Maotai[*]: View-Oriented Parallel Programming on CMT processors

Jiaqi Zhang† Zhiyi Huang‡ Wenguang Chen† Qihang Huang‡ Weimin Zheng†
†Department of Computer Science
Tsinghua University, Beijing, China
Email:zhang-jq06@mails.tsinghua.edu.cn, {cwg;zwm-dcs}@tsinghua.edu.cn
‡Department of Computer Science
University of Otago, Dunedin, New Zealand
Email:{hzy;tim}@cs.otago.ac.nz

Abstract

View-Oriented Parallel Programming (VOPP) is a novel parallel programming model which uses ***views*** *for communication between multiple processes. With the introduction of views, mutual exclusion and shared data access are bundled together, which offers both convenience and high performance to parallel programming. This paper presents the implementation of VOPP on Chip-Multithreading processors, e.g. UltraSPARC T1. We demonstrate that our implementation of VOPP on multi-core platforms (namely Maotai) shows significantly better performance than directly applying the original DSM implementation of VOPP(namely VODCA) on our platform. Besides, we compare the performance of VOPP with MPI and OpenMP. The experimental results demonstrate that VOPP has better scalability than both MPI and OpenMP on our platform.*

Key Words: *Chip-Multithreading, View-Oriented Parallel Programming, OpenMP, Message Passing Interface, Distributed Shared Memory (DSM)*

1. Introduction

Computer architectures and the computer industry are being transformed by the advent of multi-core and Chip-Multithreading (CMT) technologies [18]. These technologies offer massive increase in processing capacity on a single computer and open new opportunities for system- and application-level software. With conservative estimation, in the near future there will be hundreds or even thousands of cores in a single, economical chip [4]. Thus parallel programmers are challenged with the task of utilizing this computer power by writing efficient parallel programs. In this sense, parallel programming models and related environments become more important to the programmers.

To facilitate programmability, the underlying parallel programming models should be friendly to programmers and thus help increase the productivity. On the other hand, it should be scalable in order to guarantee a fairly good speedup on multiple processors. Traditionally, there are two camps in parallel programming methodologies. One is based on message passing such as MPI, and the other is based on shared memory such as OpenMP. Parallel programming with message passing is commonly known as difficult and complex. Programmers are burdened with the task of orchestrating inter-process communication through explicit message passing. While MPI is often a *de facto* standard for distributed memory systems due to its high performance, it is less efficient for shared memory systems due to the overhead of data transfer.

Using shared memory for communications between processes is natural and straightforward for parallel programmers, but the problems such as data race and deadlock hinder parallel programming with shared memory. Recently, OpenMP became a *de facto* standard for shared memory environments. However, it suffers from performance penalties due to the fork-join pattern in its compiler generated code. Also it has difficulties to express certain patterns of parallelism, as to be discussed in Section 2.2.

View-Oriented Parallel Programming (VOPP)[8, 10] is a recently proposed parallel programming model which has demonstrated its high performance on cluster computers[9]. While all previous papers show the advantages of VOPP on DSM systems, in this paper, with the CMT technology of UltraSPARC T1 (aka Niagara)[3], we present our implementation of VOPP on multi-core systems (namely Mao-

[*]Maotai, is arguably the most famous Chinese liquor, and has a history of about 300 years.

tai), which shows significantly better performance than directly applying the original DSM implementation of VOPP (namely VODCA) [2, 9]. Besides, we compare the performance of the above three models and make detailed discussions in terms of both programmability and performance.

This paper has the following contributions. First, we present the first implementation of VOPP on multi-core processors–Maotai, which provides an alternative parallel programming environment for shared memory systems. We also discuss the difference in performance of Maotai and VODCA on our CMT platform. Second, we use four applications written in VOPP, MPI, and OpenMP to compare the performance of these three parallel programming styles on a CMT system. Third, we give a detailed analysis on the differences between VOPP and the other two popular parallel programming environments. The analysis is based on both experimental results and programmability.

The rest of this paper is organized as follows. Section 2 briefly describes the VOPP programming style and compares it with that of MPI and OpenMP. In Section 3, we introduce the implementation of Maotai on CMT. Section 4 presents the performance results and analysis. Finally, our future work is suggested in Section 5.

