JIAJIA User's Manual

(Version 2.1)

Weiwu Hu
Weisong Shi
Zhimin Tang
Rasit Eskicioglu*

Center of High Performance Computing
Institute of Computing Technology
Chinese Academy of Sciences

*Department of Computing Science
University of Alberta

May 1999

New Features of Version 2.1

Compared to last version (Version 2.0) of JIAJIA, this version includes the following new features:

- A home migration scheme is implemented to migrate home pages adaptively according to the application sharing pattern. In the scheme, pages that are written by only one processor between two barriers are migrated to the single writing processor. Migration messages are piggybacked on barrier messages and no additional communication is required for the migration. Though very simple, performance evaluation with SPLASH program suite and NAS Parallel Benchmarks shows that home migration can reduce diffs dramatically and improve performance significantly.

- A write vector technique is implemented to reduce message amount in home-based software DSMs. Other than fetching a whole page on a page fault as in traditional home-based software DSMs, the write vector technique divides a page into blocks and fetches only those blocks that are modified since the faulting processor fetched the page last time. Performance evaluation with some popularly accepted benchmarks shows that the write vector technique can reduce message amounts dramatically and consequently improve performance significantly in some benchmarks.

- An adaptive write detection scheme is implemented to reduce write faults on read-only pages. It automatically recognizes single write to a shared page by its home host and assumes the page will continue to be written by the home host in the future until the page is written by remote hosts. During the period the page is assumed to be singly written by its home host, no write detection of this home page is required and page faults caused by home host write detection can be avoided. Evaluation with some well-known DSM benchmarks reveals that the new write detection can reduce page faults dramatically and improve performance significantly. Since the adaptive write detection scheme works well, the cache-only write detection in Version 2.0 is removed.

- A new function call `jia_config()` is provided to turn home migration, write vector, adaptive write detection and other optimization methods on and off in the application program.

Contents

1 Introduction

JIAJIA is a software DSM which supports scope consistency. It has two distinguishing features compared to other recent software DSM systems such as TreadMarks. First, it takes the NUMA-like architecture and combines physical memories of multiple computers to form a larger shared space. Second, it implements the lock-based cache coherence protocol which totally eliminates directory and maintains coherence through accessing write notices kept on the lock.

JIAJIA runs on UNIX-like operating systems. Current version of JIAJIA can be run on Solaris 2.4, AIX4.1, and Linux 2.0. JIAJIA which runs on SUNOS 4.1 is also available but has not been integrated into the released version. One can email to `dsm@water.chpc.ict.ac.cn` to get source of SUNOS 4.1 version of JIAJIA.

The Windows NT version of JIAJIA can already work now. It will be released soon after more test are done.

Although JIAJIA is designed with a C programming interface, it is easy to refine the interface for FORTRAN77 programs if the FORTRAN compiler supports POINTER statement. We have successfully implemented the JIAJIA-FORTRAN interface in both SPARCstations and SP2, which provides more chances for JIAJIA to run real applications.

JIAJIA can be used both on networks of stand-alone workstations or workstations with NFS. The UDP/IP communication is used in the first release. Future versions might use other network interfaces to improve performance.

Currently, JIAJIA is an alpha software. It will be constantly maintained and improved to get better performance, to provide more functions, to be more user-friendly, and to support more platform. You can email to `dsm@water.chpc.ict.ac.cn` for being informed whenever a new version JIAJIA is released.

2 Installation

2.1 Getting JIAJIA

One can get JIAJIA sources through sending an message to `dsm@water.chpc.ict.ac.cn`, or through visiting the Center of High Performance Computing (CHPC) web pages. You can reach CHPC at `www.ict.ac.cn/chpc/index.html`.

2.2 JIAJIA Directory Hierarchy

When untarred JIAJIA creates a directory called `JIA`. This main directory includes the source directory `src/`, the library directory `lib/`, the documentation directory `docs/`, the applications directory `apps/`, and a `README` file. Figure 1 shows the general directory hierarchy of JIAJIA.

The `lib/` and each application directory under `apps/` contain a subdirectory for each platform. Currently, JIAJIA runs on Sun Sparc workstations running Solaris 2.4 or above and

IBM SP-2 or Dawning 1000A nodes running AiX 4.1 and use UDP/IP protocol stack on both platforms.

2.3 Compiling JIAJIA

Running the `Makefile` in the corresponding platform directory of `lib/` (e.g. `lib/solaris/` or `lib/aix41/`) will create the `libjia.a` library for the given platform.

Running the `Makefile` in corresponding platform directory of the specified application directory (e.g. `apps/sor/solaris/` or `apps/sor/aix41/`) will build the application for the given platform.

We use `gcc` version 2.7.2 and `gmake` 3.7.4 on all platforms. One can use other C compliers as wanted.

Although JIAJIA is designed with a C programming interface, it is easy to refine the interface for FORTRAN77 programs if the FORTRAN compiler supports POINTER statement. We have successfully implemented the JIAJIA-FORTRAN interface in SPARCstations, SP2, and Dawning 1000A, which provides more chances for JIAJIA to run real applications.

3 JIAJIA Running Environment

3.1 Configuration

JIAJIA can be used both on networks of stand-alone workstations or workstations with NFS. If you're using JIAJIA in a NFS environment, you must define an `NFS` flag in file `src/global.h` or in `lib/Makefile`.

In a stand-alone environment (i.e., when the `NFS` flag is not defined), the master should copy the executable program to remote hosts. Hence, each machine should be configured to make `rcp` from master to slaves possible. This can be done by adding slaves into file `/etc/hosts` of the master and adding the master into the file `.rhosts` of each slave.

JIAJIA looks for a configuration file called `.jiahosts` in the directory where the application runs. This file contains a list of hosts to run the applications, one per line. Each line contains 3 entries: the name of the host, the user name, and the password. Each host is identified by the combination of the host name and the user name. Multiple copies of the same program can run by different users on the same machine. The first line of `.jiahosts` should be the master on which the program is started. An `#` in `.jiahosts` starts an annotation line.

For example, suppose we want to run JIAJIA application on four hosts named `dsm@host1`, `dsm1@host1`, `dsm@host2`, and `dsm@host3`, and we indicate that `dsm@host1` is the master. Then, `.jiahosts` file would look like this:

