□ 기술해설 □

병렬화 컴파일러의 소개

한국과학기술원 안준선* · 최광훈* · 김성훈** · 한태숙*** · 최광무***

● 목 차 ●

1. 서 론

병렬 컴퓨터를 위한 프로그램 작성은 순차 프로그램보다 일반적으로 훨씬 복잡한 작업이 된다. 병렬 프로그램의 작성자는 병렬로 수행이 가능한 작업들을 추출해 내고 적절한 시기에 동기화를 시켜 주어야 하며 자료의 공유로 인한 부담을 감소하기 위하여 자료의 분할이나, 중복을 고려하여야 한다. 또한 이러한 작업들은 특정 기계의 구조에 의존적인 특성을 가지고 있다.

이러한 병렬 프로그램 작성의 어려움을 극복하기 위한 두가지 접근법을 찾을 수 있다. 즉, 프로그래머가 비교적 이해하기 쉽고, 컴파일러가 효율적인 목적코드를 생성할 수 있는 병렬 언어를 사용하는 방법[6]과, 프로그래머가 작성한 순차 프로그램에 대하여 자동으로 병렬 프로그램을 생성하는 방법이 그것이다.

순차 프로그램을 병렬 프로그램으로 자동으로 바꾸어주는 프로그램을 병렬화 컴파일러(parallelizing compiler)라고 한다. 병렬화 컴파일러는 C나 Fortran과 같은 순차 프로그램 입력에 대하여 다양한 분석과 코드 변환 과정을 거쳐, 병렬 언어 프로그램이나 Message Passing Interface(MPI)[15], 병렬 쓰레드(thread)와 같은 병렬 라이브러리를 호출하는 프로그램을 생성해낸다.

이러한 병렬화 컴파일러는 다음과 같은 유용성을 가진다. 우선 이미 개발된 순차 프로그램을 새로운 병렬 컴퓨터에서 신속하게 사용할 수 있게 된다. 그리고 프로그래머가 병렬 프로그램 대신에 순차 프로그램을 작성하도록 하므로써 기존의 순차 프로그램에 익숙해져 있던 프로그래머가 병렬 컴퓨터를 위한 프로그램을 쉽게 작성할 수 있게 된다. 병렬 컴퓨터를 위한 프로그램은 실행시킬 컴퓨터 구조에 따라 병렬 수행 구조(primitive)나, 고려 사항들이 다르기 때문에, 병렬화 컴파일러를 사용하므로써 프로그램의 이식성(portability)도 높일 수 있다[38].

지금까지 병렬화 컴파일러의 연구는 메시지 교환(message passing)방식의 병렬 프로그램 생성 보다는, 공유 메모리(shared memory) 방식의 병렬 컴퓨터에서 유용한 병렬 반복문(loop)생성이나 벡터 연산의 생성을 중심으로 수행되어 왔다. 그 결과 Parafrase[3], Ptran[38], SuperB[1], Suif[23,42], Polaris[2,40],

*비 회 원
**학생회원
***종신회원

PFC[1], KAP[21] 등의 다양한 병렬화 컴파일러가 발표 되었다. 국내의 연구로서는 주전산기IV 개발 사업의 일부로서 Fortran을 위한 병렬화 컴파일러의 개발을 한국과학기술원(KAIST)에서 수행중에 있다[36, 37].

본 논문의 내용은 다음과 같다. 다음 절에서는 병렬화 컴파일러의 일반적인 구조와 중요한 분석, 변환 작업 등을 기술한다. 그리고 외국의 병렬화 컴파일러 개발 사례와 한국과학기술원에서 개발된 병렬화 컴파일러를 소개하고 결론을 맺는다.

2. 병렬화 컴파일러

2.1 병렬화 컴파일러의 구조

병렬화 컴파일러는 일반적으로 그림 1과 같은 과정을 갖는다[38]. 입력된 프로그램은 전단부를 거쳐서 분석부에서 프로그램의 최적화와 병렬 프로그램의 생성에 필요한 제어 및 자료 흐름 분석과 의존성 분석(dependence analysis)이 수행되며 이 결과를 바탕으로 목적 기계에 적합한 병렬 프로그램을 생성하게 된다.

프로그램 분석은 프로시져내 분석(intra-procedural analysis)과 프로시져간 분석(interprocedural analysis)으로 구분되며, 프로그램의 제어 흐름(control flow)과 자료 흐름(data flow)을 분석한다[1, 38]. 제어 흐름 분석은 프로그램 명령어들 사이에 수행 순서 관계와 프로시져 간의 호출 관계를 분석하며, 자료 흐름 분석은 자료의 정의 및 사용 관계와 프로시져 호출시의 자료 전달에 관한 다양한 정보를 분석한다. 의존성 분석은 배열(array) 원소의 첨자(index)를 분석하여 반복문 내의 배열 원소로 인한 의존성을 분석하는 것으로서 반복문 병렬화에 있어 가장 중요한 분석이 된다.

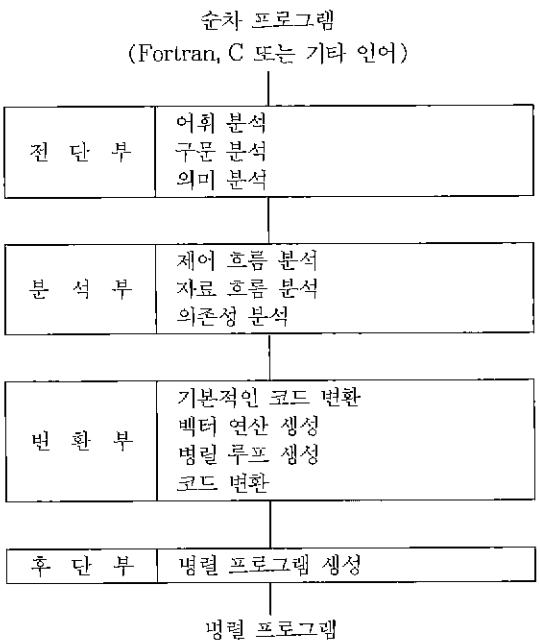


그림 1 일반적인 병렬화 컴파일러의 작업

입력 프로그램에 대한 분석이 수행되면, 병렬 프로그램 생성을 위한 코드 변환이 수행된다. 이러한 작업은 벡터 연산의 생성(vectorization), 병렬 반복문의 생성(parallelization) 등과 병렬화의 효과를 높히기 위한 다양한 프로그램 변환 작업으로 이루어진다[1, 25]. 이렇게 생성된 병렬 작업에 대하여 프로세스간의 통신이나 자료의 분할 등을 고려하여 목적 기계에서 수행 가능한 병렬 프로그램을 생성하게 된다.

이어지는 부절들에서는 병렬화 컴파일러에 있어서 기본적인 작업이 되는 프로시져내의 분석, 프로시져간의 분석, 의존성 분석, 병렬성 검출, 코드 변환에 대하여 설명한다.

2.2 프로시져내 분석

프로시져내 분석이란 한 프로시져 내에서의 자료 흐름이나 제어 흐름에 관한 정보를 분석하는 것을 말한다. 분석해 내는 정보에는 제어 흐름 정보로써 프로그램 문장 간의 우선 수행(domination) 관계, 문장간의 수행 결정 관계(control dependence) 등이 있으며, 자료 흐름 정보로서 변수값 도달 정의 탐색(reaching definition), 유용한 수식(available expression), 생존 변수(live variable), 상수 전달(constant propagation), 유도 변수(induction variable) 등이 있다.

