PARALLEL LOGIC PROGRAMMING ON DISTRIBUTED SHARED MEMORY SYSTEM

Zhiyi Huang, Chengzheng Sun, Abdul Sattar and Wanzu Lei

Knowledge Representation and Reasoning Unit
School of Computing & Information Technology
Griffith University, Nathan, Qld 4111, Australia

ABSTRACT

This paper presents an implementation of a parallel logic programming system on a distributed shared memory (DSM) system. Firstly, we give a brief introduction of Andorra-I parallel logic programming system implemented on multiprocessors. Secondly, we outline the concurrent programming environment provided by a distributed shared memory system—TreadMarks. Thirdly, we discuss the implementation issues of Andorra-I system based on TreadMarks, such as, shared memory consistency, synchronization, mutual exclusion, shared memory allocation. Finally, we discuss implementation results and further research issues for building a practically useful logic-based reasoning system on distributed shared memory systems.

Key Words: Distributed Shared Memory, Parallel Logic Programming System, Lazy Release Consistency, AND-parallelism, OR-parallelism

1. INTRODUCTION

Distributed Shared Memory (DSM) systems have been rapidly developed these years, with various kinds of memory consistency models proposed, such as sequential consistency [12], release consistency [7], entry consistency [4], lazy release consistency [11]. This sort of systems can provide a virtual shared memory space for the users on a network of workstations with comparably high performance, nearly as that on multiprocessors with physically shared memory in the area of numerical computing. The initial experimental results have demonstrated that they can support parallel numerical computing efficiently [11]. The significant achievements are mainly due to the following two reasons: the increased bandwidth of networks and the optimization of memory consistency models. The rapid increase of network bandwidth has made the distributed systems more and more attractive than the multiprocessors which are expensive and hard to scale up. However, there is little research on how to implement AI applications on distributed shared memory systems. We made the first attempt to combine them together by executing parallel logic programming system on DSM system. There are twofold research topics in this area. On one hand, we need do research on how to support symbolic reasoning and AI applications on the platform of DSM systems. Many AI applications are implemented on logic programming systems, such as Prolog, or on logic-based systems, such as hypothetical reasoning systems [15]. Parallel logic programming systems have been developed on multiprocessors since 1980s. However, because they are very expensive and hard to scale up, multiprocessors are not easily accessible. This limitation also makes the parallel logic programming systems uneasy to be available. To solve this problem, we need to investigate how to support their implementation on DSM systems which are based on networking communication. The nature of parallel logic reasoning is quite different from the nature of parallel numerical computing in the following aspects. First, memory access is often intensive and irregular. It requires intensive accesses of large shared memory. Memory addressing is often content-based, and locality of reference may not hold. Shared memory is randomly accessed by multiple processors. So the programmer can not always tell if one block of data is only used by one processor at a specific time period. Therefore, it is very hard for the programmer to optimize the access of shared data. Second, communication patterns are varied. Message passing varies in size and destination. Granularity of message units changes from time to time. The above aspects of nature very likely make the communications among processors frequent and manifold in DSM systems. How to decrease the communication overhead for AI applications based on DSM systems is one of the key issues in our research of AI-oriented parallel computing on distributed shared memory systems.

On the other hand, implementation techniques of AI applications on shared memory multiprocessors are, to some extent, different from those on DSM systems. The communications among processes in the latter cost more time than those in the former. Therefore, the scheduler, which is responsible for the load balancing, should be care-
fully designed to avoid excessive interactions among processes.

As the first step in the research area of AI-oriented parallel computing on distributed shared memory systems, we select the state-of-the-art parallel logic programming system—Andorra-I [16], which is implemented on the multiprocessor—Sequent Symmetry, as the example system, and port the system on TreadMarks [2], which is the up-to-date achievement in the area of DSM systems. From this porting project, we can accumulate knowledge and experience of implementing our own parallel logic programming systems [17, 18, 6, 9, 10] on DSM systems, while investigating how to improve DSM to support the AI applications.

