워크스테이션 클러스터에서 순차적 부하가 병렬 부하에 미치는 영향에 대한 연구

장 영 민† · 심 영 철††

요 약

네트워크로 연결된 워크스테이션들인 NOW(Network of Workstations)가 순차 프로그램과 병렬 프로그램들을 모두 효율적으로 수행할 수 있는 환경을 제공하려면 이 두 종류의 프로그램들이 어떻게 상호 영향을 주는지 이해할 필요가 있다. 본 논문에서는 순차 프로그램의 부하의 변화에 따른 병렬 프로그램의 성능 변화를 알아보았다. 이러한 목적으로 NOW 시뮬레이터를 구현하고 실험을 하였다. 구현에서 순차 프로그램과 병렬 프로그램들은 합성 부하를 사용하였고 병렬 프로그램의 수행 환경으로는 코스케쥴링과 프로세스 이주가 가능하다고 가정하였다. 실험을 통해서 순차 프로그램들에 의해 변하는 유휴 워크스테이션의 수가 병렬 프로그램의 성능에 어떠한 영향을 미치는지와 동시에 수행되는 여러 개의 병렬 프로그램들이 서로의 성능에 어떠한 영향을 미치는 가에 대해 결과를 수집하고 분석하였다.

A Study on Effects of Sequential Workloads on Parallel Workloads in Workstation Clusters

Young-Min Chang† · Young-Chul Shim††

ABSTRACT

In order that NOW(Network of Workstations) which is a cluster of workstations connected by a network can provide an efficient environment for both sequential programs and parallel programs, we should understand how these two kinds of programs affect the performance of each other. In this paper, we examine how the change in the sequential workloads affect the performance of parallel programs. With this purpose we implemented and experimented with a NOW simulator. In the implementation, we used synthetic workloads for sequential programs and parallel programs and we assumed coscheduling and process migration for the execution environment of parallel programs. In the experimentation we collected and analyzed data on how the number of idle workstations which is controlled by sequential programs affects the performance of parallel programs and how the performance of a parallel program is affected if several parallel programs are executed simultaneously.

1. 서 론

컴퓨터 하드웨어 기술의 발달과 네트워크 기술의 발달로 인해 컴퓨터의 성능이 급속히 좋아지고 있으며 컴퓨터간 통신 속도도 ATM[1], Myrinet[2]과 같은 기술로 인해 대단히 빨라지고 있다. 최근 이러한 조건을 기반으로 자치적으로 운영되고 있는 워크스테이션들을 네트워크로 연결시켜 MPP (Massively Parallel Processors)와 같은 수퍼컴퓨터의 성능을 가질 수 있

※ 본 논문은 1996년도 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었슴.
† 준 회 원 : 홍익대학교 대학원 전자계산학과 석사과정
†† 종신회원 : 홍익대학교 컴퓨터 공학과 조교수
논문접수 : 1998년 2월 5일, 심사완료 : 1998년 6월 16일

도록 하는 연구가 진행중이다. 이러한 시스템을 가리켜 NOW(Network of Workstations)[3]라고 한다.

NOW 연구의 배경은 다음과 같다. 학교나 회사에서 사용되고 있는 많은 수의 워크스테이션들은 Ethernet 등을 통하여 하나로 연결되어 있다. 그 중에서 항상 사용중인 것이 그렇게 많지 않다. 워크스테이션의 계산 수행 능력의 비약적인 발전과 통신 대역폭의 확대로 사용되고 있지 않은 워크스테이션들을 이용하여 병렬 프로그램을 수행할 수 있다. 사용되지 않는 워크스테이션들을 사용하므로 일반 사용자가 수행시키는 순차 프로그램에 영향을 끼치지 않으면서 병렬 프로그램을 수퍼컴퓨터보다 저렴한 가격으로 수행할 수 있다.

NOW 환경은 네트워크에 연결되어 있는 여러 대의 워크스테이션들을 논리적으로 하나로 묶어 MPP와 같은 수퍼컴퓨터로 생각할 수 있고 순차 프로그램과 병렬 프로그램이 공존할 수 있다. 순차 프로그램은 일반 사용자가 워크스테이션에 로긴하여 문서를 에디팅 하거나 프로그램을 컴파일 하는 것과 같은 것을 말한다. 순차 프로그램은 하나의 워크스테이션만을 사용하며 순차 프로그램이 있는 워크스테이션은 병렬 프로그램을 수행할 워크스테이션에서 제외된다. 왜냐하면 병렬 프로그램으로 인해 순차 프로그램의 성능이 저하되는 영향을 주어서는 안되기 때문이다. 이러한 규칙을 지키면서 순차 프로그램과 병렬 프로그램이 공존할 수 있다는 결과가 있다[4]. 이렇게 순차 프로그램과 병렬 프로그램이 공존하게 되면 병렬 프로그램은 순차 프로그램들로부터 영향을 받을 것이다. 왜냐하면 병렬 프로그램이 수행되고 있던 워크스테이션에 사용자가 로긴하여 순차 프로그램을 수행시키게 되면 그 워크스테이션에서는 더 이상 병렬 프로그램이 수행될 수 없게 된다. 정책에 따라서 병렬 프로그램의 우선 순위를 낮추거나 다른 워크스테이션으로 이주시키는 것이 가능하다. 어떤 경우이든 순차 프로그램으로 인하여 병렬 프로그램이 영향을 받는 것은 피할 수 없다.

한 연구에 의하면 하루 중 아무리 바쁜 시간이더라도 20%~60%의 워크스테이션들이 사용되지 않고 있다는 통계가 있다[5]. 이렇게 사용되지 않는 유휴 워크스테이션을 이용하는 것에 대한 연구가 아주 활발하다[6]. 그러나 대부분의 연구는 프로세스 통신과 프로세스 이주등 메커니즘의 구현에 관한 것이었고 실제로 이러한 환경에서 순차 프로그램과 병렬 프로그램이 동시에 수행되었을 때의 성능에 대한 분석은 없었다. NOW에서 이 두 프로그램 사이의 상호 영향에 대한 분석은 UC Berkeley에서 최근에 행해졌다[7]. 그러나 실험에서는 실제 트레이스를 사용하였기 때문에 유휴 워크스테이션의 수, 병렬 프로그램의 병렬도, 병렬 프로그램 수행시간의 평균과 분산 등의 파라미터들이 병렬 프로그램의 성능에 어떠한 영향을 미치는가에 대한 분석이 없었다.

본 논문에서는 이러한 파라미터들의 변화가 병렬 프로그램의 성능에 어떠한 영향을 미치는가에 대해 합성부하를 사용하여 정량적 분석을 하였고 또 여러 개의 병렬 프로그램이 동시에 수행될 때 시간 분할(time-sharing)과 공간 분할(space- sharing)의 두 할당 정책이 어떠한 성능의 차이를 보이는가에 대해 분석하였다. 본 논문에서 살펴볼 내용은 다음과 같다.

- 평균 유휴 워크스테이션 수, 병렬 프로그램 수행시간의 평균과 분산, 병렬 프로그램의 병렬도에 따라 병렬 프로그램의 수행시간이 어떻게 변하는지 알아본다.
- 여러 개의 병렬 프로그램이 두 프로세스 할당 정책(time-sharing, space-sharing)에 따라 할당되었을 때 프로그램들의 성능에 미치는 영향과 병렬프로그램간의 영향에 대해 알아본다.

본 논문의 목적은 위의 내용들에 대해 실험을 통하여 자료를 수집하고 정량적인 분석을 하는 것이다. 이러한 것들을 알아보기 위해 본 연구에서는 NOW 상의 병렬 프로그램 스케쥴링을 시뮬레이션 하였다. 시뮬레이터는 순차 부하와 병렬 부하를 입력으로 받아 병렬 프로그램의 수행 시간을 출력한다. 여기서 사용된 순차 부하는 평균 유휴 워크스테이션 수를 조절하기 편하게 하기 위해 다른 자료들을 참고하여 합성한 순차 부하를 사용하였으며 병렬 부하는 실제로 사용되는 병렬 프로그램의 특성들을 구하기 어려운 이유로 역시 다른 자료들을 참조하여 합성하였다.

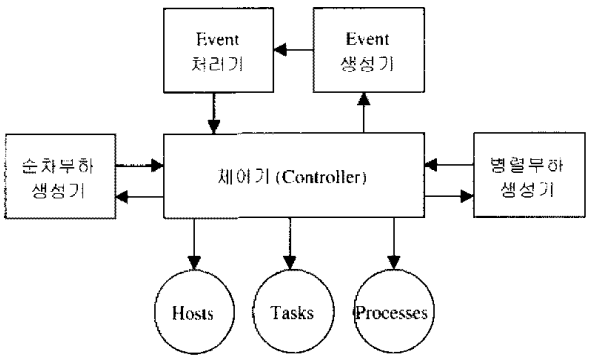
2장에서는 본 논문에서 사용한 시뮬레이터에 대해 알아본다. 시뮬레이터에서 사용한 순차 부하, 병렬 부하와 더불어 병렬 프로그램 스케쥴링을 이루는 할당 정책, 스케쥴링 정책, 이주 정책 등에 대하여 알아본다. 3장에서는 실험 내용과 결과를 다루고 4장에서 결론을 맺는다.

2. 시뮬레이터

본 장에서는 본 논문에서 사용한 시뮬레이터에 대해 알아본다.

2.1 시뮬레이터의 구조

시뮬레이터의 구조는 그림 1과 같다. 순차 부하와 병렬 부하를 입력으로 받아 병렬 프로그램의 수행 시간을 출력하게 된다. 시뮬레이터는 이벤트 처리 방식으로 진행된다. 그림 1에서의 'Event'는 시뮬레이터에서 처리해야 하는 이벤트를 뜻한다. 'Hosts'는 워크스테이션들을 뜻하며 본 논문에서는 때에 따라 워크스테이션, 호스트, Host를 적절히 사용할 것이다. 'Tasks'는 병렬 프로그램들을 뜻하는 것이다. Host와 마찬가지로 병렬 프로그램, 병렬 작업, 병렬 Task를 때에 따라 적절히 사용한 것이다. 'Processes'는 하나의 병렬 프로그램을 이루는 프로세스들을 뜻하며 하나의 병렬 task는 '병렬도' 만큼의 프로세스들로 구성된다. 이중에서 순차부하 생성기, 병렬 부하 생성기, 제어기에 대해서는 뒤에서 자세히 설명할 것이다.



