□ 기술해설 □

하드웨어와 무관한 루프 병렬화

# 병렬화를 위한 루프 구조의 변환

충남대학교 이 만 호*

목 차



## 1. 서 론

순차 프로그램을 병렬 프로그램으로 자동적으로 변환하고자 하는 연구가 병렬 컴퓨터가 출현한 이후로 꾸준히 진행되어오고 있다. 이 작업은 최초의 전자식 컴퓨터가 출현한 이후로 오랫동안 단일 프로세서의 성능을 향상시키는 방향으로 하드웨어와 소프트웨어가 발전해 오는동안 축적된 많은 양의 순차 프로그램을 병렬 컴퓨터에서 성능 향상과 함께 무리없이 수행시키기 위해서 절대적으로 필요한 것이다. 또한 오늘날의 고성능 병렬 컴퓨터를 사용하여 프로그램을 작성할 필요성을 느끼는 사람들은 과학자나 공학자들로서, 그들은 기종마다 다양한 하드웨어적인 특성을 가지고 출현하고 있는 병렬 컴퓨터를 위한 병렬 프로그램을 작성하는데는 서툴고, 이제까지 익숙하게 사용해온 순차 프로그램에는 불편함을 별로 느끼지 못하기 때문에, 여전히 계속하여 순차 프로그램으로 자신들의 작업을 수행하고 있는 실정이다. 그러면서도 매우 빠르고 신속한 작업처리를 매우 강하게 필요로 하고 있다. 따라서 대부분의 프로그램 작성자가 작성한 순차 프로그램을 자동적으로 병렬 프로그램으로 변환할 수 있는 시스템을 절실히 필요로 하고 있다.

병렬 프로그램을 작성하는데 사용되는 언어는 기존의 순차 컴퓨터에서 널리 사용되고 있는 Fortran이나 C와 같은 언어를 병렬 프로그램용으로 확장된 것이거나[6,7,10], 병렬 프로그램을 위하여 새로 설계된 언어이다[15,27]. 전자는 주로 병렬 컴퓨터를 판매하는 회사에서 제공되며, 후자는 대학이나 연구소 등지에서 수행되는 연구 결과로서 나오고 있다. 그리고 하드웨어적 특징이 다양한 기종간에 사용될 수 있도록 호환성이 있는 병렬 프로그래밍 언어를 설계하기 위한 시도도 진행되고 있다[35].

기존에 사용하고 있는 순차 프로그램을 확장하는 경우는, 병렬성을 구조적으로 표현할 수

* 종신회원

있는 새로운 구조를 추가하는 방법[7], 컴파일러 지시어(directive)를 사용하여 병렬 코드를 생성하도록 지시하는 방법[11], 그리고 병렬 수행을 제공하는 라이브러리를 이용하는 방법[6] 등이 있다. 새로운 병렬 구조를 추가하는 방법과 컴파일러 지시어를 사용하는 방법은 주로 Fortran에서 사용하는 방법이고, 라이브러리를 이용하는 방법은 C에서 사용하는 방법이다. 이런 경향은 Fortran의 구조가 비교적 간단 명료한 반면에, C 언어는 하나의 구조가 다양한 방법으로 사용될 수 있기 때문이다. 그래서 대부분의 병렬 프로그램에 관한 연구는 그 대상을 주로 Fortran, 또는 이와 비슷한 syntax를 사용하고 있다. 따라서 본 고에서도 Fortran syntax를 사용하여 설명하려한다.

프로그램의 길이는 실행시 수행에 필요한 시간과 항상 비례하지 않는다. 참고문헌 [18]에 따르면 프로그램 길이의 4% 이하의 길이가 차지하는 프로그램이 전체 수행 시간의 50% 이상을 소비하고 있다고 한다. 이러한 사실은 대부분의 수행 시간이 지극히 적은 부분, 즉 루프(loop) 구조에서 소비되고 있음을 반증한다. 따라서 루프 구조를 효율좋게 작성하면, 그것이 전체적인 프로그램의 성능에 미치는 영향은 막대하며, 이러한 연유로 순차 프로그램을 병렬 프로그램으로 변환하는 연구의 대부분이 루프 구조의 변환에 치중되고 있다. 또한 많은 연구 결과들을 모아서 좀더 발전된 형태의 병렬 프로그래밍 환경(parallel programming environment)을 구성하는 데 관한 연구가 많은 대학에서 수행되고 있다. 예를 들어 Illinois 대학의 Parafrase[33]와 Faust[14], Rice 대학의 PFC[5], Bonn 대학의 SUPERB[39], Georgia 대학의 Start/Pat[8], IBM의 PTRAN[3] 등이 바로 그것이며, 이 연구들도 그 대상을 루프 구조의 변환에 중점을 두고 있으므로 본 고에서도 루프 구조의 변환만을 다루기로 한다.

루프 구조의 변환에 관한 연구는 루프 내에 존재하는 데이타 종속성을 분석하는 연구가 필수적이다. 이런 데이타 종속성 분석은 프로그램의 작업 수행 순서에 따라 하는 것이고, 프로그램이 작업을 수행할 하드웨어의 특성과는 전혀 무관하다. 그리고 데이타 종속성 분석 결과에 따라 병렬화를 위한 변환을 하게 되는데, 변환된 병렬 프로그램이 효과적으로 수행되기 위해서는 작업이 수행될 병렬 컴퓨터의 하드웨어적 특성을 잘 활용하여야만 한다. 그러나 병렬 컴퓨터는 기종간에 독특한 특성이 있기 때문에 일반적으로 효율이 좋은 병렬 프로그램을 작성하기는 불가능에 가깝다. 따라서 루프 구조 변환에 관한 연구는 하드웨어의 특성과 관계없이 진행되고 있다. 다만 종속성의 특성에 따라 병렬화에 적합한 변환 방법을 제시하고 있다. 그 연구 결과를 특정 병렬 컴퓨터에 적용할 때는 하드웨어의 특성을 활용하는 극히 일부분만 추가적으로 고려하면 충분하다. 따라서 루프 구조의 변환에 관한 연구는 하드웨어의 특성과 무관하게 연구를 진행할 수 있는 것이다.