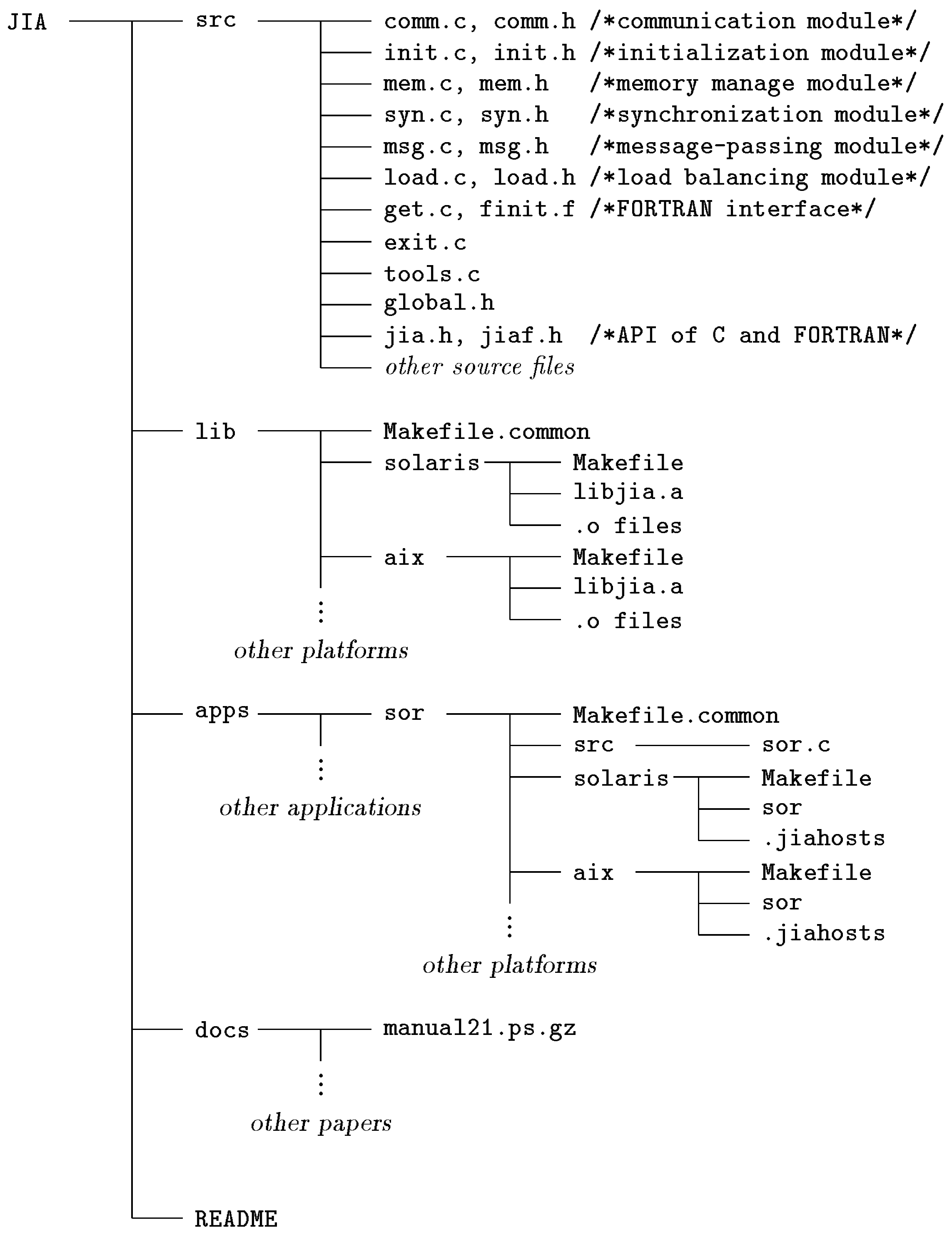
JIA
src
comm.c, comm.h /*communication module*/
init.c, init.h /*initialization module*/
mem.c, mem.h /*memory manage module*/
syn.c, syn.h /*synchronization module*/
msg.c, msg.h /*message-passing module*/
load.c, load.h /*load balancing module*/
get.c, finit.f /*FORTRAN interface*/
exit.c
tools.c
global.h
jia.h, jiaf.h /*API of C and FORTRAN*/
other source files
lib
Makefile.common
solaris
Makefile
libjia.a
.o files
aix
Makefile
libjia.a
.o files
other platforms
apps
sor
Makefile.common
src
sor.c
solaris
Makefile
sor
.jiahosts
aix
Makefile
sor
.jiahosts
other applications
other platforms
docs
manual21.ps.gz
other papers
README

Figure 1: Directory Hierarchy of JIAJIA

```
host1    dsm    xxxx
host1    dsm1   xxxx
host2    dsm    xxxx
host3    dsm    xxxx
```

If, for some reason, we decides to run an application on two hosts `dsm@host1` and `dsm@host3`, then we can either delete the two middle line of the above `.jiahosts`, or annotate the two middle line as follows:

```
host1     dsm    xxxx
#host1    dsm1   xxxx
#host2    dsm    xxxx
host3     dsm    xxxx
```

3.2 Running Applications

The user can start the application on the master once it is built. The master will automatically start the application on all slaves specified in `.jiahosts` (after copying the executable to the slaves first, if running in a stand-alone environment).

4 Application Programming Interface of JIAJIA

4.1 JIAJIA Calls At a Glance

JIAJIA provides a simple yet powerful API to the applications. This interface is defined in `<jia.h>`, which should be included by each application. JIAJIA provides following basic routines supporting shared memory parallel programming to the applications:

- `jia_init(argc,argv)`—initialize JIAJIA. It should be called at the beginning of the application. The main task of `jia_init()` is to start copies of the application on the hosts specified in `.jiahosts`. Also, `jia_init()` initializes internal data structures of JIAJIA.

- `jia_alloc3(int size,...)`—allocate shared memory. The parameter `size` indicates the number of bytes allocated. Other parameters allow the programmer to control data distribution arcoss hosts to improve performance. Techniques of distributing shared memory to improve performance will be discussed in the following subsection.

- `jia_lock(int lockid)`, `jia_unlock(int lockid)`—acquire and release a lock specified by `lockid`. `jia_lock()` and `jia_unlock()` provide a synchronization mechanism to ensure exclusive access to a critical section. `jia_lock()` and `jia_unlock()` should appear in pairs for obvious reasons. The maximum number of locks is specified in `jia/src/global.h`.

- `jia_barrier()`—performs a global barrier. A barrier provides a global synchronization mechanism by preventing any process from proceeding until all processes reach the barrier. Note that `jia_barrier()` can not be called inside a critical section enclosed by `jia_lock()` and `jia_unlock()`.

- `jia_exit()`—shut JIAJIA down.

Normally, the above basic routine is enough to write parallel programs with JIAJIA. To ease the parallel programming and improve performance, JIAJIA also provides the following subsidiary functions:

- `jia_config(int, int)`—System configuration. It is used to turn some optimization methods, such as home migration, write vector, and adaptive write detection, on and off.

- `jia_divtask(int *begin, int *end)` and `jia_loadcheck()`—load balance primitives. `jia_divtask()` divides tasks across processors and `jia_loadcheck()` checks loads of all processors.

- `jia_setcv(int cv)`,`jia_resetcv(int cv)`, and `jia_waitcv(int cv)`—set, reset, and wait on a conditional variable `cv`. Conditional variable provides another method of synchronization other than lock and barrier. However, no coherence of shared variables is enforced on conditional variables and conditional variable is normally used together with locks. The use of conditional variable will be explained further in the following section. The maximum number of locks is specified in `jia/src/global.h`.

- `jia_wait()`—similar to `jia_barrier()` in that both require the arrival of all processes before any one can proceed. They are different in that `jia_wait()` does not enforce any coherence operation across processors. Therefore, `jia_wait()` is a simple synchronization mechanism that requires all processes to wait altogether before going ahead.

- `jia_clock()`—return elapsed time since the start of application in seconds in `float` type. It can be used to summarize the time expended by the application.

- `jia_error(char *str)`—print out the error string `str` and shut down all processes started by `jia_init()`.

- `jia_startstat()` and `jia_stopstat()`—start and stop statistics. These two calls are valid only when the `DOSTAT` option is defined in `Makefile` when the system is made.

- `jia_send(char *buf, int len, int topid, int tag)`—an MPI-similar call, send `len` bytes of `buf` to host `topid`.

- `jia_recv(char *buf, int len, int frompid, int tag)`—an MPI-similar call, receive `len` bytes from host `frompid` to `buf`.

- `jia_bcast(char *buf, int len, int root)`—an MPI-similar call, send `len` bytes from `buf` of host `root` to `buf` of all hosts.

- `jia_reduce(char *snd, char *rcv, int cnt, int op, int root)`—an MPI-like call, reduce `cnt` numbers from all hosts to host `root` with operation `op`.

4.2 JIAJIA Program Structure

To write program with JIAJIA, one should include `jia.h` in the program. The program of JIAJIA implements the SPMD programming model, in which each processor run the same program on different parts of the shared data. Figure 2 shows the structure of a typical JIAJIA program.

Each JIAJIA `main()` program starts with creating multiple processes and initializing the internal data structures of JIAJIA by calling `jia_init(argc,argv)`. This is followed by allocating shared memory by `jia_alloc()` calls. `jia_alloc()` can be repeatedly called to allocate multiple shared regions. Normally, all shared memory should be allocated together befored they are used. A barrier is suggested to be called after allocating shared memory to ensure synchronization. The `slave()` routine is then called by each created process to work on the allocated shared data. Finally, the master waits for all slaves and then terminates by calling `jia_exit()`.

