Maotai 3.0: Automatic Detection of View Access in VOPP

K. Leung and Z. Huang
Department of Computer Science
University of Otago
Dunedin, New Zealand
Email:{kcleung;hzy}@cs.otago.ac.nz

***Abstract*—This paper proposes a scheme for automatic detection of view access in the View-Oriented Parallel Programming (VOPP) model. VOPP is a shared-memory-based, data-centric model that uses "views" to bundle mutual exclusion with data access. Based on the automatic detection scheme, a view is automatically acquired when first accessed, and automatically released at proper time. This scheme simplifies the VOPP model and prevents programming errors. With this scheme, the programmability of VOPP is similar to transactional memory models. In addition, VOPP can eliminate data races without compromising performance. A new VOPP implementation, Maotai 3.0, has been developed and incorporated the above features. Experimental results demonstrate that the performance of Maotai 3.0 surpasses transactional memory models such as TL-2.**

***Keywords*-View-Oriented Parallel Programming (VOPP), Transactional Memory, data race free, deadlock free, parallel programming, multicore.**

I. Introduction

Parallel programming is becoming mainstream since multicore processors have become pervasive. There is a pressing need for parallel programming models to facilitate the performance and reliability of applications. Yet current shared-memory programming models, such as OpenMP [1], Pthreads [2] and the transactional memory model TL-2 [3], are prone to errors, e.g. data race, which are difficult to debug due to their non-deterministic behavior. Impacts of these programming errors will be magnified in grid and cloud environments, where applications are used by thousands of users across the world.

Data race, which happens when multiple processes/threads access the same object concurrently and at least one of them writes to the variable, is a chronic problem in parallel programming. The View-Oriented Parallel Programming (VOPP) model is designed to eliminate data race [4, 5].

VOPP is a data-centric model, where shared data objects are partitioned into "views" by the programmer according to the memory access pattern of a program. The grain (size) and content of a view are decided by the programmer as part of the programming task, which is as easy as declaring a data structure or allocating a block of memory space. Each view can be created, resized, merged, or destroyed at any time in a program. The most important property for views is that they do not intersect with each other. Before a view is accessed (read or written), it must be acquired; after the access of a view, it must be released. Normally, single-writer view (SWV) will be used, which adopts the single-writer-multiple-reader (SWMR) protocol to allow multiple readers (i.e. at any given time, a view can be either read by multiple processes or written by a single process) [6, 7]. Like other data-centric models [8], VOPP bundles mutual exclusion and data access together and therefore relieves the programmer from managing locks directly to safeguard shared data.

Data-centric models are safer than code-centric models that use locks to protect critical code sections. Data-centric models are only interested in which shared object is used (and thus held when being used). Using a shared object longer than necessary will only decrease the performance of the program, but will *not* cause data races. In contrast, in code-centric models, programmers must demarcate each critical section of the program that accesses shared data, and guard it with locks or atomic transactions. Very often mistakes in such demarcations can result in data races.

However, in Maotai 2.0 [5], which is a VOPP implementation for multicore machines, views must be explicitly acquired before access and released after access. It is often troublesome to manage view acquire/release constructs.

For example, below we show a serial version of a list traversal program and its Maotai 2.0 version:

```
/* serial version */            /* Maotai 2.0 */

typedef struct Node_rec Node;   typedef struct Node_rec Node;

struct Node_rec {               struct Node_rec {
  Node *next;                     Node *next;
  Elem elem;                      Elem elem;
};                              };

Node *list_search(Elem elem,    Node *list_search(Elem elem,
                  Node *list)                     vid_type vid)
{                               {
  while (NULL != list) {          Node *list =
    if (elem == list->elem) {        Vpp_acquire_view(vid);
       return list;
    }                               while (NULL != list) {
    list = list->next;                if (elem == list->elem) {
  }                                     Vpp_release_view();
  return NULL;                          return list;
}                                     }
                                      list = list->next;
                                    }
                                    Vpp_release_view();
                                    return NULL;
                                  }
```

The above list traversal in Maotai 2.0 is prone to error, because it is easy to forget to release the view before calling *return* within the while loop. If that happens, the next process that is acquiring the view will wait forever. Our scheme for automatic detection of view access can resolve such problems by acquiring and releasing views automatically, which has greatly improved the programming interface of VOPP (refer to Section II for details). Additionally, the cost for the automatic detection is relatively small according to our experimental results.

A. *Contributions of this paper*

This paper has the following contributions. First, we have proposed and implemented the scheme for automatic detection of view access, which improves the programmability of VOPP and no longer requires programmers to use explicit view acquire/release constructs. A view is automatically acquired when first accessed and released when leaving the scope of the view acquisition. Second, we have shown the automatic detection scheme improves the programming convenience of VOPP, the programmability of which is similar to transactional memory models in many cases. Third, we have implemented the parallel programming system, Maotai 3.0, which is based on the VOPP model with the automatic detection scheme. Performance results show that Maotai 3.0 has superior performance over transactional memory models like TL-2 0.9.6 [3].

