스레드를 이용한 계층적 태스크 그래프(HTG)의 복합 노드 스케줄링 기법
(Scheduling Scheme for Compound Nodes of Hierarchical Task Graph using Thread)

김 현 철 † 김 효 철 ††
(Hyun-Chul Kim) (Hyo-Cheol Kim)

요 약 본 논문은 공유 메모리 시스템에서 계층적 태스크 그래프(Hierarchical Task Graph, HTG)의 복합 노드 태스크들을 효율적으로 수행하기 위한 새로운 태스크 스케줄링 기법을 소개한다. 함수 병렬성 추출을 위해 제안된 기법은 별도의 전역 스케줄러가 필요 없이 프로세서 스스로가 스케줄링 기능을 행하는 자동 스케줄링이다. 제안된 스케줄링 기법을 단일처리기 시스템을 비롯한 여러 플랫폼에 적용하기 위해 자바 스레드를 이용하여 구현하였으며, 기존의 비트 벡터 방법과 성능을 비교 분석하였다. 실험 파라메터 값을 이용한 실험 결과, 제안된 스케줄링 기법은 수행 시간 측면에서 효율적이며 양호한 부하 균형을 유지하였다. 또한, 제안된 기법은 기존의 방법에 비해 메모리 사용량을 줄일 수 있었다.

키워드 : 함수병렬성, 계층적 태스크 그래프, 복합소스, 자동스케줄링, 비트벡터 알고리즘

Abstract In this paper, we present a new task scheduling scheme for the efficient execution of the tasks of compound nodes of hierarchical task graph(HTG) on shared memory system. The proposed scheme for exploitation functional parallelism is autoscheduling that performs the role of scheduling by processor itself without any dedicated global scheduler. To adapt the proposed scheduling scheme for various platforms, including a uni-processor systems, Java threads were used for implementation, and the performance is analyzed in comparison with a conventional bit vector method. The experimental results showed that the proposed method was found to be more efficient in its execution time and exhibited good load-balancing when using the experimental parameter values. Furthermore, the memory size could be reduced when using the proposed algorithm compared with a conventional scheme.

Key words : Functional Parallelism, HTG(Hierarchical Task Graph), Compound Node, Autoscheduling, Bit Vector Algorithm

1. 서 론

병렬 처리는 현재의 컴퓨팅 환경에 많은 공헌을 하고 있으며, 하드웨어의 발전에 따라 병렬 컴퓨터를 위한 소프트웨어와 병렬화 컴파일러 연구에 많은 관심을 두게 되었다. 병렬화 컴파일러를 이용하여 하나의 프로그램을 여러 개의 모듈들로 분할 후, 그들의 선행 관계(predecessor relation)를 고려하여 다중처리기에 할당하므로 수행 시간을 최소화하며, 부하균형을 유지하는 효율적인 스케줄링 기법들이 소개되어지고 있다[1-6]. 이러한 기법들은 스케줄링 단위의 성김도(granularity) 크기에 따라 두 가지로 분류될 수 있다. 그 중 하나는, 큰 성김도(coarse grain)의 루프 또는 자료 병렬성(data parallelism) 추출을 위한 루프 스케줄링 기법이다. 다른 하나는, 성김도가 작은(fine grain) 태스크 또는 함수 병렬성(functional parallelism) 추출을 위한 태스크 스케줄링 기법이다[1-3]. 지금까지 관심을 가진 많은 연구는 병렬 루프에서의 자료 병렬성 추출에 관한 것들이다. 왜냐하면, 대부분의 계산적인 응용이 여기에 속하며 또한, 많은 양의 병렬성을 제공하기 때문이다[7-13].

다양한 새로운 구조의 병렬 시스템이 개발되고 서브루틴간의 종속성 분석에 관해 연구함에 따라, 루프 외에 태

† 비 회 원 : 재능대학 컴퓨터정보계열 교수
hckim@mail.jnc.ac.kr
†† 정 회 원 : 계명문화대학 컴퓨터정보계열 교수
khc@km-c.ac.kr
논문접수 : 2000년 10월 25일
심사완료 : 2002년 6월 5일

스크 레벨에서의 함수 병렬성 추출을 위한 스케쥴링 기법에 관한 연구의 중요성을 인식하게 되었다[1,6,14,15,16]. 프로그램에서 함수 병렬성의 양은 자료 병렬성 보다 훨씬 적지만, 고수준의 파이프라인(highly pipelined), 멀티스레드, 슈퍼스칼라, VLIW 구조에서는 유용하게 사용되어진다. 또한, 함수 병렬성 추출은 서브루틴이나 순차 루프 내에서 독립적인 태스크들의 병렬 수행에도 적용될 수 있다[1,6].

계층적 태스크 그래프(hierarchical task graph, 이하 HTG)는 프로그램의 중간적인 표현으로써, 작업들간의 선행 관계를 표현한 자료 종속성(data dependence)과 제어 종속성(control dependence) 정보를 포함하고 있다. 여러 개의 태스크들로 이루어진 HTG의 복합 노드에서의 함수 병렬성 추출을 위한 태스크 스케쥴링에 관한 연구는 그다지 많지 않으며, 소개되어진 스케쥴링 기법으로는 Moreira가 제안한 비트 벡터(bit vector) 할당 기법이 있다[16]. 이 방법은 동적 태스크 스케쥴링 기법 중의 하나인 자동 스케쥴링(autoscheduling)이다. 일반적으로, 태스크의 생성과 스케쥴링은 운영체제나 런-타임 라이브러리에 의해 이루어지나, 자동 스케쥴링에서는 병렬화 컴파일러에 의해 생성되는 드라이브 코드(drive code)가 프로세서가 수행 할 코드의 앞, 뒤에 삽입됨으로써 스케쥴링이 이루어진다. 이렇게 됨으로써, 프로세서 각각이 추가된 스케쥴링 코드 부분도 수행하므로 별도의 고정된 스케쥴러가 필요하지 않다[14,15,16]. 만족해야 될 종속성이 하나만 있는 태스크들은 종속적 선행자에 의해서 별도의 스케쥴링 연산을 하지 않고 수행을 위해서 작업 준비 큐에 들어갈 수 있다. 하지만, 기존의 비트 벡터 할당 기법에서는 이러한 태스크들에 대해서 불필요한 스케쥴링 연산을 수행하므로 수행 시간 측면에서 효율적이지 못하다. 그리고, 비트 벡터 할당 기법은 스케쥴링을 위해 복잡한 자료 구조와 많은 메모리를 필요로 한다. 따라서, 스케쥴링 오버헤드와 사용되는 메모리 양을 줄일 수 있는 간단한 태스크 스케쥴링에 관한 연구가 필요하다.

