고성능 언어에서의 병렬 태스크 생성에 관한 연구

박 성 순† · 구 미 순††

요 약

포트란 M 등의 태스크 병렬언어에서는 프로그래머가 태스크 병렬구조를 사용하여 프로그래밍한다. 그런데 응용 프로그램에서 프로시쥬어간에 종속성 관계가 존재하는 경우 프로그래머가 이 종속성을 고려하여 태스크 병렬 프로그램을 작성하기는 쉽지 않다. 그러므로 컴파일러 단계에서 묵시적 병렬성을 추출한 후, 태스크 병렬 언어에서 제공하는 병렬구조로 변환하는 병렬화가 필요하다. 그러나 현재의 태스크 병렬언어 컴파일러에서는 이러한 기능을 제공하지 못하고 있다. 본 논문에서는 종속성 관계에 따라 각 경우를 분석하여, 순차 수행되어야 하는 루프 구조에 대해 컴파일러 단계에서 포트란 M의 태스크 병렬 구조인 PROCESSDO 루프와 PROCESSES 블록구조로 병렬화하기 위해 묵시적 병렬성을 가지고 있는 경우를 추출하는 방안을 제안한다. 그리고 PROCESSDO 루프와 PROCESSES 블록 구조 모두로 병렬화 가능한 경우, 조건에 따라 어느 구조로 변환하는 것이 효과적인가를 분석한다.

A Study on Generation of Parallel Task in High Performance Language

Sung-Soon Park† · Mi-Soon Koo††

ABSTRACT

In task parallel language like Fortran M, programmer writes a task parallel program using parallel constructs which is provided. When some data dependencies exist between called procedures in various applications, it is difficult for programmer to write program according to their dependencies. Therefore, it is desirous that compiler can detect some implicit parallelisms and transform a program to parallelized form by using the task parallel constructs like PROCESSES block or PROCESSDO loop of Fortran M. But current task parallel language compilers can't provide these works. In this paper, we analyze the cases according to dependence relations and detect the implicit parallelism which can be transformed to task parallel constructs like PROCESSES block and PROCESSDO loop of Fortran M. Also, for the case which program can be paralleized both PROCESSES block and PROCESSDO loop, we analyze that which construct is more effective for various conditions.

1. 서 론

고성능 태스크 병렬언어에서는 프로그래머가 독립 수행가능한 프로시쥬어 단위의 병렬성을 고려하여, coarse grain 단위의 병렬성인 태스크 병렬구조로 프로그래밍한다. 이때 프로시쥬어들간에 어떠한 자료 종속관계도 존재하지 않는다면, 여러 프로시쥬어들을 태스크 병렬구조로 쉽게 변환하여 태스크 병렬 프로그램을 작성할 수 있다. 현재 사용되는 많은 대규모

※이 논문은 1995년 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음.

† 정 회 원:안양대학교 전자계산학과 조교수
†† 정 회 원:고려대학교 컴퓨터학과 박사과정

논문접수:1996년 10월 24일, 심사완료:1997년 4월 25일

응용 프로그램들이 독립 수행가능한 프로시쥬어 단위로 작성되어 있기 때문에, 포트란 M, Fx 포트란 등의 태스크 병렬언어로 변환하여 병렬 수행함으로써 많은 효과를 보고 있다[8, 10].

그런데 응용 프로그램에는 프로시쥬어들간에 매개변수 전달이나, 전역변수(global variable : 또는 nonlocal variable) 사용으로 인한 자료종속 관계가 존재하는 경우가 많다. 이러한 경우 프로그래머가 종속성을 고려하여 통신 프리미티브를 삽입하면서 프로그래밍하는 것이 어려워져 프로그래머의 부담(overhead)이 커진다. 그러므로 컴파일러 단계에서 그 특성을 분석하여 묵시적 병렬성(implicit parallelism)을 추출한 후, 태스크 병렬언어에서 제공하는 병렬구조로 변환하는 병렬화 및 최적화가 필요하다.

그러나 대표적인 태스크 병렬언어인 포트란 M과 Fx 포트란 컴파일러에서는 현재 이러한 기능까지는 아직 제공하지 못하고 있다[6, 11, 12]. 따라서 본 논문에서는 종속성으로 인해 순차 수행되어야 하는 루프구조를 컴파일러 단계에서 종속성에 따라 분석하여 묵시적으로 내포되어 있는 그 병렬수행 가능성을 추출하는 방안을 제시한다. 그래서 병렬수행이 가능한 경우에는 포트란 M의 PROCESSDO 루프와 PROCESSES 블록구조 등의 태스크 병렬구조로 변환할 수 있도록 하고, 불가능한 경우에는 가능한 일부분에 대해 병렬화되도록 최적화하는 방안을 제시한다. 그리고 PROCESSDO 루프와 PROCESSES 블록구조 모두로 병렬화 가능한 경우에 다양한 조건에 따라 어느 구조로 변환하는 것이 효과적인가를 분석하고, 일반 프로그래머에 의해 작성되는 프로그램과 비교한다.

본 논문에서는 컴파일러 단계에서 다양한 자료 종속성이 존재하여 순차 수행되어야 하는 순차 프로그램을 각 경우별로 가장 적절한 태스크 병렬 구조로 변환하고자 하므로, 다른 태스크 병렬언어들 - Fx 포트란[15, 16, 17], Linda[7], CODE[14], HeNCE (Heterogeneous Network Computing Environment)[4] - 에 비해 다양한 태스크 병렬 구조가 보다 잘 정의되어 있는 포트란 M을 대상 언어로 한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 다루고자 하는 경우들과 병렬화 과정에서 프로그램의 시맨틱을 유지하기 위해 필요한 자료 종속성에 대해 논하고, 병렬화 대상 언어인 포트란 M의 병렬 태스크 구조의 정의 및 그 수행 특성을 기술한다. 3장에서는 종속성으로 인해 프로그래머가 태스크 병렬구조로 프로그래밍을 할 수 없는 경우들을 분석한다. 그래서 각 경우에 대해 병렬화 가능성 여부를 판단하여 가능한 경우에는 태스크 병렬구조로 병렬화하고, 불가능한 경우에는 일부분에 대해 병렬화하는 최적화를 기술한다. 4장에서는 PROCESSDO 루프구조와 PROCESSES 블록구조 모두로 병렬화 가능한 경우에 대해 상황에 따라 어느 구조를 사용하는 것이 더 효과적인가를 분석한다. 또 프로그래머에 의한 일반적인 병렬화에 비해 본 논문에서 제안하는 컴파일러 단계에서의 자동 병렬화가 갖는 장점들을 비교하여 논한다. 마지막으로 5장에서 결론을 맺는다.

2. 연구배경

대부분의 과학 및 공학용 응용 프로그램들은 대부분 루프들로 구성되고, 이들 루프내에 많은 병렬성을 내포하고 있어 병렬화 및 최적화 연구의 대상이 되고 있다[12]. 그리고 현재의 태스크 병렬언어에서 별도의 태스크로 생성될 수 있는 기본 단위는 부프로그램 단위이다. 그러므로 본 논문에서 묵시적 병렬성을 추출하는 대상은 중첩 DO 루프구조내에서 여러 개의 부프로그램들이 호출되는 경우이다.

또한, 본 논문에서는 프로그래머가 쉽게 병렬화할 수 없는 경우에 대한 컴파일러 단계에서의 묵시적 병렬성 추출을 다루므로, 중첩 루프내에서 호출되는 부프로그램간에는 종속성이 존재한다. 즉, 연속적으로 호출되는 두 부프로그램의 매개변수 리스트상에 동일한 배열 변수가 존재하고, 두 부프로그램내에서 그 배열 변수가 새로이 재정의되어 사용된다. 이와같이 호출되는 부프로그램간에 자료 종속성 존재여부를 분석하기 위해서는 프로시쥬어간 분석(Interprocedural Analysis)이 필요하다[5]. 그리고 각 부프로그램을 표현 유지하는 중간표현 형태에서 각 기본블록과 부프로그램 블록상에 그 블록내에서 정의되고 사용되는 변수들의 리스트를 유지한다.

종속성이 존재하는 두 부프로그램이 병렬 태스크로 수행되기 위해서는, 제어 독립적인 두 부프로그램 사이에 존재하는 자료 종속성 관계가 두 부프로그램

를 병렬로 수행하더라도 수행의미(execution semantics)가 변하지 않아야 한다.

이하에서는 본 논문에서 묵시적 병렬성 추출을 위해 사용하는 자료 종속성 개념에 대해 간략히 알아본다. 그리고 본 논문의 병렬화 대상 언어인 포트란 M에서 제공하는 태스크 병렬구조로 PROCESSDO 루프구조와 PROCESSES 블록구조의 정의와 특징을 살펴본다.

