Technical Report OUCS-2009-01

Data Race: Tame the Beast

Authors:

Kai-Cheung Leung, Zhiyi Huang, Qihang Huang, Paul Werstein

University of Otago, New Zealand

Status:
submitted to P2S2, but rejected, material
reused in PDCAT2009 and SISC submissions (both are due on 1st June)

Data Race: Tame the Beast

K. Leung, Z. Huang*, Q. Huang, P. Werstein
Department of Computer Science
University of Otago, Dunedin, New Zealand
Email:{kcleung;hzy;tim;werstein}@cs.otago.ac.nz

Abstract

This paper proposes a data race prevention scheme, which can eliminate data races in the View-Oriented Parallel Programming (VOPP) model. VOPP is a novel shared-memory data-centric parallel programming model, which uses views to bundle mutual exclusion with data access. We have implemented the data race prevention scheme with a memory protection mechanism. Experimental results show that the extra overhead of memory protection is trivial in our data race free applications. We also present a new VOPP implementation–Maotai 2.0, which has advanced features such as deadlock avoidance, producer/consumer view and system queues, apart from the data race prevention scheme. The performance of Maotai 2.0 is evaluated and compared with modern programming models such as OpenMP and Cilk.

1 Introduction

Multicore and chip-multithreading (CMT) technologies are now becoming mainstream. These technologies allow multiple processors packed into a chip in a single computer, which often provides shared memory and cache [26]. However, parallel programming with shared memory can be prone to errors such as data race, which is difficult to debug due to its non-determinism and thus can severely affect programmability.

In a parallel multithreaded computation, a data race occurs if concurrent threads access the same memory location without mutual exclusive primitives such as locks, and at least one of the threads writes to the location. There have been many studies on debugging data races. Some perform a post-mortem analysis based on program execution traces [7, 10, 12, 20, 21], while others perform on-the-fly analysis during program execution [2, 9, 19, 25]. Among modern shared-memory parallel programming models [6, 8, 22, 24], only Cilk++ [8] provides a data race detector called Cilkscreen [2, 8, 15].

Even though race detectors can help debug some data races, they often have the following problems.

- Race detectors are often expensive to run, both in terms of computation and memory space. For example, Cilkscreen can take up to 30 times the normal execution time of the debugged program to run and the memory footprint can be "several times" the memory footprint of the original application[8].
- Race detectors can only detect data races for one given input of a program. If data races do not occur when the program is run with a given input, this does not imply the program is data race free. The reason is that a different input may result in threads being executed in different order, and the resultant interaction may cause data races.
- To a novice programmer, race detectors can be difficult to use. For example, Cilkscreen gives a detailed trace of memory addresses and their associated function names and line numbers, which can be very scary and confusing to inexperienced programmers. In addition, this trace is of little help to programmers about the dynamic nature of the data races, e.g. when and how the data races happen.

In this paper, instead of data race detection, we propose a data race prevention scheme, which can prevent data races from occurring in the first place. This scheme is implemented in our View-Oriented Parallel Programming (VOPP) model [13, 34]. 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 *Vpp_acquire_view*); after the access of a view, it must be released (e.g. with *Vpp_release_view*). The most important property for views is that they do not intersect with each other (refer to [13, 34] for details).

VOPP is a data-centric programming model [3, 5, 30], which bundles mutual exclusion and data access together. 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 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

*Zhiyi Huang is currently a visiting scientist at the Cilk group of MIT CSAIL.

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 techniques such as helper threads [14, 16, 18, 34].

This paper has the following contributions. First, we have proposed and implemented a data race prevention scheme for VOPP based on a virtual memory system. Second, we have proved the efficiency of the scheme with performance evaluation against other parallel programming models. Third, we have implemented a shared-memory parallel programming system: Maotai 2.0, which has enhanced VOPP with advanced features such as deadlock avoidance and producer/consumer view. Maotai 2.0 is based on Maotai 1.0 [34], but is enhanced with the features such as data race prevention.

The rest of this paper is organized as follows. Section 2 describes a data race prevention scheme that can eliminate data races in VOPP. In Section 3, we introduce the advanced features of Maotai 2.0 for improving programmability and performance. Section 4 discusses the programmability of VOPP and Section 5 presents the performance evaluation of Maotai 2.0. Finally, our future work is suggested in Section 6.

2 Data race prevention scheme in VOPP

In VOPP, shared data is defined through views. Unlike most shared memory parallel programming models, variables are private to a process by default in VOPP. Shared objects must be *explicitly* defined as “views”.