본 논문에서는 HTG에서의 함수 병렬성 추출을 위한 새로운 동적 스케쥴링 기법을 제안한다. 성김도 크기가 작은 태스크들의 병렬 수행을 위해서는 빈번한 스케쥴링이 요구되어지며, 고성능의 병렬 시스템이 아닌 단일 처리기의 멀티스레드 구조에서도 수행 가능하기에 제안된 기법과 비교 대상 스케쥴링 기법을 다양한 플랫폼에 적용하기 위해 Java 언어를 사용하여 프로세서의 기능을 스레드로 구현하였다. 성능 평가는 스케쥴링 오버헤드를 포함한 작업 시간, 부하 균형과 할당 기법에서 필요로 하는 메모리 양을 기존의 방법과 비교 분석한다.

본 논문의 구성은 다음과 같다. 제2장에서는 프로그램을 계층적으로 표현한 HTG와 이의 스케쥴링에 대해 알아보며, 3장에서는 HTG의 복합 노드를 표현한 ATG에서의 함수 병렬성 추출을 위해 제안된 동적 스케쥴링 기법을 소개한다. 4장에서는 스레드 프로그래밍 모델에서의 할당 기법의 구현과 스케쥴링 기법의 성능 평가에 대해 알아보며, 마지막으로 본 연구에 대한 결론을 기술한다.

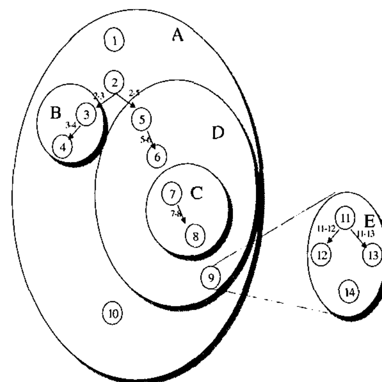
2. 관련 연구

프로그램의 중간 표현인 태스크 그래프를 계층적으로 표현하는 여러 가지 방법들이 제시되고 있으며, 알려진 연구는 인터벌(interval)을 기본으로 계층 구조를 만들었으나[17], 계층적 태스크 그래프(Hierarchical Task Graph, 이하 HTG)는 루프 구조를 기본으로 하여 계층을 정의한다[14,15]. HTG는 단순, 복합, 단순 루프, 복

```
1  V1 =V2
2  if (C1.gt.100) then
3     do 4   i = 1, n
4          V3 = V3 + i
   else
5     do 9 j = 1, n
6          V4 = V4 + j
7          do 8 k = 1, l
8               V5 = ·V5 +k
9          T = mul(j)
10  stop

   function mul(x)
11 if (x.le.10) then
12    mul = 10
   else
13    mul = x * 10
14 return
   end
```

(a) 예제 프로그램

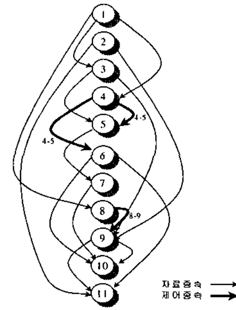


(b) 예제 프로그램의 HTG

그림 1 예제 프로그램과 HTG

```
1   A = N ** 2
2   B = N ** 4
3   C = A * 5
4   if (A.gt.0) then
5     D = C + 7
6     E = E * 10
    endif
7   F = D * 100 + 7
8   if (A.lt.20) then
9     G = G * C - B
    endif
10  H = E + 3 * G + F
11  I = E + G - B
```

(a) 예제 프로그램



(b) 예제 프로그램의 ATG

그림 2 예제 프로그램과 ATG

합 루프의 네 종류 노드들과 이들간의 자료와 제어 종속 관계를 표현한 간선으로 구성되는 방향성 비순환 그래프이다[14]. 그림 1은 예제 프로그램과 이에 해당하는 HTG를 나타낸 것이다. 여기서, 노드를 포함한 그림자를 가진 타원은 하나의 계층 혹은 레벨을 표현하며, 제어 종속 간선의 레이블 b는 종속성을 만족한 제어 흐름 간선을 나타낸다. 그림에서 노드 B와 C는 단순 루프 노드에 해당하며, D는 복합 루프 노드이다. 노드 1, 2, 10 등은 단순 노드에 해당되며, E는 서브루틴을 표현한 복합 노드이다. 이러한 HTG의 생성 방법과 병렬성 추출에 관한 연구는 Girkar[14,15]에 의해 상세히 다루어졌으며, 노드들은 성김도(granularity)의 크기에 따라 서로 다른 레벨로 구성될 수 있다. 즉, 성김도가 큰 루프 레벨과 작은 태스크 레벨에서 병렬성을 찾을 수 있기에 HTG의 스케쥴링은 병렬 루프의 루프 스케쥴링과 순차 루프나 서브루틴(HTG 복합 노드)를 표현한 ATG의 태스크 스케쥴링으로 나눌 수 있다[14,15,16].

본 논문에서 제안된 함수 병렬성 추출을 위한 스케쥴링 기법의 대상이 되는 ATG는 HTG의 복합 노드를 표현한 그래프이다. ATG 노드는 제어 흐름 그래프(Control Flow Graph, 이하 CFG)의 노드로 구성되며, 간선은 태스크들 간의 자료와 제어 종속성을 표현한다. 간선에 있는 레이블 $p-q$ 는 제어 흐름 그래프에서 노드 p에서 q로의 제어 종속성이 만족함을 의미한다. 그림 2는 예제 프로그램과 그에 해당하는 ATG이다.

HTG 복합 노드에서의 함수 병렬성 추출을 위한 태스크 스케쥴링에 관한 연구는 그다지 많지 않으며, 소개되어진 자동 스케쥴링 기법으로는 Moreira가 제안한 비트 벡터 방법이 있다[16]. 이 기법은 ATG의 상태를 기록하기 위해 비트 벡터 *SATISFIED*를 사용한다. 노드 수가 n 인 ATG에 대해, 자료 종속 정보를 표현하기 위한 자료 구조로 $DONE[1..n]$을 사용하며, 제어 종속 정보를 표현하기 위해서 배열 $CONT[1..n]$을 사용한다. 이들 자료 구조는 각각 임계 영역에 위치하여야 한다. 이 영역의 접근 횟수를 줄이기 위해 두 자료 구조를 결합하여 $SATISFIED[1..2n]$로 표현한다. 여기서, 상위 n 비트는 자료 종속 정보를 저장하는데 이용되며, 하위 n 비트는 제어 종속성을 기록하는데 사용된다. 스케쥴링을 위해서 각각의 노드 k는 l 개의 분기 개수에 해당하는 $SETBIT_{kl}[1..2n]$ 배열을 가진다. 이 자료 구조는 작업을 마친 노드 k가 분기 b로 제어가 옮겨갈 때, 노드 k의 종속적 후행자 노드들에게 노드 k의 수행이 완료되었음을 알리는 정보를 포함하고 있다. 배열 *SETBIT*는 현재 만족된 종속성 정보들을 담고 있는 *SATISFIED*와 비트 연산을 수행하여 *SATISFIED*를 갱신한다. 노드 k의 종속적 후행자인 노드 i가 수행을 하기 위해서 만족되어야 하는 자료와 제어 종속성 정보들을 나타내고 있는 자료 구조 $TESTBIT_i[1..2n]$와 변경된 *SATISFIED* 배열을 비트별 비교 연산을 한다. 연산 결과에 따라서 노드 i의 수행 조건이 만족되었으면 노드 k의 작업을 마친 프로세서에 의해 노드 i가 실행 준비 큐에 들어가게 된다[16].