2.2.1 유용한 정보

가. 프로그램 문장 사이의 수행 순서(control flow)

프로그램의 자료의 흐름을 알기 위해서는 일반적으로 프로그램의 제어의 흐름을 먼저 알아야 하며, 이러한 정보는 문장간의 다음과 같은 수행 순서 관계를 바탕으로 이루어진다[1].

문장간의 수행 순서 관계에는 우선 수행(dominate) 관계와 나중 수행(post dominate) 관계가 있다. 우선 수행 관계는 문장 s가 수행되면 반드시 문장 s'이 그전에 수행되는 관계를 말하며 나중 수행 관계는 문장 s가 수행되면 언제나 프로그램이 끝나기 전에 문장 s'이 수행되는 것을 말한다.

나. 변수값 도달 정의 탐색(reaching definition)

변수값 정의 탐색이란 프로그램 그레프 상의 어떤 노드에 도달하는 모든 변수의 정의를 찾아내는 것이다[25]. 이는 프로그램 그래프 상의 한 노드에서 사용되는 변수가 어디서 정의된 변수인지, 그리고 정의된 변수가 어디까지 새로 정의되지 않고 사용되는지를 알아내는 것이다. 그림 2에서 노드 G에서의 변수값 정의는 노드 B의 b, 노드 D의 c와 b, 그리고 노드 E의 a와 b이다. 이러한 정보는 스칼라 변수간의 자료의 의존성을 나타낸다고 할 수 있다.

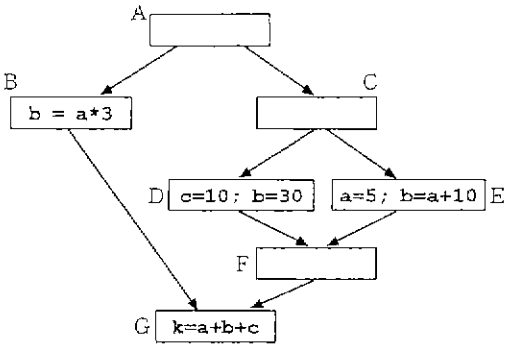


그림 2 변수값 도달 정의 탐색을 위한 예제

다. 상수 전달(constant propagation)

상수 전달은 변수가 하나의 값으로 할당되어 그 값이 변하지 않을 때 그 변수를 상수로 치환하는 것을 말한다[1].

그림 3을 보면 변수 a는 상수로 치환될 수 있음을 알 수 있다. 그리고 b의 경우는 10과 30중 어떤 값이 대입될지 알 수 없으므로 상수로 치환될 수 없다. c,d의 경우도 a+b의 값을 예측할 수 없으므로 상수로 치환될 수 없다. 이렇게 모든 프로그램 구간에서 값이 한 번 대입되어 사용되는 a와 같은 변수를 간단한 상수(simple constant)라고 한다.

하지만 그림 3을 주의깊게 살펴보면 노드 B에서는 오른쪽 방향으로만 간다는 것을 알 수 있다. 즉, 30>30은 항상 거짓이므로 수 b에는 항상 30 이 대입되어 사용된다. 그리고 이 값은 노드 E에서 다시 대입되는 a변수의 값과 같으므로 b도 상수로 치환될 수 있다. 따라서 a+b 또한 상수값이 되며 변수 d도 상수로 치환될 수 있다. 이와 같이 조건문이 항상 참이거나 거짓이어서 파생되는 상수를 조건 상수(conditional constant)라고 한다[14].

이러한 상수 전달 분석은 일반적인 컴파일러의 최적화에도 유용할 뿐 아니라 병렬화를 위한 의존성 분석의 성능에 영향을 미치게 된다.

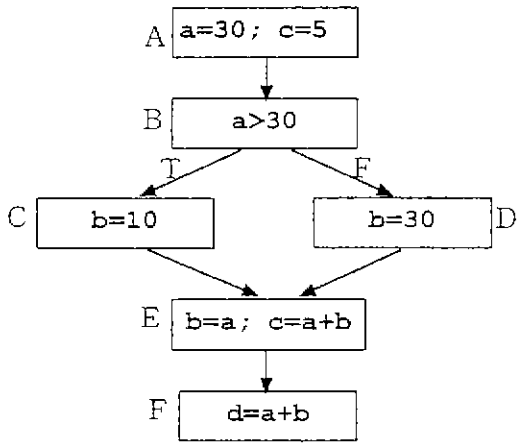


그림 3 상수 전달을 위한 예제

라. 유도 변수

유도 변수(induction variable)란 반복문 내에서 일정한 비율로 변화하는 변수를 가리킨다[17]. 일반적으로 반복문의 반복횟수를 나타내는 변수가 유도 변수일 수 있으나 여기에서 설명하고자 하는 변수는 확장된 개념으로서 반복문 내에서 일정한 비율로 변화하는 모든 변수를 가리킨다.

그림 4에서 변수 A는 명백하게 반복문을 한 번 돌 때마다 값이 1씩 증가하고 있으므로 유도변수가 된다. 그리고 변수 B의 경우도 반복

```
A=S
B=S
DO 10 I=1, 10, 1
  A=A+1
  B=B+1
  ...

10 CONTINUE
```

그림 4 유도 변수를 위한 예제

이 한번 수행될 때 마다 I 값만큼 증가하고 있으므로 유도 변수임을 알 수 있다. 이러한 유도 변수는 반복문의 병렬화에서 반복문 간의 의존성을 없애주는 중요한 정보가 된다.

2.2.2 프로시져 내의 분석을 위한 자료 구조

이러한 프로시져 내의 분석은 프로그램을 나타내는 구조를 바탕으로 단조 자료 흐름 분석 체계(monotone data flow analysis system)를 통하여 이루어 진다[1].

단조 자료 흐름 분석 체계는 자료 흐름 분석을 위한 도구로써 Fortran 프로그램과 같이 비교적 간단한 의미 구조를 가진 프로그램을 분석할 때 사용되는 방법이다. 각 분석 정보에 따라 일정한 초기값과 입력값, 선행 노드의 정보에 대한 새로운 정보의 생성 방법, 정보가 정의되는 래티스(lattice) 공간 등이 주어지고 프로그램 그래프의 각각의 노드에 대하여 계산되는 정보의 변화가 없을 때까지 반복하는 알고리즘을 수행하므로써 정보를 얻어내게 된다.

이러한 단조 자료 흐름 분석은 프로그램의 정보를 표시하는 자료구조를 필요로 하는데, 사용되는 자료 구조는 분석의 효율성에 영향을 미친다. 이러한 자료 구조로는 제어 흐름 그래프[25], 정의 사용 체인[12], SSA(Static Single Assignment) 그래프[26], 성긴 자료 흐름 그래프(sparse dataflow evaluation graph)[8], SESE(Single Entry Single Exit) 그래프[20], 점진적 자료 흐름 그래프(incremental dataflow graph)[39] 등이 제안되었다.

2.3 프로시져간 분석

프로시져간 분석은 프로시져 사이의 호출/피호출 관계에 나타나는 자료의 흐름에 관한 분석을 말한다.

