다중스레드 데이타 병렬 프로그램의 표현 : PCFG(Parallel Control Flow Graph)

(A Representation for Multithreaded Data-parallel Programs : PCFG(Parallel Control Flow Graph)

김 정 환 †

(Junghwan Kim)

요 약 데이타 병렬 모델은 대규모 병렬성을 용이하게 얻을 수 있는 장점이 있지만, 데이타 분산으로 인한 통신 지연시간은 상당한 부담이 된다. 본 논문에서는 데이타 병렬 프로그램에 내재되어 있는 태스크 병렬성을 추출하여 이러한 통신 지연시간을 감추는데 이용할 수 있음을 보인다. 기존의 태스크 병렬성 추출은 데이타 병렬성을 고려하지 않았지만, 여기서는 데이타 병렬성을 그대로 유지하면서 태스크 병렬성을 활용하는 방법에 대해 설명한다. 데이타 병렬 루프를 포함할 수 있는 다수의 태스크 스레드들로 구성된 다중스레드 프로그램을 표현하기 위해 본 논문에서는 PCFG(Parallel Control Flow Graph)라는 표현 형태를 제안한다. PCFG는 단일 스레드인 원시 데이타 병렬 프로그램으로부터 HDG(Hierarchical Dependence Graph)를 통해 생성될 수 있으며, 또한 PCFG로부터 다중스레드 코드를 쉽게 생성할 수 있다.

키워드 : 데이타 병렬성, 태스크 병렬성, 다중스레드, 제어 흐름 그래프, 병렬 제어 흐름 그래프, 제어 종속성, 데이타 종속성, 통신 지연시간.

Abstract In many data-parallel applications massive parallelism can be easily extracted through data distribution. But it often causes very long communication latency. This paper shows that task parallelism, which is extracted from data-parallel programs, can be exploited to hide such communication latency. Unlike the most previous researches over exploitation of task parallelism which has not been considered together with data parallelism, this paper describes exploitation of task parallelism in the context of data parallelism. PCFG(Parallel Control Flow Graph) is proposed to represent a multithreaded program consisting of a few task threads each of which can include a few data-parallel loops. It is also described how a PCFG is constructed from a source data-parallel program through HDG(Hierarchical Dependence Graph) and how the multithreaded program can be constructed from the PCFG.

Key words : data parallelism, task parallelism, multi-thread, control flow graph, parallel control flow graph, control dependence, data dependence, communication latency.

1. 서 론

Fortran D나 HPF와 같은 언어에서 이용하는 데이타 병렬성은 많은 과학 계산 응용에서 추출될 수 있는 병렬성 중의 하나이다. 데이타 병렬성은 다수의 데이타 집합에 대해 동일한 연산을 수행할 수 있는 형태로 벡터 컴퓨터, 분산 메모리 컴퓨터 등에 효과적으로 적용될 수 있다. 이와 관련해서는 데이타 종속성 분석에 기반한 루프 변환 및 병렬화에 대한 많은 연구가 이루어져 왔다[1]. 한편, 이와는 다른 형태의 병렬성으로 태스크 병렬성[2], 또는 함수 병렬성이 있는데, 이는 한 프로그램 내의 서로 다른 두 개 이상의 코드 부분을 병렬적으로 수행하는 형태를 말한다. 이때 병렬적으로 수행되는 코드 부분(태스크)은 하나의 문장일 수도 있고, 루프나 함수일 수도 있다. 하나의 태스크는 스레드로 사상하여 스케줄링하는 것이 가능하다.

본 논문에서는 데이타 병렬 프로그램으로부터 태스크 병

† 종신회원 : 건국대학교 컴퓨터·응용과학부 교수
jhkim@kku.ac.kr
논문접수 : 2002년 4월 18일
심사완료 : 2002년 10월 22일

렬성을 자동으로 추출하여 다중스레드 프로그램으로 변환하는 과정에 대해 기술한다. 일반적으로 Fortran D나 HPF 등으로 작성된 데이타 병렬 프로그램은 단일 태스크(스레드)로 이루어지는데, 이러한 프로그램은 본 연구에서 제안한 중간 표현 형태인 PCFG(Parallel Control Flow Graph)로 변환된다. PCFG는 이미 다중스레드를 나타내고 있는 형태로써, 이것으로부터 다중스레드 프로그램의 생성은 간단히 그래프 순회를 통해 이루어진다.

데이타 병렬 프로그램에서의 중요한 이슈 중의 하나는 통신 지연시간이다. 병렬 루프는 대개 배열 참조를 포함하고, 이러한 배열들은 분산 메모리 컴퓨터의 경우 여러 개의 계산 노드들에 분산되어 있기 때문에 종종 통신을 유발하게 된다. 통신 지연시간은 프로세서의 계산 지연시간에 비해 매우 크기 때문에 이를 효과적으로 다루는 것은 중요한 과제로 인식되었으며, 메시지 벡터화, 통신/계산 중첩, 중복 통신의 회피 등 여러 최적화 방법들이 연구되어 왔다[3]. 그러나 이들 연구는 단일 스레드 실행 모델에 국한되어 왔다.

다중스레드 실행 모델은 스레드 간의 문맥 전환을 통해 긴 지연시간을 감춤으로써 프로세서의 효율을 높인다. 본 연구에서는 데이타 병렬 모델에 다중스레드 방식을 결합함으로써, 원격 배열 참조 등으로 유발된 통신 지연시간을 효과적으로 감추고자 한다. 생성된 다수의 스레드를 다중프로세서에 스케줄링함으로써 실행 속도를 향상시킬 수도 있지만, 여기서는 단지 통신 지연시간을 감추기 위한 목적으로 스레드를 생성하며 이러한 태스크 병렬성은 하나의 프로세서에 사상된다. 즉, 태스크 병렬성은 통신 지연시간을 감추기 위한 목적으로 이용되며, 복수의 프로세서에 사상되는 것은 데이타 병렬성이다.

본 논문의 2절에서는 관련 연구를 소개하고, 3절에서는 다중스레드 데이타 병렬 프로그램을 표현하기 위한 PCFG를 제안한다. 4절에서는 단일 스레드의 원시 데이타 병렬 프로그램으로부터 PCFG를 생성하는 방법을, 그리고 5절에서는 PCFG로부터 다중스레드 프로그램을 생성하는 방법에 대해 기술한다. 6절에서는 PCFG 생성 예를 살펴보고, 마지막으로 7절에서 결론을 맺는다.

