SMP 클러스터를 위한 소프트웨어 분산 공유메모리의 구현 및 성능 측정

(Implementation and Performance Evaluation of Software Distributed Shared Memory for SMP Clusters)

이 동 현 † 이 상 권 † 박 소 연 †† 맹 승 렬 †††
(Dong-Hyun Lee) (Sang-Kwon Lee) (Soyeon Park) (Seungryoul Maeng)

요 약 가격대비 성능이 좋은 저가의 상업용 SMP가 클러스터 시스템의 노드로 많이 사용되고 있다. 본 논문에서는 이러한 SMP 클러스터 상에서 KDSM을 확장해 소프트웨어 분산공유메모리를 구현하고 성능을 평가하였다. 본 논문의 SDSM 시스템은 HLRC 메모리 모델을 제공한다. 또한 같은 SMP 노드내에서 실행되는 프로세스 간에는 메모리 공유를 통해 페이지 획득 및 메시지 전달을 줄여 성능을 향상시켰다. 100Mbps Fast Ehternet으로 연결된 8노드의 2-way 펜티엄-III SMP 클러스터 상에서 구현되었고 통신 계층은 TCP/IP를 사용한다. 8개의 응용프로그램을 실행시켜 얻은 성능 평가에서는 기존의 단일프로세스 프로토콜과 비교해 최대 33%의 성능 향상과 13%-52%의 페이지 획득 감소가 나타났다.

키워드 : 소프트웨어 분산공유메모리, 공유가상메모리

Abstract Low-cost commodity SMP(Symmetric Multiprocessor) is widely used as a node of cluster system. In this paper, we implement and evaluate the performance of SDSM system for SMP clusters. Our SDSM system provides HLRC(Home-based Lazy Release Consistency) memory consistency model. Our protocol utilize shared memory within same SMP node, so that page fetch and message passing through network can be reduced. It is implemented on 8 node of 2-way Pentium-III SMP interconnected with 100Mbps Fast Ethernet, and uses TCP/IP for transport/network layer protocol. The experiment with eight applications shows that our SMP protocol achieves maximum 33% speedup improvement and 13%-52% reduction of page fetch compared with uniprocessor protocol.

Key words : Software Distributed Shared Memory, Shared Virtual Memory

1. 서 론

최근들어 대량생산되는 저가의 상업용 PC나 워크스테이션의 성능이 향상되고 고속 네트워크의 등장에 따라 NOW(Networks of Workstations)와 같은 클러스터 시스템을 이용하여 고성능 병렬 컴퓨터를 구성하려는 연구가 활발히 진행중이다. 또한 단일 프로세서에 비해 가격대비 성능이 좋은 소규모 SMP(Symmetric Multiprocessors) 시스템을 클러스터의 노드로 많이 사용한다.

소프트웨어 분산공유메모리(SDSM : Software Distributed Shared Memory)[1] 시스템은 특별한 하드웨어의 지원없이 공유메모리 모델을 제공할 수 있기 때문에 물리적으로 메모리가 분산되어 있는 클러스터 환경에 효과적으로 사용될 수 있다. LRC(Lazy Release Consistency)[2]와 HLRC(Home based LRC)[3]는 SDSM 에서 대표적인 완화된 일관성 메모리 모델이다. LRC와 HLRC는 모두 다중 기록자 프로토콜을 채용하는데 이것은 쓰기 가능한 페이지가 동시에 여러 개 존재할 수 있고, twin과 diff를 이용해 공유메모리상의 변경내용을 찾아서 적용한다. HLRC가 LRC와 다른 점은 모든 공유 페이지에 지정된 홈 프로세스가 존재하여 항

· 본 연구는 국가지정연구실 사업의 지원을 받았습니다.
† 비 회 원 : 한국과학기술원 전자전산학과
donghyun@kt.co.kr
sklee@camars.kaist.ac.kr
†† 학생회원 : 한국과학기술원 전자전산학과
sypark@camars.kaist.ac.kr
††† 종신회원 : 한국과학기술원 전자전산학과 교수
maeng@camars.kaist.ac.kr
논문접수 : 2002년 6월 24일
심사완료 : 2003년 3월 31일

상 최신의 내용을 가지고 있다는 점이다.[1)]

멀티프로세서 노드 내에서는 하드웨어적인 공유를 하고 노드간에는 소프트웨어적으로 공유메모리를 제공하는 것이 국부성(locality)이 높고 적은 동기화가 이루어지는 응용 프로그램에서 하드웨어 분산 공유메모리에 근접하는 성능이 나타났다[4]. 이에 따라, 기존의 SDSM 시스템 프로토콜을 SMP 클러스터 환경에 적합한 프로토콜로 확장하는 연구가 많이 진행되고 있다[5,6,7,8]. SMP 시스템은 노드 내의 프로세서들이 메모리를 공유하고 그들간의 캐쉬 일관성은 하드웨어적으로 지원된다. 따라서 노드 내의 프로세서들 간에 메모리를 통해 통신이 가능하다는 장점을 가진다. 이러한 장점을 이용하기 위해서는 SDSM 시스템의 통신 구조가 노드 내의 통신과, 노드와 노드 간의 통신의 두 단계로 구성되어야 한다. 그리고 노드 내에서 응용 프로그램 및 프로토콜 데이타의 공유 메커니즘이 필요하다.