Views can be created, destroyed, merged, or resized, but a process must acquire a view (read-only or read-write) before accessing it and must release it after finishing with the view. However, this rule can be relaxed with compiler/run-time support (refer to Section 4). VOPP adopts the Single-Writer Multiple-Reader (SWMR) model. At any given time, a view can either be read/written by one process *or* allow read-only access to multiple processes. In our current implementation, a view uses a contiguous memory space to store shared variables. Below is a simple example of VOPP in C.

```
typedef struct {int a[ARRAY_SIZE];
                int result;} Foo;
Foo *ptr;
if (0 == Vpp_proc_id) {
  /* master allocates view 0 with
     type SWV, which is a shared object
     with "Foo" type */
  Vpp_alloc_view(0, sizeof(Foo), SWV);
}
Vpp_barrier();

ptr = Vpp_acquire_view(0);
ptr->result = compute(ptr->a);
Vpp_release_view(0);
/* attempts to dereference ptr after
   view is released will cause
   segmentation fault */
```

As illustrated in the above example, if a data structure should be shared by multiple processes, a view has to be created for it with *Vpp_alloc_view*. For exclusive access to the view, the view type is SWV, which means “Single Writer View”. However, we also provide other advanced views in Maotai 2.0 to enhance the programmability and flexibility of VOPP (refer to Section 3).

If a process wants access to a view, the view must be acquired with *Vpp_acquire_view*. The view must be released with *Vpp_release_view* after accessing it.

In our data race prevention scheme, a data race is prevented by a memory protection mechanism available in most UNIX systems. All views are initially protected from access using system calls such as *mprotect()*. Only after a view is acquired is a process allowed to access the memory space of the view via *mprotect()*. When a view is released, the process is again denied access to the view.

If a process accesses a view before *Vpp_acquire_view* or after *Vpp_release_view*, a segmentation fault will occur. Our system will handle the fault, send a warning message to the programmer that a view is accessed without acquisition, and quit the program execution.

In this way, only one process can access the view at a time. Programmers do not need to worry about the data race bugs. If a view is accessed by calling *Vpp_acquire_view*, mutual exclusion of the view access is automatically done by the system, i.e., Maotai 2.0. If a view is accessed without view acquisition, a segmentation fault will occur, and the system will alert the programmer about the error, which can be easily fixed by the programmer inserting *Vpp_acquire_view* and *Vpp_release_view* into the faulted code section.

The extra cost of this data race prevention scheme is the overhead of the memory protection. In Maotai 2.0, this cost is very low. On a Sun T2000 Server equipped with a 1GHz UltraSPARC T1 processor [27], micro-benchmarking results demonstrate that the overhead of memory protection added to the view primitives is generally very low (around 2-3μs). The exception is *Vpp_acquire_view*, requiring up to 35μs extra, which covers the essential overhead of the memory protection mechanism(see Table 1). Note that *Vpp_acquire_Rview* and *Vpp_release_Rview* means acquiring and releasing read-only views.

Table 1: Breakdown of view primitive costs (in μs)

Primitive	no prot	prot	cost
Vpp_acquire_view()	3.14	39.08	35.94
Vpp_acquire_Rview()	3.60	6.32	2.72
Vpp_release_view()	1.91	4.54	2.63
Vpp_release_Rview()	1.99	4.64	2.65

However, in our application benchmarks, this overhead does not cause noticeable difference in application speedup. Table 2 shows the speedups (at 32 processes) of our applications with and without memory protection in Maotai 2.0. We have six benchmark applications: Successive Over-Relaxation (SOR), Gaussian Elimination (GE), Integer Sort (IS), Neural Network (NN), Mandelbrot, and Mergesort. For details of these applications, refer to Section 5. As we can see from Table 2, in all 32-process benchmark cases, the difference is around 0.5%.

Table 2: Effects of memory protection on benchmark application speedups with 32 processes / threads

Application	no prot	prot
SOR	16.82	16.77
GE	22.41	22.36
IS	16.51	16.47
NN	16.98	16.92
Mandelbrot	17.80	17.79
Mergesort	12.52	12.50

One issue about the implementation is that memory protection such as *mprotect* is page-based. Therefore, in order to protect view data properly, memory space allocated to a view is aligned by pages. This can result in memory space wastage. Table 3 shows the requested and actual sizes of the memory space allocated by VOPP in our benchmark applications. The page size is 8kB and 32 processes are used when the data are collected. From this table, it can be seen that some applications like GE and Mandelbrot, which have many views that do not exactly fit a page, have a higher proportion of memory wastage (up to 51%), though other applications have less than 7% wastage.

Table 3: Requested vs actual VOPP shared size (in bytes) in different applications

Algorithm	Requested	Actual	Wasted	Proportion wasted
SOR	4097024	4194304	97280	0.0232
GE	64016004	98328576	34312572	0.349
IS	4194304	4194304	0	0
NN	271612	294912	23300	0.0790
Mandelbrot	2000000	4096000	2096000	0.512
Mergesort	1600001280	1600274432	273152	0.000171

However, fortunately, with architectural support of variable-size pages [4, 33], this memory wastage can be greatly reduced.

