Orchestrating a brighter world

NEC

How to Use Fortran Compiler for Vector Engine

2nd Edition November 2019

\Orchestrating a brighter world

NEC brings together and integrates technology and expertise to create the ICT-enabled society of tomorrow.
We collaborate closely with partners and customers around the world, orchestrating each project to ensure all its parts are fine-tuned to local needs.

Every day, our innovative solutions for society contribute to greater safety, security, efficiency and equality, and enable people to live brighter lives.

Table of Contents

In this document, “parallel processing” stands for automatic parallelization of compiler or shared memory parallel processing with OpenMP Fortran.

NEC Fortran Compiler for Vector Engine

Product Name: NEC Fortran Compiler for Vector Engine

- Conformed Language Standards
  - ISO/IEC 1539-1:2004 Programming languages – Fortran
  - ISO/IEC 1539-1:2010 Programming languages – Fortran (Partially)
  - OpenMP Version 4.5
- Major Features
  - Automatic Vectorization
  - Automatic Parallelization and OpenMP Fortran
  - Automatic Inline Expansion

How to Use Fortran Compiler

Usage of Fortran Compiler

```
$ nfort -mparallel -O3 a.f90 b.f90
                 … Compile and link Fortran program(a.f90 b.f90)
```

-O4	… Automatic vectorization with the highest level optimization
-O3	… Automatic vectorization with high level optimization
-O2	… Automatic vectorization with default level optimization
-O1	… Automatic vectorization with optimization without side-effects
-O0	… No vectorization and optimization

High ↕ Low

Options to control the level of automatic vectorization and optimization.

-fopenmp	… Enable OpenMP Fortran
-mparallel	… Enable automatic parallelization

Parallelization controlling options.

Do not specify these options when you do not use shared memory parallelization.

Example of Typical Compiler Option Specification

`$ nfort a.f90`	Compiling and linking with the default vectorization and optimization.
`$ nfort -O4 a.f90 b.f90`	Compiling and linking with the highest vectorization and optimization.
`$ nfort -mparallel -O3 a.f90`	Compiling and linking using automatic parallelization with the advanced vectorization and optimization.
`$ nfort -O4 -finline-functions a.f90`	Compiling and linking using automatic inlining with the highest vectorization and optimization.
`$ nfort -O0 -g a.f90`	Compiling and linking with generating debugging information in DWARF without vectorization and optimization.
`$ nfort -g a.f90`	Compiling and linking with generating debugging information in DWARF with the default vectorization and optimization.
`$ nfort -E a.f90`	Performing preprocessing only and outputting the preprocessed text to the standard output.
`$ nfort -fsyntax-only a.f90`	Performing only grammar analysis.

Program Execution

```
$ nfort a.f90 b.f90
$ ./a.out
```

Executing a compiled program.

```
$ ./b.out data1.in
```

Executing a program getting input file and parameter from command line.

```
$ ./c.out < data2.in
```

Executing with redirecting an input file instead of standard input file.

```
$ nfort -mparallel -O3 a.f90 b.f90
$ export OMP_NUM_THREADS=4
$ ./a.out
```

Executing a parallelized program with specifying the number of threads.

```
$ env VE_NODE_NUMBER=1 ./a.out
```

Executing with number of VE.

Performance Analysis

Performance Information of Vector Engine

PROGINF

- Performance information of the whole program.
- The overhead to get performance information is slightly.

FTRACE

- Performance information of each function.
- It is necessary to re-compile and re-link the program.
- If functions are called many times, the overhead to get performance information and the execution time may increase.

PROGINF

Performance information of the whole program

```
$ nfort -O4 a.f90 b.f90 c.f90
$ ls a.out
a.out
$ export VE_PROGINF=DETAIL
$ ./a.out
         ********  Program  Information  ********
  Real Time (sec)                         :            11.329254
  User Time (sec)                         :            11.323691
  Vector Time (sec)                       :            11.012581
  Inst. Count                             :           6206113403
  V. Inst. Count                          :           2653887022
  V. Element Count                        :         619700067996
  V. Load Element Count                   :          53789940198
  FLOP count                              :         576929115066
  MOPS                                    :         73492.138481
  MOPS (Real)                             :         73417.293683
  MFLOPS                                  :         50976.512081
  MFLOPS (Real)                           :         50924.597321
  A. V. Length                            :           233.506575
  V. Op. Ratio (%)                        :            99.572922
  L1 Cache Miss (sec)                     :             0.010847
  CPU Port Conf. (sec)                    :             0.000000
  V. Arith. Exec. (sec)                   :             8.406444
  V. Load Exec. (sec)                     :             1.384491
  VLD LLC Hit Element Ratio (%)           :           100.000000
  Power Throttling (sec)                  :             0.000000
  Thermal Throttling (sec)                :             0.000000
  Max Active Threads                      :                    1
  Available CPU Cores                     :                    8
  Average CPU Cores Used                  :             0.999509
  Memory Size Used (MB)                   :           204.000000
```

Set the environment variable "**VE_PROGINF**" to "**YES**" or "**DETAIL**" and run the executable file.

- "**YES**" … Basic information.
- "**DETAIL**" … Basic and memory information.

Time information (Real Time, User Time, Vector Time)

Number of instruction executions (Inst. Count through MFLOPS (Real))

Vectorization, memory and parallelization information (A. V. Length through Memory Size Used)

FTRACE

Performance information of each function

```
$ nfort -ftrace a.f90 b.f90 c.f90          (Compile and link a program with -ftrace)
$ ./a.out
$ ls ftrace.out
ftrace.out                                  (At the end of execution, ftrace.out file is generated in a working directory)
$ ftrace                                    (Type ftrace command and output analysis list to the standard output)
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Mar 22 17:32:54 2018 JST
Total CPU Time : 0:00'11"163 (11.163 sec.)


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS    MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                     RATIO V.LEN      TIME    MISS     CONF HIT E.%

    15000     4.762( 42.7)     0.317  77117.2  62034.6  99.45 251.0     4.605   0.002    0.000  100.00 funcA
    15000     3.541( 31.7)     0.236  73510.3  56944.5  99.46 216.0     3.554   0.000    0.000  100.00 funcB
    15000     2.726( 24.4)     0.182  71930.2  27556.5  99.43 230.8     2.725   0.000    0.000  100.00 funcC
        1     0.134(  1.2)   133.700  60368.8  35641.2  98.53 214.9     0.118   0.000    0.000    0.00 main
------------------------------------------------------------------------------------------------------------------
    45001    11.163(100.0)     0.248  74505.7  51683.9  99.44 233.5    11.002   0.002    0.000  100.00 total
```

For an MPI program, multiple **ftrace.out** files are generated. Specify them by **-f** option.

```
$ ls ftrace.out.*
ftrace.out.0.0   ftrace.out.0.1   ftrace.out.0.2   ftrace.out.0.3
$ ftrace -f ftrace.out.0.0 ftrace.out.0.1 ftrace.out.0.2 ftrace.out.0.3
```

Notes of Performance Analysis

- In FTRACE, performance information is collected at the function entry/exit. So if many functions are called, the execution time would increase.

```
$ nfort -ftrace -c a.f90
$ nfort -c main.cpp b.f90 c.f90
$ nfort -ftrace a.o main.o b.o c.o
$ ./a.out
```

- Compile with "**-ftrace**" only the file contains the target function.
- Also specify "**-ftrace**" for linking.

- Performance information of functions in the files compiled without **-ftrace** are contained in that of the caller function.
- In FTRACE, performance information of the inlined functions are contained in that of the caller function.
- Performance information of system library functions
  - PROGINF result contains the performance information of system library functions called from a program.
  - FTRACE result contains the performance information of system library functions called from a program. They are included in the performance information of the caller function.

Debugging

Traceback Information

Compile and link with **-traceback**.

Set the environment variable "**VE_TRACEBACK**" to "**FULL**" or "**ALL**" at execution.

Set the environment variable "**VE_FPE_ENABLE**" to catch arithmetic exceptions.

- "**DIV**" ... Divide-by-zero exception
- "**INV**" ... Invalid operation exception
- "**DIV,INV**" ... Both exceptions

```
PROGRAM MAIN
REAL :: A, B
A = 1.0
B = 0.0
PRINT *, A/B
END
```

Note: "**VE_FPE_ENABLE**" can be set to any other value but traceback basically uses "**DIV**" or "**INV**".

PRINT *, A/B ← Occur "divide-by-zero"

```
$ nfort -traceback main.f90
$ export VE_TRACEBACK=FULL
$ export VE_ADVANCEOFF=YES
$ export VE_FPE_ENABLE=DIV
$ ./a.out
Runtime Error: Divide by zero at 0x6000000105d0
[ 1] Called from 0x600000010750
[ 2] Called from 0x7f8f41e307a8
[ 3] Called from 0x600000003700
Floating point exception
$ naddr2line -e a.out -a 0x6000000105d0
0x00006000000105d0
/.../main.f90:5
```

- `$ nfort -traceback main.f90` ← Compile and link with **-traceback**
- `$ export VE_TRACEBACK=FULL` ← Use traceback information
- `$ export VE_ADVANCEOFF=YES` ← Advance-mode is off
- `$ export VE_FPE_ENABLE=DIV` ← Catch exception of "divide-by-zero"
- Runtime Error … [3] Called from 0x600000003700 ← Traceback information
- `$ naddr2line -e a.out -a 0x6000000105d0` ← Specify where the exception occurs
- `/.../main.f90:5` ← Notice that divide-by-zero is occurring in the 5th line in the main.f90 file

Using GDB

Specify **-g** to the files including the functions which you want to debug, in order to minimize performance degradation

```
$ nfort -O0 -g -c a.f90
$ nfort -O4 -c b.f90 c.f90
$ nfort a.o b.o c.o
$ gdb a.out
(gdb) break sub
Breakpoint 1 at sub
(gdb) run
Breakpoint 1 at sub
(gdb) continue
...
```

- `$ nfort -O0 -g -c a.f90` ← Only a.f90 is compiled with **-O0 -g**(avoid performance degradation)
- `$ nfort -O4 -c b.f90 c.f90` ← The others are compiled without **-g**
- `$ gdb a.out` ← Run GDB

- When debugging without **-O0**, compiler optimization may delete or move code or variables, so the debugger may not be able to reference variables or set breakpoints.
- The exception occurrence point output by traceback information can be incorrect by the advance control of HW. The advance control can be stopped to set the environment variable **VE_ADVANCEOFF=YES**. But the execution time may increase substantially to stop the advance control. Please take care it.

Strace: Trace of System Call

```
$ /opt/nec/ve/bin/strace ./a.out
...
write(2, "delt=0.0251953, TSTEP".., 27)           = 27
open(“MULNET.DAT", O_WRONLY|O_CREAT|O_TRUNC, 0666)= 5
ioctl(5, TCGETA, 0x8000000CC0)                    Err#25 ENOTTY
fxstat(5, 0x8000000AB0)                           = 0
write(5, "1 2 66 65", 4095)                       = 4095
write(5, "343 342", 4096)                         = 4096
write(5, "603 602", 4096)                         = 4096
write(5, "863 862", 4094)                         = 4094
write(5, "1105 1104", 4095)                       = 4095
write(5, "1249 1313 1312", 4095)                  = 4095
write(5, "1456 1457 1521 1520", 4095)             = 4095
write(5, "1727", 4095)                            = 4095
...
```

System call arguments　　　　System call return values

- **Arguments and return values of system calls are output**
  - You can check if the system library has been called properly.
  - You should carefully select system calls to be traced by **-e** of **strace**, because the output would be so many.

Automatic Vectorization

Vectorization Features

- An orderly arranged scalar data sequence such as a line, column, or diagonal of a matrix is called vector data. Vectorization is the replacement of scalar instructions with vector instructions.

Execution image of scalar instructions

```
A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
A(3) = B(3) + C(3)
…
A(100)=B(100) + C(100)
```

A(1)	=	B(1)	+	C(1)
A(2)	=	B(2)	+	C(2)
		…		
A(100)	=	B(100)	+	C(100)

Execute one calculation 100 times

Execution image of scalar instructions

```
DO I = 1, 100
    A(I) = B(I) + C(I)
END DO
```

A(1)		B(1)		C(1)
A(2)	=	B(2)	+	C(2)
…		…		…
A(100)		B(100)		C(100)