2.1 자료 종속성

본 논문에서 순차 부프로그램들을 병렬 태스크로 변환하는 것은 순차 제어 흐름을 역전시키는 것이 아니라 동시수행 형태로 바꾸고자 하는 것이다. 그런데 반 종속성(anti-dependence)이나 출력 종속성(output-dependence)은 이같은 동시수행 형태로 변환하려는 시도에 영향을 주지 않고, 흐름 종속성(flow-dependence)만이 영향을 준다. 이렇게 수행의미에 영향을 주는 경우를 참 종속성(true dependence)이라 하고, 아무런 영향을 주지 않는 경우를 거짓 종속성(false dependence)이라 한다[1, 3, 18, 19]. 그러므로 본 논문에서는 태스크 병렬성 추출에서 의미있는 참 종속성인 흐름 종속성만을 고려한다.

그리고 루프 반복(iteration)간의 종속성 여부에 따라 LCD(loop carried dependence)와 LID(loop independent dependence)로 나누어진다[2, 3]. 서로 다른 루프 반복간에 종속성이 존재하면 LCD가, 동일 루프 반복내에 종속성이 존재하면 LID가 존재한다고 한다.

또 종속관계에 있는 두 문장의 위치 관계에 따라 분류하는 전향 종속성(forward data dependence: 이하 FDD와 혼용)과 역향 종속성(backward data dependence: 이하 BDD와 혼용)이 있다[3, 18]. 프로그램의 뒷문장의 내용이 앞문장의 결과에 종속되면 FDD가, 프로그램의 앞문장이 뒷문장의 내용에 종속되면 BDD가 존재한다고 한다.

2.2 태스크 병렬구조의 정의

태스크 병렬언어인 포트란 M에서 제공하는 태스크 병렬구조로는 PROCESSDO 루프와 PROCESSES 블록구조가 있다[9, 11]. 먼저 PROCESSES 블록의 문법 구조는 다음과 같다.

```
PROCESSES
    statement_1
    . . .
    statement_n
ENDPROCESSES
```

여기서 각 문장(statement)들은 부프로그램 호출문을 의미한다. 포트란 M에서는 이들을 프로세스라 한다. 여기서 각 문장들, 즉 프로세스들은 병렬로 수행된다. 다시 말해서 각 부프로그램들은 동시에 호출되어 서로 다른 프로세서들에 의해 병렬 수행된다.

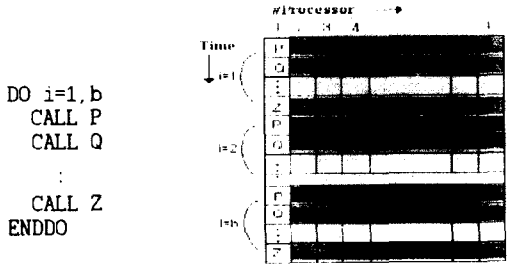
다음으로 PROCESSDO 루프의 문법구조는 다음과 같다.

```
PROCESSDO i = 1, n
    processcall myprocess
ENDPROCESSDO
```

PROCESSDO 루프는 같은 프로세스에 대해 여러 실체(instance)들을 생성한다. 즉, PROCESSDO 루프는 한 개의 프로세스를 여러 루프 인덱스에 대하여 병렬 호출하여 수행한다. 위의 구조에서 루프 인덱스 i가 1부터 n까지에 해당하는 프로세스 myprocess를 병렬로 호출하여 수행한다. PROCESSDO 루프구조에는 단 하나의 PROCESSDO 루프나 PROCESSES 호출문만 포함한다. 이' PROCESSDO 루프는 PROCESSDO 루프구조와 PROCESSES 블록구조내에 중첩될 수 있으나, PROCESSES 블록은 PROCESSDO 루프내에 중첩될 수 없다.

2.3 태스크 병렬구조의 특성

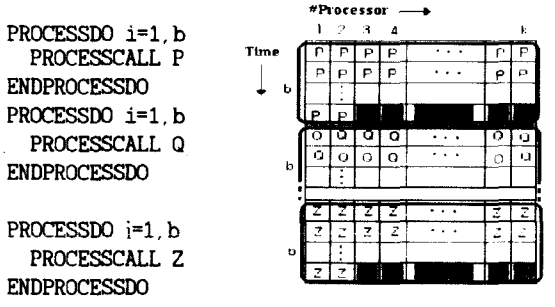
포트란 M의 두 태스크 병렬구조는 서로 상이한 특성을 가질 뿐만 아니라 일반적인 병렬 코드와도 구분되는 특징을 갖는다. 이 절에서는 두 병렬구조의 수행형태를 모형화하여 그 특성을 분석한다. 1차원 DO 루프구조는 (그림 1)과 같은 형태의 수행 구조를 갖는다. 여기서 b는 루프 반복횟수(loop bound)를 나타내고, k는 프로세서의 수를 나타낸다. 이때 여러 개의 프로세서가 사용 가능하더라도 각 부프로그램들은 한 프로세서에만 할당되어 순차적으로 수행된다. (이하의 수행형태에서 음영부분은 사용되지 않는 부분이다.)



(a) 순차 DO 루프 예제 (b) 수행 형태

(그림 1) 순차 DO 루프의 수행형태
(Fig. 1) Execution model of sequential DO loop

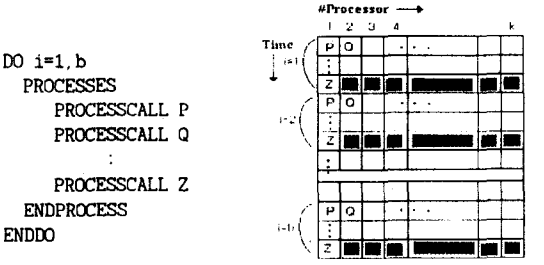
그러나 부프로그램들이 병렬수행 가능하다면, PROCESSDO 루프구조와 PROCESSES 블록구조 형태로 변환할 수 있다. 먼저 (그림 2)에서 볼 수 있듯이 PROCESSDO 루프구조는 순차 DO 루프의 루프 몸체내에 있던 부프로그램들이 프로세스 형태의 병렬 태스크로 바뀌어 하나씩 수행된다. 여기서 각 프로세스는 i=1부터 i=k까지의 모든 루프 인덱스에 대해 병렬수행 가능하다. 이때 가용 프로세서 수가 k라고 가정하면, (그림 2(b))의 수행형태처럼 가용 프로세서 수 k만큼 병렬로 수행된다. 이러한 태스크의 수행 형태는 문장에 대해 벡터화된 벡터코드의 수행형태와 유사하다.



(a) PROCESSDO 루프 예제 (b) 수행 형태

(그림 2) PROCESSDO 루프의 수행 형태
(Fig. 2) Execution model of PROCESSDO loop

본 논문에서는 병렬화 가능한 루프구조를 PROCESSES 블록구조로 표현하기 위해 DO 루프내에 내포된 PROCESSES 블록구조로 모형화하여 사용한다. (그림 3(a))에서 볼 수 있듯이 PROCESSES 블록구조에서는 PROCESSES 블록 몸체내에 있는 프로세스들이 병렬 수행될 수 있다. 이때 동시에 병렬수행 가능한 프로세스들은 각 루프 인덱스에 대해 가용 프로세서의 수 k만큼 병렬 수행한다. 따라서 DO 루프내의 PROCESSES 블록구조는 병렬화 코드인 DOALL 구조나 FORALL 구조와는 상이한 특성을 갖는다[18].



(a) PROCESSES 블록 예제 (b) 수행 형태

(그림 3) PROCESSES 블록구조의 수행 형태
(Fig. 3) Execution model of PROCESSES loop

정리해 보면, 순차 DO 루프구조를 병렬화하려는 관점에서 볼 때 PROCESSDO 루프구조는 하나의 프로세스를 모든 루프 인덱스에 대해 invoke하여 병렬 수행하는 형태이고, PROCESSES 블록구조는 루프 인덱스의 한 반복에 대해 여러 프로세스들을 병렬수행하는 형태이다.

한편 중첩 DO 루프구조를 병렬화하는 관점에서 분석해 볼 수 있다. 이때 종속성 검사를 하여 일괄적으로 병렬화 가능하면, PROCESSDO 루프구조는 중첩 DO 루프구조의 중첩된 n차원의 깊이만큼 PROCESSDO 루프구조를 중첩시킬 수 있다. 그러나 PROCESSES 블록구조는 그 중첩정도에 관계없이 항상 루프 인덱스에 대해서는 순차수행 의미를 가지며, 한 루프 인덱스의 각 반복내에서 여러 프로세스들을 병렬 수행하는 형태이다. 본 논문에서는 이와같은 특성을 고려하여 다양한 종속관계에 따른 변환 가능성을 분석한다.