루프 구조 변환에 관한 연구에서 주로 다루고 있는 루프 구조는 그림 1(b)에서 보는 것과 같이 루프 변수의 초기값이 1이고, 증가값도 1인 표

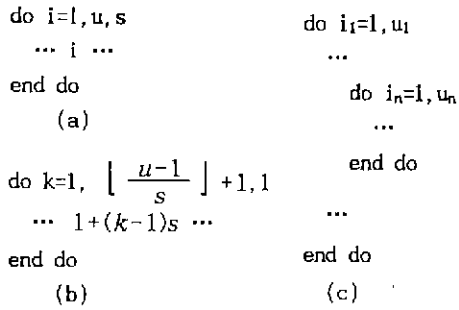


그림 1 표준화된 루프 구조

준화된 루프이다. 그러나 일반적인 루프 구조는 표준화 되어있지 않은 경우도 많이 발견된다. 여기서 그림 1(a)에 있는 것과 같이 표준화 되어 있지 않은 루프 구조는 일정한 공식에 의해서 그림 1(b) 와 같이 표준화 시킬 수가 있다. 루프 변수가 i인 루프에서 루프 변수의 초기값이 l, 시험값이 u, 그리고 증가값이 s 일 때, 표준화된 루프는 루프 변수를 k라고 할때 시험값은 $1+(u-1)/s$이고, 원래의 루프의 내부에서 발생하는 모든 루프 변수 i에 대해서는 $1+(k-1)s$로 대체하면 된다. 따라서, 루프 구조와 관련된 모든

이론은 표준화된 루프 구조를 대상으로 하여도 일반적인 루프 구조에 적용할 수가 있게 된다.

일반적인 루프 구조 여러개가 내포되어 있는 경우에도 각각을 표준화 하는 과정을 적용하면, 우리는 표준화된 루프 구조 여러개가 내포되어 있도록 변환할 수 있다. 이후에 사용하는 모든 루프 구조는 그림 1(c)와 같이 이미 표준화 되어 있는 형태만을 다루기로 한다.

## 2. 데이타 종속성

하나의 프로그램이 수행되는 과정은 일련의 문장들이 정해진 순서(control flow)에 따라 작업을 수행하는 것이다. 각 문장에서는 상수 또는 이미 정의된 변수들을 사용하여 새로운 값을 계산하여 변수에 저장하고(정의), 그 값은 다음에 수행되는 문장에 의해서 다시 사용되기도 한다(사용). 변수의 정의와 사용 관계에 따라, 몇 문장들은 동시 수행이 가능할 수 있으나, 몇 문장들은 반드시 순서에 입각해서 수행되어야만 하는 경우도 있다. 후자와 같이 반드시 순서대로 수행해야만 하도록 만드는 성질들을 종속성이라고 하며, 특히 변수의 정의와 사용과 관계된 종속성을 데이타 종속성이라고 한다.

### 2.1. 데이타 종속성의 유형

데이타 종속성은 두개의 문장사이에서 동일한 변수에 대해 값을 정의하거나 사용하는 순서에 따라 다음의 세가지 유형으로 분류할 수 있다[20].

1) 흐름 종속 (flow dependence)
2) 출력 종속 (output dependence)
3) 역 종속 (anti dependence)

흐름 종속은 그림 2(a)에서 δ로 표시된 것이며, 두 문장 사이의 종속관계를 ($S_1$ δ $S_2$), ($S_3$ δ $S_4$)와 같이 표현한다. 이는 먼저 한 문장에서 변수의 값을 정의하고 이후에 실행되어질 문장에서 그 변수를 사용하는 경우이다. 출력 종속은 ($S_1$ $\delta^o$ $S_3$)이며, 두 문장이 모두 순서에 따라 차례대로 동일한 변수를 정의하는 경우이다. 역 종속성은 ($S_2$ $\delta^a$ $S_3$)이며, 흐름 종속과는 반대로 변수값을

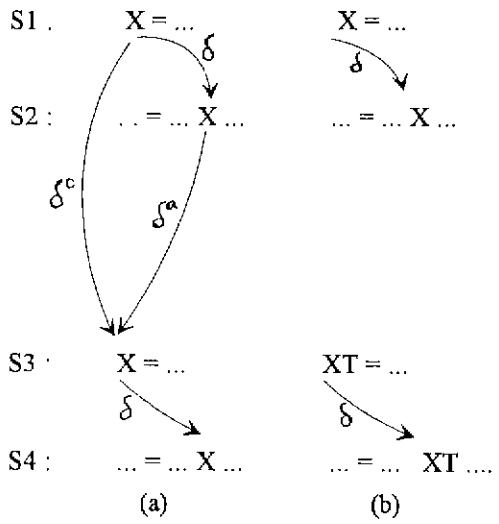


그림 2 데이타 종속성 관계

정의하기 전에 먼저 사용하는 경우이다. 이중에서 흐름 종속은 참 종속 (true dependence)이라고 하고 다른 두가지는 인위 종속 (artificial dependence)이라고 한다. 그 이유는 출력 종속과 역 종속은 그림 2(b)와 같이 프로그램을 변환함으로써 제거할 수 있으나, 흐름 종속은 제거할 수 없기 때문이다.

### 2.2. 종속관계의 표현

종속관계를 나타내기 위한 표현방법은 distance vector[9]와 direction vector[37]가 주로 사용된다. Distance vector는 두 문장사이에 존재하는 종속관계가 iteration space 상에서 얼만큼 멀리 떨어져 있나를 나타낸다. 그림 3을 예로들면, 하나의 종속관계는 루프 변수 i와 j의 어느 방향으로도 차이가 없으므로 (0,0)이고, 다른 하나는 i 방향으로는 차이가 없으나 j 방향으로는 1만큼 차이가 있으므로 (0,1)이 된다. Direction vector는 distance vector를 구성하는 각 요소의 부호를 사용하여 +, −, 그리고 0 등으로 표현하기도 하며 또는 〈, 〉 및 = 등으로 표현한다. 본 고에서는 후자의 표현방법을 이용하여 direction vector를 나타내기로 한다.

Direction vector를 구하기 위해서 반드시 distance vector를 먼저 알아야 하는 것은 아니다.

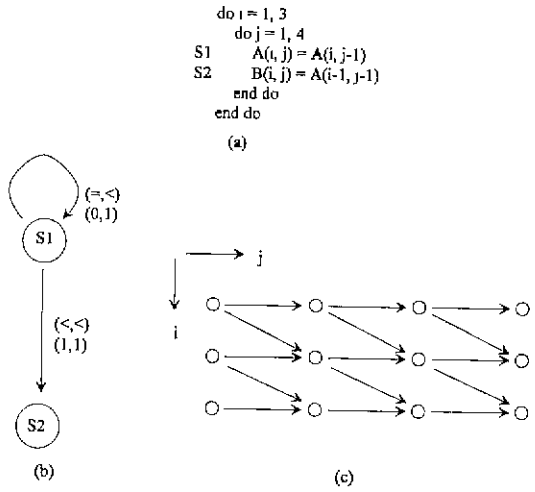


그림 3 종속관계 표현 방법

많은 경우에 distance vector를 구하는 것이 매우 어려우나, direction vector는 비교적 용이하게 구할 수 있다. 또한 활용면에서 보면, distance vector는 주로 캐쉬(cache)나 개별메모리(local memory)의 효율 증대에 효과적으로 이용될 수 있으나, direction vector는 병렬화의 조건을 파악하는 데 주로 이용된다[13]. 예를 들어서 그림 3(a)와 같은 이중으로 내포된 루프 구조를 대상으로 살펴보자. 루프 변수 i가 1, 2, 3으로 변하면서 그 내부의 문장을 실행하므로, 각각의 i값에 따라 변수 j는 1부터 1씩 증가하여 4까지 변하면서 그 내부 문장을 실행한다. 따라서 이중 루프의 내부가 실행되는 어느 한 순간에는 i와 j가 각각 [1,3]과 [1,4] 범위에서 어느 한 정수값을 가지고 있다. 정수형 영역에서 Cartesian product [1,3] [1,4]를 iteration space라고 하며 그림 3(c)와 같이 12개의 점으로 표현한다.

