Adsmith User Interface

William W. Y. Liang
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan
E-mail: wyliang@orchid.ee.ntu.edu.tw

Chung-Ta King
Department of Computer Science
National Tsing Hua University, HsinChu, Taiwan
E-mail: king@cs.nthu.edu.tw

Feipei Lai
Department of Electrical Engineering and
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan
E-mail: king@cs.nthu.edu.tw

1 Introduction

Adsmith [1] is an *object-based* DSM [2], which is built completely on top of PVM [3]. In Adsmith, the memory is viewed as consisting of many objects, which can be shared among multiple process on different processors. Adsmith provides primitives to create and allocate shared objects, accesses to shared objects, and operations to synchronize among processes.

Adsmith presents to the users as a user level library, which is implemented in C++. 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. This document describes the primary functions provided in Adsmith and its programming conventions. Advanced primitives for performance optimization are also introduced. optimization.

One example of Adsmith is shown in Appendix A. The complete Adsmith interface is listed in Appendix B. More information on Adsmith can be retrieved from Adsmith homepage:

http://archiwww.ee.ntu.edu.tw/~wyliang/adsmith.

2 Prologue

Before getting started, a few things should be noted. These include the process and host identification, the memory model, and the access style.

2.1 Process and Host Identification

Adsmith uses PVM to manage the processes. Traditionally, PVM users use the *task id* to identify a process. Under the DSM model, it is generally not necessary to directly communicate between processes. But in some cases, it may be useful to identify the hosts or the processes, for example, to allocate a shared object to a specified host. The host and process identification in Adsmith is simple. Each host is assigned a unique integer from 0 to the number of hosts minus one. Similarly, each host is assigned a unique integer from 0 to the number of processes minus one.

2.2 Memory Model

Release consistency (RC) [5] is implemented in Adsmith. Shared accesses in RC are classified as *competing accesses* (special accesses) and *non-competing accesses* (ordinary accesses). Competing accesses mean that two or more accesses may refer to the same shared memory location at the same time and at least one is a write access. Special accesses are further categorized as *synchronization accesses* and *non-synchronization accesses*. Non-synchronization accesses are competing accesses which are not used for synchronization purposes. Synchronization accesses are further divided into *acquire accesses* and *release accesses*. Adsmith provides all these access operations. (See Section 6.) Programmers are responsible for writing properly-labeled programs [5] by utilizing these operations.

2.3 Access Style

Adsmith adopts a *load/store-style memory accesses*. When an object is allocated, an address is returned. The address points to a buffer in local memory. The time between executing an acquire and its immedidate following release in one process is called an *interval*. Before an object is first read during an interval, a load operation must be called to get its up-to-date data from the DSM system to the local buffer. At the end of the interval, if the object was modified, a store operation must be called to write its data back to the DSM system. Other accesses to the object during the interval can be performed locally through the buffer. In this way, most references in an interval are local. Section 5 will give more the details of accessing a shared object.

3 System Initialization

With Adsmith, programmers need not do any initialization or termination explicitly. All these works are done automatically via the object initialization facilities provided by C++. In C++, before a program begins, the constructors of all global objects will be invoked.

Similarly, the destrctors are invoked when the program ends. Teh Adsmith library also contains a system object, which is responsible for initialization and termination of the whole system.

4 Process Creation

Two forms of process creation are provided:

> int **adsm_spawn**(char **file*, int *count*);
>
> int **adsm_spawn**(char **file*, char ***argv* = 0, int *flags* = 0, char **where* = 0, int *count*, int **tids* = 0);

In the above declaration, *file* is the program name and *count* is the number of processes to create. The second form has the same set parameters as that of `pvm_spawn()` [4]. Here is an example which creates 10 processes from the program "sample".

```
adsm_spawn("sample",10);
```

Although PVM has provided the function `pvm_spawn()`, we require that child processes in Adsmith be created through `adsm_spawn()`. In this way, proper system information will be collected during the process creation time.

Adsmith maintains a tree-like parent-child relationship among processes. The parent process will not terminate until all its children have terminated. There are functions in Adsmith which allow a parent process to explicitly wait for its children to terminate. There are three forms of such wait functions:

> int **adsm_wait**( );
>
> int **adsm_wait**(int *tid*);
>
> int **adsm_wait**(int **tids*, int *num*);

The first function will cause the caller, i.e. the parent, to wait for all its children to terminate. The second specifies a particular child to wait. The *tid* is an integer PVM task id returned by the second form of `adsm_spawn()`. The last form specifies a list of processes to wait.

5 Shared Object Allocation and Basic Manipulation

In Adsmith shared objects must be allocated before they are used. The allocation functions are used similarly as the `malloc()` function in C language. Three forms of the allocation function are supported:

> void ***adsm_malloc**(char **identifier*, int *size*, int *hint* = *AdsmDataDefault*);
>
> void ***adsm_malloc**(char **identifier*, int *size*, void **init*, int *hint* = *AdsmDataDefault*);

void *$\ast$***adsm_malloc**(char *$\ast$*identifier*, int *size*, void *$\ast$*initdata*, short *home*, short *hint* = *AdsmDataDefault*);

In the declaration, *size* is the size in bytes of the shared object and *identifier* is the string name to refer to the shared object. The parameter, *init*, in the last two forms is used to set the initial value for that shared object. Specifying *init* as NULL makes the shared object uninitialized. In the third form, the parameter *home* allows the user to specify the host where the shared object is to be allocated. However this parameter is only a hint. The object may be allocated on a host other than that specified, depending on the actual situation. The range of *home* is from 0 to the number of hosts minus one.

Several options can be set through the *hint* parameter to affect the access behavior of a shared object. Currently, the value of *hint* may be *AdsmDataCache, AdsmDataLocal*, *AdsmDataUpdate*, and *AdsmDataMWriter*. AdsmDataCache means that the shared data will be cached in application processes and managed through the coherent protocol. AdsmDataLocal means that the shared object will be allocated on the local host. This is the same as specifying the host id through the third form. When most accesses of the object are performed by the local processes, AdsmDataLocal can be used to increase the locality. AdsmDataUpdate means that write-update will be used as the coherence protocol for the declared object. By default, write-invalidate will be used.

