A Compiler Supporting Distributed Shared Memory System

Rung-Ji Shang[†], Wen-Yew Liang[†], Jen-Chiun Lin[††], Feipei Lai[†††]

[†]Dept. of Computer Science and Information Engineering
[††]Dept. of Electrical Engineering
[†††]Dept. of Computer Science and Information Engineering
and Dept. of Electrical Engineering
National Taiwan University, Taipei, Taiwan

Abstract

Distributed systems receive much attention because parallelism and scalability are achieved with relatively low costs. Many researches favor distributed shared memory (DSM) systems since the concept of shared-memory can increase programmability without much performance loss. In this paper, a new language for distributed computing is introduced. We have implemented a compiler which will generates executables running on the Adsmith DSM system. What we provide is a simple, efficient, general purposed, and portable solution. Programming in the language can greatly reduce the inconveniences programmers need to face when they only have low level library routines at their hands. With the help of our compiler, effective optimizations including bulk transfer, non-blocking lock and non-exclusive lock are done to help programmers from doing routinely hand-tuned optimizations.

1 Introduction

Even though current computer technologies have been greatly advanced in architecture and VLSI designs, many scientific application still can not be solved in acceptable time. A good chance to solve this problem is to explore the parallelism in these applications and to utilize the multiprocessor to solve the application.

Due to the cost-effective considerations, more and more current parallel processing systems are constructed by connecting low cost personal computers or workstations with cheap local area networks. These kind of environment are often called *network of workstations* (NOW). To make parallel program development in these environment easier, *distributed share-memory* (DSM) was proposed[10, 3, 2, 1]. It provides the users shared-memory programming environment on top of physically distributed memory systems. This can prevent the tedious and error-prone problems which usually come with directly using message-passing paradigm to write parallel programs. The architecture of a typical DSM system is shown in Fig. 1.

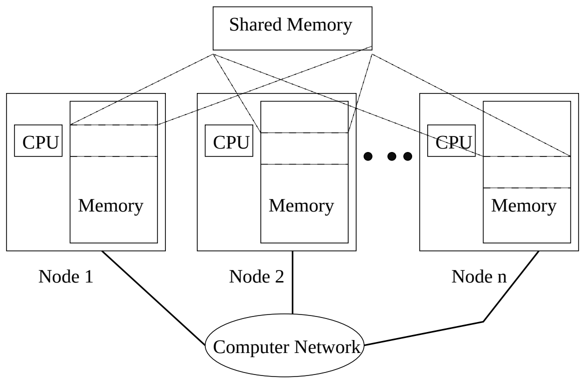

Figure 1: DSM Architecture

Many of current DSM systems are software DSMs. The advantages of a software DSM is that it can easily incorporate optimization techniques such that many of the message passing overheads can be eliminated. The programmers can also then affect the behaviors of the underlying DSM through the provided interface to improve the performance of the applications. Since adopting an suitable optimization options is not an easy work, new languages and compilers are usually designed to help the programmers to accomplish this work easily.

We have designed such a parallel language based on the C programming language. The parallel language provides the users a shared memory environment. Our purpose is to design a simple, portable, and powerful language for developing general purpose distributed applications. We also have implemented a compiler for the language, which will generate parallel codes for the Adsmith DSM system[8]. The compiler eases programmers off the responsibilities such as caring the consistency maintenance and the performance tuning. It will perform simple optimizations such as data prefetching, computations reordering, and data partition, etc. The programmers can then focus on more subtle aspects of

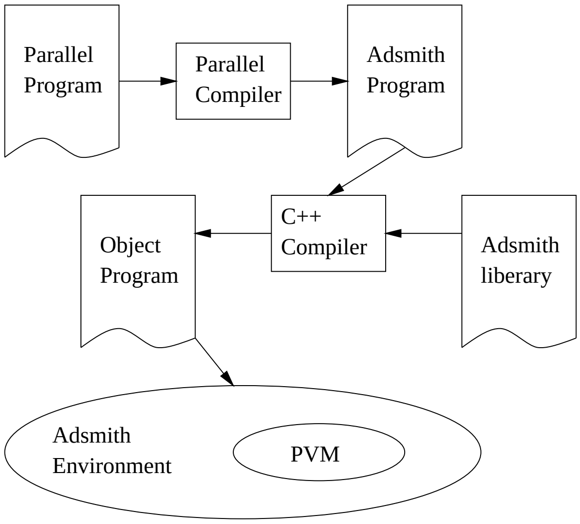

Figure 2: System Architecture

applications.