주어진 루프 구조에서의 데이타 종속관계를 살펴보면, 두 개의 종속관계가 존재함을 알 수 있다. 그 하나는 $S_1$에서 $S_1$으로의 종속관계로서, $i=\alpha$, $j=\beta$일 때 정의한 배열 요소를 $i=\alpha$, $j=\beta+1$일 때 사용하고 있고, 다른 하나는 $S_1$에서 $S_2$로의 종속관계로서, $i=\alpha$, $j=\beta$일 때 정의한 배열 요소를 $i+\alpha+1$, $j+\beta+1$일 때 사용하고 있다. 이와같이 데이타 종속관계가 서로 다른 루프 변수 값 사이에서 발생하는 경우의 종속성을 loop-carried dependence라고 한다.

데이타 종속성을 distance vector나 direction vector로 표현하는 것은 데이타 종속성과 관련된 여러가지 분석을 수행할 때 사용되나, 이를 그림으로 표현하여 데이타 종속관계를 쉽게 이해할 수 있도록 하기도 한다. 그림 3(b)는 루프 내의 문장을 node로 하고, 문장 사이에 존재하는 종속관계를 방향이 있는 arc로 하는 digraph를 그려서 데이타 종속관계를 나타내고 있다. 각 종속관계를 나타내는 arc에는 direction/distance vector를 label로 표현한다. 이런 데이타 종속관계의 graph 표현방법은 종속관계에 있는 문장 사이의 관계를 직관적으로 표현한다. 또한, 그림 3(c)는 단지 데이타 종속관계의 distance vector를 iteration space와 함께 표현하고 있다. 그림에서 화살표는 loop-carried dependence를 의미한다. 이런 표현 방법은 데이타 종속관계의 direction과 distance를 시각적으로 보여주고 있기 때문에 병렬화 가능성을 쉽게 판별할 수 있는 장점이 있다. 이상에서 살펴본 데이타 종속관계의 표현 방법은 각각이 나타낼 수 있는 표현력이 특징이 있으므로, 데이타 종속관계를 사용하고자 하는 목적에 따라 편리한 표현 방법을 선택하여 사용하면 효과적이다.

### 2.3 데이타 종속성의 특성

데이타 종속관계는 항상 프로그램의 실행 순서에 따라 먼저 정의/사용한 변수를 후에 정의/사용 하는 경우에 발생한다. 내포된 루프에서는 루프 변수가 변하는 관계를 이용하여 실행 순서를 알 수가 있기 때문에, distance vector나 direction vector를 이용하여 종속관계를 좀 더 자세히 기술할 수가 있었다. 이제 내포된 루프에서 존재하는 모든 종속관계에서 일반적으로 성립되는 특성에 대해서 살펴보려 한다.

그림 4는 이중 루프에서 발생 가능한 모든 종속관계를 포함하고 있는 프로그램과 각 종속관계를 direction vector로 나타낸 것이다. 그림 4(b)에서 첫번째 5개의 종속관계는 흐름 종속이지만, 나머지 4개는 역 종속관계이다. 배열의 index 함수가 달라짐에 따라 종속관계의 방향이 달라지게 된다. 또한 중요한 특성중의 하나는

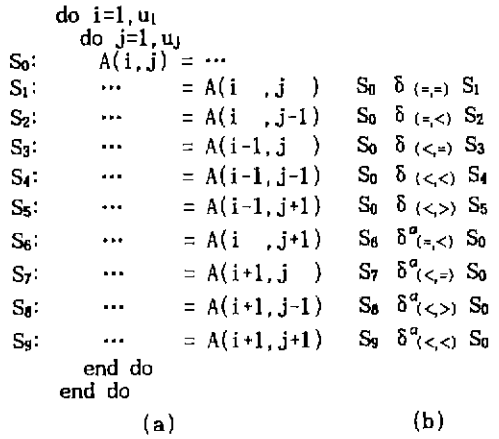


그림 4 이중 루프에서 데이타 종속관계

각각의 direction vector에서, 모든 요소가 '=' 이던가, 아니면 '='이 아닌 첫번째 요소는 반드시 '<'이어야 한다. 따라서 이와 같은 종속관계의 방향성에 관한 특성을 일반적인 다중 루프에도 확장하여 적용한다면, 모든 종속관계의 direction vector는 (=, ···, =, <, *, ···)와 같이 표현된다[36].

## 2.4 데이타 종속성 분석 방법

효과적이고도 정확한 데이타 종속성 분석은 루프 구조를 병렬화하는데 매우 중요하다. 그러나 정확한 데이타 종속성을 찾기가 쉽지 않으므로, 계속적으로 많은 연구가 진행되고 있다. 일반적으로는 동일한 변수를 포함하고 있는 두개의 문장 사이의 종속관계를 분석하는 데에는 해석적인 방법이 사용된다. 단지 index 함수가 선형(linear)이어야 한다는 제약 조건이 있다. 분석 방법의 기본 원리는 다음과 같다. 먼저 동일한 변수에 대해 사용한 두개의 index 함수가 같도록 하는 루프 변수의 정수값들이 루프 변수의 영역(range)안에서 존재하는 가를 판단한다. 여기서 우리는 배열의 dimension에 따라 연립 방정식을 얻게 되는데, 이를 dependence equation이라고 부른다. 이 dependence equation이 루프 변수의 영역 안에서 정수해를 가진다면 종속관계가 있고, 그렇지 않으면 종속관계는 없는 것이다. 예를

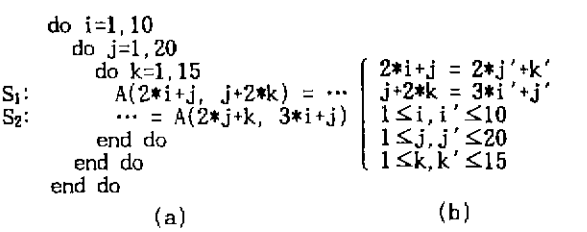


그림 5 Dependence Equation

들어서, 그림 5(a)와 같은 삼중 루프가 주어졌을 때, dependence equation은 그림 5(b) 와 같이 표현 된다.