2. View-Oriented Parallel Programming (VOPP)

In VOPP, shared data is partitioned into views. A view is a set of memory units (bytes or pages) in shared memory. Each view, with a unique identifier, can be created, merged, or destroyed at any time in a program. Before a view is accessed (read or written), it must be acquired (e.g., with *acquire_view*); after the access of a view, it must be released (e.g. with *release_view*). The most significant property for views is that they do not intersect with each other (refer to [8] for details).

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 lock-based parallel programming. Mutual exclusion is automatically achieved when a view is acquired using *acquire_view*. In this way, mutual exclusion and data access are bundled together.

Some programming interfaces that bundle mutual exclusion and data access have also been proposed [5, 12, 13]. CRL (C Region Library)[13] focuses on low-level memory mapping, and limits a region to contiguous memory space. In contrast, a view in VOPP is a higher level shared object whose memory space may be non-contiguous, e.g., Automatically Detected Views [8]. Entry Consistency (EC)[5] and Scope Consistency (ScC) [12] also bundle mutual exclusion and data access like in VOPP. However, their programming interfaces are very different from VOPP (refer to [9] for details).

Bundling mutual exclusion and data access together is a convenient way for parallel programming. It has the following advantages: First, programmers can be relieved from data race issues. In VOPP, when a view is acquired, mutual exclusion is automatically achieved, so it is not possible for other processes to access the same view at the same time. If a view is accessed without being acquired, either the programmer can be notified of the problem by the compiler with some VOPP related support, or the runtime system can report the problem with the support of the underlying virtual memory system. Second, debugging is more effective. In VOPP, views are the only shared data between processes. Since views can be tracked down with view primitives, they can be easily monitored by a debugger while a program is running. Third, since the memory space of a view can be known, view access can be made more efficient with cache prefetching technique such as helper threads [14, 15, 17].

2.1. Comparison with MPI

MPI is different from VOPP in that it is based on message passing. Despite its difficulties in programming, MPI is very suitable and effective to utilize the computing power of distributed computers because of the user-controlled data transfer.

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 that view primitives are used whenever shared memory is accessed). At the same time, VOPP provides experienced programmers an opportunity to finely tune the performance of their programs by carefully dividing the shared data into views and thus offers the potential to make VOPP programs perform as well as MPI programs even on clusters. A view in VOPP can be regarded as a message with transparent location, and therefore a VOPP program can be finely tuned so that its behavior can match that of its MPI counterpart. That is, a VOPP program can imitate the MPI program in a way that wherever there is data sharing through message passing between processors in the MPI program, the VOPP program can allocate a view for the shared data. We have demonstrated that the performance of VOPP is comparable to that of MPI on cluster computers[9, 10].

The shared memory model has been attracting more and more attention with the advent of CMT processors, which provide physical shared memory and shared caches. Since all processes share the same physical memory, the high overhead of maintaining memory consistency that hinders the speedup of parallel programs on DSM can be entirely removed as discussed in Section 3. That means, besides a guaranteed much better programmability, they can even overwhelm the message passing model in terms of performance. To demonstrate the difference in programming style, Figure1(a) and 1(b) show a typical producer/consumer problem written in both VOPP and MPI. In these programs, a master process produces the tasks, which are later distributed to consumers. In VOPP, this work is done in a straight forward manner: the master produces the tasks, which are later acquired by consumers. In contrast, the master in MPI has to send all the tasks one by one using send/recv primitives, which not only incur complexity in programmability but also greatly hinder the performance. What's more, collective operations can hardly be used here since the tasks are usually not stored in a contiguous space. Note that the *acquire_Rview* primitive in Figure1(a) is acquiring a view for read-only accesses.

```
if (0==proc_id) {
        aquire_view(view_id);
        /*produce the data*/
        release_view(view_id);
}
barrier(bar_id);
acquire_Rview(view_id);
/*do something with the data*/
release_Rview(view_id);
```

(a) VOPP style program

```
if (0==rank) {
        /*produce the data*/
        for (i=1; i<nprocs; i++) {
              send(data,i);
        }
}
if (0!=rank) {
        recv(data,0);
}
/*do something with the data*/
```

(b) MPI style program

Figure 1. producer/consumer program written in VOPP and MPI

These two simple programs are also used to test the extra overhead of data transfer in MPI on shared memory systems. Their performance results are shown in Section 4.4.2, which suggests that VOPP is more scalable than MPI on multi-core systems.

2.2. Comparison with OpenMP

As a popular shared memory model, OpenMP has been appreciated due to its ease of use. For certain types of programs, a few OpenMP directives will "magically" turn a sequential program into a parallel one.