3 Advanced features in Maotai 2.0

Apart from data race prevention, Maotai 2.0 also offers primitives for acquiring multiple views in order to avoid deadlocks, producer/consumer views and system queues to enhance programmability and performance. These features are discussed below.

3.1 Deadlock avoidance

Similar to data race, deadlock is another pain that can happen easily but is difficult to debug in shared-memory parallel programming. In VOPP, deadlock can happen if views are acquired in a nested way and different processes acquire them in different orders.

For example:

```
 P1                       P2
 A_V(1)
                          A_V(2)
 A_V(2)
                          A_V(1)
/* here P1 holding view 1 waits for view 2
   which is held by P2, but P2 will not
   release view 2 until it gets view 1
   ----> DEADLOCK */
  .....                   ......

  R_V(2)                    R_V(1)
  R_V(1)                    R_V(2)
```

In the above example, A_V is *Vpp_acquire_view* and R_V is *Vpp_release_view*. The example shows that if $P1$ is holding view 1 while $P2$ is holding view 2, deadlock occurs.

To avoid deadlocks due to acquiring multiple views in different orders, Maotai 2.0 offers primitives for acquiring multiple views. Programmers can list all views to be acquired with these primitives which will acquire these views in a specific, same order. In this way, there is no chance for deadlocks to happen.

Below is an example illustrating the use of the primitives for acquiring multiple views:

```
/* acquire access to both view 0 and 1 */
Vpp_acquire_multiviews(0, &ptr0, 1, &ptr1);
ptr0->result = compute0(ptr0->a, ptr1->a);
ptr1->result = compute1(ptr1->a, ptr0->a);
Vpp_release_view(); /* release all views */
```

In the above example, the process acquires both view 0 and 1 with *Vpp_acquire_multiviews* which put the view base addresses into $ptr0$ and $ptr1$. Finally the process releases both views with *Vpp_release_view*.

Note that the above solution cannot eliminate deadlocks from VOPP programs as the data race prevention scheme does data races. There are two reasons: first, the programmer may choose not to use *Vpp_acquire_multiviews* for nested view acquisition; second, even if the programmer would like to use the primitive, it is difficult to know which views to acquire in advance in some programs where inner views can only be decided after the outer views are processed.

Nevertheless, the above primitives provide an avenue for novice programmers to avoid unnecessary deadlocks.

3.2 Producer/consumer view

Producer/consumer view is provided to allow direct expression of producer/consumer relationships in parallel algorithms. Traditionally barriers are used to synchronize the producer and the consumers in shared memory parallel programming. With the introduction of the producer/consumer view, programming with producer/consumer problem is more straightforward (see examples below) and thus increases programmability. In addition, producer/consumer view can avoid expensive barriers, which makes all processes wait and whose cost would increase with increasing number of processes.

The producer/consumer view is implemented as a queue. The producer enqueues a new version of the view by acquiring the view, producing the data, and finally releasing the view. The consumer dequeues a version of the view by acquiring read-only access to the view. After it finishes with the view, it releases its version of the view whose buffer may be recycled by the producer.

There are two types of producer/consumer views:

Producer/Consumer Single (PCS) is used in situations where all consumers share the same queue. That means, when a version of the view is dequeued by a consumer, it is not accessible to other consumers.

Producer/Consumer Multicast (PCM) is used for situations where each consumer has its own queue. The producer makes a copy of each version of the view for each consumer. Therefore, each version of the view is broadcast to all consumers.

The following example demonstrates the use of PCS.

```
Vpp_alloc_view(0, sizeof(Foo), PCS);
Vpp_barrier();
......
producer:                  consumer:
ptr=Vpp_acquire_view(0);  ptr = Vpp_acquire_Rview(0);
produce(ptr->a);             consume(ptr->a);
Vpp_release_view();        Vpp_release_view();
```

However, for the same problem, the following barrier version has to worry about the synchronization between the producer and the consumer.

Barrier version:

```
Vpp_alloc_view(0, sizeof(Foo), SWV);
Vpp_barrier();
......
producer:                  consumer:
ptr=Vpp_acquire_view(0);
produce(ptr->a);
Vpp_release_view();
Vpp_barrier();             Vpp_barrier();
                           ptr=Vpp_acquire_Rview(0);
                           consume(ptr->a);
                           Vpp_release_view();
```

In our experiments, the SOR and GE benchmark applications demonstrate that producer/consumer views (both PCS and PCM) give a better speedup than all other barrier based implementations, including their VOPP versions that use barriers. Figure 1 and 2 shows the speedup difference between applications using barriers and those using producer/consumer views.