(그림 1) 시뮬레이터의 구조
(Fig. 1) The structure of the simulator

2.2 순차부하 생성기

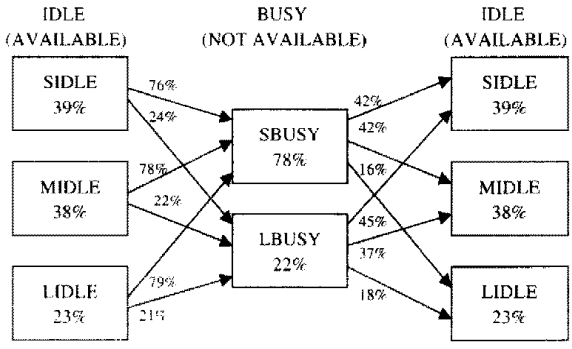
순차부하란 병렬 프로그램에 의해서 생기는 부하를 제외한 사용자에 의해 수행되는 순차 프로그램에 의해 생기는 부하를 말한다. 많은 사람들에 의해서 시스템의 순차부하에 대한 연구가 이루어졌으나 구체적으로 어떤 기준에 의해 시스템이 유휴 상태인지 바쁜 상태인지는 잘 알려져 있지 않다. 다른 논문들에서는 로긴한 사람이 없는 시스템을 유휴 상태라고 정의하기도 하고 1분 동안 시스템에 사용자에 의한 이벤트가 없으면 유휴 상태라고 정의하기도 했다. 본 논문에서는 시스템의 유휴 상태인지 바쁜 상태인지 판단하는 기준에 대해서는 논하지 않는다. 단지 순차부하의 형태를 유휴 상태(가용 Host), 바쁜 상태(불가용 Host)로 제한하였다.

시뮬레이터의 입력이 되는 순차 부하는 두 가지 종류가 있다. 실제로 네트워크에 연결되어 있는 워크스테이션들로부터 트레이스를 얻는 것이 있고 통계에 의한 합성이 있다. 본 연구에서는 후자의 방법을 사용하였다. 왜냐하면 합성에 사용되는 여러 변수들을 - 지속 시간, 상태 변화 확률 - 변화시킴으로써 유휴 워크스테이션들의 평균수를 조절하기 위해서이다. 본 연구에서 사용한 합성은 [5]의 방법을 사용했다. 이에 대해 간단히 알아보면 다음과 같다.

호스트의 상태가 유휴 상태일 때의 평균 지속 시간은 3분, 25분, 300분이다. 이것들을 각각 SIDLE, MIDLE, LIDLE 상태라고 하자. 또 호스트의 상태가 바쁜 상태일 때의 평균 지속 시간은 7분, 55분이다. 각각 SBUSY, LBUSY 상태라고 하자. 이때 상태 지속 시간은 다음 식에 의해 계산된다.

$$T = -t_{avg} \cdot \ln(1 - R)$$

where T : 상태 지속 시간
t_{avg} : 평균 지속 시간
R : $0 < R < 1$인 난수



(그림 2) 상태 전이 확률
(Fig. 2) State transition probabilities

호스트가 어떠한 상태에 있을 확률과 하나의 상태에서 다른 상태로의 전이가 일어날 확률은 그림 2와 같다. 그림 3은 40대의 호스트가 있고 그림 2의 상태 변화 확률과 평균 상태 유지 시간값을 그대로 사용했을

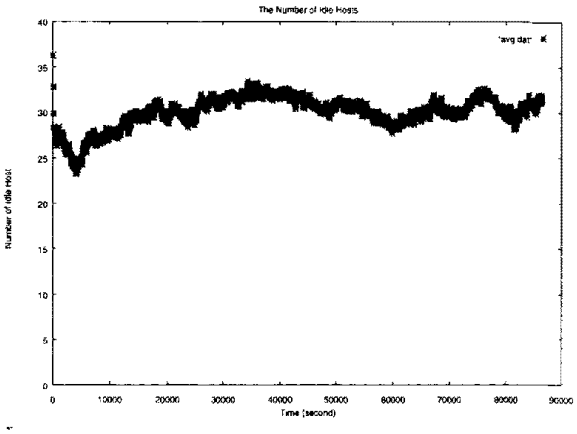
〈표 1〉 호스트 평균 상태 유지 시간과 상태 전이 확률
〈Table 1〉 The average time of a WS's being in a stateand state transition probabilities

SI : SIDLE, MI : MIDLE, LI : LIDLE, SB : SBUSY, LB : LBUSY

평균 유휴 호스트 수	평균 상태 유지 시간 (단위 : 초)				
	SIDLE	MIDLE	LIDLE	SBUSY	LBUSY
35	180	1500	18000	120	1300
33	180	1500	18000	300	2343
30	180	1500	18000	420	3300
28	180	1500	18000	600	4179
26	180	1500	18000	600	4179
	상태 전이 확률				
	SI -> SB	MI -> SB	LI -> SB	SB -> SI	LB -> SI
				SB -> MI	LB -> MI
	SI -> LB	MI -> LB	LI -> LB	SB -> LI	LB -> LI
35	0.76	0.78	0.79	0.42	0.45
				0.42	0.37
	0.24	0.22	0.21	0.16	0.18
33	0.76	0.78	0.79	0.42	0.45
				0.42	0.37
	0.24	0.22	0.21	0.16	0.18
30	0.76	0.78	0.79	0.42	0.45
				0.42	0.37
	0.24	0.22	0.21	0.16	0.18
28	0.76	0.78	0.79	0.42	0.45
				0.42	0.37
	0.24	0.22	0.21	0.16	0.18
26	0.65	0.65	0.65	0.42	0.45
				0.42	0.37
	0.35	0.35	0.35	0.16	0.18

때 매 60초마다 유휴 호스트 수를 나타낸 것이다. 평균 30개 정도로 약 75%의 호스트가 항상 유휴 상태라는 것을 나타내고 있다. 본 연구에서는 표 1과 같이 평균 지속 시간과 확률을 조절하여 평균 유휴 호스트 수를 조절하였다.

호스트 상태 변화 event가 발생하면 제어기는 호스트의 이전 상태를 순차 부하 생성기로 넘긴다. 호스트의 상태에는 앞에서 설명한 SIDLE, MIDLE, LIDLE, SBUSY, LBUSY의 다섯 가지 상태가 있다. 호스트의 상태를 넘겨받은 순차 부하 생성기는 그림 2에서 설명한 대로 다음 상태를 결정하고 결정된 상태에

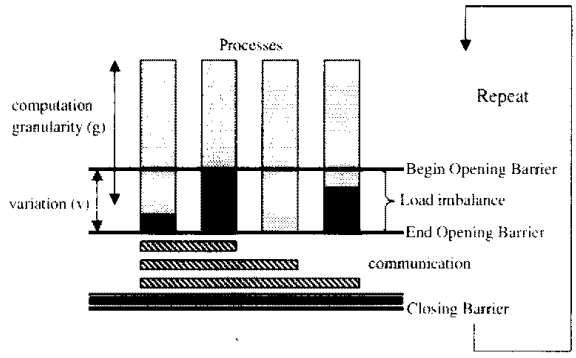


(그림 3) 하루 동안 유휴 상태인 호스트의 수
(Fig. 3) The number of idle WS during a day

따라서 상태 유지 시간을 결정하여 제어기에 넘긴다. 제어기는 호스트의 상태를 순차 부하 생성기로부터 받은 상태로 변화시키고 상태 유지 시간을 호스트에 저장시킨다. 또한 상태 유지 시간 뒤에 발생할 호스트 상태 변화 event를 새로 생성하여 event 생성기에 넘기게 된다.

2.3 병렬부하 생성기

병렬부하란 병렬 프로그램에 의해서 생기는 부하를 말한다. 본 논문에서는 병렬 프로그램들 중에서 Single Program Multiple Data(SPMD)를 위주로 설명할 것이다. 본 논문에서 고려하는 SPMD는 그림 4와 같은 방식으로 동작한다. 병렬 프로그램을 구성하는 각 프로세스들은 각각의 계산을 마치고 통신이 필요한 프로세스들끼리 통신을 한다. 이런 계산과 통신을 합쳐 한 phase라고 한다. 병렬 프로그램은 정해진 수만큼 phase를 반복하면 종료하게 된다.



(그림 4) 병렬 프로그램 SPMD 모델
(Fig. 4) SPMD model of parallel program

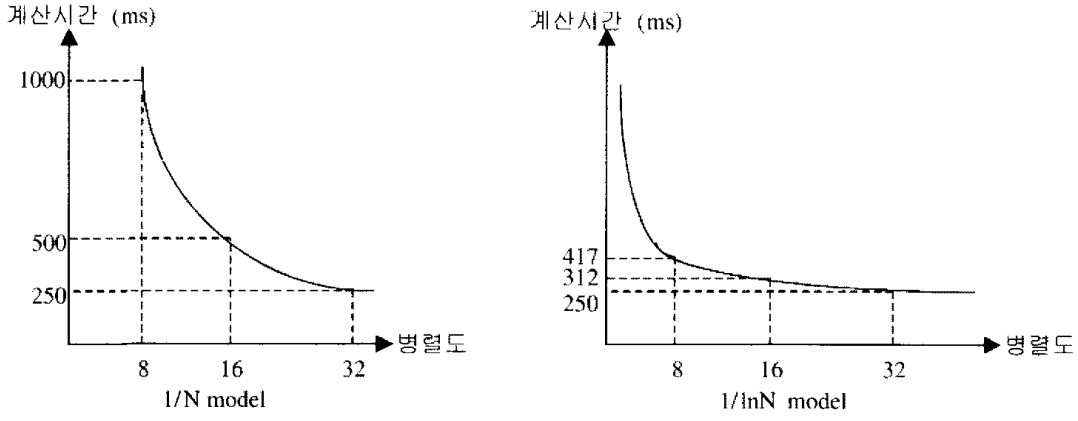
병렬부하 생성기는 병렬 프로그램에 대한 자료를 가지고 있다. 병렬 프로그램 번호, 통신 형태, 평균 계산 시간(g), load imbalance, 통신 시간, 반복 회수, 병렬도 등이다. 병렬 프로그램 번호는 여러 병렬 프로그램을 구분하는 것이다. 통신 형태는 병렬 프로그램을 구성하는 프로세스들이 어떤 형태로 통신하는지에 관련된 것으로 BARRIER, NEWS, TRANSPOSE 등이 있을 수 있다[8]. 평균계산 시간(g)은 한 phase 내에서 각 프로세스가 수행해야 할 계산의 평균 시간을 말한다.

