# 최소의 스케줄 길이를 유지하는 결함 허용 태스크 스케줄링 알고리즘

민 병 준†


## 요 약

고성능 병렬 컴퓨터 시스템에서는 분산되어 있는 태스크의 수행 중 발생하는 결함에 대비하기 위하여 하나의 태스크를 서로 다른 프로세서에 중복하여 할당한다. 본 논문에서는 태스크 중복에 기반을 둔 스케줄링 알고리즘을 이용하여 최소의 스케줄 길이를 보장하면서 모든 태스크를 두 개 이상의 서로 다른 프로세서에 할당하는 태스크 스케줄링 알고리즘을 제시하고, 이 때 필요한 프로세서의 개수를 언신시간 대비 통신비용과 작업부하와의 관계로 규명한다. 다양한 형태의 태스크 그래프에 대한 시뮬레이션 결과, 산출될 수 있는 최소의 스케줄 길이를 그대로 유지하면서 모든 태스크를 중복 할당하는데 필요한 프로세서 개수는 최소 태스크 중복에 기반을 둔 스케줄링 알고리즘의 경우에 비하여 대략 30% ~ 75% 정도 증가하는 것으로 나타났다.


# A Fault-tolerant Task Scheduling Algorithm Supporting the Minimum Schedule Length

Byoung-Joon Min†


## ABSTRACT

In order to tolerate faults which may occur during the execution of distributed tasks in high-performance parallel computer systems, tasks are duplicated on different processors. In this paper, by utilizing the task duplication based scheduling algorithm, a new task scheduling algorithm which duplicates each task on more than two different processors with the minimum schedule length is presented, and the number of processors required for the duplication is analyzed with the ratio of communication cost to computation time and the workload of the system. A simulation with various task graphs reveals that the number of processors required for the full-duplex fault-tolerant task scheduling with the obtainable minimum schedule length increases about 30% to 75% when compared with that of the task duplication based scheduling algorithm.


## 1. 서 론

반도체와 통신 기술의 발달에 따라 고성능 병렬 컴퓨터가 보편화되고 있다. 특히 안전도가 중요시되는 국방, 의료, 금융 등의 응용 분야에서 고성능 병렬 컴퓨터의 수요가 늘고 있다. 분산 태스크의 실행 중에 발생 가능한 소프트웨어 및 하드웨어 결함을 포용할 수 있도록 각 태스크를 서로 다른 두 개 이상의 프로세서에서 동시 수행시키는 기법들이 이러한 응용 분야에 적용될 수 있다[10, 11, 12, 14]. 특히 복구블럭(recovery block) 기법을 분산 시스템에 적용시킨 분산복구블럭(distributed recovery block)과 같은 기법을 활용하면 효과적으로 하드웨어와 소프트웨어의 결함을 포용할 수 있다. 이

※ 본 연구는 한국과학재단 특정기초연구(96-0101-07-01-3) 지원에 의한 것임
† 종신회원 인천대학교 컴퓨터공학과 교수
논문접수 2000년 2월 16일, 심사완료 2000년 4월 1일

기법에서는 동일한 태스크를 수행하도록 설계된 소프트웨어 모듈이 두 개 이상의 서로 다른 프로세서에 할당되어 동시에 수행되고 수행 결과에 대한 수용검사(acceptance test)를 통과한 주(primary) 모듈이 결과를 다음 태스크에 전달한다. 만일 주 모듈이 수용검사에서 실패하거나 다른 이유로 정해진 시간 내에 결과를 출력하지 못하면 다른 부(shadow) 모듈이 주 모듈의 다음 임무를 수행하여 외부에서 볼 때 투명하게 결함에 대한 대응이 이루어진다. 이를 위해서는 모든 태스크들을 중복하여 프로세서에 할당하되 최소한의 수행시간을 유지하면서 프로세서의 개수를 가능한 줄일 필요가 있다.

일반적으로 태스크 스케줄링은 각 프로세서에 할당할 태스크들과 태스크가 수행되는 순서를 결정하는 작업을 말한다. 여러 개의 태스크로 분할된 응용 프로그램을 추상화한 태스크 그래프는 DAG(directed acyclic graph) 형태로 표현된다. 각 노드는 태스크를 나타내고 노드를 연결하는 화살표는 노드 간의 통신비용을 나타낸다. 스케줄링 기법들은 대개 두 가지 목표를 추구하는데 하나는 전체 태스크의 수행시간을 줄이는 것이고, 또 다른 목표는 시스템 자원의 활용도를 높이는 것이다[6, 7]. 다중 프로세서 시스템에서 태스크 스케줄링은 NP-complete 문제로 밝혀졌고[8], 최근에 다양한 휴리스틱을 기반으로 한 알고리즘들이 발표되었다 가장 간단한 형태는 우선순위에 기반을 둔 것이다[1, 4] 사용 가능한 프로세서가 나타나면 준비가 된 태스크들 중에서 우선순위가 가장 높은 것을 선택하는 방법이다. 이 방법의 단점은 프로세서간 통신비용을 고려하지 않는다는 것이다 또 다른 방법은 클러스터 개념을 이용한 것으로 통신량이 많은 태스크들을 같은 프로세서에 할당하는 방법이다[2, 3, 5]. [6, 7]에서는 태스크 중복을 기반으로 최적의 스케줄 길이를 갖는 알고리즘(task duplication based scheduling)을 제시하였다. 이 알고리즘은 태스크들 간의 통신비용을 고려하여 비용이 많이 드는 태스크를 이와 연결된 태스크가 할당된 동일한 프로세서에서 할당하면 스케줄 길이를 최소화 할 수 있다는 개념에 바탕을 둔 것이다. 여기서는 중복이 반드시 필요한 경우에만 하도록 하고 있으며, 간단한 조건을 만족하면 최적의 수행시간을 제공하는 것으로 증명되었다. [11]에서는 결함 허용을 위한 태스크 스케줄 문제가 하이퍼큐브나 메쉬와 같은 특정 다중 프로세서 구조에서 다루어졌으며, [9, 13]에서는 결함허용을 위해 복제된 태스크가 정해진 시간 내에 스케줄될 수 있는가를 판단하는 방법들이 연구되었다.

본 논문에서는 태스크 중복을 기반으로 하는 알고리즘을 이용하여 모든 태스크들이 서로 다른 두개 이상의 프로세서에 중복 할당되도록 하여 결함허용 기법을 수용하도록 하면서 최소수행시간을 갖는 알고리즘을 제시한다. 모든 태스크들을 중복 할당하는데 추가로 요구되는 프로세서의 개수를 최대한 줄일 수 있도록 고안하였으며 요구되는 프로세서의 개수를 연산시간 대비 통신비용과 작업부하와의 관계로 분석하였다.

