Chapter 7
Tuning Tool

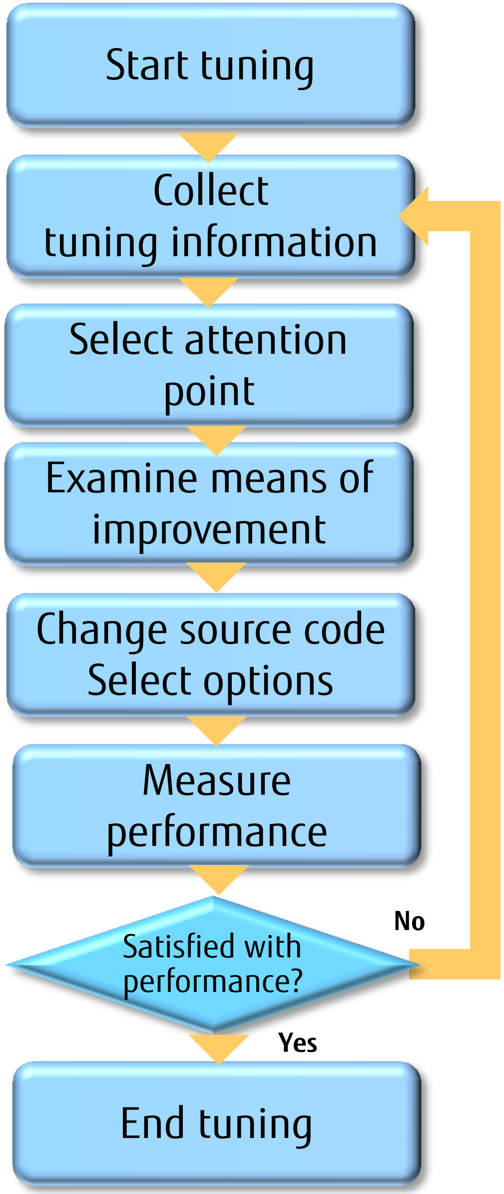
FUJITSU LIMITED
April 2016

Contents

Tuning Workflow

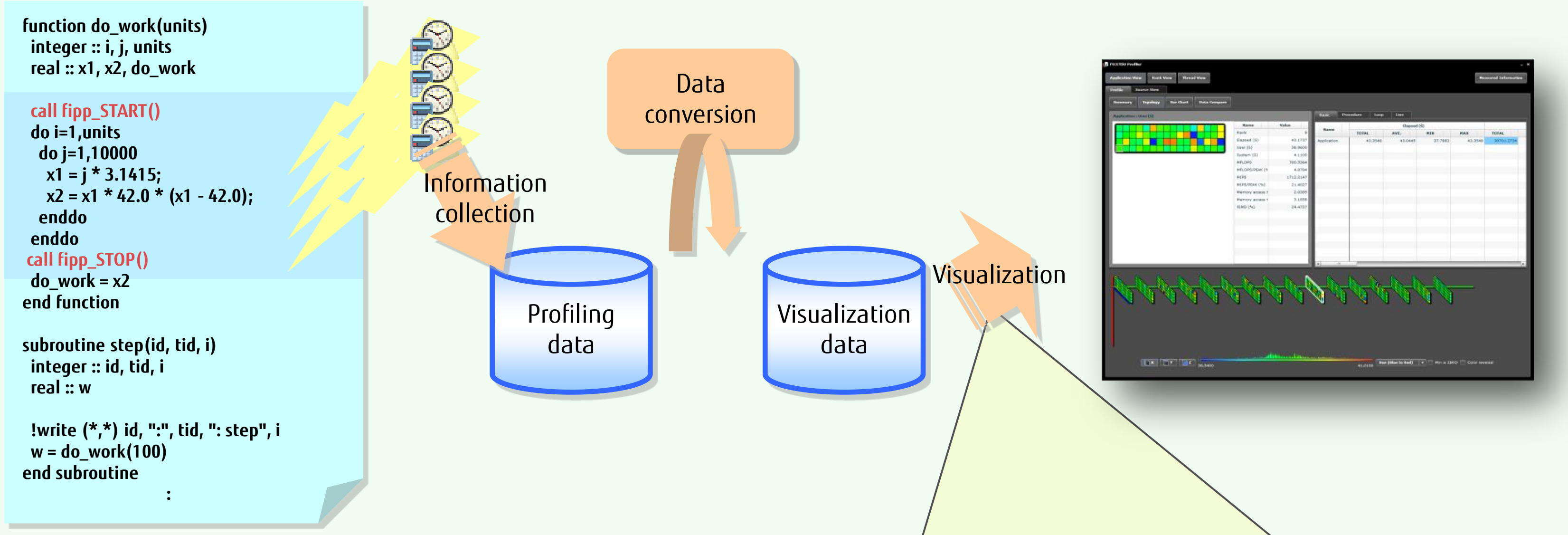
Tuning work and profiler utilization

- To tune an application, perform a series of tasks, including collecting tuning information, examining means of improving the application, correcting the application, and measuring performance.
- Generally, identifying the places that take a long time to execute in an application can have the significant tuning effect of increasing the speed there. You can obtain tuning information, such as the distribution of execution times, by using a profiler. We recommend you start tuning work by analyzing the application with a profiler.
- A profiler can collect tuning information for applications created with the compiler of this product.



Profiler Operation Flow

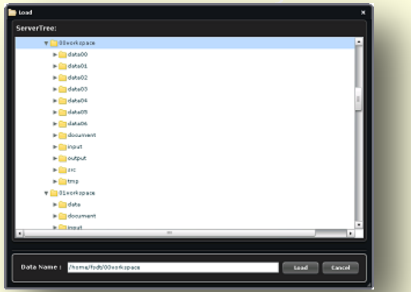
Conceptual diagram of profiler



```
function do_work(units)
  integer :: i, j, units
  real :: x1, x2, do_work

  call fipp_START()
  do i=1,units
    do j=1,10000
      x1 = j * 3.1415;
      x2 = x1 * 42.0 * (x1 - 42.0);
    enddo
  enddo
 call fipp_STOP()
  do_work = x2
end function

subroutine step(id, tid, i)
  integer :: id, tid, i
  real :: w

  !write (*,*) id, ":", tid, ": step", i
  w = do_work(100)
end subroutine
                    :
```

Counter/Timer interrupts, etc. used to collect, convert, and visualize information on application to be analyzed

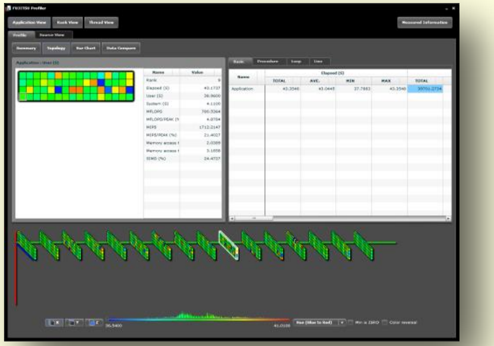
GUI visualization tool started from user client



Data Name : /home/fsdb/00workspace

Data selected for visualization

GUI visualization tool



Profiler Overview

- Features of Each Component
- Information Retrieval and Analysis Procedure (Instant Profiler)
- Instant Profiler Use Example
- Retrievable Information (Instant Profiler)
- Information Retrieval and Analysis Procedure (Advanced Profiler)
- Advanced Profiler Use Example
- Retrievable Information (Advanced Profiler)

Profiler Overview

Profiler configuration

- The provided profilers are the instant profiler and advanced profiler.
- Use each of the two profilers according to the required tuning information.

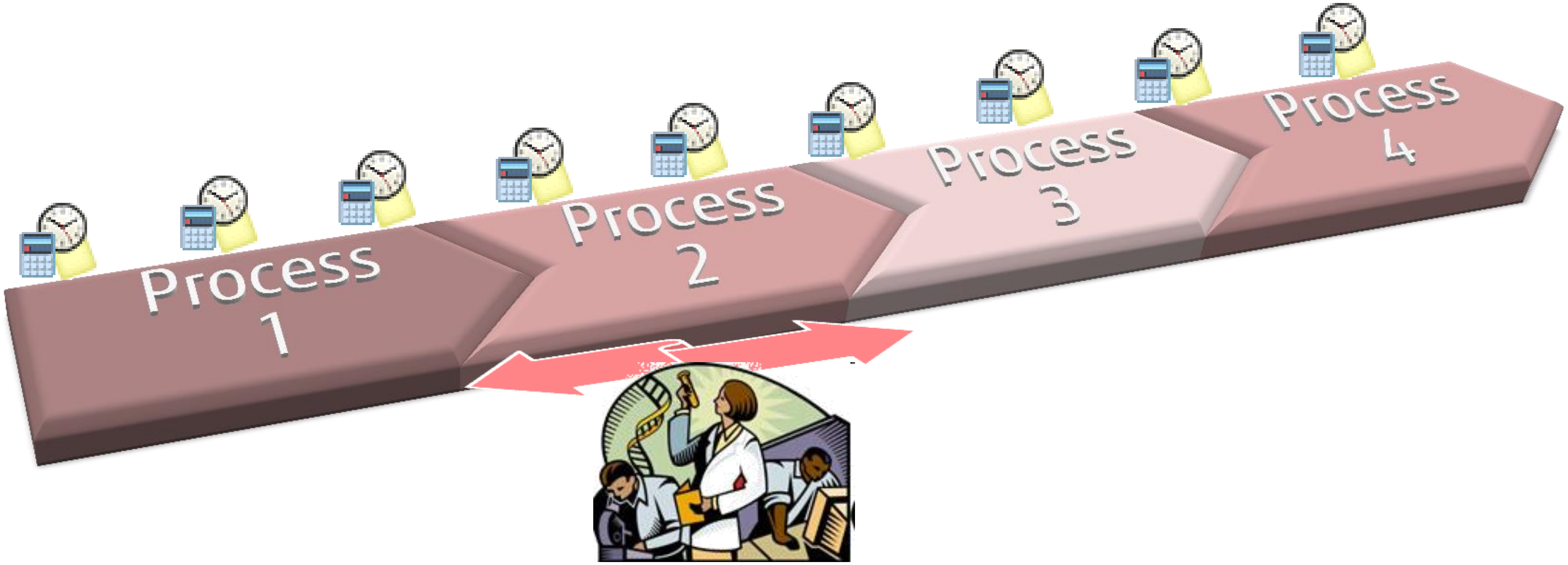
Profiler	Instant profiler ***fipp***	Cost analysis based on sampling. The profiler can be applied at low overhead to any program, but there are some occasions where the profiler has difficulty making a detailed analysis.
	Advanced profiler ***fapp***	Cost and behavior analysis based on the event counter. The profiler can make a detailed analysis of an application, though a measurement section must be set in the program source.