The last value is AdsmDataMWriter. It tells the system that the *Multiple Writer Protocol* [6] will be applied on the declared shared object. This protocol allows several processes to write to different parts of the shared object at the same time. In this way the *false-sharing* problem can be eliminated and the critical section manipulation can be avoided, which is normally used to serialize accesses to the same object by multiple processes. The only restriction is that the user must ensure that at any time, no one part of the shared object is written by more than one process simultaneously.

All these values can be set together by the *or* operation in C language. Similar to the *home* parameter, *hint* serves only as a reference. The actual value will be set by the first declaration of an object, and can not be changed.

After a shared object has been allocated, an address will be returned, which points to the buffer space of the shared object. There are some differences between the concepts of "buffer" and "cache" in Adsmith. Every shared object has a buffer space in the process which has allocated the object. Data in a buffer will be up-to-date only when the acquire access is performed, according to the definition of Release Consistency. Without caching *every* load operation needs to be forwarded to the home node to fetch the data. Caching will avoid unnecessary communication if the up-to-date data exist in the buffer. Of course, caching may induce extra coherence messages.

Adsmith uses *load/store-style memory accesses.* As described in Section 2.3, a load operation should be performed before the first read reference in an interval, and a store operation should be performed after the last write reference. Other accesses for the shared object in the interval can be performed locally through the buffer. The above load and store operations are specified with the `adsm_refresh`() and `adsm_flush`() functions respectively. Detailed usage of these two functions will be described later in Section 6.1.

When a shared object is no longer referenced, it can be freed by the following function:

adsm_free(void **ptr*);

This function will discard the local buffer space. For better memory space utilization, the programmers are encouraged to free shared objects which will not be used in the near future. Freed objects can be reused by reallocating them again. Here is a simple example for the allocation and basic access functions.

```
float *pi=(float*)adsm_malloc("pi",sizeof(float));
...other accesses...
... lock ...
adsm_refresh(pi); // load(read) the content of pi
  *pi=*pi+100;    // add pi by 100
adsm_flush(pi);   // store(write) the content of pi
... unlock ...
adsm_free(pi);    // free pi
```

6 Accesses of Shared Objects

In Release Consistency model, memory operations are classified as ordinary accesses, synchronization accesses, and non-synchronization accesses. This section describes how these accesses are supported in Adsmith.

6.1 Ordinary Accesses

The following two functions are for ordinary accesses:

void **adsm_refresh**(void **ptr*);

void **adsm_flush**(void **ptr*);

Since no hardware or operating system related facilities are used in Adsmith to support shared memory accesses, data should be manually refreshed (loaded) from the shared memory by the programmer before they are accessed. Similarly, they should be flushed (stored) back to the memory after they are modified. Under RC, `adsm_refresh()` is for ordinary loads, and `adsm_flush()` is for ordinary stores. The value refreshed is guaranteed to be as recent as that at the time of the last acquire. (See Section 6.2.) Under the load/store-style memory accesses, the refresh operation is invoked only once before the first load access after an acquire, and the flush operation is invoked only once after the last store access before a release. (See Section 2.3.)

6.2 Synchronization Accesses

Ordinary accesses require that the programmer use enough synchronization accesses to ensure program correctness. Adsmith provides three classes of synchronization operations: semaphore, mutex and barrier. They are listed below as class prototype with their main public methods.

```
class AdsmSemaphore {
public:
    AdsmSemaphore( char *identifier, int init = 1 );
    void wait();
    void signal();
    void set( int value );
    int get();
};

class AdsmMutex {
public:
    AdsmMutex( char *identifier );
    void lock();
    void unlock();
};

class AdsmBarrier {
public:
    AdsmBarrier( char *identifier );
    void barrier( int count );
};
```

Since all three class instances are shared between processes, the identifier should also be specified.

Semaphore is the most basic synchronization operation. It is implemented as a counting semaphore in Adsmith. Mutex is designed specifically for mutual exclusion, but there is a restriction on its usage: mutex lock and unlock operations must appear in pairs. Barrier supports barrier synchronization, and the number of processes participating in the synchronization must be specified.

Among the synchronization functions, semaphore wait, mutex lock and barrier are acquire accesses, and semaphore signal, mutex unlock and barrier are release accesses. An acquire is needed in order to gain the access right to a set of data, and a release is used to grant the access right. The RC model guarantees that ordinary accesses after an acquire will obtain the data available at the time of the acquire. Examples of synchronization accesses can be found in the example program given later.

In addition to the above three synchronization functions, process creation and termination are also one kind of synchronization accesses. The release accesses will be performed before child processes are spawned and before the process terminates. And the acquire accesses will be performed at the process initialization time.

6.3 Non-synchronization Accesses

Synchronization accesses are competing accesses. However not all competing accesses are for synchronization purposes. This kind of accesses is called *non-synchronization accesses.* Adsmith has two basic non-synchronization functions:

void **adsm_refresh_now**(void **ptr*);

void **adsm_flush_now**(void **ptr*);

Although most ordinary accesses may be performed locally, non-synchronization accesses are always sent to home nodes directly. The tail "_now" in the function names indicates that these two accesses will be performed without waiting for previous ordinary accesses. That is, a write through `adsm_flush_now`() will be seen immediately by all the following loads through `adsm_refresh_now`(), even when they are invoked by other processes. Adsmith currently support RC_{sc} [5], competing accesses including both synchronization and non-synchronization accesses are sequentially consistent.

7 Advanced Functions

Portable software DSM systems usually have the efficiency problem. One primary source of inefficiency is excessive messages. Two methods are generally used to solve this problem. The first is to overlap the communications, and the second is to reduce the number of messages. Adsmith implements both strategies and provides several advanced functions to improve the performance. These include prefetch, bulk transfer, atomic accesses, and collect accesses.

