성능 제고를 위한 재구조화 프로그램의 프로시저 변환

송 월 봉*

요 약

병렬처리 시스템 상에서 처리 속도의 극대화를 꾀하기 위해서는 하드웨어뿐만 아니라 소프트웨어적인 요소들이 함께 고려되어야 한다. 즉 병렬컴퓨터의 능력은 그것을 탐지하기위한 소프트웨어의 능력 없이는 불가능하다. 이에 따라 본 논문에서는 재구조화된 순차루프를 이용하여 성능 제고를 위한 프로시저 변환 방법을 제안 하고자 한다. 이것은 컴파일 시간에 중첩 병렬 DOALL루프로 바꾸어주는 순차루프의 자동 변환에 관한 절차이다. 이를 위해서 병렬컴파일러의 한 종류인 Parafrase Ⅱ 병렬화 컴파일러의 원시 프로그램을 분석하였으며 이를 토대로 기존의 Parafrase Ⅱ를 적용하여 중첩루프에서 효율적인 병렬처리를 위한 향상된 프로시저 변환 방법을 보였다. 이러한 병렬화 기법은 반복문에서 더 많은 병렬화 효과를 얻는 것을 확인하였다.

The Procedure Transformation of Restructured Program for Higher Performance

Worl-Bong Song*

ABSTRACT

For the improvement in ability of processing speed, the parallel processing system should be considered not only the elements of hardware but also the elements of software. Exactly the capability of parallel computer is impossible without software technic for detection the parallelism. In this paper, Using the restructured sequential loop, The procedure transformation method for the advanced performance is proposed. This is a procedure for the automatic conversion of a sequential loop into a nested DOALL loops at the compile time. For this, the source program of Parafrase Ⅱparallel compiler is analyzed and through this existing Parafrase parallel compiler, a improved procedure transformation method to parallel processing effectively in the nested loop is implemented. This proposed parallel scheme is certificated to obtain a more parallelism effect in the reiterative sentence.

Key words : procedure transformation, restructured program, parallelism, DOALL loop

* 인천대학교 컴퓨터공학부 (✉wbsong@incheon.ac.kr)
・제 1 저자(First Author) : 송월봉 ・교신저자(Correspond Author) : 송월봉
・접수일(2013년 3 월 4일), 수정일(1차: 2013 년 3월 20일), 게재확정일(2013 년 3 월 26일)

Ⅰ. 서 론

다수의 벡터 처리기들이 한 시스템 내에서 상호 연결되어 병렬 코드 프로그램들에 대해서 병렬 처리를 수행할 수 있는 형태의 시스템들이 등장하였다. 그러나 단순히 더 많은 프로세서를 시스템에 부착하는 것만으로는 시스템의 성능 향상에 커다란 영향을 미치지는 못한다. 따라서 병렬처리 시스템 상에서 처리 속도의 극대화를 꾀하기 위해서는 하드웨어뿐만 아니라 소프트웨어적인 요소들이 함께 고려되어야 한다. 즉 병렬컴퓨터의 능력은 그것을 탐지하기위한 소프트웨어의 능력 없이는 불가능하다. 이러한 요구에 맞추어서 병렬화[1,2,3]를 탐지하는 기술[4,5]과 병렬코드를 생성하는 기술[6,7]들이 발전되어 왔다. 이러한 소프트웨어적인 요소들은 기존의 프로그래밍 환경과는 다른 것으로 병렬환경이라고 부른다. 병렬환경 중에서도 직접 병렬 언어로 프로그램을 작성하는 것이나 임의로 제작된 순차 프로그램을 새로운 병렬언어로 수정하는 작업은 주어진 병렬처리 시스템의 구조를 이해하고 병렬처리에 관한 전문 지식을 새로이 익혀야 하는 등 프로그램 작성자에게 상당한 부담이 된다. 따라서 순차 프로그램을 자동적으로 병렬처리가 가능한 형대로 변환해 주는 병렬화 컴파일러(parallelizing compiler) 기법이 가장 현실적이며 가장 효과적인 방법이라고 볼 수 있다. 지금까지 병렬화 컴파일러의 연구는 메시지 교환 방식의 병렬 프로그램 생성보다는, 공유 메모리(shared memory) 방식의 병렬 컴퓨터에서 유용한 병렬 루프 생성이나 벡터 연산의 생성을 중심으로 수행되어 왔다. 그 결과 Illinois대학의 Parafrase I, II[7] Rice대학의 PFC, IBM의 PTRAN, Kuck and Association의 KAP, Pacific Sierra의 VAST, Bonn 대학의 SUPERB, Stanford대학의 SUIF, Illinois대학의 Polaris[8] 등이 있다.

