韓國 컴퓨터情報學會 論文誌
第13卷 第1號, 2008. 1.

2008-13-1-1-4

리눅스 클러스터에서 MPI 기반 병렬 프로그램의 동적 동시 스케줄링 기법

김 혁*, 이 윤 석*

A Dynamic Co-scheduling Scheme for MPI-based Parallel Programs on Linux Clusters

Hyuk Kim*, Yunseok Rhee**

요 약

빈번한 메시지를 주고 받는 MPI 기반의 병렬 프로그램에서 효과적으로 통신이 이뤄지기 위해서는 송수신 프로세스들이 각 노드에서 동시에 스케줄되어야 한다. 그러나, 일반적으로 클러스터 컴퓨터를 구성하는 각 노드는 범용 시분할 운영체제를 기반으로 하며, 이 경우 병렬 프로그램을 구성하는 프로세스들은 각 스케줄러에 의해 자율적으로 관리되므로 이들을 동시에 함께 실행시키는 것은 쉽지 않다. 본 연구에서는 리눅스 클러스터에서 효과적으로 병렬 MPI 프로그램을 실행시키기 위해, 메시지 교환 정보를 활용하여 통신에 참여하는 프로세스들이 동시에 스케줄되는 기법을 제안하고 실제 구현을 통해 성능을 살펴보았다. NPB 병렬 벤치마크의 수행을 통해 측정한 결과에 따르면, 통신량이 높은 프로그램에서 33-56%의 실행 시간 감소 효과를 보였다.

Abstract

For efficient message passing of parallel programs, it is required to schedule the involved two processes at the same time which are executed on different nodes, that is called 'co-scheduling'. However, each node of cluster systems is built on top of general purpose multitasking OS, which autonomously manages local processes. Thus it is not so easy to co-schedule two (or more) processes in such computing environment. Our work proposes a co-scheduling scheme for MPI-based parallel programs which exploits message exchange information between two parties. We implement the scheme on Linux cluster which requires slight kernel hacking and MPI library modification. The experiment with NPB parallel suite shows that our scheme results in 33-56% reduction in the execution time compared to the typical scheduling case, and especially better performance in more communication-bound applications.

▸ Keyword : 동적 동시스케줄링(Dynamic Co-scheduling), MPI(Message Passing Interface), 병렬 프로그램(Parallel Programs), 리눅스 클러스터(Linux Cluster)

• 제1저자 : 김혁
• 접수일 : 2007. 12. 5, 심사일 : 2008. 1.11, 심사완료일 : 2008. 1.25.
* 한국외국어대학교 전자정보공학부 석사과정 **한국외국어대학교 전자정보공학부 교수
※ 이 논문은 2006학년도 한국외국어대학교 학술연구비 지원에 의하여 이루어진 것임.

Ⅰ. 서 론

저비용 SMP(symmetric multiprocessor) 하드웨어를 고속네트워크로 연결한 리눅스 클러스터들이 고성능 병렬어플리케이션의 해결에 활용되고 있다. 따라서 이와 같은 클러스터의 성능을 높이기 위해 병렬 작업을 효과적으로 관리하는 것은 중요한 일이 되었으며, 많은 슈퍼컴퓨팅센터에서는 이들 클러스터의 작업관리를 위해 LoadLeveler[10]와 같은 batch 시스템을 사용하고 있다. 그러나, 이들 시스템에서는 장시간 작업들을 효과적으로 스케줄하여 시스템 전체의 처리 성능을 높이는 대신, 개별 프로세스의 응답성은 매우 낮은 결과를 초래한다[11]. 이의 대안으로, 클러스터를 구성하는 각 노드의 시분할(time-sharing) 스케줄링을 활용하여 시스템의 응답성을 개선하는 연구들이 제안되었고, 특히 이와 같은 시분할 환경에서 메시지 패싱(message passing)에 의존하는 각 작업(job)의 프로세스들을 어떻게 여러 노드에서 동시에 스케줄할 것인가는 매우 중요한 문제이다.

리눅스 클러스터와 같이 각 노드가 자율적인 시분할 스케줄러를 갖는 시스템에서 MPI (Message Passing Interface) 기반의 병렬 프로그램을 효과적으로 수행하기 위해서는 작업에 참여하는 프로세스들을 각 프로세서에서 동시에 실행시키는 것이 필요하며, 이를 '동시 스케줄링 (co-scheduling)'이라고 한다[1].