2. 관련 연구

CFG는 여러 최적화 방법에서 기본적으로 사용되는 프로그램 표현 형태이다[4]. CFG가 단일 태스크에 의한 프로그램 순차 흐름을 표현한데 비해, PCFG는 FORK와 JOIN을 통해 병렬 태스크 스레드를 표현할 수 있다. CFG와 PCFG는 둘 다 데이타 종속성 정보는 포함하고 있지 않지만, PCFG는 데이타 종속 정보를 이용하여 생성되었으므로 여기에서 표현된 태스크 스레드들은 병렬 실행이 보장된다.

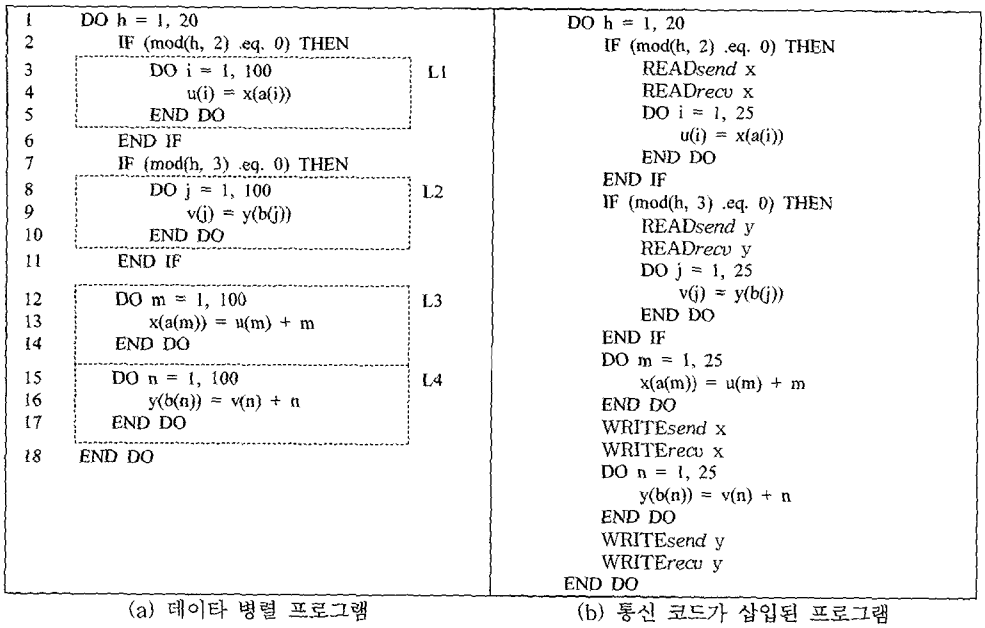
PDG(Program Dependence Graph)는 제어 종속성과 데이타 종속성을 모두 표현함으로써, 순차 프로그램에서의 병렬성을 표현하였다[5]. PDG는 종속 정보들을 표현하지만, 실행 의미를 갖고 있지는 않다. DFG(Dependence Flow Graph)에서는 단순히 종속 정보를 통한 병렬성 표현이 아닌, 실행 가능한 표현 형태를 제안하였다[6, 7]. DFG는 데이타 종속성 그래프와 데이타플로우 모델의 합성으로 볼 수 있다. 이 그래프에서 각 종속 간선은 두 노드 사이의 종속 관계를 의미할 뿐 아니라 데이타플로우 모델에서처럼 노드 실행을 지시하는 토큰 흐름을 나타낸다. PDW(Program Dependence Web)는 Fortran과 같은 명령형(imperative) 언어를 동적 데이타플로우 컴퓨터 구조에 사상하기 위한 표현 형태를 제공한다[8]. DFG와 PDW는 모두 데이타플로우 실행 모델을 제공함으로써 세밀한(fine-grain) 병렬성을 추구하는데, 이는 명령형 언어에 내재되어 있는 지역성과 루프 효율성을 잃는 문제점을 안고 있다.

HTG(Hierarchical Task Graph)는 데이타 종속성과 제어 종속성을 모두 표현하되, 루프 구조는 각 계층에 따라 단일 노드로 추상화되어 있다[9, 10]. 이 그래프에서 각 노드의 실행 조건은 종속 간선들의 논리식으로 표현되고, 이를 통한 자동적인 스케줄링이 가능하다. HTG는 병렬 태스크들의 계층적 표현을 제공함으로써, 크기별 수준에 따른 다양한 병렬성을 내포한다. 따라서 적절한 수준의 루프 구조를 포함한 태스크를 이용할 수 있다. 그러나 서로 다른 노드에 대해 제어 종속적인 노드들 사이의 데이타 종속 간선은 복잡한 동기화를 유발하고, 또한 병렬 태스크 내에서 통신 작업과 같은 긴 지연시간을 유발하는 작업에 대한 고려가 되어 있지 않다.

3. 데이타 병렬성과 다중스레드

3.1 데이타 병렬 프로그램에서의 통신 지연 문제

병렬 루프가 효과적으로 수행되기 위해서는 루프 내에서 참조되는 배열 요소들이 프로세서들에 적절히 분산되어 있어야 한다. 데이타 병렬 프로그램에서 데이타 분산은 일반적으로 프로그래머에 의해 명시적으로 제시된다. 또한 계산 분할은 소유자 계산 법칙(Owner Computes Rule)에 따라 계산하고자 하는 데이타(즉, LHS에서 참조되는 데이타)를 소유한 프로세서에 계산을 할당하는 방식으로 이루어진다. 할당된 계산을 수행할 때 참조되는 모든 배열 요소를 항상 소유하고 있는 것은 아니므로 병렬 루프 수행에서는 종종 메시지 전달과 같은 통신이 발생하게 된다.