본 논문에서는 Illinois대학의 Parafrase Ⅱ 병렬화 컴파일러의 원시코드를 분석하고 병렬처리의 핵심이 되는 루프의 재구조화 프로그램에 대한 제어흐름분석과 자료흐름분석을 토대로 종속성분석[9,10]을 수행하고자하며 이를 토대로 재구조화 프로그램의 성능 제고를 위한 프로시저 변환 방법을 제안 하고자 한다. 이러한 병렬화 컴파일러는 실행될 병렬 컴퓨터의 구조에 따라 병렬 수행 구조나 고려할 사항이 다르기 때문에 병렬 컴파일러를 사용함으로써 프로그램의 이식성(portability)을 높일 수 있으며, 프로그래머가 병렬 프로그램 대신 순차 프로그램을 작성하도록 함으로서 프로그래머의 부담을 덜 수 있다.

Ⅱ. 관련이론

2.1 흐름종속을 갖는 중첩루프

병렬화 컴파일러[8]는 주로 프로그램 명령어들 사이에 수행 순서 관계와 프로시저간의 호출 관계를 분석하는 제어 흐름 분석, 자료의 정의 및 사용 관계와 프로시저를 호출할 때 자료 전달에 관한 다양한 정보를 분석하는 자료 흐름 분석, 그리고 배열 원소 첨자를 분석하여 루프 내의 배열 요소로 인한 종속성 관계에 대한 분석을 수행하며 이 결과를 바탕으로 병렬 프로그램

```
DIM A(5:N1,3:N2)
DIM B(5:N1,3:N2)
   do 100 i = 5, N1
   do 100 j = 3, N2
                A(i,j) = B(i-3, j-5)
                B(i,j) = A(i-2, j-4)
100  continue
end
```

그림 1. 프로그램 예
Fig. 1. Sample program

생성을 위한 코드 변환이 수행된다. <그림 1>의 경우 첨자 A, B가 사이클을 이루며 자료 종속성을 갖기 때문에 병렬 컴퓨터에서 한꺼번에 수행시킬 수가 없

다. <그림 1>의 중첩루프에 대한 자료 흐름 종속성은 그림 2와 같으며(일부만 표현), <그림 1>에서 병렬로 처리할 수 있는 문장의 수를 살펴보면 다음과 같다.

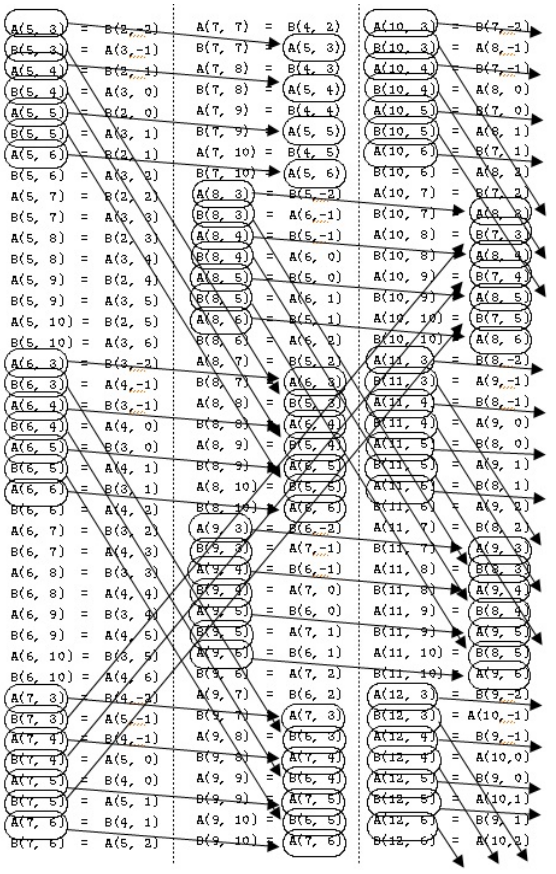

그림 2. 그림 1의 흐름종속 관계도
Fig. 2. Flow dependence diagram of Fig. 1.

OUT(X)를 문장 X에서 좌측 값 즉, 결과 값이라 하고, IN(X)를 문장 X에서 우측 값인 입력 값이라 하자. 이때, <그림 1>에서 5번째 문장인 A(i, j) = B(i-3, j-5)를 S5라 하고, 6번째 문장인 B(i, j) = A(i-2, j-4)를 S6라고 할 때, $OUT(S5) \cap IN(S6) \neq \varnothing$이므로 <그림 1>의 5번째 문장과 6번째 문장은 흐름 종속(flow dependence)을 가진다. <그림 2>는 <그림 1>에서 생기는 자료의 흐름 종속을 나타낸 그림으로써 <그림 1>의 첫 문장의 좌측 값인 A(5, 3)과 21번째 반복에서 생기는 문장의 우측 값인 A(5, 3)을 살펴보면, 21번째 반복에서 생기는 문장의 A(5, 3)의 값은 첫 번째 문장에서 A(5, 3)의 값이 대치되어야 한다. 그러나 만약 이 두 문장을 병렬로 처리할 때는 21번째 반복에서 생기는 문장이 먼저 수행될 수 있으므로 21번째 반복문에서 생기는 문장의 좌측 값인 B(7, 7)에는 엉뚱한 값이 대입될 수 있다. 그러므로 첫 번째 반복과 21번째 반복에서 생기는 각각의 문장들은 서로 동시에 처리할 수 없다. <그림 1>의 처음 나오는 종속은 20번 반복 후에 생기는 흐름 종속(flow dependence)이며, 20번의 반복 동안 생기는 40개의 문장은 서로 종속성을 가지지 않는다. 따라서 <그림1>은 한 번에 40개의 문장을 서로 병렬로 처리가 가능하다.