이 병렬 작업의 프로세스들은 대개 메시지 교환을 통해 협력하므로, 대부분의 경우 관련된 모든 프로세스들이 메시지 수수 작업을 완료하기 전까지는 어느 프로세스도 더 이상 진행할 수 없다. 따라서, 만일 메시지 수수 시점에서 관련 프로세스들이 각 노드에서 동시에 스케줄되지 않는다면, 프로세스 간 통신(IPC)이 작업의 진행을 방해하는 주요소가 된다. 만일 동시 스케줄링이 지원되지 않는다면, 각 프로세스는 메시지 전달을 기다리는 대기시간과 시분할 스케줄러에 의한 작업 전환(context switching) 시간을 고스란히 작업 실행시간에 포함해야 한다. 이와 같은 상황은 결국 시스템 전체의 성능을 저하시키는 요인이다[12].

각 노드가 자율적인 시분할 스케줄러를 동작시키는 환경에서 병렬 작업의 프로세스들을 동시에 스케쥴링하는 것은 쉽지 않은 일이며, 한편으로 이를 지원할 수 있는 새로운 실행 환경을 설계 제공할 필요가 있다. 지금까지 SMP 클러스터나 NOW(network of workstation) 환경에서 동시 스케줄링을 지원하는 연구들이 다수 제안되었는데, 이 가운데 가장 단순한 접근법의 하나는 gang scheduling이다[2,3,13]. 이 방법은 gang matrix라는 스케줄링 정보를 사용하여 모든 노드의 스케줄링 정보를 관리한다. 이 matrix의 각 열(column)은 프로세스 정보를 나타내고, 각 행(row)에는 해당 프로세스의 실행 시간을 기록한다. 이를 통한 동시 스케쥴링은 각 행의 모든 프로세스들을 동시에 각 노드에서 실행시키도록 함으로써 이뤄지는데, 이를 위해 일반적으로 독립된 노드에서 실행되는 중앙 관리자(central manager, CM)가 필요하다. 결국 이 CM이 각 노드에서의 스케쥴에 변화가 일어날 때마다 gang matrix를 업데이트하여 각 노드에 배포한다. 이 방법은 병렬 작업의 동시 스케쥴링을 보장하는 비교적 단순한 방법이다. 실제 이 방식의 후속 시스템들이 실제 개발된 사례가 있지만[3,14], 이 방법은 크게 몇 가지 제약을 안고 있다. 첫째, 정확한 스케쥴링이 완전히 분산 스케쥴링 정보의 무결성에 여부에 따라 좌우된다. 만일 이 스케쥴 정보 가운데 일부가 불안정한 네트웍 상황에 의해 훼손되면, 전체 프로세스들이 동시 스케쥴되지 않은 가능성이 매우 높다. 둘째로, 이 gang scheduler의 CM에 부하가 가중되거나 문제가 생겨 노드가 다운될 경우 동시 스케쥴링이 어렵게 된다. 또한 시스템의 노드 수가 점차 증가하면

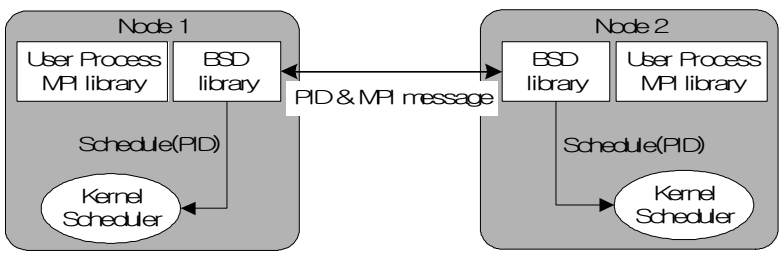

그림 1. 노드 간 동시 스케쥴링을 지원하는 소프트웨어 구조
Fig 1. Software Architecture for Co-scheduling between Nodes

서, gang matrix의 크기 뿐아니라 CM과 각 노드가 주고 받는 제어 메시지도 자연 증가한다. 이 제어 메시지에는 각 노드의 현재 부하 상태, 실행 중의 작업의 수 등의 정보를 담게 되고, 이와 같은 정보를 수집하기 위해 CM의 부하가 과도해 질 경우 gang scheduling의 확장성을 제약한다. 또한 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) 기법들이 연구되었는데 [4, 5], 이들에서는 각 지역 스케줄러가 특정 이벤트를 근거로 독자적인 스케줄링을 수행하고 결국 이로 인해 동시 스케줄링을 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.