(a) 데이타 병렬 프로그램 (b) 통신 코드가 삽입된 프로그램

그림 1 데이타 병렬 프로그램의 CFG 표현

그림 1의 (a)는 데이타 병렬 프로그램을, (b)는 컴파일되어 통신 코드가 삽입된 4-프로세서 버전을 각각 보여준다. 루프 L1에서 참조하는 배열 x의 요소에는 지역 메모리에 존재하지 않는 것이 있을 수 있으므로 통신이 필요하다. 이 때의 통신 코드는 원격 메모리에서 필요로 하는 데이타를 가져오는 것이기 때문에 READ로 표현한다. 이 READ는 다시 자신이 소유하고 있는 데이타를 필요로 하는 프로세서에게 송신해주는 READ*send*와, 반대로 필요로 하는 데이타를 원격 프로세서로부터 수신받는 READ*recv*의 두 단계로 구분할 수 있다. 따라서 (b)에서 루프 L1 이전에 READ*send* x와 READ*recv* x가 삽입되었다. 마찬가지로 루프 L2 이전에는 배열 y에 대한 READ가 삽입되었다. 루프 L3와 L4의 경우에는 각 프로세서에서 정의된(계산된) 배열 x 및 y의 요소들을 소유하고 있지 않을 수 있다. 따라서 계산된 데이타를 소유자 프로세서에게 전송하는 작업 WRITE가 필요하다. READ의 경우에서처럼 WRITE도 WRITE*send* 및 WRITE*recv*로 구분할 수 있다.

그림 1의 예는 배열 인덱스로 배열 요소를 사용하는 간접 참조를 사용하였다. 이와 같은 불규칙 문제의 경우 참조되는 배열 요소들은 컴파일 시에 정적으로 결정할 수 없고, 실행 시에만 결정할 수 있다. 불규칙 문제의 해결에 대해서는 *INSPECTOR-EXECUTOR* 방식[11, 12]이 알려져 있는데, 위 예에서는 배열 인덱스로 사용되는 배열 a 및 b가 재정의 되고 있지 않으므로 INSPECTOR 단계는 초기에 한번 수행되면 되므로 생략하였다. 앞으로 논의의 편의상 그림 1의 불규칙 문제를 예제로 사용할 것이다. 규칙 문제의 경우에도 PCFG의 생성 과정은 동일하지만, 여러 가지 통신 최적화가 가능하므로[13] 이러한 것들이 함께 고려되어야 할 것이다. 통신 최적화 기법들에 대해서는 본 논문에서 다루지 않는다. 다만, PCFG의 생성 단계에서 필요한 통신 코드의 배치 방법으로 메시지 벡터화(vectorization)나 합착(coalescing)의 경우와 유사하게 통신 코드는 가능한 루프 밖에 위치시킴으로써 통신 회수를 줄이는 기법이 사용될 것이다. 또한 READ*send*와 READ*recv*는 문제에 따라 점대점(point-to-point) 통신, 군집(collective) 통신 또는 다른 형태의 통신이 될 수 있지만, 여기서는 단지 지연시간이 발생하는 지점이라는 차원에서 다룰 것이다.

3.2 태스크 병렬성의 활용

일반적으로 READ*send*와 READ*recv* 사이에 다른 독립적인 계산 코드를 배치하면 통신 지연시간을 감출 수 있다[1]. 즉, READ*send*는 가능한 일찍 수행하고, READ*recv*는 가능한 늦게 수행하도록 배치하는 것이 성능 향상에 유리하다. 이와 같은 정적인 코드 배치에 대한 연구 중의 하

1) two-sided 통신의 경우에는 송신자와 수신자의 랑데부가 필요하기 때문에, one-sided 통신에 비해 지연 시간 감춤 효과가 작을 수 있다.

나로 GIVE-N-TAKE를 들 수 있다[14]. 그러나, 그림 1의 경우 이러한 컴파일 시의 정적인 코드 배치만으로는 최대 효율을 얻을 수 없는데, 그 이유는 *if* 블럭 안에 통신 코드를 갖게 되기 때문이다. 원격 READ는 *if* 문의 조건에 따라 실행될 수도 있고, 그렇지 않을 수도 있다. 따라서, READ*send* 또는 READ*recv*를 *if* 블록 밖으로 옮길 수는 없다. 반대로 *if* 블럭 밖의 다른 문장들을 READ*send*와 READ*recv* 사이로 옮기기는 것도 곤란하다. 특히 그림 1의 경우 두 *if* 블럭 사이에 종속성이 없으므로, 어느 한 *if* 블럭을 다른 *if* 블럭 안으로 이동시킬 수는 있지만, 이와 같은 코드 이동에서의 이득은 극히 제한적이다. 한편 두 *if* 블럭을 서로 다른 스레드에서 실행시키면, 통신 지연시간은 동적인 스레드 스케줄링에 의해 자연스럽게 감춰질 것이다.

그림 1에서 2~17까지의 문장은 병렬 수행될 수 있는 두 개의 그룹으로 나눌 수 있는데, (2, 3, 4, 5, 6, 12, 13, 14)와 (7, 8, 9, 10, 11, 15, 16, 17)이다. 각 그룹 내에서는 제어 종속성(control dependence)와 데이타 종속성(data dependence)로 인해 더 이상의 태스크 병렬성을 얻을 수 없다. 가령 (2, 3), (3, 4), (12, 13)들에는 제어 종속이 존재하고, 또한 (4, 13)에는 데이타 종속이 존재한다. 여기서 13은 4와 12에 종속적이므로 4와 12의 실행 이후에는 실행 가능하다. 그러나 4는 궁극적으로 3과 2에 제어 종속적이므로 2의 실행 이전에 13의 실행은 불가능하다. 또 4는 2의 실행 결과에 따라 선택적으로 실행되므로 13의 선행 조건이 반드시 4라고 할 수 없다. 따라서 단순 제어 종속과 데이타 종속만으로 실행의 선행 관계를 얻을 수는 없다. [2]에서는 제어 종속과 데이타 종속의 이행 클로저(closure)를 통해 이러한 관계를 파악하고 있다. 본 연구에서는 데이타 종속성을 제어 종속 서브 그래프간의 관계로 다룸으로써, 병렬 루프와 *if* 블럭 구조는 그대로 유지하면서 태스크 병렬성을 추구하고자 한다. 이러한 방식은 규모가 큰 병렬성인 데이타 병렬성은 그대로 유지하고, 병렬 태스크(스레드) 사이의 동기화를 줄임으로써 태스크 병렬성 추구에 따른 추가적인 부담을 적게 하였다.

3.3 CFG와 PCFG

1) CFG

CFG는 프로그램 최적화 정보를 얻기 위한 유용한 표현 형태로 널리 활용되어 왔다[4, 5, 9, 7, 1, 13]. CFG에 대한 정의를 정리하면 다음과 같다[4, 5].