일반적으로 프로시져간 분석을 하기 위해서는 먼저 각 프로시져 사이의 호출/피호출 관계를 나타내는 호출 그래프(call graph)를 작성하고, 이를 이용하여 프로시져간 자료의 흐름을 분석하게 된다. 각 프로시져 사이에 자료가 전달되는 경로는 형식인자(formal parameter)와 실인자(actual parameter) 사이의 바인딩(binding)과 전역 변수에 의해 이루어지므로, 인자들 사이의 자료 흐름을 나타낼 수 있는 바인딩 그래프를 사용하여 바인딩과 전역 변수에 의한 자료의 흐름을 분석한다[1].

2.3.1 호출 그래프(call graph)

호출 그래프는 각 프로시져간의 호출/피호출 관계를 그래프로 표시한 것으로서 노드는 각 프로시져를 나타내며 노드 사이의 화살표(directed edge)는 프로시져의 호출을 나타낸다.

프로시져간에 함수를 인자로 주고 받을 수 없다면 프로그램 상에 나타난 호출문을 그대로 호출 그래프의 화살표로 구성하기만 하면 된다. 그러나 프로시져의 인자(parameter)에 함수값이 바인딩 될 수 있다면 각각의 형식인자에 바인딩이 가능한 함수값들을 증가시켜 가면서 더 이상 변화가 없을 때까지 반복하는 알고리즘을 수행하여야 한다[1].

2.3.2 바인딩 그래프[1]

바인딩 그래프는 프로시져 인자에 의하여 전달되는 자료의 흐름을 표현하기 쉽도록 하기 위하여 인자들 사이의 바인딩 관계를 그래프로 표현한 것이다. 노드는 프로시져들의 각각의 형식인자를 나타내며, 두 형식인자 사이에 바인딩이 존재할 때 바인딩 그래프상에서 화살표를 생성한다. 호출 그래프가 생성되어 있으면, 바인딩 그래프는 호출하는 프로시져와 호출된 프로시져의 형식인자와 호출문의 실인자를 이용하여 쉽게 구성될 수 있다.

2.3.3 USE, MOD 집합[1]

프로시져 호출시에 호출된 프로시져 내에서 사용하거나 변경한 변수를 USE, MOD 집합으

로 나타내는데, 이를 통해 프로시져 호출문의 자료 의존 관계를 알 수 있다. 이러한 USE, MOD 집합을 구하지 않을 경우 함수 호출시에 인자로 사용된 변수와 모든 전역 변수가 변경, 사용되었다고 가정하여야 하므로 분석의 정확도와 효과가 떨어지게 된다. 이러한 USE, MOD 집합의 계산과 프로시져내의 자료 흐름 분석은 상호 보완적인 관계를 가진다[8].

2.3.4 이명 분석(aliasing analysis)

다른 이름을 가진 두 개의 변수가 프로그램 수행 중에 같은 메모리의 위치를 가리키게 될 때 이를 이명 관계라고 한다. 이러한 이명 관계는 두가지 형태가 있는데 정적(static) 이명과, 동적(dynamic) 이명이 그것이다. 정적 이명 관계는 서로 다른 두 개의 변수가 같은 메모리를 가리키고 있다는 것이 명시적으로 나타난 경우로 Fortran의 EQUIVALENCE 문장의 경우가 그러한 경우이다. 동적 이명은 어떤 문장의 수행의 결과로 이명 관계가 발생하는 경우로서 포인터 연산이나 프로시져 호출시의 인자로 인하여 발생한다. 포인터 연산으로 인한 이명 관계의 분석은 명확히 분석해 내기 어려운 것으로 간주되었으나, 최근 포인터 연산으로 인한 이명 관계 분석을 위한 연구가 활발히 진행되고 있다[22,24,34,35]. 여기서는 프로시져간의 분석과 명확한 연관성이 있는 프로시져 호출로 인한 이명 관계 분석에 대하여 알아보기로 한다.

프로시져 호출로 인한 이명 관계의 발생은 다음과 같은 두 경우로 나타낼 수 있다[8]. 첫째는 두 개의 다른 형식인자가 이명 관계에 있는 실인자나 동일한 실인자에 바인딩 되어있는 경우이고, 둘째는 형식인자에 전역 변수가 바인딩되어 있는 경우이다. 그림 5의 Fortran 프로그램을 살펴 보면 SUBA의 첫 번째 호출에서 SUBA의 형식 인자인 X와 Y는 모두 MAIN의 I1에 바인딩 되어 있으므로 첫 번째 경우에 해당하여, X와 Y는 이명 관계를 가지게 된다. 두 번째 호출에서는 Y는 전역 변수 G에 바인딩 되므로 두 번째 경우에 해당하여 프로시져 내에서 G와 Y는 이명 관계에 놓이게 된다.

```
PROGRAM MAIN
INTEGER I1, I2, G
COMMON G
CALL SUBA(I1, I1)
CALL SUBA(I2, G)
END

SUBROUTINE SUBA(X, Y)
COMMON G
...
END
```

그림 5 이명 관계의 예제

이러한 부프로그램의 호출로 인해 발생하는 이명 관계를 나타내기 위해서는, 각 프로시져에서 볼 수 있는 모든 변수에 대하여 각각의 프로시져 호출시에 발생될 수 있는 이명 관계를 변수에 대한 분할(partition)로 나타내게 된다. 그림 5의 예제 프로그램에서 SUBA에 대한 이명 관계의 분석 결과는 { {{X, Y}, {G}}, {{X}, {Y, G}} }가 된다.

2.4 의존성 분석(dependence analysis)

의존성 특히 자료 의존성이란 같은 메모리 장소를 사용하는 두 문장 사이의 수행 순서가 유지되어야 하는 성질을 가리킨다. 두 문장 사이에 의존성이 없을 경우에 두 문장은 병렬로 수행이 가능하게 된다.

순차 프로그램의 병렬화는 반복문의 병렬화가 성능에 많은 영향을 끼치며, 따라서 이러한 반복문 내의 문장 또는 반복 사이의 의존성을 규명하는 것이 중요한 작업이 된다. 그런데 반복문의 계산 내용을 보면 배열 원소들에 대한 동일한 연산을 수행하는 경우가 대부분이므로, 배열로 인한 의존성을 정확히 분석하는 것이 중요해진다[1,28].

이전의 자료 분석 단계에서는 배열 원소의 첨자를 고려하지 않으므로 의존성 분석에서는 배열 원소의 첨자를 고려하여 문장간의 의존성을 찾아내는 작업을 수행한다. 이러한 의존성 분석은 반복문 병렬화의 성능에 매우 큰 영향을 주는 것으로 간주되어 많은 연구가 진행되었다.

