COIS 프로그램에 대한 PVM 코드 생성기

우 제 화†

요 약

본 논문에서는 COIS 프로그램을 PVM 환경의 C 코드로 변환해 주는 PVM 코드 생성기를 구현한다. COIS는 Chandy와 Misra에 의해 제안된 UNITY 이론에 바탕을 두고 고안된 병렬 프로그래밍 언어이다. UNITY 이론은 병렬 프로그램을 비결정적 상태 전이 시스템으로 규정한다. COIS는 이 모델을 채용하여 실질적으로 활용 가능한 언어로 설계되었다. PVM은 네트워크로 연결된 컴퓨터들을 하나의 병렬 가상 기계처럼 보이게 하는, 병렬 프로그래밍 환경을 제공하는 소프트웨어이다. 이 PVM 상에서 병렬 프로그램을 작성하기 위해서는 프로그래머가 직접 프로그램의 분할과 매핑 과정을 수행해야 한다. COIS에 대한 PVM 코드 생성기는 이 과정을 수행하지 않고도 병렬 프로그램을 작성하는 것을 가능하게 한다. PVM 코드 생성기는 COIS-to-C 번역기 형태로 구현된다. COIS 프로그램의 구성 요소인 문장의 단위 작업 특성을 고려하여, 태스크들의 교착 상태, 공정성, 병행성 문제등을 해결하기 위한 스케쥴링 방식이 제안된다. 스케쥴된 태스크들은 PVM의 노드 프로세서들로 매핑이 이루어 진다. COIS 예제 프로그램을 작성하고, 그 프로그램이 어떠한 PVM 코드로 변환되는 지를 설명한다.

A PVM Code Generator for COIS Program

Je-Hak Woo†

ABSTRACT

In this paper, we implement a PVM code generator, which transforms a COIS program into an executable C code on PVM environment. COIS is a parallel programming language based on UNITY theory proposed by Chandy and Misra. UNITY theory regards a parallel program as a non-deterministic state transition system. COIS adopts this execution model, and becomes a practical language. PVM is a software which provides a parallel programming environment, making networked computers looked as a parallel virtual machine. In order to describe a parallel program on PVM, a programmer should perform the partition of program and mapping process directly. The PVM code generator makes this process unnecessary. It is implemented as a COIS-to-C translator. Considering the characteristic of atomic operation for a statement in COIS program, we proposes a scheduling scheme solving some problems: deadlock freedom, fairness rule, and concurrency on tasks. Scheduled tasks are mapped into node processors in PVM. We describe some example program, and then show which PVM code is generated.

1. 서 론

PVM(Parallel Virtual Machine)은 기존 하드웨어 상에서의 효율적인 병렬 프로그램 개발을 지원하는 소프트웨어이다.[1, 2, 3] PVM은 여러대의 이종 컴퓨터 시스템들이 마치 하나의 병렬 가상 기계처럼 보이도록 한다. 이를 위해 PVM은 네트워크로 연결된 컴퓨터들에 걸쳐서 메시지 라우팅, 데이터 변환, 태스크 스케쥴링등의 기능을 수행한다. PVM에 의해 제공되는

† 종신회원 : 대진대학교 컴퓨터공학과
논문접수 : 1997년 9월 27일, 심사완료 : 1997년 12월 29일

컴퓨팅 모델은 아주 간단하고 일반적인 프로그램 구조를 가지고 있어서, 프로그램이 직관적인 방법으로 작성되는 것을 가능하게 한다. 사용자는 응용 프로그램을 여러개의 태스크들이 협력하는 형태로 분할해야 하며, 이 태스크들은 PVM에서 제공되는 라이브러리 함수들을 이용하여 프로그램의 초기화와 종료 과정, 그리고 태스크들 사이의 통신 및 동기화를 이루게 된다. 이러한 간단한 프로그래밍 인터페이스와 가상 기계 개념은 PVM이 과학 기술 계산의 병렬 처리 분야에서 널리 확산되는 근본적인 이유가 되고 있다.

그러나, PVM에 의해 제공되는 병렬 프로그래밍 방식은 낮은 수준의 메시지 전달 프리미티브들을 사용하는 것으로, 프로그래머가 병렬 실행을 위한 데이터 및 기능상의 분할 작업과 특정 컴퓨터 구조에 맞는 매핑(mapping) 과정을 직접 기술해야 한다는 문제점을 내포하고 있다. 이러한 문제점은 병렬 프로그래밍을 어렵게 만드는 중요 요인으로 작용한다. 따라서 네트워크로 연결된 기존 컴퓨터들을 병렬 처리를 위한 플랫폼으로 사용하게 해 주는 PVM의 장점을 활용하기 위해서는 PVM 환경을 지원해 주는 별도의 프로그래밍 환경이 필요하다. 이를 위해서 환경 개선을 위한 연구가 세가지 방향으로 진행되고 있다. 첫째는 프로그래밍 실행 환경을 그래픽 인터페이스로 구현하는 것이다.[4, 5] 이 방식은 PVM 환경의 이용을 쉽게 해주는 효과를 볼 수 있지만, 아직까지 병렬성을 도출하는 문제는 프로그래머의 몫으로 남게 된다. 둘째로 시도되고 있는 것은 부가적인 계층을 PVM 위에 두는 것이다. 예를 들어 객체나 프로시져 라이브러리를 사용하여 프로그래밍 작업이 이루어 지게 하거나,[6, 8] 공유 메모리 프로그래밍이 가능하도록 환경을 변환하여 제공하는 것이다.[7] 끝으로, 기존의 병렬 컴퓨터 상에서 구현되어진 병렬 프로그래밍 환경이나 언어들을 PVM 환경에서 동작하도록 구현하는 것이다. 현재 병렬 프로그래밍을 위하여 많이 사용되고 있는 언어는 Fortran에 기반을 두고 발전된 것들이기 때문에, Fortran 기반의 프로그래밍 도구를 PVM 상에서 구현하고 있다.[9, 10] 이외에 실험적인 언어들을 기반으로 한 구현이 시도되고 있다.[11] 그러나, 이러한 접근 방식이 병렬 프로그래밍을 하부 구조로부터 완전히 독립시키 지는 못하고 있다. 따라서 프로그램의 병렬성을 컴파일러나 실행 시간 시스템이 자동으로 찾아 주는 내재적인 병렬 프로그래밍 언어의 개발이 계속 시도되어야 한다.

COIS(COncurrent Iterative Statements)는 병렬 프로그래밍을 위해서, 위에서 언급했던 명시적인 병렬화 작업이 필요하지 않도록 해 주는 병렬 프로그래밍 언어이다.[12] COIS는 Chandy와 Misra에 의해 제안된 UNITY [13, 14]라는 이론을 바탕으로 고안된 언어이다. 이 이론은 프로그래밍을 계속적이고 비결정적이며 반복적인 변환 과정으로 나타내는 계산 모델로, 프로그램의 정확성을 위한 자체적인 검증 시스템을 포함한다. 이러한 모델은 병렬 프로그램을 비결정적 상태 전이 시스템으로 볼 수 있게 해주며, 프로그램의 실행이 하나의 전역 상태와 그에 대해 비결정적으로 작용하는 병행 전이들(concurrent transitions)로 보이게 해 준다. 따라서 프로그램 내에 명시적인 프로세스의 개념이 존재하지 않는다. COIS는 이러한 UNITY의 계산 모델을 그대로 채용하고 있다. COIS 프로그램은 여러개의 문장들(statements)로 구성되며, 각 문장은 하나의 상태 전이를 나타낸다. 문장은 동시에 실행되는 한 개 이상의 치환식들(assignments)로 표현된다. 각 문장의 동시적인 치환식들은 여러 변수들을 읽거나 쓸 때 단위 트랜잭션(atomic transaction) 방식으로 실행된다. 그래서, 변수들은 상호 배타적으로 사용되어야 한다. 즉, 한 문장이 변수들을 사용하고 있는 동안, 다른 문장들은 그 변수들에 접근해서는 안되는 것이다.

