The Journal of Natural Science,
Pai Chai University, Korea
Volume 5(2):39-43, 1992

Parallel Machine에 있어서의 Functional, Declarative 언어의 Algorithm

金 珍 秀
배재대학교 국제산업대학 전자계산학과.

Algorithm for Functional and Declarative Language in Parallel Machine

Jin-Soo Kim
Dept. of Computer Science, Pai Chai University

사용자의 관점에서 볼때는 compiler가 parallelism을 발견할 수 있게 하는것이 매우 요구되지만, 아무리 잘 만들어진 compiler 라 할지라도 conditional, functional 또는 I/O statement 등 프로그램내에 존재하는 많은 parallelism을 인식 하기가 결코 쉬운 일이 아니다.

경우에 따라서는 compiler의 parallelism 결정이 곤란할경우 사용자에게 feedback 시키는 경우가 종종 있다. 이런 입장에서 프로그래머는 parallelism을 바로 전달하기 위해서 프로그램의 일부를 재구성 시킬 필요가 자주 발생한다.

그러한 관점에서 Functional, Declarative 언어의 잇점이 있다고 할 수 있고, 그러기 위해서는 parallel machine에 적합한 Algorithm 이 필요하다. 그러나, 이미 사용중인 Algorithm이 Parallel Machine에 부적절 하다는 것을 의미 하는것은 아니다.

본 연구에서는, Fortran을 이용하여 Parallel Algorithm을 구현 시키기위한 Declarative 언어에 있어서 Array 및 Matrix 를 다루기위한 Abstraction 방법을 제시 하고자 한다.

Detection of parallelism by a compiler is very desirable from a user's point of view. However, even the most sophisticated techniques to detect parallelism trip on trivial impediments, such as conditionals, function calls, and input/output statements, fail to detect most of the parallelism present in a program.

Some parallelizing compilers provide feedback to the user when they have difficulty in deciding about parallel execution. Under these circumstances, a programmer has to restructure the source code to aid the detection of parallelism.

But, functional and declarative languages can be said to offer many advantages in this context. Functional programs are easier to reason about because their output is determinate, that is, independent of the order of evaluation. However, functional languages traditionally have lacked good facilities for manipulating arrays and matrices. In this paper, a declarative language called Id has been proposed as a solution to some of these problems.

keywords : parallelism, functional language, declarative language, parallel programming, array, matrix

서 론 : Declarative 언어의 배경

a) *Functions.*

2개의 수를 합하는 Id 형태의 *function*으로 표현하면 다음과같다.

```
add (i,j) = i + j;·····(1)
```

declarative 언어에서는 프로그램의 기능적 의미가 rewrite rule로 표현될 수 있어야 하므로(모든 definition이 rewrite rule로서 받아들여질 수 있기때문) 위의 함수는,

```
c_add i j = i + j;·····(2)
```

와 같이 표현할 수 있다.

결국 *f(a)* 와 같은 표현 보다는 *f a* 로 표현하는것이 function application을 기술하기위한 functional language형태이다.

또한 문장에서 "left associative" 로 간주하게 하므로써 괄호를 없앨 수 있다. 따라서 아래의 모든 표현은 동일하다.

```
fab   (fa)b   ((fa)b)   f(a)b
```

여기서, 위의 *add* 와 *c_add*의 차이점은 매우 중요한 의미를 갖는다.

add function은 한개의 argument (tuple이 2개임) 인데 반해서 *c_add*는 2개의 argument를 갖는점이다. 결국 functional language 입장에서보면 *add*는 arrity가 1 일때 *c_add*는 arrity가 2 이다. 이러한 의미의 차이는 다음과같은 expression이 binding 되는 경우를 예로 생각 해 볼 수 있다.

```
successor = c_add 2;·····(3)
```

successor 나 *c_add 2* 는 동일하게 어떤 수에 2 를 합하는 function으로서 *successor n* 은 *c_add 2 n* 과 같은 수행을 하게된다.

b) *Tuple structures.*

자료구조로서의 *tuple*은 *Id*에 있어서 콤마(comma)를 이용하여 *tuple*을 형성하게되고 가령, *(1, n), (1, n)* 은 *2*개의 *tuple*을 가진 하나의 *tuple*이 된다.

```
grid n = ((1,n), (1,n));·····(4)
```

위의 function은 n 을 argument로 받아서 n * n matrix를위한 bound를 나타내는데 사용될 수 있다.

c) *Blocks.*

Declarative 언어에서 *block*은 하나의 binding set을 의미한다. *block*은 common subexpression의 계산을 분할하여 사용하는데 편리한 기능을 제공한다.

```
acceleration node =
        ( d = nodal_mass node;^·····(5)
          n1 = -(line_integral p z node)
               -(line_integral q z node);
          n2 =  (line_integral p r node)
               +(line_integral q r node);
      in  n1/d, n2/d );

          line_integral, nodal_mass : function
          p, q, r, z : constants
```

위의 *block*은 *d, n1, n2*의 3개의 binding을 갖고있다.

