자료 병렬 언어 프로그램의 병렬구조 변환을 위한 최적화기 설계

구 미 순† · 박 명 순††

요 약

대부분의 자료 병렬 언어 컴파일러들은 대부분 source-to-source 형태이다. 표준적 자료 병렬 언어로 인식되고 있는 HPF의 대부분의 컴파일러는 HPF로 작성된 프로그램을 메시지 패싱 프리미티브가 삽입된 포트란 77 프로그램으로 변환한다. 그런데 병렬 구조를 포함하는 병렬 언어로 작성된 프로그램을 순차 프로그램으로 변환하는 과정에서 병렬 프로그램의 시맨틱 유지를 위해 상당량의 비효율적 코드가 생성되고 있다. 특히 HPF의 특징 중 하나인 병렬 수행 구조 FORALL을 포트란 77의 DO 루프로 변환하는 과정에서 다수의 루프를 생성함으로 인해 코드의 성능이 저하되고 있다. 본 논문에서는 병렬 수행구조인 FORALL 구조로부터 최적화된 DO 루프를 생성하기 위해 루프 정렬을 적용하는 최적화기를 제시한다. 그리고 이 방안을 위해 필요한 정보 유지수단으로 관계거리벡터를 정의하여 사용한다. 끝으로 본 논문에서 제안한 방안에 의해 생성된 코드와 기존의 HPF 컴파일러 가운데 PARADIGM에 의해 생성된 코드의 성능을 비교 평가한다.

A Design Of An Optimizer For Conversion of Parallel Constructs Of Data Parallel Language Programs

Mi-Soon Koo† · Myong-Soon Park††

ABSTRACT

Most data parallel language compilers are source-to-source translators. Most Compilers of HPF which is recognized as a standard data parallel language convert a parallel program in HPF into a Fortran 77 program inserted message passing primitives. By the way, they currently generate significant amount of ineffective codes in the course of the conversion. Especially, FORALL construct is converted into several DO loops, so loop overhead of these codes is very increased. In this paper, we propose an optimizer to generate optimized DO loops from FORALL construct using loop alignment. For this, we define and use relation distance vector to keep necessary informations. Then we evaluate and analyze execution time for the codes converted by our method and by PARADIGM method for various array sizes.

1. 서 론

병렬 컴퓨팅은 빠른 연산처리를 요구하는 과학자와 공학자들에게 컴퓨터의 연산 처리 속도를 지속적으로 증가시킬 수 있는 유일한 방법으로 현재 인식되고 있다. 특히 분산 메모리 컴퓨터는 매우 빠른 연산 처리 속도를 제공하고, 확장성과 이식성이 뛰어난 병렬 컴퓨터이다. 그러나 이러한 많은 장점에도 불구하고 병렬 컴퓨터는 프로그래밍의 어려움과 프로그램 작성 시간이 오래 걸리는 등의 이유로 인해 그 사용 범위가

* 이 논문은 1997년 고려대학교의 교내특별연구비에 의하여 연구되었음.
† 정 회 원 : 고려대학교 대학원 컴퓨터학과
†† 정 회 원 : 고려대학교 컴퓨터학과 교수
논문접수 : 1998년 5월 11일, 심사완료 : 1998년 12월 18일

제한되어 왔다. 병렬 컴퓨터를 이용하여 적절한 성능을 얻기 위해 프로그래머는 기존의 프로그래밍 방법과는 다른 병렬 언어로 프로그램을 작성하여야 함은 물론, 컴퓨터 기종에 따라 변하는 부가적인 사항도 고려하여야만 한다[4,7,14,17]. 프로그램을 개발, 디버깅 및 최적화 한 후에도 여전히 새로운 기종의 컴퓨터로 대체되거나 다른 병렬 컴퓨터에 이식될 때 쉽게 보완·적용될 수 있다는 것을 보장하기 어렵다.

분산 메모리 구조의 대규모 병렬 처리 시스템에서의 이러한 프로그래밍 문제점을 프로그래밍 언어 차원에서 해결하기 위해 제안된 것이 자료 병렬 언어이다[1,2,4,5,17]. 자료 병렬 언어는 공유 메모리 구조에서 장점으로 인식되었던 전역적 데이터 참조 기능을 분산 메모리 구조에서 제공하면서, 프로그래머가 분산 메모리 구조에 맞게 프로그램의 데이터를 손쉽게 분할할 수 있는 여러 종류의 지시어(directive)를 제공한다. 따라서 프로그래머는 지시어를 사용하여 분산 저장되고 처리될 데이터의 분산 형태를 지정하고, 배열 연산이나 FORALL 등의 병렬 구조로 프로그램 내 병렬 수행 부분을 표현함으로써 병렬성을 표현한다. 자료 병렬 프로그램은 순차수행부분과 병렬수행부분으로 구성된다. 이렇게 구성된 자료 병렬 언어 프로그램이 다수의 프로세서로 이루어진 병렬처리환경에서 실행될 때, 순차수행부분은 모든 프로세서가 같은 데이터에 대해 동일한 연산을 병렬로 중복수행하고, 병렬수행부분은 모든 프로세서가 각 프로세서별로 분할된 서로 다른 데이터에 대해 동일한 연산을 병렬로 수행한다.

자료 병렬 언어 프로그램은 과학 및 공학 연산의 특징인 동질성을 이용하는 것이므로, 프로그램은 방대한 양의 데이터에 대해 동일하게 적용되는 일련의 연산으로 구성된다. 많은 양의 데이터에 연산이 병렬로 똑같이 적용되기 때문에 병렬성의 정도는 문제의 크기에 따라 달라진다. 그러므로 많은 양의 계산을 요구하는 큰 문제(Grand Challenge Problems) 해결에 적용될 때 보다 높은 병렬성을 얻을 수 있다.

자료 병렬 언어의 종류는 매우 다양한데, 그 중에서도 기존의 순차 프로그래밍 언어에 자료 병렬성을 지원하기 위한 사양이 추가된 형태가 주류를 이룬다. 특히 과학 기술 계산 프로그램에서 유용하게 사용되던 포트란 언어에 자료 병렬성을 지원하도록 확장된 형태가 가장 일반적이다. 포트란 언어에 새로운 명세들이 추가된 대표적인 자료 병렬 언어로는 CM 포트란, 포트란 D, 포트란 90D/HPF, 비엔나 포트란, PCF 포트란, HPF(High Performance Fortran) 등이 있다[1,4,5,7,11,17].

프로그래머에 의해 작성된 자료 병렬 프로그램을 실제 분산 메모리 컴퓨팅 환경에서 수행될 수 있는 통신 라이브러리를 포함한 병렬 코드로 바꾸어주는 일은 자료 병렬 언어 컴파일러의 몫이다. 자료 병렬 언어별로 컴파일러도 각각 연구·개발되어 발표되고 있는데, 이들 대부분은 "source-to-source" 형태의 컴파일러라는 공통점이 있다. Source-to-source 컴파일러는 자료 병렬 언어로 작성된 프로그램을 입력으로 하여 여러 가지 프로그램 분석을 진행한 후, 분산 메모리 다중 컴퓨터 시스템 구조에서 병렬로 실행되기 위해 데이터 및 연산이 분할되고 프로세서간 통신에 필요한 메시지 패싱 프리미티브가 삽입된 형태의 새로운 소스 프로그램을 생성한다. 새롭게 생성된 소스 프로그램은 실행될 컴퓨터에 존재하는 포트란 77 컴파일러 등의 일반적인 컴파일러에 의해 컴파일되고 메시지 패싱 라이브러리와 링크되어, 그 컴퓨터 구조에 가장 알맞은 형태의 실행 코드가 생성된다.