7.1 Prefetch

Adsmith supports nonblocking load, i.e., data prefetching, through the following function.

void **adsm_prefresh**(void **ptr*);

The function corresponds to the load function `adsm_refresh()`. The programmer can insert the prefetch function before the load function as far as possible. The sequence of shared data accesses in Adsmith can be depicted as follows.

Acquire → Prefresh → Refresh → Local Accesses → Flush → Release

where Refresh and Flush are ordinary accesses discussed above. Nonblocking load and store will be performed between prefresh-refresh and flush-release pairs respectively. The code segment below is a typical example to perform computations on a shared object within a critical section:

```
typeA *A=(typeA*)adsm_malloc("name of A",sizeof(typeA));
AdsmMutex mutex("mutex name"); // Mutex is a synchronization class
mutex.lock();
adsm_prefresh(A);
... prologue of computation ...
adsm_refresh(A);
... computations with local access on A ...
adsm_flush(A);
... epilogue of computation ...
mutex.unlock();
```

Note that `adsm_refresh`() is still required for the load access to ensure that the requested data has arrived. Prefetch can be used most effectively in compilers, because compilers know the best places to insert these accesses.

7.2 Bulk Transfer

Consecutive accesses can be aggregated to reduce the number of messages sent. Adsmith provides a *bulk-transfer* version of several functions for this purpose. In current implementation, these functions include allocation functions, prefetch accesses, refresh accesses, and flush accesses. The mechanism is to add the following functions in pairs to the beginning and the end of the corresponding group of functions or accesses.

void **adsm_malloc**(AdsmBulkType **type*);

void **adsm_prefresh**(AdsmBulkType **type*);

void **adsm_refresh**(AdsmBulkType **type*);

void **adsm_flush**(AdsmBulkType **type*);

AdsmBulkType is defined as follows:

enum **AdsmBulkType** {
  AdsmBulkBegin,
  AdsmBulkEnd
}

For example, to aggregate the refresh to a group of shared objects, one may use the following code:

```
// assume int *group[N] points to N integer shared objects
adsm_refresh(AdsmBulkBegin);
for (int i=0; i<N; i++) adsm_refresh(group[i]);
adsm_refresh(AdsmBulkEnd);
```

By default, bulk transfer is always performed for `adsm_flush`() on release time. Thus it is not necessary to specify bulk transfer for `adsm_flush`() in most cases. It should be noted that other operations to the objects specified between AdsmBulkBegin and AdsmBulkEnd are not allowed, since the bulk transfer will not be actually performed until the AdsmBulkEnd time. To reduce the function calls when accessing a group of related shared objects with bulk transfer, Adsmith provides the form `adsm_*_array`() for the above `adsm_*`() functions. Please refer to Appendix B for the interface.

7.3 Atomic Accesses

Suppose we want to control the access to a shared object through a critical section. As described in Section 6.1, the conventional sequence for an ordinary access may look like this: acquire → refresh → local accesses → flush → release. Assume that the synchronization arbitrator for the critical section and the host of the shared object are located in different machines. Then accessing the shared object may require seven messages (two for acquire, two for refresh, two for flush, and one for release), if the shared object was invalidated by other processes before the acquire and was modified during local accesses. It is thus necessary to reduce the amount of messages for this kind of accesses. It should be noted that such a situation will also happen on most DSMs, which support a similar consistency model.

The problem can be solved by allocating the synchronization arbitrator to the home node of the shared object and combining these two operations. During an acquire, the requested data can be piggy-backed on the lock grant message. After the computation, the modified data can also be sent back with the release message. In this way, the required number of messages will be reduced to only four (two for acquire and refresh, and two for flush and release).

Adsmith provides *atomic accesses* to support this kind of accesses. Two functions are supported:

void **adsm_atomic_begin**(void **ptr*, int *type* = AdsmAtomicWrite);

void **adsm_atomic_end**(void **ptr*);

The function `adsm_atomic_begin()` can be viewed as a combination of acquire and refresh, while the function `adsm_atomic_end()` as a combination of flush and release. Note that *currently atomic accesses are categorized as non-synchronization accesses*[1]. `adsm_atomic_begin()` only performs pure lock and refresh operations on the target object. Similarly, `adsm_atomic_end()` only performs flush and pure unlock operations on the target object. No interactions will happen between atomic accesses and ordinary accesses. They are also sequentially consistent with any other competing accesses.

Let the program segment between `adsm_atomic_begin()` and `adsm_atomic_end()` be called *atomic section*. Two types of atomic operations can be specified in the *type* parameter in `adsm_atomic_begin()`: AdsmAtomicWrite and AdsmAtomicRead. The former is to indicate that both read and write accesses are included in the atomic section; and the latter is to indicate that only read accesses exist in the section. Adsmith implements a *single-writer/multiple-readers* protocol. For a writer, `adsm_atomic_begin()` can be performed only when there is no reader nor writer in the atomic section. For a reader, `adsm_atomic_begin()` can be performed only when there is no writer in the atomic section. For fairness purpose, Adsmith implements the *writer first* protocol. That is, when a writer is waiting to enter the atomic section, readers which come after the writer will be blocked until the writer has finished its atomic section. Of course, readers before the writer can proceed until they all exit their atomic sections.

