중복 단계를 개선한 병렬 타스크 할당 알고리즘

°이재관 김창수
부경대학교 전자계산학과
leejaegn@baesan.psks.ac.kr cskim@dolphin.pknu.ac.kr

A Parallel Task Allocation Algorithm improved Duplication Steps

°Jae-Gwan Lee Chang-Soo Kim
Department of Computer Science
Pukyong National University
leejaegn@baesan.psks.ac.kr cskim@dolphin.pknu.ac.kr

요 약

병렬 프로그램의 스케쥴링 기법에 있어, 타스크 중복 알고리즘은 리스트 스케쥴링 알고리즘에 비해 상대적으로 새로운 접근 방식이다. 타스크 중복이란 어떤 프로세서에 할당되어 있는 중요한(critical) 타스크들을 다른 프로세서에 중복시켜, 그 타스크들이 중복 실행하도록 프로그램을 스케쥴하는 것이다. 따라서, 중요한 타스크들이 같은 프로세서내에 있게 되어, 다른 타스크들의 시작 시간(start time)을 줄일 수 있게 된다. 이는 결국 전체 프로그램의 스케쥴 길이(schedule length)를 줄이게 된다. 병렬 프로그램의 스케쥴링 목적은 프로그램의 스케쥴 길이를 최소화하고, 스케쥴의 complexity를 줄이는 것이다. 그러나, 스케쥴 길이와 complexity는 상호 trade-off 관계이다. 본 논문에서는 기존의 중복 알고리즘과 비교하여, 스케쥴 길이를 증가 시키지 않으면서, complexity를 같거나 더 적게하는 알고리즘을 제시하여 컴파일 시간을 향상시키고자 한다.

1. 서론

병렬 컴퓨터 시스템이 가지고 있는 성능을 최대한 활용하기 위하여, 병렬 프로그램을 프로세서들에게 효율적으로 스케쥴하는 것은 매우 중요한 일이다. 스케쥴링은 컴파일 시간에 정적으로(statically) 이루어진다. 스케쥴링의 목적은 스케쥴 길이를 최소화하고, 스케쥴의 complexity를 줄이는 것이다. 그러나, 스케쥴 길이와 complexity는 상호 trade-off 관계이다. 타스크 그래프에 대한 멀티프로세서 스케쥴링 문제는 NP-complete로 잘 알려져 있다[3]. 따라서, 제안된 많은 알고리즘들은 발견적 방법(heuristics)에 의한 것들이다[7].

정적인 타스크 스케쥴링 알고리즘은 타스크 중복을 사용하는 스케쥴링(duplication scheduling)과 중복을 사용하지 않는 스케쥴링(non-duplication scheduling)으로 분류할 수 있다[2]. 중복을 사용하지 않는 스케쥴링은 리스트 스케쥴링 알고리즘에 기인한 것들이다[6].

타스크 중복 알고리즘은 리스트 스케쥴링 알고리즘에 비해 상대적으로 새로운 접근 방식이다. 타스크 중복 알고리즘은 리모트 프로세서(remote processor)에 할당되어 있는 부모 타스크들(parent tasks)을 로컬(local) 프로세서에 중복시켜서, 통신으로 인한 오버헤드를 줄일 수 있다. 중복 스케쥴링은 부분적인 중복 스케쥴링(Scheduling with Partial Duplication)과 완전한 중복 스케쥴링(Scheduling with Full Duplication)으로 구분할 수 있다.

SPD 알고리즘은 join node의 부모 노드를 critical 한 경우에만 중복시킨다. 그리고, join node는 critical parent node가 있는 프로세서에 스케쥴된다. 이 알고리즘은 제한적 중복으로 인하여 복잡도가 낮은 반면, 통신 오버헤드가 큰 시스템에는 적합하지 않다. FSS(Fast and Scalable Scheduling) 알고리즘[4]이 여기에 속한다.

SFD 알고리즘은 join node의 부모 노드들 뿐만 아

니라, 모든 조상 노드(ancestor node)들에 대해서도 중복 알고리즘을 적용한다. 그러므로, 이 부류의 알고리즘은 복잡도가 높은 반면, SPD보다 나은 성능을 보여준다. SFD 알고리즘은 통신 오버헤드가 큰 시스템에 보다 효과적이며, CPFD(Critical Path Fast Duplication)[2], DFRN(Duplication First and Reduction Next)[1] 등의 알고리즘이 여기에 속한다.