Although both OpenMP and VOPP are based on shared memory, they are different in methodology. In OpenMP, everything is shared by default, and threads that share the whole memory are maintained dynamically. However, VOPP emphasizes the isolation of data, and the processes are forked initially and run in parallel through out the whole program.

The above parallel model in OpenMP brings performance penalties on shared memory architectures. On a DSM system, the unnecessary shared memory incurs severe false sharing problems that lead to heavy network traffic. Although efforts are made to extend OpenMP for cluster computers[7], the result is still not satisfactory. Although this problem is not so significant on physical shared memory, it still suffers due to maintaining threads dynamically. While this overhead can be amortized in coarse-grained parallelism, it becomes prominent for fine-grained parallelism and for applications with small data size. This problem is demonstrated in Section 4.3. In contrast, while VOPP does use shared memory for communications between processes, it emphasizes the use of private memory whenever possible. By creating and acquiring views explicitly, programmers are reminded of the cost of data sharing and are discouraged of using unnecessary shared data. This philosophy helps VOPP achieve a high performance on both DSM and CMT platforms as illustrated later.

```
do{
   /*calculate with data in current
node*/
   /*calculate whether perform
pruning*/
   if (!prun){
        node−>rtree=new_rnode;
   }
   node=node−>ltree;
}while(searchnotfinished);
```

Figure 2. Generation of a tree with pruning

```
bucksort(){
   for(i=0; i<count; i++)
       key_den[key[i]]++;

   for(i=1; i<MAXKEY;i++)
       key_den[i]+=key_den[i−1];

   for(i=0;i<count;i++)
       rank[i]=−−key_den[key[i]];
}
```

Figure 3. A counting sort algorithm

Despite its effort in easing the burden of parallelization, OpenMP still has disadvantages in programmability:

Some programs cannot be parallelized in OpenMP conveniently. A classical example in artificial intelligence is the search of a decision tree. Pruning is used while the tree is explored. Pruning could largely reduce the computation, but it results in an unpredictable amount of work to do, and thus brings data dependencies in the program. A typical code pattern for such a program is shown in Figure 2. With VOPP, however, we can parallelize the program in Figure 2 based on a producer/consumer pattern. When an rtree is identified, it is put into a shared task queue. And all processes look up the queue for new tasks. Although this problem is supposed to be solved in the upcoming OpenMP Specification3.0, it will still affect the programmability negatively since more new concepts will make it a more complicated language.

Another example is a counting sort program in Figure 3. This program aims to compute the rank of each integer in an array *key[]* . At first glance, the program looks perfect for OpenMP because it has three *for* loops. However, by carefully examining the behavior of the program, we find the second loop cannot be parallelized directly with OpenMP directives due to inherently sequential dependency. Furthermore, the first loop cannot be parallelized as well, because *key[i]* is a random value and accessing *key_den[key[i]]* concurrently incurs data races when there are many repeated keys in the problem set such as in Section 4.3. While the third loop has the same data race problem if parallelized, the data races in that loop only affect the rank of the integers of the same value. With VOPP, we can parallelize the program by dividing the key array into several parts. After the first and the second loop are done by each process, all processes work in parallel to construct a shared *key_den* using the values of their local *key_den* array.

The uncontrolled shared data may lead to severe correctness problems. From the example above, we can realize that novice OpenMP programmers may easily get such programs parallelized incorrectly by directly applying OpenMP directives. The implicitly shared data may prevent programmers from realizing that they are accessing shared data concurrently, which is the very source of data races. And the seemingly easy OpenMP interfaces have incurred a lot of traps that may result in correctness problems for new parallel programmers, as listed in [19].

Of course, the program in Figure 2 can be parallelized by hardcoding the parallel code with parallel sections. However, in this way, it falls back to the traditional lock-based model, which exposes the problems such as data race conditions to the programmers and is not what OpenMP is supposed to advocate. NPB[1] has efficiently implemented Integer Sort based on counting sort using OpenMP. However, the algorithm NPB adopts is far from being straight forward to novice parallel programmers.

3. Implementation of Maotai

In Maotai, we have implemented VOPP primitives [8] in Linux kernel 2.6.20 running on UltraSPARC T1. They are implemented as a kernel module supporting a shared memory device. This device provides both shared memory and synchronization mechanisms for VOPP. UltraSPARC T1 has eight cores, each of which can support four hardware threads. In total, it can support up to 32 simultaneous threads. There is a 12-way 3MB L2 cache on the chip, shared by all cores. Each core has a 16KB instruction cache and a 8KB data cache (L1 caches) and is clocked at 1.0GHz [3].

