PC 클러스터에서 스케줄링 기법의 구현

강 오 한[†] · 송 희 헌[††] · 정 중 수[†††]

요 약

본 논문에서는 버스 기반의 클러스터 구조에 적합한 새로운 태스크 스케줄링 기법을 소개하고, PC 클러스터에 구현하여 스케줄링 기법의 성능을 분석한다. 구현된 스케줄링 기법은 태스크 그래프를 입력으로 받아 PC 클러스터로 스케줄링하며, 휴리스틱을 사용하여 태스크를 선택적으로 중복함으로써 병렬연산시간을 단축한다. PC 클러스터는 리눅스 OS가 설치된 6대의 PC가 Gigabit Ethernet으로 연결되어 있다. 통신을 위해 TCP/IP 프로토콜을 사용하며, 메시지 교환을 위해 표준화된 병렬 프로그래밍 도구로 MPI를 사용한다. 실험을 한 결과 본 논문에서 소개한 스케줄링 기법이 비교 기법보다 병렬연산시간 측면에서 성능이 우수함을 확인하였다.

Implementation of Scheduling Strategies on PC Clusters

Oh-Han Kang[†] · Hee-Heon Song[††] · Joong-Soo Chung[†††]

ABSTRACT

In this paper, we propose a new task scheduling scheme for bus-based cluster architectures and analyze performance of the scheduling scheme which has been implemented in a PC cluster. The implemented scheme schedules the tasks of a task graph to the processors of a PC cluster, and it reduces parallel execution time by selectively duplicating critical tasks using heuristic. Experimental results show that the proposed scheduling scheme produces better parallel execution time than the other scheduling scheme.

키워드 : PC 클러스터(PC Cluster), 스케줄링(Scheduling), 휴리스틱(Heuristic), 태스크 그래프(Task Graph), 태스크 중복(Task Duplication)

1. 서 론

하드웨어 기술 변화와 통신 기술의 발전에 따른 네트워크의 보급이 확산되면서 클러스터 시스템에 대한 관심이 높아지고 있다. 클러스터 시스템은 여러 대의 PC나 워크스테이션을 고속 네트워크로 연결하여 고성능 서버의 성능을 발휘하는 시스템으로 범용성, 확장성, 가격에 대한 성능의 우수성으로 인하여 병렬 연산을 위한 효과적인 시스템으로 정착되고 있다[1, 2]. 그러나 클러스터는 다중프로세서 시스템과 비교하여 통신 대역폭이 낮아서 통신속도가 지연되는 또 다른 특성을 가지고 있다. 클러스터의 성능 저하에 가장 큰 영향을 미치는 요인중의 하나가 프로세서 사이의 통신지연이다. ATM과 같은 고속의 네트워크에서도 프로세서간의 통신이 병렬 연산을 위한 병목구간(bottleneck)이 되고 있다[3]. 현재까지 클러스터 환경에서 병렬 연산을 위한 다양한 네트워크 유형(topology)이 사용되지만 구조가 간단한 버스(bus) 구조가 광범위하게 사용되고 있다. 그러나 버스 구조는 네트워크 통신자원을 공유할 수 없으므로 다른 구조와 비교할 때 중요한 한계중의 하나가 통신자원 사용을 위한 비용(cost)이 크다는 것이다. 따라서 버스 구조의 클러스터에서 병렬 연산을 위한 효과적인 스케줄링 기법을 사용함으로서 병렬 연산을 위한 비용을 절감할 수 있다.

클러스터에서 프로세서의 활용도를 높이고 성능을 향상시키기 위한 태스크 스케줄링 기법과 프로세서간의 통신지연 단축에 관한 연구가 최근에 활발히 추진되고 있다. 병렬 환경에서 시스템의 성능을 향상시키기 위하여 효과적인 스케줄링 기법의 개발이 중요한 연구 분야가 되어왔다. 태스크 스케줄링 기법은 입력된 태스크 중에서 다음에 처리할 태스크를 선택하는 방법이며, 태스크를 프로세서로 할당한다. 병렬 프로그램의 태스크를 효과적으로 스케줄링 함으로서 시스템 자원의 활용도를 높이고 프로그램의 병렬연산시간을 단축할 수 있다[4-6]. 병렬 연산을 위하여 현재까지 제안된 태스크 스케줄링 기법들은 휴리스틱을 기반으로 한 것[7]과 태스크 중복(duplication)을 기반으로 한 것[8, 9]이 주류를 이루고 있다. 태스크 중복을 기반으로 하는 스케줄링 기법은 태스크를 다수의 클래스(class)로 나누고 각 클래스를 프로세서에 할당한다. 또한 각 클래스에 태스크를 선택적으로 중복하여 할당함으로써 통신을 위한 비용을 절감할 수 있다.

※ 이 논문은 2003년도 안동대학교 학술연구지원사업에 의하여 연구되었음.
† 종신회원 : 안동대학교 컴퓨터교육과 교수
†† 정 회 원 : 안동대학교 컴퓨터교육과 교수
††† 정 회 원 : 안동대학교 전자정보산업학부 교수
논문접수 : 2004년 3월 26일, 심사완료 : 2004년 11월 22일

현재까지 병렬연산을 위한 다중프로세서나 클러스터 시스템을 위한 다양한 스케줄링 기법들이 제안되어 성능이 평가되었으나 대부분이 시뮬레이션에 의한 비교였다[4-9]. 클러스터의 보급이 확산되면서 최근에 태스크 스케줄링 기법을 실제 시스템에 구현하여 성능을 비교한 연구들이 발표되고 있다[10-12].

본 논문에서는 태스크 중복을 기반으로 버스 구조의 클러스터에 적용할 수 있는 휴리스틱 스케줄링 알고리즘을 실제 클러스터 시스템에 구현하여 성능을 비교한다. PC 클러스터에 구현된 스케줄링 기법은 중복할 태스크를 선택할 때 휴리스틱을 사용하여 병렬 시간을 단축할 수 있는 태스크들을 중복시킨다. 스케줄링 기법이 구현된 PC 클러스터는 리눅스가 설치된 6대의 PC와 Gigabit Ethernet을 사용하며, 메시지 교환을 위해 표준화된 메시지 전달방식 통신 라이브러리인 MPI[13]를 사용한다.

