KDSM(KAIST Distributed Shared Memory) 시스템의 설계 및 구현

(Design and Implementation of KDSM(KAIST Distributed Shared Memory) System)

이 상 권 † 윤 희 철 †† 이 준 원 ††† 맹 승 렬 †††
(Sang-Kwon Lee) (Hee-Chul Yun) (Joonwon Lee) (Seungryoul Maeng)

요 약 본 논문에서는 KDSM(KAIST Distributed Shared Memory) 시스템의 설계 및 구현에 관해서 설명한다. KDSM은 Linux 2.2.13 상에서 실행되는 사용자 수준 라이브러리로 구현되었고, TCP/IP를 기반 통신 구조로 사용한다. KDSM은 페이지 기반 무효화 프로토콜(page-based invalidation protocol)과 다중 쓰기 프로토콜(multiple writer protocol)을 기반으로 하고, HLRC(Home-based Lazy Release Consistency) 메모리 일관성 모델을 사용한다. KDSM의 성능을 측정하기 위해서 4개의 과학계산용 응용 프로그램을 실행하여 JIAJIA와 성능 비교를 하였다. 그 결과, 2개의 응용은 같은 결과를 냈고, 나머지 2개는 KDSM의 성능이 우수하였다.

키워드 : 소프트웨어 분산공유메모리, 공유가상메모리, 병렬처리

Abstract In this paper, we give a detailed description of KDSM(KAIST Distributed Shared Memory) system. KDSM is implemented as a user-level library running on Linux 2.2.13, and TCP/IP is used for communication. KDSM uses page-based invalidation protocol, multiple-writer protocol, and supports HLRC(Home-based Lazy Release Consistency) memory consistency model. To evaluate performance of KDSM, we executed 4 scientific applications and compared the result to JIAJIA. The results showed that performance of KDSM is almost equal to JIAJIA for 2 applications and performance of KDSM is better than JIAJIA for 2 applications.

Key words : Software Distributed Shared Memory, Shared Virtual Memory, Parallel Processing

1. 서 론

최근 고성능 마이크로 프로세서와 고속 네트워크의 등장으로 인해서 NOW(Networks Of Workstations)와 같은 클러스터 시스템을 병렬 처리에 사용하고자 하는 연구들이 활발히 진행중이다. 소프트웨어 분산공유메모리(Software Distributed Shared Memory)는 특별한 하드웨어를 필요치 않고 구현하기 쉽기 때문에 NOW 상에서 공유메모리를 제공하는데 효과적이다.

† 비 회 원 : 한국과학기술원 전산학과
sklee@camars.kaist.ac.kr
†† 비 회 원 : 한국전자통신연구원 컴퓨터소프트웨어기술연구소 연구원
hcyun@etri.re.kr
††† 종신회원 : 한국과학기술원 전산학과 교수
joon@camars.kaist.ac.kr
maeng@camars.kaist.ac.kr
논문접수 : 2001년 4월 12일
심사완료 : 2002년 2월 20일

일반적으로 소프트웨어 분산공유메모리는 주소공간을 페이지 단위로 분할하며, 각 페이지 단위로 복제(*replication*) 및 이동(*migration*)을 시킨다. 페이지 기반의 소프트웨어 분산공유메모리 시스템의 문제점은 (1) 높은 통신 오버헤드와 (2) 캐쉬 및 통신 단위(granularity)인 페이지 크기가 크다는 점이다. 특히 두 번째 문제는 거짓 공유(false sharing) 문제를 야기시킨다. 이와 같은 문제들을 해결하기 위해서 다양한 알고리즘 및 메모리 모델들이 제안되었다[1,2,3,4,5,6].

IVY[1]는 최초의 소프트웨어 분산공유메모리 시스템으로 SC(Sequential Consistency) 모델을 사용하였다. SC는 직관적이라는 장점을 가지지만 프로토콜 오버헤드가 크다는 단점을 가진다. Munin[3]은 소프트웨어적인 방식으로 RC(Release Consistency)[7] 메모리 모델을 구현한 ERC(Eager Release Consistency) 프로토콜을 사용하였다. ERC는 무효화 메시지를 임시 버퍼에 저장

했다가 동기화 지점에서 다른 노드들에게 그것을 전달하는 식으로 통신 횟수를 줄였다. 또한 다중 기록자(multiple writer) 프로토콜을 사용해서 거짓 공유(false sharing) 문제를 해결하고자 노력하였다. TreadMarks[2]는 LRC(Lazy Release Consistency) 모델을 사용하여 실제 락(lock)을 얻는 프로세스에게만 무효화 메시지가 전달되도록 하였다. 따라서 LRC와 ERC 모두 RC 모델을 구현하는 프로토콜이지만, LRC가 ERC 보다 한 단계 더 완화된 프로토콜이라고 할 수 있다. JIAJIA[4]는 ScC(Scope Consistency)[6] 모델을 사용해서 LRC와 동일한 인터페이스를 사용하면서도 거짓 공유로 인한 성능 저하를 한단계 더 줄이고자 하였다. 또한 각 공유 페이지마다 홈을 가지는 홈 기반 프로토콜을 이용하였다.

