병렬성 검출을 위한 C++ 클래스 설계 및 종속성 분석

황 득 영[†] · 권 오 진[††] · 최 영 근[†††]

요 약

C++ 프로그램을 병렬 실행시키려면 재구성 컴파일러가 함수 호출로 발생하는 요약된 정보를 수집해야 한다. 객체의 참조 매개변수를 사용하는 경우와 객체의 참조를 반환하는 메소드에 대해서 메소드간의 요약된 정보를 발견하여 병렬성을 검출하는 것은 어렵다. 본 논문에서는 순차 C++ 프로그램을 병렬 프로그램으로 변환하기 위해 클래스 인터페이스 내에 메소드간의 관계를 명시하는 병렬 정보 GERINFO절과 순서 정보 SEQ절을 삽입하므로서 명시적인 병렬성을 얻고 재구성 컴파일러는 명시된 정보를 이용하여 프로그램에 내재한 묵시적 병렬성을 얻는 방법을 제안한다. 또한 본 논문에서 제시하는 종속 분석 방법을 이용하여 순차 C++ 프로그램을 병렬 코드로 변환하는 병렬코드 변환기를 구현하였다.

Dependence Analysis and Class Design for Exploiting Implicit Parallelism in C++

Deuk-Young Hwang[†] · Oh-Jin Kwon[††] · Young-Gun Choi[†††]

ABSTRACT

For the parallel execution of C++ program it is required for the restructuring compiler to collect summary information generated by function calls. It is not so easy to find summary information between called methods and extract parallelism for using object reference parameters and returning object reference method.

This paper presents an annotation mechanism which helps the parallelization of a sequential C++ program. GERINFO and SEQ clauses are adopted, respectively, to describe parallel and sequential relations between methods within a C++ class interfaces. Explicit parallel information being provided, a restructuring compiler then proceeds an efficient program parallelization. Parallel code translator converting C++ program into parallel code is implemented using the dependence analysis method described in this paper.

1. 서 론

정보의 처리 내용이 고도화 및 다양화됨에 따라 최근 병렬처리는 객체지향의 개념이 도입되어 연구되고 있다. 절차형 언어에서 데이터의 흐름은 제어의 흐름과 독립되어 있는 반면 객체지향은 데이터와 제어가 함께 이동하는 메시지 호출로서 연산이 이루어진다. 즉, 객체지향은 모듈성이 강하고 모듈 상호간에 메시지를 통해서 연산을 수행하며 객체지향 개념 자체에 병행성(concurrency)을 포함한다. 따라서 객체지향의 개념은 병렬처리에 적합하며 실세계의 개체를 하나의 객체로 모델링할 수 있으므로 문제를 기술하는 것만으로 다수

† 정 회 원 : 삼척산업대학교 컴퓨터과학과
†† 정 회 원 : 산업기술정보원 데이터베이스 사업부 연구원
††† 정 회 원 : 광운대학교 전자계산학과

논문접수 : 1998년 2월 17일, 심사완료 : 1998년 3월 27일

의 계산 모듈을 생성하게 된다. 즉 하나의 객체를 하나의 프로세스로 모델링하게 되면 한 시스템 내부에서는 여러 프로세스가 동시에 존재하게 되며, 이러한 객체지향 모델은 병렬 프로그램을 위한 모델로 자연스럽게 이용될 수 있다[2,17].

본 논문에서는 기존에 작성된 C++ 프로그램을 재구성 컴파일러에 의해 병렬 프로그램으로 변환을 용이하게 하기 위해 클래스 인터페이스에 주석(annotation)을 사용하여 병렬성을 명시한다. 클래스 인터페이스에 병렬 정보를 기술함으로써 재구성 컴파일러는 객체의 참조 매개변수를 사용하는 경우와 객체의 참조를 반환하는 메소드간의 관계(병렬성)를 알 수 있다. 또한, 재구성 컴파일러는 명시적인(explicit) 병렬성 뿐만 아니라 명시된 정보를 이용하여 프로그램에 내재한 묵시적인(implicit) 병렬성도 얻을 수 있다.

지금까지 객체의 병렬성 표현에 관한 방법은 OSI 표준안을 명세하기 위해 제안된 LOTOS, 그리고 병렬성을 추구하는 대신에 순차성을 부여하는 패스 표현(path expression)이 있다[6,14]. 패스 표현은 패스 파스칼(path pascal)에서 사용된 방법으로서 컴파일러는 프로그램에서 사용된 패스 표현을 제어 엔진(control engine)으로 변환하고 메소드의 모든 호출은 제어 엔진을 통해 전달된다. 그러나, 패스 표현은 제어 엔진의 현재 상태에 따라 함수의 실행이 지연 호출될 수 있고, 메소드가 종료했을 때 제어 엔진의 상태를 갱신하기 위해 함수가 종료되었다는 것을 알려줘야 하는 부담이 있다. 또한, 패스 표현은 기존에 작성된 C++ 응용 프로그램을 병렬 프로그램으로 변환하는데 적합하지 않으며 C++에서 메소드간의 관계를 표현하는데도 적합하지 않다.

따라서, 본 논문에서 제안하는 C++의 병렬성 표현 방법은 각 클래스 내에 메소드간의 병렬성과 외부 세계와의 인터페이스를 위해 정규식을 사용하여 순서 정보와 병렬 정보를 클래스 인터페이스에 삽입하여 객체내에 메소드간의 관계를 표현한다. 클래스 인터페이스에 주석을 삽입하여 얻는 이점은 지역화된 병렬성과 사용자의 병렬성 검출에 대한 오버헤드를 줄이고, 재구성 컴파일러에 의한 효과적인 병렬성 검출을 가능하게 한다.

2. 병렬 프로그래밍 언어의 유형

2.1 병렬화 컴파일러를 이용한 언어

대부분의 순차 프로그램들은 루프와 서브루틴, 함수 등을 처리하는데에 많은 시간을 소요하므로 속도향상을 위해 프로그램을 변환시켜 주는 작업이 필요하다. 이러한 기능을 하는 자동변환 컴파일러(restructuring compiler)는 PARAFRASE, KAP, PFC, PTRAN, BULLDOG 등이 있으며 이 컴파일러들은 순차 프로그램을 입력으로 받아들여 병렬 컴퓨터 상에서 벡터(vector) 연산이 가능하도록 벡터 구문으로 변환하거나, 병렬처리가 가능하도록 병렬 구문으로 자동 변환하여 번역한다. 이 컴파일러들 대부분은 포트란 언어의 루프 구조에서 종속(dependence) 분석에 의한 변환 과정을 통하여 벡터화(vectorization)나 병렬화(parallelization)한다[4,7,12].

병렬화 컴파일러에 의한 방법은 프로그래머가 병렬성을 명시하거나 찾아낼 필요가 없다는 장점이 있다. 그러나 순차 C++ 프로그램 자체는 메소드 호출, 즉 함수 호출의 형태를 취하므로 컴파일러가 함수 호출로 발생하는 요약된 정보(summary information)를 발견하는 것은 어렵다. 객체의 참조 매개변수(reference parameter)를 사용하는 경우와 객체의 참조를 반환하는 메소드에 대해서 메소드간의 병렬성을 검출하는 것은 일부분에 지나지 않는다.

2.2 명시적인 병렬 언어

명시적인 병렬 언어는 자원관리, 특정 기계의 세부사항에 대한 책임이 프로그래머에게 있고 프로그래머가 기술한 행위는 병렬로 수행한다. 명시적인 병렬 언어는 크게 두 가지로 분류할 수 있다. 첫 번째는 순차 프로그래밍 언어에 병렬 프리미티브(primitive) 추가에 의해 명시적인 병렬 언어를 만드는 방법으로 여기에 포함되는 언어는 Concurrent Pascal, MultiLisp, QLisp, Concurrent Prolog, PARLOG 등이 있다. 두 번째는 병렬 처리를 위해 새롭게 언어를 설계하는 방식으로 트랜스퓨터에 사용하는 Occam과 SAL, C-Linda, ACTOR 계열의 언어들이 여기에 포함된다.

