논문
8-3-3

루프인터체인지 병렬컴파일러 구현
(A Implementation of Loop Interchange Parallel Compiler)

송월봉(Worl-Bong Song)[1)]

요 약

일반적으로 응용프로그램에서 병렬성 추출에 대한 핵심 부분은 루프이다. 따라서 본 논문에서는 Loop Interchange를 자동으로 처리할 수 있는 병렬컴파일러를 구현하고자한다. Loop Interchange는 반복문에서 cedar Fortran의 CDOALL문장을 바깥쪽으로 보냄으로서 특히 더 많은 병렬화 효과를 얻을 수 있기 때문이다. 이는 향후 선형변환과 혼합하여 더욱 효과적인 실행 결과를 기대하고 불완전 중첩루프에 적용하는 방법의 연구에 기여할 것으로 판단된다.

Abstract

Generally, In a application program the core part for parallel processing is a loop. therefore in this paper, loop interchange parallel compiler is proposed. this is a procedure for the automatic conversion of a loop interchange. According to execution to the outside CDOALL statements of cedar fortran, loop interchange is more effectively method the extracting parallelism in order to parallel processing in iterations. This method will be expected to effectively execution result with mixed into linear conversion and go far toward solving the effectively implementation of the non-unimodular nested loop.

논문접수 : 2007. 5. 15.
심사완료 : 2007. 6. 10.

1) 정회원 : 시립인천전문대학 컴퓨터정보과 교수

본 논문은 인천전문대학 연구지원비에 의한 것임

1. 서 론

병렬화 컴파일러는 일반 컴파일러의 전 단계로서 기존의 순차 프로그램에서 묵시적인 병렬성을 탐지해내고 그를 병렬처리 시스템에서 효율적으로 처리하도록 함으로써 컴퓨터의 성능을 높이는 방법이다. 병렬화 컴파일러를 만드는 것은 미국의 Illinois대학처럼 기술 축적이 많이 되어 있고 연구원이 많은 경우에도 다년간 걸리는 힘들고, 비용이 많이 들어가는 일이지만 개방화와 국제경쟁력을 고려해 볼 때 우리 나라가 정보화와 세계화에서 경쟁력을 갖기 위해서는 꼭 필요로 하는 시스템 소프트웨어인 병렬화 컴파일러의 개발은 매우 필수적인 사안이며 이런 시스템 소프트웨어를 가격 경쟁 면에서 뒤진다고 하여 개발을 하지 않을 때 우리 나라는 경쟁력을 모두 잃고 외국의 컴퓨터 시스템을 모두 사들여야 할 것이다. 이런 상황은 정보화 사회에서 우리나라가 선진국의 속국을 벗어나기가 어렵다고 판단된다. 이런 의미에서 병렬화 컴파일러의 개발은 반드시 필수적이라고 하겠다.

본 논문에서는 병렬컴파일러의 생성과정을 토대로 Loop Interchange를 자동으로 할 수 있는 병렬컴파일러를 구현하고자 한다.

2. 병렬화 컴파일러의 생성과정

병렬화 컴파일러는 순차 프로그램을 입력으로 하여 코드 분석 과정을 거쳐 병렬 언어 프로그램이나 메시지 전달 인터페이스(massage passing interface)[8], 병렬 스레드(thread)와 같은 병렬 라이브러리를 호출하는 프로그램을 생성한다. 병렬화 컴파일러는 일반적으로 다음과 같은 과정으로 수행된다. 입력된 순차 프로그램의 어휘 및 구문에 대한 분석들이 전단부에서 점검되는 기존 컴파일러의 문제와 같다. 전단부를 거친 입력 프로그램들은 중간코드를 출력하며, 출력된 중간코드는 분석부에 입력되어 프로그램의 최적화와 병렬 프로그램의 생성에 필요한 제어 흐름 및 자료 흐름 분석과 종속성 분석(dependence analysis)이 수행된다. 이러한 프로그램 분석은 프로시져내 분석과 프로시져간 분석으로 구분되며, 주로 프로그램 명령어들 사이에 수행 순서 관계와 프로시져간의 호출 관계를 분석하는 제어 흐름(control flow)분석, 자료의 정의 및 사용 관계와 프로시져를 호출할 때 자료 전달에 관한 다양한 정보를 분석하는 자료 흐름(data flow)분석 그리고 배열(array) 원소 첨자(index)를 분석하여 루프 내의 배열 요소로 인한 의존성 관계에 대한 분석을 수행한다[3]. 이 결과를 바탕으로 병렬 프로그램 생성을 위한 코드 변환이 수행된다. 코드 변환 작업에서는 주로 벡터 연산 생성(vectorization)과 병렬 루프 생성(parallelization)등의 다양한 프로시져 변환 작업[3]을 거쳐 목적 기계(target machine)에 적합한 병렬 프로그램을 생성하게 된다. 생성된 병렬 프로그램은 병렬 컴퓨터에서 P개의 프로세서에 의해 동시에 병렬로 스케쥴링이 된다.