COIS 프로그램은 전역 상태를 계속 유지해야 하기 때문에, 공유 메모리 구조상에 구현하는 것은 쉽게 이루어 질 수 있다. 그러나, 분산 메모리를 갖는 병렬 컴퓨터에서 COIS 프로그램을 실행시키기 위해서는 공유된 전역 상태를 분산시키는 문제가 해결되어야 한다. 이를 위해서 작업의 분할과 매핑, 그리고 분산 종료등에 대한 연구가 이루어 져야 한다. COIS는 이에 대한 연구를 바탕으로 분산 메모리 구조를 갖는 하이퍼큐브 다중컴퓨터상에서 구현되었다.[12] PVM은 논리적으로 분산 메모리 환경을 제공하기 때문에, 하이퍼큐브와는 다른 위상(topology)을 가질 지라도 근본적으로 동일한 특성을 갖고 있다. 따라서, COIS는 유사한 방식으로 PVM상에서 구현될 수 있다.

본 논문에서는 PVM을 위한 내재적 병렬 프로그래밍 환경을 구축하기 위하여, COIS 프로그램을 PVM

에서 실행가능한 코드로 변환해 주는 PVM 코드 생성기(code generator)를 구현한다. 이 코드 생성기는 COIS-to-C 번역기로서의 역할을 수행한다. 번역 과정의 기본 방식은 각 문장을 하나의 태스크로 간주하는 것이다. 이렇게 생성된 태스크들은 병렬 실행의 가능성을 갖고 있지만, 문장의 단위 작업(atomic operation) 조건을 준수하기 위하여 태스크들의 공유 변수 사용 형태가 분석되어야 한다. 이러한 제약 조건은 태스크들 사이의 병렬성을 제한하게 되며, 상호 배타적 실행을 위한 방안이 필요하게 된다. 번역기에 의해 생성되는 PVM 코드는 정적으로 각 프로세서에 매핑되어 실행되는 SPMD(single program, multiple data) 형태의 명시적 병렬 프로그램이다.

본 논문의 2절에서는 COIS 병렬 프로그래밍 언어의 설계 사항 및 간단한 예제를 기술한다. 다음, PVM 코드 생성기 구현을 위한 고려 사항과 그에 대한 해결책이 3절에서 다뤄진다. 4절은 PVM 코드 생성기의 정상적인 기능 수행을 보이기 위해 응용 프로그램을 제시하고, 생성된 결과 코드를 설명한다. 끝으로 5절에서 결론이 이루어 진다.

2. COIS의 설계

2.1 COIS 개요

COIS는 UNITY 이론에 바탕을 둔 병렬 프로그래밍 방식을 지원하는 병렬 프로그래밍 언어이다. COIS는 UNITY의 동작 체계인 비결정적 상태 전이 모델을 그대로 채용한다. 따라서 프로그래머는 프로그램의 초기 상태와 그에 대한 가능한 전이들만을 기술해 주면 된다. 이렇게 이루어지는 프로그래밍은 특정 하드웨어 구조를 가정하고 있는 것이 아니기 때문에 하드웨어 독립적인 프로그래밍이 가능해 진다. COIS 프로그램을 작성하기 위한 최소한의 구성 요소는 변수 선언, 종료 조건의 지정, 다중 치환식을 갖는 문장들로 이루어 진다. 이 세가지 기본 구성 요소는 각각 **declare, terminate, assign**의 세 섹션에서 기술된다.

declare 섹션은 프로그램에서 사용되는 변수들의 형과 이름을 지정하기 위하여 사용된다. COIS는 한정 변수(bound variable)를 제외한 모든 변수들에 명시적 선언 규칙을 적용한다. COIS에서 변수를 선언하기 위해 사용되는 구문은 C 언어와 유사한 구조를 가진다. 변수가 가질 수 있는 기본 형은 int, char, float, double 형이 지원되며, 구조형으로는 배열형이 가능하다. 함수의 원형도 **declare** 섹션에서 미리 선언될 필요가 있다. 수식에 대한 형을 정의하는 형 시스템은 C 언어의 형 시스템을 그대로 사용한다. **declare** 섹션에서 선언된 변수들은 전역 명칭 영역(global naming scope)을 가진다. 따라서 선언된 변수들은 프로그램내의 전체 위치에서 사용 가능하다.

UNITY에서 고정 점(fixed point:FP)이라 불리는 프로그램 상태는 프로그램의 어떤 문장도 상태를 바꿀 수 없게 되는 그런 특별한 상태를 의미한다. 이 상태가 됐을 때 프로그램은 자체적으로 종료 결정을 내릴 수 있게 된다. 따라서 프로그래머가 종료 조건을 명시할 필요가 없다. 종료 상태를 정의하는 고정 점 조건은 프로그램의 치환문에 나타나는 모든 치환 연산자들을 등호로 바꿈으로써 얻어지는 방정식에 의해 표현된다. 고정 점 상태에 이르게 되면 프로그램의 계속적인 실행이 변수들의 값을 변경시킬 수 없는 상태가 되며, 결국 프로그램을 계속 실행하거나 종료하는 것이 차이가 없게 된다. 이때 프로그램 실행을 중지시키는 것이 가능하다. 그러나, 고정 점이 모든 프로그램에 대해 존재하는 것은 아니다. 예를 들어 k:=k+1 같은 문장에 대한 고정 점 조건은 항상 거짓으로 계산된다. 그래서 이런 조건은 종료 조건으로 사용될 수 없는 것이다. 더구나, 고정 점 조건을 종료 조건으로 사용하는 것은 구현을 비효율적으로 만들 가능성이 있다. 왜냐하면 조건식의 수가 프로그램에 있는 모든 치환식의 개수와 같아지기 때문이다. 내재적인 종료 방식이 프로그래머가 종료를 생각할 필요가 없게 해 주지만, 위의 문제들 때문에 프로그래머가 직접 종료 조건을 제시하는 명시적 종료 방식이 COIS에서는 채택되었다. 물론 그 종료 조건이 프로그램의 고정 점 조건과 일치할 수도 있다. 명시적인 종료 조건은 **terminate** 섹션에 포함된 수식에 의해 정의된다.

COIS에서 계량화(quantification)는 나열 형태를 갖는 문장들과 치환식, 그리고 수식들을 표기하기 위해 사용될 수 있다. 계량화는 각 한정 변수(bound variable)에 대한 범위를 지정하는 한정 목록(bound list)과 조건식에 의해 표현되는 제한식의 두 부분으로 구성된다. 계량화의 한 인스턴스(instance)는 범위와 제

한식에 의해 정적으로 결정되는 한정 변수들의 가능한 값들의 집합으로 정의된다. 각 인스턴스에 대해 문장이나 치환식, 수식의 한 표현이 얻어질 수 있다. 계량화의 정적인 특성에 따라, 프로그램에서의 데이터 사용 형태를 분석하는 것이 가능하며, 태스크들을 프로세서들로 정적으로 매핑하는 것이 가능해 진다.

COIS에서 사용되는 수식 표기법은 C 언어의 것을 그대로 채용한다. 단 예외적으로 COIS에서 사용되는 치환 연산자는 C와 다른 ':=' 연산자를 사용한다. 프로그램내에 주석문이 가능하며 '/*'와 '*/' 기호에 의해 포함된다. 프로그램은 C 언어에서 제공되는 표준 수학 라이브러리 함수들을 사용할 수 있으며, 이는 목적 코드에 대한 컴파일러가 이를 지원할 것을 가정한 것이다.