In this paper, we will describe the language specification and the implementation of our compiler. The rest of the paper is organized as follows. In Section 2, we take a look at the overview of the system. In Section 3, the language specification is introduced. Implementation of the compiler and optimization techniques are described in Section 4. Finally, we conclude this paper with future works in Section 5.

2 Overview

This section describes the system overview, including the language, the compiler and the runtime system.

2.1 System Architecture

Fig. 2 illustrates the architecture of our system. First, a source program will be translated by our compiler to a C++ program with appropriate Adsmith system calls. Then, *g++* compiles the intermediate file into an object code and it will be linked with Adsmith library to produce an executable code. The code can be executed on a group of homogeneous machines running Adsmith runtime environment. Since the system is ultimately based on PVM, it should be able to run on every machine where PVM exists[11]. Currently, our system have been tested on Sparc workstations running SunOS and personal computers running Linux.

2.2 The Adsmith DSM system

Adsmith is implemented as a user-level library in C++ with PVM as the communication subsystem. The primary goal of Adsmith is to provide

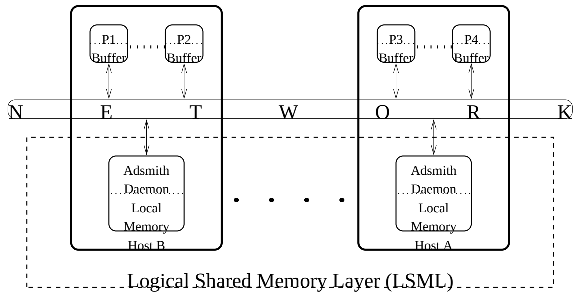

Figure 3: Organization of Adsmith

a low-cost, portable, and efficient DSM for networks of workstations (NOW). It can be viewed as adding a DSM layer on top of PVM. Both the PVM message-passing library and the Adsmith shared-memory library are accessible at the same time. Adsmith is an object-based DSM system; it differs from traditional page-based DSM systems in that it views the shared memory as consisting of many objects sharable by all processes. The basic organization of Adsmith is shown in Fig. 3.

Each daemon will interact with the application processes to provide shared-memory services. Shared objects are distributed to the memories of the participating hosts. Adsmith provides primitives to create and allocate shared objects, accesses to shared objects, and operations to synchronize among processes. Adsmith also incorporates many techniques to optimize its performance, including supports for release consistency memory model[5], different coherence protocols, load/store-style memory accesses, object-based multiple writer protocol, bulk transfer, prefetch, nonblocking store, and other specialized accesses, etc.

2.3 The Language and the Compiler

We have designed a general purposed parallel language based on the C language, and have implemented a compiler conforming to the specification. The detail will be discussed in section 3 and section 4 respectively.

3 The Language Specification

The language we have designed is a superset of the C programming language, which is the most widely used programming language today. We add language constructs supporting shared memory management, synchronization, dynamic memory allocation, and process management. Some miscellaneous functions that are not added into the language but useful are provided in a library.

3.1 Process Management

A process is declared and defined mostly like an ordinary function except the adding of a new keyword `process` to distinguish one from a function.

We use keyword `create` to spawn a new process, the value returned is of a special type `process*` which holds the relative attributes of the created process. The process structure is implementation dependent, all functionalities are provided by the library routines and direct access to the members of it should be avoided. To kill a process, just destroy the corresponding process management structure of that process. If the process is already terminated, the destroy operation does nothing. The `create/destroy` pairs acts similarly to `new/delete` pairs.

3.2 Shared Memory

Shared memory is the major issue in our research. The interpretation of a variable in C depends on three main attributes: storage specification, type specification and qualification. Following the convention of C, a shared variable can be allocated statically, automatically or dynamically.We introduce a new keyword `shared`, which is a type qualifier, to distinguish shared data and non-shared data.

