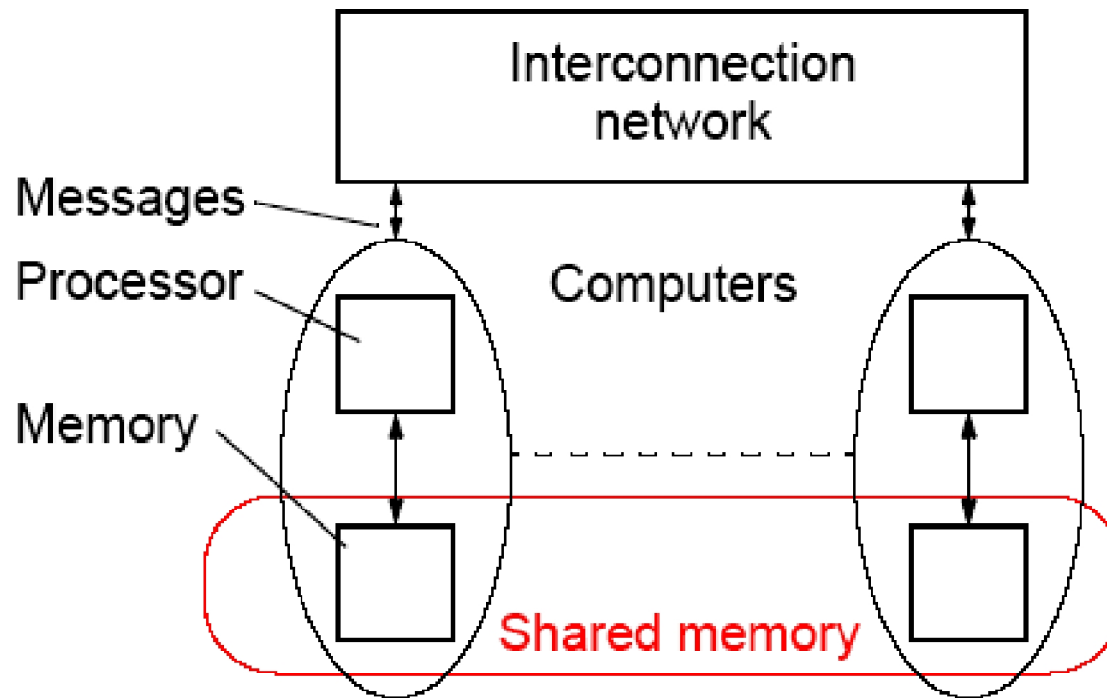# Distributed Shared Memory

Distributed Shared Memory

Making the main memory of a cluster of computers look as though it is a single memory with a single address space.

Then can use shared memory programming techniques.

DSM System

Still need messages or mechanisms to get data to processor, but these are hidden from the programmer:

Advantages of DSM

- System scalable

- Hides the message passing - do not explicitly specific sending messages between processes

- Can us simple extensions to sequential programming

- Can handle complex and large data bases without replication or sending the data to processes

Disadvantages of DSM

- May incur a performance penalty

- Must provide for protection against simultaneous access to shared data (locks, etc.)

- Little programmer control over actual messages being generated

- Performance of irregular problems in particular may be difficult

5

Methods of Achieving DSM

- Hardware

   Special network interfaces and cache coherence circuits

- Software

   Modifying the OS kernel
   Adding a software layer between the operating system
   and the application - most convenient way for teaching
   purposes

Software DSM Implementation

- Page based - Using the system's virtual memory

- Shared variable approach- Using routines to access shared variables

- Object based- Shared data within collection of objects. Access to shared data through object oriented discipline (ideally)

Consistency Models

- Strict Consistency - Processors sees most recent update, i.e. read returns the most recent wrote to location.
- Sequential Consistency - Result of any execution same as an interleaving of individual programs.
- Relaxed Consistency- Delay making write visible to reduce messages.
- Weak consistency - programmer must use synchronization operations to enforce sequential consistency when necessary.
- Release Consistency - programmer must use specific synchronization operators, acquire and release.
- Lazy Release Consistency - update only done at time of acquire.