노드 k의 종속적 후행자들 중에서 하나의 종속성만을 가진 노드들은 k의 작업 완료 후에 바로 수행 가능하기에 별도의 스케쥴링 오퍼레이션을 행하지 않고 작업 준비 큐에 들어갈 수 있다. 하지만, 기존의 비트 벡터 할당 기법에서는 노드 k의 모든 종속적 후행자 노드들의 수행 가능 여부를 검사하기에 많은 비교 연산을 필요로 하여 수행시간 측면에서 비효율적이다. 그리고, 비트 벡터 할당 기법은 스케쥴링을 위해 복잡한 자료 구조와 많은 메모리를 필요로 한다.

3. 태스크 스케줄링

3.1. 수행 조건 평가

계층적 태스크 그래프의 복합 노드를 표현한 ATG의 동적 스케줄링은 종속성 이론과 Girkar[14,15]의 수행 조건들에 기초를 두고 있다. ATG 각 노드는 수행 태그를 가지며, 그 값이 참이 될 때 그 노드는 수행을 위한 준비 상태가 된다. 노드 v_i의 수행 태그 ε는 아래와 같이 표현된다[14,15,16].

$$\varepsilon(v_i) = \varepsilon_c(v_i) \wedge \varepsilon_d(v_i) \tag{1}$$

여기서, $\varepsilon_c(v_i)$는 v_i 노드의 제어 종속 조건을 나타내며, $\varepsilon_d(v_i)$는 노드 v_i의 자료 종속 조건의 만족 여부를 나타낸다. 수행 태그는 부울 표현으로 나타낼 수 있으며, 노드 v_i가 수행되기 위한 조건은 v_i 노드의 자료 종속 조건이 만족되어야 하고 또한, v_i 노드의 제어 종속 조건도 참이 되어야 한다.

노드 v_i가 여러 개의 노드들로부터 자료 종속 관계를 가졌다면, 각각 해당하는 모든 노드들의 자료 종속 조건이 모두 만족해야만 한다. 이것을 식으로 표현하면 아래와 같다[14,15].

$$\varepsilon_d(v_i) = \bigwedge_{v_j \in DPred(v_i)} \varepsilon_d^{v_j} \tag{2}$$

여기서, $DPred(v_i)$는 노드 v_i의 자료 종속적 선행자 노드들의 집합이다.

```
Entry Block :
    Lock(TaskQueue)
        get the task from front of the queue
    Unlock(TaskQueue)

Execution Block :
    execute the task v_i

Exit Block :
1.  for each one_successor v_j ∈ Succ*(v_i, b) do

2.      enqueue v_j for execution
3.  end for

4.  Lock(DepCount)
5.      DepCount = Fetch_Add(DepCount, AddBit_{v_i,b})
6.      Temp = DepCount
7.  Unlock(DepCount)

8.  for each successor v_k ∈ Succ*(v_i, b) − v_j do
9.      if ((Temp && MaskBit_{v_k}) == CountBit_{v_k}) then
10.         enqueue v_k for execution
11.     end if
12. end for
```

그림 3 제안된 비트 카운트 기법

노드 v_i가 v_j에 자료 종속적이라 할 때, v_i가 자료 종속 조건이 만족되는 경우는 노드 v_j의 수행이 완료되거나 혹은, 노드 v_j를 비키어 가는 분기들 중 하나가 선택되어지면 조건이 만족되게 된다. 그것을 표현하면 식 3과 같다[14,15].

$$\varepsilon_d^{v_j} = v_j \vee \Big(\bigvee_{b \in Neg(v_j)} b \Big) \tag{3}$$

여기서, $Neg(v_j)$는 v_j를 수행하지 않는 분기이다.

3.2. 태스크 스케줄링 기법

본 논문에서 제안한 태스크 스케줄링 기법은 HTG 복합 노드를 표현한 비순환 태스크 그래프(Acyclic Task Graph, 이하 ATG)의 태스크들을 공유 메모리 다중 처리기 환경에서 효율적으로 수행하기 위한 자동 스케줄링 기법이다. 함수 병렬성 추출을 위한 비트 카운트 알고리즘은 그림 3과 같으며, 태스크들이 만족해야 할 자료와 제어 종속성의 개수 정보를 이용하여 간단히 스케줄링 한다. 수행을 위한 태스크 혹은 노드들은 식별자를 가지며, 수행 가능한 태스크들은 하나의 작업 큐(*TaskQueue*)에 적재된다. 독립적인 노드들이 작업 큐에 먼저 들어가게 된다. 제안된 태스크 할당 기법의 수행에 필요한 자료 구조들은 기억 장소 크기를 줄이기 위해서 필요한 정보를 비트 형태로 저장하며, 실제 구현은 자바의 부울 자료형을 사용하였다. 제안된 태스크 스케줄링 기법은 기능에 따라 크게 세 부분으로 나누어진다. 그것은 작업 큐로부터 태스크를 가져오며(그림 3의 입구 부분), 가져온 태스크의 코드를 실행한다(그림 3의 수행 부분). 그리고, 작업 완료된 노드의 종속적 후행자 노드들 중에 현재 수행 가능한 태스크들을 작업 준비 큐에 집어넣는다(그림 3의 출구 부분).

수행 코드 앞부분에 스케줄링을 위해 추가되는 그림 3의 입구 부분은 프로세서들이 모든 작업이 끝날 때까지 작업 큐의 프론트(front)로부터 하나의 태스크를 가져온다. 이때, 여러 프로세서들이 접근하는 임계 영역에 해당하는 작업 큐에 잠금(lock)과 해제(release)를 이용하여 직렬화(serialization)를 유지한다. 수행 부분에서는 입구 부분에서 가져온 태스크를 실행하게 된다. 그림 3의 출구 부분(줄 1-12)은 노드 v_i의 수행 후, 분기 b가 선택되었을 때에 스케줄링을 위해 추가된다. 출구 부분은 크게 세 부분으로 나누어진다. 그것은 현재 수행이 끝난 노드 v_i 에게만 종속적인 후행자 노드들(v_j)의 실행을 위해 작업 큐에 넣는 부분(그림 3의 줄 1-3), 노드 v_i가 작업 완료됨을 종속적 후행자 노드들에게 알리는 부분(줄 4-7), 그리고, 노드 v_i 외에 다른 노드들과도

종속 관계가 있는 후행자 노드들(v_k)의 수행 가능 여부를 검사하여 수행 조건이 만족된 태스크를 작업 큐에 넣은 부분이다(그림 3의 줄 8-12).