We should mention that implementation of parallel logic programming systems on DSM systems is quite different from that on a particular distributed system, for there are some works done on transputer-based systems [20]. The reason is that our implementation is based on a well-defined DSM platform, which is supposed to be portable to various operating systems and architectures. Thus, based on this sort of platform, the applications are more portable on various distributed systems.

2. ANDORRA-I PARALLEL LOGIC PROGRAMMING SYSTEM

A lot of works have been done on parallel execution of logic programs [14, 5, 1, 16, 8, 17]. There are a few systems implemented on multiprocessors, one of which is the Andorra-I system which exploits both determinate dependent AND-parallelism and OR-parallelism.

The Andorra-I system is based on the Basic Andorra Model, which transparently exploits determinate dependent AND-parallelism and OR-parallelism in logic programs. In Andorra-I, a group of workers, which are abstract processing agents (processes or processors), cooperatively explore AND- and OR-parallelism. A worker can be regarded as an integration of engine and scheduler. An engine is responsible for the execution of logic programs. Andorra-I engine is essentially based on the efficient stack management techniques of WAM [19]. It has four stacks, heap, local stack, choicepoint stack and trail. They are composed of the so-called execution tree. All these data structures are allocated in the shared memory of the multiprocessor, and are shared by multiple workers. The workers move around the execution tree in order to find tasks.

The computation in Andorra-I has the following properties. First, every engine's stacks are accessed intensively, expanding or contracting very rapidly. The engine just operates on these stacks to get the inference results. Second, most of accesses to memory are content-based, e.g., dereference operation. Third, communication among workers are frequent and greatly varied. For every inference step, busy workers have to check some flag variables to see if other workers are requiring work. Idle workers are busy checking if some worker has enough work to be given. Therefore, if there is no work transfer between workers, the passed message is only several bytes; if there is work transfer between workers, however, the passed message is a package of several stacks and registers. In Andorra-I this kind of communication is saved by the physically shared memory. These computation properties may degrade the performance of the current DSM systems since DSM is based on message passing and maintenance of its consistency is time-consuming.

Most shared memory multiprocessors support the canonical consistency model, sequential consistency (SC) [12]. In sequential consistency model, every update in the shared memory is immediately visible to all of the processors. This property has been taken advantage of very often in Andorra-I system.

3. TREADMARKS SYSTEM

The TreadMarks is a DSM system for standard Unix systems such as SunOS and Ultrix. It is based on the lazy release consistency model, which can greatly avoid unnecessary message passing compared with other consistency models. Experimental results indicate good performance in the applications of numerical computing [11].

Lazy release consistency (LRC) is an improvement of release consistency (RC) [7]. LRC and RC are based on the presumption that "mutual exclusion of accesses to a variable is needed if there is at least one write operation among the accesses"¹. They permit a process to delay its updates on shared data until certain synchronization accesses occur. Therefore, the time for making the shared memory consistent is actually the same as the time of mutual exclusion. The difference between RC and LRC is that RC requires ordinary read or write accesses to be performed globally at the next release, whereas LRC requires only that ordinary read or write accesses be performed with respect to some processes which acquire the released lock. Through the delay of invalidation, the unnecessary message passing is greatly reduced and better performance is achieved in LRC.

TreadMarks provides a library of functions as its API for the users. The synchronization between processes is through the operations Tmk.lock.acquire, Tmk.lock.release and Tmk.barrier. Their parameters are lock or barrier identifiers whose status are operated by TreadMarks. The interface is simple but powerful.

¹according to a seminar given by Prof. Willy Zwaenepoel.
4. IMPLEMENTATION ISSUES

From the porting project, we find the following issues discussed below are important for the improvement of performance.