II. 제안 기법의 설계 및 구현

본 논문에서 구현한 동시 스케줄러는 리눅스 커널 버전 2.4.35과 MPICH 라이브러리 버전 1.2.7 [8]를 기반으로 구현하였다. 메시지에 따른 스케줄링을 지원하기 위해 리눅스 커널의 메시지 전송 계층인 BSD TCP/IP 프로토콜 계층을 수정하였고, MPICH 라이브러리의 ADI(Abstract Device Interface) 계층의 수정이 요구되었다.

기본적인 시스템 동작 방식은 그림 1과 같다. Node 1의 송신 프로세스에서 MPI 메시지를 전송할 때, 해당 메시지가 MPI 메시지임을 알리는 tag 정보와 Node 2에 위치한 수신 프로세스 번호(PID)를 첨부하여 전송한다. 이 때 Node 2의 BSD 계층에서는 해당 메시지를 해석하여 MPI 메시지임과 수신 프로세스 번호를 알아내고 이를 스케줄러에게 전달하여 해당 프로세스가 스케줄링되도록 한다.

2.1 MPI 구현부의 수정

본 연구를 위해 그림 2와 같은 MPI 메시지 형식의 재정의가 필요하다. 메시지의 선두에 MPI를 알리는 태그 "MPIX"를 추가하고, 이어 수신 PID를 기록함으로써 커널 TCP 계층에서 MPI 메시지를 식별하고, 해당 PID를 갖는 수신 프로세스의 우선순위를 높여 가능한 신속하게 스케줄링되도록 구현하였다. 본 연구에서는 특히 기존의 MPI 응용 프로그램의 수정이 필요 없고, MPI 라이브러리의 수정을 최소화 하기위하여 MPI의 ADI(Abstract Device Interface) 계층을 수정하였다. 모든 MPI 기능은 ADI 계층에서 구현되고, ADI 계층은 실제 메시지의 전송기능과 API와 H/W사이의 데이터 전달, 메시지의 리스트관리와 실행 환경에 대한 정보를 다루고 있다.

MPI 라이브러리에서는 MPI 응용 프로그램에서 생성된 병렬 작업 프로세스의 PID를 관리하고 메시지를 전송 시 해당 메시지를 받을 프로세스의 PID를 같이 전송하는 방식으로 구현하였다. 이를 위해, 그림 3에서와 같이 MPI_Init을 통해 병렬 프로그램의 PID가 테이블에 저장되고 사용자 수준에서 MPI_Send를 호출하면 ADI계층의 MPID_SendDatatype, MPID_Contig 순으로 사용자 메시지가 전달되며, MPID_Contig에서 MPI 태그와 PID 정보를 추가하여 메시지를 전송한다. 블록킹 없는 (non-blocking) 방식도 마찬가지로 사용자가 MPI_Isend를 호출하여 메시지를 보내면 MPID_SendDatatype과 MPID_Icontig 순으로 메시지가 전달되며 MPID_Icontig에서 MPI 태그와 PID를 추가하여 전송하게 된다.

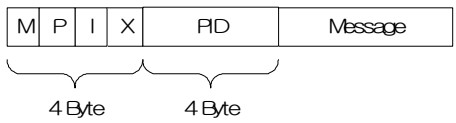

그림 2. 수정된 MPI 메시지 형식
Fig 2. Modified MPI Message Format

수신 노드 TCP/IP 계층의 recv 기능은 수신된 MPI 메시지에서 본 논문에서 추가한 헤더를 제거하고 사용자 프로그램으로 전달한다. MPI 메시지의 전송이 완료되면 MPID_ Recv Completed가 호출되고 MPID_IrecvContig에서 헤더와 실제 MPI 메시지를 분리하게 된다. 그 후 사용자 프로그램에서는 MPI_Recv를 호출하는데, 이는 순차적으로 MPID_recvContig과 MPID_IrecvDatatype, MPID_ RecvDatatype을 거쳐 본

래의 메시지를 얻는다.