본 논문에서는 SMP 클러스터를 위한 HLRC 프로토콜을 설계, 구현하고 그 성능을 분석하였다. 멀티프로세스 모델을 사용하였고 노드 내의 프로세스들은 모든 응용 프로그램 데이타와 twin, write notice 테이블 등의 프로토콜 데이타를 서로 공유한다. 서로 다른 노드간의 통신은 TCP/IP를 사용하여 네트워크를 통한 메시지 전달로 이루어지지만, 같은 노드 내에서의 통신은 공유메모리를 통해 이루어진다. 그리고 같은 노드 내에서는 소프트웨어의 간섭 없이 직접 응용 프로그램 데이타의 공유가 가능하므로 원거리 노드로부터 페이지를 가져오는 횟수가 줄어든다. 결과적으로 네트워크를 통한 데이타 전달량이 줄고 통신 지연 시간이 줄어들어 전체 성능이 향상된다. 실제 구현에서는 운영체제의 제한으로 노드 내에서 공유될 수 있는 자료구조의 크기가 한정되기 때문에 프로토콜 데이타의 크기를 줄여야 할 필요가 있다. 이를 해결하기 위해 write notice 테이블의 재사용 방법을 제안하고 적용하였다.

성능 측정을 위한 하드웨어 환경은 100Mbps Switched Fast Ehternet으로 연결된 8노드 클러스터이며, 각 노드는 2개의 Pentium-III 500MHz 프로세서와 512MB의 메인 메모리로 구성되어 있다. 운영체제는 Linux를 사용하며 KDSM(KAIST Distributed Shared Memory)[9,10]의 HLRC 프로토콜 버전 위에서 구현되었고, 8개의 응용 프로그램을 수행하여 성능을 측정하였다. 성능 측정 결과 SMP 노드 내에서 메모리 공유를 통해 단일 프로세서 프로토콜에 비해 최대 33%향상을 보였다. 또한 네트워크를 통한 원격 페이지 전달의 횟수와 프로토콜 메시지 수가 최대 60% 줄어들었고, 이는 메모리 액세스의 국지성이 높은 응용 프로그램에서 특히 감소의 폭이 컸다. 이를 통해 일반적으로 많이 사용되는 2-way SMP에서도 성능 향상이 충분히 있음을 확인할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 구현에서 제공하는 HLRC에 대해 살펴본다. 3장에서는 구현의 기반이 된 KDSM에 대해 살펴보고, 본 논문에서 제안하고 구현된 SMP 클러스터를 위한 프로토콜에 대해 상세히 기술한다. 4장에서는 성능 측정 결과 및 응용프로그램의 특성에 따른 SMP 프로토콜의 분석을 기술한다. 5장에서는 관련 연구로 기존의 SMP 클러스터 상에서의 SDSM 시스템에 대해 살펴본다. 마지막으로 6장에서 결론을 맺는다.

2. 배경 지식

이 장에서는 SDSM에서 대표적인 메모리 일관성 모델이고 본 논문에서 기반으로 하고 있는 HLRC (Home-based Lazy Release Consistency)에 대해 설명한다.

2.1 HLRC

HLRC[3] 메모리 일관성 모델은 기본적으로 RC(Release Consistency)[11]의 변형이다. HLRC는 응용 프로그램 수행을 interval 단위로 구분한다. Interval은 acquire나 release같은 동기화 연산으로 구분되고, 동기화 연산시마다 증가한다.

메모리 일관성 유지를 위해 한 interval 내에서 변경된 공유 페이지들의 정보를 따로 기록해 두는데 이를 write-notice라 한다. 이 write-notice는 동기화 연산시에 다른 프로세스로 전달되어 변경된 페이지 정보를 알 수 있게 한다. 그리고 write-notice를 전역적인 write-notice 테이블이라는 자료 구조에 저장을 해 두어 테이블의 참조만으로 특정 프로세스의 특정 interval동안의 변경 사항을 알 수 있다.

또한 일관성 유지를 위해 각 프로세스는 독립적으로 벡터 타임스탬프(vector timestamp)를 유지한다. 벡터 타임스탬프는 해당 프로세스가 알고 있는 다른 프로세스의 최신의 interval 값들의 리스트이며, 이 벡터 타임스탬프를 통해 다른 프로세스의 어느 interval까지의 수정된 정보를 가지고 있는지를 알 수 있다.

HLRC는 다중 기록자 프로토콜(multiple writer protocol)이다. 어떤 페이지에 대해 쓰기가 발생했을 때 페

1) 일반적으로 흔 노드란 용어를 사용하나, SMP 노드에서는 여러 프로세스가 존재할 수 있기 때문에 본 논문에서는 용어의 혼란을 막기 위해 흔 노드 대신 흔 프로세스란 용어를 사용한다.

이지의 원본을 따로 복사해 두는데 이를 twin이라고 한다. Release시에 프로세스는 변경된 페이지와 twin을 비교하여 변경 사항을 얻는다. 이를 diff라고 한다. 다른 프로세스가 변경된 페이지에 접근하면 diff를 순서대로 적용하여 변경 사항을 액세스하도록 한다.

HLRC가 LRC[2]와 다른 점은 각 공유메모리 페이지에 지정된 홈 프로세스가 할당되어 있다는 점이다. HLRC에서는 release시 수정된 페이지에 대한 변경 사항, 즉 diff를 홈 프로세스에 적용시켜 홈 프로세스는 항상 최신의 내용을 유지한다. 홈이 아닌 노드에서 페이지 부재(page fault)가 발생하면 단순히 홈으로부터 페이지를 얻어옴으로써 최신의 복사본을 얻는다.

3. SMP 클러스터를 위한 HLRC 프로토콜 구현

이 장에서는 본 논문 구현의 기반이 된 KDSM (KAIST Distributed Shared Memory) 시스템의 HLRC 버전에 대해 살펴보고, 본 논문의 SMP 클러스터를 위한 HLRC 프로토콜의 구현에 대해 자세히 기술한다.