Maotai is implemented with multi-processing support. Multiple processes are created when a program is started with the primitive *Vdc_startup*. We prefer multi-processing to multi-threading, because we believe independence and isolation are better than sharing in parallel computing. In the same line, we encourage more independence and isolation than sharing in VOPP. With multi-processing, we can keep the sharing of data among processes to the minimal, since the sharing of data in VOPP programs can only be achieved through views. In contrast, threads have lots of unnecessary sharing which expose programs to potential problems like data race condition. Besides, the overhead of multi-processing has been much reduced with Light-Weight Process (LWP) and Copy-On-Write (COW) techniques.

On cluster computers, minimizing data sharing helps VOPP reduce large amount of data transfer and false sharing effect. This principle also benefits from the shared cache (L2 cache) on multi-core platforms. Since minimizing the shared data can reduce the footprint of the data in memory, the shared data can be more often kept in the cache instead of the RAM for fast accesses.

Physical shared memory architectures such as CMT provide more suitable platforms for shared memory models and benefit VOPP in several ways:

- On CMT platforms, no data transfer is needed by a shared memory programming model,and processes directly access memory instead of transferring view data off-chip via TCP/IP stacks or even network cards.
- For mutual exclusion access of shared data, the consistency of shared memory is achieved atomically on physical shared memory and thus the consistency maintenance can be totally removed from the implementation.
- Logical shared memory on DSM requires each process has its own duplicate of the shared memory. This requirement is removed on physical shared memory.
- The initialization of VOPP can be much faster due to fork approach instead of starting via SSH remotely.

The performance gain brought by these features and the detailed analysis of the implementation are presented in Section 4.2 along with the experimental results.

4. Performance evaluation

The performance evaluation is divided into three parts. First, we use a micro-benchmark and four scientific applications to show how the two implementations of VOPP–Maotai and VODCA [2, 9]-differ in performance on CMT. Second, we use the four applications to compare the general performance of VOPP (i.e. Maotai) with MPI and OpenMP. In the third part, we perform experiments using micro-benchmarks to further compare VOPP against OpenMP and MPI in details.

4.1. Experimental Methodology

All performance tests are carried out on Sun Microsystems' T2000 server. The server has UltraSPARC T1 as its processor and 16GB RAM. Its operating system is Linux 2.6.20 for sparc64. We use a gcc version 4.2.1, which supports *-fopenmp* option to compile OpenMP programs. To compile and run MPI programs, we use MPICH2, which is configured with "ch3:shm" to avoid the overhead of TCP sockets on shared memory platforms. All the applications in this section are compiled with *-O2* optimization switch.

The four scientific applications we used are Integer Sort (IS), Gauss Elimination (GE), Successive Over-Relaxation (SOR), and Neural Network (NN). Since UltraSPARC T1 has only one floating point unit, we replaced floating point calculations with integer calculations without affecting the amount of computation of these programs, in order to avoid the bottleneck problem of the floating point unit. The details of these applications can as well be found on [10].

IS ranks an unsorted sequence of N keys using a bucket sort algorithm shown in Figure 3. For the OpenMP version, we used NPB IS. In our test, the problem size is 2^{26} integers with a *Bmax* of 2^{15}, and 40 iterations are performed.

GE implements the Gauss Elimination algorithm in parallel. In our test, the matrix size is $4000 * 4000$.

SOR uses a simple iterative relaxation algorithm with a two-dimensional grid as input. Every element is updated to a function of its neighbors' values in each iteration. In our test, a matrix with a size of $8000 * 4000$ is processed in 40 iterations.

NN trains a back-propagation neural network in parallel using a training data set. In our test, the size of the neural network is $9 * 40 * 1$ and the maximum number of epochs is 200.

4.2. Maotai vs. VODCA

To analyze how Maotai is different from VODCA, we run both implementations on our T2000 server.