2.4.1 문제 정의

다음은 일반적인 n차원 반복문의 형태이다.

```
DO I1 = t1, u1
  DO I2 = t2, u2
    ...
    DO In = tn, un
      A[f(I1, I2, ..., In)] = ...
        ...= ... A[g(I1, I2, ..., In)] ...
    END DO In
    ...
  END DO I2
END DO I1
```

여기에서 A가 m차원 배열일 경우 f와 g는 m차원 벡터를 결과로 가지는 함수가 되며, 배열의 인덱스는 선형식으로 가정하여 $I_1, I_2, \ldots, I_n$에 대한 선형 함수로 가정한다. 이때에 반복 공간 내에서 $f(I_1, I_2, \ldots, I_n) = g(I_1', I_2', \ldots, I_n')$을 만족하는 $I_1, I_2, \ldots, I_n, I_1', I_2', \ldots, I_n'$ 이 존재할 경우에 위의 두 문장은 배열 A의 같은 원소를 접근하게 되므로 의존성이 존재하게 된다. 이러한 작업은 m개의 f와 g의 각각의 대응하는 벡터 원소 선형 함수에 대하여 다음을 만족하는 $I_1, I_2, \ldots, I_n, I_1', I_2', \ldots, I_n'$ 이 존재하는지를 검사하는 작업이 된다[1].

$$a_0 + \sum_{i=1}^{n} a_i \cdot I_i - (b_0 + \sum_{i=1}^{n} b_i \cdot I_i') = 0$$

$$(u_i \le I_i, I_i' \le t_i)$$

2.4.2 기존의 의존성 분석 기법들

의존성 검사는 병렬화를 위한 가장 중요한 분석 작업의 하나로써 다양한 검사법이 연구되었다[28]. 그러나 검사 방법들간의 정확도에 대하여 정확히 규명된 연구 결과는 거의 없으며, 일부 검사 방법들 간에 우수성에 관한 증명이 이루어 졌거나 실험적인 연구가 이루어졌다[32].

의존성 분석 방법중 대표적인 것으로는 GCD 테스트[1], Banerjee 테스트[1], 아이 검사[10,31], 람다 검사[33] 등이 있다.

가. GCD 검사[1]

반복문의 첨자가 정수임을 이용하여 최대공약수의 정리를 사용하는 검사 방법이다. 즉 $\sum_{i=1}^{n} a_i \cdot x_i = c$, 가 정수해를 가질 필요 충분 조건은 $(\gcd(a_1, a_2, \ldots, a_n) \bmod c = 0)$임을 이용하여 의존성의 존재 여부를 검사한다.

나. Banerjee 검사[1]

Banerjee 검사는 2.4.1의 마지막 식에서의 $a_0 - b_0$의 값이 $(\sum_{i=1}^{n} b_i \cdot I_i' - \sum_{i=1}^{n} a_i \cdot I_i)$의 범위에 속하는지를 검사하는 방법이다. 그러나 이 방법은 Ii와 Ii'이 정수라는 성질을 이용하지는 못한다.

다. 아이 검사(I test)[10,31]

GCD 검사와 Banerjee 검사를 조합하여, 최대 공약수 검사의 해의 정수 여부와 Banerjee 검사의 범위를 고려하는 방법이다.

라. 람다 검사(Lambda test)[33]

위의 검사들은 배열 원소들의 각각의 차원을 각각 검사하여 의존 관계가 없는 차원을 찾아내면 의존성이 없는 것으로 결론을 내린다. 이러한 방법은 모든 차원에서 의존 관계가 존재하더라도 어떤 한 차원에서 구한 해집합이 다른 차원에서 포함되지 않는 경우가 생긴다.

람다 검사는 각각의 차원에 대한 선형식들의 모든 조합이 해를 가지는지를 검사하게 된다. 이러한 람다 검사는 배열의 여러 첨자에 공통으로 쓰이는 반복문의 변수가 있을 경우에 유용한 방법이 된다.

2.5 병렬성 검출

병렬성 검출은 주어진 의존성 분석 결과를 바탕으로 병렬로 수행될 수 있는 부분을 찾아내는 작업이다. 병렬 수행이 유용한 계산 집약적인 응용들은 수행시간의 많은 부분을 반복문 내에서의 계산에 소비하게 되므로, 병렬성 검출 작업도 반복문의 병렬화에 중점을 두고 연구되어 왔다. 이러한 작업은 수행될 컴퓨터의 구조에 따라 벡터 명령어 생성(vectorization), 병렬 반복문 생성(parallelization) 등이 있다[1].

2.5.1 벡터 명령어 생성

벡터 명령어는 여러개의 배열 원소에 대하여

동일한 작업을 하는 명령어를 말한다. 이러한 벡터 명령어를 생성하기 위해서는 여러개의 문장을 가진 반복문을 여러개의 작은 반복문들로 분할하고 하나의 명령어를 가진 병렬 반복문에 대하여 적절한 벡터 명령어로 대치하게 된다[1]. 벡터 명령어로 대치시에는 반복간의 의존성(loop-carried dependence)이 없어야 프로그램의 의미를 보존할 수 있다.

2.5.2 병렬 반복문(doall loop) 생성

병렬 반복문의 생성은 벡터 명령어 생성과 반대로 여러개의 반복문을 합쳐 병렬 반복문의 크기를 크게 하는 것이 목적 프로그램의 성능을 높이게 된다. 이것은 병렬 작업의 크기가 클수록 동기화(synchronization), 프로세스 발생 등으로 인한 부담(overhead)을 줄일 수 있기 때문이다. 따라서 병렬 반복문 생성에서는 반복문 병합(loop fusion) 변환이 유용한 작업이 된다[1].

기본적인 병렬 반복문의 생성은 벡터 명령어 생성과 마찬가지로 반복간의 의존성 존재 여부에 따라 병렬 반복문으로 바꾸어 주는 간단한 과정이 되나, 방법면에서 조금씩의 차이가 있다.

가. Parafrase2의 방법[3,41]

Parafrase2에서는 병렬 반복문 생성시에 단순히 반복간의 의존성 유무만을 검사하여 DO 문을 DOALL문으로 바꾸어 주기만 한다. 그러나 그 전후에 반복문의 분할과 반복문의 병합 작업을 수행한다.

나. Allen 등의 알고리즘[1]

Allen, Callahan, Kennedy 등이 제안한 알고리즘은 중첩된 반복문의 의존성 그래프를 바탕으로 하여 바깥쪽 반복문부터 반복문의 분할과, 문장 재배열, 병합을 동시에 수행하면서 병렬 반복문을 검출하는 방법을 취한다.

이러한 알고리즘은 IBM의 RP3를 위한 병렬화 컴파일러와 PFC+에서 사용되었다.

다. SUIF 병렬화 컴파일러

의존성을 갖는 두 반복간의 거리를 벡터로 나타낸 것을 의존성 벡터라고 한다. 이러한 의존성 벡터의 값이 모두 양수이면 중첩된 반복문의 순서를 바꾸어도 의존성이 유지되는 성질이 있으며 이러한 반복문을 교환 가능(fully permutable)하다고 한다.

Wolf, Lam 등이 SUIF 병렬화 컴파일러에서 사용한 방법[12]은 교환 가능한 반복문을 생성하고, 바깥쪽으로 병렬 수행이 가능한 반복문을 이동한 후에, 안쪽의 반복문을 의존성 벡터와 무관한 방향으로(wavefront method)[11] 수행하도록 병렬화를 행한다. 이 방법의 단점은 반복문의 분할이나 병합을 고려하지 않기 때문에 병렬 반복문의 생성과 별개로 분할과 병합을 수행해야 한다는 점이다.

2.6 코드 변환

코드 변환은 프로그램의 문장을 변형하므로써, 병렬화의 효과나, 메모리 사용의 최적화, 수행시간 단축 등을 꾀하는 것을 말한다. 이 절에서는 병렬화와 관련된 코드 변환중 특히 반복문과 관련되어 많이 사용되는 기법들을 살펴보기로 한다[1, 4, 25, 30].