4.1. One Writer Multiple Readers

We start the discussion of implementation issues in distributed shared memory systems with the One Writer Multiple Readers scenario. Consider the following abstracted programs:

```c
P1
for(;;){
    ...
    shared_richness++;
    if(shared_richness>100){
        /* get task from P1 */
    }
    ...
}
```

```c
P2
for(;;){
    ...
    Tmk_lock_acquire(0); Tmk_lock_release(0);
    shared_richness++;
    richness=shared_richness;
    Tmk_lock_release(0);
    ...
    if(richness > 100){
        /* get task from P1 */
    }
    ...
}
```

Process P1 is the only writer for shared.richness. Other processes are readers for shared.richness. Suppose P2 is one of the readers. Please note there is no mutual exclusion primitive, such as, LOCK, UNLOCK, in the programs. Since writing an integer is an atomic operation, it is impossible for the readers to get the wrong value (part of new value and part of old value) even though there is race condition for the accesses of shared.richness. If the readers do not mind whether they get the newest value of shared.richness or not, i.e., the correctness of the programs does not depend on whether the value of shared.richness is the most up-to-date one or not, the above programs will correctly run on a multiprocessor. Moreover, under the sequential consistency model, the readers can get the up-to-date value.

However, if we run the programs under lazy release consistency model, P2 may never get task from P1. The reason is that the shared memory consistency is carried out by piggy-backing on the implementation of synchronization in TreadMarks. If there is no synchronization primitive, such as, Tmk_lock_acquire, Tmk_lock_release, in the programs, there is no chance to make the values of shared.richness in P1 and P2 consistent. So even if the value of shared.richness in P1 becomes 10000, the value of shared.richness in P2 may still be the initial value 0. This scenario is contradicted with the LRC presumption in Section 3. Associating the shared memory consistency with mutual exclusion is not proper in some applications. For example, in AI algorithms, there are some parameters which are used as hints. These values are updated by only one process and read by other processes. The accurate values are not important for the readers. Provided the writer can be sure the readers will not read the partial value of them, there is no need to use mutual exclusion when accessing them. Even though the readers do not mind if they get the newest values or not, they still want to know the change of the values sometime later. But in TreadMarks the reader will never get the changed values if the synchronization primitives are not used.

To solve the problem in our porting project, one approach is to add synchronization primitives in the programs. The following is the programs revised for TreadMarks.

```c
P1
for(;;){
    ...
    Tmk_lock_acquire(0);
    Tmk_lock_release(0);
    shared_richness++;
    richness=shared_richness;
    Tmk_lock_release(0);
    ...
    if(richness > 100){
        /* get task from P1 */
    }
    ...
}
```

Another approach is to provide a mechanism which can periodically make the shared memory consistent. In this way, we need not add lock primitives in the programs and the programs can run correctly without any change. We call this approach periodic flush. In the approach, the user can set a time period for the system to make the shared memory consistent periodically. If the time period is set to \( T \), then after every \( T \) time period the system will automatically make the shared memory consistent. We have done experiments on different time period \( T \), from 1 microsecond to 100 microseconds [13]. For our benchmark programs, if the time period is too small, e.g., 1 microsecond, the programs with periodic flush run slower than those with lock primitives. However, if the time period \( T \) is set 10 microseconds, the programs with periodic flush run 10 times faster than those with lock primitives. Of course, if the time period is set much larger, the programs will run much faster, but the readers will not notice the change of values in time. So the time period for the writer to flush its updated values should be decided by the user according to different applications. We think that integrating the periodic flush mechanism into LRC will make LRC more flexible for programming.

4.2. Synchronization and mutual exclusion

The Andorra-I uses special lock variables to implement synchronization between workers. But TreadMarks only provides one kind of lock primitives. We have two methods to implement synchronization in Andorra-I based on TreadMarks. One is to use TreadMarks' lock primitives to
implement lock variables. Another is to replace the lock variable with lock identifier one by one. The following macro definitions are used in the first method.

```c
#define LOCK(X) \
    { \
        while(1){ \
            Tmk_lock_acquire(0); \ 
            if((X)==0){ \
                (X) = 1; \
                Tmk_lock_release(0); \ 
                break; \
            } \
            Tmk_lock_release(0); \
        } \
    }

#define UNLOCK(X) \
    { \
        Tmk_lock_acquire(0); \ 
        (X) = 0; \ 
        Tmk_lock_release(0); \
    }
```

This method is more nature for the implementation of lock variables in TreadMarks. You need not change too much in the source code and can keep the semantics of LOCK and UNLOCK consistent with the original ones. But since only one lock is used, the bottle neck problem may arise in the method.

