2001년 한국정보처리학회 춘계 학술발표논문집 제8권 제1호

프로시저 호출을 가진 루프에서 병렬성 추출

장 유 숙*, 박 두 순*
*순천향대학교 정보기술공학부
e-mail:usjang@unitel.co.kr

The extraction parallelism in loops with procedure call

Yu-Sug Chang*, Doo-Soon Park*
*Division of computer science and computer engineering,
College of Engineering Soonchunhyang University

요 약

프로그램 수행시간의 대부분이 루프 구조에서 소비되고 있기 때문에 순차 프로그램을 병렬 프로그램으로 변환하는 연구들이 많이 행해지고 있고 그 연구들은 하나의 프로시저 내 루프 구조의 변환에 치중되고 있다. 그러나 대부분의 프로그램들은 프로시저 간 잠재된 병렬성을 가지고 있다. 본 논문에서는 프로시저 호출을 가진 루프에서 병렬성 추출 방식을 제안한다. 프로시저 호출을 포함하는 루프의 병렬화는 대부분 uniform 형태의 코드에서만 집중되었다. 본 논문에서는 uniform 코드, nonuniform코드 그리고 복합된(complex) 코드를 제안된 알고리즘과 loop extraction, loop embedding방법을 CRAY-T3E로 성능 평가하였다. 그리고 제안된 알고리즘이 효율적인 방법이라는 것을 보여준다.

1. 서론

현대 슈퍼컴퓨터 구조적 개선의 주요 목적은 병렬화를 증가하는 것이다. 병렬 하드웨어의 능력은 그것을 탐지하기 위한 소프트웨어의 능력 없이는 불가능하다. 이러한 요구에 주요한 과제는 병렬화를 탐지하는 기술을 컴파일러에서 만들고 병렬화를 강화하기 위한 변환을 수행하고 코드에서 종속성을 최소화하는 것이다.

프로그램 수행시간의 대부분이 루프 구조에서 소비되고 있기 때문에 순차 프로그램을 병렬 프로그램으로 변환하는 연구들이 루프 구조의 변환에 치중되고 있다. 루프에 프로시저 호출을 포함하는 경우 컴파일러는 그것을 최적화하기를 원할 것이다.

본 논문에서는 프로시저분석 방법과 프로시저 변환 방법을 서술한다. 그리고 병렬화를 위한 최적화 방법과 uniform 코드, nonuniform코드 그리고 복합된(complex) 코드를 프로시저 변환 알고리즘을 적용한 중간 코드와 loop extraction, loop embedding방법으로 성능평가하고 세 가지 코드를 자료 종속성 제거 알고리즘으로 병렬화 하여 CRAY-T3E로 성능 평가한다.

2. 프로시저 분석과 변환

프로시저 분석은 프로시저 내 분석과 프로시저 간 분석으로 구분하고 프로시저 변환도 프로시저 내 변환과 프로시저 간 변환으로 구분한다[5,6].

2.1 프로시저 내 분석

프로시저 내 분석이란 한 프로시저 내에서의 자료 흐름이나 제어 흐름에 관한 정보를 분석하는 것을 말한다. 프로시저 내 분석 방법은 프로그램 문장 사이의 순행순서, 변수 값 도달 정의 탐색, 상수 전달, 유도변수, 복사 전파, 공통 부분 식, 불필요한 코드 제거, 코드 이동, 식의 연산 순서 변경, 중복된 로드와 저장 명령문 제거, 불필요한 jump 제거, 레지스터 할당 등이 있다[8,10].

2.2 프로시저 간 분석

프로시저 간 분석은 프로시저 사이의 호출/피 호

출 관계에 나타나는 자료의 흐름에 관한 분석을 말한다. 프로시저 간 분석을 하기 위해서는 먼저 각 프로시저 사이의 호출/피 호출 관계를 나타내는 호출 그래프를 작성하고, 이를 이용하여 프로시저간 자료의 흐름을 분석하게 된다. 각 프로시저 사이에 자료가 전달되는 경로는 형식 인자와 실 인자사이의 바인딩과 전역 변수에 의해 이루어지므로, 인자들 사이의 자료 흐름을 나타낼 수 있는 바인딩 그래프를 사용하여 바인딩과 전역 변수에 의한 자료의 흐름을 분석한다. 그리고 이명 관계, 상수 전달, 참조 정보 등을 분석한다[1,10].