The `slave()` routine works on the shared memory. JIAJIA provides a process identifier, called `jiapid`, to each process. The `jiapid` of the master is 0, the `jiapid` of other processes is set according to the position of the associated host in the configuration file `.jiahosts`, i.e., the process identifier of the host specified in the nth line (excluding blank and/or comment lines) of `.jiahosts` is $n-1$ (this requires the master to be specified at the first line of `.jiahosts`). Besides, JIAJIA provides the global variable `jiahosts` as the current number of hosts. With `jiapid` and `jiahosts`, different processes can be set to work on different things.

4.3 Shared Memory Allocation and Distribution

Figure 3 shows JIAJIA's organization of the shared memory. In JIAJIA, each shared page has a designated home node and homes of shared pages are distributed across all nodes. References to home pages hit locally, references to non-home pages cause these pages to be fetched from their home and cached locally. A cached page may be in one of three states: Invalid (INV), Read-Only (RO), and Read-Write (RW). When the number of locally cached remote pages is larger than the maximum number allowed, some aged cache pages must be replaced to its home to make room for the new page. This allows JIAJIA to support shared memory that is larger than physical memory of one machine.

For example, suppose somebody allocates a shared memory of size 4MB in four hosts, 1MB in each. Then when host 0 want to access a location in the first 1MB (saying loaction 1024), the reference hits directly in the its home part of the shared space. If, however, host 0 wants to access a shared location whose home stays at host 3 (saying, location $3M + 1024$), then host

```
#include <jia.h>

char *ptr1, *ptr2, ...

slave()
{
  /* Program of slave, operate on allocated shared memory, synchronize
     with locks, conditional variables, or barrier as required */
}

main(argc,argv)
{
  /* initialize jiajia, create multiple processes */
  jia_init(argc,argv);

  /* allocate shared memory */
  ptr1=jia_alloc(size1);
  ptr2=jia_alloc(size2);
  ...
  ...
  ...
  jia_barrier();
  /* start the slave program, multiple processes work on the shared data */
  slave();
  ...
  ...
  ...
  /* exit jiajia */
  jia_exit();
}
```

Figure 2: Program structure of JIAJIA applications

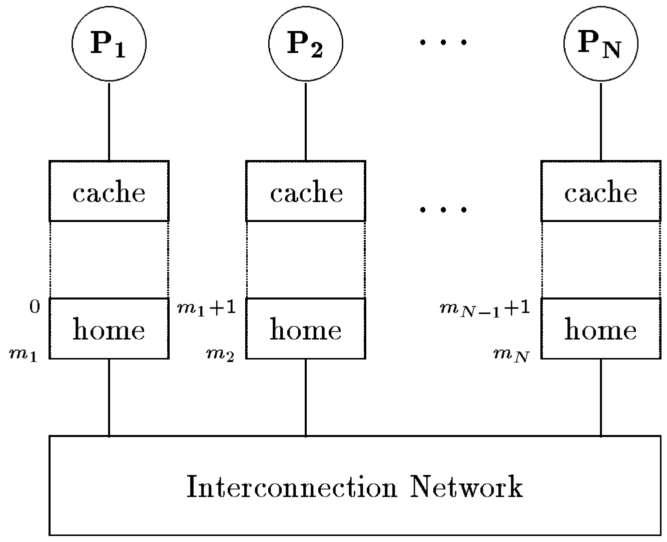

Figure 3: Memory Organization of JIAJIA

0 fetches the page which contains the referenced location from host 3 and keeps the page in its cache. Further references of host 0 to the same page (e.g., to location $3M + 1028$) will hit directly in the cache.

We can see from the memory organization of JIAJIA that, references to local home pages has less system overhead than those to remote pages. Hence, good program of JIAJIA should allocate and distributed shared memory properly so that most reference hit in home. JIAJIA provides flexible shared memory allocation calls to allow the programmer to control the initial distribution of homes of shared locations. The basic shared memory allocation function in JIAJIA is `jia_alloc3(size, blocksize, starthost)` which allocates `size` bytes cyclically across all hosts, each time `blocksize` bytes. The `starthost` parameter specifies the host from which the allocation starts. JIAJIA also defines three simpler shared memory allocation calls: `jia_alloc2(size, blocksize)` which equals to `jia_alloc3(size, blocksize, 0)`, `jia_alloc1(size)` which equals to `jia_alloc3(size, size, 0)`, and `jia_alloc(size)` which equals to `jia_alloc3(size, Pagesize, 0)`.

For example, consider a configuration of four nodes each with 16MB physical memory. Suppose an application uses a 24MB-size shared integer array `a`.

- If the programmer want to distribute the shared array in the way shown in Figure 4(a), then he(she) can allocate the array with the following call:

$$\mathtt{a} = (\mathtt{int}*)\mathtt{jia\_alloc3}(\mathtt{0x1800000}, \mathtt{0x600000}, \mathtt{0})$$

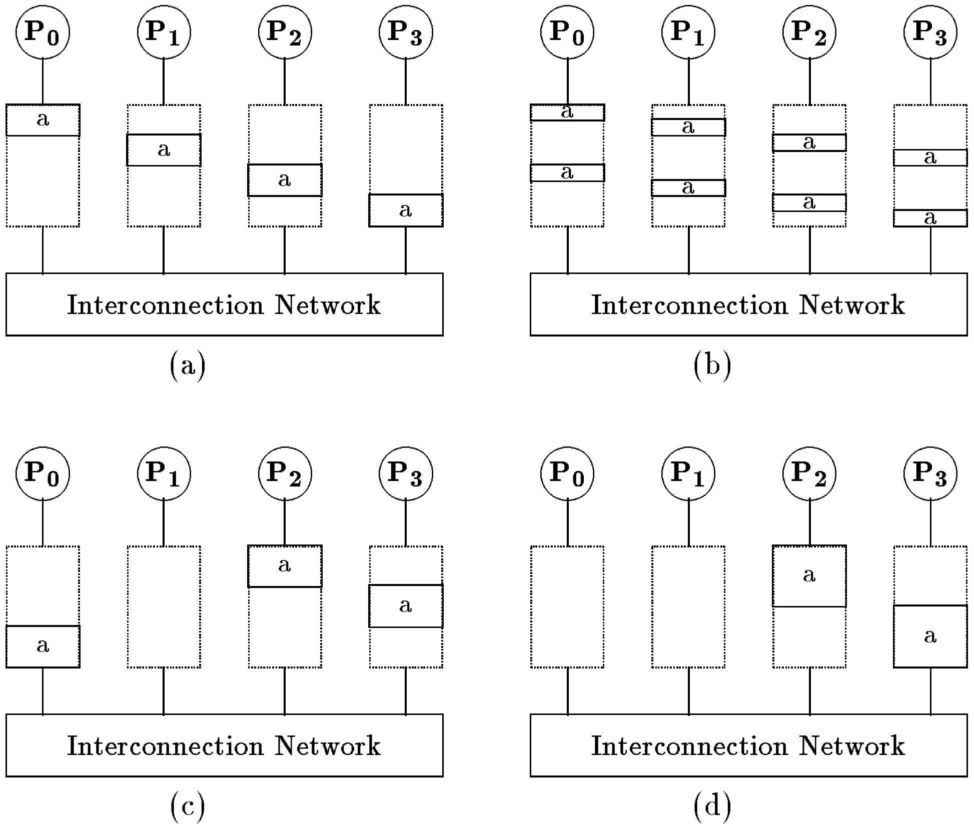

Figure 4: Memory Allocation Example

- The following call

  $$\texttt{a} = \texttt{(int*)jia\_alloc3(0x1800000, 0x300000, 0)}$$

  allocates shared memory in the way shown in Figure 4(b).

- The following call

  $$\texttt{a} = \texttt{(int*)jia\_alloc3(0x1800000, 0x800000, 2)}$$

  allocates shared memory in the way shown in Figure 4(c).

- The following call

  $$\texttt{a} = \texttt{(int*)jia\_alloc3(0x1800000, 0xc00000, 2)}$$

  allocates shared memory in the way shown in Figure 4(d).