본 논문의 나머지 구성은 다음과 같다. 2장에서는 태스크 중복 스케줄링에 사용되는 용어와 본 논문에서 해결하려는 문제에 대하여 정의한다. 3장에서 결함허용을 위한 태스크 중복 스케줄링 알고리즘을 제시하고 태스크 그래프의 예를 들어 동작 과정을 설명한다. 4장에서는 제시된 알고리즘을 성능 분석을 위하여 수행된 시뮬레이션 결과를 소개하고 그 결과 분석에 대하여 논한다. 그리고 마지막으로 5장에서 결론을 맺는다.

## 2. 용어 및 문제 정의

### 2.1 용어 정의

일반적으로 응용 프로그램은 태스크를 단위로 하여 나누어지고 그 결과는 DAG(directed acyclic graph) 형태의 태스크 그래프로 묘사된다 태스크 $t_i$를 수행하는데 필요한 연산시간은 $\tau(t_i)$이고, 태스크 $t_i$에서 태스크 $t_j$로의 통신비용은 $c(t_i, t_j)$이다.

$$pred(t_i) = \{ t_j \mid c(t_j, t_i) \neq \infty \} \quad (1)$$

$$succ(t_i) = \{ t_j \mid c(t_i, t_j) \neq \infty \} \quad (2)$$

하나의 태스크 그래프에는 하나의 입력 태스크 노드와 하나의 출력 태스크 노드가 있는 것으로 간주한다. 실제 그렇지 않은 경우에는 연산시간이 0이고 이와 연결되는 통신비용이 0인 가상의 입력 태스크 노드와 출력 태스크 노드를 둘 수 있다. 태스크들 간의 관계는 부모(predecessor) 또는 자식(successor)으로 나타낼 수 있다. 예를 들어 태스크 $t_i$에서 $t_j$로 연결되는 화살표가 존재하면 $t_i = pred(t_j)$, $t_j = succ(t_i)$의 관계가 성립된다.

$$est(t_i) = 0, \text{ if } pred(t_i) = \phi \quad (3)$$

$$est(t_i) = \min(\max(ect(t_j), ect(t_k)+c(t_k,t_i))), \quad \forall t_j, t_k \text{ IN } pred(t_i), \text{where } j \neq k, \text{ if } pred(t_i) \neq \phi \tag{4}$$

$$ect(t_i) = est(t_i) + \tau(t_i) \tag{5}$$

$$fpred(t_i) = t_j \mid (ect(t_j)+c(t_j,t_i)) \geq (ect(t_k)+c(t_k,t_i)), \forall t_j, t_k \in pred(t_i), \text{where } j \neq k \tag{6}$$

태스크 그래프가 주어지면 태스크 수행을 완료하는데 필요한 최소 수행시간, 즉 최소 스케줄 길이를 구할 수 있다 태스크 $t_i$의 수행을 시작하고 완료할 수 있는 가장 빠른 시간을 각각est($t_i$) (earliest start time of $t_i$)와 ect($t_i$) (earliest completion time of $t_i$)로 나타낸다. 입력 태스크에 대한 est를 0으로 시작하여 마지막으로 출력 태스크에 대한 ect을 구하는데 출력 태스크의 ect가 바로 전체 태스크의 스케줄 길이가 된다. 두 개 이상의 pred를 갖는 결합(join) 노드는 모든 pred로부터 결과를 받아야만 수행을 시작할 수 있다. 예를 들어서 태스크 $t_i$에 세 개의 pred 태스크, $t_a$, $t_b$, $t_c$가 있다고 하면 $t_i$가 $t_a$와 같은 프로세서에 할당되었다고 가정할 경우, 또, $t_i$가 $t_b$와 같은 프로세서에 할당되는 경우, 마지막으로 $t_i$가 $t_c$와 같은 프로세서에 할당되는 경우의 각각의 빠른 시작 시간을 비교하여 세 가지 중 가장 적은 값을 est($t_i$)로 한다. 이 때 최소값을 제공하는 pred를 fpred($t_i$) (preferred predecessor of $t_i$)라고 한다.

$$lct(t_i) = ect(t_i). \text{ if } succ(t_i) = \phi \tag{7}$$

$$lct(t_i) = \min(\min(lst(t_j)-c(t_i,t_j)), \min(lst(t_k))), \quad \forall t_j, t_k \in succ(t_i), \text{where } t_i \neq fpred(t_j) \text{ and } t_i = fpred(t_k), \text{if } succ(t_i) \neq \phi \tag{8}$$

$$lst(t_i) = lct(t_i) - \tau(t_i) \tag{9}$$

$$level(t_i) = \tau(t_i), \text{ if } succ(t_i) = \phi \tag{10}$$

$$level(t_i) = \max(level(t_k)) + \tau(t_i), \forall t_k \in succ(t_i), \text{ if } succ(t_i) \neq \phi \tag{11}$$

출력 태스크 노드의 ect가 구해지면 이제부터는 거꾸로 즉, 출력 태스크 노드에서 입력 태스크 노드의 방향으로 진행하면서 최소 스케줄 길이를 보장하기 위해서 태스크 $t_i$ 가 실행을 완료해야 할 마감시간인 lct($t_i$) (latest completion time of $t_i$)와 가장 늦은 시작 시간인 lst($t_i$) (latest start time of $t_i$)를 구하게 된다. 태스크 $t_i$의 succ중 하나인 태스크 $t_k$ 가 있고 태스크 $t_i$가 태스크 $t_k$의 fpred라면 두 태스크는 같은 프로세서에 할당될 수 있으므로 그것을 가정하여 lct($t_i$)는 lst($t_k$)로 하고 그렇지 않으면 두 태스크 사이의 통신비용을 고려하여 정한다. level은 출력 태스크 노드에서부터 입력 태스크 노드의 순서로 연결된 태스크 연산시간의 합이다.

위에서 정의된 용어와 정의식 들은 [5, 6, 7]에서 사용된 것들을 바탕으로 다시 기술한 것이다. 지금부터는 결함 허용 태스크 중복 알고리즘을 위한 새로운 용어 정의에 대하여 논한다. 태스크가 특정 프로세서에 할당되면 해당 프로세서의 상황에 따라서 est, lct의 값이 달라지게 되므로 다음과 같은 정의가 필요하다. 이에 대한 예는 3장에서 설명하기로 한다.