데이타 종속성을 판단하는 가장 간단한 방법이 separability test[3,37]이다. 이 방법은 두 문장에서 사용된 동일한 변수에 대하여 사용된 두 index 함수가 공통적인 루프 변수를 한개 이하로 표현된 경우에만 적용할 수 있는 방법이다. 예를 들어서 배열 A가 한 문장에서는 A(2*i+2) 로 쓰이고, 다른 한 문장에서는 A(3*i−2)로 쓰인 경우에 적용할 수 있다. 이 방법의 특징은 index 함수에 공통적으로 나오는 루프 변수의 영역 안에서 두 문장 사이의 종속관계 유무를 정확하게 판별할 수 있다.

Gcd test[3]는 루프 변수의 영역과는 상관없이 dependence equation이 정수 해를 가지는 지의 여부를 판별하는 검사방법이다. 만약에 정수 해가 없으면 두 문장 사이에는 종속관계가 없다고 확실히 말할 수 있지만, 정수해가 있다고 판별이 되면 종속관계 유무를 확실히 말할 수가 없다. Banerjee test[3]는 루프 변수의 영역안에서 dependence equation이 실수해를 가지는 지의 여부를 판단하는 방법으로, 만약에 루프 변수의 영역 안에서 실수 해가 존재하지 않는다고 판별이 되면 종속관계는 없지만, 그렇지 않으면 확실한 판별을 할 수가 없다. 따라서 gcd test와 Banerjee test는 종속관계가 존재하기 위한 필요조건만 판별할 수 있다. 이외에도 Power test [38], I test[19,34], λ test[24], Delinearization 방법[26] 등이 좀 더 정확한 데이타 종속성 분석을 위해서 사용되기도 한다.

루프 구조에서 존재하는 데이타 종속성의 특성에 따른 병렬화 조건을 그림 6에서 살펴보기로

```
do i=1,n                    do i=1,n
  A(i) = B(i) + 3             A(i) = B(i) + 3
  B(i) = A(i) * 2             B(i) = A(i-1) * 2
end do                      end do
     (a)                         (b)
```

그림 6 병렬화 조건 비교

하자. 그림 6(a)와 같이 주어진 프로그램에서 루프 변수 i가 루프 영역 내에서 취하는 각 값에 따라 하나의 task가 생성된다고 하고 이들을 $T_1$,…,$T_n$라고 하자. 그러면 각각의 task는 서로 영향을 미치지 않아 병렬화 할 수가 있다. 반면에, 그림 6(b)의 프로그램에서, $T_k$의 첫번째 문장에서 정의된 변수 A(k)를 $T_{k+1}$의 두번째 문장에서 A((k+1)−1)로 사용하기 때문에 $T_k$와 $T_{k-1}$은 병렬수행을 할 수가 없으므로, 모든 task들을 병렬화 할 수가 없다. 한마디로 표현하면, 루프 구조의 병렬 수행 가능성은 loop-carried dependence가 없어야만 한다[37].

## 3. 루프 구조의 변환

우리가 많이 사용하는 병렬화 (parallelization)라는 용어는 보통 넓은 의미로 쓰이고 있다. 병렬 프로그램을 작성하는 경우, 병렬 컴퓨터의 특성에 따라 fine grain 또는 coarse grain parallelism을 찾아 프로그램을 작성하게 되는데, 두 경우 모두에게 넓은 의미의 병렬화라는 용어를 사용한다. 그러나 두가지 경우를 세분하여 부르는 용어가 있는데, fine grain parallelism을 찾아 병렬화하는 것을 벡터화(vectorization)라 하고, coarse grain parallelism을 찾아 병렬화하는 것을 좁은 의미의 병렬화(parallelization)라고 부른다 (concurrentization이라고 부르는 논문도 있다). 넓은 의미 혹은 좁은 의미의 병렬화는 보통 그 용어가 사용된 앞뒤 문맥을 살펴보면 그 사용 의미를 분명히 알 수가 있다. 이 장은 주어진 프로그램의 병렬화를 위해 사용되는 여러가지 방법에 대해 설명을 하고 있다. 주어진 프로그램을 병렬 프로그램으로 변환하는 과정에서, 벡터화가 가능한 문장은 벡터 문장을 사용하여 변환할 수 있고, 병렬화가 가능한 문장은 병렬 프로그래밍 언어가 제공하고 있는 병렬 구조를 사용하여 변환할 수 있다. 병렬 루프 구조의 형태는 프로그래밍 언어에 따라 다양하게 표현될 수 있으나, 본 고에서는 많은 프로그래밍 언어가 제공하고 있는 doall 구조를 사용하고자 한다[15,27].

### 3.1 병렬화(Parallelization)

순차적으로 수행하는 루프 구조는 그 안에 존재하는 데이타 종속관계가 병렬화 조건을 만족하면 병렬 루프로 바꾸어 주는 기능이 있다. 그림 7(b)와 그림 7(c)는 그림 7(a)의 순차 루프를 병렬화 한 루프를 보여주고 있다. 이 중으로 내포된 루프 안의 문장에는 어떠한 종속관계도 없으므로 모든 do 변수에 대한 수행을 병렬화할 수 있다. 그림 7(b)는 coarse grain parallelism을 찾아 병렬화한 것이고, 그림 7(c)는 fine grain parallelism을 찾아 병렬화한 것이다.

```
do i=1,n
  do j=1,m
    A(i,j) = B(i,j) + C(i,j)
  end do
end do
          (a)

doall i=1,n                          do j=1,m
  do j=1,m                             doall i=1,n
    A(i,j) = B(i,j) + C(i,j)             A(i,j) = B(i,j) + C(i,j)
  end do                               end doall
end doall                            end do
          (b)                                (c)
```

그림 7 병렬화 변환

### 3.2 루프 교환(Loop Interchanging)

다중으로 내포된 루프 구조에서, 데이타 종속관계가 2.3절에서 설명하고 있는 특성만 유지된다면 루프의 순서를 교환해도 결과는 영향을 받지 않는다[37]. 예를 들어서, 그림 8(a)의 루프 구조에서 데이타 종속관계를 iteration