본 논문의 나머지 부분의 구성은 다음과 같다. 2장에서는 본 논문에서 제안하는 스케줄링 기법을 설명한다. 3장에서는 PC 클러스터 구축과 스케줄링 기법을 클러스터에 구현하는 방법을 설명한다. 4장에서는 PC 클러스터에 구현된 기법의 성능을 비교한다. 5장에서는 본 논문에 대한 결론을 나타낸다.

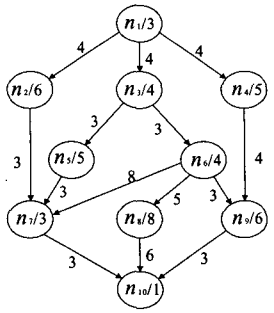
2. 태스크 스케줄링 기법

클러스터 시스템에서 태스크 스케줄링의 주된 목적은 태스크를 서로 다른 프로세서에 할당함으로써 응용 프로그램의 병렬연산시간을 단축할 수 있도록 스케줄링 길이를 줄이는 것이다. 이러한 응용 프로그램은 태스크 스케줄링 기법의 입력으로 사용되는 태스크 그래프로 나타낼 수 있다.

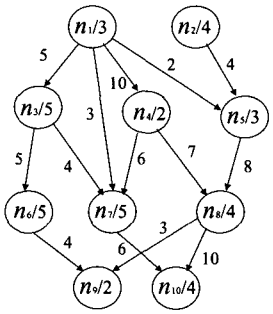
2.1 태스크 그래프

태스크 그래프는 V가 태스크 노드이고, E가 통신 링크일 때 튜플(tuple) (V, E, t, c)로 정의할 수 있다. 여기서 집합 t는 연산비용(computation cost)으로 구성되며, 각각의 태스크 $i \in V$는 $t(i)$로 표시하는 연산비용을 갖는다. c는 통신비용(communication cost)의 집합으로 구성되며, 태스크 i에서 태스크 j로 연결되는 링크 $e_{ij} \in E$는 통신비용 c_{ij}를 갖는다. 태스크 그래프에서 링크는 두 태스크 사이의 선행 관계(precedent relation)를 나타낸다. 따라서 하나의 태스크는 모든 선행 태스크의 수행이 완료되어야 실행될 수 있으며, 두 태스크가 서로 다른 프로세서에 할당되면 통신비용이 필요하다. (그림 1)은 태스크 그래프의 예를 나타낸 것이다.

하나의 태스크는 독립된 작업의 단위로 하나의 노드로 나타내며, 노드 안에 태스크의 정보를 표시한다. 노드에서 태스크 i에 대한 노드 번호는 n_i로 나타내고, 슬래쉬(/) 다음의 숫자는 태스크의 연산비용을 나타낸다. 태스크 그래프에서 각 링크에는 태스크 사이의 통신비용을 표시한다. 태스크는 하나의 독립된 작업 단위이며 비선점(nonpreemptive)으로 동작한다. 각 프로세서에는 통신 전용 프로세서가 존재하여 연산과 통신을 동시에 처리할 수 있다.



(a) entry node와 exit node가 각각 1개인 경우



(b) entry node와 exit node가 각각 2개인 경우

(그림 1) 태스크 그래프

응용 프로그램 분할 알고리즘은 응용 프로그램을 적당한 실행단위(grain size)를 갖는 태스크로 분할하며, 응용 프로그램을 (그림 1)과 같은 형태의 태스크 그래프(DAG : Directed Acyclic Graph)로 표현한다[14, 15]. DAG를 입력받는 태스크 스케줄링 알고리즘은 Bellman-Ford, Systolic, Master-Slave, Cholesky decomposition 등 다양한 응용 프로그램에 활용할 수 있으며, 태스크 스케줄링 알고리즘의 성능평가를 위한 시뮬레이션에 이들 응용 프로그램의 실제 DAG를 사용하고 있다[15].

2.2 수식 및 정의

본 논문에서 제안한 스케줄링 기법에서 스케줄링 길이와 병렬연산 시간의 계산을 위해 태스크 i에 대하여 아래와 같은 수식들을 사용한다.

$$pred(i) = \{j | e_{ji} \in E\} \quad (1)$$

$$suce(i) = \{j | e_{ij} \in E\} \quad (2)$$

$$EST(i) = 0 \quad \text{if } pred(i) = \Phi \quad (3)$$

$$EST(i) = \min\left(\max_{j \in pred(i)} \max_{k \in pred(i),\, k \neq j} (ECT(j), ECT(k) + c_{ki}) \right) \quad \text{if } pred(i) \neq \Phi \quad (4)$$

$$ECT(i) = EST(i) + t(i) \quad (5)$$

$$CPT(i) = \{j | (ECT(j) + c_{ji}) = (ECT(k) + c_{ki})\} \quad \forall_j \in pred(i); k \in pred(i), k \neq j \quad (6)$$

$$LCT(i) = ECT(i) \quad \text{if } suce(i) = \Phi \quad (7)$$

$$LCT(i) = \min\left(\min_{j \in suce(i),\, i \neq CPT(i)} (LST(j) - c_{ij}), \min_{j \in suce(i),\, i \neq CPT(j)} LST(j) \right) \quad (8)$$

$$LST(i) = LCT(i) - t(i) \quad (9)$$

$$level(i) = t(i) \quad \text{if } suce(i) = \emptyset \quad (10)$$

$$level(i) = \max_{k \in suce(i)} (level(k)) + t(i) \text{ if } suce(i) = \emptyset \quad (11)$$

본 논문에서 제안한 스케줄링 기법에서는 태스크 그래프에 대한 스케줄링 길이를 계산하기 위하여 아래와 같은 정의를 사용한다.

정의 1 : 태스크 i의 모든 부모 노드 j에 대하여 $ECT(j)+c_{ji}$값이 최대인 노드를 $CPT(i)$(Critical Parent)라고 한다.

정의 2 : 태스크 i의 $CCPT(i)$(Critical Parent of CPT)는 $CPT(i)$의 CPT로 정의한다.

정의 3 : 스케줄링이 완료되었을 때 태스크 i의 확정된 시작 시간과 종료 시간을 각각 $RST(i)$ (Real Start Time)과 $RCT(i)$ (Real Completion Time)로 정의한다. $RCT(i)$는 $RST(i)$에 태스크 i의 연산비용을 합한 것이다.