정의 1. CFG=(N, E, $TYPE$)는 방향 그래프로써, 모든 노드는 루트로부터 도달 가능하다. 각 구성요소는 다음과 같다.

- N, 그래프를 구성하는 노드들의 집합. 각 노드는 임의의 순차 계산(기본 블록, 또는 문장)을 나타낸다.
- $E \subseteq N \times N \times LABEL$, 제어 흐름을 나타내는 레이블을 갖는 간선들의 집합.
- $TYPE$, 노드 유형에 대한 사상. $TYPE(n)$은 노드 n의 유형으로 다음 중의 하나이다: START, STOP, PREDICATE, COMPUTE.

모든 CFG는 START와 END의 특별한 2개 노드를 갖는다. CFG 내의 모든 노드는 START로부터 도달 가능하며, 또한 END에 이르는 경로를 갖는다. COMPUTE 유형의 모든 노드는 정확히 한 개의 외향(outgoing) 간선을 갖는다. PREDICATE 노드 유형은 한 개 이상의 간선을 가지며 또한 각 간선은 구별되는 레이블을 갖는다. PREDICATE 노드로부터 나가는 이들 간선들에 붙어 있는 레이블들은 $LABEL$ 집합을 형성한다. PREDICATE 이외의 노드들에 부속된 간선은 "U"(Unconditional) 레이블을 갖는다. 본 논문에서는 혼란의 여지가 없는 한, 레이블을 생략하기로 한다.

2) PCFG

본 논문에서 PCFG는 루프 구조와 *if* 블럭을 그대로 유지하면서 태스크 병렬성을 추구하기 위해 제안되었다. PCFG에는 CFG가 갖고 있는 노드 유형에 몇 개의 새로운 노드 유형이 더 추가되었다. $\Yinyang_R$ 노드는 분산되어 있는 배열에 대한 원격 READ 동작을 의미한다. 앞서 그림 1의 READ*send*와 READ*recv* 를 합쳐서 $\Yinyang_R$ 노드라고 할 수 있다. 마찬가지로 $\Yinyang_W$는 원격 WRITE를 나타낸다. 두 개의 수평 막대 중 위쪽 막대는 READ*send*(또는 WRITE*send*)를, 아래쪽 막대는 READ*recv*(또는 WRITE*recv*)를 의미하며, 긴 지연시간이 발생하는 지점임을 나타내다. {와 } 사이에는 READ 또는 WRITE하고자 하는 배열 이름을 열거한다. 가령, $\Yinyang_R\{x\}$는 배열 x에 대한 원격 READ를 나타낸다. 일반적으로 스칼라는 지역적으로 갖고 있기 때문에, 원격 READ가 발생하지 않는다고 가정한다. PCFG에 대한 정의는 다음과 같다.

정의 2. PCFG=(N, E, $TYPE$)는 방향 그래프로서, 모든 노드는 루트로부터 도달 가능하다. 각 구성요소는 다음과 같다.

- N, 그래프를 구성하는 노드들의 집합. 각 노드는 임의의 순차 계산(기본 블록, 또는 문장)을 나타낸다.
- $E \subseteq N \times N \times LABEL$, 제어 흐름을 나타내는 레이블을 갖는 간선들의 집합.
- $TYPE$, 노드 타입에 대한 사상. $TYPE(n)$은 노드 n의 타입으로 다음 중의 하나이다: START, STOP, PREDICATE, COMPUTE, FORK, JOIN, $\Yinyang_R$, $\Yinyang_W$

PCFG는 CFG의 모든 사항을 그대로 유지하면서 다중스레드를 표현하기 위해 단지 FORK와 JOIN 노드가 추가되

었으며, 또한 통신 지점을 나타내기 위해 $\div_R$와 $\div_W$의 2가지 노드가 추가되었다. CFG에서 PREDICATE는 여러 개의 간선을 가질 수 있는 유일한 노드였다. PCFG의 FORK 노드도 다수의 간선을 가질 수 있지만, 여기에서는 "U"(Unconditional) 레이블만 허용된다. FORK 노드는 여러 개의 스레드를 생성하는, 그리고 JOIN 노드는 여러 개의 스레드를 하나로 합치는 의미를 갖는다. 그림 2는 그림 1에 대한 PCFG를 보여준다. $\div_R$이나 $\div_W$에 도달한 스레드는 통신으로 인한 블럭 상태가 되고, 프로세서는 다른 스레드를 스케줄링함으로써 지연시간 감춤 효과를 얻게 된다.

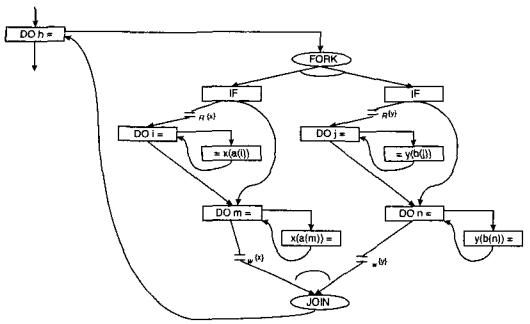


그림 2 PCFG의 예

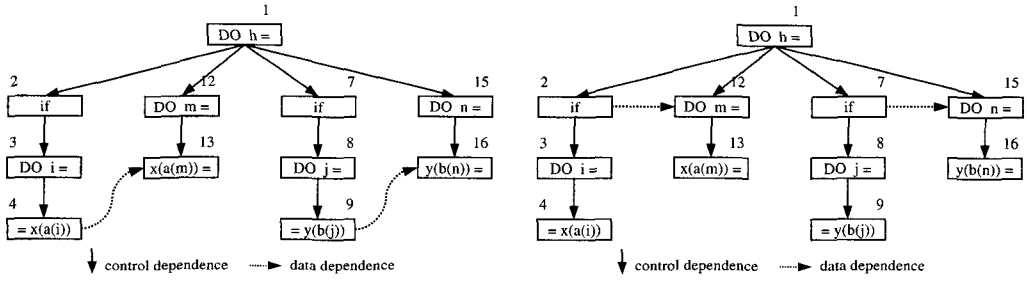
4. PCFG의 생성

PCFG는 CFG에 대한 분석으로부터 출발하여 중간 단계인 HDG(Hierarchical Dependence Graph)를 거쳐 생성된다. HDG는 제어 종속성 그래프에 데이타 종속 간선을 추가한 그래프이다. 여기서 데이타 종속 간선은 단순히 노드 간의 데이타 종속성을 나타내는 것이 아니라, 제어 종속성 서브그래프 간의 데이타 종속을 포괄한다.

