instead of synchronization, in which case barriers can be removed in VOPP programs. Integer Sort (IS) has such a barrier that can be moved from inside a loop to outside.

3.3 Shared memory for frequently-shared data

Frequently-shared data is often mixed with infrequently shared data in traditional DSM programs. For example, the SOR program processes a matrix with multiple processors, each of which gets a portion of the matrix. Each processor works on its portion most of the time, but needs the border elements between portions from the portion of its neighbour processors after every iteration. Each portion of the matrix is not frequently shared except the border elements. However, the SOR program allocates a block of shared memory to the whole matrix and multiple processors directly access the shared memory, which causes the false sharing effect. The pseudo-code of the traditional SOR program is shown below.

```
processor 0 reads in the matrix
            into the shared memory;
barrier(0);
   //executed by each processor;
```

```
for(i=0;i<max_iterations;i++){
    read border elements
         from its neighbor’s share;
    update its share of the matrix
         with the border elements;
    barrier(0);
}
read and print out the whole matrix;
```

In VOPP, we allocate a local buffer for the portion of the matrix of each processor, since it is not frequently shared. We use separate views for those border elements which are frequently shared. At the end of each iteration, border elements of the views are updated by copying them from the local buffers to their respective views. At the beginning of the next iteration, the border elements in the views are read by the corresponding processors. In this way, only the border elements of the views are passed between processors through the cluster network instead of other irrelevant elements of the same page. The pseudo-code is as below.

```
processor 0 reads in the matrix
            into the shared memory;
barrier(0);
acquire_view(proc_id);
read its share of the matrix
                into a local buffer;
release_view(proc_id);
bdv = nprocs;
for(i=0;i<max_iterations;i++){
    acquire_view(bdv+prev_pid);
    read border elements from
          the previous processor;
     release_view(bdv+prev_pid);

    acquire_view(bdv+next_pid);
    read border elements from
          the next processor;
     release_view(bdv+next_pid);

     update the local buffer
           with the border elements;

    acquire_view(bdv+proc_id);
    update the border elements of
          the current processor;
     release_view(bdv+proc_id);
    barrier(0);
}
if(proc_id==0){
  for(j=0;j<nprocs;j++)
               acquire_Rview(j);
  read and print the whole matrix;
  for(j=0;j<nprocs;j++)
               release_Rview(j);
}
barrier(0);
```

The above optimization is again enforced by VOPP so that frequently-shared data can be accessed at the same time as the infrequently-shared data is accessed. Likewise there is an overhead of copying between the shared memory and the local buffers. However the overhead is negligible if the processing on the local buffers is sustained for relatively long time. Another overhead is the view primitives called in the loop, which will result in more messages than the traditional DSM program. However, there is a big performance difference between the barriers in VOPP and those in traditional programs. Barriers in VOPP simply synchronize the processors without any consistency maintenance, while barriers in traditional programs have to maintain the consistency of the shared memory. Maintaining consistency in barriers is a centralized way for consistency maintenance and becomes time-consuming when the number of processors increases. Even though there are many view primitives in the above VOPP program, consistency maintenance is optimally achieved by them in a distributed way. Therefore, overall the above VOPP program will still perform better than its traditional version, especially when the number of processors is large.

3.4 *acquire_Rview* for read-only data

Read-only views can be accessed with $acquire\_Rview$ and $release\_Rview$ in VOPP. Programmers can use them to improve the performance of VOPP programs, since multiple read-only accesses to the same view can be granted simultaneously, so that the waiting time for acquiring access to read-only views is very small. Programmers can use them to replace barriers and read/write view primitives ($acquire\_view$ and $release\_view$) wherever possible to optimise VOPP programs. In NN, a global weight matrix is shared by all processors. After each iteration, the weight matrix is updated by every processor. At the beginning of each iteration, every processor needs to read the weight matrix to update the neural network. We use $acquire\_Rview$ to enable every processor to read the matrix concurrently rather than sequentially.

3.5 *merge_views* for merging views

When there is a need to rearrange the views at some stage in a program, *merge_views* can be used to update all views of every processor so that each processor has an up to date copy of the whole shared memory. This operation is expensive but convenient for programmers. However, we have not seen any program that has such a need so far.