Load-imbalance는 한 phase에서 가장 짧은 계산 시간과 가장 긴 계산 시간의 차이를 만드는 변수로서 본 연구에서는 0.0, 0.25, 0.5, 1.0, 1.5, 2.0 여섯 단계로 나누었다[8]. Load-imbalance에 따라서 v는 0 ~ 2g 사이의 값을 갖게 된다. 그래서 한 프로세스가 한 phase에서 가지는 계산 시간은 다음과 같이 결정된다.

$$T_{comp} = g \mp v/2$$

where T_{comp} : 계산 시간
g : 평균 계산 시간
v : 계산 시간 변화량

통신 시간은 한 phase 내에서 모든 프로세스가 통신을 완료하는데 걸리는 시간이다. 반복 회수는 병렬 프로그램이 몇 번의 phase를 거쳐야 종료하는가를 나타낸다. 병렬도는 하나의 병렬 프로그램이 몇 개의 프



(그림 5) 병렬도와 한 phase 내의 평균 계산 시간과의 관계
(Fig. 5) The relationship between parallelism and average computation time in a phase

프로세스들로 이루어져 있는가를 나타낸다.

본 연구에서는 복잡도를 줄이기 위해 병렬 프로그램의 특성을 고정시켰다. 우선 병렬 프로그램의 병렬도는 32, 16, 8 세 가지로 제한하고 통신 형태는 BARRIER만 고려하며 한 phase 내에서 평균 계산 시간은 병렬도가 32일 때 250ms로 고정하였다.

같은 양의 계산을 하기 위해서 병렬도가 낮아짐에 따라 각 프로세스가 한 phase 내에서 수행해야 할 평균 계산 시간이 늘어나게 했다. SPMD 병렬 프로그램의 여러 특성에 따라 다른 결과들이 나올 수 있으나 본 연구에서는 병렬도와 한 phase 내의 평균 계산 시간과의 관계가 그림 5와 같은 두 가지 모델만을 고려하였다. 두 모델을 고려한 것은 병렬도가 작아지면서 한 phase 내의 평균 계산 시간이 증가되는 것 외에 프로세스간 통신 회수나 통신량 등의 변화를 반영하기 위해서이다. 병렬도가 16일 때 1/N 모델의 경우 평균 계산 시간은 500ms, 1/lnN 모델의 경우 312ms가 된다. 또 병렬도가 8일 때 1/N 모델에서는 1000ms, 1/lnN 모델에서는 417ms가 된다. 통신 시간은 아주 빠른 네트워크를 가정하여 10μs로 고정시켰다. 반복회수는 1000회로 고정시켰다.

시뮬레이터가 초기화할 때 병렬 부하 생성기는 병렬 프로그램들의 특성들을 읽어들인다. 병렬 프로그램 시작 event가 처리되면서 새로운 Task를 생성하고 Task는 병렬 프로그램의 병렬도, 시작 시간, 종료 시간, phase 회수 등의 변수를 가진다. 병렬도는 프로세스들을 생성할 때 사용되며 시작 시간과 종료 시간은 병렬 프로그램이 수행되는데 걸린 시간을 측정할 때 사용된다. 또 phase 회수는

병렬 프로그램이 진행되는 동안 몇 번의 phase를 수행했는지 검사할 때 사용된다.

병렬 부하 생성기는 제어기로부터 병렬 프로그램 번호와 프로세스 번호를 넘겨받아 한 phase 내에서 프로세스가 수행해야 하는 계산시간이나 통신 시간을 제어기로 돌려준다. 제어기는 받은 시간을 사용하여 한 phase 내의 계산 종료 event나 통신 종료 event를 생성한다.

2.4 제어기

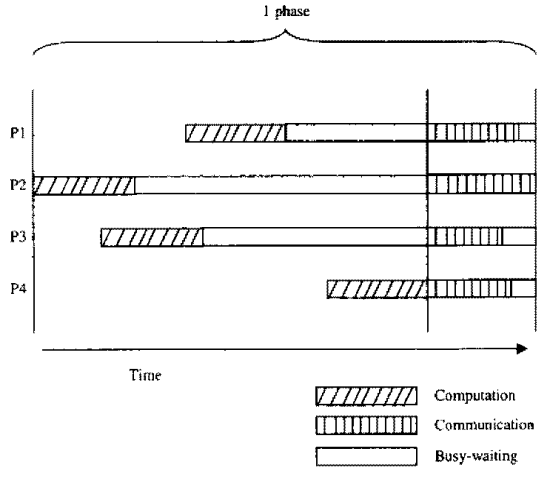
제어기는 시뮬레이터에서 가장 중요한 부분으로서 다음과 같은 역할을 한다.

- 병렬 프로그램 스케쥴링
- Host, Task, Process 관리
- 새로운 event 생성

이중에서 병렬 프로그램 스케쥴링에 대해서 구체적으로 살펴보겠다.

주로 병렬 프로그램을 수행하는 MPP와 같은 수퍼컴퓨터에서는 병렬 프로그램을 스케쥴하기 위해 만들어진 운영체제가 있다. 그런 스케쥴링 기술 중에 잘 알려진 것이 coscheduling[9]이다. 이 기술은 병렬 프로그램의 수행 성능을 높일 수 있음이 알려져 있다[10].

그림 6과 같이 병렬도가 4인 병렬 프로그램이 있을 때 프로세스들끼리 빈번하게 통신하면서 동작한다고 보자. 시스템의 스케쥴 정책이 각 프로세스를 개별적으로 스케쥴 하는 것(local scheduling)이었다면 한 phase 내에서 각각의 프로세스가 계산을 시작하는 시각이 모두 틀릴 것이다. 그리하여 모든 프로세스가 한 phase에서 계산을 마치는데 많은 시간이 걸리게 된다. 이런 현상이 반복되면 병렬 프로그램의 성능은 아주 나빠지게 된다.

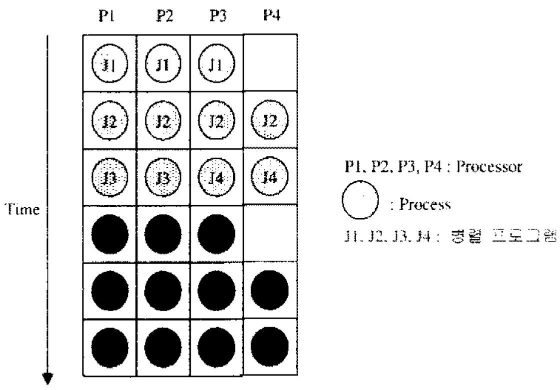


(그림 6) 부적당한 병렬 프로그램 스케쥴링 예
(Fig. 6) An example of inappropriate parallel program scheduling

Ousterhout는 이런 단점을 극복하는 방법으로 coscheduling을 제안했다. 즉 병렬 프로그램을 이루는 모든 프로세스들이 동시에 계산하고 통신하도록 하여 메시지를 기다리면서 블록되는 일이 없도록 한 것이다. 또한 동시에 여러 병렬 프로그램을 수행할 수 있도록 했다. Ousterhout가 제시한 coscheduling 알고리즘

들(Matrix, Contiguous, Un divided)을 살펴보면 이런 사실을 알 수 있다. 가장 간단한 모델인 Matrix를 예를 들면 쉽게 알 수 있는데 그 방법은 그림 7에 나타나 있다. 이 예는 사용할 수 있는 프로세서가 4개이며 병렬 프로그램 J1은 세 개의 프로세스로 구성되어 있고 J2는 네 개의 프로세스로 구성되어 있다. 또 J3과 J4는 각각 두 개의 프로세스로 구성되어 있다. 각 time slot에 있는 프로세스들은 동시에 스케쥴링 되므로 그림 6과 같은 현상은 생기지 않는다. J3과 J4는 병렬도가 2이기 때문에 두 병렬 프로그램은 한 time slot에 들어갈 수 있다. 이와 같이 병렬 프로그램 4개가 Round-Robin 방식으로 context switch에 의해 교대로 또는 동시에 수행하고 있음을 알 수 있다.

본 연구에서는 NOW 상에서의 coscheduling을 고려하였다. MPP와 달리 NOW에서는 사용 가능한 프로세서(호스트)의 수가 변하고 호스트 상태의 변화에 따라 사용 가능 여부도 달라지게 된다. 또한 MPP에서의 coscheduling에서는 고려하지 않았던 프로세스 이주가 발생할 수 있다. 이런 이유들로 인해 MPP에서의 coscheduling을 NOW에서는 그대로 사용할 수 없다. MPP에서의 coscheduling과 마찬가지로 NOW에서의 coscheduling은 병렬 프로그램의 각 프로세스를 적당히 사용 가능한 유휴 호스트들에 할당하는 것과 병렬 프로그램의 스케쥴링, 그리고 프로세스 이주가 발생했을 때의 처리 방식들에 대한 정책을 가져야 한다. 이를 하나씩 살펴보도록 하겠다.

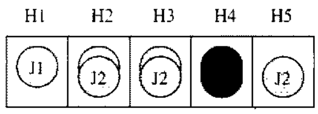


(그림 7) 병렬 프로그램 4개가 수행중인 모습
(Fig. 7) 4 parallel programs running simultaneously

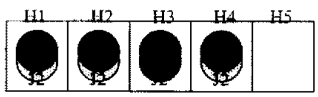
할당 정책

NOW 환경에서는 병렬 프로그램이 이용할 수 있는 호스트들이 변하기 때문에 MPP 환경에서와 같이 정적으로 할당할 수 없다. NOW에서 사용 가능한 호스트는 유휴 상태인 것들뿐이다. 여기서 유휴 상태인 호스트도 두 가지 경우가 있다. 유휴 호스트 중에서 병렬 프로그램의 프로세스가 동작중인 것이 있고 그렇지 않은 것이 있다. 전자를 단순히 유휴 호스트, 후자를 순수 유휴 호스트라고 하겠다. 즉 할당 대상이 될 수 있는 호스트는 순수 유휴 호스트와 유휴 호스트이다.