본 연구에서 제안하는 알고리즘, TADS(Task Allocation improved Duplication Steps)는 SFD 알고리즘에 속하며, join node의 부모 노드들과 그 노드들이 포함되어 있는 프로세서내의 노드들에 대해서만 중복 알고리즘을 적용한다.

2. 용어 정의

병렬 프로그램은 보통 타스크 그래프(task graph)로 표현된다. 본 논문에서는 타스크 그래프를 4개의 요소(n, e, w, c)로 나타내는데[5], n은 노드들(nodes)을, e는 통신 경로(communication edges)를, w는 연산 비용(computation costs)을, c는 통신 비용(communication costs)을 각각 의미한다. 각 노드 번호는 $n_1, n_2, ..., n_i$로 표기하고, $w(n_i)$는 노드 n_i의 연산 시간을 나타낸다. n_i에서 n_j사이의 통신 시간은 c_{ij}로 표기한다.

두 노드 사이에 선행 관계(precedence constraint)가 존재한다면, 즉, n_j는 반드시 n_i가 실행되고 난 후에 실행해야 한다면, n_i를 부모 노드(parent node), n_j를 자식 노드(child node)라 한다. 부모 노드가 없는 노드를 entry node라 하고, 자식 노드가 없는 노드를 exit 노드라 한다. 노드 n_i가 프로세서 J에 스케쥴될 때, 프로세서 J에서 n_i가 시작할 수 있는 시간(start time)을 $ST(n_i, J)$로 표기한다.

본 논문에서 사용되는 몇 가지의 용어를 정의하면 다음과 같다.

정의 1: 한 노드의 outgoing degree가 2이상인 노드를 fork node라 한다.

정의 2: 한 노드의 incoming degree가 2이상인 노드를 join node라 한다.

그림 2에서 n_1은 fork node이고, n_7, n_8은 join node이다. 어떤 노드는 join node이면서 fork node도 될 수 있다.

정의 3: 통신 제약(communication constraint)을 고려하여, 노드 n_i가 프로세서 PE에서 가장 빠르게 실행을 시작할 수 있는 시간을 $EST(n_i, PE)$로 표기하고, 가장 빠르게 실행을 완료할 수 있는 시간을 $ECT(n_i, PE)$로 표기한다.

정의 4: n_i에서 n_j까지 메시지가 도달하는데 걸리는 시간을 $MAT(n_i, n_j)$라 한다.

두 노드 n_i, n_j가 같은 프로세서 J에 스케쥴되면 $MAT(n_i, n_j) = ECT(n_i, J)$이 되고, 서로 다른 프로세서에 스케쥴되면, $MAT(n_i, n_j) = ECT(n_i, J) + c_{ij}$이 된다.

정의 5: join node n_i는 m개의 부모 노드를 가지며, n_{p_k}는 k번째 부모 노드를 나타낸다고 가정하면, 다음의 조건을 만족하는 부모 노드를 critical parent node(n_{p_c})라 한다.

$$\max_{1 \le k \le m} \{MAT(n_{p_k}, n_i)\}$$

n_{p_c}가 스케쥴된 프로세서는 PE_c로 표기한다.

정의 6: 프로세서 J에 노드 n_m, n_n 순으로 스케쥴되어 있을 때, n_m과 n_n사이에 다음의 조건을 만족하는 idle time slot이 있다면, 노드 n_i는 그 사이에 스케쥴될 수 있다.

$$EST(n_n, J) - \max\{ECT(n_m, J), EST(n_i, J)\} \ge w(n_i)$$

3. 관련 연구

여기서는 SFD 알고리즘에 속하는 CPFD와 DFRN 알고리즘에 대하여 간단히 언급하고, 나중에 TADS 알고리즘과 이들과의 성능을 비교할 것이다.

3.1 Critical Path Fast Duplication(CPFD) 알고리즘

CPFD 알고리즘은 타스크 그래프의 노드들을 3가지 종류로 분류한다[2]. (1)Critical Path Node(CPN), (2)In-Branch Node(IBN), (3)Out-Branch Node(OBN). CPN 노드는 critical path에 있는 노드들이고, IBN은 CPN으로 갈 수 있는 경로(path)에 있는 노드들이다. 그리고, OBN은 CPN과 IBN이 아닌 노드들을 말한다.

CPFD 알고리즘은 CPN 노드들을 우선 스케쥴한다. 하나의 CPN 노드를 스케쥴할 때, 그 노드와 관련된 IBN 노드가 스케쥴되어 있지 않다면, 그와 관련된 모든 IBN 노드들을 찾아서 스케쥴한다. CPN과 IBN이 모두 스케쥴되고 나면, OBN 노드들을 우선 순위에 따라 스케쥴한다.