2.2 프로그램 구조

COIS 프로그램의 전체 구조는 다섯 개의 섹션들로 이루어 진다. 그 구문 구조를 EBNF로 표기하면 다음과 같다:

coisprog: **program** *program_name*
macro_section
declare_section
initially_section
terminate_section
assign_section
end

여기서, **macro**와 **initially** 섹션은 생략될 수 있다. *program_name*은 이 프로그램의 이름을 나타내는 C 언어 식별자이다. 이 이름은 PVM 코드 생성기에 의해 생성되는 호스트 프로그램의 함수 이름으로 사용된다.

macro 섹션은 프로그램에서 사용되는 상수들과 매크로 함수들을 정의하기 위하여 사용된다. 각 정의의 모양은 = 기호의 왼쪽에 상수나 함수 이름이 나타나고, 오른쪽에 수식이 나타나는 방정식처럼 보인다. 각 매크로 정의는 여러 줄에 걸쳐서 표기될 수 있으며, 정의들사이의 구분은 ';' 기호에 의해 이루어 진다. 변수들을 선언하기 위해 사용되는 **declare** 섹션의 선언들도 매크로의 경우와 마찬가지로 ';' 기호에 의해 구분된다.

프로그램의 **initially** 섹션의 구문 구조는 **assign** 섹션과 동일하다. 단지, 치환 연산자인 ':=' 기호가 등호 기호로 바뀐다는 차이만 있을 뿐이다. 이 섹션에서 정의되는 식들은 몇 개의 변수들을 초기화 시키게 된다. 초기화 연산은 순서적인 방식으로 실행되는 것을 가정한다. 따라서 아직 초기화 되지 않은 변수들을 사용하여 다른 변수를 초기화하는 것은 프로그래머의 책임이다. 초기화 되지 않은 변수들의 기본 값은 0으로 간주 된다.

프로그램의 종료 상태를 나타내는 종료 조건은 **terminate** 섹션에서 정의된다. 그 조건은 논리식에 의해 표현된다. **assign** 섹션은 프로그램의 실제 실행 부분을 정의한다.

2.3 Assign 섹션

assign 섹션은 '[]' 기호에 의해 구분되는 문장들의 집합으로 구성된다. 그 구문 구조를 보면 다음과 같다:

assign_section	: **assign** *statement_list*
statement_list	: *statement*{'[]' *statement*}
statement	: *assignment_statement*
	| *quantified_statement_list*
quantified_statement_list	: '{''[]' *quantification statement_list*'}'
quantification	: *bound_list*['.' *boolean_expr*]'::'
bound_list	: *bound*{, *bound*}
bound	: *variable*'(' *expression*':' *expression*')'

위에 나타난 것처럼 문장은 나열형과 계량형의 두 가지 형태로 표현된다. *statement_list*는 문장들의 나열을 표현한다. *quantified_statement_list*는 *quantification*에 의해 *statement_list*의 계량화를 정의한다. *quantification*에 포함된 변수는 한정 변수라 하며, 그 값은 주어진 범위와 *boolean_expr*에 의해 선택적으로 제한된다. 계량화 인스턴스는 이 한정 변수들이 갖는 값들의 집합으로, 한 인스턴스의 값들이 *statement_list*에 의해 정의된 문장들의 한정 변수들에 대입돼, 한 개의 문장을 생성해 낸다. 따라서, 계량형의 표현 방식은 일정 형태를 갖는 많은 문장들의 표현에 적합하다. 한정 변수들의 적용 영역은 중괄호 '{'와 '}' 사이로 제한되며, **declare** 섹션에서 선언될 필요가 없다.

인스턴스 집합은 유한해야 하며, 정적으로 결정되어야 하는데, 이는 프로그래머의 책임이다.

COIS의 한 문장은 동시에 실행되어야 하는 여러개의 치환식들로 구성될 수 있다. 이 치환 문장은 '//' 기호에 의해 구분되는 한 개 이상의 치환식들을 포함할 수 있다. 그 구문 구조는 다음과 같다:

assignment_statement : *assignment_component*
{'//' *assignment_component*}
assignment_component : *enumerated_assignment*
| *quantified_assignment*
enumerated_assignment : *variable_list* := *expr_list*
expr_list : *simple_expr_list*
: *conditional_expr_list*
simple_expr_list : *expression*{, *expression*}
conditional_expr_list : *simple_expr_list* **if** *boolean_expr*
{'~' *simple_expr_list* **if** *boolean_expr*}
quantified_assignment : '{'// *quantification assignment_statement* '}'

위에서 *assignment_component*는 몇 개의 치환식들로 구성될 수 있으며, *statement_list*와 마찬가지로 나열형과 계량형 두가지 형태로 표현될 수 있다. 치환식의 오른쪽 부분이 *conditional_expr_list*이고 그 목록의 여러 조건식들이 참이 될 때는, 그 조건에 연결되는 모든 *simple_expr_list*들의 결과가 일치해야만 한다. 따라서 선택 문제는 발생하지 않는다. 그 *conditional_expr_list*의 모든 조건들이 거짓이라면, 치환되는 변수들은 값이 전혀 변경되지 않는다.

2.4 간단한 프로그램

이절에서는 COIS 프로그래밍의 개념적인 이해를 돕기 위해 간단한 예제 프로그램을 제시해 본다. 이 프로그램은 한 그룹의 모든 사람들에 대해 가능한 회의 시간들중 가장 빠른 시간을 찾아 내는 문제를 해결한다. 프로그램에 대한 기본적인 특성은 참고문헌 [13]에서 기술된다. 프로그램에서 시간은 양의 정수 값으로 표현된다. 예제 프로그램에서는 구조를 단순하게 하기 위해서 F, G, H의 세명으로 구성되는 그룹을 다룬다. 함수 f, g, h는 각각 F, G, H 세명에 관련된다. 이 함수들은 시간에 대한 시간 값 계산을 정의한다. 임의의 시간 t에 대해, $f(t) \geq t$는 F가 $f(t)$ 시간에 회의를 참석할 수 있으며, $t \leq u < f(t)$인 시간 u에는 참석할 수 없음을 의미한다. 그래서 $f(t)$는 F가 시간 t 이후에 참석할 수 있는 가장 이른 회의 가능 시간을 의미한다. (그림 1)은 이 문제를 풀기 위한 COIS 프로그램 *meeting*을 보여 준다. 이 프로그램에서 배열 f[ ], g[ ], h[ ]는 각각 함수 f, g, h를 나타낸다. 가능한 시간 값은 0부터 23까지의 값들을 가정한다. 마지막 가능한 회의 시간 이후의 시간에 대해서는 배열은 24라는 시간 값을 갖는다. 원소들의 개수가 TN이기 때문에, 배열 마지막 원소인 f[TN-1]은 반드시 24를 값으로 갖는다. 이 값은 프로그램 상태의 발산을 막기 위해 사용될 수 있다. 그래서, 그 값은 사람들이 공통되는 회의 시간을 갖고 있지 않음을 의미한다.

```
program meeting

macro
        TN = 25;

declare
        int   t;
        int   f[TN], g[TN], h[TN];      /* schedule for each person */

initially
        t = 0

terminate
        t == f[t] && t == g[t] && t == h[t]

assign
        t := f[t] []
        t := g[t] []
        t := h[t] []
end
```

(그림 1) 간단한 COIS 프로그램 *meeting*
(Fig. 1) A simple COIS program *meeting*

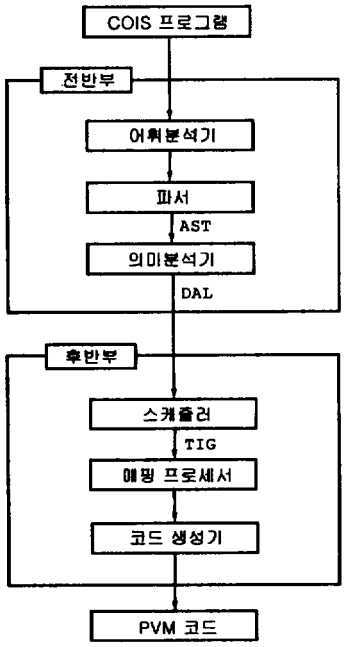
프로그램은 세 개의 치환식으로 구성되어 있음을 알 수 있다: t:=f[t], t:=g[t], t:=h[t]. 이 프로그램에 대한 계산 과정은 치환문들 중 하나가 비결정적으로 선택되고, 그 문장이 상호 배타적으로 실행되게 한다. 이러한 선택 과정은 공정성 규칙(fairness rule)을 만족하도록 이루어 져야 한다. 즉, 각 문장은 무한히 자주 실행될 수 있어야 한다는 것이다. 프로그램에서 예상 회의 시간인 변수 t는 0으로 초기화 된다. 그룹의 한 사람이 현재의 회의 시간을 지킬 수 없으면, 예상 시간을 자신의 가능 시간으로 증가시킨다. 궁극적으로, 프로그램은 t 값이 변화되지 않는 종료 조건을 만족하는 상태에 다다르게 된다. 이 상태에서 시간 t는 공통되는 회의 시간을 갖게 되며, 프로그램은 종료될 수 있다. 공통되는 회의 시간이 존재하지 않으면, 변수 t는 24란 값을 갖게 될 것이다. 이 경우에도 또한

프로그램은 종료될 수 있다.