$$atc(p_x) = \{ t_{x,i} \mid t_{x,i} \text{ . task allocated to processor } p_x, \text{ where } t_{x,i} \text{ precedes } t_{x,j} \text{ in } p_x, \text{ if } \langle i, \text{where } i,j = 1,2,\ldots, n_x, n_x \text{ is the total number of tasks allocated to } p_x \} \tag{12}$$

$$pat(t_{x,i}) = \{ t_{x,j} \mid 1 \leq j < i \} \tag{13}$$

$$sat(t_{x,i}) = \{t_{x,j} \mid i < j \leq n_x\} \tag{14}$$

태스크 그래프의 태스크들 중에서 태스크 $t_a$, $t_b$ , $t_c$가 같은 프로세서 $p_x$에 할당되었다고 가정하면 이들의 집합을 atc($p_x$) (allocated task cluster on processor $p_x$)라고 한다. 그리고 태스크 $t_a$, $t_b$, $t_c$가 수행되는 순서를 나타내기 위하여 i번째 수행되는 태스크를 $t_{x,i}$로 나타낸다 예를들어 $t_a$, $t_b$, $t_c$의 순서대로 수행된다면 $t_{x,1} = t_a$, $t_{x,2} = t_b$, $t_{x,3} = t_c$가 되고 $t_a$는 pat($t_b$) (preceding allocated task of $t_b$)이고 $t_c$는 sat($t_b$) (succeeding allocated task of $t_b$)가 된다.

$$esta(t_{x,i}) = est(t_{x,i}), \text{ if } est(t_{x,i}) = lst(t_{x,i}) \tag{15}$$

$$esta(t_{x,1}) = 0,$$

$$esta(t_{x,i}) = \max(ecta(t_{x,i-1}), ecta(t_{y,j}) + c(t_{y,j}, t_{x,i})), \forall t_{y,j} \in pred(t_{x,i}), \text{ where } x \neq y, 1 < i \leq n_x, \text{ if } est(t_{x,i}) \neq lst(t_{x,i}) \tag{16}$$

$$ecta(t_{x,i}) = esta(t_{x,i}) + \tau(t_{x,i}) \tag{17}$$

태스크가 특정 프로세서에 할당되면 해당 프로세서의 상황에 따라서 실제로 시작 가능한 시간과 종료되는 시간이 달라질 수 있다 프로세서 $p_x$에서 i번째 수행되도록 스케줄된 태스크 $t_{x,i}$의 실제 가장 빠른 시작 시간과 종료

시간을 각각 esta($t_{x,j}$) (earliest start time of task allocated as $t_{x,j}$)와 ecta($t_{x,j}$) (earliest completion time of task allocated as $t_{x,j}$)라고 한다. est($t_{x,j}$) 와 lst($t_{x,j}$)가 서로 같다면 태스크 $t_{x,j}$를 수행하는데 있어서 여유 시간이 없음을 의미한다. 따라서 실제 esta($t_{x,j}$) = est($t_{x,j}$)을 만족하도록 해야 한다. 그렇지 않은 경우에는 pat($t_{x,j}$) 중에서 바로 먼저 수행되는 태스크의 빠른 종료 시간 ecta($t_{x,j-1}$) 이후, 그리고 모든pred($t_{x,j}$)로부터 결과값을 받은 이후에 시작할 수 있게 된다.

$$\begin{aligned} &\text{lcta}(t_{x,j}) = \text{lct}(t_{x,j}), \text{ if } \text{est}(t_{x,j}) = \text{lst}(t_{x,j}) \\ &\qquad (\text{that is, if } \text{ect}(t_{x,j}) = \text{lct}(t_{x,j})) \end{aligned} \tag{18}$$

$$\begin{aligned} &\text{lcta}(t_{x,nx}) = \text{lct}(t_{x,nx}), \\ &\text{lcta}(t_{x,i}) = \min(\text{lsta}(t_{x,i+1}), \text{lsta}(t_{y,j}) \\ &\qquad - c(t_{x,i}, t_{y,j})), \ \forall t_{y,j} \in succ(t_{x,i}), \\ &\qquad \text{where } x \neq y, 1 \le i \le n_x, \\ &\qquad \text{if } est(t_{x,i}) \neq \text{lst}(t_{x,i}) \end{aligned} \tag{19}$$

$$\text{lsta}(t_{x,i}) = \text{lcta}(t_{x,i}) - \tau(t_{x,i}) \tag{20}$$

각 프로세서에 대해서 esta와 ecta를 계산하고 난 후 lcta($t_{x,i}$) (latest completion time of task allocated as $t_{x,i}$)와 lsta($t_{x,i}$) (latest start time of task allocated as $t_{x,i}$)를 구하게 된다 모든 succ($t_{x,i}$)가 lsta이전에 시작될 수 있도록 통신비용이 고려된다.

$$\begin{aligned} &sit(x,j) = \text{ecta}(t_{x,j-1}), eit(x,j) = \text{lsta}(t_{x,j}), \\ &\qquad \text{if } (\text{lsta}(t_{x,j}) - \text{ecta}(t_{x,j-1})) > 0 \end{aligned} \tag{21}$$

$$\begin{aligned} &sit(x,k) = ecta(t_{x,nx}), eit(x,k) = schedule\ length, \\ &\qquad \text{if } \text{ecta}(t_{x,nx})) < \text{schedule length} \end{aligned} \tag{22}$$

$$\begin{aligned} &\text{netidletime}(t_{x,j}) = \text{schedule length} \\ &\qquad - \text{ecta}(t_{x,j}) - \sum(\tau(\text{sat}(t_{x,j})) \end{aligned} \tag{23}$$

프로세서에 태스크들이 할당되고 나면 아무런 태스크를 수행하지 않아도 되는 여유 시간(idle timeslot)이 존재할 수 있다. sit(x,j)와 eit(x,j)는 프로세서 $p_x$에 존재하는 j번째 시간 블록의 시작과 끝(start and end of idle timeslot)을 말한다. 그 프로세서에서 마지막으로 스케줄되는 태스크 $t_{x,m}$의 ecta($t_{x,nx}$)가 스케줄 길이 보다 작으면 $ecta(t_{x,nx})$와 스케줄 길이 사이의 시간이 마지막 여유 시간 블록이 된다. netidletime ($t_{x,j}$)는 $t_{x,j}$ 수행 이후에 프로세서 $p_x$에 남아있는 순수한 전체 여유 시간을 나타낸다.

### 2.2 문제 정의

본 논문에서 다룰 문제는 주어진 태스크 그래프에 대해서 최소의 스케줄 길이를 유지하면서 그래프 내의 모든 태스크들이 서로 다른 두개 이상의 프로세서에 중복 할당되도록 하는 것이다. 이와 동시에 모든 태스크 중복 할당하는데 요구되는 프로세서의 개수를 최대한 줄일 수 있는 방법을 찾는 것이다.

이를 해결하기 위하여 다음과 같은 모델을 가정한다.

(1) 프로세서

모든 프로세서는 동기종이다 각 프로세서는 충분한 메모리에 접근할 수 있어서 하나의 프로세서에서 여러 개의 태스크 수행을 위한 충분한 공간이 확보될 수 있다. 그리고, 같은 프로세서 내에서의 태스크간 통신비용은 무시한다

(2) 상호연결망

프로세서 간 연결을 위한 상호연결망은 모든 프로세서에 대칭이며 충돌이 일어나지 않는 망이다 또한 프로세서의 망 접속에는 별도의 I/O 처리장치가 있어서 태스크 연산과 I/O 처리가 동시에 진행될 수 있다.