2.6.1 반복 재배열(reodering) 변환

반복을 재배열한다는 것은 반복 공간내에서 수행되는 각각의 반복들의 순서를 바꾸는 것을 말한다. 반복 재배열 변환에는 반복문 교환(loop interchange), 반복문 역전(loop reversal), 반복 모양 변환(loop skewing), 반복 공간 분할(loop blocking, tiling) 등이 있다[12, 13].

가. 반복문 교환(loop interchange)

반복문 교환은 여러 개의 중첩된 반복문이 있을 경우 바깥쪽의 반복과 안쪽의 반복을 바꾸는 것이다. 중첩된 반복문의 순서를 바꿈으로써, 병렬화가 가능한 반복문을 안쪽으로 옮겨서 벡터 명령어로 대치하거나, 반대로 바깥쪽으로 병렬 반복문을 모아서 병렬 작업의 크기를 증가시킬 수도 있다. 또한 자료 접근의 지역성(locality)를 향상시키는데도 유용하게 사용된다.

나. 반복문 역전(loop reversal)

반복문 역전은 반복의 방향을 반대로 바꾸는 것이다. 즉 1부터 n까지 반복하는 반복문을 n부터 1까지로 바꾸는 것을 말한다. 반복문 역전은 의존성 벡터의 해당 원소의 부호를 바꾸

어 주기 때문에 다른 반복문 재배열 변환과 병행하여 유용하게 쓰인다.

다. 반복 모양 변환(loop skewing)

반복 모양 변환은 반복 공간의 모양을 바꾸는 것이다. 반복공간의 모양을 바꿈으로서 반복간의 의존성을 반복내의 의존성으로 바꿀 수 있다.

그림 6의 (a)의 반복문은 두 개의 중첩된 반복 방향으로 모두 의존성이 존재한다. 그러나 대각선 점선 방향의 반복 사이에는 의존성이 존재하지 않는다. 따라서 반복 모양이 바뀐 그림 6의 (b)를 보면 일정한 J 값에 대하여 I 반복문의 반복 사이에는 의존성이 없음을 볼 수 있다. 그림 (d)는 (b)의 반복 공간의 모양과 의존성을 나타낸다.

그림 6의 (b)의 변환된 반복문에서는 기존의 의존성 벡터가 양의 값을 가진 반복문이므로 반복문 교환이 가능하다. (b)의 반복문에 반복문 교환을 적용하므로써 결과적으로 (e)의 안쪽 반복문은 병렬 수행이 가능해진다.

역전, 교환, 모양 변환 등의 반복 공간 재배열 변환은 유니모듈러 행렬(unimodular matrix)[1]을 통한 반복공간의 사상을 통하여 나타낼 수 있다[12]. 이때 의존성 벡터가 0벡터보다 순서적으로(lexicographically) 크게 유지되면 기존의 의존성을 보존하게 된다.

라. 반복 공간 분할(loop blocking, tiling)

반복 공간 분할은 각각의 반복을 두 개의 차원으로 나누는 것이다. 일반적으로 이러한 변환은 n차원의 반복 공간에서 각각의 차원을 분할 하여 분할된 덩어리를 순서대로 수행하므로써 자료 참조의 지역성(locality)을 향상시키는데 사용된다[13].

2.6.2 반복문 분할 및 병합

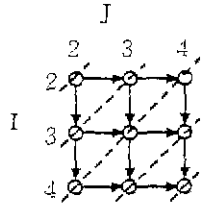
반복문 분할(distribution)은 한 반복문 내의 여러 문장들을 나누어 여러개의 반복문을 만드는 것을 말한다. 이러한 반복문 분할은 반복간의 의존성을 서로 다른 반복문과 반복문 사이의 의존성으로 바꾸어 병렬 수행이 가능한 반복문을 만들거나, 벡터 명령어를 만들기 위해서 사용된다.

```
do I=2, n-1
  do j=2, m-1
    a[i,j]=(a[i-1, j]+a[i,j-1]+
            a[i+1,j]+a[i,j+1])/4
  end do
end do
```

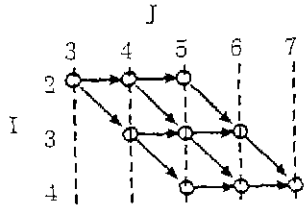
(a) 입력 프로그램(의존성 벡터={(1,0), (0,1)})

```
do i=2, n-1
  do j=i+2, i+m-1
    a[i,j-1]=(a[i-1, j-1]+a[i,j-i-1]+
              a[i+1. j-1]+a[i, j-i+1])/4
  end do
end do
```

(b) 반복 모양 변환 프로그램
(의존성 벡터={(1,1), (0,1)})



(c) 입력 프로그램 반복 공간



(d) 변환된 프로그램의 반복공간

```
do j=4, m+n-2
  do i=max(2, j-m+1), min(n-1, j-2)
    a[i, j-i]=(a[i-1, j-i]+a[i, j-i-1]+
               a[i+1, j-i]+a[i, j-i+1])/4
  end do
end do
```

(e) 모양 변환과 반복문 교환을 수행한 결과
(의존성 벡터={(1, 0),(1,1)}

그림 6 반복 모양 변환을 위한 예제[4]

1) 디터미넌트(determinant)가 1인 행렬을 사용하여 반복 모양 변환을 하면 반복문 증가값의 절대값이 유지되며, 해당 변환과 역변환이 반복문의 증가값을 정수로 유지한다.

반복문 병합(fusion)은 서로 다른 반복문을 하나로 합쳐 하나의 반복내에 수행되는 작업을 크게 하는 것을 말한다. 이렇게 함으로써 반복으로 인한 부담을 줄일 수 있고 병렬 반복문의 경우 병렬 수행의 부담을 줄일 수 있다. 이러한 반복문 병합은 주로 반복문 분할과 병행하여 이루어진다.

2.6.3 기타 변환

위의 기본적인 방법 외에도 병렬화의 효과를 높이기 위한 다양한 변환 방법들이 제안되었고 구현되었다. 이러한 변환 방법에는 반복문 정규화(loop normalization), 스칼라 확장(scalar expansion), 유도 변수 대치(induction variable substitution), 반복문 펼침(loop unrolling), 배열 확장(array expansion), 축약식 규명(reduction recognition), 변수 지역화(privatization), 전향 대치(forward substitution), 변수 재명명(variable renaming) 등이 있다[1,4].

본 절에서는 병렬화와 관련하여 일반적으로 사용되는 반복문 정규화, 스칼라 확장, 유도 변수 대치, 축약식 규명에 대해서만 설명하기로 한다.

가. 반복문 정규화(loop normalization)

반복문 정규화는 반복문의 첨자가 1씩 증가하도록 변환하는 것을 말한다. 이 방법은 의존성을 없애서 병렬 수행의 범위를 확장시키는 변환이라기 보다는, 의존성 분석과 기타 변환을 쉽게 하기 위한 전처리라고 할 수 있다[1].