Strict Consistency

Every write immediately visible
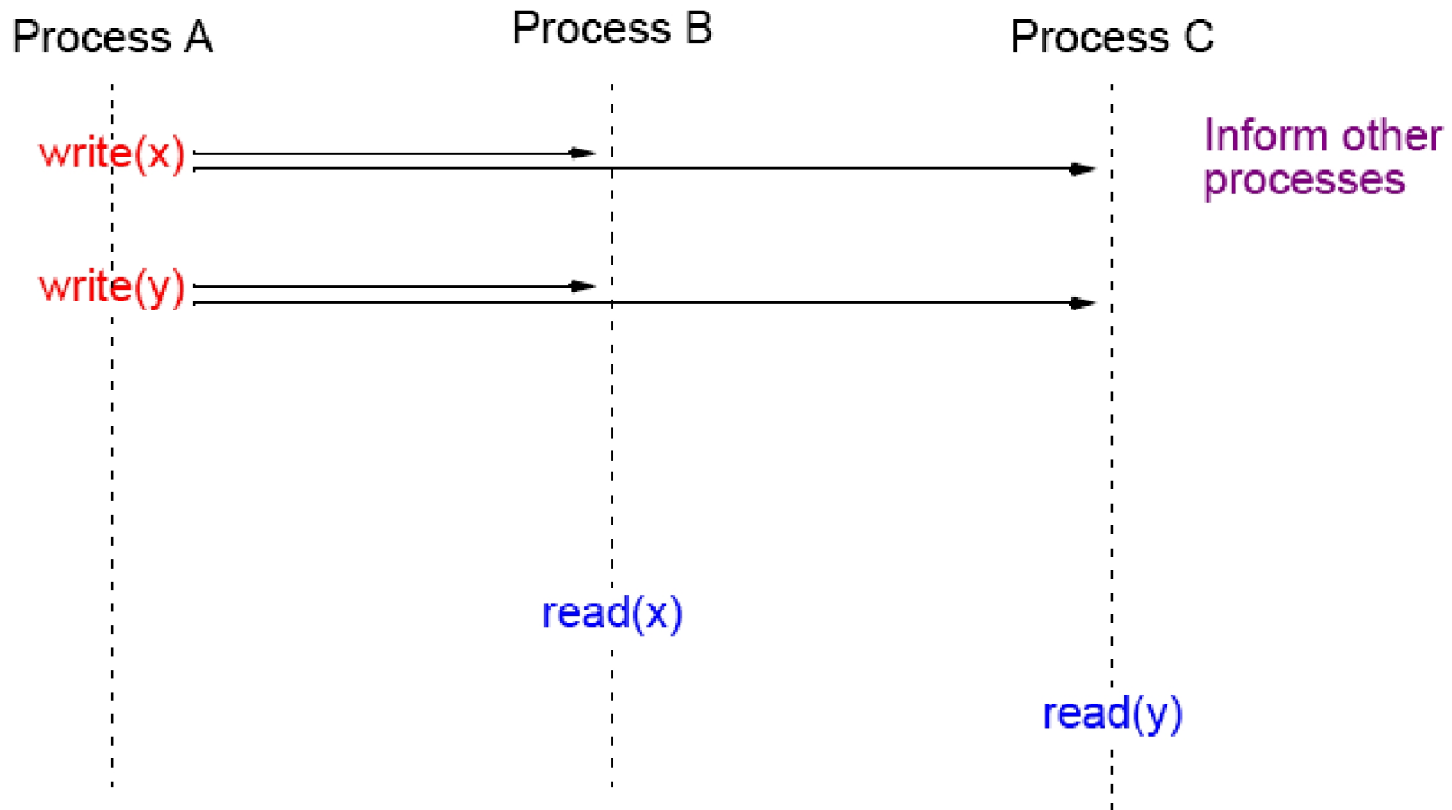
Disadvantages: number of messages, latency, maybe unnecessary.
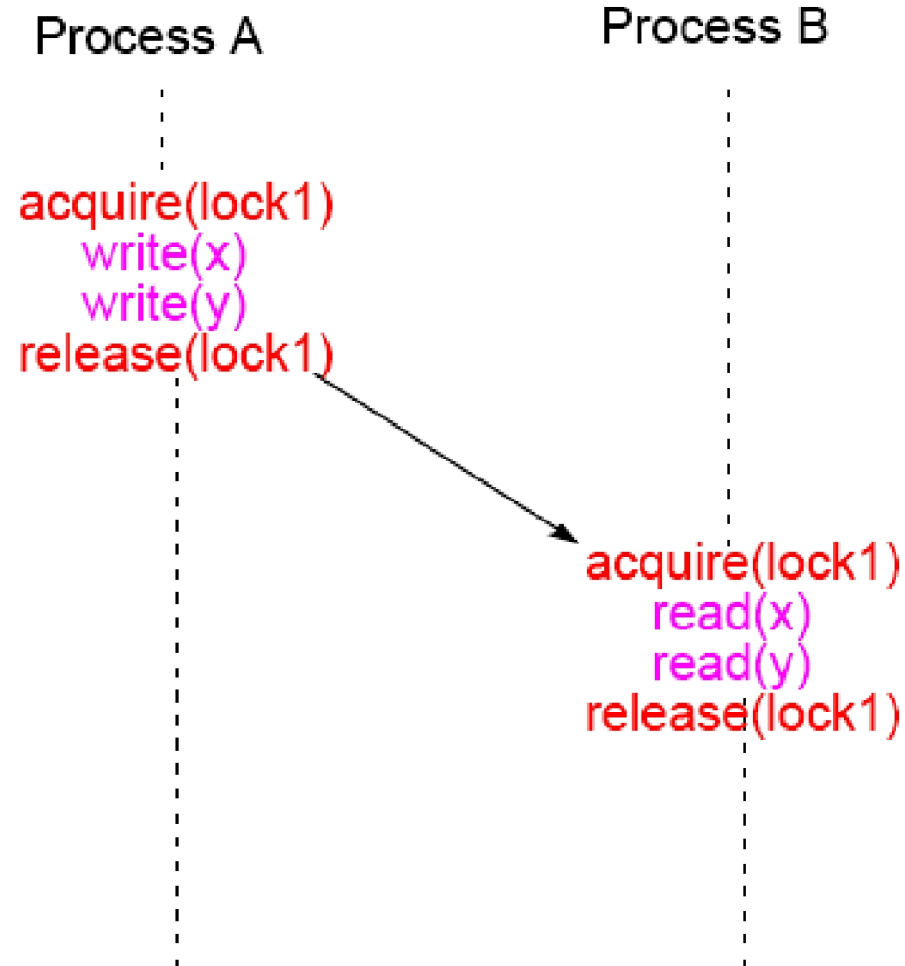
Consistency Models used on DSM Systems
Release Consistency