Although global variables are visible by all processes, they are not shared by default. Many existing libraries, such as the standard I/O library, use global variables more or less. Making global variables shared by all processes will confuse the linker and reduce the re-usability of the original libraries. A private global variable has multiple individual copies on different processes. Global shared variables can be static or extern, they share the same scope rules as ordinary global variables. The use of static shared variables is encouraged because the compiler knows that they are visible only in the file scope and can optimize them more aggressively. An extern shared variable means it may be declared elsewhere.

Automatic shared variables are also supported. They obey the same scope rules as ordinary automatic variables. Unlike an global shared variable, whose life-time begins after its declaration and ends at the end of the file, a automatic shared variable's life-time ends at the end of the enclosing scope. We must carefully examine the execution of processes that use automatic shared variables and ensure that those variables are not out of scope when they are in use. For instance,

```
/* process declaration */
void p( shared int* psi ) process;
shared int gi;  /* global */

void main()
{
  {
     shared int ai;  /* automatic */
     static shred int si; /*static*/
     process* p1=create p( &gi );
     process* p2=create p( &ai );
     process* p3=create p( &si );
     /* p2 is waited here
        but p1 & p3 is not */
     /* then ai goes out of scope */
  }
}
```

we can see that since process `p2` uses variable `ai` and during the execution of process `p2`, automatic shared variable ai should be alive, so the parent process must wait for process `p2` to terminate and then leave the scope. Conversely, process `p1` uses only global shared variable `gi`, therefore the parent process does not need to wait for `p1`'s termination. From the above example, it should be clear that automatic shared variable should be used with care since it could lead to lower parallelism. Shared static variables declared in scopes other than the global scope have the same visibility as automatic shared variable but the same life-time as global shared variable. Referencing these variables will not cause 'implicit barriers' as those automatic ones will do.

The C programming language does not provide dynamic memory management facilities. Instead, it is achieved by using standard library calls `malloc()` to dynamically allocate memory and `free()` to dynamically free the allocated memory. If we follow this convention and provide library routines such as `shared_malloc()` or `shared_free()`, the semantic will be complicated since the programmers have to use different functions for shared data and non-shared data. Thus our language borrows the `new/delete` constructs from the C++ language to handle dynamic memory management in a uniform way, for both shared memory management and non-shared memory management. Here is an simple example,

```
void main()
{
  /* non-shared data */
  int* poi=new int[10];
  /* shared data */
  shared int* psi=new shared int[10];
}
```

3.3 Structures and Unions

Structures and unions are important constructs for data aggregation. A member of a structure or a union can not be declared shared, but a structure or union can be instantiated as shared data. A structure or a union can contain semaphore or barrier members. (See Section 3.5. If a structure or a union definition contains semaphore or barrier members, it can only be instantiated as shared. The underlying principle is that a structure or a union is viewed as a piece of continuous memory, so all members in a structure or a union can only be totally shared or non-shared.

3.4 Arrays and Pointers

Declaration of shared arrays is similar to declaration of shared variables, it can be allocated statically, automatically or dynamically. Pointers can be categorized into three classes: *ordinary pointers* (conventional C pointers), *pointers to shared data*, and *shared pointers*. An ordinary pointer points to a piece of private memory and it cannot be assigned an address of a shared data, including explicit casts. A pointer to shared data points to a piece of shared memory and it can be casted to an ordinary pointer with explicit casts. Although casting a pointer to shared memory to a ordinary pointer is dangerous, it may be useful when we need to use existing library functions on shared memory. A shared pointer is itself a shared data and can only point to a piece of shared memory. Here is an example for various situations,

```
void main()
{
  int i;
  shared int si;
  int* pi;
  /* pointer to shared memory */
  shared int* psi;
  /* shared pointer
     to shared memory */
  shared int* shared spsi;
  /* shared pointer to
     non-shared memory*/
  int* shared spi;  /* wrong! */
  psi=&si;       /* ok */
  pi=(int*)psi;  /* ok */
  spsi=psi;      /* ok */
  psi=spsi;      /* ok */
  pi=(int*)spsi; /* ok */
}
```

3.5 Synchronization

To guarantee correct serialization [4], our language also support some synchronization mechanisms. Three forms of synchronization constructs are provided: semaphores, locks, and barriers.