4.4 Synchronization

JIAJIA supports three kinds of synchronization mechanisms: lock, barrier, and conditional variable.

A pair of `jia_lock()` and `jia_unlock()` encloses a critical section. `jia_lock()`s and `jia_unlock()`s can be called embeddedly provided they appear in pair.

A barrier provides a global synchronization mechanism by preventing any process from proceeding until all processes reach the barrier. Note that `jia_barrier()` can not be called inside a critical section enclosed by `jia_lock()` and `jia_unlock()`.

Conditional variable provides another method of synchronization other than lock and barrier. `jia_setcv(int cv)` and `jia_resetcv(int cv)` set and reset the conditional variable `cv` respectively, while `jia_waitcv(int cv)` waits on the conditional variable `cv` until it is set. Multiple hosts can wait on a conditional variable which is set by one host.

As has been stated, JIAJIA supports the scope consistency in which a new value written by a host is delivered to another (others) through a `jia_unlock()`—`jia_lock()` pair or a `jia_barrier()`, as is shown in the example of Figure 5.

Current version of JIAJIA does not enforced coherence arcoss a `jia_setcv()`—`jia_waitcv()` pair (Though last version of JIAJIA maintains coherence among hosts in a `jia_setcv()`—`jia_waitcv()` pair) because the concept of "scope" is unclear for conditional variables. However, the conditional variable provides a useful synchronization method in parallel programs with producer-consumer sharing pattern. Take the program segment in Figure 6 as an example. The producer process write a value "1" into the shared variable `x`, while the consumer reads the new value. Figure 6(a) and (b) provides two synchronization methods between the producer and the consumer. The synchronization method in Figure 6(b) is obviously much more efficient than that in Figure 6(a) because it does not require the consumer to constantly send request message for the lock as does the program in Figure 6(a). It worth to mention that, it will yield incorrect result if the `while` statement is put inside the critical section in the consumer of Figure 6(a), or if references to the shared variable `x` is not protected by locks in Figure 6(b).

It worth to re-emphasize that, since scope consistency is adopted as the memory consistency model, conflicting accesses (two accesses are conflicting if they accesses the same shared location and at least one is a write) from different processes must be separated by a release–acquire pair on the same lock or by a barrier.

4.5 Message-Passing Primitives

In our experience with users, we found that some users still need some message-passing primitives in the shared memory programming environment. For large problems, one would like to write message passing program for some modules (or just port existing message passing modules) and write shared memory program for other modules. On the other hand, using some message passing primitives in shared memory programs helps to improve performance in many cases.

```
             P1                  P2                  P3
---------------------------------------------------------------
         jia_lock(0);
           x=1;
         jia_unlock(0);
         jia_lock(1);
           y=1;
         jia_unlock(1);
                             jia_lock(0);
                               a=x;
                               b=y;
                             jia_unlock(0);
                                                    jia_lock(1);
                                                      a=x;
                                                      b=y;
                                                    jia_unlock(1);
---------------------------------------------------------------
Shared variable: x,y;
Local variable: a,b;
Initial: x=y=0;
Final:                    P2: a=1,b=0;          P3: a=0,b=1;
--------------------------------------------------------------
```

(a). Write Propagation in locks

```
             P1                  P2                  P3
--------------------------------------------------------------
           x=1;                y=1;                z=1;
         jia_barrier(1);     jia_barrier();     jia_barrier();
           a=x;                a=x;                a=x;
           b=y;                b=y;                b=y;
           c=z;                c=z;                c=z;
--------------------------------------------------------------
Shared variable: x,y,z;
Local variable:  a,b,c;
Initial: x=y=z=0;
Final:   P1: a=b=c=1;     P2: a=b=c=1;       P3: a=b=c=1;
--------------------------------------------------------------
```

(b). Write Propagation in barrier

Figure 5: Write Propagation in JIAJIA

```
     Producer              Consumer
-------------------------------------------
   jia_lock(0);
     x=1;
   jia_unlock(0);
                         while (a==0){
                           jia_lock(0);
                             a=x;
                           jia_unlock(0);
                         }
-------------------------------------------
```

(a). Producer-Consumer with Lock

```
     Producer                 Consumer
-------------------------------------------
   jia_lock(0);
     x=1;
   jia_unlock(0);
   jia_setcv(0);
                           jia_waitcv(0);
                           jia_lock(0);
                             a=x;
                           jia_unlock(0);
-------------------------------------------
```

(b). Producer-Consumer with Lock and Conditional Variable

Figure 6: Example of Conditional Variables

These observations motivate us to provide some simple message passing primitives in JIAJIA.

As has been stated, JIAJIA provides four MPI-like message passing primitives: `jia_send()`, `jia_recv()`, `jia_bcast()`, and `jia_reduce()`. Their usage is similar to their MPI counterparts. It should be mentioned that there is a receiving buffer in each host, so that it is unnecessary for the receiving host to sit waiting for the message before the sending host can send the message out. The `jia_bcast()` or `jia_reduce()` should be called by all processors for correct operation. For `jia_reduce()`, current version of JIAJIA supports summing, maximizing, and minimizing operations for integer, float, and double data types. The operations supported by `jia_reduce()` are defined in `opt.h`.

4.6 The `jia_config()` Call

The `jia_config()` call provides a flexible mechanism to configure the JIAJIA system at runtime while the library remains the same. Currently, it is used to switch on/off some performance optimization techniques. It accepts two parameters: the first parameter indicates the name of the optimization and the second parameter is either `ON` or `OFF`. All optimizations are initially closed until is is turned on with `jia_config(XXX, ON)`.

The optimizations provided by `jia_config()` include:

- Home Migration—A `jia_config(HMIG, ON)` call turns the home migration optimization on. The home migration scheme is implemented to migrate home pages adaptively according to the application sharing pattern. In the scheme, pages that are written by only one processor between two barriers are migrated to the single writing processor. Normally, the home migration optimization should not be turned on until shared data is initialized. If `jia_config(HMIG, ON)` is called before shared data initialization, all shared pages will be migrated to the initialization processor. Performance evaluation with SPLASH program suite and NAS Parallel Benchmarks shows that home migration can reduce diffs dramatically and improve performance significantly. See [3] for detail.

- Reducing Message Overhead—JIAJIA implements a write vector technique to reduce message amount. Other than fetching a whole page on a page fault as in traditional home-based software DSMs, the write vector technique divides a page into blocks and fetches only those blocks that are modified since the faulting processor fetched the page last time. A call `jia_config(WVEC, ON)` turns the write vector optimization on. Performance evaluation with some popularly accepted benchmarks shows that the write vector technique can reduce message amounts dramatically and consequently improve performance significantly in some benchmarks. See [4] for detail.

- Adaptive Write Detection—An adaptive write detection scheme is implemented in JIAJIA to reduce write faults on read-only pages. It automatically recognizes single write to a shared page by its home host and assumes the page will continue to be written by the home host in the future until the page is written by remote hosts. During the period the page is assumed to be singly written by its home host, no write detection of this home page is required and page faults caused by home host write detection can be avoided. The `jia_config(ADWD, ON)` call turns this optimization on. Evaluation with some well-known DSM benchmarks reveals that the new write detection can reduce page faults dramatically and improve performance significantly. See [5] for detail.

- Broadcast Barrier Messages—After all hosts arrive at a barrier, the barrier manager need to send barrier acknowledgement to all hosts. JIAJIA provides both one-by-one and tree-structure broadcast methods for barrier acknowledgement. The `jia_config(BROADCAST, ON/OFF)` call switches between these two methods.
- Load Balancing—The `jia_config(LOADBAL,ON)` call turns a simple load balancing optimization on. See next subsection for detail.

4.7 Load Balancing Primitives

JIAJIA provides a simple load balancing method for non-dedicated environments. It is mainly designed for regular applications and provides two primitives `jia_divtask(int *begin, int *end)` and `jia_loadcheck()` to divide tasks across processors and check computation power of each processor.

