Orchestrating a brighter world

NEC

How to Use C/C++ Compiler for Vector Engine

2nd Edition, November 2019

© NEC Corporation 2019

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

- **How to Use C/C++ Compiler**
  - Performance Analysis
  - Debugging
- **Automatic Vectorization**
  - Extended Vectorization Features
  - Program Tuning
  - Tuning Techniques
  - Notes on Using Vectorization
- **Automatic Parallelization and OpenMP C/C++**
  - OpenMP Parallelization
  - Automatic Parallelization
  - Behavior of Parallelized Program
  - Tuning Parallelized Program
  - Notes on Using Parallelization

The information disclosed in this document is the property of NEC Corporation (NEC) and/or its licensors. NEC and/or its licensors, as appropriate, reserve all patent, copyright and other proprietary rights to this document, including all design, manufacturing, reproduction, use and sales rights thereto, except to the extent said rights are expressly granted to others.
The information in this document is subject to change at any time, without notice.

In this document, "parallel processing" stands for automatic parallelization of compiler or shared memory parallel processing with OpenMP C/C++.

All product, brand, or trade names in this publication are the trademarks or registered trademarks of their respective owners.

NEC C/C++ Compiler for Vector Engine

Product Name: NEC C/C++ Compiler for Vector Engine

- Conforming Language Standards
  - ISO/IEC 9899:2011 Programming languages – C
  - ISO/IEC 14882:2014 Programming languages – C++
  - OpenMP Version 4.5
- Major Features
  - Automatic Vectorization
  - Automatic Parallelization and OpenMP C/C++
  - Automatic Inline Expansion

© NEC Corporation 2019

How to Use C/C++ Compiler

\Orchestrating a brighter world NEC

Usage of C / C ++ Compiler

```
$ ncc -mparallel -O3 a.c b.c      … Compile and link C program
$ nc++           -O4 x.cpp y.cpp … Compile and link C++ program
```

-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 C/C++
-mparallel ... Enable automatic parallelization

Parallelization controlling options.

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

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Example of Typical Compiler Option Specification

```
$ ncc a.c b.c
```

Compiling and linking with the default vectorization and optimization.

```
$ nc++ -O4 a.C b.C
```

Compiling and linking with the highest vectorization and optimization.

```
$ ncc -mparallel -O3 a.c b.c
```

Compiling and linking using automatic parallelization with the advanced vectorization and optimization.

```
$ nc++ -O4 -finline-functions a.cpp b.cpp
```

Compiling and linking using automatic inlining with the highest vectorization and optimization.

```
$ ncc -O0 -g a.c b.c
```

Compiling and linking with generating debugging information in DWARF without vectorization and optimization.

```
$ ncc -g a.c b.c
```

Compiling and linking with generating debugging information in DWARF with the default vectorization and optimization.

```
$ ncc -E a.c b.c
```

Performing preprocessing only and outputting the preprocessed text to the standard output.

```
$ nc++ -fsyntax-only a.cpp b.cpp
```

Performing only grammar analysis.

© NEC Corporation 2019

Program Execution

```
$ ncc a.c b.c
$ ./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.

```
$ ncc -mparallel -O3 a.c b.c
$ 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.

© NEC Corporation 2019

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.

© NEC Corporation 2019

PROGINF

Performance information of the whole program

```
$ ncc -O4  a.c b.c c.c
$ 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)

© NEC Corporation 2019

FTRACE

Performance information of each function

```
$ ncc -ftrace a.c b.c c.c        (Compile and link a program with -ftrace to an executable file)
$ ./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
```


© NEC Corporation 2019
\Orchestrating a brighter world NEC

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.

```
$ nc++ -ftrace -c a.cpp
$ nc++ -c main.cpp b.cpp c.cpp
$ nc++ -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.
- 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.

© NEC Corporation 2019

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

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

```
#include <stdio.h>
Int main(void) {
    printf("%f¥n",1.0/0.0);
}
```

Occur "divide-by-zero"

```
$ ncc -traceback main.c
$ export VE_TRACEBACK=FULL
$ export VE_ADVANCEOFF=YES
$ export VE_FPE_ENABLE=DIV
$ ./a.out
Runtime Error: Divide by zero at 0x600000000cc0
[ 1] Called from 0x7f5ca0062f60
[ 2] Called from 0x600000000b70
Floating point exception
$ naddr2line -e a.out -a 0x600000000cc0
0x0000600000000cc0
/.../main.c:3
```

- `$ ncc -traceback main.c` — 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 … Floating point exception — Traceback information
- `$ naddr2line -e a.out -a 0x600000000cc0` — Specify where the exception occurs
- `/.../main.c:3` — Notice that divide-by-zero is occurring in the 3rd line in the main.c file

© NEC Corporation 2019

\Orchestrating a brighter world NEC

Using GDB

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

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

- Only a.c is compiled with **-O0 -g**
- The others are compiled without **-g**
- 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**. The execution time may increase substantially to stop the advance control. Please take care it.

© NEC Corporation 2019

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.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

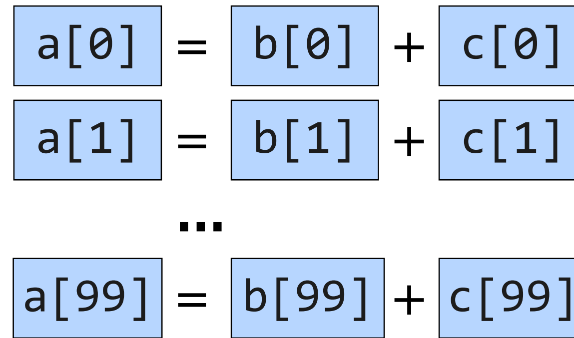
Automatic Vectorization

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

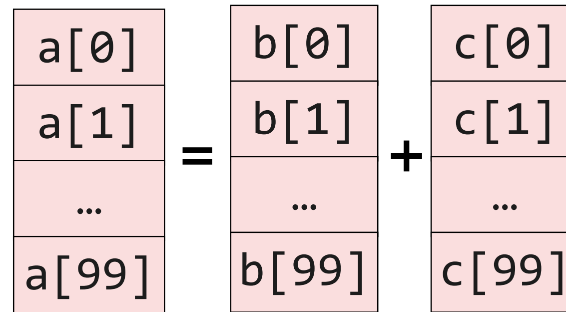
```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
…
a[99]=b[99] + c[99];
```



Execute one calculation 100 times

Execution image of scalar instructions

```
for (i=0; i<100; i++)
  a[i] = b[i] + c[i];
```



Execute 100 calculation at once

At most 256 calculation at once

© NEC Corporation 2019

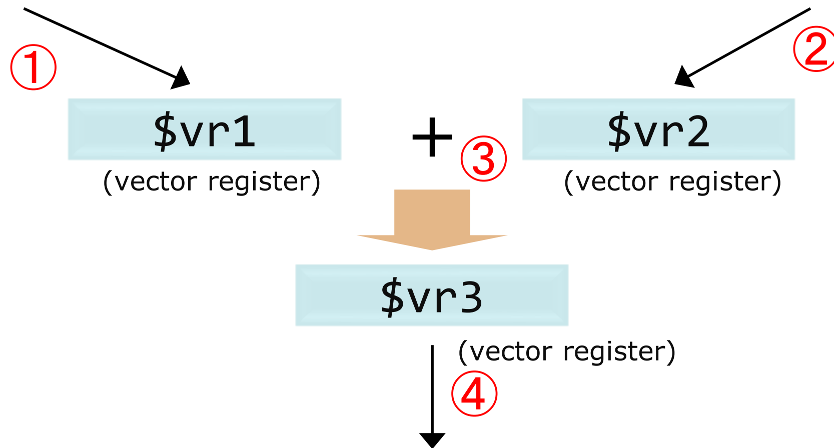
Comparison of HW Instruction

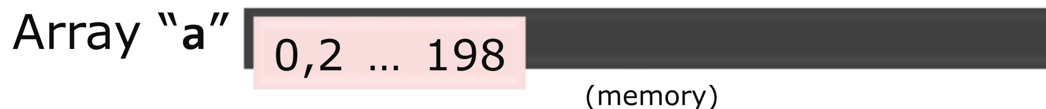
In the case of a scalar machine, these four instruction sequences must be repeated 100 times.

```
a[0]  = b[0]  + c[0];
a[1]  = b[1]  + c[1];
…
a[99] = b[99] + c[99];
 ④      ①     ③  ②
```

```
① VLoad  $vr1, b[0:99]
② VLoad  $vr2, c[0:99]
③ VAdd   $vr3, $vr1, $vr2
④ VStore $vr3, a[0:99]
```

Array "**b**"   Array "**c**"





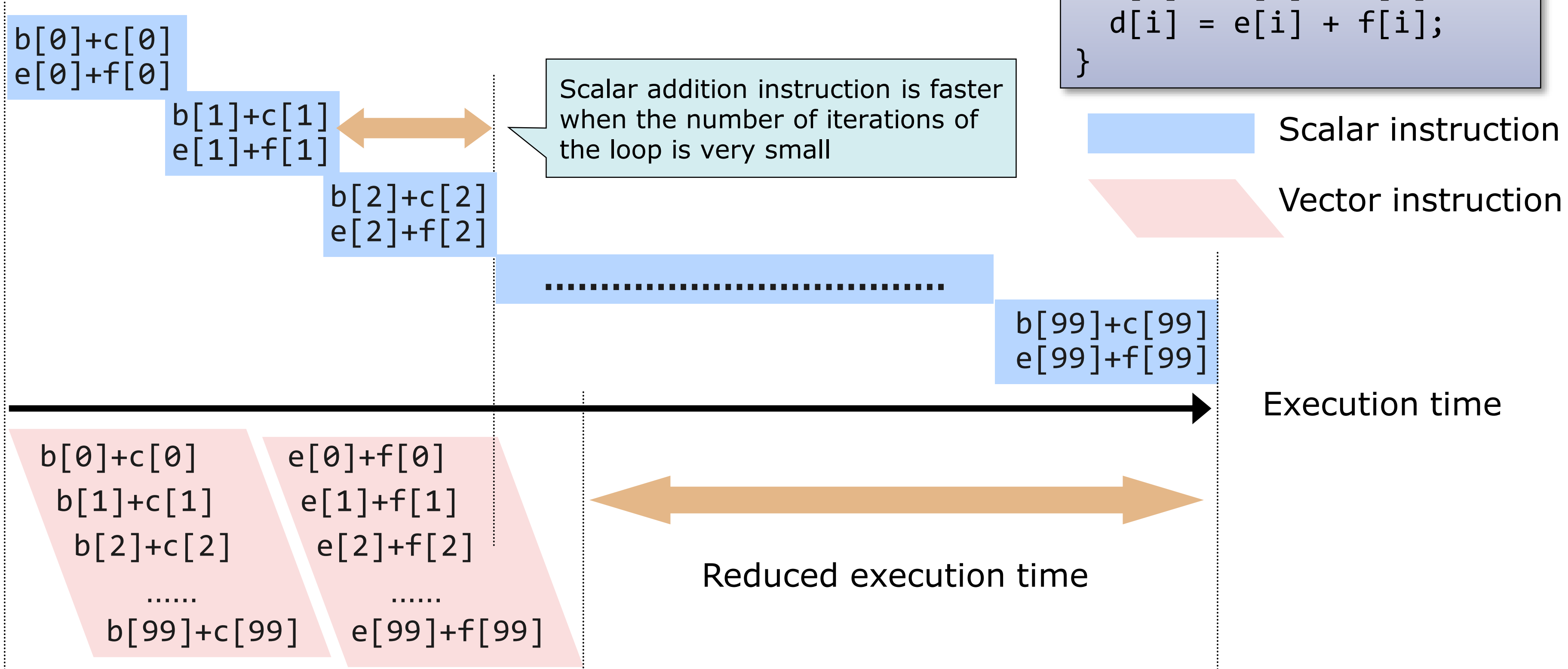
In Vector Engine, up to 256 array elements can be collected into vector register and calculation can be executed at once.