나. 스칼라 확장(scalar expansion)

반복마다 임시 기억장소로 사용되는 스칼라 변수를 배열로 확장하여 각 원소가 각각의 반복에서 사용하도록 하므로서 기억장소의 재사용으로 인한 의존성을 없애는 것을 말한다. 이와 비슷한 접근법으로 이러한 변수를 한 반복내의 지역변수로 선언하는 변수 지역화(privatization)가 있으며, 이러한 기법을 배열로 확장한 배열 확장(array expansion)[19], 배열 지역화(array privatization)[2] 등이 있다.

다. 유도 변수 대치(induction variable substitution)

유도 변수에 대해서는 이미 분석부에 관한 설명에서 언급한 바 있다.

유도 변수는 반복마다 일정하게 변화하므로 당연히 반복간의 의존성을 발생시키게 되는데 이러한 유도 변수를 반복문 첨자의 선형식으로 대치하는 것이 유도 변수 대치이다[17]. 그림 4의 예를 보면 유도 변수인 A와 B는 반복내에서 S+I와 S+I*(I+1)/2 로 대치될 수 있다.

라. 축약식 규명(reduction recognition)

축약(reduction)이란 반복문의 작업이 합산이나 최대값 계산 등과 같은 하나의 식으로 표현될 수 있는 경우를 말한다. 대부분의 병렬 언어들은 이러한 계산을 병렬로 수행하는 라이브러리를 제공하고 있으므로 이러한 반복문을 해당 라이브러리 함수로 대치하므로써 프로그램의 속도를 향상시킬 수 있다.

3. 병렬화 컴파일러 개발 사례

3.1 국외 개발 사례

3.1.1 Parafrase[1, 3, 41]

Parafrase는 Univ. of Illinois at Urbana-Champaign에서 개발된 최초의 병렬화 컴파일러로서 많은 병렬화 컴파일러에 영향을 주었다.

Parafrase의 입력은 Fortran 프로그램이며 출력은 확장된 병렬 Fortran 프로그램과 변환에 대한 오류, 성능 예상 등의 기록이 된다. Parafrase에서는 100여개 이상의 코드 변환 및 병렬화 알고리즘을 제공하고 있는데, 각각의 알고리즘에 대해서는 선행되어야 할 분석이나 변환 알고리즘이 정의되어 있다. 사용자는 변환 과정들의 순서와 종류를 선택할 수 있으며, 이를 통해 다양한 응용에 적절한 병렬화 작업을 수행할 수 있도록 하고 있다.

Parafrase의 후속 컴파일러인 Parafrase2는 병렬화 정보가 추가된 추상 구문 트리를 내부 구조로 하여 C, Fortran, Pascal 등의 다양한 언어를 지원하는 병렬화 컴파일러이다. Parafrase2는 변수값 추적 분석(symbolic analysis)[18]을 이용한 의존성 분석, 유도 변수 검출과 프로시져간 분석 등이 보강되었다.

3.1.2 PFC[1]

PFC는 Rice 대학에서 개발한 벡터화 컴파일러(vectorizing compiler)이다. 중간 표현으로는 추상 구문 트리가 사용되며 Fortran66 또는 Fortran77 프로그램을 입력으로 받아 프로시져간 분석, 전처리 코드 변환, 의존성 분석, 벡터 코드 생성의 네 가지 과정을 거쳐 Fortran 90 프로그램을 출력한다.

PFC는 Parafrase에 비하여 상대적으로 적은 수의 분석 및 변환 알고리즘을 적용하고 있으며, 각각의 패스가 내부 자료 구조에 한꺼번에 적용되기 때문에 각각의 패스가 독립적으로 적용되는 Parafrase에 비하여 실행속도가 빠른 특징을 갖는다.

3.1.3 Ptran[38]

Ptran은 IBM에서 개발된 공유 메모리 병렬 컴퓨터를 위한 병렬화 컴파일러이다. 병렬화 컴파일러의 입력 언어는 병렬 반복문을 추가한 VS Fortran77이며 출력은 VS Fortran2.5나 Tobey와 같은 병렬 언어 프로그램을 출력한다.

Ptran 분석부는 프로시져간의 분석과 프로시져내의 분석을 상호보완적으로 사용한다[5]. 분석부에서는 먼저 호출 그래프를 하향(top-down)으로 추적하면서 프로시져간 이명 분석과 상수 전달을 수행한다. 그 후에 다시 호출 그래프를 상향(bottom-up)으로 추적하면서 프로시져내의 상수 전달 및 의존성 분석, 프로시져 호출시의 부작용, 자료 흐름 분석 등을 수행하게 된다.

또한 병렬 코드 생성시에는 수행될 병렬 컴퓨터에 따른 병렬 수행의 부담을 고려하여 수행시간을 추정하므로써 속도의 향상을 최적화 할 수 있도록 하였다.

3.1.4 KAP[21]

KAP은 Parafrase의 영향으로 개발된 상용 병렬화 및 최적화 컴파일러이다. KAP의 입력은 Fortran77, Fortran90, C 등이 되며 HP, SPARC, Alpha 등의 프로세서에 대한 최적화된 병렬 프로그램을 생성한다.

KAP은 입력 프로그램에 사용자가 다양한 방법으로 병렬화를 위한 정보를 줄 수 있도록 하여 적당한 병렬화 방법과 실행시의 조건을 알려주거나 사용자가 직접 병렬 수행을 표시할 수 있도록 하고 있다. 이러한 정보는 의존성 분석이나 병렬 프로그램 생성시에 사용된다.

의존성 분석 단계에서는 사용자가 제공한 실행 조건을 최대한 활용하도록 하고 있으며, 실행시의 상태에 따른 여러 종류의 의존성 결과를 냄으로써 실행시의 상태에 따라 적절한 순차 또는 병렬 수행을 할 수 있도록 하고 있다.

KAP의 병렬 반복문 생성부는 반복문의 교환, 분할, 교환을 통하여 생성 가능한 모든 병렬 반복문에 대하여 자료의 지역성(locality)과 병렬 수행의 효과를 계산하고, 가장 성능이 좋은 병렬 반복문을 선택한다. 또한 자료의 지역성의 고려시에 SMP(Symmetric Multiprocessor)와 같은 목적 프로세서의 구조를 고려한다는 점이 특징이라고 할 수 있다.

3.1.5 SUIF[23,42]

SUIF은 Stanford 대학에서 개발한 컴파일러 개발 환경이다. SUIF에서는 중간 표현으로서의 SUIF 언어와 Fortran-to-C 변환기, C-to-SUIF 변환기, SUIF-to-C 변환기 등의 개발 도구 및 SUIF 중간 표현을 조작할 수 있는 라이브러리, 각종 프로그램 변환 라이브러리를 제공하고 있다. SUIF을 이용하여 프로그램 최적화기와 병렬화 컴파일러, MIPS 목적 코드 생성기 등이 개발된 상태이며 또한 다양한 최적화 컴파일러들이 여러 대학에서 개발되고 있다.

SUIF 병렬화 컴파일러는 Fortran과 C를 입력으로 받아 쓰레드 라이브러리를 호출하는 C 프로그램을 생성한다. 코드 변환부에서는 Wolf등이 제안한 유니모듈러 transformation을 사용하여 병렬 반복문을 생성하는 것이 특징이며, 반복문 분할과 병합은 지원하지 않는다.

3.1.6 Polaris[2,40]

Polaris는 UIUC에서 개발된 최신의 병렬화 컴파일러중의 하나이다.