The `jia_divtask(int *start,int *end)` call regards the initial value of `(*end)-(*start)` as the total task to be divided and returns the task of each processor in `begin` and `end`. The normal way to parallelize the sequential loop

```
for (i=0;i<N;i++){
    ......
  }
```

is

```
start = N/jiahosts*jiapid;
end   = start + N/jiahosts;
if (jiapid == (jiahosts-1)) end = N;
for (i=start;i<end;i++){
    ......
  }
```

With the `jia_divtask()` call, the above parallization can be written as follows

```
start = 0;
end   = N;
jia_divtask(&start,&end);
for (i=start;i<end;i++){
```

```
                ......
           }
```

If the `LOADBAL` switch is closed, then the above program segment distributes loops across all processors evenly. If the LOADBAL switch is turned on by `jia_config(LOADBAL, ON)`, then loops are distributed across processors according their computation power.

The `jia_loadcheck()` call checks and records computation power of each processor in the system. The computation power of a processor is calculated by dividing its old computation power by the computation time since last `jia_loadcheck()` call. The computation time between two `jia_loadcheck()` calls is calculated as the elapsed time minus synchronization waiting time. The total computation power of all processors are always normalized to 1.

Though the above load balancing optimization scheme is very simple, preliminary evaluation result seems encouraging.

4.8 I/O

Normally, all I/O operations should be done in the master (host 0). Currently, no I/O operations on the standard I/O is allowed in slaves. If I/O in slaves is required, file I/O can be used. In JIAJIA, we redirect any write to stand output to the file `apps.log` (in stand-alone workstation environment) or apps-*i*.log (in NFS environment), where `apps` is the name of the executable application program and *i* is the host id.

5 Tuning the Performance

5.1 Exploiting Locality

In a DSM system, remote accesses takes much longer time to complete than those hit locally. Therefore, exploiting reference locality plays an important role in improving performance, and consequently in writing parallel applications of a DSM system. In a software DSM system, exploiting reference locality is even more important, not only because of the large latency of inter-process communication in network of workstations, but also because of serving remote access will cost CPU cycles in a software DSM system. In JIAJIA, two kinds of reference locality can be exploited when writing a parallel program: home locality and cache locality.

As has been stated, JIAJIA allows the programmer to fully control the distribution of data across processors. Experiences show that totally different performance can be get with different distribution of shared data. For example, consider the distribution of matrices in a matrix multiplication application. Suppose `a[1024][1024]` and `b[1024][1024]` are two float matrices to be multiplied, and `c[1024][1024]` is the result of multiplication of matrices `a` and `b`. Figure 7(a) and Figure 7(b) show two different allocation schemes for these three matrices. In the allocation scheme of Figure 7(a), the home of matrix `a`, `b`, and `c` are allocated row by row across hosts, while in the allocation scheme of Figure 7(b), the home of matrix `a`, `b`, and `c` are distributed band by band across all four hosts, in the way that each host homes the band of `a` and `c` it processes. Evaluation results on a four-processor SPARCstation 20 show that, when multiplied with middle product algorithm shown in Figure 8(a), the multiplication of 1024×1024 matrices takes 93 seconds when shared matrices are allocated in the way of Figure 7(a), while only 64 seconds are required when the data distribution method of Figure 8(b) is taken.

JIAJIA caches remote pages in a cache whose size is defined by the `Cachepages` constant in `jia/src/global.h`. The cache size can be adjusted when necessary to reach high reference locality. A good program of JIAJIA should take full advantage of the cache and avoid remote accesses as much as possible.

Consider again the matrix multiplication example. Figure 8(a) and 8(b) show the middle product and inner product algorithm of matrices multiplication. Suppose the matrices are distributed in the way of Figure 7(b), i.e., the matrices are distributed in the way that all memory references to matrix `a` and matrix `c` hit locally at the home, while only one quarter of references to matrix `b` hit locally, references to other three quarters of matrix `b` cause remote accesses. When the middle product algorithm in Figure 8(a) is employed, all reference to one row of `b` finish before next row is referenced. As a result, cache size has little influence to the performance of the middle product algorithm. It takes 64 seconds to finish a 1024×1024 multiplication in a four processor SPARCstation 20. When the middle product algorithm in Figure 8(a) is employed, the full matrix `b` is referenced for each iteration of `i`. To get acceptable performance, the cache should be able to hold the whole matrix `b`. Our eveluation results in the four processor SPARCstation 20 show that, when the cache size is 1024 pages with page size of 4096 bytes, the multiplication of 1024×1024 matrices takes 76 seconds, while 1164 seconds are required when the cache size is 512 pages. Statistics show that 768 remote page accesses happens in each processor when the cache size is 1024 pages, while the number increases to 114,684 when the cache size is 512 pages.

The page size defined in `jia/src/global.h` may also have influence on reference locality. One can adjust it (must be a multiple of the operating system page size) to reach high reference locality. For example, for a 8192×8192 float matrix, eight remote get page request is required to access a remote row when the page size is 4096 bytes, while just one remote get page request can bring a remote row to local cache if the page size is set to 32768 bytes.

5.2 Using `jia_config()` Call

As indicated in [2], [3], [4], and [5], different optimization works for different applications depending the sharing pattern of the application. Some optimization may even deteriorate performance because of the additional overhead of the optimization or because the optimization changes memory access behavior of the application.

Normally, the home migration and adaptive write detection techniques work well for regular problems with a large shared data set and with barrier as the main synchronization method.

```
#include <jia.h>
#define N 1024

float (*a)[N],(*b)[N],(*c)[N]; /*a,b,c are pointers to N-element arrays*/

main(argc,argv)
{
 jia_init(argc,argv);

 a=(float (*)[N])jia_alloc(N*N*sizeof(float));
 b=(float (*)[N])jia_alloc(N*N*sizeof(float));
 c=(float (*)[N])jia_alloc(N*N*sizeof(float));

 jia_barrier();
 worker();
 jia_barrier();

 jia_exit();
}
```

(a). Allocate a, b, and c row by row across hosts

```
main(argc,argv)
{
 jia_init(argc,argv);

 a=(float (*)[N])jia_alloc3(N*N*sizeof(float),N*N*sizeof(float)/jiahosts,0);
 b=(float (*)[N])jia_alloc3(N*N*sizeof(float),N*N*sizeof(float)/jiahosts,0);
 c=(float (*)[N])jia_alloc3(N*N*sizeof(float),N*N*sizeof(float)/jiahosts,0);

 jia_barrier();
 worker();
 jia_barrier();

 jia_exit();
}
```

(b). Allocate a, b, and c band-by-band across hosts

Figure 7: Shared Memory Allocation in Matrix Multiplication

```
void worker()
{int i,j,k;
 int start,end;
 float temp;

 start=(N/jiahosts)*jiapid;
 end=start+(N/jiahosts);

 for (j=0;j<N;j++){
   for (i=start;i<end;i++){
     temp=0.0;
     for (k=0;k<N;k++)
       temp+=a[i][k]*b[j][k];
     c[i][j]=temp;
   }
 }
}
```

(a). Middle Product Algorithm

```
void worker()
{int i,j,k;
 int start,end;
 float temp;

 start=(N/jiahosts)*jiapid;
 end=start+(N/jiahosts);

 for (i=start;i<end;i++){
   for (j=0;j<N;j++){
     temp=0.0;
     for (k=0;k<N;k++)
       temp+=a[i][k]*b[j][k];
     c[i][j]=temp;
   }
 }
}
```

(b). Inner Product Algorithm

Figure 8: Matrices Multiplication Process(matrix b is supposed to be transposed)

The write vector technique works well for some irregular applications in which the shared data set is not large and the shared pages are frequently referenced. Follows are some conclusion remarks of our evaluation.