Semaphores are special shared variables which are provided for synchronizing different processes. They need be declared with the keyword shared, since they are always shared. Semaphores are treated by the compiler in a different way as that of shared variables. Only four operations are allowed: initialization, signal, wait and reading the value of a semaphore (which will be an integer value). A semaphore is only initialized once and only the other three operations are allowed after its initialization. Signals and waits need not be paired in a process or a function scope.

Unlike semaphores, locks are not special shared variables. Instead, it is a conceptual flag associated with each shared variable other than the special shared variable for semaphore and barrier. When shared data are to be accessed in exclusive mode, locks are often used. For example,

```
shared int i, j;

void main()
{
  ...
  /* may print old value */
  printInt( i );

  lock i;
  /* print the newest value */
  printInt( i );
  unlock;

  /* may be useless */
  /* a warning generated */
  j++;

  lock j;
  j++; /* exclusive-mode access */
  unlock;
}
```

Our language also support multi-locks, i.e. more than one variable can be locked at the same time. Multi-locks are atomic operations. If all locks can not be acquired at one time, none of them will be granted. A multi-lock does not always act the same as the corresponding multiple single-locks. For example, the following program segment may produce dead-lock if the parallel program is not well designed.

```
shared int i, j;

void p() process {
  lock i; lock j;
  unlock; /* unlock j */
  unlock; /* unlock i */
}
void q() process {
  lock j; lock i;
  unlock; unlock;
}
```

While the following segment will be dead-lock free.

```
shared int i, j;

void p() process {
  lock i, j; /* a multi-lock */
  unlock;    /* unlock i, j */
}
void q() process {
  lock j, i;
  unlock;
}
```

Although multi-locks have the above advantages, they may suffer from the problem of worse performance because of the complex design. Thus further supports by the underlying run-time system are necessary. After using locked shared variables, the programmer must explicitly unlock them. An unlock statement without operands unlocks variables in the nearest locked statement. The other things that should be noted when using locks include: locks can only be applied to shared variables; a shared variable cannot be locked again by the same process if it is still locked and of course it can not be unlocked if it is not locked; and at the end of any process scope, all locked shared variables must be unlocked.

The third form of synchronization is barrier. A barrier is a special shared variable for synchronization of different phases among a group of processes. A barrier cannot be initialized and cannot be locked or unlocked. The number of processes needed to pass a barrier must be given at the barrier point (i.e. the `barrier_until` construct). The value of a barrier can be read as an integer that indicates the current number of processes that have arrived at the barrier point. In release consistency memory model, all subsequent references to shared data after a barrier point will get the up-to-date value.

It is programmers' responsibility to use synchronization mechanisms carefully to ensure the correctness of their programs. For example, the result of the following program may be 1 ,or 2, since `barrier_until` operation can not make the updating to `i` exclusively and race condition occurs.

```
shared int i=0;
barrier b;

void p() process {
  i++;
  barrier_until( b, 3 );
}

void q() process {
  i++;
  barrier_until( b, 3 );
}

void main()
{
  process* p1=create p();
  process* q1=create q();
  barrier_until( b, 3 );
  printInt( i ); /* maybe 1 or 2 */
}
```

To make the above program correct, each `i++` statement should be translated to:

```
  lock i;
  i++;
  unlock;
```

4 Implementation

In this section, we describe the implementation of our compiler. We also emphasize how our compiler optimize shared memory accesses and synchronizations.

4.1 The Runtime System

The object code is built with Adsmith runtime library and run on top of Adsmith runtime environment. Adsmith [8] is an *object-based* DSM [10], which is built completely on top of PVM [11]. In Adsmith, the shared memory consists of many shared objects. Accesses to the shared memory must go through these shared objects. Adsmith provides primitives to create and allocate shared objects, accesses to shared objects, and operations to synchronize among processes[9].

4.2 Shared Memory

In the C programming language, a programmer can allocate memory from static storage (usually data segment), stack or free storage. From programmers' point of view, shared memory can also be allocated like private memory. All the shared memory allocations are through Adsmith system call `adsm_malloc()`, and the deallocations are through `adsm_free()`. Even though static shared memory is declared in global area, it is still allocated dynamically (in our implementation, they are initialized in the main procedure). They are very different from static private memory which may reside in a data segment. For automatic shared variables, they are called automatic not because they reside on the stack as ordinary ones, but because they shared the same scope rules as ordinary ones. So use automatic shared variables are not more efficient than dynamically allocated shared memory, they are allocated dynamically in the underlying system.