정의 4 : 스케줄링 알고리즘에 의해 동일한 프로세서에 할당된 태스크의 집합을 태스크 클래스(class)라고 한다.

<표 1>은 (그림 1)(a)와 (그림 1)(b) 태스크 그래프에 대하여 식 (1)~식 (11)까지의 값을 구한 결과를 나타낸 것이다.

〈표 1〉 (그림 1)의 태스크 그래프에 대한 수식의 값

노 드		EST		ECT		LST		LCT		CPT		CCPT		level	
(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
n_1	n_1	0	0	3	3	0	0	3	3	-	-	-	-	20	17
n_2	n_2	3	0	9	4	6	1	12	5	1	-	-	-	10	15
n_3	n_3	3	3	7	8	3	3	7	8	1	1	-	-	17	14
n_4	n_4	3	3	8	5	3	3	8	5	1	1	-	-	12	11
n_5	n_5	7	5	12	8	7	5	12	8	3	2	1	-	9	11
n_6	n_6	7	8	11	13	8	8	12	13	3	3	1	1	13	7
n_7	n_7	15	11	18	16	15	11	18	16	6	3	3	1	4	9
n_8	n_8	11	12	19	16	13	13	21	17	6	5	3	2	9	8
n_9	n_9	12	17	18	19	12	17	18	19	6	8	3	5	7	2
n_{10}	n_{10}	21	22	22	26	21	22	22	26	8	8	6	5	1	4

2.3 태스크 스케줄링 기법

본 논문에서 제안한 스케줄링 기법은 클러스터 환경을 버스 구조로 가정하며, 태스크의 중복을 허용하고 이를 위해 휴리스틱을 사용한다. 본 논문에서는 이를 HTDS(Heuristic Task Duplicaton Scheduling) 기법이라고 한다. HTDS 기법은 태스크 그래프를 입력으로 받아서 클래스를 생성하는 부분과 생성된 클래스를 프로세서에 할당하여 각각의 태스크를 스케줄링하는 부분으로 구성된다. 현재까지 제안된 다중 프로세서와 클러스터 환경의 스케줄링 기법은 프로세서 사이에 완전연결을 가정하였으며 다수의 프로세서가 동시에 통신이 가능하였다. 버스 기반의 클러스터 환경에서는 네트워크에서의 통신 충돌이 병렬 응용 프로그램의 연산 시간에 매우 큰 영향을 주며, 네트워크 통신 자원의 동시 사용을 위한 공유가 불가능하다. 따라서 본 논문에서 개발한 스케줄링 기법에서는 서로 다른 프로세서에 할당된 태스크 사이의 통신을 위하여 네트워크 통신 자원을 우선 배정하여 스케줄링 함으로써 네트워크 충돌을 방지한다. 본 논문에서는 HTDS 기법의 성능을 비교하기 위하여 다중프로세서 환경에서 성능이 우수하다고 알려진 태스크 중복기반의 STDS [8] 스케줄링 기법을 버스 구조에 맞도록 수정하였다. 본 논문에서는 이 수정된 스케줄링 기법을 MSTDS(Modified STDS)라 한다. STDS 기법은 프로세서 사이에 완전연결을 가정하였으므로 MSTDS 기법에서는 STDS 알고리즘[8]의 'Step 3'을 위에서 설명한 버스 구조에 맞도록 수정하였다.

2.3.1 휴리스틱 개발

HTDS 기법에서는 중복할 태스크를 선택할 때 다음과 같은 휴리스틱을 사용함으로써 스케줄링 길이를 줄여서 병렬연산시간을 단축한다.

① CPT 노드가 스케줄링하고 있는 노드의 유일한 부모 노드이면 CPT 노드를 중복한다.

② 스케줄링하고 있는 결합 노드와 동일한 클러스터에 배정할 CPT 노드가 이미 다른 클러스터에 배정된 경우에는 결합 노드의 $CCPT$ 노드가 결합 노드와 같은 클러스터에 배정될 수 있는 경우에만 CPT 노드를 중복하여 배정한다.

③ 결합 노드에서 모든 부모 노드가 이미 다른 프로세서에 배정된 경우에는 CPT 노드를 중복한다.

태스크 그래프에서 결합 노드에 대한 스케줄링이 병렬연산시간에 가장 큰 영향을 미친다. 위의 휴리스틱에서 두 번째와 세 번째 내용은 결합 노드와 어떤 부모(parent) 노드를 동일한 프로세서에 할당할 것인지를 결정한다. 본 논문에서 개발한 알고리즘의 중요한 개념은 결합 노드와 스케줄링 길이에 결정적인 영향을 줄 수 있는 CPT 노드를 선택적으로 중복하여 동일한 프로세서에 배정함으로써 스케줄링 길이와 병렬연산시간을 단축하는 것이다. 태스크 그래프에서 하나의 노드가 두 개 이상의 노드에 대한 CPT 노드로 나타날 수 있다. 이때 CPT 노드를 계속적으로 중복하여 클러스터에 할당하는 것은 스케줄링 길이를 연장할 수 있는 가능성이 있다. 그 이유는 식 (6)의 $CPT(i)$ 정의에서 확인할 수 있다. 정의에 따르면 태스크 i의 EST는 $CPT(i)$에 의해서 결정되므로 CPT 노드를 중복함으로써 자식 노드의 EST를 지연시켜 스케줄링 길이가 늘어날 수 있다. 따라서 알고리즘에서는 위의 휴리스틱에서 두 번째 방법을 사용하여 CPT 노드를 선택적으로 중복한다. 이 방법은 결합 노드의 CPT 노드 중복을 위하여 CPT 노드의 CPT인 $CCPT$ 노드가 동일한 클러스터에 할당 가능한지를 체크한다. 결합 노드와

CCPT 노드가 동일할 클러스터에 할당될 수 있으면 결합 노드, *CPT*, *CCPT* 노드를 하나의 클러스터에 할당하여 스케줄링 길이를 단축하는 것이다. 위에서 세 번째 경우는 결합 노드에서 모든 부모 노드가 이미 다른 클러스터에 할당된 상태에서 새로운 클러스터 생성을 위하여 부모 노드 중에서 하나를 선택하는 경우이다. 따라서 결합 노드와 *CPT* 노드를 동일한 클러스터에 할당함으로써 스케줄링 길이를 단축하기 위한 것이다.