* Sampling is a method of collecting information on the process, thread, procedure, loop, or line that is currently being executed, by interrupting the application at a constant user CPU interval.

* The event counter is a function for collecting PA (Performance Analysis) information.

Features of Each Component

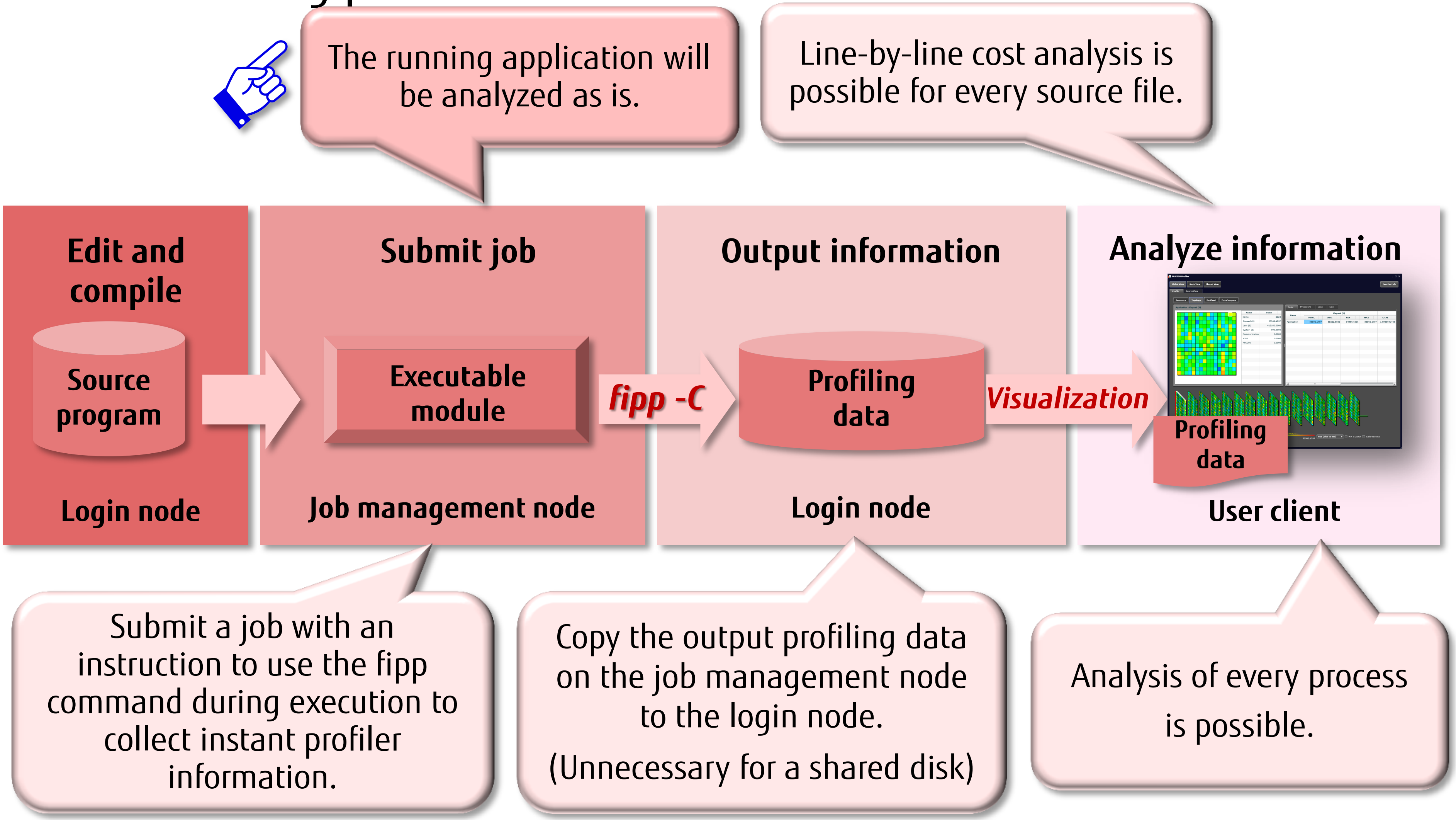
The instant profiler (**fipp**) collects general information by sampling at a constant interval.



The advanced profiler (**fapp**) collects precise information by using a counter within a measurement section.
However, it requires changes in the program.

Profiler Overview: Information Retrieval and Analysis Procedure (Instant Profiler)

- The instant profiler analyzes the costs of an application by using the following procedure.



The running application will be analyzed as is.

Line-by-line cost analysis is possible for every source file.

Submit a job with an instruction to use the fipp command during execution to collect instant profiler information.

Copy the output profiling data on the job management node to the login node.

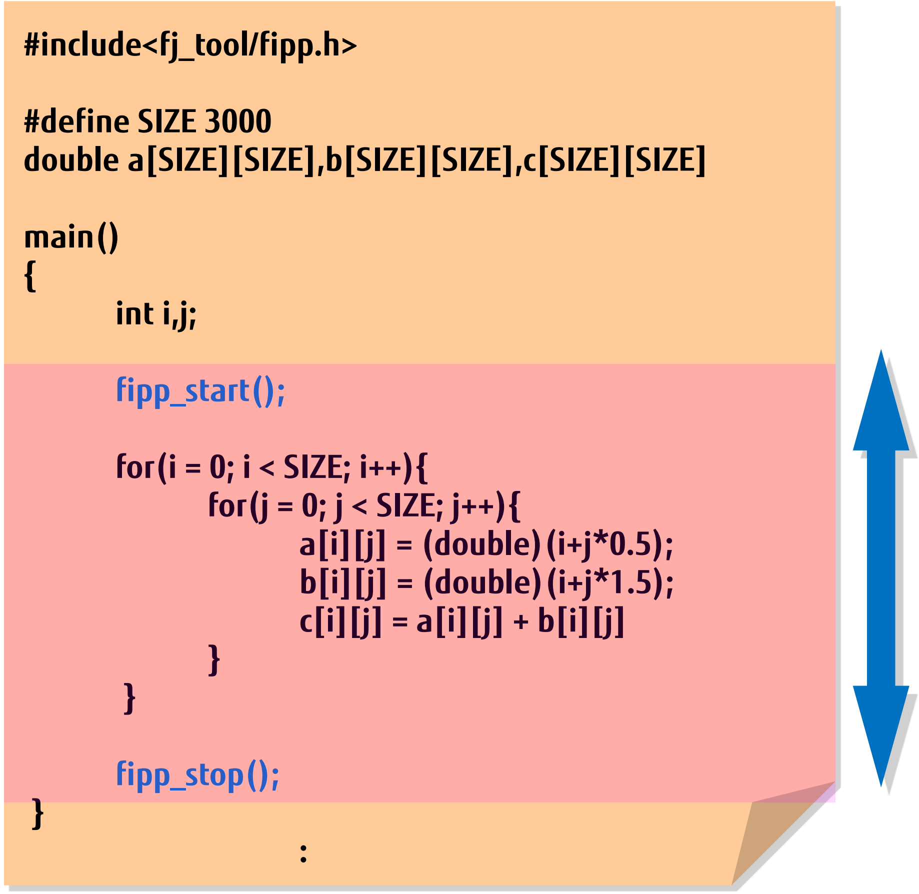
(Unnecessary for a shared disk)

Analysis of every process is possible.

Profiler Overview: Instant Profiler Use Example

- The instant profiler can specify not only an entire application but also the measurement region for cost information.

Language type	Header file	Function name	Function	Argument
Fortran	None	***fipp_start***	Start information measurement	None
		fipp_stop	Stop information measurement	None
C/C++	fj_tool/ fipp.h	***void fipp_start***	Start information measurement	None
		void fipp_stop	Stop information measurement	None



```
#include<fj_tool/fipp.h>

#define SIZE 3000
double a[SIZE][SIZE],b[SIZE][SIZE],c[SIZE][SIZE]

main()
{
        int i,j;

        fipp_start();

        for(i = 0; i < SIZE; i++){
                for(j = 0; j < SIZE; j++){
                        a[i][j] = (double)(i+j*0.5);
                        b[i][j] = (double)(i+j*1.5);
                        c[i][j] = a[i][j] + b[i][j]
                }
        }

        fipp_stop();
}
                        :
```

To use the section specification function of the instant profiler, specify the ***fipp a.out -Srange*** option.