3.1 KDSM

KDSM(KAIST Distributed Shared Memory) 시스템[9,10]은 Linux 2.2.13 상에서 실행되는 사용자수준 라이브러리로 구현되었다. 프로세스 간의 통신은 TCP/IP를 통해서 이루어지고, 비동기 메시지의 처리를 위해서 SIGIO 시그널을 가로채는 방식을 사용한다. KDSM은 페이지 기반 무효화 프로토콜(page-based invalidation protocol), 다중읽기 다중쓰기(multiple reader multiple writer) 프로토콜을 바탕으로 HLRC 프로토콜을 구현한다. 원거리 페이지를 저장하는 캐쉬 페이지들은 다음과 같은 네가지 상태 중 하나를 가진다 : RO(읽기전용 상태), RW(읽기쓰기가 가능한 상태), INV(무효화 상태), 그리고 UNMAP(메모리 맵핑이 안된 상태).

3.2 개요

본 논문에서 구현한 SMP 클러스터를 위한 프로토콜(SMP 프로토콜)은 KDSM의 HLRC버전을 SMP 환경을 이용하도록 확장시킨 것이다. 기본적으로 2 way SMP의 클러스터를 가정하여 구현되었고, 4 way 이상의 SMP 클러스터를 위한 프로토콜로 확장이 가능하다.

SMP 노드를 사용하는 이점으로는 다음과 같은 점들을 들 수 있다.

- SMP 노드 내에서는 프로세스간에 하드웨어 캐쉬 일관성 메커니즘을 사용하기 때문에 일관성 유지를 위한 오버헤드가 줄어든다.
- 같은 노드 내에서는 공유메모리를 사용해 통신하므로 원거리 노드간의 네트워크를 통한 통신에 비해 빠르다.
- 물리적 메모리를 노드 내 프로세스들이 공유하므로 소프트웨어 간섭 없이 메모리가 공유되고, 페이지 요청이나 diff 생성 등이 감소된다.
- 어떤 프로세스가 페이지 폴트가 발생하면 원거리 노드로 부터 페이지를 읽어온다. 이후에 같은 노드의 다른 프로세스가 같은 페이지에 대해서 폴트가 발생하면 이 프로세스는 원거리 노드로 부터 페이지를 읽어 오지 않고, 적절한 프로토콜 처리를 통해서 공유된 페이지를 이용할 수 있다. 따라서 메모리 국부성(locality)이 높은 응용프로그램에서 선인출(prefetch)의 효과가 있다.

A. Bilas의 논문[12]에서 구분한 SMP 노드 내의 공유 모델 중 HLRC의 laziness를 얻을 수 있는 프로세스 모델을 채택해 구현하였다.

3.3 통신 구조

SMP 프로토콜은 같은 노드 내의 프로세스간 통신은 공유된 프로토콜 데이타를 읽고 쓰는 것으로 이루어지고, 원거리 노드의 프로세스와의 통신은 TCP/IP를 사용해 이루어지는 두 단계의 통신 구조를 갖는다.

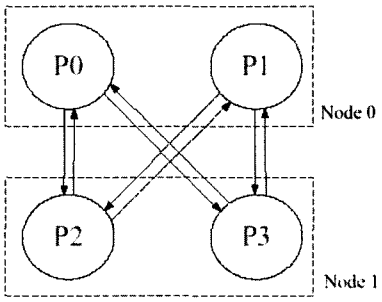


그림 1 SMP 프로토콜에서 프로세스간의 커넥션

응용프로그램의 초기화 단계에서 프로세스간에 소켓 커넥션을 그림 1과 같이 생성한다. 그림 1에서 점선으로 나타낸 사각형은 SMP 노드의 경계를 의미한다.

3.4 공유메모리 관리

SMP 프로토콜 기반 환경에서는 그림 2와 같이 한 노드 내의 프로세스들은 응용프로그램의 공유메모리 및 소프트웨어 캐쉬를 공유한다.

공유메모리 페이지에 대한 정보를 유지하기 위해서 디렉토리 구조를 유지한다. 디렉토리에는 각 페이지의

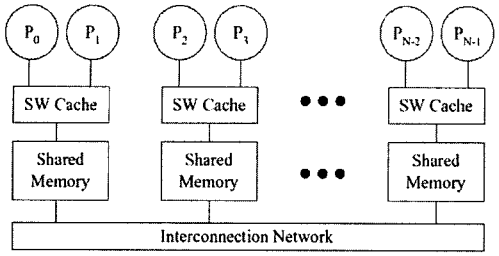


그림 2 메모리 구조

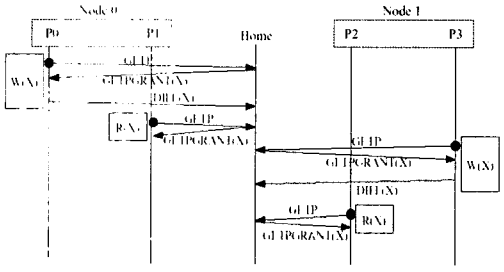
홈 프로세스, 페이지의 상태, twin의 포인터와 페이지 버전을 나타내는 diffver 벡터, write-notice를 받아 요구되는 페이지 버전을 나타내는 wtntver 벡터를 가진다.

공유메모리의 관리는 공유메모리 페이지를 가상 주소 공간에 사상시켜 메모리를 참조할 때 발생하는 SIGSEGV 시그널을 처리해 줌으로써 이루어진다. SIGSEGV 시그널이 발생하면 시그널 핸들러가 불려지고, 시그널 핸들러는 파라미터로 넘어 온 폴트난 주소를 다음과 같이 처리함으로써 분산 공유메모리를 유지한다.