HTDS 기법에서는 위의 휴리스틱을 사용하여 태스크 클래스의 생성과 함께 중복할 태스크를 선택하며, 알고리즘의 '단계 2'에서 이것을 구현하였다. 예를 들면, (그림 1)(a)를 사용하여 태스크 클래스를 생성하면 중복되는 태스크는 (그림 2)(b)에서와 같이 n_1, n_3, n_6이 된다.

2.3.2 알고리즘 구조

본 논문에서 개발한 알고리즘은 태스크 그래프를 입력으로 받아서 태스크 클래스를 생성하는 부분과 생성된 클래스에 속한 태스크들에 대한 시작 시각과 완료 시각을 확정하는 부분으로 구성된다. 알고리즘의 단계 1에서는 앞 절에서 나타낸 수식들을 이용하여 알고리즘에서 사용할 필요한 파라미터 값들(식 (1)~식 (11))을 구한다. 이 단계는 다중 프로세서 환경의 스케줄링 알고리즘과 유사하다.

단계 2에서는 단계 1에서 구한 값들을 이용하여 중복된 태스크로 구성된 태스크 클래스를 생성한다. 각 노드는 스케줄링되기 전에 태스크 그래프에서의 레벨을 기준으로 오름차순으로 저장한다. 태스크 클래스의 생성은 태스크 그래프에서 레벨이 가장 낮은 노드에서 시작하여 각 노드의 *CPT* 노드를 동일한 클래스에 할당하면서 진행한다. 각 노드의 중복 여부는 휴리스틱에 의하여 결정되는 것으로, 클래스 생성을 위해 반드시 필요한 노드와 스케줄링 길이를 줄일 수 있는 노드를 클래스에 중복한다. 스케줄링하는 현재 노드의 *CPT* 노드가 이미 다른 클래스에 할당된 경우에는 *CCPT* 개념을 적용하여 *CPT* 노드의 중복을 위한 휴리스틱을 사용한다.

단계 3에서는 단계 2의 결과를 이용하여 태스크의 *RST* (Real Start Time)와 *RCT*(Real Completion Time)를 확정하고, 병렬 처리 시간을 예측하기 위한 스케줄링 길이를 계산한다. 이 단계에서 알고리즘은 버스 구조의 클러스터 특성을 고려하여 클래스에 배정된 태스크를 네트워크 자원의 충돌이 발생되지 않도록 스케줄링한다. 네트워크 통신 자원을 독립된 두 개의 통신 태스크가 동시에 사용할 때 충돌이 발생되므로 스케줄링 알고리즘에서 이를 방지하도록 통신 자원을 할당한다. 이를 위하여 알고리즘에서는 슬롯(slot) 개념을 사용한다. 두 개의 태스크가 통신하는 시간 동안을 하나의 슬롯으로 보며, 프로그램이 시작되기 전에 모든 슬롯은 비어있다고 가정한다. 알고리즘에서는 태스크가 통신하기 전에 통신비용에 해당하는 길이의 사용하지 않는 첫 번째 슬롯을 태스크에 배정한다. 각 클래스에 저장된 태스크를 레벨이 낮은 것부터 스케줄링하며, 스케줄링 할 태스크의 데이터 선행 관계가 만족되지 않으면 다음 클래스의 태스크를 스케줄링한다. 스케줄링 할 노드의 모든 부모 노드의 스케줄링이 완료되어 데이터 선행 관계가 만족되는 경우에는 네트워크 통신 자원의 충돌이 발생되지 않도록 태스크를 스케줄링한다. 다음은 HTDS 스케줄링 알고리즘을 단계별로 나타낸 것이다.

단계 1 :
// 입력 : 태스크 그래프 (*V, E, t, c*)
출력 : *EST, ECT, LST, LCT, level, CPT, CCPT* //

compute *EST(i), ECT(i), LST(i), LCT(i), level(i), CPT(i), CCPT(i)* for all nodes $i \in v$.

단계 2 :
// 입력 : 태스크 그래프 (*V, E, t, c*), *queue* (레벨에 따라 오름차순으로 정렬된 태스크 큐)
출력 : 중복된 태스크를 갖는 태스크 클러스터들 //

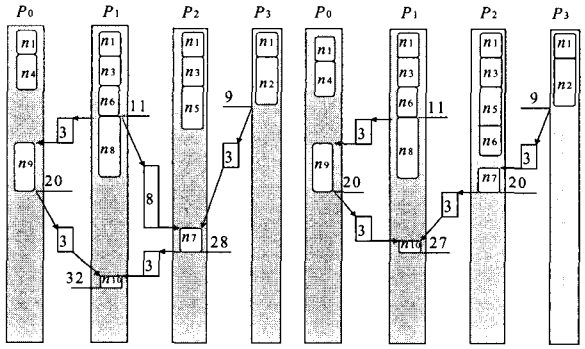
$i = 0$
n_x = first element of *queue*
assign n_x to W_i
while (not all tasks are assigned to a workstation) {
  for (qindex=0 ; num_of_task-1 ; qindex++) dupflag[qindex] = 0
  $n_y = CPT(x)$
  if (n_y already been assigned to another workstation) {
    n_k = another parent of n_x which has not yet been assigned to a workstation
    If ($(LST(x) - LCT(y)) \geq c_{xy}$) then // n_y is not critical for n_x //
    { $n_y = n_k$; dupflag[y] = 1}
  else
    candflag = 0
    for another parent n_z of n_x, $z \neq y$
    if ($(ECT(x) + c_{yx}) = (ECT(z) + c_{zx})$ && n_z has not yet been assigned to a workstation) then
    { $n_y = n_z$; candflag = 1}
  endif
  if (candflag = 0) { $n_y = n_k$; dupflag[y] = 1}
  endif
}
endif
assign n_y to W_i
$n_x = n_y$
if n_x is entry node
  assign n_x to W_i
for (qindex=0 ; num_of_task - 1 ; qindex++)
  if (dupflag[qindex] = 1 && *CPT*(qindex) is in W_i)
    assign n_{qindex} to W_i
  n_x = the next element in *queue* which has not yet been assigned to a workstation
  increment i
assign n_x to W_i
endif
}