본 논문에서는 페이지 기반의 소프트웨어 분산공유메모리인 KDSM(KAIST Distributed Shared Memory) 시스템에 관해서 설명한다. KDSM 시스템은 여러 대의 PC를 100Mbps Fast Ethernet으로 엮은 클러스터 시스템 상에서 실행된다. 각 PC들은 500 Mhz Pentium III 프로세서를 탑재하고 있으며, 운영체계로 Linux 2.2.13를 사용한다. KDSM 시스템은 사용자 수준 프로세스(user-level process)로 동작한다. 기반 통신 구조로는 TCP/IP를 사용하고, 외부 노드로부터 전송되는 메시지를 처리하기 위해서 SIGIO 시그널을 가로채는 방식을 사용한다. 캐쉬 일관성 프로토콜은 페이지 기반 무효화 프로토콜(page-based invalidation protocol)과 다중 쓰기(multiple reader multiple writer) 프로토콜을 바탕으로, HLRC(Home-base Lazy Release Consistency)[5] 메모리 일관성 모델을 사용한다.

본 논문의 구성은 다음과 같다. 2장에서는 실행 및 프로그래밍 환경에 대해서 설명한다. 3장에서는 공유메모리 구조, 메모리 맵핑, 메모리 할당 방법에 대해서 설명한다. 4장에서는 분산 공유메모리의 기반 통신 환경에 대해서 설명한다. 5장에서는 기본 HLRC 프로토콜에 대해서 설명한다. 6장에서는 여러 가지 응용을 사용하여 시스템의 성능 측정 결과를 설명하고, 7장에서 현재 진행중인 연구 및 향후 연구 방향에 대해서 설명한다.

2. 실행 및 프로그래밍 환경

KDSM은 Linux 커널 2.2.13 버전에서 개발되고 테스트되었다. KDSM 시스템을 이용하여 응용 프로그램을 수행하기 위해서는, 응용 프로그램의 실행파일이 있는 디렉토리에 .config 라는 환경 설정 파일을 만들어야 한다. 이 파일은 프로세스들을 실행시킬 노드에 대한 정보를 가지는 파일로써 각 라인마다 호스트의 이름을 기록한다. 그림 1의 예를 들면, can1, can2, can3, can4라는 네 개의 호스트에 각각 하나의 프로세스를 실행하겠단 의미이다.

```
can1
can2
can3
can4
```

그림 1 config 파일의 예

KDSM 시스템은 공유메모리 응용 프로그램 개발을 위해 다양한 API(Application Programming Interface)들을 제공한다. API 인터페이스는 dsm.h 파일에 정의되어 있으며, 각 응용 프로그램은 이 파일을 포함해야 한다. KDSM 시스템이 응용 프로그램에 제공하는 API는 표 1과 같다.

표 1 KDSM이 제공하는 사용자 API

API	설명
DsmInit(argc, argv)	DSM 시스템을 초기화시키는 함수로, 응용 프로그램의 첫부분에서 호출되어야 한다. DsmInit()은 환경 설정 파일을 읽어서 각 호스트에 응용 프로그램을 실행시킨다.
DsmExit()	모든 프로세스를 동기화시킨 후 DSM 시스템을 종료한다.
DsmAlloc(sz)	sz 크기 만큼의 공유메모리를 할당한다.
DsmAllocAt(sz, pid)	sz 크기 만큼의 공유메모리를 주어진 프로세스(pid)에 할당한다.
DsmAllocBlock(sz, blksz)	sz 크기 만큼의 공유메모리를 할당하는데, 프로세스들을 번갈아가며 blksz 만큼 할당한다.
DsmAllocBlockAt(sz, blksz, pid)	sz 크기 만큼의 공유메모리를 할당하는데, 주어진 프로세스(pid)에서 부터 시작해서 blksz 만큼 프로세스들을 돌아가면서 할당한다.
DsmLock(lockid)	주어진 lock에 대해서 lock을 얻는다(acquire).
DsmUnlock(lockid)	주어진 lock에 대해서 lock을 해제한다(release).
DsmBarrier(barrid)	전역적인 동기화를 위해서 barrier 연산을 수행한다.
DsmWriteFlag(flagid, val)	플래그 값을 설정한다.
DsmWaitFlag(flagid, val)	플래그 값이 설정될 때까지 대기한다.
DsmGetPid()	응용 프로그램이 실행되고 있는 DSM 시스템의 프로세스 번호를 알아낸다. 프로세스의 번호는 .config 파일에 기록되어 있는 순서로써 0부터 N-1 까지 할당된다.
DsmGetProcNum()	DSM 시스템에서 실행 중인 프로세스 수를 알아낸다. 이것은 .config 파일에 기록되어 있는 호스트 수와 일치한다.
DsmGetNodeNum()	DSM 시스템이 운용되는 전체 노드의 갯수를 반환한다. (KDSM은 한 노드에 여러 프로세스를 실행할 수 있기 때문에 노드 수는 프로세스 수와 다를 수 있다).