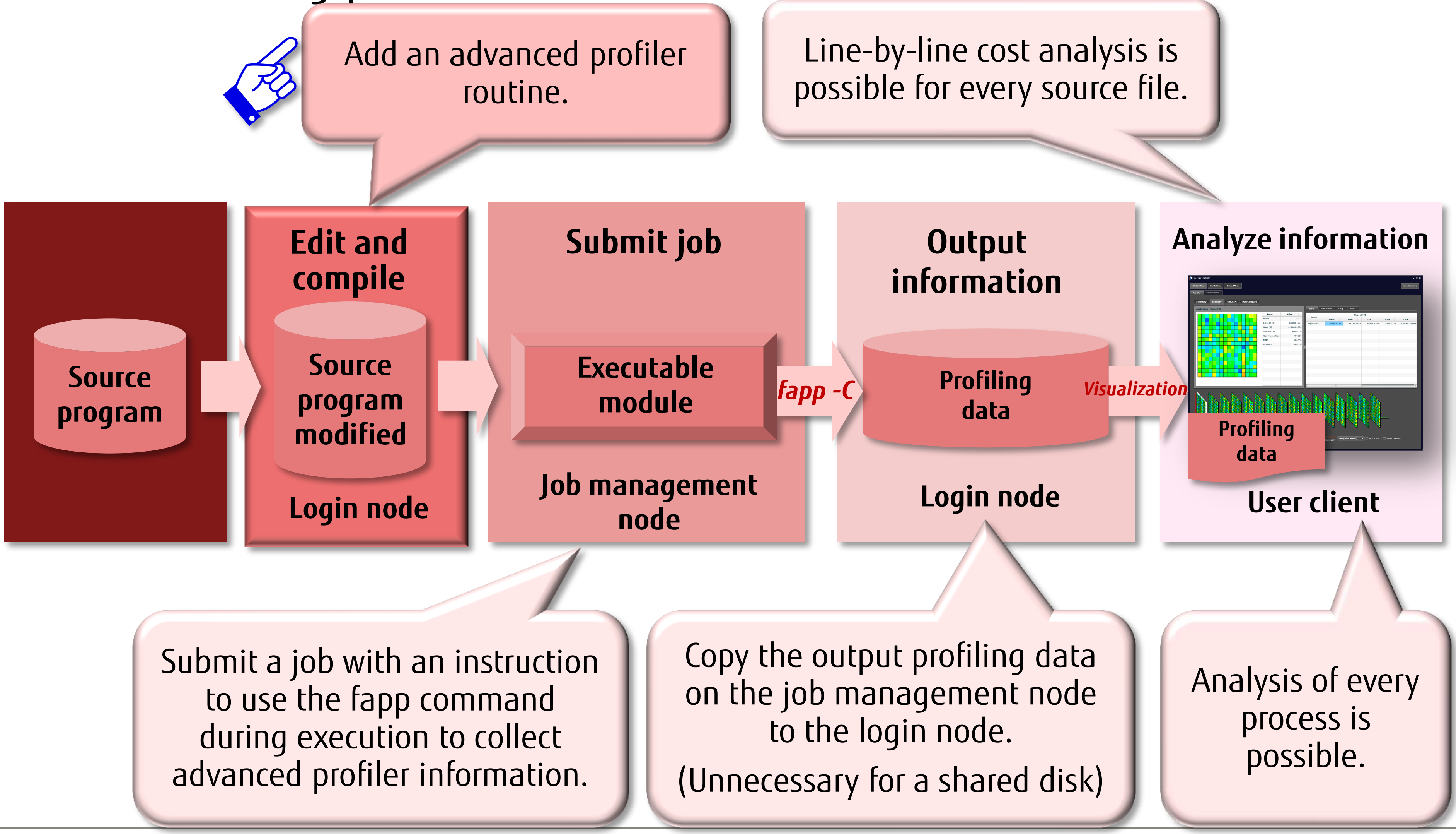
Profiler Overview: Retrievable Information (Instant Profiler)

- The instant profiler can retrieve the following information by:
  - Collecting tuning information from the entire program through sampling
  - Collecting the elapsed times, user CPU times, and system CPU times managed by the OS (Linux)

Classification	Details
Time statistical information	Elapsed time, breakdown of user CPU time and system CPU time, and other information
Cost information	Costs based on sampling in units of procedures, loops, or lines, cost of waiting for inter-thread synchronization, and MPI library communication cost
Hardware monitor information	Processor behavior at application execution time
Call graph information	Procedure call path and cost
Source code information	Each line of source code is output with cost information added

Profiler Overview: Information Retrieval and Analysis Procedure (Advanced Profiler)

- The advanced profiler analyzes the costs of an application by using the following procedure.



Add an advanced profiler routine.

Line-by-line cost analysis is possible for every source file.

Submit a job with an instruction to use the fapp command during execution to collect advanced profiler information.

Copy the output profiling data on the job management node to the login node.

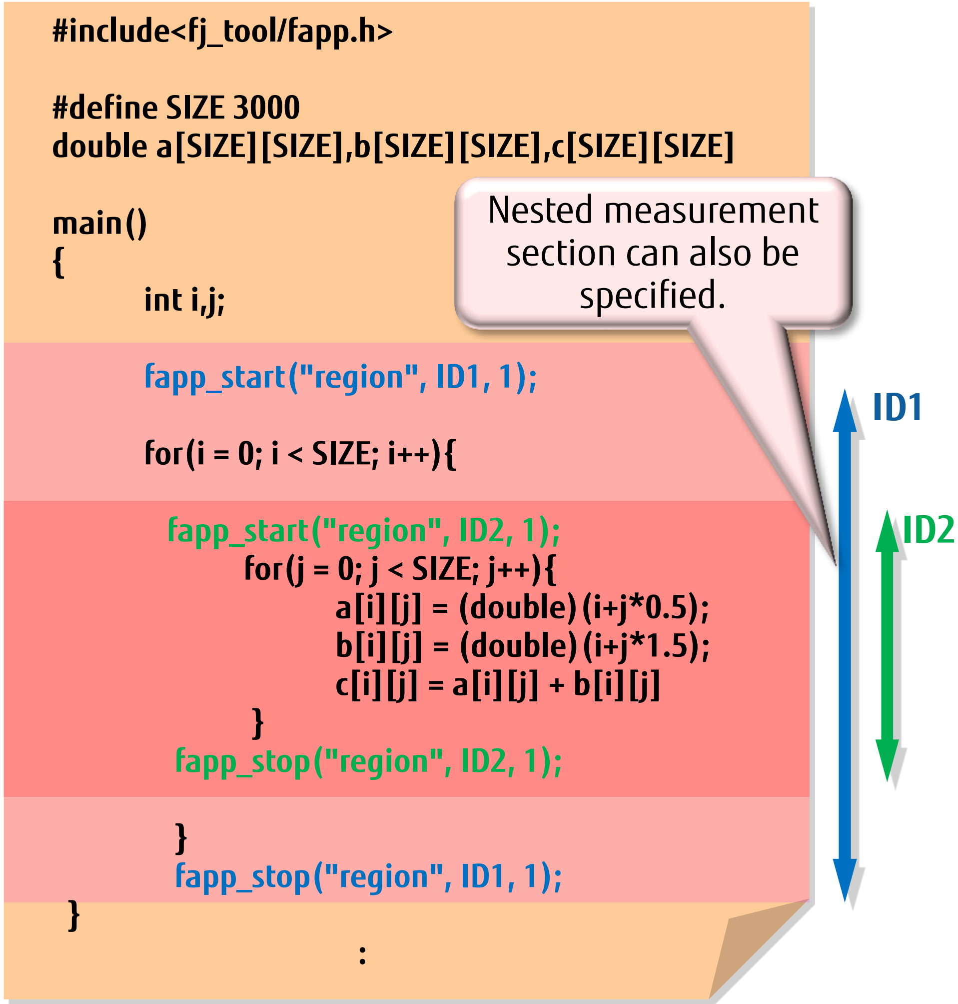
(Unnecessary for a shared disk)

Analysis of every process is possible.

Profiler Overview: Advanced Profiler Use Example

- The advanced profiler can collect execution performance information on the specified section of an application.

Language type	Header file	Function name	Function	Argument
Fortran	None	***fapp_start***	Start information measurement	(name,number, level)
		fapp_stop	Stop information measurement	(name,number, level)
C/C++	fj_tool/fapp.h	***void fapp_start***	Start information measurement	(const char *name, int number, int level)
		void fapp_stop	Stop information measurement	(const char *name, int number, int level)



```
#include<fj_tool/fapp.h>

#define SIZE 3000
double a[SIZE][SIZE],b[SIZE][SIZE],c[SIZE][SIZE]

main()
{
        int i,j;

        fapp_start("region", ID1, 1);

        for(i = 0; i < SIZE; i++){

          fapp_start("region", ID2, 1);
                for(j = 0; j < SIZE; j++){
                        a[i][j] = (double)(i+j*0.5);
                        b[i][j] = (double)(i+j*1.5);
                        c[i][j] = a[i][j] + b[i][j]
                }
          fapp_stop("region", ID2, 1);

          }
          fapp_stop("region", ID1, 1);
 }
                              :
```

Nested measurement section can also be specified.

Profiler Overview: Retrievable Information (Advanced Profiler)

- The advanced profiler can retrieve the following information by:
  - Collecting tuning information from a program measurement section by using a counter
  - Collecting the elapsed times, user CPU times, and system CPU times managed by the OS (Linux)

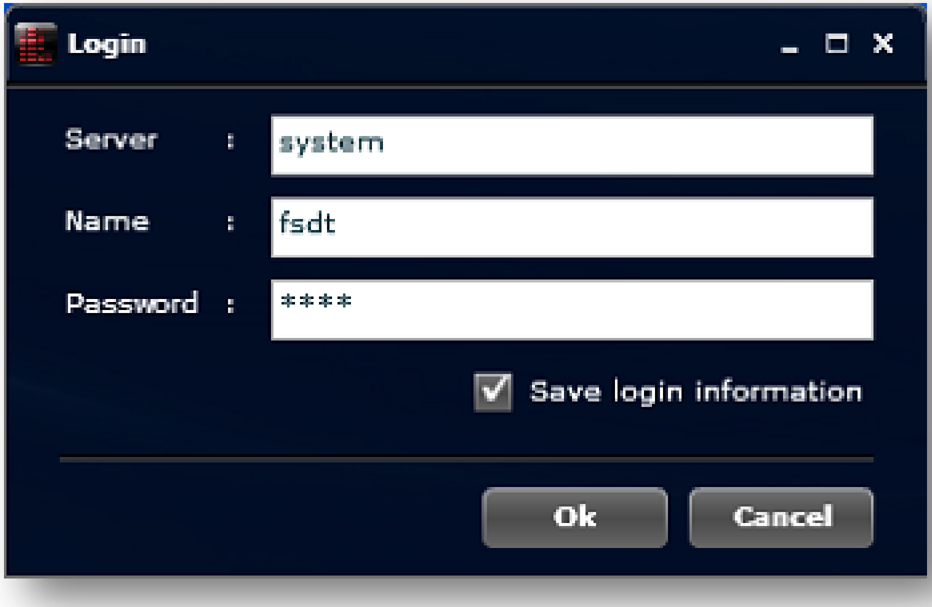
Classification	Details
Basic information	Number of calls, elapsed time, breakdown of user CPU time and system CPU time, and other information in measurement section
MPI information	MPI library execution information in measurement section
Hardware monitor information	Hardware monitor information in measurement section

Profiler Visualization

- Summary View
- Topology View
- Source View
- Call Graph

Profiler Visualization

- First, start the programming support tool to display the main screen (launcher).



