Chapter 4
Fortran

FUJITSU LIMITED
April 2016

Contents

- How to Compile/Execute a Program
- Aspects of Compile Information
- Typical Compiler Optimizations
  - SIMD Optimization
  - Software Pipelining
  - Loop Optimization
  - Automatic Parallelization
- Recommended Options (Effects and Impact)
  - Description of Recommended Options
  - Options with an Impact on Operation Results and Execution Results
  - Other Compile Options
  - Optimization Options
- Other Optimization Options
- Debug Functions

How to Compile/Execute a Program

- How to Compile a Program
- How to Execute a Program

* See "Chapter 2 Compiling and Linking Fortran Programs" and "Chapter 3 Executing Fortran Programs" in the *Fortran User's Guide* for details.

How to Compile a Program

- **Fortran**

Format (cross): **frtpx** [option list] [file name list]

(own) : **frt** [option list] [file name list]

■Specify the following options when using thread parallelization processing:

-Kparallel	To use automatic parallelization
-Kopenmp	To use OpenMP

Compile example:

```
$ frtpx sample.f90                      (Compile with sequential processing)

$ frtpx -Kparallel sample.f90           (Compile with automatic parallelization)
```

How to Execute a Program (Sequential)

● Sequential execution

Execute the executable module generated at compilation.

```
./{executable module name}   [argument]
```

Example: Script for sequential execution

```
#!/bin/sh
#PJM -L "node=1"                              # Number of nodes

./a.out
```

* The files for standard output and standard error output of the job are as follows:
{pjm-script}.o{req-id}: Standard output
{pjm-script}.e{req-id}: Standard error output

pjm-script: PJM script name, req-id: Request number

How to Execute a Program (Thread Parallelization Processing)

● Thread parallelization execution (automatic parallelization)

Specify the number of threads for parallel execution in the environment variable PARALLEL, and execute the program.

```
PARALLEL={number of threads} ;export PARALLEL
```

Example: Script for executing automatic parallelization

```
#!/bin/sh
#PJM -L "node=1"           # Number of nodes
PARALLEL=16 ;export PARALLEL
./a.out
```

● Thread parallelization execution (OpenMP)

Specify the number of threads for parallel execution in the environment variable OMP_NUM_THREADS, and execute the program.

```
OMP_NUM_THREADS={number of threads} ;export OMP_NUM_THREADS
```

Example: Script for OpenMP execution

```
#!/bin/sh
#PJM -L "node=1"                     # Number of nodes
OMP_NUM_THREADS=16 ;export OMP_NUM_THREADS
./a.out
```

Aspects of Compile Information

- Aspects of Compile Information
- Operation Confirmation for Optimization

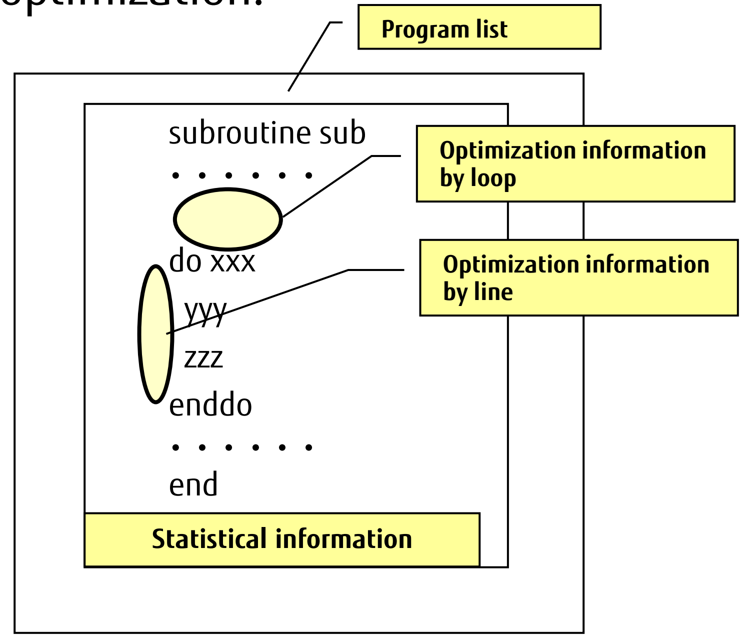
* See “4.1.2 Compilation Information” in the *Fortran User’s Guide* for details.

Aspects of Compile Information

Aspects of compile information and an output example are described below to provide prerequisite knowledge for compiler optimization.

The following information is output as compile information:

- Program list
- Optimization information by loop
- Optimization information by line
- Statistical information
- Error message



Compiler options format:

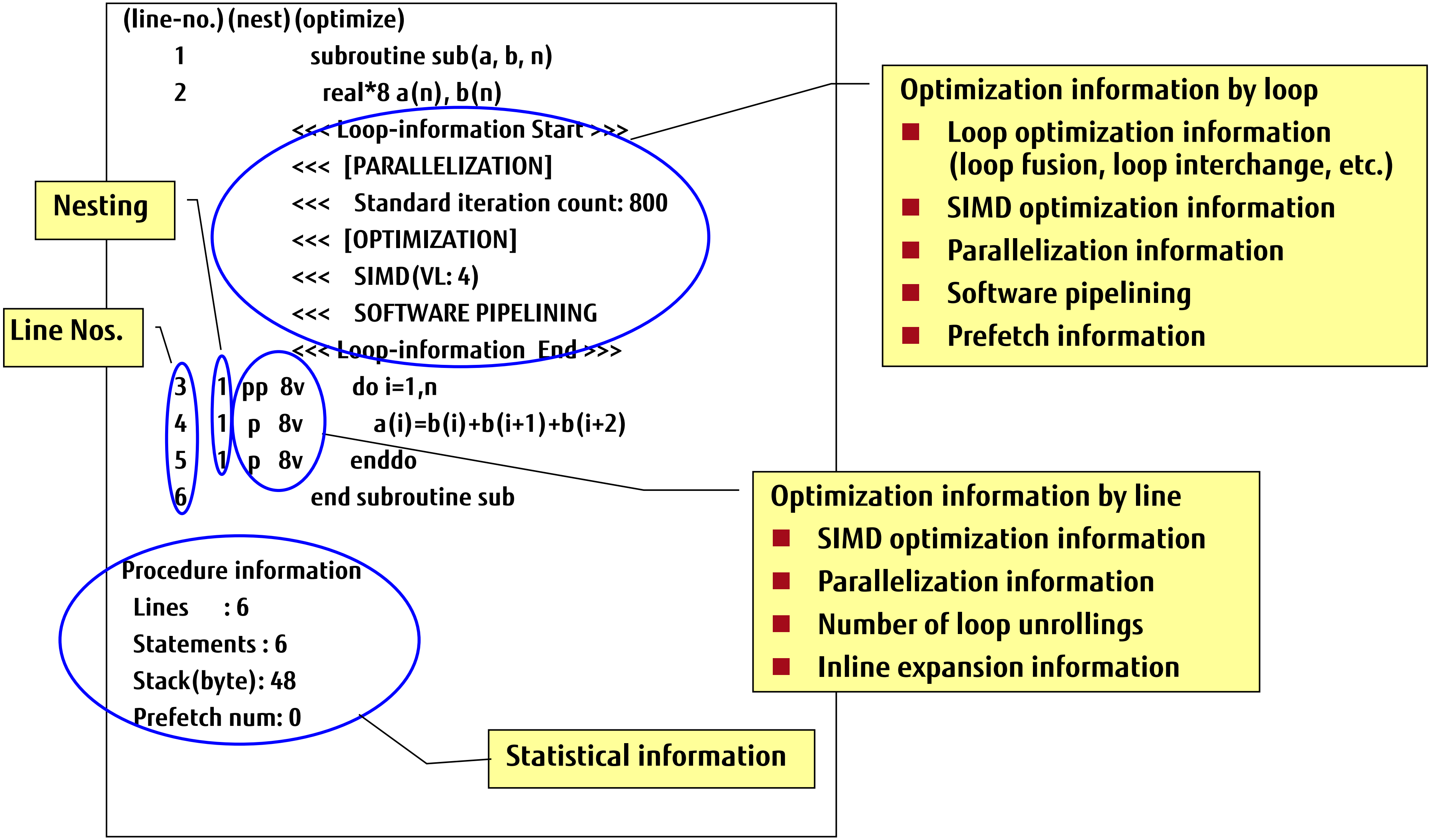
-Nlst[=*lst_arg*] *lst_arg*:{ a | d | i | m | p | t | x }

Compile example

```
$ frtpx –Kfast,parallel –Nlst=t sample.f90
```

⇨ sample.lst is output as compile information.

Operation Confirmation for Optimization



```
(line-no.) (nest) (optimize)
     1                  subroutine sub(a, b, n)
     2                    real*8 a(n), b(n)
                        <<< Loop-information Start >>>
                        <<<  [PARALLELIZATION]
                        <<<    Standard iteration count: 800
                        <<<  [OPTIMIZATION]
                        <<<    SIMD(VL: 4)
                        <<<    SOFTWARE PIPELINING
                        <<< Loop-information  End >>>
     3      1  pp  8v       do i=1,n
     4      1   p  8v         a(i)=b(i)+b(i+1)+b(i+2)
     5      1   p  8v       enddo
     6                  end subroutine sub

  Procedure information
    Lines      : 6
    Statements : 6
    Stack(byte): 48
    Prefetch num: 0
```

Optimization information by loop

- Loop optimization information (loop fusion, loop interchange, etc.)
- SIMD optimization information
- Parallelization information
- Software pipelining
- Prefetch information

Optimization information by line

- SIMD optimization information
- Parallelization information
- Number of loop unrollings
- Inline expansion information

Statistical information

Typical Compiler Optimizations

- SIMD Optimization
- Software Pipelining
- Loop Optimization
- Automatic Parallelization

SIMD Optimization
(Single Instruction Multiple Data)

- Basic Principles of SIMD Optimization
- SIMD Optimization of Contiguous Data
- Use of Single Precision and Double Width (8 SIMD)
- SIMD Optimization of Stride and Indirect Access
- SIMD Optimization of Complex Types
- Masked SIMD Optimization
- Confirmation of SIMD Optimization
- Loops Suitable for SIMD Optimization

* See “9.1.1.7 SIMD” in the *Fortran User’s Guide* for details.

Basic Principles of SIMD Optimization

- SIMD (Single Instruction Multiple Data)
  - Parallel processing of multiple operations from a single instruction
- SIMD features of SPARC64™ XIfx
  - Parallel processing of four operations from a single instruction
  - Support of multiply-add operations
  - Capable of executing two SIMD instructions concurrently with a single core (double-precision instructions)

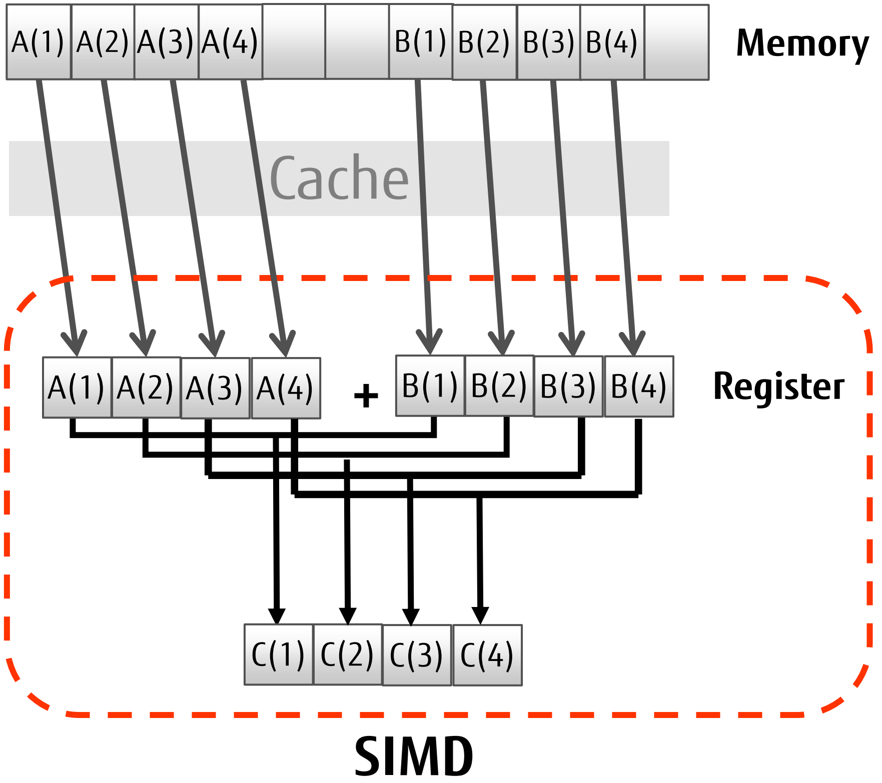
Capable of processing 16 operations concurrently on a single core (512 operations by 1 chip)