The rest of the paper is organized as follows. Section II describes the automatic detection scheme, the language constructs, and implementation details of Maotai 3.0. Section III compares the programmability of Maotai 3.0 with transactional memory models. Section IV covers experimental results and performance evaluation. Section V discusses related work. Finally, Section VI gives the conclusions and the future work.

II. THE PROGRAMMING MODEL AND IMPLEMENTATION DETAILS OF MAOTAI 3.0

The introduction of automatic detection of view access helps remove the explicit view acquiring and releasing, and thus greatly simplifies the programming interface to shared data access in Maotai 3.0. It reduces extra code instrumentation needed to parallelize existing serial code. This section will discuss the language constructs and their semantics used in Maotai 3.0, as well as the implementation details of the automatic detection scheme.

A. *Automatic detection of view access*

In this scheme, a view is automatically acquired when its memory is first accessed. Then the view is automatically released when control leaves the *scope of view acquisition*.

The scope of view acquisition is often the function that first accesses the view. During the execution of such a function, the executing process acquires the view when it is first accessed, and automatically releases the view when the function returns. Our applications show that, in most cases, this functional scope of view acquisition is the intention of the programmer. Below is an example that shows at what time a view, called *foo*, is acquired and released automatically in the function *func*.

```
VPP void func(void) {
  Foo *foo = Vpp_alloc_view(sizeof(foo[0]), SWV);

  .....
  foo->index = 5;             /* view foo acquired */
  printf("%d\n", foo->val);
  ....
  ....
}                             /* view foo released */
```

Bearing in mind the above scope of view acquisition, our list traversal code becomes as follows.

```
/* Maotai 3.0 */

typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

VPP list *list_search(Elem elem,
                      Node *list) {
  while (NULL != list) {
    if (elem == list->elem) { /* view acquired */
       return list;           /* view released */
    }
    list = list->next;
  }
  return NULL;              /* view released */
}
```

In the list traversal code above, there is very little code changes compared with the original serial code. The only changes to the serial code are adding the keyword *VPP* as an attribute of the function *list_search()*.

Compared with Maotai 2.0 [5], the programmers do not need to keep track of view IDs and the acquire/release statements.

The keyword *VPP* is used to declare that a function will have effect on the scope of view acquisition. When a *VPP* function returns, it will automatically release all views acquired during the execution of the function, including those views acquired in the callee functions. Also the subsequent callee functions have access to the views once they are acquired.

For example, in the example code below, *func1* acquires view 1 and then calls *func2*. *func2* inherits the acquisition of view 1 throughout its scope (Figure 1).

```
/* a third-party non-VPP library function */
void func3() {        /*  inherit v1 and v2  */
  ptr_3->val = 0;     /*  acquire v3,
  ........                but v3 belongs to
                          immediate VPP ancestor (func2())
                          and will only be released
                          at the end of func2() */
}

VPP void func2(Object *ptr_1) { /* inherit v1 */
  ptr_1->done = 1;
  ....
  ptr_2->index =      /*  acquire v2  */
    ptr_1->index + 1;
```

```
  ....
  func3();            /*  func3 inherits v1 and v2 */
  ....
  ....
}                     /*  release v2, v3 */

VPP void func1() {
  ptr_1->index = 0;   /*  acquire v1  */
  ....
  func2(ptr_1);    /* func2 inherits v1 */
  ....
}                     /*  release v1  */
```

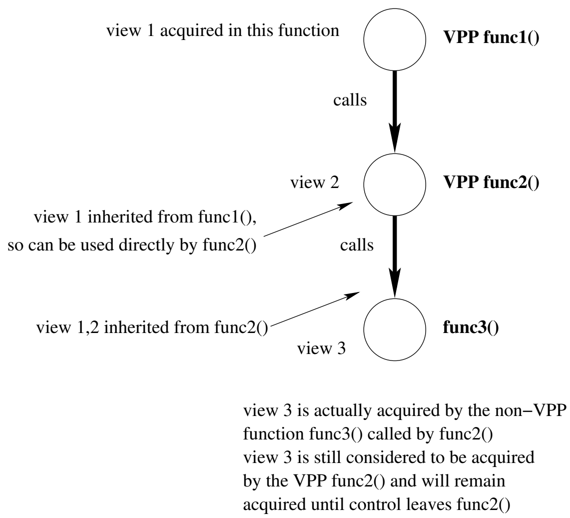

Figure 1. Inheritance of views acquired by VPP functions

Similarly, *func2* acquires view 2 and calls *func3*. *func3* inherits the acquisition of view 1 and 2 throughout its execution.

However, since *func3* does *not* have the *VPP* attribute, it has no effect on the scope of view acquisition. Therefore, view 3 that is acquired during the execution of *func3* will *not* be released when *func3* returns. The scope of view acquisition for view 3 is *func2*, the immediate VPP ancestor of *func3*, which will release view 3 as well as view 2 when it returns.