2.2 loop interchange

2개의 열(행)을 교환한다.

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \overset{interchanging}{\rightarrow} \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$

k와 k+1 loop interchange는 일치하는 distance(direction)vector를 변환한다.

int(k, k+1) ⇒

(d1,d2,--,dk,dk+1,--)→ (d1,d2,--,dk+1,dk,--)

예를 들면, 이중 루프에서는 종속관계(=,<)를 (<,=)로 변환하고 삼중 루프는 IJK를 IJK→IKJ→KIJ→KJI나 IJK→JIK→JKI→KJI로 변환한다.

2.3 병렬컴파일러 분석

Parafrase Ⅱ는 소스 코드의 크기가 매우 방대하기 때문에 흐름도와 결정 테이블, N-S도표 등의 분석방법이 거대한 원시코드를 분석함에는 적합하지가 않다. 따라서 본 논문에서는 프로시저 변환을 통한 재구조화 프로그램의 성능향상을 위하여 Parafrase Ⅱ의 특징 중의 하나인 패스들을 분석하고, 해당 패스들이 참조하는 파일들을 파악하고 나서, 각각의 파일들이 참

조하는 함수들을 분석하는 방법을 택하고자 한다. Parafrase Ⅱ는 여러 단계의 패스들을 실행시킴으로써 결과를 출력하게 된다.

주요 패스들의 실행 순서는 다음과 같다.

fixup → callgraph → sumfcn → param_al → donest → depend → flow → constant → induction → builddep → hier → dotodall → codegen.

각 패스들은 각각 여러 개의 원시코드 파일들을 포함하고 있으며 주요 패스들의 각 특징을 살펴보면 다음과 같다.

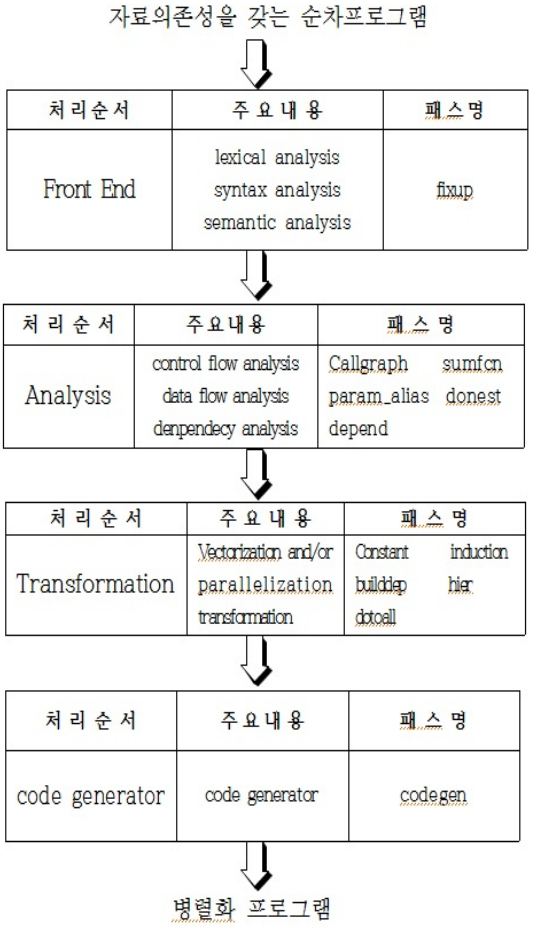

그림 3. Parafrase Ⅱ의 실행 순서

Fig. 3. Processing flow graph of Parafrase Ⅱ

2.3.1 sumfcn과 libsum

sumfcn은 Parafrase Ⅱ의 프로시저들의 참조 정보를 분석하고 요약하는 역할을 하는 패스이다. 이 패스가 존재하는 디렉토리는 src/analyses/interprocedure/procedure_reference_info이다.