Polaris의 개발은 Perfect Benchmark의 분

석을 기반으로 하여 이루졌는데, 기존의 실용적인 응용들이 실제로 효과적인 병렬화가 가능함을 확인하고 이를 자동으로 수행하기 위하여 필요한 변환 및 분석 기법을 구현하는 접근법을 취하였다.

Polaris에서 강조한 분석 및 변환 기법으로는 프로시져간 분석시에 강력한 프로시져 확장(inline expansion)을 사용한 것과 변수값 추적 분석을 사용한 비선형 첨자의 의존성 분석, 유도 변수와 축약식의 대치, 스칼라 및 배열 지역화(privatization), 수행시(run time)의 프로그램 분석을 통한 병렬 수행 등이 있다.

3.2 주전산기IV를 위한 병렬화 컴파일러

본 절에서는 한국과학기술원(KAIST)에서 수행중인 병렬화 컴파일러 개발 연구[36, 37]에 대하여 소개한다. 본 연구는 주전산기IV(SPAX) 개발 연구의 일부로서, 현재 2차년도의 전체 구현 작업이 마무리 단계에 있으며 3차년도의 성능 평가 및 안정화 작업을 수행중에 있다.

SPAX의 병렬화 컴파일러는 Fortran77과 C 프로그램을 입력으로 받아 병렬 라이브러리를 호출하는 C 프로그램을 생성한다. 중간 표현으로는 SUIF을 선택하여 SUIF의 프로그램 접근 및 변환 라이브러리가 사용되었다. 입력된 Fortran 프로그램은 일단 C로 번역된 후에 SUIF 프로그램으로 변환되며 분석부와 코드 변환, 병렬 루프 검출, C 프로그램으로의 재번역을 거쳐 병렬 라이브러리를 호출하는 C 프로그램으로 변환되어진다.

분석부에서는 상수 전달 분석과 이명 분석을 수행한다. 상수 전달 분석은 조건 상수를 고려하였으며 점진적 자료 흐름 그래프(IDG)를 분석을 위한 자료 구조로 사용하였다. IDG는 기본적으로는 정의-사용 그래프나 SSA 형태와 유사하지만 확장성을 고려한 구조라는 장점을 가진다[39].

병렬 코드 생성을 위한 코드 변환으로는 스칼라 변수 확장과 유도 변수 대치, 반복문 정규화를 수행한다. 의존성 검사는 GCD, Banerjee, 아이, 람다 검사를 구현하였으며 병렬화부에서는 병렬 반복문 생성을 위하여 반복 모양 변환과 함께 반복문의 병합과 분할을 병행하였다.

목적 병렬 프로그램은 SPAX가 병렬 쓰레드 라이브러리와 MPI를 모두 지원하므로, 쓰레드 라이브러리를 호출하는 공유 메모리(shared memory) 방식의 C 프로그램과 MPI를 호출하는 SPMD(Single Program Multiple Data) 방식의 C 프로그램을 선택적으로 생성하도록 하였다. 개발중인 병렬화 컴파일러의 전체 과정은 그림 7과 같다.

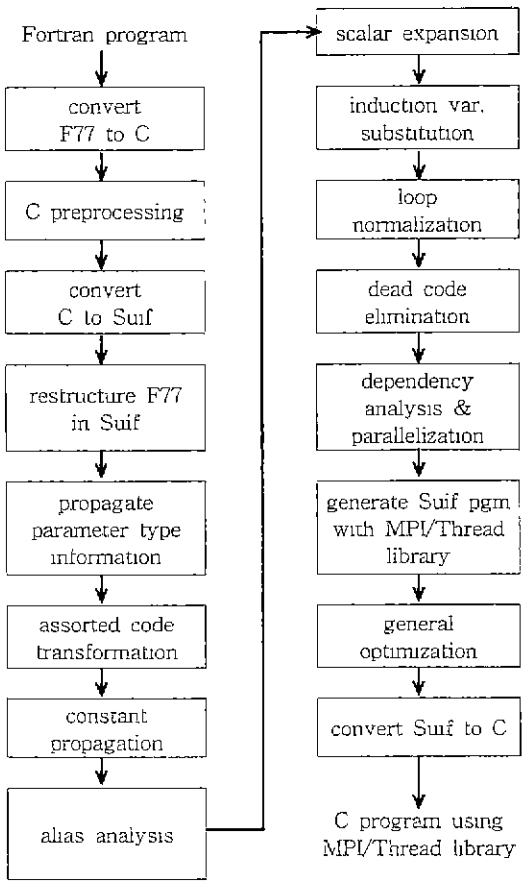


그림 7 병렬화 컴파일러의 동작 과정

4. 결 론

본 논문에서는 병렬화 컴파일러를 위한 일반적인 분석, 프로그램 변환 방법을 설명하고 그 개발 사례를 소개하였다.

지금까지 병렬화 컴파일러의 개발은 주로 공

유 메모리 방식의 컴퓨터와 벡터 컴퓨터를 위한 병렬 프로그램의 생성을 중심으로 연구되어 왔으며 현재 상용화 단계에 있다. 분산 메모리 구조 병렬 컴퓨터를 위한 연구는 병렬 언어 컴파일러 연구를 중심으로 수행되어 왔으나[6], 최근들어 자료의 자동 분할 등에 관한 연구를 중심으로 활발히 연구되는 추세에 있다[27].

이러한 병렬화 컴파일러 관련 기술은 병렬 컴퓨터의 일반화와 프로세서의 병렬 구조화 추세를 고려할 때 컴파일러 개발의 중심 기술이 될 것으로 판단된다.

참고문헌

[1] Barbara Chapman and Hans Zima, Supercompilers for Parallel and Vector Computers, Addison-Wesley, 1990.

[2] Bil Blume et. al., "Polaris : The Next Generation in Parallelizing Compilers," Proceedings of the Workshop on Languages and Compilers for Parallel Computing, pp. 10.1-10.18, 1994.

[3] C.D. Polychronopoulos et al., "The Structure of Parafrase-2 : An Advanced Parallelizing Compiler for C and Fortran," In Proceedings of "89 International Conference on Parallel Processing, pp.II-39~48, IEEE Computer Society.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, "Compiler transformation for high-performance computing," Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, 1993.

[5] Frances Allen, Michael Burke, Philippe Charles, Rob Crypton and Jeanne Ferrante, "Overview of the Ptran Analysis System," Journal of Parallel and Distributed Computing, Vol. 5, No. 5, pp. 617-640, 1988.

[6] Hans P. Zima and Barbara Mary Chapman, "Compiling for distributed-memory systems," Proceedings of the IEEE, Vol. 81, No. 2, pp. 264-286, 1993

[7] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam, "Data and computation transformation for multiprocessors," Proceedings on the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995.

[8] Jond-Deok Choi, Ron Cytron, and Jeanne Ferrante, "Automatic Construction of Sparse Dataflow Evaluation Graph," ACM Symposium on Principles of Programming Languages, pp. 55-66, 1991.

[9] Karen L. Pieper, Parallelizing Compilers : Implementation and Effectiveness, Phd thesis, Stanford University, Computer Systems Laboratory, 1993.

[10] Kleanthis Psarris, Xiangyun Kong, and David Kalppholz, "The direction vector I test," IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 11, pp.1280-1290, 1993.