KDSM 라이브러리를 사용하기 위해서는 dsm.h 파일을 프로그램내에 포함해야 한다. 그림 2는 KDSM 라이브러리를 이용하는 일반적인 프로그램의 구조를 나타낸다. 각 프로세스는 제일먼저 DsmInit(argc, argv)를 호

출하여 KDSM 라이브러리를 사용하기 위한 내부 초기화 과정을 거친다. 그 다음으로는 DsmAlloc(), DsmAllockAt(), DsmAllocBlock(), DsmAllocBlockAt() 등을 이용하여 공유메모리를 할당한다. 공유메모리를 할당한 후에는 반드시 DsmBarrier() 를 호출하여 모든 프로세스가 공유메모리 할당이 끝났음을 보장해야 한다. Worker() 루틴은 실제로 공유메모리를 이용하여 동작하는 부분이다. 각 프로세스의 Worker() 루틴은 DsmLock(), DsmUnlock(), DsmBarrier() 등을 이용하여 공유메모리 상에서 상호 협력하에 동작한다. Worker() 루틴이 끝난 후에는 DsmExit()를 호출하여 모든 Worker() 프로세스가 끝나는 것을 기다린 후 종료한다.

```
#include "dsm.h"

char *ptr1, *ptr2;

void Worker()
{
 /* shared memory program using various synchronization
    primitives like DsmLock(), DsmUnlock(), DsmBarrier()
  */
}

int main(int argc, char* argv[])
{
   /* initialize KDSM */
   DsmInit(argc, arg);
   /* allocate shared memory */
   ptr1 = DsmAlloc(size1);
   ptr2 = DsmAllocAt(size2, pid);
   ...
   /* synchronize for memory allocation */
   DsmBarrier(0);

   /* start application code */
   Worker();

   /* terminate KDSM */
   DsmExit();
   return 0;
}
```

그림 2 KDSM 응용 프로그램의 일반적인 구조

3. 메모리 관리

KDSM 시스템은 NUMA(Non-Uniform Memory Access) 형태의 메모리 구조를 가진다 (그림 3). 공유메모리 주소 공간은 페이지 단위로 분할되고, 각 공유페이지는 미리 지정된 홈 노드를 가진다. 또한 각 노드들은 원거리 페이지를 저장하기 위해 소프트웨어 캐쉬를 가진다. 홈 노드에 있는 페이지에 대한 참조는 지역적으로 해결되나, 원거리 페이지에 대한 참조는 폴트가 발생하여 홈 노드로부터 페이지를 읽어 와서 캐쉬에 저장한다. 캐쉬된 페이지들은 다음과 같은 네 가지 상태 중 하나를 가진다: RO(읽기전용 상태), RW(읽기쓰기가 가능한 상태), INV(무효화 상태), UNMAP(메모리 맵핑이 안 된 상태).

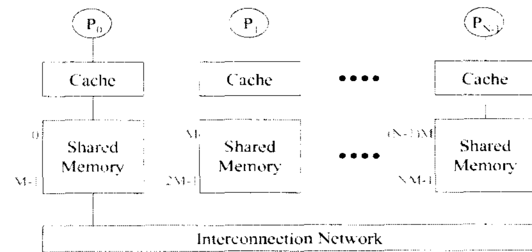


그림 3 KDSM의 메모리 구조

KDSM 시스템은 mmap() 시스템 호출을 사용해서 공유메모리의 주소 공간을 특수 화일인 /dev/zero로 맵핑함으로써 이루어진다. /dev/zero 화일에 맵핑하는 것은 맵핑된 길이만큼의 길이를 가지면서 0으로 초기화된 메모리 영역를 생성한다. STARTADDR는 전체 주소 공간에서 공유메모리로 사용할 부분의 시작 주소이다. GlobalOffset은 현재까지 할당된 공유메모리 양을 표시한다. 공유 페이지는 초기에 홈 노드에 읽기전용 상태로 맵핑되는데, 이것은 페이지에 쓰기를 했을 때 그것을 검출하기 위해서이다.

```
fd = open("/dev/zero", O_RDWR, 0);
...
mmap(STARTADDR + GlobalOffset, len, PROT_READ, MAP_PRIVATE | MAP_FIXED, fd, 0);
```

그림 4 공유메모리 맵핑 방법

공유메모리를 유지하기 위해서 다음과 같은 자료 구조가 사용된다:

- gPage[MAXLOCALPAGE]

홈에 할당된 페이지에 관한 정보를 유지한다: 페이지 주소, 페이지가 수정되었는 지를 표시하는 플래그, diff 버전.