단계 3 :
// 입력 : 1) 중복된 태스크로 이루어진 태스크 클러스터들
2) *bus_slot_list* : 네트웍 통신 자원에서 사용 가능한 슬롯을 나타냄
초기값 : [0, ∞], 형식 : [s_1, e_1], [s2, e2], ⋯ , [s_n, ∞] //

let there exit n task clusters on n workstations, i.e. { W_0, W_1,⋯, W_{n-1})
while (not all nodes be scheduled) do {
for (i = 0 ; $i < 0$; i++) {
  select the node n_j to be scheduled from W_i
  let there exist p parents of n_j, i.e. { $n_{j_1}, n_{j_2}, \cdots, n_{j_p}$}
  if (all n_j's p parents have been scheduled) {
    for ($k = 1$; $k \leq p$; k++) {
      If (n_{j_k} was scheduled onto W_i)
        if (tmp_rst of n_j < tmp_rct of n_{j_k}) tmp_rst of n_j = tmp_rct of n_{j_k}
      else {
        let[s_i, e_i] be the first slot in *bus_slot_list* for n_{j_k}
        if ((($e_i - s_i + 1$) ≥ $c_{j_k j}$) && ($s_i \geq RCT$ of n_{j_k})) {

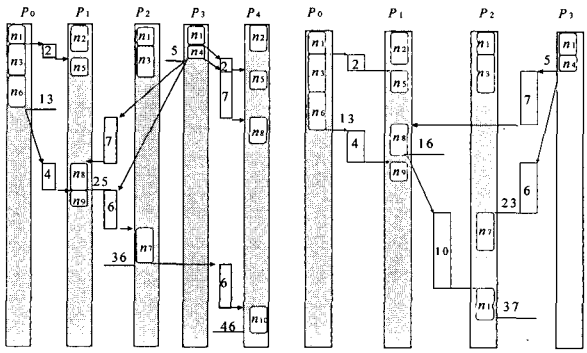
```
            slot_busy = s_i + C_{j_k j}
            substitute [s_i, e_i] to [s_i + C_{j_k j}, e_i]
          }
        else
          if ((RCT of n_{j_k} + C_{j_k j}) ≤ e_i)) {
            slot_busy = RCT of n_{j_k} + C_{j_k j}
            substitute [s_i, e_i] with [s_i, RCT of n_{j_k}] and [RST of n_{j_k} +
                C_{j_k j}, e_i]
          }
          if (tmp_rst of n_j < slot_busy) tmp_rst of n_j = slot_busy
        }
        if (tmp_rst of n_j < W_i_busy) {
          tmp_rst of n_j = W_i_busy
          tmp_rst of n_j = tmp_rst of n_{j_k} + C_{j_k j}
          W_i_busy = tmp_rct of n_j
        }
      }
    }
    if (RST of n_j is not assigned) {
    RST of n_j = tmp_rst of n_j ; RCT of n_j = tmp_rct of n_j
    }
    else
      if (tmp_rst of n_j < RST of n_j) {
      RST of n_j = tmp_rst of n_j ; RCT of n_j = tmp_rct of n_j
    }
  }
}
```

(그림 2)는 (그림 1)(a)의 태스크 그래프에 스케줄링 알고리즘을 적용한 결과이며, (그림 3)은 (그림 1)(b)의 태스크 그래프에 스케줄링 알고리즘을 적용하여 스케줄링한 결과를 나타낸 것이다.



(a) MSTDS 기법을 적용한 경우 (b) HTDS 기법을 적용한 경우

(그림 2) (그림 1)(a)를 스케줄링한 결과

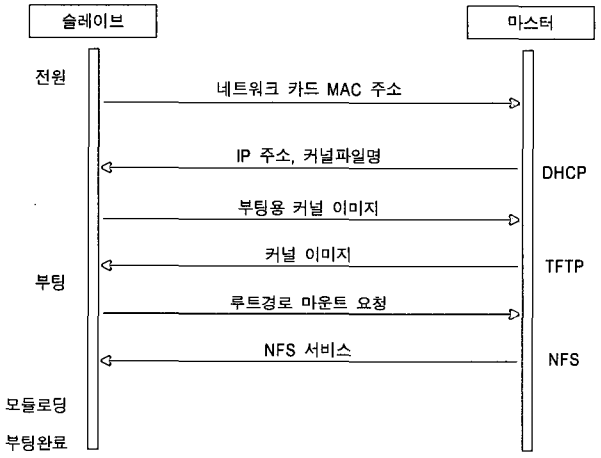


(a) MSTDS 기법을 적용한 경우 (b) HTDS 기법을 적용한 경우

(그림 3) (그림 1)(b)를 스케줄링한 결과

3. PC 클러스터 구축과 스케줄링 기법의 구현

본 논문에서는 PC 클러스터를 구축하고 태스크 스케줄링 기법을 구현하여 성능을 비교하였다. 리눅스(Linux) OS가 설치된 클러스터 PC는 기가비트 이더넷(Gigabit Ethernet)으로 연결되었으며, 메시지 전송을 위하여 병렬 프로그래밍 도구인 MPI를 사용한다. 본 논문에서 구축한 클러스터는 하나의 마스터 PC와 5대의 슬레이브 PC로 구성하였다. 클러스터 구축에 사용된 PC 사양은 Pentium IV CPU, 256M-512M 메모리, 1Gbps NIC로 구성되어 있다. 클러스터 구축에 필요한 소프트웨어는 시스템의 기본 운영환경을 제공하는 운영체제, 각 노드에 IP주소와 커널을 제공하기 위한 서비스 프로그램, 마스터 노드의 디스크를 공유하기 위한 파일 시스템, 각 노드들의 계정 정보를 공유하기 위한 서비스 프로그램, 병렬프로그래밍을 위한 API 등으로 구성된다. (그림 4)는 본 논문에서 구축한 클러스터의 슬레이브 노드의 부팅 과정과 각 단계에서 필요한 소프트웨어를 나타낸 것이다.



