Department of Computer Science,
University of Otago

Technical Report OUCS-2008-01

View-Oriented Parallel Programming on CMT processors

Authors:

J. Zhang, W. Chen, W. Zheng
Department of Computer Science, Tsinghua University, Beijing, China

Z. Huang, Q. Huang
Department of Computer Science, University of Otago

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.html

View-Oriented Parallel Programming on CMT processors

J. Zhang† Z. Huang‡ Q. Huang‡ W. Chen† W. 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 performance results of VOPP on Chip-Multithreading processors, e.g. UltraSPARC T1. We have compared VOPP with MPI and OpenMP in terms of programmability and performance. An implementation of helper threaded prefetching for VOPP has also been discussed and evaluated.

Key Words: Chip-Multithreading, View-Oriented Parallel Programming, OpenMP, Message Passing Interface, Helper Threaded Prefetching

1 Introduction

Computer architectures and the computer industry are being transformed by the advent of multi-core and Chip-Multithreading (CMT) technologies [20]. 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 [2].

The challenge for us is how to efficiently utilize this computing power. This task will eventually fall on the shoulders of application programmers, who should make sure that their programs run correctly and efficiently on multiple processors. 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. A well designed, easy model will help increase the productivity largely. On the other hand, it should perform well in efficiency and scalability, which is needed to guarantee a fairly good

speedup for most applications. 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 which is used for communications between computing entities such as processes.

Parallel programming with message passing is commonly known as difficult and complex, especially when there are hundreds of processes communicating with messages. 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. The reason is that the advantage of message passing has turned out to be a potential disadvantage due to its overhead of data transfer in a shared memory system.

Using shared memory for communications between processes is natural and straightforward for programmers, but the problems such as data race and deadlock hinder parallel programming with shared memory. Recently, OpenMP becomes a *de facto* standard for shared memory environments because of its ease of use. However, it suffers from performance penalties due to the fork-join pattern in its compiler-automated code. Also it is not always convenient in programmability, as to be discussed in Section 2.2.

View-Oriented Parallel Programming (VOPP)[8, 11] is a recently proposed parallel programming model which has demonstrated its high performance on cluster computers[9]. This paper will show that, as a model based on shared memory, VOPP can achieve good performance on shared memory systems such as multi-core systems, besides its advantages in programmability.

In this paper, with the CMT technology of UltraSPARC T1 (aka Niagara)[1], we will compare the performance of the above three models and make detailed discussions in terms of both programmability and performance. Additionally, the unique features of VOPP enable us to adopt the idea of helper threaded prefetching [14], in order to reduce memory access latency of shared data.

This paper has the following contributions. First, we present the first implementation of VOPP on multi-core processors, which provides an alternative parallel programming environment for shared memory systems. 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. Fourth, we implement helper threads for prefetching data for parallel programs and give a performance evaluation and analysis of the helper threads.

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 VOPP on CMT with a helper threaded prefetching feature. 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, and 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.

The following classes of views are identified in [8] for parallel programming: Single-Writer View (which includes Consumable View and Atomic View), Multiple-Writer View, and Automatically Detected View.

There are a number of requirements for VOPP programmers. First, the programmer should partition shared data into a number of views according to the data sharing pattern of the parallel algorithm. Second, each view should consist of data objects that are always processed as an atomic set in the program. Third, when any data object of a view is accessed, view primitives such as *acquire_view* and *release_view* must be used (refer to [8] for details of the primitives).

VOPP allows programmers to participate in performance optimization through wise partitioning of shared data into views. Views can be carefully designed and tuned in order to reduce the communication overhead between processes. VOPP does not place any extra burden on programmers since the partitioning of shared data is an implicit task in parallel programming. This task is just made explicit in VOPP by adding view primitives, which renders parallel programming less error-prone in handling shared data.

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*.

Some programming interfaces that bundle mutual exclusion and data access have also been proposed [3, 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. Entry Consistency (EC)[3] 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 run-time 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. We will demonstrate this advantage shortly in this paper.