Execute 100 calculation at once

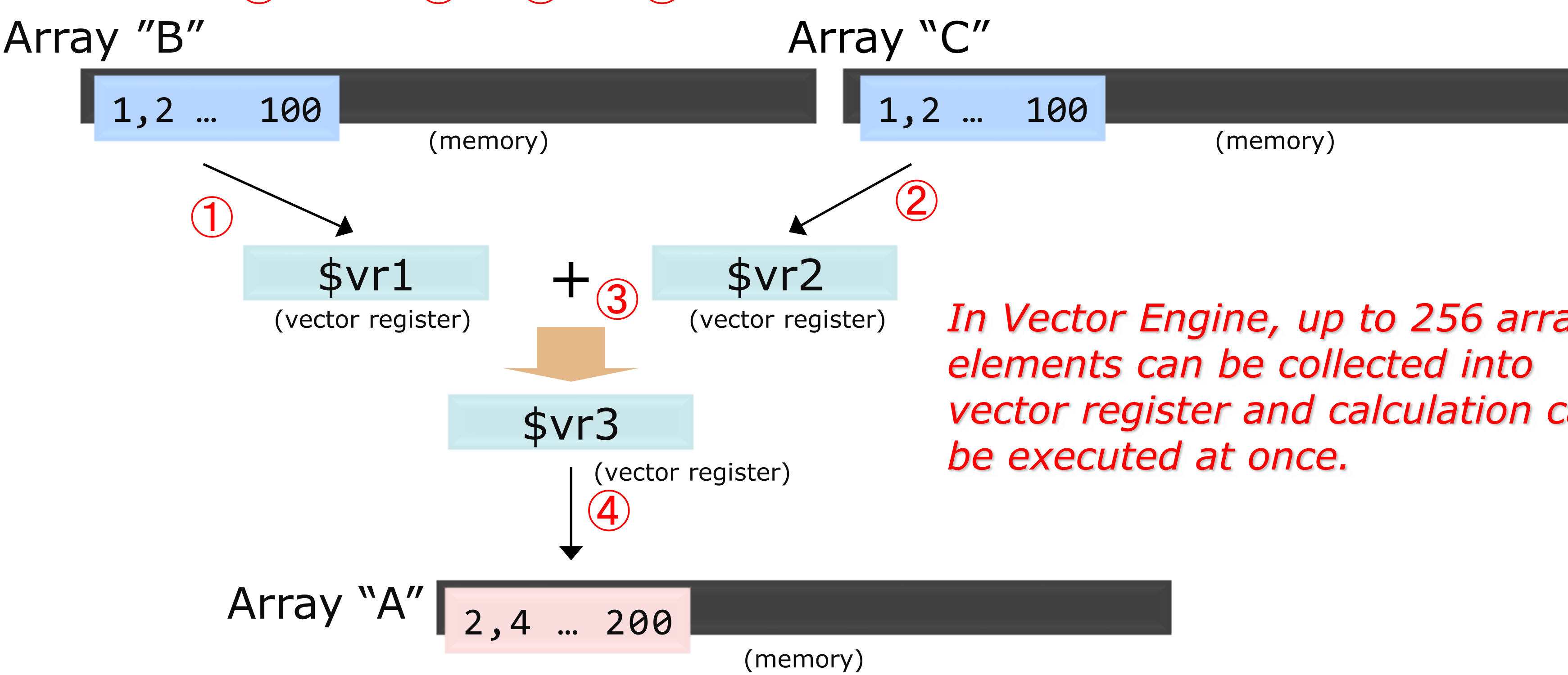
At most 256 calculation at once

Comparison of HW Instruction

In the case of a scalar machine, these four instruction sequences must be repeated 100 times.

```
A(1)   =  B(1)    +  C(1)
A(2)   =  B(2)    +  C(2)
…
A(100)=   B(100)  +  C(100)
  ④         ①     ③    ②
```

① VLoad $vr1, B(1:100)
② VLoad $vr2, C(1:100)
③ VAdd $vr3, $vr1, $vr2
④ VStore $vr3, A(1:100)

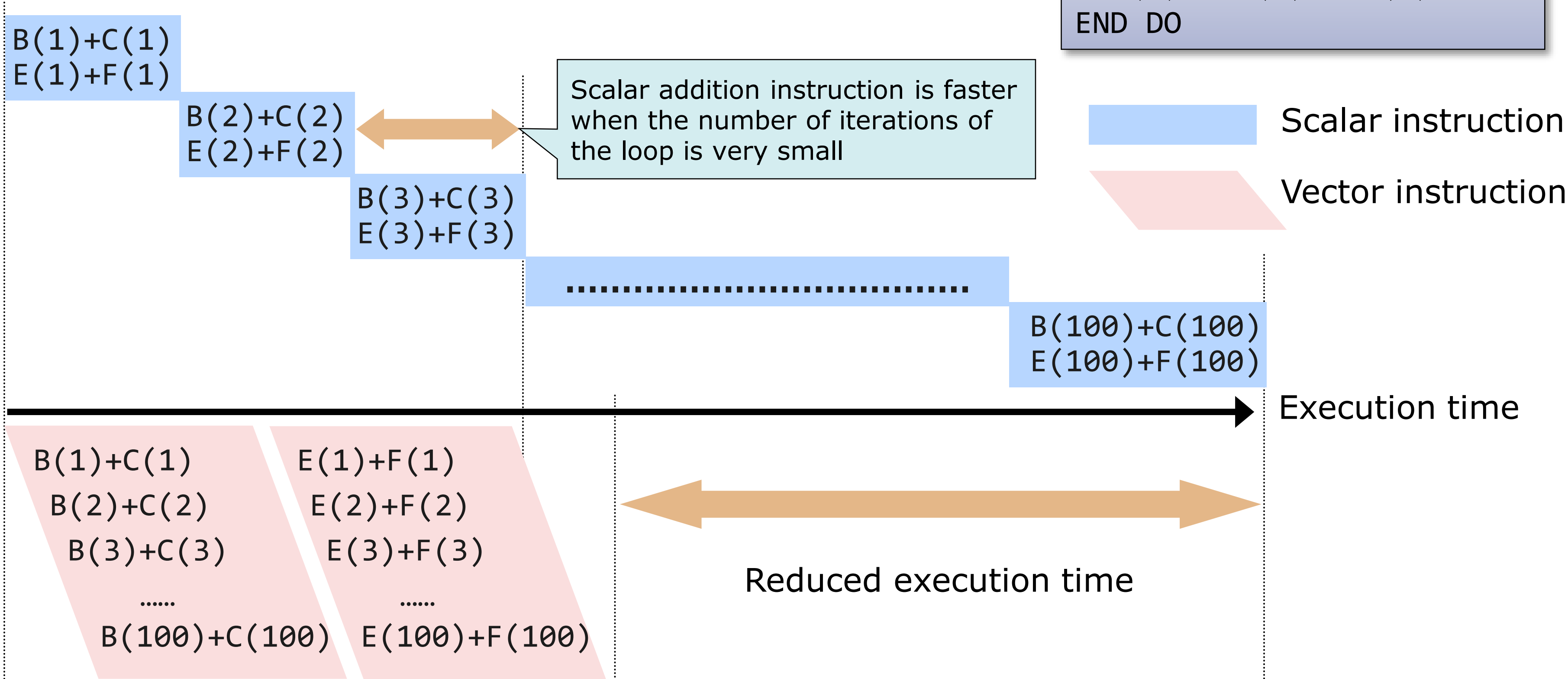


In Vector Engine, up to 256 array elements can be collected into vector register and calculation can be executed at once.

Comparison of Instruction Execution Time

```
DO I = 1, 100
  A(I) = B(I) + C(I)
  D(I) = E(I) + F(I)
END DO
```

Execution image of scalar addition instruction
(when two instructions are simultaneously executed)



Note that the order of addition has changed.
(“**B (2) + C (2)**” is added faster than “**E (1) + F (1)**”)

When the number of loop iterations is large enough, vector instructions can achieve maximum performance.

Execution image of vector addition instruction

Vectorizable Loop

- A loop which contains only vectorizable types and operations.
  - Not include 1-byte, 2-byte and 16-byte data types.
    - These types are rarely used in numerical calculations.
    - There are no corresponding type of vector operation instructions.
  - Not include function call.
    - Except trigonometric functions, exponential functions and logarithmic functions. These are vectorizable.

- There are no unvectorizable dependencies in the definition and reference of arrays and variables.
  - It is possible to change the calculation order.
- Performance improvement can be expected by vectorization.
  - Loop length (number of loop iterations) is sufficiently large.

Unvectorizable Dependencies (1)

The calculation order cannot be changed, when array elements or variables which defined in the previous iteration are referred in the later iteration.

Example 1

```
DO I = 2, N
  A(I+1) = A(I) * B(I) + C(I)
END DO
```

Unvectorizable, because the updated "A" value cannot be referenced.

Calculation order in scalar

A(3) = A(2) * B(2) + C(2);
A(4) = A(3) * B(3) + C(3);
A(5) = A(4) * B(4) + C(4);
A(6) = A(5) * B(5) + C(5);
:

Calculation order in vector

A(3) = A(2) * B(2) + C(2);
A(4) = A(3) * B(3) + C(3);
A(5) = A(4) * B(4) + C(4);
A(6) = A(5) * B(5) + C(5);
:

A(n) : Updated "A" value

before update

Example 2

```
DO I = 2, N
  A(I-1) = A(I) * B(I) + C(I)
END DO
```

Vectorizable, because the order of calculation does not change.

Calculation order in scalar

A(1) = A(2) * B(2) + C(2);
A(2) = A(3) * B(3) + C(3);
A(3) = A(4) * B(4) + C(4);
A(4) = A(5) * B(5) + C(5);
: :

Calculation order in vector

A(1) = A(2) * B(2) + C(2);
A(2) = A(3) * B(3) + C(3);
A(3) = A(4) * B(4) + C(4);
A(4) = A(5) * B(5) + C(5);

Check that there is no lower right arrow between loop iterations.

Unvectorizable Dependencies (2)

Example 3

```
DO I = 1, N
   A(I) = S
   S = B(I) + C(I)
END DO
```

Unvectorizable, because the reference of “S” appears before its definition in a loop.

Calculation order in scalar

```
A(1) = S
S = B(1) + C(1)
A(1) = S
S = B(1) + C(1)
      :
```

Calculation order in vector

```
A(1) = S
A(2) = S
      :
A(N) = S
S = B(1) + C(1)
S = B(2) + C(2)
      :
```

```
A(1) = S
DO I = 2, N
   S = B(I-1) + C(I-1)
   A(I) = S
END DO
S = B(N) + C(N)
```

It can be vectorized by transforming the program.

Calculation order in scalar

```
A(1) = S
S = B(1) + C(1)
A(2) = S
S = B(2) + C(2)
      :
```

Calculation order in vector

```
A(1) = S
S = B(1) + C(1)
S = B(2) + C(2)
      :
A(2) = S
A(3) = S
      :
```

Unvectorizable Dependencies (3)

Example 4

```
S = 1.0
DO I = 1, N
   IF (A(I) .LT. 0.0) THEN
      S = A(I)
   END IF
   B(I) = S + C(I)
END DO
```

Cannot be vectorized when a variable definition may not be executed, even if its definition precedes its reference.

Example 5

```
DO I = 1, N
   IF (A(I) .LT. 0.0) THEN
      S = A(I)
   ELSE
      S = D(I)
   END IF
   B(I) = S + C(I)
END DO
```

Can be vectorized, because there is always a definition of “S” before its reference.

Example 6

```
DO I = 1, N
   A(I) = A(I+K) + B(I)
END DO
```

Cannot be vectorized. It is not possible to determine whether there is a dependency or not, because the value of “K” is unknown at compilation.

Unknown pattern in Example 1 or 2

Vectorization of Array Expression

Program

```
A(1:M,1:N) = B(1:M,1:N) + C(1:M,1:N)
B(1:M,1:N) = SIN(D(1:M,1:N))
```

Image of transformation by compiler 1

```
DO J = 1, N
  DO I =1, M
    A(I,J) = B(I,J) + C(I,J)
  END DO
END DO
DO J = 1, N
  DO I =1, M
    B(I,J) = SIN(D(I,J))
  END DO
END DO
```

Image of transformation by compiler 2

```
DO J = 1, N
  DO I =1, M
    A(I,J) = B(I,J) + C(I,J)
    B(I,J) = SIN(D(I,J))
  END DO
END DO
```

An array expression is vectorized on optimal dimension after the compiler internally transforms it to **DO** loop format and performs optimizations such like loop fusion, loop collapse and so on.

Vectorization of IF Statement

Conditional branches (**IF** statements) are also vectorized.

```
DO I = 1, 100
  IF (A(I) .LT. B(I)) THEN
    A(I) = B(I) + C(I)
  END IF
END DO
```

Execute with vector operations

```
mask(1)   = A(1) .LT. B(1)
mask(2)   = A(2) .LT. B(2)
  :           :          :
mask(100) = A(100) .LT. B(100)
```

```
if (mask(1) == .TRUE.)   A(1) = B(1) + C(1)
if (mask(2) == .TRUE.)   A(2) = B(2) + C(2)
 :            :              :
if (mask(100) == .TRUE.) A(100) = B(100) + C(100)
```

Diagnostic Message

- You can check the vectorization status from output messages and lists of the compiler.
  - Standard error … **-fdiag-vector=2** (detail)
  - Outputs diagnostic list … **-report-diagnostics**

```
$ nfort -fdiag-vector=2 abc.f
…
nfort: vec( 103): abc.f, line 23: Unvectorized loop.
nfort: vec( 122): abc.f, line 24: Dependency unknown. Unvectorizable dependency is assumed.: RHO
nfort: vec( 122): abc.f, line 25: Dependency unknown. Unvectorizable dependency is assumed.: RHO
nfort: vec( 101): abc.f, line 50: Vectorized loop.
…
$ nfort -report-diagnostics abc.f
…
$ less abc.L
FILE NAME: abc.f
…
PROCEDURE NAME: SUB
DIAGNOSTIC LIST

 LINE                DIAGNOSTIC MESSAGE

    23: vec( 103): Unvectorized loop.
    24: vec( 122): Dependency unknown. Unvectorizable dependency is assumed.: RHO
    25: vec( 122): Dependency unknown. Unvectorizable dependency is assumed.: RHO
    50: vec( 101): Vectorized loop.
…
```

A message indicating that pointer RHO is considered to have a dependency that cannot be vectorized and has not been vectorized

List file name is "source file name. L"

Format List

Loop structure and vectorization, parallelization and inlining statuses are output with the source lines

- A format list is output when **-report-format** is specified.

```
$ nfort -report-format a.f90 -c
...
$ less a.L
          :
PROCEDURE NAME: SUB
FORMAT LIST

 LINE   LOOP      STATEMENT

     1:           SUBROUTINE SUB(A, B, C, X, Y, Z, N)
     2:             INTEGER :: N
     3:             REAL(KIND=4) :: A(N), B(N), C(N)
     4:             REAL(KIND=16) :: X(N), Y(N), Z(N)
     5:             INTEGER :: I
     6:
     7: V------>    DO I = 1, N
     8: |             A(I) = B(I) * C(I)
     9: V------     END DO
    10:
    11: +------>    DO I = 1, N
    12: |             X(I) = Y(I) * Z(I)
    13: +------     END DO
    14:
    15:           END SUBROUTINE SUB
```

List file name is ”source file name.L”

The whole loop is vectorized.

The loop is not vectorized.

Extended Vectorization Features

Extended Vectorization Features

When the basic conditions for vectorization are not satisfied, the compiler performs as much vectorization as possible by transforming the program and using the special vector operations.