Table 1. Execution Time of Producer/Consumer

	Total	Startup	Acquire	Release	Barrier	Compute
VD	17506859	5437667	10064528	1805	1752658	249223
MT	322014	5966	329	318	90981	223688

Table 1 shows the execution time (in milliseconds) of each part in producer/consumer program presented in Figure 1 using 10 processes. In the table, VD and MT stand for VODCA and Maotai respectively. From the table we can see that VODCA spends large amount of time on the *Startup* process. This is because despite the other processes being also on localhost, it has to start them via SSH. The extra time cost on *Acquire* in VODCA is two fold: data transfer via off-chip TCP/IP stack and consistency maintenance. In VODCA, consistency is maintained efficiently by transferring only diffs [11], which is calculated and merged every time when changes are made to views. The diffs are transferred when a process acquires a view, and then is decoded and applied to the local memory by acquiring process, which works very well for DSM. However, the above overhead of transferring and coding/decoding diffs can be eliminated in Maotai as suggested in Section 3, and thus a significant performance gain is achieved. Similarly, the overhead of *Barrier* is also reduced in Maotai since barriers are implemented with shared variables instead of message passing as in VODCA.

More interestingly, Maotai outperforms VODCA in the computing process. This seems to be counter-intuitive because the amount of computation should be always the same. However, when looking into the implementation of VODCA, we shall see that diffs are created by means of page fault handling. During the computing process, whenever a process is trying to modify or read a protected page, a page fault handler is invoked. This overhead has been removed from Maotai, which contributes to the faster computing process of Maotai.

In the following discussion, we ignore the time difference caused by the procedure of *Startup* because, although it occupies a lot of time, the overhead is constant when the number of processes is fixed.

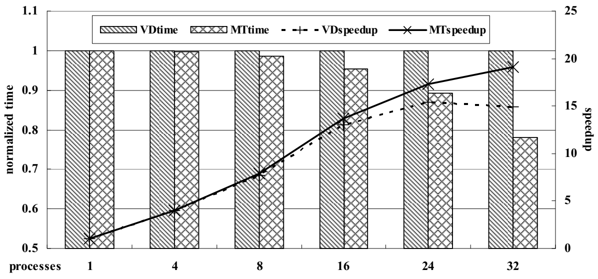


Figure 4. Execution time of IS

Figure 4 demonstrates the execution time and speedup of IS on Maotai and VODCA. The execution time is normalized according to VD_{time}, which is the execution time of VODCA. It is obvious that more performance gain can be achieved when we have more processes. This is consistent with the discussion above, as more processes incur more data transfers and more work to manipulate consistency in VODCA. The speedup curve also shows that the unnecessary overhead severely affects the scalability of VODCA. In contrast, more processes can still help Maotai while VODCA shows a decreased speedup on 32 processes.

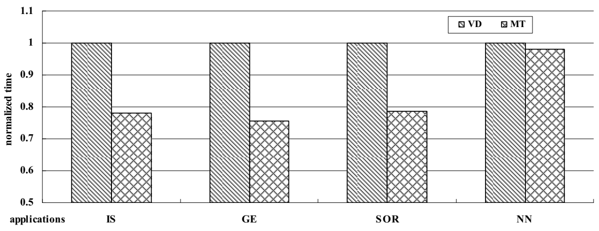


Figure 5. Execution time of 4 applications

Figure 5 is the normalized execution time of the four applications introduced in Section 4.1 with 32 processes on Maotai and VODCA. The results show significant performance gains of Maotai from 22% to 25% for IS, GE, and SOR. However, the improvement of NN is not so prominent (only about 2%). This is due to the small amount of shared memory used in NN. As discussed above, since it is the consistency maintenance of shared memory that incurs the overhead of data transfer in VODCA, the benefit of removing the overhead is not so significant when there is little shared memory.

4.3. VOPP performance against MPI and OpenMP

In this part, we compare the performance of VOPP (i.e. Maotai) against MPI and OpenMP on CMT with the four applications described above. All speedups are calculated with respect to the sequential code of the application. The speedup of IS, GE, SOR, and NN are shown in Figure 6, Figure 7, Figure 8, and Figure 9 respectively.