© NEC Corporation 2019

Comparison of Instruction Execution Time

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

```
for (i = 0; i < 100; i++)
{
  a[i] = b[i] + c[i];
  d[i] = e[i] + f[i];
}
```



Note that the order of addition has changed.
(“**b[1]+c[1]**” is added faster than “**e[0]+f[0]**”)

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

Execution image of vector addition instruction

© NEC Corporation 2019

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.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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

```
for (i=2; i < n; i++)
  a[i+1] = a[i] * b[i] + c[i];
```

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];
  :       :
```

a[n] : Updated "a" value

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];
```

before update

Example 2

```
for (i=2; i < n; i++)
  a[i-1] = a[i] * b[i] + c[i];
```

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.

© NEC Corporation 2019

Unvectorizable Dependencies (2)

Example 3

```
for (i = 0; i < n; i++) {
    a[i] = s;
    s = b[i] + c[i];
}
```

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

Calculation order in scalar

```
a[0] = s ;
s = b[0] + c[0] ;
a[1] = s ;
s = b[1] + c[1] ;
      :
```

Calculation order in vector

```
a[0] = s ;
a[1] = s ;
      :
a[n-1] = s ;
s = b[0] + c[0] ;
s = b[1] + c[1] ;
      :
```

```
a[0] = s
for (i = 1; i < n; i++) {
    s = b[i-1] + c[i-1];
    a[i] = s;
}
s = b[n-1] + c[n-1];
```

It can be vectorized by transforming the program.

Calculation order in scalar

```
a[0] = s ;
s = b[0] + c[0] ;
a[1] = s ;
s = b[1] + c[1] ;
      :
```

Calculation order in vector

```
a[0] = s ;
s = b[0] + c[0] ;
s = b[1] + c[1] ;
      :
a[1] = s ;
a[2] = s ;
      :
```


© NEC Corporation 2019
\Orchestrating a brighter world NEC

Unvectorizable Dependencies (3)

Example 4

```
s = 1.0;
for (i=0; i < n; i++) {
    if (a[i] < 0.0)
        s = a[i];
    b[i] = s + c[i];
}
```

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

Example 5

```
for (i=0; i < n; i++) {
    if (a[i] < 0.0)
        s = a[i];
    else
        s = d[i];
    b[i] = s + c[i];
}
```

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

Example 6

```
for (i=1; i < n; i++) {
    a[i] = a[i+k] + b[i];
}
```

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

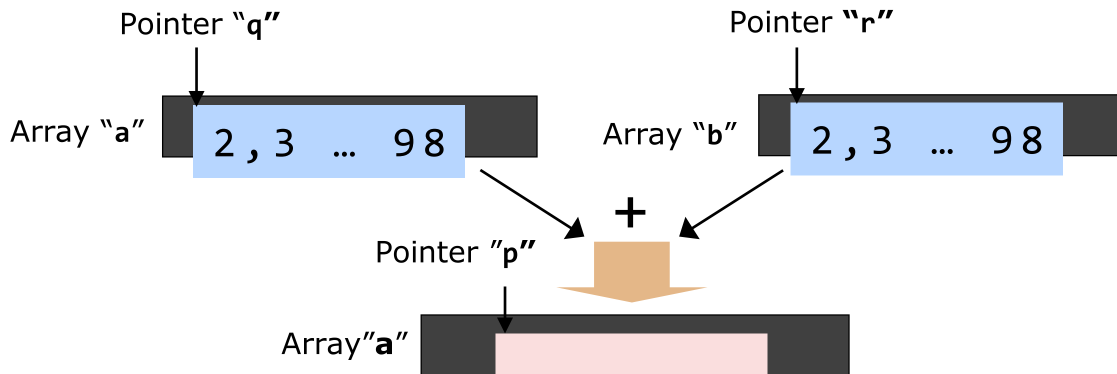
© NEC Corporation 2019

C/C++ Pointer and Vectorization

Ex 1: Cannot be vectorized when **p = & a [3], q = & a [2]**

Pattern of a[i+1]=a[i]+…

```
for (i = 2 ; i < n; i++) {
  *p = *q + *r;
  p++, q++, r++;
}
```



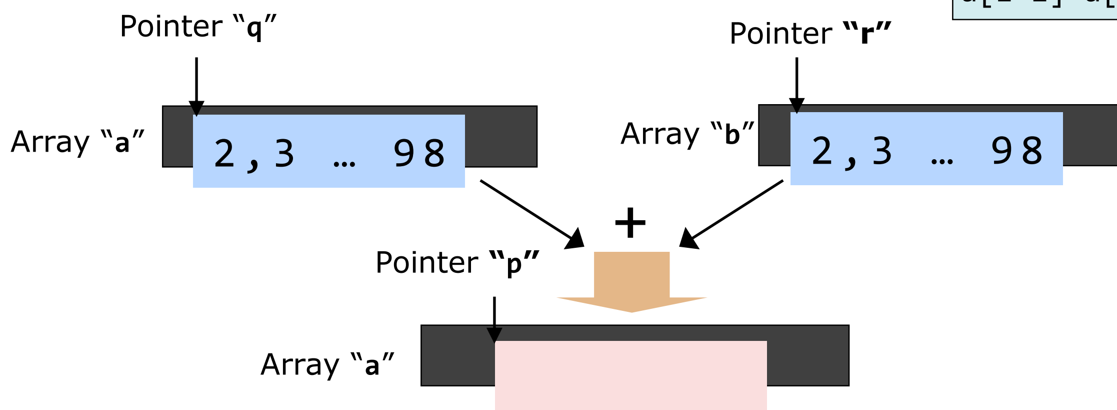
The pointer value is determined when program is executed.

It is regarded as unvectorizable dependency and not vectorized to avoid generating incorrect results, unless it is clear that there are no dependencies.

*Specifying the compiler option or **#pragma** to indicate that there are no dependecies.*

Ex 2: Can be vectorized when p=&a[1], q=&a[2]

Pattern of a[i-1]=a[i]+…

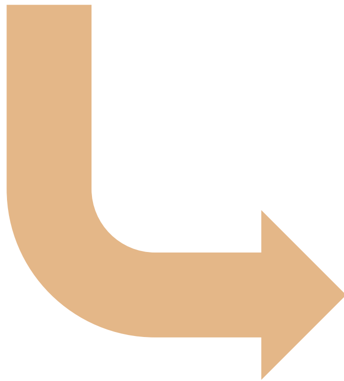


© NEC Corporation 2019

Vectorization of IF Statement

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

```
for (i = 0, i < 100; i++) {
  if (a[i] < b[i]) {
    a[i] = b[i] + c[i];
  }
}
```



Execute with vector operations

```
mask[1]   = a[1]  <  b[1]
mask[2]   = a[2]  <  b[2]
  :           :        :
mask[100] = a[100] < 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]
```


© NEC Corporation 2019

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**

```
$ ncc –fdiag-vector=2 abc.c
…
ncc: vec( 103): abc.c, line 1181: Unvectorized loop.
ncc: vec( 113): abc.c, line 1181: Unvectorizable dependency is assumed.: *(p)
ncc: vec( 102): abc.c, line 1234: Partially vectorized loop.
ncc: vec( 101): abc.c, line 1485: Vectorized loop.
…
$ ncc –report-diagnostics abc.c
…
$ less abc.L
FILE NAME: abc.c
…
FUNCTION NAME: func
DIAGNOSTIC LIST

 LINE               DIAGNOSTIC MESSAGE

  1181  vec( 103): Unvectorized loop.
  1181  vec( 113): Unvectorizable dependency is assumed.: *(p)
  1234  vec( 102): Partially vectorized loop.
  1485  vec( 101): Vectorized loop.
…
```

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

List file name is "source file name".L

© NEC Corporation 2019

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.

```
$ ncc -report-format a.c -c
$ less a.L
        :
FUNCTION NAME: func
FORMAT LIST

 LINE   LOOP      STATEMENT

     5:           void func(double *x, double *y, int n )
     6:           {
     7: +------>      for (int j = 0; j < n; j++) {
     8: |V----->          for (int i = 0; i < m; i++)
     9: |V-----               a[i] += b[i] * c[j];
    10: +------       }
    11:
    12: +------>      for (int j = 0; j < n; j++) {
    13: |+----->          for (int i = 0; i < m; i++)
    14: |+-----               x[j] = y[j] * a[i];
    15: +------       }
    16:           }
…
```

List file name is "source file name".L

The whole loop is vectorized.

The loop is not vectorized.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Extended Vectorization Features

Orchestrating a brighter world NEC

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

© NEC Corporation 2019

Statement Replacement

Source Program

```
for (i = 0; i < 99; i++) {
    a[i] = 2.0;
    b[i] = a[i+1];
}
```

Transformation Image

```
for (i = 0; i < 99; i++) {
    b[i] = a[i+1];
    a[i] = 2.0;
}
```

When this loop is vectorized, all the value from b[0] to b[98] 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.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Loop Collapse

Source Program

```
double a[M][N], b[M][N], c[M][N];
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      a[i][j] = b[i][j] + c[i][j];
```

Transformation Image

```
double a[M][N], b[M][N], c[M][N];
for (ij = 0; ij < M*N; ij++)
    a[0][ij] = b[0][ij] + c[0][ij];
```

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

© NEC Corporation 2019
\Orchestrating a brighter world NEC

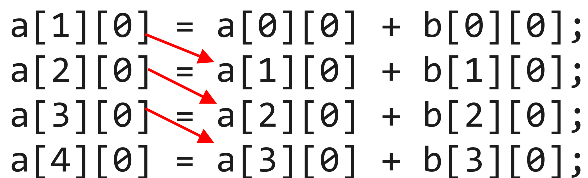
Loop Interchange

Source Program

```
for (j = 0; j < M; j++) {
  for (i = 0; i < N; i++) {
    a[i+1][j] = a[i][j] +  b[i][j];
   }
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      a[i+1][j] = a[i][j] + b[i][j];
   }
}
```



```
a[1][0] = a[0][0] + b[0][0];
a[2][0] = a[1][0] + b[1][0];
a[3][0] = a[2][0] + b[2][0];
a[4][0] = a[3][0] + b[3][0];
```

```
a[1][0] = a[0][0] + b[0][0];
a[1][1] = a[0][1] + b[0][1];
a[1][2] = a[0][2] + b[0][2];
a[1][3] = a[0][3] + b[0][3];
```

The loop "`for (i=0; i<N; i++)`" has unvectorizable dependency about the array `a`.