(3) 태스크 그래프

주어진 DAG 형태의 태스크 그래프에는 하나의 입력 태스크 노드와 하나의 출력 태스크 노드가 존재한다. 태스크 $t_i$에 두 개 이상의 pred 태스크가 존재해서 ect(pred($t_i$)) + c(pred($t_i$), $t_i$)가 가장 큰 값을 갖는 pred를 $t_m$, 그 다음으로 큰 값을 갖는 pred를 $t_n$이라고 할 때 다음 조건을 만족한다.

$$\begin{aligned} &\tau(t_m) \ge c(t_{n,i}), \text{if } est(t_m) \ge est(t_n) \\ &\tau(t_m) \ge c(t_n, t_i) + \text{est}(t_n) - \text{est}(t_m), \text{if } \text{est}(t_m) < \text{est}(t_n) \end{aligned}$$

위 조건은 기존의 중복 기반 태스크 스케줄링 알고리즘 결과 얻어진 스케줄 길이가 최적이기 위한 것이다[6, 7].

(4) 태스크 실행 규칙

하나의 태스크는 모든 pred 태스크의 수행 결과가 얻어져야 실행될 수 있다. 하나의 태스크는 할당된 프로세서에서만 수행되며 비선점(non-preemptive)이다. 중복 할당된 모든 태스크가 완료되어야 해당 태스크가 완료된 것으로 간주하고 중복 할당된 태스크들이 결함 허용을 위해 추가 수행해야 하는 연산이나 태스크들

간의 통신비용은 무시한다.

## 3. 결함허용 태스크 스케줄링 알고리즘

3장에서는 2장에서 정의된 문제 해결을 위한 알고리즘에 대하여 설명한다.

본 논문에서 제시하는 알고리즘은 개념적으로 볼 때 크게 두 부분으로 나누어 생각할 수 있다. 첫번째 부분은 [6, 7]의 알고리즘을 이용하여 태스크 스케줄을 만들어 내는 것이다. 이 과정에서는 태스크간의 통신비용이 상대적으로 큰 경우 이 태스크들를 동일한 프로세서에 할당하여 최소 스케줄을 산출하고 태스크 중복은 반드시 필요한 경우에만 한다 그리고, 두 번째 부분에서 중복되지 않은 태스크들에 대해서 중복을 실시한다. 중복을 만들어 내는 방법으로 현재 스케줄에 사용 중인 프로세서에 존재하는 여유 시간 블록을 찾아내어 그 안에 들어갈 수 있으면 그 프로세서에 중복시키고, 그렇지 않으면 새로운 프로세서를 추가하여 태스크를 중복 할당하는 것이다. (그림1)에 알고리즘의 전 과정을 자세히 나타내었다. 이 알고리즘은 태스크의 연산시간과 태스크들간의 통신비용을 입력 받아 크게 네 단계의 처리를 거친 후 모든 태스크들이 중복 할당되고 최소의 스케줄 길이를 보장하는 스케줄 결과를 출력한다. 그림의 1단계와 2단계는 앞서 설명한 알고리즘 개념적 구성의 첫번째 부분에 해당하고 3단계와 4단계가 두 번째 부분에 해당한다.

1단계에서는 식 (3)부터 식 (11)에 정의된 바와 같이 입력 태스크에서 출력 태스크의 순서로 est, ect, fpred를 계산하고 반대의 순서로 lct, lst, level을 계산한다. 태스크들을 level 값의 오름차순으로 정렬하여 큐 Qlevel에, lct 값의 오름차순으로 정렬하여 큐 Qlct에 저장한다.

2단계에서 최소 스케줄 길이를 갖는 태스크 할당을 한다. (2.1)에서 Qlevel의 첫번째 태스크를 프로세서에 할당한다 그리고 (2.2)에 따라 그 태스크의 fpred 노드도 같은 프로세서에 할당한다. 이 과정을 입력 태스크 노드에 이르게 될 때까지 계속한다. (2.3)에서 새로운 프로세서를 추가하고 (2.4)에 명시된 바와 같이 Qlevel에서 아직 할당되지 않은 태스크를 선택한다 (2.2)단계를 반복 수행하되 만일 fpred가 이미 다른 프로세서에 할당되어 있는 경우에는 태스크들이 불필요하게 중복 할당되는 것을 피하기 위하여 그 fpred외에 아직 할당되지 않은 다른 pred가 존재하고 이 pred를 같은 프로세서에 할당해도 무방하다면 fpred 대신에 그 pred를 할당한다. 이렇게 하면 최소의 스케줄 길이를 보장하면서 일부 태스크가 중복 할당된 태스크 스케줄이 얻어진다.

3단계에서는 2단계에서 얻어진 결과에 대하여 각 프로세서 별로 식 (15)부터 식 (20)의 정의에 따라 할당된 후 태스크의 가장 빠른 시작 시간과 종료 시간, esta와 ecta를 입력 태스크에서 출력 태스크의 순서로 계산하고, 반대 순서로 할당 후 태스크 가장 늦은 시작 시간과 종료 시간, lcta와 lsta를 계산한다. 이어서 식 (21)과 식 (22)에 정의된 여유 시간 블록을 프로세서 별로 구한다.

Input : a set of tasks $\{t_i\}$ with the corresponding computation time $\tau(t_i)$ and a set of communication costs between tasks $\{ c(t_i,t_j) \}$

Step 1. compute $est(t_i)$, $ect(t_i)$, $fpred(t_i)$, $lct(t_i)$, $lst(t_i)$, and $level(t_i)$ in the order for all tasks and sort the tasks in ascending order of level and in ascending order of lct and store them in queues, $Q_{level}$, $Q_{lct}$, respectively

Step 2 allocate all tasks to processors

*allocate the first task in the queue Qlevel to a processor;* (2.1)

*while (not all the tasks are assigned) {*

*while (not reached to the input task node) {*

*if fpred($t_i$) is already allocated {*

*if (lct(fpred($t_i$)) + c(fpred($t_i$),$t_i$))≤lst($t_i$) then find another pred($t_i$) which is not allocated yet and ect(pred($t_i$)) + c(pred($t_i$),$t_i$) is maximum;*

*if found, allocate the pred($t_i$) to the same processor as $t_i$ ;*

*} else allocate fpred($t_i$) to the same processor as $t_i$ ;* (2.2)

*}*

*get a new processor;* (2.3)

*get a none-allocated task $t_i$ from the queue and allocate it to the new processor;* (2.4)