- Statement Replacement
- Loop Collapse
- Loop Interchange
- Partial Vectorization
- Conditional Vectorization
- Macro Operations
- Outer Loop Vectorization
- Loop Fusion
- Inlining

Statement Replacement

Source Program

```
DO I = 1, 99
   A(I) = 2.0
   B(I) = A(I+1)
END DO
```

Transformation Image

```
DO I = 1, 99
   B(I) = A(I+1)
   A(I) = 2.0
END DO
```

When this loop is vectorized, all the value from B[1] to B[99] will be 2.0. This loop do not satisfy the vectorization conditions.

The compiler replaces the statements in the loop to satisfy the vectorization conditions.

Loop Collapse

Source Program

```
REAL A(M,N), B(M,N), C(M,N)
DO J = 1, N
  DO I = 1, M
    A(I,J) = B(I,J) + C(I,J)
  END DO
END DO
```

Transformation Image

```
REAL A(M,N), B(M,N), C(M,N)
DO IJ = 1, M*N
  A(IJ,1) = B(IJ,1) + C(IJ,1)
END DO
```

A loop collapse is effective in increasing the loop iteration count and improving the efficiency of vector instructions.

Loop Interchange

Source Program

```
DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J) + B(I,J)
  END DO
END DO
```

```
A(2,1) = A(1,1) + B(1,1)
A(3,1) = A(2,1) + B(2,1)
A(4,1) = A(3,1) + B(3,1)
A(5,1) = A(4,1) + B(4,1)
```

The loop “`DO I=1,N`” has unvectorizable dependency about the array A.

Transformation Image

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J) = A(I,J) + B(I,J)
  END DO
END DO
```

```
A(2,1) = A(1,1) + B(1,1)
A(2,2) = A(1,2) + B(1,2)
A(2,3) = A(1,3) + B(1,3)
A(2,4) = A(1,4) + B(1,4)
```

Interchanging loops removes unvectorizable dependency, and enable the loop “`DO J=1,M`” to be vectorized.

Partial Vectorization

Source Program

```
DO I = 1, N
  X = A(I) + B(I)
  Y = C(I) + D(I)
  WRITE(6,*) X, Y
END DO
```

Transformation Image

```
DO I = 1, N
  WX(I) = A(I) + B(I)
  WY(I) = C(I) + D(I)
END DO
DO I = 1, N
  WRITE(6,*) WX(I), WY(I)
END DO
```

Vectorizable (first loop)

Unvectorizable (second loop)

If a vectorizable part and an unvectorizable part exist together in a loop, the compiler divides the loop into vectorizable and unvectorizable parts and vectorizes just the vectorizable part.

To do this, work vectors (the array WX and WY in above example) are generated if necessary.

Conditional Vectorization

Source Program

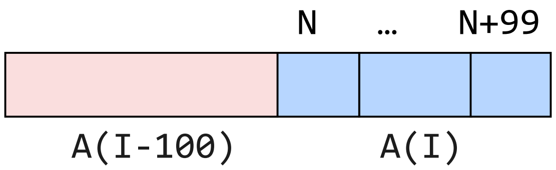
```
DO I = N, N+99
  A(I) = A(I+K) + B(I)
END DO
```

Transformation Image

```
IF((K.GE.0) .OR. (K.LT.-99)) THEN
  ! Vectorized Code
ELSE
  ! Unvectorized Code
END IF
```

The compiler generates a variety of codes for a loop, including vectorized codes and scalar codes, as well as special codes and normal codes. The type of code is selected by run-time testing at execution when conditional vectorization is performed.

```
(When k=-1)
    A(I) = A(I-1) + B(I)

(When k=-100)
    A(I) = A(I-100) + B(I)
```



Macro Operations

Sum

```
DO I = 1, N
  S = S + A(I)
END DO
```

Iteration

```
DO I = 1, N
  A(I) = A(I-1) * B(I) + C(I)
END DO
```

Maximum or minimum values

```
DO I = 1, N
  IF (XMAX .LT. X(I)) THEN
    XMAX = X(I)
  END IF
END DO
```

Although patterns like these do not satisfy the vectorization conditions for definitions and references, the compiler recognizes them to be special patterns and performs vectorization by using proprietary vector instructions.

Outer Loop Vectorization

Source Program

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = 0.0
  END DO
  B(I) = 1.0
END DO
```

Transformation Image

```
DO I = 1,N
  DO J = 1,N
    A(I,J) = 0.0
  END DO
END DO
DO I = 1, N
  B(I) = 1.0
END DO
```

In this case, these loops are collapsed.

The compiler basically vectorizes the innermost loop.
If a statement which is contained only in the outer loop exists, the compiler divides the loop and vectorizes the divided outer loop.

Loop Fusion

Source Program

```
DO I = 1, N
  A(I) = B(I) + C(I)
END DO
DO I = 1, N
  D(I) = SIN(E(I))
END DO
```

Transformation Image

```
DO I = 1, N
  A(I) = B(I) + C(I)
  D(I) = SIN(E(I))
END DO
```

```
A(1:M) = B(1:M) + C(1:M)
D(1:M) = E(1:M) * F(1:M) + S
```

```
DO I = 1, M
  A(I,J) = B(I,J) + C(I,J)
  D(I,J) = E(I,J) * F(I,J) + S
END DO
```

The compiler fuses consecutive loops which have the same iteration count and vectorizes the fused loop.
If the same shape array and loop structure are continuous, they can be fused. But if there are the different shape arrays, loop structures, and other sentences, they cannot be fused.
In order to increase speed, it is better to make same shape arrays and loop structures continuous as much as possible.

Vectorization with Inlining

Source Program

```
DO I = 1, N
  CALL SUB(B(I),C(I))
  A(I) = B(I)
END DO
   :
SUBROUTINE SUB(X,Y)
  X = SIN(Y)
END
```

Transformation Image

```
DO I=1,N
  B(I) = SIN(C(I))
  A(I) = B(I)
END DO
   :
SUBROUTINE SUB(X,Y)
  X = SIN(Y)
END
```

When the **-finline-functions** option is specified, the compiler expands the function directory at the point of calling it if possible. If the function is called in a loop, the compiler tries to vectorize the loop after inlining the function.

Program Tuning

"Tuning" is to increase executing speed of a program (reduce the execution time) by specifying compiler options and #pragma directives.

The performance of Vector Engine system can be derived at the maximum by tuning.

Point of View in Tuning

Raising the Vectorization Ratio

- The vectorization ratio is the ratio of the part processed by vector instructions in the whole program.
- The vectorization ratio can be improved by removing the cause of unvectorization.
  - Increase the part processed by vector instructions.

Improving Vector Instruction Efficiency

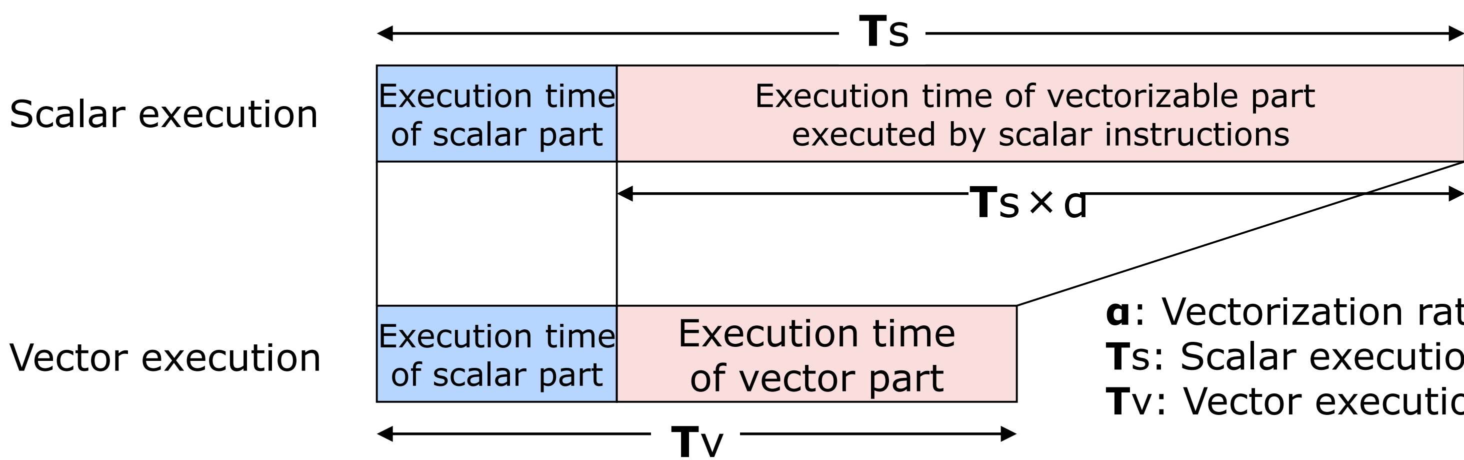
- Increase the amount of data processed by one vector instruction.
  - Make the iteration count of a loop (loop length) as long as possible.
- Stop vectorization when the loop is so short.
  - See p.21 “Comparison of instruction execution time”.

Improving Memory Access Efficiency

- Avoid using a list vector.

Vectorization Ratio

- The ratio of the part processed by vector instructions in whole program



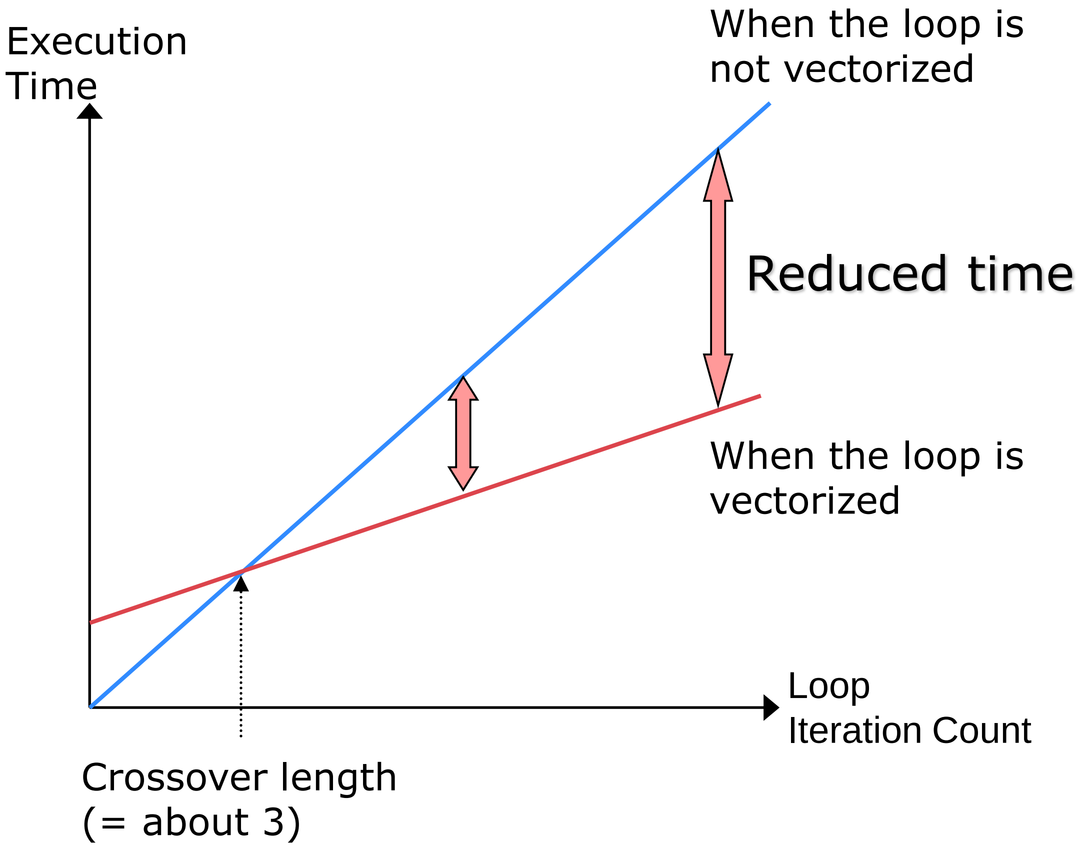
α: Vectorization ratio
Ts: Scalar execution time
Tv: Vector execution time

- The vector operation ratio is used instead of the vectorization ratio

$$\textbf{Vector operation ratio} = 100 \times \frac{\text{Number of vector instruction execution elements}}{\text{Execution count of all instructions} - \text{Execution count of vector instructions} + \text{Number of vector instruction execution elements}}$$

Loop Iteration Count and Execution Time

- To maximize the effect of vectorization, the loop iteration count should be made as long as possible
  - Increase the amount of data processed by one vector instruction.



It is difficult to analyze iteration count for each loops.

Analyze **average vector length**.

The average number of date processed by one vector instruction.
The maximum number is 256.

Process of Tuning

- Finding the function whose execution time is long, vector operation ratio is law and average vector length is short from the performance analysis information
  - PROGINF
    - Execution time, vector operation ratio and average vector length of the whole program.
  - FTRACE
    - Execution time, execution count, vector operation ratio and average vector length of each function.

↓

- Finding unvectorized loops in the function from diagnostics for vectorization

↓

- Improving vectorization by specifying compiler options and directives

Output example

```
   ********  Program  Information  ********
  Real Time (sec)                   :           11.336602
  User Time (sec)                   :           11.330778
  Vector Time (sec)                 :           11.018179
  Inst. Count                       :          6206113403
  V. Inst. Count                    :          2653887022
  V. Element Count                  :        619700067996
  V. Load Element Count             :         53789940198
  FLOP count                        :        576929115066
  MOPS                              :        73455.206067
  MOPS (Real)                       :        73370.001718
  MFLOPS                            :        50950.894570
  MFLOPS (Real)                     :        50891.794092
  A. V. Length                      :          233.506575
  V. Op. Ratio (%)                  :           99.572922
  L1 Cache Miss (sec)               :            0.010855
  CPU Port Conf. (sec)              :            0.000000
  V. Arith. Exec. (sec)             :            8.410951
  V. Load Exec. (sec)               :            1.386046
  VLD LLC Hit Element Ratio (%)     :          100.000000
  Power Throttling (sec)            :            0.000000
  Thermal Throttling (sec)          :            0.000000
  Max Active Threads                :                   1
  Available CPU Cores               :                   8
  Average CPU Cores Used            :            0.999486
  Memory Size Used (MB)             :          204.000000
```

A.V.Length (Average vector length)

- Indicator of vector instruction efficiency.
- The longer, the better (Maximum length: 256).
- If this value is short, the iteration count of the vectorized loops is insufficient.