- gCache[MAXCACHEPAGE]

원거리 페이지들에 관한 정보를 유지한다: 페이지 주소, twin을 가르키는 포인터, 캐쉬 상태, write-notice 버전 정보, diff 버전 정보.

- gGPage[MAXGLOBALPAGE]

전체 공유메모리 페이지에 대한 정보를 유지하며, 페이지마다 다음의 정보를 가진다: 페이지의 홈 프로세스

식별자(pid), 홈 페이지에 대한 색인, 캐쉬 페이지에 대한 색인, 페이지가 수정되었는지를 표시 플래그.

그림 5는 SIGSEGV 시그널 처리 알고리즘을 의사 코드 형식으로 보여준다. SIGSEGV 시그널이 발생하면 등록된 시그널 핸들러가 불려지고, 시그널 핸들러는 파라미터로 넘어 온 값을 분석해서 폴트가 난 페이지의 주소를 알아낸다. SIGSEGV 시그널은 두 가지 경우에 발생한다.

- 현재 사용할 수 없는 원거리 페이지를 참조했을 때

시그널 핸들러는 폴트난 페이지를 저장할 공간을 캐쉬에 할당한 후 메모리 맵핑한다. 폴트난 페이지의 홈에 GETP 메시지를 전송해서 페이지를 요청한다. GETP GRANT 메시지가 도착할 때까지 대기하다가, 메시지가 오면 폴트 페이지를 캐쉬에 할당된 페이지로 복사한다. 캐쉬 상태는 접근 형태에 따라 RO 또는 RW로 기록된다.

- 읽기전용으로 지정된 메모리 영역에 쓰기를 시도했을 때

홈 페이지에 대한 쓰기라면 페이지가 수정되었음을 표시한 후, 페이지 보호 모드를 읽고쓰기 모드로 바꾼다. 원거리 페이지에 대한 쓰기라면 페이지가 수정되었음을 표시한 후, 페이지 보호 모드를 읽고쓰기 모드로 바꾼다. 또한 나중에 diff를 생성하기 위해서 twin 생성하고, 캐쉬 상태를 RW로 바꾼다.

```
Input: address of faulted page
Output: none
Algorithm:
  if (write fault on local RO page)
     set page protection to (PROT_READ|PROT_WRITE);
  else // remote fault
  {
    if (write fault on RO page) {
      set page to (PROT_READ|PROT_WRITE);
      create twin and set cache state to RW;
    }
    else {
      if (read or write fault on INV page)
         set page protection to (PROT_READ|PROT_WRITE);
      else // read or write fault on UNMAP page
         map the faulted page using mmap()
                  with (PROT_READ|PROT_WRITE) mode;
      send a GETP message to home of the page;
      if (write fault) {
        create twin and set cache state to RW;
        wait until GETPGRANT is received;
        copy the page content into cache;
      }
      else { // read fault
        set cache state to RO;
        wait until GETPGRANT is received;
        copy the page content into cache;
        set protection mode of the page to PROT_READ;
      }
    }
  }
```

그림 5 SIGSEGV 시그널을 통한 페이지 폴트 처리 알고리즘

4. 통신 구조

KDSM의 통신 모듈은 TCP/IP 상에서 구현되었고, 통신 모듈 초기화 단계에서 모든 프로세스 사이에 TCP 커넥션을 만든다. 각 프로세스는 다른 프로세스에 대한 서버 역할을 하는 동시에 다른 프로세스에게 서비스를 요청하는 클라이언트의 역할을 수행하기 때문에 완전 그래프(complete graph) 형태의 커넥션을 형성한다. 그림 6은 프로세스 간의 커넥션을 초기화하는 함수인 InitComm() 함수를 보여준다.

```
create server socket;
install SIGIO signal handler;
for (process p in all processes except for itself)
  connect to p;
```

그림 6 InitComm() 함수

모든 메시지는 공통 헤더 부분과 각 메시지 별로 다른 데이타 부분으로 구성되고, Send(TMsg *msg) 함수를 통해 전송된다. 메시지는 TMsg 타입으로 표현되고, 그림 7과 같은 필드를 가진다. 메시지를 받는 프로세스는, 미리 정해진 포트로 메시지가 도착했을 때 운영체제가 생성해주는 SIGIO 시그날을 핸들링하여 비동기적으로 메시지를 처리한다.

```
typedef struct {
  int type; // message type
  int src;  // source pid
  int dest; // destination pid
  int size; // data size
  char data[MSG_MAXSIZE]; // message data
} TMsg;
```