Server : system

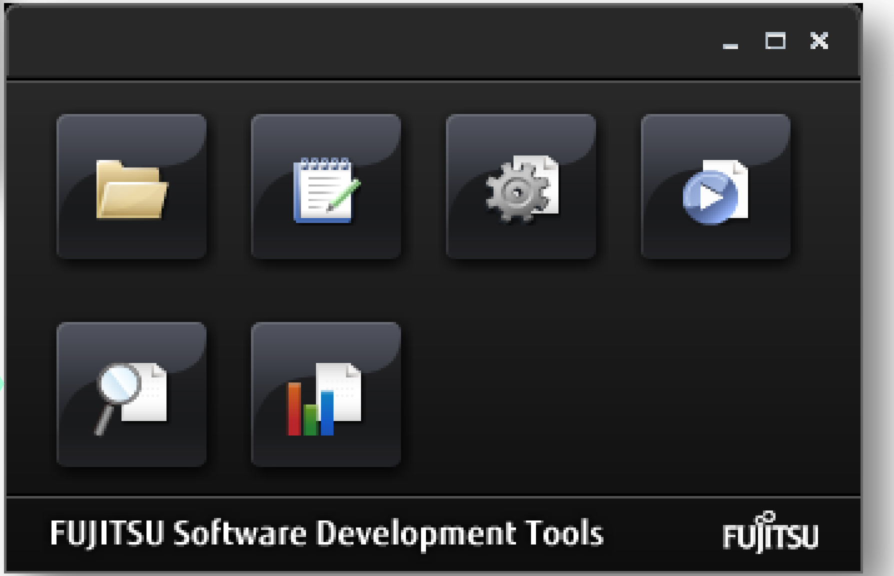
Name : fsdt

Password : ****

☑ Save login information

- Login to a login node
  - Server: Enter the IP address or host name of the login node to which you are logging in.
  - Name: Enter the user name of your login account.
  - Password: Enter the password of your login account.

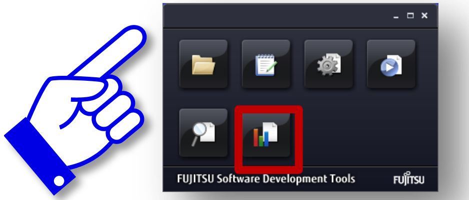
- Various development support functions
  - File operations (file explorer)
  - File editing (editor)
  - Application building (builder)
  - Application execution (executor)
  - Interactive debugger
  - **Profiler visualization**



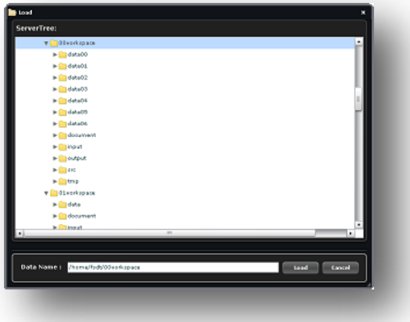
Main screen (launcher)

Profiler Visualization: Starting the Profiler Visibility Function

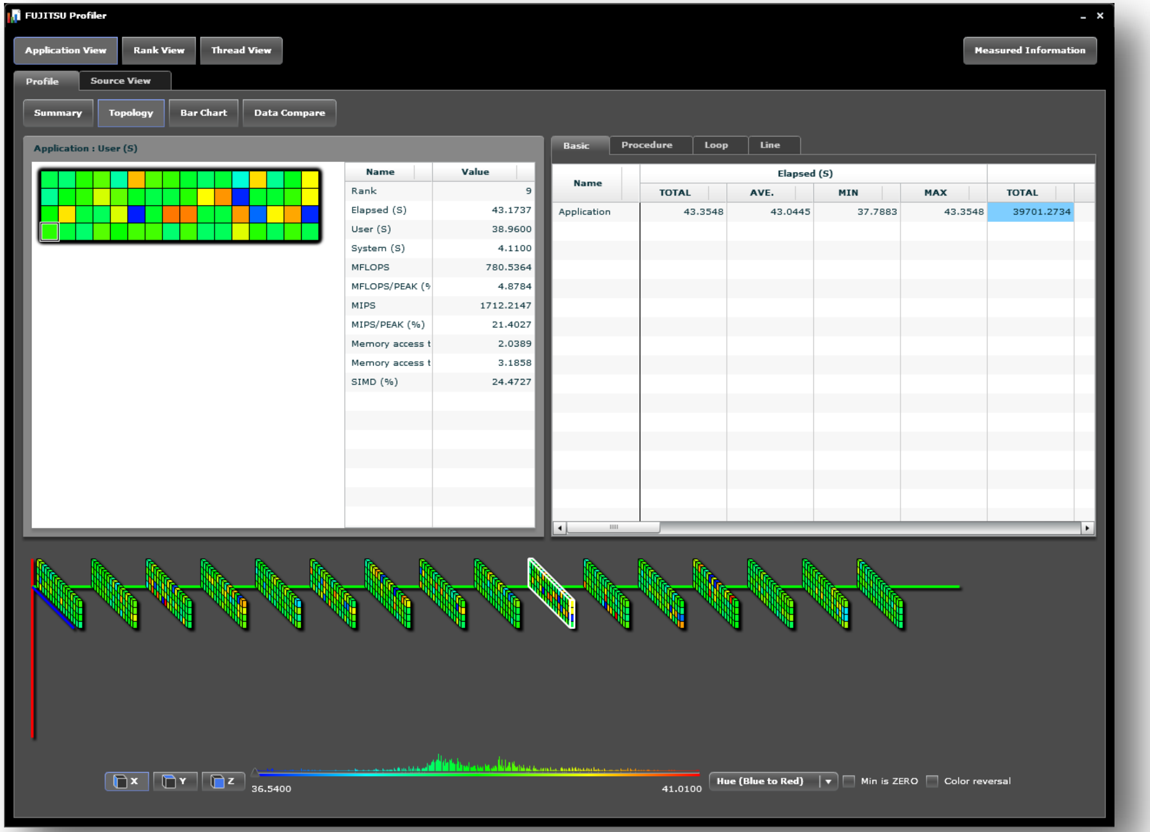
- Start profiler visualization from the main screen.
- After it starts, load profiling data.



Main screen



Load



Name	Value
Rank	9
Elapsed (S)	43.1737
User (S)	38.9600
System (S)	4.1100
MFLOPS	780.5364
MFLOPS/PEAK (%	4.8784
MIPS	1712.2147
MIPS/PEAK (%)	21.4027
Memory access t	2.0389
Memory access t	3.1858
SIMD (%)	24.4727

Name	Elapsed (S)				
	TOTAL	AVE.	MIN	MAX	TOTAL
Application	43.3548	43.0445	37.7883	43.3548	39701.2734

Profiler screen

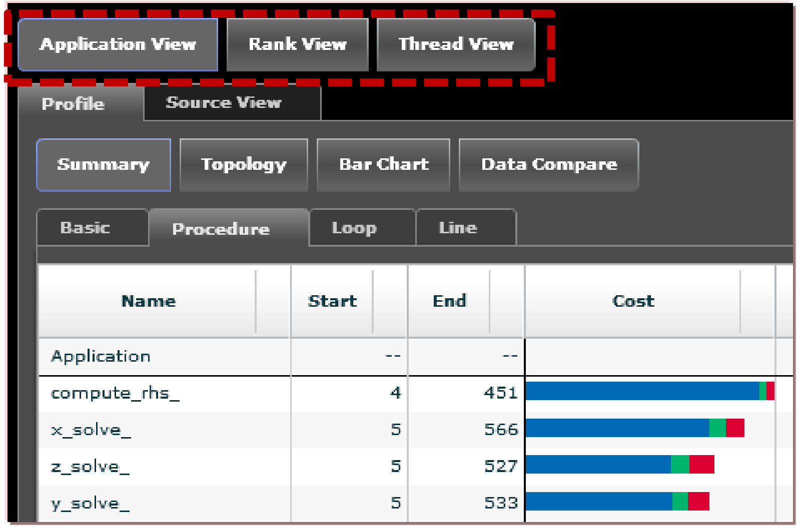
Selecting a GUI View (Mode)

- Select a mode for analyzing profiling data.
  - ☐ **Application View**

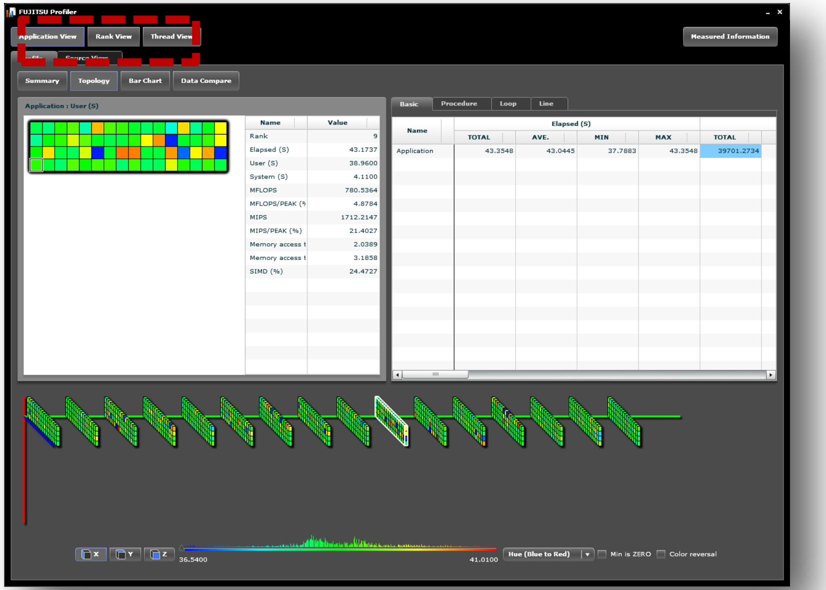
    This view displays information on the retrieved profiling data as a whole.
  - ☐ **Rank View**

    This view displays detailed information focusing on a single rank.
  - ☐ **Thread View**

    This view displays detailed information focusing on a single thread.