- Double-precision SIMD load acceptable even at the 8-byte boundary
- Capable of processing 32 operations concurrently, in the case of single precision
- SIMD optimization also possible for the integer type

Achieves "easy-to-use SIMD for applications" and "acceleration of computing"

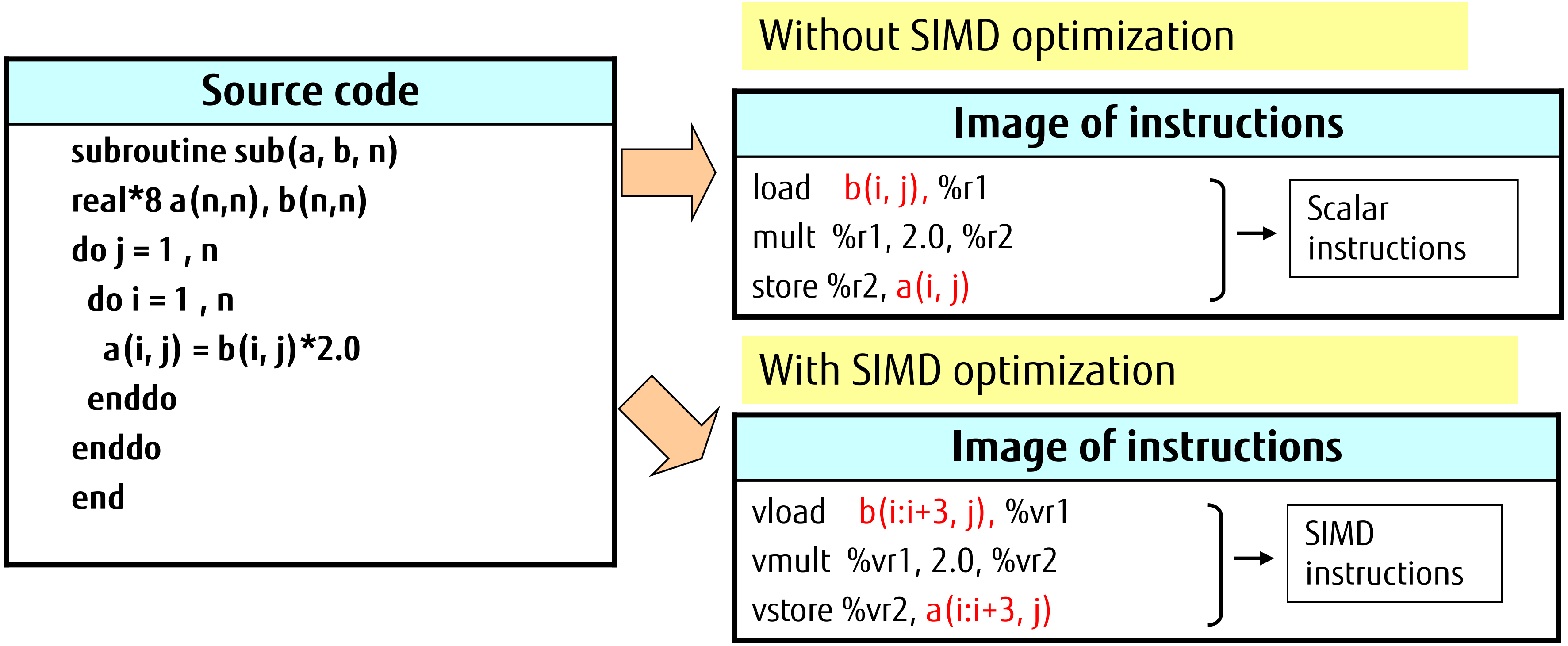
Program example

```
 do i=1,4
   c(i)=a(i)+b(i)
 enddo
```



SIMD Optimization of Contiguous Data

SIMD optimization of contiguous data



Source code

```
subroutine sub(a, b, n)
real*8 a(n,n), b(n,n)
do j = 1 , n
 do i = 1 , n
  a(i, j) = b(i, j)*2.0
 enddo
enddo
end
```

Without SIMD optimization

Image of instructions

```
load   b(i, j), %r1
mult  %r1, 2.0, %r2
store %r2, a(i, j)
```

→ Scalar instructions

With SIMD optimization

Image of instructions

```
vload   b(i:i+3, j), %vr1
vmult  %vr1, 2.0, %vr2
vstore %vr2, a(i:i+3, j)
```

→ SIMD instructions

A single instruction execute the data of four elements, like b(i:i+3, j).

Conversion of the array elements of the innermost loop and the array elements of the second, third, and fourth iterations into SIMD instructions reduces the number of executed instructions to one-fourth, so processing is accelerated.

Use of Single Precision and Double Width (8 SIMD)

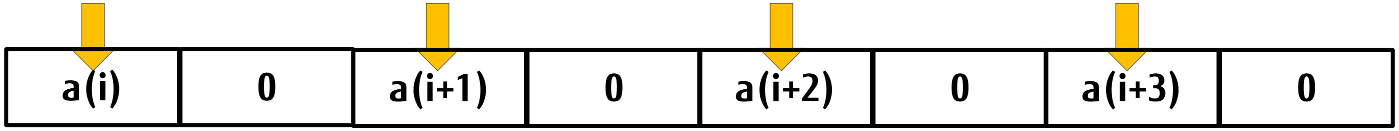
Source code

```
subroutine sub(a,b,c,n)
real(4),dimension(1:n) :: a,b,c
do i=1,n
  a(i) = b(i) ● c(i)
enddo
end subroutine
```

The processor supports the use of double width only for the following instructions for single-precision floating-point operations:

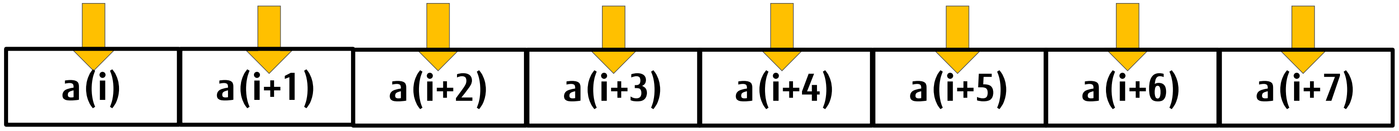
- FMA
- Add
- Subtract
- Multiply

Single-precision floating-point 4 SIMD load



a(i)	0	a(i+1)	0	a(i+2)	0	a(i+3)	0

Single-precision floating-point 8 SIMD load



a(i)	a(i+1)	a(i+2)	a(i+3)	a(i+4)	a(i+5)	a(i+6)	a(i+7)

If an instruction other than the supported ones appears, 4 SIMD execution is performed.

SIMD Optimization of Stride and Indirect Access

Source code

```
subroutine sub(a,b,m,n)
real(8),dimension(1:n) :: a,b
integer(4),dimension(1:n) :: m
do i=1,n,2
  a(i) = b(m(i))
enddo
end subroutine
```

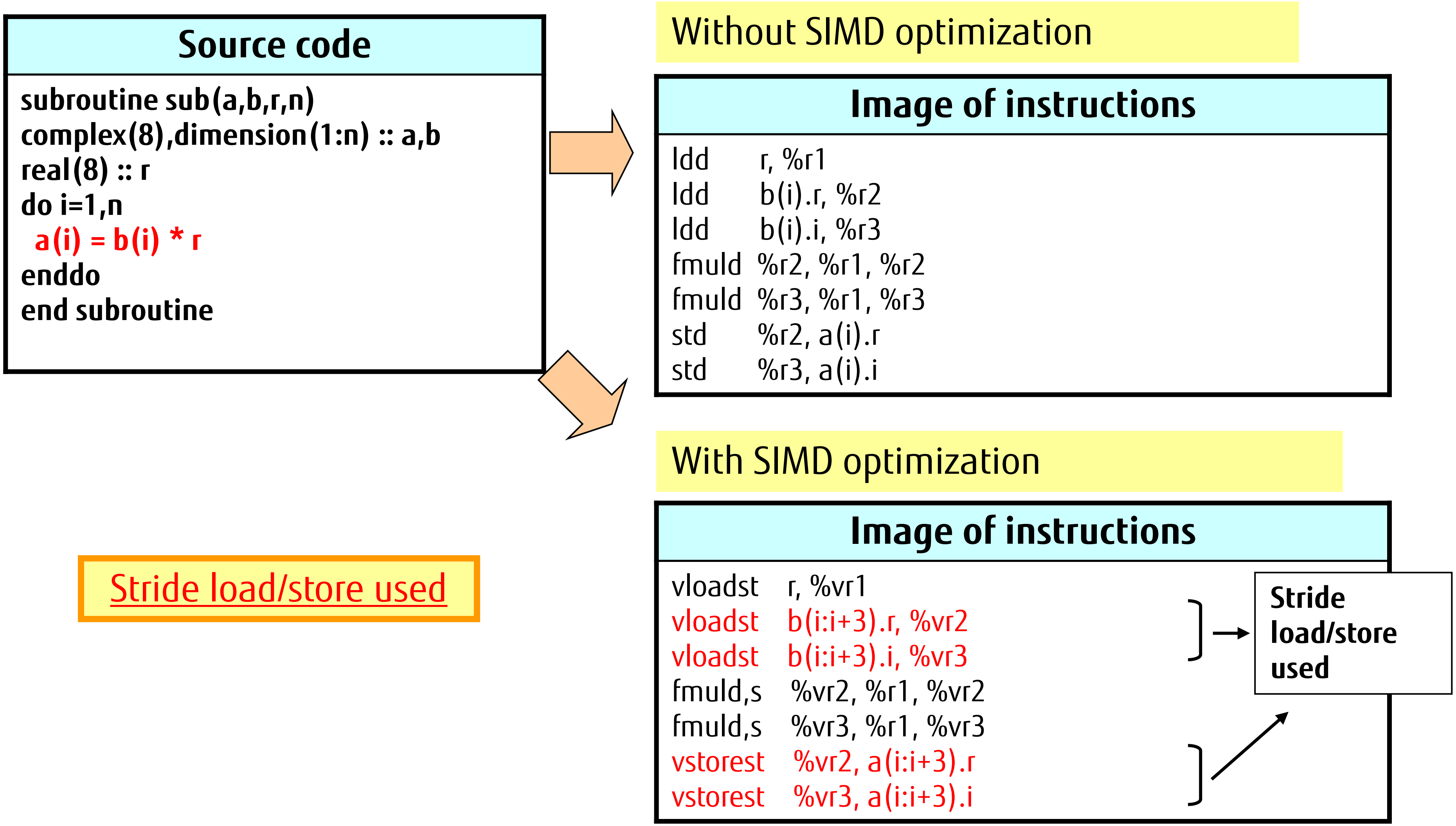
Image of instructions

vloadst	m(i)@2,%vr1	Stride access for loading array m
vloadid	b(%vr1),%vr2	Indirect access for loading array b
vstorest	%vr2,a(i)@2	Stride access for storing array a

Stride lengths ranging from 2 to 7 are available for stride load and store. For a stride length greater than the upper limit, use indirect load and store.

SIMD Optimization of Complex Types

- SIMD optimization of complex types



Source code

```
subroutine sub(a,b,r,n)
complex(8),dimension(1:n) :: a,b
real(8) :: r
do i=1,n
 a(i) = b(i) * r
enddo
end subroutine
```

Without SIMD optimization

Image of instructions

```
ldd     r, %r1
ldd     b(i).r, %r2
ldd     b(i).i, %r3
fmuld   %r2, %r1, %r2
fmuld   %r3, %r1, %r3
std     %r2, a(i).r
std     %r3, a(i).i
```

With SIMD optimization

Image of instructions

```
vloadst   r, %vr1
vloadst   b(i:i+3).r, %vr2
vloadst   b(i:i+3).i, %vr3
fmuld,s   %vr2, %r1, %vr2
fmuld,s   %vr3, %r1, %vr3
vstorest  %vr2, a(i:i+3).r
vstorest  %vr3, a(i:i+3).i
```

Stride load/store used

Stride load/store used

Masked SIMD Optimization

FUJITSU

Masked SIMD optimization

Masked SIMD optimization available

Source code

```
subroutine sub(a, b, c, x, n)
 real*8  a(n), b(n), c(n), x(n)
 do i = 1 , n
   if ( x(i) .gt. 0.0 ) then
      a(i) = b(i) * c(i)
     else
      a(i) = b(i) / c(i)
   endif
 enddo
 end
```

Image of instructions

```
vgt          x(i:i+3) , 0.0, %vr1         } (1)
vnot1         %vr1, %vr2
: Operations                                 (2)
vmaskstore %vr3, %vr1, a(i:i+3)          } (3)
vmaskstore %vr4, %vr2, a(i:i+3)
```

You can remove a conditional branch instruction by using a masked instruction as follows.