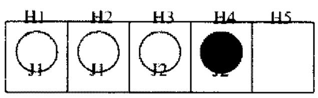
그림 8과 같이 다섯 개의 유휴 호스트(H1 ~ H5)에 병렬도가 4인 두 개의 병렬 프로그램(J1, J2)의 프로세스들 중 일부만 같은 호스트들에 할당되면 H2, H3, H4에서 J1과 J2가 교대로 수행되는 동안 H1, H5의 프로세스는 그림 6에서 Ousterhout가 지적했던 대로 메시지를 기다리며 블록되는 경우가 발생하여 coscheduling의 장점이 사라지게 된다. 그러므로 NOW에서는 그림 9와 같이 병렬 프로그램들을 완전히 겹치게 호스트들에 할당하여 J1, J2를 교대로 수행시키는 시간 분할(time-sharing)을 하거나 그림 10과 같이 완전히 분리시켜 할당하여 공간 분할(space-sharing)을 한다. 그림 10에서는 병렬도를 2로 낮추어 할당한 것이다. 이를 정리하면 다음과 같이 요약할 수 있다.



(그림 8) 부분 중복 할당
(Fig. 8) Partly overlapped assignment



(그림 9) 완전 중복 할당
(Fig. 9) Completely overlapped assignment



(그림 10) 완전 분리 할당
(Fig. 10) Completely separated assignment

```
if (병렬로 수 순수 유휴 워크스테이션 수)
    모든 프로세스를 각각 유휴 워크스테이션에 할
    당한다
else if (수행중인 병렬 프로그램이 있는가?)
        수행중인 병렬 프로그램의 프로세스들이 있
        는 워크스테이션들에 각각 할당
    else
        병렬 프로그램을 할당할 수 없다.
```

병렬 프로그램의 프로세스들을 할당할 호스트들을 선정하는 방법도 여러 가지가 있을 수 있다. 첫째, 단순히 유휴 호스트들의 리스트를 관리하면서 병렬 프로그램이 시작될 때 차례로 할당하는 방법이 있다. 이 방법은 단순하여 구현하기 쉽다는 장점이 있지만 바쁜 상태의 빈도가 많은 호스트에 할당할 가능성이 있어 병렬 프로그램의 성능을 저하시킬 수 있는 단점이 있다. 둘째, 호스트의 유휴 상태, 바쁜 상태의 빈도를 보고 유휴 상태가 많은 순서로 유휴 호스트 리스트를 관리하여 할당하는 방법이 있다. 이 방법은 병렬 프로그램 성능에 좋은 영향을 끼칠 수 있지만 구현이 복잡하다는 단점이 있다. 본 연구에서는 구현의 편리함을 위해 첫 번째 방법으로 할당을 하였다. 또한 동시에 수행되는 병렬 프로그램의 병렬도는 모두 같다고 가정하였다.

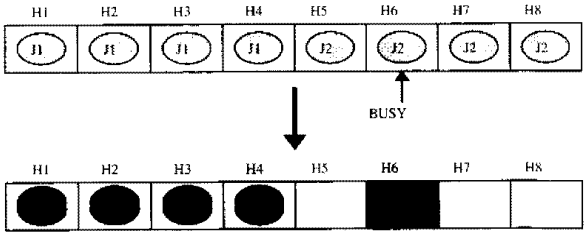
스케쥴링 정책

먼저 하나의 병렬 프로그램에 대해서 알아보면 병렬 프로그램의 모든 프로세스들은 동시에 계산을 시작하고 모든 프로세스가 계산을 마치면 동시에 통신을 시작한다. 역시 통신이 모두 끝나면 그 다음 계산을 수행한다. 이런 식으로 정해진 반복 회수만큼 phase를 반복한다. 만약 두 개 이상의 병렬 프로그램이 있을 때 완전히 분리되어 할당된 경우는 하나의 병렬 프로그램에서 썼던 방식을 그대로 사용하고 완전히 겹쳐서 할당된 경우는 병렬 프로그램 사이는 Round-Robin 방식으로 교대로 스케쥴링하고 병렬 프로그램 자체는 앞서 설명한 하나의 병렬 프로그램에서 썼던 방식을 따른다.

Round-Robin 방식으로 병렬 프로그램들이 돌아가면서 스케쥴링이 이루어 질 때 context switch가 발생하는데 context switch가 발생하는 빈도는 100ms로 고정시켰다[11]. 또한 context switch가 이루어지는 시간을 200μs로 고정하였다[8].

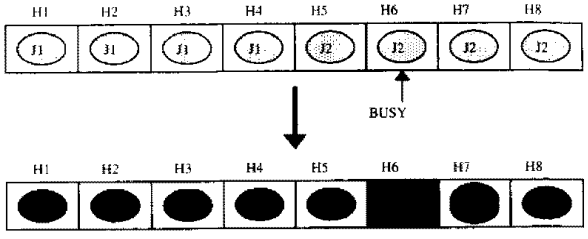
이주 정책

MPP에서 coscheduling은 병렬 프로그램의 각 프로세스들을 프로세서에 할당하고 할당된 프로세스들은 병렬 프로그램이 종료할 때까지 다른 프로세서로 이주하지 않는다. 왜냐하면 개인 사용자에 의한 순차 프로그램 같은 것이 없기 때문이다. 그러나 NOW 환경에서는 병렬 프로그램이 수행 중이던 유휴 호스트에 사용자가 로긴하여 순차 프로그램을 수행시킬 수 있으며 그러면 그 호스트는 더 이상 유휴 호스트가 아니게 된다. 그러므로 병렬 프로그램의 수행 우선 순위가 낮아지게 하거나 다른 유휴 호스트로 이주하게 해야 한다. 본 연구에서는 우선 순위를 낮추는 것은 고려하지 않고 다른 유휴 호스트로 이주시키는 것만을 고려한다. 우선 순위를 낮추는 것은 병렬 프로그램의 프로세스를 이주시키지 않아도 되므로 이주 비용 절감의 이점이 있지만 사용자의 순차 프로그램의 우선 순위보다 크게 낮기 때문에 프로세서 할당을 거의 받지 못하게 되는 단점이 있다. 따라서 병렬 프로그램의 다른 프로세스들은 계속해서 블록되어 있는 프로세스로부터 메시지를 받기 위해 블록되어야 한다. 반면 이주를 하는 경우는 이주 비용이라는 추가적인 비용이 필요하지만 이주가 끝나면 곧바로 수행을 계속할 수 있다는 장점이 있다.



(그림 11) 다른 병렬 프로그램이 있는 곳으로 완전히 중복되게 이주

(Fig. 11) Completely overlapped migration to WS's where other parallel program runs



(그림 12) 같은 병렬 프로그램의 다른 프로세스가 있는 곳으로 이주

(Fig. 12) Migration to WS where another process of the same parallel program runs

NOW 상에서 병렬 프로그램이 수행되다가 한 호스트가 더 이상 유휴 상태가 아니게 되었을 때 해당 호스트의 병렬 프로세스 또는 프로세스들은 이주를 해야 한다. 이때 여러 가지 경우가 가능하다. 우선 순수 유휴 호스트로 이주하는 것이다. 이 경우는 이주 후에 스케쥴링에 별다른 문제점이 없다. 그러나 순수 유휴 호스트가 없는 경우는 세 가지의 방법이 있을 수 있다. 첫째, 그림 11과 같이 이주되는 프로세스가 속한 병렬 프로그램(J2)의 모든 프로세스들을 완전히 분리되어 수행중인 병렬 프로그램(J1)의 프로세스들이 있는 호스트들로 옮기는 방법이다. 이 방법은 적어도 하나 이상의 병렬 프로그램이 완전히 분리되어 수행 중이어야 한다는 가정이 필요하다. 만약 완전 분리되어 수행 중인 병렬 프로그램이 없다면 두 번째 방법을 사용하거나 세 번째 방법을 사용해야 한다. 두 번째, 그림 12와 같이 이주되는 프로세스가 속한 병렬 프로그램(J2)의 프로세스들이 있는 곳 중에서 한 곳으로 이주하는 방법이다. 이것은 어떤 경우의 이주가 발생해도 진행이 가능하지만 앞에서 설명한 그림 6과 같은 문제가 발생한다. 병렬 프로그램마다 다르겠지만 최악의 경우 병렬 프로그램의 모든 프로세스 중에 두 개만 같은 호스트에서 동작하더라도 병렬 프로그램 전체의 성능이 저하될 수 있다. 세 번째, 순수 유휴 호스트가 생길 때까지 기다렸다가 새로 생긴 순수 유휴 호스트로 이주시키는 방법이다. 이 방법도 이주가 발생하는 모든 경우에 진행이 가능하지만 새로이 순수 유휴 호스트가 생기지 않으면 계속 블록되어야 한다는 단점이 있다.

본 연구에서는 세 번째 방법을 택했다. 왜냐하면 첫 번째 경우는 하나의 순수 유휴 호스트가 없어서 병렬 프로그램의 모든 프로세스를 이주시켜야 하는 경우가 발생하고 곧 새로이 순수 유휴 호스트가 생겨도 부하균형(load balancing)을 하지 않기 때문에 충분한 유휴 호스트가 있음에도 두 병렬 프로그램이 겹쳐서 수행해야 하는 불합리한 점이 있기 때문이고 두 번째 경우는 앞서 설명한 대로 coscheduling의 장점을 살릴 수 없기 때문이다.

프로세스가 이주하는데 필요한 시간, 즉 이주 비용은 시스템 성능뿐 아니라 네트워크의 속도에 따라 다르다. 예를 들면 Ethernet에 연결된 전형적인 UNIX 호스트의 경우 1~4초 정도가 걸리고 Sprite[10]에서는 0.33초, ATM으로 연결된 GLUNIX[12]에서는 0.25~0.5초 정도가 걸린다고 한다. 또한 프로세스가 사용하고 있는 메모리의 크기에 따라서 1Mbyte당 0.1초에서 2초 정도가 추가로 소요된다고 한다[13]. 본 연구에서는 아주 좋은 성능의 호스트와 매우 빠른 네트워크를 가정하여 이주 비용을 0.2초로 고정시켰다.