2.1 Comparison with MPI

MPI is different from VOPP in that it is based on message passing. Although MPI is difficult for programmers, it is very suitable and effective to utilize the computing power on computers that are connected by networks, such as cluster computers. Since it is the programmers' responsibility to perform the actual message passing, the overhead of data transfer can be minimized by carefully selecting the data to be transferred.

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). Like MPI, VOPP provides experienced programmers an opportunity to finely-tune the performance of their programs by carefully dividing the shared data 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 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 and uses view acquisition to get the data. In this way, the overhead of message passing for VOPP on distributed shared memory (DSM) can be almost the same as that in MPI program, since the cost of view acquisition is almost the same as that of sending and receiving a block of data in MPI. We have demonstrated that the performance of VOPP is comparable to that of MPI on cluster computers[9, 11]. However, VOPP still suffers from performance penalties incurred by certain critical routines such as barrier[9], which is common for DSM on cluster computers.

Fortunately, 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. Therefore, shared memory models can take full advantages on these systems. That means, besides a guaranteed much better programmability, they can even overwhelm the message passing model in terms of performance. A typical producer/consumer problem written in both VOPP and MPI, shown in Figure1(a) and 1(b), can demonstrate their significant difference in programming style. In these programs, a master process produces the data, and then distributes it for other processes to consume. 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 4.3, 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 essentially different in methodology. In OpenMP, everything is shared by default. While this concept sounds straightforward for a shared memory model, it brings performance penalties especially on distributed shared memory (DSM). Although an effort is being made to extend OpenMP for cluster computers[7], its performance is still not satisfactory. 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 and its efficient View-Based consistency protocol help VOPP achieve a high performance on DSM.