그림 3의 줄 1-3은 분기 b에 대해 v_i의 후행자 노드가 충족해야 될 종속성이 하나뿐이라면, 이러한 노드들은 v_i의 작업 완료 후에 바로 수행 가능하기에 비교 연산(줄 9)을 행하지 않고 작업 큐에 들어가게 된다. 이렇게 됨으로, 다른 휴면 프로세서들은 빠른 시간 내에 작업 큐에 새로 들어온 태스크를 가져와 수행 할 수 있다. 하지만, 기존의 비트 벡터 알고리즘에서는 노드 v_i의 모든 후행자 노드들의 수행 가능 여부를 검사하기에 많은 비교 연산을 필요로 한다. 그림 3에서 표현 $Succ^*(v_i, b)$는 식 4와 같다.

$$Succ^*(v_i, b) = [DSucc(\{v_i\}) \cup DSucc\{BranNeg(b)\}] - BranNeg(b) \quad (4)$$

여기서, $DSucc(\{v_i\})$는 노드 v_i의 자료 종속적인 후행자 노드들의 집합이며, $BranNeg(b)$는 분기 b로 인하여 수행되지 않는 노드들의 집합이다. 그림 3의 *one_successor*는 후행자 노드 중에서 노드 v_i에 대해서만 종속 관계를 가지는 후행자 노드들이다.

그림 3의 줄 4-7은 HTG 복합 노드의 태스크 v_i가 프로세서에 의해 실행이 끝났음을 종속적 후행자 노드들에게 반영하기 위해 현재 각 노드가 만족된 자료와 제어 종속성 개수를 담고있는 *DepCount*에 접근한다. 스케쥴링 연산(*Fetch_Add*)에 의해 현재의 *DepCount* 값과 프로세서가 작업한 노드 v_i의 상태 변수 $AddBit_{v_i b}$ 값을 비트별 덧셈을 하여 *DepCount* 값을 갱신한다. 노드 v_i는 l 개의 분기 개수에 해당하는 *AddBit* 값을 가진다. 자료 구조 $AddBit_{v_i b}$는 노드 v_i가 분기 b로 제어를 옮길 때, 후행자 노드들이 노드 v_i의 종속성을 만족했음을 나타내는 정보를 담고 있다. 그림 3의 줄 5에 의해 노드 v_i의 후행자 노드들의 현재 만족된 종속성 개수가 1 증가된다. 이것은 후행자 노드 입장에서 보면 자신이 고려해야 할 여러 종속 관계 중 하나가 만족됨을 의미하며, 모든 종속 관계가 만족 될 때 비로써, 실행을 위한 준비 상태가 된다. 변경된 *DepCount*를 이용하여 후행자 노드의 수행 가능 여부를 검사한다. 즉, 만족해야 할 종속 관계가 모두 만족되었는지 살펴본다. 이때, 임계 영역인 *DepCount*에 락을 걸고 수행 비교 연산을 하는 동안 다른 프로세서들이 *DepCount* 변수에 접근시 대기 상태가 된다. 프로세서가 공유 변수 *DepCount*에 접근 해 있는 시간을 줄이기 위해 이를 지역 변수 *Temp*에 저장한다. 스케쥴링을 위해 프로세서들이 참조하는 *DepCount*는 임의의 노드가 수행될 때마다 갱신되며, 초기 값은 모두 0으로 채워진다. 응용에 따라 *DepCount*가 필요로 하는 비트 수는 식 5와 같다.

$$\sum_{i=1}^{n} \lceil \log_2(NodeCount(v_i)+1) \rceil \quad (5)$$

식 5에서 $NodeCount(v_i)$는 노드 v_i가 수행을 하기 위해 만족해야 될 자료와 제어 종속성의 전체 수이며, 식 6과 같이 표현된다.

$$NodeCount(v_i) = DepData(v_i) + DepCntl(v_i)$$
$$DepCntl(v_i) = 1 \quad \text{If, } InCntl(v_i) \geq 1$$
$$DepCntl(v_i) = 0 \quad \text{otherwise} \quad (6)$$

여기서, $DepData(v_i)$는 노드 v_i로 들어오는 자료 종속 간선의 개수이며, $DepCntl(v_i)$는 제어 종속 간선과 관계된다. $InCntl(v_i)$는 노드 v_i로 들어오는 제어 종속 간선의 전체 수이다. 각 노드는 스케쥴링을 위하여 자신에게로 들어오는 자료와 제어종속 간선 수의 합 정보를 가진다. 만약, 노드 v_i에 들어오는 제어 종속 간선이 여러 개일 경우에는 한 개만 고려한다. 왜냐하면, 이 노드로 들어오는 여러 개의 제어 종속적인 분기 중에 하나만 선택이 되어지면 제어 종속성이 만족되기 때문이다. 메모리 사용량과 계산 효율(스케쥴링 오버헤드)을 감안하여 *DepCount*가 필요로 하는 비트 수를 최적화하면 식 7과 같다.

$$\sum_{i=1}^{n} \lceil \log_2(NodeCount(v_i)+1) \rceil - Noincomming \quad (7)$$

여기서, *Noincomming*은 $NodeCount(v_i)$가 1인 노드들이 필요한 비트 수의 합이다.

식 7에 의해 종속적 선행자가 하나인 노드들의 정보는 *DepCount*에 표현 할 필요가 없다. 왜냐하면, 이러한 노드들은 선행자의 작업이 완료되면 바로 수행할 수 있기에 부가적인 스케쥴링 오퍼레이션 없이 작업을 마친 프로세서에 의해 작업 준비 큐에 들어가게 된다. 스케쥴링의 초기화 단계에서 $NodeCount(v_i)$가 0인 노드들은 병렬적으로 먼저 실행된다. 이러한 노드들은 만족해야 할 자료나 제어 종속성이 없는 독립적인 태스크이기 때문이다.