Our compiler will distinguish dynamic memory allocation of shared memory and non-shared memory. It will translate a `new` expression to a `adsm_malloc()` or a `malloc()`, and a `delete` expression to a `adsm_free()` or a `free()` respectively.

In Adsmith, each shared object (no matter how much memory it occupies) can be associated with a "lock flag". Currently our compiler do not support fine-grain locks of a structure, therefore, to lock a member of a structure is the same as to lock the whole structure. For example,

```
typedef struct p {
  int x, y;
} Point;

void main() {
  shared Point* p1=new shared Point;
```

```
    /* the same as lock the whole *p1 */
    lock p1->x;
    unlock;
  }
```

locking member x of p1 is the same as locking p1.

A pointer to shared memory is not like a pointer to non-shared memory. This is because the address of a shared object in one process may not be the same as that in the other processes. Thus, the compiler will translate the local address of a shared object to a globally recognizable address before the address is passed to other processes through the shared memory. Currently, the compiler will not do any optimization about pointer usage, except a very special case involving indexed array access. For example,

```
void main() {
  shared int *a=new shared int[100];
  shared int *amax=a+100;
  int i;
  /* encouraged,
     optimization can be done */
  for( i=0; i<100; i++ )
    a[i]*=2;
  /* discouraged,
     no optimization can be done */
  for( amax=a+100; a<amax; a++ )
    *a*=2;
}
```

As the above example, current implementation does not support pointer/index conversions.

4.3 Synchronizations

Semaphores and barriers are supported in Adsmith and the compiler directly translates them to the corresponding ones provided by Adsmith. The compiler's flow-analysis guarantees that a semaphore is initialized once; if it cannot determine whether a semaphore is initialized or initialized more than twice, it will report a warning message. The use of each barrier_until operation will create a barrier point. The compiler will generated the needed load and store function calls at proper location according to the load/store-style access method supported by Adsmith.

Current version of Adsmith does not support multi-locks, so simulating multi-locks with multiple single-locks is not so efficient. The compiler let the programmers choose whether to simulate atomic multi-locks, or simply replaces each multi-lock with a sequence of single-locks. For multi-locks, the compiler will generate code that locks each shared variable in a special order so as to minimize the chances that deadlocks may happen. Currently, the compiler can guarantee that multi-locks of statically declared shared variables deadlock-free. For pointers to dynamically allocated shared memory, deadlock-free cannot be guaranteed.

4.4 Bulk Transfer

Adsmith provides *bulk transfer* on shared memory accesses, i.e. continuous accesses can be aggregated to send in fewer messages. Our compiler will optimize shared variable accesses (none pointer-based) and indexed array accesses within two barrier points using bulk transfer. It will group individual accesses to several *bulk* accesses to reduce the number of messages. But for general pointers and irregular indexed array accesses (such as an array using its elements as its indexes), the compiler will conservatively generate codes that guarantee the correctness of the execution. The algorithm for bulk-transfer optimization is listed in Fig. 4.

bulk_transfer_for_barriers() {
  parse_tree tree;
  for (*each barrier point* bp $\in$ tree) {
    write_set wset = ϕ;
    for (*each barrier point before* bp) {
      for (*each write access* w)
        wset = wset $\cup$ {w};
    }
    generate code of wset;
    generate code of bp;
    read_set rset = ϕ;
    for (*each barrier point after* bp) {
      for (*each read access* r)
        rset = rset $\cup$ {r};
    }
    generate code of rset;
  }
}

Figure 4: Algorithm of bulk transfer optimization for barriers

For any critical region bounded by a lock/unlock pair, the compiler will also try to aggregate the accesses into bulk accesses, just like barrier points. The algorithm is shown in Fig. 5.