(그림 4) PC 클러스터에서 Diskless 슬레이브 노드의 부팅과정

본 논문에서는 클러스터 구축에 폭넓게 사용되고 많은 사용자와 지원프로그램을 확보하고 있는 레드햇 리눅스(Red-hat Linux)를 PC의 OS로 사용하였다. 슬레이브 노드가 마스터 노드로부터 커널 이미지를 전송받기 위해 파일전송 서비스가 필요한데, 사용자 인증과정 없이 파일전송 서비스를 제공할 수 있는 서비스인 TFTP(Trivial File Transfer Protocol)를 사용하였다. 본 논문에서 구축한 클러스터는 각 노드들간 메시지 및 파일전송, 파일시스템 공유를 위해 TCP/IP 프로토콜을 기반으로 이루어진다. 따라서 클러스터에 참여하는 모든 노드에게 TFTP를 이용해 커널 이미지를 다운로드 받기 전에 TCP/IP를 사용하기 위한 정보가 주어져야 한다. DHCP(Dynamic Host Configuration Protocol)는 마스터 노드에서 슬레이브 노드에게 IP주소를 비롯한 TCP/IP 환경 정보를 전달하기 위한 서비스이다. 이러한 정보의 전달은 슬레이브 노드가 구동한 직후에 이루어진다. 클러스터에 참여하는 노드들 중 마스터 노드는 로컬 디스크에 운영

체제가 설치되지만 하나의 마스터 노드를 제외한 나머지 슬레이브 노드들은 디스크가 없으며 모든 정보는 마스터노드의 디스크를 공유하여 사용하였다. 이를 위해 네트워크로 연결된 호스트의 파일시스템을 로컬 파일시스템처럼 사용할 수 있는 NFS(Network File System)이 필요하다.

일반적인 응용프로그램은 하나의 CPU 혹은 하나의 호스트에서 실행되도록 개발된다. 클러스터와 같은 병렬 시스템에서 이러한 응용프로그램을 실행한다면 클러스터 규모와는 관계없이 하나의 시스템만 동작하게 된다. 여러 시스템이 협력하여 작업을 처리하도록 하려면 응용프로그램이 병렬처리를 지원하도록 개발되어야하며 병렬프로그래밍을 지원해주는 라이브러리가 필요하다. 병렬프로그래밍 라이브러리는 MPI(Message Passing Interface), PVM(Parallel Virtual Machine) 등이 있으며, MPI는 본 논문에서 구축하는 클러스터와 같이 동일한 플랫폼으로 구성된 클러스터 시스템에 적합하다.

본 논문에서 클러스터 구축을 위하여 커널 환경설정, DHCP 설정, TFTP 설정, NSF 설정, MPI 설정의 작업을 수행하였다. 병렬 프로그래밍 도구인 MPI 설정을 위하여 MPI 설치, 사용자 추가, rlogin과 rsh 설정, rhosts 생성, lam-bhost와 lamhosts 생성의 작업을 수행하였다.

본 논문에서 DAG의 연산비용을 위한 구현은 연산비용에서 제시한 시간(ms)만큼을 지연시켜 실제 작업을 처리하기 위해 시간을 소요하는 것으로 구현하였다. 프로그램에서는 주어진 시간만큼 대기하다가 노드의 번호와 작업번호를 출력하는 calculate() 함수를 제작하였다. 아래 코드는 calculate() 함수를 나타낸 것이다.

```
void calculate(double msecs, int rank, char *tasknode) {
    double start, stop, secs ;
    secs = msecs / 1000.0 ;
    start = MPI_Wtime() ;
    stop = start + secs ;
    while(MPI_Wtime() < stop) ;
    printf("Finished [%s] By P[%d].\n," tasknode, rank) ;
}
```

DAG에서 통신비용은 메시지의 크기와 통신의 대역폭에 의해 결정될 수 있다. 정확한 통신비용을 구현하기 위해 먼저 각 노드간의 전송속도를 측정하고 통신비용을 곱하여 실제 전송할 데이터의 크기를 구하게 된다. 속도 측정 프로그램은 두 개의 노드를 지정하면 512Kbyte를 전송한 시간을 계산하고 단위시간(1ms)당 전송량을 계산한다. 프로그램을 사용하여 본 논문에서 구축한 클러스터의 노드간 평균 전송속도를 측정한 결과 약 45Kbytes/ms 정도로 나타났다. 프로그램의 실행시간이나 메시지 전달에 필요한 오버헤드의 영향을 줄이기 위하여 비용의 시간단위를 10ms로 하였다. 아래 코드는 클러스터의 노드간 평균 전송속도를 측정하는 프로그램을 나타낸 것이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <sys/time.h>

static struct timeval start_time ;
static struct timeval finish_time ;

static char *buffer ;
static int iproc ;
static int nproc ;

double roundtrip(int count, int size)
{
    MPI_Status status ;
    int i ;
    double elapsed_time ;
    gettimeofday(&start_time, (struct timezone *) 0) ;
    if ( iproc == 0 ) for (i=0 ; i<count ; i++)
        MPI_Send(buffer, size, MPI_BYTE, 1, 0, MPI_COMM_WORLD) ;
    else for (i=0 ; i<count ; i++)
        MPI_Recv(buffer, size, MPI_BYTE, 0, 0, MPI_COMM_WORLD,
        &status) ;
    gettimeofday(&finish_time, (struct timezone *) 0) ;
    elapsed_time = ((double)((finish_time.tv_usec - start_time.tv_usec) *
            0.001 + ((finish_time.tv_sec - start_time.tv_sec) * 1000.0 ))) ;
    return (elapsed_time / ((double) count)) ;
}

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv) ;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc) ;
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc) ;

    if (nproc != 2 ) perror("Fatal run time error : number of process-
      ors must be 2") ;
    {
        int i ;
        double elapsed_time ;
        buffer = (double *) malloc(512*1024*sizeof(char)) ;
        for (i = 0 ; i < 512 * 1024 ; i++) buffer[i] = 1 ;