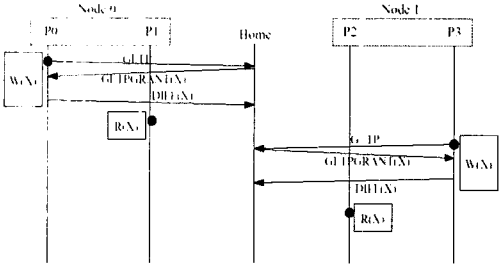
- 현재 사용할 수 없는 메모리를 접근했을 때
  이 경우는 원거리 페이지를 참조했을 때이다. 시그널 핸들러는 자기 노드에 해당 페이지의 캐쉬가 존재하지 않거나 존재하더라도 원하는 새로운 버전이 아닌 경우 폴트난 페이지의 홈 프로세스에게 페이지 요청 메시지를 전송한다. 그리고 요청한 페이지가 도착할 때까지 대기하다가, 메시지가 오면 폴트 페이지를 캐쉬에 할당된 페이지로 읽어 들인다. 캐쉬에는 접근 형태에 따라서 RO 또는 RW 상태를 기록한다. 만약 캐쉬의 twin이 존재한다면 이웃 프로세스(같은 노드 내의 다른 프로세스)에 의해 캐쉬가 읽고쓰기되고 있는 상태이므로 단순히 복사하면 안 된다. 이 때는 전송받은 페이지를 twin과 비교해 diff를 생성하여 twin과 해당 캐쉬에 각각 적용한다. 이렇게 함으로써 이웃 프로세스에 의해 업데이트 된 내용을 잃어버리지 않고 새로운 버전의 페이지를 가져올 수 있다. 이 과정은 Cashmere 2L [6]에 사용된 2-way diffing 방법과 유사하다.
- 읽기 전용으로 지정된 메모리 영역에 대해서 쓰기를 시도했을 때
  만약 지역 페이지에 대한 쓰기라면 페이지가 수정되었음을 표시한 후, 페이지의 보호 모드를 읽고쓰기 모드로 바꾸면 된다. 이웃 페이지(홈 프로세스가 같은 노드 내의 다른 프로세스인 페이지)에 대한 쓰기도 마찬가지로 페이지가 수정되었음을 표시한 후, 페이지의 보호 모드를 읽고쓰기 모드로 바꾼다. 캐쉬에 저장된 원거리 페이지에 대한 쓰기라면 페이지가 수정되었음을 표시한 후, 페이지에 대한 보호 모드를 읽고쓰기 모드로 바꾼다. 또한 나중에 diff를 생성하기 위해서 twin을 생성하고, 캐쉬 상태를 RW로 바꾼다.

그림 3은 기본 HLRC의 프로토콜과 SMP를 위한 프로토콜의 차이점을 나타낸다. 그림 3에서 점선의 사각형은 노드의 경계를 나타낸다. 즉, P0와 P1이 같은 노드에, P2와 P3이 같은 노드 내에 존재한다. Home은 페이지 X의 홈 프로세스이다. 검은 점은 페이지 폴트가 일어나 시그널 핸들러가 실행되는 시점을 나타낸다.

그림 3(a)는 노드 내의 각 프로세스가 읽기, 쓰기 폴트가 일어날 때마다 홈으로부터 페이지를 가져와 프로세스의 메모리 공간에 캐쉬 카피를 만든 후 접근하는 것을 보여준다. 즉, 같은 노드 내의 두 프로세스는 동일한 페이지 X에 대해서 서로 독립된 메모리 카피를 가진다. (b)의 SMP 프로토콜에서는 노드 내의 모든 프로세스들은 캐쉬된 페이지를 공유하기 때문에 P1과 P3이 읽기 폴트를 낸 시점에 이웃 프로세스에 의해 홈으로부터 가져온 최신의 캐쉬된 페이지가 노드 내에 이미 존재한다. 따라서 원거리에 있는 홈으로부터 페이지를 다시 가져올 필요가 없다. 이러한 페이지 공유를 통해서 SMP 프로토콜에서는 페이지 획득의 수가 줄어든다.



(a) 페이지 획득의 HLRC 기본프로토콜



(b) 페이지 획득의 SMP를 위한 프로토콜

그림 3 페이지 획득의 예

3.5 자료 구조

본 논문의 SMP 프로토콜은 프로세스 단위로 interval을 관리한다. 즉, 프로세스가 acquire, release 등의 동기화 작업을 수행할 때마다 interval값이 증가한다. 또한 프로세스들은 독립적으로 벡터 타임스탬프를 가지며, 같은 노드 내의 프로세스들은 서로 벡터 타임스탬프를 공유한다. 같은 노드 내의 프로세스들은 벡터 타임스탬프와 함께 write notice를 기록하는 테이블을 공유한다.

Interval	P0	P1	P2	P3
4				
3	7, 8	4, 5, 6		
2	3, 5	2, 3, 4	1, 2	11
1	1, 3, 5	2, 3, 8	9, 10	8, 10, 12

그림 4 write-notice 테이블의 데이타 구조

그림 4는 write-notice 테이블의 데이타 구조를 표시한 것이다. 각 테이블 박스 안의 숫자는 update-list1로 프로세스가 특정 interval에 업데이트를 한 페이지들의 번호를 나타낸다. 즉, 그림 4에서 프로세스 P0는 interval 2에 페이지 3과 5를 변경했다는 것을 알 수 있다. 또한 lock과 barrier의 프로토콜 자료 구조를 프로세스마다 따로 갖되 서로 공유한다. 그리고 이웃 프로세스의 프로토콜 데이타를 변경하는 것으로 lock과 barrier메시지를 대신하게 된다.