Name	Start	End	Cost
Application	--	--	
compute_rhs_	4	451	
x_solve_	5	566	
z_solve_	5	527	
y_solve_	5	533	



Name	Value
Rank	9
Elapsed (S)	43.1737
User (S)	38.9600
System (S)	4.1100
MFLOPS	780.5364
MFLOPS/PEAK (%	4.8784
MIPS	1712.2147
MIPS/PEAK (%)	21.4027
Memory access t	2.0389
Memory access t	3.1858
SIMD (%)	24.4727

Name	Elapsed (S)				
	TOTAL	AVE.	MIN	MAX	TOTAL
Application	43.3548	43.0445	37.7883	43.3548	39701.2734

Profiler Visualization: GUI View Configuration

- The profiler screen provides the following views for each of the application, rank, and thread modes.
- Select the Summary tab, Topology tab, or Source View tab to display the corresponding view.
  - **Summary (summary view)**

    This view displays cost information.

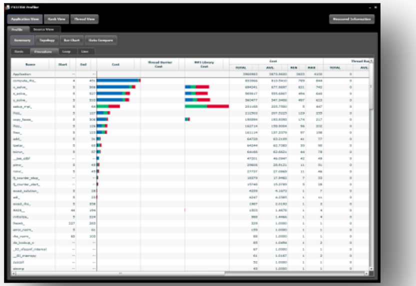
    Use the view to look for hot spots by focusing on the cost of each procedure, loop, and line.
  - **Topology (topology view)**

    This view displays cost information along with the corresponding topology view.

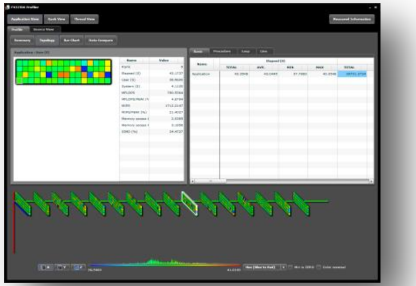
    Use the view to look among all processes (ranks) and threads for those that are hot spots, by focusing on the cost information.
  - **Source View (source view)**

    This view displays cost information on the source code.

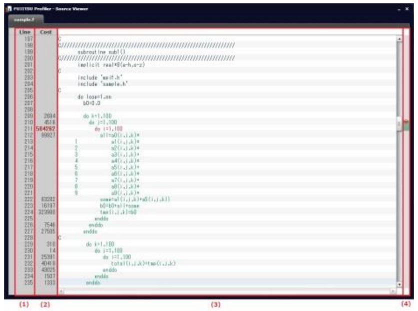
    You can check what processing is performed at hot spots, by focusing mainly on line costs.



Summary view



Topology view



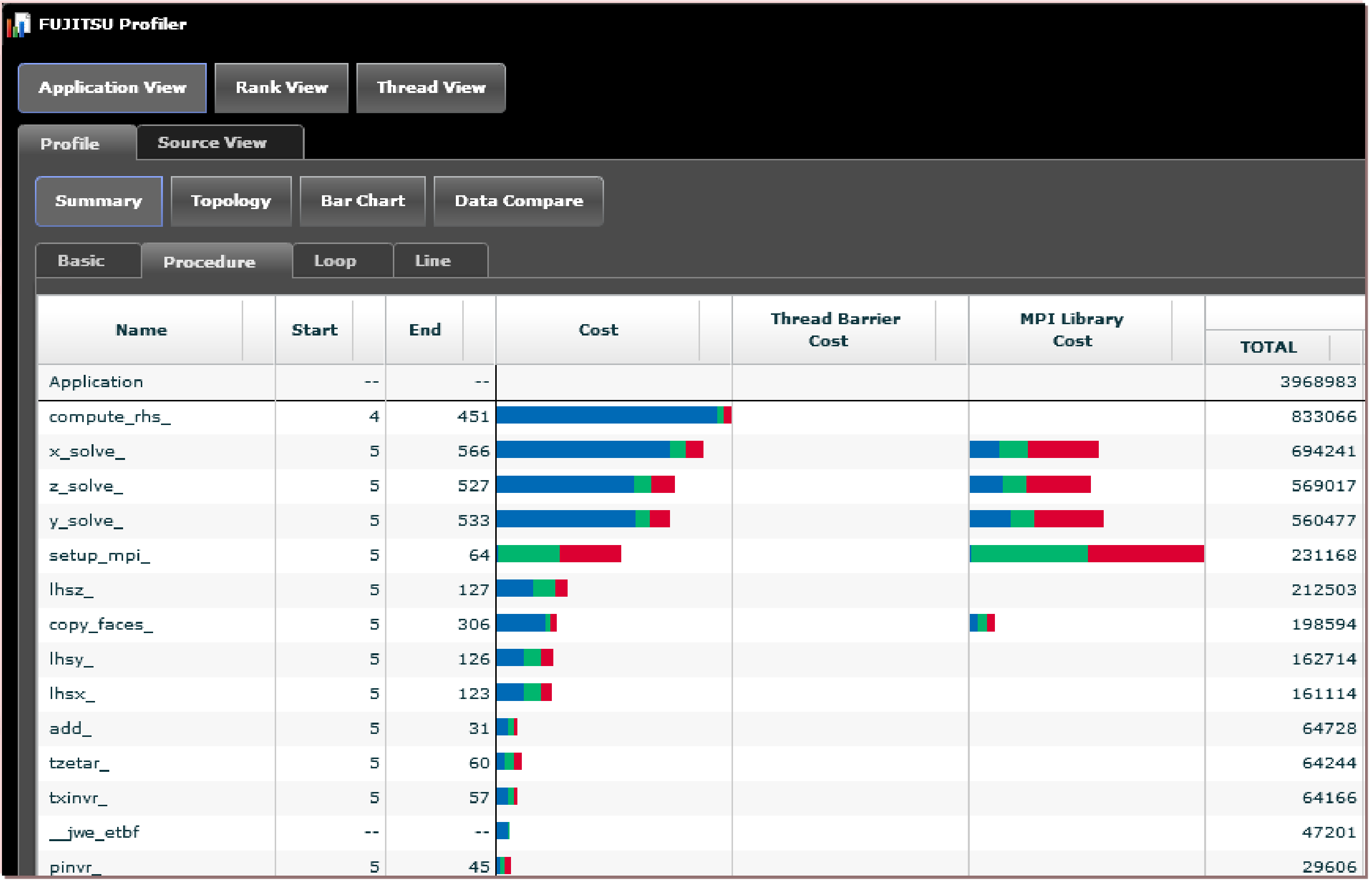
Source view

Summary View

Summary View (1/2)

- This view displays cost information for the entire application.
- You can look by procedure, loop, or line for the places that are hot spots based on the cost information.

Summary view

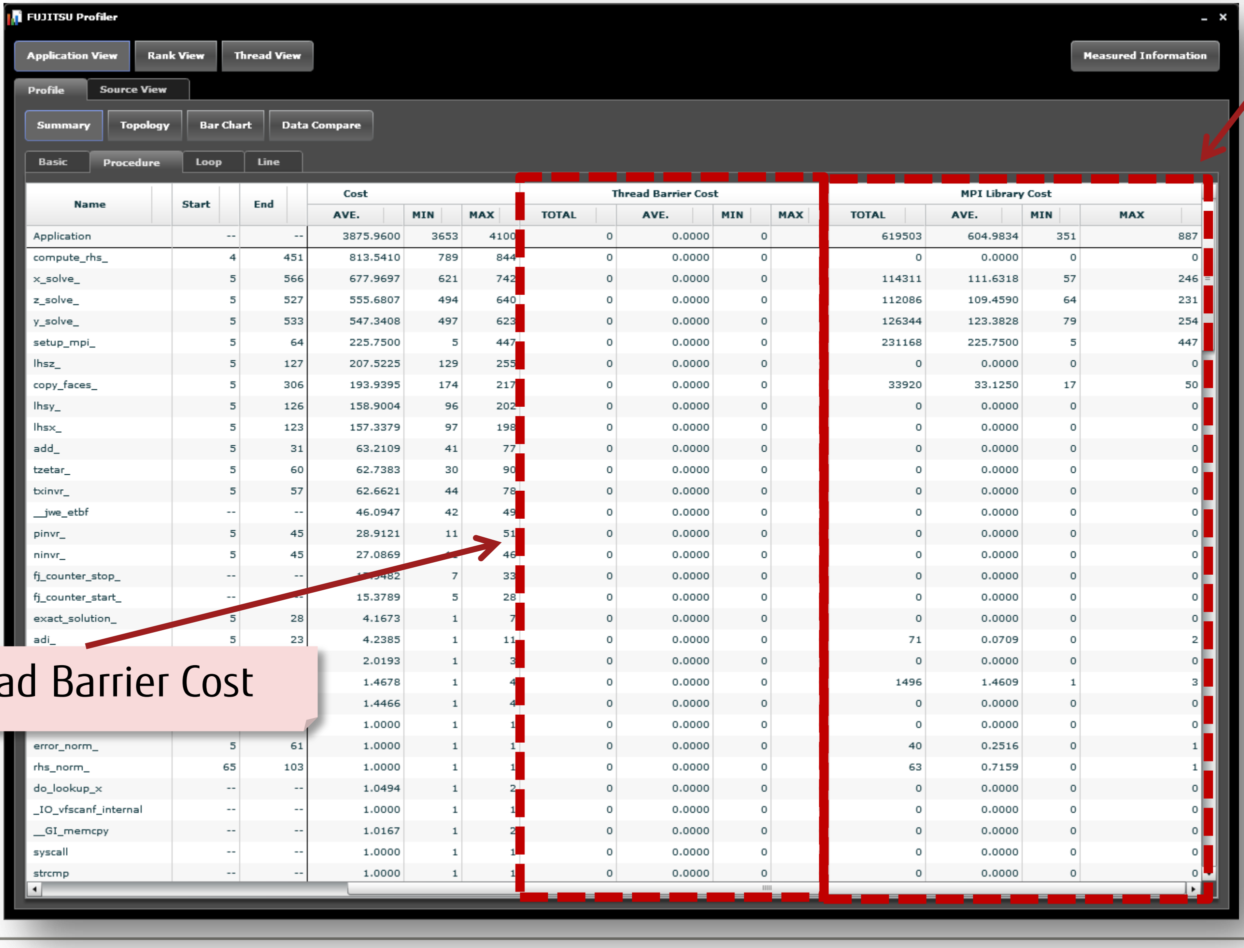


Name	Start	End	Cost	Thread Barrier Cost	MPI Library Cost	TOTAL
Application	--	--				3968983
compute_rhs_	4	451				833066
x_solve_	5	566				694241
z_solve_	5	527				569017
y_solve_	5	533				560477
setup_mpi_	5	64				231168
lhsz_	5	127				212503
copy_faces_	5	306				198594
lhsy_	5	126				162714
lhsx_	5	123				161114
add_	5	31				64728
tzetar_	5	60				64244
txinvr_	5	57				64166
__jwe_etbf	--	--				47201
pinvr_	5	45				29606

Summary View (2/2)

Thread Barrier Cost/MPI Library Cost

- Thread Barrier Cost lists the costs of waiting for inter-thread synchronization.
- MPI Library Cost lists MPI library costs.