4.1 HDG의 생성

노드 y가 노드 x에 제어 종속적이라 함은 노드 x의 실행 결과에 따라 노드 y의 실행이 선택적으로 이루어질 수 있음을 의미하며, $(x \rightarrow_c y)$로 표기한다. *then* 블럭이나 *else* 블럭은 *if* 문에 제어 종속적이라고 할 수 있다. 또한 루프 몸체는 루프 엔트리에 제어 종속적이다. 제어 종속성 그래프(Control Dependence Graph)는 CFG에서 후위-도미네이트(post-dominate)를 이용하여 생성할 수 있다[5, 10]. 데이타 종속성은 동일 메모리 위치에 대한 흐름(flow), 역(anti), 출력(output) 종속성이 있는 경우를 의미하며, 노드 y가 노드 x에 데이타 종속적인 경우 $(x \rightarrow_d y)$로 표기하기로 한다.

그림 3은 그림 1의 (a)로부터 생성된 CDG에 데이타 종속 정보를 추가한 그래프를 보여준다. $(4 \rightarrow_d 13)$와 $(9 \rightarrow_d 16)$의 데이타 종속이 존재하므로 (a)에 이를 점선 화살표로 표기하였다. 그런데 이와 같은 형태의 데이타 종속은 실행 모델에서 유용한 정보를 제공하지 못한다. 가령, 노드 13은 노드 4에 데이타 종속적이므로 노드 4의 실행 이후에 실행되어야 하지만, 노드 4는 실제로 실행이 안될 수도 있다. 왜냐하면 노드 4는 노드 3 및 노드 2에 제어 종속적이므로 이들 노드의 실행 결과에 따라 실행이 결정된다. 한편, 노드 4가 실행되지 않는 경우는 노드 2와 3의 실행을 통해서만 알 수 있으므로, 노드 4의 실행 여부에 무관하게 노드 13의 선행 조건에 노드 2와 3이 포함됨을 알 수 있다. [2]에서는 제어 종속들과 데이타 종속의 이행 클로저(transitive closure) 그래프를 제안하고, 이를 통해 실행 선행 관계를 얻는다. 그러나 이와 같은 선행 관계를 얻더라도, 실제 실행을 위해서는 노드 2, 3, 4로부터 토큰이 전달되어야 하며, 노드 13에서는 토큰 매칭을 통해 선행 조건이 완료되었음을 확인해야 한다. 이와 같은 경우에서는 매 루프 반복에서 동기화가 이루어지므로 데이타 병렬 루프의 이점을 살릴 수 없다. 인터벌(interval) 분석[4] 등을 통해 루프를 하나의 노드로 표현할 경우 데이타 병렬 루프의 이점을 유지할 수 있지만, 여전히 노드 2와 같은 *if* 문으로부



(a) 단순 데이타 종속 정보 추가 (b) 포괄적 의미의 데이타 종속 정보 추가

그림 3 CDG에 데이타 종속 정보 추가

터의 토큰 전달은 피할 수 없다.

본 논문에서는 궁극적으로 (b)의 형태와 같이 데이타 종속 정보를 그래프에 추가함으로써 동기화를 최소화하고자 한다. 여기서 추가된 간선들은 ($G_2 \rightarrow_d G_{12}$)와 ($G_7 \rightarrow_d G_{15}$)에 대한 것이며, G_2, G_{12}, G_7, G_{15}는 각각 노드 2, 12, 7, 15를 루트로 한 서브 그래프들을 의미한다. 서브 그래프 사이의 데이타 종속은 실제로 각 루트 노드들 사이의 간선으로 표현할 것이다. 또한 (b)에서 각 데이타 종속에 대해 반대 방향의 데이타 종속도 역시 존재하지만, 루프 자체에 대한 변환이나 해체는 고려하지 않으므로 이러한 루프 종속(loop-carried dependence)은 표시하지 않는다.

그림 3의 (b)와 같이 데이타 종속 정보를 추가한 그래프는 다음과 같은 과정을 거쳐 얻는다. 먼저 각 노드에 대해 R과 W의 두 가지 집합을 구한다. R은 CDG에서 자기 자신 또는 후손 노드들이 참조하는 변수들의 집합이며, W는 자기 자신 또는 후손 노드들이 정의하는 변수들의 집합이다. 이때 다음과 같은 등식이 성립된다: $Ra = Ra_1 \cup Ra_2 \cup \ldots \cup Ra_n$. 여기서 a_i는 노드 a의 후손 노드를 나타낸다. 마찬가지로 $Wa = Wa_1 \cup Wa_2 \cup \ldots \cup Wa_n$이 성립한다. 그림 4는 R과 W 집합이 표시된 그래프이다. 이러한 R과 W는 CDG에 대한 후위 순회(post-order traversal)을 통해 얻을 수 있다. 단, 원시 프로그램은 구조화 프로그램임을 가정한다. 그림에서 배열과 스칼라는 특별히 구분하지 않고 표현하였으며, 또한 x_a는 $x(a(i))$와 같이 배열 a를 통한 간접 배열 참조를 나타낸다.

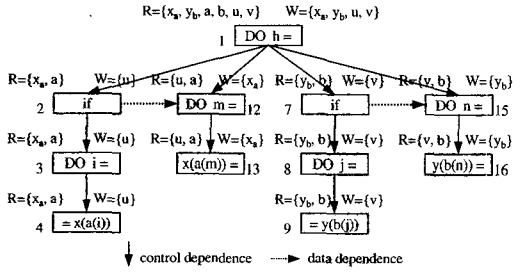
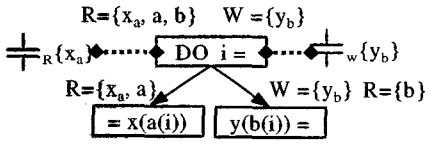


그림 4 R과 W 집합의 추가

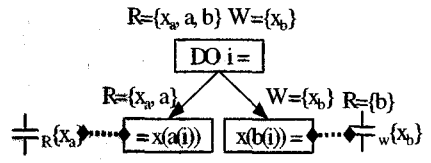
그림 4에서 서브 그래프 사이의 데이타 종속성 여부는 루트 노드로부터 출발하여 순회하면서 형제 노드들 사이의 R, W를 조사함으로써 파악할 수 있다. R과 W 또는 W와 W 사이의 교집합이 공집합이 아니면, 데이타 종속이 존재한다.