[11] Lesile Lamport, "The parallel execution of do loops," Comm. of the ACM, Vol. 17, No. 2, pp. 83-93. 1974

[12] M. E. Wolf and Monica S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," IEEE Trans. on Parallel and Distributed Systems, Vol. 2, October, 1991.

[13] M. E. Wolf, Improving Locality and Parallelism in Nested Loops, Phd thesis, Stanford University, Computer Systems Laboratory, August 1992.

[14] Mark N. Wegman and F. Kenneth Zadeck, "Constant Propagation with conditional branches," ACM Trans. on Programming Languages and Systems, Vol. 13, No. 2, 1991.

[15] Message Passing Interface Forum, MPI : A Message Passing Interface Standrd, final report version 1.0 edition, 1995.

[16] Michael Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996

[17] Michael Wolfe, "Beyond induction varia-

bles," ACM SIGPLAN Notices, Vol. 27, No. 7, pp. 162-174, 1992.

[18] Mohammad R. Haghighat and Constantine D. Polychronopoulos, "Symbolic Program Analysis and Optimization for Parallelizing Compilers," CSRD Report 1237, UIUC, 1992.

[19] Paul Feautrier, "Array Expansion," In Proceedings of International Conference on Supercomputing, pp. 429-441, 1988.

[20] Richard Johnson, Efficient program analysis using dependence flow graphs, PhD thesis, Department of Computer Science, Cornell University, 1994.

[21] Robert H. Kuhn, Bruce Leasure, and Sanjiv M. Shah, "The KAP Parallelizer for DEC Fortran and DEC C Programs," Digital Technical Journal, Vol. 6, No. 3, pp. 57-70, 1994

[22] Robert P. Wilson and Monica S. Lam, "Efficient context-sensitive pointer analysis for C programs," In Proceedings on the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, 1995.

[23] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson. Steve W. K. Tjiang, Shih-Wei Liao, C.W. Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy, "SUIF : An infrastructure for research on parallelizing and optimizing compilers," ACM SIGPLAN Notices, Vol. 29, No. 12, pp. 31-37, 1994.

[24] Jong-Deok Choi, Michael Burke and Paul Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Analysis and Side Effects," In Proc. of ACM Symposium on Principles of Programming Languages. pp. 232-246, 1993.

[25] Ron Cytron, "Compiler Construction Tutorial Notes," In SIGPLAN'93 Conference on PLDI, 1993.

[26] Ron Cytron, Jeanne Ferrante, and Barry K. Rosen, "An Efficient Method of Computing Static Single Assignment form," ACM Symposium on Principles of Programming Languages, pp. 25-35, 1989.

[27] S.P.Amarasinghe and M.S. Lam, "Communication optimization and code generation for distributed memory machines," In Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, June, 1993.

[28] Utpal Banerjee, Dependence Analysis for Sypercomputing, Kluwer Academic Publishers, 1988.

[29] Utpal Banerjee, Rudolf Eigemann, Alexandru Nicolu, and David A. Padua, "Automatic program parallelization," In Proceedings of the IEEE, Vol. 81, No. 2, pp. 211-243, 1993.

[30] Utpal, Banerjee, Loop Transformations for Restructuring Compilers : The Foundations, Kluwer Academic Publishers, 1993.

[31] Xiangyun Kong, David Klappholz, and Kleanthis Psarris, "The I test : An improved dependence test for automatic parallelization and vectorization," IEEE Trans. on Parallel and Distributed Systems, Vol. 2, No 3, pp. 342-349, 1991.

[32] Z. Shen, Z. Li, and P. Yew, An emprical study of Fortan programs for parallelizing compilers, IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 3, pp. 356-364, 1990.

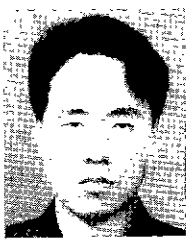
[33] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu, "An efficient data dependence analysis for parallelizing Compilers," IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 1, pp.26-34, 1990.

[34] S. Horwitz, P. Pfeiffer, and T. Reps. "Dependence analysis for pointer variables," In Proceedings on the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, 1989.

[35] V.A.Guarna Jr., "A Technique for Analyzing Pointer and Structure References in Parallel Restructuring Compilers," Proc.

of ICPP, Vol. II, pp.212-220, 1988
[36] 병렬화 트랜스레이터 개발에 관한 연구, 최종연구보고서(연구수행기관 : 한국과학기술원 전산학과), 한국전자통신연구소, 1996.
[37] 병렬화 트랜스레이터와 노드간 연결망의 검증 및 성능 평가 시스템 개발, 최종 연구 보고서(연구수행기관 : 한국과학기술원 전산학과), 한국전자통신연구소, 1995.
[38] 최종덕, "프로그램 병렬화를 위한 고급 컴파일러 기술", '91 컴퓨터 과학 하계 세미나 자료집, 인공지능연구센터, 한국과학기술원, 1991.
[39] 허재원, 점진적 자료 흐름 그래프를 이용한 효율적인 상수전달 기법, 석사 논문, 한국과학기술원 전산학과, 1996
[40] Polaris Home Page, http : //csrd.uiuc.edu/polaris/polaris.html.
[41] Parafrase2 Home Page, http : //csrd.uiuc.edu/parafrase2.
[42] SUIF Home Page, http : //suif.stanford.edu.

안 준 선



1992 서울대학교 계산통계학과 졸업(학사)
1994 한국과학기술원 전산학과 졸업(석사)
1994～현재 한국과학기술원 전산학과 박사과정
관심분야 : 함수형 언어, 병렬 언어, 병렬화 컴파일러, 프로그램 최적화 등임.

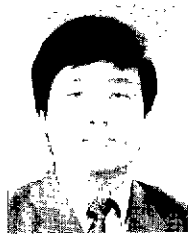
최 광 훈

1994 한국과학기술원 전산학과 졸업(학사)
1996 한국과학기술원 전산학과 졸업(석사)
1996～현재 한국과학기술원 전산학과 박사과정
관심분야 : 함수형 언어, 병렬화 컴파일러 등임

김 성 훈

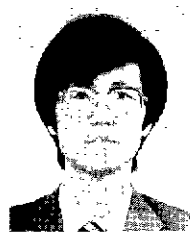
1991 한국과학기술원 전산학과 졸업(학사)
1993 한국과학기술원 전산학과 졸업(석사)
1993～현재 한국과학기술원 전산학과 박사과정
관심분야 : 프로그래밍 언어 의미론, 프로그램 정적 분석 등임

한 태 숙



1976 서울대학교 전자공학과 졸업(학사)
1978 한국과학기술원 전산학과 졸업(석사)
1990 Univ. of North Carolina at Chapel Hill 졸업(박사)
현재 한국과학기술원 전산학과 조교수
관심분야 : 프로그래밍 언어론, 함수형 언어 등임

최 광 무



1976 서울대학교 전자공학과 졸업(학사)
1978 한국과학기술원 전산학과 졸업(석사)
1984 한국과학기술원 전산학과 졸업(박사)
1985～86 AT&T Bell Labs. Member of Technical Staff
현재 한국과학기술원 전산학과 교수
관심분야 : 형식 언어 이론, 논리 프로그램 병렬 수행, 최적화 컴파일러 등임