*}*

Step3. compute idle timeslots on each processor between time zero and the schedule length for each processor $p_x$ with the allocated tasks $t_{x,1}, t_{x,2}, .., t_{x,m}$, compute esta, ecta, lcta, lsta in the order such that

if (lsta($t_{x,j}$) - ecta($t_{x,j-1}$)) > 0

then sit(x,k) = ecta($t_{x,j-1}$), eit(x,k) = lsta($t_{x,j}$)

if ecta($t_{x,m}$)) < schedule length

then sit(x,k) = ecta($t_{x,mx}$), eit(x,k) = schedule length

Step 4. duplicate a (non-duplicated) task node and go back to Step 3 if not all the tasks are duplicated

*pick a non-duplicated task $t_i$ from $Q_{lct}$ ($t_i$ was allocated on $p_x$ as $t_{x,j}$ in Step 2);* (4.1)

*search for all the idle time slots among currently used processors (e.g., $p_y$) except $p_x$ which satisfy the following conditions;* (4.2)

*[condition 1] (eit(y,k)-sit(y,k) ≥ $\tau(t_i)$) and (eit(y,k) ≥ lst($t_i$))*

*[condition 2] (pred($t_i$) ∈ pat($t_{y,k'}$)) or (ecta(pred($t_i$)) + $c$(pred($t_i$),$t_i$) ≤ lsta($t_{y,k'}$)) for all pred($t_i$)), where k' represents the location which task $t_i$ is inserted into*

*[condition 3] netidletime($t_{y,k'}$) ≥ Σ $\tau$ (non-duplicated sat($t_{x,j}$)) , where k' represents the location which task $t_i$ is inserted into*

*if such idle time slot is found on one processor then duplicate $t_i$ on the (earliest) time slot on the processor and adjust pat and sat relationship;* (4.3)

*else if such idle time slots are found on more than two processors then select a processor satisfies the following condition;* (4.4)

*[condition 4] pat($t_{x,j}$) = pat($t_{y,k'}$) or*

*[condition 5] (eit(y,k)-sit(y,k)- $\tau(t_i)$) is minimum*

*else (not found) then get a new processor and allocate all the pat($t_{x,j}$) and $t_{x,j}$ on it;* (4.5)

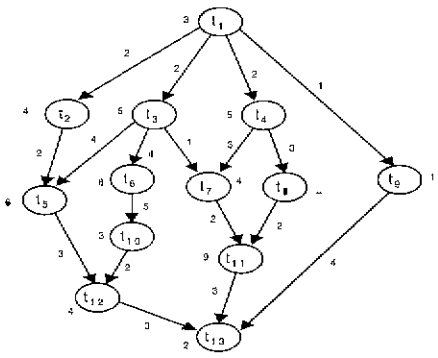
*if not all the tasks duplicated then go back to Step 3.* (4.6)

Output : full-duplex task clusters with the optimal schedule length

(그림 1) 태스크 중복 할당에 의한 결함 허용 태스크 스케줄링 알고리즘

마지막 4단계에서는 중복 할당되지 않은 태스크를 찾아내어 가능한 새로운 프로세서를 이용하지 않고 중복 할당하고 모든 태스크가 중복될 때까지 3단계로 돌아가 반복 수행한다. 이를 위해서 (4.1)에서와 같이 태스크의 lct 오름차순으로 정렬된 큐Qlct에서 아직 중복 할당되지 않은 태스크를 선택한다. lct가 적은 태스크부터 중복 할당 여부를 확인하고 중복되어 있지 않으면 (4.2)에서 우선 이미 사용 중인 프로세서의 여유 시간 블록에 할당 가능한가를 살핀다. 이 때 모두 세 가지의 조건을 만족시켜야 한다. 조건 1은 발견된 여유 시간 블록이 크기가 충분히 커서 정해진 종료시간을 만족시켜야 한다는 것이다. 그 다음 조건 2는 pred 조건으로 해당 태스크 $t_{x,j}$의 모든 pred들이 현재 조사하고 있는 프로세서 $p_y$에 이미 할당되어 있거나 다른 프로세서에 할당되어 있는 경우는 통신비용을 감안해서 lst이전까지 모든 pred 결과를 받을 수 있어야 한다는 것이다. 조건 3은 sat($t_{x,j}$) 의 중복 할당을 미리 고려한 것이다. 이것에 대해서는 태스크 그래프 예를 들어서 나중에 설명하기로 한다. 이와 같은 조건을 만족하는 하나의 프로세서가 발견되면 (4.3)에서 중복 할당하고 pat와 sat관계를 다시 설정한다. 조건을 충족하는 프로세서가 두 개 이상 발견되면 (4.4)의 조건 4와 조건 5에 따라서 가장 적합한 것을 선택한다 우선 pat가 프로세서 $p_x$와 일치하는 프로세서를 선택한다. 그래도 두 개 이상의 프로세서가 존재하면 하는 여유 시간 블록 크기가 가장 잘 맞는 경우를 선택한다. 앞서 설명한 세 조건을 충족하는 프로세서가 없으면 (4.5)에서 새로운 프로세서에 pat($t_{x,j}$)와 $t_{x,j}$를 할당한다 $t_{x,j}$의 중복 할당이 완료되면 (4.6)에 나타낸 바와 같이 3단계로 돌아가서 필요한 계산을 다시 하고 다음 중복되지 않은 태스크의 중복 할당을 진행한다. 이를 반복하면 프로세서를 추가로 사용하여 최소 스케줄 길이를 유지하면서 모든 태스크가 중복 할당되는 스케줄 결과가 산출된다.

일반적인 태스크 그래프의 예를 (그림 2)에 나타내었다. 노드를 나타내는 작은 원 안에 태스크 번호가 있고 바로 옆에 태스크 연산시간이 표기되어 있다. 화살표 위에는 해당 태스크들간의 통신비용이 표기되어 있다. $t_1$은 입력 태스크이고 $t_{13}$은 출력 태스크이다. 이 태스크 그래프에 대해서 알고리즘의 1단계를 시행한 결과는 (그림 3)과 같다.



