

Second Paradigm. (simplest out of all but limited)

Data Parallel Programming

Jingke Li

Portland State University

Initially Created for Cray machines!

Programming Vector/SIMD Computers

↳ many operations happen at the same cycle.

Vector computers and SIMD computers have a common programming interface:

- ▶ a single stream of instructions → all operations must be the same
- ▶ each instruction specifies *uniform* operations over a set of data
- ▶ the operations are carried out in a synchronized step.

This style of computation is called *data parallel*.

Two Programming Approaches: (using these machines)

- Initially →
- ▶ Regular programs + smart compiler → usual approach due to high complexity.
 - compiler detects data parallelism in user program; converts them to vector form of data parallelism
- Later →
- ▶ Explicit data-parallel programs use additional constructs
 - requires language support that satisfy.

SIMD – requires all incoming instructions to be the same.

1. The Compiler Approach (used ^{but} mostly for Fortran)

Source code:

```
for i = 1 to n do
  a[i] = b[i] + c[i]
  if a[i] > 0 then
    b[i] = b[i] + 1
  endif
endfor
```

⇕

Vector code:

```
for i = 1 to n by 64 do
  VL = max(n-i+1, 64)
  vfetch b[i] -> v1
  vfetch c[i] -> v2
  vadd v1+v2 -> v3
  vstore v3 -> a[i]
  loadi #0 -> r1
  vcmp v3, r1 -> m1
  loadi #1 -> r2
  vaddc v1+r1 -> v1, m1
  vstorec v1 -> b[i], m1
endfor
```

Loop is partitioned
each out of 64
iteration will be
executed 64 times

Key compiler techniques:

- ▶ data dependence analysis
- ▶ loop transformations

Mature Compiler (At a time)
worked well, but not interested
for us!

Explicit Programming Approach (still welcome because user is not involved).

Gives more control than blind compiler approach.

Use simple, high-level constructs to specify data parallelism — uniform computations performed on all elements of a data domain.

- ▶ Key constructs: array operations and parallel loops

→ only these two
terms are required!

- ▶ Synchronizations are implicitly

- ▶ Communications are not allowed → no shared information (not allowed)

Languages: (That can qualify as data parallel)

- ▶ Fortran 90/95 → First wide spread language for Parallel
- ▶ OpenCL and Cuda → later approaches (only)
- ▶ Chapel → Newer (but supports all three paradigms)

One of the earliest languages, still widely used in 90 because of lack of features) **Fortran 90/95** (previously Fortran 77), was drastically redesigned

Goals: 1. Modernize 2. Introduce data parallel

Fortran 90 is a major extension to Fortran 77 for

- ▶ data-parallel style parallel programming → only array parallelization (but not) loops.
- ▶ language modernization (e.g. dynamic memory management, recursive functions, derived types, pointers, bit functions, etc.)

Fortran 77:

```
real sums
dimension sums(2,3)
do 20 j = 1, 3
  do 10 i = 1, 2
    sums(i,j) = 0.0
  10 continue
20 continue
```

initialize array to 0

Fortran 90:

```
= real,dimension(2,3):: sums = 0.0
```

Loops are added in Fortran 95.
↳ parallel loops.

More extended example Whole Array Operations

```
program simple
integer a, b, c, n, max
parameter (n=5)
dimension a(n), b(n), c(n) ! define three arrays,
data a /1,2,3,4,5/ ! array initialization
b = 2 ! scalar expansion and shape as b and move it to b.
c = a**2 + b**2 ! whole array operations
print *, 'array c contains:' !
print *, c
max = maxval(c) ! predefined in the language (intresit parallel function)
print *, 'the largest value in c is ', max

stop
end
```

initialize
↪
every element of a gets squared.

Operate on whole arrays!

Array Section Operations (Selection of separate array elements)

Triplet Notation: (lbound : ubound : stride) → subset

column row
`integer, dimension(5,5):: array` 5x5 array.
`integer, dimension(5):: row1, row2, row3, col5`
`integer, dimension(3,3):: inner`
`integer, dimension(25):: linear`
`integer, dimension(5):: diagonal`

Index for array starts with 1 not 0!

Triplet notation allows regular slicing of the arrays for uniform operation

`Row1 = array(1,1:5:1)`
`Row2 = array(2,1:5)`
`Row3 = array(3,:)`
`Col5 = array(:,5)`
`inner = array(2:4,2:4)`
`linear = pack(array, .true.)`
`diagonal = linear(1:25:6)` *Trick to get diagonal*

Array Functions

- ▶ *Array attributes:* lbound, ubound, shape, size

`real, dimension(-1:2, -3:4):: array` *query*
`print *, lbound(array)` ! -1 -3
`print *, ubound(array)` ! 2 4
`print *, shape(array)` ! 4 8
`print *, size(array)` ! 32 *array properties.*

- ▶ *Rearranging elements:* reshape, pack, unpack

`integer, dimension(3,3):: array`
`= reshape((/ 1,4,7,2,5,8,3,6,9 /), (/ 3,3 /))` *Just a view*
`integer, dimension(9):: vector = (/ (i,i=1,9) /)` *Reshaping Arrays.*
`logical, dimension(3,3):: filter = .true.`
`integer:: missing = 0`
`filter(:,2) = .false.` ! [1 4 7] [1 0 7]
`filter(2,:) = .false.` ! [2 5 8] -> [0 0 0]
`array = unpack(vector, filter, missing)` ! [3 6 9] [3 0 9]

- ▶ *Intrinsic Functions:* (Built-in parallel functions)

– sum, product, max, min, any, all, count, matmul, spread, etc.