객체지향 개념에서의 연산은 다수의 객체들이 서로 메시지를 주고받음으로써 발생하고 이러한 개념은 병렬처리 시스템에서 연산처리 방식과 유사하다. 객체지향의 장점을 병렬처리에 응용한 기존의 모델로는 ACTOR, ABCL/1, pC++, pSather, CHARM++ 등이 있다[5,8,9,11,13].

그러나 명시적인 객체지향 언어 모두 새로운 병렬

프로그래밍 언어를 습득해야 하는 오버헤드가 있고, 병렬성의 검출을 사용자에게 요구하는 공통된 단점을 가지고 있다. 기존에 작성된 많은 C++ 프로그램을 병렬 프로그램으로 변환하는 데에는 객체지향의 개념을 사용하였다 할지라도 많은 시간이 소모된다. 이와 같은 문제점을 해결하기 위해 본 논문에서는 클래스 인터페이스에 병렬 정보를 사용함으로써 사용자에게 병렬성 검출에 대한 오버헤드를 줄이고 재구성 컴파일러에 의한 효과적인 병렬성 검출이 가능하다. 또한, 이 방법은 잘 구성된 명시적인 병렬 언어보다는 성능면에서 다소의 차이는 있다고 할지라도 사용자는 단지 클래스 인터페이스내에 병렬성 정보만을 기술하면 되고 프로그램에 내재한 병렬성은 컴파일러가 검출하므로 쉽게 병렬 프로그램으로 변환할 수 있다.

3. C++의 데이터 공유 방식과 병렬성 표현의 문제점

3.1 메소드에 의한 프로그램 코드의 공유

대부분의 순차 C++ 컴파일러에서는 (그림 1)과 같이 동일한 클래스에 속하는 객체 각각에 대해 별도의 멤버 데이터를 가지고 있지만 메소드는 클래스내에 하나만 존재하여 이를 공유한다. 따라서 순차 C++ 프로그램을 객체 단위로 병렬 실행한다고 가정하면 문제가 발생한다.

순차 C++ 컴파일러는 자기 참조(this pointer)를 사용하여 각 멤버함수를 변환한다. 각 클래스 멤버 함수는 묵시적으로 자신의 클래스형에 대한 this 포인터 [illegible] 정적 메모변수를 취하고, 멤버 함수가 호출될 때 클래스 객체의 주소를 멤버 함수에 건네준다[16].

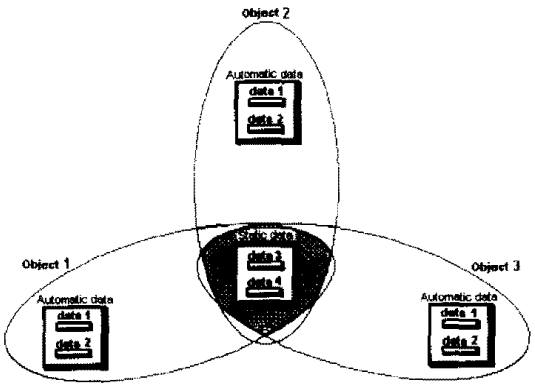
따라서, 컴파일러가 이러한 순차 C++ 프로그램을 병렬 프로그램으로의 변환시 서로 독립적인 객체간에 병렬 실행을 하지 못하고 상호배제로 실행되어야 하는 문제점을 내포하고 있다. 또한 상속의 장점인 상위 클래스(super class)와 하위 클래스(sub class) 사이의 프로그램 코드의 공유는 불가능하게 된다. 이러한 문제점을 해결하기 위한 한가지 방법으로 순차 C++ 프로그램에서는 사용하지 않는 증식 함수(proliferate function)를 사용하여 해결한다. 즉 한 클래스로부터 존재하는 인스턴스 수만큼 복사된 코드를 가져야 한다.

3.2 정적 멤버 데이터

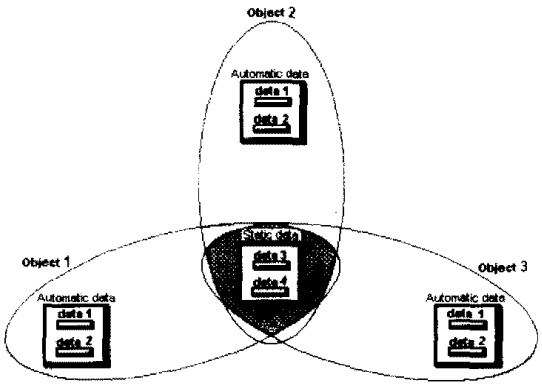
C++에서는 동일한 클래스에 속하는 인스턴스가 정보를 공유하는 수단으로 정적 멤버(static member)를 제공한다[3,15]. (그림 2)는 정적 멤버에 의한 데이터의 공유를 보여주고 있다. 이 정적 변수를 액세스하기 위해서는 상호배제(mutual exclusion)로 실행되어야 하므로 병렬성을 떨어뜨리게 된다. 즉, 정적 데이터는 모든 객체에 의해 공유되는 변수이기 때문이다. 따라서 정적 변수는 가능한 한 사용하지 않는 것이 병렬성 표현이나 구현에 바람직하다.

3.3 포인터

클래스의 데이터 멤버로서 포인터를 사용하고 (그림 3)과 같이 HerObject와 MyObject는 클래스의 데이터 멤버를 공유한다고 가정하자. 이때, 한 객체의 수정



(그림 1) 메소드에 의한 프로그램 코드의 공유
(Fig. 1) Sharing of Program Code by Method in C++



(그림 2) 정적 멤버 데이터에 의한 자료의 공유
(Fig. 2) Sharing of Data using Static Member Data in C++

은 다른 객체에 영향을 미치게 되어 부작용(side effect)이 발생된다. 객체의 메소드 호출로 인한 부작용은 객체 그 자체에 제한되는 것이 아니라 다른 객체에 영향을 미치게 되어 객체지향의 장점을 이용할 수 없게 된다. 즉, HerObject와 MyObject는 서로 틀린 객체임에도 불구하고 HerObject의 수정은 MyObject의 내용을 수정하게 되어 각각의 객체는 다른 객체에 대해 독립성을 상실한다.

(그림 3) 포인터를 사용하는 경우의 부작용
(Fig. 3) Side Effect in case of using Pointer

프로그램내에 포인터를 사용하는 경우 순차 C++ 프로그램에서는 부작용이 발생하더라도 올바른 결과를 생성할 수 있으나 순차 프로그램을 병렬 프로그램으로 변환시 병렬 프로그램 고유의 비결정성(nondeterminism)으로 인하여 올바른 결과를 보장할 수 없다. 포인터를 사용하여 복잡한 자료구조를 표현하는 경우 컴파일러가 사용자에 의해 기술된 자료구조를 알고 이를 해결해 주는 방법은 없다. 이와 같은 문제점을 제거하기 위해 포인터를 사용하는 경우는 인터페이스내에 할당 연산자, 즉 operator=()를 사용자가 기술해야 한다.

3.4 참조 변수

메소드 p는 형식 매개변수 FV를 갖는다고 가정하자. p(FV)에 의해 간접 호출되는 메소드 IDC(p,FV)는 아래와 같다.

$$IDC(p,FV) = DC(p,FV) \bigcup \sum_{\forall q_i} IDC(q_i, FV_i)$$

DC(p,FV)는 p내에 FV's 메소드가 직접 호출되는 집합을 표기한다. q_i는 p에서 호출되는 집합, FV_i는 q_i에 전달되는 p의 형식 매개변수 집합, $IDC(q_i,FV_i)$는 q_i의 간접 호출 집합을 표기한다.

```
Indirect_Example(OBJECT& object1, OBJECT& object2)
{
  object1[1] = data1;
  object2.InRect(object1,y) = data2;
}
```

위 프로그램은 간접 함수 호출의 예를 나타내며, 메소드 object1.[](1).=는 object1.[](1)의 호출에 의해 반환되는 객체의 메소드로서 간접 호출된다. object1[1]의 메소드 호출은 아니다. 그리고 object2.subm(object1,y).=은 object1이나 object2의 메소드가 아님에도 불구하고 간접 호출된다. <표 1>은 p(V)호출에 의해서 발생하는 간접함수 호출의 요소를 보여주고 있다. object1.[](1).=호출과 object2.subm(object1,y).=의 호출은 어떤 객체의 메소드 호출인지 불명확하고 발견하는 것도 어렵다. 즉, 간접 함수 호출로 인해 메소드간의 관계를 정확히 분석하는 것은 어렵다.