(1) Compare FP registers, and write the results into the FP registers. (Create a mask.)

(2) Execute the operations.

(3) Selectively store FP register values in memory according to the FP register values.

Use a masked instruction to enable optimization by software pipelining and to increase the efficiency of execution of a loop containing an IF construct.

To use a masked SIMD instruction, the -Ksimd=2 option may be required.

Depending on the true ratio of the IF construct, execution performance may deteriorate because of redundant execution of the instruction in the IF construct.

Confirmation of SIMD Optimization

Example of compile information output

```
(line-no.)(nest)(optimize)
                                      :
3                real*8 a(n,n), b(n,n)
4    1            do j = 1 , n
               <<< Loop-information Start >>>
               <<<  [OPTIMIZATION]
               <<<   SIMD(VL: 4)
               <<< Loop-information  End >>>
5    2    v          do i = 1 , n
6    2    v             a(i, j) = b(i, j)*2.0
7    2    v          enddo
8    1            enddo
```

DO loop SIMD optimization information

- v: SIMD-optimized
- m: Includes SIMD-optimized part and part that is not SIMD-optimized
- s: Not SIMD-optimized
- Blank: Not subject to SIMD optimization

4 SIMD-optimized

Executable statement SIMD optimization information

- v: Can be SIMD-optimized
- m: Includes part that can be SIMD-optimized and part that cannot be SIMD-optimized
- s: Cannot be SIMD-optimized

Loops Suitable for SIMD Optimization

- Loop characteristics required for SIMD optimization by the compiler
  - The compiler should be able to determine the number of loop iterations before entering the loop.
  - There should be no branches (e.g., if statement) in the loop. (*1)
  - There should be no subroutine calls in the loop. (*2)
  - There should be no overlap of array areas between the right-hand and left-hand sides of an expression in the loop.
  - Computation number n of the loop should not depend on the results of computation number n-1.

(*1) In some cases, the effect of masked SIMD may depend on the true ratio of an if statement or other conditions.

(*2) In some cases, inline expansion may enable SIMD optimization.

Software Pipelining

- Basic Principles of Software Pipelining
- Confirmation of Software Pipelining

* See “9.1.1.6 Software Pipelining” in the *Fortran User’s Guide* for details.

Basic Principles of Software Pipelining

- Software pipelining improves parallelism at the instruction level in a kernel loop by overlapping the next iteration of the loop.

The loop that is the most representative of the features of the program is called the kernel loop.

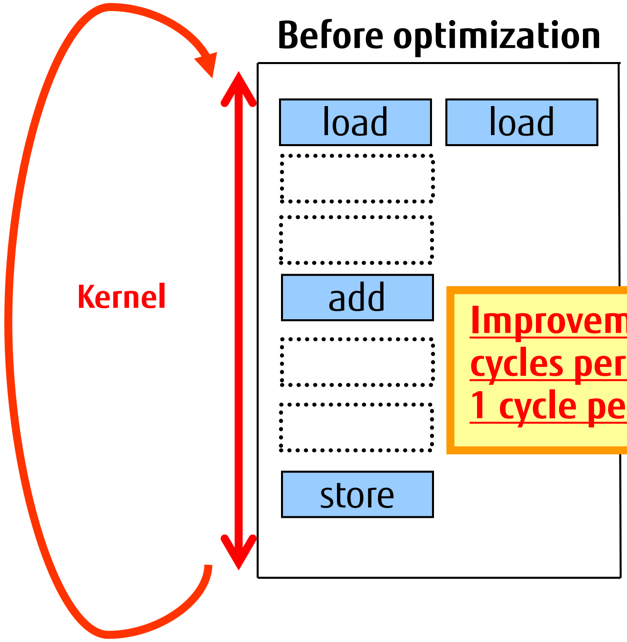
Machine model assumed in the conceptual description

Latency of load = 3 cycles
Latency of add = 3 cycles
Latency of store = 1 cycle
Number of operation units for load and store = 3
Number of commits = 4 (For load and store, up to 3 can be issued simultaneously.)

Program example

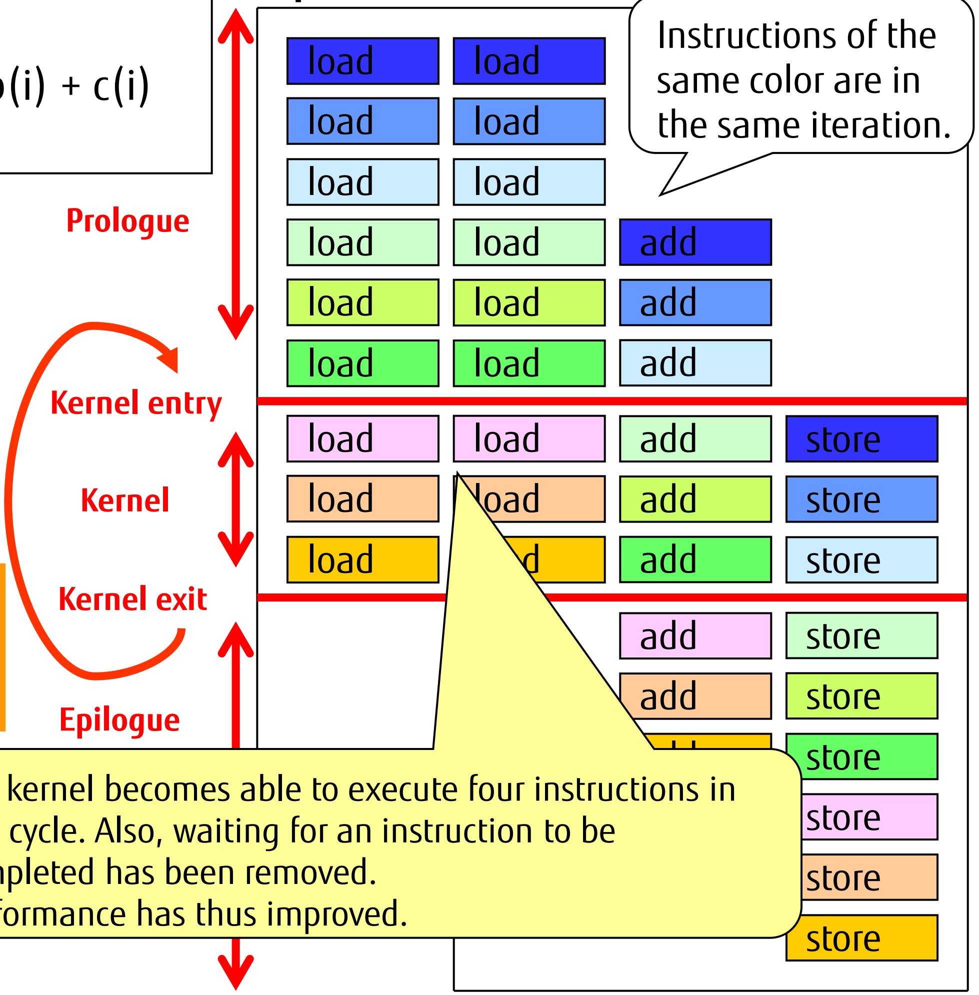
```
do i=1,n
  a(i) = b(i) + c(i)
enddo
```

Before optimization



Improvement from 7 cycles per iteration to 1 cycle per iteration

After optimization



Instructions of the same color are in the same iteration.

The kernel becomes able to execute four instructions in one cycle. Also, waiting for an instruction to be completed has been removed.
Performance has thus improved.

Confirmation of Software Pipelining

Example of compile information output

```
 (line-no.) (nest) (optimize)
                                    :
2                   real*8 a(n), b(n)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 4)
                <<<    SOFTWARE PIPELINING
                <<< Loop-information  End >>>
3     1      8v       do i=1,n
4     1      8v          a(i)=b(i)+b(i+1)+b(i+2)
5     1      8v       enddo
                                   :
```

Indicates that software pipelining of loop was done

Loop Optimization

- Loop Optimization
- Loop Interchange
- Loop Fission
- Loop Fusion
- Loop Unrolling
- Loop Collapse

* See “12.2.3.1 Automatic Parallelization” in the *Fortran User’s Guide* for details.

Loop Optimization

Loop optimization

- The following table lists typical types of automatic loop optimization by the compiler.

Loop optimization and transformation type	Effect
Loop interchange	- Data localization - Parallelization in the outer loop (coarse granularity)
Loop distribution or loop fission	- Facilitation of optimization
Loop fusion	- Data localization - Improved parallelism at the instruction level
Loop unrolling	- Reduction in instructions - Improved parallelism at the instruction level
Loop collapse	- Improved scheduling efficiency - Improved load balancing

Loop Interchange

Purpose

Data localization

- Array b is accessed sequentialy, thereby improving cache use efficiency.

<table>
<tr><th>Original source</th><th>Appearance after compiler optimization</th></tr>
<tr><td><pre>
subroutine sub(a, b, n, m)
real*8 a(n, m), b(n, n, m)
do j=1,n
 do k=1,n
  do i=1,m
   a(j,k)=a(j,k)+b(j,i,k)
  enddo   Stride access
 enddo
enddo
end
</pre></td><td><pre>
subroutine sub(a, b, n, m)
real*8 a(n, m), b(n, n, m)
do k=1,m
 do i=1,n
  do j=1,n
   a(j,k)=a(j,k)+b(j,i,k)
  enddo   Sequential access
 enddo
enddo
end
</pre></td></tr>
</table>

Confirmation of Loop Interchange

- Example of compile information output (loop interchange)

```
(line-no.)(nest)(optimize)
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    INTERCHANGED(nest: 3)
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
                 <<< Loop-information  End >>>
 3    1    8v        do j=1,n
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    INTERCHANGED(nest: 1)
                 <<< Loop-information  End >>>
 4    2    8          do k=1,n
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    INTERCHANGED(nest: 2)
                 <<< Loop-information  End >>>
 5    3    8            do i=1,m
 6    3    8v             a(j,k)=a(j,k)+b(j,i,k)
 7    3    8            enddo
 8    2    8          enddo
 9    1    8v        enddo
```

Loops were interchanged, so
loop j becomes nest 3,
loop k becomes nest 1,
and loop i becomes nest 2