Motivation behind reshaping: F77 didn't allow dynamic arrays. Every subroutine had dozens of parameters, had to create global arrays. Each array could have a view which doesn't reshape the array but gives you a different view.

Conditional Operations

- ▶ The original "if" statement: Standard Fortran IF
 - cond is a scalar boolean expression

```
integer, dimension(10,10):: a, b, c
if (k > 5) then
    c = b/a
end if
```

how to apply it to single array element?
 scalar condition (single variable)

- ▶ The new parallel "where" statement:
 - cond is an array boolean expression

```
integer, dimension(10,10):: a, b, c
where (a .ne. 0)
    c = b/a
elsewhere
    c = -1
end where
```

True → False
 vectors, condition applied to all elements of an array.
 all true statements executed in 1st cycle and all the false in separate 2nd cycle

Prime-Finder Program Comparison

Fortran 77:

```
program prime
integer i, j, n
parameter (n=999)
logical primes(n)

do i = 1, n
    primes(i) = .true.
end do
primes(1) = .false.
do i = 1, int(sqrt(real(n)))
    if (primes(i)) then
        do j = i+i, n, i
            primes(j) = .false.
        end do
    end if
end do
do i = 1, n
    if (primes(i)) print *, i
end do
end
```

find all primes between 1 and 1000
 potential prime

Fortran 90:

```
program rimes
integer i, n, next
parameter (n=999)
logical, dimension(n):: primes
integer, dimension(n):: ident = (/ (i,i=1,n) /)

primes = .true.
primes(1) = .false.

next = 2
do while (next .lt. int(sqrt(real(n))))
    primes(next*2:n:next) = .false.
    next = minval(ident(next+1:n),
                    primes(next+1:n))
end do

print *, "Number of primes:", count(primes)
do i = 1, n
    if (primes(i)) print *, i
end do
end program
```

Declaration is almost the same.
 generate 1...n
 additional array.
 data parallel statement
 regular scalar statement
 not totally intuitive
 but doable

Test if prime → true cross out all multiples of the prime
 → repeat.

Fortran 90 Prime-Finder Example

- Initialization (assume $n=10$):

```
n = 10
primes = (f,t,t,t,t,t,t,t,t,t)
ident  = (1,2,3,4,5,6,7,8,9,10)
next = 2
```

- 1st iteration:

```
primes(4:10:2) = .false.
primes = (f,t,t,f,t,f,t,f,t,f)
next = minval((3,4,5,6,7,8,9,10), (t,f,t,f,t,f,t,f))
⇒ next = 3
```

- 2nd iteration:

```
primes(6:10:3) = .false.
primes = (f,t,t,f,t,f,t,f,f,f)
next = minval((4,5,6,7,8,9,10), (f,t,f,t,f,f,f))
⇒ next = 5 (> loop bound)
```

The Fortran 95 Forall Statement

The intent of the Forall statement is that its “iterations” can be executed *concurrently*. For example,

```
forall (i = 1:n)
  a(i) = b(i) + c(i)
end forall
```

The n iterations imply n concurrent threads. At the end of the loop, there is an implicit *barrier* synchronization to synchronize all n threads.

However, defining its semantics is not as easy as it appears. Consider:

```
forall (i = 2:n)
  a(i) = b(i) + a(i-1)
end forall
```

```
forall (i = 1:n-1)
  a(i) = b(i) + c(i)
  a(i+1) = a(i) + a(i+1)
end forall
```

Both contain inter-iteration dependencies. The second example is further complicated by having multiple statements in the body of the loop. How should they be executed?

forall Statement Semantics

- ▶ Only assignment statements, where statements, and nested forall statements are allowed inside a forall statement.
- ▶ *All* reads from a given assignment statement, in *all* iterations, must occur before *any* write to the left-hand side, in *any* iteration.
- ▶ The writes of the left-hand side must occur before *any* reads in the following assignment statement.

Back to the Examples

```
forall (i = 2:n)
  a(i) = b(i) + a(i-1)
end forall
```

- ▶ The value of $a(i-1)$ used in iteration i is the *old* value from before the loop execution starts; it is not the value produced in iteration $i-1$.

```
forall (i = 1:n-1)
  a(i) = b(i) + c(i)
  a(i+1) = a(i) + a(i+1)
end forall
```

- ▶ The value of $a(i)$ used in the second statement is the *new* value produced by the first statement of the same iteration.
- ▶ The value of $a(i+1)$ used in the second statement is the *old* value from before the loop execution starts.

Sequential Execution of Forall Statement

The forall statements are best implemented on a vector or a SIMD machine. However, there will be times they need to be executed sequentially, for instance, when there is not enough processors to match the number of iterations, or when we want to test and debug a parallel program on a sequential machine.

When running on a single processor, a forall statement can be expensive, because in general, it cannot be simply turned into a do loop. In other words, the following two loops may produce different results:

```
forall (i = 2:n)
  a(i) = b(i) + a(i-1)
end forall
```

```
do i = 2 to n
  a(i) = b(i) + a(i-1)
end do
```

The correct implementation calls for buffering a copy of the old values of a.