3. 묵시적 태스크 병렬성의 추출

이 장에서는 프로그램의 종속성 관계를 분석하여

종속관계에 따라 병렬화 가능성을 분석하고, 가능한 경우에 대해 병렬화한 예제를 보인다. 2장에서 언급한 LCD와 LID 각각에 대해 FDD와 BDD가 존재하는 경우로 분류하여 컴파일 단계에서 병렬구조 형태로 변환할 수 있는 경우를 분석한다. (이하에서는 표현을 용이하게 하기 위해 순차 중첩 루프내에 존재하는 호출문은 부프로그램 호출문이라 하고, PROCESSDO 루프나 PROCESSES 블록구조와 같은 병렬 태스크 구조내의 호출문은 프로세스 호출문이라 한다.)

3.1 한 가지 종속성만 존재하는 경우

3.1.1 LID가 존재하는 경우

같은 반복내에서 종속성이 존재하는 LID의 경우는 동일한 반복내의 두 부프로그램 호출문을 병렬 수행하는 형태로는 변환할 수 없고, 동일한 반복내에서는 두 프로그램 호출문을 순차 수행하면서 루프 인덱스에 대해 반복 횟수만큼 병렬 수행할 수 있다. 그러므로 LID가 존재하는 경우 PROCESSES 블록구조로는 변환할 수 없으나, PROCESSDO 루프구조로는 변환할 수 있다. 이러한 성질을 명확히 설명하기 위해 [정리 1]과 [정리 2]를 제시한다. 먼저 PROCESSES 블록구조로 변환할 수 없음을 구체적으로 보이기 위해 [정리 1]을 유도한다.

[정리 1] 중첩 루프내에서 호출되는 부프로그램들 간에 LID가 존재하면 태스크 병렬구조인 PROCESSES 블록으로 변환할 수 없다.

[증명] 병렬 수행단위인 PROCESSES 블록구조는 PROCESSES 블록문 진입부(entry)로 제어가 들어와 구조내 프로세스들이 동시에 병렬로 수행되고, ENDPROCESSES문에서 동시에 종결된다. 그런데 순차 수행시 두 부프로그램간에 LID가 존재하는 경우는 한 루프 인덱스 값에 대해 동일 반복내에서 자료 종속성이 존재하는 경우이다. 즉, 한 반복내에서 선행 부프로그램의 수행결과가 후속 부프로그램의 수행에 영향을 미치는 경우이므로, 이 부프로그램들은 순차 수행되어야만 수행의미가 변하지 않는다. 그러므로 동일 반복내에서 프로세스들을 동시 수행하는 PROCESSES 블록구조로는 변환할 수 없다. □

이하에서 부프로그램 P_1에서 $A(i_1, i_2)$가 정의(define)되고, P_2에서 $A(i_1, i_2)$가 사용(use)될 때, 두 부프로그램 호출문의 매개변수 앞에 각각 'D:'와 'U:'를 붙여 표현하기로 한다. 예를들어루프 인덱스 $i_1=k$이고 $i_2=l$인 루프 반복에서, 부프로그램 P_1에서 $A(i_1, i_2)$가 정의되고, P_2에서 $A(i_1, i_2)$가 사용되는 경우는 (그림 4)의 호출문 형태로 표현한다.

```
DO i1=1,M
   DO i2=1,N
      CALL P1(D:A(i1,i2),...)
      CALL P2(U:A(i1,i2),...)
   ENDDO
ENDDO
```

(그림 4) 예제 프로그램
(Fig. 4) Example program

(그림 4)에서는 부프로그램 P_1에서 P_2로 LID가 존재한다. 루프 인덱스 $i_1=k$이고 $i_2=l$인 같은 반복내에서 부프로그램 P_1과 P_2를 병렬 수행하면, 부프로그램 P_2에서의 $A(i_1, i_2)$는 P_1에서 계산된 $A(i_1, i_2)$ 값을 사용하지 못한다. 따라서 부프로그램 P_1과 P_2는 이 종속성을 유지하면서 PROCESSES 블록으로 변환될 수 없다.

그러나 LID가 존재하는 경우 PROCESSDO 루프구조를 사용하는 병렬 태스크 형태로는 변환 가능하다. 이 성질을 [정리 2]에서 증명한다.

[정리 2] 중첩 루프내에서 호출되는 부프로그램들 간에 LID가 존재하면, 부프로그램 호출문 각각을 PROCESSDO 루프구조를 사용하여 중첩된 병렬 태스크 구조로 변환할 수 있다.

[증명] 순차 DO 루프구조에서 호출되는 두 부프로그램간에 LID가 존재하는 경우는 같은 루프 인덱스 값에 대해 선행 부프로그램의 수행결과가 후속 부프로그램의 수행에 영향을 미치는 경우이다. 이 순차 DO 루프구조내의 두 부프로그램 호출문 각각을 병렬 수행단위인 PROCESSDO 루프구조로 변환했을 때, 한 PROCESSDO 루프구조에서 그 루프 진입부로 제어가 들어와 모든 루프 인덱스들에 대해 동시에 병렬 수행되고 ENDPROCESSDO 문을 만나 수행이 동시에 종결된 후 다음 PROCESSDO 루프구조가 같은 방식으로 수행된다. 순차 DO 루프구조에서 두 부프로그램간의 LID가 독립된 두 PROCESSDO 루프구

조에서도 동일하게 유지되므로, 동일 반복내에서 두 프로세스를 각각 수행하는 PROCESSDO 루프구조로 변환할 수 있다. □

```
PROCESSDO i1=1,M
  PROCESSDO i2=1,N
    PROCESSCALL P1(A(i1,i2),...)
  ENDPROCESSDO
ENDPROCESSDO
PROCESSDO i1=1,M
  PROCESSDO i2=1,N
    PROCESSCALL P2(A(i1,i2),...)
  ENDPROCESSDO
ENDPROCESSDO
```

(그림 5) 예제 프로그램
(Fig. 5) Example program

(그림 4)의 예제 프로그램에서 LID를 갖는 경우인 부프로그램 P_1과 P_2는 위의 [정리 2]에 의해 중첩 PROCESSDO 루프구조로 변환하여 병렬 수행하여도 수행의미가 변하지 않는다. 따라서 컴파일러에 의해 (그림 5)와 같이 중첩 PROCESSDO 루프구조로 병렬화된다.

이상에서 LID는 FDD만을 고려하였다. 그 이유는 루프 인덱스의 표현상 BDD의 LID는 참 종속성이 아니기 때문이다. 앞에서 언급한 바와 같이 본 논문에서는 거짓 종속성은 무시한다.

3.1.2 LCD가 존재하는 경우

LCD는 다른 반복구간 사이에 종속성이 존재하는 경우이다. LCD가 존재하는 서로 다른 반복들은 동시 수행할 수 없고, 동일 반복내의 여러 부프로그램들은 동시수행 가능하다. 이러한 LCD가 존재하는 경우, PROCESSES 블록구조로의 병렬화 가능성과 PROCESSDO 루프구조로의 병렬화 가능성을 다음과 같이 구분하여 살펴볼 수 있다. 이하에서 LCD이면서 FDD의 의미를 나타내는 종속성은 LCD(FDD)로 표현하고, BDD의 의미를 나타내는 종속성은 LCD(BDD)로 표현한다.

①PROCESSES 블록구조로의 변환

부프로그램들 간에 LCD가 존재하는 경우에는 이 LCD가 FDD 또는 BDD 여부에 관계없이 PROCESSES 블록구조로 변환이 가능하다. 이 특성을 구체적으로 보이기 위해 [정리 3]을 유도한다.

[정리 3] 중첩 루프내에 존재하는 부프로그램들간에 LCD가 존재하면, LCD(FDD) 또는 LCD(BDD)에 관계없이 PROCESSES 블록형태의 병렬 태스크로 변환할 수 있다.