2.3 프로시저 내 변환

프로시저 내에서 사용되는 변환 방법으로 루프 분리(Loop distribution), 루프 융합(Loop Fusion), 루프 반전(Loop Reversal), 루프 왜곡(Loop Skewing), 루프 교환(Loop Interchange), 루프 타일링(Loop Tiling) 등의 변환 방법이 있다[2,7,9].

2.4 프로시저 간 변환

프로시저 사이에 적용되는 변환 방법으로 inline substitution은 피 호출 프로시저의 문장들을 프로시저 정보에 의하여 프로시저 호출 위치에 모두 대체한다. Loop Extraction변환 방법은 루프에서 호출을 가진 프로시저에서 피 호출 프로시저의 루프를 프로시저의 호출 위치 외부로 이동하는 방법이다. Loop Embedding은 프로시저 호출을 포함하는 루프 헤더를 피 호출 프로시저로 이동하는 변환 방법이다. Procedure Cloning은 프로시저가 여러 차례 호출될 때 프로시저를 최적화한 프로시저로 복사하여 많은 프로시저가 호출하게 하는 변환 방법이다[3,4].

3. 병렬화를 위한 최적화

프로시저를 포함한 코드의 병렬화와 병렬성 증가를 위한 최적화 방법으로 inlining 후 최적화 방법과 최적화 후 inlining방법이 있다. inlining 후 최적화 방법은 호출 프로시저의 호출 위치들을 피 호출 프로시저의 코드로 대체하는 inlining을 수행한 후 inline된 코드를 프로시저 내 변환 방법을 사용하여 최적화하는 방법이다. 최적화 후 inlining방법은 각 프로시저 단위로 프로시저 내 변환 방법으로 최적화 한 후 inlining을 제외한 프로시저 간 변환 방법으로 최적화한다. 본 논문에서는 프로시저를 포함한 코드의 병렬화와 병렬성 증가를 위한 최적화 알고리즘을 서술한다.

<알고리즘 1>

1. 호출 멀티그래프 작성
2. 확장된 호출 그래프(augmented call graph)로 확장
3. 프로시저간 정보 계산
4. 종속성 분석
5. IF (하나의 프로시저가 한번 이상 호출되고 호출 매개변수 값이 변하지 않으면)
   THEN goto 알고리즘 3
   ELSE goto 알고리즘 4
6. 중간 코드에 자료 종속성 제거 알고리즘 대입 병렬 코드 생성

<알고리즘 2>

1. 초기화 과정
   S = 문장 수
   동적 기억장소 배열 DMA, DMB, DMC선언
   sw = 0
2. diophantine방식 계산 위해 GCD계산 함수 호출하여 pass=1인 $S*S$ 크기의 2차원 종속 행렬 DMB구함
   IF (인덱스 변수가 같으면) THEN rename
3. 중첩 루프의 인덱스 변수를 i, j라하고 최대치 N_1, N_2인 경우
   Forall DMB행렬의 i, j에 대하여
   IF $((a_{ij} \oplus u_{ij}) > N_1 \mid\mid (c_{ij} \oplus w_{ij}) > N_2)$
   THEN 그 원소는 0으로 치환
   IF sw == 0 THEN DMA = DMB, sw = 1
4. DMB 행렬 이용하여 0이 아닌 모든 원소에 대해
   Forall i, j에 대하여 IF $(u_{ij}$ & $w_{ij} == 1)$
   THEN uniform
   ELSE IF $(u_{ij}$ & w_{ij} != 1) THEN nonuniform
   ELSE complex형태의 doall S_i 문장으로 변환
5. IF 똑같은 원소가 존재한다면 THEN 하나만 남겨 놓고 제거
6. IF 모든 원소 = 0 THEN go to 8
   ELSE pass 증가하기 위해 행렬의 곱 이용하여
   $S*S^{pass+1}$ 크기의 종속행렬
   DMC = DMA * DMB 계산
   pass++
   DMB = DMC
   free DMC
   ENDIF
7. *Go to 3*
8. 변형된 문장을 제외한 모든 문장에 대해 doall S_i 문장으로 변환

<알고리즘 3>

/* 프로시저 내 변환 먼저 적용하고 inlining을 제외한 프로시저 간 변환 적용 후 inlining 적용 */
{

```
  1. repeat(각 프로시저가 최적화될 때까지)
     { /* 프로시저 내 변환 적용 */ }
  2. repeat(프로시저 간 최적화될 때까지)
     { /* inlining을 제외한 프로시저 간 변환 적용 */ }
  3. reverse topological order로 inlining 적용
}
```

```
<알고리즘 4>
/* inlining 후 프로시저 내 변환 적용 */
{
  1. reverse topological order로 inlining 적용
  2. repeat(각 프로시저가 최적화될 때까지)
     { /* 프로시저 내 변환 적용 */ }
}
```