Interchanging loops removes unvectorizable dependency, and enable the loop "`for (j=0; j<M; j++)`" to be vectorized.

© NEC Corporation 2019

Partial Vectorization

Source Program

```
for (i = 0; i < N; i++) {
    x = a[i] + b[i];
    y = c[i] + d[i];
    func(x, y);
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
    wx[i] = a[i] + b[i];
    wy[i] = c[i] + d[i];
}
for (i = 0; i < N; i++) {
    func(wx[i], wy[i]);
}
```

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.

© NEC Corporation 2019

Conditional Vectorization

Source Program

```
for (i = N; i < N+100; i++) {
    a[i] = a[i+k] + b[i];
}
```

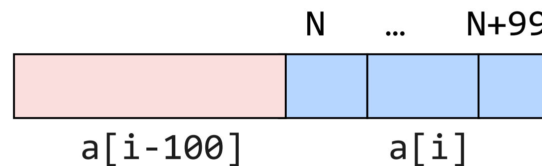
Transformation Image

```
if (k >= 0 || k < -99) {
    // Vectorized Code
}
else {
    // Unvectorized Code
}
```

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];
```



© NEC Corporation 2019

Macro Operations

Sum

```
for (i = 0; i < N; i++)
    s = s + a[i];
```

Iteration

```
for (i = 0; i < N; i++)
    a[i] = a[i-1]*b[i]+c[i];
```

Maximum or minimum values

```
for (i = 0; i < N; i++) {
    if (xmax < x[i])
        xmax = x[i];
}
```

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.

© NEC Corporation 2019

Outer Loop Vectorization

Source Program

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
        a[i][j] = 0.0;
    b[i] = 1.0;
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
    for (j = 0; j <N; j++)
        a[i][j] = 0.0;
}
for (i = 0; i < N; i++)
    b[i] = 1.0;
```

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.

© NEC Corporation 2019

Loop Fusion

Source Program

```
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
for (j = 0; j < N; j++)
    d[j] = e[j] * f[j];
```

Transformation Image

```
for (i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] * f[i];
}
```

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

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Vectorization with Inlining

Source Program

```
for (i = 0; i < N; i++) {
    b[i] = func(a[i]);
    c[i] = b[i];
}
…
double func(double x)
{
    return x*x;
}
```

Transformation Image

```
for (i = 0; i < N; i++) {
    b[i] = a[i] * a[i];
    c[i] = b[i];
}
…
double func(double x)
{
    return x*x;
}
```

When the **`-finline-functions`** option is specified, the compiler expand 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.

© NEC Corporation 2019

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”.

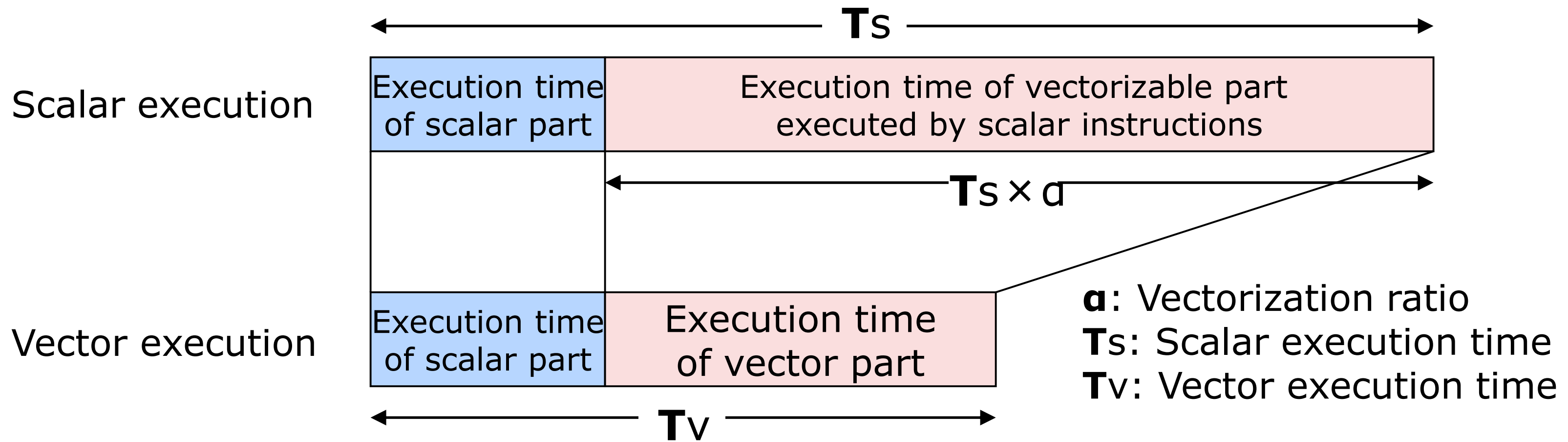
Improving Memory Access Efficiency

- Avoid using a list vector.

© NEC Corporation 2019

Vectorization Ratio

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



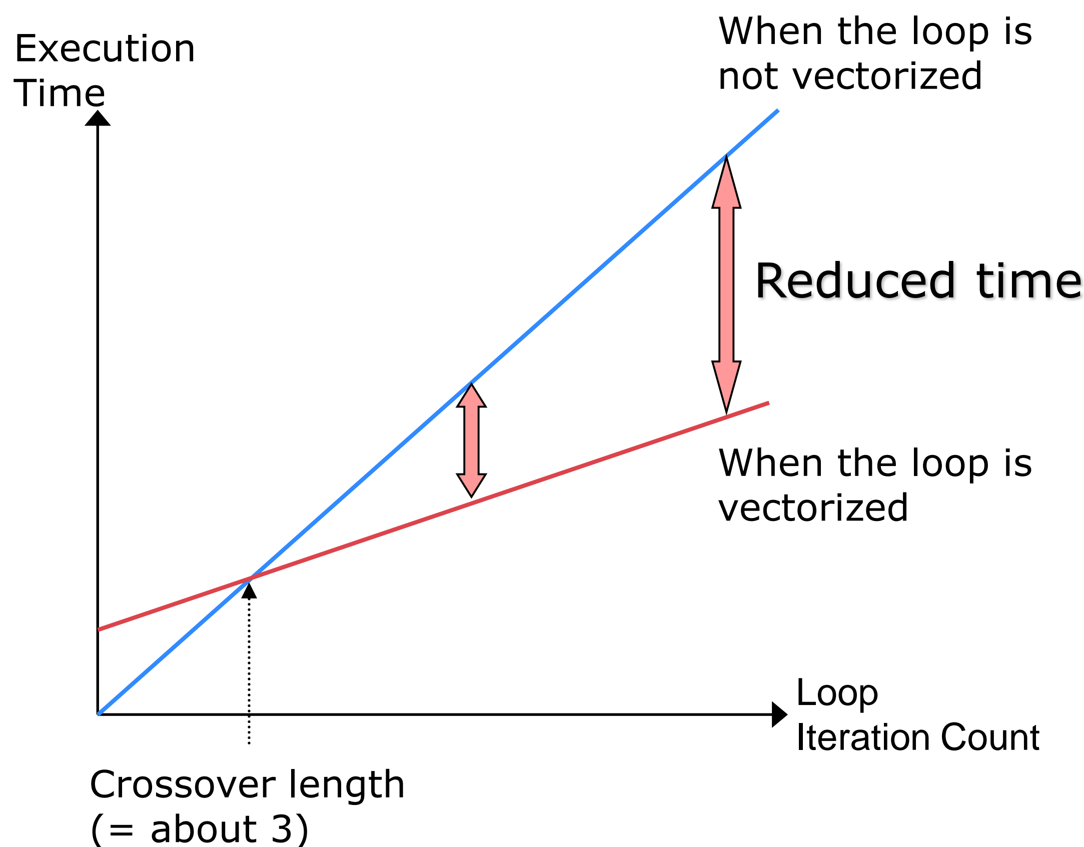
- The vector operation ratio is used instead of the vectorization ratio

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

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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 **`#pragma`** directives

© NEC Corporation 2019

PROGINF

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.

© NEC Corporation 2019

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 funcA
    15000     3.541( 31.7)     0.236  73505.6  56940.8  99.46 216.0     3.555    0.000    0.000  100.00 funcB
    15000     2.726( 24.4)     0.182  71930.1  27556.5  99.43 230.8     2.725    0.000    0.000  100.00 funcC
        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
```


© NEC Corporation 2019

Tuning Techniques

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 **#pragma**.
  - The compiler directive format is as follows.

    #pragma _NEC *directive-name* [*clause*]

  - 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.

```
#pragma _NEC ivdep
  for (i = 2 ; i < n; i++)
   {
    *p = *q + *r;
    p++, q++, r++;
  }
```

- 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.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Dealing with Unvectorizable Dependencies (1)

Raising Vectorization Ratio

```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 113): a.c, line 16: Overhead of loop division is too large.
ncc: vec( 121): a.c, 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

```
for (i=0; i<N; i++) {
   if (x[i] < s)
      t = x[i];
   else if (x[i] >= s)
      t = -x[i];
   y[i] = t;
}
```

Vectorized Loop

```
for (i=0; i<N; i++) {
   if (x[i] < s)
      t = x[i];
   else
      t = -x[i];
   y[i] = t;
}
```

Modified so that variable "**t**" is always defined.

Compiler cannot recognizes sum type macro operation

Unvectorized Loop

```
for (i=0; i<N; i++) {
   if (a[i] < 0.0)
      s = s + b[i];
   else
      s = s + c[i];
}
```

Vectorized Loop

```
for (i=0; i<N; i++) {
   if (a[i] < 0.0)
      t = b[i];
   else
      t = c[i];
   s = s + t;
}
```

Vectorization as a sum type macro operation.