As a shared memory model, OpenMP can bring net performance gain for many sequential programs on shared memory systems, such as multi-core systems. While sequential code like *for* loops can be parallelized by OpenMP compiler, there are programs that cannot be parallelized in such a convenient way. 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. Since it brings data dependency in the program, it is hardly parallelizable by OpenMP in the convenient way. A typical code pattern for such a program is shown in Figure 2.

```
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 typical bucketsort algorithm

With VOPP, 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. All processes look up the queue for new tasks. Since this problem is already under discussion in recent work on OpenMP, hopefully it will be solved in the upcoming OpenMP Specification 3.0.

Another example is a bucket 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 with OpenMP directives due to data 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. 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. From this example,

we also realize that novice OpenMP programmers may easily get such programs parallelized incorrectly by directly applying OpenMP directives. In fact, The seemingly easy OpenMP interfaces have incurred a lot of traps that may result in *correctness* mistakes for new parallel programmers[21]. 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.

Of course, the above two problems can also be addressed by OpenMP by hardwiring 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.

In terms of performance, OpenMP suffers from penalties due to spawning and 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 4.2.

3 Implementation of VOPP

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 in 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. There are four DDR2 channels with a total throughput of 23GB/s for accessing RAM [1].

3.1 Implementation principle

VOPP 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. By the way, the overhead of multi-processing has been much reduced with the Light-Weight Process (LWP) and Copy-On-Write (COW) techniques.

VOPP conforms to this principle of minimizing sharing. Every time there is a sharing of data, a view has to be created by the programmer. Every time a shared data object is accessed, view primitives have to be used. In this way, sharing is discouraged and the programmer is reminded to carefully budget the amount of sharing.

On cluster computers, minimizing data sharing helped VOPP reduce large amount of data transfer and false sharing effect [10]. 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.

3.2 Helper Threaded Prefetching for VOPP

In order to help VOPP programs tolerate the increasing gap of memory latency, we have tried to take advantages of the prefetching techniques [4–6, 14, 15, 17–19], which have received much attention recently with the advent of chip-level multithreading technology. To prefetch data accurately and efficiently, efforts are put into region selection, which identifies the appropriate regions to include a piece of helper code, and phase detection which identifies the right timing to run the helper code [17]. However, without the help of the above techniques, VOPP can provide the right information for both the prefetching regions and the prefetching timings. When a view is acquired, it is almost for sure that the memory space of the view is about to be accessed due to the view-oriented feature of VOPP. When a view is created with the *alloc_view* primitive, the address and the length of the memory space of the view are recorded. With this information, we can prefetch the memory space of a view at the view acquiring time.

For our view prefetching, we have tried to use the PREFETCH instruction provided by UltraSPARC T1. The instruction can prefetch a cache line to L2 cache each time it is executed. However, UltraSPARC T1 only allows three PREFETCH instructions in flight at the same time. This limitation renders it impossible to use them in *acquire_view* because a view can be very large and the PREFETCH instructions cannot preload them into cache in time.