이와 같이 source-to-source 컴파일러는 자료 병렬 프로그램으로부터 직접 실행 코드를 생성해내는 컴파일러에 비해 그 제작이 용이하고, 목적 코드가 새로운 소스 코드이므로 특정 컴퓨터 구조에 종속되지 않아 코드의 이식성이 좋아진다. 이러한 이유로 대부분의 자료 병렬 언어 컴파일러가 source-to-source 형태를 택하고 있는 것이다.

최근 다양한 자료 병렬 언어 가운데 분산 메모리의 대규모 병렬 시스템에 적합한 자료 병렬 언어의 표준으로 인식되고 있는 HPF의 대부분의 컴파일러도 HPF로 작성된 병렬 프로그램을 입력으로 받아 병렬 수행 가능한 포트란 77 프로그램으로 변환하는 source-to-source 형태이다[1,2,3,6,12,13]. HPF source-to-source 컴파일러가 HPF 프로그램을 병렬 수행 가능한 포트란 77 프로그램으로 변환하기 위해서는 우선 HPF 구문을 포트란 77 구문으로 변환하고 난 뒤, 포트란 77 구문만으로 구성된 프로그램에 대해 병렬화하는 순서로 진행되는 것이 일반적이다. 이처럼 FORALL 구조를 병렬화된 DO 루프로 직접 변환하지 않고 순차 DO 루프로 변환한 뒤 다시 병렬화하는 이유는 병렬화 가능한 순차 DO 루프의 병렬화와 관련된 연구가 이미 많이 이루어져 다양한 최적화 방안들이 제시되어 있기 때문

에 이들을 그대로 적용하기 위함이다[9,10,11,12,13,14, 15,16,17,18].

반면, 병렬화 단계의 선행 과정으로 HPF 고유의 구문을 그 시맨틱이 유지되는 포트란 77 구문으로 변환하는 과정은 소홀히 다루어져 왔다. 그 결과 목적 코드가 포트란 77 프로그램인 대부분의 source-to-source형 HPF 컴파일러들은 현재 HPF 프로그램을 포트란 77 코드로 변환하는 과정에서 상당량의 비효율적 코드를 생성하고 있다. 특히 HPF의 병렬 구조 FORALL을 그 병렬 수행의 의미 유지를 위해 다수의 포트란 77 DO 루프로 변환하고 있다. FORALL 구조에 포함된 문장수가 많아질수록 이러한 현상은 심화되어 변환 과정을 통해 생성된 다수의 DO 루프의 오버헤드로 인해 결과적으로 프로그램 전체의 성능이 저하되고 있다.

대량의 데이터에 대해 동일한 연산을 병렬로 실행시키는 병렬 수행 구조가 자료 병렬 언어의 주요 특성이고 병렬화 단계에서 주된 병렬성 추출의 대상이 루프임을 고려할 때, 컴파일러가 최적화된 목적 프로그램을 생성하기 위해서는 병렬화 단계에 앞서 HPF의 병렬구조로부터 포트란 77의 DO 루프로의 변환 과정에서도 루프 오버헤드를 줄이기 위한 최적화를 고려해야 한다. 따라서 본 논문에서는 HPF 프로그램의 병렬 구조를 포트란 77의 DO 루프로 변환하는 과정에서 가능한 한 적은 루프 오버헤드를 갖는 코드를 생성하기 위한 최적화기를 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 관련 연구와 그들의 문제점에 대해 기술한다. 3장에서는 본 논문에서 설계하는 최적화기의 필요 정보를 유지하는 수단으로 관계거리벡터를 정의하고, 이를 기반으로 하여 병렬 구조 FORALL의 구문 변환시 적용될 최적화기를 제시한다. 4장에서는 기존의 HPF 컴파일러 가운데 PARADIGM 컴파일러에 의해 생성된 코드와 본 논문에서 제시한 최적화기를 적용하여 생성된 코드에 대해 실시한 비교 성능평가 결과를 논한다. 마지막으로 5장에서 결론과 향후 연구에 대해 기술한다.

2. 기존 연구

기존의 HPF 컴파일러 중 본 논문에서 다루려고 하는 FORALL에서 DO 루프로의 변환에 대해 연구된 컴파일러로 미국 일리노이 대학의 PARADIGM[6]과 시라큐스 대학의 포트란 90D/HPF[1]를 들 수 있다. 이 장에서는 두 컴파일러에서 행해지는 FORALL에서 DO 루프로의 변환 방법과 그 문제점에 대해 기술한다.

2.1 PARADIGM 컴파일러

PARADIGM에서 FORALL을 DO 루프로 변환할 때, FORALL의 시맨틱을 유지하기 위해 복제(clone)와 루프 분할 방법이 사용된다. 복제 방법은 변환된 DO 루프에서 FORALL의 시맨틱 유지를 보장할 수 없는 경우 임시 배열을 생성하여 DO 루프 이전에 원래 배열을 임시 배열로 복사하는 것이다. (그림 1)은 복제 방법을 이용하여 FORALL문을 DO 루프로 변환한 예이다.

```
FORALL (I=1:100, X(I) .EQ. 0) X(I) = X(I-1) + X(I+1)
```

(a) FORALL문

```
CALL MEM_COPY(X_CLONE, X, 100)
DO I = 1, 100
  IF (X_CLONE(I) .EQ. 0)
    X(I) = X_CLONE(I-1) + X_CLONE(I+1)
  END IF
END DO
```

(b) 변환된 DO 루프

(그림 1) 복제 방법에 의해 FORALL문을 DO 루프로 변환한 예

(Fig. 1) A convsersion example of FORALL statement into DO loops using clone method

다음으로 다중 문장 FORALL 구조를 중첩 DO 루프로 변환할 때는, 반복 공간에 관계없이 선행 문장의 수행 결과가 후행 문장에 반영되어야 하는 FORALL의 시맨틱을 유지하기 위해 루프 분할 방법이 사용된다. (그림 2)는 다중 문장 FORALL 구조를 PARADIGM의 루프 분할 방법에 의해 변환한 예이다.

```
FORALL (J = 2:99, I = 2:99, A(I, J) .NE. B(I, J))
  C(I, J) = SQRT(A(I, J) * A(J, I))
  B(I, J) = (C(I+1,J+1)+C(I-1,J-1))/(B(I,J)-A(I,J))
END FORALL
```

(a) FORALL문

```
DO J = 2, 99
  DO I = 2, 99
    IF (A(I, J) .NE. B(I, J))
      C(I, J) = SQRT(A(I, J) * A(J, I))
    END IF
  END DO
END DO
DO J = 2, 99
  DO I = 2, 99
```

```
      IF (A(I, J) .NE. B(I, J))
        B(I, J) = (C(I+1, J+1) + C(I-1, J-1)) /
                  (B(I, J)   A(I, J))
      END IF
    END DO
END DO
```