B. View Scope Construct

The automatic view access detection model described above works well in most cases. However, in some cases, a view acquired by a callee function is actually intended to be held until the end of the *current* function. Even though we can achieve this by making the callee function a non-VPP function, this restricts the use of the callee function as a VPP function. The following example illustrates the problem in more details.

```
VPP void bar(char *shared_str) {
  ......
  shared_str[0] = 'a'; /* view shared_str acquired */
    .
    .
    .
}                      /* view shared_str released */

VPP void foo(char *shared_str) {
  .....
  bar(shared_str); /*view shared_str acquired and
                      released by bar, but
  ....                actually intended to be
  ....                acquired until end of foo()*/
  ....
  ....
  str[1] = str[0] + 1;  /* RW access to view
  ....                     shared_str reacquired */
  ....
}                       /* shared_str released */
```

In the example above, *foo()* intends to hold the view *shared_str* during its execution, though it is acquired during the execution of *bar()*. However, since *bar()* is a VPP function, and *shared_str* is acquired during the execution of *bar()*, under the automatic detection scheme, *shared_str* will be released when *bar()* returns. Therefore, *shared_str* will be re-acquired when it is accessed again in *foo()*. In this situation, the view acquisition of *shared_str* is unwittingly fragmented. Though there is no data race involved in this situation, it may affect the atomicity of the operation on *shared_str* intended by the programmer.

To address this problem, the view scope construct *VPP_View(access_type, pointer_to_view,...)* {...} is proposed to allow views to be automatically acquired at the *beginning* of the declared scope, where *access_type* can either be *VPP_RO* or *VPP_RW*, which stand for read-only access and read-write access respectively. Programmers can use view scope constructs to manually define the scope of view acquisition. Any views automatically acquired within a view scope construct, including those acquired in non-VPP callee functions, will be release when control leaves the declared view scope.

In summary, the view scope construct works according to the following rules:

1) A view scope construct acquires the listed views at the *beginning* of the scope according to the listed order.
2) Within the scope, accesses of other unacquired views are still automatically detected and acquired at their first access.
3) All views acquired within the view scope, including those acquired by the non-VPP callee functions, will be released automatically when control leaves the *view scope construct*.

With the view scope construct, the above example can be written as below to achieve the programmer's intended scope of view acquisition.

```
VPP void bar(char *shared_str) { /* view shared_str */
  ......                              inherited
  shared_str[0] = 'a';
    .
    .
    .
}

VPP void foo(char *shared_str) {
  .....
  VPP_View(VPP_RW, shared_str) { /* view shared_str acquired */
    bar(shared_str);                         |
    ....                                     |
    ....                                     |
```

```
    str[1] = str[0] + 1;                    |
    ....                                    |
  }                                  /* view shared_str no longer
                                        needed, and is released at
                                        the end of VPP_View scope
  /* more calculations.... */           instead of the end of foo()
  ........                              in order to maximize the
                                        concurrency of view
                                        accesses */
}
```

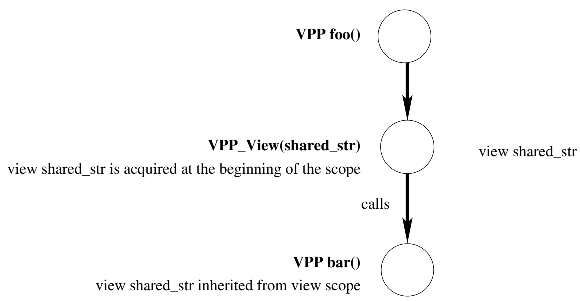

Figure 2. Inheritance of views acquired by VPP functions and view scopes

In the example above, a view scope construct is used to acquire the view *shared_str* in the caller *foo()*, as intended by the programmer, and the callee *bar()* inherits the view *shared_str* from *foo()* (refer to Figure 2 for the inheritance of views).

In addition, view scope constructs allow programmers to define exactly when views are acquired and released, so views can be released as soon as they are not needed. In this way, views are not unnecessarily held until the end of the function and thus do not unnecessarily hinder concurrent accesses of views.

C. Deadlock free mode

The automatic view access detection scheme acquire views automatically as they are first accessed, therefore views may be acquired in different orders by different processes. As a result, the compiler cannot guarantee that views are acquired in a consistent order, deadlock remains possible with the automatic detection scheme.

```
Process 0                   Process 1
{                           {
  /* foo acquired */
  foo->index++;

                              /* bar acquired */
                              bar->index++;

  /* acquire bar,
     but held by P1 */
  bar->val = foo->index;

                              /* acquire foo,
                                 which is held by P0 */
                              /* DEADLOCK */
                              foo->next = bar->val;
}                           }
```

In the example above, *foo* and *bar* are separate views. Process 0 (P0) acquires *foo* and process 1 (P1) acquires *bar* first, then P0 tries to acquire *bar*, but *bar* is already held by P1, so P0 blocks. At the same time, P1 tries to acquire *foo*, which is held by P0 (also blocked). As a result, deadlock occurs. This scenario can happen when inexperienced programmers fail to ensure that views are accessed in a consistent order.