<Diagnostic message after vectorization>

```
ncc: vec( 101): a.c, line 16: Vectorized loop.
ncc: vec( 126): a.c, line 21: Idiom detected.: Sum.
```

Sum type macro operation is vectorized using special HW instruction

© NEC Corporation 2019

Dealing with Unvectorizable Dependencies (2)

Raising Vectorization Ratio

```
ncc: vec( 103): vec_dep2.c, line 7: Unvectorized loop.
ncc: vec( 113): vec_dep2.c, line 7: Overhead of loop division is too large.
ncc: vec( 122): vec_dep2.c, line 8: 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 exit.

Unvectorized Loop

```
#define N 1024
double a[N],b[N],c[N];
void func(int k, int n)
{
    int i;

    for (i=1; i < n; i++)
        a[i+k] = a[i] + b[i];
}
```

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

```
#define N 1024
double a[N],b[N],c[N];
void func(int k, int n)
{
    int i;
#pragma _NEC ivdep
    for (i=1; i < n; i++)
        a[i+k] = a[i] + b[i];
}
```

When it is clear that the pattern is **a[i-1] = a[i]**, specify "**ivdep**" to vectorized.

<Diagnostic message after vectorization>

```
ncc: vec( 101): a.c, line 7: Vectorized loop.
```


© NEC Corporation 2019
\Orchestrating a brighter world NEC

Dealing with Pointer Dependencies


```
ncc: vec( 103): a.c, line 12: Unvectorized loop.
ncc: vec( 122): a.c, line 13: Dependency unknown. Unvectorizable dependency is
assumed.: *(p)
```

- 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.

Vectorized Loop

```
main() {
double *p = (double *) malloc(8*N);
double *q = (double *) malloc(8*N);
…
func(p,q);
…
}
void func(double *p, double *q) {
  …
#pragma _NEC ivdep
  for (int i = 0; i < n; i++) {
      p[i] = q[i];
  }
}
```

There is no unvectorizable dependency between **p[i]** and **q[i]** because it is an area secured separately by **malloc(3C)**, but it is not known in function "**func ()**"

It is clear to the programmer that there is no unvectorizable dependency, so you can specify "**ivdep**".

Even if "**ivdep"** is specified, the compiler ignores it and does not vectorize the loop when there is a clearly unvectorizable dependency.

NOTE: Specifying ivdep may result in invalid results when there is a dependency that cannot be vectorized in practice

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Dealing with Pointer Dependencies: restrict Keyword

Raising Vectorization Ratio

```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 121): a.c, line 18: Unvectorizable dependency is assumed: *(p)
```

restrict is a keyword that can be used in pointer declarations

- It indicates that only the pointer itself will be used to access the object to which it points.
- The compiler assumes that the pointers with **restrict** keyword point to different locations and there is no unvectorizable dependency between them, and can vectorize the loop which contains them.

Vectorized Loop

```
void func(double * restrict p, double * restrict q)
{
  …
  for (int i = 0; i < N; i++) {
      p[i] = q[i];
  }
  …
}
```

Even if "**restrict**" is specified, the compiler does not vectorize the loop when it is clear that the object is accessed by another pointer or variable.

*NOTE: Specifying "**restrict**" may result in invalid results if the object is acutually accessed by another pointer or variable.*

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Vectorization by Improving Loop Structure

Raising Vectorization Ratio

```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 121): a.c, line 16: Unvectorizable loop structure.
```

Unvectorizable loop structure

- The induction variable is type converted.
- The equality operator (**==**) or the inequality operator (**!=**) appears in a loop-termination-expression.

The iteration count of the loop cannot be determined at compilation.

- A logical AND operator (**&&**) or a logical OR operator (**||**) appears in a loop-termination-expression.

Multiple branches in loop-termination-expression

```
for (j=0; j < m; j++) {
  for (i=0; i < n; i++) {
    a[i] = b[j] + c[j];
  }
}
```

- "**i**" and "**j**" are induction variables.
- "**j<m**" and "**i<n**" are loop-termination-expression.

The induction variable is a variable that is increased or decreased by a fixed amount per loop iteration.

© NEC Corporation 2019

Equality Operator in Loop-termination-expression

Raising Vectorization Ratio

- When the equality operator (==) or the inequality operator (!=) appears in a loop-termination-expression, it cannot be determined whether the expression becomes true or not during the loop execution.
  - Use the relational operators <, >, <= or >= in the loop-termination-expression to vectorize the loop.

Unvectorized Loop

```
for (i=0; i != n; i+=2) {
    ......
}
```

The condition is not satisfied when n is an odd number

Vectorized Loop

```
for (i=0; i < n; i+=2) {
    ......
}
```

Fix to "i <n"

Unvectorized Loop

```
double *first, *last, *p;
......
for (p=first; p != last; p++)
{
    ......
}
```

C ++ iterator type array

Vectorized Loop

```
double *first, *last, *p;
......
for (p=first; p < last; p++)
{
    ......
}
```

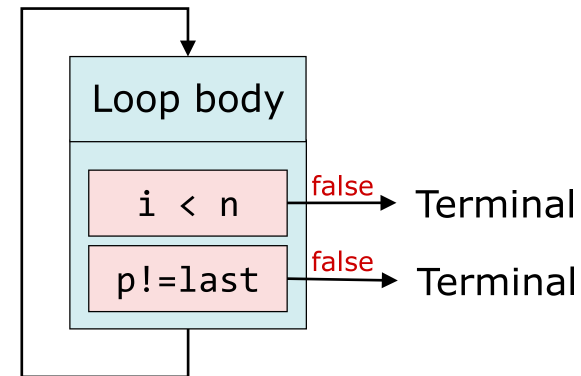

© NEC Corporation 2019

Logical AND/OR Operator in Loop-termination-expression

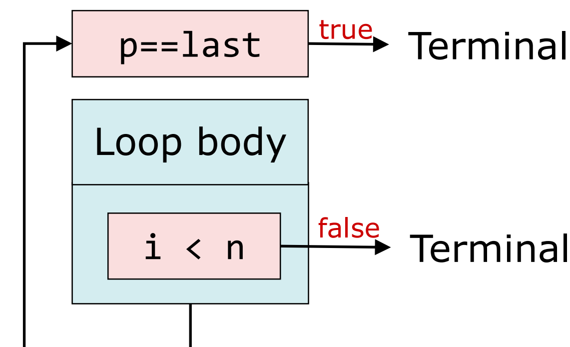
- When a logical AND operator (**&&**) or a logical OR operator (**||**) appears in a loop-termination-expression, two branches are generated for the expression and the loop cannot be vectorized.
  - Modify the source code so as to avoid using (**&&**) or (**||**) the loop-termination-expression.
  - Part of the loop-termination-expression is moved into the loop body to remove the branch from the loop-termination-expression.

```
double func(double *first, double *last, double *a, int n)
{
    double *p = first;
    double sum = 0.0;
                      /* Unvectorizable loop structure */
    for (int i = 0; i < n && p != last; i++, p++) {
        sum += a[i] * (*p);
    }
    return sum;
}
```

```
double func(double *first, double *last, double *a, int n)
{
    double *p = first;
    double sum = 0.0;
                      /* Vectorizable */
    for (i = 0; i < n; i++, p++) {
        if (p == last) break;
        sum += a[i] * (*p);
    }
    return sum;
}
```



Processing of loop-termination-expression



© NEC Corporation 2019
\Orchestrating a brighter world NEC

Inlining: Improving of Vectorization


```
ncc: vec( 103): a.c, line 9: Unvectorized loop.
ncc: vec( 110): a.c, line 10: Vectorization obstructive procedure reference.: fun
```

- When a function call prevents vectorization, above messages are output
- Try to inlining with either of the following
  - Specify "**-finline-functions**" option
  - Specified as inline function at function declaration

```
#include <math.h>
double fun(double x, double y)
{
  return sqrt(x)*y;
}
…
for (i=0; i<N; i++) { // Unvectorized
    a[i] = fun(b[i], c[i]) + d[i];
  }
…
```

"**double sqrt (double)**" is vectorizable function, so it does not prevent vectorization

<When specifying inline function>

```
#include <math.h>
inline double fun(double x, double y)
{
  return sqrt(x)*y;
}
…
for (i=0; i<N; i++) {  // Vectorized
    a[i] = fun(b[i], c[i]) + d[i];
  }
…
```

<When specifying compiler option>

```
$ ncc -finline-functions a.c
```


© NEC Corporation 2019
\Orchestrating a brighter world NEC

A Loop Contains an Array with a Vector Subscript Expression

Raising Vectorization Ratio

```
ncc: vec( 103): a.c, line 8: Vectorized loop.
ncc: vec( 126): a.c, line 9: Idiom detected.: List Vector
```

Specifying **ivdep** for the list vector further improves 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)

```
#pragma _NEC list_vector
for (i=0; i < n; i++) {
    a[ix[i]] = a[ix[i]] + b[i];
}
```

Vectorized Loop (**"ivdev"** Directives)

```
#pragma _NEC ivdep
for (i = 0; i < n; i++) {
    a[ix[i]] = a[ix[i]] + b[i];
}
```

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, if there are no duplicate values in "ix[i]", *more efficient vector instructions can be generated by specifying **ivdep** instead of **list_vector***.

<Message after vectorization by **ivdep**>

```
ncc: vec( 101): a.c, line 8: Vectorized loop.
```


© NEC Corporation 2019

Outer Loop Unrolling

Raising Vectorization Ratio

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.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}
```

Instert **outerloop_unroll(4)** directives

```
#pragma _NEC outerloop_unroll(4)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        a[i][j] = b[i][j] + c[j]
    }
}
```

specify 2^x times unrolling in parentheses.