[증명] 순차 수행 형태인 부프로그램들의 LCD는 LCD(FDD)이거나 LCD(BDD)이다. 이때 같은 반복내에서는 종속성이 존재하지 않고 다른 반복간에만 종속성이 존재하므로, 같은 반복내의 부프로그램들만을 병렬 수행하도록 하는 구조인 PROCESSES 블록으로 묶어 병렬화하여도 종속성 관계나 부프로그램들의 수행의미는 변하지 않는다. 따라서 PROCESSES 블록구조로 병렬화할 수 있다. □

(그림 6)의 (a)에서 루프 인덱스 $i_1=k$이고 $i_2=l$인 루프 반복내의 부프로그램 P_1에서 배열변수 A가 정의되고, 루프 인덱스 $i_1=k$이고 $i_2=l+1$인 루프 반복내의 P_2에서 A가 사용되므로, 부프로그램 P_1에서 P_2로 LCD가 존재한다. 같은 루프 반복내에서 부프로그램 P_1과 P_2를 병렬 수행하여도 다른 반복간의 순서는 바깥 루프에 의해 그대로 유지되므로, P_2에서의 A는 P_1에서 계산된 A 값을 사용하게 된다. 따라서 부프로그램 P_1과 P_2는 (그림 6)의 (b)와 같이 PROCESSES 블록으로 병렬화된다.

```
DO i1=1,M
   DO i2=1,N
      CALL P1(D:A(i1,i2),...)
      CALL P2(U:A(i1,i2+1),...)
   ENDDO
ENDDO
```

(a) 예제 프로그램

```
DO i1=1,M
   DO i2=1,N
      PROCESSES
         PROCESSCALL P1(A(i1,i2),...)
         PROCESSCALL P2(A(i1,i2+1),...)
      ENDPROCESSES
   ENDDO
ENDDO
```

(b) PROCESSES 블록구조로 변환한 형태

(그림 6) LCD(FDD)가 존재하는 예
(Fig. 6) An example which exists LCD(FDD)

(그림 7)의 (a)에서는 루프 인덱스 $i_1=k$이고 $i_2=l$인

루프 반복내의 부프로그램 P_1에서 A가 사용되고, 루프 인덱스 $i_1=k$이고 $i_2=l+1$인 루프 반복내의 P_2에서 A가 정의되는 경우로 부프로그램 P_2에서 P_1으로 LCD가 존재한다. LCD(FDD)가 존재하는 경우와 마찬가지로, 같은 루프 반복내에서 부프로그램 P_1과 P_2를 병렬수행하여도 다른 반복간의 순서는 바깥 루프에 의해 그대로 유지되므로 선행 P_1에서 사용된 A는 후속 P_2에서 정의될 수 있다. 따라서 [정리 3]에 의거하여 부프로그램 P_1과 P_2는 (그림 7)의 (b)와 같이 PROCESSES 블록으로 병렬화된다.

```
DO i1=1,M
   DO i2=1,N
      CALL P1(U:A(i1,i2-1),...)
      CALL P2(D:A(i1,i2),...)
   ENDDO
ENDDO
```

(a) 예제 프로그램

```
DO i1=1,M
   DO i2=1,N
      PROCESSES
        PROCESSCALL P1(A(i1,i2-1),...)
        PROCESSCALL P2(A(i1,i2),...)
      ENDPROCESSES
   ENDDO
ENDDO
```

(b) PROCESSES 블록구조로 변환한 형태

(그림 7) LCD(BDD)가 존재하는 예
(Fig. 7) An example which exists LCD(BDD)

②PROCESSDO 루프구조로의 변환

PROCESSES 블록구조의 경우와는 달리 PROCESSDO 루프구조의 경우에는 FDD를 갖는 경우와 BDD를 갖는 경우가 상이한 특징을 갖는다. 먼저 FDD를 갖는 경우에는 LID를 갖는 경우와 마찬가지로 모든 경우에 대해 PROCESSDO 루프구조로 변환할 수 있다. 이 성질을 구체적으로 설명하기 위해 [정리 4]를 유도한다.

[정리 4] 중첩 루프내에 존재하는 부프로그램들 간에 LCD(FDD)가 존재하면, PROCESSDO 루프구조의 병렬 태스크로 변환할 수 있다.

[증명] LCD(FDD)가 존재하는 경우는 다른 반복구간에서 선행 부프로그램으로부터 후속 부프로그램으로 종속성이 존재하는 경우이다. 따라서 선행 부프로그램을 모든 루프 인덱스에 대해 병렬수행하고, 그 후에 후속 부프로그램을 모든 루프 인덱스에 대해 병렬수행하는 형태로 변환하여도 전체 수행의미는 변하지 않는다. 그러므로 별개의 PROCESSDO 루프구조로 병렬화할 수 있다. □

[정리 4]에 의거하여 (그림 6)의 예제 프로그램을 (그림 8)과 같이 PROCESSDO 루프구조로 병렬화된다.

```
PROCESSDO i1=1,M
  PROCESSDO i2=1,N
    PROCESSCALL P1(A(i1,i2),...)
  ENDPROCESSDO
ENDPROCESSDO
PROCESSDO i1=1,M
  PROCESSDO i2=1,N
    PROCESSCALL P2(A(i1,i2+1),...)
  ENDPROCESSDO
ENDPROCESSDO
```

(그림 8) 그림 6을 PROCESSDO 루프구조로 변환한 형태
(Fig. 8) Transformed PROCESSDO loop of Fig. 6

이렇게 LCD(FDD)가 존재하는 경우는 [정리 3]과 [정리 4]에서 볼 수 있듯이 PROCESSES 블록구조와 PROCESSDO 루프구조로 병렬화가 가능하다. 이때 조건에 따라 어느 구조가 더 효과적인가에 대한 분석은 4장에서 기술한다.

다른 반복간의 종속관계인 LCD(BDD)가 존재하는 경우에는 [정리 4]에서 기술하였듯이 PROCESSES 블록구조로의 병렬화는 가능하지만, PROCESSDO 루프구조로의 병렬화는 불가능하다. 이 특징을 보이기 위해 다음 [정리 5]를 유도한다.

[정리 5] 중첩 루프내에 존재하는 부프로그램들 간에 LCD(BDD)가 존재하면, PROCESSDO 루프구조로 변환할 수 없다.

[증명] LCD(BDD)가 존재하는 이 경우는 서로 다른 반복간에 후속 부프로그램으로부터 선행 부프로그램으로 종속성이 존재하는 경우이다. 따라서 선행 부프로그램을 모든 루프 인덱스에 대해 먼저 병렬수행하고, 다음에 후속 부프로그램을 모든 루프 인덱스에 대해 병렬 수행하는 PROCESSDO 루프구조로 변환

하면 수행의미가 변하게 된다. 따라서 PROCESSDO 루프구조로는 변환할 수 없다. □

앞에서 살펴본 (그림 7)의 (a)를 PROCESSDO 루프구조로 병렬화하면 그 수행의미가 변하기 때문에 PROCESSDO 루프구조로 병렬화할 수 없다. 그러나 이 경우에 대해 보다 상세한 종속성 특성을 분석하여 병렬화가능한 경우를 분류하거나 루프 교환(loop interchange)을 적용하여 병렬화 가능성을 높이거나 부분적인 병렬화를 수행할 수 있다. 이 방안은 3. 3절의 최적화에서 살펴본다.

한 가지 종속성만 존재하는 경우의 분석 결과를 종합해 보면 종속성의 종류에 따라 가능한 병렬화 형태를 〈표 1〉과 같이 정리할 수 있다. 〈표 1〉에서 LID(FDD)의 경우에는 PROCESSDO 루프구조로의 병렬화는 가능하나 PROCESSES 블록구조로의 병렬화는 할 수 없고, LID(FDD)가 존재하는 경우에는 PROCESSDO 루프구조 뿐만 아니라 PROCESSES 블록구조로도 병렬화가 가능하다. 그리고 LCD(BDD)가 존재하는 경우에는 PROCESSDO 루프구조로는 병렬화할 수 없으나 PROCESSES 블록구조로는 병렬화가 가능하다.

〈표 1〉 병렬화 가능형태 1
〈Table 1〉 Parallelizable form 1

가능 병렬화 형태 / 종속성 종류	PROCESSDO 루프구조	PROCESSES 블록구조
①LID(FDD)	○	×
②LCD(FDD)	○	○
③LCD(BDD)	×	○

3.2 여러 종속성들이 존재하는 경우

앞에서 분석 및 유도된 〈표 1〉의 성질을 기반으로 중첩 루프 블록내에 여러 종속성들이 존재하는 경우에 대한 변환 가능성을 이끌어 낼 수 있다. 예를들어, (그림 9)의 (a)와 같이 LCD(FDD)와 LCD(BDD)가 동시에 존재하는 경우, 두 종속성 모두 PROCESSES 블록구조로 변환이 가능하다. 그 이유는 [정리 3]에 의거하여 LCD(FDD)나 LCD(BDD)에 관계없이 PROCESSES 블록구조로 병렬화 가능하기 때문이다.