이러한 병렬화 컴파일러는 다음과 같은 유용성이 있다.

- 이미 개발되어 사용되는 순차 프로그램을 병렬 컴퓨터에서 신속하게 사용할 수 있다.
- 프로그래머가 병렬 프로그램 대신 순차 프로그램을 작성하도록 함으로서 프로그래머의 부담을 덜 수 있다.
- 병렬 프로그램들은 실행될 병렬 컴퓨터의 구조에 따라 병렬 수행 구조나 고려할 사항이 다르기 때문에 병렬 컴파일러를 사용함으로써 프로그램의 이식성(portability)을 높일 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 Illinois 대학의 Parafrase I, II[6], Rice대학의 PFC[2], IBM의 PTRAN[1], Kuck and Association의 KAP, Pacific Sierra의 VAST, Bonn 대학의

SUPERB[5], Stanford대학의 SUIF[9], Illinois 대학의 Polaris[4]등이 있다. 국내의 연구로서는 주전산기 IV 개발 사업의 일부로서 Fortran을 위한 병렬화 컴파일러가 있다.

3. 병렬 컴파일러 구현

본 장에서는 2장에서의 병렬컴파일러의 생성 과정을 토대로 loop interchange를 자동으로 할 수 있는 병렬 컴파일러를 구현하여 본다.

Parafrase-2에서는 기본적으로 병렬화 코드보다 벡터화 코드를 우선시하여 코드를 생성하므로, 중첩된 루프에서 병렬화가 가능한 루프는 기본적으로 안쪽에 위치하여 코드가 생성된다. 만약 Parafrase-2를 사용할 때, 병렬 가능한 루프를 바깥쪽에 위치하게 하여 병렬화를 우선시하는 코드를 생성하기 위해서는 loop interchange를 하여야 하며, Parafrase-2에서는 이러한 기능을 유틸리티 형태로 제공된다.

본 장에서는 이러한 기능을 개선하여, 병렬화를 우선시하여 병렬 코드를 생성하는 컴파일러를 구현하였다.

이를 위해서 Parafrase-2에서 쓰인 여러 패스들 중, 다른 패스들은 그냥 사용하고 병렬 코드를 생성하는 패스인 codegen 패스에 loop interchange와 연관된 함수를 추가하여 구성하였다.

Loop interchange는 초기에 traverse_modules() 함수를 호출한다. 이때 인수로 test_all_for_swap()를 호출하는데, 여기에 연관된 함수는 크게 3가지다.

첫째, ok_to_interchange로써 바깥쪽 반복문(lout)과 안쪽 반복문(lin)이 서로 안전하게 바뀔 수 있는지, 즉 종속성이 없는지를 검사하여 바뀔 수 있으면 TRUE값을 반환하고, 바뀔 수 없으면 FALSE값을 반환한다.

이 함수가 호출하는 함수는 nest_distance, this_nest_level, gen_intrchg_info, free_intrchg이며, 이 함수를 호출하는 함수는 test_all_fro_swap, interchange_2_loops이다.

두 번째 함수로는 interchange_2_loop 함수로써 바깥쪽 반복문인 'lout'와 안쪽 반복문인 'lin'을 바꾸는 것을 시도한다. 이 두 반복문은 반드시 완전 중첩 반복문이어야 한다. 만약, 두 반복문사이에 종속성이 없으면 두 반복문은 바꾸고 TRUE값을 반환하고, 그렇지 않으면 FALSE값을 반환한다.

마지막으로 연관된 함수는 do_interchange이며, 이 함수는 종속성을 검사하지 않고, 완전 중첩된 두 반복문을 바꾸어 준다.