(b) 2개의 중첩 DO 루프로 변환한 예

(그림 2) 루프 분할에 의해 FORALL 구조를 중첩 DO 루프로 변환한 예

(Fig. 2) A conversion example of FORALL construct into nested DO loops using loop distribution

PARADIGM 컴파일러에서 사용되는 FORALL에서 DO 루프로의 변환 방법들의 문제점은 다음과 같다. 복제 방법은 FORALL의 시맨틱 유지를 위해 필수적인 방법이긴 하지만, 임시 배열을 위한 메모리가 추가 사용되어야 하는 문제가 있다. 그러나 현재로서는 별다른 대안이 없는 것으로 보인다. 루프 분할 방법은 다중 문장 FORALL의 시맨틱을 유지하기 위한 간단한 변환 방법이긴 하지만, FORALL 구조내 문장의 수가 증가함에 따라 생성되는 DO 루프의 수도 증가됨으로 인해 루프 오버헤드가 커지게 되어 프로그램의 전체 성능이 저하되는 문제가 있다. 그런데 이 문제는 루프 오버헤드를 줄이기 위한 최적화 방안을 적용함으로써 충분히 개선될 수 있다. 그래서 본 논문에서는 다중문장 FORALL 구조로부터 최적화된 DO 루프를 생성하는 최적화기를 제안한다.

2.2 포트란 90D/HPF 컴파일러

포트란 90D/HPF 컴파일러에서 FORALL을 DO 루프로 변환할 때 사용하는 방법으로는 임시 기억장소 이용, 루프 교환, 마스크 삽입 기법이 있다[1]. 여기서 임시 기억장소 이용 방법은 2.1절에서 언급한 PARADIGM 컴파일러에서의 복제 방법과 동일하므로 생략한다.

다음으로 루프 교환은 FORALL이 중첩 DO 루프로 변환될 때, 중첩 루프의 두 레벨을 바꾸는 것을 말한다. 포트란은 열 우선 순서(column-major order)로 배열을 저장하므로, 포트란 90D/HPF 컴파일러는 FORALL triplet을 열 우선으로 정렬한다. (그림 3)은 (a)의 FORALL을 루프 교환에 의해 (b)의 DO 루프로 변환한 예이다.

```
FORALL (I=1:N, J=1:N) A(I, J) = B(I, J)
```

(a) FORALL문

```
DO J = 1, N
  DO I = 1, N
    A(I, J) = B(I, J)
  END DO
END DO
```

(b) 변환된 중첩 DO 루프

(그림 3) 루프 교환에 의해 FORALL문을 중첩 DO 루프로 변환한 예

(Fig. 3) A conversion example of FORALL statement into nested DO loops using loop interchange

마지막으로 마스크 삽입 기법에 대해 알아보자. (그림 4)(a)는 Gaussian Elimination 코드에 존재하는 FORALL문으로 마스크가 인덱스 j와는 관계없고 인덱스 i만 관계된다. 따라서 변환된 코드에서 IF문의 위치는 (그림 4)(b)에서와 같이 인덱스 j루프 바깥쪽이 되어, 내부의 루프는 벡터화 가능한 루프가 되도록 하는 것이다.

```
FORALL (i=1:N, j=1:NN, INDX(i) .EQ. -1) A(i, j) = A(i, j) -
        FAC(i) * ROW(j)
```

(a) FORALL문

```
DO i = 1, N
  IF (INDX(i) .EQ. -1) THEN
    DO j = 1, NN
      A(i, j) = A(i, j) - FAC(i) * ROW(j)
    END DO
  END IF
END DO
```

(b) 변환된 중첩 DO 루프

(그림 4) 마스크 삽입 방법에 의해 FORALL문을 중첩 DO 루프로 변환한 예

(Fig. 4) A conversion example of FORALL statement into nested DO loops using mask insertion

포트란 90D/HPF 컴파일러에서 사용되는 FORALL에서 DO 루프로의 변환 방법들의 문제점은 다음과 같다. 첫 번째 임시기억장소 이용은 PARADIGM 컴파일러에서와 마찬가지로 추가의 메모리 사용이 문제가 된다. 나머지 두 가지 방법은 HPF의 병렬 구조인 FORALL의 구문 변환시 적용될 최적화 방안이긴 하지만, 앞에서 지적한 바 있는 다수의 DO 루프 생성으로 인한 루프 오버헤드 증가 문제는 해결하지 못한다. 그러므로 컴파일러가 프로그램으로부터 종속성 등의 정보

를 추출하여 보다 일반적으로 적용할 수 있는 최적화 방안이 요구된다.

3. 제안 방안

포트란 90으로부터 확장된 HPF에서 제공하는 병렬 구조는 FORALL과 배열 연산(array operation)이다. 그런데 병렬 수행 구조 FORALL이 포트란 90의 배열 연산의 일반화된 형태이므로, 본 논문에서는 FORALL을 위주로 하여 DO 루프로의 변환시 적용할 최적화 방안을 제시한다.

병렬 구조 FORALL은 단일 문장 FORALL문과 다중 문장 FORALL 구조로 나누어진다[5,7]. 단일 문장 FORALL문에서는 모든 반복에 대해 배정문의 RHS (Right-Hand Side)를 계산한 후 LHS(Left-Hand Side)로의 배정이 이루어진다. 다중 문장 FORALL 구조에서도 각 문장에 대해 단일 문장 FORALL문에서의 시맨틱이 적용된다. 또한, 여러 문장으로 이루어진 FORALL 구조에서는 앞 문장의 수행결과가 반복 공간에 관계없이 뒤 문장에 반영되어야 한다는 시맨틱이 존재한다. 즉, FORALL 구조 내에서는 앞 문장에서 뒤 문장으로의 전향 종속성(Forward Data Dependence : 이하 FDD라 함)만이 존재한다.

HPF 컴파일러가 FORALL 구조를 DO 루프로 변환한 후 변환된 DO 루프에서 이러한 FORALL 구조의 시맨틱이 그대로 유지되어야 한다. 다시 말해서, FORALL 구조에서는 전향 종속성만이 발생할 수 있으므로 그 시맨틱이 유지되도록 변환된 DO 루프에서도 전향 종속성만이 발생해야 한다. 이러한 이유로 기존의 HPF 컴파일러들이 하나의 FORALL 구조로부터 다수의 DO 루프를 생성하고 있는 것이다.

본 논문에서 제안하는 최적화기는 FORALL 구조가 순차 DO 루프로 변환되면서 문장들간에 역향 종속성(Backward Data Dependence : 이하 BDD라 함)이 발생하는 경우에 대해 관계거리벡터라는 종속관계 정보를 기반으로 기존의 루프 변환 방법 가운데 루프 정렬(loop alignment)[13,14,15,18]을 적용하여 FORALL의 시맨틱이 유지되면서도 가능한 한 적은 수의 DO 루프를 생성하도록 한다. 이렇게 하여 변환된 코드는 기존의 HPF 컴파일러들에서 생성해내는 코드보다 적은 수의 DO 루프를 생성함으로써 루프 오버헤드를 줄여 결과적으로 프로그램의 성능이 향상 되도록 하고자 함이다.