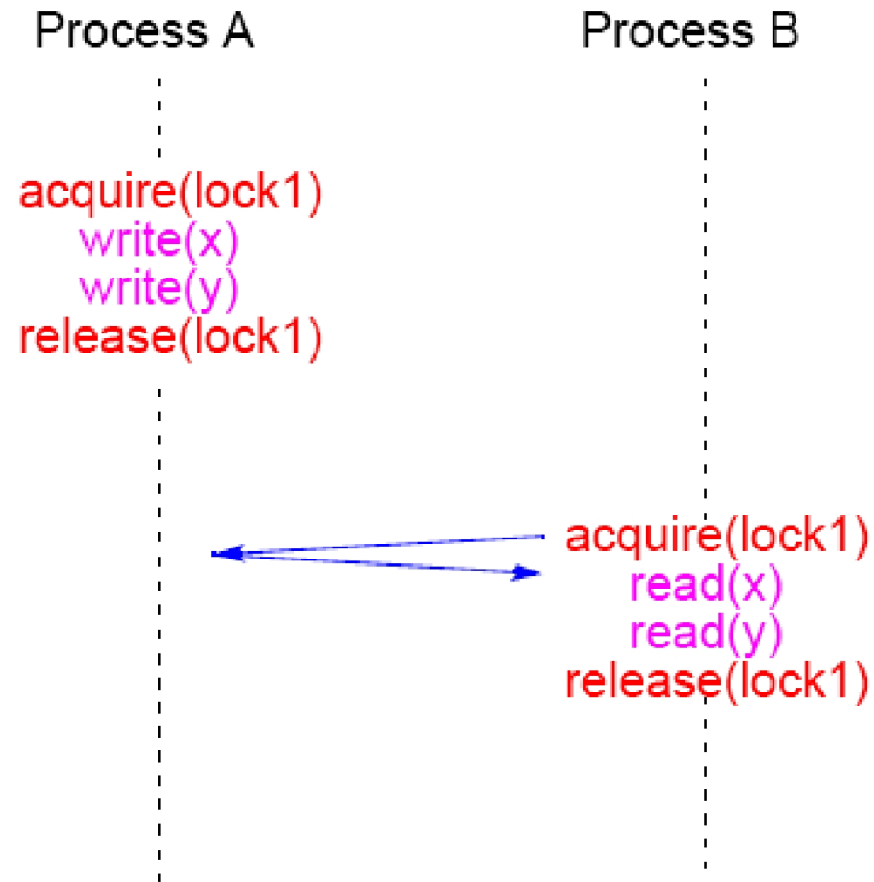An extension of weak consistency in which the synchronization operations have been specified:

- acquire operation - used before a shared variable or variables are to be read.

- release operation - used after the shared variable or variables have been altered (written) and allows another process to access to the variable(s)

Typically acquire is done with a lock operation and release by an unlock operation (although not necessarily).

Release Consistency

Process A   Process B

acquire(lock1)
  write(x)
  write(y)
release(lock1)

                              acquire(lock1)
                                 read(x)
                                 read(y)
                              release(lock1)

Lazy Release Consistency


```
Process A                          Process B

acquire(lock1)
  write(x)
  write(y)
release(lock1)



                                   acquire(lock1)
                                     read(x)
                                     read(y)
                                   release(lock1)
```

Advantages: Fewer messages

Process

To start a new process or processes:

adsm_spawn(filename, count)

Example

adsm_spawn("prog1",10);

starts 10 copies of prog1 (10 processes). Must use Adsmith routine to start a new process. Also version of adsm_spawn() with similar parameters to pvm_spawn().

Process "join"

adsmith_wait();

will cause the process to wait for all its child processes (processes it created) to terminate.

Versions available to wait for specific processes to terminate, using pvm tid to identify processes. Then would need to use the pvm form of adsmith() that returns the tids of child processes.