이 함수가 호출하는 함수는 in_perfect_nest, rectangular_loop, triangular_loop이며, 이 함수를 호출하는 함수는 interchange_2_loops, test_all_for_swap이다.

[그림1]과 [그림3]은 loop interchange 시에 주요 코드를 보여준다.

```
int call_codegen(p2_module, s)
module_t *p2_module;
char *s;
{
 module_t *modls;
 int lineflag;
 char *getnext();
 int flag;
 char ch;
 char *t;
DB0(5,"BEGIN Loop Interchange₩n");
traverse_modules(P2_module,FALSE,NULL,test_all
_for_swap);
DB0(5,"END Loop Interchange₩n");
 printf("₩n");
 if (Opus == F77) {
    printf("GENERATING FORTRAN SOURCE₩n");
    Print_IMPLICIT_NONE = 1;
 } else if (Opus == KNRC) {
    printf("GENERATING C SOURCE₩n");
 } else {
    yyerror("What language are you using?");
 }
 modls = p2_module;
 lineflag = FALSE;
 getnext(s, &ch, &flag);
 while ((t = getnext(NULL, &ch, &flag)) !=
NULL){
    if(flag){
      switch(ch){
      case 'l':
        lineflag = TRUE;
        break;
      case 'i':
         if (Opus == F77) Print_IMPLICIT_NONE =
atoi(t);
      default:
        break;
      }
    } else {
      break;
    }
 }
 while (modls != NULL) {
    gen_code(p2stdout,modls, lineflag);
    modls = T_NEXT(modls);
 }
 fflush(p2stdout);
 return 0;
}
```

[그림1] loop interchange시 주요 소스코드

[Fig.1] The main source code when the loop interchange is converted

[그림1]에서 traverse_modules()함수를 호출함으로써 loop interchange가 진행된다. 이때 인수로 쓰인 test_all_for_swap()의 원시코드는 [그림 3]과 같다.

```
DO  I = 3,  N1
    DO J = 5, N2
        A(I, J) = B(I-3,  J-5)
        B(I, J) = A(I-2,  J-4)
```

[그림 2] 불변거리에 대한 예

[Fig. 2] The example of uniform

```
static void test_all_for_swap(f)
function_t    *f;
{
    statement_t *I1, *I2;
    DB1(10,"test_all_for_swap(%x)₩n",f);
    fprintf(p2stdout,"testing `%s' for potential loop
interchange₩n", S_IDENT(T_FCN(f)));
    for (I1=T_FIRSTDO(f); I1 && T_NEXTDO(I1); ) {
    I2 = T_NEXTDO(I1);
    while (I2 && in_perfect_nest(I1,I2)) {
        if (OK_to_interchange(I1,I2)) {
    fprintf(p2stdout,"swap loops lines %d and %d₩n",
        T_LINENO(I1), T_LINENO(I2));
if (manual && inquire(I1,I2)) {
        if (!do_interchange(I1, I2)) {
        WARN2("loops %d & %d not swapped",
    T_LINENO(I1), T_LINENO(I2));
    }
    }
    } else {
fprintf(p2stdout,"cannot swap loops lines %d and
%d₩n",
        T_LINENO(I1), T_LINENO(I2));
    }
    I2 = T_NEXTDO(I2);
}
I1 = T_NEXTDO(I1);
}
}
```

[그림 3] test_all_for_swap() 원시 코드

[Fig. 3] The source code of test_all_for_swap()

```
DIMENSION A(5:20,3:10)
  DIMENSION B(5:20,3:10)
    do 1200 i = 5, 20
    do 1200 j = 3, 10
```

```
        A(i,j) = B(i-3, j-5)
        B(i,j) = A(i-2, j-4)
    1200  continue
end
```

[그림 4] 반복 순차 프로그램 예
[Fig. 4] The example of iteration seq. pro.

```
IMPLICIT NONE
      REAL a(5:20,3:10)
      REAL b(5:20,3:10)
      INTEGER i, j
      CDOALL 1200 j = 3,10
        integer i
        DO 1200 i = 5,20
          a(i,j) = b(i - 3,j - 5)
          b(i,j) = a(i - 2,j - 4)
       1200   CONTINUE
END
```

[그림 5] [그림 2] 및 [그림 4] 예제를 이용한 병렬화 코드
[Fig. 5] The converted code into parallel for Fig.2 and Fig. 4

[그림 5]는 [그림 2] 및 [그림 4] 반복 순차 프로그램 예를 이용하여 loop interchange후에 개선되어 나온 병렬코드를 보여준다. 내용에서와 같이 병렬로 처리 가능한 CDOALL문장이 중첩 루프의 바깥쪽에 위치함을 알 수 있다. 이렇게 병렬 수행이 가능한 루프를 중첩 루프의 바깥쪽에 끌어냄으로써 스케줄링의 용이성과 더 나은 로드 밸런스를 이끌어 낼 수 있다.