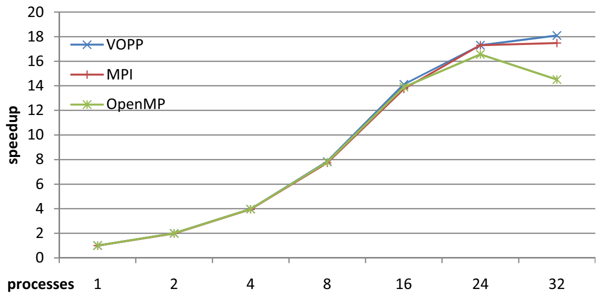


Figure 6. Speedup of IS

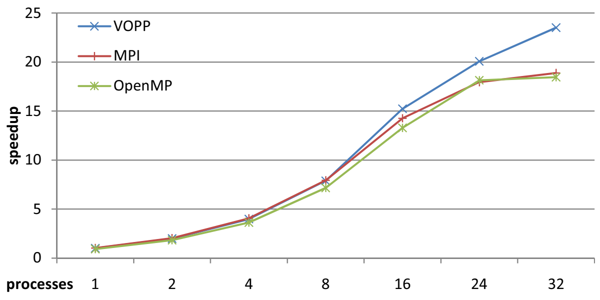


Figure 7. Speedup of GE

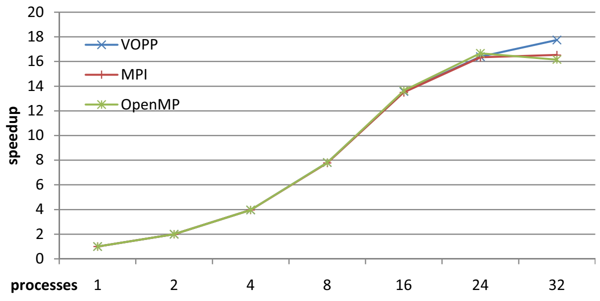


Figure 8. Speedup of SOR

The performance results of these applications show that VOPP outperforms both MPI and OpenMP, especially when

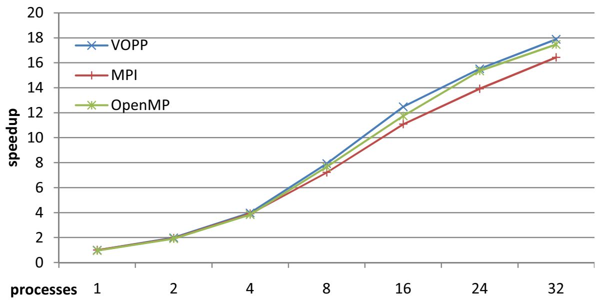


Figure 9. Speedup of NN

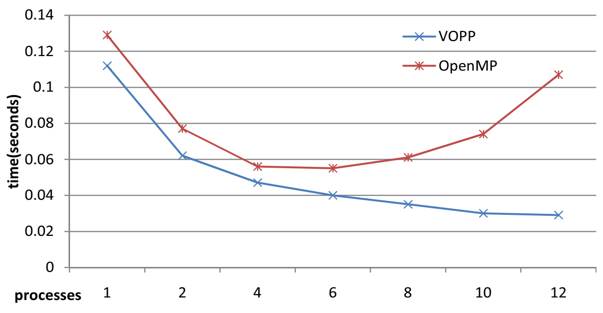
the number of processes is large. The results are consistent with the prior discussions in Section 2. For example, when running on 32 processes, VOPP performs up to 20% better than both MPI and OpenMP in GE. The program pattern in GE is similar to the producer-consumer pattern in Section 2.1. In each iteration of the computation, a process first calculate the pivot row, which is then transferred to all the other processes to compute their own rows. And the transfer of data in MPI, although via memcpy on shared memory platforms, severely affects its performance. OpenMP also performs worse than VOPP in GE due to the overhead of dynamically maintaining threads with fork-join patterns. Although we have optimized the OpenMP program by merging different loops into one parallel section, there are still many fork-joins because of the large number of iterations in the outer loop. From the figures, we can also notice that while maintaining a good scalability, even with a simple implementation, VOPP incurs rare overhead when running on single process. The overhead discussed above can also be applied to other applications, whose overheads differ in effects due to their different data sharing patterns. These overheads are amortized in the programs when the granularity of parallelism is large. They are more prominent for fine-grained parallelism.

4.4. Detailed Performance Analysis

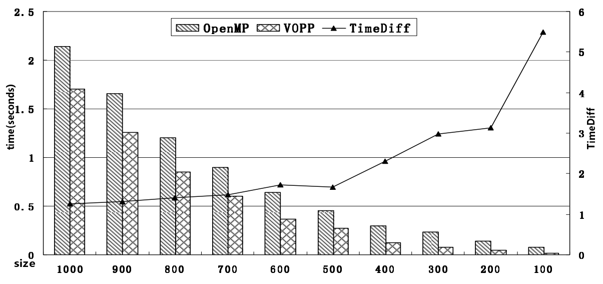
To identify how VOPP, MPI, and OpenMP differ in performance more clearly, we demonstrate their performance on some micro-benchmarks and specific data sizes in the following sections.