The second method replaces every lock variable with lock identifier. But the parameter X in LOCK(X) and UNLOCK(X) is changed from a lock variable to a lock identifier. From the experiments [13], the second method is more efficient.

In Andorra-I, logical variables may be shared by several workers. Thus, when these variables are accessed, mutual exclusion is required. The straightforward method is to assign a unique lock to each variable. Because there may be hundreds or thousands of variables in the system, it is not economical to do so and the program may become intolerably complex. To avoid those problems, we reserve a set of locks and use simple hash function to map the variables to the lock set. The following macros map the variables to locks 1 to 4096. The function parameter is the address of the variable.

```c
#define V_LOCK(X) \ 
Tmk_lock_acquire((unsigned int)(X)%4096 + 1)
#define V_UNLOCK(X) \ 
Tmk_lock_release((unsigned int)(X)%4096 + 1)
```

From our experiments, this method does not cause performance degradation compared with the first straightforward one.

When implementing Andorra-I on DSM systems, there are three factors to effect the system performance. The first one is frequent memory consistency. The second one is the fine task granularity in original Andorra-I. The fine granularity causes a lot of communication overhead between processes, since the tasks migrate from one process to another very often. The third one is the dependent AND-parallelism, whose shared variables result in more communication between processes. On multiprocessors, it may not cause serious performance problem because of the support of hardware. But on a distributed shared memory system, communication cost plays an important role in performance problem and can make it even worse. Therefore, granularity control of parallelism should be carefully implemented in the ported Andorra-I system, and the kinds of parallelism exploited should also be carefully chosen. These are important issues which need to be further investigated.

4.3. Allocation of shared memory

There are two ways to allocate shared memory in Andorra-I. One is static declaration with the decoration `shared` in front of variable definitions. Another is dynamic allocation using the function `malloc`. But TreadMarks only provides dynamic allocation for the shared variables. Thus we have to transform the statically declared variables into dynamically allocated variables. The strategy is to put all the statically declared variables into a structure and to allocate shared memory space for it using `Tmk_malloc`. In this way, we need not change too much Andorra-I source code except one header file (see [13] in detail).

Even though the above problem is solved for porting, we should be careful with the shared memory allocation, which may cause performance problem if it is not properly done in TreadMarks. In Andorra-I, the engine in every worker has several stacks in shared memory. Most of the time they are accessed by the engine itself, while sometimes they may be accessed by other workers. If the stacks in each worker are located with those in other workers together in a mixed way, then the possibility for the stacks in different workers to share one page is very high, and this will cause extra message passing, such as, page `diff` in TreadMarks. To avoid this kind of unnecessary message passing, we locate the stacks in the same worker in a continuous memory space. Therefore, the stacks in two different workers share at most one page. From our experiments [13], the performance is greatly improved by this arrangement.

5. CONCLUSIONS

We discussed the implementation issues of parallel logic programming system on distributed shared memory system. These issues are very important and need to be
further investigated in parallel computing based on distributed shared memory systems. Through porting Andorra-I to TreadMarks and making preliminary experiments, we think improving the performance of AI applications running on distributed shared memory systems is very promising. This task should be fulfilled by twofold investigation: one is to better support AI applications by improving the DSM systems; another is to adapt the AI applications to the DSM systems by changing the design ideas which originated from the multiprocessors. Further research will be conducted on these issues.

Acknowledgments
The research is supported by an ARC(Australian Research Council) large grant(A49601731).

6. REFERENCES