MPI Library Cost

Thread Barrier Cost

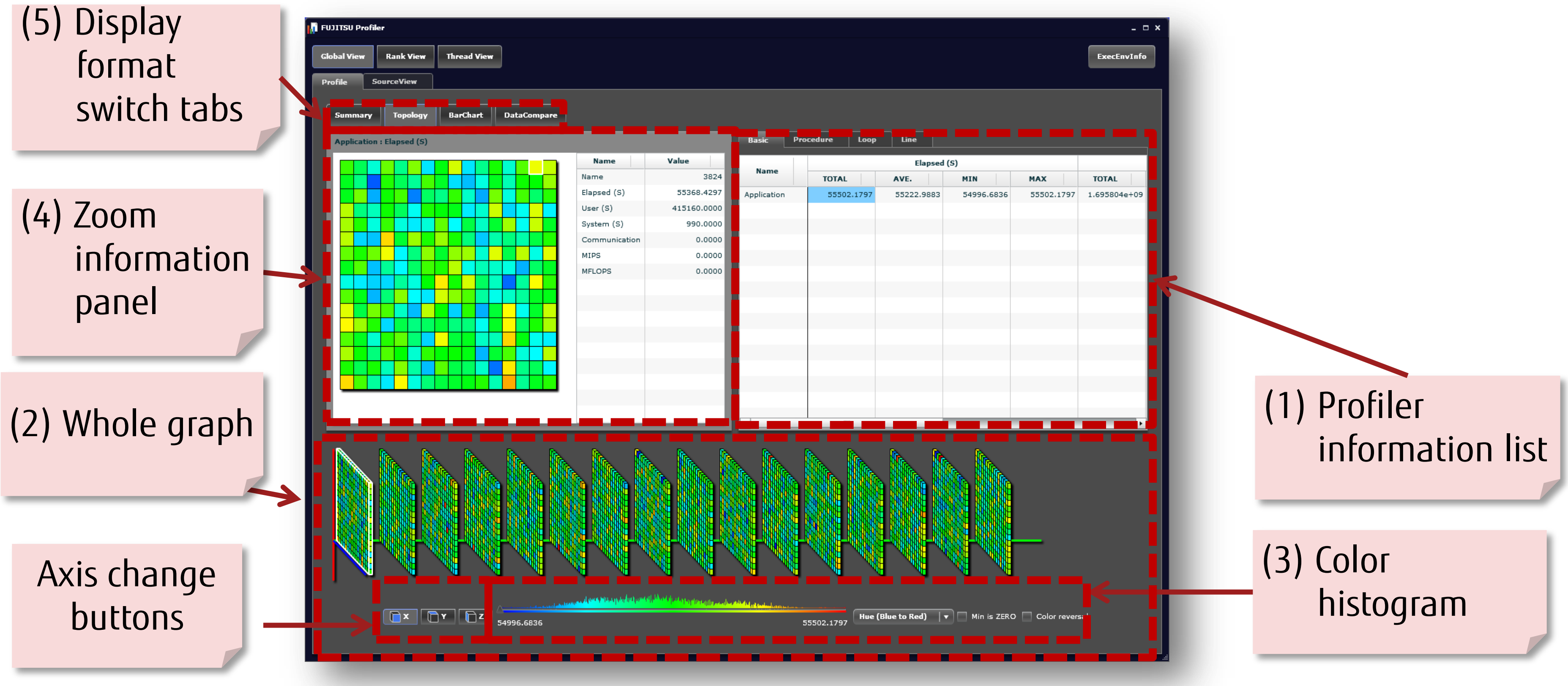
Name	Start	End	Cost			Thread Barrier Cost				MPI Library Cost			
			AVE.	MIN	MAX	TOTAL	AVE.	MIN	MAX	TOTAL	AVE.	MIN	MAX
Application	--	--	3875.9600	3653	4100	0	0.0000	0		619503	604.9834	351	887
compute_rhs_	4	451	813.5410	789	844	0	0.0000	0		0	0.0000	0	0
x_solve_	5	566	677.9697	621	742	0	0.0000	0		114311	111.6318	57	246
z_solve_	5	527	555.6807	494	640	0	0.0000	0		112086	109.4590	64	231
y_solve_	5	533	547.3408	497	623	0	0.0000	0		126344	123.3828	79	254
setup_mpi_	5	64	225.7500	5	447	0	0.0000	0		231168	225.7500	5	447
lhsz_	5	127	207.5225	129	255	0	0.0000	0		0	0.0000	0	0
copy_faces_	5	306	193.9395	174	217	0	0.0000	0		33920	33.1250	17	50
lhsy_	5	126	158.9004	96	202	0	0.0000	0		0	0.0000	0	0
lhsx_	5	123	157.3379	97	198	0	0.0000	0		0	0.0000	0	0
add_	5	31	63.2109	41	77	0	0.0000	0		0	0.0000	0	0
tzetar_	5	60	62.7383	30	90	0	0.0000	0		0	0.0000	0	0
txinvr_	5	57	62.6621	44	78	0	0.0000	0		0	0.0000	0	0
__jwe_etbf	--	--	46.0947	42	49	0	0.0000	0		0	0.0000	0	0
pinvr_	5	45	28.9121	11	51	0	0.0000	0		0	0.0000	0	0
ninvr_	5	45	27.0869	[illegible]	46	0	0.0000	0		0	0.0000	0	0
fj_counter_stop_	--	--	[illegible]	7	33	0	0.0000	0		0	0.0000	0	0
fj_counter_start_	--	--	15.3789	5	28	0	0.0000	0		0	0.0000	0	0
exact_solution_	5	28	4.1673	1	7	0	0.0000	0		0	0.0000	0	0
adi_	5	23	4.2385	1	11	0	0.0000	0		71	0.0709	0	2
			2.0193	1	3	0	0.0000	0		0	0.0000	0	0
			1.4678	1	4	0	0.0000	0		1496	1.4609	1	3
			1.4466	1	4	0	0.0000	0		0	0.0000	0	0
			1.0000	1	1	0	0.0000	0		0	0.0000	0	0
error_norm_	5	61	1.0000	1	1	0	0.0000	0		40	0.2516	0	1
rhs_norm_	65	103	1.0000	1	1	0	0.0000	0		63	0.7159	0	1
do_lookup_x	--	--	1.0494	1	2	0	0.0000	0		0	0.0000	0	0
_IO_vfscanf_internal	--	--	1.0000	1	1	0	0.0000	0		0	0.0000	0	0
__GI_memcpy	--	--	1.0167	1	2	0	0.0000	0		0	0.0000	0	0
syscall	--	--	1.0000	1	1	0	0.0000	0		0	0.0000	0	0
strcmp	--	--	1.0000	1	1	0	0.0000	0		0	0.0000	0	0

Topology View

- (1) Profiler Information List
- (2) Whole graph
- (3) Color Histogram
- (4) Zoom Information Panel
- (5) Display Unit Switching Buttons

Topology View

- This view displays cost information, by process, for an application.
- From the cost information on each process, you can check the information among all processes which have varying times and check the rank of a process that is a hot spot.
- The figure below shows the names of the topology view areas.



Name	Value
Name	3824
Elapsed (S)	55368.4297
User (S)	415160.0000
System (S)	990.0000
Communication	0.0000
MIPS	0.0000
MFLOPS	0.0000

Name	Elapsed (S)				
	TOTAL	AVE.	MIN	MAX	TOTAL
Application	55502.1797	55222.9883	54996.6836	55502.1797	1.695804e+09

(5) Display format switch tabs

(4) Zoom information panel

(2) Whole graph

Axis change buttons

(1) Profiler information list

(3) Color histogram

Topology view

(1) Profiler Information List

Procedure/Loop/Line

- Procedure outputs information in units of user procedures and system functions.
- Loop displays information in units of loop equivalents within a user procedure.
- Line displays information in units of line equivalents within a user procedure.

Procedure

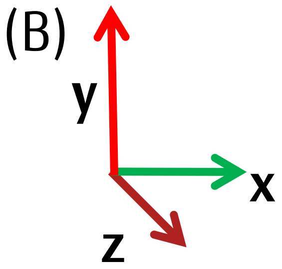
Loop

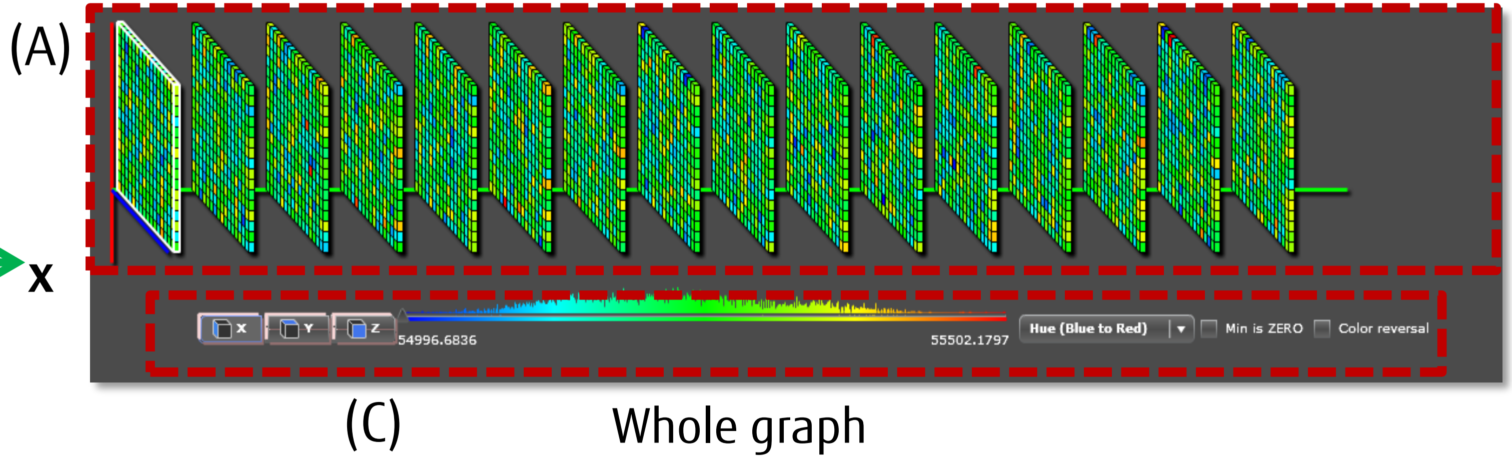
Line

Topology View: (2) Whole Graph (1/2)

- This area displays the (A) information by process within the measurement section selected in the profiler information list.
- The information by process appears in the shape of the topology specified at the job submission time.