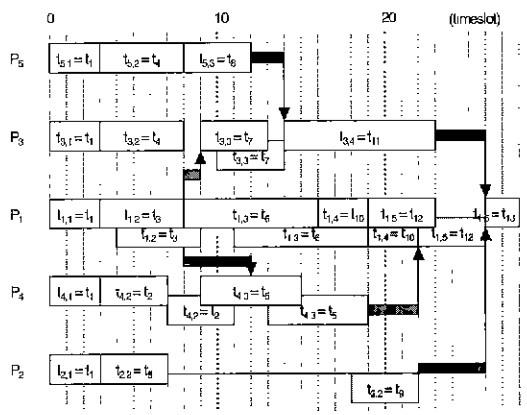
(그림 2) 일반적인 태스크 그래프의 예

| task | <est,ect> | [lst,lct] | fpred | level |
|---|---|---|---|---|
| $t_1$ : | < 0, 3> | [ 0, 3] | 0 | 25 |
| $t_2$ : | < 3, 7> | [ 7,11] | 1 | 16 |
| $t_3$ : | < 3, 8> | [ 4, 9] | 1 | 22 |
| $t_4$ : | < 3, 8> | [ 3, 8] | 1 | 20 |
| $t_5$ : | < 9,15> | [13,19] | 3 | 12 |
| $t_6$ : | < 8,16> | [11,19] | 3 | 17 |
| $t_7$ : | < 9,13> | [10,14] | 4 | 15 |
| $t_8$ : | < 8,12> | [ 8,12] | 4 | 15 |
| $t_9$ | < 3, 7> | [18,22] | 1 | 6 |
| $t_{10}$ : | <16,19> | [19,22] | 6 | 9 |
| $t_{11}$ : | <14,23> | [14,23] | 7 | 11 |
| $t_{12}$ : | <19,23> | [22,26] | 10 | 6 |
| $t_{13}$ : | <26,28> | [26,28] | 12 | 2 |

$Q_{level}$ : $t_{13}$, $t_{12}$, $t_9$, $t_{10}$, $t_{11}$, $t_5$, $t_8$, $t_7$, $t_2$, $t_6$, $t_4$, $t_3$, $t_1$
$Q_{lct}$ : $t_1$, $t_4$, $t_3$, $t_2$, $t_8$, $t_7$, $t_6$, $t_5$, $t_{10}$, $t_9$, $t_{11}$, $t_{12}$, $t_{13}$

(그림 3) 알고리즘의 1단계 수행 결과

(그림 4)는 알고리즘의 2단계에 따라서 태스크를 할당한 결과이다. 우선 프로세서 $p_1$에 $t_1$을 할당하고 이어서 fpred($t_1$)인 $t_{12}$, 그리고 순차적으로 $t_{10}$, $t_6$, $t_3$, $t_4$이 할당된다. 다음 프로세서 $p_2$에는 $Q_{level}$에서 아직 할당되지 않은 $t_9$이 선택되고 fpred($t_9$)인 $t_4$이 할당된다. 마찬가지 방법으로 $p_3$에는 $t_{11}$, $t_7$, $t_4$, $t_1$이 할당된다. 그 다음은 $t_5$의 순서이다. fpred($t_5$)는 $t_3$이지만 이미 $t_3$은 $p_1$에 할당되어 있고 lct($t_3$) + c($t_3$,$t_5$) ≤ lst($t_5$) 이므로 아직 할당되지 않은 pred($t_5$)인 $t_2$를 같은 프로세서에 할당한다.




(그림 4) 알고리즘의 2단계 수행 결과

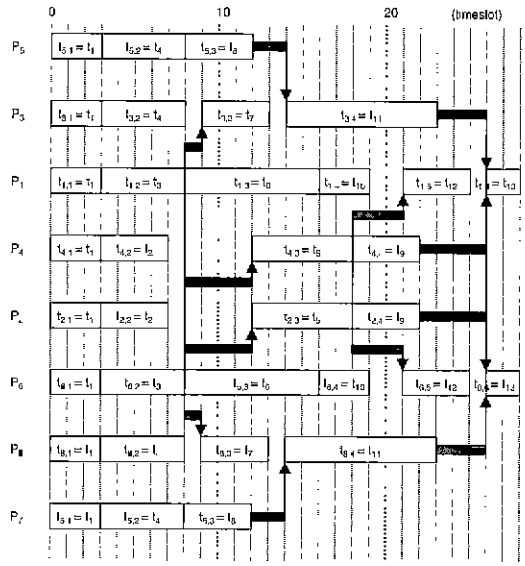
그림에서 네모 상자는 각 태스크의 est과 ect 수행 시간을, 그림자 처럼 진하게 표시된 네모 상자는 태스크의 lst와 lct에 맞추어 나타낸 것이다. 굵은 화살표는 서로 다른 프로세서에 할당된 태스크들간의 통신을 의미한다. 진하게 표시된 네모 상자가 나타나 있지 않은 태스크는 est와 lst가 같은 경우이다. (그림 3)에 보이듯이 원래 $t_5$의 est와 ect는 각각 9, 15이지만 $p_3$에 할당되고 난 후에는 가장 빠른 시작시간과 종료시간이 12, 18로 각각 바뀌게 된다. 즉, esta($t_{1,4}$) = 12, ecta($t_{1,3}$) = 18이 된다 알고리즘의 3단계에서 이와 같이 각 프로세서 별로 esta, ecta, lcta, lsta를 계산하고 여유 시간 블록 [sit(x,k), eit(x,k)]를 계산한다. (그림 5)는 각 프로세서 별 여유 시간 블록을 나타낸다.

$p_1$ : [ 3, 4] [ 8,11] [16,19] [19,22] [23,26]
$p_2$ : [ 3,18] [ 7,28]
$p_3$ : [ 8,10] [13,14] [23,28]
$p_4$ : [ 3, 9] [ 7,13] [18,28]
$p_5$ : [12,28]

(그림 5) 프로세서의 여유 시간 블록

이제 알고리즘의 마지막 4단계를 시작할 준비가 되었다 소요되는 프로세서의 개수를 최대한 줄이면서 문제를 해결하는 접근 방법으로 두 가지를 생각할 수 있다. 하나는 우선 lct가 적은 태스크부터 가능한 기존의 프로세서에 삽입해 나가다가 그것이 불가능해지면 새로 프로세서를 추가하는 방법이다. 또 다른 접근 방법은 우선 순수 여유 시간 블록의 합이 최소인 프로세서의 태스크이거나 할당된 태스크들의 est과 lst이 모두 같은 프로세서의 태스크들을 새로운 프로세서에 그대로 복제한 후에 아직까지도 중복되지 않은 태스크들을 사용 중인 프로세서에 삽입시키는 방법이다 본 논문의 알고리즘은 두 가지 접근방법을 절충하여 첫번째 방법으로 시작하되 중간에 두 번째 접근방법에 해당하는 태스크들이 발견되면 모든 pal 태스크들이 하나의 새로운 프로세서에 할당되도록 한다. (그림 6)에 알고리즘 수행 결과를 도시하였다. 큐 $Q_{lct}$에서 중복 할당되지 않은 첫번째 태스크는 $t_3$이다. $p_2$에는 3부터 18까지의 여유 시간 블록이 있으므로 (그림 1)의 알고리즘 조건 1을 만족하고 $p_2$에는 pred($t_2$)인 $t_1$이 이미 할당되어 있으므로 조건 2도 만족한다. 만일 $t_3$를 $p_2$에 중복 할당하여 $t_{2,2} = t_3$, $t_{2,3} = t_9$을 가정하면 netidletime ($t_{2,2}$)는 식 (23)에 의해 28-8-4 = 16이 된다. 즉, 일고리즘에 나타낸 조건 3의 좌측 항은 16이다 한편 sat($t_{1,2}$)인 $t_6$,