V.Op.Ratio (Vector operation ratio)

- Ratio of data processed by vector instructions.
- The larger, the better (Maximum rate: 100).
- If this value is small, the number of vectorized loops is small or there are few loops in the program.

FTRACE

- A feature used to obtain performance information of each function
  - Focus on V.OP.RATIO (Vector operation ratio) and AVER.V.LEN (Average vector length) as well as PROGINF, and analyze the performance of each function.

```
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Mar 22 15:47:42 2018 JST
Total CPU Time : 0:00'11"168 (11.168 sec.)


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS      CONF HIT E.%

    15000     4.767( 42.7)     0.318  77030.2  61964.6  99.45 251.0     4.610   0.002     0.000  100.00 FUNC_A
    15000     3.541( 31.7)     0.236  73505.6  56940.8  99.46 216.0     3.555   0.000     0.000  100.00 FUNC_B
    15000     2.726( 24.4)     0.182  71930.1  27556.5  99.43 230.8     2.725   0.000     0.000  100.00 FUNC_C
        1     0.134(  1.2)   133.700  60368.9  35641.3  98.53 214.9     0.118   0.000     0.000    0.00 MAIN
----------------------------------------------------------------------------------------------------------------
    45001    11.168(100.0)     0.248  74468.3  51657.9  99.44 233.5    11.008   0.002     0.000  100.00 total
```

Tuning Techniques

Orchestrating a brighter world NEC

Compiler Directives

- The compiler directive is to give the compiler the information that it cannot obtain from source code analysis alone to further the effects of the vectorization and parallelization, writing **!NEC$**.
  - The compiler directive format is as follows.

    !NEC$ *directive-name* [*clause*] (free format / fixed format)

    ***NEC$** *directive-name* [*clause*] (fixed format)

    cNEC$ *directive-name* [*clause*] (fixed format)
  - Major vectorized compiler directives.
    - **VECTOR**/**NOVECTOR** : Allows [Disallows] automatic vectorization of the following loop
    - **IVDEP** : Regards the unknown dependency as vectorizable dependency during the automatic vectorization.

```
!NEC$ IVDEP
DO I = 1, N
  A(IX(I)) = A(IX(I)) + B(I)
END DO
```

- Specify the vectorization directive option just before the loop by delimiting with the specified space.
- It works only for the loop immediately after the directive.

Dealing with Unvectorizable Dependencies (1)

Raising Vectorization Ratio

```
nfort: vec( 103): a.f, line 16: Unvectorized loop.
nfort: vec( 113): a.f, line 16: Overhead of loop division is too large.
nfort: vec( 121): a.f, line 18: Unvectorizable dependency.
```

Such messages may be displayed to attempt partial vectorization.

It cannot be vectorized. Because compiler cannot recognizes the variable “T” is defined or not.

Unvectorized Loop

```
DO I = 1, N
  IF (X(I).LT.S) THEN
    T = X(I)
  ELSE IF (X(I).GE.S) THEN
    T = -X(I)
  END IF
  Y(I) = T
END DO
```

Vectorized Loop

```
DO I = 1, N
  IF (X(I).LT.S) THEN
    T = X(I)
  ELSE
    T = -X(I)
  END IF
Y(I) = T
```

Modified so that variable “T” is always defined.

Compiler cannot recognizes sum type macro operation.

Unvectorized Loop

```
DO I = 1, N
  IF (A(I).GT.0.0) THEN
    S = S + B(I)
  ELSE
    S = S + C(I)
  END IF
END DO
```

Vectorized Loop

```
DO I = 1, N
  IF (A(I).GT.0.0) THEN
    T = B(I)
  ELSE
    T = C(I)
  END IF
  S = S + T
END DO
```

Vectorization as a sum type macro operation.

<Diagnostic message after vectorization>

```
nfort: vec( 101): a.f, line 16: Vectorized loop.
nfort: vec( 126): a.f, line 22: Idiom detected.: Sum.
```

Sum type macro operation is vectorized using special HW instruction

Dealing with Unvectorizable Dependencies (2)

Raising Vectorization Ratio

```
nfort: vec( 103): dep.f90, line 5: Unvectorized loop.
nfort: vec( 122): dep.f90, line 6: Dependency unknown. Unvectorizable dependency is
assumed.: A
```

- Specify “**IVDEP**” if you know that there are no unvectorizable data dependencies in the loops, even when the compiler assumed that some unvectorizable dependencies exist.

Unvectorized Loop

```
SUBROUTINE SUB(A, B, C, N, K)
  INTEGER I, N, K
  REAL A(N), B(N), C(N)

  DO I = 1, N
    A(I+K) = A(I) + B(I)
  END DO
END SUBROUTINE SUB
```

It is not vectorized because it is unknown whether the pattern of `A(I-1) = A(I)` or the pattern of `A(I+1) = A(I)`

Vectorized Loop

```
SUBROUTINE SUB(A, B, C, N, K)
  INTEGER I, N, K
  REAL A(N), B(N), C(N)
!NEC$ IVDEP
  DO I = 1, N
    A(I+K) = A(I) + B(I)
  END DO
END SUBROUTINE SUB
```

When it is clear that the pattern is `A(I-1) = A(I)`, specify “**IVDEP**” to vectorize

<Diagnostic message after vectorization>

```
nfort: vec( 101): dep.f90, line 5: Vectorized loop.
```

Inlining: Improving of Vectorization

Raising Vectorization Ratio

```
nfort: vec( 110): a.f90, line 4: Vectorization obstructive procedure reference.: FUN
nfort: vec( 103): a.f90, line 4: Unvectorized loop.
nfort: opt(1025): a.f90, line 5: Reference to this procedure inhibits optimization.: FUN
```

- When a function call prevents vectorization, above messages are output
- Try to inlining with specifying **"-finline-functions"** option

```
SUBROUTINE SUB(A, B, C, D, N)
  INTEGER I, N
  REAL A(N), B(N), C(N), D(N)
  DO I=1, N              ! Unvectorized
    A(I) = FUN(B(I), C(I)) / D(I)
  END DO
END

FUNCTION FUN(X, Y)
  REAL X, Y
  FUN = SQRT(X) * Y
END FUNCTION FUN
```

<Specifying compiler option >

```
$ nfort –finline-functions a.f90
```

<Transformation Image>

```
DO I=1, N              ! Vectorized
  A(I)= SQRT(B(I))*C(I) / D(I)
END DO
```

SQRT is a vectorizable function, so it does not prevent vectorization.

```
nfort: vec( 101): func.f90, line 4: Vectorized loop.
nfort: inl(1222): func.f90, line 5: Inlined: FUN
```

A Loop Contains an Array with a Vector Subscript Expression

Raising Vectorization Ratio

```
nfort: vec( 101): a.f90, line 5: Vectorized loop.
nfort: vec( 126): a.f90, line 6: Idiom detected.: LIST VECTOR
```

Specifying **IVDEP** for the list vector further improve performance

- List vector is an array with a vector subscript expression.
- When the same list vector appears on both the left and right sides of an assignment operator, it cannot be vectorized because its dependency is unknown.

Vectorized Loop ("**LIST_VECTOR**" Directives)

```
!NEC$ LIST_VECTOR
  DO I = 1, N
    A(IX(I)) = A(IX(I)) + B(I)
  END DO
```

Vectorized Loop (**"IVDEV"** Directives)

```
!NEC$ IVDEP
  DO I = 1, N
    A(IX(I)) = A(IX(I)) + B(I)
  END DO
```

If **LIST_VECTOR** is specified, the loop can be vectorized.
If the same element of array "A" is not defined twice or more in the loop, in other words, <u>if there are no duplicate values in "IX(I)"</u>, *more efficient vector instructions can be generated by specifying **IVDEP** instead of **LIST_VECTOR***.

<Message after vectorization by **IVDEP**>

```
nfort: vec( 101): a.f90, line 5: Vectorized loop.
```

Outer Loop Unrolling

Outer loop unrolling will reduce the number of load and store operations in the inner loops.

- Unrolling the outer loop when there are multiple loop nests reduces the number of loads and stores that use only the inner loop's induction variable.

```
DO J = 1, N
  DO I = 1, M
    A(I,J) = B(I,J) + C(I)
  END DO
END DO
```

Insert **OUTERLOOP_UNROLL(4)** directive

```
!NEC$ OUTERLOOP_UNROLL(4)
DO J = 1, N
  DO I = 1, M
    A(I,J) = B(I,J) + C(I)
  END DO
END DO
```

specify 2^x times unrolling in parentheses.

Program after unrolling the outer loop 4 times.

```
DO J = 1, N%3
  DO I = 1, M
    A(I,J) = B(I,J) + C(I)
  END DO
END DO

DO J = N%3+1, N, 4
  DO I = 1, M
    A(I,J)   = B(I,J)   + C(I)
    A(I,J+1) = B(I,J+1) + C(I)
    A(I,J+2) = B(I,J+2) + C(I)
    A(I,J+3) = B(I,J+3) + C(I)
  END DO
END DO
```

4 times vector operations can be performed per one vector load in array “C”

Specifying **OUTERLOOP_UNROLL** directive or **-fouterloop-unroll** option shortens the loop length of the outer loop (induction variable “I”) and reduces the number of vector loads of the array “C”.

<Message after outer loop unroll by **OUTERLOOP_UNROLL** directive>

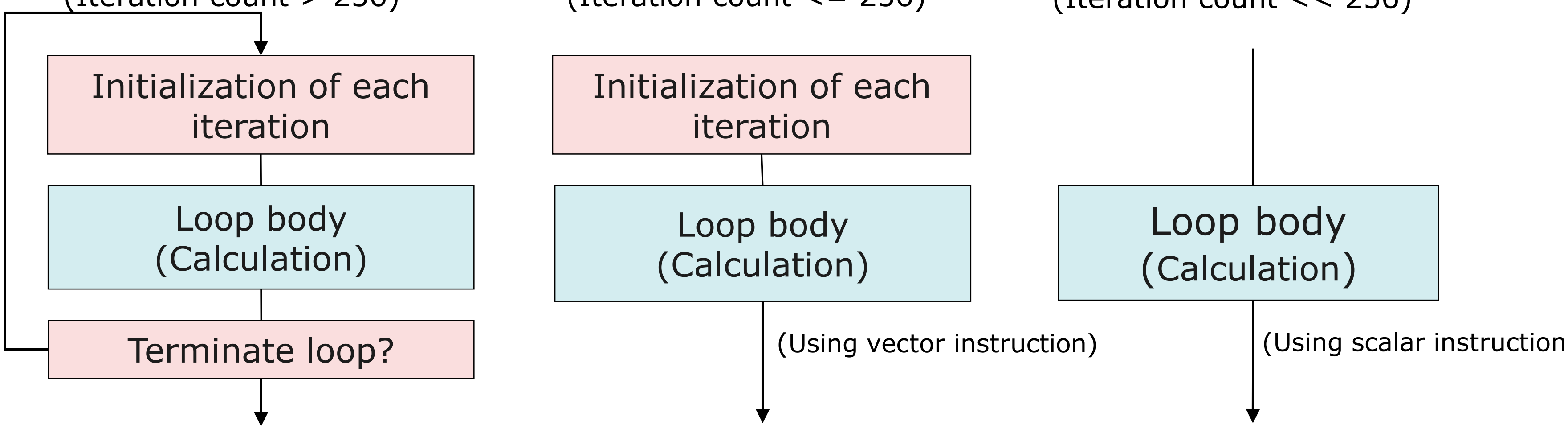
```
nfort: opt(1592): a.f90, line 5: Outer loop unrolled inside inner loop.: J
nfort: vec( 101): a.f90, line 6: Vectorized loop.
```

Small Iteration Loop

Raising Vectorization Ratio

- When the iteration count is small, loop controlling expressions can be eliminated
  - The iteration count <= 256 : A short-loop which does not have “terminate loop?” is generated.
  - The iteration count << 256 : The loop is expanded and loop controlling expressions are eliminated.

Normal Loop (Iteration count > 256) | **Short-Loop** (Iteration count <= 256) | **Loop Expansion** (Iteration count << 256)




```
DO I = 1, N
...
END DO
```

```
!NEC$ SHORTLOOP
DO I = 1, N
...
END DO
```

```
!NEC$ UNROLL(7)
DO I = 1, 7
...
END DO
```

Notes on Using Vectorization

Level of Automatic Vectorization and Optimization Applied

- The following vectorization and optimization are applied automatically when changing the level of automatic vectorization at "**-O4**", "**-O3**" and "**-O2**"

high ←→ low

Applied vectorization and optimization	***-O4***	***-O3***	***-O2***
Vectorization by condition vectorization (`-m[no-]vector-dependency-test`)	○	○	○
Vectorization by loop collapse, loop interchange and transform matrix multiply loops into a vector matrix library function call. (`-f[no-]loop-collapse`, `-f[no-]loop-interchange`, `-f[no-]matrix-multiply`)	○	○	—
Disallows the compiler to assume that the object pointed-to-by a named pointer are aliasing in vectorization. (`-fnamed-[no]alias`)	○	○	—
Allows outer-loop unrolling (`-f[no-]outerloop-unroll`)	○	○	—

Remark: Only the major options are listed, () is the compiler option when specifying separately.

Influence on Result by Vectorization

Results may differ within an error range with and without vectorization

- "Conversion of division to multiplication" or "reordering of operations" may cause "loss of trailing digits", "cancellation" and "rounding error".
- The vector versions of mathematical functions do not always use the same algorithms as the scalar versions.
- An integer iteration macro operation is vectorized by using a floating point instruction. So when the result exceeds 52 bits or when a floating overflow occurs, the result differs from that of scalar execution.
- When vector fusion product-sum operation (FMA) is used, since addition is performed without rounding up the integration result in the middle, the operation result may be different from when it is not used.