3.1절에서는 FORALL 구조를 최적화된 DO 루프로 변환하는 과정에서 필요한 데이터 종속성 정보와 흐름 정보를 모두 표현하는 수단으로 관계거리벡터를 정의하고, 3.2절에서는 관계거리벡터 정보를 기반으로 하여 설계된 최적화기의 알고리즘에 대해 구체적으로 기술한다.

3.1 관계거리벡터

앞에서 언급한 바와 같이, 다중 문장 FORALL 구조에서는 앞 문장에서 뒤 문장으로의 전향 종속성이 존재한다. 다시 말해 FORALL 구조에서는 어떤 역향 종속성도 발생할 수 없다. FORALL 구조의 이러한 시맨틱을 변환된 DO 루프에서 유지하기 위해서는 문장간의 데이터 종속 정보는 물론 문장들간의 실행순서를 나타내는 흐름 정보도 필요하다.

현재 데이터 종속성을 표현하는 수단으로 방향벡터(direction vector)와 거리벡터(distance vector)가 많이 사용되고 있다[13,14,15,16]. n-단계 중첩루프에서 i번째 루프 인덱스에 대하여 반복 a내의 어떤 문장으로부터 반복 b내의 문장으로 데이터 종속성이 존재할 때, 같은 인덱스에 대한 반복 값의 차이를 거리벡터라 하고, 거리벡터 값이 양이면 방향벡터를 '+' 또는 '<'로, 거리벡터 값이 음이면 방향벡터를 '-' 또는 '>'로, 거리벡터 값이 0이면 방향벡터를 0 또는 '='로 표현한다. 이와 같이 거리벡터와 방향벡터는 종속성이 존재하는 두 문장 사이에서 '종속하는 문장에서 종속되는 문장으로의 종속방향 관점에서 같은 루프 인덱스 값의 차이'를 표현하고 있다. 따라서 종속성이 존재하는 두 문장 가운데 어느 문장이 먼저 나오고, 어느 문장이 나중에 나오는가 하는 내용을 나타내지 못한다. 즉, 거리벡터나 방향벡터의 값들만으로는 종속성 정보와 제어 흐름 정보를 표현하지 못한다.

한편, 관계벡터(relation vector)는 종속성이 존재하는 두 문장 사이에서 '선행문장에서 후속문장으로의 종속방향 관점에서 대응되는 루프 인덱스 값의 차이' 관계를 부등호와 등호 기호인 '>', '<', '='로 표현한다[9,10]. 그러므로, 종속성 정보와 흐름 정보를 동시에 나타낼 수는 있으나 종속거리 정보는 표현하지 못한다.

따라서 본 논문에서는 프로그램 문장들 사이의 종속거리를 포함한 데이터 종속 정보와 문장간의 실행순서인 흐름 정보를 모두 나타내도록 관계벡터를 수정한 관계거리벡터(relation distance vector)를 다음과 같이 정의하여 사용한다. 먼저, 프로그램 코드 상에서 종

속성이 존재하는 두 문장을 그 순서에 따라 구분할 때, 먼저 나오는 문장을 선행문장(predecessor statement)이라 하고, 나중에 나오는 문장을 후속문장(successor statement)이라 하자.

중첩 루프 내 i번째 문장 S_i의 최상위 루프로부터 k번째 루프에 해당하는 인덱스 식을 l_{ik}라고 할 때, 선행문장 S_i와 후속문장 S_j 사이의 k번째 루프 인덱스 식의 차이 관계를 rd_k라고 정의한다. 여기서 rd_k는 인덱스 식의 차이 관계에 따라 양수, 음수, 영(zero) 등으로 표현되고 다음의 세 경우가 존재한다(단, C는 양의 정수).

(1) rd_k = '+C' : l_{ik} = m, l_{jk} = m - C이고, '$l_{ik} - l_{jk}$'의 결과가 양수인 경우

(2) rd_k = '-C' : l_{ik} = m, l_{jk} = m + C이고, '$l_{ik} - l_{jk}$'의 결과가 음수인 경우

(3) rd_k = '0' : l_{ik} = l_{jk} = m이고, '$l_{ik} - l_{jk}$'의 결과가 영(zero)인 경우

n-단계의 중첩 루프 하에서 두 문장 S_i와 S_j 사이에 종속 관계가 존재할 때, n개의 루프 인덱스 식들의 차이관계의 집합을 관계거리벡터 RD_{ij}라고 하고, 다음과 같이 나타낸다.

$$RD_{ij} = (rd_1, ..., rd_k, ..., rd_n),\ 1 \le k \le n$$

여기서 n은 루프 인덱스의 최대 값이고, rd_k는 문장 S_i와 S_j 사이에 존재하는 종속 관계의 최상위 루프로부터 k번째 루프 인덱스 식의 차이 값을 나타낸다. 그리고 관계거리벡터의 각 인덱스별 항목 값들의 절대값 $|rd_k|$이 각 루프 인덱스에 대한 종속거리이다.

단일 항목을 갖는 관계거리벡터에 대하여 그 값이 양수이면 전향 종속성을, 음수이면 역향 종속성을 나타낸다. 이 관계를 정리하면 다음과 같다.

(1) RD_{ij} = +C이면, S_j가 S_i에 전향 종속(FDD), LCD(Loop Carried Dependence)된다.

(2) RD_{ij} = -C이면, S_i가 S_j에 역향 종속(BDD), LCD된다.

(3) RD_{ij} = 0이면, S_j가 S_i에 전향 종속(FDD), LID(Loop Independent Dependence)된다.

여기서 LCD는 서로 다른 두 반복(iterations)내 두 문장간에 데이터 종속 관계를 나타내고, LID는 같은 반복 내 두 문장간에 데이터 종속 관계를 나타낸다[14,15].

이와 같은 성질을 중첩 루프 내 여러 문장이 존재하고 그 문장들에서 참조하는 여러 변수간에 종속관계가 존재하는 경우로 확장하면 다음과 같다.

"여러 문장간의 종속관계를 나타내는 관계거리벡터의 항목들 가운데 음수 항목 값이 적어도 하나 존재하면 루프 내 문장들 사이에는 역향 종속성이 발생한다".

3.2 변환 방법

본 논문은 선행 연구의 결과 논문으로 이미 발표된 바 있는 [8]을 일반화한 것이다. [8]에서는 고려대상을 두 개의 배정문으로 이루어진 FORALL 구조내 한 쌍의 변수사이에 역향 종속성이 존재하는 경우로 한정하였고, 관계거리벡터의 유형이 (-C, 0) 또는 (0, -C)인 경우를 CASE 1, (-C, +C)인 경우를 CASE 2, (-C, -C)인 경우를 CASE 3으로 각각 나누어 CASE 1과 CASE 3에는 루프 정렬 방법을, CASE 2에는 루프 교환(loop interchange) 방법을 적용하여 최적화된 DO 루프를 생성하는 최적화 방안을 제안하였다(단, C는 양의 정수). 그리고 비교 대상이 되었던 PARADIGM 컴파일러의 변환 방법에 의해 생성된 코드보다 적은 수의 DO 루프가 생성됨에 따라 루프 오버헤드가 줄어들어 결과적으로 수행시간이 짧아지는 결과를 보였다.

본 논문에서는 [8]의 고려대상을 확장하여 다음과 같이 일반화하고자 한다. 본 논문에서 제안하는 최적화기가 고려하는 대상은 다음과 같다.