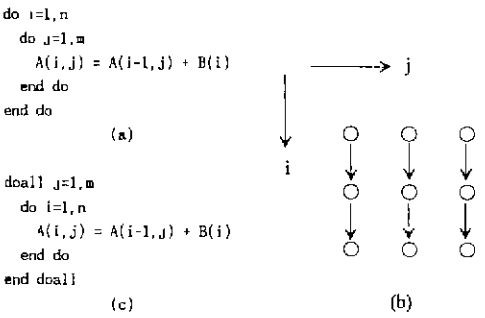


그림 8 루프 교환에 의한 병렬화

space로 표현하면 그림 8(b)와 같다. 여기에서, i 루프와 j 루프를 교환해도 데이타 종속관계가 그대로 유지되므로 두개의 루프를 교환할 수 있다. 그리고 종속관계는 i 루프 사이에는 loop-carried 종속관계가 있으나, j 루프 사이에는 loop-carried 종속관계가 없으므로, j 루프를 병렬화할 수 있다. 만약 벡터화가 목적이라면, 원래의 루프 구조에서, 안 쪽에 있는 j 루프만 벡터화하면 된다. 그런데 병렬화가 목적이라면, 그림 8(c)와 같이 두 루프를 교환한 후 밖으로 옮겨진 j 루프를 병렬화 하면 된다[16,37].

### 3.3 루프 나누기(Loop Distribution)

루프 나누기는 루프 안에 있는 여러개의 문장을 데이타 종속관계에 따라 좀더 작은 단위의 그룹으로 나누어 두 개 이상의 루프로 분할하는 것으로, 그 방법은 다음과 같다. 먼저 데이타 종속관계 그래프를 그린 후 strongly connected component 들을 찾고, 하나의 component는 하나의 루프가 되도록 한다. 그리고 각 component 사이에 존재하는 종속관계에 따라 해당 루프들이 순서대로 코딩될 수 있도록 한다. 예를 들면, 그림 9(a)는 프로그램과 그의 종속관계 그래프를 보여주고 있다. 종속관계 그래프에서 strongly connected component는 $\{S_1, S_3\}$와 $\{S_2\}$이다. 따라서 두개의 루프로 나누고, 벡터화가 가능한 두번째 루프를 벡터화 한 프로그램이 그림 9(b)에 주어져 있다. 이 변환은 loop fission 또는 fission by

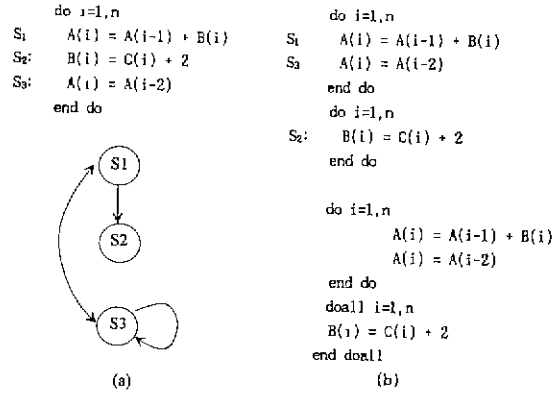


그림 9 루프 나누기와 벡터화 변환

name이라고도 부르는데, 이는 같은 변수 (또는 배열) 이름을 가지고 있는 문장들은 같은 component 에 속하게 되기 때문이다. 또한 하나의 배열은 기억장소에서 연속된 장소를 사용할 가능성이 매우 높기 때문에 캐쉬와 같은 기억장치의 효율도 높일 수 있다[2].

### 3.4 루프 융합(Loop Fusion)

루프 융합은 coarse grain parallelism이 목적인 변환에서 이용된다[25]. 이 방법은 이웃하는 루프를 하나의 루프로 융합시키는 것으로, 루프 변수의 영역이 동일해야 하며, 융합했을 때 데이타 종속관계가 변하지 않아야 한다.

```
do i=1,n
  A(i) = B(i) + C(i)
end do
do i=1,n
  D(i) = C(i-1)
end do
```

(a)

```
doall i=1,n
  A(i) = B(i) + C(i)
  D(i) = C(i-1)
end doall
```

(b)

그림 10 루프의 융합

예를 들면, 그림 10(a)에 주어진 프로그램은 그림 10(b)와 같이 하나의 루프로 융합시킨 후 병렬화할 수 있다. 이 변환은 융합한 후 병렬화가 가능하지 않아도 iteration overhead를

줄이는 변환으로서 순차 컴퓨터를 위한 프로그램에서도 프로그램의 효율을 높이기 위한 방법으로도 이용된다.

### 3.5 Scalar 확장(Expansion)

루프 안에 scalar 변수가 사용되면 항상 loop-carried dependence가 존재하게 된다. 그림 11 (a)에서 사용된 변수 X와 관련된 종속관계를 보면 흐름, 출력, 역 종속관계가 모두 존재한다. 그러나 흐른 종속을 제외한 출력 종속과 역 종속관계는 프로그램 수행상 아무런 의미가 없는 관계이다. 그래서 루프 안에서 사용된 scalar 변수대신에 루프의 영역만한 크기의 temporary 배열을 정의해서 사용하면 출력 종속과 역 종속관계를 없앨 수 있어서 병렬화 가능성을 높여준다. 그러나, 변환된 프로그램의 마지막에는 반드시 원래의 scalar 변수에 최종값을 정의해 주어서, 이후에 출현하는 동일한 scalar 변수가 옳바른 종속관계를 유지할 수 있도록 하여야 한다[22,37]. 변환된 결과는 그림 11(b)와 같다.

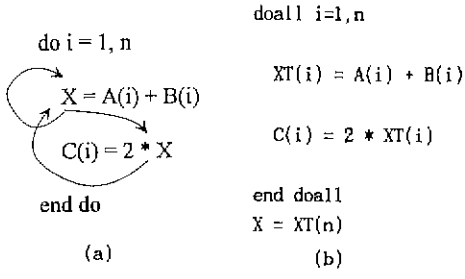


그림 11 Scalar Expansion

### 3.6 Strip Mining(Loop Sectioning)

이 변환은 기억 장치의 효율이나, 벡터 processor를 가지고 있고 한번에 처리할 수 있는 벡터의 최대 크기가 하드웨어로 고정되어 있는 컴퓨터를 위한 것으로서, 하나의 루프를 이중으로 내포된 루프 구조로 변환하는 것이다[25]. 변환된 루프의 안쪽 루프에서는 기억장치의 한 블럭 단위로 데이타를 처리할 수 있

```
do i=1,n                    do k1=1,n,64
  A(i) = B(i) + 2             k2 = min(k1+63,n)
  C(i) = B(i) * 2             doall i=k1,k2
end do                          A(i) = B(i) + 2
                                C(i) = B(i) * 2
                              end doall
                            end do
(a)                         (b)
```

그림 12 Strip Mining 변환

도록 변환한다. 이 변환은 하드웨어의 특성과 매우 밀접한 관계를 가지고 있다. 예를 들어서, cache의 블럭이 64 word이거나, 벡터 프로세서의 크기가 64 word인 경우, 그림 12(b)와 같은 변환이 효과적이다. 먼저 64 word씩 처리할 수 있도록 이중으로 내포된 루프를 만든 후, 안 쪽의 doall 루프를 벡터화 한다.