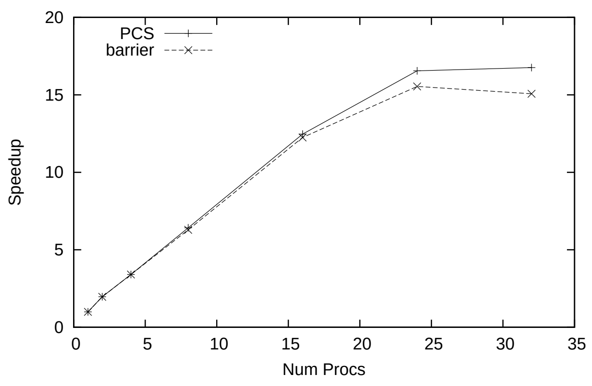

Figure 1: Speedup of SOR in VOPP

Figure 1 shows the speedup of SOR which uses PCS to improve its performance. Compared with its barrier implementation, the improvement of speedup is 11.2% at 32 processes.

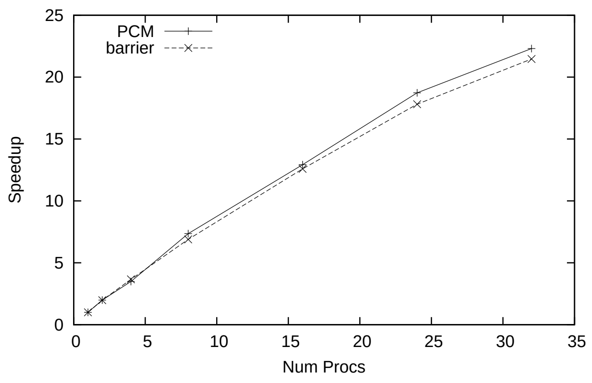

Figure 2: Speedup of GE in VOPP

Figure 2 shows the speedup of GE which uses PCM to improve its performance. Compared with its barrier implementation, the improvement of speedup is 4.2% at 32 processes.

3.3 System queues

System queues are provided in Maotai 2.0 to store view IDs. This facility allows easy implementations for task queues. Task queues are good for load balancing parallel applications (e.g. Mandelbrot and tree search algorithms), where the data for each job or node can be put in a view and its ID is simply enqueued in a system queue for other processes to work on.

Below is an example showing how a system queue serves as a task queue in VOPP.

```
if (0 == Vpp_proc_id) {
  /* each row is a job (and a view), and master
     process enqueue all jobs (view IDs) */
  for (y = 0; y < ydim; y++) {
    Vpp_alloc_view(y, xdim * sizeof(int), SWV);
    init_view(y);
    Vpp_enqueue_view(0, y);
  }
}
Vpp_barrier();
/* now all processes start working ... */
while ((vid = Vpp_dequeue_view(0))
      >= 0) {
  ptr=Vpp_acquire_view(vid);
  /* work on the row ....... */
  Vpp_release_view();
}
```

In the above example, the master process allocates and enqueues all jobs through enqueuing their view IDs into the system queue (numbered 0). Then all processes dequeue from system queue 0, get a view ID and process the jobs until the queue is empty. Without the system queues, programmers have to set up similar queues by themselves.

In Maotai 2.0, the enqueue and dequeue calls are efficient. In a microbenchmark test on a Sun T2000 server, an enqueue call only takes $2.65\mu s$ and a dequeue call takes $2.56\mu s$.

4 Programmability of VOPP

VOPP has improved the programmability of shared memory parallel programming since it eliminates data races and can avoid unnecessary deadlocks. In addition, producer/consumer views and system queues are provided in Maotai 2.0 to further improve its programmability. However, its distinct difference from other shared memory programming models is that VOPP requires views to be defined before accessing them.

From a syntactic point of view, view definition does not impose extra burden on programmers. In our current implementation, views are defined with *Vpp_alloc_view*, which is similar to *malloc*. With compiler support, views are defined as:

```
View struct {int a[SIZE]; int result;} my_v;
```

The programmer only needs to add a decorator "View" in front of the normal variable definition.

For most applications such as IS and SOR, views do not change once they are created. For those applications, view definition is very natural and straightforward. However, there is some small group of applications such as *Mergesort* that have changing views and have to re-organize the views (create new views and destroy old views). For those applications, VOPP does trade off some programming convenience for data race prevention.

Fortunately, Maotai 2.0 has provided a Multiple Writer View (MWV) to offer the programming convenience for experienced programmers. A MWV is a view that can be accessed simultaneously by multiple processes. Therefore, it is up to the programmer to avoid data races in a MWV. However, in contrast to other programming models such as Cilk++ [8] and OpenMP [6], the data races are confined in the current MWV should they occur.

In some shared memory parallel programming models [6, 8, 24], there are constructs like *reducer* for programmers to avoid data races. Apart from the fact that the syntax of reducer definition is more complex than view definition of VOPP, the operations of a reducer have to be pre-defined, which rules out any ad-hoc operations on the reducer from third-party software. In addition, reducers require the operations to be commutative, which restricts their usability. Fortunately, the view constructs in VOPP are free from these restrictions.