3. 실 험

본 연구에서는 앞서 설명했던 시뮬레이터를 사용하여 세 가지 실험을 하였다. 모든 실험은 40대의 호스트 중 평균 유휴 호스트의 수가 35, 33, 30, 28, 26개일 때 load-imbalance 별로 병렬 프로그램을 수행시켜 병렬 프로그램의 수행 시간을 측정하였다. 통계적인 값을 구하기 위하여 이런 측정을 각각 100번씩 수행시켜 최고 시간과 최저 시간을 뺀 나머지를 평균하였다. 이 수행 시간을 바탕으로 병렬 프로그램 1개를 수행시켰을 때 병렬도 32와 병렬도 16의 성능을 비교하였고 병렬 프로그램 2개를 수행시켜 병렬도 32로 시간 분할한 것과 병렬도 16으로 공간 분할한 것의 성능을 비교하였다. 또 병렬 프로그램 4개의 병렬도와 할당 정책을 변화하며 성능의 변화를 비교하였다.

모든 실험은 다음과 같은 가정 하에서 이루어 졌다. 먼저 NOW를 구성하는 워크스테이션들은 40대이며 모두 같은 기종의 시계이며 운영체제도 같다. 그리고 병렬 프로그램을 스케쥴링 하는데 소요되는 모든 오버헤드는 무시한다. 실제로는 병렬 프로그램을 스케쥴링 하기 위해 많은 오버헤드가 있을 것이다. 예를 들면 유휴 워크스테이션들을 관리하는데 소요되는 비용, 병렬 프로그램의 프로세스들을 할당하는 비용, 프로세스 이주가 발생했을 때 소요되는 비용등 많은 부분이 있을 것이다.

3.1 실험 내용

앞에서 설명한 시뮬레이터로 병렬 프로그램 1개를 수행시키는 실험과 병렬 프로그램 2개를 동시에 수행시키는 실험, 병렬 프로그램 4개를 동시에 수행시키는 실험을 하였다. 한 가지씩 살펴보면 다음과 같다.

실험 1 : 병렬 프로그램 1개를 수행시키는 실험이다. 이것은 평균 유휴 호스트의 개수, 병렬 프로그램 계산 시간의 평균(g)과 분산(v)이 변함에 따라서 병렬 프로그램의 수행 시간이 어떻게 변하는지 알아보기 위한 것이다. 병렬 프로그램의 병렬도가

32인 경우와 병렬도가 16인 경우를 실험하였다. 이 실험에서는 또 어느 시점에서 어느 정도의 병렬도를 가진 프로그램이 더 좋은 성능을 나타내는지 알 수 있을 것이다.

실험 2 : 병렬 프로그램 2개를 동시에 수행시키는 것으로 두 가지 종류가 있다. 하나는 그림 13과 같이 두 병렬 프로그램(J1, J2)의 병렬도를 32로 하여 완전히 겹쳐서 수행시키는 시간 분할(time-sharing)이고 다른 하나는 그림 14와 같이 병렬도를 16으로 낮추어서 완전히 분리하여 수행시키는 공간 분할(space- sharing)이다. 이 실험에서는 여러 병렬 프로그램이 동시에 수행될 때 시간 분할의 성능이 좋은지 공간 분할의 성능이 좋은지 알 수 있을 것이다.

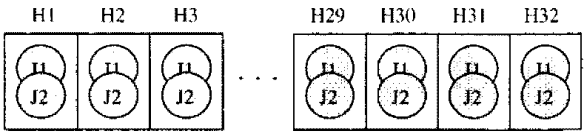
- 실험 3 : 병렬 프로그램 4개를 동시에 수행시키는 실험이다. 이 실험에는 세 종류를 고려하였다. 그림 15와 같이 4개의 병렬 프로그램의 병렬도를 32로 하고 모두 겹쳐서 수행시키는 것과 그림 16과 같이 병렬도를 모두 16으로 하여 두 개씩 겹쳐서 분리시켜 수행시키는 것, 그리고 마지막으로 그림 17과 같이 병렬도를 모두 8로 하여 모두 분리시켜 수행시키는 것이다. 이 실험에서는 실험 2에서 알 수 있는 내용을 알 수 있을 뿐 아니라 같은 조건에서 병렬 프로그램 하나를 수행시킬 때보다 여러 개를 수행시킬 때 얼마만큼 성능이 변하는지를 알 수 있을 것이다.

3.2 실험 결과

본 연구에서 행한 실험은 Sun Sparc20, Solaris 2.5에서 이루어졌으며 시뮬레이터는 C++ 코드 4000라인 정도로 구현되었고 GNU C++ 컴파일러인 g++로 컴파일 하였다.

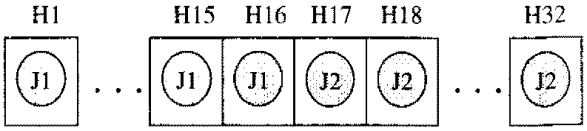
실험 1, 2, 3의 결과는 기준값에 대한 느려짐(Slowdown)의 정도로 나타내었다. 기준값은 MPP환경과 같은 순차부하의 영향이 전혀 없고 병렬 프로그램의 병렬도가 32이며 load-imbalance가 0.0일 때 병렬 프로그램 하나를 수행시켜 걸린 시간으로 250초 정도이다.

그림 18은 순차 부하의 영향에 따라 평균 유휴호스트의 수가 35, 33, 30, 28, 26일 때 병렬도가 32인 병렬 프로그램 하나를 수행시켰을 때 수행시간의 느려짐을 나타낸 것이다. 그림 18을 보면 평균 유휴 호스트



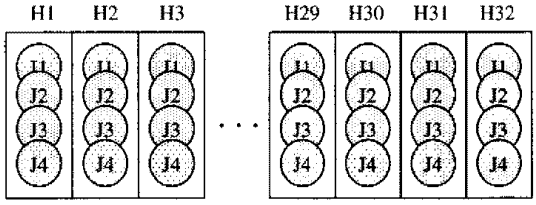
(그림 13) 병렬도를 32로 하여 두 병렬 프로그램을 완전히 중복되게 할당

(Fig. 13) Completely overlapped assignment of 2 parallel programs with parallelism 32



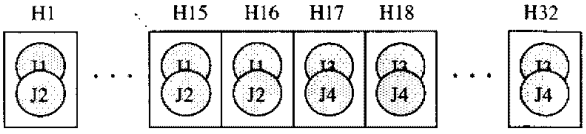
(그림 14) 병렬도를 16으로 하여 두 병렬 프로그램을 완전히 분리시켜 할당

(Fig. 14) Completely separated assignment of 2 parallel programs with parallelism 16



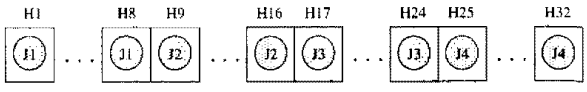
(그림 15) 병렬도를 32로 하여 네 개의 병렬 프로그램을 완전히 중복되게 할당

(Fig. 15) Completely overlapped assignment of 4 parallel programs with parallelism 32



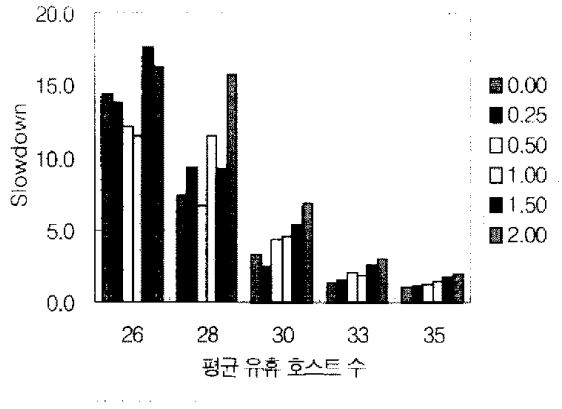
(그림 16) 병렬도를 16으로 하여 네 개의 병렬 프로그램을 두 개씩 완전히 중복되게 할당

(Fig. 16) Completely overlapped assignment of 2 parallel programs and separated assignment of 2 couple of parallel programs with parallelism 16



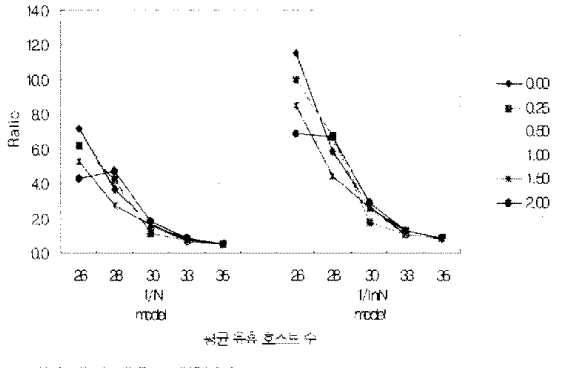
(그림 17) 병렬도를 8로 하여 네 개의 병렬 프로그램을 완전히 분리하여 할당

(Fig. 17) Completely separated assignment of 4 parallel programs with parallelism 8



(그림 18) 병렬도 32인 병렬 프로그램 1개를 수행시켰을 때의 느려짐

(Fig. 18) The slowdown when 1 parallel program(parallelism = 32) is executed



(그림 19) 병렬 프로그램 1개, 병렬도 32의 느려짐(그림 18)과 병렬도 16의 느려짐과의 비율

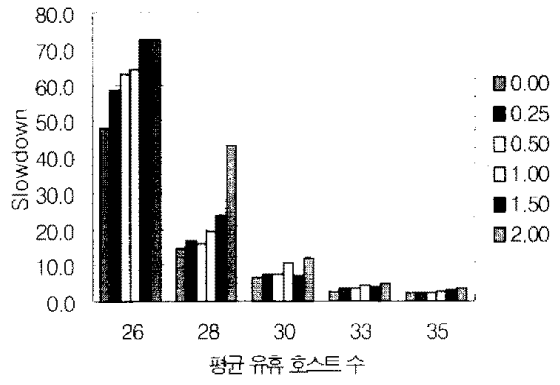
(Fig. 19) The ratio of slowdown when 1 parallel program (parallelism = 32) is executed(Fig 18) to slow down when 1 parallel program(parallelism = 16) is executed

의 수 병렬도보다 낮아지는 30부터 전반적으로 병렬 프로그램의 성능이 급격히 나빠짐을 알 수 있다. 가장 나쁜 경우는 평균 유휴 워크스테이션 수가 26이고 load-imbalance가 1.5일 때로서 기준값보다 17.6배가 걸렸다. 평균 유휴 워크스테이션 수가 35, 33일 때 계산 시간의 분산(v)이 클수록 성능이 나쁜 모습을 볼 수 있다. 그러나 병렬도보다 적은 26, 28, 30에서는 계산 시간의 분산(v)의 영향보다는 평균 유휴 워크스테이션 수에 의해 많은 영향을 받았다. 이것은 계산 시간의 분산(v)에 의한 계산량의 차이와 병렬 프로그램이 블록 당하는 [illegible] 생기는 현상이라고 할 수 있다.