요약하면 본 논문의 SMP 프로토콜에서는 다음과 같은 자료 구조를 공유한다.

- 응용프로그램 데이타
  - 지역 공유메모리 페이지
  - 캐쉬된 페이지
- 프로토콜 데이타 중 노드 내에 한 카피만 존재하면서 프로세스 간의 공유되는 것들
  - 페이지 디렉토리
  - Twin
  - Write-notice 테이블
- 프로토콜 데이타 중 프로세스 단위로 존재하면서 프로세스 간의 공유되는 것들
  - Lock 자료 구조
  - Barrier 자료 구조
  - 벡터 타임스탬프

프로토콜 데이타는 리눅스의 시스템 V IPC 중 공유메모리 함수를 사용하여 공유하였다. 프로세스 간의 공유를 위해서는 두 프로세스 사이에 미리 알려진 주소를 이용해야 한다. 응용프로그램의 데이타의 경우, 모든 프로세스가 동일한 가상 주소를 보기때문에 문제가 없다. 프로토콜 데이타의 경우 주소가 미리 고정되어야 하기 때문에 공유되는 모든 자료 구조는 메모리 할당을 고정으로 받아야 한다(static memory allocation). 또한 프로토콜 데이타가 여러 프로세스 사이에서 공유되어 읽고 쓰이기 때문에 적절한 상호 배제가 필요하다. 이를 위해서 리눅스 커널 SMP 버전에서 제공하는 spinlock을 이용해 공유데이타의 상호 배제를 하였다.

3.6 write-notice 테이블 재사용

프로토콜 자료 구조 중 그림 4에서 나타낸 write-notice 테이블은 정적 크기로 할당될 경우 가장 큰 프로토콜 자료 구조가 되어 물리적 메모리의 크기를 넘어가기 때문에 디스크 접근이 발생할 수 있다. 예를 들어, 한 interval에서 수정될 수 있는 페이지가 최대 4096개라고 하고, 최대 interval이 1024, 최대 프로세스 수가 16개라고 했을 때, write-notice 테이블의 크기는256 MB(=16×1024×4096×4B)를 차지한다. 하지만, 노드의 물리적 메모리가 한계가 있고, 리눅스의 최대 공유메모리 크기가 제한되어 있기 때문에 프로토콜 내부에서 사용하는 자료 구조의 크기를 최대한 줄여 메모리 오버헤드를 줄일 필요가 있다. 따라서 본 논문에서는 write-notice 테이블 크기를 작게 잡고 재사용을 해서 메모리 오버헤드를 줄였다.

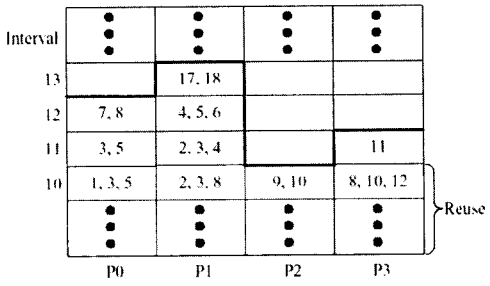


그림 5 write-notice 테이블의 데이타 구조

그림 5는 특정 순간의 write-notice 테이블을 가정한 것이다. 두꺼운 실선은 프로세스의 벡터 타임스탬프를 나타내고 그 값은 (12,13,10,11)이다. 이같은 벡터 타임스탬프를 보면 그 프로세스가 현재 어디까지의 뷰를 보고 있는지 알 수 있다. 만일 모든 프로세스가 interval X이상을 보고 있다면 X이하의 write notice는 필요없는 정보가 된다. 따라서 그 부분은 재사용이 가능하다.

각 프로세스가 다른 프로세스들의 벡터 타임스탬프를 저장하기 때문에 이 작업은 각 프로세스에 대해 지역적으로 수행될 수 있다.

3.7 동기화

Lock의 이동은 같은 노드 내의 이동과 서로 다른 노드간의 이동으로 나누어 생각한다.

- 서로 다른 노드간의 lock의 이동은 기존 프로토콜과 마찬가지로 네트워크를 통한 lock 토큰, 벡터 타임스탬프, write-notice의 전달 메시지를 통해 이루어진다.
- 노드 내 프로세스들이 lock 구조, 벡터 타임스탬프, write-notice 테이블을 공유하고 있기 때문에 같은 노드 내의 lock의 이동은 인터럽트를 발생시키지 않고 공유메모리의 참조만으로 이루어진다.

SMP 프로토콜에서 barrier는 계층적 barrier를 사용하였다. 보통 하나의 barrier 관리자를 두고 모든 프로세스로부터 barrier 요청을 받도록 하는 순차적 방법을 사용한다. 그러나 이는 프로세스 수가 많아지면 barrier 관리자가 병목이 될 수 있다. 계층적 barrier를 사용하면 순차적 방법에 비해 barrier 시간을 단축시킬 수 있다[13]. 본 논문의 구현에서는 각 노드 내의 한 프로세스가 그 노드의 write-notice를 모아서 barrier 관리자에게 보내고, barrier 관리자는 모든 노드로부터 write-notice를 모아 다시 각 노드로 보내면, 각 노드 내의 여러 프로세스들이 그 write-notice를 보고 해당 페이지들을 무효화시키는 방법을 사용하였다.

4. 성능 측정

4.1 플랫폼

성능 측정을 위한 하드웨어 환경은 100Mbps Switched Fast Ehternet으로 연결된 8노드 클러스터이며, 각 노드는 2개의 Pentium-III 500MHz 프로세서와 512MB의 메인 메모리로 구성되어 있다. Linux2.2.16-22smp를 운영체제로 사용한다. 또한 통신 계층으로는 TCP/IP를 사용하였다.