- **Home Migration:** Performance evaluation with Water, LU, Ocean, MG, SOR, ILINK, and EM3D shows that most tested benchmarks exhibits single writer sharing behavior and the home migration scheme is effective in migrating home of pages to their single writer. Diffs are reduced dramatically in all benchmarks tested. As a side effect, home migration removes the effect of data pre-sending when a non-home writer sends diffs of a modified page to its home. Six of seven tested benchmarks achieve significant performance gains with home migration, and home migration makes host 0 the bottleneck and degrades the performance in ILINK.

- **Write vector:** Performance evaluation with some popularly accepted benchmarks (Water, Barnes, LU, SOR-Z, SOR-NZ, ILINK, and TSP) shows that the write vector technique reduces message amounts by times in four (LU, SOR-Z, ILINK, and TSP) of seven benchmarks, and consequently improves performance significantly in three (LU, ILINK, and TSP). The extra time overhead of the write vector technique is negligible in the tested benchmarks.

- **Adaptive Write Detection:** The effect of the adaptive write detection is evaluated with some matrix-based DSM benchmarks, include LU from SPLASH2, SOR from TreadMarks benchmarks, MG from NAS Parallel Benchmarks, and a real application EM3D from Institute of Electronics, Chinese Academy of Sciences. Evaluation results show that the adaptive write detection can reduce page faults dramatically and improve performance by 5% for LU, 43% for SOR, 17% for MG, and 19% for EM3D. Evaluation result also implies that the major overheads of virtual memory write detection is not caused by factors such as `mprotect()` and SIGSEGV signal handler call, but caused by other factors such as processor pipeline interruption, cache and TLB pollution, etc.

- **Broadcast** We observe a speedup of 30-50% for barrier operation in Dawning 1000A which connects eight Power PC 604 with an 100bps switched Ethernet). For bus structure network like Ethernet, the one-by-one barrier acknowledgement method is suggested because the tree-structure broadcast method does introduce additional overhead.

For detailed evaluation results, see [3], [4], and [5].

5.3 Trade Performance With Memory Space

The original design of JIAJIA (on network of SPARCstation 10 workstations, each with 32MB memory) regards the memory as a critical resource. Memory spaces for twins, messages, and write notices are dynamically allocated and freed to reduce memory overhead. Our experiments in Dawning 1000A indicates that, frequent allocation and free of memory space introduce a heavy oveahead to the operating system, and may often break the system down (Illegal Instruction error). Hence, to improve performance and to make JIAJIA more robust, we provide an option to reserve memory space for messages, twins, and write notices. Define `RESERVE_TWIN_SPACE` in `jia/src/mem.h` will switch the memory reservation on. The `RESERVE_TWIN_SPACE` option has already been defined `jia/src/mem.h` as the default choice. For environments (each node in Dawning 1000A has a 256MB memory) and applications which are not memory critical, the default memory reservation scheme is suggested.

6 Advanced Topics

6.1 Options and Values

Follows are some options and values in JIAJIA that some advanced users of JIAJIA may feel interest in changing for performance or other purpose.

- `Cachepages`—defined in `jia/src/global.h`. Specifies the cache size in number of pages.
- `Pagesize`—defined in `jia/src/global.h`. Specifies the page size of JIAJIA. Page size is the basic unit of coherence and communication in software DSM systems.
- `Maxhosts`—defined in `jia/src/global.h`. Maximum number of hosts of a parallel system.
- `Maxlocks`—defined in `jia/src/global.h`. Maximum number of locks in JIAJIA. The value of the `lockid` parameter of `jia_lock(lockid)` and `jia_unlock(lockid)` should be in the range `[0,Maxlocks)`.
- `Maxcvs`—defined in `jia/src/syn.h`. Maximum number of conditional variables in JIAJIA. The value of the `condv` parameter of `jia_setcv(condv)`, `jia_resetcv(condv)`, and `jia_waitcv(condv)` should be in the range `[0,Maxcvs)`.
- `Startaddr`—defined in `jia/src/global.h`. The start virtual address from which the allocation of shared space starts. JIAJIA allocates shared space through mapping them into a fixed virtual address. The `[Startaddr, Startaddr+Maxmemsize)` should not overlap with other space used by the same process.
- `Maxmemsize`—defined in `jia/src/global.h`. Maximum number of bytes of the shared memory allocated in JIAJIA.
- `Homepages`—defined in `jia/src/mem.h`. Maximum number of home pages in a host.
- `Maxfileno`—defined in `jia/src/init.h`. Maximum number of file descriptors that can be concurrently opened in UNIX. It should be no less than `4*Maxhosts*Maxhosts` in JIAJIA.

- `Maxqueue`—defined in `jia/src/comm.h`. The size of input and output queue for communication. Normally, it should no less than `2*maxhosts`.
- `RESERVE_TWIN_SPACE`—defined in `jia/src/mem.h`. JIAJIA will reverve space for twin if `RESERVE_TWIN_SPACE` is defined. See last section for its function.
- `DOSTAT`—defined in `jia/src/global.h`. JIAJIA will print out statistics (such as message numbers, remote get page numbers, ...) at the end of the program if `DOSTAT` is defined.
- `JIA_DEBUG`—defined in `jia/src/global.h`. `JIA_DEBUG` is used by the developer to debug JIAJIA system. It is not suggested to use by common users.

6.2 Error Messages

JIAJIA prints error messages and exits in exceptions such as incorrect returned value of system calls. Follows are some exception messages.

- "`req_fdcreate-->bind()`" or "`rep_fdcreate-->bind()`" — Bind error in initializing communication sockets. Current version of JIAJIA employs UDP protocol for communication. Each JIAJIA application requires `4*jiahosts*jiahosts` communication ports which are process id related numbers range from 10000 to 30000. As a result, communication ports of different applications may overlap and binding of a port to a socket will be failed. Rerun the application will correct this error.
- "`Access shared memory out of range` ⋯" — Address error, please check the program to make sure all its references to shared address are correct and meet the requirement of scope consistency.

(to be continued)

6.3 The `argc` and `argv` Parameters in JIAJIA

JIAJIA adds a parameter `-PStartport` in the command line when starting remote processes. Hence, both the `argc` and `argv` parameters are changed by JIAJIA in hosts other than host 0. Cares should be taken when you use these two parameters in your application program.

6.4 FORTRAN Interface

Currently, JIAJIA can only be used with FORTRAN 77 compilers supporting POINTER statement. Fortunately, both Solaris FORTRAN and AIX FORTRAN support POINTER.

To use JIAJIA in FORTRAN programs, you should include a file named `jiaf.h`, which contains necessary information about all subroutines and functions provided by JIAJIA. All

JIAJIA provided C functions can be called in FORTRAN as functions or subroutines, depending if it returns a value. `jiaf.h` also defined a COMMON block named `jia`, which contains two integer members, `jiapid` and `jiahosts`, for work sharing control among parallel processes. Since those two variables are defined in a COMMON block, any subroutine or functions using them must contain the following declaration:

```
COMMON /jia/ jiapid, jiahosts
```

Owing to the difference between C and FORTRAN in transferring function parameters, most JIAJIA subroutines must be called with their parameters specified by the `%val` modifier. For example, `jia_lock` can be called as

```
CALL jia_lock(%val(1))
```

and the following code allocates shared memory for matrix A and assigns value to one of its elements:

```
PARAMETER (N=1024)
POINTER (PA, A)
REAL A(N,N)
PA = jia_alloc(%val(N*N*4))
A(2,102) = 3.14159
```

To allocate a COMMON block in shared space, you should not list your data variables in COMMON statement. Instead, listed in COMMON blocks are only the associated pointer variables, since FORTRAN usually does not allow pointer-based variables to be used in COMMON statements. For example, to allocate the following block to shared space:

```
COMMON /shared/ a(N,N), b(N,N), c(N,N)
REAL a, b, c
```

you should write:

```
COMMON /shared/ pa, pb, pc
POINTER (pa, a), (pb, b), (pc, c)
REAL a(N,N), b(N,N), c(N,N)
pa = jia_alloc(%val(N*N*4))
pb = jia_alloc(%val(N*N*4))
pc = jia_alloc(%val(N*N*4))
```

To ease JIAJIA initialization in FORTRAN programs, a FORTRAN subroutine `jiaf_init`, which needs no parameters, is provided. You can initialize JIAJIA by simply calling it.