병렬도를 32로 했을 때 병렬도 16보다 얼마나 더 많은 시간이 걸리는지 알아보기 위해 병렬도 32의 느려짐과 병렬도 16의 느려짐의 비율을 구해보았다. 그림 19의 1/N 모델 부분은 그림 18과 병렬도가 16인 1/N 모델의 병렬 프로그램의 느려짐과의 비율이고 1/lnN 모델 부분은 그림 18과 병렬도 16인 1/lnN 모델의 병렬 프로그램의 느려짐과의 비율을 나타낸 것이다. 평균 유휴 호스트 수가 26이고 load-imbalance가 0.0일 때 병렬도 32의 느려짐을 병렬도 16의 느려짐으로 나누는 방식으로 각 load-imbalance와 평균 유휴 호스트에 대하여 그림 19에 나타내었다. 그림 19를 보면 평균 유휴 호스트 수가 33, 35일 때까지는 병렬도가 32인 것보다 병렬도 16인 것의 느려짐이 작게는 1, 크게는 1.8배정도 커서 성능이 좋지 않았지만 30부터는 더 좋은 것을 알 수 있다. 이것은 병렬도가 32인 프로그램은 평균 유휴 호스트 수가 병렬도 이하로 떨어지면 순차 부하에 의해서 유휴 상태이던 호스트가 자주 바쁜 상태로 변하고 이에 따라 병렬 프로그램의 프로세스들이 자주 이주를 해야 하는 일이 발생하지만 순수 유휴 호스트가 없기 때문에 병렬 프로그램의 수행이 블록 당하기 때문에 생기는 현상이라고 할 수 있다.

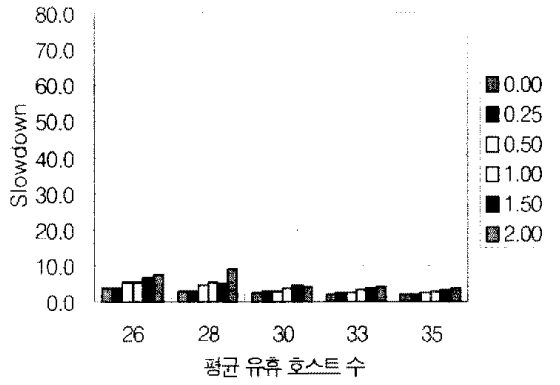
그림 20은 병렬도가 32인 병렬 프로그램 2개를 동시에 수행시켰을 때 즉, 두 병렬 프로그램을 완전히 겹쳐서 수행시켜 시간 분할을 했을 때 수행시간의 느려짐을 나타낸 것이다. 평균 유휴 호스트 수가 35일 때의 느려짐은 2배에서 3.6배로 두 개의 병렬 프로그램이 번갈아 수행하여 생기는 느려짐과 계산 시간 분산(v)의 영향만 있을 뿐 순차부하에 의한 느려짐은 거의 없다. 그러나 26일 때의 느려짐은 47.7배에서 72.7배로 두 병렬 프로그램이 번갈아 수행하면서 생기는 느려짐보다는 순차부하에 의한 느려짐이 더 크다. 그림 21과 그림 22는 병렬도를 16으로 하여 두 병렬 프로그램을 완전히 분리시켜 수행시켜 시간 분할을 했을 때 수행시간의 느려짐을 나타낸 것이다. 두 모델 모두 평균 유휴 워크스테이션 수가 30이하가 되면서 느려짐의 정도가 커졌다. 1/N 모델의 경우 3.6배에서 9배의 느려짐이 있었다. 이것은 병렬도 16인 두 병렬 프로그램이 순차부하의 영향만을 받아 생기는 영향이다.

병렬도를 32로 하여 두 프로그램을 완전히 겹치게 했을 때 병렬도를 16으로 하여 두 프로그램을 완전히



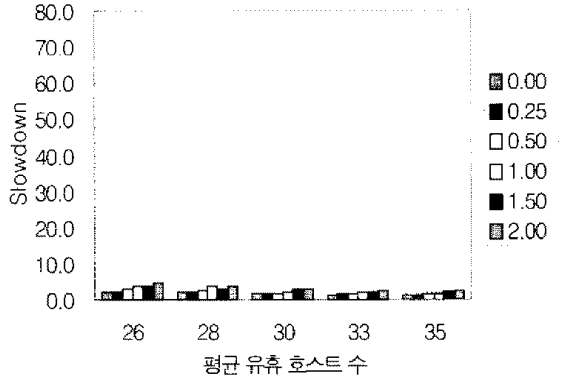
(그림 20) 병렬도 32인 병렬 프로그램 2개를 동시에 수행시켰을 때의 느려짐

(Fig. 20) The slowdown when 2 parallel programs(parall -elism = 32) are executed simultaneously



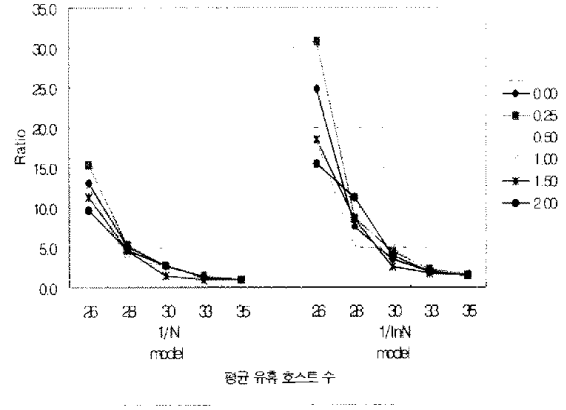
(그림 21) 병렬도 16, 1/N 모델의 병렬 프로그램 2개를 동시에 수행시켰을 때의 느려짐

(Fig. 21) The slowdown when 2 parallel programs(parall -elism = 16, 1/N model) are executed simult aneously



(그림 22) 병렬도 16, 1/lnN 모델의 병렬 프로그램 2개를 동시에 수행시켰을 때의 느려짐

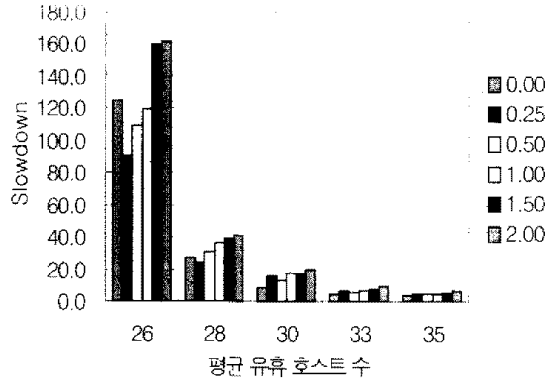
(Fig. 22) The slowdown when 2 parallel programs(paral -lelism = 16, 1/lnN model) are executed sim ultaneously



(그림 23) 병렬 프로그램 2개, 병렬도 32의 느려짐(그림 20)과 병렬도 16의 느려짐(그림 21, 그림 22)과의 비율

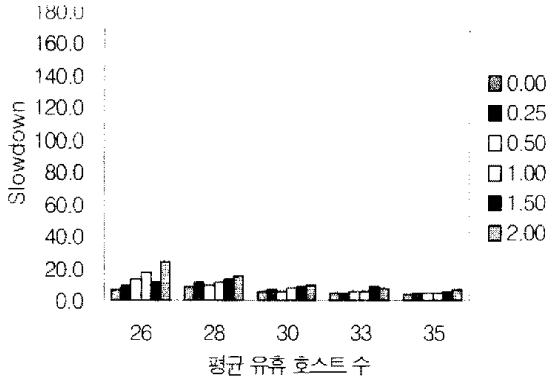
(Fig. 23) The ratio of slowdown when 2 parallel programs (parallelism = 32) are executed simultaneou-sly(Fig 20) to slowdown when 2 parallel progr -ams(parallelism = 16) are executed simulta -neously(Fig 21, Fig 22)

분리 시켰을 때보다 얼마나 더 많은 시간이 걸리는지 알아보기 위해 병렬도 32의 느려짐과 병렬도 16의 느려짐의 비율을 구해보았다. 그림 23의 1/N 모델 부분은 그림 20과 그림 21의 비율이고 1/lnN 모델 부분은 그림 20과 그림 22의 비율을 나타낸 것이다. 그림 23을 나타낸 방법은 그림 19에서 설명한 방법과 같다. 그림 23을 보면 평균 유휴 호스트 수가 35, 33일 때는, 즉 병렬도보다 많을 때 1/N 모델의 경우 0.96에서 0.99의 값이 나와서 병렬도를 높여서 겹치게 수행시키는 것이 극소한 차이로 좋았으나 30이하부터는 1.5에서 15.4의 값이 나와 시간 분할을 하는 것 보다 공간 분할하는 것이 더 좋음을 알 수 있다. 이것은 병렬 프로그램들이 완전히 겹쳐서 수행될 경우 두 프로그램이 번갈아 수행해야 하므로 context switch등에 의해 서