To prevent deadlocks, deadlock-free mode is offered in Maotai 3.0. The deadlock-free mode can be specified during initialization of a VOPP session and is effective for the whole VOPP session. A VOPP program starts with serial execution. When parallel processing is desired, a VOPP session is started with *Vpp_session()* which creates multiple processes to execute the same function in parallel. When a VOPP session is finished, the program reverts to serial execution, but can start another VOPP session anytime later.

The following rules are applied in the deadlock-free mode to avoid deadlocks:

- Automatic detection of view accesses is disabled.
- All views must be *explicitly* listed in the *VPP_View(access_type, ptr_to_view,...)* {...} view scope construct. The system will acquire the listed views at the beginning of the view scope construct in the same system-determined order to prevent deadlocks.
- Access to unacquired views will result in termination of the program and an error message will be printed to notify the user where the violation occurs, as in Maotai 2.0.
- Nesting of view scope constructs is forbidden.

D. Implementation details and overheads

In the automatic detection of view access scheme, we use virtual memory protection (such as *mprotect()*) to detect view access. Initially a view (consisting of a number of pages) is protected against any access. When it is accessed, a page fault will occur and the page fault handler will be invoked to process the fault. We use this mechanism to implement the automatic detection scheme.

A view is automatically identified and acquired when its memory is first accessed (which can be detected by the page fault handler that subsequently acquires the view). The view is automatically released when control leaves the scope of view acquisition, as defined by either the control scope of the VPP function or the view scope construct.

Like Maotai 2.0, view acquisition is lock-based and is implemented using Pthreads *rwlock* [2], which is based on futex [9].

Therefore the overheads of Maotai 3.0 include:

- futex lock overhead
- virtual memory protection overhead
- view identification overhead (for calculating the view identity at runtime from the accessed memory address)
- page fault handler overhead (only for automatic detection of view access)

The lock-based view acquisition itself incurs the futex lock overhead and virtual memory protection overhead, the automatic detection of view access incurs the rest of the listed overheads.

To examine these overheads, a microbenchmark is run on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running at 800MHz. Overhead is measured for:

- basic pthread_rwlock operations (pthread_rwlock)
- explicit view acquire without runtime protection (still requires view identification mechanism) (no_prot)
- explicit view acquire with runtime protection (manual)
- automatic view access detection (automatic)

To amortize measuring noises, we have collected the results by first measuring the execution time of 100,000 sequentially-executed identical operations and then calculating the average execution time of one operation. The results are presented in Table I.

Table I
BREAKDOWN OF VIEW PRIMITIVE COSTS (IN μs)

Primitive	pthread_rwlock	no_prot	manual	automatic
a_v()	0.08	0.94	4.15	15.24
a_rv()	0.08	0.92	4.14	15.17
r_v()	0.08	0.08	2.74	N/A
r_rv()	0.08	0.08	2.73	N/A

Note: "a" stands for acquire; "r" stands for release; "v" stands for view and "rv" stands for read-only view. In the pthread_rwlock test, a_v() stands for pthread_rwlock_wrlock(); a_rv() stands for pthread_rwlock_rdlock(); r_v() stands for releasing the wrlock and r_rv() stands for releasing the rdlock.

The above results show that automatic detection mechanism does incur runtime computation overheads for view identification, virtual memory protection and the page fault handler. In automatic detection mode, it takes $15\mu s$ to acquire a view, whereas it only takes $80ns$ to acquire a pthread_rwlock. The automatic detection overhead of $15\mu s$ is small enough for most applications, as shown in performance comparison between Maotai 2.0 (which has no automatic detection) and Maotai 3.0 in Section IV. However, this overhead would make applications requiring fine-grain view partition and frequent view accesses unscalable. Also, due to the page-based memory protection mechanism, all view allocations must be page-aligned, which may waste memory space.

In the future, to reduce the runtime overheads and the waste of memory space, we will investigate compiler support of VOPP to allow compile-time tracking of view allocation and access, so that data-race free feature in VOPP can be partially implemented at compile time.

III. PROGRAMMABILITY OF MAOTAI 3.0 AND TRANSACTIONAL MEMORY MODELS

As mentioned in Section I and Section II-A, automatic detection of view access improves programmability of Maotai by eliminating programming errors in Maotai 2.0 arising from forgetting to release acquired views, especially when control leaves the scope not at the end of the scope (e.g. by keywords such as *break* or *return*) as illustrated in the list search example:

```
/* Maotai 2.0 */

typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

Node *list_search(Elem elem,
                  vid_type vid)
{
  Node *list =
     Vpp_acquire_view(vid);

  while (NULL != list) {
    if (elem == list->elem) {
      Vpp_release_view();     /* the list will be held forever
                                 by this process if forgotten
                                 to be released */
      return list;
    }
    list = list->next;
  }
  Vpp_release_view();
  return NULL;
}
```

However, this type of programming errors is eliminated by automatic detection of view access in Maotai 3.0 (its code snippet is shown below), since views are automatically acquired and released by the runtime system.

```
/* Maotai 3.0 */

typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

VPP list *list_search(Elem elem,
                      Node *list) {
  while (NULL != list) {
    if (elem == list->elem) { /* view acquired */
       return list;          /* view released */
    }
    list = list->next;
  }
  return NULL;             /* view released */
}
```

Comparing the above two code snippets, we can see that the lines-of-code (LOC) of the Maotai 3.0 version is around 20% fewer than the Maotai 2.0 version.