In the near future, the programming convenience of VOPP could be improved further with compiler/run-time support. The access of a view can be automatically detected by the memory protection mechanism at run-time. Once an access is detected, the system can find out which view is accessed and then acquire the view accordingly. The programmer only needs to tell the system about the atomic section of code. The following example illustrates our future VOPP language constructs.

```
View Foo V1, V2;
......
View() {
V1.result = compute1(V1.a, V2.a);
V2.result = compute2(V2.a, V1.a);
}
```

In the above VOPP code, the syntax is much simpler and free from errors such as missing *Vpp_release_view*. The underlying system acquires each view automatically once the view is accessed. To avoid deadlocks, the programmer could supply view information to the *View()* {} constructs so that the system can guarantee the same order of view acquisitions.

5 Performance evaluation with other models

In this section, we compare the performance of Maotai 2.0 with other modern shared memory parallel programming models like OpenMP, Cilk and Pthreads. Our benchmark applications include Successive Over-Relaxation (SOR), Integer Sort (IS), Gaussian Elimination (GE), Neural Network (NN), Mandelbrot and Mergesort. The experiments are carried out on a Sun T2000 server with an UltraSPARC T1 processor and 16GB memory. The UltraSPARC T1 has eight cores, each of which is clocked at 1GHz and supports four hardware threads. In total, the UltraSPARC T1 processor supports up to 32 hardware threads [27]. Linux kernel 2.6.24-sparc64-smp and the compiler gcc-4.4 are used during benchmarking. The benchmark applications are implemented on Maotai 2.0, Cilk-5.4.6 [28], OpenMP 3.0 [6] and Pthreads [22], respectively. All programs are compiled with the optimization flag "-O2". In each case, speedup is measured against the serial implementation of the benchmark algorithm. The elapsed time calculated in each case excludes initialization and finalization costs, because they are one-off and are difficult to measure within the program in models that involve source-translation, such as Cilk and OpenMP. Instead, startup and finalization times for each model are measured separately. Runtime of functions that are irrelevant to the original application, such as generation of random sequences and result-verification, are also excluded.

Successive Over-relaxation (SOR) is a multiple-iteration algorithm where each element is updated by the values of the neighbouring elements from the last iteration. In this experiment, the implementation is adapted from [34]. Matrix size is set to $8000 * 4000$ and 40 iterations are performed.

The Integer Sort (IS) algorithm used in this experiment is based on the NPB version [29]. This is a counting-sort algorithm. In this experiment, the problem size is 2^{26} integers with a B_{max} of 2^{15} and 40 repetitions are performed.

The Gaussian Elimination (GE) implementation from [31, 34] is used in this experiment and the matrix size is set to $4000 * 4000$.

The parallel Neural Network (NN) implementation is based on Pethick's work [23]. This algorithm trains a back-propagation neural network in parallel using a training data set. In this experiment, the size of the neural network is set to 9 * 40 * 1 and the number of epochs is set to 200.

The Mandelbrot algorithm is embarassingly-parallel. However, the workload of pixels is extremely uneven, and thus requires a load-balancing mechanism to prevent process starvation [11, 32]. In this experiment, the size of the screen is set to 500 * 500, the maximum number of iterations is set to 500 and each pixel is calculated 5000 times. The maximum number of processes / threads is set to eight for this experiment because hyperthreading relies on memory latency. Since this application has very few memory accesses, there is little speedup when more processes / threads than the number of CPU cores are used (the UltraSparc T1 has eight cores).

The parallel Mergesort algorithm in Cilk is recursive [17, 28] and is implemented verbatim in Cilk and OpenMP to test performance of the newly-available task-parallelism feature in OpenMP [1]. The array consists of 200 million integers. This algorithm is converted to the iterative version for VOPP and Pthreads. The iterative version requires the number of processes to be a power of 2. This version first divides the array equally between the processes and each process sorts its own subarray. Then the merge procedure largely models the recursive version of the parallel merge algorithm. Both MWV and SWV versions of the VOPP implementation are included to test effects of the extra memory copying needed in SWV.

Since the UltraSPARC T1 has only one floating-point unit, all floating-point calculations in the above algorithms are converted to integer calculation to avoid the bottleneck at the floating-point unit.

5.1 Experimental results

The experimental results are illustrated with speedup curves. For each application, we give its speedup curves on Maotai 2.0, Cilk, OpenMP and Pthreads. In the discussion below, n refers to the number of processes / threads.

For SOR (Figure 3), Maotai 2.0 has the best performance. At $n = 32$, Maotai 2.0 is 13.6% better than Cilk, 17.9% better than OpenMP and 12.0% better than Pthreads.