Here is an example of atomic accesses modified from the example in Section 6.1.

```
typeA *A=(typeA*)adsm_malloc("name of A",sizeof(typeA));
adsm_atomic_begin(A);
... computation with local access on A ...
adsm_atomic_end(A);
```

There is another form of atomic access which may be useful.

void **adsm_atomic**(void **ptr*, char **expr*);

[1]Although atomic accesses will be categorized as synchronization accesses in the future, programs written in the current version will run correctly in the later versions

where *expr* is of the form

[*type*] *expression*

The parameter *type* describes the type of the target object pointed by *ptr*, which currently include the C basic types only. The parameter *expression* is a C expression. The only variable in *expression* is the target object, which is represented by the symbol "@". The shared object will be calculated according to the expression specified by *expression* atomically. This kind of access is also called *active access* in Adsmith, since the expression is executed at its home. Here is an example of the function.

```
int *A=(int*)adsm_malloc("A",sizeof(int));
adsm_atomic(A,"[int] @=@+100");
```

Although this function currently does not support complex data types, it can be very useful to reduce the number of messages. In fact, each `adsm_atomic()` function call involves only two messages.

7.4 Collect Access

Sometimes one may use a shared object as an accumulator. For example, the following code sequence calculates the total sum by cumulating partial sums.

```
AdsmBarrier bsum("bsum");
int *sum=(int*)adsm_malloc("sum",sizeof(int));
int partial_sum=....;  // calculate the partial sum
char expr[20];
sprintf(expr,"[int] @+=%d",partial_sum); // prepare the expression
adsm_atomic(sum,expr); // add partial sum
bsum.barrier(nproc);   // barrier on nproc processes
adsm_refresh(sum);     // retrieve the result
```

The code from `adsm_atomic()` to `adsm_refresh()` generates $nproc{*}6$ messages. Adsmith provides a function called *collect accesses* to optimize for this kind of cases. The functions are listed as follows:

void **adsm_collect_begin**(void **ptr*, int *num*);

void **adsm_collect_end**(void **ptr*);

num is the number of processes participating in the access. Both `adsm_collect_begin()` and `adsm_collect_end()` must appear in pairs. The usage is similar to that of `adsm_atomic_begin()` and `adsm_atomic_end()`. It effects as if the following functions were called in order: `adsm_atomic_begin()`, `adsm_atomic_end()`, barrier(), and `adsm_refresh()`. But the number of messages generated is much reduced. The above example can now be re-written as follows.

```
int partial_sum=....;  // calculate the partial sum
adsm_collect_begin(sum,nproc);
sum+=partial_sum;      // add partial sum
adsm_collect_end(sum); // total sum is returned
```

The number of messages involved now is reduced to $nproc * 4$.

8 Other Functions

This section describes other functions provided in Adsmith.

8.1 Pointer

Pointers to shared memory are supported in Adsmith, but with some restrictions. This is because the address of a shared object in one process may not be the same as that in the other process. Thus, the programmers are required to translate the local address of a shared object to a globally recognizable address before that address is passed to other processes through the shared memory. Functions for pointer manipulations are as follows:

> int **adsm_gid**(void **ptr*);
>
> void ***adsm_attach**(int *gid*);

The function `adsm_gid()` translates the local address of a shared object into its global address, which is represented by an integer. The function `adsm_attach()`, on the other hand, translates a global address back to the local address for the requesting process. Its function is like `adsm_malloc()` if the target object of the pointer is not present.

For example, if the programmer wants to pass the address of shared object T from process P1 to process P2 through the shared pointer object S, the following two sections of code can be used for the two processes.

```
// process P1 set the pointer
AdsmBarrier Bpointer("barrier for this code");
sometype *T=(sometype*)adsm_malloc("target data",sizeof(sometype));
int *S=(int*)adsm_malloc("point to T",sizeof(int));

*S=adsm_gid(T); // get global address of T
adsm_flush_now(S); // flush immediately
Bpointer.barrier(2); // done

// process P2 get the pointer
AdsmBarrier Bpointer("barrier for this code");
int *S=(int*)adsm_malloc("point to T",sizeof(int));

Bpointer.barrier(2); // wait P1 done
adsm_refresh_now(S); // get the pointer value
sometype *T=(sometype*)adsm_attach(S); // attach the pointer
```

8.2 Supporting Shared Memory and Message Passing

Adsmith allows programmers to use message passing primitives in PVM together with its shared memory primitives. However, cares must be taken when using PVM message passing primitives. As mentioned in Section 4, processes must be spawned with `adsm_spawn()` instead of `pvm_spawn()`. Another thing to note is the range of message tags that are allowed when passing messages. In PVM message tags are integers and used in send and receive primitives to distinguish message channels. Adsmith currently occupies seven channels, which are from MAXINT down to MAXINT−6, where MAXINT is the largest integer defined in the system.

8.3 Information Retrieval

Section 2.1 discussed how to identify hosts and processes. Adsmith supports the following functions to retrieve the identifier and translate it to the corresponding PVM task id.

> int **adsm_hostno**(int *procno* = -1);
>
> int **adsm_procno**( );
>
> int **adsm_procno2tid**(int *procno*);
>
> int **adsm_tid2procno**(int *tid*);

The function `adsm_hostno()` returns the host id where the process specified by the process number *procno* resides. If *procno* is ignored, the host id of the calling process will be returned. The host id is in the range of zero to the number of hosts minus one. The function `adsm_procno()` returns the process id of the calling process. It is in the range of zero to the number of processes minus one. The function `adsm_procno2tid()` translates the process id to the corresponding PVM task id, while `adsm_tid2procno()` translates the PVM task id back to the corresponding process id in Adsmith.

9 Summary

Writing parallel programs is easier in Adsmith than in PVM. There are only two things that programmers of Adsmith have to note. First, synchronization operations must be inserted properly to ensure the correctness of ordinary accesses (see Section 6.2). This effort is needed for any shared-memory parallel programming. Second, the load (refresh) operation must be used before a shared object is referenced and the store (flush) operation is used after the object is modified (see Section 6.1).

Since programmer's knowledge about the programs are helpful for optimizing performance, Adsmith exploits this knowledge by providing users and/or compilers with several useful options. These include property settings, data distribution, multiple writer protocol (Section 5), prefetch, bulk transfer, atomic access, and collect access (Section 7). Our initial experimental results show that programs written in Adsmith have comparable performance with those written in PVM. Unfortunately, for performance reason, Adsmith currently only supports homogeneous environments to avoid data transformation between different architectures (e.g., XDR used in PVM).