Moreover, as seen in the code snippets, converting a serial program to Maotai 3.0 requires very little code instrumentation, apart from tagging some functions with the keyword **VPP**. If programmers want to optimize the program performance, they can easily fine-tune the program by using the view scope construct to control how long a view is held.

However, in Transactional Memory (TM) models, programmers must *manually* instrument all code that access shared data to put them into atomic constructs. For example, for the same list traversal example, TM models often have the following code snippet:

```
typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

/* search a list in shared memory */
list *list_search(Elem elem,
                  Node *list) {
  atomic {
    while (NULL != list) {
      if (elem == list->elem) {
         return list;
      }
      list = list->next;
    }
  }
  return NULL;
}
```

The above list traversal code (which accesses a shared list) must be included in an atomic construct. Failure to put code that access the shared data into an atomic construct can result in data race bugs.

In contrast, Maotai 3.0 is always *data-race free*. Sub-optimal programming only compromises performance by holding views longer than necessary, but does not cause data races in Maotai 3.0. Violation of safe view accesses can be detected by the system. Therefore, Maotai 3.0 is safer than TM models.

While TM does not suffer from deadlocks, Maotai 3.0 can avoid deadlocks by using the dead-lock free mode.

IV. Performance Evaluation and Discussion

In this section, we compare the performance of Maotai 3.0 with Maotai 2.0 [5] and the software transactional memory system TL-2 version 0.9.6 [3]. Our benchmark applications include Mergesort, Raytrace, Barnes-Hut, Parallel Neural Network (PNN), Binary-tree (BT), Linked-List (LL) and Travelling Salesman Problem (TSP), representing different classes of applications. The experiments are carried out on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running at 800MHz and 16GB DDR2 memory. Linux kernel 2.6.31 and the compiler gcc-4.4 are used during benchmarking.

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 includes initialization and finalization costs. However, runtime of functions that are irrelevant to the original application, such as generation of random input sequences and result-verification, are excluded.

The experimental results are illustrated with speedup curves. For each application, we give the speedup curves using Maotai 2.0, Maotai 3.0 and TL-2. In the discussion below, N refers to the number of processes.

To ensure fair comparison, the same serial implementation of each benchmark application is used as a baseline for calculating speedups of all parallel programming platforms. Each run is repeated for 10 times and the geometric mean is used.

A. Maotai 3.0 outperforms TL-2 in high-contention cases TSP, LL and BT

The Travelling-Salesman Problem (TSP) algorithm [10] uses the branch-and-bound depth-limited search approach to identify the shortest path solution. The 33-city case ftv33.atsp from TSPLIB95 [11] is used.

In this algorithm, the priority queue (storing partially-evaluated tours) is the shared object. First, the master process pushes the root tour into the priority. Then, in a loop, each process pops a tour. If the tour is small, it will be evaluated serially; otherwise, sub-tours will be created and pushed into the priority queue.

In the TL-2 implementation, the shared priority queue is pushed and popped by transactions. High contention of the priority queue results in the poor speedup of 7.03 in TL-2, as shown in Figure 3.

In both Maotai 2.0 and 3.0 implementations, the priority queue is allocated as a view. The speedup of Maotai 3.0 is 12.92, which is 84% better than the TL-2 implementation, as shown in Figure 3. However, Maotai 3.0 is only 3% slower than Maotai 2.0, which has a speedup of 13.28. This small overhead can be attributed to the automatic detection of view accesses.

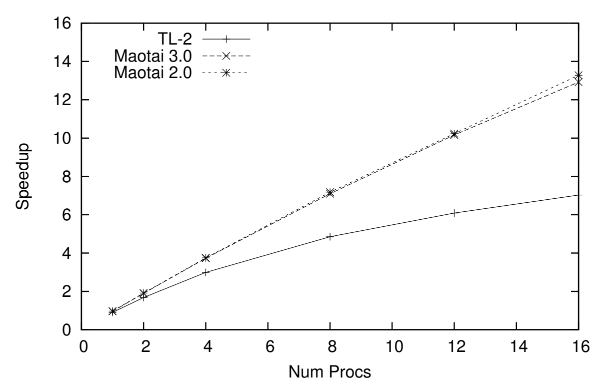

Figure 3. Speedup of TSP

Linked-list (LL) inserts nodes in an ascending-ordered singly-linked list, and deletes the nodes afterwards.