그리고 LCD(FDD)가 존재하는 경우에는 [정리 4]에 의해 PROCESSDO 루프구조로의 변환이 가능하나, LCD(BDD)의 경우에는 [정리 5]에 의해 PROCESSDO 루프구조로 변환할 수 없다. 그러므로 (그림 9)의 (b)와 같이 PROCESSES 블록구조로의 병렬화만 가능하다. 이와 같은 분석을 통하여 여러가지 종속성이 존재하는 경우의 병렬화 가능성은 〈표 1〉에서 살펴 본 종속성별 병렬화 가능형태에 대한 교집합(intersection set) 관계를 가짐을 알 수 있다.

```
DO i1=1,M
   DO i2=1,N
      CALL P1(D:A(i1,i2),U:B(i1,i2-1),...)
      CALL P2(U:A(i1,i2-1),D:B(i1,i2),...)
   ENDDO
ENDDO
```

(a) 예제 프로그램

```
DO i1=1,M
   DO i2=1,N
      PROCESSES
        PROCESSCALL P1(A(i1,i2),B(i1,i2-1),...)
        PROCESSCALL P2(A(i1,i2-1),B(i1,i2),...)
      ENDPROCESSES
   ENDDO
ENDDO
```

(b) PROCESSES 블록구조로 변환한 형태

(그림 9) LCD(FDD)와 LCD(BDD)가 존재하는 경우 예
(Fig. 9) An example which exists LCD(FDD) and LCD(BDD)

블록내에 서로 다른 종속성들이 존재하는 경우, 가능한 경우의 수는 기본적인 상태가 LID(FDD), LCD(FDD), LCD(BDD) 세 가지이므로 $2^3-1=7$가지이다. 따라서 서로 다른 종속성이 존재하는 경우에 대해 〈표 2〉와 같이 PROCESSDO 루프나 PROCESSES 블록구조로 변환 가능한 경우를 분석할 수 있다. 〈표 2〉의 ⑤는 LCD(BDD)와 LID(FDD)가 존재하는 경우로 PROCESSDO 루프나 PROCESSES 블록구조 어느 것으로도 변환할 수 없음을 보이고 있다.

이상의 분석결과로 부터 두 가지 이상의 서로 다른 종속성이 존재하는 경우, LCD(BDD)의 존재는 PROCESSDO 루프구조로의 변환을 불가능하게 하고, LID(FDD)의 존재는 PROCESSES 블록구조로의 변환을 불가능하게 함을 알 수 있다. 따라서 두 부프로그램간에 LCD(BDD)가 존재하지 않는 경우는 PRO-

〈표 2〉 병렬화 가능형태 2
〈Table 2〉 Parallelizable form 2

가능 병렬화 형태 / 종속성 종류	PROCESSDO 루프구조	PROCESSES 블록구조
④LID(FDD)+LCD(FDD)	○∩○=○	×∩○=×
⑤LID(FDD)+LCD(BDD)	○∩×=×	×∩○=×
⑥LCD(FDD)+LCD(BDD)	○∩×=×	○∩○=○
⑦LID(FDD)+LCD(FDD)+LCD(BDD)	○∩○∩×=×	×∩○∩○=×

CESSDO 루프구조로 병렬화가 가능하고, LID(FDD)가 존재하지 않는 경우는 PROCESSES 블록구조로 병렬화가 가능하다. 그리고 LCD(BDD)가 존재하여 병렬화 변환이 불가능하게 되는 경우들에 대해서는 최적화를 적용하여 일부분에 대해서라도 병렬화할 수 있는데 이 방안에 대해서는 다음 3.3절에서 논한다.

3.3 최적화

이 절에서는 LCD(BDD)가 존재하는 경우 최외곽 부등호 항목(outmost non-equal entry: 이하 ONEE와 혼용) 개념을 적용하여 태스크 병렬구조로 일부 변환하는 방안과 루프 교환을 통하여 태스크 병렬구조로 일부 변환하는 방안을 제시한다.

3.3.1 최외곽 부등호 항목의 이용

앞 절에서는 종속성 종류별 특성을 분석하여 태스크 병렬구조로의 변환 가능성을 기술하였다. 이때 태스크 병렬구조로 부분적으로 병렬화하는데 사용하기 위해 [20]에서 정의한 ONEE 개념을 수정한다. 본 논문에 적용하기 위해 수정된 ONEE의 정의는 다음과 같다.

〈정의 1〉 중첩 루프하의 두 부프로그램간에 종속성이 존재할 때, 종속관계에 있는 선행 부프로그램 변수와 후속 부프로그램 변수의 대응하는 인덱값의 차이를 계산할 수 있다. 그래서 선행 부프로그램 변수의 인덱스 값이 후속 부프로그램 변수의 인덱스 값보다 크면 '〉', 작으면 '〈', 같으면 '='로 표시한다. 이때 루프 인덱스 값의 차이가 '〉' 또는 '〈'인 가장 바깥 루프 인덱스 항목을 ONEE라 정의한다. □

```
DO i1=1,M
   DO i2=1,N
      DO i3=1,R
            :
         CALL P1(U:A(i1,i2-1,i3,...),...)
         CALL P2(D:A(i1,i2,i3,...),...)
            :
      ENDDO
   ENDDO
ENDDO
```

(그림 10) ONEE 예
(Fig. 10) An example of ONEE

예를들어 (그림 10)에서는 배열 변수 A(i_1, i_2, i_3, ...)에 대해 부프로그램 P_2에서 P_1으로 LCD(BDD)가 존재하는데, 루프 인덱스 항목 값들의 차이는 i_1-i_1 = '=', $(i_2-1)-i_2$='〈', i_3-i_3='=' 등이 된다. 두번째 루프 인덱스인 i_2에 대한 차이 값이 부등호 형태인 '〈'이므로, 두번째 인덱스 항목 i_2가 ONEE가 된다.

다음 (그림 11)은 중첩 루프구조에서 LID(FDD)와 LCD(BDD)를 갖는 경우의 한 예이다. [정리 5]에서는 이 경우에 LCD(BDD)로 인하여 병렬화될 수 없음을 보이고 있다. 논문 [20]의 결과에 의하면 배열 변수들간의 종속성은 ONEE인 루프 인덱스에 의해 결정된다. 즉 ONEE 값이 '〉'이면 FDD가 존재하고, '〈'이면 BDD가 존재한다. (그림 11)의 예에서 배열변수 B(i_1-1, i_2)와 B(i_1, i_2)사이의 ONEE인 루프 인덱스 i_1이 BDD를 결정함을 알 수 있다.

```
DO i1=1,M
   DO i2=1,N
      CALL P1(D:A(i1,i2),U:B(i1-1,i2),...)
      CALL P2(U:A(i1,i2),D:B(i1,i2),...)
   ENDDO
ENDDO
```

(a) 예제 프로그램

```
                    ========================
i1=1, i2=1          P1(...A(1,1) = B(0,1)...)
                    P2(...B(1,1) = A(1,1)...)
                    ------------------------
i1=1, i2=2          P1(...A(1,2) = B(0,2)...)
                       :
                    ========================
i1=2, i2=1          P1(...A(2,1) = B(1,1)...)
```

(b) 펼친 경우

(그림 11) LID(FDD)와 LCD(BDD)가 존재하는 경우 예
(Fig. 11) An example which exists LCD(FDD) and LCD(BDD)

이러한 성질을 PROCESSDO 루프구조로의 병렬화에 이용하기 위하여 [정리 6]을 유도한다.

[정리 6] 중첩 루프내에 LCD(BDD)가 존재하는 두 부프로그램간에 배열변수의 최외곽 부등호 항목 아래단계(innerlevel 또는 lower level)의 루프 인덱스에 대해서는 PROCESSDO 루프구조로 변환할 수 있다.
[증명] 두 부프로그램간의 FDD나 BDD 여부는 ONEE 값에 의해 결정된다. 따라서 루프 인덱스 i가 ONEE라고 가정할 때, i 아래에 존재하는 루프 인덱스 값은 FDD나 BDD 여부에 영향을 주지 않는다. 즉, i 아래에 존재하는 루프 인덱스의 반복순서가 변하여도 프로그램 수행의미는 변하지 않는다. 따라서 i 아래의 루프 인덱스에 대해서는 PROCESSDO 루프구조로 변환할 수 있다. □

(그림 11)의 예에서 i_1 인덱스에 의해 이미 종속방향이 결정되어 있으므로, i_1보다 아래단계의 루프 인덱스인 i_2는 전체적인 종속방향에 영향을 주지 않음을 알 수 있다. 즉, i_2 인덱스에 대해서는 PROCESSDO 루프구조로 병렬화하여도 수행의미가 변하지 않는다.

```
DO i1=1,M
   PROCESSDO i2=1,N
     PROCESSCALL P1(A(i1,i2),B(i1-1,i2),...)
   ENDPROCESSDO
   PROCESSDO i2=1,N
     PROCESSCALL P2(A(i1,i2),B(i1,i2),...)
   ENDPROCESSDO
ENDDO
```

(그림 12) 그림 11을 PROCESSDO 루프구조로 변환한 형태
(Fig 12) Transformed PROCESSDO loop of Fig. 11

따라서 i_1에 대해서는 PROCESSDO 루프구조로 변환할 수 없지만, i_2에 대해 PROCESSDO 루프구조로 변환하여 (그림 12)와 같이 부분적으로 병렬화된다.