Program after unrolling the outer loop 4 times.

```
for (int i = 0; i < (n%3); i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}

for (int i = (n%3); i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j]   = b[i][j]   + c[j];
        a[i+1][j] = b[i+1][j] + c[j];
        a[i+2][j] = b[i+2][j] + c[j];
        a[i+3][j] = b[i+3][j] + c[j];
    }
}
```

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_unrol**l" directives>

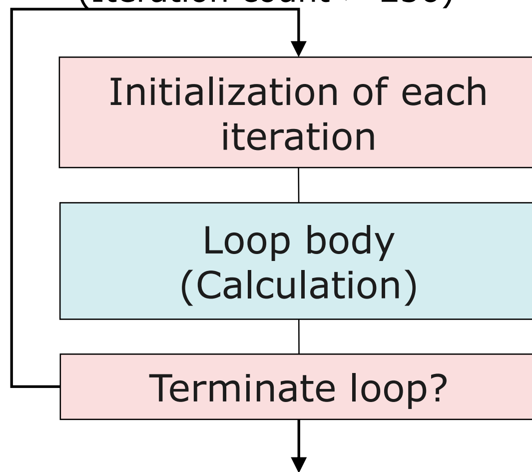
```
ncc: opt(1592): a.c, line 3: Outer loop unrolled inside inner loop.: I
ncc: vec( 101): a.c, line 4: Vectorized loop.
```


© NEC Corporation 2019
\Orchestrating a brighter world NEC

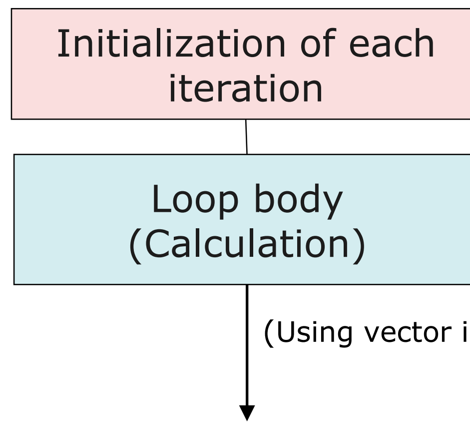
Small Iteration Loop

Raising Vectorization Ratio

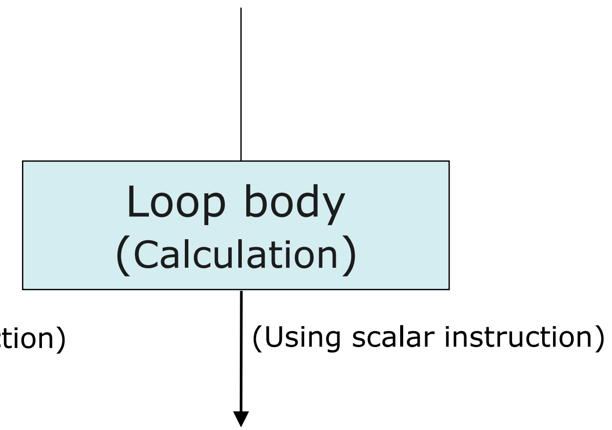
- When the iteration count is small, loop controlling expressions can be eliminated
  - The iteration count <= 256 : A short-loop which does not have “terminate loop?” is generated.
  - The iteration count << 256 : The loop is expanded and loop controlling expressions are eliminated.




```
for (i = 0; i < n; i++) {
...
}
```




```
#pragma _NEC shortloop
for (i = 0; i < n; i++)
{
...
}
```




```
#pragma _NEC unroll_completely
for (i = 0; i < 7; i++) {
...
}
```


© NEC Corporation 2019

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`)	○	○	—
Replaces "`!=`" and "`==`" operator with "`<=`" or "`>=`" at the loop-termination-expression. (`-f[no-]replace-loop-equation`)	○	—	—

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

© NEC Corporation 2019

Influence on Result by Vectorization

Results may differ within error range with and without vectorization

- “Conversion of division to multiplication” or “reordering of arithmetic 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”** directives. The loop is not vectorized.
- Specify the **“nofma”** directives. Vector fused-multiply-add instruction does not generated.

```
#pragma _NEC novector
for (i=0; i < n; i++) {
    sum += a[i];
}
```


© NEC Corporation 2019

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, the float type (aligned in 4 bytes) arrays "**a**" and "**b**" which are passed as arguments are casted to **uint64_t**. Therefore, vector load/sore for 8 bytes elements is executed for them.
- 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.

```
void func1(){
  float a[512],b[512];
  func2(a,b);
}
```

```
void func2( void* a, void* b ){
  for(int i=0; i<256; ++i){ //!!!<---vectorized loop
    ((uint64_t*)b)[i] = ((uint64_t*)a)[i];
  }
}
```

Access the array as data of 4 bytes data type or specify the **novector** directive to the loop to stop vectorization

Access an array as 4 bytes data type

```
void func2( void* a, void* b ){
  for(int i=0; i<512; ++i){
    ((uint32_t*)b)[i] = ((uint32_t*)a)[i];
  }
}
```

Specify **novector** directive

```
void func2( void* a, void* b ){
#pragma _NEC novector
  for(int i=0; i<256; ++i){
    ((uint64_t*)b)[i] = ((uint64_t*)a)[i];
  }
}
```


© NEC Corporation 2019

Automatic Parallelization and OpenMP C/C++

\Orchestrating a brighter world NEC

Automatic Parallelization Features

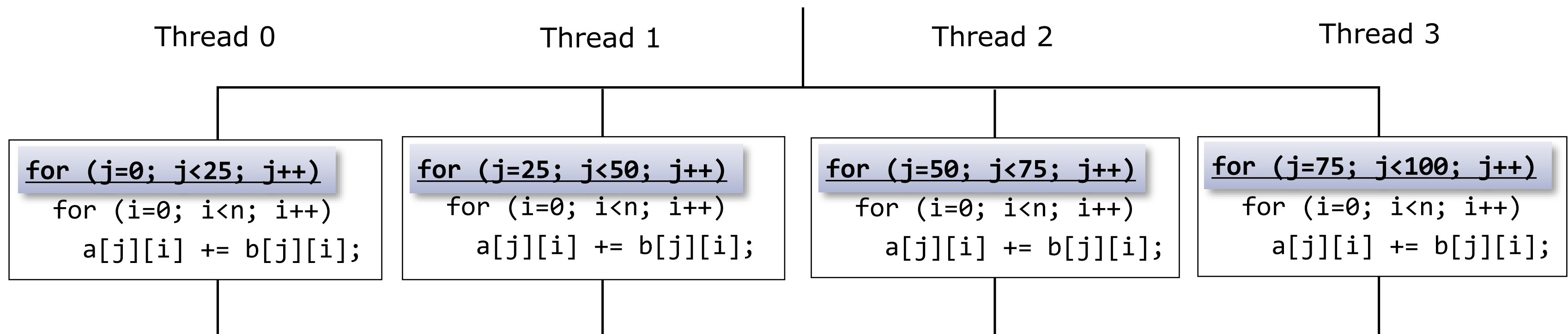
Split one job and execute it simultaneously in multiple threads

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

```
for (j=0; j<100; j++)
  for (i=0; i<100; i++)
    a[j][i] += b[j][i];
```

Serial execution

Example when loop iteration is split into four



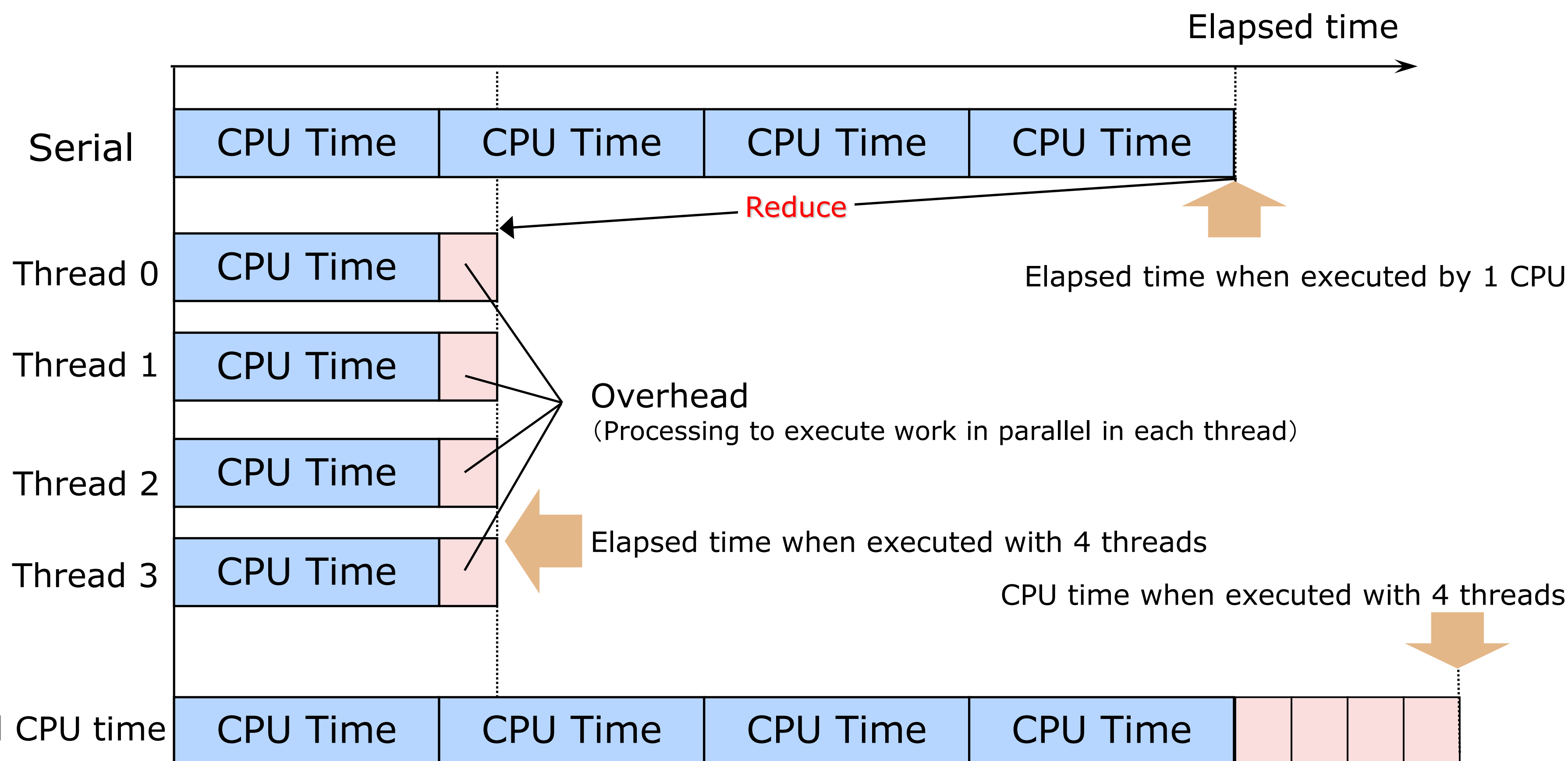
Parallel execution

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Reduce the Elapsed Time by Parallelization

Reduce the elapsed time by parallelization

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



© NEC Corporation 2019

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

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;

 for (j=0; j<n; j++)  {
    for (i=0; i<n; i++)
       sum += a[j] + b[i];
  }

  return sum;
}
```

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

```
$ ncc -mparallel a.c
ncc: par(1801): ex1.c, line 6: Parallel routine generated.: sub$1
ncc: par(1803): ex1.c, line 6: Parallelized by "for".
ncc: vec( 101): ex1.c, 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.

© NEC Corporation 2019

Parallelization Programming Available on Vector Engine

OpenMP C/C++

- 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 C/C++ (**`-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.

© NEC Corporation 2019

OpenMP Parallelization

OpenMP C/C++

```
$ ncc -fopenmp a.c b.c
```

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

- International standards of directives and libraries for shared memory parallel processing
  - "NEC C/C++ 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.