4.5 Non-blocking Lock Optimization

Since using locks usually involves message exchanges among processes, which is a very expensive operation, the compiler will use non-blocking and scheduling to overlap communications and computations to enhance parallelism. For example,

```
shared int i;

void main() {
  int j, k;
  j=i*2;  /* 1 */
  lock i;
  i++;    /* 2 */
  unlock;
  k=j+1;  /* 3: depends on 1 */
```

```
bulk_transfer_for_locks() {
    parse_tree tree;
    for ( each lock l ∪ tree ) {
        for ( each matching unlock ul of l ) {
            find critical section cs
                bounded by l and ul;
            read_set rset = φ;
            write_set wset = φ;
            for( each read access r )
                rset = rset ∪ {r};
            for( each write access w )
                wset = wset ∪ {w};
            generate code of l;
            generate code of rset;
            generate code of cs;
            generate code of wset;
            generate code of ul;
        }
    }
}
```

Figure 5: Algorithm of bulk transfer optimization for locks

```
  j=i+k;  /* 4: depends on 2,3 */
}
```

The the compiler finds computations which are independent of the lock, it will generate code like

```
shared int i;

void main() {
  int j, k;
  pre-lock i;
  j=i*2;
  k=j+1;
  lock i;
  i++;
  unlock;
  j=i+k;
}
```

the `pre-lock` is the non-blocking version of `lock`. Both `j=i*2` and `k=j+1` are moved to the location after the `pre-lock` as dependencies are not violated. The `lock` in this translated code is to ensure the accomplishment of the `pre-lock` Fig. 6 presents the algorithm for the scheduling.

The *prelock_schedule* algorithm try to do more computation before the lock is granted to the process. Since the basic block enclosing the critical section is usually small, the computations will most probably ends before the specific locks are really grabbed. Because the message send/receive time is usually much more than simple computation time, thus the situation that the duration occupied by a lock increases due to this optimization can be expected to happen less often.

```
prelock_schedule() {
    for ( each lock l ) {
        move_set mset = φ;
        find matching unlock ul of l;
        find critical section cs
            bounded by ul and l;
        find basic block b1 before lock l;
        find basic block b2 after ul;
        b3 = cs ∪ b2;
        for ( each expression e ∈ b3 ) {
            if ( e can be reordered with respect
                to all other expressions ⊆ b3 )
                mset = mset ∪ {e};
        }
        if ( b1 == φ ∧ mset == φ ) {
            generate lock code of l;
            generate code of cs;
            generate code of ul;
            generate code of b2;
        }
        else {
            generate prelock code of l;
            generate code of b1;
            generate code of mset;
            generate lock code of l;
            generate code of (cs - mset);
            generate code of ul;
            generate code of (b2 - mset);
        }
    }
}
```

Figure 6: Algorithm of non-blocking lock optimization

4.6 Non-exclusive Lock

Logically, accesses with locks are performed in exclusive mode, but there are cases where it is unnecessary to go into exclusive mode. The compiler will treat these special read only accesses with more efficient calls. For instance, in the following example

```
shared int i, j;

void main() {
  lock i, j;
  j=i;
  unlock;
}
```

variable `i` is read-only in the critical section. Thus it is unnecessary to access them in exclusive mode, instead, it can be transformed to the following code

```
shared int i, j;

void main() {
  read-lock i;
  write-lock j;
  j=i;
  unlock i, j;
}
```

The `read-lock` and `write-lock`, which are provided by Adsmith, allow the shared data to be accessed in the *single-writer/multiple-readers* protocol. However, this optimization has side effects, for example, the following program segment will perform better if we lock `i` with a `write-lock` at the first time, because when we update it in the next critical region, it is likely that the data is already available locally.

```
shared int i, j;

void main() {
  lock i, j;
  j=i;
  unlock;
  lock i;
  i++;
  unlock;
}
```

For simple exclusive operations, the compiler will generate special code that migrates the computation rather than migrates data. For example,

```
shared int i;

void main() {
  lock i;
  i++;
  unlock;
}
```

will be transformed to

```
shared int i;

void main() {
  remote-computation( i, "[int] @++" );
  /* the latest value of i
     will be available here */
}
```

where `remote-computation` is provided in Adsmith as *active accesses*. However, this optimization has its limitation. Our compiler currently cannot analyze across different critical region, so programmers should have good disciplines using locks to maximize the performance. For example, the code

```
lock i;
 ... /* do some computation */
unlock;
lock i;
 ... /* do some computation */
unlock;
```

is better written as

```
lock i;
 ... /* do some more computation */
unlock;
```

because the compiler has more chance to perform optimization.