<표 1> 간접 함수 호출의 요소
<Table 1> Element of Indirect Function Call

집 합	내 용
FV	{object1,object2}
DC(p,FV)	{object1[](1), object2.InRect(object1,y)}
q_i(0<i≤4)	{object1[](1)} {object1.[](1).=} {object2.InRect(object1,y)} {object2.InRect(object1,y).=}
FV_i(0<i≤4)	{object1} {object1} {object1,object2} {object1,object2}

또한, 순차 C++프로그램에서 컴파일러가 간접 호출에 대한 병렬성을 검출하는 것은 불가능하다. 클래스를 설계하는 프로그래머가 병렬 정보에 대한 기술 없이는 참조 매개변수를 사용하는 경우와 객체의 참조를 반환하는 메소드에 대해서 메소드간의 병렬성을 검출하는 것은 어렵다[18]. 따라서, 참조 변수에 의한 부작용은 본 논문에서 제시한 클래스 인터페이스내의 주석을 사용하여 해결할 수 있다.

4. 병렬 정보를 갖는 C++ 클래스 인터페이스 설계

순차 C++ 프로그램에서 데이터를 액세스하는 것은 메소드 형태를 취하므로 문장간의 종속성을 분석하기 위해서는 프로시져간의 종속성을 분석해야 한다. 프로시져간의 종속성 분석은 수정되어지거나 사용되어진 변

수의 집합이나 요약된 정보를 얻는 것이 어렵다[7].

이러한 문제점을 해결하기 위한 한가지 방법으로 클래스 인터페이스에 함수간의 관계, 즉 병렬성의 명시적인 사항을 표기하므로서 함수간의 종속 분석을 효과적으로 할 수 있다.

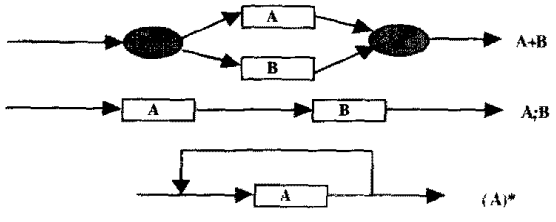
3.4절에서 기술한 참조 변수를 사용하는 메소드내의 호출이 어느 객체의 메소드인지 찾고 이러한 호출이 다른 메소드에 영향을 주는지 안 주는지 결정하는 문제를 컴파일러가 전적으로 해결하는 것은 어렵다. 따라서, 본 논문에서는 참조 객체를 사용하는 메소드와 다른 메소드간의 관계를 클래스에 기술하여 이러한 문제를 해결하였고, 컴파일러가 메소드간의 관계를 분석하기 위한 오버헤드를 감소시킬 수 있다.

클래스에 존재하는 메소드들간의 관계는 다음 중 하나에 해당한다.

- 메소드 A는 메소드 B와 병렬 실행한다.
- 메소드 A는 메소드 B와 병렬 실행할 수 없다.
- 메소드 A는 하나 이상 병렬 실행할 수 있다.
- 메소드 A 다음에 반드시 메소드 B가 실행해야 한다(순서).

4.1 정규식

본 논문에서는 정규식 연산자를 사용하여 메소드들간의 관계를 나타내며, 객체내에 정의된 메소드의 집합은 정규식을 사용하여 메소드간의 병렬성을 정의할 수 있다. (그림 4)는 정규식을 도식화한 것이며, 정규식은 세 가지 연산자를 갖는다. 즉 +연산자는 병렬, *는 반복, ;는 순서를 각각 나타낸다.



(그림 4) 정규식 표현
(Fig. 4) Representation of Regular Expression

A+B의 의미는 A 또는 B가 순서에 관계없이 병렬 수행할 수 있다는 것을 의미한다. A;B는 정규식을 사용하는 절에 따라 서로 다른 두 가지의 의미를 갖는다. 병렬 정보인 GERINFO절에서는 단지 병렬 수행할 수 있다는 것을 의미하며, 순차 실행인 SEQ절에서는 A가 먼저 실행한 후 반드시 B를 실행해야 한다는 것을 의미한다. 또한 (A)*는 하나 이상 병렬로 수행할 수 있다는 것을 의미한다.

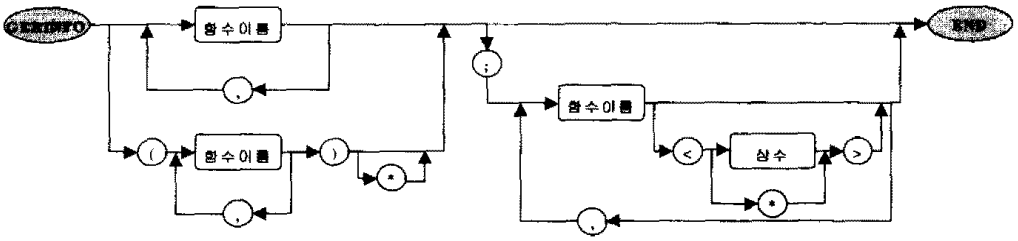
C나 파스칼에서는 존재하지 않지만, C++은 함수에 전달되는 매개변수의 데이터 타입 또는 함수에 전달되는 매개변수가 틀릴 경우 하나의 프로그램에 동일한 이름을 가진 함수가 여러개 존재할 수 있다. 컴파일러는 오버로딩 함수에 대해 이름 압착(name mangling) 이란 기법을 사용하여 각각의 함수에 유일한 함수 이름을 부여한다[1]. 각각의 오버로딩된 함수는 동일한 이름을 갖지만 그것들은 서로 다른 기능을 하고, 메소드들간의 관계 또한 다양하다. 예를 들면 A메소드와 오버로딩된 B메소드 B(int)와 B(char)의 관계에 있어서 A메소드와 B(int)는 병렬 실행 가능하지만 B(char)와는 상호배제로 실행된다고 했을 때 함수 이름만 사용할 경우 이를 구별할 수 있는 방법은 없다. 따라서 메소드간의 병렬 정보를 표현하기 위해서는 표현식내의 각 함수를 구별하는 방법이 필요하다. 오버로딩된 함수의 첫 번째 메소드는 함수 이름 다음에 〈1〉, 두 번째 메소드는 〈2〉, 모든 메소드는 〈*〉로 표기한다.

아래 프로그램에서 '++'는 함수 호출의 형태를 취하고 동일한 이름이 한 클래스내에 두 개 존재한다. 동일한 이름의 첫 번째 메소드 operator++(int)를 표기하는 방법은 ++〈1〉, 두 번째 메소드 operator++()는 ++〈2〉, '++' 이름의 모든 메소드의 표현은 ++〈*〉로 표기한다.

```
class ArrayClass {
    int size;
    int *ip;
     .....
public:
    ArrayClass operator++(int);
    ArrayClass operator++();
```

4.2 병렬 정보

프로그래머가 병렬처리를 위해서는 각 객체 내부에 메소드간의 관계와 외부 세계와의 인터페이스를 위해 병렬성을 표기하는 방법이 필요하다. 프로그래머가 C++ 클래스내에 병렬성을 명시해야 하는 부담은 있지만 이러한 방법은 클래스를 설계한 프로그래머는 객



(그림 5) 병렬 정보의 문법 구문도
(Fig. 5) Syntax Diagram of General Parallel Information

체에 대한 세부적인 지식을 가지고 있기 때문에 메소드 간의 상호 작용을 명확히 표현할 수 있다. 컴파일러는 사용자에 의해 기술된 병렬 정보를 이용하여 묵시적인 병렬성을 검출하고 프로그램을 병렬화한다.