그림 3은 MPI ADI 계층을 이용한 메시지 수신 과정을 간략하게 보여주고 있다. MPID_RecvComplete로 패킷을 받고, 그 상위의 MPID_IrecvContig에서 헤더부분과 실제 MPI 메시지 부분을 분리해서 메시지 부분은 앞에서 언급한 ADI 계층의 여러 함수들을 거쳐 사용자에게 전달된다.

2.2 커널 부분 구현

관련된 프로세스를 같은 시간에 스케줄링하기 위해서는 각 클러스터의 로컬 스케줄러에서 특정 PID를 찾아 우선순위를 높일 수 있는 로컬 스케줄러가 필요하였다. 본 논문에서 기존의 리눅스 스케줄러에 다음의 기능을 갖도록 수정하였다.

1. MPI 메시지를 받을 프로그램의 스케줄링을 우선적으로 할 수 있도록 nice값을 -99, 스케줄링 policy를 SCHED_RR로 변경한다.

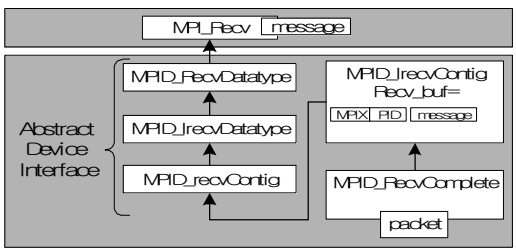

그림 3. ADI interface를 통한 recv 함수 구현부
Fig 3. recv function by ADI interface

2. 스케줄된 MPI 응용 프로그램은 우선순위를 원래로 변경을 한다. (nice = 0, policy = SCHED_OTHER)

앞서 설명한 바와 같이, 커널에서 MPI메세지의 구별을 하기 위해서 그림 2와 같이 MPI 메시지 필드에 MPI 태그 "MPIX"를 추가하고 수신 프로세스의 PID를 추가했다. 따라서 수신 노드의 네트웍 계층에서 MPI 메시지를 확인하면, 뒤따르는 정수형의 PID 값을 이용하여 해당 프로세스를 찾게 된다.

해당 PID의 프로세스를 찾아내면, 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 신속한 스케줄링을 유도할 수가 있게 된다. 해당 프로세스가 스케줄되고난 후 최대한 다른 프로세스와 동등한 스케줄링을 위해서 nice와 policy를 원래 상태로 변경한다.

간단히 정리하면, 동시 스케줄링에 필요한 정보인 PID정보를 MPI 메시지에 포함시켜 전송하고, 메시지가 커널에 도착하면 인터럽트에 의해 인터럽트 함수가 호출되어 TCP계층에 도착하며, socket buffer에 저장이 되어있는 메시지의 헤더 부분을 커널에서 분석한다. 헤더 정보에서 MPI 메시지임을 분석하고, PID 부분을 프로세스 리스트에서 찾아 프로세스의 스케줄링 policy를 SCHED_RR로 바꾸어 높은 우선순위로 CPU를 할당받을 수 있다. 그림 4는 MPI 메시지가 실제로 커널에 도착하여 커널의 여러 계층를 거쳐 TCP계층에 도착했을 때, TCP계층에서 메시지의 PID를 분석해 해당 프로세스를 찾은 뒤, 스케줄링 정책과 nice 값을 조정하여 우선적으로 MPI 프로세스가 CPU를 할당받는 것을 보여주고 있다. 이후 MPI 라이브러리에서는 정상적인 작동을 위해 추가된 헤더를 제외한 메시지를 어플리케이션에 전달한다.