4.2 응용 프로그램

성능 측정을 위해 DSM 벤치마크로 널리 쓰이는 8개의 응용 프로그램을 수행하여 수행 시간 및 메시지 전송량등의 여러 값들을 측정하였다. 사용한 응용 프로그램은 SPLASH 2[14]의 LU, Ocean, Water, Barnes, Radix와 NAS 벤치마크[15]의 3-D FFT, Rice 대학[16]의 TSP, SOR이다. Barnes, 3-D FFT, SOR, TSP, Water의 경우 CVM[17]의 배포판에 포함된 것을 이용하였고, 나머지는 SPLASH2에서 KDSM 용으로 포팅한 것을 이용하였다.

표 1은 각 응용 프로그램의 사용된 문제 크기, 사용된 락과 배리어의 수, 그리고 단일 프로세스에서 실행시킨 시간(Seq. time)을 보여준다.

표 1 응용 프로그램의 특징

Appl.	Size	Barns	Locks	Seq.time
LU	1024×1024	132	0	178.87
3 D FFT	128×128×128	12	0	44.33
Ocean	258×258	901	207	4.71
Water	1728 mols	31	30	169.55
Barnes	8192	24	0	35.46
Radix	8M keys	11	3	6.50
TSP	20 cities	1107	2	134.16
SOR	4096×4096	41	0	27.18

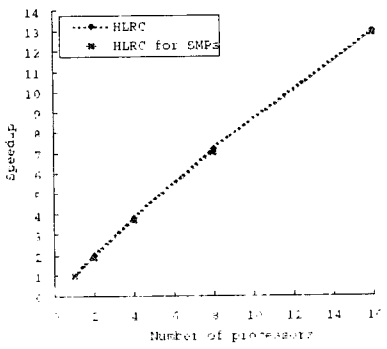
5. 성능 측정 결과

그림 6은 기존의 HLRC와 본 논문의 SMP 프로토콜에 대해 프로세서 수를 1, 2, 4, 8, 16개로 늘려가며 각 응용 프로그램의 속도 향상을 측정한 것이다. 측정 결과 8개의 응용 프로그램 중 Ocean, Barnes, TSP가 SMP 프로토콜에서 성능 향상이 두드러졌고, 8노드, 16 프로세서에서 수행시켰을 때 Ocean은 26%, Barnes는 33%, TSP는 18%의 성능 향상이 있었다. FFT, SOR, Radix는 프로세서의 수에 따라 성능 향상 폭이 달라졌다. LU와 Water는 통신 대 계산 비율이 작기 때문에 SMP 프로토콜의 이점이 적다. 따라서 기존의 HLRC와 성능 차이가 거의 없고, 통신 오버헤드가 작기 때문에 확장성(scalability)이 높게 나타났다.

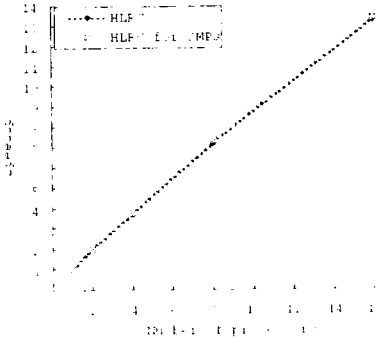
그림 6의 (e),(g),(h)를 보면 SMP 프로토콜에서 프로세서 2개일 때 높은 속도 향상이 나타났는데 이는 하나의 SMP 노드만 사용하기 때문에 네트워크를 통한 통신이 전혀 없어서 높게 나타나는 속도 향상이다.

표 2는 8노드에서 각각의 응용 프로그램을 실행시켰을 때의 페이지 폴트 횟수(page faults), 원거리 노드로부터 페이지를 가져오는 횟수(page fetches), diff 전송량(diff recv)을 비교한 것이다. 페이지 폴트 횟수는 홈이 원거리 노드인 페이지를 접근하는 횟수를 말한다.

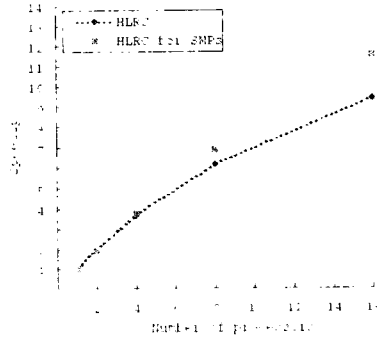
TSP를 제외한 모든 응용 프로그램에서 SMP 프로토콜을 적용시켰을 때 page fault가 줄어들었고, 모든 응용 프로그램에서 page fetches, diff recv가 줄어드는 것을 알 수 있다. 페이지 폴트 횟수는 공간 국부성이 높은 응용 프로그램인 Ocean, SOR 등에서 줄어드는 폭이 크고, Ocean, Barnes, SOR의 경우 페이지 획득 횟수가 크게 줄어들었다. TSP의 경우 페이지 폴트 횟수