If you care about the error range

- Specify the "**NOVECTOR**" directive. The loop is not vectorized.
- Specify the "**NOFMA**" directive. Vector fused-multiply-add instruction does not generated.

```
!NEC$ NOVECTOR
DO I = 1, N
  SUM = SUM + A(I)
END DO
```

The Bus Error Caused by Vectorization

It may occur because vector load/store for 8 bytes elements is executed for the array aligned in 4 bytes

- In the following example, sub.f90 is compiled with **`-fdefault-real=8`**. Therefore, the arrays "A" and "B" of type **`REAL`** are vector loaded/stored for 8 bytes elements.
- Vector load/store for 8 bytes elements requires an array aligned in 8 bytes. If the array is aligned in 4 bytes, the execution failed by the bus error for an invalid memory access.

main.f90

```
PROGRAM MAIN
  REAL :: A(512), B(512)
  ...
  CALL SUB(A,B,512)
END
```

sub.f90

```
SUBROUTINE SUB(A, B, N)
  INTEGER :: N
  REAL :: A(N), B(N)
  B = A                     !!!<---vectorized
END SUBROUTINE SUB
```

```
$ nfort -c main.f90
$ nfort -c -fdefault-real=8 sub.f90
$ nfort main.o sub.o
$ ./a.out
Bus error
```

Declare an array as 4 bytes data type explicitly or specify the **NOVECTOR** directive to the loop to stop vectorization

```
SUBROUTINE SUB(A, B, N)
  INTEGER :: N
  REAL(KIND=4) :: A(N), B(N)
  B = A
END SUBROUTINE SUB
```

Explicitly specify 4 bytes data type.

```
SUBROUTINE SUB(A, B, N)
  INTEGER :: N
  REAL :: A(N), B(N)
!NEC$ NOVECTOR
  B = A
END SUBROUTINE SUB
```

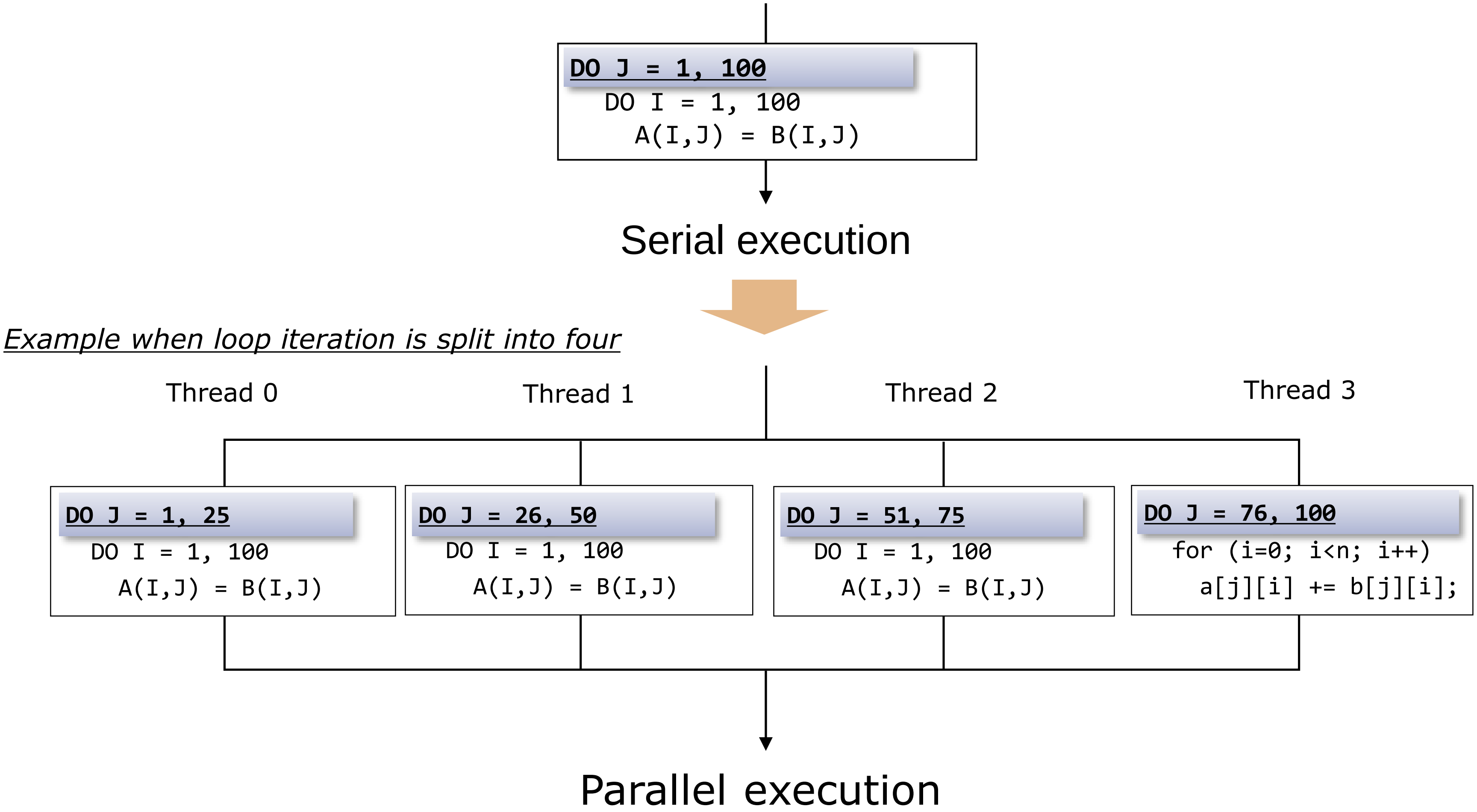
Specify **NOVECTOR** directive.

Automatic Parallelization and OpenMP Fortran

Automatic Parallelization Features

Split one job and execute it simultaneously in multiple threads

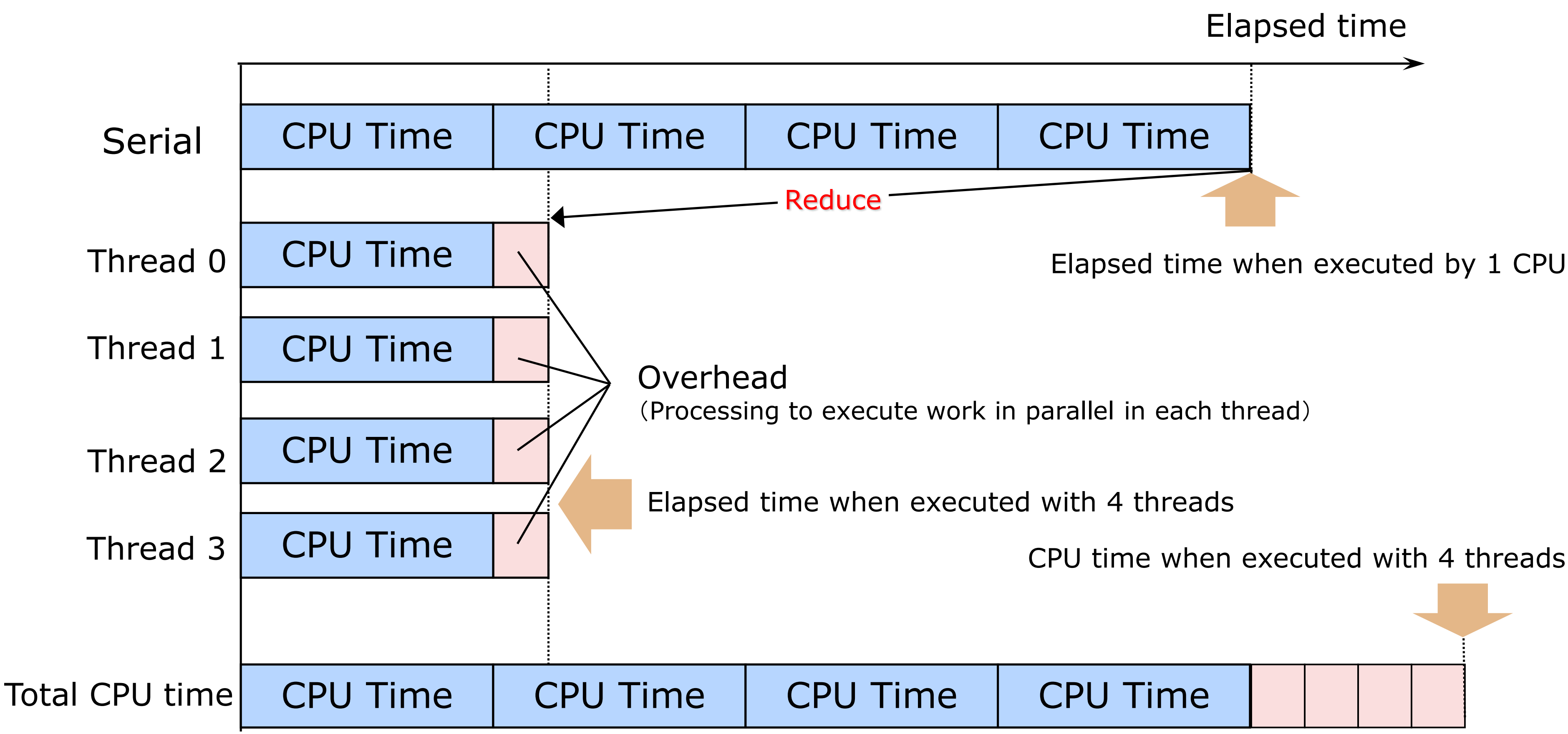
- Split loop iteration.
- Split a series of processing (a collection of sentences) in a program.



Reduce the Elapsed Time by Parallelization

Reduce the elapsed time by parallelization

- Increase total CPU time due to overhead for parallel processing.



Program Parallelization

Program to execute in parallel in multiple threads

- Select loops and statements and extract code that can be execute in parallel.
- Generate executable code to execute in parallel with automatic parallelization or OpenMP.

Example 1: Parallelization by automatic parallelization

```
SUBROUTINE SUB(A, N)
  INTEGER :: N, I, J
  REAL(KIND=8) :: A(N), B(N)
  REAL(KIND=8) :: SUM = 1.0

  DO J = 1, N
    DO I = 1, N
      SUM = SUM + A[j] + B[I]
    ENDDO
  ENDDO

  RETURN
END SUBROUTINE SUB
```

Specify "**-mparallel**" to enable automatic parallelization.

```
$ nfort -mparallel a.f90
nfort: par(1801): a.f90, line 6: Parallel routine generated.: SUB$1
nfort: par(1803): a.f90, line 6: Parallelized by "do".
nfort: vec( 101): a.f90, line 7: Vectorized loop.
```

Vectorize the inner loop.

Extract as another function to execute the loop in parallel.

Search loops that can be execute in parallel.

Remark: Other part of loop is regarded as impossible to execute in parallel.

Parallelization Programming Available on Vector Engine

OpenMP Fortran

- The programmer selects a set of loops and statement blocks that can be executed in parallel, and specifies OpenMP directives indicating how to parallelize them.
- The compiler transforms the program based on the instruction and inserts a directives for parallel processing control.

Automatic parallelization

- The compiler selects loops and statement blocks that can be executed in parallel and transforms the program into parallel processing control.
- The compiler automatically performs all the work of loop detection and program modification and directives insertion of "Example 1" on the previous page.

Programming method	Select loops / blocks	Insert directives	Program modification	Difficulty
OpenMP Fortran (**-fopenmp**)	○	○	–	High
Automatic parallelization (**-mparallel**)	–	–	–	Low

○ : Handwork is needed.
– : Handwork is not needed because the compiler automatically executes it.

Remark: At the time of tuning, even if it is a section of "-", Handwork may be needed.

OpenMP Parallelization

OpenMP Fortran

```
$ nfort -fopenmp a.f90 b.f90
```

Specify "**-fopenmp**" also when linking

International standards of directives and libraries for shared memory parallel processing

- "NEC Fortran Compiler for Vector Engine" supports some features up to "OpenMP Version 4.5".

Programming method

- The programmer extracts a set of loops and statements that can be executed in parallel, and specifies OpenMP directives indicating how to parallelize them.
- The compiler modifies the program based on the instruction and inserts processing for parallel processing control.
- Compile and link with "**-fopenmp**".

Feature

- Higher performance improvement than automatic parallelization is expected because the programmer can select and specify the parallelization part.
- Easy to program because the compiler performs program transformation involving extraction of parallelized part, barrier synchronization and shared attribute of variables.

Example: Writing in OpenMP Fortran

Parallelize subroutine "SUB" of Example 1 with OpenMP Fortran

```
SUBROUTINE SUB(A, N)
  INTEGER :: N, I, J
  REAL(KIND=8) :: A(N), B(N)
  REAL(KIND=8) :: SUM = 1.0
!$OMP PARALLEL DO
  DO J = 1, N
    DO I = 1, N
      SUM = SUM + A(J) + B(I)
    ENDDO
  ENDDO

  RETURN
END SUBROUTINE SUB
```

Insert OpenMP directives.

Search loops that can be execute in parallel.

Specifying with"**-fopenmp**". And OpenMP directives is enable.

```
$ nfort -fopenmp a.f90
nfort: par(1801): a.f90, line 5: Parallel routine generated.: SUB$1
nfort: par(1803): a.f90, line 6: Parallelized by "do".
nfort: vec( 101): a.f90, line 7: Vectorized loop.
```

The Compiler modifies the program so that the compiler can execute in parallel.

- The OpenMP directives follows "**!$OMP**" to specify the parallelization method.

```
!$OMP PARALLEL DO
```

PARALLEL

Specify start of parallelization region

DO

Specify parallelization of for loop

Terms

OpenMP thread

- A unit of logical parallelism. Sometimes abbreviated as thread.

Parallel region

- A collection of statements executed in parallel by multiple OpenMP threads.

Serial region

- A collection of statements executed only by the master thread outside of a parallel region.

Private

- Accessible from only one of the OpenMP threads that execute parallel regions.

Shared

- Accessible by all OpenMP threads executing parallel regions.

OpenMP Directives

Explain only frequently used items

!$OMP PARALLEL DO [*schedule-clause*] [NOWAIT]

SCHEDULE(STATIC[,*size*]) … SCHEDULE(STATIC) is default value

- Perform round-robin allocation and execution on OpenMP threads with *size* iterations grouped together.
- When the specification of *size* is omitted, the value obtained by dividing *size* by the number of threads is regarded as specified.