3.2 Duplication First and Reduction Next(DFRN) 알고리즘

DFRN 알고리즘은 fork node와 join node에서 중복 알고리즘을 다르게 적용한다[1]. fork node의 중복에는 SPD 알고리즘이 적용되고, join node에는 DFRN 알고리즘이 적용된다. DFRN 알고리즘은 SFD 알고리즘과는 달리 중복의 효과를 평가하지 않고, join node에서부터 bottom-up 방식으로 모든 조상 노드들을 검색하여, join node의 critical parent가 스케쥴되어 있는 프로세서에 우선 중복시킨다. 그리고 난 후, 중복된 각각의 노드들을 점검하여 중복 조건에 맞지 않으면 삭제한다.

4. TADS 알고리즘 및 시스템 가정

전체적인 타스크의 스케쥴 길이를 최소화하기 위해서는 다음의 사항들을 고려하여야 한다. 1) 자식 노드는 부모 노드가 스케쥴되어 있는 프로세서에 할당되어야 한다. 2) 부모 노드가 여러 개인 경우에는 통신 비용이 가장 큰 critical parent node가 있는 프로세서에 자식 노드가 할당되어야 한다. 3) 부모 노드들을 중복시킴으로써 자식 노드들의 시작 시간(start time)이 단축된다면, 부모 노드들을 중복시킨다. 노드의 중복은 fork node와 join node에서 발생한다.

본 논문에서 제안한 알고리즘은 대부분의 기존 연구[1][2][4]에서 가정하고 있는 것과 동일하게 다음 상황을 가정하고 있다.

가정 1: 모든 프로세서들간의 연결은 1 : 1로 완전히 연결된 것(fully-connected network)으로 간주하며, 프로세서의 수는 무한개이다.

가정 2: 각 프로세서들은 동일한 성능을 가지며, 연산중에도 통신을 할 수 있도록 통신을 전담하는 하드웨어가 부착되어 있다.

가정 3: 두 노드가 동일한 프로세서에 스케쥴되면 두 노드간의 통신 비용은 0가 된다.

4.1 Fork node의 중복

하나의 fork node는 여러 개의 자식 노드를 가지고 있으며, fork node 자신은 그들의 부모 노드가 된다. 각각의 자식 노드들은 부모 노드와의 통신 시간을 줄이기 위해 부모 노드가 있는 프로세서에 스케쥴되길 원한다. 따라서, fork node가 프로세서내의 마지막 노드라면, 자식 노드를 그 뒤에 바로 스케쥴하고, fork node가 마지막 노드가 아니라면, 프로세서내의 첫 노드부터 fork node 까지를 새로운 프로세서에 중복시킨 후, 자식 노드를 스케쥴한다. 그림 1은 fork node가 중복되는 간단한 형태를 보여주고 있다. fork node n_1을 프로세서 P1, P2, P3에 중복시킴으로써 자식 노드 n_2, n_3, n_4의 ST가 감소하는 것을 볼 수 있다.

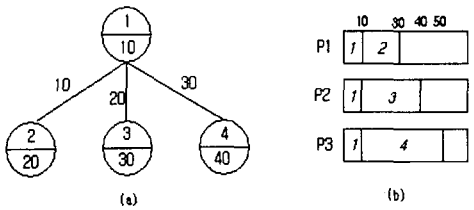

그림 1. (a)fork node; 원의 윗 부분: 노드번호, 아랫 부분: computation cost (b) 스케쥴된 모습

4.2 Join node에서의 중복

하나의 join node는 2개 이상의 부모 노드를 가지며 모든 부모 노드들과의 선행 관계를 유지해야 한다. 만약, n_i의 부모 노드들의 MAT가 $n_{p_1}, n_{p_2}, \ldots, n_{p_k}$의 순서로 크고, n_{p_k}, $n_{p_{k-1}}$이 각각 프로세서 J, P에 스케쥴되어 있다면, n_i를 n_{p_k}과 같은 프로세서 J에 스케쥴함으로써 $ST(n_i, J)$를 줄일 수 있다. 만약, n_{p_k}이 J의 마지막 노드가 아니라면, J의 첫 노드부터 n_{p_k} 까지를 새로운 프로세서 J'에 중복시킨 후, n_i를 스케쥴하면 된다. 이 경우, n_i의 ST는 다른 프로세서 P에 할당되어 있는 $n_{p_{k-1}}$의 $MAT(n_{p_{k-1}}, n_i)$에 달려있다. 따라서, 만약, n_i를 스케쥴하기 전에 $n_{p_{k-1}}$을 J(혹은 J')에 중복시킴으로써 n_i의 ST가 더 감소하게 된다면, 노드 $n_{p_{k-1}}$을 J(혹은 J')에 먼저 중복시킨다. 또, $n_{p_{k-1}}$의 ST는 P에 있는 $n_{p_{k-1}}$의 부모 노드 MAT에 의존하게 되며, 이는 결국 n_i의 ST에 영향을 미치게 된다. 그러므로, $n_{p_{k-1}}$의 부모 노드를 중복시키고, 이런 방법으로 프로세서 P에 있는 노드들을 J(혹은 J')에 계속 중복시킨다.