In both Maotai implementations, the entire linked-list is allocated as a SWV, while in the TL-2 implementation, natually each insertion/deletion is put into a transaction. Size of the linked-list is set to 4096.

At $N = 16$, speedup of Maotai 3.0 is 13.59, which is 26% better than TL-2 (10.79) as shown in Figure 4. Maotai 3.0 is only 4% slower than Maotai 2.0.

Binary Tree (BT) constructs a binary tree in parallel and uses a task queue for load balancing. When a node is explored, a small amount of dummy work is done, then

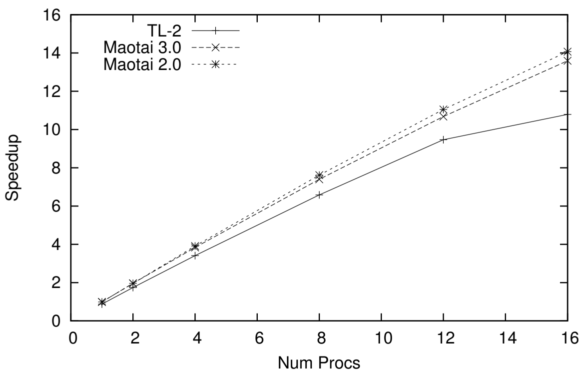

Figure 4. Speedup of LL

based on the id of the node, it works out whether the node has a left child and/or right child. The left child id is $curr.id * 2$ and right child id is $curr.id * 2 + 1$. Left children are always evaluated immediately and right children are pushed into the task queue for future evaluation. Idle processes pop unexplored nodes from the task queue, until the entire tree is explored. In Maotai 2.0 and 3.0 implementations, the task queue is allocated as a SWV, whereas in the TL-2 implementation, the task queue is accessed by short transactions. The depth of the tree is set to 21.

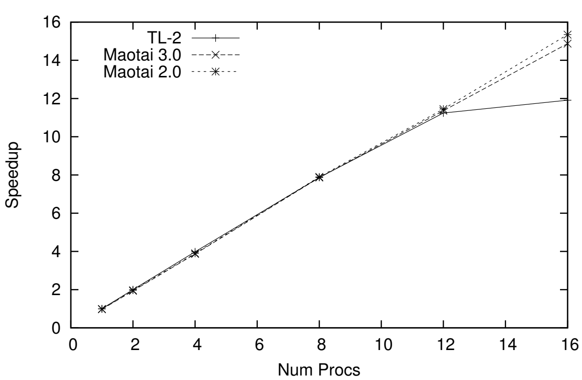

Figure 5. Speedup of BT

At $N = 16$, speedup of Maotai 3.0 is 35% better than TL-2 as shown in Figure 5. This is another case that lock-based implementations performs better than transactional memory. Again, speedup of Maotai 3.0 is only 3% worst than Maotai 2.0.

The above applications show that TL-2 is inferior to Maotai 3.0 in terms of performance. The slight performance drop of Maotai 3.0 against Maotai 2.0 in the above applications can be attributed to the automatic detection overhead described in Section II-D, as these applications have ten thousands of automatic view acquisitions throughout their executions.

B. PNN - multiple iteration algorithm updating a shared array

Parallel Neural Network (PNN) [12, 13] trains a back-propagation neural network in parallel using a training data set. In this experiment, the size of neural network is set to $9 * 40 * 1$, and the number of epochs is set to 400.

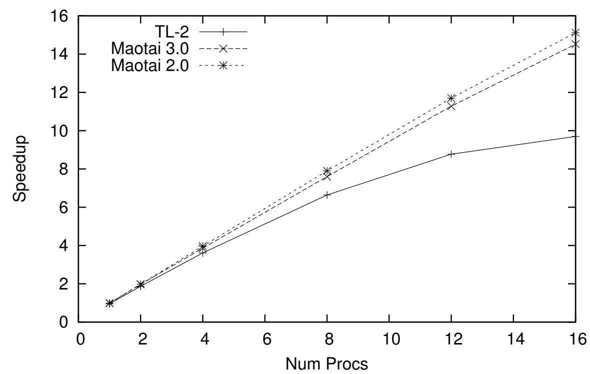

Figure 6. Speedup of PNN

At $N = 16$, speedup of Maotai 3.0 is 50% better than TL-2 as shown in Figure 6. In TL-2, there is a shared array with size of 4800, which all processes need to *increment* each element in this array at the end of each iteration. A short transaction would need to increment *each* element. This arrangement results in millions of transactions. Since the overhead of the transaction itself (start and commit) is not negligible, the sheer number of transactions has unnecessarily compromised the performance of TL-2. It is not possible to simply cover the entire array incrementation with a single transaction, because if one element aborts, the entire array operation will be aborted and redone, which would make performance worse.

However in Maotai 2.0 and 3.0, the entire array is allocated as a single-writer view (SWV), which removes unnecessary overheads from the TL-2 implementation and does not complicate the programmability since there is only one lock in the application and thus no deadlock issue in this case. As a result, there are only 25000 view acquires. At $n = 16$, speedup of Maotai 3.0 is only 4% slower than Maotai 2.0.