병렬 정보의 문법 구문도는 (그림 5)와 같으며, 키워드 GERINFO과 END 사이에 병렬 정보에 대한 식으로 구성한다.

예를 들어, 아래와 같은 병렬 정보를 사용하여 ArrayClass의 메소드들 간의 병렬성을 표현한 예를 살펴보자. 병렬 정보는 단순히 'getsize', '[]', '+', '-'는 하나 이상 병렬로 실행할 수 있지만 메소드 '='과 '++(*)'와는 병렬로 실행할 수 없다는 것을 표기한다.

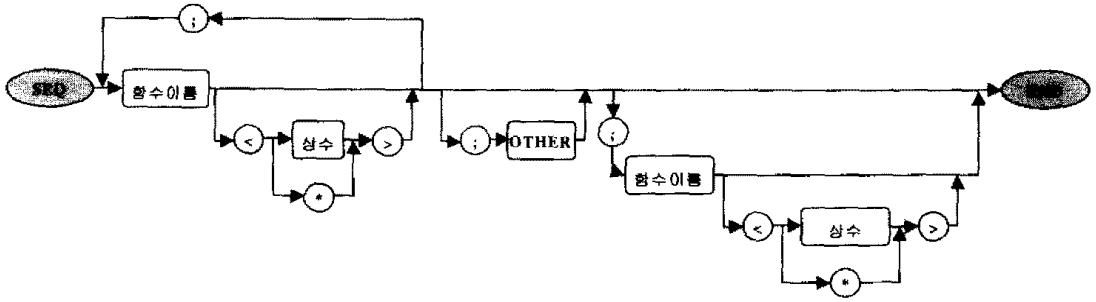
```
class ArrayClass {
  int size;
  int *ip;
  void check(int);
public:
  GERINFO(getsize, [], +, -)* ; =, ++(*) END;
  ArrayClass(int sz=ARRSIZE);
  ArrayClass(ArrayClass&);
  int getsize() { return size; }
  ArrayClass& operator=(ArrayClass&);
  int& operator[](int);
  ArrayClass operator+(ArrayClass&);
  ArrayClass operator-(ArrayClass&);
  ArrayClass operator++(int);
  ArrayClass operator++();
S1:  A.getsize();
S2:  A.getsize();
S3:  B[1] = A[1];
S4:  C   = A + B;
S5:  D   = C + A;
  .....
};
```

문장 S1과 S2는 같은 객체에 속하는 함수 getsize를 호출하고 있다. ArrayClass내에서는 getsize 메소드가 하나 이상 병렬 실행할 수 있다고 명시되어 있으므로 문장 S1과 S2는 병렬 실행할 수 있다. 또 문장 S3과 S4는 A 객체에 대해 []과 + 메소드 호출이 일어나고, ArrayClass내의 병렬 정보는 []메소드와 + 메소드는 병렬 실행할 수 있다는 것을 보여 주고 있다. 그러므로 문장 S3과 S4 또한 병렬 실행할 수 있다. ArrayClass내에는 =과 + 메소드는 병렬 실행할 수 없다고 명시되어 있으므로 문장 S4와 S5는 병렬 실행할 수 없다. 즉 클래스내에 기술된 병렬 정보를 이용하여 재구성 컴파일러는 메소드간의 종속성을 구하여 프로그램에 내재된 병렬성을 탐지할 수 있다.

4.3 순서 정보

병렬 정보에서는 동일 클래스내에 있는 메소드간의 실행 순서에 관한 정보가 없다. 일반적으로 어떤 시기에 모든 메소드가 호출되는 것은 아니다. 한 객체에 있는 특정 메소드는 동일한 객체의 다른 메소드에 선행해야 한다[10].

예를 들면 화일 처리에서 읽기나 쓰기 연산을 수행하기 전에 반드시 화일을 개방해야 한다. 또한 이러한 연산들은 반드시 화일을 닫은 후에 실행되어서는 안된다. 이러한 것을 해결하는 방법은 시스템에 미리 행위에 대한 정의가 되어 있는 내장 검사(embedded check) 방법으로 처리하거나, 실행하기 전에 조건을 만족해야 하는 최소한의 선조건(pre-condition)을 명시해야 한다. 본 논문에서는 병렬 정보 이외에 순서 정보를 사용하며, 순서 정보의 문법 구문도는 (그림 6)과



(그림 6) 순서 정보의 문법 구문도
(Fig. 6) Syntax Diagram of Sequence Information

같이 구성한다.

아래 프로그램의 SEQ절은 Open 메소드 이후에 DataBase 클래스에 있는 다른 메소드가 실행될 수 있고, 그러한 메소드는 반드시 Close 메소드 이전에 실행되어야 한다는 것을 표기한 것이다. OTHER 프리미티브는 SEQ절에 표기되지 않는 모든 메소드들이 OTHER 위치에 올 수 있다는 것을 의미한다.

```
class DataBase {
    .....
public:
    GERINFO ..... END:
    SEQ   Open: OTHER: Close END:
    ......
    Open();
    FileSeek(int pos, int Current );
    BlockRead(int size);
    Close();
    ......
};
```

5. 종속성 분석

재구성 컴파일러의 중요한 부분은 데이터 종속 분석과 메소드간의 종속 분석이다. 일반적인 데이터 종속 분석은 문장 단위의 종속성 분석과 프로시져간의 종속성 분석으로 나눌 수 있다. 흐름 분석이 끝난 다음 인접하는 문장 S1과 S2간의 데이터 종속성 분석 방법은 사용되어지는 변수의 집합 S1.IN과 S2.IN, 수정되어지는 변수의 집합 S1.OUT과 S2.OUT이 처음에 결정된다. 문장 S1과 S2가 S1.IN ∩ S2.OUT, S1.OUT ∩ S2.IN 그리고 S1.IN ∩ S2.IN이 공집합이면 두 문장은 종속이 존재하지 않는다. 프로시져간의 종속성 분석 방법은 문장간의 종속성 분석 방법보다 복잡하다. 프로시져간의 종속성을 분석하기 위해서는 프로시져내에서 수정되어지거나 사용되어진 변수의 집합이나 요약된 정보를 수집하여야 한다. C++에서 데이터를 액세스하는 것은 프로시저 호출의 형태를 취한다. 따라서 데이터간의 종속성을 알려면 문장간의 종속성 방법보다는 프로시져간의 종속성을 분석해야 한다. 그러나 프로시져간의 종속성 분석은 정확한 정보가 아닐 경우가 많기 때문에 프로그래머에 의한 클래스 인터페이스내의 병렬성 정보를 사용하면 더욱 효과적이다. 일단 병렬성이 클래스 인터페이스에 기술되어지면 객체의 메소드 호출에 대한 부작용은 객체 그 자체에 제한되고 부작용에 의해 영향받는 모든 메소드는 클래스 인터페이스 내에 기술된 병렬정보와 상호작용을 갖는다.

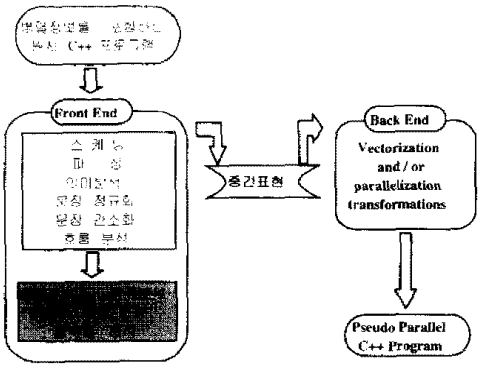
5.1 재구성 컴파일러의 구조

재구성 컴파일러의 구조는 (그림 7)과 같으며, 전처리부(front end)에서 강조된 부분은 일반적인 재구성 컴파일러 부분에 본 논문에서 제안한 병렬 정보의 정적 분석, 순서 정보 분석, 종속성 분석을 추가한 부분이다.