Access to shared data (objects)

Adsmith uses "release consistency." Programmer explicitly needs to control competing read/write access from different processes.

Three types of access in Adsmith, differentiated by the use of the shared data:

- Ordinary Accesses - For regular assignment statements accessing shared variables.
- Synchronization Accesses - Competing accesses used for synchronization purposes.
- Non-Synchronization Accesses - Competing accesses, not used for synchronization.

Ordinary Accesses - Basic read/write actions

Before read, do:

adsm_refresh()

to get most recent value - an "acquire/load." After write, do:

adsm_flush()

to store result - "store"

Example

int *x;   //shared variable

.

.

adsm_refresh(x);

a = *x + b;

Synchronization accesses

To control competing accesses:

- Semaphores

- Mutex's (Mutual exclusion variables)

- Barriers.

available. All require an identifier to be specified as all three class instances are shared between processes.

Semaphore routines

Four routines:

       wait()
       signal()
       set()
       get().

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

};
```

Mutual exclusion variables – Mutex

Two routines
lock
unlock()

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

Example

```
int *sum;
AdsmMutex x("mutex");
x.lock();
        adsm_refresh(sum);
        *sum += partial_sum;
        adsm_flush(sum);
x.unlock();
```

Barrier Routines

One barrier routine

barrier()

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

Example

AdsmBarrier barrier1("sample");

.

.

barrier1.barrier(procno);

Non-synchronization Accesses

For competing accesses that are not for synchronization:

adsm_refresh_now(void *ptr);

And

adsm_flush_now(void *ptr);

refresh and flush take place on home location (rather than locally) and immediately.

Features to Improve Performance

Routines that can be used to overlap messages or reduce number of messages:

- Prefetch

- Bulk Transfer

- Combined routines for critical sections

Prefetch

adsm_prefetch(void *ptr)

used before adsm_refresh() to get data as early as possible.

Non-blocking so that can continue with other work prior to issuing refresh.

Bulk Transfer

Combines consecutive messages to reduce number. Can apply only to "aggregating":

adsm_malloc(AdsmBulkType *type);
adsm_prefetch(AdsmBulkType *type)
adsm_refresh(AdsmBulkType *type)
adsm_flush(AdsmBulkType *type)

where AdsmBulkType is defined as:

enum AdsmBulkType {
adsmBulkBegin,
AdsmBulkEnd
}

Use parameters **AdsmBulkBegin** and **AdsmBulkEnd** in pairs to "aggregate" actions.
Easy to add afterwards to improve performance.

Example

adsm_refresh(AdsmBulkBegin);
adsm_refresh(x);
adsm_refresh(y);
adsm_refresh(z);
adsm_refresh(AdsmBulkEnd);

Routines to improve performance of critical sections

Called "Atomic Accesses" in Adsmith.

adsm_atomic_begin()
adsm_atomic_end()

Replaces two routines and reduces number of messages.

```
Acquire              adsm_atomic_begin()
Refresh
local code
Flush                adsm_atomic_end()
Release
```

Sending an expression to be executed on home process

Can reduce number of messages. Called "Active Access" in Adsmith. Achieved with:

adsm_atomic(void *ptr, char *expression);

where the expression is written as [type] expression.
Object pointed by ptr is the only variable allowed in the expression (and indicated in this expression with the symbol @).

Example

```
int *x = (int*)adsm_malloc)"x",sizeofint(int));
adsm_atomic(x,"[int] @=@+10");
```

Collect Access

Efficient routines for shared objects used as an accumulator:

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

where num is the number of processes involved in the access, and *ptr points to the shared accumulator

Example
(from page 10 of Adsmith User Interface document):