An alternative solution is a helper thread that does prefetching for a task thread. Helper threaded prefetching is a technique which proved to be promising on multi-core and hyper-threading platforms [14, 15, 18]. With the help of the view information discussed above, a helper thread can adapt to the dynamic behavior of a running application. That means, it works effectively despite a different input data set each time an application is given. The communication between the helper thread and the task thread is achieved by a shared variable that contains the identifier of the view being acquired. In our implementation, we make the helper sleep in a wait queue initially. When a view is being acquired, the task thread wakes up the helper, which then checks the shared variable to find out which view should be prefetched.

In previous research work, helper threaded prefetching is used in both chip-level multiprocessors (CMP), which have multiple cores inside one chip, and Simultaneous Multi-Threading (SMT) [22] processors, which physically support simultaneous threads in a single core. The implementation of a helper with a hardware thread inside an SMT processor is called tightly-coupled helper, while the helper implemented with another core in a CMP is called loosely-coupled. It had been suggested that a tightly-coupled helper incurs the contention of the same core that is shared among multiple threads [14, 18]. However, a helper thread that is located in the same core as the task thread can actually help prefetch the data into the L1 cache which is much closer to the CPU than the L2 cache. Although the difference of the speed between the L1 cache and the L2 cache is not so significant as that between the L2 cache and the memory, there are chances that tightly-coupled helpers would provide further performance gain when we perform read access (e.g. *acquire_Rview*). Previous research work could not compare and evaluate both the loosely-coupled and the tightly-coupled approaches with experimental results. Fortunately, with the CMT technology in UltraSPARC T1, which supports both CMP and SMT, we can now evaluate them on the same architecture. The experimental results are shown in Section 4.4.

Using helper threads to increase the speed of task threads is a promising approach to high performance computing on multi-core systems. It can offer further performance gain on top of parallel computing technology. Since there will be hundreds (maybe thousands) of cores in a chip, a helper thread may well use some idle core to speed up a sequentially executed task thread.

4 Performance evaluation

The performance evaluation is divided into four parts. First, we use four applications to compare the general performance of VOPP (without use of the helper threads) with MPI and OpenMP. In the second part, we use the Gaussian Elimination problem with small data sizes to compare the performance of VOPP and OpenMP for fine-grained parallelism. In the third part, the producer/consumer programs shown in Figure 1 are used to demonstrate the overhead of data transfer in MPI. Finally, we use a sum program to demonstrate the performance of VOPP with the helper threads.

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. MPICH2 is used to compile and run MPI programs. All the applications in Section 4.1, 4.2 and 4.3 are compiled with *-O2* optimization switch. However, in Section 4.4, in order to make sure the compiler does not disturb the memory access pattern, we do not use any optimization provided by gcc.

4.1 VOPP performance

In this part of our evaluation work, the applications we use 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 correctness of these programs, in order to avoid the bottleneck problem of the floating point unit in T1. We only use up to 30 processes in our experiments in order to avoid interference from other system processes.

IS ranks an unsorted sequence of N keys using a bucket sort algorithm shown in Figure 3. 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, Bmax]*. In order to guarantee the effectiveness of the parallelization, we do not put any restriction on the order of the keys with same value. As discussed in Section 2.2, this approach makes it possible for OpenMP to parallelize the third loop, despite the potential data race which only affects the rank of the integers of the same value. In our test, the problem size is 2^{26} integers with a *Bmax* of 2^{15}, and 40 iterations are performed. The speedup of IS is shown in Figure 4.

GE implements the Gauss Elimination algorithm in parallel. In our test, the matrix size is $4000 * 4000$. The speedup of GE is shown in Figure 5.

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 its neighboring elements. We use local buffers for those infrequently-shared data in the VOPP program. In contrast, we use shared memory (a set of views) for those frequently-shared data such as the boundary elements shared by two adjacent processes. In our test, a matrix with a size of $8000 * 4000$ is processed in 40 iterations.