3. PVM 코드 생성기의 구현

3.1 COIS-to-C 번역기의 구성

이절에서는 PVM 코드 생성기의 구현에 대하여 기술한다. PVM 코드 생성기는 COIS 프로그램을 PVM 환경에서 실행 가능한 C 프로그램으로 변환하는 COIS-to-C 번역기의 형식으로 구현된다. 공정성 규칙에 의해 선택되고, 실행되는 문장들의 집합으로 구성되는 COIS 프로그램을 네트워크로 연결된 컴퓨터들에서 실행시키기 위하여 각 문장을 반복적으로 실행하는 능동적 존재로서 태스크(task)가 정의된다. 이 태스크는 COIS 프로그램으로부터 추출되는 병렬성의 기본 단위로서 사용된다. 그래서, 태스크의 한 반복 실행은 관련 문장을 한번 실행하는 것에 해당한다. 프로그램의 모든 문장은 각각 자신의 태스크에 할당되며, 이 태스크들의 집합이 실행 시간의 프로그램을 구성한다.



(그림 2) 번역기의 구성
(Fig. 2) Organization of translator

(그림 2)는 단계적으로 실행되는 여섯 개의 모듈들로 구성되는 COIS-to-C 번역기를 보여 준다. 번역기의 전반부는 소스 프로그램으로부터 여러 가지 특징적인 요소들을 추출하며, 이 요소들은 추상 구문 트리(abstract syntax tree:AST)와 데이터 접근 목록(data access list:DAL)에 의해 표현된다. 이러한 자료 구조들은 후반부로 전달되고, 후반부는 이들을 바탕으로 실행 시간 환경을 구축하며, PVM 환경에서 실행가능한 C 프로그램을 목적 코드로 생성해 낸다.

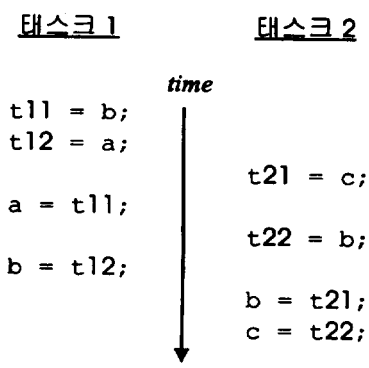
번역기의 어휘 분석기(lexical analyzer) 모듈은 소스 프로그램으로부터 바이트열을 읽어 들이고, 그것을 토큰열로 변환한다. 이 토큰열은 파서(parser)로 전달되고, 파서는 문맥 자유 문법에 기반을 두고 구문 분석을 수행한다. 프로그램의 구문 구조는 AST 자료 구조에 의해 표현된다. 이 구조는 다음 단계인 의미 분석기(semantic analyzer)로 전달되고, 여기서 COIS의 계량화된 문장들이 나열형으로 확장된다. 이 확장은 문장들 사이에 계층 관계가 존재하지 않는 평형 구조의 문장 집합을 생성하게 되고, 그 결과는 AST에 반영되게 된다. 다음으로, 태스크들에 의한 데이터 사용 형태가 분석되는데, 그 결과는 DAL에 기록된다. 스케줄러(scheduler)는 이를 바탕으로 태스크들 사이의 단위 작업 조건을 보장하면서, 병렬성을 최대화시키기 위한 스케줄링 작업을 실행한다. 태스크 상호작용 그래프(task interaction graph:TIG)는 이 단계에서 생성되는데, 계산과 통신 부하에 대한 평가값과 태스크들 사이의 통신 형태 정보가 포함된다. 매핑 프로세서(mapping processor)는 이 TIG를 받아서, 결과 프로그램이 효율적으로 실행될 수 있도록 태스크들을 PVM 환경의 노드 프로세서들에 할당하는 작업을 수행한다. 코드 생성기는 위의 모듈들에서 분석된 결과를 가지고 하나의 호스트 프로그램과 노드 프로그램을 생성해 낸다.

3.2 스케줄링

3.2.1 단위 작업 문제

COIS 프로그램의 작업 모델은 문장이 상호 베타적으로 실행될 것을 요구하기 때문에 하나의 단위 작업 내에 존재하는 중간적인 상태는 다른 문장을 실행하는 태스크들에게 보여져서는 안된다. 이것은 태스크의 문장에 대한 반복 부분이 다른 태스크들에 의해

간섭받지 않고 일관된 상태 전이를 보장해야 한다는 것을 의미한다. 프로그램의 상태는 변수들의 현재 값들에 의해 결정되기 때문에 한 반복 부분에서 사용되는 모든 변수들을 상호 배타적으로 변경할 필요가 있다. 문장은 동시에 실행되는 여러개의 치환식들로 구성될 수 있기 때문에 각 태스크의 단위 반복 부분은 다음 순서를 갖고 실행된다:먼저, 치환식의 오른쪽 부분에서 사용되는 모든 변수들이 읽혀 진다. 다음, 계산 과정이 이루어 지고, 그 결과들이 식의 왼쪽 부분에 있는 변수들에 치환된다.



(그림 3) 단위 작업 조건 위배 예제
(Fig. 3) Example of atomicity violation

태스크들 사이에 단위 작업이 보장되지 않는 경우를 보이기 위하여 다음 두 문장을 포함하는 프로그램을 고려해 보자:a, b:=b, a와 b, c:=c, b. 이 문장들이 태스크 1과 태스크 2에 각각 할당된다고 가정하자. (그림 3)은 잘못된 결과를 야기하는 가능한 실행 형태를 보여 준다. 작업들이 전체 시간 축에서 순차적으로 실행된다고 가정하자. 변수들 t11, t12, t21, t22는 동시 치환식을 구현하기 위한 임시 기억 장소들로 사용된다. 두 개의 태스크들이 변수 b를 공유하고 있기 때문에 그림과 같은 순서로 실행된다면, 변수 a에 있던 값이 소실되게 된다. 이것은 각 문장에 대한 단위 작업이 이루어지지 않았기 때문에 발생하는 문제이다.