(1) **세 개이상의 배정문**으로 이루어진 FORALL 구조로 MASK 수식은 포함하지 않으며,

(2) 문장들에서 참조(정의-사용 또는 사용-정의)되는 **변수들은 *2차원 배열 요소*이고, 데이터 종속관계에 있는 변수들은 *일정한 종속거리***를 갖는다.

본 논문에서 제안하는 최적화기에서 변환시 적용되는 기본 원칙은 다음과 같다.

첫째, 포트란 90D/HPF 컴파일러에서 적용되었던 루프 교환 방법[1]을 적용한다. 즉, FORALL 구조를 DO

루프로 변환할 때 포트란 77의 열 우선 저장 방식을 감안하여 루프 내에서 배열 요소들이 열 우선 순서로 참조될 수 있도록 루프를 위치시킨다. 2차원 배열을 참조하는 FORALL 구조가 변환된 DO 루프에서는 열 인덱스 j에 대한 루프를 바깥쪽에, 행 인덱스 i에 대한 루프를 안쪽에 위치하도록 하여 메모리 접근시간을 줄인다.

둘째, FORALL 구조내 두 문장간에 발생하는 역향 종속성을 제거하기 위해 루프 변환 방법 가운데 루프 정렬 방법을 적용한다. 역향 종속성은 루프 내 선행문장에서 참조되는 배열의 인덱스 값이 후속문장에서 참조되는 배열 인덱스 값보다 작기 때문에 발생한다. 그런데 루프 정렬은 루프 내 문장들에서 서로 다른 반복 공간을 접근함으로써 발생하는 종속성을 제거하여 병렬화를 용이하게 하는 최적화 방법이다[12, 13, 14, 15, 16]. 그러므로 본 논문에서 다루고자 하는 경우에 발생하는 역향 종속성을 제거하기 위한 방법으로 적절하다.

병렬 구조 FORALL을 최적화된 DO 루프로 변환하는 최적화기는 FORALL 구조 내 문장들에서 종속관계에 있는 배열들에 대해 종속성 검사에 의해 구해진 모든 관계거리벡터를 기반으로 하여 다음과 같이 작동된다.

(1) 구해진 관계거리벡터 항목 값들 가운데 음수 항목 값을 갖는 항목이 존재하는지 검사하고, 한 개 이상의 음수 항목 값이 존재할 때 모든 항목 값들 가운데 최소 값의 절대값(이하 최대 종속거리라 함)을 구한다.

(2) 관계거리벡터 항목 값들 가운데 음수 항목 값을 갖는 종속관계가 적어도 하나 존재하면 변환된 DO 루프내 문장들간에 역향 종속성이 발생하므로, 이를 제거하기 위해 FORALL 구조 내 첫 번째 문장을 기준으로 이후 문장들에 대해 루프 정렬 방법을 적용하여 DO 루프로 변환한다. 이 때 루프 정렬은 모든 루프 인덱스에 대해 적용되고 최대 종속거리 만큼의 문장이 정렬되어야 모든 역향 종속성이 제거된다.

(3) 관계거리벡터에 음수 항목 값이 존재하지 않으면 문장들 간에 어떤 역향 종속성도 발생하지 않으므로, FORALL 구조를 별다른 변환없이 그대로 DO 루프로 변환한다.

예제 코드를 통해 변환 과정을 구체적으로 살펴보자. (그림 5)(a)는 FORALL 구조내 세 문장에서 세 쌍의 변수간에 서로 다른 종속거리를 갖는 종속관계가 존재하는 예제이다. 우선 (그림 5)(a)에서 문장 S1, S2, S3 간의 관계거리벡터는 배열 A에 대해 (-1, -1)과 (-2, -2), 배열 B에 대해 (0, 1)이다. 세 관계거리벡터 가운데 (-1, -1)과 (-2, -2)에 음수 항목 값이 존재하므로 문장들간에 배열 A에 대해 역향 종속성이 발생한다는 것을 알 수 있다. 그러므로 FORALL의 시맨틱 유지를 위해 루프 정렬을 적용하여 DO 루프로 변환하여야 한다. 역향 종속성을 발생시키는 관계거리벡터의 항목 값들 가운데 최대 종속거리는 |-2| = 2이다. 그러므로 FORALL 구조내 첫 번째 문장과 이후 문장들 각각에 대하여 최대 종속거리 2만큼 두 인덱스 i, j에 대해 루프 정렬을 적용하여 생성된 코드가 (그림 5)(c)이다. (그림 5)(c)의 1라인~4라인과 6라인~7라인까지가 첫 번째 문장에 대해, 13라인~14라인과 19라인~20라인까지가 두 번째 문장에 대해, 15라인~16라인과 21라인~22라인까지가 세 번째 문장에 대해 최대 종속거리 2만큼 루프 정렬을 적용하여 새로이 생성된 코드이다. 이와 같이 하여 생성된 코드에서 발생하는 루프 오버헤드와 기존의 PARADIGM 컴파일러에서 루프 분할을 적용하여 변환해 낸 (그림 5)(b)의 코드에서 발생하는 루프 오버헤드를 비교해 보자. (그림 5)(c)에서 발생하는 루프 반복 횟수는 '(n-2) + (m-5)×(n-4) + (n-2)'이고, (그림 5)(b)에서 발행하는 루프 반복 횟수는 '3×(m-3)×(n-2)'이다. 따라서 (그림 5)(b) 코드가 상대적으로 적은 루프 반복을 수행하는 (그림 5)(c) 코드에 비해 큰 루프 오버헤드를 발생시켜 이로 인한 성능 저하를 보이게 된다.

```
    FORALL(i=1:n-2,  j=2:m-2)
S1    A(i, j) = B(i, j) + 1
S2    B(i, j-1) = A(i+1, j+1) + 3
S3    C(i, j) = A(i+2, j+2) + B(i, j-1)
    END FORALL
```

(a) Original FORALL 구조

```
    DO j = 2, m-2
      DO i = 1, n-2
S1      A(i, j) = B(i, j) + 1
      END DO
    END DO
    DO j = 2, m-2
      DO i = 1, n-2
S2      B(i, j-1) = A(i+1, j+1) + 3
      END DO
    END DO
    DO j = 2, m-2
      DO i = 1, n-2
S3      C(i, j) = A(i+2, j+2) + B(i, j-1)
```

```
    END DO
  END DO
```

(b) 루프 분할을 적용하여 변환된 DO

```
1   DO i = 1, n-2
2     A(i, 2) = B(i, 2) + 1
3     A(i, 3) = B(i, 3) + 1
4   END DO
5   DO j = 4, m-2
6     A(1, j) = B(1, j) + 1
7     A(2, j) = B(2, j) + 1
8     DO i = 3, n-2
9       A(i, j) = B(i, j) + 1
10      B(i-2, j-3) = A(i-1, j-1) + 3
11      C(i-2, j-2) = A(i, j) + B(i-2, j-3)
12    END DO
13    B(n-3, j-3) = A(n-2, j-1) + 3
14    B(n-2, j-3) = A(n-1, j-1) + 3
15    C(n-3, j-2) = A(n-1, j) + B(n-3, j-3)
16    C(n-2, j-2) = A(n, j) + B(n-2, j-3)
17  END DO
18  DO i = 1, n-2
19    B(i, m-4) = A(i+1, m-2) + 3
20    B(i, m-3) = A(i+1, m-1) + 3
21    C(i, m-3) = A(i+2, m-1) + B(i, m-4)
22    C(i, m-2) = A(i+2, m) + B(i, m-3)
23  END DO
```