The speedup of SOR is shown in Figure 6.

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 or it reaches the max epoch number. In our test, the size of the neural network is $9 * 40 * 1$ and the maximum number of epochs is 200. The speedup of NN is shown in Figure 7.

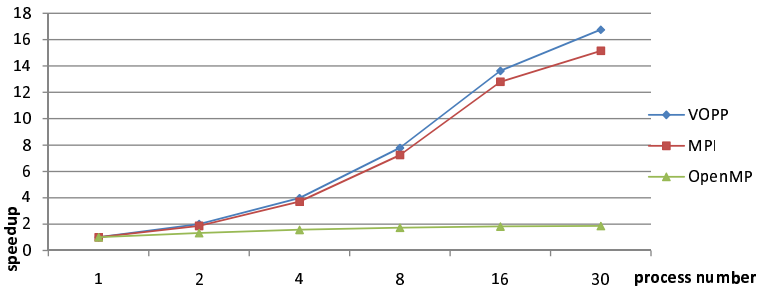


Figure 4: Speedup of IS

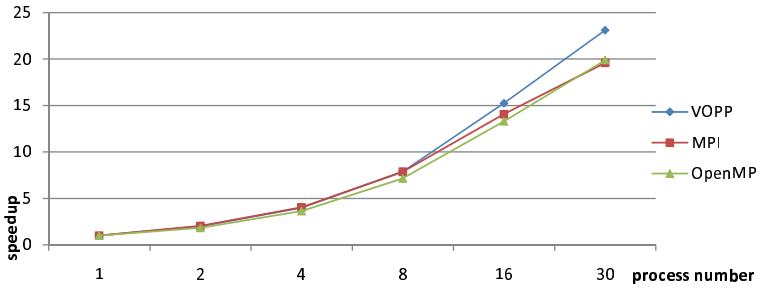


Figure 5: Speedup of GE

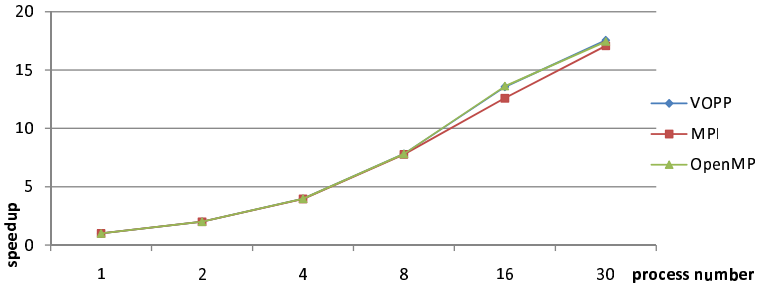


Figure 6: Speedup of SOR

The performance results of these applications show that VOPP outperforms both MPI and OpenMP, especially when the number of processes is large. The results are consistent with the prior discussions in Section 2. For example, for the GE application, VOPP performs up to 18% better than OpenMP and up to 16% better than MPI for 30 processes. The program pattern in GE is similar to the producer-consumer pattern in Section 2.1. Every time a process finishes the calculation of the pivot row, all other processes begin to process their rows with the pivot row. GE in MPI should thus transfer the pivot row to all other processes, which 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,

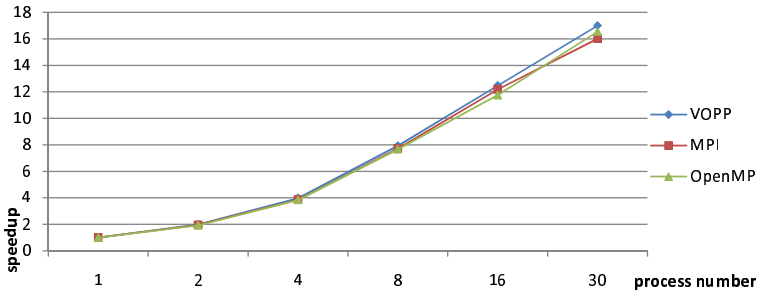


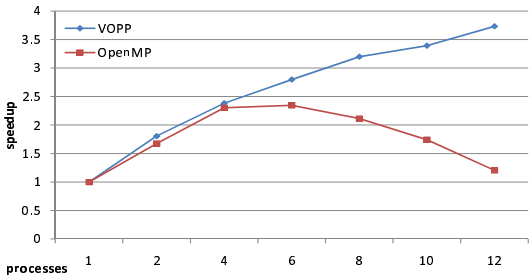
Figure 7: Speedup of NN

there are still many fork-joins because of the large number of iterations in the outer loop. The overhead of data transfer in MPI and the fork-join overhead in OpenMP can also be applied to other applications, which differ in these overheads due to the different data sharing patterns they follow. These overheads are amortized in the programs when the granularity of parallelism is large. They are more prominent for fine-grained parallelism, which will be shown shortly in the next section.

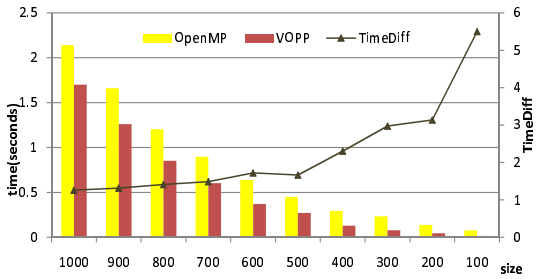
The only exception from the above results is IS in OpenMP. As discussed in Section 2.2, to follow the typical way of OpenMP programming, we can only parallelize the third loop, which makes the speedup of the program is lower than 2 due to Amdal's Law.

4.2 OpenMP for fine-grained parallelism

To demonstrate the performance problem of OpenMP 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 Figure8(a). Also we show the running time of GE with various matrix sizes ranging between $100 * 100$ and $1000 * 1000$ using 16 processes in Figure8(b). In order to make an optimal OpenMP program, we merge three separate loops into one parallel section so that OpenMP generates threads only once in one iteration.



(a) different process number, 200*200



(b) different data size

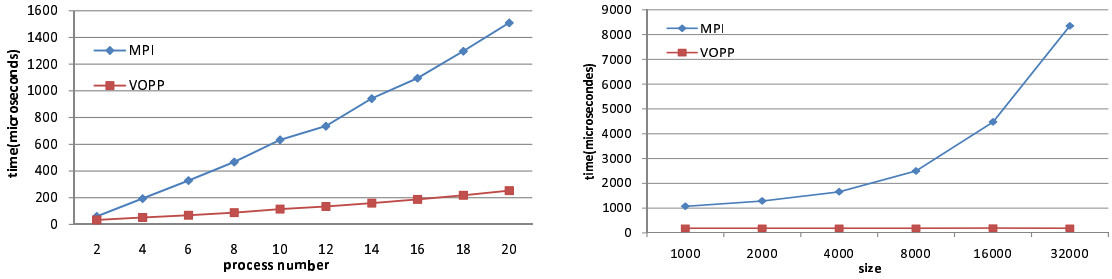
Figure 8: Performance Comparisons

We can see from Figure 8(a) that OpenMP reaches its peak of speedup much earlier than VOPP when dealing with the small data size. From Figure 8(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 time difference between VOPP and OpenMP becomes larger. The 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 fine-grained parallelism.

4.3 Overhead of data transfer in MPI

Figure 9(a) and 9(b) show the time cost of the producer/consumer program mentioned in Section 2.1. They depict the running time for various number of processes and various 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.



(a) different process number (b) different data size

Figure 9: Performance Comparisons

We use 1000 integers as the shared data in Figure 9(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, and is relatively small. However, the communication overhead for MPI becomes not negligible when the number of processes is large. Data transfer becomes a significant overhead in the MPI program when more processes are involved in message passing. We have tried to use a broadcast operation to reduce the overhead in the MPI program. However, the program using the broadcast operation performs even worse in our experiments, of which the reason is unknown yet.

Figure 9(b) shows the time cost with variable data sizes. 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.

4.4 Performance of the helper thread

To evaluate our preliminary implementation of helper threads, we divide our experiments into two parts. The first part involves cache misses and the second part involves general performance of the helper threads.