3.2.2 분산 다중 데이터 접근 문제

COIS 프로그램으로부터 유도되는 태스크들의 단위 작업 문제는 분산 다중 데이터 접근(distributed multiple data accessing) 문제로 모델화될 수 있다. 즉, 분산 환경에서 여러개의 태스크들이 다중 데이터를 중첩되는 모양으로 반복되게 접근하는 문제인 것이다. 이 문제에 대한 해결책을 찾기 위해서는 다음 제약 조건들에 대한 고려가 필요하다:교착 상태 해결(deadlock freedom), 강한 공정성(strong fairness), 병행성(concurrency), 메시지 교환의 최소화(low message traffic). 교착 상태 해결 문제는 프로그램의 중요한 안정성 문제이다. 모든 태스크들은 프로그램이 종료될 때까지 교착 상태없이 반복적으로 실행 가능해야 한다. 강한 공정성은 모든 태스크들이 궁극적으로 무한히 자주(infinitely often) 실행될 수 있음을 의미한다. 여기서 "무한히 자주"라는 조건은 "연속적으로"라는 조건보다 더 약한 조건으로 볼 수 있다. 그래서 강한 공정성이란 연속적이지는 않지만 궁극적인 실행 보장을 의미한다. 병행성이 높을수록 더 많은 태스크들이 병렬적으로 실행될 수 있다. 태스크들 사이에 교환되는 메시지들의 수가 최소화되면, 프로그램의 성능 향상에 기여할 수 있다. 이절에서는 분산 데이터 접근 문제에 대해 위의 제약 조건들을 만족시키는 정적인 해결책을 고려하고자 한다.

위의 문제들을 풀기 위하여 분산 문제에서 많이 사용되는 토큰(token) 개념을 확장한 캐리어(carrier)라는 객체를 도입한다. 이 캐리어는 프로그램의 각 변수에 대해 하나씩 생성되며, 해당 변수의 현재 값을 포함하고, 그 변수를 사용하는 태스크들 사이를 움직여 다닌다. 그래서, 변수를 공유하고 있는 태스크들중 하나만이 어느 한순간에 해당 캐리어를 소유하고 있게 된다. 태스크는 자신이 사용하는 모든 변수들의 캐리어가 확보되었을 때만 문장의 한 반복 부분을 실행할 수 있다. 그후, 그 태스크에 의해 사용된 모든 캐리어들은 해제된다. 해제된 캐리어들은 각각 다음 태스크들로 보내진다. 캐리어가 움직이는 경로는 번역 과정에 정적으로 결정된다. 각 변수에 대해 그것을 사용하는 모든 태스크들을 연결함으로써 캐리어의 정적 순회 경로가 구해질 수 있다. 해제된 후에, 캐리어는 그 경로를 통해 한번씩 태스크들을 전부 방문하기 전까지는 되돌아 오지 않는다. 캐리어는 분산 환경에서 메시지 전달 방식에 의해 이동된다.

위에서 제안한 방식이 다중 데이터 접근 문제를 풀기 위하여 교착 상태 문제없이 실행될 수 있는 지를

보이기 위하여 n개의 태스크와 m개의 데이터를 갖는 일반적인 경우를 고려해 보자. 여기서 t_i와 d_j를 각각 태스크와 데이터 식별자라고 하자.

정의 1:접근 순서 매트릭스(access order matrix) AOM은 $n \times m$ 매트릭스다. AOM의 k번째 행 벡터 aom_k는 순서 번호 k를 갖는 태스크에 해당된다. 태스크가 행 벡터에 할당되는 순서는 임의로 결정된다. t_i가 d_j를 사용하고, k번째 순서에 실행될 때, $AOM[k, j]$는 t_i의 값을 갖는다. 여기서, $1 \le i \le n$, $1 \le j \le m$, $1 \le k \le n$. 나머지 모든 매트릭스 원소들은 0 값을 갖는다. ■

정의 2:AOM이 결정되었을 때, 전체 태스크들 사이에 존재하는 순서 관계를 전체 순서 관계 $\ll$로 정의한다. 그래서, $t_i \ll t_j$는 t_i의 순서 번호가 t_j의 순서 번호보다 작음을 의미한다. ■

정의 3:AOM이 결정되었을 때, 같은 데이터를 공유하는 태스크들 사이에 존재하는 순서 관계를 부분 순서 관계 $<$로 정의한다. 그래서, $t_i < t_j$는 두 태스크 사이에 $t_i \ll t_j$의 관계가 존재하고, 그들이 같은 데이터를 공유함을 의미한다. ■

보조 정리 1:각 데이터의 캐리어가 부분 순서 관계에 의해 정해지는 경로를 따라 태스크들을 순회하면, 태스크의 실행 순서는 전체 순서 관계에 의해 결정된다. 즉, $t_k \ll t_j$일 때 태스크 t_k가 t_j보다 먼저 실행된다. ■

정리 1:초기에, 각 데이터의 캐리어를 그 데이터를 공유하는 태스크들중 순서 번호상 제일 앞쪽 태스크에 할당하자. 이때, 각 캐리어가 부분 순서 관계에 의해 정해지는 경로를 따라 태스크들을 순회하고, 부분 순서상 마지막 태스크 다음에 제일 앞쪽 태스크로 이동하면, 태스크들 사이에 교착 상태가 발생하지 않는다.

증명:초기에, 캐리어들의 할당이 부분 순서상 제일 앞쪽 태스크들에 이루어 지기 때문에 AOM에 있는 순서 번호 1을 갖는 태스크는 필요한 모든 캐리어들을 할당받게 되고, 따라서 반드시 실행된다. 그래서, 각 캐리어는 부분 순서에 의해 결정되는 다음 태스크로 전달된다. 전체 태스크들이 필요한 캐리어들을 할당받아서 실행 가능하게 되는 지를 보이기 위하여 우선, 순서 번호 k를 갖는 태스크 t_i가 실행 가능하다고 가정하자. $(k+1)$번째 순서에 있는 태스크 t_j는 두종류의 데이터들을 사용한다. 하나는 t_i와 공유되는 것들이고, 다른 하나는 공유되지 않는 것들이다. 공유된 데이터를 사용하는 경우에 $t_i \ll t_j$이므로, $t_i < t_j$이다. 따라서 t_i의 실행이 끝난후에 부분 순서를 따라서 순회하는 캐리어들은 전부 t_j에 전달된다. 공유되지 않는 데이터에 대해서는 그 데이터가 t_j에 의해 처음 사용된다면, 그 캐리어는 초기에 t_j에 할당되었을 것이다. 그렇지 않은 경우에, k번째 이전 태스크들중에 그 데이터를 사용하는 태스크 t_h가 존재하며, $t_h \ll t_i$의 관계를 갖는다. 그래서, $t_i \ll t_j$이므로 $t_h \ll t_j$의 관계가 성립한다. 따라서, $t_h < t_j$의 관계가 성립하고, t_h의 실행이 이미 이루어진 상황이므로 그 데이터에 대한 캐리어는 이미 t_j에 전달되어 있는 상태이다. 태스크 t_i는 일정 시간 이후에 궁극적으로 실행이 완료되고, 공유된 데이터들에 대한 캐리어들은 t_j에 전달될 것이다. 따라서, 태스크 t_j는 필요한 모든 캐리어들을 얻게 되고, 실행 가능하게 된다. 결론적으로 모든 태스크들이 항상 실행 가능하게 되고, 교착 상태는 발생하지 않는다. ■

이 정리는 임의의 AOM이 캐리어들의 정적인 순회 경로를 결정할 수 있고, 그 경로를 통해 캐리어들이 이동한다면, 교착 상태 없이 태스크들이 실행될 수 있음을 나타낸다. 이것은 태스크들 사이의 병행성을 보장하는 것은 아니다. 그러면, 어떻게 병행성이 형성될 수 있는가? 동시에 실행될 수 있는 태스크들이 많이 존재하면 할수록 병행성이 증가하게 된다. 태스크들이 동시에 실행 가능하기 위해서는 그들 사이에 공유되는 데이터가 존재하지 않아야 한다. 또한, 이들이 동시에 실행되기 위해서는 AOM상에서 인접할 필요가 있다.