Parafrase Ⅱ는 반복문 교환을 패스에서 하도록 되어있으나, 본 논문에서는 반복문 교환 작업이 코드를 생성할 때 자동으로 하도록 한다. 반복문 교환은 중첩 반복문을 서로 바꿈으로써 반복문을 변형시키는 것이다. 이것은 다음과 원인으로 수행된다. 병렬화를 극대화시키기 위하여 반복의 횟수가 많은 반복문을 밖으로 내보내거나, 적은 병렬화 즉, 벡터 머신에 적당하게 끔 반복 횟수가 많은 반복문을 안으로 내보내거나, 주어진 아키텍처에 알맞게 메모리 접근 패턴을 바꾸기 위함이다.

본 논문에서는 반복의 횟수가 많은 반복문을 밖으로 내보내어서 벡터화보다는 병렬화를 하게끔 만들어 보았다.

4. 결론 및 향후 과제

병렬화를 위해서 기존의 Parafrase Ⅱ를 갱신하여 간단하게 Loop Interchange를 자동으로 할 수 있는 병렬 컴파일러를 구현하였다. 이러한 Loop Interchange는 반복문에서 CDOALL 문장을 바깥쪽으로 보냄으로써 더 많은 병렬 효과를 주는 것을 확인하였다.

연구결과에 대한 효과로는 병렬성 제거 알고리즘을 선형변환과 혼합하여 더욱 효과적인 실행결과를 기대하는 것과 불완전 중첩루프에 적용하는 연구에 기여할 것으로 보이며,

향후 과제로는 제안된 Loop Interchange 병렬 알고리즘을 토대로 하여 서로 다른 프로세서의 성능 및 반복들의 전송 시간을 반영함으로서 공유메모리 시스템에서뿐만 아니라 분산 메모리 시스템에서도 효율적인 스케쥴링을 달성할 수 있는 알고리즘에 대한 연구가 필요할 것으로 판단된다.

참고문헌

[1] Shen, Z., Z. Li, P.C. Yew, "An Emperical Study on Array Subscripts and Data Dependencies," *Proc. of International Conference Parallel Processing*, pp. 145-152, Aug. 1989

[2] Wolf, M.E. and M.S. Lam, "A Loop Transformation theory and an Algorithm to Maximize Parallelism." IEEE Trans. on Parallel Distributed Systems, Vol. 2, No. 4, pp 452-471, Oct. 1991

[3] BBN Advanced Computers Inc., "Prorgramming in Fortran with the uniform system," Cambridge, MA, Nov., 1988

[4] Blume, B. et. al., "Polaris : The Next

Generation in Parallelizing Compilers," Proceedings of the Workshop on Languages and Compilers for Parallel Computing, 1994
[5] Cosnard, M., K. Ebcioglu, J. Gaudiot, "Architectures and Compilation Techniques for Fine and Medium Grain Parallelism", North-Holland, 1993
[6] Kuck, D.J., E.S. Davidson, D.H. Lawrie, and A.H. Sam도, "Parallel supercomputing today and the CEDAR approach," *Science*, 231, Feb., 1986
[7] Polaris Home Page, http://csrd.uiuc.edu/polaris/polaris.html
[8] Allen, F., M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," *Journal of parallel and distributed computing*, vol. 5, No. 5, Oct. 1988
[9] Allen, J.R. and K. Kennedy, "PFC:A program to convert Fortran to parallel form," Tech. Rept. MASC-TR82-6, Rice University, Houston, Texas, Mar., 1982

송월봉
1974년 숭실대 공학사
1982년 한양대 공학석사
1998년 순천향대학교 공학박사(전산학)
1978년~현재 시립인천전문대학 컴퓨터정보과 교수
관심분야 : 병렬처리컴파일러, 알고리즘