libsum은 분석 시 즉시 사용하지 않는 library routine들을 다룬다. 루틴들에 관해서는 어떤 것도 알려져 있지 않기 때문에 조심스러운 추측을 해야 하고, 그 추측에 모든 전역 변수들이 연관되어져 있어야 한다. 이러한 루틴들은 call graph의 끝에서 발생하기 때문에 루틴들은 프로시저간 관련 분석의 이득들을 모두 제거할 수 있고, 어떤 경우들에서는 이득을 무의미하게 한다.

```
common gv
.
.
.
do i=1, N
        C(i) = gv * B(i)
        A(i) = sin(B(i))
        enddo
.
.
.
```

그림 4. Library Summary 예

Fig. 4. Example of Library Summery

간단한 프로시저간 분석은 <그림 4>에 나타난 반복문이 충분히 병렬화 할 수 있음을 알 수 있다. 그러나 sin함수를 호출하기 때문에 어떤 경우의 최적화가 방해된다. sin이란 함수에 관하여 아무것도 알려지지 않았기 때문에 실인수(actual parameter) B와 전역변수 gv를 갱신하여야한다. 따라서 이 반복문은 안전하게 병렬화 할 수 없다. 만약 sin이라는 함수가 어떤 인수의 값도 변화시키지 않는다면, 반복문은 병렬화 할 수 있다.

library 루틴들의 관련 정보가 가지고 있는 데이터베이스를 유지함으로써 보다 정확한 정보를 얻을 수 있다. 그러나 이런 정확한 정보를 얻을 수 있는 방법은 단지 데이터베이스를 통해서만 얻을 수 있는 것은 아니다. 데이터베이스를 이용하면 너무 많은 메모리를 사용하므로, 루틴들의 집합들을 분석하고 관련 정보가 하나의 파일에 저장할 수 있다. 게다가 컴파일러 분석은 조심스럽게 진행되므로, 컴파일러가 결정할 수 없는 부분을 사용자에게 유용한 정보를 사용하기 위하여 직접 옵션을 정할 수 있다. 주어진 파일들은 데이터베이스를 이루고, 컴파일러는 이런 정보들을 읽고 종속성(dependence)과 변환(transformation)과 분석(analysis)을 계산할 때 이런 정보를 이용한다.

2.3.2 param_alias

param_alias 패스는 가인수(formal parameter)로 인해 생기는 aliasing 정보를 만들고 심볼 테이블에 저장한다. 또한 가인수뿐만 아니라, 전역변수도 고려한다. 즉, 이 패스는 이명 분석(aliasing analysis)을 거쳐 생긴 정보를 심볼 테이블에 저장한다. 다른 이름을 가진 두 개의 변수가 프로그램 수행 중에 같은 메모리의 위치를 가리키게 될 때 이를 이명관계라고 한다. 이러한 이명관계는 두 가지 형태가 있는데 정적(static) 이명관계와 동적(dynamic) 이명관계가 그것이다. 정적 이명관계는 서로 다른 두 개의 변수가 같은 메모리를 가리키고 있다는 것이 명시적으로 나타난 경우로 Fortran의 EQUIVALENCE 문장의 경우가 그러한 경우이다. 동적 이명은 어떤 문장의 수행의 결과로 이명관계가 발생하는 경우로서 포인터 연산이나 프로시저 호출시의 인자로 인하여 발생한다. 포인터 연산으로 인한 이명관계의 분석은 명확히 분석해 내기 어려운 것으로 간주되었으나, 최근 포인터 연산으로 인한 이명관계 분석을 위한 연구가 활발히 진행되고 있다. 여기서는 프로시저간의 분석과 명확한 연관성이 있는 프로시저 호출로 인한 이명관계 분석에 대하여 알아보기로 한다. 프로시저 호출로 인한 이명관계의 발생은 다음과 같은 두 경우로 나타낼 수 있다. 두 개의 다른 형식인자가 이명관계에 있는 실인자나 동일한 실인자에 바인딩되어 있는 경우와 형식인자에 전역 변수가 바인딩되어 있는 경우가 그것이다. 그림 5의 프로그램을 살펴보면 SUBA의 첫 번째 호출에서 SUBA의 형식 인자인 X와 Y는 모두 MAIN의 I1에 바인딩 되어 있으므로 첫 번째 경우에 해당하며, X와 Y는 이명관계를 가지게 된다. 두 번째 호출에서는 Y는 전역 변수 G에 바인딩 되므로 두 번째 경우에 해당하여 프로시저 내에서 G와 Y는 이명관계에 놓이게 된다.

```
PROGRAM MAIN
INTEGER I1, I2, G
COMMON G
CALL SUBA(I1, I1)
CALL SUBA(I2, G)
END

SUBROUTINE SUBA(X, Y)
COMMON G
.....
END
```