$t_{10}$, $t_{12}$, $t_{13}$ 모두가 아직 중복 할당되지 않았으므로 조건 3의 우측 항은 8+3+4+2=17이 된다. 따라서 조건 3을 만족시키지 못한다. 이 조건의 의미는 앞서 설명한 두 번째 접근방법을 반영하기 위한 것이다. 이 조건을 무시하고 $t_3$를 $p_2$에 중복 할당하고 나면 그 이후에 $t_6$, $t_{10}$을 모두 $p_2$에 중복 할당할 수 없어서 결국 새로운 프로세서에 $t_1$, $t_3$, $t_6$, $t_{10}$, $t_{12}$, $t_{13}$을 중복 할당하게 된다. (그림 6)에 나타낸 본 논문에서 제시하는 알고리즘의 최종 결과와 비교할 때, 조건 3을 무시한 경우에는 $t_2$를 중복 할당하기 위하여 한 개의 프로세서를 더 필요로 하게 된다.

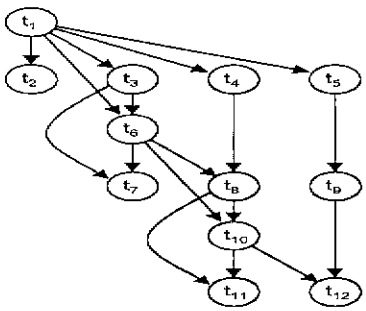



(그림 6) 알고리즘 수행 최종 결과

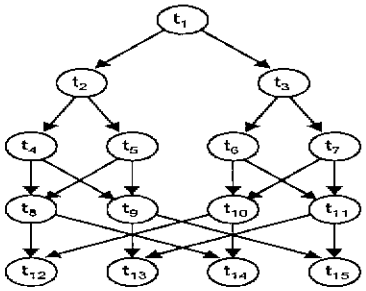
## 4. 실험 및 고찰

이 장에서는 지금까지 설명된 알고리즘의 구현과 다양한 태스크 그래프에 대해 시뮬레이션한 결과에 대하여 논한다.

제시된 알고리즘의 성능을 분석하기 위하여 앞서 설명된 예제 태스크 그래프 외에 최대 100개 태스크로 구성되는 임의(general) 태스크 그래프를 생성하였으며, (그림 7)에 나타낸 것과 같은 실제 태스크 그래프의 모형을 이용하였다. 임의 수를 발생시켜서 태스크 연산시간을 정하고 2장에서 정의된 조건에 부합하는 임의 통신비용으로 각각 약 5,000회에 걸쳐서 실험을 행하였다 (그림 8)에 나타낸 결과는 결합허용 태스크 스케줄에 요구되는 프로세서 증가 비율의 평균 값을 작업부하와 수행시간 대비 통신비용 평면에 샘플링하여 나타낸 것이다.



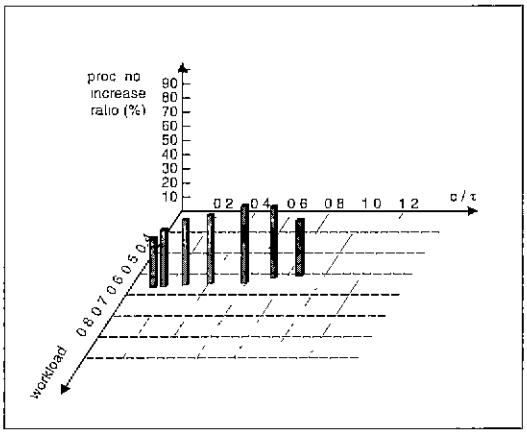

(a) Gauss Elimination Task Graph



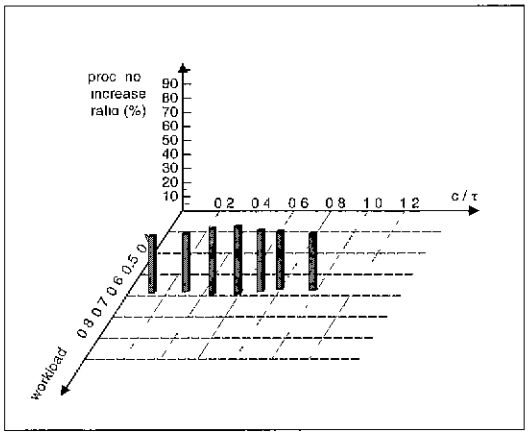

(b) Fast Fourier Transform Task Graph

(그림 7) 시뮬레이션을 위한 태스크 그래프 모형

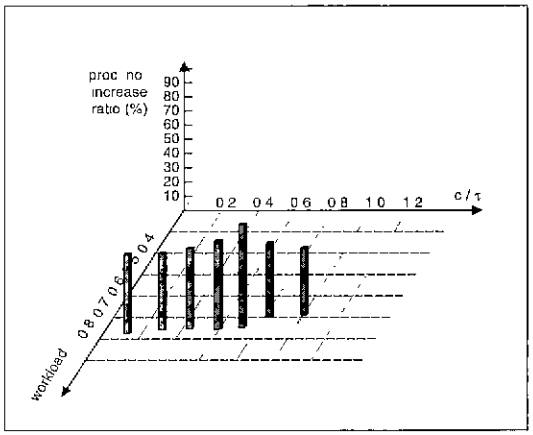
(그림 8)의 가로 축 ($c/\tau$)은 주어진 태스크 그래프에서 얻어진 전체 통신비용을 전체 태스크 연산시간으로 나눈 값, 즉 $\Sigma\, c(t_i, t_j) / \Sigma\, \tau(t_k)$ 을 나타낸다. 비스듬한 세로 축 (workload)은 최소 스케줄 길이를 얻기 위해서 부분적으로 태스크 중복을 한 결과, 즉, (그림 1)의 알고리즘 2단계에서 얻어진 스케줄에서 각 프로세서의 평균 작업부하를 나타낸 것이다. 이 두 가지 좌표 값이 변함에 따라서 알고리즘의 4단계에서 모든 태스크를 중복 할당하는데 추가로 요구되는 프로세서 개수의 증가 비율이 달라지는 것을 알 수 있다. 프로세서 개수 증가 비율은 100 * (알고리즘 단계 4에서 필요한 프로세서 개수 - 알고리즘 단계 2에서 필요한 프로세서 개수) / (알고리즘 단계 2에서 필요한 프로세서 개수)이다. 3장에서 설명한 예의 경우는 프로세서 개수가 5개에서 8개로 증가되었으므로 60%가 증가한 것이다.