C. Barnes-Hut, Raytrace and Mergesort - low to moderate contention cases shows very little overhead in Maotai 3.0 automatic view access detection

Barnes-Hut [14] is a multiple-iteration algorithm where in each iteration, the master process constructs an octree for all particles in the model space based on their current location and mass, then the force acting on each particle is calculated using the octree, and based on the force, acceleration, velocity and position of the particle for the next iteration is also calculated. In our experiment, the number of bodies is set to 32768 and the number of iterations is set to 160. Due to the complexity of parallelizing the octree construction and its relatively small share of the workload, the octree construction is not parallelized. However, the workload of force calculation on each particle is unpredictable; therefore a work queue is implemented for load balancing. In the TL-2 implementation, accessing the work queue (i.e. incrementing the index) is handled by transactions; therefore Barnes-Hut is a short transaction case. In Maotai 3.0, the work queue index is implemented as a SWV.

Raytrace [14] is an embarrassingly-parallel case with uneven workload, so a task queue is used for load balancing (a row of pixels is a unit). The input file car.env is used, and anti-aliasing level is set to 400.

The Mergesort algorithm sorts a 1,000,000,000-element array. The Maotai 3.0 implementation comes from [5], and the TL-2 implementation is derived from the Pthreads implementation [5]. This is a barrier-based case, and transactions are not needed in the TL-2 implementation.

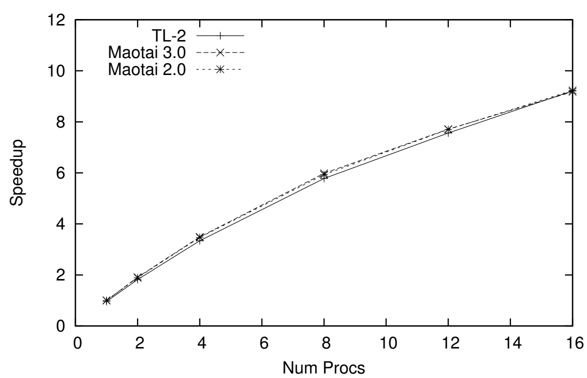

Figure 7. Speedup of Barnes-Hut

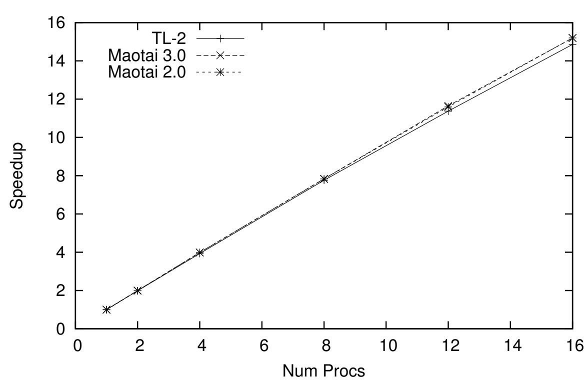

Figure 8. Speedup of Raytrace

In Barnes-Hut (Figure 7), Raytrace (Figure 8) and Mergesort (Figure 9), speedup of Maotai 2.0, Maotai 3.0 and TL-2 are nearly identical, except at $n = 16$, TL-2 is 2% worse than Maotai 2.0 and Maotai 3.0 in Raytrace. (Figure 9).

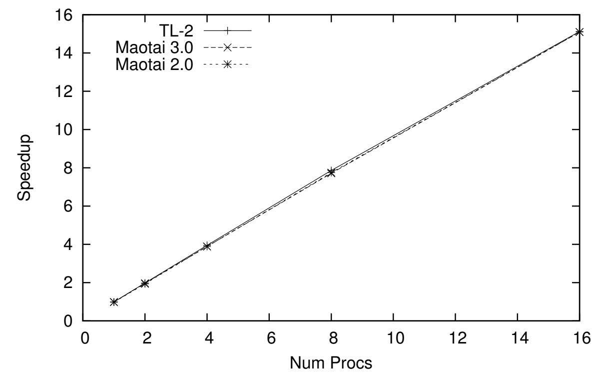

Figure 9. Speedup of Mergesort

The same performance of Maotai 3.0 and 2.0 demonstrates the extra overhead of automatic detection of view access in Maotai 3.0 is relatively trivial when the view acquisition is not frequent.

V. Related work

A. Colorama

Colorama is a data-centric shared memory model [8]. Like VOPP, shared objects are explicitly defined as "colours", as opposed to views in VOPP, and multiple blocks of share data can be allocated with the same colour in a similar fashion to the VOPP model. Under the colorama scheme, access to data owned by colours is also automatically acquired and released:

- Colour is acquired when its memory is first accessed.
- Colour is released when control leaves the scope of the colour acquisition.

The automatic detection of view access in Maotai 3.0 is similar to this model, but our scope of view acquisition is defined differently from the scope of colours. Also our automatic detection is implemented at run time with memory protection, while Colorama implements the automatic detection with compiler support.