그림 5. 이명 관계 설명을 위한 소스코드
Fig. 5. Source code for aliasing relation

2.3.3 depend

depend 패스는 각 문장의 In/Out 집합을 생성한다. C언어는 변수 이름을 마음대로 가질 수 있기 때문에 각 문장에 대한 In/Out 집합은 꽤 복잡할 수 있다. In/Out 집합의 요소들은 변수들(심볼 테이블을 가리키는 포인터들)과 변수들이 될 수 있는 조건의 목록들이다. In/Out 집합들이 연산될 수 없을 때마다 보편적인 (universal) In/Out 집합으로 간주된다. 이것은 이 문장이 어떤 메모리의 위치에도 읽고, 쓸 수 있다는 것을 의미한다. 에러 메시지와 출력은 이 패스에 인수로써 정해진 파일이름으로 저장된다. 이 패스를 이루는 소스 파일의 디렉토리는 src/misc/control이다.

flow graph를 생성하는 flow 패스의 경우 -d 옵션을 통해 디버그 플래그가 양수 값을 가지면, flow graph와 그에 상응하는 연결하는 배열과 flow graph의 dominator tree에 관한 정보가 출력된다. flow 패스와 관련된 주요 스트럭쳐는 두 개가 있다. 이 두 개의 주요 스트럭쳐는 노드의 정보를 담고 있는 flowgraph_t 스트럭쳐와 edge의 정보를 담고 있는 flowedge_t 스트럭쳐이다. flow graph의 노드는 매크로를 통하여 접근하는 필드를 가지고 있다. 다음 <표 1> 은 flowgraph_t의 매크로 목록들이다.

표 1. flowgraph_t 스트럭쳐의 매크로
Table 1. Macro list of flowgraph_t structure

매크로명	의 미
F_BLOCK	그래픽 인터페이스에 의해 사용되는 유일한 값의 basic block number.
F_FIRST	노드의 첫 문장을 가리킨다.
F_LAST	노드의 마지막 문장을 가리킨다.
F_EDGEIN	노드에 입력되는 선(edge)들의 목록을 가리킨다.
F_EDGEOUT	노드로부터 출력되어지는 선(edge)들의 목록을 가리킨다.
F_DOMINATOR	노드의 바로 전의 우선 수행된 노드를 가리킨다.
F_NEXT	다음 flow graph 노드를 가리킨다.

flowedge_t에 관계된 매크로는 <표 2>와 같다.

표 2. flowedge_t 스트럭쳐의 매크로
Table 2. Macro list of flowedge_t structure

매크로명	의 미
F_HEAD	edge가 시작되는 flow graph의 노드를 가리킨다.
F_TAIL	edge가 끝나는 flow graph의 노드를 가리킨다.
F_IN_NEXT	다음에 입력되는 edge를 가리킨다.
F_OUT_NEXT	다음에 노드에서 빠져나가는 edge를 가리킨다.

2.3.4 constant와 induction

constant 패스는 프로그램의 전역에 상수를 전파한다. 이것은 constant folding과 constant propagation의 작업을 한다. constant folding은 컴파일 시간에 상수식을 계산하여 그 결과를 필요한 곳에 사용하며, constant propagation은 고정된 값을 갖는 변수를 상수로 대치하는 것이다. 이 패스는 induction 패스의 결과 값과 함께 수행될 때 정확한 constant propagation이 이루어진다. 이 패스의 원시코드 파일들은 src/transformations/constant_propagation에 존재한다.

induction 패스는 근본적으로 Parafrase Ⅱ의 심볼 분석을 한다. 이 패스는 프로그램 내에서 루프 첨자 변수와 루프 invariant라는 용어로써 표현 가능한 모든 induction 식을 인식한다. 이 패스는 IF나 GOTO 문과 같은 문장들을 포함한 루프도 인식하여 분석한다.

2.3.5 builddep와 hier

builddep 패스에서는 depend 패스에서 생성된 정보를 사용하여 종속 그래프(dependence graph)를 만든다. 이 패스는 문장들 사이의 모든 종속관계를 찾고, 종속관계를 나타내는 적당한 선을 구성한다. 이 작업 중에, 종속 방향 벡터(dependence direction vector)또한 만들어진다. In/Out 집합들이 보편적인 집합(어떤 메모리 위치에서도 읽기와 쓰기가 가능하다)일 때, 모든 가능한 종속 선은 출력되지 않으나, 이 문장에서 종속 노드는 이 사실이 표시되어진다. 출력은 정해진 파일에 적을 수 있고, 또한 draw_depend 패스를 이용하여 그래픽으로 나타낼 수 있다. 이 패스가 존재하는 디렉토리는 src/misc/control이다. 자료 종속 그래프의 선언과 정의는 include/depgraph.h와 include/depmac.h에 존재한다. 전자는 종속그래프의 노드와 edge들의 스트럭쳐 정의를 담고 있으며, 후자는 이런 스트럭쳐의 각 필드를 접근하는 매크로의 정의를 포함하고 있다. <표 3>은 dep_node 스트럭쳐의 필드들을 설명한다.