© NEC Corporation 2019

Example: Writing in OpenMP C/C++

*Parallelize function "**sub**" of Example 1 with OpenMP C/C ++*

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;
#pragma omp parallel for
  for (j=0; j<n; j++)  {
     for (i=0; i<n; i++)
        sum += a[j] + b[i];
  }
  …
  return sum;
}
```

Insert OpenMP directives

Search loops that can be execute in parallel

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

```
$ ncc -fopenmp a.c
ncc: par(1801): ex1_omp.c, line 5: Parallel routine generated.: sub$1
ncc: par(1803): ex1_omp.c, line 6: Parallelized by "for".
ncc: vec( 101): ex1_omp.c, line 7: Vectorized loop.
```

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

- The OpenMP directives follows "**#pragma omp**" to specify the parallelization method.

#pragma omp parallel for

parallel

Specify start of parallelization region

for

Specify parallelization of for loop

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

OpenMP Directives

Explain only frequently used items

`#pragma omp parallel for [schedule-clause] [nowait]`

schedule(static[,*size*]) … **schedule(static)** is the 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.

`#pragma omp single`

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

`#pragma omp critical`

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

© NEC Corporation 2019

Automatic Parallelization

Automatic Parallelization

In automatic parallelization, compiler does everything suggested in "Example: Writing in OpenMP C/C++".

```
$ ncc -mparallel a.c b.c
```

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

  #pragma _NEC ***directive-option***

- Major directive options
  - **concurrent/noconcurrent** … parallelize/not-parallelize a loop right after this.
  - **cncall** … parallelize a loop including function calls.

© NEC Corporation 2019

Shared Attribute of Data

Compiler decides shared attributes of data automatically.

- **Shared**
  - Variables outside function scope or declared with **`extern`**, **`static`**.
  - Arguments of functions including parallelized loops and so on.
- **Private**
  - Variables which does not satisfy condition of shared variable.

```
double a[N], b[M];
static double x[M];
double func()
{
    double wk[M];
    double sum = 0.0;

    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            wk[i] = a[i] + b[j];
            sum += x[j]*wk[i];
        }
    }
    return sum;
}
```

•Arrays **a**, **b** and **x** are referenced as shared.
•Array **wk** to contain internal results in a loop and loop control variables **i** and **j** are referred as private.
•Variable **sum** is referred as shared because the calculation result of each thread needs to be added together.

Amount of memory used increases significantly when the size of array **wk** is big because it is shared attribute and allocated in each thread.
Hence, **wk** have better be replaced with scalar variable if possible.

© NEC Corporation 2019

Control Automatic Parallelization with Directives

noconcurrent ... Do not parallelize a loop right after this directive.

```
(void) func(4);              // function call
...
void func(int m) {
#pragma _NEC noconcurrent
    for (j=0; j < m; j++) {  // m is small
        for (i=0; i < n; i++)
            a[i] = b[j] / c[j];
    }
}
```

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.

```
#pragma _NEC cncall
for (i=0; i < m; i++) {
  a[i] = func(b[i], c[i]);
}
```

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

Parallelize by **cncall** when functions can be parallelized.

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

© NEC Corporation 2019

Apply Both OpenMP and Automatic Parallelization

```
$ ncc -fopenmp -mparallel a.c b.c
```

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**.

```
double sub (double *a, int n)
{
  int i, j;
  double b[n][n];
  double sum = 1.0;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      b[i][j] = i * j;

#pragma omp parallel for
  for (j=0; j<n; j++) {
    for (i=0; i<n; i++)
      sum += a[j] + b[i][j];
  }

  return sum;
}
```

Automatic parallelized

OpenMP parallelized

```
$ ncc -fopenmp -mparallel t.c
ncc: par(1801): t.c, line 7: Parallel routine generated.: sub$1
ncc: par(1803): t.c, line 7: Parallelized by "for".
ncc: par(1801): t.c, line 11: Parallel routine generated.: sub$2
ncc: vec( 101): t.c, line 8: Vectorized loop.
ncc: par(1803): t.c, line 12: Parallelized by "for".
ncc: vec( 101): t.c, line 13: Vectorized loop.
```


© NEC Corporation 2019

Behavior of Parallelized Program

Orchestrating a brighter world NEC

Execution Image of Program Parallelized with OpenMP

When parallelized with OpenMP

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;
  double derive;
#pragma omp parallel private(derive)
   {
      derive = 12.3;
#pragma omp for
      for (i = 0; i < n; i++)
        b[i] = derive;
   }
   ….
#pragma parallel omp for   ¥
                        reduction(+:sum)
      for (j=0; j<n; j++)  {
         for (i=0; i<n; i++)
            sum += a[j] + b[i];
       }
   …
  return sum;
}
```

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

Threads are generated before main function

Master thread

Thread 1 Thread 2 Thread 3

Serial region

Execute the same code

Parallel region is taken out as another function by compiler. The function name is **sub$1**.

Parallel region

Execute parallelized loop

Serial region

Barrier sync is done

The function name is **sub$2**

Execute parallelized loop

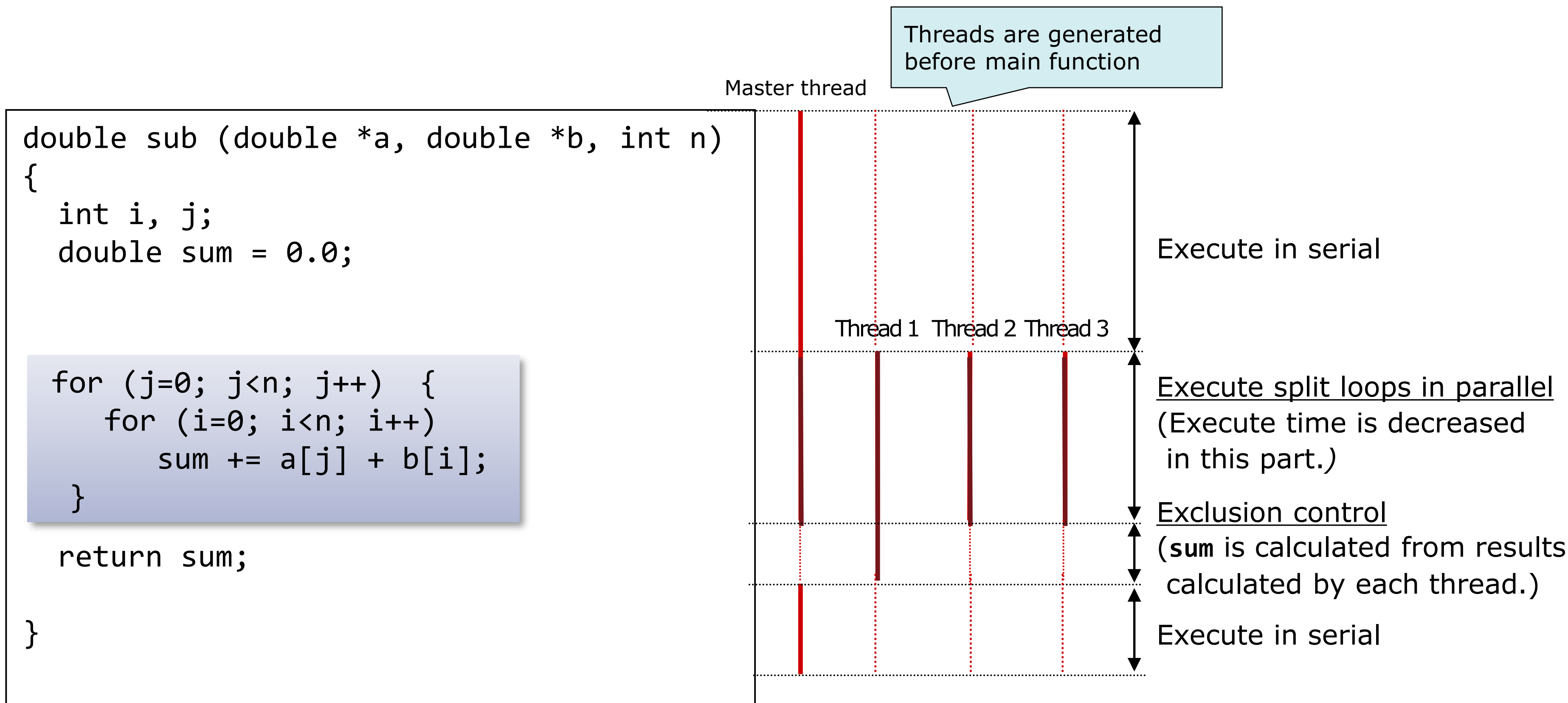
Parallel region

Serial region

Note: VE does not support nested parallelism.

© NEC Corporation 2019

Execution Image of Automatically Parallelized Program

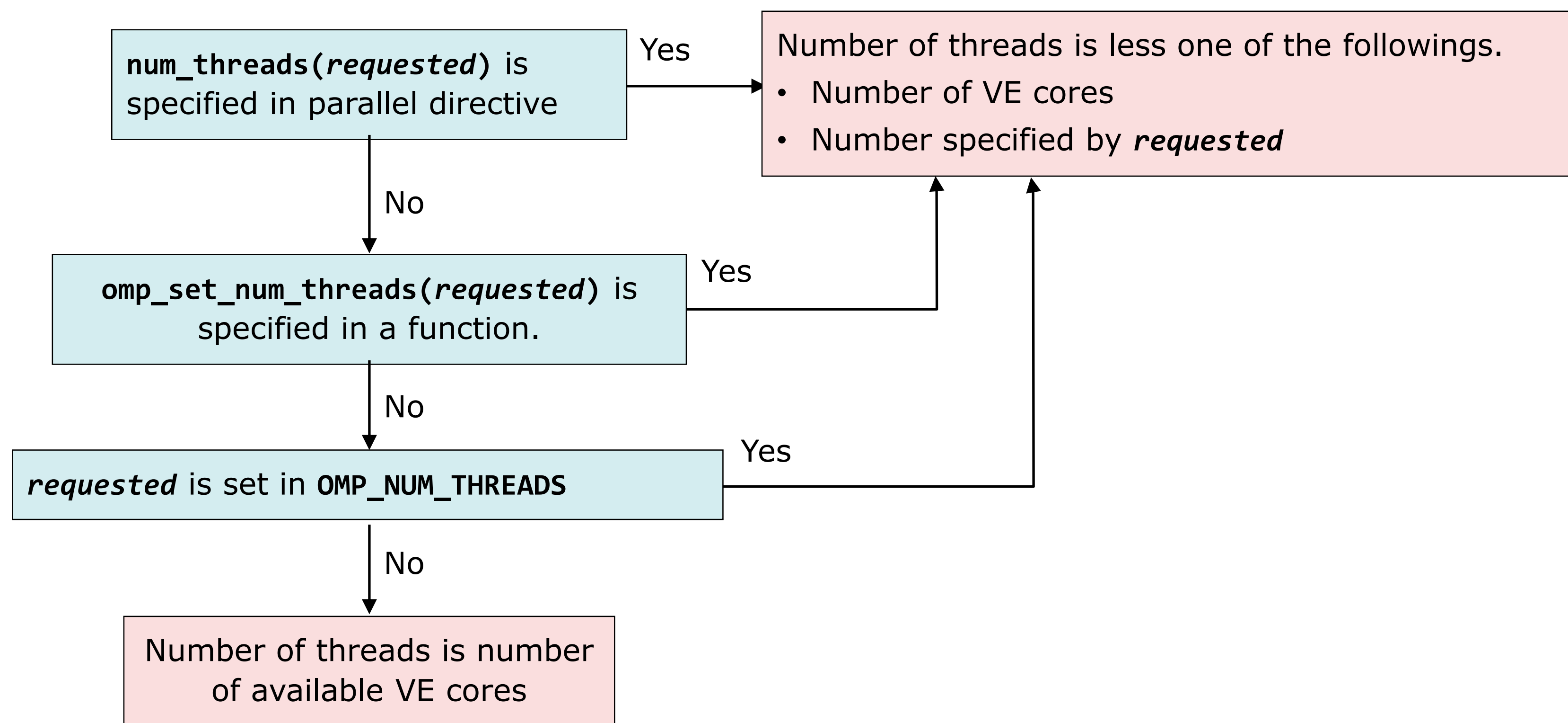


(Solid line：Program execution, Dashed line：Waiting process)

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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.