3.6 Basic rule of thumb

The following basic rule of thumb can help VOPP programmers optimize view partitioning and parallel algorithms proactively: the more views are acquired, the more messages there are in the system; and the larger a view is, the more data traffic is caused in the system when the view is acquired.

4 Comparison with related work

VOPP is different from the programming style of Entry Consistency in terms of the association between data objects and views (or locks). Entry Consistency [2] requires the programmer to explicitly associate data objects with locks and barriers in programs, while VOPP only requires the programmer to implicitly associate data objects with views (in the programmer's mind). The actual association is achieved in view detection in the implementation of the VC model. Since the association is achieved dynamically, VOPP is more flexible than the programming style of Entry Consistency.

VOPP is also different from the programming style of Scope Consistency (ScC) 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. Programs based on ScC are extended from the traditional DSM programs, i.e., lock primitives are normally used in programs while scope primitives such as *open_scope* are used only when required by memory consistency. However, in contrast to the traditional DSM programs, the focus of VOPP is shifted towards shared data (views) rather than synchronization and mutual exclusion.

VOPP is more convenient and easier for programmers than the message-passing programming style such as MPI [4], since VOPP is still based on the concept of shared memory (except the consistency of the shared memory is maintained according to views). Moreover, VOPP provides experienced programmers an opportunity to fine-tune the performance of their programs by carefully dividing the shared memory into views. Partitioning of shared data into views becomes part of the design of a parallel algorithm in VOPP. This approach offers the potential for programmers to make VOPP programs perform as well as MPI programs, which is the ultimate goal of our VOPP-based DSM system.

5 Experimental evaluation

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

- LRC_d is a diff-based implementation of the Lazy Release Consistency (LRC) model [10]. It is the original implementation of LRC in TreadMarks [1], which uses diffs to represent modifications of a page.

- VC_d is our implementation of VC which uses diffs to represent modifications of a page. It uses the same implementation techniques (e.g. the invalidation protocol) as the LRC_d.

- VC_{sd} is our implementation of VC based on a diff integration scheme [5], which uses a single diff to represent modifications of a page and piggy-backs diffs of a view on the view acquiring message. It is an optimal implementation of VC and renders better performance for applications than other DSM implementations.

Since VC_d and LRC_d use the same implementation techniques, the performance advantage of VOPP over traditional DSM programs can be demonstrated by running applications on these two implementations. The overall performance advantage of VOPP (including the potential for an optimal implementation) can be demonstrated by comparing VC_{sd} with LRC_d.

All tests are carried out on our cluster computer called Godzilla. The cluster consists of 32 PCs running Linux 2.4, which are connected by a N-way 100 Mbps Ethernet switch. Each of the PCs has a 350 MHz processor and 192 Mbytes of memory. The page size of the virtual memory is 4 KB. Though our processors are relatively slow, a cluster with faster processors can more favorably demonstrate the advantage of the VC model. The reason is that VC significantly reduces data traffic of the network which is the bottle neck of a cluster with faster processors.

The applications used in our tests include Integer Sort (IS), Gauss, Successive Over-Relaxation (SOR), and Neural network (NN).

5.1 Integer Sort (IS)

The problem size of IS in our tests is ($2^{25} \times 2^{15}$, 40). Table 1 shows the statistics of IS running on 16 processors, which can typically demonstrate the performance of our applications.

	LRC_d	VC_d	VC_{sd}
Time (Sec.)	78.4	53.4	25.8
Barriers	682	682	682
Acquires	0	20,479	20,479
Data (GByte)	1.236	1.279	0.174
Num. Msg	123,994	180,207	80,387
Diff Requests	38270	38,398	0
Barrier Time (usec.)	34,492	5467	2211
Rexmit	114	14	0