4.4.1. Comparison with OpenMP for fine-grained parallelism

To illustrate the performance difference of OpenMP and VOPP with fine-grained parallelism, we show the running time of the GE application with various number of processes working on a matrix of size $200 * 200$ in Figure 10(a). Also we show the running time of GE with various matrix sizes ranging between $100 * 100$ and $1000 * 1000$ using 16 processes in Figure10(b).

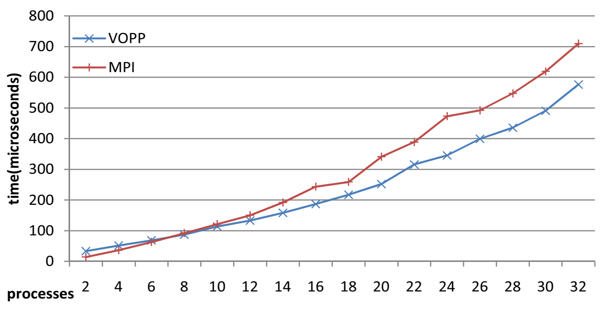


(a) different process number, 200*200

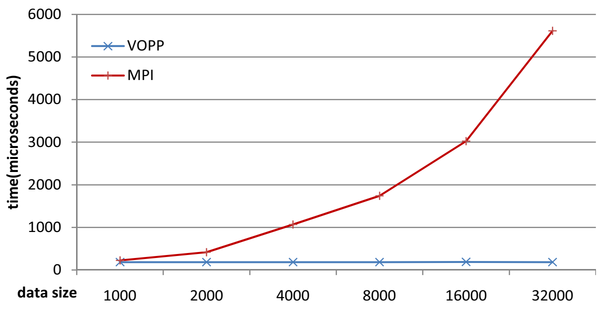


(b) different data size

Figure 10. Performance Comparisons



(a) different process number



(b) different data size

Figure 11. Performance Comparisons

We can see from Figure 10(a) that not only VOPP incurs little overhead on a single process, but it also reaches its peak of speedup much later than OpenMP when dealing with small data sizes. From Figure 10(b), we can also find that when we decrease the problem size, the time cost by OpenMP is decreasing more slowly than VOPP, and the performance difference between VOPP and OpenMP becomes larger. The performance difference presented by the curve is calculated using the time of OpenMP divided by the time of VOPP. When we perform this test on a 100*100 matrix, OpenMP is 4.5 times slower than VOPP. Therefore, VOPP is more scalable than OpenMP for finer-grained parallelism.

4.4.2. Overhead of data transfer in MPI

The major reason of performance difference between MPI and VOPP is that MPI requires memory copy operations to complete send/recv, which increase in number with the number of processes and data sizes. Figure 11(a) and 11(b) show the time cost of the producer/consumer program mentioned in Section 2.1. They depict the running time for varied number of processes and varied data sizes, respectively. Note that we only demonstrate the time cost of data sharing between the producer and the consumers, which excludes the computation time.

We use 1000 integers as the shared data in Figure 11(a), where the increasing time cost of both programs is expected because synchronization overhead increases when the number of processes increases. The overhead of VOPP is attributed to the barriers synchronizing the processes. It makes VOPP slightly slower than MPI when there are less than 8 processes. However, VOPP shows better scalability when there are more processes due to the large number of send/recv operations in MPI.

Figure 11(b) shows the time cost with varied data sizes with 16 processes. The time cost of MPI increases significantly because it has to do more data transfers. However, there is no extra overhead for VOPP because there is no data transfer and the overhead of *acquire_Rview* is trivial and constant.

5. Conclusions and future work

We present an implementation of VOPP, called Maotai, on a latest CMT processor, UltraSPARC T1, and demonstrate the advantages brought by multi-core platforms to VOPP by showing that Maotai achieves much better performance than directly porting VODCA on CMT. We have also illustrated the differences and advantages of VOPP compared to two popular parallel programming models–MPI and OpenMP. Our experimental results show that VOPP is more scalable than MPI and OpenMP on CMT processors. VOPP outperforms OpenMP especially for fine-grained parallelism, and also outperforms MPI especially when there is large data sharing between processes such as in our producer/consumer problem.

In the near future, we would like to test these programming models on other multi-core architectures such as Intel

Core 2 and AMD multi-core processors.