### 3.7 Vertical Spreading

이 변환은 병렬 작업(task)의 스케쥴링 부하를 줄이기 위한 것으로서, strip mining과 마찬가지로 하나의 루프 구조를 이중으로 내포된 루프로 변환하는 것이다[39]. 변환된 루프의 바깥 쪽 루프는 프로세서의 갯수와 같은 갯수의 병렬 작업를 생성하는 역할을 하도록 하는 것이다. 예를 들면, n개 $(T_1,\cdots,T_n)$의 병렬작업이 p개의 프로세서 $(P_1,\cdots,P_p)$를 가지고 있는 MIMD 구조의 병렬 컴퓨터에서 수행할 때, 모든 작업들의 grain size가 거의 비슷하다면, 보통의 경우, $P_k$에는 $T_k$, $T_{p+k},\cdots$의 작업들이 할당되어 작업이 진행된다. 이와같은 스케쥴 방법을 horizontal spreading이라 부르는데. 스케쥴링 부하가 매우 크다. 이러한 부하를 줄이는 한 방법으로 작업의 grain size를 가능한한 크게 만들어 준다. 각 프로세서에 n/p개의 작업을 묶어서 하나의 작업으로 만들어, $P_k$에는 $\{T_{(k-1)*n/p+1},\cdots,T_{kn/p}\}$를 할당하는 방법으로 프로그램을 작성하면 부하를 최소한으로 줄일 수 있게 된다. 그림 13(b)는 그림 13(a)에 이 방법을 적용하여 변환한 결과이다.

```
doall i=1,n                doall k=1,p
  ...                        do i=(k-1)*ceil(n/p)+1, min(k*ceil(n/p),n)
end doall                      ...
                             end do
                           end doall
```

(a) (b)

그림 13 Vertical Spreading

### 3.8 Synchronization

루프 안에서 loop-carried dependence가 존재할 경우에는 병렬화가 가능하지 않다. 그러나 경우에 따라서는 semaphore와 같은 것을 이용하여 종속관계를 synchronization 시켜줌으로써 제한적인 병렬 수행을 할 수 있도록 한다. 이 경우에는 loop-carried dependence가 있으므로, 루프의 iteration 들은 반드시 루프 변수가 증가하는 방향으로 스케쥴 되어야 한다[12]. 그림 14(a)에 주어진 프로그램은 배열 A와 관련된 loop-carried dependence가 존재한다. 그림 14(b)와 같이 semaphore를 이용한 signal/wait를 사용함으로써 종속관계를 유지하면서 부분적인 병렬처리가 가능해 진다.

```
do i=2,n                   doacross i=2,n
  A(i) = B(i)                A(i) = B(i)
  C(i) = A(i-2)              signal(i)
end do                       if (i≥4) wait(i-2)
                             C(i) = A(i-2)
                           end doacross
```

(a) (b)

그림 14 종속관계 Synchronization

### 3.9 Cycle Shrinking

데이타 종속관계를 그래프로 표현했을 때 cycle이 존재하면, 병렬화하는 데 많은 제약이 따른다. Cycle을 이루는 종속관계 중에서 역종속성이나 출력 종속성이 있다면 이들을 제거하는 변환을 먼저 수행하여 cycle을 없앤 후에 병렬화를 수행할 수가 있다. 그러나 cycle을 이루는 종속관계가 단지 흐름 종속만으로 이루어져 있다면 전체적으로 병렬화 하는 것이 불가능하다. 이러한 경우에는 부분적으로 병렬수행이 가능하도록 루프 구조를 변환할 수 있는데, 이런 방법 중의 하나가 cycle shrinking이다[32]. 단, 이 방법을 적용하기 위해서는 모든 데이타 종속관계의 종속거리가 1보다 커야만 한다.

먼저 가장 간단한 경우로, 루프가 하나로 구성된 단일 루프를 살펴 보기로 하자. 단일 루프의 경우에는, 두개의 완전 내포된 루프로 변환하게 된다. 종속 그래프에서 cycle을 이루는 종속관계들의 종속거리(dependence distance) 중에서 최소값이 바깥쪽 루프의 증가값(stride)이 되고, 안쪽 루프는 최소의 종속거리 만큼의 병렬 작업을 수행하는 병렬 루프로 변환된다. 그러면 변환된 루프는 최소의 종속거리 값 만큼 실행시에 효율이 증가하게 되는데, 이때의 증가율을 reduction factor(λ)라 한다. 그림 15(a)의 루프는 cycle을 이루는 두 개의 흐름종속이 존재하고, 종속거리는 10과 15이다. 따라서 λ는 10이므로 그림 15(b)와 같이 병렬화될 수 있다.

```
do i = 1, N                do i = 1, N, 10
  A(i+10) = B(i) - 1         doall j = i, Min(N,i+9)
  B(i+15) = A(i) + C(i)        A(i+10) = B(i) - 1
end do                         B(i+15) = A(i) + C(i)
                             end doall
                           end do
```

(a) (b)

그림 15 단일 루프에서의 Cycle shrinking

다중 내포된 루프의 경우에 적용할 수 있는 cycle shrinking에는 다음과 같은 세가지 방법이 있다.

1) Simple shrinking
2) Selective shrinking
3) True Dependence shringking(TD shrinking)

그림 16의 루프 구조를 사용하여 열거한 세가지 방법을 설명하기로 한다.

Simple shrinking 방법은 내포된 각 루프에 대해서 독립적으로 종속 그래프를 고려한다. 즉, 그림 16에 주어진 루프 구조에서, 각 i 루

```
do i = 3, N1
  do j = 5, N2
    A(i, j) = B(i-3, j-5)
    B(i, j) = A(i-2, j-4)
  end do
end do
```

**그림 16 Cycle shrinking을 위한 루프 구조**

프와 j 루프에 대해서 종속 그래프를 고려한다. 루프 i에 대해서는, 종속거리가 2와 3이므로 최소값인 2가 reduction factor이고, 루프 j에 대해서는 종속거리가 4와 5이므로 최소값인 4가 reduction factor이다. 따라서, 각 루프에 대해서 단일 루프에서와 같이 변환하면 그림 17이 된다. 이런 변환은 iteration space에 표시한 것과 같이, 점선으로 구획되어 있는 각 구역 안에 있는 iteration들 사이에 종속관계가 없으므로 이들을 병렬 수행할 수가 있다. 이 때의 reduction factor는 2*4가 된다.