Adsmith is an on-going project. Several new features are currently under investigation and will be implemented into the system in the near future. Performance of Adsmith is also being evaluated more extensively. New algorithms or techniques will be used to eliminate possible performance bottlenecks and to make the system more robust. With further refinements, we expect Adsmith to be a really useful environment for shared-memory programming on networks of workstations.

References

[1] Wen-Yew Liang, Chun-Ta King, Fepei Lai, "Adsmith: An Efficient Object-Based DSM Environment on PVM," In the proceedings of the 1996 International Symposium on Parallel Architecture, Algorithms and Networks, Beijing, China, pp. 173-179, June 1996.

[2] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, Vol. 24, No. 8, pp. 52–60, Aug 1991.

[3] V.S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience,* Vol. 2, No. 4, Dec. 1990.

[4] A. Geist, et al., "PVM 3.0 User's Guide and Reference Manual," Oak Ridge National Laboratory, 1993.

[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,* pp. 15–26, May 1990.

[6] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," In *Proceedings of the 1994 Winter Usenix Conference*, pp. 115-113, Jan 1994.

A An Example of Adsmith

This section lists an example program of Adsmith. More examples can be found in the package.

```
/////////////////////////////////////////////////////////////
//
// Sample program for Adsmith
// Purpose: Calculate the summation of values in a vector
// Paradigm: SPMD
//
// Algorithm:
//   Shared Objects:
//     the vector and the summation
//   Procedure:
//     0. Declare initialize the shared objects.
//     1. Spawn a process on each host.
//     2. Every process calculates a partial sum.
//     3. Cumulate the partial sum into the global sum
//
// Usage: sample [size]
//
// Author: William W. Y. Liang
//
/////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
#include "adsm.h" // standard libray of Adsmith
#include "adsmutil.h" // non-standard utilities of Adsmith
#include "adsmtime.h" // a timing class

void slave();

// system parameters
int seqno;
int nhost;
AdsmBarrier Bsample("sample"); // for barrier synchronization

// initial values
int zero=0;
int deflen=100000; // default length

// shared variable
int *len; // length of the vector
int *vec; // source of the vector
int *sum; // result of the sum

main(int argc,char *argv[]) {
  // get my sequence no. and host number
  seqno=get_seqno(); // get_seqno() and get_nhost()
  nhost=get_nhost(); // both are in adsmutil.h

  // initialize content of the vector and spawn child processes
  if (seqno==0) { // the 1st process
```

```
    // determine the size of vector
    if (argc>1) deflen=atoi(argv[1]);

    cout<<nhost<<" host(s) detected"<<endl;
    cout<<"Length of vector = "<<deflen<<endl;

    // initialize the vector
    int initvec[deflen];
    for (int i=0; i<deflen; i++) initvec[i]=i;

    // allocate shared objects
    adsm_malloc(AdsmBulkBegin); // allocate in aggregate mode
    len=(int*)adsm_malloc("len",sizeof(int),&deflen);
    sum=(int*)adsm_malloc("sum",sizeof(int),&zero);
    vec=(int*)adsm_malloc("vec",(*len)*sizeof(int),initvec);
    adsm_malloc(AdsmBulkEnd);

    // spawn processes on each host
    adsm_spawn(execname(argv[0]),nhost); // extract program name

    // timing
    Timing Tsample("sample");
    Bsample.barrier(nhost+1);
    Tsample.start(); // timing start
    Bsample.barrier(nhost+1);
    Tsample.stop();   // timing end

    // obtain the final result
    adsm_refresh(sum);
    cout<<"Sum = "<<(*sum)<<endl;
  } else { // the workers
    // allocate shared objects
    adsm_malloc(AdsmBulkBegin); // allocate in aggregate mode
    len=(int*)adsm_malloc("len",sizeof(int));
    sum=(int*)adsm_malloc("sum",sizeof(int));
    adsm_malloc(AdsmBulkEnd);

    // read the parameters
    adsm_refresh(len);

    // allocate shared vector and read the value
    vec=(int*)adsm_malloc("vec",(*len)*sizeof(int));
    adsm_refresh(vec);

    slave();
  }

  adsm_free(len);
  adsm_free(sum);
  adsm_free(vec);
}

void slave() {
  // calculate the range to be computed by this process
  int size=((*len)+nhost-1)/nhost;
  int begin=size*(seqno-1);
  int end=begin+size;
  if (end>(*len)) end=(*len);

  // ready to compute
  Bsample.barrier(nhost+1);
```

```
  // calculate partial sum
  int partial=0;
  for (int i=begin; i<end; i++) partial+=vec[i];

  // add partial sum to the total sum

  // Using Mutex

  // AdsmMutex mutex("mutex");
  // mutex.lock(); // lock the mutex
  // adsm_refresh(sum); // load the content of sum
  //   *sum+=partial; // add partial sum to total sum
  // adsm_flush(sum); // store the content of sum
  // mutex.unlock(); // unlock the mutex

  // Using Atomic Begin-End

  // atomic access is faster than the above mutex/refresh method
  // adsm_atomic_begin(sum); // begin of an atomic access on 'sum'
  //   *sum+=partial;        // add partial sum to total sum
  // adsm_atomic_end(sum);   // end of the atomic access

  // Using Atomic Expression

  char expr[20];
  sprintf(expr,"[int] @+=%d",partial);
  adsm_atomic(sum,expr);

  // wait for all processes done
  Bsample.barrier(nhost+1);
}
```

B The Manual Pages

This section contains an alphabetical listing of all the Adsmith routines.