표 3. dep_node 스트럭쳐의 필드
Table 3. Fields of dep_node structure

필드명	내용
DEP_IN_LIST	현재 노드에 들어오는 edge들의 목록을 가리킨다.
DEP_OUT_LIST	현재 노드로부터 나오는 edge들의 목록을 가리킴.
DEP_NEXT	다음 종속 그래프 노드를 가리킨다.
DEP_STMT	현재 종속 노드와 상응하는 문장을 가리킨다.
DEP_NUM	그래픽 인터페이스에서 노드를 위해 사용된 숫자
DEP_ALL	정상적으로 표현될 수 없는 노드로부터의 종속 edge가 있는지 없는지를 결정하는 flag

<표 4>는 dep_edge 스트럭쳐의 필드들을 설명한다.

표 4. dep_edge 스트럭쳐의 필드
Table 4. Fields of dep_edge structure

필드명	내용
DEP_TYPE	종속의 종류; flow, anti, output
DEP_VAR	종속을 일으키는 변수
DEP_HEAD	종속의 관계선의 출발점
DEP_TAIL	종속의 관계선의 도착점
DEP_OUT_NEXT	출발점으로부터 밖으로 나가는 다음 edge를 가리킨다.
DEP_IN_NEXT	도착점에 입력되는 다음 edge를 가리킨다.
DEP_SC_OR_VEC	종속의 종류; 스칼라, 벡터
DEP_DIRS	종속 방향 벡터

hier 패스에서는 flow/induction/depence 패스에서 생성된 정보를 이용하여 계층적인 그래프를 만든다. 이 패스는 프로그램을 나타내기 위한 계층조직을 만든다. 계층조직은 반복문에 근간을 두고 있다. 또한 반복문은 induction 패스의 한 부분으로써 식별된다. 각 단계에서 비순환 제어 흐름 그래프(acyclic control flow graph)가 나중 수행 트리(post dominator tree)로서 인정될 때, 제어종속 그래프와 자료종속 그래프가 만들어진다. 나머지 종속은 이 패스의 경험적인 방법으로 인식되어진다. hier 패스는 패스 파일에 옵션을 첨가하여 상세화 할 수 있다. 표 5는 hier 패스에 사용되는 옵션들을 설명한다. 이러한 옵션들은 scalar privatization이 dotodoall 패스에 의해 발견된 병렬 루프의 수를 증가시키게 한다.

builddep 패스는 hier 패스 전에 사용되고, hier 패스는 종속그래프 안의 잘못된 연결선을 삭제한다. <그림 6>은 패스 파일에 hier 패스의 옵션들이 실제 적용된 예를 보여준다. 이 패스의 원시코드 파일이 존재하는 디렉토리는src/analyses/hierarchical_task_graph이다.

표 5. hier 패스의 옵션
Table 5. Option of hier path

Option	내용
-o	hier 패스를 호출한다. 이 옵션은 다른 옵션과 함께 쓰이지 않는다.
-k	좀더 정확한 종속관계를 얻기 위해서, KILL 집합을 계산한다. 사용되기 전에 정의된 변수들은 복잡한 노드의 IN 집합에서는 포함되지 않기 때문에, KILL 집합을 계산하는 것은 복잡한 노드에서 IN 집합을 정확히 계산할 수 있다.
-l	각 노드의 모든 지역 변수를 찾는다.
-s	이 옵션은 코드들 변환시키지 않음에도 불구하고, 더 많은 DOALL 반복문이 발견되게끔 한다. 이것은 종속 그래프 상에서 각 노드의 모든 지역 변수를 찾는다.
-v	DO 반복문들에서 지역 변수들의 선언을 생성한다. 이 옵션도 -l옵션을 포함하며, 만약 dotodoall 패스가 반복문을 Cedar의 병렬 반복문으로 바꾼다면, 지역변수 선언은 결과 코드에서 나타날 것이다.

```
builddep
hier -klsv
dotodoall
```

그림 6. hier 패스의 옵션이 적용된 예
Fig. 6. Imple. example of the hier path option

2.3.6 dotodoall

dotodoall 패스에서는 builddep에서 생성된 정보를 사용한다. 프로그램 내에 있는 모든 반복문을 검사하여 반복문들이 병렬화 될 수 있는지 없는지를 검사한다. 이것은 '=' 방향에서 종속이 존재하는지 검사하는 것과 동일하다. 이 패스는 먼저 바깥쪽의 반복문이 병렬화 되는지를 확인한다. 이것은 내부 루프가 '<'나 '>'인 종속관계이더라도 만약 외부 루프의 순차적 성질에 의해 이들 방향이 만족되면 병렬화 할 수 있음을 의미한다. 만약 하나의 반복문이 병렬화 될 수 없다면, 병렬화를 막는 모든 종속관계들이 지적된다. 출력은 정해진 출력 파일에 저장된다. 이 패스의 소스 파일이 존재하는 경로는 /src/misc/control이다. 패스가 존재하는 디렉토리를 요약하면 <표 6>과 같다.