Loop Fission

Purpose

Facilitation of optimization

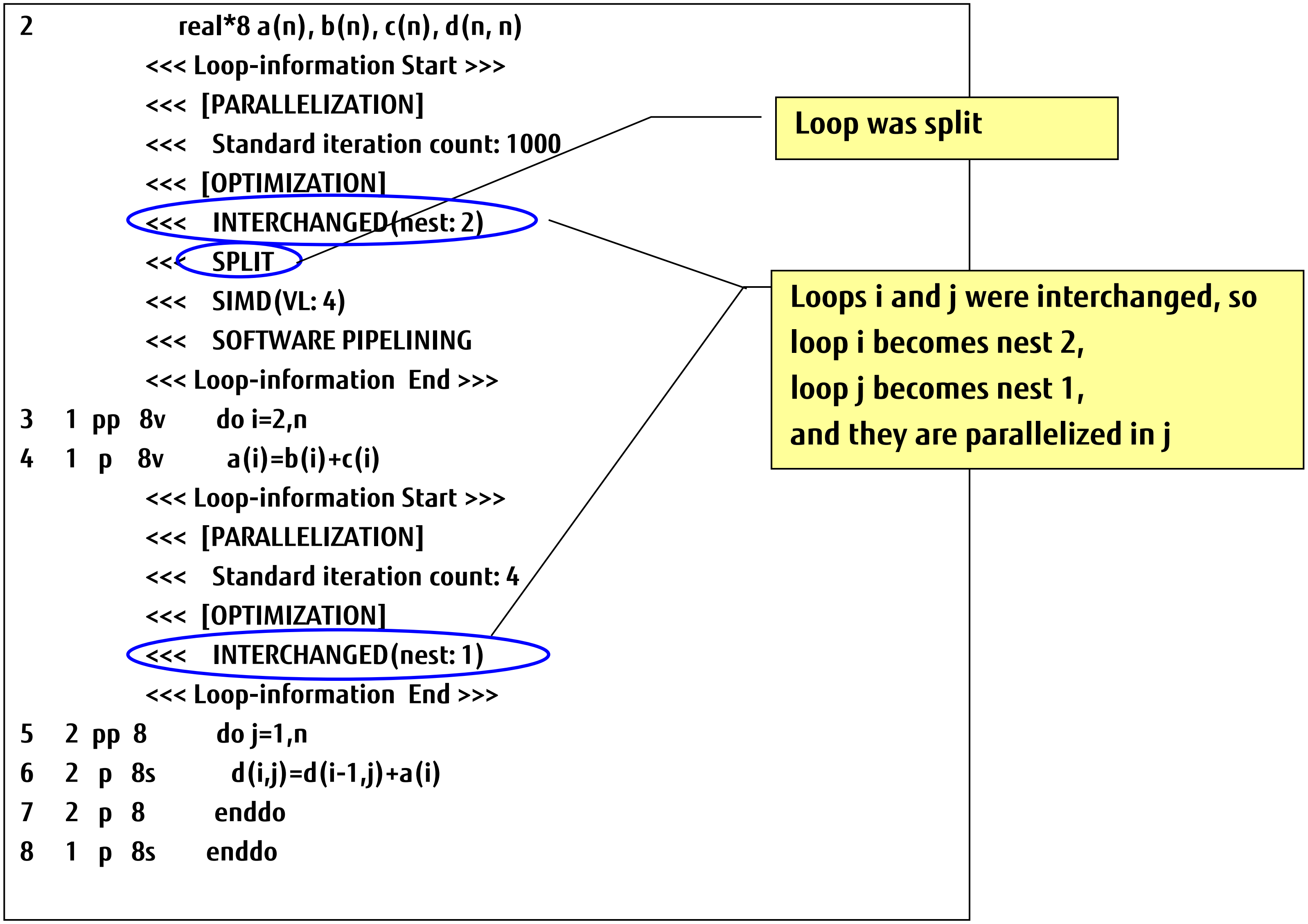
- The loop fission process interchanges loops too, and this enables parallelization of the outer loop.

Original source	Appearance after compiler optimization

subroutine sub(a, b, c, d, n)
real*8 a(n), b(n), c(n), d(n, n)
do i=2,n
a(i)=b(i)+c(i)
do j=1,n ⇨ Parallel
d(i,j)=d(i-1,j)+a(i)
enddo
enddo
end subroutine sub
	subroutine sub(a, b, c, d, n)
	real*8 a(n), b(n), c(n), d(n, n)
	do i=2,n ⇨ Parallel
	a(i)=b(i)+c(i)
	enddo
	do j=1,n ⇨ Parallel
	do i=2,n
	d(i,j)=d(i-1,j)+a(i)
	enddo
	enddo
	end subroutine sub

Confirmation of Loop Fission

- Example of compile information output (loop fission)



```
2               real*8 a(n), b(n), c(n), d(n, n)
             <<< Loop-information Start >>>
             <<<  [PARALLELIZATION]
             <<<    Standard iteration count: 1000
             <<<  [OPTIMIZATION]
             <<<    INTERCHANGED(nest: 2)
             <<<    SPLIT
             <<<    SIMD(VL: 4)
             <<<    SOFTWARE PIPELINING
             <<< Loop-information  End >>>
3    1  pp   8v        do i=2,n
4    1   p   8v          a(i)=b(i)+c(i)
             <<< Loop-information Start >>>
             <<<  [PARALLELIZATION]
             <<<    Standard iteration count: 4
             <<<  [OPTIMIZATION]
             <<<    INTERCHANGED(nest: 1)
             <<< Loop-information  End >>>
5    2  pp  8          do j=1,n
6    2   p  8s           d(i,j)=d(i-1,j)+a(i)
7    2   p  8          enddo
8    1   p  8s       enddo
```

Loop was split

Loops i and j were interchanged, so
loop i becomes nest 2,
loop j becomes nest 1,
and they are parallelized in j

Loop Fusion

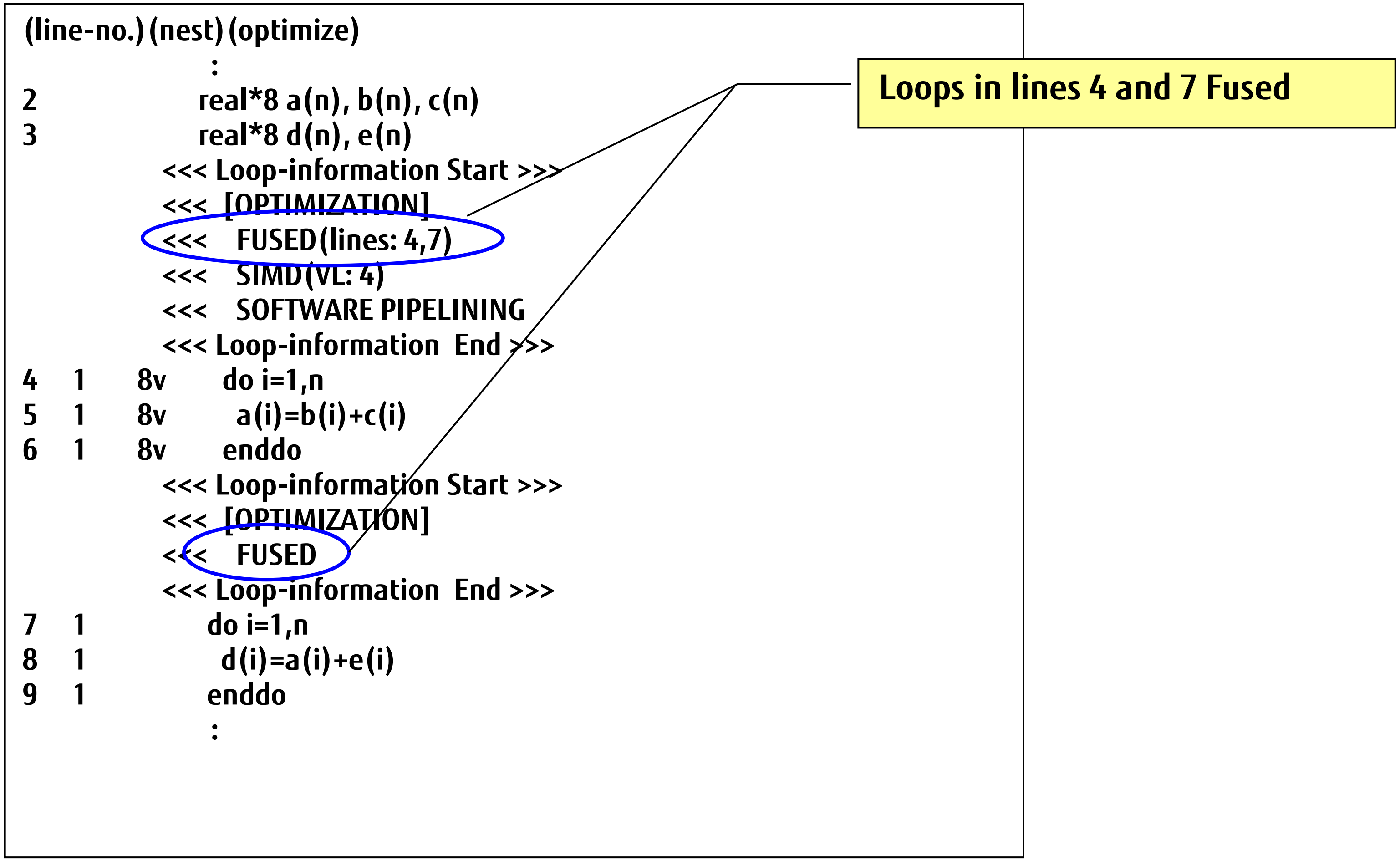
- **Purpose**
  - **Data localization**
    - Loop fusion enables reuse of array a.
  - **Improved parallelism at the instruction level**
    - Loop fusion increases the number of instructions in a loop and improves parallelism at the instruction level through instruction scheduling.

Original source	Appearance after compiler optimization

subroutine sub(a, b, c, d, e, n)
real*8 a(n), b(n), c(n)
real*8 d(n), e(n)
do i=1,n
  a(i)=b(i)+c(i)
enddo
do i=1,n
  d(i)=a(i)+e(i)
enddo
end
	subroutine sub (a, b, c, d, e, n)
	real*8 a(n), b(n), c(n)
	real*8 d(n), e(n)
	do i=1,n
	  a(i)=b(i)+c(i)
	  d(i)=a(i)+e(i)
	enddo
	end

Confirmation of Loop Fusion

- Example of compile information output (loop fusion)



```
(line-no.)(nest)(optimize)
                          :
2                       real*8 a(n), b(n), c(n)
3                       real*8 d(n), e(n)
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    FUSED(lines: 4,7)
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
                 <<< Loop-information  End >>>
4     1      8v        do i=1,n
5     1      8v          a(i)=b(i)+c(i)
6     1      8v        enddo
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    FUSED
                 <<< Loop-information  End >>>
7     1                do i=1,n
8     1                  d(i)=a(i)+e(i)
9     1                enddo
                          :
```

Loops in lines 4 and 7 Fused

Loop Unrolling

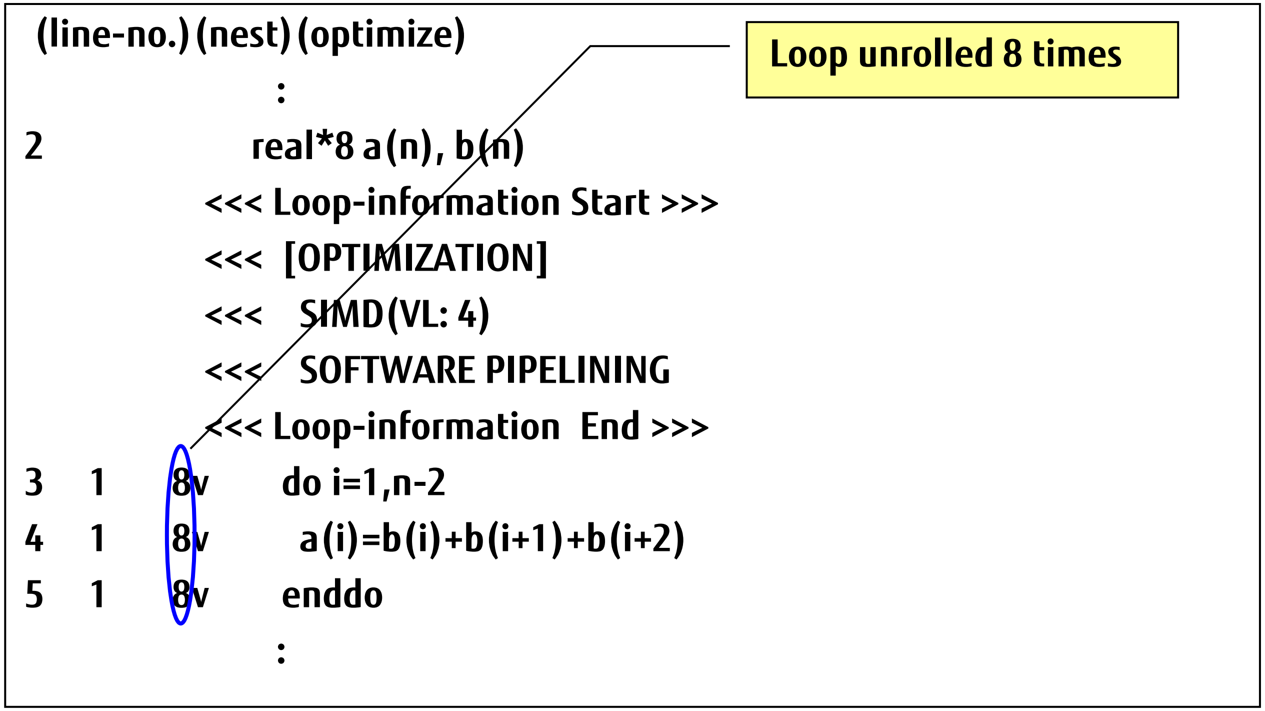
- **Purpose**
  - **Reduction in instructions**
    - Branch instructions are reduced because the number of iterations is halved.
    - Instructions are reduced because b(i+1)+b(i+2) is used as a common expression.
  - **Improved parallelism at the instruction level**
    - The increased number of instructions per iteration increases the leeway for scheduling and improves parallelism at the instruction level.