다음으로 그림 4의 그래프에 대해 통신 노드를 추가하게 된다. 통신 노드의 추가는 R 또는 W 집합에 통신을 유발하는 데이타의 참조 또는 정의가 포함되어 있는지를 조사함으로써 이루어진다. 최적화된 통신 노드의 추가에 대해서는 다른 많은 연구들이 있으며[3, 13], 본 논문에서는 논의하지 않는다. 여기서는 불규칙 문제를 예로 사용하였으므로, $x(a(i))$와 같은 간접 참조가 있으면 단순히 통신이 필요한 것으로 간주한다.

먼저 루트 노드에 대해 R과 W를 조사하여 통신을 유발하는 변수가 포함되어 있다면, 통신 노드를 추가한다. 그러나 만일 그 변수가 R과 W 모두에 포함되어 있다면, 통신 노드를 추가하지 않고 제어 종속 간선을 따라 하위 노드에 대해 조사를 진행한다. 이 과정은 R과 W 어느 한 쪽에만 동일 통신 유발 변수가 있을 때까지 진행된다. 그림 5는 통신 노드 $\div_R$와 $\div_W$를 첨가하는 예를 보여준다. (a)에서 *do-loop* 노드는 배열 x에 대한 원격 참조와 배열 y에 대한 원격 정의를 갖고 있으므로 이에 대한 통신 노드가 첨가되었다. (b)에서는 *do-loop* 노드가 동일한 배열 x에 대해 원격 참조와 원격 정의를 모두 갖고 있으므로 루프 밖에 통신 노드를 배치할 수 없다. 따라서 하위 노드에 대한 조사를 계속 진행하여 통신 노드를 첨가하게 된다.



(a)



(b)

그림 5 통신 노드 추가 예

마지막으로 이행 종속(transitive dependence)을 제거하고 나면 HDG가 구성된다. 이행 종속의 제거란, ($x \rightarrow_c y$), ($y \rightarrow_d z$), ($x \rightarrow_c z$)의 3개의 종속이 존재하는 경우 ($x \rightarrow_c z$)을 나타내는 간선을 제거하는 것을 의미한다. 그림 4에서는 (1, 12)와 (1, 15)의 간선이 제거될 것이다. 노드 12의 경우를 예로 들면, 노드 12의 실행을 위해서는 노드 1과 2가 선행되어야 하지만, 노드 2가 실행되면 이미 노드 1이 실행되었으므로 (1, 12) 간선을 제거한다. 생성된 그래프는 그림 6과 같다.

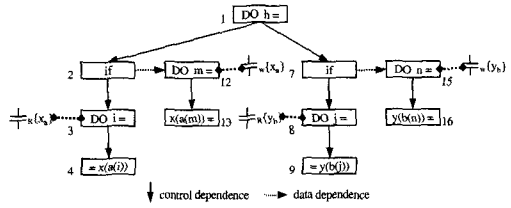


그림 6 HDG

4.2 HDG로부터 PCFG 생성

HDG는 제어 종속성의 계층 구조를 가지면서, 동시에 각 계층 수준별로 데이타 종속성을 가질 수 있는 그래프이다. 또한 HDG는 순환적으로 정의 가능하다. 그림 7은 순환적 정의에 기초한 PCFG 생성 규칙들이다. 만일 HDG의 어떤 노드 n이 다수의 제어 종속 간선들을 갖고 있다면, 이들 간선들에 의해 연결되어 있는 서브 HDG들은 병행적으로 실행 가능하다. (a)는 이러한 경우의 PCFG 생성 규칙을 보여준다. G_i'과 G_j'은 각각 서브 HDG인 G_i와 G_j로부터 생성된 PCFG를 의미한다. 또한 HDG의 어떤 노드 n이 다수의 데이타 종속 간선들을 갖고 있다면, 이들 간선들에 의해 연결되어 있는 서브 HDG들은 병행적으로 실행 가능하다. (b)는 이러한 경우의 PCFG 생성 규칙을 보여준다. G_n은 노드 n을 루트로 하는 서브 HDG를 의미한다. (a)와 (b)에서 G_i와 G_j는 분리(disjoint)되어 있지 않을 수 있는데, 그 이유는 서브 HDG들 사이에서 (c)의 G_k와 같이 다수의 내향 데이타 종속 간선을 가질 수 있기 때문이다. 한편 구조화 프로그램을 가정하기 때문에 다수의 내향 제어 종속 간선은 존재하지 않는다.

HDG에서 통신 노드 $\doteqdot_R$와 $\doteqdot_W$는 일반 노드에 부속된 것으로 처리한다. PCFG 생성 시에는 $\doteqdot_R$은 부속되어 있는 노드의 앞에, 그리고 $\doteqdot_W$는 반대로 부속되어 있는 노드의 뒤에 연결한다.

5. 다중스레드 코드

PCFG로부터 다중스레드 코드를 생성하는 것은 매우 간단한 과정이다. 단지 PCFG를 순회하면서 각 노드로부터 해당 문장들을 생성한다. 예외적인 경우는 FORK, JOIN 그리고 통신 노드들($\doteqdot_R$, $\doteqdot_W$) 뿐이다.

만일 어떤 FORK 노드가 n개의 자식 노드를 갖고 있다면, 다음 문장을 생성한다.

FORK t_1, t_2, ..., t_n;
(t_i는 i번째 스레드에 대한 스레드 식별자)

만일 어떤 JOIN 노드가 n개의 부모를 갖는다면, 다음 문장을 생성한다.

JOIN v, n; (v는 join 변수)

통신 노드의 경우는 다음과 같은 문장을 생성한다.

$\doteqdot_R\{x_1, x_2, ..., x_k\}$ ⇒ READ*send* $\{x_1, x_2, ..., x_k\}$;
READ*recv* $\{x_1, x_2, ..., x_k\}$;

$\doteqdot_W\{x_1, x_2, ..., x_k\}$ ⇒ WRITE*send* $\{x_1, x_2, ..., x_k\}$;
WRITE*recv* $\{x_1, x_2, ..., x_k\}$;

그림 8은 그림 2의 PCFG로부터 생성된 다중스레드 코드이다.