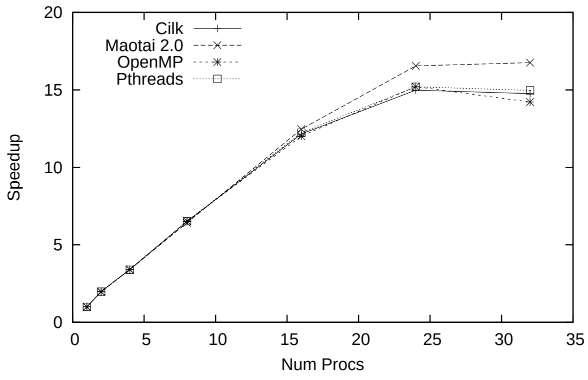

Figure 3: Speedup of SOR

For GE (Figure 4), Maotai 2.0 again has the highest speedup. At $n = 32$, Maotai 2.0 is 7.4% better than Cilk; 33% better than OpenMP and 7.8% better than Pthreads.

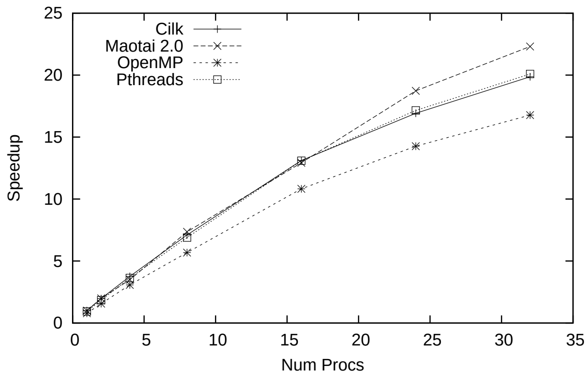

Figure 4: Speedup of GE

In IS (Figure 5), there are less variations in speedups in different models. However at $n = 32$, Maotai 2.0 is 5% faster than Cilk; 15% faster than OpenMP and 7% faster than Pthreads.

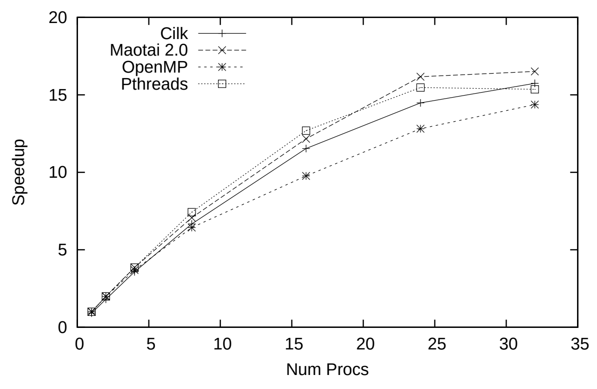

Figure 5: Speedup of IS

In NN (Figure 6), all models have similar speedups. Maotai 2.0 is 3.1% faster than OpenMP, but it is 1.8% slower than Cilk and 0.2% slower than Pthreads.

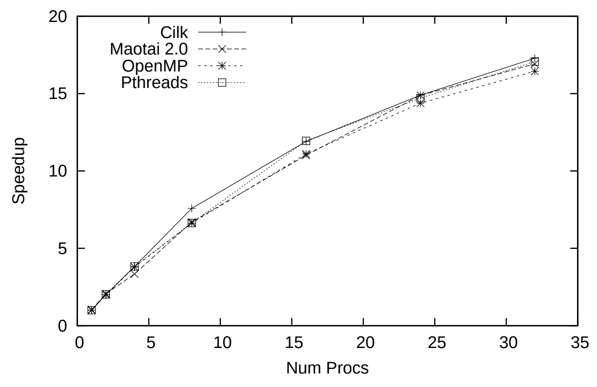

Figure 6: Speedup of NN

In Mandelbrot (Figure 7), there are relatively little differences between speedups of different models. At $n = 8$, Maotai 2.0 is 0.8% faster than Cilk; 7.2% faster than OpenMP and 3.3% faster than Pthreads.

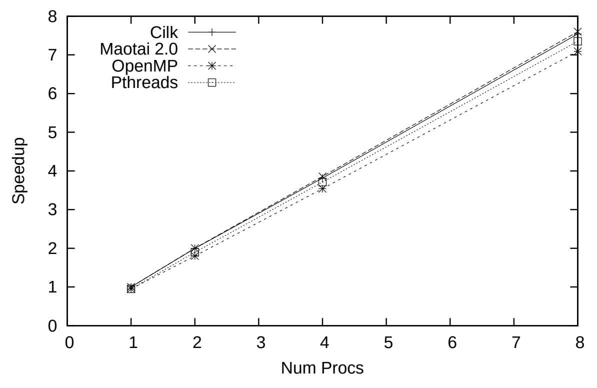

Figure 7: Speedup of Mandelbrot

For Mergesort, speedup of Maotai 2.0 is slower but comparable with other shared-memory models, though the SWV version is clearly not scalable (Figure 8). At $n = 32$, Maotai 2.0 is 9 % slower than Cilk; 1% faster than OpenMP and 2 % slower than Pthreads. The MWV version is 3.5 times faster than the SWV version.