3.3.2 루프 교환

(그림 11)과는 달리 ONEE 아래단계에는 더 이상의 루프 인덱스가 존재하지 않고, ONEE보다 윗단계 루프 인덱스 차이 값이 등호(=)인 루프 인덱스가 존재하는 경우가 있다. 이러한 경우의 예를 (그림 13)에서 볼 수 있다.

이 경우도 (그림 11)처럼 (그림 13(a))의 배열변수 B들간에 LCD(BDD)때문에 PROCESSDO 루프구조로 병렬화할 수 없는 경우이다. 그런데 〈정의 1〉에 의해 ONEE가 i_2이므로 윗단계에 있는 루프 인덱스 i_1는 전체 종속방향에 영향을 미치지 않는다. 따라서 루프 인덱스 i_1과 i_2를 갖는 루프문을 교환하여 (그림 13(b))처럼 변환하여도 전체 수행의미는 변하지 않는다. (그림 13(b))는 ONEE 값이 I_2이므로, [정리 6]에 의해 i_2 아래 단계의 루프 인덱스인 i_1에 대해서는 PROCESSDO 루프구조로 변환할 수 있다. 따라서 (그림 13(a))의 프로그램에 대해 루프교환을 수행하여 (그림 13(b))처럼 PROCESSDO 루프구조로 부분 병렬화된다.

```
DO i1=1,M
   DO i2=1,N
     CALL P1(D:A(i1,i2),U:B(i1,i2-1),...)
     CALL P2(U:A(i1,i2),D:B(i1,i2),...)
   ENDDO
ENDDO
```

(a) 예제 프로그램

```
DO i2=1,N
   PROCESSDO i1=1,M
     PROCESSCALL P1(A(i1,i2),B(i1,i2-1),...)
   ENDPROCESSDO
   PROCESSDO i1=1,M
     PROCESSCALL P2(A(i1,i2),B(i1,i2),...)
   ENDPROCESSDO
ENDDO
```

(b) PROCESSES 블록구조로 변환한 형태

(그림 13) LID(FDD)와 LCD(BDD)를 갖는 경우 예
(Fig. 13) An example of LID(FDD) and LCD(BDD)

4. 분석 및 비교

묵시적 프로그램 병렬화를 수행하는 과정에서 PROCESSDO 루프와 PROCESSES 블록구조 두가지 모두로 병렬화 가능한 경우, 최적화된 코드를 생성하기 위해 어느 것을 선택하는 것이 더 효과적인 지를 판별할 수 있는 기준이 필요하다. 이 장에서는 이러한 경우에 대해 두 경우의 수행 형태를 모형화하여 각각의 수행 비용을 비교함으로서 이 판별 기준을 분석한다.

또한 이 장에서는 일반 프로그래머에 의해 작성되는 프로그래밍 방식과 본 논문에서 제안하는 묵시적

추출방안을 적용하여 컴파일러 단계에서 자동 병렬화하는 경우를 비교한다.

4.1 PROCESSDO 루프구조와 PROCESSES 블록구조 비교

4.1.1 수행환경 모델

본 논문에서 대상으로하는 병렬 시스템 환경하에서의 수행비용 계산에 영향을 미치는 요소는 다음 4가지이다.

- 사용가능한 프로세서 수(p#)
- 호출되는 부프로그램 수(k)
- 호출되는 부프로그램의 수행 사이클 수
  - P_i : i번째 호출되는 부프로그램의 수행 사이클 수
  - P_{max} : 호출되는 부프로그램들의 수행 사이클 수 가운데 가장 긴 수행 사이클 수
  - P_{avg} : 호출되는 부프로그램들의 평균 수행 사이클 수
- 루프 반복횟수 (b)

그리고 루프구조의 수행비용은 루프 제어변수와 루프 반복횟수를 초기화하기 위한 루프 진입(loop entry) 비용, 루프 몸체(loop body)를 수행하는데 드는 비용, 매 반복시마다 현재의 루프 반복횟수와 전체 루프 반복 수를 비교하여 루프수행을 종료시키기 위한 루프 이탈(loop exit) 비용으로 구성된다. 여기서 루프 관련 수행비용을 다음과 같이 가정한다[13].

- E_n : 루프 진입비용 (1 사이클로 가정-MIPS 코드 기준)
- E_x : 루프 이탈비용 (5 사이클로 가정-MIPS 코드 기준)

이와같은 가정하에 PROCESSDO 루프구조와 PROCESSES 블록구조를 다음과 같이 모형화할 수 있다.

① PROCESSDO 루프

PROCESSDO 루프는 루프 몸체내 여러 문장 중 한 문장(P_i)에 대해 여러 반복들을 현재 가용한 프로세서(p#)에 동시에 할당하여 병렬수행한다. 그러므로 루프 몸체내의 문장은 $\lceil b/p\# \rceil$ 만큼의 루프 반복으로 한 문장을 실행할 수 있다. 따라서 한 반복에서 한 문장의 수행비용은 다음과 같다.

$$E_n + P_i \times \lceil b/p\# \rceil + E_x \times \lceil b/p\# \rceil$$

PROCESSDO 루프구조는 루프 몸체내 문장 각각에 대해 각각 위의 비용이 소비되므로, 총 수행비용은 다음과 같다.

$$\begin{aligned}
&E_n + P_1 \times \lceil b/p\# \rceil + E_x \times \lceil b/p\# \rceil - \text{첫번째 문장} \\
&+ E_n + P_2 \times \lceil b/p\# \rceil + E_x \times \lceil b/p\# \rceil - \text{두번째 문장} \\
&\quad \vdots \\
&\quad \vdots \\
&+ E_n + P_k \times \lceil b/p\# \rceil + E_x \times \lceil b/p\# \rceil - \text{k번째 문장} \\
&= En \times k + \lceil b/p\# \rceil \times (\Sigma_{i=1}^{k} P_i) + \lceil b/p\# \rceil \times E_x \times k
\end{aligned}$$

② PROCESSES 블록

PROCESSES 블록구조는 한 루프 반복에 대해 루프내 여러 문장들이 가용한 프로세서에 1대1로 할당되어 병렬로 실행된다. 따라서 루프구조내 문장수(k)와 가용 프로세서 수(p#)가 일치하는 경우에는 루프 반복수(b) 만큼 반복 수행된다. 그러나 대개 일치하지 않는 경우가 많으므로 $\lceil k/p\# \rceil \times b$ 만큼 반복 수행된다.

PROCESSES 블록구조에서는 가용한 프로세서에 k개의 문장이 각각 할당되어 병렬로 수행되기 때문에, 수행 시간이 짧은 문장들은 긴 문장에 대해 수행시간의 중첩(overlap) 효과가 발생한다. 그러므로 한 반복의 수행 시간은 루프구조내 문장들(부프로그램들)중 가장 긴 수행 시간을 갖는 문장 P_{max}에 의해 결정된다. 그러므로 PROCESSES 블록구조에서의 수행비용은 다음과 같다.

$$\begin{aligned}
&E_n + (P_{max} + E_x) \times \lceil k/p\# \rceil \times b \\
&= E_n + P_{max} \times \lceil k/p\# \rceil \times b + E_x \times \lceil k/p\# \rceil \times b
\end{aligned}$$

4.1.2 비교 및 분석

일반적으로 현실성있는 프로그램의 경우, 호출되는 부프로그램의 수(k)보다 루프 반복 횟수(b)가 큰 경우(k<b)가 대부분이다. 따라서 본 논문에서는 이 경우를 〈표 3〉과 같이 분류하여 각 경우에 대해 PROCESSDO 루프와 PROCESSES 블록의 수행비용을 비교한다.

'k<b'인 경우를 가용한 프로세서 수(p#), 호출되는 부프로그램의 수, 루프 반복횟수의 관계에 따라, 〈표

3)에서와 같이 CASE 1과 CASE 2 등의 두 가지로 나눌 수 있다. 여기서 'b<p#'인 경우를 고려하지 않는 이유는 일반적으로 고성능 병렬언어 환경에서 프로그램의 루프 반복횟수는 100 단위 이상의 크기임에 비해, 프로세서의 수는 100 단위 이하인 경우를 고려하고 있기 때문이다.