Name

`AdsmAtomicType` - atomic type (Advance)

Synopsis

enum `AdsmAtomicType` {
AdsmAtomicWrite,
AdsmAtomicRead
};

DESCRIPTION

Atomic types is used in `adsm_atomic_begin`() below. The atomic section is defined as the code region between `adsm_atomic_begin`() and `adsm_atomic_end`(). *AdsmAtomicWrite* means that there is write accesses in the atomic section. *AdsmAtomicRead* means that there is only read accesses in the atomic section. Atomic section follows the single writer / multiple reader protocol. That is, at any time there will be only one writer in the atomic section. Multiple readers are allowed to be in the atomic section at the same time.

See Also

`adsm_atomic`, `adsm_atomic_begin`, `adsm_atomic_end`

Name

`AdsmBarrier` - the class of shared objects used as barrier (Primary Function)

Synopsis

```
class AdsmBarrier {
public:
    AdsmBarrier( char *id );
    ~AdsmBarrier();
    int barrier( int count );
private:
    int *tag;
};
```

DESCRIPTION

Barrier is for barrier synchronizations. As `AdsmSemaphore`, it also need an identifier *id*. `barrier()` is the only method. The number of processes participating in the barrier operation should be specified as the *count* parameter, and it must be the same for all the participating processes. `barrier()` will block the caller until all participating processes have arrived the barrier point. It implies both the acquire and release accesses in the RC model.

See Also

`AdsmSemaphore`, `AdsmMutex`

Name

AdsmBulk - Bulk transfer delimiter function for `adsm_malloc`(), `adsm_refresh`(), `adsm_flush`(), and `adsm_prefresh`() (Advance Functions)

Synopsis

void `adsm_malloc`(AdsmBulkType *type*, int *num* = 0);

void `adsm_refresh`(AdsmBulkType *type*);

void `adsm_flush`(AdsmBulkType *type*);

void `adsm_prefresh`(AdsmBulkType *type*);

DESCRIPTION

Each of these functions must appear in pairs of *type AdsmBulkBegin* and *AdsmBulkEnd*. *num* of `adsm_malloc`() is the approximate number of objects included in the bulk allocation. *num* is only used on *AdsmBulkBegin* and can be ignored. In fact, `adsm_flush`() is implicitly performed on release time. Thus, it is rarely used.

See Also

`AdsmBulkType`, `adsm_malloc`, `adsm_malloc_array`, `adsm_refresh`, `adsm_refresh_array`, `adsm_flush`, `adsm_flush_array`, `adsm_prefresh`, `adsm_prefresh_array`

Name

AdsmBulkType - bulk transfer type (Advance Function)

Synopsis

enum `AdsmBulkType` {
    AdsmBulkBegin,
    AdsmBulkEnd
};

DESCRIPTION

This is used in the following bulk transfer delimiter functions.

See Also

`AdsmBulk`

Name

AdsmHint - the properties of the shared objects (Primary)

Synopsis

AdsmDataLocal, *AdsmDataCache*, *AdsmDataUpdate*,
AdsmDataMWriter, *AdsmDataMovable*, *AdsmDataDefault*

DESCRIPTION

These are used as the *hint* parameters of the allocation functions below. *AdsmDataLocal* will allocate the object at the caller's host. *AdsmDataCache* lets the object utilize caching mechanisms; by default, caching is not used. If *AdsmDataCache* is set, *AdsmDataUpdate* will use write-update as its coherence protocol; otherwise write-invalidate is used. Specifying *AdsmDataMWriter* will enable multiple writer protocol to be applied on accessing to the object. *AdsmDataMovable* will make the home to be movable, But it is not implemented yet. These properties can be OR'ed.

See Also

`adsm_malloc`, `adsm_malloc_array`

Name

AdsmMutex - the class of shared objects used as mutex (Primary Function)

Synopsis

```
class AdsmMutex {
public:
    AdsmMutex( char *id );
    ~AdsmMutex();
    void lock();
    void unlock();
private:
    int *tag;
};
```

DESCRIPTION

Mutex is for critical sections. As `AdsmSemaphore`, it also need an identifier *id*. `lock()` will block the caller if the mutex has been locked by other process. `unlock()` will unlock the mutex. `lock()` and `unlock()` are acquire and release accesses respectively in the RC model.

See Also

`AdsmSemaphore`, `AdsmBarrier`

Name

AdsmSemaphore - the class of shared objects used as semaphores (Primary Function)

Synopsis

```
class AdsmSemaphore {
public:
    AdsmSemaphore( char *id, int val = 1 );
    ~AdsmSemaphore();
    int get();
    void set( int val );
    void wait();
    void signal();
private:
    int *value;
};
```

DESCRIPTION

A semaphore object is a integer counting semaphore, which is also a shared object. It requires an identifier *id*. The identifier can be the same as that appears in `adsm_malloc()`'s, since they are distinguished from the normal shared object names internally. An initial value greater than zero can be set by the *val* parameter. The default value is one. Operations and accesses to semaphores must be performed through the methods provides by the class. `get()` and `set()` are used to retrieve and set the semaphore value respectively. `wait()` will block the caller if the semaphore value is less than one, otherwise it will decrease the semaphore value by one. `signal()` will wake up a process if there are waiting processes, otherwise it will increase the semaphore value by one. `wait()` and `signal()` are acquire and release accesses respectively in the RC model.

See Also

`AdsmMutex`, `AdsmBarrier`

Name

adsm_atomic - atomic expression delimiter function (Advance Functions)

Synopsis

void `adsm_atomic`(void **data*, char **expr*);

DESCRIPTION

Perform the expression *expr* on the object atomically. *expr* is of the format: " [<*type*>] <*expression*> ", where <*type*¿ includes most of C's base types, namely `char`, `short`, `int`, `long`, `uchar`, `ushort`, `uint`, `ulong`, `float`, `double`; <*expression*> includes most of the C expression format. The only variable in <*expression*> is '@', which represents the shared objects. The shared object will be evaluated in type of <*type*> in the expression. Atomic accesses are assumed to be the Non-Synchronization accesses in the RC model.