이 문제를 해결하기 위하여 서로 공유 데이터를 갖지 않는 태스크들을 찾아 내고, 그들의 실행 순서를 가능하면 연속적이 되도록 AOM의 행 벡터 위치를 바꿀 필요가 있다. 그래서, 캐리어가 순회하는 경로가 바뀌게 된다. 이 문제는 가로 좌표에 제한되는 스트립 패킹(abscissa-constrained strip packing) 문제에 의해 모델화 될 수 있다.[15] 이 문제에서 가로 좌표는 프로그램의 각 데이터에 해당되고, 그 데이터를 사용하는 태스크는 사각형에 의하여 모델화 될 수 있다. 이때, 그 사각형들을 총 높이가 최소화 되도록 사각 통 속에 쌓는 문제로, 그 쌓여진 순서에 의해 태스크 실행이 스케쥴링되는 것으로 생각할 수 있다. 그래서, 사각형의 폭에 해당되는 가로 좌표를 데이터로, 높이를 그 태스크의 실행 시간으로 보면 스트립 패킹의 총 높이

```
LWF(ℛ,ℒ,P)
  ℛ : a set of rectangles
  ℒ : a set of lowest-well-fit rectangles
  P[] : an array for implementation of packing function
1 o := 1;
2 while ℛ ≠ ϕ do
3    ℒ := ϕ;
4    while ℒ = ϕ do
5       for each rectangle in ℛ do
6          if it is lowest-well-fit then
7             insert it into ℒ;
8          end if
9       end
10      if ℒ = ϕ then
11         grow current lowest well to next level;
12      end if
13   end
14   r := prefer(ℒ);
15   delete r from ℛ;
16   P[r] := o;
17   o := o + 1;
18   update current outline;
19   update current lowest well;
20 end
```

(그림 4) *LWF* 알고리즘
(Fig. 4) *LWF* algorithm

는 스케줄의 총 실행 시간(makespan)에 해당된다.

스케줄링을 위하여 각 태스크를 사각형으로 표현하고 (그림 4)의 *LWF* 알고리즘을 적용할 수 있다.[15] 이 알고리즘에서 패킹을 위한 사각형을 선택하는 과정에서 현재까지 패킹된 사각형들에 의해 형성되는 외곽선으로부터 가장 낮게 존재하는 우물 모양에 적합한 사각형을 찾게 된다. 이런 특성을 만족하는 사각형들의 집합이 알고리즘에서 $\mathcal{L}$에 의하여 표현된다. 이 집합에 속하는 사각형들이 여러개일 때 그들중 하나를 선택하기 위하여 선호 함수 *prefer()*가 정의되어야 한다. 이 함수는 여러 규칙에 의하여 구현될 수 있으나, 본 논문에서는 폭과 높이가 가장 큰 사각형을 우선적으로 선택하는 규칙을 사용한다. 각 사각형이 패킹될 때마다 그 사각형에 해당하는 태스크의 순서 번호가 1번부터 순서대로 결정된다. 따라서 알고리즘이 종료된 후, 새롭게 결정된 순서 번호에 의해 *AOM*을 재구축할 수 있다. 이렇게 만들어진 *AOM*은 각 캐리어의 정적 순회 경로를 설정하는데 이용된다.

```
TASK()
1 while TRUE do
2   if all carriers acquired then
3      excute a statement;
4      send each carrier to the next task in AOM column;
5   end if
6   receive carriers from any task;
7 end
```

(그림 5) 실행 시간에 태스크의 동작 형태
(Fig. 5) Operation pattern of a task at run-time

3.2.3 TIG의 생성

(그림 5)는 태스크의 동작 형태를 보여 준다. 태스크는 사용되는 데이터에 대한 모든 캐리어들이 얻어졌을 때 자신의 반복 부분을 실행한다. 실행후에, 각 캐리어는 *AOM*에 의해 결정된 정적 순회 경로를 따라 다음 순서의 태스크에 전달된다.

스케줄러는 *AOM*으로부터 TIG를 생성한다. 캐리어들만이 태스크들 사이에 전송될 필요가 있기 때문에 *AOM*으로부터 통신 패턴을 발견할 수 있다. TIG의 정점(vertex)은 태스크를 표현하고, 그 가중치는 평가된 계산 부하값을 나타낸다. 각 태스크의 부하는 문장의 한 반복에 걸리는 시간을 계산하여 얻어질 수 있다. TIG에서 두 정점 사이의 간선(edge)은 해당되는 두 태스크 사이에 캐리어가 전송되는 것을 의미한다. 간선에 대한 가중치는 그 간선을 통해 순회하는 캐리어들의 수가 된다.

3.3 매 핑

이 절에서는 병렬 태스크들을 PVM 환경의 노드 프로세서들에게 할당하는 매핑 문제를 다룬다. 매핑의 목적은 프로세서들 사이에 부하 균형을 이루면서 프로세서간 통신을 최소화시키는 것이다. 매핑 문제는 노드 프로세서들의 연결 구조에 의존적이기 때문에 여기서는 버스 형태의 네트워크에 의해 연결된 컴퓨터들을 가정한다. 병렬 태스크들은 스케줄러에 의해 생성된 TIG에 의해 표현된다. TIG에서 정점의 가중치는 태스크의 계산 부하를 나타내고, 간선의 가중치는 태스크들 사이의 통신 비용을 의미한다.

매핑 과정을 처리하기 위하여 본 논문에서는 [16]에서 제안된 *MAPPING*알고리즘을 사용한다(그림 6). 이 알고리즘은 $\mathcal{G}$에 의해 표현되는 n개의 태스크들을

```
MAPPING(G, m)
    G : a task interaction graph
    m : number of processors
    M[1..n] : an array of tasks assigned to processor
    gain[1..n][1..m] : gain[i][j] = g(t_i,p_j)
    state[1..n] : 0 = free, 1 = locked
    history[1..n][1..2] : information on move operation
    temp[1..n] : temporary gain for previous move
1   construct an initial random mapping;
2   for i := 1 to n do
3       state[i] := 0;
4       for j := 1 to m do
5           gain[i][j] := g(t_i,p_j);
6       end
7   end
8   for i := 1 to n do          // until all nodes are moved
9       select free t_d in p_x with maximum gain g(t_d,p_l);
10      M[d] := l, state[d] := 1;
11      history[i][1] := d, history[i][2] := x;
12      temp[i] := gain[d][l];
13      for j := 1 to n do
14          if M[j] = x then
15              gain[j][l] := gain[j][l] + 2c_dj;
16          else if M[j] = l then
17              gain[j][x] := gain[j][x] - 2c_dj;
18          else
19              gain[j][x] := gain[j][x] - c_dj;
20              gain[j][l] := gain[j][l] + c_dj;
21          endif
22      end
23  end
24  choose k to maximize T = sum_{j=1}^{k} temp[j];
25  if T > 0 then       // if an improvement is obtained
26      for j := k+1 to n do
27          M[history[j][1]] := history[j][2];
28      end
29      goto 2;
30  else                // if no improvement
31      for j := 1 to n do
32          M[history[j][1]] := history[j][2];
33      end
34  endif
```

(그림 6) 매핑 알고리즘
(Fig. 6) Mapping algorithm

부하 균형과 통신 비용의 최소화가 이루어 지도록 m 개의 그룹으로 분할하기 위해 사용된다. 분할된 각 그룹의 태스크들은 하나의 프로세서에 할당되어 실행될 수 있다.

3.4 분산 종료

실행 시간에, 노드 프로세서들에 할당된 태스크들은 종료 조건이 만족될 때까지 자신의 문장 반복 부분을 계속해서 실행한다. 종료 상태는 각 태스크가 자신의 종료 조건이 만족되는 지를 검사하므로써 이루어진다. 각 태스크에 할당되는 개별적인 종료 조건은 COIS 프로그램의 **terminate** 섹션에 기술된 전체 종료 조건을 분할해서 얻어진다. 전체 종료 조건은 모든 태스크들에 할당된 개별 종료 조건들의 논리적 ***and*** 에 해당한다. 분산된 종료 조건으로부터 프로그램의 종료를 결정하기 위하여 분산 알고리즘이 필요하다.