7 Documentations

Our web site `http://www.ict.ac.cn/chpc/dsm` include following documentations of JIAJIA.

- Papers.
- Technique reports (include a detail design report with flow charts of JIAJIA Version 1.0).
- Source codes of all versions.
- Presentations and Talks.
- User's manual.

Any work related to JIAJIA should cite one of [1], [2], [3], [4], or [5].

References

[1] Weiwu Hu, Weisong Shi, and Zhimin Tang, "JIAJIA: A Software DSM system Based on a New Cache Coherence Protocol", in *Proc. of 7th International Conference on High Performance Computing and Networking Europe*, LNCS 1593, pp. 463–472, Amsterdam, Apr. 1999. , available at `http://www.ict.ac.cn/chpc/dsm`.

[2] Weiwu Hu, Weisong Shi, and Zhimin Tang, "Reducing System Overheads in Home-based Software DSMs", in *Proc. of 13th Int'l Parallel Processing Sym.*, pp. 167–172, San Juan, Apr. 1999, available at `http://www.ict.ac.cn/chpc/dsm`.

[3] Weiwu Hu, Weisong Shi, and Zhimin Tang, "Home Migration in Home-based Software DSMs", accepted by *The 1st Workshop on Software Distributed Shared Memory*, available at `http://www.ict.ac.cn/chpc/dsm`.

[4] Weiwu Hu, "Reducing Message Overhead in Home-based Software DSMs", accepted by *The 1st Workshop on Software Distributed Shared Memory*, available at `http://www.ict.ac.cn/chpc/dsm`.

[5] Weiwu Hu, Weisong Shi, and Zhimin Tang, "Adaptive Write Detection in Home-Based Software DSMs", accepted by *The 8th IEEE International Symposium on High Performance Distributed Computing*, available at `http://www.ict.ac.cn/chpc/dsm`.

A Matrix Multiplication in C

Followed is the source code of middle product matrix multiplication. In order to reduce miss rate, we allocate the matrix `a`, `b`, `c` on the processors band by band uniformly. As a result, only $(n-1)/n$ of matrix `b` need to be fetched from remote processors during the course of computation.

```
#include <jia.h>

#define N  1024
float (*a)[N], (*b)[N], (*c)[N];

void seqinit()
{int i,j;
   if (jiapid==0) {
     for (i=0;i<N;i++)
       for (j=0;j<N;j++){
         a[i][j]=1.0;
         b[i][j]=1.0;
     }
   }
}

void worker()
{int i,j,k;
 int start, end;
 float temp;

 begin=(N/jiahosts)*jiapid;
 end=start+(N/jiahosts);

 for (j=0;j<N;j++)
   for (i=begin;i<end;i++){
       temp=0.0;
       for (k=0;k<N;k++)
         temp+=a[i][k]*b[j][k];
       c[i][j]=temp;
       if (i==start)
         printf("c[%d][%d]=%f\n",i,j,c[i][j]);
   }
}
```

```
void main(int argc,char **argv)
{int i,j,size;
 float t1,t2;

 jia_init(argc,argv);
 size = N*N*sizeof(float);
 a=(float (*)[N])jia_alloc3(size,size/jiahosts,0);
 b=(float (*)[N])jia_alloc3(size,size/jiahosts,0);
 c=(float (*)[N])jia_alloc3(size,size/jiahosts,0);

 jia_barrier();
 seqinit();
 jia_barrier();
 jia_startstat();
 t1=jia_clock();
 worker();
 jia_barrier();
 t2=jia_clock();
 if (jiapid==0)
   printf("Total time for matric multiply is == %10.2f seconds\n", t2-t1);
 jia_exit();
}
```

B Matrix Multiplication in FORTRAN

```
      program matrix

      include 'jiaf.h'

      parameter  (n=1024)
      integer i, j
      common /shared/ pa, pb, pc
      pointer (pa, a), (pb, b), (pc, c)
      real a(n,n), b(n,n), c(n,n), t1, t2

      call jiaf_init()
      call jia_barrier()
      pa = jia_alloc(%val(4*n*n),%val(4*n*n/jiahosts),%val(0))
      pb = jia_alloc(%val(4*n*n),%val(4*n*n/jiahosts),%val(0))
      pc = jia_alloc(%val(4*n*n),%val(4*n*n/jiahosts),%val(0))
      if (jiapid .eq. 0) then
        do j = 1, n
              do i = 1, n
                a(i,j) = 1.0
                b(i,j) = 1.0
              end do
        end do
      endif
      call jia_barrier()
      t1 = jia_clock()
      call worker()
      call jia_barrier()
      t2 = jia_clock()
      if (jiapid .eq. 0) then
      print *, 'Elapsed time is', t2-t1, ' seconds'
      endif
      call jia_exit()
      end

subroutine worker()
      include 'jiaf.h'
      integer begin, end
      parameter  (n=1024)
      common /shared/ pa, pb, pc
```

```
pointer (pa, a), (pb, b), (pc, c)
real a(n,n), b(n,n), c(n,n)

begin = n/jiahosts*jiapid + 1
end = n/jiahosts*(jiapid+1)
do j = begin, end
  do i = 1, n
        t = 0
        do k = 1, n
          t = t + a(k,i)*b(k,j)
        enddo
        c(i,j) = t
        if ((jiapid .eq. 0) .and. (i .eq. 1)) then
        print *, 'c(', i, j, ') = ', c(i,j)
        endif
  enddo
enddo
call jia_barrier()
return
end
```

C *LU* Factorization in C

Followed is the source code of *LU* factorization. The matrix is factored column by column in the algorithm. To improve reference locality, the matrix is distributed across processors in a round-robin manner in which columns are allocated contigyously and entirely in the local memory of processors that "own" them.

As the factorization procudure proceed (j), the update of the current column is finished by processor that "owns" it. Since all the rest updates of the trailing submatrix depend on the new value of the current column, a `jia_barrier()` is called to propagate the up-to-date value of the current column to all processors. Barrier is the only synchronization mechnism used in *LU* factorization.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <jia.h>
#define N          1024
#define CHECK      0
#define EPSILON    1e-5
#define MAXRAND    32767.0
#define min(a,b)   (((a)<(b)) ? (a) : (b))
#define fabs(a)    (((a)>0.0) ? (a) : (0.0-(a)))

double (*a)[N];
double **old,**new;
char luerr[80];

void seqinita()
{int i,j;
 if (jiapid==0) {
   if (CHECK==1){
     old=(double **)malloc(N*sizeof(double *));
     for (i=0;i<N;i++){
       old[i]=(double *)malloc(N*sizeof(double));
     }
     new=(double **)malloc(N*sizeof(long));
     for (i=0;i<N;i++){
       new[i]=(double *)malloc(N*sizeof(double));
     }
   }
   srand48((long) 1);
   for (i=0; i<N; i++){
     for (j=0; j<N; j++) {
       a[j][i] = ((double) lrand48())/MAXRAND;
       if (i==j) a[j][i] *= 10.0;
       if (CHECK==1) old[j][i] = a[j][i];
     }
   }
 }
}

void lua()
{register int i,j,k;
 int begin;
 double temp;

 for (j=0;j<N;j++){
   if ((j%jiahosts)==jiapid){
     if (fabs(a[j][j])>EPSILON){
       temp=a[j][j];
       for (i=j+1;(i<N);i++){
         a[j][i]/=temp;
       }
     }else{
       sprintf(luerr,"Matrix a is singular, a[%d][%d]=%lf",j,j,a[j][j]);
       jia_error(luerr);
     }
   }
   jia_barrier();
   begin=j+1-(j+1)%jiahosts+jiapid;
   if (begin<(j+1)) begin+=jiahosts;
   for (k=begin;k<N;k+=jiahosts){
       temp=a[k][j];
       for (i=j+1;(i<N);i++){
         a[k][i]-=(a[j][i]*temp);
       }
   }
 }
}