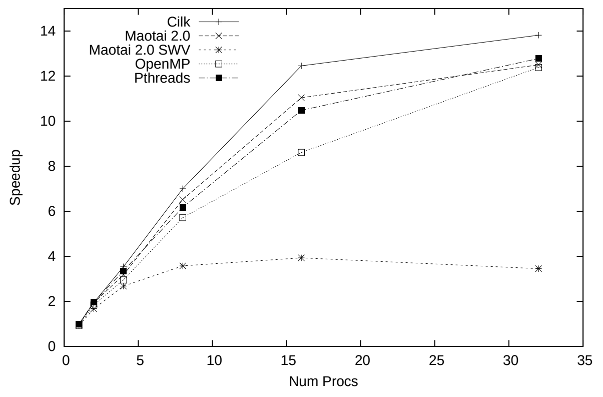

Figure 8: Speedup of Mergesort

Note that, in the above collected results, the standard deviations of the elapsed time at $n = 32$ for Maotai 2.0, Cilk and Pthreads cases are less than 0.1s, but the standard deviations of the elapsed time for OpenMP are between 0.2 to 0.5s, which may be due to the random nature of the OpenMP task scheduler.

Table 4 presents the startup and finalization time of each system. As expected, startup and finalization costs for thread-based models including Cilk, OpenMP and Pthreads are lower than process-based system like Maotai 2.0.

Table 4: Combined startup and finalization time (in ms) for different number of processes/threads on a Sun T2000 server

	1	2	4	8	16	24	32
Cilk	2	2	2	2	2	2	2
OpenMP	2	2	2	2	2	2	2
Pthreads	2	2	2	2	2	2	2
Maotai 2.0	9	10	11	13	15	19	22
Serial	2						

All thread-based models have the same combined startup and finalization time as the serial version regardless of the number of threads. Maotai 2.0 has a startup/finalization cost of 9ms (at $n = 1$) and the cost grows to 22ms at $n = 32$, almost linear to the number of processes. Despite Maotai 2.0 having a larger startup/finalization overhead, the 22ms is still negligible compared to the time consumed in $n = 32$ cases, which is at least 10 seconds. Also the startup/finalization time in Maotai 2.0 is only a one time event, therefore this overhead should have negligible effect on the speedup curves.

5.2 Discussion

The following is an analysis on why Maotai 2.0 performs better or worse than other systems.

As we mentioned before, the producer/consumer view in Maotai 2.0 enhances both programmability and performance of SOR and GE. In SOR, PCS is used to pass boundary rows to neighbour processes, thus allowing the natural expression of the message-passing relationship without the use of barrier, which would hold up irrelevant processes. Apart from programmability, the resultant performance gain is reflected in Figure 1, where the PCS VOPP version is 11.2% faster than the barrier-based SOR version.

Similarly in GE, PCM is used to *broadcast* the pivot row and the swap index, which improves programmability by mimicking the broadcasting semantics in the parallel algorithm. Also the removal of barriers by PCM improves the VOPP performance by 4.2% (Figure 2). Time is saved by replacing lock and barrier primitives with a PCM primitive.

The introduction of system queues for programmability in Maotai 2.0 does not come at the expense of performance. The efficiency of the system queue primitives can explain the slight performance advantage over other models in Mandelbrot.

In IS, the performance advantage seen in Maotai 2.0 over other models can be attributed to the split of global keyden array into *nproc* views. In the global keyden construction step, each process updates *all* global keyden parts in the round-robin fashion, starting from the $proc\_id^{th}$ part. Here, the SWMR view access pattern removes the need for barriers for preventing data race due to multiple processes updating an element simultaneously. This removal of barriers can contribute to the performance gain by the VOPP program.

In NN, since multiple items are updated by multiple processes at the end of the iteration, barriers are still used in the VOPP program. However the performance of Maotai 2.0 is still comparable to other models, which shows that being data race free has little impact on performance.

However, the SWMR model in VOPP does have its limitations in cases where the access pattern changes in every iteration, as we mentioned in Section 4. In those cases, view data must be copied to a local buffer of a process, where the process works on the data. After the data is processed, the view is acquired again by the process and the results copied back to the view. In our application Mergesort, the resultant excessive *memory-copying* renders the implementation unscalable (Refer to VOPP-SWV in Figure 8). However, the alternative MWV implementation allows multiple processes to work directly on the view and avoid memory copying. This flexible multiple write view (MWV) made the speedup of Maotai 2.0 comparable to other shared-memory models, though the programmer has to take the risk of data races within the view.

Although Cilk is internally implemented using Pthreads, there are cases, such as Mergesort, GE, IS and Mandelbrot where Cilk performs better than Pthreads (also, in Mergesort and NN, it is better than Maotai 2.0). This can be attributed to the recursive task decomposition of Cilk ensuring cache locality [17].