그림 7 메시지 포맷

KDSM은 응용 프로그램 코드를 실행하는 중 다른 프로세스로부터 메시지를 받으면 이를 비동기적으로 처리한다. SIGIO 시그널은 새로운 커넥션 요청이 있을때와 메시지가 전송된 경우에 발생한다. 그림 8은 SIGIO 시그널이 발생했을 때, SIGIO 핸들러가 어떻게 처리하는지를 보여준다. MsgServer(TMsg* msg) 함수는 msg의 type 필드에 따라서 적절한 메시지 처리 루틴을 호출한다.

```
disable SIGIO signal;
while (there is something to be served) {
  if (there is a connection request)
    accept the connection;
  for (all connection)
    if (there is a message to be read) {
      read(msg);
      MsgServer(msg);
    }
}
enable SIGIO signal;
```

그림 8 SIGIO 시그널 핸들러

5. HLRC(Home-base Lazy Release Consistency)의 구현

KDSM은 HLRC 메모리 모델을 지원하는데, 페이지 기반 무효화 프로토콜(page-based invalidation protocol), 다중쓰기 프로토콜(multiple writer protocol), 그리고 홈 기반의 프로토콜(home-based protocol)을 사용해서 구현되었다. 홈 기반의 프로토콜에서, 페이지에 가해진 수정은 그 페이지의 홈으로 직접 전달되기 때문에 홈 노드(프로세스)의 페이지들은 항상 최신 정보를 유지한다. 따라서 페이지 폴트가 났을 때 여러 노드가 아닌 홈 노드에서 전체 페이지를 읽어온다.

HLRC는 DSM 응용 프로그램의 수행을 interval들로 구분하는데 한 interval은 release와 다음 release 사이를 말한다. interval은 release 마다 증가하며 interval 내에서의 모든 변경된 페이지에 대한 정보는 특정한 테이블에 기록한다. HLRC에서 여러 프로세스간의 메모리 일관성은 interval을 기본 단위로 하여 이루어진다. 일관성의 유지를 위해 각 프로세스는 각각 독립적으로 vector timestamp를 유지한다. vector timestamp는 어떤 프로세스가 알고 있는 다른 프로세스의 최신의 interval 값의 리스트이며 이 값을 통해 다른 프로세서의 어느 시간까지의 변경사항을 알고 있는지를 알 수 있다.

5.1 상호배제(Mutual Exclusion)

상호배제는 토큰을 기본으로 하는 분산 lock 모델을 사용하였다. 이 방법에서 lock을 acquire하기 위해서는 lock 토큰을 받아야 한다. 어떤 프로세스가 n번재로 lock을 요청했을때 만일 lock 토큰을 가지고 있지 않다면 n-1 번째로lock을 요청한 프로세스에게서 lock 토큰을 받아온다. 여기에서 n-1번째 프로세스가 어떠한 것인가는 지정된 manager를 두어 관리한다. 이와 같은 처리를 위해서 각 프로세스는 *local, held, saved*의 세 필드(field) 를 유지한다. *local*은 lock 토큰을 가지고 있는지의 여부를 나타내며, *held*는 lock 토큰을 사용하고 있는지의 여부를, 그리고 *saved*는 아직 처리하지 못한 포워딩된 lock 요청 메시지를 저장하는데 쓰인다.

그림 9는 P1이 lock을 사용하고 있는 상태에서 P1이 lock을 release 하고 P2가 lock 요청을 하였을때 전송되는 메시지와 각 프로세스의 lock과 관련된 필드들의 내용을 보여주고 있다.

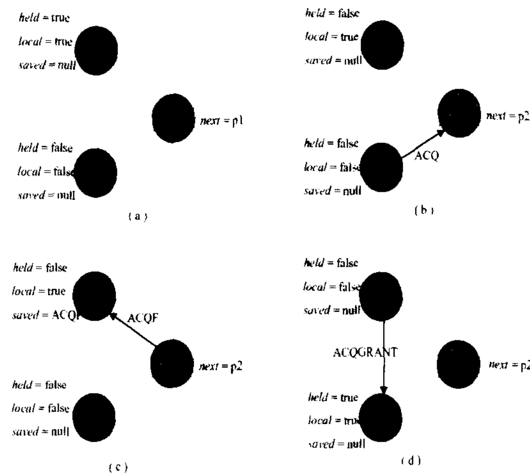


그림 9 분산 Lock 관리

5.2 메모리 일관성(Memory Consistency)