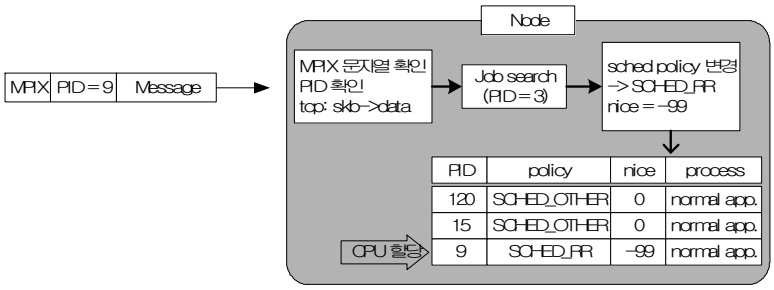

그림 4. 커널에서의 데이터 수신 및 스케줄링
Fig 4. Data Receiving and Scheduling in Linux Kernel

III. 실험 및 성능 평가

본 논문의 실험을 위해 NPB (NAS Parallel Benchmark) [9]를 이용하였으며, 벤치마크 workload로는 EP, LU, BT, CG를 수행하였고, 프로그램 크기는 W와 A를 선택하였다. 실험 환경은 펜티엄 III의 450Mhz 128MB 8노드 클러스터 시스템, 100Mbps 네트워크 환경에서 실험하였고, 한 노드에 한 개의 프로세스를 생성하는 형태로 병렬 프로그램을 수행하였다. 측정 자료는 동시 스케줄러를 적용했을 때와 적용하지 않았을 때의 실행 시간을 중심으로 성능을 평가하였다.

3.1 실험 1

이 실험에서는 대표적으로 통신량이 많은 프로그램 LU를 대상으로 성능을 비교하였다. 쉽게 예측할 수 있는 것처럼, 각 노드의 부하가 높을수록 (즉, 스케줄할 프로세스의 수가 매우 많은 경우) 송수신 프로세스 간에 동기화 가능성이 낮아 통신 지연의 영향이 클 것이다. 따라서 본 실험에서는 각 노드의 부하를 점차적으로 증가시키면서 제안 기법의 효과를 살폈으며, 부하를 가중시키는 방법은 CPU-bound 프로세스의 수를 점차 늘려갔다.

그림 5의 load1의 경우처럼 각 노드의 부하가 거의 없는 경우에는 각 프로세서(CPU)를 해당 병렬 프로세스가 거의 독점하므로 기존 스케줄링과 동시 스케줄링의 차이가 아주 크게 나타나지는 않았다. 그러나, 시스템 부하가 증가하면서 동시 스케줄링 기법이 월등한 성능을 보여 33-56%의 실행 시간 감소를 보였다.

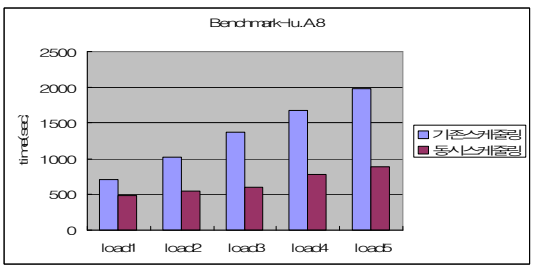

그림 5. Benchmark LU.A.8의 수행 결과
Fig 5. Execution Result of Benchma가 LU.A.8

한편 본 연구의 제안 기법에서는, 사용자 수준에서 전달한 메시지를 커널에서 다시 재복사해야 하는 복사 오버헤드가 발생하며 이 부분이 효과를 감쇠시킬 수 있었다. 그러나, 실험 결과를 볼 때 다행히 그리 심각한 영향을 끼치지는 않는 것으로 나타났다.

3.2 실험 2

실험 2에서는 여러 형태의 MPL (Multi-Programming Level)을 구성하였다. 그림 6은 MPL level 2 (lu.A.8, cg.A.8)에서의 실험 결과이다. 통신량이 많은 workload와 중간인 workload를 함께 수행시킨 결과, 동시 스케줄링 기법을 적용한 경우 현저하게 실행 시간이 짧게 나타났다. 또한 실험 1에서와 마찬가지로 시스템 부하가 올라갈수록 동시 스케줄링과 기존 스케줄링 간의 성능 격차가 커져, 약 7%-38%의 실행 시간 감소를 보이는 동시 스케줄링의 효과가 있었다. 하지만 부하가 낮을 때는 동시 스케줄링의 효과가 크게 나타나지 않고 있는데, 이는 통신량이 많은 LU의 재복사 오버헤드가 동시 스케줄링의 효과에 비해 상대적으로 큰 영향을 미쳤기 때문으로 분석된다.