재구성 컴파일러는 클래스 인터페이스내에 기술되어진 병렬 정보를 고려하여 순차 프로그램을 병렬 프로그램으로 변환한다.

5.2 병렬 정보의 정적 분석

병렬 정보의 정적 분석은 사용자에 의해 병렬 정보가 잘못 사용되었을 때 병렬화된 프로그램은 오류를 유발할 가능성이 있고, 사용자에 의해 표기된 병렬 정보를 분석하기 위한 방법이 필요하다. 재구성 컴파일러의



(그림 7) 재구성 컴파일러 구조
(Fig. 7) Restructuring Compiler Structure

전처리 부분에서 이러한 작업이 이루어진다.

사용자가 기술한 병렬 정보가 정당하다는 것을 토대로 종속 분석이 이루어지기 때문에 재구성 컴파일러의 병렬 정보를 위한 정적 분석기는 단지 경고 메시지를 표시한다. 프로그램의 수행결과가 올바르게 생성되지 않았을 때 사용자는 정적 분석기에서 출력된 경고 메시지를 살펴봄으로써 병렬 정보의 잘못된 표기를 정정할 수 있다.

알고리즘 1은 정적 분석 알고리즘이다. 수행 과정에서 partition 함수는 형식 매개 변수로 ObjectPATH, Delimeter, Cset, Mset을 입력 받고, 기능은 클래스 인터페이스에 기술된 병렬 정보를 병렬절과 순차절로 분리하여 Cset과 Mset에 할당하여 반환한다. next 함수는 X 다음의 원소가 존재하면 해당 원소를 반환하고, 존재하지 않으면 NULL을 반환한다.

state_data_dependence_check 함수는 객체의 멤버 데이터를 갱신하는 부분에 대해서만 종속성 분석을 한다. 참조 매개변수를 취하는 경우와 참조를 반환하는 메소드에 대한 종속 분석은 최악의 경우(종속이 존재하는 것으로)로 가정한다. 이 함수는 메소드 e와 e1에 대해 종속 검사를 하고 종속이 존재하면 참을 반환하고 종속이 존재하지 않으면 거짓을 반환한다.

GERINFO_static_analyzer 함수는 partition 함수에 의하여 분리된 병렬절과 순차절의 정보를 가지고 메소드들에 대한 종속 분석을 실행한다. 병렬절에 대한 종속 분석 결과 종속이 발견되면 경고 메시지("메소드간에는 상태 데이터에 대한 종속이 존재한다.")를 출력시킨다. 또한 병렬절과 순차절에 대한 종속 분석을 실시하고 종속이 존재하지 않으면 경고 메시지("메소드간에는 상태 데이터에 대한 종속이 없다.")를 출력한다.

알고리즘 1. 병렬 정보의 정적 분석 알고리즘

입력 : 해당 객체의 병렬 정보
출력 : 만약 데이터 종속 규칙을 만족하면 메시지 출력, 그렇지 않으면 다음 작업을 계속 진행

```
void partition(SET ObjectPATH, Element Delimeter,
               SET Cset, SET Mset);
SET next(SET X);
Boolean state_data_dependence_check(SET e, SET e1);
void GERINFO_static_analyzer(SET ObjectPATH)
{
  SET Cset, Mset;
  partition(ObjectPATH, ':', Cset, Mset);
  for every e ∈ Cset do
    if ((e1 = next(e) ∈ Cset) and next(e) != NULL )
      if (state_data_dependence_check(e, e1))
        Message("경고 : 메소드 e와 e1간에는 상태 데이터에
                 대한 종속이 존재한다.");
      end if
    end if
  end for
  for every e ∈ Cset do
    for every e1 ∈ Mset do
      if (!state_data_dependence_check(e, e1))
        Message("경고 : 메소드 e와 e1간에는 상태 데이터에
                 대한 종속이 없다.");
    end for
  end for
} /* End Of GERINFO_static_analyzer */
```

아래의 프로그램은 병렬절의 잘못된 표기 예를 나타내며, Check 클래스의 병렬 정보는 Add 메소드와 Sub 메소드 사이에 종속이 존재하지 않는 것으로 기술되었다. 그러나 알고리즘 1의 partition함수에 의해 병렬절과 순차절은 Cset = {Add, Sub}가 되고, Mset = {Sang}이 된다. 따라서, 실제로는 메소드 Add와 Sub는 데이터 종속이 존재한다. 또한 Add와 Sang 메소드간에는 종속이 존재하지 않음에도 불구하고 병렬 정보에는 종속이 존재하는 것으로 기술되었다. 이와 같은 경우에 컴파일러는 경고 메시지(메소드 Add와 Sang간에는 상태 데이터에 대한 종속이 존재한다.)를

출력한다.

```
class Check {
  int   CFlag;
  int   DFlag;
public:
  GERINFO Add, Sub; Sang END;
  void Add()  { Cflag += 1; }
  void Sub()  { Cflag -= 1; DFlag--; }
  void Sang() { DFlag++; }
}
```

5.3 순서 정보 분석

하나의 객체에 존재하는 메소드들은 어떤 시기에 모든 메소드가 호출되는 것은 아니다. 동일한 객체에 있는 어떤 메소드들은 다른 메소드에 선행해야 한다. 컴파일러는 클래스 인터페이스에 기술된 순서 정보와 흐름 분석을 통해 생성된 객체의 호출 그래프(call graph)를 비교한다.

알고리즘 2에서 SEQ_POS 함수는 제어흐름 분석을 통해 생성된 호출 그래프의 각 노드에 대해 클래스 인터페이스에 기술된 순서정보의 위치를 찾는다. ADD_Element 함수는 ObjectSEQ에서 index번째 원소를 반환하고 집합 Temp에 집어넣는다. Compatible 함수는 각 원소에 대해 처리가 끝난 다음 Temp와 ObjectSEQ가 호환되는지 안되는지를 검사한다.

SEQ_Analyzer 함수는 객체에 대한 제어흐름 분석에 의해 생성된 호출 그래프와 클래스 인터페이스에 기술된 순서 정보를 비교하여 객체의 호출 순서가 정당한지 아닌지를 분석한다.

알고리즘 2. 순서 정보 분석 알고리즘

```
입력 : 해당 객체의 호출 그래프(G = (N, E)), 클래
       스에 표기된 순서 정보
출력 : 객체의 호출 순서에 대한 정당성 유무
int SEQ_POS(char *s, SET ObjectSEQ);
char *ADD_Element(SET ObjectSEQ, int index);
BOOLEAN Compatible(SET Temp, SET ObjectSEQ);
SEQ_Analyzer(CALL_GRAPH CG, SET ObjectSEQ)
{
SET Temp;
  for every n_i ∈ CG(N,E) (i=1~edge number, n_i ≠ Node
number) do
    index = SEQ_POS(n_i, ObjectSEQ);
    Temp = ADD_Element(ObjectSEQ, index);
  end for
  if (Compatible(Temp, ObjectSEQ))
      return 1;
  else  return 0;
}
```

예를 들어, 아래의 프로그램은 객체의 호출 순서를 나타내며, 컴파일러의 흐름 분석을 통해 생성된 호출 그래프는 S: Open→Read→Close, T: Open→Close →Read일 때, Temp집합은 알고리즘 2의 수행에 의해 S 객체는 Open; OTHER; Close, T 객체는 Open; Close; OTHER가 생성된다. 따라서 S 객체는 ObjectSEQ절에 표기된 순서와 같지만 T 객체는 순서가 다르다는 것을 알 수 있다.

```
class DEFFILE {
  .....
public:
  SEQ Open; OTHER; Close END
  Open(); Read(); Write(); Close();
};
main()
{
  DEFFILE S, T;
  S.Open(); S.Read();
  S.Close();
  T.Open(); T.Close();
  T.Read();
}
```

5.4 클래스의 병렬 정보를 사용한 데이터 종속 분석기

종속성 분석은 문장에서 호출되는 집합이 결정된 후에 두 문장이 종속인지 아닌지를 분석한다. C++에서 데이터 종속 분석을 하기 위한 첫 번째 단계는 문장에서 호출되는 메소드 집합을 결정하는 것이다. 메소드는 문장에서 두 가지 형태로 호출된다. 일반적인 형태는 A.getsize()와 같이 직접 호출되는 경우이고, 또 다른 형태는 할당 문장에서 V.=(OBJECT) 호출과 동일한 V=OBJECT가 될 수 있다. 메소드 호출의 또 다른 형태는 참조 매개변수로서 객체를 다

는 메소드에 건네주는 것에 의해 간접 호출이 발생한다. 일단 모든 문장 S에 의해 호출되는 메소드 S.C 집합이 결정되면, 두 문장이 종속적인지 아닌지를 분석하기 위해 본 논문에서 제시한 알고리즘 3을 사용하여 종속 분석을 한다.