메모리 일관성 정보는 diff와 write-notice 정보이다. diff 는 한 인터벌 동안 특정 페이지가 변경된 내용으로 lock acquire, release 시에 해당 페이지의 홈으로 전송되어 홈이 항상 페이지에 대한 최신의 정보를 유지하도록 한다. write-notice는 페이지가 변경되었는지의 유무를 알려주는 것으로 lock을 release하는 프로세스에서 lock을 acquire하는 프로세스에게로 lock grant메시지를 통해 전달된다. write-notice는 interval 단위로 저장되므로 write-notice를 전송할때 어느 interval까지를 전송하는가를 알아야 할 필요가 있다. 이것을 결정하는 것은 vector timestamp의 비교에 의해 이루어진다. write-notice를 받은 프로세스는 해당 페이지들을 무효화 시키고 자신의 vector timestamp 갱신한다. 추후 무효화된 페이지에 대해 접근하려고 하면 페이지 폴트가 발생하여 페이지의 홈으로부터 페이지 전체를 읽어온다.

그림 10은 HLRC 프로토콜에서의 write-notice 정보의 흐름의 예를 보여준다. 이 그림은 P_0, P_1, P_2가 차례로 lock을 얻고 페이지 X, Y, Z를 수정하고 마지막으로 다시 P_0이 lock을 얻어 X, Y, Z를 읽었을 때

write-notice 정보가 어떻게 흘러가는지를 보여준다. 그림에서 앞절에서 설명한 lock manager를 통해 lock 요청이 포워딩되는 과정은 생략하였다. ACQ 메시지의 괄호안의 숫자는 자신의 vector timestamp를 나타낸다. ACQGRANT 메시지는 write-notice 정보를 포함하는데 write-notice는 (프로세스, interval, 수정된 페이지 리스트)와 같은 형태로 나타내었다. 표 2는 그림 10의 각 위치에서 해당 프로세스의 vector timestamp 값을 보여준다.

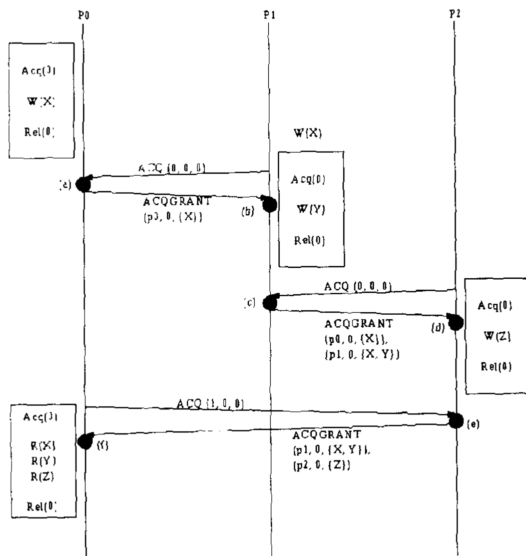


그림 10

표 2 write-notice 정보의 흐름

위치	프로세스	vector timestamp
(a)	P0	(1,0,0)
(b)	P1	(1,0,0)
(c)	P1	(1,1,0)
(d)	P2	(1,1,0)
(e)	P2	(1,1,1)
(f)	P0	(1,0,0)

5.3 Barrier의 구현

barrier는 release와 acquire로 생각할 수 있다. 즉 barrier에 도착하는 것은 lock release, barrier를 떠나는 것은 lock acquire로 생각할 수 있다. barrier 도착 시 고정된 barrier 서버에게 자신의 vector timestamp 와 write-notices를 보내면 barrier 서버는 모든 프로세스에게 이를 종합하여 각 requester의 vector timestamp와 비교하여 write-notices를 보내준다.

6. 성능 측정 결과

KDSM의 기본적인 성능을 측정하기 위해서 8대의 PC로 구성된 Linux 클러스터 상에서 실험을 하였다. 각 PC는 Pentium III 500 MHz CPU와 256 MB의 메인 메모리로 가지며, 100 Mbps Switched Fast Ethernet 연결되었다.

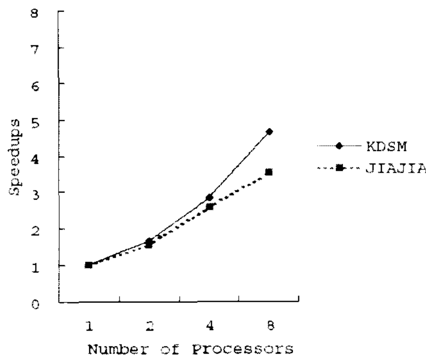
성능 측정에 사용된 응용은 SPLASH2[8] 응용 중 Water와 LU, Rice 대학에서 개발된 SOR와 TSP를 선택하였다. LU는 1024x1024 행렬을 LU decomposition을 통해 lower triangular matrix와 upper triangular matrix의 곱으로 나누는 일을 한다. Water는 N-body 문제를 통해 1728 개의 물분자의 운동을 실험한다. SOR는 Red-Black Successive Over-Relaxation 방법을 통해서 편미분 방정식을 푸는 응용으로, 1024x1024 데이타를 사용하였다. TSP는 branch and bound 알고리즘을 사용해서 20 개의 도시를 방문하는 Traveling Sales Person 문제를 해결한다.