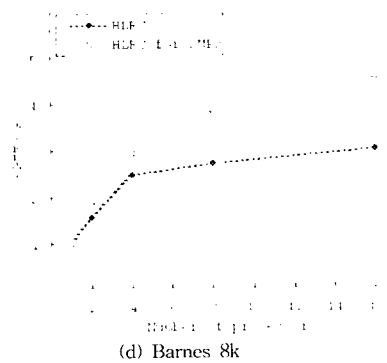
(a) LU 1024×1024 matrix



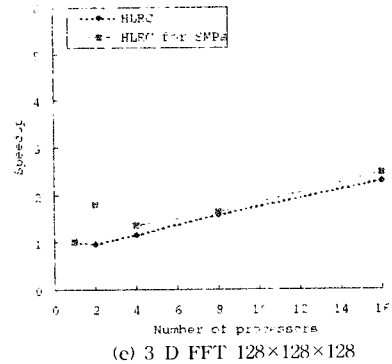
(b) LU 1024×1024 matrix



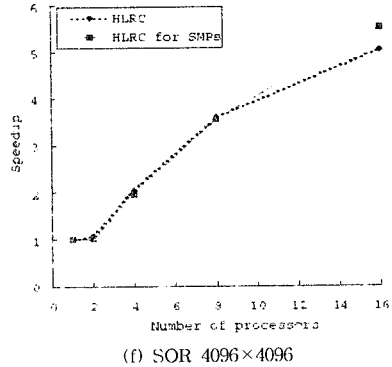
(c) TSP 20 cities



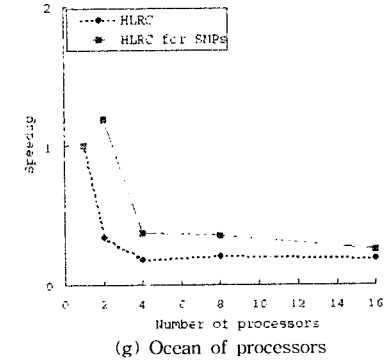
(d) Barnes 8k



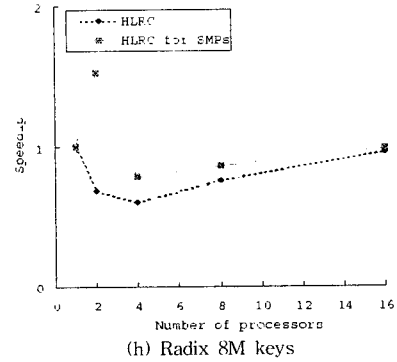
(e) 3 D FFT 128×128×128



(f) SOR 4096×4096



(g) Ocean of processors



(h) Radix 8M keys

그림 6 속도 향상 결과

표 2 페이지, diff 전송량 측정 결과

Application	Page faults		Page fetches		diff recv(kBytes)	
	HLRC	SMP prot.	HLRC	SMP prot.	HLRC	SMP prot.
LU	8552	6333	8552	4084	0	0
FFT	120604	111888	59160	52920	338912	316318
Ocean	210775	178672	119884	56208	188384	175677
Water	9722	8934	6224	3342	7297	6752
Radix	71613	68576	24265	21447	93152	86874
TSP	9338	9521	8029	5771	259	244
SOR	2455	1168	2455	1167	0	0

가 SMP 프로토콜에서 오히려 늘어났는데, TSP의 경우 동기화 수단으로 락만을 사용하고 프로세스들이 락을 획득하는 순서는 매 실행시마다 달라질 수가 있기 때문이다. 즉, 홈이 아닌 프로세스가 락을 자주 얻을 경우 페이지 폴트 횟수가 약간 늘어날 수 있다.

그림 7은 8노드, 16프로세서에서 실행시간 분할 그래프이다. 시간 분할은 계산 시간(Application), 페이지 부재 처리 시간(Page), 락 수행 시간(Lock), 배리어 수행 시간(Barrier), 노드 내의 동기화 시간(Intra-node sync)으로 나누었다. 각각의 실행 시간의 값은 16프로세스의 실행 시간 분할값의 평균을 사용하였다. 각각의 시간값에는 원거리 노드로부터 온 메시지를 처리하는데 걸리는 시간이 포함되어 있다. 원거리 노드에서 온 메시지는 계산 도중이나 락 또는 배리어 대기 도중에 언제라도 SIGIO 시그널이 불려 처리될 수 있다. 따라서 이 메시지를 처리하는 시간은 따로 떼어 측정하기가 불가능하다.

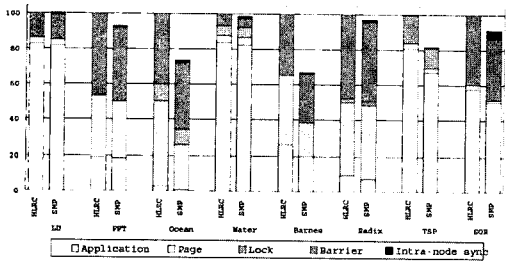


그림 7 실행 시간 분할

표 2에서 나타났듯이 Ocean과 Barnes가 페이지 획득 횟수가 현저하게 줄어들었고 이는 페이지 부재 처리 시간의 감소로 이어졌다. 따라서 Ocean과 Barnes가 SMP 프로토콜을 적용했을 때, 전체 실행시간이 두드러지게 감소하였다.

모든 응용 프로그램에서 계산 시간이 다소 변화했는데 이는 순수 계산 시간이 줄어든 것이 아니라 외부로부터의 메시지를 처리하는 시간이 줄어든 것으로 해석된다. 같은 노드 내에서의 락이나 write-notice 전달은 이웃 프로세스를 방해하지 않고 공유메모리를 통해서 이루어지기 때문에 단일프로세스 프로토콜에서 소비되던 메시지 처리 시간이 많이 줄어들 수 있다. 락을 빈번하게 사용하는 TSP의 경우이 효과가 크게 나타나 전체 실행시간의 감소가 이루어졌다.

LU나 Water와 같이 통신 대 계산 비율이 작은 응용 프로그램은 고유하게 소비되는 계산 시간이 크기 때문에 페이지 획득 횟수나 diff 전달 양이 줄어들어도 전체 실행시간에 큰 영향을 끼치지 못했다.

