리눅스 클러스터의 병렬 MPI 프로그램을 위한 동시 스케줄링 기법

정 평재° 조 현웅 이 윤석
한국외국어대학교 전자정보공학부
{system9°, hwcho, rheeys}@dslab.hufs.ac.kr

A Dynamic Co-scheduling Scheme for parallel MPI programs on Linux clusters

Pyoung-Jae Jung° Hyun-Woong Cho Yunseok Rhee
School of Electronics & Information Eng., Hankuk Univ. of Foreign Studies

요 약

본 연구에서는 리눅스 클러스터에서 효과적으로 병렬 MPI 프로그램을 실행시키기 위해, 메시지 전달 이벤트를 토대로 통신에 참여하는 프로세스들이 동시에 스케줄링되는 기법을 설계 구현하였다. 실제 병렬 프로그램의 실행을 통해 측정한 결과에 따르면, 통신량이 높은 프로그램에서 33-56%의 실행 시간 감소 효과를 보였다.

1. 서론

리눅스 클러스터와 같이 각 노드가 자율적인 시분할 스케줄러를 갖는 시스템에서 MPI 병렬 프로그램을 효과적으로 수행하기 위해서는, 작업에 참여하는 프로세스들을 동시 스케줄링(co-scheduling)시키는 것이 중요한 문제이다 [1]. 병렬 작업은 대부분 각 프로세스 간의 빈번한 통신이 요구되는데, 만일 송수신에 참여하는 프로세스의 어느 한 쪽이 스케줄링되지 않은 상황에서는 전체 프로그램의 실행이 진전될 수 없고, 대기 루프를 수행하거나 작업 전환이 발생하여 통신 지연이 가중되고, 결국 처리 성능 저하를 가져온다.

전통적인 동시 스케줄링 기법으로는 갱(gang) 스케줄링[2]을 들 수 있는데, 이 방법은 중앙 관리자가 갱 매트릭스라는 스케줄링 정보를 작성, 배포하고 각 노드의 데몬 프로세스가 이 정보에 따라 각 프로세스에게 CPU를 할당하는 방식이다. 비교적 구현 방법이 단순하여 실제 상용 시스템에서 구현된 사례[3]가 있지만, 분산 환경에서 정확한 스케줄링 정보를 구성하기 어렵고 관리자에 결함이 발생하거나 부하가 집중되면 가용성과 확장성이 저하되는 단점이 있다.

중앙 관리자를 두지 않는 방법으로 동적 동시스케줄링(dynamic co-scheduling, DCS) 기법들[4][5]이 연구되었는데, 이들에서는 각 지역 스케줄러가 특정 이벤트를 근거로 독자적인 스케줄링을 수행하고 결국 이로 인해 동시스케줄링을 일으키는 방법을 사용한다. '통신 메시지의 도착'이 대표적인 이벤트이며, 이 경우에는 메시지가 수신되면 지역 스케줄러는 수신 프로세스를 바로 깨워 스케줄링함으로써 송수신 프로세스가 거의 동시에 스케줄링되도록 지원하는 것이다. 그러나, 대부분의 DCS 기법들이 Myrinet과 같은 고속 스위치를 기반으로 사용자 수준 메시지 계층을 통해 구현되고 있으며, 대부분 NIC의 펌웨어 프로그래밍을 요구한다 [6][7]. 그러나, 이와 같은 전용 하드웨어와 펌웨어 수정 요구는 범용 장비로 대규모 클러스터를 구성하는데 있어 확장성을 제약하는 요소이다.

본 연구에서는 리눅스를 탑재한 범용 PC로 구성된 클러스터 시스템에서 약간의 MPI 라이브러리와 커널 수정을 통해 효과적으로 동시스케줄링을 성취하는 기법을 제안하고 이를 설계 구현하고, 최종 구현된 시스템에서 병렬 작업을 수행시켜 성능을 평가하였다.