표 6. 해당 패스별 디렉토리 위치와 파일 이름
Table 6. Directory position & File name of Path

패스 이름	위치하는 디렉토리와 파일 이름
builddep	src/misc/control/interface.c
callgraph	src/analyses/call_graph/callgraph.c
codegen	src/misc/control/interface.c
constant	src/transformations/constant_propagation/constant.c
deadcode	src/transformations/dead_code_elimination/dead_code.c
debug_symtab	src/misc/control/interface.c
debug_codegen	src/misc/control/interface.c
depend	src/misc/control/interface.c
distribute	src/transformations/loop_distribution/distribute.c
donest	src/misc/control/interface.c
dotodoall	src/misc/control/interface.c
draw_flow	src/misc/control/interface.c
draw_depend	src/misc/control/interface.c
draw_call	src/misc/control/interface.c
expand	src/transformations/inline_expansion/xpand.c
f2c	src/transformations/Fortran_to_C
fixup	src/lexers_and_parsers/common/fixup.c
gen3ad	src/transformations/source_to_3_address/gen3ad.c
hcodegen	src/code_generators/Htgl/hcodegen.c

III. Parafrase II의 병렬코드

Parafrase Ⅱ의 패스의 실행순서와 패스의 원시코드파일이 존재하는 디렉토리와 각 패스의 처음을 호출하는 함수를 살펴보았다. 지금까지의 내용을 토대로 <그림 1>에 대한 Parafrase Ⅱ[p2fpp -p pass.dat source.c : source.c라는 파일을 병렬화 하려하고, 패스 파일이 pass.dat라고 가정할 경우]를 이용하여 생성한 병렬 코드는 <그림 7>과 같다.

<그림 7>에서 Sn을 n번째 문장이라고 할 때, S6의 CDOALL문장은 3~10의 반복구간 동안 문장 S7와 문장 S8를 순차로 처리하므로, 한번에 8개의 문장을 병렬로 처리가 가능하다. 즉 프로그램 예<그림 1>에서는 한 번에 40개의 문장이 병렬 처리가 가능하나, Parafrase Ⅱ에서 생성된 병렬코드는 한 번에 8개의 문장이 한 번에 병렬로 처리가 가능하다.

```
 IMPLICIT NONE
REAL A(5:N1,3:N2)
REAL B(5:N1,3:N2)
INTEGER i, j
    DO 100 i = 5,N1
        CDOALL 100 j = 3,N2
            A(i,j) = B(i - 3,j - 5)
            B(i,j) = A(i - 2,j - 4)
100  CONTINUE
END
```

그림 7. 그림 1에 대한 Parafrase Ⅱ병렬 코드
Fig. 7. Parafrase ⅡParallel code of Fig. 1

따라서 Parafrase Ⅱ에서 생성하는 코드는 최적의 코드가 아님을 알 수 있다. 즉, 이러한 점을 개선하여 재구조화 프로그램의 성능을 개선할 수 있다.

Ⅳ. 성능향상을 위한 프로시저변환

본 논문 2.1에서 분석한 내용과 같이 종속성이 분석된 재구조화 프로그램의 성능 즉 병렬성을 극대화하기 위하여 본 논문에서는 Parafrase Ⅱ의 다른 패스들은 그냥 사용하고, 2.2절의 관련이론 루프 인터체인지 기법[11,12]을 적용하여 중첩된 루프에서 병렬 코드를 중첩된 루프의 바깥으로 끌어내는 기법을 추가시키고자 한다. 이를 위하여 codegen 패스에서 DOALL 문장을 중첩된 루프의 바깥으로 다음과 같이 끌어내었다.

```
CDOALL 100 j = 3, N2
DO 100 i = 5, N1
A(i,j) = B(i - 3,j - 5) : B(i,j) = A(i - 2,j - 4)
100 CONTINUE
```

그 결과 기존 병렬코드(그림 7)에 비하여 예제 프로그램 대비 5배의 성능 향상이 이루어 졌음을 확인할 수 있었다. 따라서 구현하고자 하는 구조는 그림 8과 같다.