```
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
```

Other Features
Pointers

Can be shared but need to use adsmith address translation routines to convert local address to a globally recognizable address and back to an local address:

To translates local address to global address (an int)

int adsm_gid(void *ptr);

To translates global address back to local address for use by requesting process

void *adsm_attach(int gid);

Message passing

Can use PVM routines in same program but must use adsm_spawn() to create processes (not pvm_spawn(). Message tags MAXINT-6 to MAXINT used by Adsmith.

Information Retrieval Routines

For getting host ids (zero to number of hosts -1) or process id (zero to number of processes -1):

int adsm_hostno(int procno = -1);
- Returns host id where process specified by process number procno resides. (If procno not specified, returns host id of calling process).

int adsm_procno();
-Returns process id of calling process.

int adsm_procno2tid(int procno);
-Translates process id to corresponding PVM task id.

int adsm_tid2procno(int tid)
translates PVM task id to corresponding process id.

DSM Implementation Projects
Using underlying message-passing software

- Easy to do

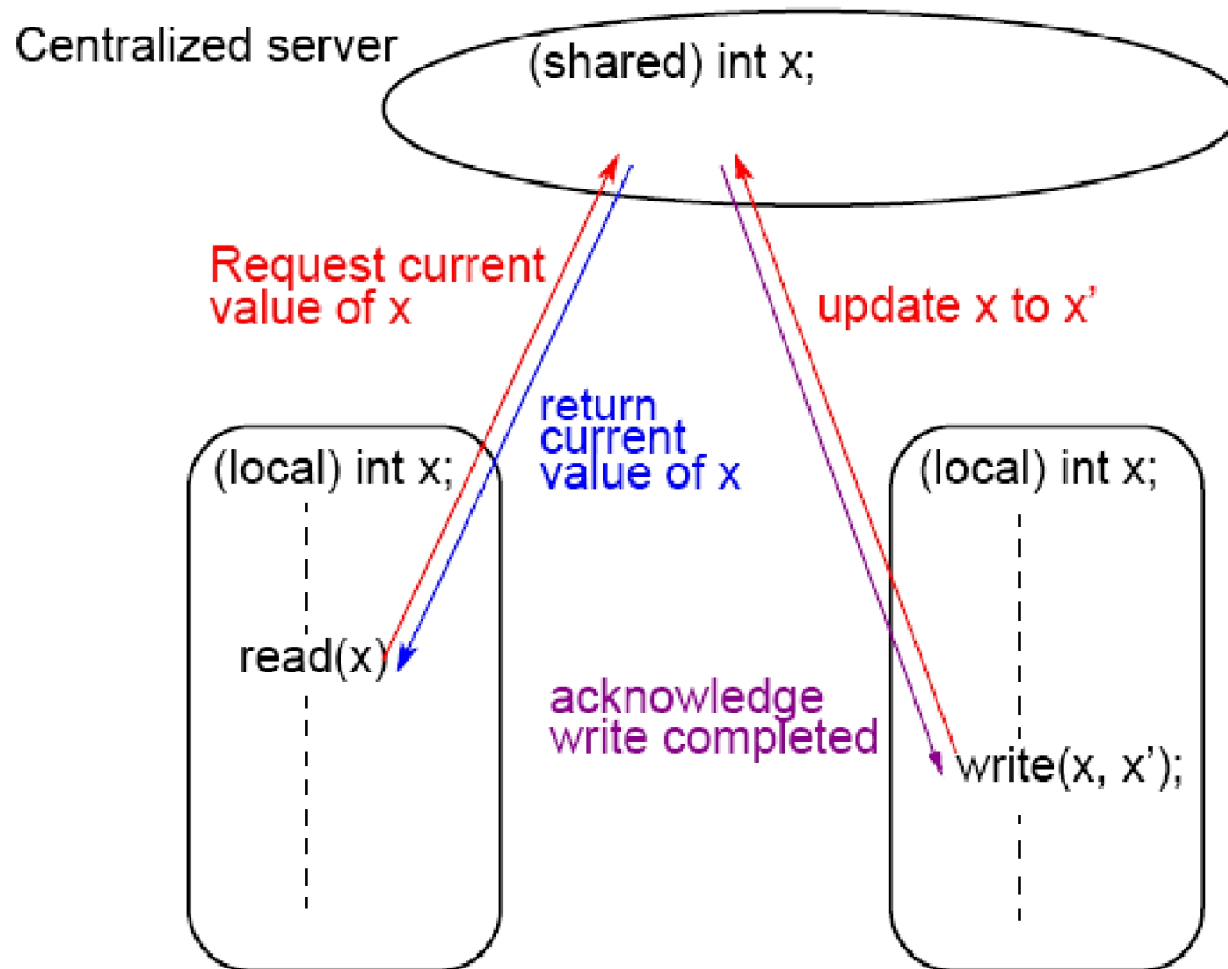- Can sit on top of message-passing software such as MPI.

Issues in Implementing a DSM System

- Managing shared data - reader/writer policies

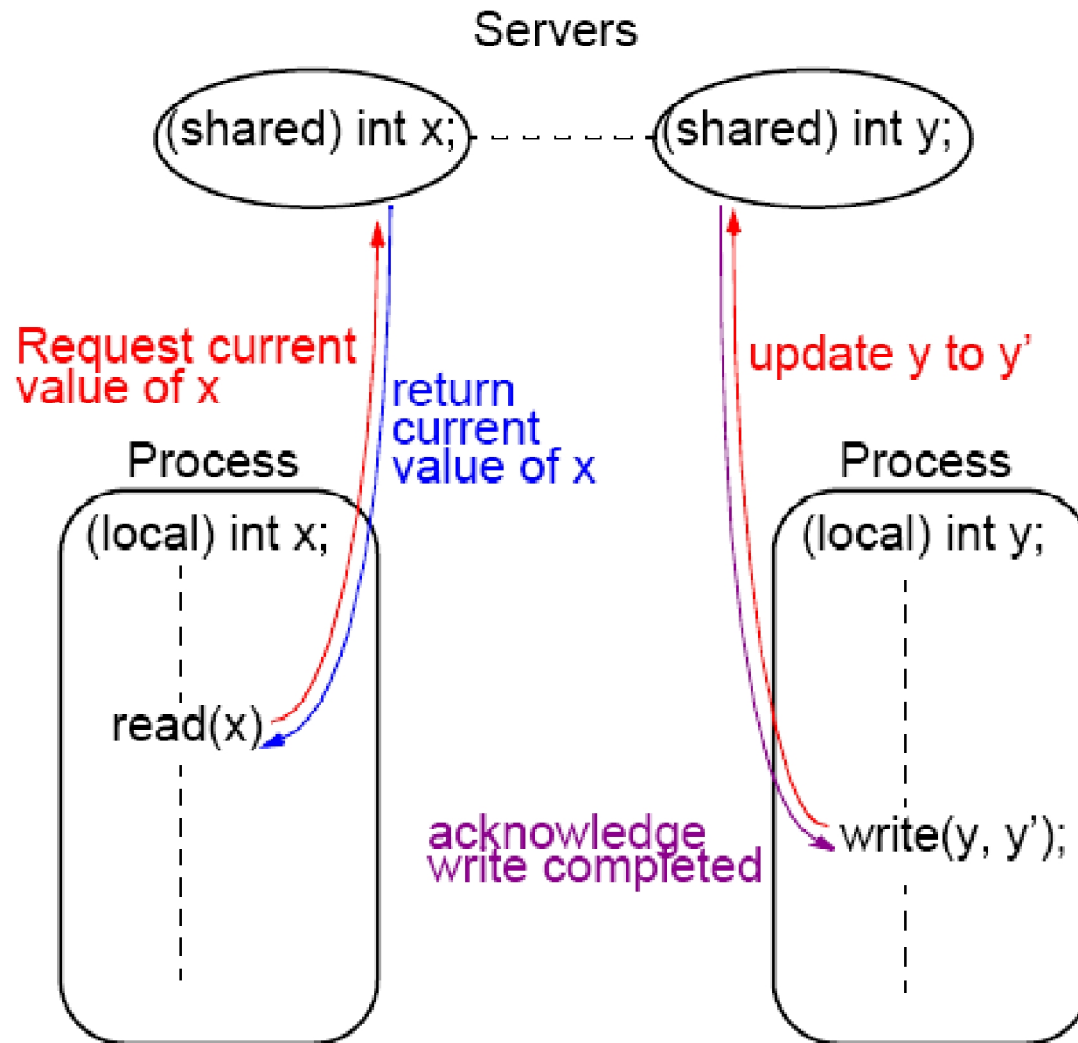- Timing issues - relaxing read/write orders

Reader/Writer Policies

• Single reader/single writer policy - simple to do with
  centralized servers

• Multiple reader/single writer policy - again quite simple
  to do

• Multiple reader/multiple writer policy - tricky