The parallel for-loop in OpenMP allows easy specification of data-parallelism. However, it would introduce a task-scheduling cost, especially when the workload is fixed and no load-balancing is required. The lower speedups of GE, SOR and NN of OpenMP can be attributed to this parallel for-loop overhead. Although Cilk++ cannot be benchmarked in this experiment because sparc64-smp is not supported, its equivalent construct **cilk_for** can also have the similar task-scheduling overhead.

As we noticed, OpenMP has bigger standard deviations in its elapsed time. It can suggest that the OpenMP scheduler has some random behavior. In Mergesort, although the recursive OpenMP implementation has benefited from cache locality like Cilk, its worse performance (compared to Cilk) can be attributed to the inefficiency of its task scheduler.

6 Conclusions and future work

Our data race prevention scheme based on views proves to be efficient and adds little extra overhead to parallel programming systems. Though there is some memory wastage due to page alignment in the implementation, architectural support for variable-size pages will significantly reduce the wastage. Even with a fixed page size, view constructs are useful to remove data races. Compared with reducers in OpenMP, Cilk and TBB, views are more flexible and allow ad-hoc operations.

With the advanced features in Maotai 2.0, the performance and programmability of VOPP are enhanced. Though strict SWV views are rigid for some application like Mergesort, Maotai 2.0 offers MWV views to avoid dynamic view re-organization in the application.

Performance results demonstrate that Maotai 2.0 is very competent among modern parallel programming models, even with the unique data race prevention scheme.

In the near future, we will investigate automatic detection of view access and compiler support of VOPP. Currently views must be *explicitly* acquired and released, which can be removed with run-time/compiler support. In automated view detection, *beginning* of view access can be defined as the first access of view data. However, to define a *releasing* point (aka. exit protocol) [5, 30] could be difficult.

To allow natural expression of recursive algorithms such as tree search and parallel mergesort, a task model such as Cilk will also be investigated. This avenue would bring the advantage of recursive task decomposition in Cilk and the features of VOPP together.

References

[1] Eduard Ayguadé, Nawai Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel And Distributed Systems*, 20(3):404–418, March 2009.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-

parallel relationships in fork-join multithreaded programs. In *SPAA04*, June 2004.

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

[4] Mihai Burcea, J. Gregory Steffan, and Cristiana Amza. The potential for variable-granularity access tracking for optimistic parallelism. In *The ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'08)*, 2008.

[5] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *HPCA'07*, 2007.

[6] Barbara Chapman, Habriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* MIT Press, 2007.

[7] J-D Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13:491–530, 1991.

[8] Cilk Arts Inc. *Cilk++ User Guide.*

[9] Anne Dining and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *The 2nd ACM SIGPLANS Symposium on Principles & Practice of Parallel Programming (PPoPP*, pages 1–10, 1991.

[10] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing'91*, pages 580–588, 1991.

[11] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI : Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 2nd edition, 1999.

[12] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Analyzing traces with anonymous synchronization. In *The 19th International Conference on Parallel Processing (ICPP'90)*, pages 1170–1177, 1990.

[13] 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.

[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] Tushara C. Karunaratna. Nondeterminator-3: A provably good data-race detector that runs in parallel. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2005.

[16] 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.

[17] Charles E. Leiserson. A minicourse in multithreaded programming. Technical report, MIT Laboratory for Computer Science, 1998.

[18] 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.

[19] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91*, pages 24–33, 1991.

[20] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN '88 conference on Programming language design and implementation*, volume 23, pages 135–144, July 1988.

[21] Robert H.B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *The 21st International Conference on Parallel Processing (ICPP'92)*, 1992.

[22] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *PThreads Programming.* O'Reilly, 1996.

[23] Mark Pethick, Michael Liddle, Paul Werstein, and Zhiyi Huang. Parallelization of a backpropagation neural network on a cluster computer. In *International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, 2003.

[24] James Reinders. *Intel Threading Building Blocks : Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, 2007.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multithreaded programs. In *The 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, 1997.

[26] 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.

[27] Sun Microsystems. *OpenSPARC T1 Microarchitecture Specification*, 2006.

[28] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, 1998.

[29] Rob F. van der Wijngaart and Michael Frumkin. NAS grid benchmarks version 1.0. Technical report, NASA Advanced Supercomputing Division, NASA Ames Research Center, 2002.

[30] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *The 33rd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'06)*, 2006.

[31] Paul Werstein, Mark Pethick, and Zhiyi Huang. A performance comparison of DSM, PVM and MPI. In P Fan and H Shen, editors, *in Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 476–482, Chengdu, August 2003. IEEE Press.

[32] Barry Wilkinson and Michael Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.

[33] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *ASPLOS-X 2002*, 2002.

[34] Jiaqi Zhang, Zhiyi Huang, Wenguang Chen, Qihang Huang, and Weimin Zheng. Maotai: View-Oriented Parallel Programming on CMT processors. In *The 37th International Conference on Parallel Processing (ICPP'08)*, 2008.