With the view information of VOPP and the shared caches in CMT, there is a good chance that we can adopt the helper threaded prefetching [14, 16, 17], which could utilize otherwise idle cores for applications that cannot be massively parallelized. The basic idea is to load the view into cache when it is acquired. The features of CMT processor give us the opportunity to implement the helper in both loosely-coupled and tightly-coupled way [14]. Figure 12 shows the execution time of a sum program utilizing our current implementation of loosely-coupled helper(MT_{lc}) and tightly-coupled helper(MT_{tc}), along with VOPP without helper(MT_{non}) and OpenMP. Although there are still a lot of work to do to improve the helper in our future work, the preliminary results demonstrate the exciting potential to further improving the performance of VOPP programs.

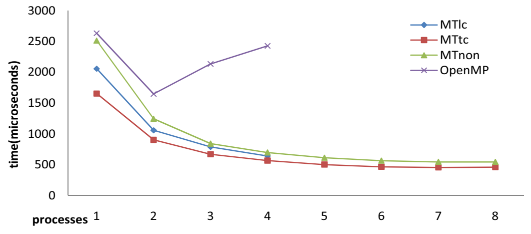


Figure 12. performance of helper threaded prefetching for VOPP

Since VOPP has demonstrated its promising performance on both CMT processors and cluster computers [9], a scalable parallel programming environment based on VOPP for multi-core clusters is desirable. This VOPP environment on multi-core clusters is promising to replace the current solution of combining OpenMP and MPI, which is both hard to program and error prone due to the two completely different models, as suggested in [6].

Acknowledgement

This work was supported by the Chinese National 973 Basic Research Program, 2007CB310900 and by the Nature and Science Fundation of China: NSFC 90718040.

References

[1] NAS Parallel Benchmarks. http://www.nas.nasa.gov.

[2] View Oriented Distributed Cluster-based Approach to parallel programming. http://vodca.org.

[3] UltraSPARC Architecture 2005 Specification. http://opensparc-t1.sunsource.net/, 2005.

[4] K. Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, December 2006.

[5] Bershad, B. N., Zekauskas, and M. J. Midway: Shared memory parallel programming with Entry Consitency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, 1991.

[6] Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *CCGRID*, pages 471–478. IEEE Computer Society, 2007.

[7] Jay P. Hoeflinger. Extending OpenMP to Clusters. White Paper, http://www.intel.com/, 2006.

[8] Z. Huang and W. Chen. Revisit of View-Oriented Parallel Programming. In *Proc. of the Seventh IEEE Inter. Symp. on Cluster Computing and the Grid*, pages 801–810, 2007.

[9] Z. Huang, W. Chen, M. Purvis, and W. Zheng. VODCA: View-Oriented, Distributed, Cluster-based Approach to parallel computing. In *Proc. of the IEEE/ACM Symp. on Cluster Computing and Grid 2006*, page 15, 2006.

[10] Z. Huang, M. Purvis, and P. Werstein. Performance Evaluation of View-Oriented Parallel Programming. In *Proc. of the 2005 Inter. Conf. on Parallel Proceessing (ICPP05)*, pages 251–258, June 2005.

[11] Z. Huang, M. Purvis, and P. Werstein. 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)*, pages 873–880, May 2005.

[12] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Conssitency. In *Proc. of the 8th Annual ACM Symp. on Parallel Algorithms and Architectures*, 1996.

[13] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP)*, volume 29, pages 213–226, 1995.

[14] C. Jung, D. Lim, L. Lee, and Y. Solinhin. Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems. In *Proc. of 20th IEEE Inter. Parallel & Distributed Processing Symp.*, 2006.

[15] D. Kim et al. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proc. of the 2004 Inter. Symp. on Code Generation and Optimization*, pages 27–38, 2004.

[16] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *the 10th Inter. Conf. on Architectural Support for Programming Languages and Operation Systems*, pages 159–170, 2002.

[17] J. Lu et al. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC CMP Processor. In *Proc. of the 38th Annual IEEE/ACM Inter. Symp. on Microarchitecture*, pages 93–104, 2004.

[18] L. Spracklen and S. G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proc. of Inter. Symp. on High-Performance Computer Architecture*, pages 248–252, 2005.

[19] M. Süß and C. Leopold. Common Mistakes in OpenMP and How To Avoid Them: A Collection of Best Practices. In *Inter. Workshop on OpenMP(IWOMP2006)*, 2006.