그림 3의 줄 8-12는 노드 v_i의 후행자 노드들 중에 만족해야 할 종속 관계가 두 개 이상인 태스크(v_k)들의 수행 가능 여부를 검사 후, 수행 조건이 만족된 태스크를 작업 큐에 집어넣는다. 그림 3의 줄 9는 공유 변수 *DepCount* 값을 저장하고 있는 *Temp* 변수에서 v_i의

후행자 노드인 v_k의 현재 만족된 종속성 개수만을 추출하기 위해서 $MaskBit_{v_k}$를 이용해 비트별 AND 연산을 한다. 비트 연산의 결과 값과 노드 v_k가 만족해야 할 전체 종속 개수를 비트 형태로 담고있는 $CountBit_{v_k}$와 비교하여 같으면 수행을 위해 프로세서가 태스크 v_k를 작업 준비 큐에 집어넣는다.

4. 구현 및 성능 평가

본 논문에서 제안된 스케쥴링 기법을 단일처리기 시스템을 비롯한 여러 플랫폼에서 수행하기 위해 JDK 1.2.2와 통합 개발 패키지인 Kawa 3.22를 이용하여 프로세서의 자동 스케쥴링 기능을 스레드에서 수행 되도록 하였다.

4.1. 자바 스레드 구현

자바는 두 종류의 스레드를 제공하며 실험에 사용된 것은 그 중, 그린 스레드(green thread)로 스레드 대 프로세서의 매핑은 M:1로 이루어진다[18,19]. 스레드 구현은 스레드 클래스로부터 상속받아 하위 클래스에서 런 메소드를 오버라이딩 하였다. 임계 영역의 동기화 구현은 잠금을 위한 *synchronized* 키워드와 *wait*, *notifyAll* 등의 동기화 메소드를 사용하였다. 다음은 함수 병렬성 추출을 위해 제안된 비트 카운트 스케쥴링 기법의 구현에 사용된 클래스들이다.

(1) Schedular ; 스케쥴링을 위한 초기화 작업을 수행한다. 먼저, 스케쥴링 방법(비트 벡터 혹은 제안된 기법)에 따라 *TaskPool* 객체를 생성하며, 독립적인 태스크들은 수행을 위해 작업 큐에 삽입된다. 그리고, 지정한 개수에 따라 스레드를 생성 후 런 상태로 만든다.

(2) TaskPool ; 임계 영역에 해당하는 작업 준비 큐의 구현을 위한 것이다. 태스크의 개수에 따라 태스크 풀의 크기가 정해진다. 작업 큐에 태스크를 삽입할 때는 동기화가 필요하므로 락을 걸고 큐의 뒤쪽에 하나의 태스크를 삽입 후 락을 해제한다. 작업 큐로부터 태스크를 가져올 때 또한 동기화가 필요하다.

(3) EnqueueLock, DequeueLock ; 태스크 풀의 동기화를 구현한 것으로 *synchronized* 키워드를 이용한 인큐, 디큐시의 락 메소드와 스레드간의 동기화를 위한 *wait*, *notifyAll* 메소드가 사용된다.

(4) Lock ; 각 노드들의 현재 만족된 종속성 정보를 담고있는 *DepCount*의 접근 제어를 위한 클래스이다. 여러 프로세서가 공유 변수 *DepCount*에 접근하기에 동기화가 필요하다. *synchronized* 키워드를 이용한 락과 언락 메소드와 스레드간의 동기화 메소드를 이용한다.

(5) LabledBit ; 스케쥴링 연산 *Fetch_Add*를 비트별로 구현하였으며, 이 연산에서 필요로 하는 비트별 가산, *and*, *or*, *equal*, 치환의 비트 오퍼레이션을 위한 메소드들과 비트 벡터 알고리즘에서 사용되는 여러 오퍼레이션들을 포함한다.

(6) ProposedAlgorithm ; 응용 프로그램에 따른 초기화 작업으로 병렬화 컴파일러에 의해 생성된 스케쥴링에 관련된 정보들을 세팅한다. 노드간의 종속 관계를 표현하는 방향성 간선과 알고리즘에서 필요로 하는 자료 구조의 생성과 초기화, 응용에 따른 *Noincomming*과 간선의 개수에 따라 필요한 비트 수를 동적으로 결정하여 배열을 생성한다. 또한, 각 노드가 필요로 하는 해당 *AddBit*, *MaskBit*, *CountBit*를 가져오는 메소드로 구성된다.

(7) ProposedThread ; 앞장에서 설명한 그림 3의 자동 스케쥴링 기능을 스레드 레벨에서 구현한 것으로, 런 메소드는 스레드가 휴면 상태이면 작업 큐의 앞 부분에서 하나의 태스크를 가져와 수행한다. 작업 큐에 접근해 있는 스레드가 있으면 대기 상태가 되며, 스레드의 동기화 메소드에 의해 다시 깨어나게 된다. 태스크의 수행을 끝낸 후에 종속적 후행자 노드를 찾아서 현재 실행 가능한지를 검사하여 수행 조건이 만족되면 작업 준비 큐에 해당 태스크를 삽입한다.

다음 두 개의 클래스는 제안된 태스크 스케쥴링 기법의 성능 평가 대상인 비트 벡터 알고리즘을 동일한 환경에서 수행하기 위해 구현에 사용된 클래스들이다. 많은 부분들이 앞에서 설명한 클래스들의 메소드와 멤버 변수를 공유한다.

(1) BitVectorAlgorithm ; 응용 프로그램을 표현한 ATG 정보와 스케쥴링을 위해서 필요한 정보를 세팅한다. 즉, 노드간의 종속성을 표현하는 방향성 간선과 알고리즘에서 필요로 하는 자료 구조를 생성 후 초기화 값을 세팅한다. 이러한 정보들은 스케쥴링을 위해 필요하다. 상태 변수 *SETBIT*와 *TESTBIT* 정보를 가져오는 메소드들이 구현된다.

(2) BitVectorThread ; 비트 벡터 할당 기법에 따라 프로세서가 수행 할 스케쥴링 기능을 스레드로 표현한 것으로 *BitVectorAlgorithm* 객체에 저장된 정보에 따라 태스크 풀에서 태스크를 가져와서 수행하고 다음에 수행할 태스크들을 태스크 풀에 저장한다.

4.2. 성능 분석

제안된 태스크 스케쥴링 기법의 성능을 평가하기 위해 HTG의 함수 병렬성 추출을 위한 대표적인 태스크 스케쥴링 방법인 비트 벡터 알고리즘을 비교 대상으로

선택하였다. 평가 요소는 동적 스케쥴링 기법의 성능을 좌우하는 스케쥴링 오버헤드를 포함하는 전체 태스크의 작업 완료 시간과 스레드의 부하 균형 그리고, 할당 기법에서 필요로 하는 자료 구조의 메모리 양을 측정하였다. 이를 위해 비트 벡터 알고리즘을 제안된 스케쥴링 기법과 동일한 형태인 자바 스레드 레벨로 구현 후, 다양한 플랫폼(윈도우즈 98/NT 4.0, 솔라리스 2.6, 래드햇 리눅스 6.1)에서 수행하였다. 실험에 사용된 환경은 다음과 같다.