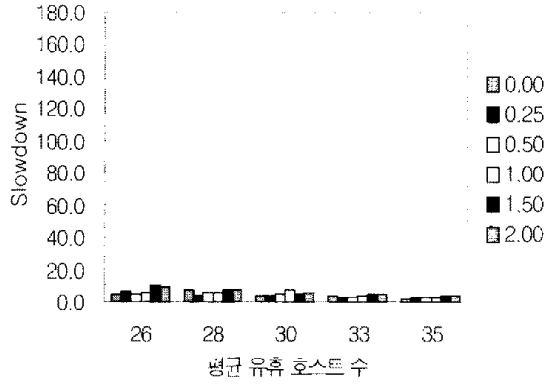
(그림 24) 병렬도 32인 병렬 프로그램 4개를 동시에 수행시켰을 때의 느려짐

(Fig. 24) The slowdown when 4 parallel programs(parall -elism = 32) are executed simultaneously



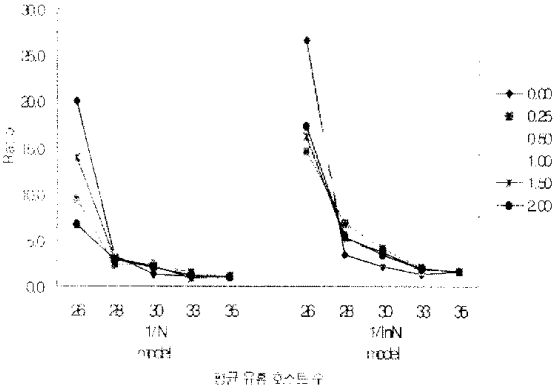
(그림 25) 병렬도 16, 1/N 모델의 병렬 프로그램 4개를 동시에 수행시켰을 때의 느려짐

(Fig. 25) The slowdown when 4 parallel programs(parall -elism = 16, 1/N model) are executed simult aneously



(그림 26) 병렬도 16, 1/lnN 모델의 병렬 프로그램 4개를 동시에 수행시켰을 때의 느려짐

(Fig. 26) The slowdown when 4 parallel programs(parall -elism = 16, 1/lnN model) are executed simu ltaneously

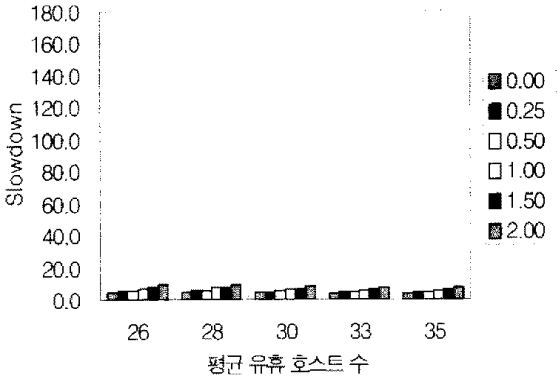


(그림 27) 병렬 프로그램 4개, 병렬도 32의 느려짐과 병렬도 16의 느려짐과의 비율

(Fig. 27) The ratio of slowdown when 4 parallel progra-ms(parallelism = 32) are executed simultane -ously(Fig 24) to slowdown when 4 parallel pr -ograms(parallelism = 16) are executed simu -ltaneously(Fig 25, Fig 26)

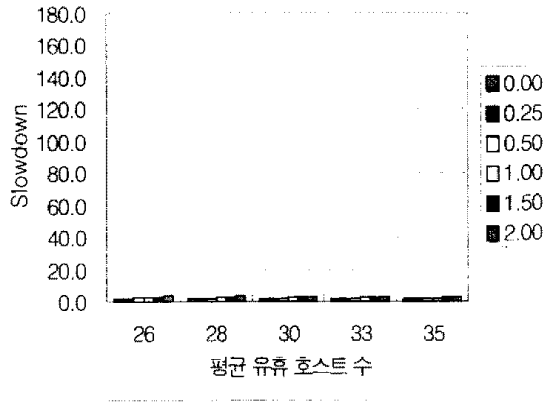
로 감소하는 양이 적어질 뿐만 아니라 응답 부하에 의해 이용할 수 있는 워크스테이션의 수가 줄어들기 때문이다.

그림 24는 병렬도 32인 병렬 프로그램 4개를 모두 완전히 겹쳐서 수행시켰을 때 수행 시간의 느려짐을 나타낸 것이다. 여기에서도 평균 유휴 호스트의 개수가 30개 이하가 되면서 성능이 많이 나빠지는 모습을 볼 수 있다. 또한 28일 때의 느려짐 보다 26일 때의 느려짐이 90.3배에서 160배로 훨씬 크다. 그림 25와 그림 26은 병렬도를 16으로하여 그림 16과 같이 수행시켰을 때 수행 시간의 느려짐을 나타낸 것이다. 병렬도를 32로 하여 네 프로그램을 완전히 겹치게 했을 때 병렬도를 16으로 하여 두 프로그램씩 완전히 분리 시켰을 때보다 얼마나 더 많은 시간이 걸리는지 알아보기 위해 병렬도 32의 느려짐과 병렬도 16의 느려짐의 비율을



(그림 28) 병렬도 8, 1/N 모델의 병렬 프로그램 4개를 동시에 수행시켰을 때의 느려짐

(Fig. 28) The slowdown when 4 parallel programs(parall -elism = 8, 1/N model) are executed simulta neously



(그림 29) 병렬도 8, 1/lnN 모델의 병렬 프로그램 4개를 수행시켰을 때의 느려짐

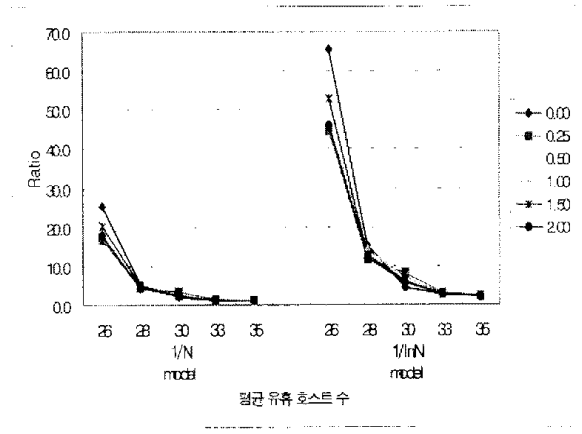
(Fig. 29) The slowdown when 4 parallel programs(parall -elism = 8, 1/lnN model) are executed simult aneously

구해보았다. 그림 27의 1/N 모델 부분은 그림 24와과 그림 25의 비율이고 1/lnN 모델 부분은 그림 24와 그림 26의 비율을 나타낸 것이다. 그림 27을 나타낸 방법은 그림 19에서 설명한 방법과 같다.

그림 27을 보면 30 이하에서는 실험 2에서와 같이 느려짐의 정도가 4개의 모든 병렬 프로그램을완전히 겹쳐서 수행시켰을 때 보다 병렬도를 16으로 하여 되도록 분리시켜 수행시켰을 때 좋은 성능을 보인다. 1/N 모델의 경우 1.3배에서 20.1배정도 좋은 성능을 나타낸다.

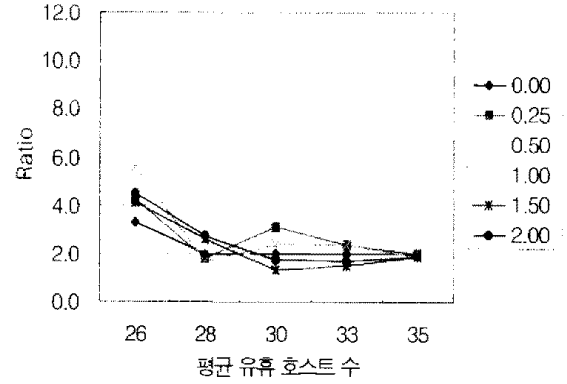
또한 그림 28과 그림 29는 그림 17과 같이 병렬도를 8로 하여 모두 완전히 분리시켜 수행시켰을 때 수행시간의 느려짐을 볼 수 있다. 1/N 모델의 경우 4배에서 9배의 느려짐을 보였다. 병렬도를 32로 하여 네 개의 프로그램을 완전히 겹치게 했을 때 병렬도를 8로 하여 네 프로그램을 완전히 분리 시켰을 때보다 얼마나 더 많은 시간이 걸리는지 알아보기 위해 병렬도 32의 느려짐과 병렬도 8의 느려짐의 비율을 구해보았다. 그림 30의 1/N 모델 부분은 그림 24와 그림 28의 비율이고 1/lnN 모델 부분은 그림 24와 그림 29의 비율을 나타낸 것이다. 그림 30을 나타낸 방법은 그림 19에서 설명한 방법과 같다. 그림 30을 보면 역시 평균 유휴 호스트 수가 30 이하일 때 병렬도를 8로 하여 모두 분리시켜 수행시키는 것이 병렬도를 32로 하여 모두 겹쳐서 수행시키는 것보다 1/N 모델의 경우 1.9배에서 25.4배 좋은 성능을 나타냄을 알 수 있다.