알고리즘 3. 데이터 종속 분석기 알고리즘

입력 : 해당 객체의 병렬 정보, 두 문장 S1과 S2의 호출 집합 STMT1.C, STMT2.C

출력 : 만약 두 문장간에 종속성이 존재하면 참(1), 종속성이 존재하지 않으면 거짓(0)을 반환

```
SET Subtract(SET ObjectPATH, SET Temp);
void Order(SET *List);
void Sort(SET *List);
BOOLEAN after(SET X, ITEM Y);
 /*SET X뒤의 원소 Y가 존재한다면 거짓을 반환, 존재하지 않
   으면 참을 반환 */
function dependence_analyzer(SET ObjectPATH, SET
            ObjectSEQ, SET STMT1.C, SET STMT2.C)
{
   SET Temp, Thin_Set;
/*0*/
/*1*/   for every e ∈ {STMT1.C, STMT2.C} do
          if (e ∈ ObjectSEQ)  return(FALSE);
        end for
/*2*/   Temp = STMT1.C + STMT2.C;
/*3*/   Order(Temp);
/*4*/   Order(ObjectPATH);
/*5*/   Thin_Set = Subtract(ObjectPATH, Temp);
/*6*/   if (Thin_Set == Temp)
          return(TRUE);
        else
          return(FALSE);
}
SET Subtract(SET ObjectPATH, SET Temp)
{
  SET SubSet;
   for every e ∈ Temp do
      for every e1 ∈ ObjectPATH do
         if(e == e1)  /* 동일한 원소이면 SubSet에 등록 */
            SubSet = e1;
         else {
            if (e1 ==';')
               SubSet+=e1;
         } /*End Of Else */
      end for
   end for
      if (after(SubSet, ';')) /* 잘못된 오퍼레이터 제거 */
         SubSet -= ';';
         return(SubSet);
}
void Order(SET *List)
{
   Sort(List); /* 사전적 순서로 정렬 */
}
```

알고리즘 3에서 dependence_analyzer 함수는 형식 매개 변수로 병렬 정보인 ObjectPATH, 순서 정보인 ObjectSEQ, 각각의 문장에 대한 호출 집합 STMT1.C, STMT2.C를 갖는다. 수행 과정은 첫 번째 단계에서 문장에 대한 호출 집합 중 OTHER 프리미티브에 속하지 않는 순서 정보와 일치되는 원소가 있으면 두 문장은 종속한다. 만약 순서 정보에 속하지 않는다면 두 문장은 '+' 연산을 하여 두 집합에서 중복되는 원소가 있으면 하나만 설정한다. 예를 들면 {A, B, C} + {A, D, E}는 {A, B, C, D, E}가 된다. 두 번째 단계에서 Order 함수는 사전적 순서로 Temp와 ObjectPATH를 병렬절과 순차절로 분리하여 정렬한다. 예를 들면 {DDA, CFG, DFG; AFG, EVENT}의 정렬은 {CFG, DDA, DFG; AFG, EVENT}가 된다. 세 번째 단계에서 Subtract 함수는 ObjectPATH집합과 Temp집합을 입력으로 받아 Temp에는 없는 ObjectPATH내의 원소를 제거한다. 예를 들면, ObjectPATH = {A, B, C}이고 Temp = {A, C}일 때 결과는 {A; C}를 반환한다. 마지막 단계는 Thin_Set과 Temp를 비교하여 같으면 종속이 존재하지 않고, 같지 않으면 종속이 존재한다. 예를 들면, 문장 S1과 S2는 S1.C내에 object1.method1()호출이 존재하고, S2.C내에 object2.method2()호출이 존재하는 경우, object1 = object2 그리고 method1과 method2는 object1내의 병렬 정보와 부합되지 않는 경우 종속한다.

5.5 적용 예

5.5.1 종속 분석기를 이용한 종속성 검사 예

본 논문에서 제시한 종속 분석 알고리즘에 따라 병렬 정보를 이용하여 루프 문장의 종속성 검사 예를 살펴 보자. 다음 프로그램은 배열의 사칙 연산을 하는 프로그램이다.

```
class ArrayClass {
  int size;
  int *ip;
void check(int) {{ if (index < 0 || index >= size) { ...}};
public:
  GERINFO(getsize, [], +, -, *, /  )* : =,++(*) END;

  SEQ ArrayClass(*): OTHER: ~ArrayClass END;
  ArrayClass(int sz=ARRSIZE) { /* sz만큼의 메모리를 ip
                              에 할당하고 초기화 */ }
  ArrayClass(ArrayClass&){ /* 복사 생성자*/ }
  ~ArrayClass() { delete ip; }
  int getSize() { return size; }
  ArrayClass& operator=(ArrayClass&) { /* 연산자 오
                                       버로딩 */ }
  int& operator[](int) { check(index); return ip[index]; }
    { /* 사칙 연산의 결과를 임시 변수 ArrayClass 변수
       temp에 저장 후 반환 */ }
  ArrayClass operator+(ArrayClass& IP)
  ArrayClass operator-(ArrayClass& IP)
  ArrayClass operator*(ArrayClass& IP);
  ArrayClass operator/(ArrayClass& IP);
    ArrayClass operator++(int): { /* postfix 증감 후
                              temp변수에 저장 반환 */ }
  ArrayClass operator++(): { /* prefix 증감 후 temp변
                              수에 저장 반환 */ }
  void GetData() { .... }
  void Display() { .... }
};
int main()
{
 ......
  /* CASE 1 */
  for (int i = 0; i < SIZE; i++) {
    /*S1*/  A[i] = B[i] + C[i];
    /*S2*/  C++;
  }
  /* CASE 2 */
  for (i = 0; i < SIZE; i++) {
    /*S1*/  A[i] = B[i] + C[i];
    /*S2*/  D[i] = B[i]  C[i];
    /*S3*/  E[i] = B[i] * C[i];
  }
}
```

본 논문에서 제시한 종속 분석 알고리즘에 따라 CASE 1, 2는 <표 2>와 같은 과정을 통해 종속의 유무를 판단한다.

<표 2> 병렬 정보를 이용한 종속 분석 결과
<Table 2> Dependence Analysis using Parallel Information

문장번호	CASE 1	CASE 2
0	ObjectPATH={getsize,[],+,*,/, :,=,++(*)} ObjectSEQ={ArrayClass(*),~ArrayClass} STMT1 = {[]} STMT2 = {++} 대상 : C 객체	ObjectPATH={getsize,[],+,-,*,/,:,=,++(*)} ObjectSEQ={ArrayClass(*),~ArrayClass} STMT1 = {+}, {+}, {-} /*S1,S1,S2*/ STMT2 = {-}, {*}, {*} /*S2,S3,S3*/ 대상 : B 객체
1	Ø	Ø
2,3,4	ObjectPATH={getsize,[],+,*,/, :,=,++(*)} Temp = {[], ++}	ObjectPATH={getsize,[],+,-,*,/, :,=,++(*)} Temp = {+, -}, {+, *}, {-, *}
5	Thin_set = {[], :, ++} Temp = {[], ++}	Thin_set = {+, -}, {+, *}, {-, *} Temp = {+, -}, {+, *}, {-, *}
6	두 집합이 같지 않으므로 거짓 반환 (즉 종속성이 존재한다.)	두 집합이 같으므로 참 반환 (즉 종속성이 존재하지 않는다.)