(1) 시스템 A ; Pentium Pro 200MHz, 64MB - Windows 98

(2) 시스템 B ; SUN Ultra 140 (ultraSPARC 167MHz, 132MB) - RedHat Linux 6.1

(3) 시스템 C ; SUN Ultra Enterprise 450 (ultraSPARC-Ⅱ 248MHz,1048MB) - Solalis 2.6

(4) 시스템 D ; 삼보 STD workstation 820 (60MHz X 2, 64MB) - SUN OS 4.1

(5) 시스템 E ; Dell Precision workstation 420 (Pentium-Ⅲ 733MHz X 2, 512MB-PC800) - Windows NT 4.0

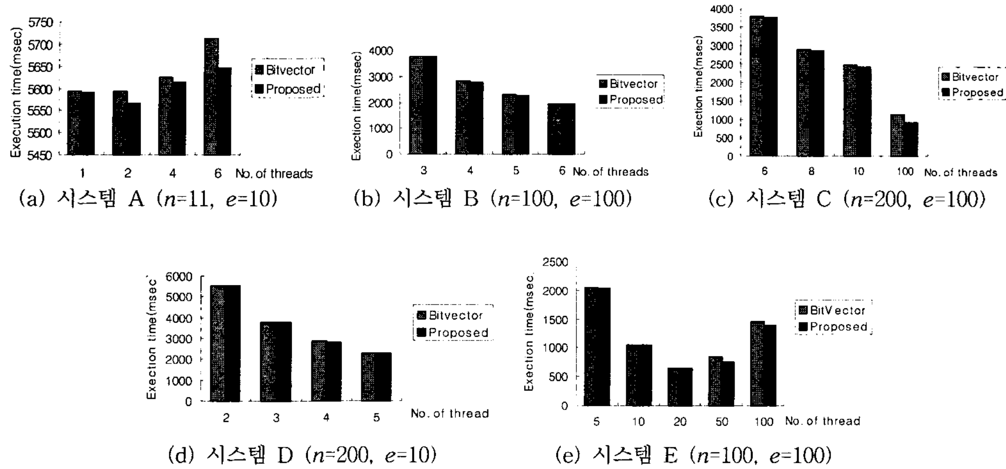
실험에 사용된 응용 프로그램의 파라메터 값으로 작업 노드의 실행 비용(e)은 5, 10, 20, 50, 100, 500, 1000, 5000 밀리 초(milliseconds, 이하 msec), ATG 노드의 수(n)는 10, 11, 20, 50, 100, 200개이다. 그리고, 그래프의 전체 간선의 수(a)는 자료와 제어 종속 간선이 각각 10, 3개인 13개에서 최대 260개이다. 하나의 노드가 가지는 최대 종속 간선의 수(a_{one})는 세 개인 응용에 대해 실험을 하였다. 시스템 파라메터 값으로 스레드 수(t)는 1개에서 응용에 따라 최대 1000개까지 랜덤 하게 생성하였으며, 사용된 프로세서 수(p)는 병렬 수행할 태스크의 성김도가 작기 때문에 최대 2개 (시스템 D, E)까지 하였다.

스레드 수, 프로세서 수, 태스크 수와 태스크의 크기에 따라 동적 태스크 할당 기법의 스케쥴링 오버헤드를 포함한 전체 수행 시간의 측정 결과는 표 1과 같다. 시스템 A, B, C, D, E에서 수행 한 결과, 수행 시간 측면에서 제안된 태스크 스케쥴링 기법은 기존의 방법에 대해 각각 0.82, 3.76, 1.81, 0.3, 0.12%의 성능 향상을 보였다. 이것은 제안된 동적 할당 기법의 스케쥴링 오버헤드가 기존의 방법에 비해 감소된 결과이다. 제안된 할당 기법은 태스크들이 만족해야 할 자료와 제어 종속성의 개수 정보를 이용하여 간단히 스케쥴링하며 또한, 만족

표 1 태스크 스케쥴링 기법의 수행 시간(단위: msec)

n	*e*	*t*	Bit vector	Proposed
시스템 A				
4	5	4	232	228
	100	4	1256	1190
20	10	2	235	233.75
		4	166.25	163.75
	30	2	598.75	596.25
		4	302.5	302.5
시스템 B				
11	10	4	307.5	280
	500	4	5662	5650
100	100	1	11012.1	11008.3
		2	5525.2	5511.1
		3	3748.7	3747.7
		4	2834.8	2805
		5	2301.5	2278.8
		6	1956.6	1926.2
	5000	4	125335.2	125304.4
		20	30259.3	30256.8
200	10	100	765.66667	626
		200	1421.667	766.6667
	100	4	5600.5	5592.5
		100	1278.667	889
		200	1683.333	1011.333
	1000	100	5794.5	5267.5
		1000	11905	6865

n	*e*	*t*	Bit vector	Proposed
시스템 C				
200	10	2	2052	2043.5
		8	792	778
	100	10	2475.7	2425.3
		100	1121.7	928
	1000	200	5573.5	5452.5
		1000	6435	5575.5
시스템 D				
100	100	2	5535.8	5533.3
		3	3796.3	3787.9
		5	2298	2274.9
시스템 E				
11	5	2	78.4	78
		4	78	78.3
100	100	1	10090.6	10085.8
		2	5061	5054.9
		4	2543.8	2539
		6	1734.6	1734.2
		100	1454.6	1389
	5000	4	125029.5	125018.8
		20	30047	30039.1
200	100	2	10089.88	10087.75
		4	5060.375	5049
		8	2552.333	2547
	1000	4	50032	50031



(a) 시스템 A (n=11, e=10) (b) 시스템 B (n=100, e=100) (c) 시스템 C (n=200, e=100)

(d) 시스템 D (n=200, e=10) (e) 시스템 E (n=100, e=100)

그림 4 스레드 수에 따른 수행 시간

해야될 종속성이 하나인 태스크들은 부가적인 스케쥴링 오퍼레이션 없이 스케쥴링 되기에 수행 시간에 밀접한 영향을 미치는 스케쥴링 오버헤드를 줄일 수 있다.