SCHEDULE(DYNAMIC[,*size*])

- Dynamically allocate and execute on OpenMP thread by grouping size iterations together.
- When the specification of size is omitted, it is assumed that 1 is specified.

SCHEDULE(RUNTIME)

- Execute according to the schedule method set in the environment variable "**OMP_SCHEDULE**".

NOWAIT

- Do not perform implicit barrier synchronization at the end of parallel loop.

!$OMP SINGLE

Execute only on one OpenMP thread. Execute with the task, not necessarily the master thread that reached the directive finally.

!$OMP CRITICAL

Do not execute in multiple OpenMP threads at the same time (exclusive control).

Automatic Parallelization

Automatic Parallelization

In automatic parallelization, compiler does everything suggested in "Example: Writing in OpenMP Fortran".

```
$ nfort -mparallel a.f90 b.f90
```

Also specify **`-mparallel`** for linking.

Compile and link with **`-mparallel`**.

- Compiler finds and parallelizes parallelizable loops and statements.
  - Automatically select loops without factors inhibiting parallelization.
  - Automatically select outermost loops in multiple loops.
    - Innermost loops should be increased speed with vectorization.

Compiler directives to control automatic parallelization.

- Compiler directive format

  `!NEC$` ***`directive-option`***

- Major directive options
  - **`CONCURRENT/NOCONCURRENT`** … parallelize/not-parallelize a loop right after this.
  - **`CNCALL`** … parallelize a loop including procedure calls.

Control Automatic Parallelization with Directives

NOCONCURRENT … Do not parallelize a loop right after this directive.

```
CALL SUB(4)           ! function call
...
SUBROUTINE SUB(M)
  INTEGER :: M
    ...
!NEC$ NOCONCURRENT
    DO J = 1, M       ! M is small actually
      DO I = 1, N
        A(I) = B(J) / C(J)
      ENDDO
    ENDDO
```

Performance sometimes degrades when small loop is parallelized because overhead of parallelization accounts for much ratio of execution.

Stop parallelization by **NOCONCURRENT**

CNCALL … Parallelize a loop including function call.

```
!NEC$ CNCALL
  DO I = 1, M
    A(I) = FUNC(B(I), C(I))
  ENDDO
```

Loops including a procedure call is not parallelized automatically because it is unknown if the procedure can be executed in parallel.

Parallelize by **CNCALL** when procedures can be parallelized.

(Programmer must ensure that procedures can be executed in parallel.)

Apply Both OpenMP and Automatic Parallelization

```
$ nfort –fopenmp –mparallel a.f90 b.f90
```

Compile and link with both **-fopenmp** and **-mparallel**.

- Automatic parallelization is applied to the loops outside of OpenMP parallel regions.
- If you don't want to apply automatic parallelization to a routine containing OpenMP directives, specify **-mno-parallel-omp-routine**.

```
SUBROUTINE SUB(A, N)
  INTEGER :: I, J, N
  REAL(KIND=8) :: A(N), B(N,N)
  REAL(KIND=8) :: S = 1.0

  DO I = 1, N
    DO J = 1, N
      B(J,I) = I * J
    END DO
  END DO

!$OMP PARALLEL DO
  DO J = 1, N
    DO I = 1, N
      S = S + A(J) + B(J,I)
    END DO
  END DO
...
END SUBROUTINE SUB
```

Automatic parallelized

OpenMP parallelized

```
$ nfort -fopenmp -mparallel t.f90
nfort: par(1801): t.f90, line 6: Parallel routine generated.: SUB$1
nfort: par(1803): t.f90, line 6: Parallelized by "do".
nfort: vec( 101): t.f90, line 7: Vectorized loop.
nfort: par(1801): t.f90, line 12: Parallel routine generated.: SUB$2
nfort: par(1803): t.f90, line 13: Parallelized by "do".
nfort: vec( 101): t.f90, line 14: Vectorized loop.
```

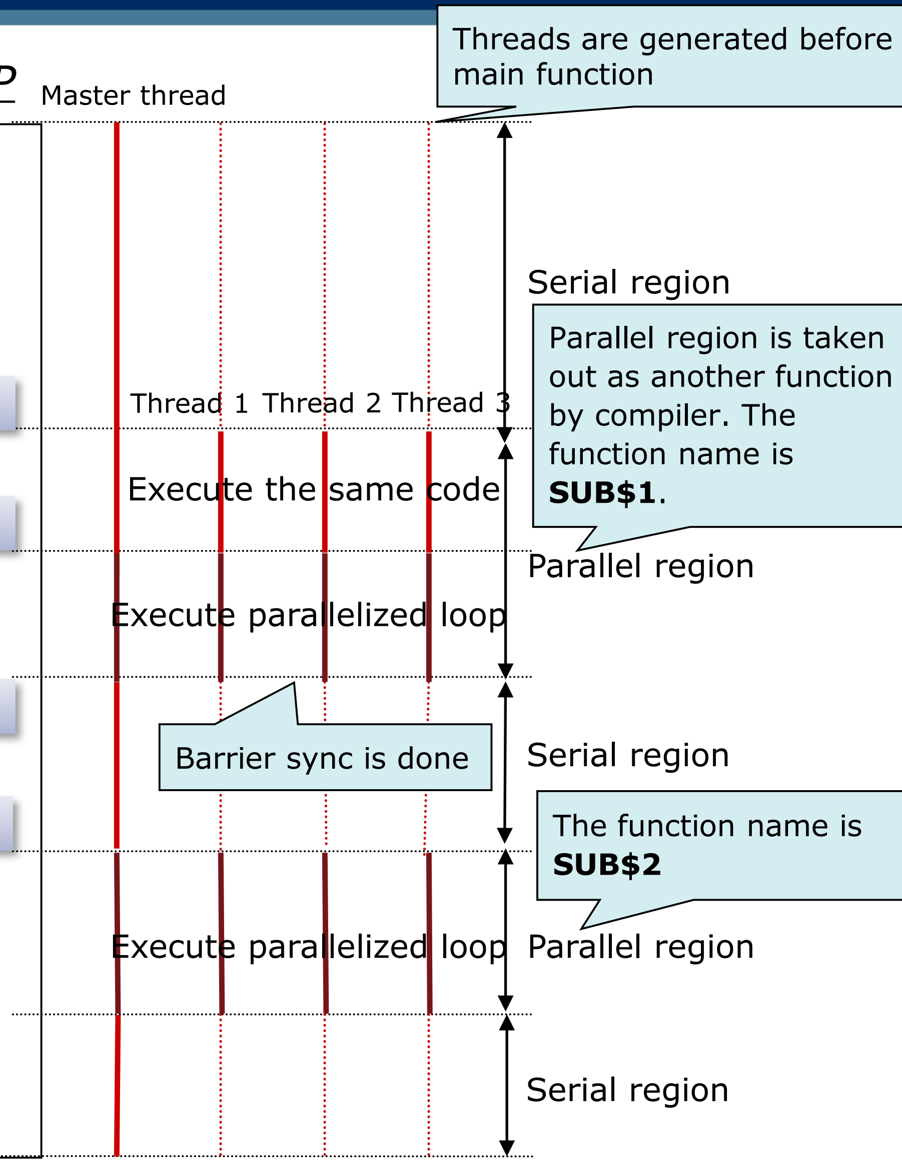
Behavior of Parallelized Program

Execution Image of Program Parallelized with OpenMP

When parallelized with OpenMP

All variables except loop control variables are shared when there are no specifications.

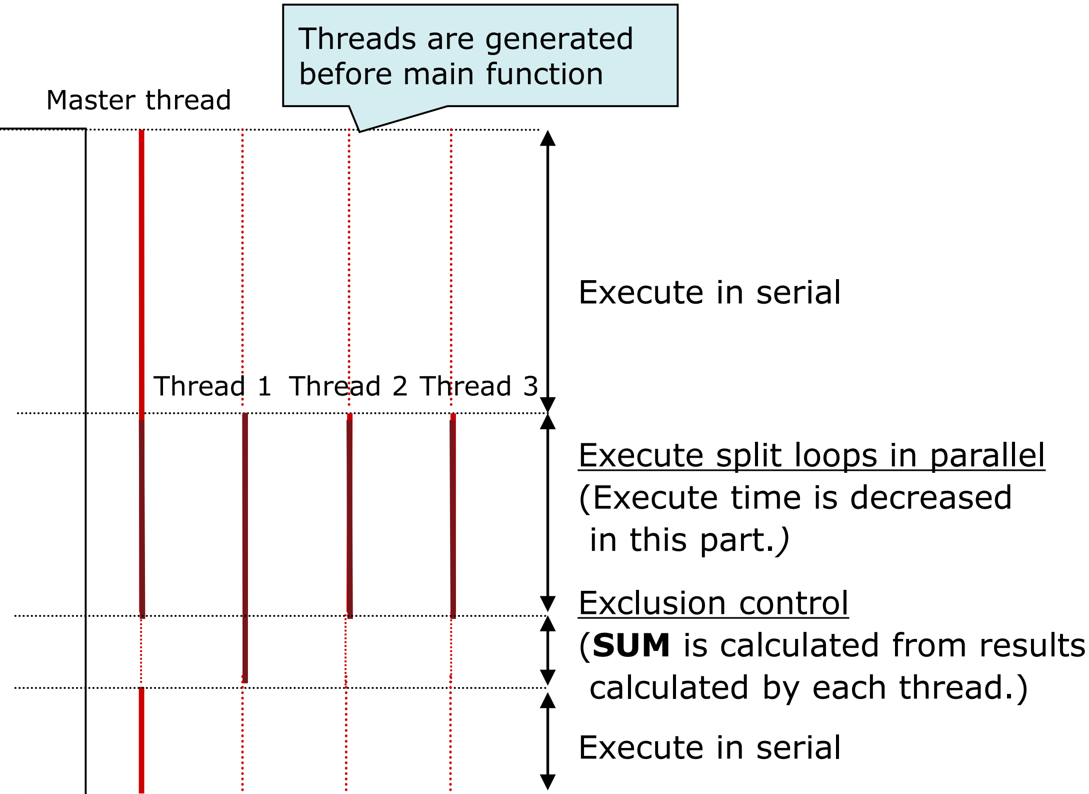
```
SUBROUTINE SUB (A, N)
  INTEGER :: N, I, J
  REAL(KIND=8) :: A(N), B(N)
  REAL(KIND=8) :: SUM = 1.0
  REAL(KIND=8) :: DERIVE

!$OMP PARALLEL PRIVATE(DERIVE)
  DERIVE = 12.3

!$OMP DO
  DO I = 1, N
    B(I) = DERIVE
  ENDDO
!$OMP END PARALLEL
  ...

!$OMP PARALLEL DO REDUCTION(+:SUM)
  DO J = 1, N
    DO I = 1, N
      SUM = SUM + A(J) + B(I)
    ENDDO
  END DO
  RETURN
END SUBROUTINE SUB
```



Note: VE does not support nested parallelism.

Execution Image of Automatically Parallelized Program

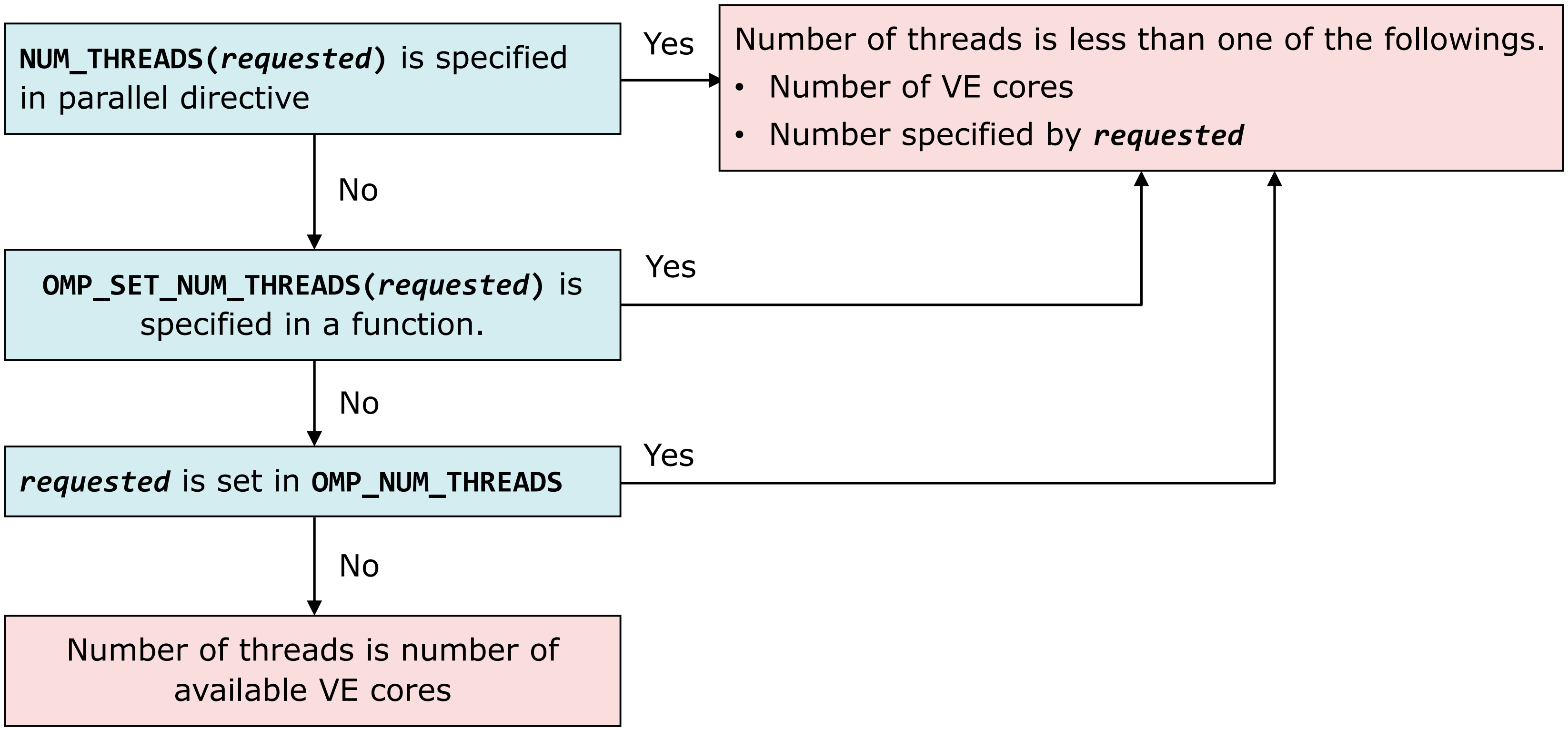
```
SUBROUTINE SUB (A, B, N)
  INTEGER :: N, I, J
  REAL(KIND=8) :: A(N), B(N)
  REAL(KIND=8) :: SUM = 0.0
  ...

 DO J = 1, N
   DO I = 1, N
     SUM = SUM + A(J) + B(I)
   ENDDO
 ENDDO

  RETURN

END SUBROUTINE SUB
```



(Solid line: Program execution, Dashed line: Waiting process)

Decide Number of Threads in OpenMP

- Number of threads used in parallel process is decided by rules as follows.