(c) 루프 정렬을 적용하여 변환된 DO

(그림 5) 관계거리벡터 RD_{12} = {(-1, -1), (0, 1)}, RD_{13} = {(-2, -2), (0, 1)}, RD_{23} = {(-1, -1)}인 경우의 예
(Fig. 5) A conversion example of the case which has relation distance vectors RD_{12} = {(-1, -1), (0, 1)}, RD_{13} = {(-2, -2), (0, 1)}, and RD_{23} = {(-1, -1)}

다음으로 (그림 6)(a)는 FORALL 구조내 세 문장에서 세 쌍의 변수간에 서로 다른 종속거리를 갖는 또 다른 예제로 이 경우는 서로 다른 배열 A와 B가 모두 역향 종속성을 갖는 경우이다. 세 문장 S_1, S_2, S_3 간의 관계거리벡터 가운데 역향 종속성을 일으키는 관계거리벡터는 배열 A에 대해 (-1, -1), 배열 B에 대해 (-2, 0)이다. 음수 항목 값이 존재하므로 문장들간에 배열 A와 B에 대해 모두 역향 종속성이 발생함을 알 수 있고, 역향 종속성을 발생시키는 관계거리벡터의 항목 값들 가운데 최대 종속거리는 |-2| = 2이다. 그러므로 FORALL 구조내 첫 번째 문장과 이후 문장들 각각에 대하여 최대 종속거리 2만큼 두 인덱스 i, j에 대해 루프 정렬을 적용하여 생성된 코드가 (그림 6)(b)이다.

```
    FORALL(i=2:n-2, j=2:m)
S1    B(i, j) = A(i-1, j-1) + 2
S2    A(i, j) = B(i, j) + 1
S3    C(i, j) = A(i, j) + B(i+2, j)
    END FORALL
```

(a) Original FORALL 구조

```
DO i = 2, n-2
  B(i, 2) = A(i-1, 1) + 2
  B(i, 3) = A(i-1, 2) + 2
END DO
DO j = 4, m
  B(2, j) = A(1, j-1) + 2
  B(3, j) = A(2, j-1) + 2
  DO i = 4, n-2
    B(i, j) = A(i-1, j-1) + 2
    A(i-2, j-2) = B(i-2, j-2) + 1
    C(i-2, j-2) = A(i, j-2) + B(i, j-2)
  END DO
  A(n-3, j-2) = B(n-3, j-2) + 1
  A(n-2, j-2) = B(n-2, j-2) + 1
  C(n-3, j-2) = A(n-3, j-2) + B(n-1, j-2)
  C(n-2, j-2) = A(n-2, j-2) + B(n, j-2)
END DO
DO i = 2, n-2
  A(i, m-1) = B(n-3, m-1) + 1
  A(i, m) = B(n-2, m) + 1
  C(i, m-1) = A(i, m-1) + B(i+2, m-1)
  C(i, m) = A(i, m) + B(i+2, m)
END DO
```

(b) 루프 정렬을 적용하여 변환된 DO

(그림 6) 관계거리벡터 RD_{12} = {(-1, -1)}, RD_{13} = {(-2, 0), (-1, -1)}, RD_{23} = {(-2, 0)}인 경우의 예
(Fig. 6) A conversion example of the case which has relation distance vectors RD_{12} = {(-1, -1)}, RD_{13} = {(-2, 0), (-1, -1)}, and RD_{23} = {(-2, 0)}

(그림 7)은 본 논문에서 제안한 FORALL 구조 변환시 적용될 최적화기의 코드 생성 알고리즘이다. (그림 7)에서 사용되는 용어들은 다음과 같다.

```
Algorithm : gen-transformed-code()
input : I1, I2, S1, S2,...,Sn, RD = {(C1^l,C1^l),...,(Cr^r, Cj^r),...,(Cd^d, Cj^d)},
        (1≤r≤d)
output : Transformed DO loops
  begin
1       /* Initialize */
2       Need_Align = 0
3       Min_Dist = 0
4
5   /* Check one or more relation distance
       vectors to examine any negative entry */
6       for each (Cr^r, Cj^r)∈RD do
7         if (Cr^r<0) or Cj^r<0) then
8           Need_Align = 1
9
```

```
10  /* Find minimum dependence distance */
11      if (Min_Dist > C_i^r) then
12          Min_Dist = C_i^r
13      end if
14      if (Min_Dist > C_j^r) then
15          Min_Dist = C_j^r
16      end if
17    end if
18  end for
19
20  /* Generate DO loops through loop alignment
       to eliminate backward dependence */
21  if (Need_Align = 1) then
22      Max_Dist = abs(Min_Dist)
23      gen(do I1 = i_l1, i_u1)
24        peel off S1 for [j_l2, j_l2 + Max_Dist - 1]
25      gen(end do)
26      gen(do I1 = j_l2 + Max_Dist, j_u2)
27        peel off S1 for [i_l1, i_l1 + Max_Dist - 1]
28        gen(do I2 = i_l1 + Max_Dist, i_u1 / S1)
29          locate S2 with aligning each index
                expression as many as Max_Dist
30          locate S3 with aligning each index
                expression as many as Max_Dist
31                  :
32          locate Sn with aligning each index
                expression as many as Max_Dist
33        gen(end do)
34        peel off S2 for [i_u1 - Max_Dist + 1, i_u1]
35        peel off S3 for [iu1 - Max_Dist + 1, iu1]
36                  :
37        peel off Sn for [i_u1 - Max_Dist + 1, i_u1]
38      gen(end do)
39      gen(do I1 = i_l1, i_u1)
40        peel off S2 for [j_u2 - Max_Dist + 1, j_u2]
41        peel off S3 for [j_u2 - Max_Dist + 1, j_u2]
42                  :
43        peel off Sn for [ju2 - Max_Dist + 1, ju2]
44      gen(end do)
45
46  /* Generate DO loops without any code
       transformation */
47      else
48        gen(do I1 = j_l2, j_u2 / do I2 = i_l1, i_u1 / S1
            / S2 / ... / Sn / end do / end do)
49      end if
    end
```

(그림 7) 코드 생성 알고리즘
(Fig. 7) Code Generation Algorithm

I_1, I_2 : 루프 인덱스 i, j의 반복 공간(iteration space)

$S_1, S_2, ..., S_n$: FORALL 구조내 배정문

RD : 문장 S_1, S_2, ..., S_n 사이에 존재하는 d가지의 종속관계에 대한 관계거리벡터들의 집합 $RD = \{(C_i^1, C_j^1), ..., (C_i^r, C_j^r), ..., (C_i^d, C_j^d)\}$, where $1 \le r \le d$

i_{l1}, i_{u1} : 루프 인덱스 i의 하한값과 상한값

j_{l2}, j_{u2} : 루프 인덱스 j의 하한값과 상한값

(그림 7)의 6~18라인은 문장 S_1, S_2, ..., S_n간의 관계거리벡터들 가운데 음수 항목 값이 존재하는지 검사하고, 존재할 경우 최대 종속 거리를 구하는 부분이다. 22~44라인은 역향 종속성이 존재하는 경우 루프 정렬을 적용하여 DO 루프로 변환하는 부분이고, 48라인은 나머지 경우 즉 어떤 역향 종속성도 발생하지 않는 경우 별다른 코드 변환 없이 DO 루프로 변환하는 부분이다. 22라인의 abs()는 절대값을 구하는 함수이고, gen()은 괄호 안에 기술된 문장을 생성하는 함수로 여러 문장을 기술할 때는 슬래쉬('/')로 구분한다.