그림 8의 코드는 추상적인 문장을 포함하기 때문에 실제 실행 가능한 코드를 얻기 위해서는 플랫폼 종속적인 변환이 필요할 것이다. 통신 문장들은 MPI(Message Passing Interface)[15] 등과 같은 통신 라이브러리의 함수를 통해 구성할 수 있을 것이다. 또한 FORK 문과 JOIN 문은 Posix thread나 시스템에 내재되어 있는 스레드를 사용하면 쉽게 구현될 수 있을 것이다. FORK 문마다 스레드를 생성하면 부담이 클 수 있으므로, 필요한 최대 스레드 수만큼 미리 생성해 놓고 스케줄링하는 것도 고려할 수 있는 방법 중의 하나이다.

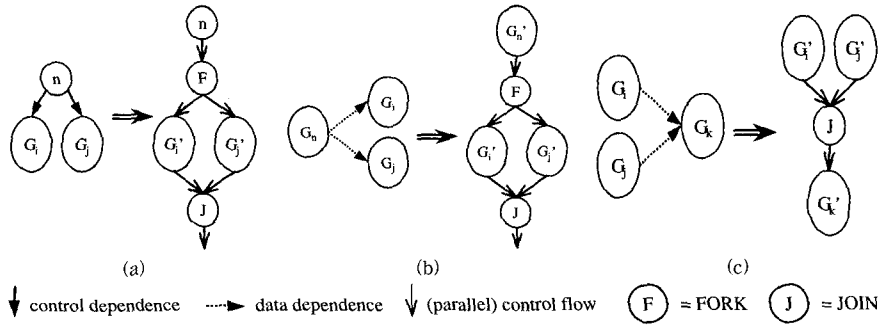


그림 7 HDG로부터 PCFG 생성

```
      DO h = 1, 20
         FORK L2
         IF (mod(h, 2) .eq. 0) THEN
            READsend xa
            READrecv xa
            DO i = 1, 25
               u(i) = x(a(i))
            END DO
         END IF
         DO m = 1, 25
               x(a(m)) = u(m) + m
            END DO
            WRITEsend xa
            WRITErecv xa
            GOTO J1
L2       IF (mod(h, 3) .eq. 0) THEN
            READsend yb
            READrecv yb
            DO j = 1, 25
               v(j) = y(b(j))
            END DO
         END IF
            DO n = 1, 25
               y(b(n)) = v(n) + n
            END DO
            WRITEsend yb
            WRITErecv yb
J1       JOIN j, 2
      END DO
```

그림 8 그림 2의 PCFG로부터 얻은 다중스레드 코드

6. PCFG 예

그림 9는 Hydrodynamics[3]에 대한 PCFG 생성 예를 보여준다. (a)는 원시 프로그램으로서 각각의 내부 루프는 두 개의 문장을 갖고 있다. 이 두 문장 사이에는 종속성이 없으므로 (b)와 같이 루프 분산(loop distribution)[1]을 한다. 루프 분산을 통해 루프들 사이의 종속성이 줄임으로써 보다 많은 태스크 병렬성을 추출할 수 있다. (c)는 (b)로부터 생성된 PCFG를 보여준다. 편의상 각 루프는 L1~L6으로 나타냈다. 그림에서 보듯이 1~3개의 태스크 스레드를 이용할 수 있음을 알 수 있다.

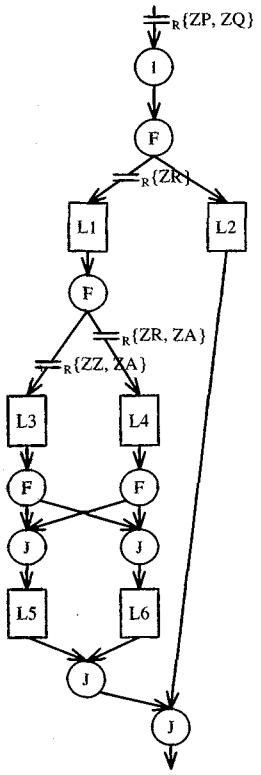
통신 노드는 3.1절과 3.2절에서 설명한 방법에 의해 생성되었지만, 보다 최적화된 통신이 가능하다. 가령, (c)에서 루프 L3와 L4에 각각 필요한 통신은 $\dashv_R${ZZ, ZA}와 $\dashv_R${ZR, ZA}으로써, ZA가 공통적으로 필요하므로 $\dashv_R${ZA}를 FORK 노드 이전으로 올리는 것이 바람직하다. 또한 L1과 L4에 공통적으로 ZR이 필요하므로 이에 대한 최적화도 가능할 것이다. 이 경우 L1과 L4가 필요로 하는 ZR의 요소들이 서로 다르므로 이에 대한 분석이 필요할 것이

```
DO l = 1, time
   DO k = 2, 99
      DO j = 2, 99
         ZA(j,k) = F1(ZP(j-1,k), ZQ(j-1,k), ZR(j-1,k), ...)
         ZB(j,k) = F2(ZP(j-1,k), ZQ(j-1,k), ...)
   DO k = 2, 99
      DO j = 2, 99
         ZU(j,k) = F3(ZZ(j-1,k), ZZ(j+1,k), ZA(j-1,k), ...)
         ZV(j,k) = F4(ZR(j-1,k), ZR(j+1,k), ZA(j-1,k), ...)
   DO k = 2, 99
      DO j = 2, 99
         ZR(j,k) = F5(ZR(j,k), ZU(j,k), ...)
         ZZ(j,k) = F6(ZZ(j,k), ZV(j,k), ...)
END
```

(a)

```
1   DO l = 1, time
2     DO k = 2, 99
3        DO j = 2, 99                                   L1
4           ZA(j,k) = F1(ZP(j-1,k), ZQ(j-1,k), ZR(j-1,k), ...)
5     DO k = 2, 99
6        DO j = 2, 99                                   L2
7           ZB(j,k) = F2(ZP(j-1,k), ZQ(j-1,k), ...)
8     DO k = 2, 99
9        DO j = 2, 99                                   L3
10          ZU(j,k) = F3(ZZ(j-1,k), ZZ(j+1,k), ZA(j-1,k), ...)
11    DO k = 2, 99
12       DO j = 2, 99                                   L4
13          ZV(j,k) = F4(ZR(j-1,k), ZR(j+1,k), ZA(j-1,k), ...)
14    DO k = 2, 99
15       DO j = 2, 99                                   L5
16          ZR(j,k) = F5(ZR(j,k), ZU(j,k), ...)
17    DO k = 2, 99
18       DO j = 2, 99                                   L6
19          ZZ(j,k) = F6(ZZ(j,k), ZV(j,k), ...)
20  END
```