<표 2>의 분석 결과에 의해 CASE 1은 C 객체에 대해 '++'과 '[]'은 ArrayClass의 병렬 정보와 부합되지 않으므로 병렬 구문으로 변환할 수 없다.

CASE 2의 경우는 B 객체를 포함하는 각각의 문장에 대해 종속성 유무를 판단해야 한다. 먼저 S1과 S2, S1과 S3, S2와 S3에 대해 종속 유무를 판단하고, 판단 결과 종속이 존재하지 않으므로, C 객체에 대해서도 위와 같은 과정을 반복한다. C 객체의 '[]'는 하나 이상 병렬로 실행할 수 있으므로 종속이 존재하지 않는다는 것을 알 수 있다.

5.5.2 재구성 컴파일러에 의한 병렬 코드 생성 예

재구성 컴파일러에서 데이터 종속 분석은 흐름 분석에 의해 생성된 호출 집합간의 종속 분석을 한다. 객체의 첨자 변수에 대한 분석은 메소드 호출에 대한 종속 분석을 하기 전에 분석되어야 한다. 지금까지 객체지향 언어가 아닌 언어에서는 첨자 변수가 문장 간에 공유하게 될 때 종속이 존재하지만 객체지향 언어에서는 공유하는 첨자 변수에 대한 객체의 호출 집합에 따라 종속의 유무가 결정된다. 따라서 C++의 종속 분석은 첨자 변수에 대한 종속 분석이 먼저 이루어지고 다음에 공유되는 첨자 변수에 해당되는 메소드 호출 분석이 이루어져야 한다.

다음은 마이크로소프트사의 Visual Basic 5.0을 사용하여, 재구성 컴파일러에서 데이터 종속 분석기를 이용하여 순차 C++ 프로그램을 병렬 코드로 변환하는 방법을 구현하였다. 행렬 A = $[a_{ij}]$의 역행렬을 구하는 순서는 다음과 같다.

① 제 i행을 대각요소의 a_{ii} 로 나눈다.

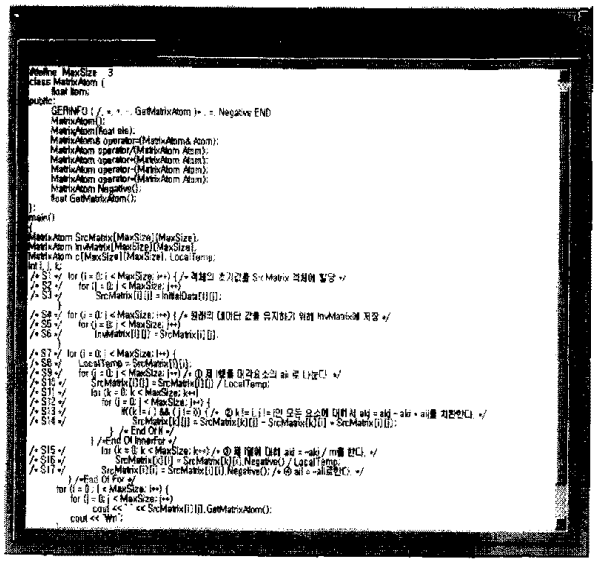
② $k \neq i$, $j \neq i$인 모든 요소에 대해서 $a_{kj} = a_{kj} - a_{ki} \times a_{ij}$를 치환한다.

③ 제 i열에 대해 $a_{ki} = -a_{ki} / m$을 한다.

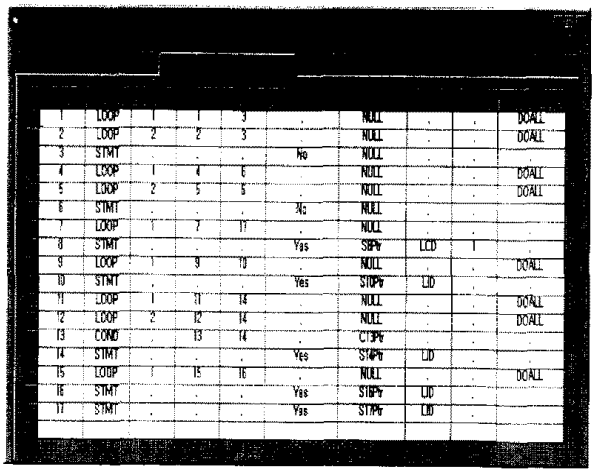
④ $a_{ii} = -a_{ii}$로 한다.

(그림 8)은 행렬의 역행렬을 구하는 순차적인 C++ 원시 프로그램이다.

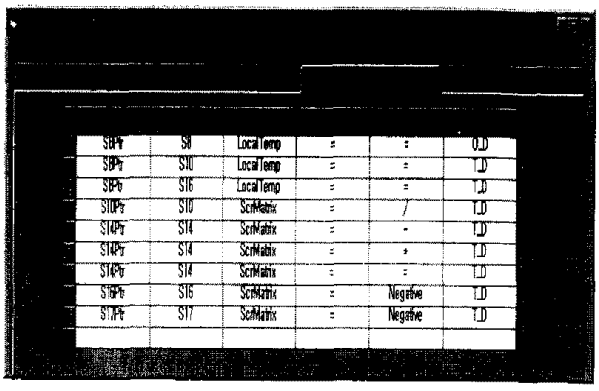
(그림 9)에서는 데이터 흐름 분석을 한 후 문장에 대한 문장 번호, 루프 내포 레벨, 루프의 시작과 끝, 블럭 내의 동일 객체 존재 유무, 문장의 종류 등의 정보가 기록된다. 기록된 정보를 토대로 데이터 종속 분석이 행해진다. 종속 레벨과 종속성 종류는 첨자 변수에 대한 정보를 근거로하여 메소드간의 종속이 존재할 때 삽입된



(그림 8) 순차 C++ 프로그램
(Fig. 8) Sequential C++ Program



(그림 9) 종속성 분석 결과
(Fig. 9) Result of Dependency Analysis



(그림 10) 호출 집합 정보
(Fig. 10) Information of Call Set

다. (그림 9)에서 LCD(Loop Carried Dependence)는 루프내의 서로 다른 이터레이션간의 종속성이 존재하는 것을 나타내고, LID(LOOP Independent Dependence)는 루프내의 서로 다른 이터레이션간의 종속성이 존재하지 않고, 동일한 이터레이션내에서만 종속성이 존재하는 경우이다.

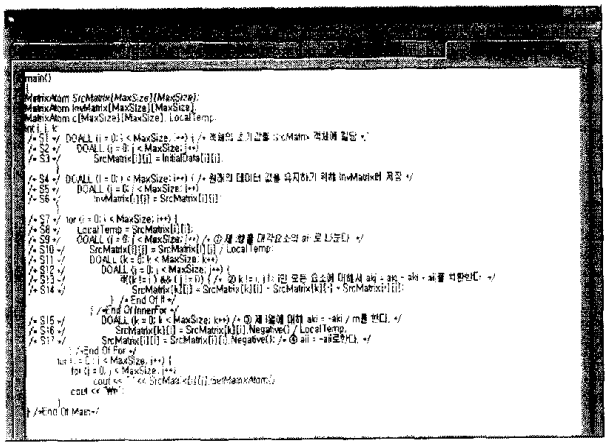
(그림 10)은 (그림 9)의 호출 집합에 대한 포인터를 나타낸다. (그림 10)에서 O_D(Output Dependence)는 출력 종속성을 나타내고, T_D(True Dependence)는 흐름 종속성을 나타낸다.