© NEC Corporation 2019

Tuning Parallelized Program

Point of View in Tuning

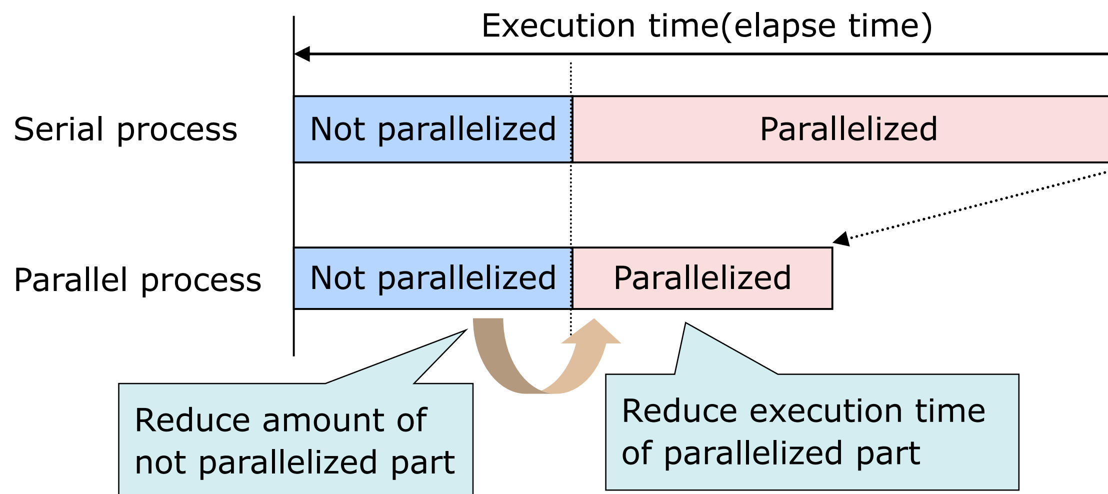
Are there many parts executed in parallel?

- 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.)



© NEC Corporation 2019
\Orchestrating a brighter world NEC

Tuning Flow

Used performance analysis

- **FTRACE**

1. Select loop/function targets of parallelization.

- Find functions whose execution time is long according to information of PROGINF and FTRACE.

2. Increase parallelized part.

- Check if loops in functions 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.

© NEC Corporation 2019

Select Loops for Parallelization

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

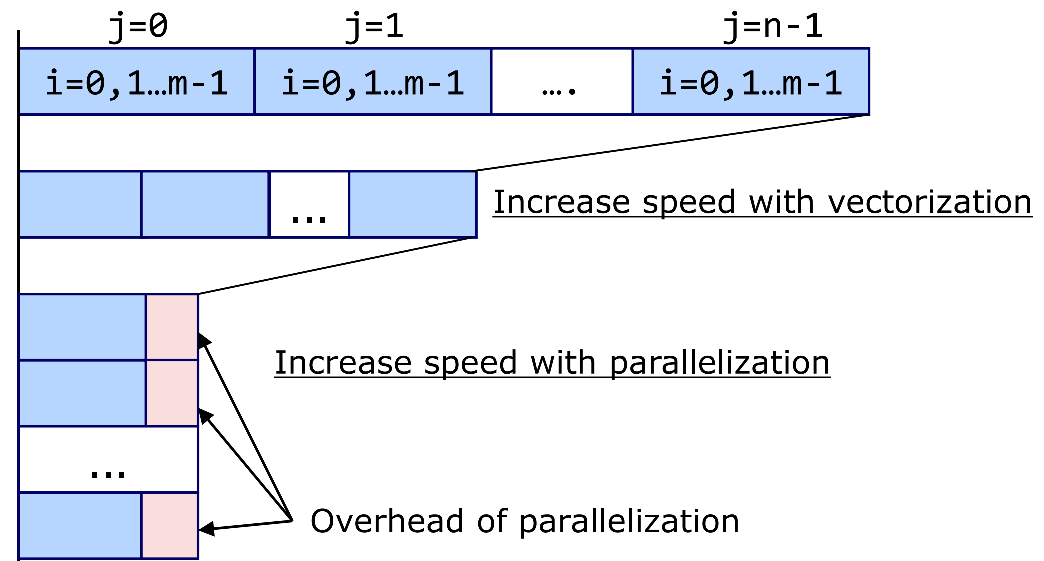
Loops without factors inhibiting parallelization

- Not parallelizable dependencies.
- Not parallelizable control flow.
- Function call like I/O functions 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.

```
for (j = 0; j < n; j++) {
   for (i = 0; i < m; i++) {
      a[j][i] = b [j][i] + c [j][i];
   }
}
```



© NEC Corporation 2019

Not Parallelizable Dependencies

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

Define・Refer the same array element.

```
for (i=0; i<n; i++) {
    a[i] = b[i+1];
  b[i] = c[i];
}
```

Iteration	Reference	Definition	
i=0	b[1]	b[0]	Executed in thread 1
i=1	b[2]	b[1]	
i=2	b[3]	b[2]	Executed in thread 2
i=3	b[4]	b[3]	
:	:	:	

The order of definition and reference of **b[2]** is undefined.

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

Same scalar variable

```
for (i=0; i<n; i++) {
    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 conditions is referred out of it.

```
for (j=0; j<n; j++) {
    for (i=0; i<m; i++) {
        if (a[j][i] >= d ) {
            T = a[j][i] - d;
        }
        c[j][i] = T;
    }
}
```

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

© NEC Corporation 2019

Not Parallelizable Control Flow

Jump from loops

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

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
          if (a[i][j] < 0.0 ) goto label100 ;
        b[i][j] = sqrt(a[j][i] );
    }
}
Label100: ;
```


© NEC Corporation 2019

Add Directives to Promote Parallelization

```
$ ncc -mparallel -fdiag-parallel=2 a.c -c
ncc: opt(1380): a.c, line 6: User function references not ok without "cncall".: func1
```

- 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.

```
void func()
{
    for (int i = 0; i < M; i++) {
        c[i] = func1(a[i], b[i]);
    }
}
```

```
void func()
{
#pragma _NEC cncall
    for (int i = 0; i < M; i++) {
        c[i] = func1(a[i], b[i]);
    }
}
```

```
$ ncc -mparallel -fdiag-parallel=2 a.c -c
ncc: par(1801): a.c, line 7: Parallel routine generated.: func$1
ncc: par(1803): a.c, line 7: Parallelized by "for".
ncc: vec( 103): a.c, line 7: Unvectorized loop.
```


© NEC Corporation 2019
\Orchestrating a brighter world NEC

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 for`** to parallelize.
  - Enable to specify parallelization for loops and statement list.
  - Compiler ignores data dependencies and parallelize them.

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

```
void func()
{
    double wk[M];
#pragma _NEC parallel for private(wk)
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++)
            wk[i] = a[i] + b[j];
        func1(x[j], wk);
#pragma _NEC atomic
        sum += x[j];
    }
}
```

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

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

© NEC Corporation 2019

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.
  - Beware C++ program where a lot of **`new`** operations are used, because **`malloc`**() is used in a **`new`** operation.
- Waiting time for barrier sync with other threads.

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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 function calls in system libraries.
  - Put together **`malloc`**`()` as much as possible.
  - Not to use **`new`** operator to allocate data used only in functions and declare them as local data to allocate them in stack area.
  - Read file contents, map them on memory and read required data from memory when there are enough available area in memory.
- Call **`xxx_unlocked`**`()` functions to use 1 byte I/O functions outside parallel region.
  - `getc(3S)` → `getc_unlocked(3S)`
  - `getchar(3S)` → `getchar_unlocked(3S)`
  - `putc(3S)` → `putc_unlocked(3S)`
  - `putchar(3S)` → `putchar_unlocked(3S)`

© NEC Corporation 2019
\Orchestrating a brighter world NEC

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.

```
#pragma omp for schedule( static )
for (j=m; j>0; j--) {
  for (i=0; i<j; i++) {
    ...
  }
}
```

```
#pragma omp for schedule( dynamic )
for (j=m; j>0; j--) {
  for (i=0; i<j; i++) {
    ...
  }
}
```


© NEC Corporation 2019

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.

```
double a[N], b[N*2], x;
void func()
{
#pragma omp parallel for
    for (i = 0; i < N; i++) {
        a[i] = …
    }
    x = 0.0;
#pragma omp parallel for
    for (i = 0; i < N*2; i++) {
        b[i] = …
    }
    …
}
```

Combine parallel regions

```
double a[N], b[N*2], x;
void func()
{
   x = 0.0;
#pragma omp parallel
  {
#pragma omp for
    for (i = 0; i < N; i++) {
        a[i] = …
    }
#pragma omp for
    for (i = 0; i < N*2; i++) {
        b[i] = …
    }
  }
    …
}
```