다음으로 두 경우를 배수관계 여부에 따라 각각 네 가지 경우와 다섯 가지 경우로 다시 세분하여 수행비용을 분석하였다. 〈표 3〉의 분류 및 분석 결과에 대한 증명은 [부록]에서 볼 수 있다. 〈표 3〉에서 보는 바와 같이, CASE 1의 ②와 ④, CASE 2의 ③과 ④의 경우는 항상 PROCESSDO 루프의 수행 비용이 PROCESSES 블록의 수행 비용보다 적었다. 그러므로 이 경우들에 대해서는 PROCESSDO 루프로 변환하는 것이 더 효과적임을 알 수 있다.

그리고 CASE 1의 ①과 ③, CASE 2의 ①, ②, ⑤의 경우들은 호출되는 부프로그램들의 최대 수행 사이클 수와 평균 수행 사이클 수간의 차이($P_{max}-P_{avg}$)와 가용한 프로세서 수, 루프 반복횟수, 호출되는 부프로그램 수, 호출되는 부프로그램들의 총 수행 사이클 수 등의 관계에 따라 PROCESSDO 루프의 수행 비용이 PROCESSES 블록의 수행 비용보다 크거나 작거나 같았다.

따라서 이 경우들에 대해서는 각 조건에 따라 적은 수행 비용을 갖는 구조로 변환하는 것이 효과적임을 알 수 있다.

〈표 3〉 PROCESSDO 루프(PDO)와 PROCESSES 블록(PSS)의 수행비용 비교
〈Table 3〉 Comparison of execution costs between PROCESSDO and PROCESSES

경 우	조 건 (k, b, p#, P_{max}, P_{avg}, P_i)			수행 비용
	대 분 류	중 분 류	소 분 류	
CASE 1	$p\# \le k < b$	① $b = n \times p\#$, $k \neq m \times p\#$	$(P_{max}-P_{avg}) > p\#(k-1)/bk$	PDO < PSS
			$(P_{max}-P_{avg}) < p\#(k-1)/bk$	PDO > PSS
			$(P_{max}-P_{avg}) = p\#(k-1)/bk$	PDO = PSS
		② $b = n \times p\#$, $k \neq m \times p\#$		PDO < PSS
		③ $k = m \times p\#$, $b \neq n \times p\#$	$P_{max}-P_{avg} > p\#((\Sigma^{k}_{i=1}P_i)+6k-1)/bk$	PDO < PSS
			$P_{max}-P_{avg} < p\#((\Sigma^{k}_{i=1}P_i)+6k-1)/bk$	PDO > PSS
			$P_{max}-P_{avg} = p\#((\Sigma^{k}_{i=1}P_i)+6k-1)/bk$	PDO = PSS
		④ $b \neq n \times p\#$, $k \neq m \times p\#$		PDO < PSS
CASE 2	$k \le p\# \le b$	① $k = p\#$, $b = n \times p\#$	$(P_{max}-P_{avg}) > (k-1)/b$	PDO < PSS
			$(P_{max}-P_{avg}) < (k-1)/b$	PDO > PSS
			$(P_{max}-P_{avg}) = (k-1)/b$	PDO = PSS
		② $k = p\#$, $b = n \times p\#$	$(P_{max}-P_{avg}) > (\Sigma^{k}_{i=1}P_i+6k-1)/b$	PDO < PSS
			$(P_{max}-P_{avg}) < (\Sigma^{k}_{i=1}P_i+6k-1)/b$	PDO > PSS
			$(P_{max}-P_{avg}) = (\Sigma^{k}_{i=1}P_i+6k-1)/b$	PDO = PSS
		③ $k < p\# = b$		PDO < PSS
		④ $k < p\# < b$, $b = n \times p\#$		PDO < PSS
		⑤ $k < p\# < b$, $b \neq n \times p\#$	$\Sigma^{k}_{i=1}P_i > 5b-6k+1$	PDO < PSS
			$\Sigma^{k}_{i=1}P_i > 5b-6k+1$	PDO > PSS
			$\Sigma^{k}_{i=1}P_i = 5b-6k+1$	PDO = PSS

(단, n, m은 임의의 양의 정수)

4.2 기존의 프로그래밍 환경과의 비교

본 논문에서는 기존의 병렬 프로그래밍 환경에서 프로그래머의 몫으로 남겨져 발생할 수 있는 문제점을 해결함으로써 다음과 같은 장점들을 갖는다.

첫째, 호출되는 부프로그램간에 종속성이 복잡하게 존재하는 경우 프로그래머는 그 종속성들을 고려하여 종속관계 유지를 위한 다수의 통신 프리미티브들을 삽입하여야 한다. 특히 그 종속관계가 복잡해질수록 프로그래머는 순차 프로그램을 병렬구조로 변환하기 위해 다수의 통신 프리미티브들을 삽입하는 오버헤드를 감수해야 하기 때문에, 프로그래머에 따라 이와 같은 오버헤드를 감수하기보다는 순차로 실행하려고 할 것이다. 본 논문에서는 컴파일러 단계에서 부프로그램들간의 종속관계를 분석하여, 적용 가능한 태스크 병렬구조로 자동 병렬화함으로써 프로그래머의 부담이 줄어들 뿐만 아니라 병렬성도 증진된다.

(그림 14)는 프로그래머에 의해 병렬화된 코드와 본 논문에서 제안한 컴파일러 단계에서의 묵시적 병렬성 추출에 의해 병렬화된 코드를 비교하기 위한 예제이다. (그림 14(a))의 순차 프로그램에서 부프로그램 P_1과 P_2간에 배열 A에 대해서는 LID(FDD)가, 배열 B에 대해서는 LID(BDD)가 존재한다. 프로그래머가 PROCESSES 블록구조를 선택하여 (그림 14(b))의 〈주 프로그램〉과 같이 병렬화하려면, 두 부프로그램간 종속관계를 고려하여 (그림 14(b))의 〈부프로그램 P_1〉과 〈부프로그램 P_2〉에서와 같이 두 부프로그램간 동기화를 위한 통신 프리미티브를 삽입해야 한다. 부프로그램간에 여러 가지 종속성이 혼재하게 되면 프로그래머는 각각의 종속성에 대해 고려해야 하고, 이에 따라 삽입해야 하는 통신 프리미티브도 증가하므로 프로그래머의 병렬화 오버헤드는 커진다. 그러나 본 논문에서는 이러한 부담을 프로그래머로부터 컴파일러로 전이시킨다.

둘째, 프로그래머는 두 부프로그램간의 종속성을 부프로그램 호출시 나열된 매개변수로부터 판단한 후, 태스크 병렬구조로 병렬화할 때 두 부프로그램간의 동기화를 위해 통신 프리미티브를 삽입한다. 반면에 본 논문에서는 [정리 1]에서 [정리 5]까지의 결과들에서 볼 수 있듯이, 다양한 종속성이 존재하더라도 컴파일러가 종속성의 종류 및 그 결합 형태에 따라 부프로그램들간에 통신이 발생하지 않는 병렬 수행구조로 자동변환한다. 즉, 본 논문에서는 통신 프리미티브를 삽입할 필요없는 'communication free' 형태의 태스크 병렬코드를 생성함으로서 통신 프리미티브 삽입 오버헤드와 불필요한 통신 수행비용을 줄일 수 있다.

(그림 14)의 예제에서 프로그래머는 자신의 판단에 의해 PROCESSES 블록구조를 선택함으로서 순차 프로그램의 시맨틱 유지를 위한 통신 프리미티브들을 부프로그램들에 삽입해야 한다. 그러나 본 논문에서는 (그림 14(a))의 순차 프로그램을 'communication free' 형태의 병렬 프로그램으로 변환하기 위해, 컴파일러가 체계적인 종속성 분석을 통하여 (그림 14(c))와 같이 부프로그램내에 통신 프리미티브가 삽입될 필요없는 PROCESSDO 루프구조로 병렬화한다. 결과적으로 통신 프리미티브를 삽입하지 않음으로서 통신 수행비용이 줄어든다.

셋째, 프로그래머는 포트란 M의 태스크 병렬구조인 PROCESSDO 루프구조나 PROCESSES 블록구조를 사용하여 프로그래밍한다. 그런데 상황에 따라 어느 구조를 사용하는 것이 더 효과적인지를 프로그래머 입장에서 판단하기는 쉽지 않다. 4.2절 〈표 3〉의 분석결과에서 보듯이 여러가지 요인에 따라 수행비용이 달라지기 때문에, 프로그래머의 직관적 판단에 의해 선택된 구조가 실제로는 다른 구조보다 비효과적일 수 있다. 그러나 컴파일러는 본 논문에서 유도한 〈표 3〉의 결과를 기반으로 루프 반복횟수, 호출되는 부프로그램 수 및 각 부프로그램의 수행 사이클수, 프로세서 수 등의 여러 가지 요인들을 종합적으로 분석하여 효과적인 병렬구조를 정확히 선택할 수 있다.