Original source	Appearance after compiler optimization

subroutine sub(a, b, n)
real*8 a(n), b(n)
do i=1,n-2
  a(i)=b(i)+b(i+1)+b(i+2)
enddo
end
	subroutine sub(a, b, n)
	real*8 a(n), b(n)
	do i=1,n-2,2
	  t = b(i+1)+b(i+2)
	  a(i)=b(i)+t
	  a(i+1)=t+b(i+3)
	enddo
	End

Confirmation of Loop Unrolling

- Example of compile information output (loop unrolling)



```
 (line-no.)(nest)(optimize)
                        :
2                    real*8 a(n), b(n)
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
                 <<< Loop-information  End >>>
3     1      8v       do i=1,n-2
4     1      8v         a(i)=b(i)+b(i+1)+b(i+2)
5     1      8v       enddo
                        :
```

Loop unrolled 8 times

Loop Collapse

Purpose

Improved software pipelining efficiency

- Loop collapsed increases the number of innermost loop iterations and improves the effect of software pipelining.

Improvement in load imbalance

- Loop collapsed increases the probability of improvement in load balancing, regardless of the value of m. If the value of m is 1 or 2, for example, parallelization outside of the original source has disadvantages.

Original source

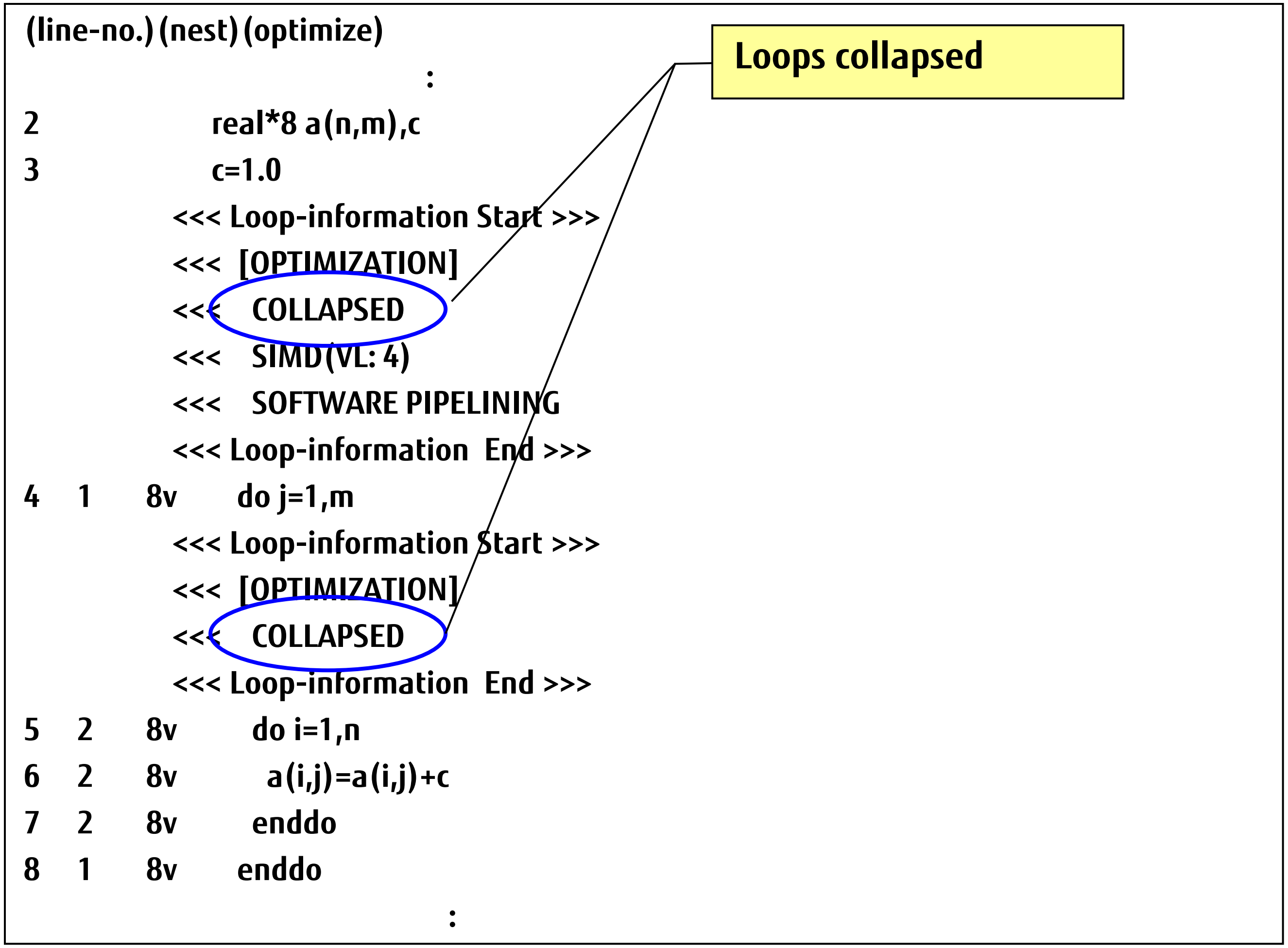
```
subroutine sub(a,n,m)
 real*8 a(n,m),c
 c=1.0
 do j=1,m
   do i=1,n
     a(i,j)=a(i,j)+c
   enddo
 enddo
end subroutine sub
```

Appearance after compiler optimization

```
subroutine sub(a,n,m)
 real*8 a(n,m),c, a1(n*m)
 equivalence (a1, a)
 c=1.0
 do ij=1,n*m
  a1(ij)=a1(ij) + c
 enddo
end subroutine sub
```

Confirmation of Loop Collapse

- Example of compile information output (loop collapse)



```
(line-no.)(nest)(optimize)
                                  :
2                  real*8 a(n,m),c
3                  c=1.0
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 4)
                <<<    SOFTWARE PIPELINING
                <<< Loop-information  End >>>
4    1      8v      do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
5    2      8v        do i=1,n
6    2      8v          a(i,j)=a(i,j)+c
7    2      8v        enddo
8    1      8v      enddo
                                    :
```

Loops collapsed

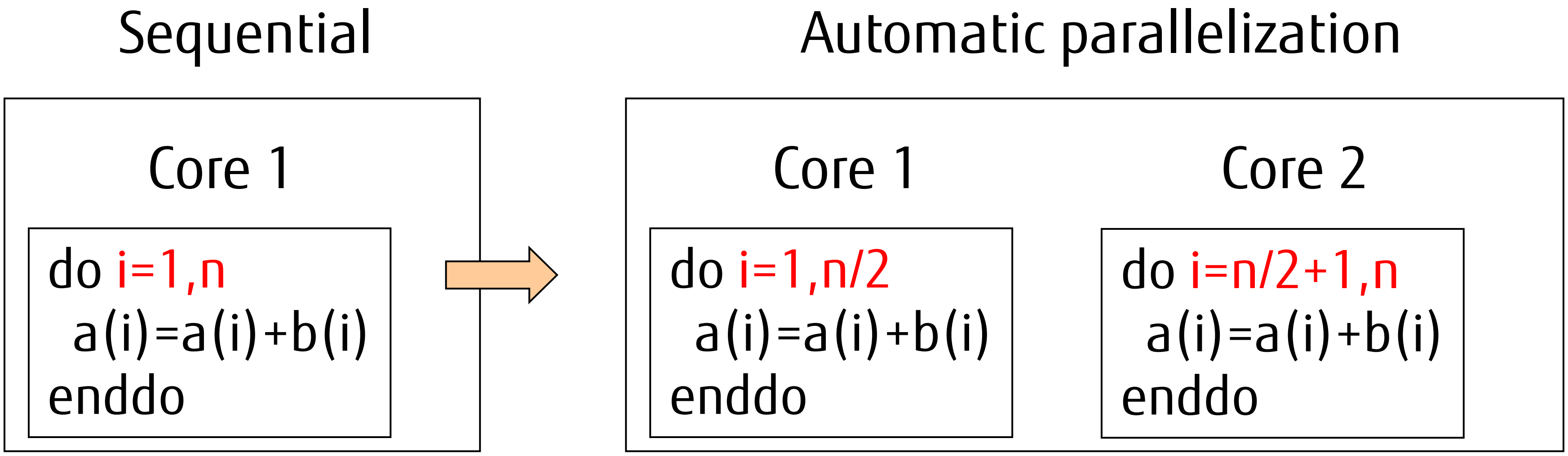
Automatic Parallelization

- Simple Loop Slicing
- Loop Slicing through Reduction
- Determination of Whether Parallelization Is Possible
- Confirmation of Automatic Parallelization
- Pipeline Parallel Processing

* See “12.2.3.1 Automatic Parallelization” in the *Fortran User‘s Guide* for details.

Simple Loop Slicing

Loop indexes are assigned to multiple threads to obtain parallelization.



Sequential

Core 1

```
do i=1,n
  a(i)=a(i)+b(i)
enddo
```

Automatic parallelization

Core 1

```
do i=1,n/2
  a(i)=a(i)+b(i)
enddo
```

Core 2

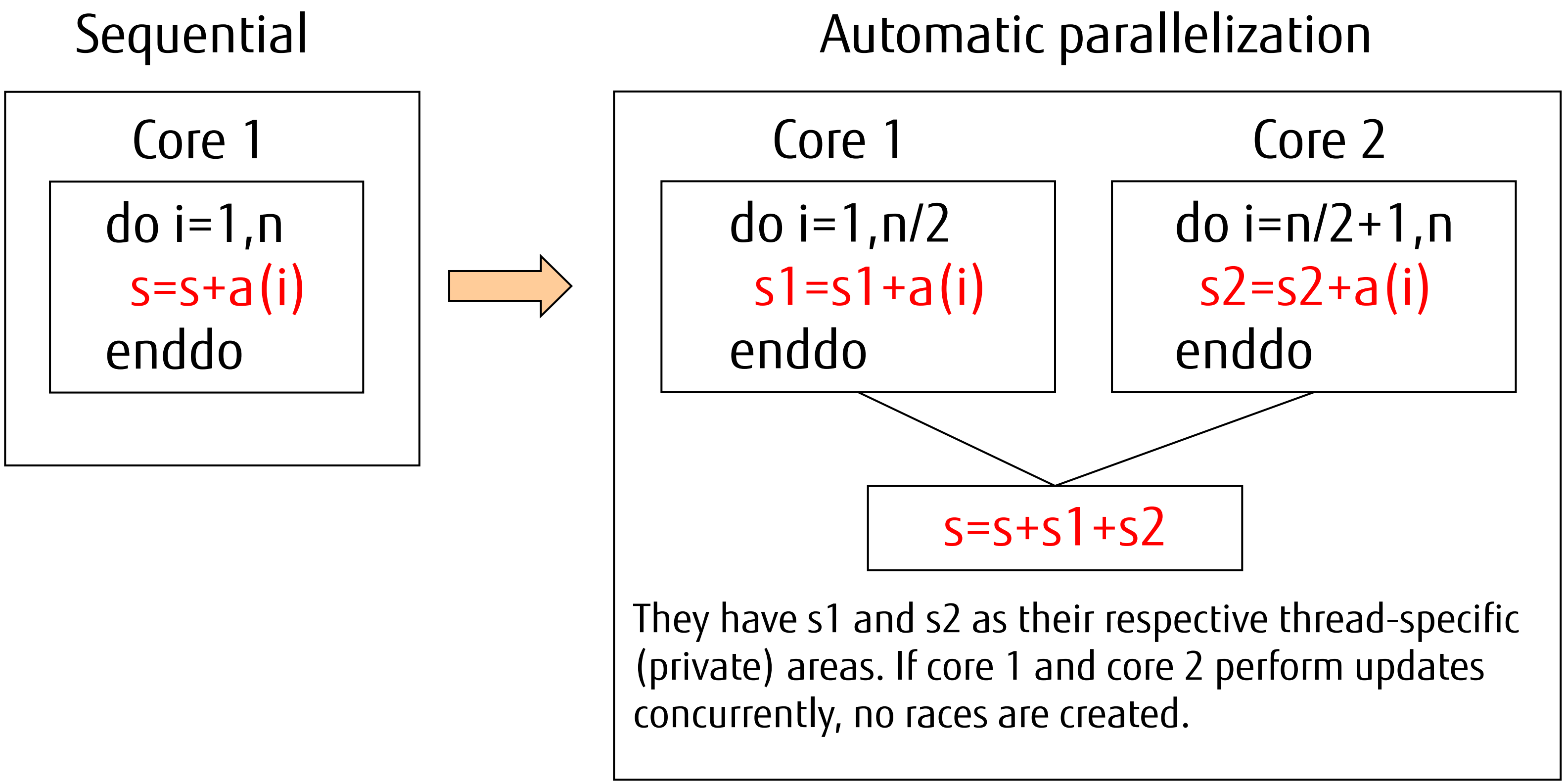
```
do i=n/2+1,n
  a(i)=a(i)+b(i)
enddo
```

(Example for 2 threads)

The number of parallels is specified by an environment variable (PARALLEL) at the execution time.