The benchmark program is a sum program in real world. It adds all the integers from a shared array. It is selected because it is a memory-intensive program that has a regular memory access pattern, which makes it an ideal program to show the effectiveness of helper threads due to its regular memory access pattern.

Since Linux dynamically schedules the processes to any physical cores, to perform our test, we have to bind the processes to specific physical cores with the *set_affinity()* system call in Sparc64 Linux.

In order to get accurate profiling of cache misses, the L2 cache and the L1 caches are thoroughly cleared before the computation starts. Since there are no libraries for performance profiling for UltraSPARC T1, we have to access directly the two performance counters, namely PIC and PCR, by calling the *perfctr()* system call in Sparc64 Linux.

Cache misses are shown in Table 1 and 2 for two data set sizes, 4K and 100K, which are used for the integer array in the sum program. The results in the tables are collected for the sum program running with one task thread and one helper thread (if helper threaded prefetching is used).

Helper	*task thread*			*helper*	
Type	*L1*	*L2*	*Tick*	*L1*	*L2*
V_{tc}	10	1	25685	245	64
V_{lc}	252	1	28484		64
V_{non}	252	63	34362		

Table 1: cache misses, 4K data

Helper	*task thread*			*helper*	
Type	*L1*	*L2*	*Ticks*	*L1*	*L2*
V_{tc}	482	131	510084	5875	1447
V_{lc}	6278	1	567828		1564
V_{non}	6278	1563	703808		

Table 2: cache misses, 100K data

In the above tables, V_{tc}, V_{lc}, and V_{non} stand for VOPP task thread with a tightly-coupled helper thread, VOPP task thread with a loosely-coupled helper thread, and VOPP task thread without any helper, respectively. The columns $L1$ and $L2$ are the L1 and L2 cache misses. The column *Ticks* is the number of CPU ticks cost by the task thread.

From Table 1 and 2, we can see that the helper thread can significantly decrease the CPU ticks of the task thread. Compared with V_{non}, the tightly-coupled helper can dramatically decrease both L1 and L2 cache misses, while the loosely-coupled helper can only decrease L2 cache misses.

However, when the data set size is larger, e.g. 100K in Figure 2, the count of L2 cache misses for V_{tc} is higher than that of V_{lc}, and its L1 cache misses is also increasing. This is largely due to the interference between the task thread and the helper thread competing for resources in the same core. Nevertheless, there is a significant performance gain by the tightly-coupled helper thread according to the CPU ticks, which is attributed to the decrease of L1 cache misses.

We can also notice that no matter whether we run the helper thread on the same core or not, the total number of L2 cache misses from both the task thread and the helper thread is larger than that of V_{non}. This also applies to L1 cache misses. The above result is expected due to two reasons. One is the interference between the two simultaneous processes in the same core. The other is the inaccurate prefetch performed by the helper.

Since our experimental results have shown that tightly-coupled helper threads can perform better for read accesses, we currently adopt the tightly-coupled approach for the *acquire_Rview* in our implementation of VOPP. However, since L1 cache is write-through, the tightly-coupled helper cannot provide the benefits mentioned above and will incur more contentions. Instead, we use loosely-coupled helpers for write accesses. The performance benefit from the reduced cache misses is reflected in the improved performance of the parallelized sum program, which is shown in Figure 10.

Figure 10 shows the performance of a simple sum program that adds up 25,000 integers randomly selected from an array of 100,000 integers. V_{tc}, V_{lc}, V_{non} have the same meaning as above. We only use up to 4 cores to perform this test for $VOPP_{lc}$, because there are only 8 cores in the T1 chip and thus we can only have 4 cores for the task threads while the other 4 cores are used by the helper threads. For comparison purposes, we show the time cost of the corresponding OpenMP program, which suffers from performance penalties due to the fine-grained parallelism in the sum program.

For VOPP with helper threaded prefetching, we can see a further improvement of performance.

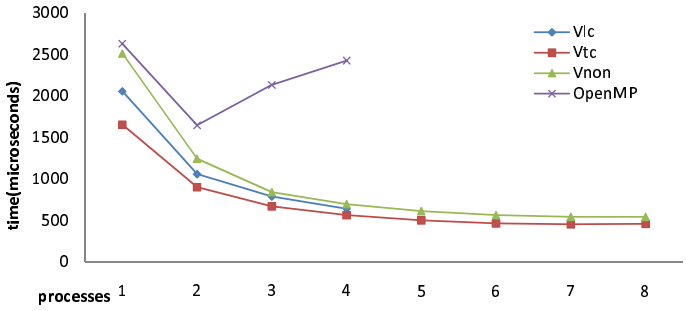


Figure 10: performance of helper threaded prefetching for VOPP