Simple DSM system using a centralized server

Centralized server

(shared) int x;

Request current value of x

update x to x'

return current value of x

(local) int x;

read(x)

acknowledge write completed

(local) int x;

write(x, x');

Simple DSM system using multiple servers

Simple DSM system using multiple servers and multiple reader policy
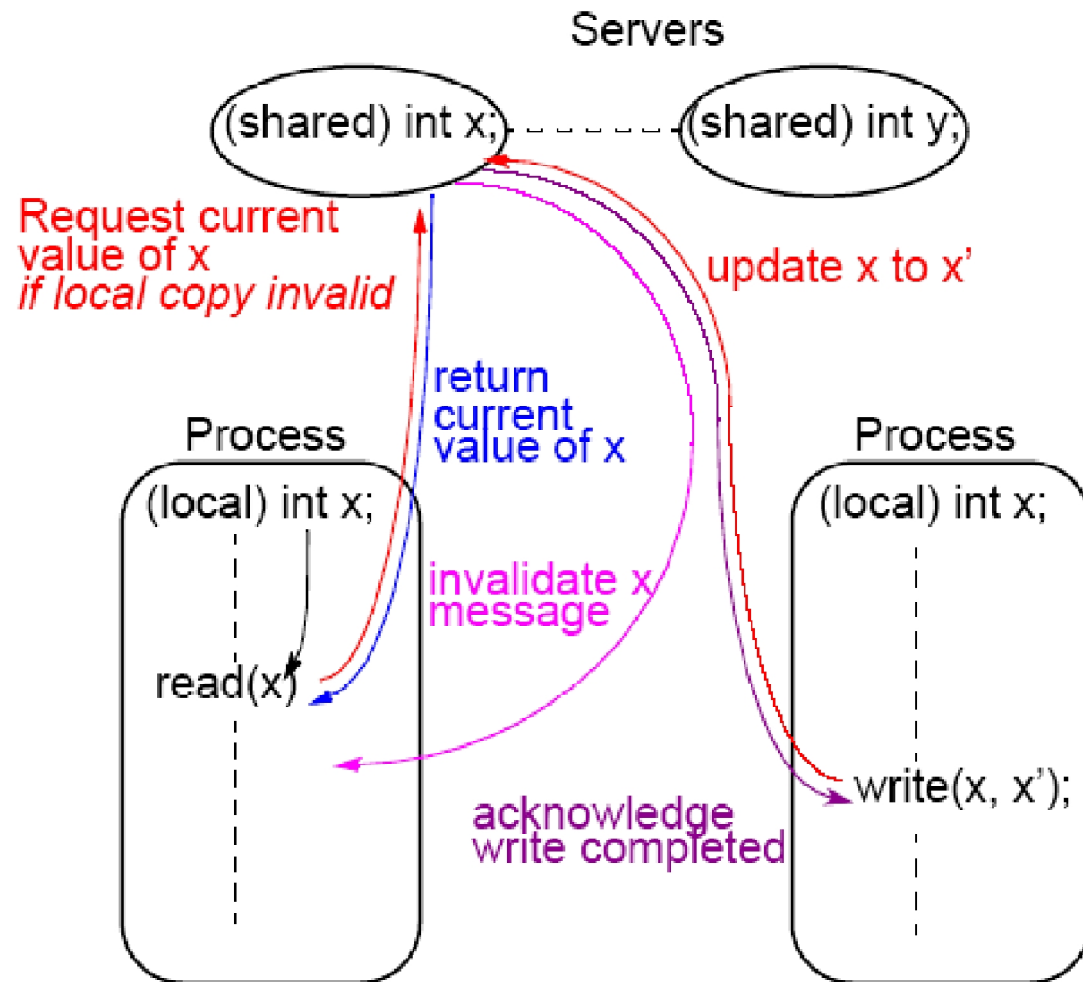
# Shared Data with Overlapping Regions A New Concept Developed at UNC-Charlotte

Based upon earlier work on so-called over-lapping connectivity interconnection networks

A large family of scalable interconnection networks devised – all have characteristic of overlapping domains that nodes can Interconnect
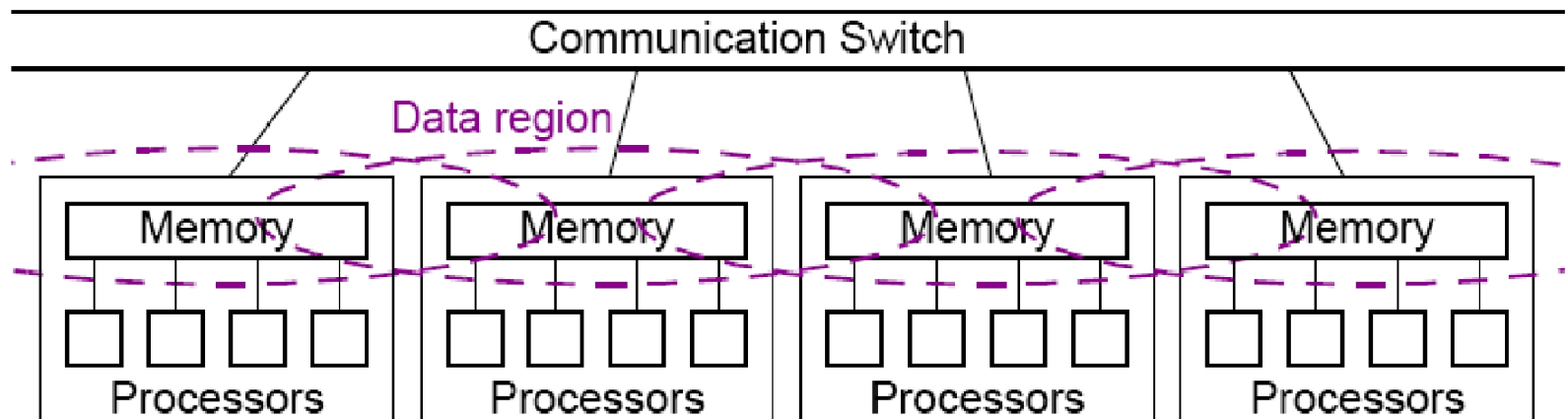
Many applications require communication to logically nearby processors

Overlapping Regions

Example

Processor/computer

P_i

Sphere of influence