KDSM의 성능을 다른 소프트웨어 분산공유메모리 시스템과 수치적으로 비교하기 위해서, Chinese Academy of Science에서 개발한 JIAJIA[4]를 비교 대상으로 삼았다. JIAJIA는 기본적으로 페이지 기반 무효화 프로토콜, 다중 기록자 프로토콜, 홈 기반 프로토콜을 쓴다는 점에서 KDSM과 비슷하지만, 메모리 일관성 모델이 Scope Consistency[6] 모델을 사용한다는 점에서 차이를 보인다.

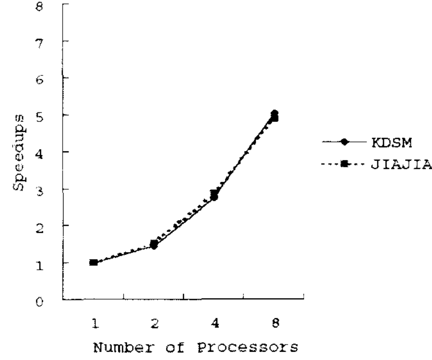
표 3은 각 응용에 대해서 프로세서의 수를 1개, 2개, 4개, 8개씩 사용했을 때 걸린 실행 시간을 보여준다. 그림 11은 각 응용에 대해서 얻을 수 있는 속도 향상을 보여준다. 4개의 응용 프로그램 중에서 SOR와 Water의 경우에는 KDSM과 JIAJIA의 성능 차이가 거의 없음을 알 수 있다. LU와 TSP의 경우, 프로세서 수가 증가함에 따라서 KDSM이 JIAJIA 보다 성능이 뛰어남을 볼 수 있는데, 이것은 크게 두가지 구현상의 차이점 때문이다.

표 3 실행 시간 (단위: 초)

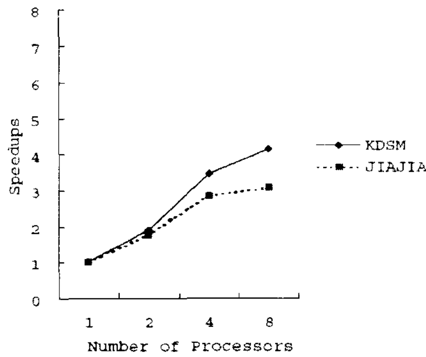
응용 프로그램	DSM 시스템	프로세서 수 1	2	4	8
LU 1024×1024	KDSM	15.0	9.0	5.3	3.3
	JIAJIA	15.0	9.7	5.8	4.3
SOR 1024×1024	KDSM	41.4	27.3	14.6	8.4
	JIAJIA	41.7	29.2	15.2	8.4
TSP 20 cities	KDSM	29.8	15.7	8.7	7.3
	JIAJIA	27.2	15.4	9.5	8.9
WATER 1728 moles	KDSM	169.7	87.7	45.3	23.6
	JIAJIA	168.7	87.6	45.6	24.6



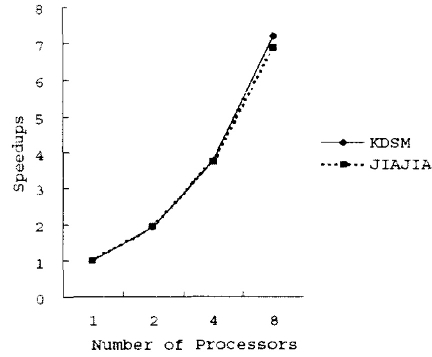
(a) LU 1024×1024



(b) SOR 1024×1024



(c) TSP 20 cities



(d) WATER 1728 moles

그림 11 속도 향상

첫째, 락의 처리시에 해당 락에 의해 보호되는 구간(HLRC에서는 interval, Scope Consistency에서는 scope) 동안 수정된 페이지들에 대해서 DIFF 메시지를 보내고 그 상태를 읽기전용인 RO로 바꾸어야 한다. 이때 JIAJIA에서는 전체 지역 페이지 집합 및 캐쉬 페이지 집합에 대해서 스캔을 하여 RW 상태에 있는 페이지들을 식별해 낸다. 하지만 KDSM에서는 구간 내에서 수정된 페이지들을 저장하기 위한 리스트를 유지하고, 어떤 페이지에 처음 쓰기 접근이 일어날 때 페이지 폴트 핸들러 내에서 해당 페이지를 리스트에 추가한다. 나중에 락 처리시 리스트에 있는 페이지들에 대해서만 DIFF 메시지를 보내고 상태를 RO로 바꾸면 된다. 따라서 KDSM은 락 처리시에 페이지 집합을 스캔할 필요가 없다. 이것은 락 처리 뿐만 아니라 배리어 처리에 똑같이 적용되며, 실제 배리어에 의해 보호되는 페이지들이 일반적으로 더 많기 때문에 배리어 처리시에 KDSM이 JIAJIA에 비해 성능이 더 우수하게 된다.