  If a 3-dimensional shape is specified, the information is displayed as a 3-dimensional structure following the directions of the axes shown in (B).
- The color of each process in (A) corresponds to a color in the (C) color histogram.

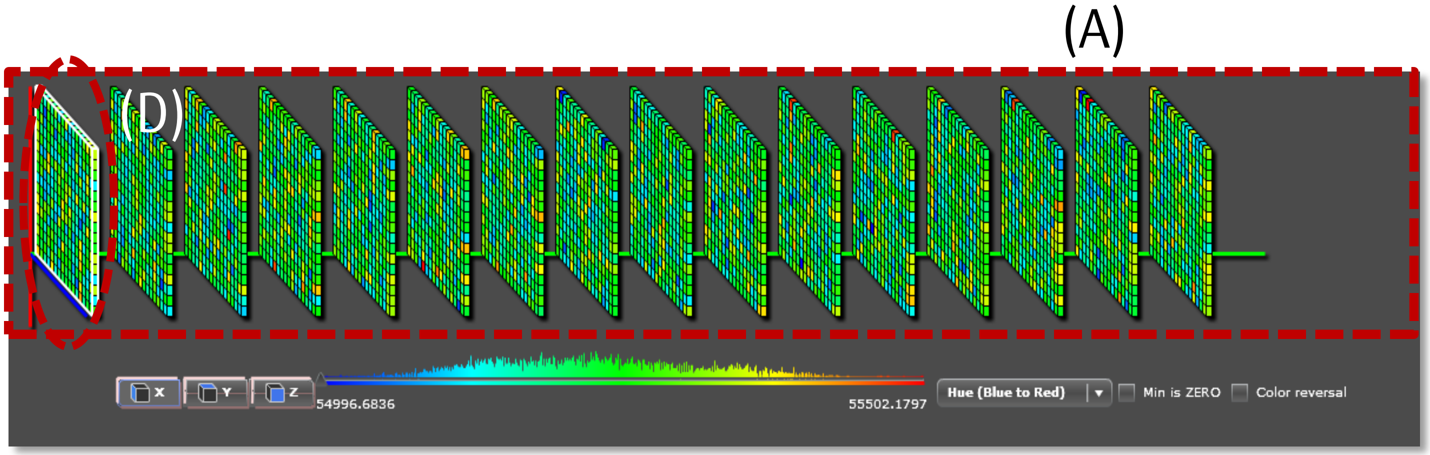




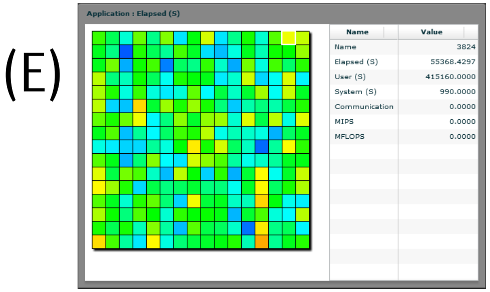
Whole graph

Topology View: (2) Whole Graph (2/2)

- The (D) white frame cursor appears as the mouse pointer is moved over the display area for (A) information by process.
- The (E) zoom information panel (4) zooms in on the process information within the range encompassed by the cursor.



Whole graph



Application : Elapsed (S)

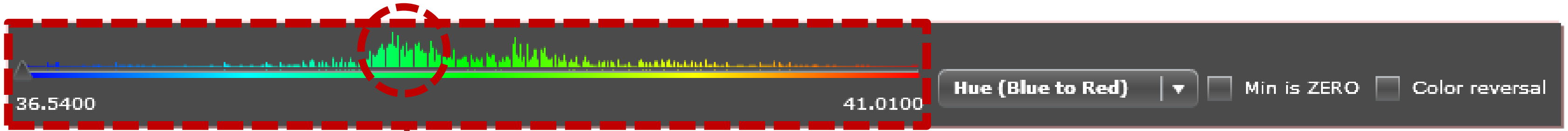
Name	Value
Name	3824
Elapsed (S)	55368.4297
User (S)	415160.0000
System (S)	990.0000
Communication	0.0000
MIPS	0.0000
MFLOPS	0.0000

Zoom information panel

Topology View: (3) Color Histogram

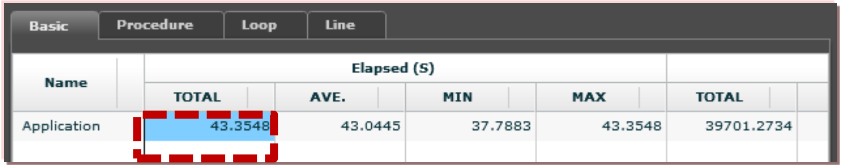
- This area displays the (A) histogram of performance-based distribution based on the process information displayed in the (2) whole graph.
- The horizontal axis represents the numeric values of retrieved information, and a vertical length in the graph represents the frequency of occurrence of the process with that numerical value.

(A)



Color histogram

Example: If you select a user CPU time from the (B) profiler information list, a histogram of the occurrence frequency of the retrieved user CPU time appears. You can see the greater number of processes that have numerical values around the average.



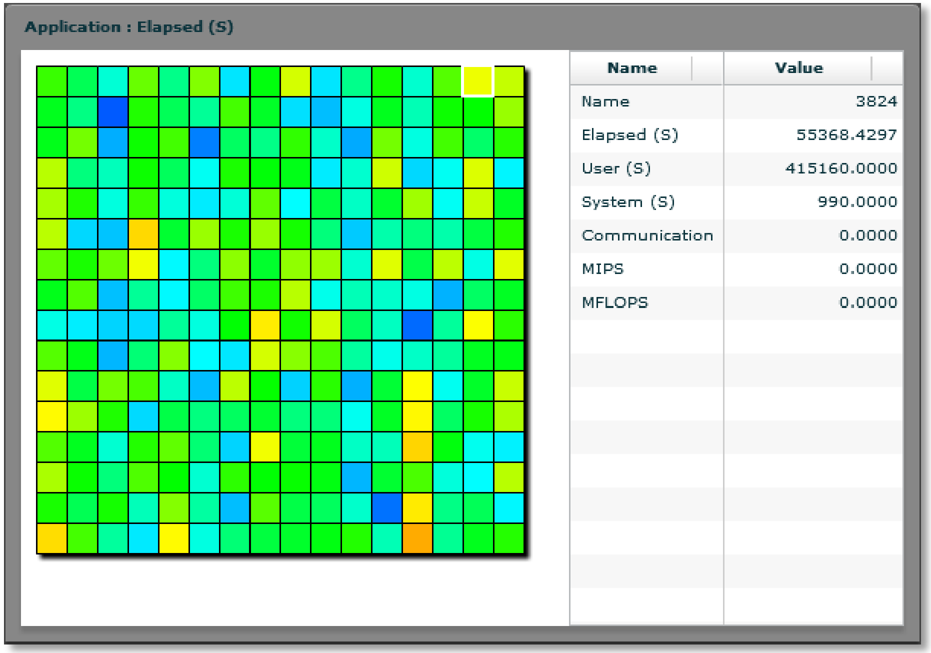
Name	Elapsed (S)				
	TOTAL	AVE.	MIN	MAX	TOTAL
Application	43.3548	43.0445	37.7883	43.3548	39701.2734

(B) Profiler information list

Topology View: (4) Zoom Information Panel

- This area displays the (A) enlarged view of the extracted portion encompassed by the white frame cursor on the (2) whole graph.
- Move the mouse pointer over a process on the (A) enlarged display to select information on the process from the (1) instant profiler information list.

(A)



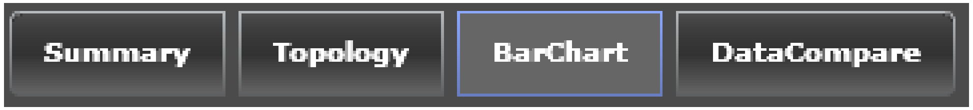
Application : Elapsed (S)

Name	Value
Name	3824
Elapsed (S)	55368.4297
User (S)	415160.0000
System (S)	990.0000
Communication	0.0000
MIPS	0.0000
MFLOPS	0.0000

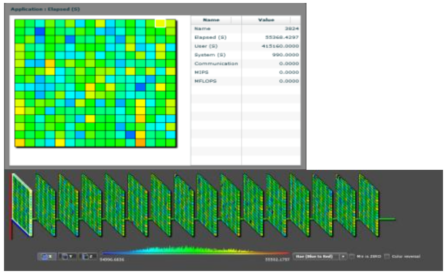
Zoom information panel

Topology View: (5) Display Unit Switching Buttons

- Use these tabs to change the display format of the process displayed in the (2) whole graph and (4) zoom information panel.



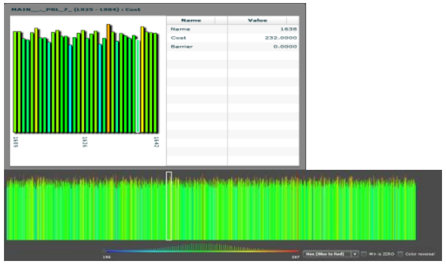
Display unit switching buttons



Topology

- **Topology**

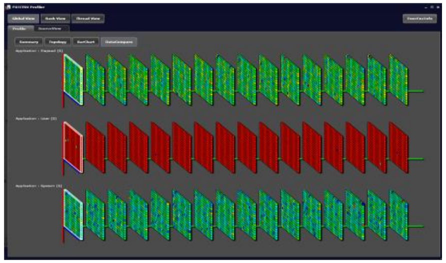
  This tab displays the process in the form of a runtime topology. (Default display)



BarChart

- **BarChart**

  This tab displays a bar chart of performance information on each process.