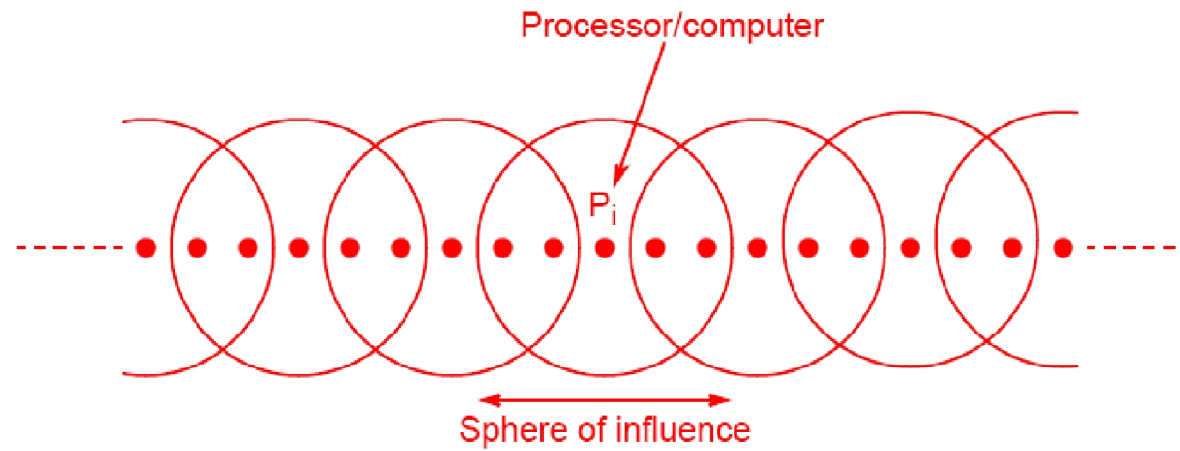
Symmetrical Multiprocessor System with Overlapping Data Regions

Static and Dynamic Overlapping Groups

- Static - defined prior to program execution – add routines for declaring and specifying these groups

- Dynamic - shared variable migration during program execution

Shared Variable Migration between Data Regions

Migrate shared variables according to a usage algorithm

Servers
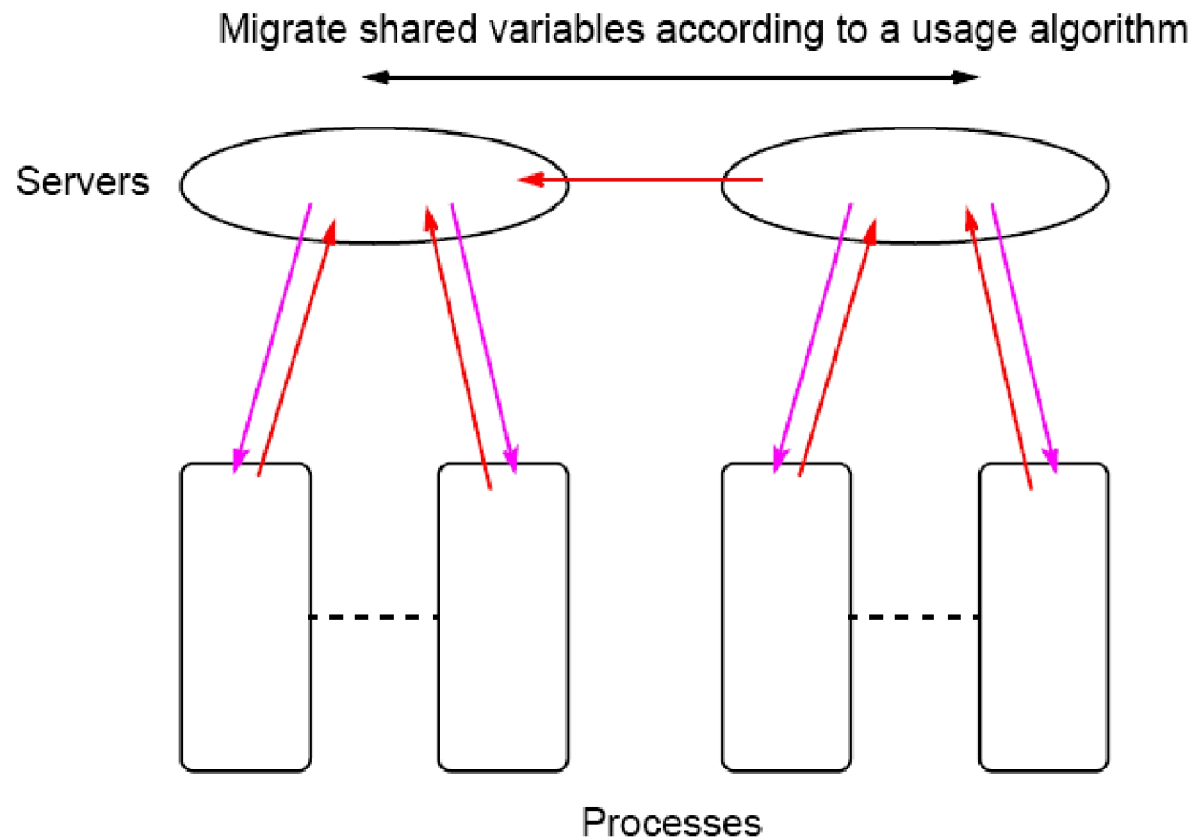
Processes

Application

- used to make System scalable

- Hides the message passing - do not explicitly specific sending messages between processes

- Can handle complex and large data bases without replication or sending the data to processes

Scope of Research

1. Migrating-Home Protocol for Software Distributed Shared Memory.

2. Dynamic Computation Migration in Distributed Shared Memory Systems