See Also

`adsm_atomic_begin`, `adsm_atomic_end`

Name

adsm_atomic_begin, **adsm_atomic_end** - atomic section delimiter function (Advance Functions)

Synopsis

void `adsm_atomic_begin`(void **data*, int *rw* = *AdsmAtomicWrite*);

void `adsm_atomic_end`(void **data*);

DESCRIPTION

`adsm_atomic_begin`() indicates the beginning of an atomic section. It acts like a lock plus a refresh operation. `adsm_atomic_end`() indicates the end of an atomic section. It acts like a flush plus a unlock operation. But these two function do not perform the release and acquire operations in the RC model. Atomic accesses are assumed to be the Non-Synchronization accesses in the RC model.

See Also

`adsm_atomic`, `adsm_collect_begin`, `adsm_collect_end`

Name

adsm_attach - attach a global address retrieval into local process (Primary Function)

Synopsis

void* `adsm_attach`(int *gaddr*);

DESCRIPTION

Attach a global address to the local process. It acts like the `adsm_malloc`() function.

Returned Value:

The buffer address of the corresponding global address is returned.

See Also

`adsm_gid`, `adsm_malloc`

Name

adsm_collect_begin, **adsm_collect_end** - collect access delimiter function (Advance Functions)

Synopsis

void `adsm_collect_begin`(void **data*, int *num*);

void `adsm_collect_end`(void **data*);

DESCRIPTION

Collect accesses can be used in cumulation operations. They are similar the atomic section functions. The different is that barrier operation of *num* processes will be performed on `adsm_collect_end`(). `adsm_collect_begin`() acts like a lock plus a refresh operations. `adsm_collect_end`() acts like a flush plus a unlock and then a barrier operations. Collect accesses are assumed to be the Non-Synchronization accesses in the RC model.

See Also

`adsm_atomic_begin`, `adsm_atomic_end`

Name

adsm_enroll - force to enroll into Adsmith

Synopsis

void `adsm_enroll`();

DESCRIPTION

If none of the library functions are used in a program, `adsm_enroll`() must be called to explicitly enroll into the Adsmith system such that it can interact with other programs in the application correctly.

See Also

`AdsmBulk`, `adsm_flush_array`, `adsm_refresh`, `adsm_flush_now`

Name

adsm_flush - ordinary store access, buffer flush function (Primary Function)

Synopsis

void `adsm_flush`(void **data*);

DESCRIPTION

Flush the buffer of the shared object pointed by *data* to the shared memory. This will be internally delayed to the next release time.

See Also

`AdsmBulk`, `adsm_flush_array`, `adsm_refresh`, `adsm_flush_now`

Name

adsm_flush_array - aggregated function of `adsm_flush`()'s (Advance Function)

Synopsis

void `adsm_flush_array`(void **data*, int *len*);

DESCRIPTION

Aggregately flush an array of *len* shared objects pointed by *data* to the shared memory. Although *data* is of type void*, it will be interpreted as void** internally.

See Also

`AdsmBulk`, `adsm_flush`

Name

adsm_flush_now - non-synchronization store access, buffer flush function (Primary Function)

Synopsis

void `adsm_flush_now`(void **data*);

DESCRIPTION

Flush the buffer of the shared object pointed by *data* to the shared memory immediately.

See Also

`adsm_flush`

Name

adsm_free - free the shared object locally (Primary Functions)

Synopsis

void `adsm_free`(void **data*);

DESCRIPTION

The shared object will be freed from the calling process.

See Also

`adsm_malloc`

Name

adsm_gid - global address retrieval function (Primary Function)

Synopsis

int `adsm_gid`(void **data*);

DESCRIPTION

`adsm_gid`() translate local address *data* to its corresponding global address represented by a integer.

Return Values

The global address is returned.

See Also

`adsm_attach`

Name

adsm_hostno - get the host number (Primary Function)

Synopsis

int `adsm_hostno`(int *procno* = -1);

DESCRIPTION

This function is used to return the host number of a specified process represented by a process number *procno*. The host number can be used to manage the location where a shared object is to be allocated. It is of the range from zero to the number of hosts in the system minus one (i.e. 0 ñhost-1).

Return Values

Without the *procno* parameter, the calling process's host number will be returned. Otherwise, the specified process's host number will be returned.

See Also

`adsm_procno`

Name

adsm_malloc - allocate a share object (Primary Function)

Synopsis

void *`adsm_malloc`(char **id*, int *size*,
short *hint* = *AdsmDataDefault*);

void *`adsm_malloc`(char **id*, int *size*, void **initdata*,
short *hint* = *AdsmDataDefault*);

void *`adsm_malloc`(char **id*, int *size*, void **initdata*,
int *home*, short *hint* = *AdsmDataDefault*);

DESCRIPTION

To allocate a shared object, a string identifier *id* must be given. *size* is the object size in bytes. *initdata* points to a datum of size *size*, which will be set as the the initial value of the shared object if the object is first declared in the system. *home* represents a host on which the object will be allocated. *home* is between the range of zero and the number of hosts in the system minus one. The properties of the object can be set in *hint*.

Return Values

An address will be returned, which points to a local buffer of size *size*. It will also be used as a parameter of most of the access functions provided by Adsmith to interact with the shared memory.

See Also

`AdsmHint`, `AdsmBulk`, `adsm_malloc_array`, `adsm_free`

Name

adsm_malloc_array - aggregated function of `adsm_flush`()'s (Advance Function)

Synopsis

void `adsm_malloc_array`(char **id*, int *size*, int *num*,
void **arrptr*, short *hint* = *AdsmDataDefault*);

void `adsm_malloc_array`(char **id*, int *size*, int *num*,
void **arrptr*, int **distarr*, short *hint* = *AdsmDataDefault*);

void `adsm_malloc_array`(char **id*, int *size*, int *begin*,
int *num*, void **arrptr*, void **init* = 0, int **distarr* = 0,
short *hint* = *AdsmDataDefault*);

DESCRIPTION

These functions are short-cuts of bulk allocations. They will allocate an array of *num* relative shared objects, each of size *size*. The caller must provide an array *arrptr* which contains *num* pointer spaces to store the addresses of the allocated objects. Although *arrptr* is of type void*, it is internally interpreted as void**. *id* is not used as the identifier of each shared objects. The name of each objects is composed by appending *id* with "[i]", where i is the index starting from 0 or *begin*. The distribution of the shared objects can be determined by the array *distarr* which contains *num* host numbers (i.e. the homes) on which each corresponding objects will be allocated. *init* points to a datum of size *size*, used to initialize every shared object. All objects with have the same properties if *hint* is provided.