B. Java S-DCS and Deterministic Parallel Java

Java S-DCS [15] and Deterministic Parallel Java (DPJ) [16, 17] are data-centric shared-memory models aiming at ensuring determinism in parallel codes.

Both models work by determining at compile-time whether it is possible for threads to have conflicting shared data access. If the compiler is sure that it is impossible to have conflicting data access between two threads, then these threads are allowed to run in parallel; otherwise, they will be run sequentially.

Both models let programmers define regions within a class, and each region in the object must be accessed atomically. DPJ allows recursive region subdivision using the region path list system, and method headers declares which regions it will read from or write to. This extra information helps the compiler to determine potential access conflicts between two threads, and thus allows more concurrency.

In cases like list/tree traversal, where access to the next node depends on result of current node (only known at runtime) and access pattern cannot be decided at compile time, execution will fall back to sequential access and performance will be affected. Since this approach depends on compiler support, complex, fine-grained applications like list traversal may not be fully parallelized.

VI. Conclusions and Future Work

The performance evaluation between Maotai 3.0 and TL-2 0.9.6 demonstrates that both performance and programmability of Maotai 3.0 surpasses TM systems. Comparison between Maotai 3.0 and Maotai 2.0 (which does not have automatic detection of view access) demonstrates that in our applications, automatic detection overhead is relatively low. Even in high-contention cases such as TSP, LL, BT and PNN, performance penalty is under 4%.

In addition, Maotai 3.0 is safer than TM systems because the data-centric nature of Maotai 3.0 ensures that it is data race free. Though the automatic detection scheme does not guarantee that there is no deadlock, deadlock can be avoided by using the VOPP sessions with deadlock-free mode offered in Maotai 3.0.

To address the deadlock problem, we are also investigating the View-Oriented Transactional Memory (VOTM) scheme, where deadlock-prone shared data are placed in a Transactional Memory View (TMV). Acquiring a TMV will begin a transaction and releasing the TMV will end the transaction. As mentioned earlier in this paper, the first access of view memory (include TMV) will acquire the view and leaving the scope of the view acquisition will release the view (end the transaction in case of TMV). Conflicting data access will result in aborting of one or more transactions, and they will be restarted at the location where the TMV is acquired. Here we only put deadlock-prone data into transactional memory, which is explicitly defined, whereas in traditional TM, the entire shared memory is transactional and requires programmers to explicitly mark sections that accesses shared memory data as atomic sections. In this way, the data-centric philosophy can be preserved in Maotai 3.0, while enjoying the deadlock-free advantage of TM.

Moreover, though the current automatic detection system has small runtime overheads, the applications with fine-grained, frequent view accesses will not scale easily. To eliminate the runtime overheads, we are investigating a high-level language for VOPP, that offers view management, garbage collection and safe pointer manipulation, so that data race can be partially detected at *compile time*, reducing the runtime overheads in the current Maotai 3.0 system.

References

[1] *OpenMP Application Program Interface Version 3.0*, OpenMP Architecture Review Board, May 2008.

[2] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. O'Reilly, 1996.

[3] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, September 2006.

[4] K.-C. Leung, Z. Huang, Q. Huang, and P. Werstein, "Data race: Tame the beast," *Journal of Supercomputing*, vol. 51, no. 3, pp. 258–278, March 2010.

[5] K. Leung, Z. Huang, Q. Huang, and P. Werstein, "Maotai 2.0: Data race prevention in view-oriented parallel programming," in *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE Computer Society, 2009, pp. 263–271.

[6] Z. Huang, M. Purvis, and P. Werstein, "Performance evaluation of view-oriented parallel programming," in *Proceedings of the 34th International Conference on Parallel Processing*. Oslo: IEEE Computer Society, June 2005, pp. 251–258.

[7] J. Zhang, Z. Huang, W. Chen, Q. Huang, and W. Zheng, "Maotai: View-oriented parallel programming on CMT processors," in *Proceedings of the 37th International Conference on Parallel Processing*, 2008, pp. 636–643.

[8] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas, "Colorama: Architectural support for data-centric synchronization," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 133–134.

[9] H. Franke and R. Russell, "Fuss, futexes and furlocks: Fast userlevel locking in linux," in *Ottawa Linux Symposium*, 2002.

[10] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

[11] G. Reinelt, "TSPLIB95," Institut für Angewandte Mathematik, Universität Heidelberg, Tech. Rep., 1995.

[12] R. F. van der Wijngaart and M. Frumkin, "NAS grid benchmarks version 1.0," NASA Advanced Supercomputing Division, NASA Ames Research Center, Tech. Rep. NAS-02-005, 2002.

[13] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2003.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of*

the 22nd Annual International Symposium on Computer Architecture, 1995, pp. 24–36.

[15] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 334–345.

[16] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[17] R. L. Bocchino, V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for Deterministic Parallel Java," University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2009-3032, 2009.