Note: Even if you requested over 8 threads, the maximum number of threads is 8, because the number of VE cores is 8.

Tuning Parallelized Program

Orchestrating a brighter world NEC

Point of View in Tuning

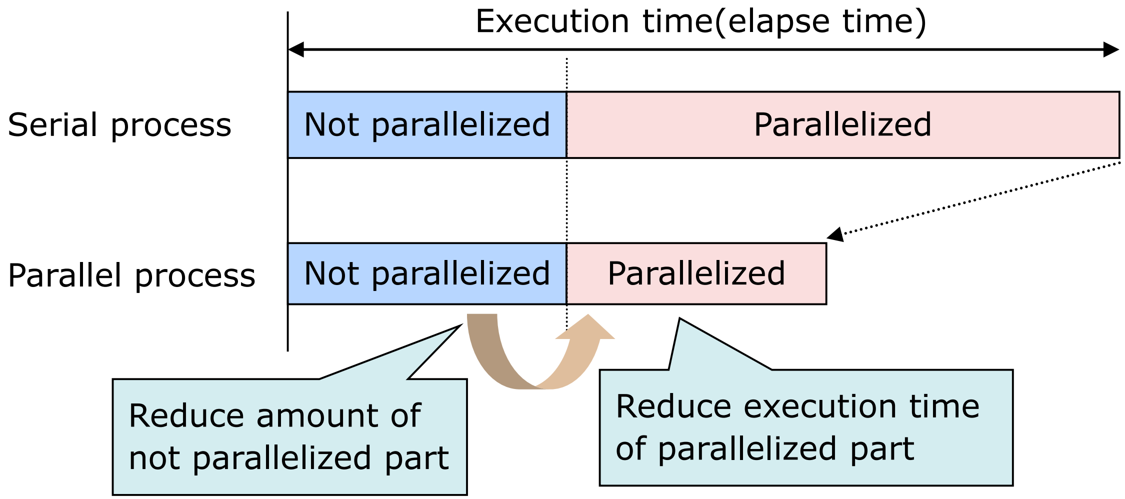
Are there many parts executed in parallel?

- Is the ratio of execution time of parallelizable part to elapse time of whole part executed in single small?

(Increase parallelized execution part/parallelized loop.)

Is parallelized effectively?

- Is execution time of parallelized loop long enough? (Parallelize suitable loops.)
- Is parallelization overhead large? (Reduce overhead.)
- Are workloads of each thread uniform? (Consider process in loops.)



Tuning Flow

Used performance analysis
- **FTRACE**

1. Select loop/procedure targets of parallelization.
   - Find procedures whose execution time is long according to information of **PROGINF** and **FTRACE**.
2. Increase parallelized part.
   - Check if loops in procedures found in **1.** is parallelizable, and add the directives and transform program to parallelize them.
3. Improve load balance.
   - Adjust load balance to make workloads of each thread uniform according to **PROGINF** and **FTRACE** information.

Note: Vectorization should be done enough before parallelization.

Select Loops for Parallelization

In automatic parallelization, correspond loops are selected and parallelized automatically.

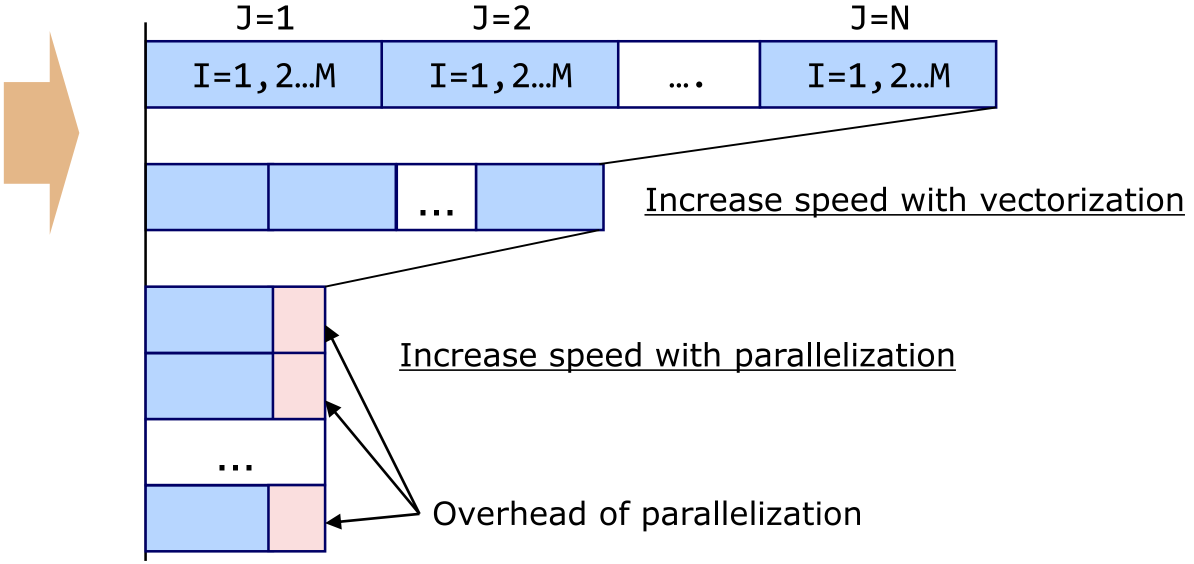
Loops without factors inhibiting parallelization

- Not parallelizable dependencies.
- Not parallelizable control flow.
- Procedure call like I/O procedures whose execution order must be ensured.

Outermost loop in multiple loops

- Loops whose execution time is long.
- Consider to increase speed of innermost loops with vectorization.

```
DO J = 1, N
  DO I = 1, M
    A(I,J) = B(I,J) + C(I,J)
  ENDDO
ENDDO
```



Not Parallelizable Dependencies

Loops where the same array element is defined and referred in different iterations.

Define and refer the same array element

```
DO I = 1, N
   A(I) = B(I+1)
   B(I) = C(I)
ENDDO
```

Iteration	Reference	Definition	
I=1	B(2)	B(1)	Executed in thread 1
I=2	B(3)	B(2)	Executed in thread 1
I=3	B(4)	B(3)	Executed in thread 2
I=4	B(5)	B(4)	Executed in thread 2
:	:	:	

The order of definition and reference of B(3) is undefined.

Loops where the same scalar variable is defined and referred in different iterations.

Same scalar variable

```
DO I = 1, N
   C(I) = T
   T = B(I)
}
```

- Parallelizable if the variable is referred after definition.
- Sum/Product patterns are parallelizable by transforming program, directives and so on. (Compiler recognizes the patterns and parallelizes automatically in automatic parallelization.)

Variable defined under a condition is referred out of it.

```
DO J = 1, N
  DO I = 1, M
    IF (A(I,J) .GE. D ) THEN
      T = A(I,J) - D
    ENDIF
    C(I,J) = T
  ENDDO
ENDDO
```

- Variable **T** is defined in **IF** branch. Defined value is referred in iterations.
- This case is not parallelizable even if the variable is referred after definition.

Not Parallelizable Control Flow

Jump from loops

- Not parallelizable because iterations must not be executed after that when condition for jumping is true.

```
DO J = 1, N
  DO I = 1, N
    IF (A(I,J) < 0.0 ) GOTO 100
    B(I,J) = SQRT(A(I,J))
  ENDDO
ENDDO
100 CONTINUE
```

Add Directives to Promote Parallelization

```
$ nfort -mparallel -fdiag-parallel=2 a.f90 -c
nfort: vec( 103): a.f90, line 5: Unvectorized loop.
```

- Loops including a procedure call is not parallelized automatically because it is unknown if the procedure can be executed in parallel.
- If the procedure can be executed in parallel, specify the directive **CNCALL** to parallelize automatically the loop.

```
SUBROUTINE SUB(A, B, N)
  INTEGER :: N, I
  REAL :: A(N), B(N), C(N)

  DO I = 1, N
    C(I) = FUNC(A(I), B(I))
  ENDDO
END
```

→

```
SUBROUTINE SUB(A, B, N)
  INTEGER :: N, I
  REAL :: A(N), B(N), C(N)
!NEC$ CNCALL
  DO I = 1, N
    C(I) = FUNC(A(I), B(I))
  ENDDO
END
```

```
$ nfort -mparallel -fdiag-parallel=2 a.f90 -c
nfort: par(1801): a.f90, line 5: Parallel routine generated.: SUB$1
nfort: par(1803): a.f90, line 5: Parallelized by "do".
nfort: vec( 103): a.f90, line 5: Unvectorized loop.
```

Forced Parallelization Directive

- Not parallelized in automatic parallelization.
- It is ensured that a correct result can be obtained even in parallel execution.

- Specify forced parallelization directive **PARALLEL DO** to parallelize.
  - Enable to specify parallelization for loops and statement list.
  - Compiler ignore data dependencies and parallelize them.

Programmer must ensure that the correct result can be obtained in parallel execution.

```
SUBROUTINE SUB(A, B, N)
  INTEGER :: N, I, J
  REAL :: A(N), B(N), X(N), WK(256)
  REAL :: SUM = 0.0
!NEC$ PARALLEL DO PRIVATE(WK)
  DO I = 1, N
    DO J = 1, N
      WK(I) = A(I) + B(J)
    ENDDO
    CALL SUB1(X(J), WK)
!NEC$ ATOMIC
    SUM = SUM + X(J)
  ENDDO
END SUBROUTINE SUB
```

Specify **ATOMIC** right before statements which need to be processed exclusively like sum and accumulation in forced parallelized loops.

Specify forced parallelization on a loop
Specify variables and arrays used for work in **PRIVATE** clause.

Overhead of Parallelization

Overhead: Increased execution time by parallelizing a program.

- Execution time of the process added by a programmer to parallelize a program.
  - Increased time by transforming a program.
  - Processing time of runtime libraries to control parallelization.
- Waiting time of exclusive control in system libraries.
  - Waiting time for exclusive control in system library functions to update and refer system data.
    - File I/O functions, **MALLOC()** and so on.
- Waiting time for barrier sync with other threads.

Exclusive Control in System Libraries

- Exclusive control is executed to inhibit the other OpenMP threads from updating data used in whole program at the same time when they are referred or updated.
  - File descriptor, management data of area allocated with **`MALLOC`**`()` and so on.

↓

- Reduce procedure calls in system libraries.
  - Put together **`MALLOC`**`()` as much as possible.
  - Declare the data used in a procedure as local data to allocate them in stack.
  - Read file contents, map them on memory and read required data from memory when there are enough available area in memory.

Reduce Waiting Time for Barrier Sync (1)

In OpenMP, barrier sync is executed automatically at places as follows.

- End of parallel loop without **NOWAIT** clause.
- End of parallel loop with **REDUCTION** clause.(*)
- Beginning of parallel region with **COPYIN** clause.(*)
- End of parallel region.(*)

In automatic parallelization, compiler makes implicit barrier sync properly.

In the cases (*), barrier sync cannot be omitted because of the mechanism of parallel process.

Make workloads of each thread uniform. (Reduce waiting time)

- **SCHEDULE(DYNAMIC)** clause is effective to make workloads of parallel loop uniform which changes in each iteration.

```
!$OMP DO SCHEDULE( STATIC )
DO J = 1, M
  DO I = 1, N
    ...
  ENDDO
ENDDO
```

```
!$OMP DO SCHEDULE( DYNAMIC )
DO J = 1, M
  DO I = 1, N
    ...
  ENDDO
ENDDO
```

Reduce Waiting Time for Barrier Sync (2)

- Remove implicit barrier sync by combining parallel regions.
- Remove unnecessary barrier sync by specifying **NOWAIT** clause.
  - Compiler ignores **NOWAIT** clause if it is specified on barrier sync unable to be removed.

```
SUBROTINE SUB()
 ...
!OMP$ PARALLEL DO
  DO I = 1, N
    A(I) = ...
  ENDDO
  X = 0.0
!OMP$ PARALLEL DO
  DO I = 1, N*2
    B(I) = ...
  ENDDO
    ...
END SUBROTINE SUB
```

Combine parallel regions

```
SUBROTINE SUB()
 ...
  X = 0.0;
!OMP$ PARALLEL
!OMP$ DO
  DO I = 1, N
    A(I) = ...
  ENDDO
!OMP$ DO
  DO I = 1, N*2
    B(I) = ...
  ENDDO
!OMP$ END PARALLEL
    ...
END SUBROTINE SUB
```

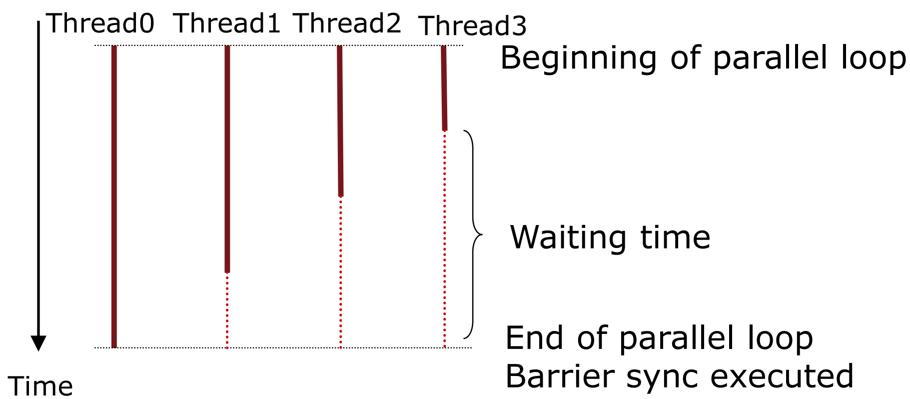
Specify **NOWAIT** clause

```
SUBROTINE SUB()
 ...
  X = 0.0;

!$OMP PARALLEL
!$OMP DO  NOWAIT
  DO I = 1, N
    A(I) = ...
  ENDDO
!$OMP DO  NOWAIT
  DO I = 1, N*2
    B(I) = ...
  ENDDO
!$OMP END PARALLEL
    ...
END SUBROTINE SUB
```

Improve Load Balance (1)

▌There is much waiting time at the end of a loop as follows because the workloads of each thread are not uniform.