MPL 단계별로 시스템의 부하를 올리면서 실험한 결과 시스템의 부하가 낮은 경우 거의 동시 스케줄링의 효과를 볼 수 없었고, 오히려 복사 오버헤드와 계산량이 많은 workload에 의해 동시 스케줄링을 적용했을 때 오히려 실행 시간이 더 길어지는 경우도 나타났다. 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 대한 이와 같은 결과는 무부하 상태의 시스템에서보다 부하가 있는 시스템에서 동시 스케줄링을 적용했을 때 병렬 작업의 실행 시간을 줄일 수 있음을 보이고 있다.

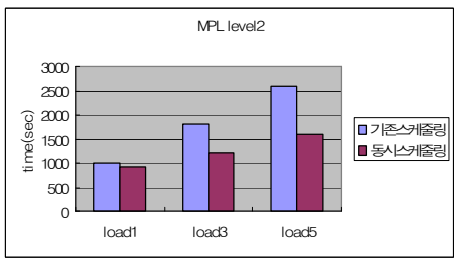

그림 6. Benchmark MPL level 2의 수행 결과
Fig 6. Execution Result of Benchmark MPL

IV. 결론

본 논문에서 구현한 동시 스케줄러는 MPI 병렬 프로그램에서의 메시지 이벤트를 이용 최대한 신속하게 수신 프로세스를 스케줄하여 클러스터 컴퓨팅에서 효과적으로 병렬 프로그램을 실행시킬 수 있는 동시 스케줄러를 구현을 하였다. 본 연구에서는 특히 MPI 구현부와 리눅스의 네트웍 계층을 간단히 수정함으로 기존의 응용 프로그램의 수정이 전혀 필요없도록 하였다.

제안 기법은 부하가 높은 환경에서 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. 우선순위를 조정함으로 스케줄링을 유도 하는 방식이 아닌 직접 해당 프로세스의 스케줄링을 지정하는 스케줄러의 설계가 향후 개선될 점이다.

또한 실험 결과로 볼 때 동시 스케줄링을 위한 메시지 작성에 빈번한 메시지 복사가 성능 감쇠로 연결이 되는 것을 볼 수가 있었다. 따라서 메시지 복사를 위한 오버헤드를 줄이는 것도 앞으로 본 연구의 결과를 개선할 부분이다.

참고문헌

〔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, pp. 96-105, 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, pp. 831-840, 1999.

〔4〕 P. Sobalvarro, Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors, PhD Thesis, MIT, 1997.

〔5〕 Patrick G. Sobalvarro et al., Dynamic Coscheduling on Workstation Clusters, Lecture Notes in Computer Science, vol. 1459, pp 231-256, 1998.

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

〔7〕 T. Eicken et al., U-Net: A User-level Network Interface for Parallel and Distributed Computing, In Proc. of 15th ACM SOSP, pp. 40-53, 1995.

〔8〕 http://www-unix.mcs.anl.gov/mpi/mpich/

〔9〕 NPB NAS Parallel Benchmark http://www.nas.nasa.gov/NAS/NPB

〔10〕 J. Skovira, W. Chan, H. Zhou, and D. Lifka, "The Easy-LoadLeveler API Project", IPPS'96 Workshop on Job Scheduling Strategies for Prarallel Processing", pp. 41-47, Apr. 1996.

〔11〕 D.H. Bailey et al., "Valuation of Ultra-Scale Computing Systems: A White Paper", Dec. 1999.

〔12〕 J.K. Ousterhout, "Scheduling Technique for Concurrent Systems", Int'l Conf. on Distributed Computing Systems, pp. 22-30, 1982.

〔13〕 M. Jette, "Performance Characteristics of Gang Scheduling in Multiprogrammed Environments", Proc. of Supercomputing'97, Nov. 1997.

〔14〕 M. Jette, "Expanding Symmetric Multiprocessor Capability Through Gang Scheduling", IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing, Mar. 1998.

저 자 소 개

김 혁 (Hyuk Kim)

2007년 2월: 한국외국어대학교 전자정보공학부 학사
2007년 3월-현재: 한국외국어대학교 전자정보공학부 석사과정
관심분야: 임베디드시스템, 센서네트워크

이 윤 석 (Yunseok Rhee)

1988년 2월: 서울대학교 계산통계학과 학사
1999년 2월: 한국과학기술원 전산학과 박사
1999년-현재 : 한국외국어대학교 교수

관심분야: 분산시스템, 임베디드컴퓨팅