(a) General Task Graph




(b) Gauss Elimination Task Graph




(c) Fast Fourier Transform Task Graph

(그림 8) 태스크 중복 할당에 필요한 프로세서 개수 증가 비율

시뮬레이션 결과 최소 프로세서 증가율은 0%이고 최대 프로세서 증가율은 100%로 나타났다. 즉, 2단계에서 사용된 프로세서 만으로 모든 태스크의 중복 할당이 가능한 경우도 있고, 최악의 경우 2단계에서 사용된 프로세서의 개수 만큼이 추가로 필요한 경우도 나타났다. 그러나, 평균적으로 볼 때, 각 태스크 그래프 모형에 따라서 절대 수치는 서로 다르지만 다음과 같은 현상을 공통적으로 갖고 있음을 알 수 있었다.

① 전체 통신비용이 전체 연산시간의 약 60 - 70% 정도일 때 2단계 스케줄 결과의 각 프로세서 평균 작업부하가 비교적 높게 나타났으며, 이 때 모든 태스크들을 중복 할당하기 위해서 추가로 요구되는 프로세서의 증가 비율이 가장 높아서 최대 75%에 이른다.

② 전체 통신비용이 전체 연산시간의 20 - 30% 이하로 들어들면 pred 태스크의 결과를 전달 받기 위해 기다리는 평균 시간이 줄어서 프로세서의 작업부하가 다소 증가한다. 2단계 스케줄 결과의 각 프로세서 평균 작업부하가 높아지게 되면 프로세서 내의 여유 시간 블록이 감소하고 그 결과 모든 태스크들을 중복 할당하기 위해서 추가로 요구되는 프로세서의 증가 비율이 다시 증가되는 현상이 나타난다.

③ 전체 통신비용이 전체 연산시간의 90% 이상이 되면 각 프로세서에는 선행 태스크의 결과를 전달 받기 위해 기다리는 평균 시간이 증가하고 그 결과 프로세서의 작업부하가 감소한다. 작업부하가 낮으면 2단계에서 사용된 프로세서 내에서 여유 시간 블록을 많이 발견하게 되므로 매우 적은 수의 프로세서를 추가해도 모든 태스크가 중복 할당되도록 할 수 있다

결론적으로, 연산시간 대비 통신비용은 최소의 스케줄링 길이를 갖는 스케줄 결과의 각 프로세서의 작업부하에 영향을 미치는데, 작업부하가 상대적으로 적은 경우에는 적은 수의 프로세서를 추가로 투입하여 최소의 스케줄링 길이를 유지하면서 모든 태스크들이 두개 이상의 서로 다른 프로세서에 할당되도록 태스크 스케줄이 가능하다.

## 5. 결 론

태스크 중복을 기반으로 하는 알고리즘을 이용하여 모든 태스크들이 서로 다른 두개 이상의 프로세서에

중복 할당되도록 하여 결함허용 기법을 수용하도록 하면서 최소수행시간을 갖는 알고리즘을 제시하였다. 최소의 태스크 수행 길이를 보장하면서 태스크 중복 할당을 가급적 줄인 스케줄링 알고리즘과 비교해 볼 때, 본 논문에서 제시된 알고리즘은 역시 같은 최소의 수행 길이를 유지하면서 모든 태스크가 중복 할당되고 프로세서 추가 사용이 억제된 스케줄 결과를 산출된다.

시뮬레이션 결과, 연산시간 대비 통신비용은 최소의 스케줄링 길이를 갖는 스케줄 결과의 각 프로세서의 작업부하에 영향을 미치는데, 작업부하가 상대적으로 적은 경우에는 약 30 % 증가된 개수의 프로세서로, 작업부하가 상대적으로 큰 경우에는 약 75% 증가된 개수의 프로세서로 최소의 스케줄링 길이를 가지면서 동시에 모든 태스크들이 두 개 이상의 서로 다른 프로세서에 할당되도록 태스크가 스케줄 되는 것을 확인할 수 있었다.

## 참 고 문 헌


[1] T. Adam, et. al. "A Comparison of List Schedules for Parallel Processing Systems," Comm. ACM. Vol.17, No.12, pp.685-690, Dec., 1974.

[2] I. Ahmad and Y.K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," IEEE Trans on Parallel and Distributed Systems, Vol 9, No 9, pp 872-891, Sept. 1998.

[3] I. Ahmad and Y.K Kwok, "On Parallelizing the Multiprocessor Scheduling Problem," IEEE Trans. on Parallel and Distributed Systems, Vol 10, No 4, pp.414-432, Apr. 1999.

[4] A. Bertossi, et al, "Fault-tolerant Rate-monotonic First-fit Scheduling in Hard-real-time Systems," IEEE Trans. on Parallel and Distributed Systems, Vol.10, No 9, pp.934-945, Sept. 1999.

[5] H. Chen, et al., "Static Scheduling Using Linear Clustering Task Duplication," Proc Int'l Conf Parallel and Distributed Computing and Systems, pp.285-290, Oct 1993

[6] S Darbha and P. Agrawal, "A Task Duplication based Scalable Scheduling Algorithm for Distributed Memory Systems," J. of Parallel and Distributed Computing, Vol.46, pp.15-27, 1997.

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

[8] R. Graham, et. al, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," Annals of Discrete Mathematics, pp.287-326, 1979.

[9] C Hou and K.G. Shin, "Module Allocation with Timing and Precedence Constraints in Distributed Real-time Systems" IEEE Proc Real Time Systems Symp. pp.146-155, Dec 1994.

[10] K.H. Kim, et. al., "Fault-tolerant Execution of Real-time Tasks through Duplex Assignment within Parallel Computers," Proc. Int'l Conf. Parallel and Distributed System, Dec 1992.

[11] K.H. Kim and B.J. Min, "Approaches to Implementation of Multiple DRB Stations in Tightly Coupled Computer Networks," Proc Int'l Computer Software and Applications Conf, Sept 1991

[12] P. Maheshwari and H Shen, "An Efficient Clustering Algorithm for Partitioning Parallel Programs," Parallel Computing Vol.24, pp.893-909, 1998.

[13] G Manimaran and C. Murthy, "A Fault-tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems and its Analysis," IEEE Trans on Parallel and Distributed Systems, Vol.9, No 11, pp.1137-1152, Nov. 1998.

[14] T Varvarigou and J. Trotter, "Module Replication for Fault-tolerant Real-time Distributed Systems," IEEE Trans. on Reliability, Vol 47, No.1, pp.8-18, Mar 1998



**민 병 준**

e-mail : bjmin@lion.inchon.ac.kr

1983년 연세대학교 전자공학과 학사

1985년 연세대학교 전자공학과 석사

1991년 미국 캘리포니아주립 대학교 (UC Irvine) 전기및컴퓨터 공학과 박사

1984년~1986년 삼성전자 연구원

1992년~1994년 한국통신 선임연구원

1995년~현재 인천대학교 컴퓨터공학과 조교수

관심분야 : 병렬처리, 분산시스템, 통신망관리