2. 제안 기법의 설계 및 구현

본 논문에서 구현한 동시 스케줄러는 리눅스 2.4.18과 MPICH[8] 1.2.5에서 구현하였다. 기본적인 동작 구조는 DCS와 동일한 방식으로 작동을 하도록 하기 위해서 리눅스 커널의 BSD 라이브러리와 MPICH의 ADI layer의 수정하였다.

기본적인 시스템 동작 방식은 그림 1과 같다. MPI 메세지에 해당 메시지를 받을 프로세스의 PID를 같이 전송을 하게 되고, BSD 라이브러리에서 메시지의 PID를 추출 해당 PID의 프로세스를 스케줄링 하도록 한다.

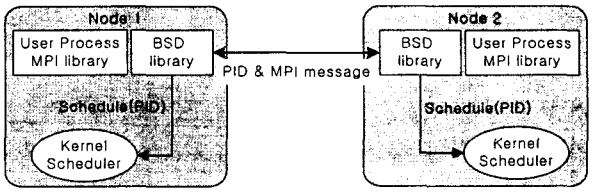

그림 1 동작 구조

2.1. MPI API 구현

MPI 라이브러리에서 그림 2에서와 같이 메시지를 보내는 API에 커널에서 동시스케줄링에 필요한 정보인 PID와 MPI 메시지임을 식별할수 있는 문자열 코드를 추가하여 커널 TCP계층에서 MPI 메시지를 구별하고 병렬 작업의 PID를 찾아 우선순위를 높일 수 있도록 구현하였다. 기존의 MPI 응용 프로그램의 수정이 필요 없고, MPI 라이브러리의 수정을 최소화 하기위하여 MPI의 ADI(Abstract Device Interface)계층에 구현하였다. 모든 MPI

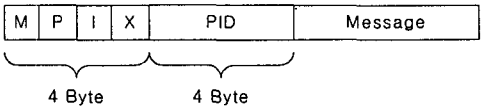

그림 2 수정된 MPI 메시지 구조

API의 구조는 ADI에 의해 구현되어 있고, 실제 메시지의 전송 기능과 API와 H/W사이의 데이터 전달, 메시지의 리스트관리와 실행 환경에 대한 정보를 다루는 기능을 한다.

MPI 라이브러리에서는 MPI 응용 프로그램에서 생성된 병렬 작업 프로세스의 PID를 관리하고 메시지를 전송시 해당 메시지를 받을 프로세스의 PID를 같이 전송하는 방식으로 구현을 하였다. 그림 3에서와 같이 MPI_Init을 통해 병렬 프로그램의 PID가 테이블에 저장이 되고 유저레벨에서 MPI_Send를 호출하게 되면 ADI레이어의 MPID_SendDatatype, MPID_Contig 순으로 유저 메시지가 전달되며, MPID_Contig에서 MPI와 PID정보

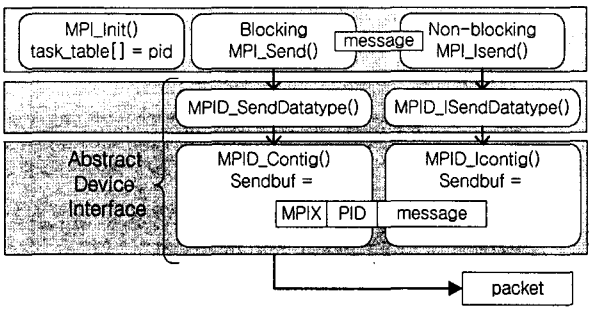

그림 3 ADI interface를 통한 send 구현부

를 추가하여 메시지를 전송하게 된다. Non-blocking 방식도 마찬가지로 유저레벨에서 MPI_Isend를 호출하여 메시지를 보내면 MPID_SendDatatype과 MPID_Icontig 순으로 메시지가 전달되며 MPID_Icontig에서 MPI와 PID를 추가하여 전송하게 된다.