Selective Shrinking 방법은 simple shrinking의 경우와 같이, 내포된 각 루프에 대해서 독립적으로 종속 그래프를 고려하여, 각 루프에 대하여 각각의 reduction factor를 구한다. 그리고 바깥쪽 루프로부터 안쪽 방향으로, 처음으로 1보다 큰 reduction factor를 가진 루프를 찾아서, 그 루프를 simple shrinking 방법을 적용하고, 그 안쪽에 있는 모든 루프를 병렬화한다. 그림 16에 주어진 루프 구조에서는, i 루프의 reduction factor가 2이므로 i 루프를 simple shrink 방법으로 변환하고, j 루프는 doall로 변환하여, 그림 18를 얻게 된다. 이런 변환도 역시, iteration space에 표시한 것과 같이, 점선으로 구획되어 있는 각 구역 안에 있는 iteration들 사이에 종속관계가 없으므로 이들을 병렬 수행할 수가 있다. 이 때의 reduction factor는 $2*(N_2-4)$가 된다.

True Dependence shringking(TD shrinking) 방법은 내포된 루프 구조에 존재하는 데이타 종속관계의 종속거리를 iteration space를 1차원적으로 생각했을 때의 거리라고 간주하여, 병렬수행할 수 있는 iteration의 갯수를 최대한도로 하는 방법이다. 즉, 다중 내포된 루프를 마치 단일 루프처럼 생각하고 cycle shrinking을 적용한다. 그림 16에서 주어진 루프 구조에서 보면, 두개의 데이타 종속관계가 존재하

```
do i1 = 3, N1, 2
  do j1 = 5, N2, 4
    doall i = i1, i1 + 1
      doall j = j1, j1 + 3
        A(i, j) = B(i-3, j-5)
        B(i, j) = A(i-2, j-4)
      end doall
    end doall
  end do
end do
```

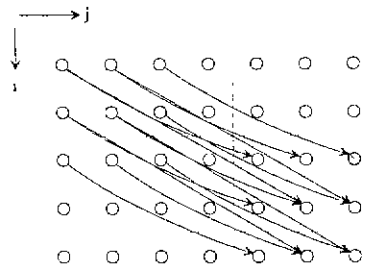


**그림 17 Simple shrinking 방법**

```
do i1 = 3, N1, 2
  doall i = i1, i1 + 1
    doall j = 5, N2
      A(i, j) = B(i-3, j-5)
      B(i, j) = A(i-2, j-4)
    end doall
  end doall
end do
```

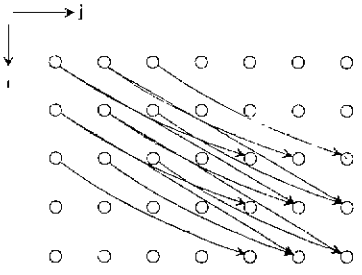


**그림 18 Selective shrinking 방법**

```
N = (N1-3)+1;  M = (N2-5)+1
λ = 2*(N2-4)+4
do K = 1, NM, λ
  T1 = ((K-1) DIV M) + 3
  T2 = ((K+λ-2) DIV M) + 3
  T3 = ((K-1) MOD M) + 5
  T4 = ((K+λ-2) MOD M) + 5
  doall i = T1, T2
    IF i ◇ T1 THEN L1 = 5
    ELSE L1 = T3
    IF i ◇ T2 THEN L2 = N2
    ELSE L2 = T4
    doall j = L1, L2
      A(i, j) = B(i-3, j-5)
      B(i, j) = A(i-2, j-4)
    end doall
  end doall
end do
```

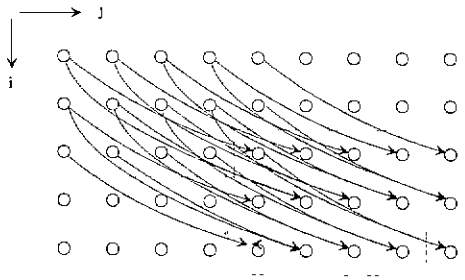


**그림 19 TD shrinking 방법**

고 각각의 종속거리는 (2,4)와 (3,5)이다. 이 종속거리를 1차원적으로 생각하면, $2*(N_2-4)+4$와 $3*(N_2-4)+5$이다. 따라서 reduction factor는 $2*(N_2-4)+4$가 되고, 그림 19의 iteration space에 표시한 것과 같이, 점선으로 구획되어 있는 각 구역 안에 있는 iteration들 사이에 종속관계가 없으므로 이들을 병렬 수행할 수가 있다. 이런 변환에서는 i 루프의 i값에 따라 병렬 수행되는 j값의 범위가 다르게 되므로, 이의 처리가 용이하지가 않다. 그림 16을 이 방법으로 변환한 것이 그림 19이다.

## 4. 결 론

본 고에서는 루프 구조에서 하드웨어와 무관한 데이타 종속관계 분석과 병렬화를 위한 루프 구조의 변환에 대해서 살펴 보았다. 종속성 관계 분석에서는 좀 더 정확하게 종속성을 분석할 수 있는 방법이 필요하다. 요즘은 데이타 종속성 분석을 위해서 행렬을 사용하여 수학적 이론을 동원하여 정확한 종속성 분석을 위해 꾸준히 연구가 진행되고 있다. 따라서 데이타 종속성에 관한 연구를 위해서는 이제 수학적 배경을 많이 필요로 하고 있다. 루프 구조의 변환에 대하여 다룬 부분은 매우 중요하고 가장 일반적인 변환 방법만을 거론하였다. 본 고에서 다루지 않은 많은 변환 방법들이 이미 발표되었고 앞으로도 꾸준히 새로운 변환 방법들이 발표되리라고 본다. 그 이유는 종속 관계 분석에 관한 연구가 진행될수록 그로부터 얻을 수 있는 정보의 폭이 넓어지게 되고, 따라서 그만큼 프로그램 변환에도 폭넓게 적용할 수 있게 되기 때문이다. 이제 병렬 컴퓨터가 차츰 대중화되어가고 있는 추세에 있으므로, 이를 효과적으로 사용하기 위한 연구가 활발히 이루어져서, 귀중한 프로세서 자원을 낭비하지 않도록 해야할 것이다.

## 참고문헌

[1] Abu-Sufah, W., Kuck, D. J., and Lawrie, D. H., "On the Performance Enhancement of Paging Systems through Program Analysis ans Transformations." *IEEE Transactions on Computers*, C-30, 5 (May 1981), 341~356.

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

[3] Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J., "An Overview of the PTRAN Analysis System for Multiprocessing." *Journal of Parallel and Distributed Computing*, 5, 5, (Oct. 1988), 617~640.