4. 성능 평가

자료 종속성 거리가 uniform, nonuniform, complex형태의 그림 5의 예제 코드로 프로시저를 포함한 코드의 성능 평가와 프로세서 수 증가에 따른 병렬 코드의 성능평가를 하였다.

```
Procedure P
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
do i = 1, n1
 do j = 1, n2
   call Q
 enddo
enddo

Procedure Q
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
  a(i, j-7)=b(i-1, j-3)+c(i-9, j+5)
  b(i+2, j+8)=a(i-5, j+4)+c(i+6, j-4)
  c(i+1, j-6)=a(i-6, j+3)+b(i-2, j+3)
```

(a) uniform

```
Procedure P
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
do i = 1, n1
 do j = 1, n2
   call Q
 enddo
enddo

Procedure Q
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
  a(4i, 5j)=b(5i, 2j)+c(4i, 6j)
  b(6i, 4j)=a(3i, 2j)+b(4i, 3j)+c(5i, 3j)
  c(7i, 5j)=a(4i, 3j)+b(5i, 3j)
```

(b) nonuniform

```
Procedure P
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
 do i = 1, n1
  do j = 1, n2
    call Q
  enddo
 enddo

Procedure Q
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
  a[i-2][ j] = b[4*i][3*j];
  b[5*i][4*j] = a[3*i+3][j-4] + b[i-4][3*j] + c[3*i][4*j];
  c[7*i][5*i] = A[i-5][i-3];
```

(c) complex

그림 5 예제 코드 III

4-1. 프로시저를 포함한 코드의 성능 평가

자료 종속성 제거 방법을 이용한 프로시저 변환 알고리즘을 적용한 중간 코드와 loop extraction, loop embedding변환 방법을 그림 5 예제 코드를 사용하여 성능평가 하였다. 그림 6은 그림 5의 예제 코드(a)의 결과 코드를 보여준다.

```
Procedure P
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
do i = 1, n1
  do j = 1, n2
    a(i, j-7) = b(i-1, j-3) + c(i-9, j+5)
    b(i+2, j+8) = a(i-5, j+4) + c(i+6, j-4)
    c(i+1, j-6) = a(i-6, j+3) + b(i-2, j+3)
 enddo
enddo
```

(a) 알고리즘이 적용된 중간 코드

```
Procedure P
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
do i = 1, n1
  do j = 1, n2
    call Q
  enddo
enddo

Procedure Q
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
  a(i, j-7) = b(i-1, j-3) + c(i-9, j+5)
  b(i+2, j+8) = a(i-5, j+4) + c(i+6, j-4)
  c(i+1, j-6) = a(i-6, j+3) + b(i-2, j+3)
```

(b) Loop Extraction

```
Procedure P
real a(n1,n2), b(n1,n2), c(n1,n2)
  call Q

Procedure Q
real a(n1,n2), b(n1,n2), c(n1,n2)
integer i, j
 do i = 1, n1
   do j = 1, n2
     a(i, j-7) = b(i-1, j-3) + c(i-9, j+5)
     b(i+2, j+8) = a(i-5, j+4) + c(i+6, j-4)
     c(i+1, j-6) = a(i-6, j+3) + b(i-2, j+3)
   enddo
 enddo
```

(c) Loop Embedding

그림 6 uniform형태 코드

변환된 코드 그림 6을 성능 평가한 결과는 그림 7과 같이 이중 차트를 이용하여 표현하였다. 세 가지 방법 모두 프로시저 변환 알고리즘을 적용한 중간 코드가 가장 효율적이었고, loop extraction을 수행한 코드가 가장 비효율적임을 알 수 있다.