        elapsed_time = roundtrip(1, 512*1024) ;
        if (iproc == 1) {
            printf("Total Time(512Kbytes) : %.2f ms\t," elapsed_time) ;
            printf("Speed : %.2f Kbytes/ms\n," 512/elapsed_time) ;
         }
         free(buffer) ;
     }
     MPI_Finalize() ;
     return 0 ;
}
```

4. 성능 평가

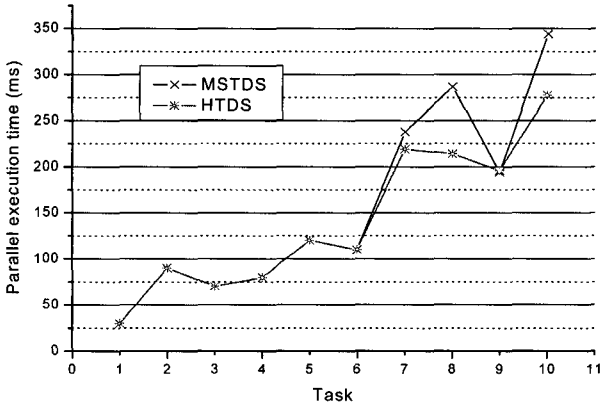
본 논문에서는 스케줄링 기법의 성능을 평가하기 위하여 HTDS 기법과 MSTDS 기법을 PC 클러스터에 구현하였다. 클러스터 환경에서 스케줄링 길이를 줄이고 병렬연산시간을 단축하기 위하여 태스크 중복을 허용한다. 또한 버스 구조의 특성에 따라 스케줄링 기법에서는 클러스터에서 통신할 때 발생되는 충돌을 방지하기 위하여 네트워크 통신 자원을 우선 할당한다. HTDS 기법에서는 스케줄링 길이를 단축하고 병렬처리 시간을 줄이기 위한 중복 태스크를 선택할 때 휴리스틱을 사용한다.

(그림 5)는 (그림 1)(a) 태스크 그래프를 실제 PC 클러스터에서 수행시켜 스케줄링되는 과정을 병렬연산시간 측면에서 나타낸 것이다. (그림 5)에서는 본 과제에서 개발한 HTDS 기법이 MSTDS 기법보다 병렬연산시간이 짧아서 HTDS 스케줄링 기법의 성능이 우수함을 보여준다. (그림 1)(a) 태스크 그래프의 병렬연산시간은 태스크 10(n_{10})이 종료되는 시

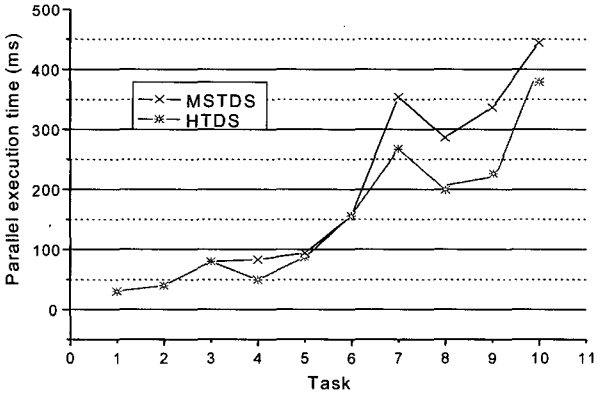
간을 비교하면 된다. 응용 프로그램의 실행이 종료되는 태스크 10을 기준으로 HTDS가 MSTDS보다 병렬연산시간 측면에서 16.5%의 성능이 향상되었음을 확인할 수 있다. 이와 같은 결과는 본 과제에서 개발한 스케줄링 기법은 클러스터의 통신특성을 파악하여 통신비용을 줄일 수 있도록 효과적인 휴리스틱을 사용하였기 때문이다.

(그림 6)는 (그림 1)(b) 태스크 그래프를 실제 PC 클러스터에서 수행시켜 스케줄링되는 과정을 병렬연산시간 측면에서 나타낸 것이다. (그림 6)은 (그림 5)와 유사한 결과를 보여주며 HTDS 기법이 MSTDS 기법보다 병렬연산시간 측면에서 성능이 우수함을 확인할 수 있다. 태스크의 실행이 종료되는 시각을 기준으로 HTDS 기법이 MSTDS 기법보다 병렬연산시간 측면에서 12.6%의 성능이 향상되었음을 (그림 6)에서 확인할 수 있다. (그림 5)와 (그림 7)의 결과를 종합하면, PC 클러스터에 구현한 스케줄링 기법의 성능비교에서 HTDS 기법이 MSTDS 기법보다 평균 14.6%의 병렬연산시간을 단축한 것을 확인할 수 있다.

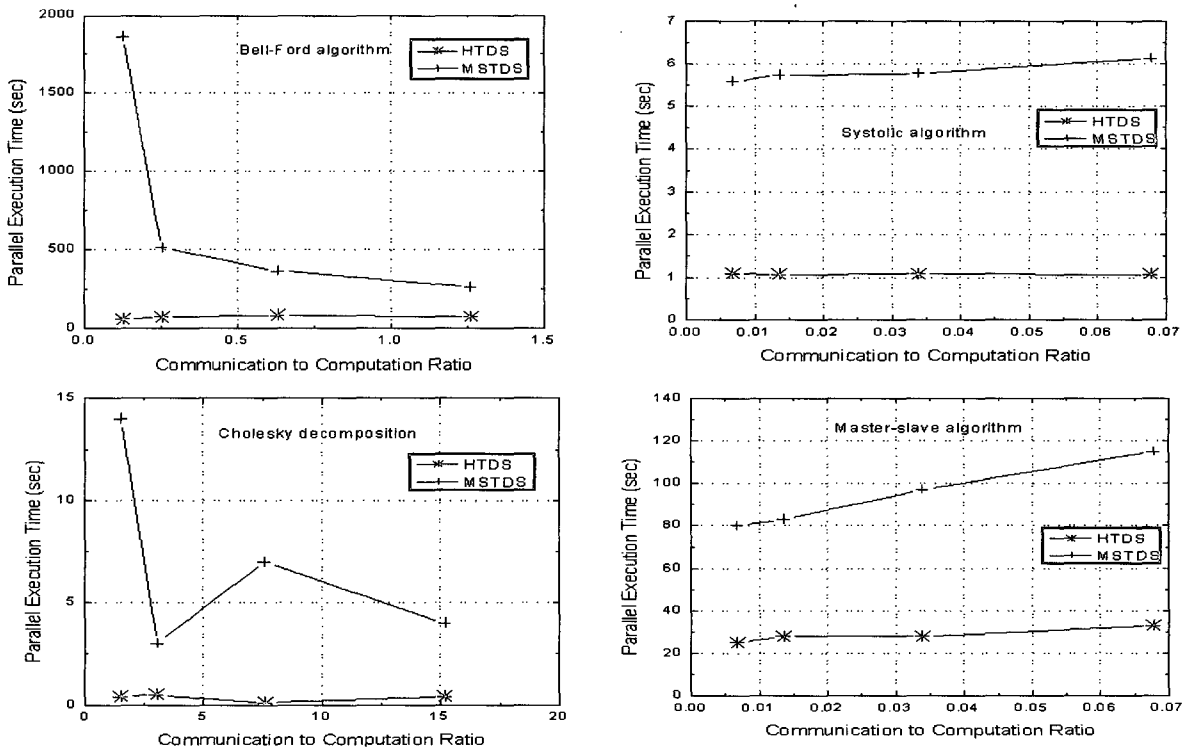
다양한 태스크 그래프와 클러스터 노드수의 변화에 대한 태스크 스케줄링 기법의 병렬연산시간을 확인하기 위한 시뮬레이션을 수행하였으며 그 결과는 (그림 7)과 같다. 4가지 응용 프로그램의 실제 DAG를 사용하여 통신비용과 연산비용의 변화에 대한 병렬연산시간을 측정하였다. (그림 7)에서 HTDS 기법이 MSTDS 기법보다 병렬연산시간 측면에서 성능이 우수한 것을 확인할 수 있으며, 이것은 HTDS 기법에서 태스크 중복을 위해 효과적인 휴리스틱을 사용한 것에 기인한 것이다.



(그림 5) 스케줄링 기법에 따른 병렬연산시간의 비교



(그림 6) 스케줄링 기법에 따른 병렬연산시간의 비교



(그림 7) 응용 프로그램 DAG를 사용한 병렬연산시간의 비교

5. 결 론

본 논문에서는 버스 구조의 클러스터 환경에서 적용할 수 있는 태스크 스케줄링 기법을 제안하고 PC 클러스터에 구현하여 성능을 분석하였다. 구현된 스케줄링 기법은 태스크 중복을 기반으로 하며, 스케줄링 길이를 줄이기 위한 휴리스틱을 사용하여 태스크를 선택적으로 중복한다. 스케줄링 기법을 구현하기 위하여 기가비트 이더넷(Ethernet)으로 연결되고 리눅스와 MPI 환경으로 동작하는 PC 클러스터를 구축하였다. 클러스터에서 스케줄링 알고리즘의 성능을 비교한 결과 본 논문에서 제안한 HTDS 기법이 기존 알고리즘인 MSTDS 기법보다 성능이 향상되었음을 보여주었다.

참 고 문 헌

[1] 특집 클러스터 컴퓨팅, 정보과학회지, 제18권 제3호, 2000.

[2] D. E. Culler, et al., A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong, "Parallel Computing on the Berkeley NOW," Joint Symp. Parallel Processing, 1997.

[3] C. Huang and P. K. Mckinley, "Communication Issues in Parallel Computing Across ATM Networks," IEEE Parallel and Distributed Technology, Vol.2, No.4, pp.73-86, 1994.