See Also

`AdsmHint`, `AdsmBulk`, `adsm_malloc`

Name

adsm_prefresh - prefetch access, buffer contents prefetch function (Advance Function)

Synopsis

void `adsm_prefresh`(void **data*);

DESCRIPTION

Prefetch the buffer contents of the shared object pointed by *data* from the shared memory.

See Also

`AdsmBulk`, `adsm_prefresh_array`, `adsm_refresh`

Name

adsm_prefresh_array - aggregated function of `adsm_prefresh`()'s (Advance Function)

Synopsis

void `adsm_prefresh_array`(void **data*, int *len*);

DESCRIPTION

Aggregately prefetch an array of *len* shared objects pointed by *data* from the shared memory. Although *data* is of type void*, it will be interpreted as void** internally.

See Also

`AdsmBulk`, `adsm_prefresh`

Name

adsm_procno - get the process number (Primary Function)

Synopsis

int `adsm_procno`();

DESCRIPTION

Adsmith process number is different from PVM task id. It is of range from zero to the number of processes in the system minus one (i.e. 0 ñproc-1). It is generally cooperated with the functions `adsm_hostno`(), `adsm_procno2tid`(), and `adsm_tid2procno`() to accomplish the work such as data distribution, or communicating with other processes using PVM message passing primitives.

Return Values

The calling process's process number is returned.

See Also

`adsm_hostno`, `adsm_procno2tid`, `adsm_tid2procno`

Name

adsm_procno2tid, **adsm_tid2procno** - translate between Adsmith process number and PVM task id (Primary Functions)

Synopsis

int `adsm_procno2tid`(int *procno*);

int `adsm_tid2procno`(int *tid*);

DESCRIPTION

`adsm_procno2tid`() translates the Adsmith process number *procno* to the corresponding PVM task id. `adsm_tid2procno`() translates the PVM task id *tid* to the corresponding Adsmith process number.

Return Values

As described in the description.

See Also

`adsm_procno`, `adsm_procno2tid`, `adsm_tid2procno`

Name

adsm_refresh - ordinary load access, buffer refresh function (Primary Function)

Synopsis

void `adsm_refresh`(void **data*);

DESCRIPTION

Refresh the buffer of the shared object pointed by *data* from the shared memory. The value refreshed is guaranteed to be as recent as that at the time of the last acquire.

See Also

`AdsmBulk`, `adsm_refresh_array`, `adsm_flush`, `adsm_refresh_now`

Name

adsm_refresh_array - aggregated function of `adsm_refresh`()'s (Advance Function)

Synopsis

void `adsm_refresh_array`(void **data*, int *len*);

DESCRIPTION

Aggregately refresh an array of *len* shared objects pointed by *data* from the shared memory. Although *data* is of type void*, it will be interpreted as void** internally.

See Also

`AdsmBulk`, `adsm_refresh`

Name

adsm_refresh_now - non-synchronization load access, buffer refresh function (Primary Function)

Synopsis

void `adsm_refresh_now`(void **data*);

DESCRIPTION

Refresh the buffer of the shared object pointed by *data* with the up-to-date data from the shared memory. Generally, `adsm_refresh_now`() is used to obtain the data stored by the most recent `adsm_flush_now`().

See Also

`adsm_refresh` `adsm_flush_now`

Name

adsm_shutdown - terminate the parallel application (Primary Function)

Synopsis

void `adsm_shutdown`();

DESCRIPTION

Kill all the processes in the application immediately.

Name

adsm_spawn - process spawn function (Primary Function)

Synopsis

int `adsm_spawn`(char **prog*, char ***argv* = 0, int *flags* = 0,
char **where* = 0, int *num* = 1, int **tids* = 0);

int `adsm_spawn`(char **prog*, int *num*);

DESCRIPTION

The first function has the same spec as `pvm_spawn()`. The second one is a short form. *prog* is the name of the program to be spawned, *num* is the number of processes to be spawned. Note that the returned *tids* are PVM task id. To find the corresponding Adsmith process numbers, please use the function `adsm_tid2procno()`. `adsm_spawn()` is also a kind of synchronization operations. It involves the release access in the RC model.

Return Values

The same as `pvm_spawn()`, the number of processes that are successfully spawned is returned. *tids* in the first form will contains the PVM task id's of the spawned processes.

See Also

`adsm_wait`, `adsm_tid2procno`

Name

adsm_version - get the Adsmith version (Primary Function)

Synopsis

const char *`adsm_version`();

DESCRIPTION

Get the Adsmith version number.

Return Values

The version is returned in the format of a string.

Name

adsm_wait - wait a process or a group of processes to finish (Primary Function)

Synopsis

int `adsm_wait`();

int `adsm_wait`(int *tid*);

int `adsm_wait`(int **tids*, int *num*);

DESCRIPTION

The first one will wait all child processes to finish. The second one will wait a specified process to finish. *tid* is a PVM task id. The last one will wait an array of processes to finish. Note that the *tids* are PVM task id, to find the corresponding Adsmith process numbers, please use the function `adsm_tid2procno`(). `adsm_wait`() is also a kind of synchronization operations. It involves an acquire access in the RC model.

Return Values

The number of process finished is returned.

See Also

`adsm_spawn`, `adsm_tid2procno`