[4] Allen, J. R., and Kennedy, K. "A Parallel Programming Environment." *IEEE Software* 2, 4 (July 1985), 21~29.

[5] Allen, J. R., and Kennedy, K. "Automatic Tran-

slation of Fortran Programs to Vector Form" *ACM Transactions on Programming Languages and Systems* 9, 4 (Oct. 1987), 491~542.

[6] Alliant Computer Systems Corp. *CONCENTRIX C Handbook*, Acton, MA, Aug. 1986.

[7] Alliant Computer Systems Corp. *FX/Fortran Language Manual*, Acton, MA, Aug. 1986.

[8] Appelbe, B. Smith, K., and McDowell, C. "Start/Pat: A Parallel-Programming Toolkit." *IEEE Software*, Jul. 1989, 29~38.

[9] Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academy Publishers, 1988.

[10] BBN Advanced Computers Inc. *Programming in Fortran with the Uniform System*. Cambridge, MA.

[11] Cray Research Inc. *CF77 Compiling System, Parallel Processing Guide*.

[12] Cytron, R., "Doacross: Beyond Vectorization for Multiprocessors." *Proceedings of 1986 International Conference on Parallel Processing*, Aug. 1986, 836~844.

[13] Gannon. D., Jalby, W., and Gallivan, K. "Strategies for Cache and Local Memory Management by Global Program Transformation." *Parallel and Distributed Computing* 5, 5 (Oct. 1988), 587~616.

[14] Guarna, V. Jr., Gannon, D., *et. al.*, "Faust: An Integrated Environment for Parallel Programming." *IEEE Software*, Jul. 1989, 20~27.

[15] Guzzi, M. *Cedar Fortran Reference Manual*. Tech. Rep. 601, Center for Supercomputer Research and Development, University of Illinois, Urbana, IL, Nov. 1986.

[16] Huson, C., Macke, T., Davies, J., Wolfe, M., and Leasure, B. "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer." *International Conference on Parallel Processing*, Aug. 1986, IEEE, pp. 827~832.

[17] Karp, A. H., and Babb, R. G. "A Comparison of 12 Parallel Fortran Dialects." *IEEE Software*, Sep. 1988, pp. 52~67.

[18] Knuth. D. E. "An Empirical Study of Fortran Programs." *Software Practice and Experience*, 1, 1971, 105~133

[19] Kong, X., Klappholz, D., and Psarris. K. "The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization." *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, Jul. 1991.

[20] Kuck, D. J. *The Structure of Computers and Computations*, vol. 1. John Wiley and Sons, New York, 1978.

[21] Kuck, D. J., Davidson, E. S., Lawrie, D. H., and Sameh, A. H. "Parallel Supercomputing Today and the CEDAR Approach." *Science*, 231, 967~974.

[22] Kuck, D. J., Kuhn, R., Leasure, B., and Wolfe, M., "The Structure of an Advanced Retargetable Vectorizer." *In Supercomputers: Design and Applications Tutorials* (Hwang, K., ed.), IEEE Society Press, Silver Spring MD, 967~974.

[23] Kuck, D. J., Kuhn, R., Padua, D., Leasure, B., and Wolfe, M. "Dependence Graphs and Compiler Optimizations." *Conference Record of the 8th Annual ACM Symposium on Principles of Programming languages*, Jan. 1981, ACM, pp. 207~218.

[24] Li, Z., Yew, P. C., and Zhu, C. Q. "An Efficient Data Dependence Analysis for Parallelizing Compilers." *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No 1, Jan. 1990.

[25] Loveman, D. "Program improvement by Source-to-Source Translation." *Journal of the ACM* 20, 1 (Jan. 1977), 121~145.

[26] Maslov, V. "Delinearization: an Efficient Way to Break Multiloop Dependence Equations." *ACM SIGPLAN*, Jun 1992.

[27] Mehrotra, P., and Van Rosendale, J. *The BLAZE Language: A Parallel Language for Scientific Programming*. Tech. Rep. 85~29, ICASE, NASA LAngley Research Center, Hampton. VA, May 1985.

[28] Ottenstein, K. J., and Ottenstein, L. M. "The Program Dependence Graph in a Software Development Environment." *SIGPLAN*, May 19 84, 177~184.

[29] Padua, D. A., and Wolfe, M. J. "Advanced Compiler Optimizations for Supercomputers." *Communications of the ACM* 29, 12 (Dec. 1986), 1184~1201.

[30] Perrott, R. H. "Parallel Languages and Parallel

Programming." *Parallel Computing* 89, 1990, 47~58.

[31] Perrott, R. H., and Zarea-Aliabadi, A. "A Supercomputer Program Development System." *Software-Practice and Experience* 17, 10 (Oct. 1987), 663~683.

[32] Polychronopoulos, C. D. "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design." *IEEE Transactions on Computers*, Vol. **37**, No. 8, Aug. 1998.

[33] Polychronopoulos, C. D., Girkar, M. B., *et. al.*, "The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran." *In Languages and Compilers for Parallel Computing*, Glenter, D., Nicolau, A., and Padua, D. Editors, MIT Press, 1990.

[34] Psarris, K., Kong, X., and Klappholz, D. "The DIrection Vector I Test." *IEEE Transactions on Parallel and Distributed Systems*, Vol. **4**, No. 11, Nov. 1993.

[35] Smith, B. "Parallel Computing Forum." *In Proceedings of the IFIP WG 2.5 Working Conference* (Amsterdam, Netherlands, Aug. 1988), M. Wright, Ed., North-Holland, p. 235.

[36] Wolf, M. E., and Lam, M. S. *An Algorithm to Generate Sequential and Parallel Code with Improved Data Locality*, Computer Systems Laboratory. Stanford University.

[37] Wolfe, M. *Optimizing Supercompilers for Supercomputers*, PhD Thesis, University of Illinois at Urbana-Champaign, Urbana, Ill, Oct. 1982.

[38] Wolfe, M., and Tseng, C. W. *The Power Test for Data Dependence.~ IEEE Transactions on Parallel and Distributed Systems*, Vol. **3**, No. 5, Sep. 1992.

[39] Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers*, Addison-Wesley, 1991.

**이 만 호**

1971 ~1975 서울대학교 공과대학 응용수학과, 학사
1975 ~1977 한국과학기술원 전산학과, 석사
1977 ~1980 국방과학연구소, 연구원
1980 ~1982 충남대학교 계산통계학과, 전임강사
1982 ~1984 충남대학교 계산통계학과, 조교수
1984 ~1991 인디아나 대학교, 박사
1991 ~1994 충남대학교 전산학과, 조교수
1994 현재 충남대학교 전산학과, 부교수
관심 분야 : 병렬 프로그래밍, 병렬 컴파일러, 병렬 알고리즘, 프로그래밍 언어