With tightly-coupled helper threads, VOPP achieves an even better speedup. However, when the number of processes is increasing, the performance gap between V_{non} and V_{lc} is decreased. This is expected because when the data loaded into each process becomes smaller, the helper threads are less effective in terms of cache prefetching. This also applies to the decreased performance gap between V_{tc} and V_{non}, which is also partially attributed to the high L2 cache misses of V_{tc} as mentioned above.

Due to limited time, we have not integrated the helper threads in other applications yet. Tests on other benchmark applications with helper threaded prefetching will be carried out in our future work.

5 Conclusions and future work

We have presented an implementation of VOPP on a latest CMT processor, UltraSPARC T1. We have shown 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 for fine-grained parallelism. It also outperforms MPI when there is large data sharing between processes such as in our producer/consumer problem. We have also adopted helper threaded prefetching in our implementation, which enables VOPP to achieve additional performance gain.

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. Since the performance of VOPP is better than OpenMP on CMT processors and comparable to MPI on cluster computers [9], an integrated parallel programming environment based on VOPP for multi-core clusters is desirable 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.

Current VOPP helper threads can only load memory for regular access patterns. They cannot guarantee a similar performance gain for applications with irregular memory access pattern. The cache contention of simultaneous processes is another problem that should be solved when the shared memory block is large. These factors can largely restrict the efficiency of prefetching in VOPP and a more efficient prefetching strategy may still rely on a more speculative approach [20]. We will address these issues with some compiler support using the techniques introduced in some related work [5, 14, 16].

Acknowledgment

The authors thank Mark Pethick for his excellent work on the implementation of shared memory device driver, on which our implementation of VOPP is based. We also thank David Miller for providing helpful advices on issues related to Sparc64 Linux.

References

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

[2] 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.

[3] 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.

[4] T.F. Chen and J.L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transations on Computers*, 44(5):609–623, May 1995.

[5] G. K. Dorai and D. Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single Thread Performance. In *Porceedings of International Conference on Parallel Architectures and Compilation Techniques 2002*, page 30, 2002.

[6] J. Dundas and T. N. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *International Conference on Supercomputing*, pages 68–75, 1997.

[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 *Proceedings of the Seventh IEEE International Symposium 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 *CD-ROM Proceedings of the Sixth IEEE/ACM Symposium on Cluster Computing and Grid*, 2006.

[10] Z. Huang et al. View-Based Consistency and False Sharing Effect in Distributed Shared Memory. *ACM SIGOPS Operating Systems Review*, 35(2):51–60, 2001.

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

[12] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Conssitency. In *Proceedings of the 8th Annual ACM Symposium 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 *Proceedings of the 15th ACM Symposium 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 *Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.

[15] D. Kim et al. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proceedings of the 2004 International Symposium 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 International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 159–170, 2002.

[17] J. Lee, Y. Solihin, and J. Torrellas. Automatically mapping code on an intelligent memory architecture. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 121–132, 2001.

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

[19] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998.

[20] L. Spracklen and S. G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of International Symposium on High-Performance Computer Architecture*, pages 248–252, 2005.

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

[22] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of International Symposium on Computer Architecture*, pages 392–403, 1995.