4. 성능 평가

이 장에서는 FORALL 구조에 대해 본 논문에서 설계한 최적화기에 의해 변환된 코드와 PARADIGM 컴파일러에 의해 변환된 코드에 대한 비교 성능 평가를 행한다. 4.1절에서는 성능 평가를 실시할 대상 코드와 평가 방법에 대해 기술하고, 4.2절에서 성능 평가 결과를 분석한다.

4.1 성능 평가 방법

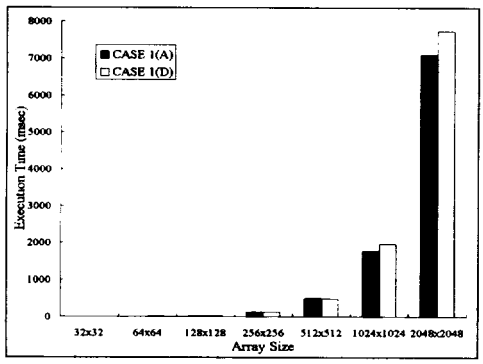
본 논문의 성능 평가 대상으로 세 개의 문장으로 이루어진 FORALL 구조에서 다양한 역향 종속성이 존재하고 최대종속거리가 2, 5, 10, 50인 경우를 각각 고려한다. 최대종속거리가 2인 경우의 FORALL 구조 예는 (그림 5)(a)이고, 이를 루프 분할에 의해 변환한 (그림 5)(b) 코드를 CASE1(D), 루프 정렬을 적용하여 변환한 (그림 5)(c) 코드를 CASE1(A)라 한다. 그리고 최대종속거리가 5인 경우의 FORALL 구조 예제를 작성하여 루프 분할 방법에 의한 코드를 CASE2(D), 루프 정렬 방법에 의한 코드를 CASE2(A)라 한다. 최대종속거리가 증가하면 루프 분할 변환 방법에 의한 코드에는 별다른 영향이 없고, 본 논문에서 제안하는 루프 정렬 변환 방법에 의한 변환 코드에서는 최대종속거리만큼 정렬되는 문장 수가 증가한다. 따라서 최대종속거리가 5, 10, 50인 경우의 예제는 (그림 5)의 코드와 유사하므로 생략한다.

HPF source-to-source 컴파일러의 목적 코드가 포트란 77이므로 성능 평가 대상 프로그램들을 모두 포트란 77로 작성한 다음, 프로그래밍의 편의를 위해 포

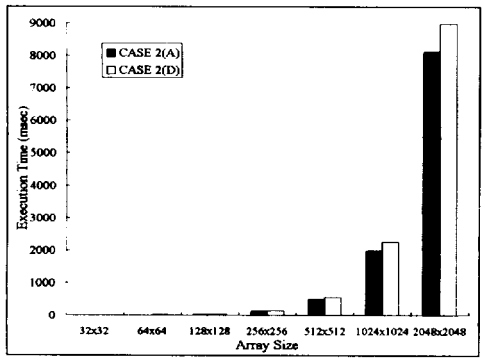
트란의 특성이 반영된 C 코드로 변환하였다. 이때 사용된 툴은 'f2c'로, 열 우선 저장 순서 등의 포트란 특징을 반영하여 포트란 코드를 C 코드로 변환한다. 이렇게 해서 생성된 C 코드의 루프 이전과 이후에 UNIX C의 시간 측정 함수인 'ftime()'을 각각 삽입한 후, gcc 컴파일러에 의해 최종 실행 코드를 생성하였다.

4.2 성능 평가 결과 및 분석

4.1절에서 기술한 방법으로 생성된 코드에 대해 배열 크기를 32×32부터 2048×2048까지 변화시켜 가며 Sun SPARC 2상에서 수행시간을 측정한 결과가 각각 (그림 8)과 (그림 9)이다.



(그림 8) 세 문장내 여러 변수간에 역향 종속성이 존재할 경우의 수행 시간(최대 종속거리 = 2)
(Fig. 8) An execution result of the case which exists backward dependences among several variables in three statements(max. dependence distance = 2)



(그림 9) 세 문장내 여러 변수간에 역향 종속성이 존재할 경우의 수행 시간(최대 종속거리 = 5)
(Fig. 9) An execution result of the case which exists backward dependences among several variables in three statements(max. dependence distance = 5)

즉, (그림 8)은 FORALL 구조내 세 개의 문장간에 최대 종속거리 2인 역향 종속성이 존재하는 경우인 CASE1의 수행시간 결과이고, (그림 9)는 FORALL 구조내 세 개의 문장간에 최대 종속거리 5인 역향 종속성이 존재하는 경우인 CASE2의 수행시간 결과이다. 최대 종속거리에 관계없이 전반적으로 모든 배열 크기에 대해 본 논문에서 제안한 루프 정렬 방안에 의한 코드의 수행시간((그림 8)의 CASE1(A)와 (그림 9)의 CASE2(A))이 PARADIGM의 루프 분할에 의한 코드의 수행시간((그림 8)의 CASE1(D)와 (그림 9)의 CASE2(D))보다 적은 것으로 나타났다. 이러한 결과는 루프 오버헤드의 차이에 기인한다. 즉, 루프 분할에 의해 변환된 코드는 원래의 세 배에 달하는 루프 반복횟수를 갖는데 비해, 본 논문에서 제안한 방법에 의해 변환된 코드는 원래의 루프 반복횟수와 거의 같은 반복횟수를 가진다. 이로 인한 루프 오버헤드의 차이가 (그림 8)과 (그림 9)의 결과를 보인 것이다. 또한 배열의 크기가 커질수록 이 차이가 커지게 되는데, 대부분의 과학 계산용 프로그램에서 사용되는 배열의 크기가 매우 큰 것을 감안하면 그 의미가 크다고 하겠다.

한편, 최대종속거리가 2, 5, 10, 50인 예제 코드에 대해 여러 차례의 실험을 거쳐 얻어진 결과를 비교해 보니, 최대종속거리가 커짐에 관계없이 루프 오버헤드를 줄임으로써 얻어지는 일정한 정도의 성능향상을 보이는 것을 알 수 있었다. 그래서 본 논문의 성능평가 결과로 최대종속거리가 2와 5인 경우만을 제시하였다.

이제 본 논문에서 제안하는 최적화기의 코드 생성 알고리즘의 시간 복잡도(time complexity)를 살펴보자. (그림 7)의 알고리즘에서 최대종속거리를 구하는 부분(6라인~18라인)은 FORALL 구조를 구성하는 문장들간에 존재하는 종속관계의 수만큼 반복된다. 즉, d개의 종속관계가 존재하는 경우 d번의 비교를 반복 수행하면서 최대종속거리를 구한다. FORALL 구조를 구성하는 문장들간에 역향 종속성이 존재하지 않는 경우(48라인)는 단순히 FORALL 구문만을 DO 구문으로 변환하는 과정이 1회 실행된다.