(b)



(c)

그림 9 Hydrodynamics 예

다. 이러한 통신 최적화에는 RSD[16]를 이용한 방법이나, 선형대수적인 방법[13]이 이용될 수 있다. PCFG는 이러한 통신 최적화를 병렬 태스크 스레드들에 대해 적용하기 위해 이용될 수 있다. 이러한 PCFG의 활용에 대한 연구는 향후 연구 과제로 남는다.

7. 결 론

데이타 병렬 프로그램은 매우 큰 병렬성을 용이하게 얻을 수 있을 뿐 아니라, 실행 효율도 우수하다. 그러나, 분산된 데이타로 인해 긴 지연시간을 필요로 하는 통신이 유발되는 단점이 있다. 본 논문에서는 원시 프로그램의 데이타 병렬성을 그대로 유지하면서 긴 통신 지연시간을 효과적으로 감추기 위해 태스크 병렬성을 활용하는 방법을 제안하였다. 이러한 태스크 병렬성을 표현하기 위해 PCFG라는 중간 표현 형태를 제안하였는데, 여기서 각 태스크 스레드는 데이타 병렬 루프들을 포함할 수 있다.

PCFG의 정의는 여러 개의 제어 흐름을 갖도록 CFG 정의를 확장한 것이다. 특별히 스레드 생성과 결합을 위해 FORK와 JOIN의 두 개 노드 유형을 추가하였다. 스레드 동기화는 FORK와 JOIN을 통해서만 이루어지기 때문에 스레드 실행 규칙이 매우 단순하고 실행 부담이 적다. 임의의 두 노드에 대한 데이타 종속 정보를 그대로 이용하는 것은 스레드 실행 규칙을 복잡하게 만들기 때문에, 데이타 종속성은 동일한 제어 종속 조상을 갖는 노드들 사이에 존재하는 것으로 추상화하였다. 이를 표현하기 위해 본 논문에서 HDG(Hierarchical Dependence Graph)를 제안하였고, 이를 통해 PCFG가 생성되도록 하였다.

PCFG는 병렬 태스크를 포함하는 여러가지 최적화 단계에 이용될 수 있으며, 특히 각 스레드에 삽입된 통신 노드들에 대한 전역 최적화를 위해 사용될 수 있다. 이러한 PCFG의 활용에 대한 연구는 향후 과제로 남는다.

참 고 문 헌

[1] David F. Bacon, Susan L. Graham and Oliver J. Sharp, "Compiler Transformation for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, No. 4, December 1994, pp. 345-420.

[2] Milind Girkar and Constantine D. Polychronopoulos, "Extracting Task-Level Parallelism," *ACM Transaction on Programming Languages and Systems*, Vol. 17, No. 4, July 1995, pp. 600-534.

[3] S. Hiranandani, K. Kennedy and C. Tseng, "Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines," *Proc. of the 1992 ACM Intl Conference on Supercomputing*, Washington DC, July 1992.

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

[5] J. Ferrante, K. J. Ottenstein and J. D. Warren, "The Program Dependency Graph and Its Uses in Optimization," *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 3, June 1987, pp. 319-349.

[6] Richard Johnson and Keshav Pingali, "Dependence-Based Program Analysis," ACM Programming Language Design and Implementation, *ACM SIGPLAN Notices, Proceedings of the Conference on Programming Language Design and Implementation*, Vol. 28, Issue 6, June 1993, pp. 78-89.

[7] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill and Paul Stodghill, "Dependence Flow Graphs: An Algebraic Approach to Program Dependencies," *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1991.

[8] Robert A. Ballance, Arthur B. Maccabe and Karl J. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Intrepretation of Imperative Languages," *ACM SIGPLAN Notices, Proceedings of the Conference on Programming Language Design and Implementation*, Vol. 25, Issue 6, June 1990, pp. 257-271.

[9] Milind Girkar, Constantine D. Polychronopoulos, "The HTG: An Intermediate Representation for Program Based on Control and Data Dependences," *Tech. Rep. No. 1046, Center for Supercomputing Res. and Dev.* - University of Illinois, May 1991.

[10] Constantine D. Polychronopoulos, "The Hierarchical Task Graph and its Use in Auto-Scheduling," *Proceedings of the 5th International Conference on Supercomputing*, June 1991.

[11] C. Koelbel and P. Mehrotra, "Programming Data Parallel Algorithms on Distributed Memory Machines using Kali," *Proceedings of the 1991 ACM International Conference on Supercomputing*, June 1991.

[12] J. Wu, J. Saltz, H. Berryman and S. Hiranandani, "Distributed Memory Compiler Design for Sparse Problems," *ICASE Report 91-13*, Hampton, VA, January 1991.

[13] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and N. Shenoy, "A Global Communication Optimization Technique Based on

Data-Flow Analysis and Linear Algebra," *ACM Transaction on Programming Languages and Systems*, Vol. 21, No. 6, November 1999, pp. 1251-1297.

[14] R. v. Hanxleden and K. Kennedy, "A Code Placement Framework and its Application to Communication Generation," *CRPC-TR93337-S*, Center for Research on Parallel Computation, Oct. 1993.

[15] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, June 12, 1995.

[16] D. Callahan and K. Kennedy, "Analysis of Inter-procedural Side Effects in a Parallel Programming Environment," *Journal of Supercomputing*, October 1988.

김 정 환

1991년 서울대학교 계산통계학과 (학사). 1993년 서울대학교 대학원 전산과학과 (이학석사). 1999년 서울대학교 대학원 전산과학과(이학박사). 1999년 ~ 2000년 삼성전자 통신연구소 연구원. 2001년 ~ 현재 건국대학교 자연과학대학 컴퓨터·응용과학부 조교수. 관심분야는 병렬처리, 분산 시스템, 결함 내성, 그룹 통신