Table 1: Statistics of IS on 16 processors

In the table, *Barriers* is the number of barriers called in the program; *Acquires* is the number of lock/view acquiring messages; *Data* is the total amount of data transmitted; *Num. Msg* is the total number of messages; *Diff Requests* is the number of diff requests; and *Barrier Time* is the average time spent on barriers; and *Rexmit* is the number of messages retransmitted. From the statistics, we find the number of messages and the amount of data transmitted in VC_d are more than in LRC_d, however VC_d is faster than LRC_d. The reason is two-fold. First, the barriers in LRC_d need to maintain consistency while those ones in VC_d do not. The consistency maintenance of barriers in LRC_d is normally time-consuming and centralized at one processor which can be a bottleneck. The consistency maintenance in VC_d is distributed among the processors through the view primitives. From the table, the average barrier time in LRC_d is 34,492 microseconds, while the average barrier time in VC_d is 5467 microseconds. Second, LRC_d has more message loss than VC_d according to the number of retransmissions ($Rexmit$ in the table). LRC_d has 114 retransmissions while VC_d only has 14 retransmissions. One message retransmission results in about 1 second waiting time. The above statistics demonstrate the distribution of data traffic in VOPP programs can help reduce message retransmissions and improve the performance of the VOPP programs. The table also shows the optimal implementation VC_{sd} has greatly reduced the amount of data and number of messages in the cluster network.

We have two VOPP versions of IS: one uses the same number of barriers as the original version (whose statistics have been shown above), and the other moves the barrier from inside the loop to outside (as we mentioned in Section 3.2). Table 2 shows the statistics of IS with fewer barriers.

Comparing Table 2 with Table 1, it is not surprising to find that the VOPP version of IS with fewer barriers is significantly faster than its counterpart with more barriers.

	VC_d	VC_{sd}
Time (Sec.)	49.6	24.2
Barriers	122	122
Acquires	20,479	20,479
Data (GByte)	1.278	0.173
Num. Msg	163,420	63,586
Diff Requests	38,398	0
Barrier Time (usec.)	9891	5540
Rexmit	14	0

Table 2: Statistics of IS with fewer barriers on 16 processors

Table 3 shows the speedups of IS running on 2, 4, 8, 16, 24, and 32 processors. From the table we find the speedups of VC_{sd} are significantly better than those of LRC_d. When the barrier is moved to outside the loop (refer to the row $VC_{sd}lb$) the speedups are further improved, especially when the number of processors becomes large.

	2-p	4-p	8-p	16-p	24-p	32-p
LRC_d	2	3.67	5.07	3.66	2.38	1.70
VC_{sd}	2	3.81	6.88	11.12	12.58	12.16
$VC_{sd}lb$	2	3.8	6.93	11.81	15.01	16.04

Table 3: Speedup of IS on LRC_d and VC_{sd}

5.2 Gauss

The matrix size of Gauss is 2048×2048 and the number of iterations is 1024 in our tests. The original *Gauss* program has the false sharing effect. The VOPP version has significantly improved the performance by removing the false sharing effect with local buffers. Table 4 shows the number of diff requests in VC_d is significantly smaller than that of LRC_d due to the removal of the false sharing effect.

	LRC_d	VC_d	VC_{sd}
Time (Sec.)	38.7	13.2	10.2
Barriers	1027	1028	1028
Acquires	0	17330	17295
Data (MByte)	255	21	20
Num. Msg	184517	119346	88521
Diff Requests	44145	15360	0
Barrier Time (usec.)	7080	3586	3610

Table 4: Statistics of Gauss on 16 processors

Even though there is an overhead for copying data between the shared memory and the local buffers (as mentioned in Section 3.1), there is a significant advantage by processing the data in the local buffers instead of in the shared memory. Due to the use of local buffers for infrequently-shared data, the work for consistency maintenance (e.g. diff requests) is greatly reduced and accordingly the amount of data and the number of messages are significantly reduced (refer to the rows $Data$ and $Num.Msg$ in Table 4).

Table 5 shows the speedups of LRC_d and VC_{sd}. The speedups of VC_{sd} is really impressive compared with those of LRC_d.