4.7 Function Calls

Due to the load/store style accesses provided by Adsmith, each data should be explicitly loaded from shared memory before they are referenced, and stored back to the shared memory after they have been modified. Foe example, the following code,

```
shared int a;

void f(shared int *i){
  (*i)++;
}

void g(){
  int b=a;
  f(&a);
  a+=1;
}
```

will be translated as follows,

```
shared int a;

void f(shared int *i){
  load *i;  /* ### */
  (*i)++;
  store *i; /* ### */
}

void g(){
  int b;
  load a;
  b=a;
  f(&a);
  load a;   /* ### */
  a+=1;
  store a;
}
```

Obviously, the above translated code involves too many unnecessary loads and stores. In fact, the number of loads and stores could be reduced through inter-procedure analysis. Thus, those statements marked by "/* ### */" can be removed. In addition, applying inter-procedure analysis can increase the chance to perform bulk transfer optimization. However, this optimization can only be used on static functions, it can not be applied to functions which may be invoked in other files.

4.8 Process Management

Our current implementation support both static and non-static processes. A static process definition ensure that the process definition is visible in current file only, and a non-static one is visible to all files. Every process will be translated into an ordinary C function which will be called by the process switcher, which is responsible for process scheduling. The compiler uses SPMD (Single Program Multiple Data) model to generate a parallel executable program. Creating a new process is actually spawning the same executable program and then switches to the corresponding function body in the main function.

Process creation and termination are directly supported by Adsmith. The compiler deals with both `new` and `delete` of processes by directly translating them to the corresponding Adsmith functions.

5 Conclusions

In this paper, we proposed a parallel language for the programmers to take the advantages of distributed shared-memory with less efforts. We also have built a compiler which processes the language and performs some effective optimizations such as data prefetching, bulk transfer, and non-blocking lock with the supports of the underlying DSM system.

Currently, one major thing we pursue is about data locality. Most shared-memory machines rely on data caching to hide the remote memory access latency. Data locality is an important issue because inappropriate data allocation and the false sharing[7, 12, 6] problem will increase many unnecessary coherence messages. This issue is especially important for distributed memory machines because the network latency is very high. Since Adsmith is an object-based shared memory system, the above problem can be reduced by contiguously allocating data accessed in a process as a shared object, and by partitioning data accessed by different processes into different objects. For applications with array-like data access behavior, data partition will perform better if spatial locality is further exploited. We will partition an array into sub-arrays of desirable size according to the access patterns to take advantage of spatial locality. The other one that we will do is to port some "real" benchmarks, such as those in Splash2[13], to test our system. It will also help to realize the system behavior and to accurately estimate the performance of various kinds of optimizations.

References

[1] Cristiana Amza, Alan L. Cox, Sandbhya Dwarkadas, Pete Keleher, Honghui Lu, ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing networks of workstations. *IEEE Computer*, February 1996.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[3] John B. Cater. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 1995.

[4] Anne Dinning. A survey of synchronization methods for parallel computers ieee computer. *IEEE Computer*, July 1989.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[6] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compiler time data transformations. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[7] Shun-Tak Leung and John Zahorjan. Optimizing data locality by array restructuring. Technical Report 95–09–01, Dept. of Computer Science and Engineering, Univ. of Washington, September 1995.

[8] Wen-Yew Liang, Chung-Ta King, and Fepei Lai. Adsmith: An efficient object-based dsm environment on pvm. In *Proc. of the 1996 International Symposium on Parallel Architecture, Algorithms and Networks*, pages 173–179, June 1996.

[9] Wen-Yew Liang, Chung-Ta King, and Fepei Lai. Adsmith user interface. Technical report, Architecture Laboratory, Dept. of Computer Science and Information Engineering, National Taiwan Univ., November 1996. http://archi1.ee.ntu.edu.tw/~wyliang /adsmith.

[10] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[11] V.S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[12] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6), 1994.

[13] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Annual International Conference on Computer ARchitecture*, pages 24–36, 1995.