Loop Slicing through Reduction

Parallelization in a way that avoids data races



Sequential

Core 1

```
do i=1,n
  s=s+a(i)
enddo
```

Automatic parallelization

Core 1

```
do i=1,n/2
  s1=s1+a(i)
enddo
```

Core 2

```
do i=n/2+1,n
  s2=s2+a(i)
enddo
```

```
s=s+s1+s2
```

They have s1 and s2 as their respective thread-specific (private) areas. If core 1 and core 2 perform updates concurrently, no races are created.

Note: The different order of operations than in sequential processing may occur calculation errors.
If -Knoeval is specified, -Knoreduction is enabled and parallelization is not possible.

Determination of Whether Parallelization Is Possible

The following cases are excluded from parallelization:

(1) Loop cost found to be low at the compile time

(2) Loop cost found to be low at the execution time

* The compiler outputs dynamic control only for parallelization at times of high loop costs.

(1)

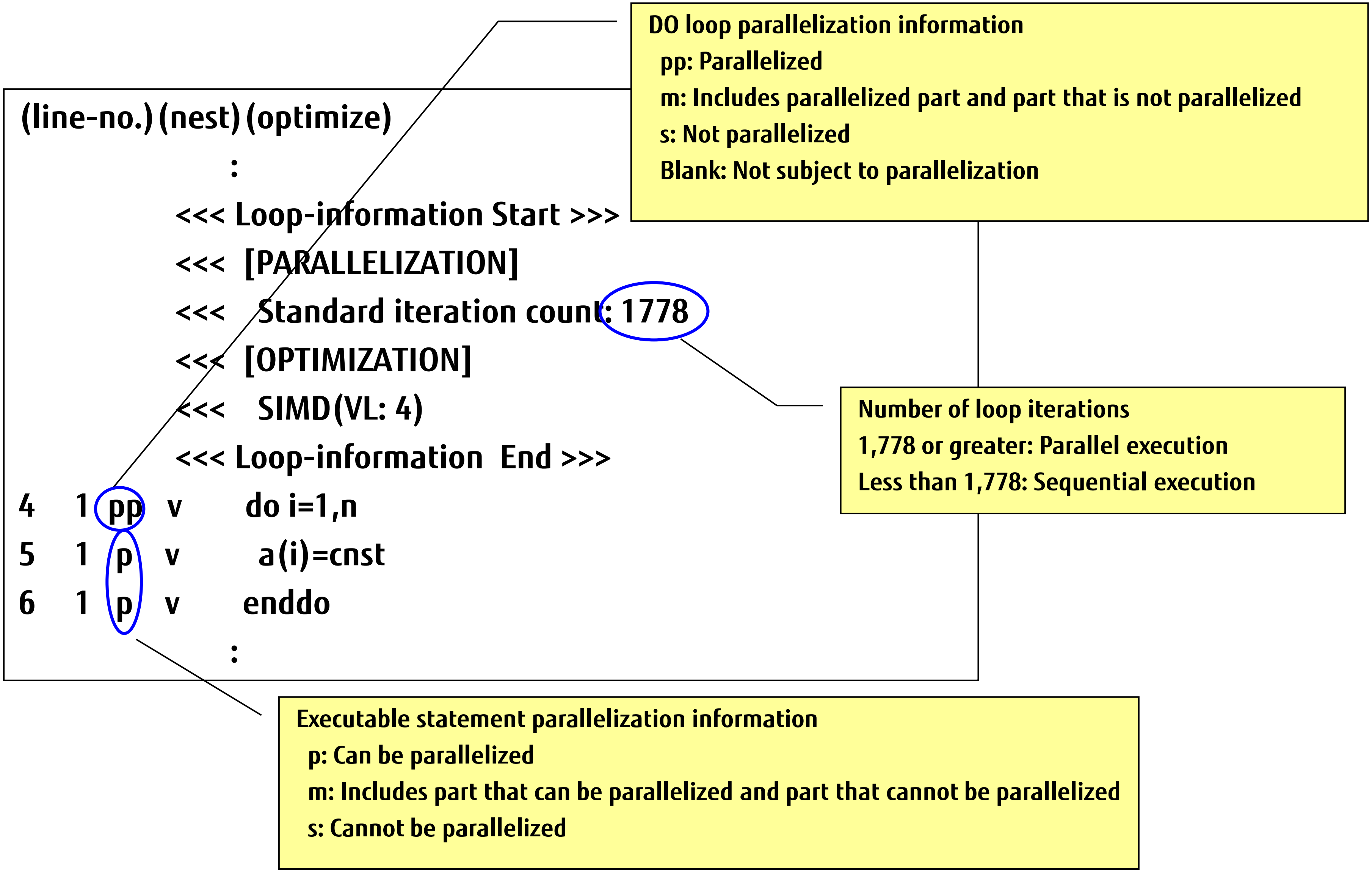
```
real*8 a(3,3),b(3,3)
do j=1,3
   do i=1,3
      a(i,j)=a(i,j)+1.0
   enddo
enddo
```

(2)

```
read*8 a(3),b(3)
call sub(a,b,3)
 :
subroutine sub(a,b,n)
   real*8 s(n),b(n)
   do i=1,n
      a(i)=b(i)+1.0
   enddo
end subroutine
```

Confirmation of Automatic Parallelization

- Example of compile information output



```
(line-no.)(nest)(optimize)
                    :
               <<< Loop-information Start >>>
               <<<  [PARALLELIZATION]
               <<<   Standard iteration count: 1778
               <<<  [OPTIMIZATION]
               <<<   SIMD(VL: 4)
               <<< Loop-information  End >>>
4     1  pp   v       do i=1,n
5     1  p    v         a(i)=cnst
6     1  p    v       enddo
                    :
```

DO loop parallelization information
pp: Parallelized
m: Includes parallelized part and part that is not parallelized
s: Not parallelized
Blank: Not subject to parallelization

Number of loop iterations
1,778 or greater: Parallel execution
Less than 1,778: Sequential execution

Executable statement parallelization information
p: Can be parallelized
m: Includes part that can be parallelized and part that cannot be parallelized
s: Cannot be parallelized

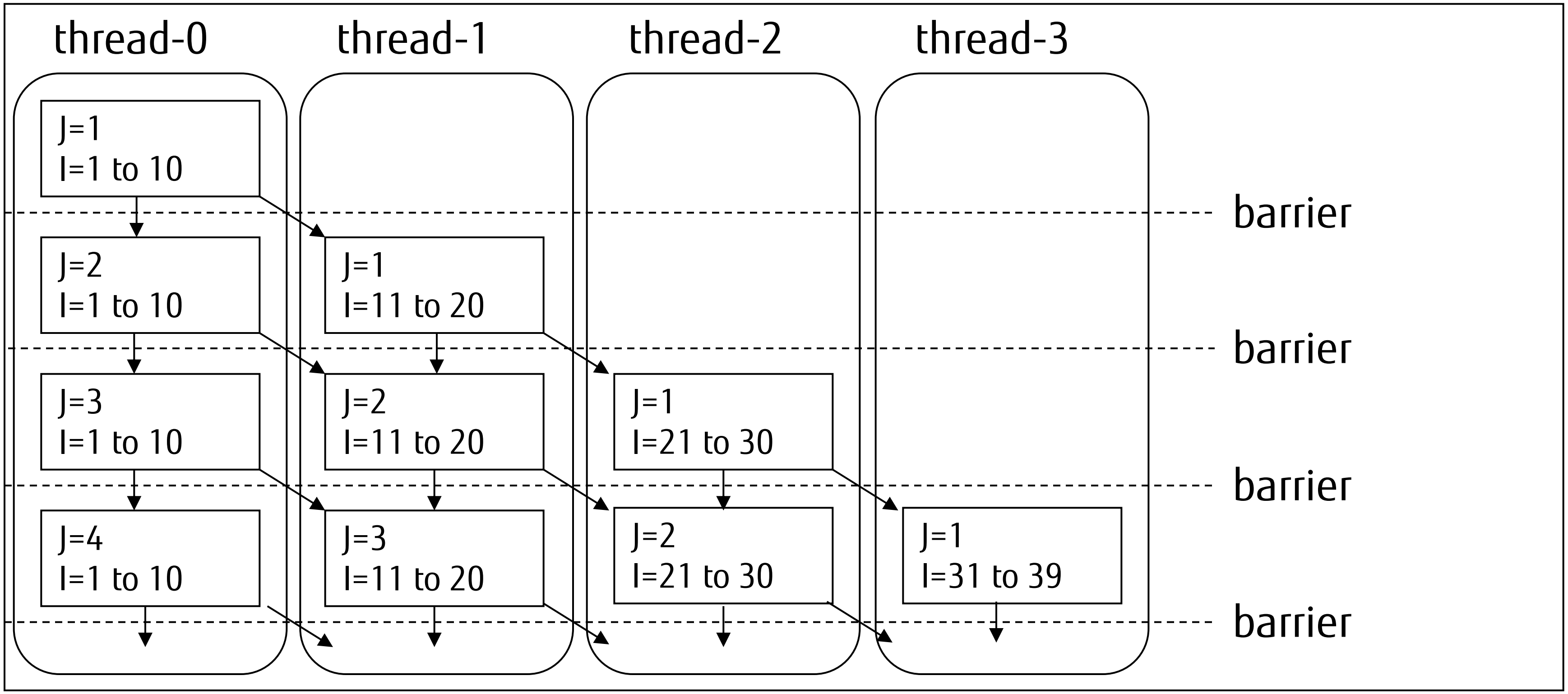
Pipeline Parallel Processing

- Pipeline Parallel Processing (Special parallelization)

```
do j=1,n
  do i=1,39
    a(i, j) = a(i+1, j)+a(i, j+1)
  enddo
enddo
```

Dependency on data across iterations
⇒ Normal parallelization impossible

No data dependency in the diagonal direction
⇒ Parallelization by diagonally slicing the inner loop



Recommended Options (Effects and Impact)

- Recommended Options
- Options with an Impact on Operation Results and Execution
- Other Compile Options
- Optimization Options

Recommended Options (Sequential)

The recommended optimization option when seeking higher execution performance with the frtpx command is -Kfast.
Compilation with this option triggers the following options internally.

Option	Meaning
-O3	Compile at optimization level 3.
-Kdalign	Generate an instruction with an assumption of alignment on a double-word boundary.
-Kns *	Initialize the FPU in non-standard floating-point mode.
-Keval *	Apply optimization to change the method of mathematical evaluation.
-Kmfunc *	Apply multi-operation functionalization.
-Kprefetch_conditional	Use the prefetch instruction for the array data included in if and case constructs.
-Kfp_contract *	Optimize by using FMA arithmetic instructions.
-Kfp_relaxed *	Execute reciprocal approximation operations.
-Kilfunc *	Inline expand an intrinsic function of the single/double-precision real type.
-Komitfp	Instruct whether to guarantee the frame pointer register for a procedure call.

The * mark indicates an option that may cause a difference in precision.

Recommended Options (Automatic Parallelization)

The recommended optimization options when seeking higher execution performance with the frtpx command are -Kfast,parallel .

Compilation with this option triggers the following options internally.

Option	Meaning
-Kfast *	Apply the sequential optimization described on the previous page.
-Kparallel	Apply automatic parallelization.

The * mark indicates an option that may cause a difference in precision.

Options with an Impact on Execution

The following table lists the commonly used compiler options that may cause the abnormal end of execution because of instruction movement involving speculative execution.

Compile option	Function	Impact of optimization
-Kpreex	Preliminary evaluation of invariant expressions (Invariant expressions are moved from IF construct in the loop to outside the loop.)	Abnormal end of execution
-Ksimd=2	SIMD optimization of an IF construct	Abnormal end of execution

If execution is abnormally terminated, specify -Knf together with -Kpreex or -Ksimd=2 to run the compiler in Non-Faulting mode so that the abnormal end of execution of the load instruction for speculative execution can be avoided.