recv에서는 수신된 MPI메세지에서 본 논문에서 추가를 한 헤더를 제거하고 어플리케이션으로 올려주는 작동을 한다. MPI 메세지의 전송이 완료가 되면 MPID_RecvCompleted가 호출이 되고 MPID_IrecvContig에서 헤더와 실제 MPI메세지를 분리하게 된다. 그 후 MPID_recvContig과 MPID_IrecvDatatype, MPID_RecvDatatype을 거쳐 MPI_Recv에서 유저로 본래의 메시지를 얻는다. 따라서 MPID_IrecvContig에 recv코드를

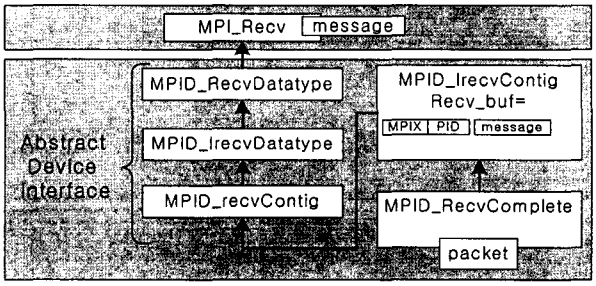

그림 4 ADI interface를 통한 recv 구현부

추가함으로써 구현 하였다.

그림 4는 MPI ADI를 이용한 메시지를 받는 것을 간략하게 보여주고 있다. MPID_RecvComplete로 패킷을 받고, 그 상위의 MPID_IrecvContig에서 헤더부분과 실제 MPI 메시지 부분을 분리해서 메시지 부분은 앞에서 언급한 ADI의 여러 계층을 거쳐 유저에게 전달된다.

2.2. 커널 부분 구현

관련된 프로세스를 같은 시간에 스케줄링하기 위해서는 각 클러스터의 로컬 스케줄러에서 특정 PID를 찾아서 우선순위를 조정할 수 있는 기능의 로컬 스케줄러 구현이 필요하였다. 본 논문에서 구현한 방식은 다음과 같다.

1. MPI 메시지를 받을 프로그램의 스케줄링을 우선적으로 할 수있도록 nice값을 -99, policy를 SCHED_RR로 변경한다.
2. 스케쥴링이 된 MPI 응용 프로그램은 우선순위를 원래로 변경을 한다. (nice = 0, policy = SCHED_OTHER)

커널에서 MPI메세지의 구별을 하기 위해서 그림 2와 같이 MPI 메시지 필드에 문자열"MPIX"를 추가하고 우선순위 수정을 위한 PID를 추가했다. 커널에서"MPIX"메시지가 확인되면 다음 4바이트의 field에서 정수형의 PID 값을 이용하여 같은 PID를 갖는 프로세스를 찾게 된다.

해당 PID의 프로세스의 nice값을 -99로 주어 우선순위를 올림과 동시에 해당 프로세스가 MPI 응용프로그램임을 표시하게 된다. 그러나 실제로 우선순위를 올리는 효과를 내는 부분은 스케쥴링 policy를 SCHED_RR로 변경하는데 있다. 이렇게 우선순위를 올린 프로세스는 다른 프로세스에 비해서 높은 우선순위를 받게 되므로 자연스럽게 스케줄링을 유도 할 수가 있게 된다. 해당 프로세스가 스케줄링이 되고난 후 최대한 다른 프로세스와 동등한 스케줄링을 위해서 nice와 policy를 원래로 변경하도록 한다.

정리를 하면, 동시 스케줄링에 필요한 정보인 PID정보를 MPI 메시지에 포함 시켜 전송을 하게 되고, 메세지가 커널에 도착하면 인터럽트에 의해 인터럽트 함수가 호출되어 TCP계층에 도착하게 되면, socket buffer에 저장이 되어있는 메시지의 헤더 부분을 커널에서 분석하게 된다. 헤더 정보에서 MPI메세지임을 분석하고, PID부분을 프로세스 리스트에서 찾아 프로세스의 스케줄링 policy를 SCHED_RR로 바꾸어 높은 우선순위로 CPU를 할당받을 수 있다. 그림 5는 MPI 메시지가 실제로 커널에 도착하여 커널의 여러 계층를 거쳐 TCP계층에 도착했을 때, TCP계층에서 메시지의 PID를 분석해 해당 프로세스를 찾은 뒤,