본 논문에서는 분산 종료 검출을 위하여 Dijkstra에 의해 제안된 토큰 전달 알고리즘을 사용한다[17]. 프로그램은 종료 검출을 위하여 한 개의 토큰을 사용한다. 이 토큰은 정수 값을 갖는 객체로, 모든 태스크들을 연속적으로 순회한다. 모든 태스크가 그 토큰을 먼저 본 후에 종료 조건을 계속해서 만족하고 있으면, 종료가 검출된 것으로 결론내릴 수 있다. 이 알고리즘을 구현하기 위하여 모든 태스크는 항상 ***red***나 ***blue***의 두가지 색을 갖는다. 각 태스크는 자신의 반복 부분을 실행할 때마다 종료 조건이 거짓이면 자신의 색을 ***red***로 설정한다. 문장의 한 반복이 끝난 후에 태스크는 (그림 7)에 기술된 *DTERM* 알고리즘을 실행한다. 정적 경로를 통해서 토큰을 받게 되는 다음 태스크는 통신을 최소화 시키도록 근접한 태스크로 결정된다. 종료를 검출해 내기 위해서 알고리즘은 토큰이 최소한 두 번 순환하는 것을 필요로 한다. 첫 번째 순환은 모든 태스크들의 색을 ***blue***로 설정하는 것이고, 두 번째는 ***blue***로 색이 유지되고 있는 것을 검사하는 것이다.

```
DTERM(Mycolor)
    Mycolor : Set with red whenever private term. condition becomes false
1   if I have the token and my termination condition satisfied then
2     if token = NTASK then
3       termination detected;
4       goto program halt operation
5     end if
6     if Mycolor = red then
7       Mycolor := blue;
8       token := 0;
9     else
10      token++;
11    end if
12    send the token through the static path;
13  end if
```

(그림 7) 종료 검출 알고리즘
(Fig. 7) Termination detection algorithm

3.5 코드 생성

COIS-to-C 번역기에 의해 생성되는 C 코드는 PVM에서 지원되는 두 개의 메시지 전달 프리미티브를 사용한다: *asynchronous blocking send, asynchronous blocking receive*. 일반적으로 PVM에서 작성되는 프로그램

들은 SPMD 형태로 많이 실행된다. SPMD 프로그래밍에서는 각 노드 프로세서가 동일한 프로그램을 실행한다. 단지, 그것의 노드 식별자에 따라 다른 코드 부분을 실행하는 것이 차이가 있다. 본 논문의 번역기에 의해 생성되는 노드 프로그램도 마찬가지로 SPMD 형식 실행 코드를 갖는다.

번역기는 두 개의 프로그램을 생성하는데, 하나는 마스터(master) 프로세서에서 실행되는 호스트 프로그램이고, 다른 하나는 여러개의 슬레이브(slave) 노드 프로세서들에서 실행되는 노드 프로그램이다. PVM의 구조상 슬레이브에서 실행되는 노드 프로그램들은 마스터의 호스트 프로그램에 의해 시작된다. 호스트 프로그램은 모든 노드 프로그램들로부터 결과 값들이 얻어지면, 프로그램을 종료한다.

호스트 프로그램에 의해 시작된 노드 프로그램은 한 개의 **while** 루프 구조를 가진다. 이 루프안에서 각 태스크의 반복 부분을 순차적으로 실행하고, 다른 노드 프로세서들로부터 메시지를 받는다. 루프는 프로그램이 중지될 때까지 계속된다. 각 태스크는 자신의 모든 캐리어들이 얻어졌을 때만 실행된다. 노드 프로세서들 사이에 전달되는 메시지의 형태는 세가지가 있다: M_CARRIER, M_TOKEN, M_HALT. M_CARRIER 메시지는 각 태스크가 캐리어들을 해제할 때 발생된다. 여러개의 캐리어들이 하나의 메시지에 실려 전달된다. 같은 노드내에 포함되는 태스크들 사이에는 캐리어 전달을 위한 메시지가 생성되지 않는다. M_TOKEN 메시지는 분산 종료 검출을 구현하기 위해 사용된다. 종료가 검출되었을 때, 프로그램은 M_HALT 메시지를 발생시키고, 결과 값을 호스트 프로그램으로 전달하고, 종료한다.

4. 예제 프로그램

4.1 정렬 프로그램

이절에서는 COIS 프로그램의 특성을 잘 나타내는 예제 프로그램을 작성하고, 그에 대한 PVM 코드 생성기의 실행 결과를 살펴본다. (그림 8)은 정수 값들에 대한 정렬을 수행하는 COIS 프로그램 *sort*를 보여준다. 이 프로그램은 10개의 원소를 갖는 정수 배열 A[ ]를 증가 순서로 정렬한다. 정렬된 결과 값은 배열 A[ ]에 저장된다. 프로그램 *sort*는 인접한 원소들의 쌍

```
program sort

macro
        N = 10;

declare
        int A[N];

terminate
        {&& j(0:N-2) ::
                A[j] <= A[j+1]
        }

assign
        {[] i(0:N-2) ::
                A[i], A[i+1] := A[i+1], A[i] if A[i] > A[i+1]
        }
end
```

(그림 8) 예제 프로그램 *sort*
(Fig. 8) An example program *sort*

을 비교하고, 감소 순서인 경우에는 그 값들을 교환한다. **macro** 섹션은 상수 N을 정의한다. 프로그램의 실행 부분은 두 개의 동시 치환식으로 구성되는 하나의 계량화 문장(quantified statement)을 포함한다. 이에 대한 계량화 인스턴스는 0부터 8까지의 정수 집합이다. 한정 변수 i가 각 인스턴스 값으로 대치된다면, 문장은 9개의 나열형 문장들로 확장된다. 하나의 문장이 하나의 태스크로 구현되기 때문에, 프로그램 *sort*에 대해서는 9개의 태스크들이 생성된다. 종료 조건은 모든 배열 원소들이 증가 순서로 나열됐는 지를 검사한다. 이 조건은 프로그램의 고정 점 조건과 일치한다.

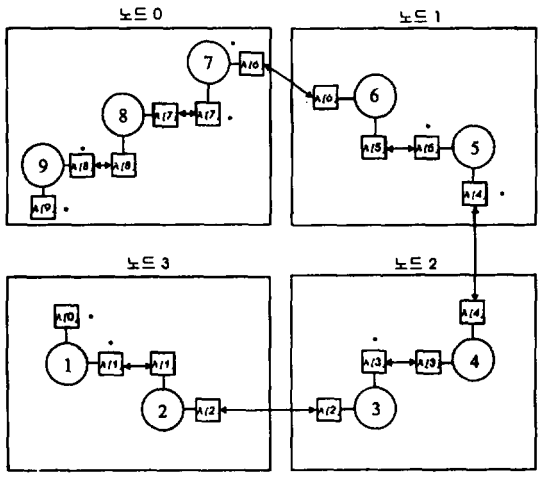
4.2 매핑 결과

(그림 9)는 4개의 노드 프로세서들로 매핑된 *sort* 프로그램의 태스크들을 보여 준다. 1부터 9까지의 태스크들이 4개의 노드들에 부하 균형을 유지하면서 분산된다. 캐리어의 초기 위치가 * 기호에 의해 표시되며, 높은 병행성을 유지할 수 있도록 설정된 모습을 볼 수 있다. 초기 상태에서 태스크 1, 3, 5와 7이나 9 사이에 병렬성이 존재한다. 노드 외부로 나가는 방향성 간선은 캐리어에 의한 메시지 전달이 존재하는 것을 나타낸다.