[4] Y. Dong, X. Du and X. Zhang, "Characterizing and Scheduling Communication Interactions of Parallel and Local Jobs on Networks of Workstations," Computer Communications, Vol.21, Issue 5, pp.470-484, 1998.

[5] X. Zhang and Y. Yan, "Modeling and Characterizing Parallel Computing Performance on Heterogeneous NOW," J. of Parallel and Distributed Computing, Vol.36, No.1, 1996.

[6] X. Du and X. Zhang, "Coordinating Parallel Processes on Networks of Workstations," J. of Parallel and Distributed Computing, Vol.46, pp.125-135, 1997.

[7] A. Gereasoulis and T. Yang, "A Comparison of Clustering Heuristics for scheduling Directed Acyclic Graphs on Multiprocessors," Journal of Parallel and Distributed Computing, Vol.16, pp.276-291, 1992.

[8] S. Darbha and D. P. Agrawal, "A Task Duplication Based Scalable Scheduling Algorithm for Distributed Memory Systems," Journal of Parallel and Distributed Computing, Vol.46, 1997, pp.15-26.

[9] G. L. Park, B. Shirazi and J. Marquis, "DFRN : A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems," Proc. of Parallel Processing Symposium, pp.157-166, 1997.

[10] S. Nagar, A. Banerjee, A. Sivasubramaniam and C. R. Das, "An Experimental Study of Scheduling Strategies for a Network of Workstations," Technical Report CSE-98-009, July, 1998.

[11] O. Sinnen and L. Sousa, "Experimental Evaluation of Task Scheduling Accuracy : Implications for the Scheduling Model," IEICE Trans., on Information and Systems, Vol. E86-D, No.9, pp.1620-1627, 2003.

[12] O. Sinnen and L. Sousa, "Scheduling Task Graphs on Arbitrary Processor Architectures Considering Contention," Proc., of High-Performance Computing and Networking, pp.373-382, 2001.

[13] http://www.mpi-forum.org.

[14] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," MIT press, Cambridge, MA, 1989.

[15] S. Darbha and D. P. Agrawal, "Optimal Scheduling Algorithm for Distributed-Memory Machines," IEEE Trans. on Parallel and Distributed Systems, Vol.9, No.1, pp.87-95, 1998.

강 오 한

e-mail : ohkang@andong.ac.kr

1982년 경북대학교 전자계열 전산모듈

1984년 한국과학기술원 전산학과 석사

1992년 한국과학기술원 전산학과 박사

1984년~1994년 (주) 큐닉스컴퓨터

1994년~현재 안동대학교 컴퓨터교육과 부교수

관심분야 : 태스크 스케줄링, OVPN

송 희 헌

e-mail : hhsong@andong.ac.kr

1986년 동국대학교 컴퓨터공학과(공학사)

1992년 충남대학교 컴퓨터과학과 (이학석사)

1995년 충북대학교 컴퓨터과학과 (이학박사)

1988년~1998년 한국전자통신연구원 선임연구원

1998년~현재 안동대학교 컴퓨터교육과 조교수

관심분야 : 컴퓨터교육(WBI), 신경망, 차세대통신망

정 중 수

e-mail : jschung@andong.ac.kr

1981년 영남대학교 전자공학과(학사)

1983년 연세대학교 전자공학과(석사)

1993년 연세대학교 전자공학과(박사)

1983년~1994년 ETRI 연구원, 선임 연구원

1987년~1989년 벨지움 Alcal/Bell Telephone사 객원연구원

2000년~2001년 미국 UMASS/Lowell 전산학과 객원교수

1994년~현재 국립 안동대학교 공과대학 전자정보산업학부 부교수