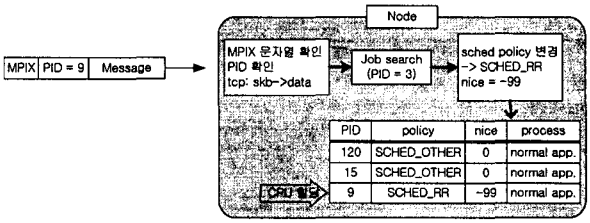

그림 5 커널에서의 데이터 수신 및 스케줄링

스케줄링 정책과 nice 값을 조정하여 우선적으로 MPI 프로세스가 CPU를 할당받는 것을 보여주고 있다. 이후 MPI 라이브러리에서는 정상적인 작동을 위해 추가된 헤더를 제외한 메시지를 어플리케이션에 전달을 한다.

3. 실험 및 성능 평가

본 논문의 실험에서는 NPB(NAS Parallel Benchmark)[9]를 이용해 수행 시간을 검토하였다. workload는 EP, LU, BT, CG

를 수행하였고, 프로그램 크기는 W와 A를 선택하였다. 실험 환경은 펜티엄 III의 450Mhz 128MB 8노드 클러스터 시스템, 100Mbps 네트워크 환경에서 동시 스케줄러를 적용했을 때와 적용하지 않았을 때의 수행 시간을 중심으로 성능 평가를 하였다.

3.1. 실험 1

LU는 통신량이 많은 대표적 workload이다. LU의 경우 통신 패턴은 링형이고, 메시지 send와 recv의 횟수가 아주 많다. 기존 스케줄링을 적용했을 때보다 동시 스케줄링 기법을 적용했을 때의 수행 시간 차이가 시스템 부하를 올리면서 수행할수록 많은 수행시간의 향상을 보였다. LU의 경우 카피오버헤드가 다른 workload에 비해 높았지만 동시 스케줄링의 효과가 시스템의 부하를 증가시킬수록 높아져 카피 오버헤드가 많은 영향을 주지 못하였다. 33%-56%까지 실행 시간이 감소하는 동시 스케줄링의 효과가 있었다.

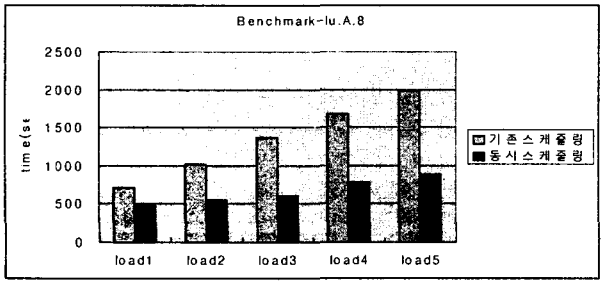

그림 7 Benchmark LU.A.8

3.2. 실험 2

실험 2에서는 MPL(MultiProgramming Level)을 구성하였다. 그림 7은 MPL level 2 (`lu.A.8, cg.A.8`)에서의 실험 결과이다. 통신량이 많은 workload와 중간인 workload를 같이 수행시킨 결과, 동시 스케줄링 기법을 적용한 것이 현저하게 실행 시간이 짧음을 알 수 있다. 그리고 시스템 부하가 올라갈수록 동시 스케줄링과 기존 스케줄링 간의 수행시간 차가 커졌다. 7%-38%의 실행 시간 감소를 보여 동시 스케줄링의 효과가 있었다. 하지만 부하가 낮을 때는 동시 스케줄링의 효과가 크게 나타나지 않고 있는데, 이는 통신 량이 많은 LU의 카피 오버헤드가 동시 스케줄링의 효과에 비해 많은 영향을 미치기 때문이다.