둘째, KDSM은 Iftode에 의해 제안된 페이지 버전 기법[9]을 이용해서 불필요한 페이지 전송을 막고 DIFF 메시지를 보낸 후 DIFFGRANT 메시지를 기다리지 않는다. 하지만 JIAJIA는 DIFF 메시지 전송 후 항상 DIFFGRANT를 받을 때까지 기다리기 때문에 락과 배리어 처리시 대기 시간이 증가하게 된다.

7. 현재 진행 상황 및 향후 연구

본 논문에서는 KDSM 시스템의 설계 및 구현에 관한

자세한 사항들을 설명하였다. 현재 KDSM은 사용자수준 통신계층을 통해 성능 향상을 꾀하기 위해서 VIA (Virtual Interface Architecture)[10]를 사용해서 구현중에 있다. 실제 구현에 사용된 VIA 라이브러리는 KAIST 컴퓨터구조 연구실에서 개발된 KVIA(KAIST Virtual Interface Architecture)[11]를 이용하고 있다. 또한 SMP 기계들을 활용하기 위해서 한 노드 내에서 실행되는 프로세스들 간의 메모리 공유를 통해서 성능을 높이는 기법에 대한 연구도 진행 중에 있다.

참 고 문 헌

[1] K. Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale University, Department of Computer Science, September 1986.

[2] C. Amza, S. Dwarkadas, P. Keleher, A. L. Cox, and Z. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. IEEE Computer, 29(2), February 1996.

[3] J .B. Carter, J. K. Bennette, and W. Zwaenepoel. Implementation and Performance of Munin. In Proceedings of the 13th SOSP, February 1991.

[4] W. Hu, W. Shi, and Z. Tang. The JIAJIA Software DSM System. Technical report, Institute of Computing Technology, Chinese Academy of Sciences, February 1998.

[5] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In Proceedings of USENIX OSDI, October 1996.

[6] L. Iftode, J. P. Singh and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures, June 1996.

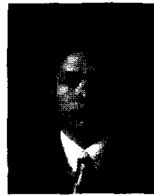
[7] K. Gharachorloo, D. Lenoski, P. Gibbons, A. Gupta and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In Proceedings of the 17th ISCA, June 1990.

[8] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 22nd ISCA, May 1995.

[9] L. Iftode. Home-Based Shared Virtual Memory. PhD thesis, Princeton University, Department of Computer Science, June 1998.

[10] Compaq and Intel and Microsoft Corporation. Virtual Interface Architecture Specification. Version 1.0, December 1997.

[11] Junglok Yu, Moonsang Lee, Seungryoul Maeng. An Efficient Implementation of Virtual Interface Architecture Using Adaptive Transfer Mechanism on Myrinet. In Proceedings of International Conference on Parallel and Distributed System, June 2001.



이 상 권

1996년 부산대학교 전자계산학과 학사. 1998년 한국과학기술원 전산학과 석사. 1998년 ~ 현재 한국과학기술원 전산학과 박사과정 재학중. 관심분야는 Software Distributed Shared Memory, Parallel Processing, Computer Architecture, Operating System

윤 희 철

1999년 한국과학기술원 전자전산학과 전산학전공 학사. 2001년 한국과학기술원 전자전산학과 전산학전공 석사. 2001년 ~ 현재 한국전자통신연구원 컴퓨터 소프트웨어 기술연구소 연구원. 관심분야는 Software Distributed Shared Memory, Parallel Processing, Computer Architecture, Operating System

이 준 원

1983년 서울대학교 계산통계학과 졸업(학사). 1990년 Georgia Tech. 전산학과(석사). 1991년 Georgia Tech. 전산학과(박사). 1983년 ~ 1986년 (주)유공 근무. 1991년 ~ 1992년 IBM 근무. 1992년 ~ 현재 한국과학기술원 전산학과 부교수. 관심분야는 운영체제, 병렬처리 등

맹 승 렬

1977년 서울대학교 공과대학 전자공학과 졸업. 1979년 한국과학기술원 전산학과 석사학위 취득. 1984년 한국과학기술원 전산학과 박사학위 취득. 1984년 ~ 현재 한국과학기술원 전산학과 정교수. 1988년 ~ 1989년 펜실바니아대학 교환교수. 1994년 ~ 1995년 텍사스대학 교환교수. 관심분야는 parallel computer architecture, dataflow machines, vision architecture, multimedia