(그림 14)의 예제에서 루프 반복횟수 b는 10이고, 호출되는 부프로그램수 k는 2이다. 이때 가용 프로세서 수인 p#가 5라 하면, 〈표 3〉의 CASE 2의 ④($k < p\# < b$, $b = n \times p\#$, n은 임의의 양수)인 경우에 해당하므로 컴파일러는 이로부터 PROCESSDO 루프가 PROCESSES 블록구조보다 효과적이라는 정보를 얻어 PROCESSDO 루프로 변환한다. 그러나 프로그래머 입장에서 이러한 여러 가지 요인들을 체계적으로 고려하여 보다 효과적인 구조로 병렬화한다는 것은 쉽지 않은 일이다. 결과적으로 본 논문에서 제안한 컴파일러 단계의 병렬화된 코드는 정확한 정보들로부터 체계적인 분석을 통해 생성되므로 프로그래머

```
PROGRAM EXAMPLE1
INTEGER A(10), B(10)
   :
DO i=1, 10
   CALL P1(D:A(i+2), U:B(i-1))
  CALL P2(U:A(i+2),D:B(i-1))
ENDDO
```

(a) 순차 프로그램

```
PROGRAM EXAMPLE1
INTEGER A(10), B(10)
INPORT  (INTEGER) PI(2)
OUTPORT (INTEGER) PO(2)

DO i=1, 10
   CHANNEL (IN=PI(i), OUT=PO(MOD(i,2)+1))
ENDDO
DO I=1, 10
   PROCESSES
        PROCESSCALL P1(A, B, i, PI(1), PO(1))
        PROCESSCALL P2(A, B, i, PI(2), PO(2))
   ENDPROCESSES
ENDDO
END
```

<주 프로그램>

```
PROCESS P1(A, B, i, PI, PO)
INTEGER A(:), B(:)
INPORT  (INTEGER) PI
OUTPORT (INTEGER) PO
   :
A(i+2) = ...
SEND(PO), A(i+2)  ------------------>  RECEIVE(PI), A(i+2)
   :                                   ... = A(i+2)
   :                                      :
   :                                   B(i-1) = ...
RECEIVE(PI), B(i-1)  <------------------ SEND(PO), B(i-1)
... = B(i-1)                              :
   :                                      :
END                                    END
```

<부프로그램 P1>　　<부프로그램 P2>

(b) 프로그래머가 포트란 M의 PROCESSES 블록구조를 사용하여 변환한 예

```
PROGRAM EXAMPLE1
INTEGER A(10), B(10)
PROCESSDO i=1, 10
   PROCESSCALL P1(A, B, i)
ENDPROCESSDO
PROCESSDO i=1, 10
   PROCESSCALL P2(A, B, i)
ENDPROCESSDO
END
```

<주 프로그램>

```
PROCESS P1(A, B, i)
INTEGER A(:), B(:)
   :
A(i+2) = ...
   :
   :
   :
... = B(i-1)
   :
END
```

<부프로그램 P1>

```
PROCESS P2(A, B, i)
INTEGER A(:), B(:)
   :
   :
... = A(i+2)
   :
B(i-1) = ...
   :
   :
END
```

<부프로그램 P2>

(c) 컴파일러가 포트란 M의 PROCESSDO 루프구조를 사용하여 변환한 예

(그림 14) 프로그래머에 의한 병렬화와 컴파일러에 의한 병렬화의 예

(Fig. 14) An example of parallelization by programmer and compiler

에 의해 병렬화된 코드보다 효과적이다.

5. 결 론

본 논문에서는 종속성으로 인해 순차수행되어야 하는 프로그램에 대해 컴파일러 단계에서 묵시적 병렬성을 탐지하여, 병렬화 가능한 경우에는 포트란 M의 태스크 병렬구조인 PROCESSDO 루프와 PROCESSES 블록구조로 병렬화하고, 불가능한 경우에는 일부분에 대해서라도 병렬화하는 최적화 방안을 제안하였다. 그리고 PROCESSDO 루프와 PROCESSES 블록구조 모두로 변환이 가능한 경우, 조건에 따라 어느 구조로 변환하는 것이 효과적인가를 분석하고 기존의 방안과 비교하였다.

그러나 태스크 병렬 언어 관련연구 가운데 가장 많이 진척된 포트란 M도 프로토타입 정도만 구현된 상태라서 다양한 적용은 아직 미흡한 실정이다. 따라서 보다 안정된 포트란 M 컴파일러에 본 연구의 결과를 적용하여 다양한 응용 프로그램에서의 성능을 정량적으로 실험하는 연구가 필요하다. 향후 이 실험결과에 따라 본 연구가 보완되면 태스크 병렬언어의 상용화에 적용 가능하게 될 것이다.

참 고 문 헌

[1] V. Aho, R. I. Sethi and J. D. Ullman, 'Compilers :Principles, Techniques, and Tools,' Addison-Wesley Publishing Company, 1986.

[2] J. R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph. D Thesis, Rice University, Apr. 1983.

[3] Uptal Banerjee, 'Dependence Analysis for Supercomputing,' Kluwer Academic Publishers, 1988.

[4] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam, "Graphical Development Tools for Network-Based Concurrent Supercomputing," in Proc. of Supercomputing, pp. 435-444, 1991.

[5] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," in Proc. of the SIGPLAN Symposium on Compiler Construction, pp. 162-175, Jun. 1986.

[6] K. M. Chandy and I. Foster, "A Deterministic Notation for Cooperating Processes," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 8, pp. 863-871, 1995.

[7] N. Carriero and D. Gelernter, "Linda in Context," Communication of the ACM, Vol. 32, No. 4, pp. 444-458, 1989.

[8] I. Foster, "Fortran M as a Language for Building Earth System Model," Technical Report CRPC-TR92444, Rice Univ., 1992.

[9] I. Foster and M. Chandy, "Fortran M Language Definition," Technical Report CRPS-TR93429, Rice Univ., 1993.

[10] I. Foster, "Task Parallelism and High-Performance Languages," IEEE Parallel & Distributed Technology, Vol. 2, No. 3, pp. 27-36, 1994.

[11] I. Foster and M. Chandy, "Fortran M : A Language for Mudular Parallel Programming," Journal of Parallel and Distributed Computing, 1994.

[12] T. Gross, D. R. O'Hallaron and J. Subhlok, "Task Parallelism in a High Performance Fortran Framework," IEEE Parallel & Distributed Technology, Vol. 2, No. 3, pp. 16-26, 1994.

[13] J. Heinrich, MIPS R4000 Microprocessor User's Manual, MIPS Technologies, Inc., 1994.

[14] P. Newton and J. C. Browne, "The CODE 2.0 Graphical Parallel Programming Language," in Proc. of the ACM International Conference on Supercomputing, pp,167-177, 1992.

[15] J. Subhlok, D. R. O'Hallaron and T. Gross, "Task Parall Programming in Fx," Technical Report CMU-CS-94-12, Carnegie Mellon Univ., 1994.

[16] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda and J. Webb, "Communication and memory requirements as the basis for mapping task and data parallel programs," in Proc. of Supercomputing, pp. 330-339, Nov. 1994.

[17] J. Subhlok, and G. Vondran, "Optimal Mapp of Sequence of Data Parallel Tasks," in Pro

the Fifth ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming, pp. 134-143, Jul. 1995.

[18] M. Wolfe, 'Optimizing SuperCompilers for Supercomputers,' MIT press, 1989.

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

[20] 박성순, 박명순, "관계벡터를 이용한 프로그램 벡터화", 한국정보과학회 논문지, 제21권 2호, pp. 369-382, 1994년 2월.

박 성 순

1984년 홍익대학교 전자계산학과 졸업(학사)
1987년 서울대학교 계산통계학과(이학석사)
1988년~1990년 공군사관학교 전산학과 전임강사
1994년 고려대학교 전산과학과(이학박사)
1994년~현재 안양대학교 전자계산학과 조교수
관심분야:병렬 컴파일러, 병렬 언어

구 미 순

1987년 고려대학교 수학과 졸업(학사)
1995년~1997년 고려대학교 컴퓨터학과(이학석사)
1997년~현재 고려대학교 컴퓨터학과 박사과정
관심분야:병렬 컴파일러, 병렬 언어