MPL 단계별로 시스템의 부하를 올리면서 실험한 결과 시스템의 부하가 낮은 경우 동시 스케줄링의 효과가 나타나지 않았고, 오히려 카피 오버헤드와 계산량이 많은 workload에 의해 동시 스케줄링을 적용했을 때 수행 시간이 더 소요됐다. 하지만 시스템의 부하가 올라갈수록 동시 스케줄링의 효과가 커지는 경향을 보였다. MPL의 단계가 높아져도 낮은 단계의 MPL에 비해 동시 스케줄링의 효과는 커지지 않았다. 각 MPL level에 이와 같은 결과는 무부하 상태의 시스템에서보다 부하가 있는 시스템에서 동시 스케줄링을 적용했을 때 병렬 작업의 수행 시간을 줄일 수 있었다.

4. 결론

본 논문에서 구현한 동시 스케줄러는 MPI 병렬 프로그램에서

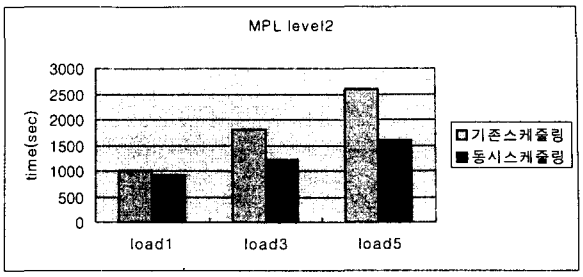

그림 6 Benchmark MPL level 2

의 메시지 이벤트를 이용 최대한 빠르게 스케줄링을 하여 클러스터 컴퓨팅에서의 스케줄링 오버헤드를 줄이는 동시 스케줄러를 구현을 하였다. 부하가 많이 걸린 시스템에서 그 효과가 두드러지게 나타나고, 특히 수행하는 workload의 통신이 빈번한 경우 높은 효과가 있음을 알 수 있었다. 그러나 부하가 없는 경우는 성능 효과가 나타나지 않는 것을 볼 수가 있었는데, 원인은 경쟁하고 있는 다른 프로세스가 없을 경우 병렬 프로세스의 우선순위를 올려서 스케줄링을 유도하는 방식으로는 그 효과를 기대하기 어려운 점에 있다.

실험한 결과로 볼 때 본 논문에서 개선을 해야 할 부분은 메시지 카피를 위한 오버헤드이다. 동시 스케줄링을 위한 메시지 작성에 빈번한 메시지 카피가 상당한 부하로 연결이 되는 것을 볼 수가 있었다. 그리고 우선순위를 조정함으로 스케줄링을 유도 하는 방식이 아닌 직접 해당 프로세스의 스케줄링을 지정하는 스케줄러의 설계로 부하가 없는 상황에서도 동시 스케줄러의 효과가 있도록 하는 것이 앞으로 필요한 연구이다.

참고 문헌

[1] S. Nagar et al., A Closer Look At Coscheduling Approaches for a Network of Workstations, In Proc. 11th ACM Symp. of Parallel Algoritms and Architectures, 1999.

[2] D. Feitelson and M. Jette, Improved Utilization and Responsiveness with Gang Scheduling, LNCS, vol. 1291, pp. 238-261, 1997.

[3] J. Moreira et al., A Gang-Scheduling System for ASCI Blue-Pacific, In Proc. Distributed Computing and Metacomputing Workshop, HPCN'99, Apr. 1999.

[4] P. Sobalvarro, Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors, PhD Thesis, EECS, MIT, 1997.

[5] Patrick G. Sobalbarro , Dynamic Coscheduling on Workstation Clusters. Scott Parkin, William E. Weihl , and Andrew A. Chien. Lecture Notes on Computer Science .vol . 1459. pp 231 - 1998.

[6] S. Pakin et al., High Performance Messaging on Workstations: Illinois Fast Messages (FM), In Proc. Supercomputing '95, Dec. 1995.

[7] T. Eicken et al., U-Net: A User-level Network Interface for Parallel and Distributed Computing, In Proc. 15th ACM SOSP, 1995.

[8] http://www-unix.mcs.anl.gov/mpi/mpich/

[9] NPB NAS Parallel Benchmark http://www.nas.nasa.gov/NAS/NPB