void checka()
{int i,j,k;
```

```
 double temp;

 if ((CHECK==1)&&(jiapid==0)){
   for (i=0;i<N;i++)
     for (j=0;j<N;j++){
       temp=0.0;
       for (k=0;k<=min(i,j);k++)
         if (i==k) temp+=a[j][k];
         else      temp+=a[k][i]*a[j][k];
       new[j][i]=temp;
       if (fabs(old[j][i]-new[j][i])>EPSILON){
         sprintf(luerr,"Incorrect! old[%d][%d]=%lf, new[%d][%d]=%lf\n",
                           i,j,old[j][i],i,j,new[j][i]);
         jia_error(luerr);
       }else if (j==0){
         printf("old[%d][%d]=%14.6lf, new[%d][%d]=%14.6lf\n",
                           i,j,old[j][i],i,j,new[j][i]);
       }
     }
 }
}

void main(int argc, char **argv)
{int i,j,k;
 float clock1,clock2;
 int size;
  jia_init(argc,argv);
  size = N*N*sizeof(double);
  a=(double (*)[N])jia_alloc3(size,size/N,0);
  jia_barrier();
  seqinita();
  jia_barrier();
  clock1=jia_clock();
  lua();
  clock2=jia_clock();
  printf("total time elapsed = %8.2f second\n",(clock2-clock1));
  jia_barrier();
  checka();
  jia_exit();
}
```

D *LU* Factorization in FORTRAN

```
program lu
include 'jiaf.h'
parameter  (n=1024)
integer i, j
integer  MAXRAND
common /shared/ pa
pointer (pa, a)
character*30 luerr
common /res/ old(0:n-1,0:n-1)
real t1, t2, temp, a(0:n-1,0:n-1)

call jiaf_init()
print *, 'jiahosts = ', jiahosts
pa = jia_alloc(%val(4*n*n), %val(n*4),0)
call jia_barrier()

MAXRAND = 32768
if (jiapid .eq. 0) then
  do j = 0, n-1
        do i = 0, n-1
          a(i,j) =1.0*(rand(0)/MAXRAND)
  if (i.eq.j) then
                a(i,j)=a(i,j)*10.0
          endif
          old(i,j) = a(i,j)
        enddo
  enddo
endif
call jia_barrier()
luerr = 'Matrix is sigular . . .'
t1 = jia_clock()
temp=lua()
if (temp.eq.-1) then
  call jia_error(luerr)
endif
call jia_barrier()
t2 = jia_clock()
if (jiapid .eq. 0) then
print *, 'Elapsed time is', t2-t1, ' seconds'
```

```
endif
temp = checka(1)
if (temp.eq.-1) then
   luerr = 'Incorrect results '
call jia_error(luerr)
else
endif
call jia_exit()
end

function lua()
include 'jiaf.h'
integer begin
real temp
parameter  (n=1024)
common /shared/ pa
pointer (pa, a)
real a(0:n-1,0:n-1)

do 20 j = 0,n-1
  if((j-j/jiahosts*jiahosts).eq.jiapid) then
        if (abs(a(j,j)).gt.EPSILON) then
           temp = a(j,j)
           do i = j+1, n-1
                 a(i,j)=a(i,j)/temp
           enddo
        else
          lua = -1
          goto 100
        endif
 endif

 call jia_barrier()
 begin=(j+1)/jiahosts*jiahosts+jiapid
 if (begin.lt.(j+1)) then
        begin=jiahosts+begin
 endif
 do 10 k=begin,n-1,jiahosts
   temp=a(j,k)
   do i=j+1, n-1
         a(i,k) = a(i,k)-a(i,j)*temp
```

```
                    enddo
10         continue
20         continue
                 lua =0
100      return
         end

                 function checka(check)
                 integer check
                 real temp, EPSILON
                 parameter  (n=1024)
                 common /shared/ pa
                 common /jia/ jiapid, jiahosts
                 pointer (pa, a)
                 real a(0:n-1,0:n-1),new(0:n-1,0:n-1)
                 common /res/ old(0:n-1,0:n-1)

                 EPSILON = 1e-5
                   if (jiapid.eq.0) then
                          if (check.eq.1) then
                            print *, 'Begin Check the result'

                            do 110 i = 0,n-1
                                  do 120 j= 0,n-1
                                    temp = 0.0
                                    do 130 k = 0, min(i,j)
                                          if (i.eq.k) then
                                            temp = temp+a(k,j)
                                          else
                                            temp = temp+a(i,k)*a(k,j)
                                          endif
130                 continue
                                    new(i,j) = temp
                                    if (abs(old(i,j)-new(i,j)).gt. EPSILON) then
                                           print *, 'Incorrect!'
                                           print *,'old(',i,',',j,')=',old(i,j)
                                           print *,'new(',i,',',j,')=',new(i,j)
                                           checka = -1
                                           goto 140
                                    endif
120               continue
```

```
110          continue
                       endif
                     endif
                     checka =0
140       return
                     end
```

E π Calculation in C

Followed is the source code of π calculation by integrating `f(x) = 4/(1 + x**2)`. Different processors are responsible for different parts of intergration. At the end of the program, each processor adds it's partial value to the final vale `pi` in a critical section. `jia_lock()` and `jia_unlock()` is used to ensure mutual exclusive references to `pi` in a critical section.

```
#include <stdio.h>
#include <math.h>
#include <jia.h>

double f(double a)
{
    return (4.0 / (1.0 + a*a));
}

void main(int argc,char *argv[])
{
    int n,i,begin,end;
    double PI25DT = 3.141592653589793238462643;
    double mypi, h, sum, x, a;
    float startt, endt;
    double *pa;

    jia_init(argc,argv);
    n = 1000000;
    pa=(double *)jia_alloc(sizeof(double));
    jia_barrier();

    if (jiapid==0) {
      *pa =0.0;
    }
    jia_barrier();
    startt = jia_clock();
    h   = 1.0 / (double) n;
    sum = 0.0;
    begin = n/jiahosts*jiapid+1;
    end = n/jiahosts*(jiapid+1);

    for (i = begin; i <= end; i++){
      x = h * ((double)i - 0.5);
      sum += f(x);
    }
    mypi = h * sum;
    jia_lock(1);
     *pa= *pa+mypi;
    jia_unlock(1);
    jia_barrier();
    endt = jia_clock();
    if (jiapid==0) {
       printf("pi is approximately %.16f, Error is %.16f\n",
       *pa, fabs(*pa - PI25DT));
       printf("Elapsed time = %f\n", endt-startt);
    }
}
```

F π Calculation in FORTRAN

```
      program main
      include 'jiaf.h'

      real*16  PI25DT
      parameter (PI25DT = 3.141592653589793238462643d0)
      real*16  pi, h, sum, x, f, a
      integer i
      integer begin, end
      real startt, endt
      pointer (pp, pi)

      f(a) = 4.d0 / (1.d0 + a*a)
      n = 10000
      call jiaf_init()
      pp = jia_alloc(%val(16))
      call jia_barrier()
 10   if ( jiapid .eq. 0 ) then
            pi = 0.0d0
      endif
      call jia_barrier()
      startt= jia_clock()
      h = 1.0d0/n
      sum  = 0.0d0
      begin = n/jiahosts*jiapid+1
      end = n/jiahosts*(jiapid+1)
      print *, 'begin = ', begin, 'end= ',end
      do 20 i = begin, end
             x = h * (dble(i) - 0.5d0)
             sum = sum + f(x)
 20   continue
      sum = h * sum

      call jia_lock(%val(1))
             pi = pi + sum
      call jia_unlock(%val(1))
      call jia_barrier()
      endt = jia_clock()

      if (jiapid .eq. 0) then
```

```
              write(6, 97) pi, abs(pi-PI25DT)
 97      format(' pi is ', F27.25, ' +- ', F27.25)
              print*, 'Elapsed time is ', endt-startt, 'seconds'
        endif

        call jia_exit()
30    stop
        end
```