(그림 9)와 (그림 10)의 결과에 의하여 문장 S1과 S2는 루프이고, 문장 S3를 내포한다. 문장 S3은 동일한 객체가 존재하지 않으므로 병렬 루프문 DOALL로 변환할 수 있다. 또한 문장 S4와 S5는 루프이고, 문장 S6을 내포하고 있으며 동일한 객체가 존재하지 않으므로 DOALL로 변환할 수 있다. 문장 S7은 문장 S8과 S17을 내포하고 있고, 문장 S8은 LocalTemp객체에 대해 LCD가 존재하므로 문장 S7은 병렬 구문으로 변환할 수 없다. 문장 S9는 루프이고 문장 S10만을 내포한다. 문장 S10은 SrcMatrix객체에 대해 LID가 존재한다. LID는 병렬 루프로 변환시 프로그램 수행에 영향을 주지 않으므로 병렬 루프문으로 변환할 수 있다.

문장 S11과 S12는 문장 S14를 내포하고 있고 문장 S14내에 LID가 존재하므로 병렬 루프문으로 변환할 수 있다. 문장 S15는 문장 S16을 내포하고 있고 문장 S16은 문장 S8과 데이터 종속이 존재하지만 문장 S7이 순차적으로 실행한다면 문장 S15 또한 병렬 루프로 변환할 수 있다. 문장 S17은 문장 S7에 내포된 문장이므로 문장 S8에 의해 병렬 실행할 수 없다. (그림 11)은 원시 C++ 프로그램을 병렬 코드로 변환한 프로그램이다.

6. 결 론

재구성 컴파일러에 의해 순차 C++ 프로그램을 병렬화하기 위해 고려해야 되는 것은 순차 C++ 프로그램의 실행이 객체간의 메소드 호출, 즉 함수 호출을 통하여 이루어진다는 점이다. 따라서 재구성 컴파일러는 함수간의 종속성을 찾아내야 한다. 그러나 전적으로 재구성 컴파일러에만 의존해야 한다면 객체의 참조 매개변수를 사용하는 경우와 객체의 참조를 반환하는 메소드에 대해서 메소드간의 병렬성을 검출하는 것은 극히 일부분에 지나지 않는다.



(그림 11) 병렬 프로그램
(Fig. 11) Parallel Program

본 논문에서는 클래스 인터페이스에 주석을 사용하여 클래스내에 메소드간의 병렬성을 명시하므로써 재구성 컴파일러에서 효과적인 병렬성 검출을 가능하게 하였다.

클래스 인터페이스에 병렬 정보를 기술하므로써 재구성 컴파일러는 명시된 정보를 이용하여 종속성을 분석하며 프로그램에 내재한 묵시적인 병렬성을 얻는다. 또한 순서 정보를 클래스 인터페이스에 삽입함으로써 처리 순서가 정해진 작업에 대해 사용자의 부주의로 인한 프로그램의 오류를 막을 수 있다. 또한 본 논문에서 제시하는 종속 분석 방법을 이용하여 순차 C++ 프로그램을 병렬 코드로 변환하는 병렬코드 변환기를 구현하였다.

객체내의 메소드에 대한 병렬성만 고려했기 때문에 본 논문에서 검출할 수 있는 병렬성은 중간 단위(medium grain)이다. 앞으로 병렬 정보를 디버깅하기 위한 시각화툴에 대한 연구와 병렬코드 변환기를 통해 나온 정보를 이용하여 스레드 프로그램으로 변환하여 시뮬레이션할 수 있는 툴에 대한 연구가 필요하다.

참 고 문 헌

[1] Allen I. Holub, "C+C++ Programming With Objects in C and C++," McGraw-Hill, 1992.
[2] Radenski, A. A, "Object-Oriented Programming and Pallelism: Introduction," Inf. Sci (USA), Vol.93, No.1-2, pp.1-7, August. 1996.
[3] B. Stroustrup, "The C++ Programming Lan-

guage," Addison-Wesley Publishing Company. Inc. 1986.

[4] C. D. Polychronopoulos, "Parallel Programmming and Compilers," Kluwer Academis Publishers, 1988.

[5] D. Gannon and J. K. Lee, "Object oriented parallelism:pC++ ideas and experiments," In Processings of 1991 Japan Society for Parallel Processing, pp.13-23, 1993.

[6] Van der Schoot, H., Ural, H., "Data flow Analysis of System Specification in LOTOS," Int. J. Softw. Eng. Knowl. Eng. Vol.7, No.1, pp.43-68, March. 1997.

[7] Hans Zima, Barbara Chapman, "Supercompilers for parallel and Vector Computers," ACM Press, 1991.

[8] J. Feldman, C-C. Lim, and T. Rauber, "The shared-memory language pSather on a distributed-memory multiprocessor," In Processing of the Second Workshop on Language, Compilers and Runtime Environments for Distributed Memory Multiprocessors, October. 1992.

[9] Agha, G. A., Mason, I. A., "A Foundation for Actor Computation," J. Funct. Program. (UK), Vol.7, No.1, pp.1-72, January. 1997.

[10] Jan Van Den Bos, "A Parallel Object Language with Protocols," 1989 OOPSLA Conference Processings, pp.95-102, October. 1989.

[11] Laxmikant V. Kale, Sanjeev Kirshman, "CHARM ++:A Portable Concurrent Object System Based On C++," ACM SIGPLAN NOTICES, Vol.28, No.10, October. 1993.

[12] Michael Wolfe, "Optimizing Supercompilers for Supercomputer," MIT Press, 1989.

[13] Norihisa Doi, Yasushi Kodama, Ken hirose, "An Implementation of an Operating System Kernel using Concurrent Object Oriented Language ABCL/c+," Proceeding of European Conference on Object-Oriented Programming, pp.250-266, August. 1988.

[14] Robert B. Kolstab, Roy H. Campbell, "PATH PASCAL USER MANUAL," URBANA, ILLINOIS 61801, Feb. 1980.

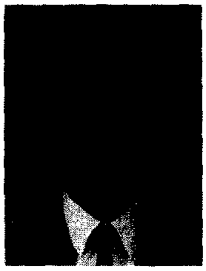
[15] Robert Lafore, "OBJECT ORIENTED PROGRAMMING IN TURBO C++," WAITE GROUP PRESS, 1993.

[16] Stanley Lippman, "Introduction to C++," 1991 OOPSLA, October, 1991.

[17] Won Kim, Frederick H. Lochovsky, "Object Oriented Concepts, Databases, and Applications," 1989 Addison Wesley, pp.79-124, 487-520, 1989.

[18] Youfeng Wu, Ted G.Lewis, "Parallelism Encapsulation in C++," 1990 International Conference on Parallel Processing, pp.1135-1142, August. 1990.

[19] 조세호, 황득영, 최영근, "재귀적 프로그램의 병렬 실행에 관한 연구," 병렬처리시스템학술발표회논문집, 제3권, 제1호, pp.57-72, 1992.



황 득 영

1988년 광운대학교 전자계산학과 (이학사)

1990년 광운대학교 전자계산학과 (이학석사)

1990년~1994년 기전여자 전문대학 조교수

1991년~현재 광운대학교 전자계산학과 박사과정

1994년~현재 삼척산업대학교 컴퓨터과학과 조교수

관심분야 : 병렬 컴파일러, 병렬 프로그래밍 언어, 객체지향 프로그래밍 언어, 시각언어, 분산처리

권 오 진

1990년 광운대학교 전자계산학과 (이학사)

1992년~1994년 수도군단 유선소대장

1994년 광운대학교 전자계산학과 (이학석사)

1994년~현재 산업기술정보원 데이터베이스 사업부 연구원

관심분야 : 병렬 컴파일러, 객체지향 소프트웨어, 검색시스템, 시각언어

최 영 근

1980년 서울대학교 사범대학 수학교육과(이학사)

1982년 서울대학교 계산통계학과(이학석사)

1989년 서울대학교 계산통계학과(이학박사)

1983년~현재 광운대학교 전자계산학과 교수

1997년~현재 광운대학교 전자계산소 소장

관심분야 : 병렬 컴파일러, 병렬 프로그래밍 언어, 객체지향 프로그래밍 언어, 객체지향 분산 컴퓨팅