이 과정에서, J(혹은 J')에 이미 있는 노드들은 중복시키지 않으며, 노드의 중복이 J(혹은 J')에 있는 자식 노드의 ST를 증가시키거나, 중복을 위한 공간(idle time slot)이 없다면, 중복 과정을 멈추고 n_i를 J(혹은 J')에 스케쥴한다.

프로세서 P에 있는 노드들이 멈춤없이 중복을 완료하였다면, n_i의 ST는 또 다른 프로세서에 할당되어 있는 부모 노드인 $n_{p_{k-2}}$의 $MAT(n_{p_{k-2}}, n_i)$에 영향을 받

게 된다. 다시 $n_{p_{i,i}}$가 J(혹은 J')에 중복이 가능하다면 위의 중복 과정을 계속하게 된다.

4.3 알고리즘 설명

본 알고리즘에서는 스케쥴을 단순화하기 위하여, 노드들의 우선 순위(priority)를 레벨(level) 순으로 하고, 레벨이 같은 경우에는 좌에서 우로 우선 순서를 정하였다. 또한, 노드들의 ST 연산을 줄이기 위하여, 자식 노드는 반드시 부모 노드가 마지막 노드로 할당되어 있는 프로세서에 스케쥴하도록 하였다. 부모 노드가 마지막 노드가 아니면, 그 프로세서내의 첫 노드부터 부모 노드 까지를 새로운 프로세서에 중복시킨다.

```
(1)  initialize()  // level순으로 node들을
                      queue에 생성
(2)  for each node n_i in the queue
(3)     find n_p. and PE_c
(4)     if n_p. does not exist // if n_i is a entry node
(5)        schedule n_i to PE_u
(6)        continue
(7)     end if
(8)     if n_p. is last node in PE_c
(9)           check_join_node( n_i , PE_c )
(10)    else
(11)       copy the schedule up to the n_p. onto PE_u
(12)       check_join_node( n_i , PE_u )
(13)    end if
(14) end for
```

스케쥴 대상이 join node가 아니면, 부모 노드가 있는 프로세서에 스케쥴한다.

check_join_node(n_i , PE_c)

```
(15) if n_i is not a join node
(16)    schedule n_i to PE_c
(17) else
(18)    attempt_duplication( n_i , PE_c )
(19) end if
```

부모 노드들의 MAT가 큰 순서대로 중복 가능성을 점검한다. 여기서, J는 현재 부모 노드가 스케쥴되어 있는 프로세서이고, PE_c는 이들이 중복될 프로세서이다 (중복이 끝나면, n_i가 스케쥴될 프로세서이기도 하다).

attempt_duplication(n_i , PE_c)

```
(20) push processors on PE_Stack
        // from the processor containing the smallest
           MAT( n_p. , n_i ) to the processor containing
           the largest MAT( n_p_m , n_i ), where n_p_m ≠ n_p.
           and n_p. , ... , n_p_m are parent nodes of n_i in
           processors.
(21) while PE_Stack is not empty
(22)    J ← pop PE_Stack
(23)    apply_duplication( n_i , PE_c , J )
(24)    if duplication is stopped
(25)       break
(26)    end if
(27) end while
(28) schedule n_i onto PE_c with EST( n_i , PE_c )
```

J에 있는 모든 노드들에 대하여, n_i의 부모 노드부터 그 조상 노드들까지 bottom-up 방식으로 중복을 시켜나간다. 중복은 임시로 DNL에 해 두었다가, 중복이 완료되면, PE_c에 순차적으로 스케쥴한다. 이는 중복된 노드들의 EST 연산을 쉽게 하기 위한 방법이다.