Specify **nowait** clause

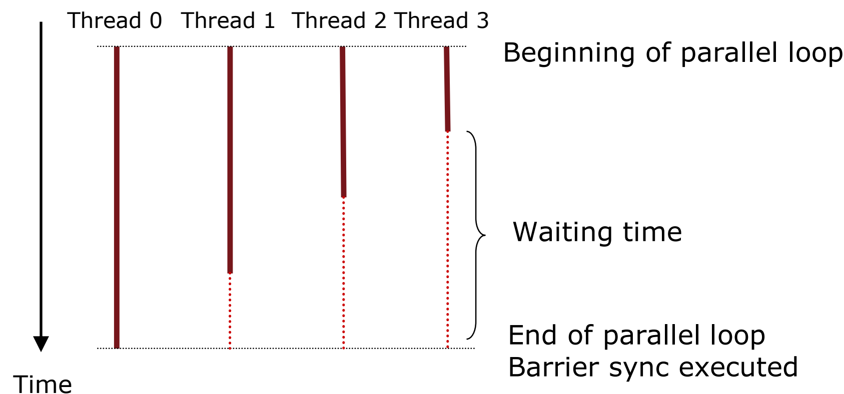
```
double a[N], b[N*2], x;
void func()
{
   x = 0.0;
#pragma omp parallel
  {
#pragma omp for nowait
    for (i = 0; i < N; i++) {
        a[i] = …
    }
#pragma omp for nowait
    for (i = 0; i < N*2; i++) {
        b[i] = …
    }
  }
    …
}
```


© NEC Corporation 2019

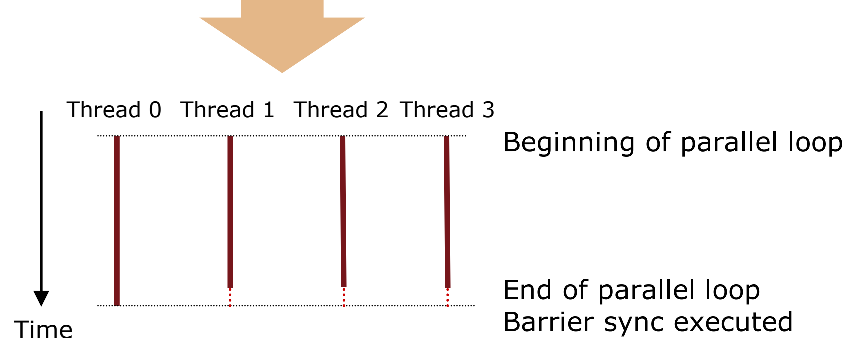
Improve Load Balance (1)

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

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



Improve load balance




```
#pragma omp for
    for ( j = 1024; j > 0; j--) {
        for (i = 0; i < j; i++) {
            …
        }
    }
```

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

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

© NEC Corporation 2019
\Orchestrating a brighter world NEC

Improve Load Balance (2)

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

OpenMP parallelization

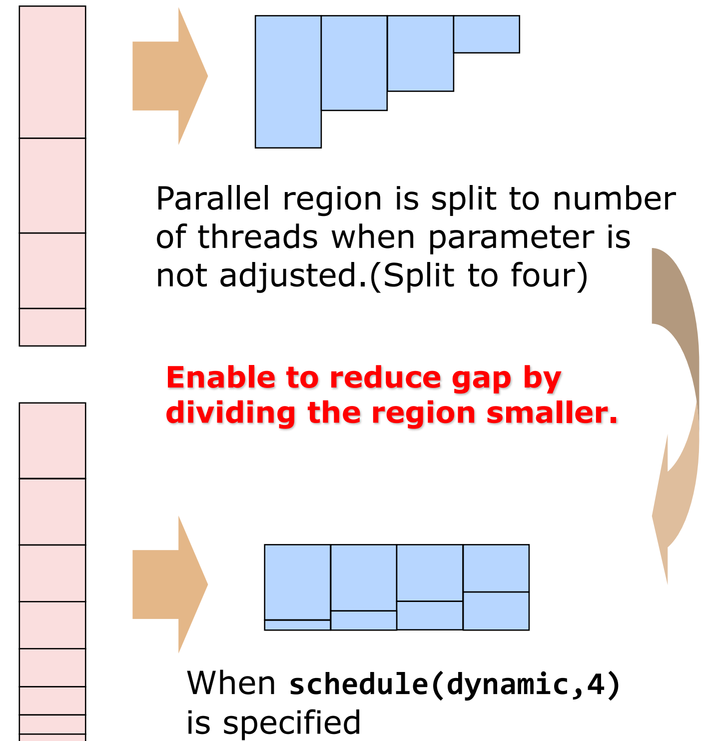
- Adjust parameter of **schedule** clause

```
#pragma omp for schedule(dynamic,4)
    for (j = 1024; j > 0; j--) {
        for (i = 0; i < j; i++) {
            …
        }
    }
```

Automatic parallelization

- Adjust parameter of **schedule** clause in **concurrent** directive as well as OpenMP.

```
#pragma _NEC concurrent schedule(dynamic,4)
    for (j = 1024; j > 0; j--) {
        for (i = 0; i < j; i++) {
            …
        }
    }
```



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

© NEC Corporation 2019

FTRACE

Load balance in functions 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 funcX$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 funcX
...
------------------------------------------------------------------------------------------------------------------
    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 **`#pragma _NEC concurrent schedule(dynamic, 4)`** right before an 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 funcX$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 funcX
...
------------------------------------------------------------------------------------------------------------------
    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 **`funcX$1`**.(Load imbalance)
After :EXCLUSIVE TIME are uniform for each threads and that of **`funcX`** is shorter(time for barrier sync and so on reduced) although that of **`funcX$1`** increases because of time to control threads.

© NEC Corporation 2019

\Orchestrating a brighter world NEC

Area Allocated with malloc(3C) · new Operation

Whether the areas allocated with **malloc**(3C) or **new** operation are shared or private is decided as follows.

- Are pointers pointing to the area shared or private?
- Is process executed in parallel when the area is allocated?

p,q,r : shared
s : private

Parallel process section

```
void func() {
    double *p = malloc(16);
    double *q;
    double *r;
#pragma omp parallel num_threads(4)
    {
        double *s = malloc(16);
#pragma omp critical
        q = malloc(16);
#pragma omp master
        r = malloc(16);
    }
}
```

p = malloc(16) is executed once.
All threads refer the same area.

q = malloc(16) is executed by all threads and four areas are allocated. However, all threads refer only the same area, so remaining three areas are useless.

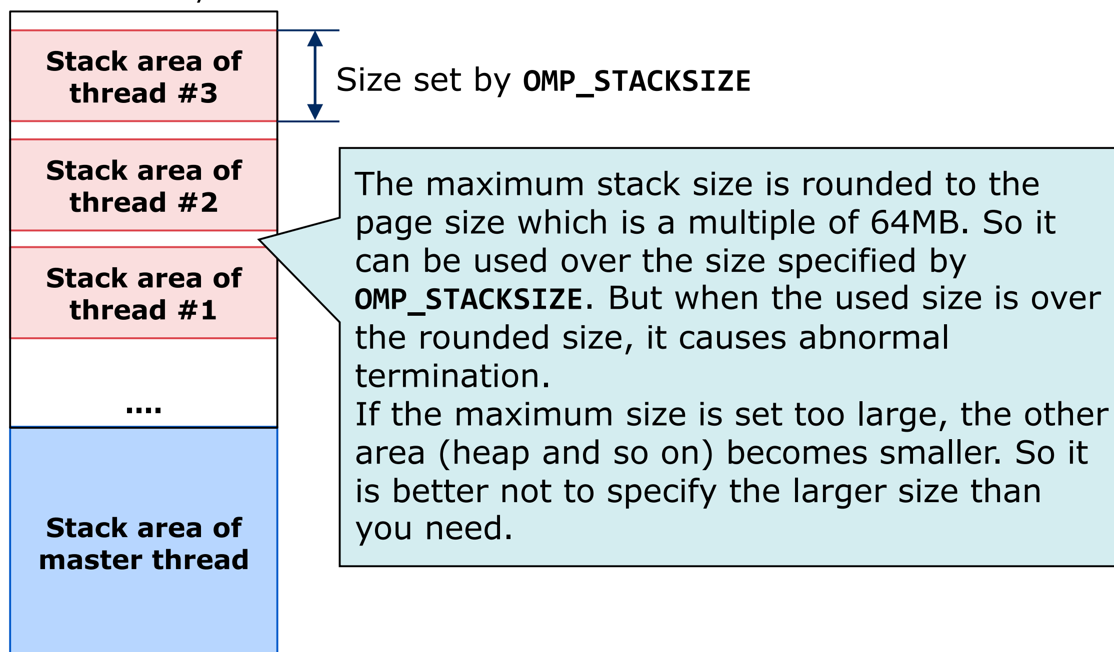
r = malloc(16) is executed by only master thread and only one area is allocated. All threads refer the same area.

s = malloc(16) is executed by all threads and four areas are allocated. Each thread uses separate area.

© NEC Corporation 2019

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.



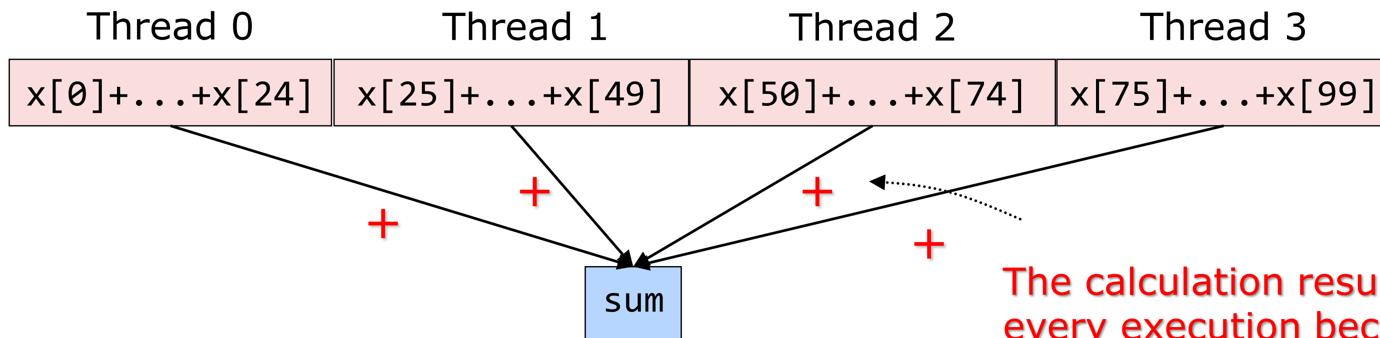

```
$ cat a.c
...
#pragma omp parallel
  {
    double x[16*1024*1024];
    double y[16*1024*1024];
    func(x,y);
  }
...
$ ncc -fopenmp a.c
$ export OMP_STACKSIZE=384M
$ ./a.out
```


© NEC Corporation 2019

Sum Operation

- Sum operation can be parallelized but the order of additions can be changed every time because the order of execution of each thread 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.

```
for (i = 0; i < 100; i++) {
    sum = sum + x[i];
}
```



The calculation results may vary at every execution because the order of these additions is not necessarily same every time.

© NEC Corporation 2019

NEC