Compile option	Function	Impact of optimization
-Knf	Instruction of whether to optimize using Non-Faulting mode	Failure to detect exceptions to the load instruction

Other Compile Options (1)

- Optimization options (optimization level specification)
  - -O0, -O1, -O2, -O3, -O
- Optimization options (inline expansion)
  - -x-, -xproc_name, -xstmt_no, -x0
- Optimization options (optimization involving rounding errors)
  - -K[no]eval, -K[no]fp_contract, -K[no]fp_relaxed, -K[no]ilfunc
- Optimization options (optimization involving speculative execution)
  - -K[no]preex, -K[no]nf
- Optimization options (prefetch)
  - -Kprefetch_[no]indirect, -Kprefetch_[no]stride
- Optimization options (SIMD)
  - -Ksimd[={1|2|auto}], -Knosimd

Other Compile Options (2)

- Optimization options (other)
  - -K[no]dalign, -K[no]auto, -K[no]ocl
- Thread parallelization processing options
  - -K[no]openmp, -K[no]dynamic_iteration, -Kparallel_strong, -K[no]region_extension, -K[no]array_private
- Other options
  - -fs , -NR[no]trap, -K[no]striping[=N], -K[no]unroll[=N]

Optimization Options (Optimization Level Specification)

Default	Option format	Function overview/Use example
-	-O0	Gives an instruction to disable optimization. [Use example] Specify this option if you do not want to optimize the compiler.
-	-O1	Gives an instruction for basic optimizations except loop-related optimizations. [Use example] Specify this option to prevent optimizations from increasing the size of the executable module. Specifically, optimizations that improve execution performance, such as loop unrolling, software pipelining, and SIMD optimization, will be stopped. In addition, specify this option when you want a shorter compile time.
Default	-O2	Gives an instruction for optimizations that improve execution performance, such as loop unrolling, software pipelining, SIMD optimization, and prefetch generation for sequential access, in addition to the optimizations of -O1. [Use example] A precision error occurs when -Kfast is specified. So if you want to improve performance without causing precision errors to occur, specify -O2 or -O3. Specify -Kdalign at the same time to further facilitate SIMD optimization.
-	-O3 (-O) *	Gives an instruction for optimizations that further increase the compile time compared with that of -O2. The optimizations include full unrolling of multiple loops and loop fission for facilitation of loop interchange, in addition to the optimizations of -O2.

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Inline Expansion)

Default	Option format	Function overview/Use example
-	-x-	Gives an instruction to inline expand a user-defined procedure. [Use example] Specify this option to facilitate optimization through inline expansion of a user-defined procedure. Use this option to have the compiler select the user-defined procedure to be inline expanded.
-	-xproc_name	Gives an instruction to inline expand the user-defined procedure proc_name. [Use example] Specify this option to facilitate optimization through inline expansion of the procedure proc_name that is called from within a high-cost loop.
-	-xstmt_no (-x0 *)	Gives an instruction to inline expand user-defined procedures whose number of executable statements is equal to or less than stmt_no. -x0 gives an instruction to disable inline expansion. [Use example] Specify this option to inline expand all the user-defined procedures whose number of executable statements is equal to or less than stmt_no when there are too many user-defined procedures to specify function names one by one.

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Optimization Involving Rounding errors (1))

Default	Option format	Function overview/Use example
- Default	-Keval * -Knoeval	Gives an instruction on whether to change the method of mathematical evaluation. [Use example] **Example of -Keval optimization of a = b + c + d + e** **a = (((b + c) + d) + e) => -Keval => a = ((b + c) + (d + e))** When -Kfast is specified, -Keval is triggered, which executes optimization that involves changing the method of mathematical evaluation as shown above. To prevent this rounding error from occurring, specify -Knoeval after -Kfast.
- Default	-Kfp_contract * -Knofp_contract	Gives an instruction on whether to output the multiply add/subtract floating-point instruction. [Use example] When -Kfast is specified, -Kfp_contract is triggered, which outputs the multiply add/subtract floating-point instruction. To prevent this precision error from occurring, specify -Knofp_contract after -Kfast.
- Default	-Kfp_relaxed * -Knofp_relaxed	Gives an instruction on whether to execute reciprocal approximation operations for floating-point division or the SQRT function. [Use example] When -Kfast is specified, -Kfp_relaxed is triggered, which outputs the reciprocal approximation instruction. To prevent this precision error from occurring, specify -Knofp_relaxed after -Kfast.

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Optimization Involving Rounding errors (2))

Default	Option format	Function overview/Use example
-	-Kilfunc *	Gives an instruction on whether to inline expand intrinsic functions. [Use example] When -Kfast is specified, -Kilfunc is triggered, which inline expands intrinsic functions. The argument error check is omitted from the inline expansion of intrinsic functions. Consequently, a precision error may occur. To prevent this error from occurring, specify -Knoilfunc after -Kfast.
Default	-Knoilfunc	

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Optimization Involving Speculative Execution)

Default	Option format	Function overview/Use example
- Default	-Kpreex -Knopreex *	Enables optimization through preliminary evaluation of invariant expressions. [Use example] The aim of the -Kpreex option is to improve execution performance through a preliminary evaluation (speculative execution beyond IF statements) of invariant expressions for the object program. However, execution may be abnormally terminated because instructions that are not supposed to be executed from the perspective of program logic are speculatively executed.
- Default	-Knf -Knonf *	Gives an instruction on whether to optimize the object program by using Non-Faulting mode. [Use example] Specify -Knf to use Non-Faulting mode so that the load that is speculatively executed by -Kpreex or -Ksimd=2 is not abnormally terminated.

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Prefetch (1))

Default	Option format	Function overview/Use example
-	-Kprefetch_indirect	Gives an instruction on whether to generate an object that uses a prefetch instruction for indirectly accessed (list access) array data used inside a loop. [Use example] As shown in the optimization control lines below, specify -Kprefetch_indirect to output prefetch instructions for list access for which subscripts specify an array, such as a(m(i)) in the following program. However, execution performance may deteriorate depending on the cache efficiency of loops, whether branches are used, and the complexity of subscripts.
Default	-Kprefetch_noindirect *	

```
do i=1,N
   /* Output an L2 prefetch instruction for a(m(i+α)) */
   /* Output an L1 prefetch instruction for a(m(i+β)) */
   b(i) = b(i) + a(m(i))
enddo
```

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Prefetch (2))

Default	Option format	Function overview/Use example

-	-Kprefetch
	_stride
		Gives an instruction on whether to generate an object that uses a prefetch instruction for array data accessed with a longer stride than the line size of the cache used inside a loop.
		[Use example]
		The program below shows stride access for which an array element accessed in the current iteration and the one accessed in the next iteration are not on the same cache line because the loop incremental value is large. For such stride access, specify -Kprefetch_stride to output prefetch instructions, as shown in the optimization control lines below. However, execution performance may deteriorate depending on the cache efficiency of loops and whether branches are used.
		`do i=1,N,100`
		`  /* Output an L2 prefetch instruction for a(i+100*α) */`
		`  /* Output an L1 prefetch instruction for a(i+100*β) */`
		`  /* Output an L2 prefetch instruction for b(i+100*α) */`
		`  /* Output an L1 prefetch instruction for b(i+100*β) */`
		`   b(i) = a(i) + 1.0`
		`enddo`

Default	-Kprefetch _nostride *	

The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (SIMD)

Default	Option format	Function overview/Use example
-	-Ksimd=1	Gives an instruction on whether to perform SIMD optimization. -Ksimd=1: Output an SIMD instruction. -Ksimd=2: Output a masked SIMD instruction in addition to that of -Ksimd=1. -Ksimd=auto: The compiler automatically determines whether to SIMD-optimize loops. [Use example] Specify -Ksimd=2 to SIMD-optimize high-cost loops containing an IF statement that meets either of the following conditions: ➢ The IF conditional clause has a high true ratio. ➢ The true ratio of the IF conditional clause is unclear, but the number of executable statements in the THEN/ELSE clause of the IF conditional clause is small compared to that in the entire loop. Doing so can facilitate software pipelining to improve instruction-level parallelization by eliminating branch instructions from the loop.
-	-Ksimd=2	
Default	-Ksimd=auto *	
-	-Knosimd	

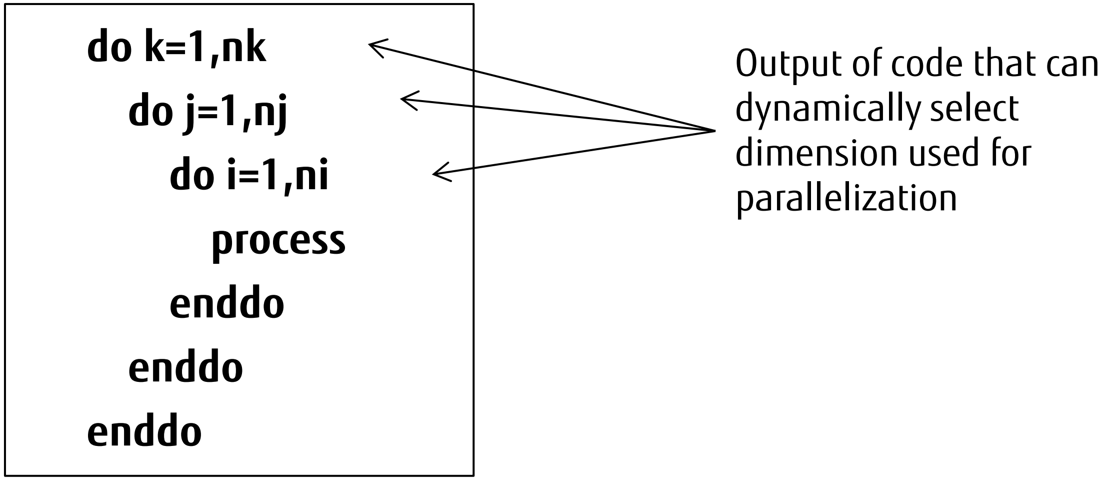
The * mark indicates an option that is enabled when -Kfast is specified.

Optimization Options (Other)

Default	Option format	Function overview/Use example
- Default	-Kdalign * -Knodalign	Gives an instruction on whether to generate an instruction with an assumption that the data type of 8 bytes or more referenced by a dummy argument or pointer is aligned with the 8-byte boundary. [Use example] When -Kfast is specified, -Kdalign is triggered, so there is no need to be aware of this option. If you want to promote execution performance by specifying the -O2 or -O3 option without specifying -Kfast, specify -Kdalign to facilitate SIMD optimization, which improves execution performance.
- Default	-Kauto -Knoauto *	Gives an instruction on whether to use a local variable as an automatic variable.
- Default	-Kocl -Knoocl *	Gives an instruction on whether to enable optimization control lines.

The * mark indicates an option that is enabled when -Kfast is specified.

Thread Parallelization Processing Options (1)

<table>
<tr><th>Default</th><th>Option format</th><th>Function overview/Use example</th></tr>
<tr><td>-
Default</td><td>-Kopenmp
-Knoopenmp *</td><td>Gives an instruction on whether to enable OpenMP Fortran directives.</td></tr>
<tr><td>-
Default</td><td>-Kdynamic_iteration
-Knodynamic_iteration *</td><td>Outputs code for parallel execution with a parallelization method that dynamically selects which loop (outer or inner) to use by considering the parallelization effect on multiple loops. (Normally, the parallelization method uses the one that is as far out as possible for parallelization.)
[Use example]
If the sizes of nk, nj, and ni are unclear and their values are likely small, parallelization with so few iterations may degrade performance. In such cases, specify -Kdynamic_iteration to output code for parallelization in the k dimension if nk is the largest, in the j dimension if nj is the largest, or in the i dimension if ni is the largest.