SMP 프로토콜의 오버헤드라고 할 수 있는 노드 내의 동기화 시간은 같은 노드 내의 프로세스간에 공유하는 프로토콜 데이타의 상호 배제를 하는데 걸리는 시간을 의미한다. 이는 SOR을 제외한 응용 프로그램에서 전체 실행 시간의 1%미만으로 큰 오버헤드가 되지 않았다.

6. 결 론

본 논문에서는 SMP 클러스터를 위한 소프트웨어 분산 공유메모리를 구현하고 실제 SMP 클러스터 상에서 성능을 분석하였다. 본 논문의 소프트웨어 분산 공유메모리 시스템의 특징은 페이지 기반 무효화 프로토콜과 홈 기반 프로토콜을 사용해서 HLRC 메모리 일관성 모델을 지원한다는 것이다. Linux 운영체제 상에서 사용자 수준 프로세스로 구현되었고, 서로 다른 노드 간의 통신은 TCP/IP를 이용하고 같은 노드 내의 통신은 네트워크를 거치지 않고 공유메모리를 이용하는 두 단계 통신 구조를 사용한다.

100 Mbps Fast Ethernet으로 연결된 8노드의 2-way Pentium-III SMP 클러스터 상에서 성능을 측정해 본 결과, 8개의 응용 프로그램 중 6개에서 단일프로세서 프로토콜에 비해 최대 33%성능 향상이 이루어졌고, 통신 대 계산 비율이 작은 2개는 SMP 클러스터를 위한 프로토콜의 이점이 별로 없기 때문에 성능 변

화가 없었다. 통신 대 계산 비율이 높고 국부성이 높은 응용 프로그램에서 SMP 노드를 사용하는 효과가 크게 나타나 성능 향상폭이 컸다. SMP 클러스터를 위한 프로토콜을 사용했을 때 모든 응용 프로그램에서 페이지나 diff등의 메시지 전송량이 13% 52% 줄어들어 네트워크 사용이 줄어듦을 확인하였다. 이를 통해 기존의 연구들과는 다른 저가의 상업용 환경에서도 SMP 노드를 사용하는 것이 전체 성능 향상을 가져온다는 것을 보였다.

참 고 문 헌

[1] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, August 1986.

[2] P. Keleher and A. L. Cox and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In Proceedings of the 19th Annual Int'l Symposium on Computer Architecture, May 1992.

[3] 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.

[4] A. Cox and S. Dwarkadas and P. Keleher and H. Lu and R. Rajamony and W. Zwaenepoel. Software versus hardware shared memory implementation: A case study. In Proceedings of the 21st Annual Int'l Symposium on Computer Architecture, April 1997.

[5] A. Erlichson and N. Nuckolls and G. Ghesson and J. Hennessy. SoftFLASH: analyzing the perfor、mance of clustered distributed virtual shared memory. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1997.

[6] R. Stets and S. Dwarkadas and N. Hardavellas and G. Hunt and L. Kontothanassis and S. Parthasarathy and M. Scott. Cashmere 2L: Software coherent shared memory on a clustered remote write network. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[7] D. J. Scales and K. Gharachorloo and A. Aggarwal. Fine grain software distributed shared memory on SMP clusters. In Proceedings of the 4th IEEE Symposium on High Performance Computer Architecture, February 1998.

[8] R. Samanta and A. Bilas and L. Iftode and J.P. Singh. Home based SVM protocols for SMP clusters: design and performance. In Proceedings of the 4th IEEE Symposium on High Performance Computer Architecture, February 1998.

[9] S. K. Lee and H. C. Yun and J. W. Lee and S. R. Maeng. Adaptive prefetching technique for shared virtual memory. In Proceedings of the 3rd International Workshop on Software Distributed Shared Memory System, May 2001.

[10] H. C. Yun and S. K. Lee and J. W. Lee and S. R. Maeng. An efficient lock protocol for home based lazy release consistency. In Proceedings of the 3rd International Workshop on Software Distributed Shared Memory System, May 2001.

[11] 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, May 1990.

[12] A. Bilas. Improving the performance of shared virtual memory on system area networks. PhD thesis, Princeton University, November 1998.

[13] W. Hu and W. Shi and Z. Tang. Reducing System Overheads in Home based Software DSMs. In Proceedings of the Second Merged Symp. IPPS/SPDP 1999. 1999.

[14] 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.

[15] D. Bailey and J. Barton and T. Lasinski and H. +Simon. The NAS Parallel Benchmarks. Technical Report 103863, NASA July 1993.

[16] Honghui Lu and Sandhya Dwarkadas and Alan L. Cox and Willy Zwaenepoel. Quantifying the Performance Differences between PVM and TreadMarks. Journal of Parallel and Distributed Computing, 43(2):65 78, 1997.

[17] P. Keleher. CVM: The Coherent Virtual Machine. Technical Report 93 215, Department of CS, University of Maryland, September 1995.

이 동 현

2000년 한국과학기술원 전자전산학과 전산학전공 학사. 2002년 한국과학기술원 전자전산학과 전산학전공 석사. 2002년~현재 KT 운용시스템연구소 인터넷망관리팀 전임연구원. 관심분야는 Software Distributed Shared Memory, Network Management System, Internet Protocol, Routing Instability

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

박 소 연

1998년 서강대학교 전자계산학과 학사
2000년 한국과학기술원 전자전산학과 전산학전공 석사. 2000년~현재 한국과학기술원 전자전산학과 전산학전공 박사과정. 관심분야는 Software Distributed Shared Memory, Fault Tolerance, Parallel Processing, Computer Architecture

맹 승 렬

정보과학회논문지 : 시스템 및 이론
제 30 권 제 5 호 참조