또한 compiler에 의해서 위의 definition (5) 는 아래와 같이 변형될 수 있을것이다.

```
acceleration' p q r z node =
        ( d = nodal_mass node;
          n1 = - (line_integral p z node)
               - (line_integral q z node) ;
          n2 = (line_integral p r node)
               + (line_integral q r node);
      in  n1/d, n2/d );

acceleration = acceleration' p q r z;
```

d) *I-structures.*

*I-structure*는 원래 parallel programming을 용이하게 하기 위해 설계되어진 것으로 run time시에 한번만 allocate되는 특별한 종류

의 배열(array) 이다. expression ***array***(l, u) 는 l 과 u 의 index bounds 를 가진 "empty array"를 allocate 한다.

array *(l, u)* => *⟨al, al+1, ... , au⟩*

각 ai 는 memory location을 나타내며, *⟨al, al+1, ... , au⟩*은 *I-structure Value* 를 나타낸다. 그리고 *I-structure* 의 각 element 들은 또다른 *I-structure*를 포함할 수 있다. 따라서, 여러개의 array를 가진 하나의 array로서의 matrix를 정의할 수가 있다.

e) loops.

concurrently 하게 반복 실행 시키기위한 하나의 binding set이다.

```
( A = array 1, 10;·····(6)
    ( for i from 1 to 10 do
        A(i) = i * i )
 in A )
```

loops 자체도 하나의 value를 return 시킬 수 있으며 그러한 loop에서의 binding set은 keyword인 ***finally*** 에 의해 가능하다. 또한 모든 순환과 return expression들은 concurrently 하게 initiate 된다. keyword ***next***는 i 번째 순환에서 계산된 값으로 (i+1) 번째에서 다시 이용할 수 있게한다. 다음의 예에서 keyword ***finally*** 는 loop expression에 의해서 반환된 값이 마지막 순환시 s 에 할당됨을 가리킨다.

```
{ s = 0;                    => s = 0
  in { for i from 1 to 10 do   s1 = s+1 * 1
       next s = s+i * i        s2 = s1+2 * 2
    finally s } }              ···
                               s10 = s9+10 * 10
```

Array 및 *Matrix*를 위한 *Algorithm.*

위의 definition (6)에서의 Array ***A***는 유한영역 (1, 10)에 대한 function *quare i = i * i;* 의 효과적인 표현으로 생각해볼 수 있다. array ***A***는 unction *square*를 마치 "cache" 형태로 취급한다.

아래의 abstraction은 그러한 관점을 보여준다.

```
make_array (l, u) generate = ····· (7)
                    { A = array (l, u);
                      { for i from l to u do
                        A(i) = generate i }
                    in A };
make_matrix ((l1, u1), (l2, u2)) generate = ··· (8)
                  { A = matrix ((l1, u1), (l2, u2));
                    { for i from l1 to u1 do
                      { for j from l2 to u2 do
                        A(i, j) = generate (i, j) } }
                  in A };
A = make_array (1, 50) successor; ····· (9)
B = make_matrix (grid 50) add; ····· (10)
```

definition (8)과 (9)에서, *generate* 는 그와 연관된 구조를 가진 elements를 생성시키는 함수이다. 예를들어 definition (9)는 i번째 요소가 (i+1) 을 가지는 50개의 element인 array를 구축하는 것이다. 동시에 definition (10)은 50 * 50 matrix를 생성한다. ((1)의 add function을 사용)

Matrix Abstractions in Fortran

abstraction을 기술 하기 위해 필요한 dynamic storage allocation 과 같은 feature 가 Fortran에는 없으므로 problem 내의 parallelism을 포착하여 Fortran 에서 구현 시키는 것을 살펴 보겠다.

make_matrix abstraction 은 Fortran으로 다음과 같이 기술할 수 있지만 몇가지 문제점은 있다.

```
      subroutine make_matrix(A, r1, r2, c1, c2, f)
      integer  r1, r2, c1, c2
      dimension  A(r2, c2)
      do  10  i = r1,  r2
      do  10  j = c1,  c2
   10 A(i, j) = f(i, j)
      return
      end
```

우선 이 프로그램이 (8)의 abstraction과 상이한 부분을 요약하면 다음과 같다. 첫째로, 'dynamic storage allocation' 기능이 없이는 이 subroutine으로 새로운 matrix를 생성할 수 없으며, 단지 argument로 전달받은 matrix를 채울 뿐이다. 둘째로, Fortran 에서의 function은 하나의 scalar값만을 return 받을 수 있다는 제약이 있다는 점이다. 사실 위의 프로그램에서의 ***f*** 는 function 이름으로써 subroutine의 argument로 passing했다고 가정한 것이지만 그 자체도 Fortran에서는 가능하지 않다는점이 문제다.