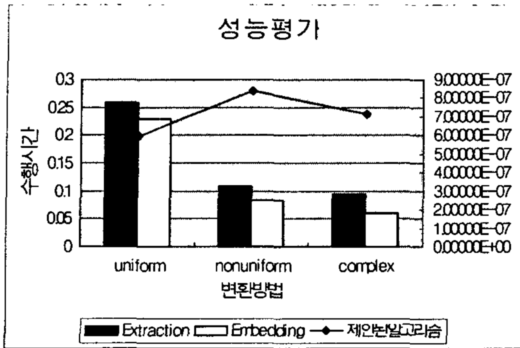

그림 7 프로시저를 포함한 코드의 성능평가

4-2. 프로세서 수 증가에 따른 병렬 코드의 성능평가

코드들을 종속성 제거 방법을 이용한 프로시저 변환 알고리즘을 사용하여 병렬 코드로 변환하였다. 데이터 수는 N_1은 20, N_2는 100으로 하고 프로세서 수는 2, 4, 8, 16, 32개를 사용하여 병렬 코드를 성능평가를 하였다. 프로세서의 수가 증가하면서 성능이 좋아짐을 알 수 있었다. 그리고 프로세서의 수를 증가하면서 성능평가 할 때 데이터 수가 적은 것에 비해 프로세서를 수를 많이 할당받으면 할당받고 작업을 하지 않는 경우가 발생하였다. 예를 들면 불변 종속거리인 경우 프로세서 32개를 할당받았을 때 21개의 프로세서만 사용하고 11개의 프로세서가 휴지 상태(idle state)가 발생했다.

프로세서 수를 증가하면서 프로세서 수 증가에 따른 병렬 코드의 성능평가는 불변 종속거리 형태가 가변 종속거리 형태나 혼합 종속거리 형태와 수행 시간 차이가 많이 나므로 그림 8과 같이 이중 차트를 이용하여 표현하였다.

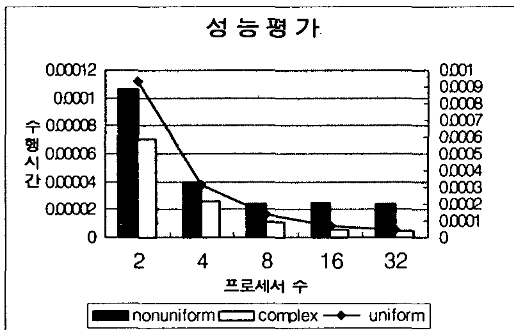

그림 8 병렬 코드의 성능 평가

5. 결론

자료 종속성 제거 방법을 이용한 프로시저 변환 알고리즘을 적용한 중간 코드와 loop extraction, loop embedding변환 방법을 적용한 예제 코드가 모두 프로시저 변환 알고리즘을 적용한 코드가 가장 효율적이었고 loop extraction을 수행한 코드가 가장 비효율적이었다. 프로세서 수 증가에 따른 병렬 코드의 성능평가는 프로세서 수가 증가하면서 비례적으로 성능이 좋아짐을 알 수 있었다.

자료 종속성 제거 방법을 이용한 프로시저 변환 알고리즘은 전역 지역(global area)에 적용하면 목적 코드와 컴파일 시간 증가와 같은 문제가 발생되므로 반복문 내부와 같이 부분 지역(local area)에 적당한 변환 방법이다.

참고문헌

[1] Dale Allan Schouten, *"An overview of Interprocedural analysis Techniques for High Performance Parallelizing Compilers"*, MS thesis, University of Illinois at Urbana-Champaign, 1986.

[2] Jr. Allen and K. Kennedy. *"Automatic loop interchange"*. In Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6, June 1984.

[3] K. D. Cooper, M. W. Hall, and Linda torczon, *"An experiment with inline substitution"*, Software-Practice and Experience, June 1991.

[4] M. W. Hall, *"Managing Interprocedural Optimization"* PhD thesis, Dept. of computer Science, Rice University, 1991.

[5] M. W. Hall, Ken Kennedy, Kathryn S. Mckinley. *"Interprocedural Transformations for Parallel Code Generation"*, Technical Report 1149-s, Dept. of computer Science, Rice University, 1991.

[6] Z. Li aand P. C. Yew, *"Efficient interprocedural analysis for program restructuring for parallel programs"*. In Proceedings of the SIGPLAN:Experience with Applications, Languages and Systems, 1988.

[7] Michael E. Wolf, *"Improving Locality and Parallelism in Nested Loops"*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, August, 1992.

[8] Hans Zima, *"Supercompilers for Parallel and Vector computers"*, ACM press, 1990.

[9] 송월봉, 박두순, *"중첩루프에서 병렬화를 위한 종속성 제거"*, 한국정보처리학회 논문지, Vol. 5, No. 8, Aug. 1998.

[10] 안준선, 최광훈, 김성훈, 한태숙, 최광무, *"병렬화 컴파일러의 소개"*, 정보과학회지 제14권 제7호, 1996