apply_duplication(n_i , PE_c , J)

```
(29) DNL ← NULL
        // DNL is the Duplication Node List
(30) for each node n_j from n_j, to n_j,
        // n_j, is the last node in the processor J and
           n_j, is the first node in the processor J
(31)    if n_j exists on PE_c
(32)       continue
(33)    end if
(34)    determine EST( n_j , PE_c )
(35)    if ST( n_i, PE_c) does not increase and
           EST( n_j, PE_c ) ≤ ECT( n_j,,J) + c_i,j
(36)       insert n_j into DNL
(37)    else
(38)       break loop
(39)    end if
(40) end for
(41) duplicate nodes on PE_c according to DNL
```

n개의 노드로 이루어진 타스크 그래프에 대한 본 알고리즘의 complexity를 살펴보면, step(1)은 O(1)이고, step(3)은 critical parent node와 해당 프로세서를 찾는데 O(n)이 걸린다. 그러므로, step(3)에서 step(13)까지, step(15)에서 step(19)까지의 루틴(routine)은 O(n)가 된다.

Join node의 부모 노드 수가 m개라면, step(20)은 이들에 대한 분류(sort) 작업이 필요하므로 O(m^2)이 된다. 또, apply_duplication(n_i, PE_c,J) 루틴은 프로세서내의 노드 수가 p개라면 O(p)가 된다. 그러므로, step(21)에서 step(27)까지는 O(mp)가 되어, attempt_duplication (n_i, PE_c) 루틴은 max{O(m^2), O(mp)}이 된다. 그리고, $m \leq n$, $p \leq n$이므로 결국 attempt_duplication(n_i , PE_c) 루틴은 O(n^2)이 된다. join node 수가 q개이면, attempt_duplication(n_i , PE_c) 루틴은 q번 호출되므로 알고리즘 전체의 complexity는 O(n^3)이 된다. 여기서, $q \leq n$이다.

4.4 알고리즘의 적용 및 비교

이 절에서는 임의의 타스크 그래프에 대해 TADS

알고리즘을 적용하여 스케줄된 결과를 살펴 보고, CPFD와 DFRN의 결과와 비교 검토해 본다.

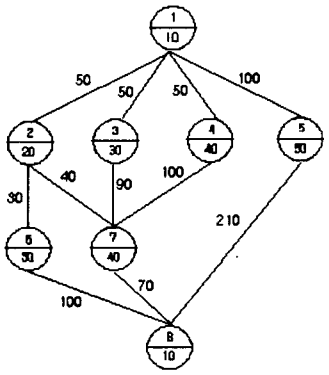

그림 2. Sample Task Graph

P1: [0,1,10][10,5,60]
P2: [0,1,10][10,4,50]
P3: [0,1,10][10,3,40][40,4,80][80,7,120][120,5,170][170,8,180]
P4: [0,1,10][10,2,30][30,6,60]

(a) schedule by CPFD (SL = 180)

P1: [0,1,10][10,5,60][60,4,100][100,3,130][130,7,170][170,8,180]
P2: [0,1,10][10,4,50][50,3,80][80,7,120]
P3: [0,1,10][10,3,40]
P4: [0,1,10][10,2,30][30,6,60]

(b) schedule by DFRN (SL = 180)

P1: [0,1,10][10,2,30][30,6,60]
P2: [0,1,10][10,3,40]
P3: [0,1,10][10,4,50][50,3,80][80,7,120]
P4: [0,1,10][10,5,60][60,4,100][100,3,130][130,7,170][170,8,180]

(c) schedule by TADS (SL = 180)

그림 3. schedule by various algorithms

그림 2는 임의의 타스크 그래프이고, 그림 3은 각각의 알고리즘에 대한 스케줄 결과를 보여 주고 있다. P_k는 프로세서 k를 나타내며, SL은 스케줄 길이를 의미한다. 그리고, [ ]속의 숫자는 [$EST(n_i, P_k)$, i, $ECT(n_i, P_k)$]을 표현한 것이다.

그림 3의 스케줄 결과를 살펴보면 모두 SL = 180으로 동일하게 나타난다. CPFD, DFRN, TADS의 complexity는 각각 $O(n^4)$, $O(n^3)$, $O(n^3)$이다. 따라서, 외견상 동일한 것으로 보이는 DFRN과 TADS를 비교함으로써 TADS의 우수성을 증명하고자한다. DFRN 알고리즘의 스케줄 과정을 살펴보면 다음과 같은 문제점이 발견된다.