응용 프로그램과 시스템 파라메터에 따른 수행 시간의 관계를 살펴본다. 그림 4는 다양한 플랫폼에서 스레드 수에 따른 제안된 기법과 비트 벡터 알고리즘의 수행 시간을 나타낸 것이다. 일반적으로, 스레드 수를 증가하므로 수행 시간을 감소시킬 수 있었지만, 두 개의 스케쥴링 기법 모두 스레드 수를 증가시킴에도 불구하고 성능이 떨어지는 경우도 있었다(그림 4(a),(e)). 스레드가 어떻게 동작하는가는 플랫폼마다 다르고 플랫폼간의 성능 차이 때문에 스레드의 실행 속도가 다르다[18, 19]. 따라서, 성능 향상을 위한 적정한 스레드 개수를 정하기는 힘이 들며, 특정 응용과 시스템에 대한 반복된 실험에 의해 알 수 있었다.

표 2에서 보듯이, 사용된 프로세서 개수 또한 수행 시간에 영향에 미쳤다. 실험 대상이 파인 그레인의 태스크이며, 스케쥴링 연산 시간이 노드의 수행 시간에 비해 상대적으로 짧기에 2개의 처리기까지 실험하였다. 예로서, n=11, e=10인 응용에 대해 4개의 스레드를 생성하여 수행한 경우, 비트 벡터 방법과 제안된 할당 기법이 시스템 A 환경에서 각각 307.5, 280 msec의 수행 시간이 소요되었으며, 프로세서의 성능이 좋은 시스템 B에서도 307.5, 280 msec로 동일한 수행 시간을 유지하였다. 반면에, 2개의 프로세서를 가진 시스템 E에서는 각각 137.4, 128 msec로 좋은 성능을 보였다. 즉, 단일 처리기 시스템에서는 별다른 성능 차이가 없었고, 처리 능력이 우수한 2개의 프로세서를 이용하여 수행 시간을 줄일 수 있었다. 또한, n=100, e=100인 응용에 대해 처리 능력이 떨어지는 2개의 프로세서를 가진 시스템 D(60MHz)와 단일 처리기 시스템B(167MHz)에서 수행한 결과, 3개까지의 스레드를 생성시켰을 경우는 단일 처리기 시스템의 성능이 우수하였다. 그러나, 4, 5개의 스레드를 사용한 경우는 다중 처리기 시스템이 짧은 수행 시간을 유지하여 프로세서 수에 따른 성능 차이를 보였다. 대체적으로 많은 수의 프로세서를 사용하여 수행 시간을 줄일 수 있었지만, 사용된 프로세서 수에 따른 성능 향상은 시스템과 응용에 따라 종속적임을 알 수 있었다.

표 2 프로세서 수에 따른 수행 시간 - 시스템 E

(단위 : msec)

p	n	e	t	Bit vector	Proposed
1	100	100	3	3748.7	3747.7
2	100	100	3	3446.6	3437.3
1	100	100	5	2301.5	2278.8
2	100	100	5	2045.3	2034.5
1	100	5000	20	30259.3	30256.8
2	100	5000	20	30047	30039

수행 시간에 영향을 미치는 응용 프로그램의 인자들 중에 작업량(태스크 수)과 태스크의 크기 변화에 따른 수행 시간을 알아보고자 한다. 먼저 작업량의 변화로, 동일한 수의 스레드를 사용하여 태스크 수를 증가함에 따라 수행 시간이 길어졌다. 예로서, 시스템 B에서 n=100, e=100인 응용에 4개 스레드를 투입한 결과, 비

표 3 스케줄링을 위해 필요한 비트 수

Bit Vector Algorithm		Proposed Algorithm	
TESTBIT	$2n^2$	*DepCount*	$\sum_{i=1}^{n} \lceil \log_2(NodeCount(v_i)+1) \rceil - Noincomming$
SATISFIED	$2n$	*AddBit*	$\sum_{k=1}^{n-1} DepCount(l_k)$
SETBIT	$\sum_{k=1}^{n-1} 2n(l_k)$	*MaskBit*	$n * DepCount$
		CountBit	$n * DepCount$
Total	$2n(n+1) + \sum_{k=1}^{n-1} 2n(l_k)$	*Total*	$2(n*DepCount) + \sum_{k=1}^{n-1} DepCount(l_k) + DepCount$

트와 제안된 알고리즘이 각각 2834.8, 2805 msec의 수행 시간을 가짐에 반해, 노드 수를 200개로 증가한 경우는 5600.5, 5592.5 msec의 시간이 걸렸다. 결과적으로, 노드 수가 두 배로 증가함에 따라 약 두 배 정도의 수행 시간을 보였다.

다음으로 태스크의 크기, 즉 노드의 연산 시간을 변화시켜 보았다. 예로서, 시스템 A에서 노드 수가 11개인 ATG에 대해 노드 당 연산 시간을 5, 10, 100, 500 msec로 변화시켜 제안된 알고리즘을 이용해 실험한 결과, 수행 시간이 각각 228, 280, 1190, 5614 msec로 길어졌다. 따라서, 응용 프로그램의 파라메터인 n, e 에 비례하여 수행 시간이 증가하였다.

부하 균형을 측정한 결과, 제안된 기법과 비트 벡터 할당 방법 모두 단일처리기 시스템에서는 동일한 개수의 태스크를 수행하였으나, 먼저 수행을 끝낸 스레드의 종료 시각과 응용의 마지막 태스크를 수행한 스레드의 종료 시각간의 차이는 매번 실행 때마다 크게 생겨 부하 균형이 둘 다 좋지 못했다. 하지만, 다중처리기 환경에서는 스레드들의 종료 시점이 모두 동일하였다. 또한, 각 스레드가 작업에 참여한 시간이 같아 두 알고리즘 모두 좋은 부하 균형을 보였다. 그렇지만 많은 수의 스레드를 생성한 경우, 실험 결과로 20개 이상의 스레드를 생성했을 때 종료 시점의 편차가 조금 생겼다.

스케줄링을 위한 할당 기법은 간단하게 구현되어야만 스케줄링 오버헤드와 필요한 메모리 양을 줄여 효율적이다. 표 3은 제안된 기법과 비트 벡터 할당 방법에서 필요로 하는 메모리 양을 나타낸 것이다. 여기서, l_k는 노드 k의 분기 개수이다. 표 3을 이용하여 실험에 사용된 응용 프로그램의 파라메터 값에 따라 필요로 하는 메모리 사용량을 나타낸 것이 표 4이다. 여기서, l_{tot}는 노드들의 전체 분기 개수이며, 하나의 노드가 가지는 최대 종속 간선은 수는 3이다. 표 4에서 보듯이, 태스크 개수가 많고 노드로 들어오는 간선의 개수가 1개 이하인 노드가 많을수록 제안된 기법은 비트 벡터 알고리즘에 비해 많은 메모리 양을 줄일 수 있다. 비트 벡터 알고리즘에서 필요로 하는 메모리 양은 ATG 노드 수에 비례하며, 제안된 알고리즘은 *Noincomming*의 비트 수에 종속적임을 알 수 있다. 즉, 응용에 따라서 두 할당 기법에서 사용된 메모리 양의 정도가 다르다. 비트 벡터 알고리즘은 노드 수에 따라 메모리 양이 결정되기에 필요로 하는 메모리 양은 동일하다. 하지만, 제안된 알고리즘은 *Noincomming*에 종속적이기에 필요한 메모리 양의 변화가 있다. 표 4(b)에서 보듯이 들어오는 간선이 1개 이하인 노드 수가 전체 노드 수에 대해 절반 이하인 응용에서는 제안된 할당 기법을 사용하여 기존 스케줄링 방법에 비해 약 26%의 메모리를 절약할 수 있다.