그림 31은 병렬도 32인 병렬 프로그램 2개를 완전히 겹쳐서 수행시켰을 때의 느려짐(그림 20)과 병렬 프로그램 1개를 수행시켰을 때의 느려짐의(그림 18) 비율이고 그림 32는 4개를 완전히 겹쳐서 수행시켰을 때의 느려짐(그림 24)과 1개를 수행시켰을 때의 느려짐(그림 18)의 비율을 나타낸 것이다. 두 그림에서 알 수



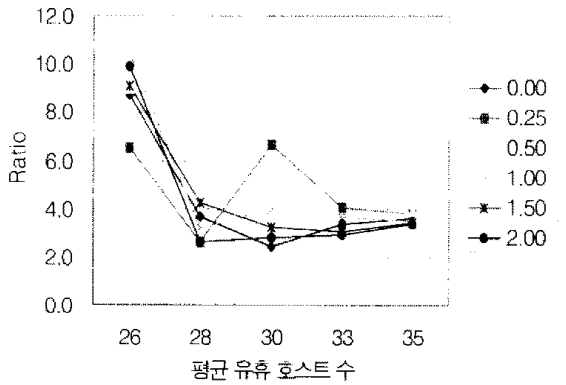
(그림 30) 병렬 프로그램 4개, 병렬도 32의 느려짐과 병렬도 8의 느려짐과의 비율

(Fig. 30) The ratio of slowdown when 4 parallel progra- ms(parallelism = 32) are executed simultaneo -usly(Fig 24) to slowdown when 4 parallel pro -grams(parallelism = 8) are executed simulta -neously(Fig 28, Fig 29)



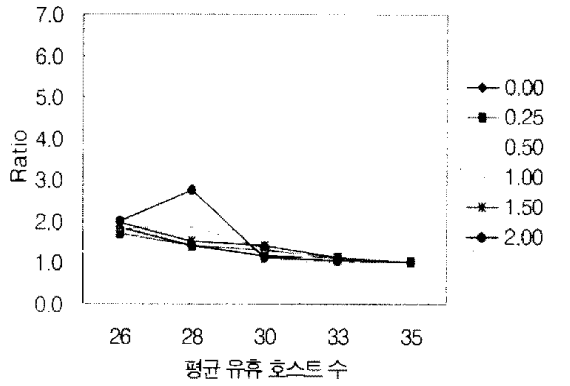
(그림 31) 병렬도 32, 병렬 프로그램 2개의 느려짐과 병렬 프로그램 1개의 느려짐과의 비율

(Fig. 31) The ratio of slowdown when 2 parallel programs (parallelism = 32) are executed simultaneously to slowdown when 1 parallel program(parallelism = 32) is executed



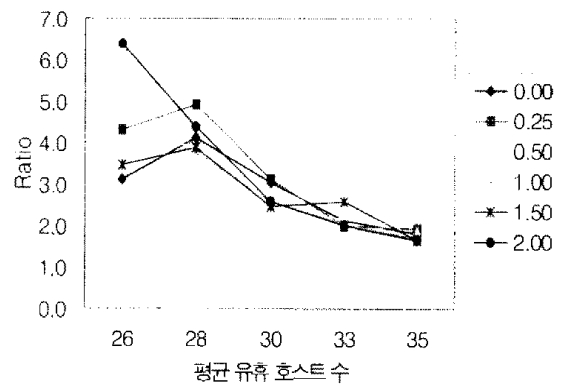
(그림 32) 병렬도 32, 병렬 프로그램 4개의 느려짐과 병렬 프로그램 1개의 느려짐과의 비율

(Fig. 32) The ratio of slowdown when 4 parallel programs (parallelism = 32) are executed simultaneously to slowdown when 1 parallel program(parallelism = 32) is executed



(그림 33) 병렬도 16, 1/N 모델, 병렬 프로그램 2개의 느려짐과 병렬 프로그램 1개의 느려짐과의 비율

(Fig. 33) The ratio of slowdown when 2 parallel programs (parallelism = 16, 1/N model) are executed sim-ultaneously to slowdown when 1 parallel program (parallelism = 16, 1/N model) is executed



(그림 34) 병렬도 16, 1/N 모델, 병렬 프로그램 4개의 느려짐과 병렬 프로그램 1개의 느려짐과의 비율

(Fig. 34) The ratio of slowdown when 4 parallel programs (parallelism = 16, 1/N model) are executed sim-ultaneously to slowdown when 1 parallel program (parallelism = 16, 1/N model) is executed

있는 것은 병렬 프로그램을 여러개 겹쳐서 수행하는 것이 하나만 수행했을 때보다 2개를 수행했을 때는 1.9배에서 5.6배, 4개를 수행했을 때는 3.3배에서 10.4배 느려진다. 그리고 많은 병렬 프로그램을 동시에 수행시켰을 때의 느려짐은 수행시키는 병렬 프로그램의 수에 따라서 선형증가 이상으로 느려진다. 충분한 수의 유휴 호스트가 있을 때 선형증가의 모습을 보이나 그렇지 않은 경우는 순차부하의 영향으로 더욱 느려진다.

그림 33은 병렬도 16인 병렬 프로그램 2개를 동시에 수행시켰을 때의 느려짐(그림 21)과 1개를 수행시켰을 때의 느려짐의 비율이고 그림 34는 4개를 동시에 수행시키되 그림 16과 같이 두 그룹으로 나누어 두 개씩은 완전히 겹쳐서, 두 그룹은 완전히 분리시켜서 수행시켰을 때의 느려짐(그림 25)과 1개를 수행시켰을

때의 느려짐의 비율이다. 병렬 프로그램 4개를 돌렸을 때 1.6배에서 3배정도 더 느림을 알 수 있다. 이것은 그림 33의 경우 두 프로그램이 각각 독립적으로 상호 영향을 주지 않고 수행된 반면 그림 34의 경우 두 프로그램이 번갈아 수행되어 그에 따른 context switch등에 의해 느려졌음을 알 수 있다.

4. 결론 및 향후 연구 과제

본 연구에서는 NOW 상에서의 병렬 프로그램 스케쥴링을 시뮬레이션 하여 순차 부하에 의한 평균 유휴 워크스테이션 수의 변화에 따라 병렬 프로그램의 수행 시간이 어떻게 변하는지 알아보았다. 우리는 다음과 같은 결론을 얻을 수 있다.

- 하나의 병렬 프로그램을 수행시킬 때 순차 부하에 의해 평균 유휴 워크스테이션 수가 적어질수록 병렬 프로그램의 성능은 전체적으로 나빠졌다. 또한 load-imbalance에 의한 계산 시간의 분산(v)이 클수록 성능이 나빠졌다. 그런데 병렬도의 80%의 유휴 호스트가 있을 때 계산 시간의 분산보다 유휴 호스트 수에 더 많은 영향을 받았고 전체적으로 15배 느려졌다.
- 여러 개의 병렬 작업을 동시에 수행시킬 경우 병렬도를 높여 완전히 겹치게 수행시키는 시간 분할 방식보다 병렬도를 낮추더라도 완전히 분리시켜 수행시키는 공간 분할 방식이 더 성능이 좋다.

지금까지는 평균 유휴 워크스테이션 수만을 고려하였지만 분산도 고려해야 할 것이다. 그 이유는 분산이 변함에 따라서도 병렬 프로그램의 수행 시간에 영향을 끼칠 것이기 때문이다. 분산이 크다면 유휴 워크스테이션 수가 크게 변하게 되고 그에 따라 병렬 작업이 사용할 수 있는 워크스테이션 수가 자주 바뀜으로써 이주가 자주 발생할 것이기 때문이다. 또한 여러 병렬 작업이 동시에 수행될 때 context switch가 발생하는 빈도도 고려해야 할 것이다. 그 이유는 전형적인 UNIX 시스템에서도 너무 잦은 태스크 전환은 오히려 성능을 저하시키는 면이 있으므로 NOW에서도 이러한 문제를 고려해야 한다. 또한 부하 균형과 이주 시간등 본 연구에서 고려하지 않았거나 고정시켰던 것에 대한 연구도 있어야 할 것이다. 그리고 유휴 호스트를 관리하는데 대한 연구가 더 있어야 한다. 호스트의 유휴 상태 빈도와 바쁜 상태 빈도에 따라 관리하여 할당 정책에서 사용한다면 더 좋은 성능을 얻을 수 있을 것이다. 더 나아가 실제 시스템으로 구현하여야 하는 과제가 있다.

참 고 문 헌

[1] D. E. McDysan and D. L. Spohn, "ATM : Theory and Application," McGraw-Hill, 1995.

[2] N. J. Boden et al., "Myrinet : A Gigabit-per-Second Local Area Network," IEEE Micro, vol. 15, no. 1, pp. 29-36, February 1995

[3] T. E. Anderson et al., "A Case for NOW (Network of Workstations)," IEEE Micro, vol. 15, no. 1, pp. 54-64, February 1995.

[4] Remzi H. Arpaci, Amin M. Vahdat, Tom Anderson, and Dave Patterson, "Combining Parallel and Sequential Workloads on a Network of Workstations," Technical Report UCB : CSD-94-838, University of California, Berkeley , 1994.

[5] M. M. Mutka and M. Livny, "The available capacity of a privately owned workstation environment," Performance Evaluation, vol. 12, no. 4, pp. 269-284, July 1991.

[6] M. Litzkow and M. Linvy, "Experiences with Condor Distributed Batch System," In Proceedings of the IEEE Workshop on Experimental Distributed Systems, pp. 97-101, Oct. 1990.

[7] R. Arpaci et al., "The Interaction of Parallel and Sequential Workloads on a Network of Workstation," Proceedings of Joint Internat-ional Conference on Measurement & Mode-ling of Computer Systems, ACM Performance Evaluation Review, vol. 23, no. 1, pp. 267-278, May 1995.

[8] A. C. Dusseau et al., "Effective Distributed Scheduling of Parallel Workloads," Proceed-ings of ACM SIGMETRICS '96 Conference on Measurement and Modeling , 1996.

[9] J. K. Ousterhout, "Scheduling techniques for concurrent systems," Proc. 3rd Internat

ional Conference on Distributed Computing Systems, pp. 22-30, Oct. 1982.
[10] F. Douglis and J. Ousterhout, "Transparent Process Migration : Design Alternatives and the Sprite Implementation," Software Practice and Experience, vol. 21, no. 8, pp. 757-785, August 1991.
[11] Andrew S. Tanenbaum and Albert S. Woodhull, "Operating Systems : Design and Implementation," Prentice-Hall, 1997.
[12] Amin M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson, "Efficient, Portable, and Robust Extension of Operating System Functionality," Technical Report UCB//CSD-94-842, University of California, Berkeley, 1994.
[13] Mor Harchol-Balter and Allen B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," In the Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, pp. 13-24, May 23-26, 1996.

장 영 민

1996년 홍익대학교 컴퓨터공학과 (학사)
1996년~현재 홍익대학교 전자계산학과 석사과정
관심 분야 : 클러스터 컴퓨팅, 운영체제, 보안

심 영 철

1979년 서울대 전자공학과(학사)
1981년 한국과학기술원 전기 및 전자과(석사)
1991년 University of Califo-nia, Berkeley 전산학 (박사)
1981년~1984년 삼성전자 대리
1991년~1993년 University of California, Berkeley 연구원
1993년~현재 홍익대학교 컴퓨터공학과 조교수
관심 분야 : 분산병렬처리, 인터넷 프로토콜, 네트워크 보안, 망관리