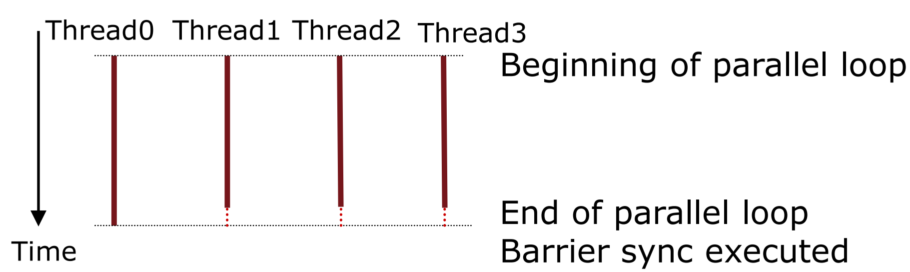
When parallel loop is split to 4 and they are executed by 4 threads




```
!$OMP DO
    DO J = 1024, 1, -1
      DO I = 1, J
          …
      ENDDO
    ENDDO
```

Iteration of inner loop or calculation amount decreased as the iteration of parallelized loop goes forward.

Improve load balance



All calculation can be done in shorter time by making workloads of each thread uniform and reducing waiting time.

Improve Load Balance (2)

Split parallel region into smaller parts and assign them to each thread to make their workloads uniform.

OpenMP parallelization

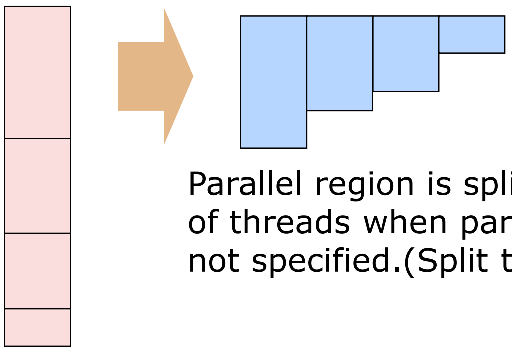
- Adjust parameter of **SCHEDULE** clause.

```
!$OMP DO SCHEDULE(DYNAMIC,4)
    DO J = 1024, 1, -1
        DO I = 1, J
            ...
        ENDDO
    ENDDO
```

Automatic parallelization

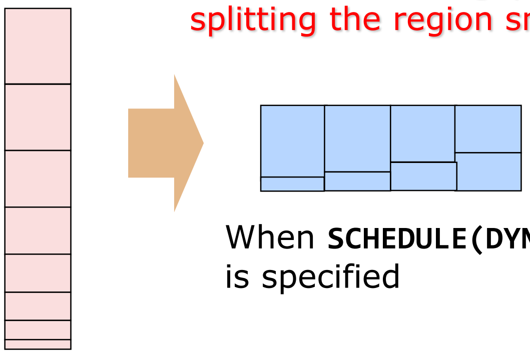
- Adjust parameter of **SCHEDULE** clause in **CONCURRENT** directive as well as OpenMP.

```
!NEC$ CONCURRENT SCHEDULE(DYNAMIC,4)
    DO J = 1024, 1, -1
        DO I = 1, J
            ...
        ENDDO
    ENDDO
```



Parallel region is split to number of threads when parameter is not specified.(Split to four)

Enable to reduce gap by splitting the region smaller.



When **SCHEDULE(DYNAMIC,4)** is specified

Make the number of regions as less as possible because the more it increases, the more time it takes to control threads.

FTRACE

Load balance in procedures are shown in information for each thread.

```
REQUENCY  EXCLUSIVE       AVER.TIME     MOPS    MFLOPS  V.OP  AVER.     VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                      RATIO V.LEN      TIME     MISS     CONF HIT E.%

    60000     62.177( 73.1)     1.036 100641.4  79931.0  99.55 248.5    62.134    0.023     0.000  100.00 SUBX$1
    15000      4.467(  5.3)     0.298 107076.2  83033.3  99.47 248.4     4.455    0.005     0.000  100.00  -thread0
    15000     11.552( 13.6)     0.770 104082.7  82404.6  99.54 248.5    11.542    0.006     0.000  100.00  -thread1
    15000     19.000( 22.3)     1.267 101390.4  80683.3  99.55 248.6    18.990    0.006     0.000  100.00  -thread2
    15000     27.157( 31.9)     1.810  97595.1  77842.2  99.56 248.6    27.147    0.006     0.000  100.00  -thread3
    15000     22.711( 26.7)     1.514   1426.9      0.0   0.00   0.0     0.000    0.015     0.000    0.00 SUBX
...
----------------------------------------------------------------------------------------------------------------
    79001     85.034(100.0)     1.076  74062.7  58500.4  98.89 248.5    62.249    0.043     0.000  100.00 total
```

Specify **!NEC$ CONCURRENT SCHEDULE(DYNAMIC, 4)** right before the outermost loop

```
REQUENCY  EXCLUSIVE       AVER.TIME     MOPS    MFLOPS  V.OP  AVER.     VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                      RATIO V.LEN      TIME     MISS     CONF HIT E.%

    60000     66.872( 99.6)     1.115  93599.2  74318.7  99.52 248.5    64.077    1.418     0.000  100.00 SUBX$1
    15000     16.766( 25.0)     1.118  92992.0  73842.7  99.52 248.5    16.022    0.409     0.000  100.00  -thread0
    15000     16.697( 24.9)     1.113  91671.0  72790.7  99.52 248.5    16.000    0.397     0.000  100.00  -thread1
    15000     16.714( 24.9)     1.114  94854.7  75312.8  99.52 248.5    16.040    0.305     0.000  100.00  -thread2
    15000     16.695( 24.9)     1.113  94880.7  75329.6  99.51 248.5    16.014    0.307     0.000  100.00  -thread3
    15000      0.129(  0.2)     0.009   1284.5      0.1   0.00   0.0     0.000    0.010     0.000    0.00 SUBX
...
----------------------------------------------------------------------------------------------------------------
    79001     67.148(100.0)     0.850  93334.5  74082.8  99.51 248.5    64.192    1.430     0.000  100.00 total
```

Before :EXCLUSIVE TIME are ununiform for **-thread0** to **-thread3** of **SUBX$1**.(Load imbalance)
After :EXCLUSIVE TIME are uniform for each thread and that of **SUBX** is shorter (time for barrier sync and so on are reduced) although that of **SUBX$1** increases because of time to control threads.

Notes on Using Parallelization

Allocated Area by ALLOCATE Statement

- Whether the areas allocated by **ALLOCATE** statement are shared or private is decided as follows.
  - Are allocated arrays or pointers shared or private?
  - Is process executed in parallel when the area is allocated?

P,Q : shared
R,S : private

Parallel process section

```
SUBROUTINE SUB()
  REAL,ALLOCATABLE :: P(:), Q(:)
  REAL,ALLOCATABLE :: R(:), S(:)

  ALLOCATE(P(16))
!$OMP PARALLEL PRIVATE(R,S)
!$OMP SINGLE
  ALLOCATE(Q(16))
!$OMP END SINGLE
!$OMP MASTER
  ALLOCATE(R(16))
!$OMP END MASTER
  ALLOCATE(S(16))
!$OMP END PARALLEL
END SUBROUTINE SUB
```

ALLOCATE(P(16)) is executed once. P is shared, so all threads refer the same area.

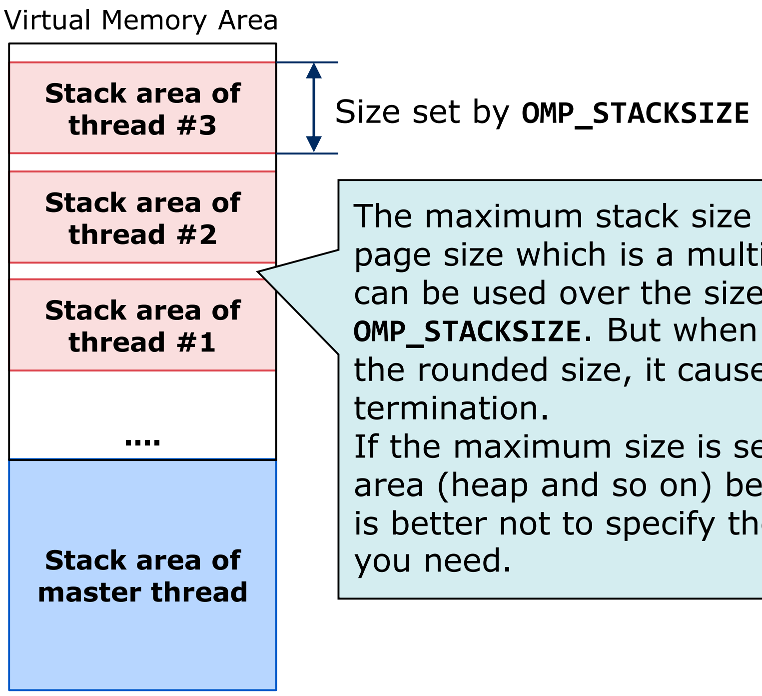
ALLOCATE(Q(16)) is executed by one thread and only one area is allocated. Q is shared, so all threads refer the same area.

ALLOCATE(R(16)) is executed by only master thread and only one area is allocated. R is private, so R is still unallocated in threads other than master thread.

ALLOCATE(S(16)) is executed by all threads and four areas are allocated. S is private, so each thread uses separate areas.

Huge Local Array

- When huge local array is used in a parallel region, set the environment variable **OMP_STACKSIZE** to a value which is larger than the size of the array.
  - **OMP_STACKSIZE** is an environment variable which sets the maximum stack size of threads other than master thread. If this is not set, the maximum stack size is 4MByte.
  - If the size of array is exceeded the size of unused area on stack, the program is terminated abnormally.



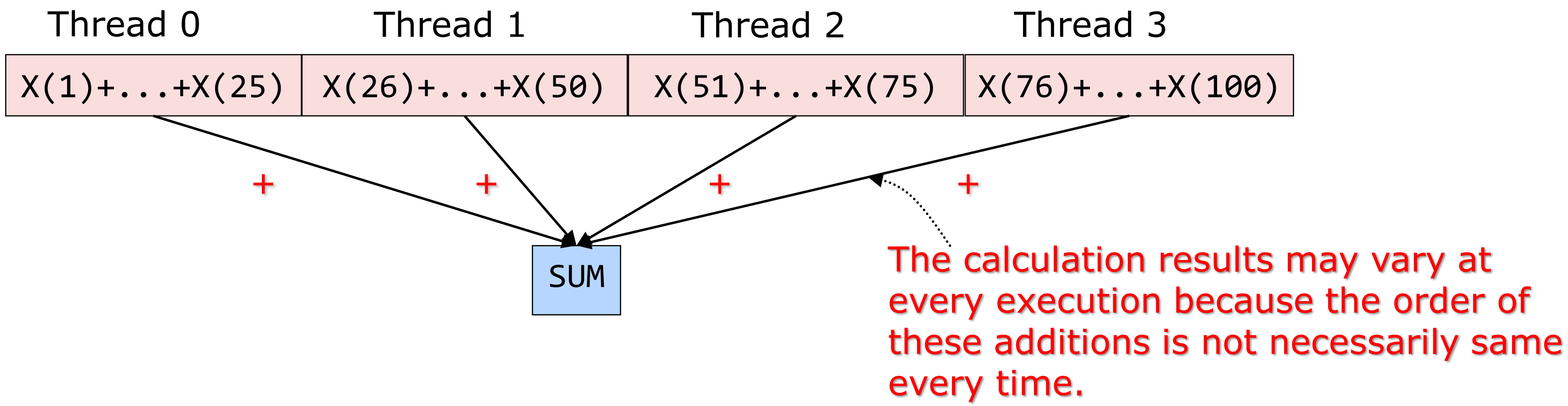
The maximum stack size is rounded to the page size which is a multiple of 64MB. So it can be used over the size specified by **OMP_STACKSIZE**. But when the used size is over the rounded size, it causes abnormal termination.
If the maximum size is set too large, the other area (heap and so on) becomes smaller. So it is better not to specify the larger size than you need.

```
$ cat a.f90
PROGRAM MAIN
...
!$OMP PARALLEL
  CALL SUB()
!$OMP END PARALLEL
...
END PROGRAM

SUBROTINE SUB()
  REAL(KIND=8) :: X(16*1024*1024)
  REAL(KIND=8) :: Y(16*1024*1024)
...
END SUBROUTINE SUB
...
$ nfort -fopenmp a.f90
$ export OMP_STACKSIZE=384M
$ ./a.out
```

Sum Operation

- Sum operation can be parallelized but the order of additions can be changed every time because the order of execution of each threads is not constant.(Execution order is not ensured.)
  - Calculation result may differ in operation error range from it in serial execution, or may vary at every execution in parallel.

```
DO I = 1, 100
  SUM = SUM + X(I)
ENDDO
```



\Orchestrating a brighter world

NEC