그러나 역향 종속성이 존재하는 경우(22라인~44라인) 첫 번째 문장을 기준으로 하여 모든 문장들에 최대종속거리만큼 루프정렬을 반복적으로 적용하여 문장들을 생성하면서 DO 루프로 변환한다. 이때 FORALL 구조내 문장수가 n, 최대종속거리가 k 라 할 때, 이 부분의 복잡도는 다음과 같다.

$O(k+k+(n-1)\times k+(n-1)\times k)=O(2kn)\approx O(kn)$

비교대상인 PARADIGM 컴파일러의 FORALL 구조 변환 알고리즘의 복잡도는 특별한 최적화를 수행하지 않으므로 문장 수에 비례하는 $O(n)$이다. 그러므로 HPF 컴파일러에 본 논문의 최적화기를 적용함에 따라 컴파일시의 시간 복잡도는 최대종속거리인 k배만큼 증가함을 알 수 있다. 컴파일러 단계에서 최적화 단계를 거치면 컴파일시 시간 복잡도는 증가하지만 그 결과 생성되는 코드의 실행시간은 단축되는 것이 일반적이고, 복잡도의 증가 정도가 그다지 크지 않으므로 가능성 있는 최적화기라 할 수 있겠다.

5. 결론 및 향후연구

대부분의 자료 병렬 언어와 마찬가지로 대다수의 HPF 컴파일러가 "source-to-source" 방식을 취하고 있다. 그러나 이러한 형태의 HPF 컴파일러가 병렬 언어로 작성된 프로그램을 그 의미가 그대로 유지되는 순차 프로그램으로 변환하는 과정에서 상당량의 비효율적 코드를 생성하고 있다. 특히, HPF의 병렬 수행 구조인 FORALL은 그 시맨틱 유지를 위해 다수의 DO 루프로 변환되어, 이 다수의 루프들로 인한 루프 오버헤드가 프로그램 성능의 저하를 초래하고 있다.

이러한 문제를 해결하려는 연구들이 PARADIGM 등의 일부 HPF 컴파일러에서 행해지기는 했으나 아직은 미흡한 단계이다. 그래서 본 논문에서는 FORALL 구조를 가능한 한 적은 수의 최적화된 DO 루프로 변환하여 다수의 루프 생성으로 인한 오버헤드를 줄이는 최적화기를 제안하였다. 그리고 이를 통해 변환된 코드와 PARADIGM 컴파일러에 의해 변환된 코드의 수행 시간을 비교 측정한 결과, 모든 배열 크기에 대해 PARADIGM 컴파일러에 의한 코드보다 나은 성능을 보였다.

앞으로 본 논문에서 설계한 최적화기를 본 연구진이 구현한 프로토타입 HPF 컴파일러에 실제 적용하고, 또 보다 폭 넓은 경우에의 적용을 위해서는 적용 대상을 가변적인 종속거리를 갖는 경우나 MASK 수식이 존재하는 FORALL 구조 등으로 확장되어야 할 것이다.

참 고 문 헌

[1] Z. Bozkus, "Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers," PhD Thesis, Syracuse Univ., June, 1995.

[2] T. Brandes, "Compiling Data Parallel Programs to Message Passing Programs for Massively Parallel MIMD Systems," GMD, St. Augustin, Germany, March, 1994.

[3] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," Supercomputing 2, pp.151-169, 1988.

[4] B. M. Chapman, P. Mehrotra and H. P. Zima, "Vienna Fortran-A Fortran Language Extension for Distributed Memory Multiprocessors," Univ. of Vienna, Vienna, Austria, 1991.

[5] High Performance Fortran Forum, "High Performance Fortran Language Specification, Version 1.0," Tech. Report CRPC-TR92225, Center for Research on Parallel Computation, Rice Univ., Houston, Tex, 1993.

[6] E. W. Hodges IV, "High Performance Fortran Support for the PARADIGM Compiler," Master's Thesis, Univ. at Urbana-Champaign, 1995.

[7] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr, M. E. Zosel, "The High Performance Fortran Handbook," The MIT Press, 1994.

[8] M. S. Koo, S. S. Park, H. G. Yook, and M. S. Park, "A New Transformation Method to Generate Optimized DO Loop from FORALL Construct," In the Proc. of 2nd Aizu Int'l Symp. on Parallel Algorithms/Architectures Synthesis, pp. 240-247, Aizu-Wakamatsu, Japan, March, 1997.

[9] S. S. Park, "The Vectorization of Program using Graph Type Intermediate Representation Form," Ph.D Thesis, Korea Univ., Dept. of Computer Science, December, 1993.

[10] S. S. Park and M. S. Park, "The Vectorization of Program using the Relation Vector," In the Proc. of Int'l Conf. on Parallel and Distributed Systems, Taipei, Taiwan, December, 1993.

[11] C. D. Polychronopoulos et. al., "Parafrase-2 : An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors," In the Proc. of the 18th Int'l Conf. on

Parallel Processing, pp.II : 39-48, St. Charles, IL, August, 1989.

[12] C. W. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines," Tech. Report Rice COMP TR93-199, January, 1993.

[13] M. Wolfe, "Optimizing Supercompilers for Supercomputers," Addison-Wesley Publishing Company, 1995.

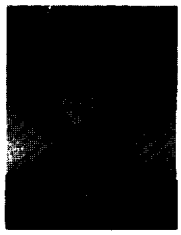
[14] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D Thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Tech. Report No.UIUCDCS-R-82-1105, Oct., 1982.

[15] M. Wolfe, "High Performance Compilers for Parallel Computing," Addison-Wesley Publishing Company, 1995.

[16] H. Zima and B. Chapman, "Supercompilers for Parallel and Vector Computers," ACM Press, 1990.

[17] H. Zima, H. Brezany, B. Chapman, P, Mehrora, and A. Schwald, "Vienna Fortran-a language specification," Austrian Center for Parallel Computation, Univ. of Vienna, Vienna, Austria, 1991.

[18] 육현규, 구미순, 박성순, 박명순, "고성능 자료병렬 언어 컴파일러에서의 최적화", 병렬처리시스템연구회지, 제7권, 제2호, pp.20-35, 1996년 10월.



구 미 순

e-mail : kms@cslab1.korea.ac.kr

1987년 고려대학교 수학과 졸업(학사)

1997년 고려대학교 컴퓨터학과(이학석사)

1997년~현재 고려대학교 컴퓨터학과 박사과정

관심분야 : 병렬 언어 및 컴파일러

박 명 순

e-mail : myongsp@cslab3.korea.ac.kr

1975년 서울대학교 전자공학과 졸업(학사)

1982년 University of Utah 전기공학과(공학석사)

1985년 University of Iowa 컴퓨터공학과(공학박사)

1975년~1980년 국방과학연구소 연구원

1985년~1987년 Marquette University 전기 및 전산과학과 조교수

1987년~1988년 포항공과대학 전자전기공학과 및 전산과학과 조교수

1988년~현재 고려대학교 이과대학 컴퓨터학과 교수

관심분야 : 컴퓨터구조, 병렬언어 및 컴파일러