4.3 PVM 코드

COIS 프로그램은 확장자로 .u를 갖는 파일에 작성된다. 그래서, (그림 8)의 예제 프로그램은 sort.u라는 이름의 파일에 저장된다. 4개의 노드 프로세서에 매핑될 때, PVM 코드 생성기는 호스트 프로그램을

hsort4.c라는 파일로 생성하고, 노드 프로그램에 대해서는 헤더 파일 nsort4.h와 각 노드별 실행 코드를 포함하는 nsort4_0.c, nsort4_1.c, nsort4_2.c, nsort4_3.c 및 main 함수를 포함하는 nsort4.c 파일을 생성한다. 이들중 노드 0에서 실행되는 코드의 핵심 부분을 살펴 보면 다음과 같다:



(그림 9) 네 개의 노드 프로세서들에 대한 *sort*의 매핑 결과
(Fig. 9) Mapping result of *sort* on 4 node processors

```
int    d_c0[10]={
  0, 0, 0, 0, 0, 0, 7, 7, 9, 9, };
node0 ()
{
  int    color=C_RED;
  int    r;
  while (1) {
    if (d_c[6]==7) {    /* Task 7 */
      if (!(A[6]<=A[6+1])) {
        color=C_RED;
      }
      stmt00 (6);
      d_c[6]=0;_carr[0].c_tid=6;_carr[0].c_id=6;
      sendcarrier (0x1, 1);
    }
    if (1) {    /* Task 8 */
      if (!(A[7]<=A[7+1])) {
        color=C_RED;
      }
      stmt00 (7);
    }
    if (1) {    /* Task 9 */
      if (!(A[8]<=A[8+1])) {
      color=C_RED;
      }
      stmt00 (8);
    }
    if (token_get
      && A[6]<=A[6+1]
      && A[7]<=A[7+1]
      && A[8]<=A[8+1]
      ) {
      chkterm(&color, 0x1);
    }
    r=arecvany (&rmsg);
    if (r >= 0) switch (rmsg.cm_type) {
    case M_TOKEN:
      token_get=1;
      token_val=* (int *)mbuf;
      break;
    case M_HALT:
      sendhalt ();
      senddbg (get_ticks());
      r=recvany (&rmsg);
      while (rmsg.cm_type != M_HALT) {
        restore (r/sizeof(carrier_t),
          (carrier_t *)mbuf);
        r=recvany (&rmsg);
      }
      sendrslt();
      if (!hinv) sendhalt();
      return;
    case M_ANGEL:
      restore (r/sizeof(carrier_t), (carrier_t *)mbuf);
      break;
      }
    }
}
```

여기서 특기할 사항은 배열 A[ ]의 원소들중 원소 6만이 다른 노드의 태스크에 의해 공유되기 때문에 캐리어 메시지가 생성되어 전달된다는 것이고, 다른 원소들에 대해서는 캐리어의 획득 여부도 검사할 필요가 없다는 것이다.

이들 프로그램이 실행되는 환경은 Windows 95를 운영체제로 갖는 4대의 PC로 구성된다. 이 PC들은 Ethernet으로 연결되며, PVM의 통신 프리미티브들은 TCP상에서 구현된다. 여기서 이용된 PVM 패키지는 [3]에서 구현된 라이브러리를 사용하였다. 이렇게 구성된 4대의 노드 프로세서상에서 SPMD 형식의 노드 프로그램 *nsort4*가 실행되며, 호스트 프로그램 *hsort4*는 이들중 노드 0에 해당되는 PC상에서 같이 실행된다.

5. 결 론

본 연구에서는 COIS 언어에 의해 작성된 프로그램을 PVM 환경에서 실행 가능한 C 프로그램으로 변환해주는 PVM 코드 생성기를 구현하였다. COIS는 Chandy와 Misra에 의해 제안된 UNITY 이론에 바탕을 두고 고안된 언어이다. COIS를 사용하여 병렬 프로그램을 작성하게 되면, PVM 코딩시에 일반적으로 고려해야 하는 프로그램의 분할과 매핑 문제를 배제할 수 있다. 이밖에 교착 상태 문제나 공정성 확보 문제, 분산 종료 문제등 하드웨어 구조에 의존적인 여러 문제들을 프로그래머가 고려할 필요가 없게 된다.

COIS는 병행 프로그램의 여러 개념을 실험하기 위해 사용될 수 있을 뿐만 아니라, 실질적인 병렬 프로그램을 작성하기 위해 사용될 수 있다. 향후 수행되어야 할 과제는 병렬 프로그램의 실행시 실질적인 성능 향상 효과를 얻기 위하여 PVM을 구성하는 네트워크를 기가비트 이더넷같은 고속의 네트워크로 구축하여 실험하는 것이다. 또한 PVM 코드 생성기의 효율적인 구현을 위하여 스케쥴링 및 매핑에 사용되는 알고리즘들의 최적해를 연구할 필요가 있다.

참 고 문 헌

[1] Al Geist, et al., 'PVM : Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing', The MIT Press, 1994.

[2] V. S. Sunderam, "PVM:A Framework for Parallel Distributed Computing," Concurrency:Practice E Experience, vol. 2, no. 4, pp. 315-339, December 1990.

[3] Markus Fischer and Jack Dongarra, "Another Architecture:PVM on Windows 95/NT," (unpublished manuscript), October 1996.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "PVM and HeNCE:traversing the parallel environment," CRAY Channels, 14 (4), pp. 22-25, Fall 1992.

[5] O. Kramer-Fuhrmann, L. Schafers, and C. Scheidler, "TRAPPER-a graphical programming environment for parallel systems," Proc. Intl. Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, pp. 3-15, October 1993.

[6] C. L. Hartley and V. S. Sunderam, "Concurrent Programming with Shared Objects in Networked Environments," Proc. Intl. Parallel Processing Symposium, pp. 471-478, April 1993.

[7] William W. Y. Liang, "Adsmith:An Efficient Object-Based DSM Environment on PVM," Proc. Intl. Symposium on Parallel Architecture, Algorithms and Networks, pp. 173-179, June 1996.

[8] S. Saarinen, "EASYPVM-An Enhanced Subroutine Library for PVM," Lecture Notes in Computer Science, Vol. 797, pp. 267-272, 1994.

[9] F. Coelho, "Experiments with HPF Compilation for a Network of Workstations," Proc. High-Performance computing and networking, pp. 423-428, April 1994.

[10] Erik H. D'Hollander and Fubo Zhang, "A PVM Code Generator for the Fortran Parallel Transformer," Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 341-344, August 1996.

[11] B. A. Sanders E. Bugnion, M. Gitsels, "Object-oriented Distributed Programming in the Oberon-PVM Environment," Proc. of Joint Modular

Languages Conference, September 1994.

[12] Je-Hak Woo, 'Design and Implementation of Parallel Programming Language COIS Based on UNITY', PhD Thesis, KAIST, 1995.

[13] K. M. Chandy and J. Misra, 'Parallel Program Design: A Foundation', Addison-Wesley, 1988.

[14] E. Shapiro, "The family of concurrent logic programming languages," ACM Computing Surveys, vol. 21, no. 3, pp. 412-510, September 1989.

[15] Je-Hak Woo, "Heuristics for Abscissa-Constrained Strip Packing," Thesis Collection, vol. 4, University of Daejin, pp. 469-478, 1996.

[16] Cheol-Hoon Lee, 'Task Assignment in Parallel and Distributed Systems', PhD Thesis, KAIST, 1992.

[17] E. Dijkstra, W. Feijen, and A. van Gasteren, "Derivation of a termination detection algorithm for distributed computation," Information Processing Letters, vol. 16, no. 5, pp. 217-219, June 1983.

[18] B. W. Kernighan and D. M. Ritchie, 'The C Programming Language', Prentice Hall, 1988.

[19] T. Mason and D. Brown, 'lex and yacc', O'Reilly and Associates, 1990.

우 제 학

1985년 2월 연세대학교 전자공학과 졸업(공학사)

1987년 2월 한국과학기술원 전기 및 전자공학과 졸업(공학석사)

1995년 8월 한국과학기술원 전기 및 전자공학과 졸업(공학박사)

1985년 3월~1991년 1월 삼성전자(주) 미니개발실 주임연구원

1996년 3월~현재 대진대학교 컴퓨터공학과 전임강사

관심분야: 병렬처리, 병렬프로그래밍언어, 분산컴퓨팅