■ DFRN 알고리즘의 단점

Join node n_i를 스케줄하기 위하여 n_i의 모든 부모 노드들(n_p는 제외)과 그의 조상 노드들을 bottom up 방식으로 찾아, 중복의 효과는 고려하지 않고, 그 노드들이 PE_c에 존재하지 않으면 무조건 중복시킨다. 중복이 끝나면, 중복의 조건(알고리즘 step(35))에 합당하지 않은 노드들을 찾아 삭제한다. 이 과정에서 불필요한 중복과 삭제 과정이 발생하여 DFRN의 성능을 저하시킨다.

■ DFRN 알고리즘의 개선책

Join node의 부모 노드들은 그 조상 노드들과의 선행관계를 유지하기 위하여, 조상 노드들의 일부는 같은 프로세서내에 할당하고, 일부는 메시지 전송 방식을 택하여 EST를 결정하였다. 그러므로, join node의 EST를 위하여, 프로세서내의 노드들에 대해서만 중복 여부를 결정하면 된다. 만약, 메시지 전송 방식을 택한 노드들을 중복시킨다면 부모 노드의 EST가 증가하게 되고, 결국 join node의 EST도 증가하게 되어 중복의 효과를 얻을 수 없게 된다. 따라서, 메시지 전송 방식을 택한 노드들은 중복을 위해 고려할 필요가 없게 된다. 따라서, 부모 노드들이 스케줄되어 있는 프로세서들을 부모 노드들의 MAT 순서대로 정렬한 후, 프로세서내의 노드들만 중복시키면 된다. 중복 과정에서 중복의 조건(알고리즘 step(35))에 위배되면, 그 노드 이후의 노드들은 중복의 효과를 얻을 수 없으므로 중복을 중단한다. 이와 같은 방법은 불필요한 중복과 중복을 위한 검색, 삭제 시간 등을 줄일 수 있다.

DFRN과 TADS 알고리즘의 complexity는 최악의 경우를 고려하면 $O(n^3)$으로 같을 수 있지만, 보통의 경우, TADS는 불필요한 중복과 삭제의 과정을 없애줌으로 DFRN보다 나은 성능을 보여줄 수 있다. 그리고, join node의 개수가 많을수록, 깊이(depth)가 클수록 TADS는 DFRN에 비해 더 좋은 성능을 발휘한다.

5. 결론

타스크 중복 스케줄링 알고리즘에서, SFD 알고리즘은 SPD 알고리즘 보다 complexity는 높은 반면, 성능은 나은 것으로 알려져 있다. CCR(Communication to Computatioon Ratio)이 큰 시스템의 경우에는 더욱 효과적이다.

본 논문에서는 SFD 알고리즘에 해당하는 TADS

알고리즘을 제시하였다. 이 알고리즘에서, 스케줄 길이는 CPFD나 DFRN과 비슷하게 나타났으나, complexity는 CPFD 보다는 낮고, DFRN과는 같았다. 그러나, DFRN 알고리즘의 단점인 불필요한 중복과 삭제의 과정을 없애줌으로써, 실제의 평균 컴파일 시간은 DFRN 보다 줄어든다. 그리고, join node의 수가 증가할수록, 깊이가 깊어질수록 TADS는 CPFD나 DFRN 보다 더 나은 성능을 발휘할 수 있다.

앞으로, TADS 알고리즘을 구현하여, 실제의 성능을 비교 분석하는 연구가 과제로 남아 있다.

참 고 문 헌

[1]G.L. Park, B. Shirazi, and J. Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems," proceedings 11th International Parallel Processing Symposium, 1997, pp. 157-166

[2]I. Ahmad and Y. K. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," Proc. of Int'l Conf. on Parallel Processing, vol. II, Aug. 1994, pp. 45-51.

[3]M.R. Gray and D.S. Johnson, Computers andIntractability, A Guide to the Theory of NP-Completeness. 1979, W.H. Freeman and Company.

[4]S. Darbha and D.P. Agrawal, "A Fast and Scalable Scheduling Algorithm for Distributed Memory Systems," Proc. of Symp. on Parallel and Distributed Processing, Oct. 1995, pp. 60-63

[5]S. Darbha and D.P. Agrawal, "SDBS: A Task Duplication Based Optimal Scheduling Algorithm," Proc. of Scalable High Performance Computing Conf., May 1994, pp. 756-763

[6]T.L. Adam, K. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing System," Communication of the ACM, vol. 17, no. 12, Dec. 1974, pp. 685-690.

[7]V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessor Scheduling, 1989, MIT Press, Cambridge, MA.