<pre>
do k=1,nk
  do j=1,nj
    do i=1,ni
      process
    enddo
  enddo
enddo
</pre>
Output of code that can dynamically select dimension used for parallelization</td></tr>
</table>

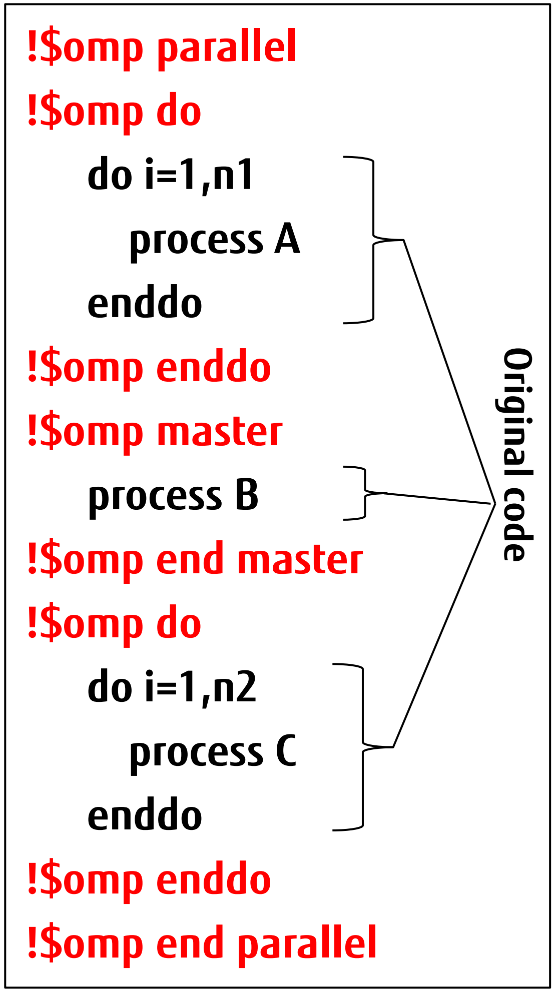
The * mark indicates an option that is enabled when -Kfast is specified. (The default of -Kparallel is also -Knodynamic_iteration.)

Thread Parallelization Processing Options (2)

Default	Option format	Function overview/Use example
-	-Kparallel_strong	Gives an instruction to parallelize all the loops that have been analyzed and determined as capable of being automatically parallelized without estimating the parallelization effect. [Use example] You can expect execution performance to improve by specifying -Kparallel_strong when you know that the number of iterations of a high-cost loop in a procedure is large. Normally, automatic parallelization considers also the cases where there are a few iterations. The output object contains such dynamic control that it executes either a sequential loop without thread processing if the number of iterations is small or thread parallelization processing if it exceeds a given threshold. -Kparallel_strong outputs an object that parallelizes threads no matter how few iterations there are, without outputting a control to handle cases where there are few iterations. Therefore, if there are actually few iterations, the overhead of thread parallelization may become larger, resulting in degraded execution performance.
Default	- *	

The * mark indicates an option that is enabled when -Kparallel is specified.

Thread Parallelization Processing Options (3)

Default	Option format	Function overview/Use example
-	-Kregion_extension *	Gives an instruction on whether to extend a parallel region. [Use example] The figure on the right is an OpenMP representation of the parallel region extended by -Kregion_extension, which is triggered by -Kparallel, to include multiple loops. As with -Kparallel_strong, -Kregion_extension does not execute generation of multiple version that considers the cost of the loops. Therefore, if the number of n1 and n2 iterations is small and extending the region affects the overall execution performance, specify -Knoregion_extension to prevent performance degradation. 
Default	-Knoregion_extension	

```
!$omp parallel
!$omp do
   do i=1,n1
      process A
   enddo
!$omp enddo
!$omp master
   process B
!$omp end master
!$omp do
   do i=1,n2
      process C
   enddo
!$omp enddo
!$omp end parallel
```

The * mark indicates an option that is enabled when -Kparallel is specified.

Thread Parallelization Processing Options (4)

Default	Option format	Function overview/Use example
- Default	-Karray_private -Knoarray_private*	Gives an instruction on whether to privatize arrays in a loop. [Use example] -Karray_private privatizes any arrays that can be privatized in each thread when multi-dimensional loops are subject to automatic parallelization. This enables you to use a loop located as far out as possible for automatic parallelization. The overhead of parallelization can be reduced as a result. However, stack usage increases by the size of the privatized arrays.

The * mark indicates an option that is enabled when -Kfast is specified.
(The default of -Kparallel is also -Knoarray_private.)

Other Options (1)

Default	Option format	Function overview/Use example
- Default	-fs - *	Gives an instruction to disable output of i- and w-level messages. [Use example] Specify this option to prevent output of a large number of compile messages.
- Default	-NRtrap -NRnotrap *	Gives an instruction on whether to output intrinsic operation diagnosis messages during execution and detect interrupt events during floating-point operations. [Use example] Specify this option to detect any unexpected operation exceptions in the program. When the -NRtrap option is enabled, performance may deteriorate because the optimization that converts the SQRT function into a reciprocal approximation operation is disabled.

*** For descriptions of other commonly used debug options, see "Debug Functions" below.**

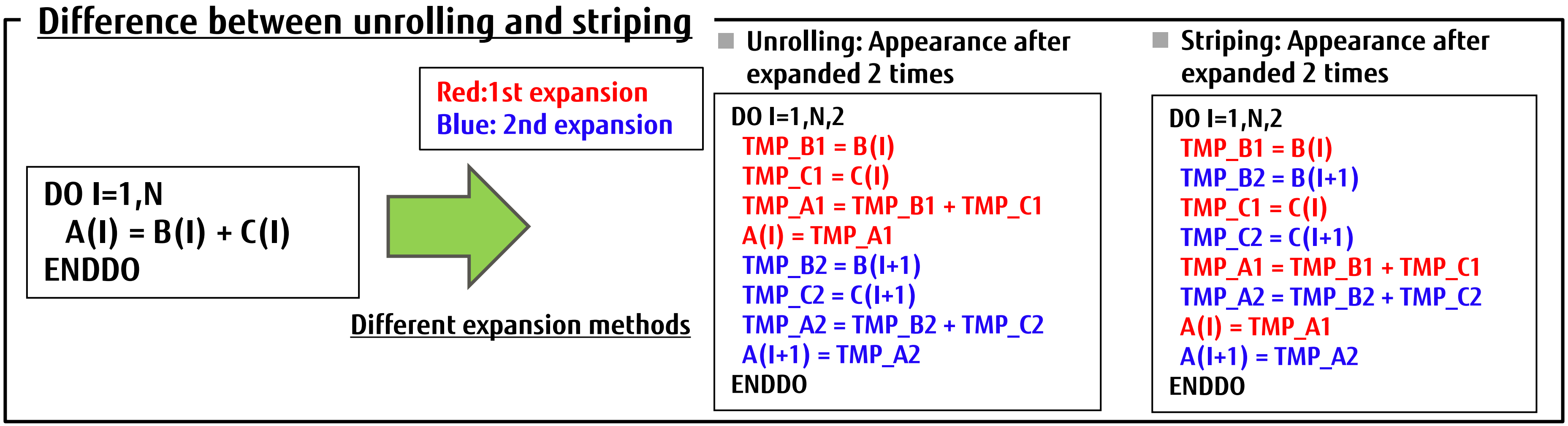
The * mark indicates an option that is enabled when -Kfast is specified.

Other Options (2)

Default	Option format	Function overview/Use example
- Default	-Kstriping[=N] -Knostriping *	Gives an instruction to optimize loop striping. [Use example] The effects of loop expansion can be expected to, for example, reduce the overhead caused by loop iterations and facilitate instruction scheduling.
-O2 or greater -O1 or less	-Kunroll[=N] * -Knounroll	Gives an instruction to optimize loop unrolling. [Use example] The effects of loop expansion can be expected to, for example, reduce the overhead caused by loop iterations and facilitate instruction scheduling.

The * mark indicates an option that is enabled when -Kfast is specified.

Difference between unrolling and striping



Red:1st expansion
Blue: 2nd expansion

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

Different expansion methods

- Unrolling: Appearance after expanded 2 times

```
DO I=1,N,2
 TMP_B1 = B(I)
 TMP_C1 = C(I)
 TMP_A1 = TMP_B1 + TMP_C1
 A(I) = TMP_A1
 TMP_B2 = B(I+1)
 TMP_C2 = C(I+1)
 TMP_A2 = TMP_B2 + TMP_C2
 A(I+1) = TMP_A2
ENDDO
```

- Striping: Appearance after expanded 2 times

```
DO I=1,N,2
 TMP_B1 = B(I)
 TMP_B2 = B(I+1)
 TMP_C1 = C(I)
 TMP_C2 = C(I+1)
 TMP_A1 = TMP_B1 + TMP_C1
 TMP_A2 = TMP_B2 + TMP_C2
 A(I) = TMP_A1
 A(I+1) = TMP_A2
ENDDO
```

Debug Functions

- Overview of Debug Functions
- Overview of the -Nquickdbg Option
- Overview of the -Haefosux Option
- Overview of the Hook Function

* See the *Fortran User's Guide* for details.

- In the case of “Overview of the -Haefosux Option” : “1.2.4 Some Compilation Options”
- Others : “8.2 Debugging Functions”

Overview of Debug Functions

The following two debug functions are available for the Fujitsu Fortran compiler.

	-Nquickdbg	-Haefosux
Execution performance	**Excellent** Reduced impact on performance	**Poor**
Check item	**Good** undef, undefnan argchk, subchk	**Excellent** undef, argchk, subchk shapechk, extchk, overlapchk, I/O chk
Output information	**Excellent** Error identification number Line number at error occurrence Variable name, subscript Procedure name, argument number	**Excellent** Error identification number Line number at error occurrence Variable name, subscript Procedure name, argument number, argument size
Support of thread parallelization processing	**Excellent** OpenMP and automatic parallelization supported	**Poor**

We recommend the -Nquickdbg option for medium- to large-scale programs and the -Haefosux option for small-scale programs.

Overview of the -Nquickdbg Option

- How to use the option
  Specify the compiler options as follows:

```
-Nquickdbg [=argchk|subchk|undef|undefnan]
```

* If no subparameter is specified, the option is equivalent to:
-Nquickdbg=argchk –Nquickdbg=subchk –Nquickdbg=undef

- Check items

Argument	Check details
argchk	Validity check of undefined references (number of references, types of references, types of functions)
subchk	Bound check when referencing an array
undef	Undefined data reference check
undefnan	Undefined data reference check due to floating-point exception

- Features
  - The check is limited to the items that frequently have problems, so that the check has a lower impact on performance. (Compared with the -H option)
  - Checking of OpenMP and automatic parallelization processing programs is already supported.

Overview of the -Haefosux Option

- How to use the option
  Specify the compiler options as follows:

```
-H{a|e|f|o|s|u|x}
```

  * You can specify any check as needed by combining parameters.

- Check items

Parameter	Argument	Check details
a	argchk	Validity check of procedure references (number of arguments, argument type, argument attribute, argument size, number of dimensions of arrays of the assumed shape, function type)
e	shapechk	Shape compatibility check
f	I/O chk	Checking Connection of a File to A Unit, I/O Recursive Call Check
o	overlapchk	Overlapping Dummy Arguments Check , SAVE attribute undefined check
s	subchk	Bound check for array references (array references, subscript overflow)
u	undef	Undefined data reference check
x	extchk	Check of undefined data in a module and common block, Checking for Unassociated Pointer

- Features
  - Debugging is limited to sequential programs.
  - Many check items are available.

Overview of the Hook Function (1)

You can use this function to check the operation of a program by calling a user-defined function from a specific location in the program.

compiler options	-Nhook_func
User-defined function name	user_defined_proc
User-defined function argument	FLAG: User-defined function call source information NAME: Call source function name LINE: Call source line number THREAD: Thread identification number (for OpenMP/automatic parallelization)
Location from which user-defined function is called	- Program entry/exit - Function entry/exit When -Kopenmp or -Kparallel is enabled, a call can be made from the following location too: - Parallel region (OpenMP/automatic parallelization) entry/exit

Overview of the Hook Function (2)

- How to use the function
  Specify the compiler options as follows:

```
-Nhook_func
```

- Format of user-defined functions
  - Format

```
SUBROUTINE USER_DEFINED_PROC(FLAG, NAME, LINE, THREAD)
INTEGER(KIND=4),INTENT(IN):: FLAG, LINE, THREAD
CHARACTER(LEN=*),INTENT(IN):: NAME
```

  - Arguments

FLAG: Shows calling source for the user-defined subroutine.
0: Program entry   1: Program exit   2: Procedure entry   3: Procedure exit
4: Parallel region entry   5: Parallel region exit
6: Regular interval   7 to 99: System reserved   100 or greater: Available to users

NAME: Shows the call source procedure name.
This can be referenced only when FLAG is 2, 3, 4, 5, 100, or greater.

LINE: Shows the call source line number.
This can be referenced only when FLAG is 2, 3, 4, 5, 100, or greater.

THREAD: Shows the identification number of the thread that called the user-defined subroutine.
(OpenMP/automatic parallelization)
This can be referenced only when FLAG is 2, 3, 4, 5, 100, or greater.

Revision History

Version	Date	Revised section	Details
2.0	April 25, 2016	-	- First published

FUJITSU

shaping tomorrow with you