즉, 기존의 code generator에서 loop interchage를 먼저 한 후 코드를 생성하게 하였다. 반복문 교환은 중첩 반복문을 서로 바꿈으로써 반복문을 변형시키는 것이다. 이것은 병렬화를 극대화시키기 위하여 반복의 횟수가 많은 반복문을 밖으로 내보내거나 적은 병렬화 즉,

벡터 머신에 적응시키기 위하여 반복 횟수가 많은 반복문을 안으로 보내어서 주어진 병렬화 구조에 알맞게 메모리 접근 패턴을 바꾸기 위함이다. 본 논문에서는 반복의 횟수가 많은 반복문을 밖으로 내보내어서 벡터화 보다는 병렬성 제고에 초점을 맞추었다.

자료의존성을 갖는 순차프로그램

⇓

처리순서		주요내용	패스명	비고
1	도입단계	lexical analysis syntax analysis semantic analysis	fixup	
2	분석단계	data flow control flow data dependency	callgraph sumfcn libsum param_alias donest depend	
3	병렬화단계	vectorization and/or parallelization transformation	constant induction builddep hier dotoall	
4	코드생성단계	**loop interchange** code generator	codegen	loop interchange 수행

⇓

병렬성이 증대된 병렬화프로그램

그림 8. 성능 제고를 위하여 개선된 구성도
Fig. 8. The new process for Performance raising

Ⅴ. 결 론

재구조화 프로그램의 수행 빈도수를 줄이고 이를 바탕으로 한 성능 제고를 목적으로, 병렬 컴파일러의 한 종류인 Parafrase Ⅱ를 분석하고 프로시저 변환을 통한 향상 방안을 제안하였다. 이는 기존의 Parafrase Ⅱ를 갱신하여 간단하게 loop interchange를 자동으로

구현하는 것이다. 이러한 loop interchange는 반복문에서 CDOALL문장을 바깥쪽으로 보냄으로서 더 많은 병렬화 효과를 주는 것으로 확인 되었다. 따라서 본 연구 결과는 프로그래머가 병렬 프로그램 대신 순차 프로그램을 작성하도록 함으로서 프로그래머의 부담을 덜 수 있고, 병렬 프로그램들은 실행될 병렬컴퓨터의 구조에 따라 병렬 수행 구조나 고려할 사항이 다르기 때문에 제안된 병렬컴파일러를 사용함으로서 프로그래머의 이식성(portability)을 높일 수 있을 것으로 사료된다. 이는 향후 병렬 처리시스템의 실행시간을 단축시킬 수 있는 고성능 병렬화 컴파일러를 개발하고, 완전하고 효율적인 병렬화 컴파일러를 구현하는데 도움이 될 것이다.

참고문헌

[1] S. K. Kim, S. Y. Han, “Performance comparison of DCOM, CORBA and Web service, PDPTA 2006, USA,”[Internation Journal] Jun. 2006

[2] SeongKi Kim, SangYong Han, “Media Frame: parallel multimedia system architecture through HTTP redirection,”정보처리학회 A, Feb. 2007

[3] S. K. Kim, S. Y. Han,“The globally scalable architecture for heterogeneous video servers without any modification,” WEBIST [Internation Journal] Portugal, Apr. 2006

[4] William E. Ryan, "An Introduction to LDPC Codes," Department of ECE, The University of Arizona, August 19, 2003

[5] Polychronopoulos, C.D., "More on advanced loop optimization," CSRD Report No. 667, Univ. of Illinois at Urbana-Champaign, Oct.2003

[6] W. B. Song, “Program Restructuring for Promotion Parallelism of the Uniform & Non-uniform Loop,” kkits, vol. 7, No.2, Apr. 2012

[7] Padua, D. A. AND M. Wolfe,“Advanced compiler optimization for supercomputers,” Comm ACM, Vol. 29, No.

[8] Blume, B. et. al., "Polaris : The Next Generation in Parallelizing Compilers," Proceedings of the Workshop on Languages and Compilers for Parallel Computing, Apr. 1994

[9] Sang Il Park, Weol Seon Park, Sung-Dae Youn, "A Data Dependency Elimination Method in Multidimensional Index Loop Using a dependency Function", 「ICIS '01」pp53-57, 2001.10

[10] Jong Chan Yun, Sung-Dae Youn, "Frequent Itemset Creation Using Sequence Association rule", 「Processing of the IASTED International Conference Parallel and Distributed Computing and Networks」pp170-174, 2008.2

[11] D. MacKay, Information Theory, Inference, and Learning Algorithms, CAMBRIDGE, 2003

[12] W. B. Song, “The Procedure Transformation Algorithm Using Dependency Elimination in Non-uniform,” kkits, vol. 6, No.2, Apr. 2011

감사의 글

본 논문은 인천대학교 2012년도 자체연구비 지원에 의하여 연구되었음.

저자소개

송월봉(Worl-Bong Song)

1974년 숭실대학교 공학사
1982년 한양대학교 공학석사
1998년 순천향대학교 공학박사

2010년~ 현재 인천대학교 컴퓨터공학과 교수
※ 관심분야: 병렬처리, 컴파일러, 알고리즘