표 4 파라메터 값에 따라 필요한 비트 수

(a) n의 변화

n	l_{tot}	*Noincomming*	필요한 비트 수	
			Bit Vector	Proposed
11	12	6	528	280
10	11	7	440	192
20	22	14	1720	756
100	110	70	42200	18660
200	220	140	168400	74520

(b) *Noincomming*의 변화

n	l_{tot}	*Noincomming*	필요한 비트 수	
			Bit Vector	Proposed
100	110	100	42,200	0
		70	42,200	18,660
		50	42,200	31,100
		30	42,200	43,540
		0	42,200	62,200

5. 결 론

본 논문에서는 공유 메모리 다중처리기 환경에서 프로그램을 계층적으로 표현한 HTG의 함수 병렬성 추출을 위한 새로운 태스크 스케줄링 기법을 제안하였다. 성김

도의 크기가 작은 태스크들의 병렬 수행을 위해서는 빈번한 스케쥴링이 요구되어지며, 고성능의 병렬 시스템이 아닌 단일처리기의 멀티스레드 구조에서도 수행 가능하기에 제안된 기법을 여러 플랫폼에 적용하기 위해 자바 스레드를 이용하여 구현하였다. HTG의 함수 병렬성을 위한 기존의 비트 벡터 알고리즘과 응용 프로그램 파라메터, 시스템 파라메터 값을 변화시켜 다양한 플랫폼에서 스케쥴링 오버헤드를 포함한 전체 수행 시간, 부하 균형과 알고리즘에서 필요로 하는 메모리 양을 비교 분석하였다. 실험 결과, 제안된 기법이 전체 지연 시간을 최소화하여 비트 벡터 기법에 비해 수행 시간을 줄여 효율적임을 보였다. 일반적으로, 많은 수의 스레드를 사용함으로 성능을 향상시킬 수 있지만, 성능 향상을 위한 최적의 스레드 개수는 특정 응용과 시스템에 종속적임을 알 수 있었다. 부하 균형 측면에서는 제안된 기법과 비트 벡터 알고리즘 둘 다 단일처리기 시스템에서는 좋지 못했으나, 다중처리기 환경에서는 모두 좋은 부하 균형을 보였다. 제안된 기법이 필요로 하는 메모리 양은 ATG의 전체 노드 수에 대해 간선이 1이하인 노드 수가 절반 이하인 응용에 대해선, 기존의 알고리즘에 비해 약 26%의 메모리를 줄일 수 있다. 향후 연구방향은 성능 향상을 위해 솔라리스 스레드나 *Pthread*를 이용하여 하나의 작업 큐 사용에 의한 병목 현상을 해결하기 위한 분산 큐 기반의 스케쥴링 기법에 대해 연구하고자 한다.

참 고 문 헌

[1] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley,1996.

[2] M.J. Quinn, *Parallel Computing-Theory and Practice*, McGraw-Hill, 1994.

[3] H. Zima and B. Chapman, *Super Compiler for Parallel and Vector Computers*, Addison-Wesley, 1991.

[4] K. Kennedy, "Compiler Technology for Machine-Independent Parallel Programming," *Inter. Journal of Parallel Programming,* vol. 22, no. 1, pp.79-97, 1994.

[5] M. Gokhale and W.Carlson, "An Introduction to Compilation Issues for Parallel Machines," *The journal of Supercomputing*, vol. 6, pp.283-314, 1992.

[6] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J. Z.Fang and C.L. Thompson, "Compiler for Instruction-Level Parallelism," *IEEE Computer*, vol.30, no.12,pp.63-69, 1997.

[7] C.D. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, vol.36, no.12, pp.1425-1439, 1987.

[8] Z.Fang, P. Tang, P.C. Yew, and C.Q.Zhu, "Dynamic Processor Self-Scheduling for General Parallel Nested Loops," *IEEE Trans. on Computers*, vol. 39, no. 7, pp.919-929, 1990.

[9] S.E. Hummel, E. Schonberg, and L.E. Flynn, "Factoring : A Method for Scheduling Parallel Loops," *Comm. ACM*, vol. 35, no. 8, pp.90-101, 1992

[10] Y.Yan, C.Jin and X. Zhang, "Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems,"*IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 1, pp.70-81, 1997.

[11] S. Subramaniam and D.L. Eager, "Affinity Scheduling of Unbalanced Workloads," *Proc. Supercomputing '94*, pp.214-226, 1994

[12] E.P. Markatos and T.J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 4, pp.379-400, 1994.

[13] M. S. Squillante and E. D. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, pp.131-143, 1993.

[14] M.Girkar and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp.166-178, 1992.

[15] M. Girkar and C.D.Polychronopoulos, "Extracting Task-Level parallelism," *ACM Trans. on Programming Languages and Systems*, vol. 17, no. 4, pp.600-634, 1995.

[16] J.E. Moreira and C.D.Polychronopoulos, "Auto-scheduling in a Shared Memory Multiprocessor," Technical Report 1337, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1994.

[17] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools,* Addison Wesley, 1986.

[18] L. Lemay, *Teach Yourself JAVA in 21 Days*, Sams.net Publishing, 1998.

[19] M.Campione, *The Java Tutorial,* Addison-Wesley, 1999.

김 현 철
1995년 경일대학교 컴퓨터공학과 졸업(공학사). 1997년 경북대학교 컴퓨터공학과 졸업(공학석사). 2002년 경북대학교 컴퓨터공학과 졸업(공학박사). 2000년 ~ 2002년 포항1대학 정보통신과 교수. 2002년 ~ 현재 재능대학 컴퓨터정보계열 교수. 관심분야는 병렬 및 분산처리, 네트워 보안

김 효 철
1987년 경북대학교 전자공학과(전산전공) 졸업(공학사). 1989년 경북대학교 전자공학과(전산전공) 졸업(공학석사). 1989년 ~ 1996년 국방과학연구소(ADD) 연구원. 2002년 경북대학교 컴퓨터공학과 졸업(공학박사). 1996년 ~ 현재 계명문화대학 컴퓨터정보계열 부교수. 관심분야는 멀티미디어, 정보보안