	2-p	4-p	8-p	16-p	24-p	32-p
LRC_d	1.9	3.08	3.5	2.5	1.84	1.44
VC_{sd}	1.98	3.75	6.55	9.42	9.13	8.3

Table 5: Speedup of Gauss on LRC_d and VC_{sd}

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* has infrequently-shared data which mixes with frequently-shared data. The VOPP version uses local buffers for those infrequently-shared data to reduce the false sharing effect. Furthermore, it uses shared memory (a set of views) for those frequently-shared data such as the border elements. Due to the wise allocation of shared memory and local buffers, the amount of data transferred in the cluster network is very small and accordingly the VOPP program performs better than the original *SOR* program. Table 6 shows the amount of data transferred in LRC_d is 65.57 Megabytes while the amount in VC_d is reduced to 2.99 Megabytes. The number of messages in VC_d is also reduced.

	LRC_d	VC_d	VC_{sd}
Time (Sec.)	11.2	4.3	4.12
Barriers	102	102	102
Acquires	0	6030	6030
Data (MByte)	65.57	2.99	3.37
Num. Msg	45,471	33,144	21,152
Diff Requests	5907	5996	0
Barrier Time (usec.)	139,100	3738	3483

Table 6: Statistics of SOR on 16 processors

Another factor contributing to the better performance of the VOPP program is faster barrier implementation in VC (as mentioned in Section 3.3). From Table 6, the average barrier time in VC_d is 3738 microseconds, while the barrier time in LRC_d is 139,100 microseconds.

Table 7 shows that the speedups of the VOPP program running on VC_{sd} is greatly improved compared with the original program running on LRC_d.

	2-p	4-p	8-p	16-p	24-p	32-p
LRC_d	1.65	2.67	3.7	4.45	4.47	4.33
VC_{sd}	1.98	3.81	6.96	11.43	14.1	14.75

Table 7: Speedup of SOR on LRC_d and VC_{sd}

5.4 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. The VOPP version of *NN* uses local buffers for infrequently-shared data and *acquire_Rview* for read-only data. The *acquire_Rview* for read-only data is very important for the VOPP program. Without it the major part of the VOPP program would run sequentially. Table 8 shows VOPP itself does not demonstrate any performance advantage in *NN* since VC_d sends more messages and data than LRC_d due to more view primitives used in the VOPP program. Thus VC_d is slower than LRC_d. However, the performance potential offered by VOPP to DSM implementation becomes larger when more view primitives are used. Table 8 shows VC_{sd} performs significantly better than LRC_d. The number of messages and the amount of data transferred in the cluster network are greatly reduced in VC_{sd} due to diff integration and diff piggy-backing.

	LRC_d	VC_d	VC_{sd}
Time (Sec.)	114	119.4	54.07
Barriers	473	473	473
Acquires	7520	22,371	22,371
Data (MByte)	335	376	64.7
Num. Msg	101,919	161,400	81,590
Diff Requests	31,228	39,900	0
Barrier Time (usec.)	122,324	147,389	13,141
Acquire Time (usec.)	2555	21,527	3872

Table 8: Statistics of NN on 16 processors

Table 9 presents the speedups of LRC_d and VC_{sd}. The table shows the speedups of the VOPP version of *NN* are significantly improved by VC_{sd}. In order 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 9. The performance of the VOPP program is comparable with that of the MPI version on up to 16 processors. On more than 16 processors, the speedup of the VOPP program 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.

	2-p	4-p	8-p	16-p	24-p	32-p
LRC_d	1.98	3.93	7.1	6.45	4.02	2.54
VC_{sd}	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 9: Speedup of NN on LRC_d, VC_{sd} and MPI

6 Conclusions

This paper presents a novel VOPP programming style for DSM parallel programs on cluster computers. Several applications are converted and optimized based on the requirements of VOPP. Our experimental results demonstrate the significant performance advantage of VOPP and its great performance potential offered to DSM implementations. VOPP is based on shared memory and is easy for programmers to use. It only requires programmers to insert view primitives when a view is accessed. The insertion of view primitives can be automated by compiling techniques, which will be investigated in our future research. We will also investigate the reasons behind the performance difference between VOPP programs and MPI programs and will develop more efficient implementation techniques for the associated VC model. 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] 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] 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., 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.: View-Oriented Parallel Programming on Cluster Computers. Technical Report (OUCS-2004-09), Dept of Computer Science, Univ. of Otago, (2004) (http://www.cs.otago.ac.nz/research/techreports.html)

[7] 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.

[8] 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)

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

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

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

[12] 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