위에서와같은 문제점을 보완하여 프로그램을 다시 작성하면 다음과 같다.

```
      subroutine make_2_matrices(A,B,r1,r2,c1,c2,f)
      dimension  A(r2,c2), B(r2,c2)
      do  10  i = r1, r2
      do  10  j = c1,c2
   10 call  f(i,j,A(i,j),B(i,j))
      return
      end
```

parallelization

Fortran 프로그램의 parallel execution 을 위한 시스템은 많이 소개되어 있다. 예를들어, 위의 subroutine make_matrix에서 'do' 대신에 doall 로 교체함으로써 두개의 loop가 parallel하게 실행되어질 수 있다.

doall은 loop내의 모든 순환을 parallel하게 실행되도록 지시하는것이고, 또다른 form인 barrier 는 모든 processor들이 각각의 처리를 끝내고 그 시점에 도달 할때까지 다음 step으로 진행하지않고 기다리는것을 지시한다.

```
      subroutine sequential_wavefront(A,n)
      dimension  A(n,n)
      do  10  j = 1, n
      A(1,j) = 55.5
   10 A(j,1) = 55.5
      do  40  i = 2, n
      do  40  j = 2, n
   40 A(i,j) = A(i-1,j) + A(i,j-1)
      return
      end
      subroutine wavefront_matrix(A,n)
      dimension  A(n,n)
      doall  10  j = 1, n
   10 A(1,j) = 55.5
      doall  20  i = 2, n
   20 A(i,1) = 55.5
      barrier
      do  45  m = 4, n+1
      doall  40  i = 2, m-2
      j = m - i
   40 A(i,j) = A(i-1,j) + A(i,j-1)
   45 barrier
      do  55  m = n+2, n+n
      doall  50  i = m-n, n
      j = m - i
   50 A(i,j) = A(i-1,j) + A(i,j-1)
   55 barrier
```

위의 예에서, 원래의 algorithm에 대한 parallelism을 관찰하여 프로그램을 restructuring시킨후 이것을 verify한다는것이 쉬운일이 아니다. 가령 "do **40**" 대신 doall로 표현했듯이 "do **45**"도 그렇게 한다든가, barrier를 한 statement 아래로 내린다든가하면 예기치 못하는 결과를 얻게될것이기 때문이다.

결 론

가장 이상적인것은 모든 high-level language들이, 가능하면 사용자로 하여금 문제의 영역에 접근한 abstraction을 쉽게 기술할 수 있는 feature를 제공해주는 것이지만 전술한 바와 같이 여러가지 어려움이 있다.

본 연구에서 제시한 Functional, Declarative 언어에서는 다소나마 그러한 문제점에대한 해결의 실마리를 제공한것이라 생각된다. 또한 Fortran에서 abstraction을 표현하기 어려운 근본적인 이유는 dynamic storage allocation의 기능이 없다는 점 이었다. 따라서, function이나 subroutine에서 array를 allocate 시켜 놓고 그것을 결과로서 return 시킬 수 없는 것이다. 대조적으로 declarative 언어 에서는 parallelism을 채아내는 어려움은 다소 줄어 들었지

만, parallel execution을 위한 자원들을 다루는 문제는 여전히 남아있다고 볼 수 있다.

이러한 각 언어들이 갖고 있는 문제점들을 보완 시키기위한 algorithm 을 계속적으로 연구하면 새롭고 보다 나은 방법을 채을 수 있으리라 본다.

참 고 문 헌

1. ARVIND and Kattamuri Ekanadham, *Future Scientific Programming on Paral. lel Machines.* 1988

2. David Cann and John Feo, *SISAL versus FORTRAN: A comparison Using the Livermore Loops,* 1990

3. 1990 INTERNATIONAL CONFERENCE on PARALLEL PROCESSING, August 1990

4. Constantine D. Polychronopoulos, *Parallel Programming and Compilers,* 1988