DataCompare

- **DataCompare**

  This tab arranges the display of data shown in the whole graph.

Source View

- Source Information Screen
- (A) Line Information Display Area
- (B) Source Code Area
- (C) Jump Map

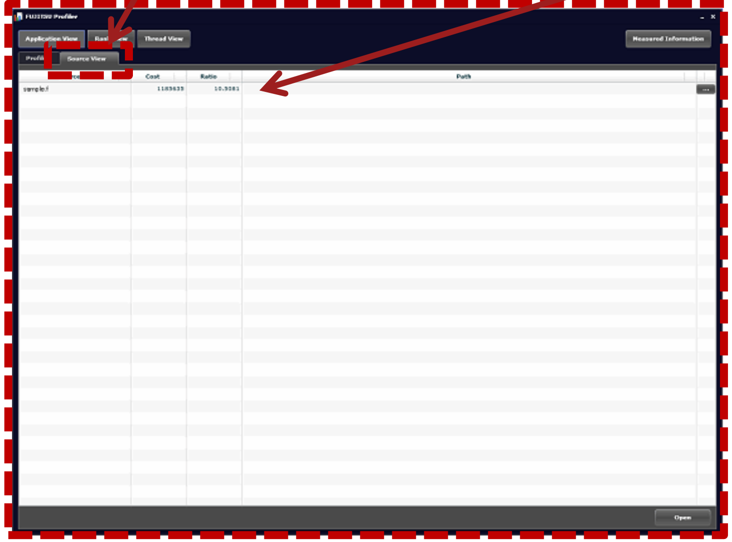
Source View

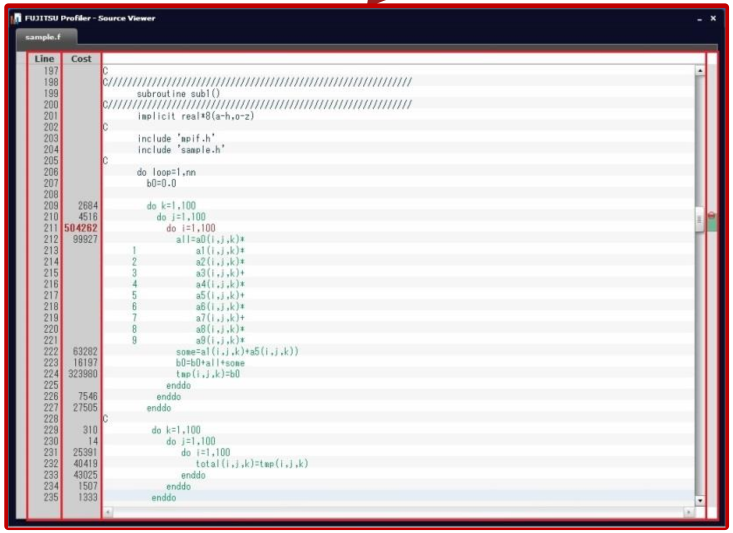
- This view displays cost information for an entire application, on the source code.
- From the cost information displayed on the actual source code, you can check what processing is performed at hot spots.
- The figure below shows the names of the source view areas.

Source View tab

Source file list area

Source information screen

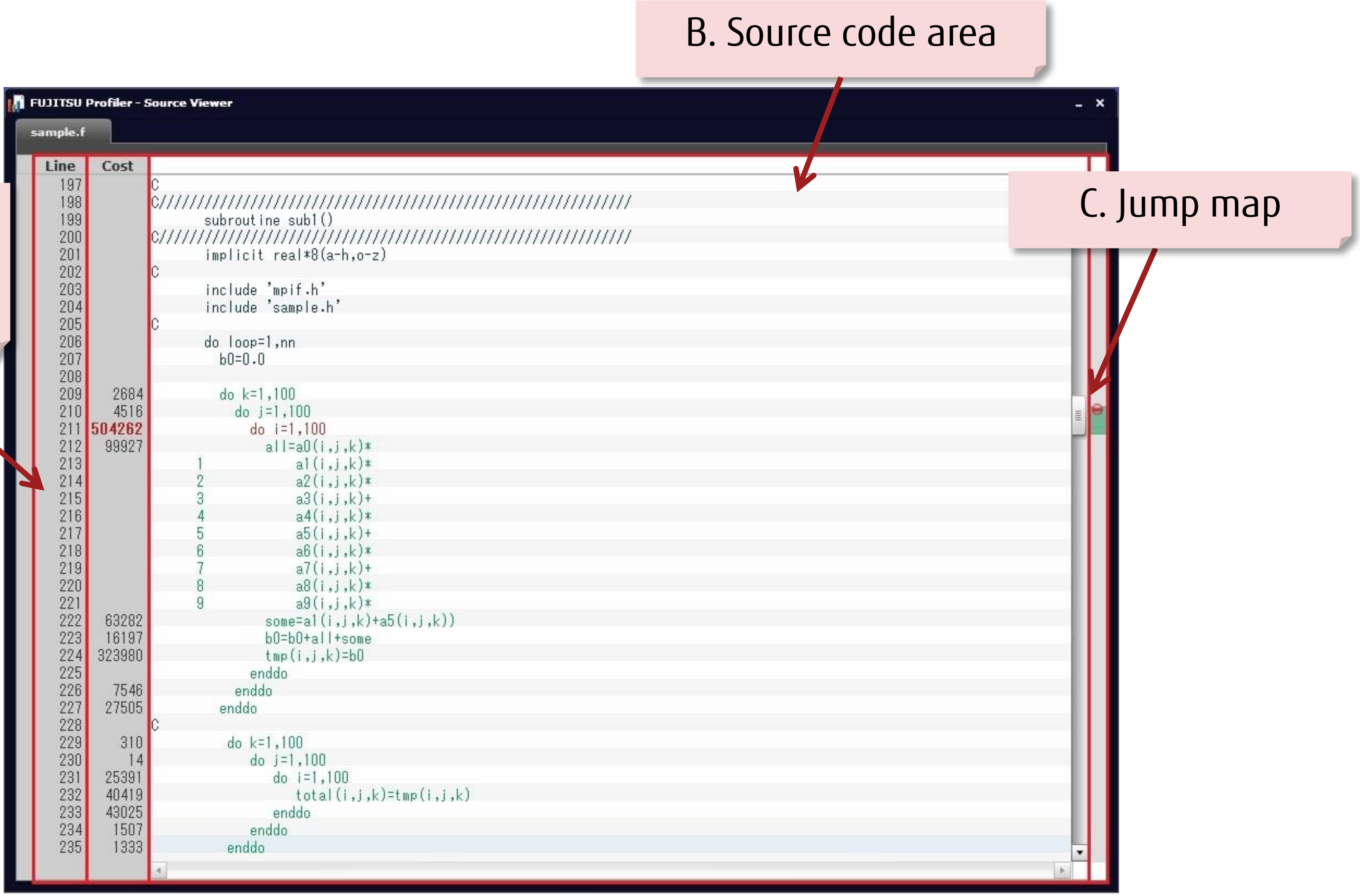




```
Line  Cost
 197         C
 198         C///////////////////////////////////////////////////////////////
 199                subroutine sub1()
 200         C///////////////////////////////////////////////////////////////
 201                implicit real*8(a-h,o-z)
 202         C
 203                include 'mpif.h'
 204                include 'sample.h'
 205         C
 206                do loop=1,nn
 207                  b0=0.0
 208
 209    2684          do k=1,100
 210    4516            do j=1,100
 211  504262              do i=1,100
 212   99927                all=a0(i,j,k)*
 213               1            a1(i,j,k)*
 214               2            a2(i,j,k)*
 215               3            a3(i,j,k)+
 216               4            a4(i,j,k)*
 217               5            a5(i,j,k)+
 218               6            a6(i,j,k)*
 219               7            a7(i,j,k)+
 220               8            a8(i,j,k)*
 221               9            a9(i,j,k)*
 222   63282                some=a1(i,j,k)+a5(i,j,k))
 223   16197                b0=b0+all+some
 224  323980                tmp(i,j,k)=b0
 225                      enddo
 226    7546            enddo
 227   27505          enddo
 228         C
 229     310           do k=1,100
 230      14              do j=1,100
 231   25391                 do i=1,100
 232   40419                    total(i,j,k)=tmp(i,j,k)
 233   43025                 enddo
 234    1507              enddo
 235    1333           enddo
```

Source View: Source Information Screen

- The source information screen consists of the following components.



B. Source code area

A. Line information display area

C. Jump map

```
Line   Cost
 197        C
 198        C///////////////////////////////////////////////////////////////
 199              subroutine sub1()
 200        C///////////////////////////////////////////////////////////////
 201              implicit real*8(a-h,o-z)
 202        C
 203              include 'mpif.h'
 204              include 'sample.h'
 205        C
 206              do loop=1,nn
 207                b0=0.0
 208
 209    2684        do k=1,100
 210    4516          do j=1,100
 211  504262            do i=1,100
 212   99927              all=a0(i,j,k)*
 213             1            a1(i,j,k)*
 214             2            a2(i,j,k)*
 215             3            a3(i,j,k)+
 216             4            a4(i,j,k)*
 217             5            a5(i,j,k)+
 218             6            a6(i,j,k)*
 219             7            a7(i,j,k)+
 220             8            a8(i,j,k)*
 221             9            a9(i,j,k)*
 222   63282              some=a1(i,j,k)+a5(i,j,k))
 223   16197              b0=b0+all+some
 224  323980              tmp(i,j,k)=b0
 225                    enddo
 226    7546          enddo
 227   27505        enddo
 228        C
 229     310         do k=1,100
 230      14            do j=1,100
 231   25391               do i=1,100
 232   40419                  total(i,j,k)=tmp(i,j,k)
 233   43025               enddo
 234    1507            enddo
 235    1333         enddo
```

Source View: (A) Line Information Display Area

- Select a source file name on the Source View tab to display the source information screen.
- This area displays source code line information and related information for the respective number of lines.
  - ☐ (1) Line number bar

    This bar displays line numbers.
  - ☐ (2) Line cost display bar

    This bar displays the costs of lines.

    Significantly high display costs appear in red.

(1) (2)

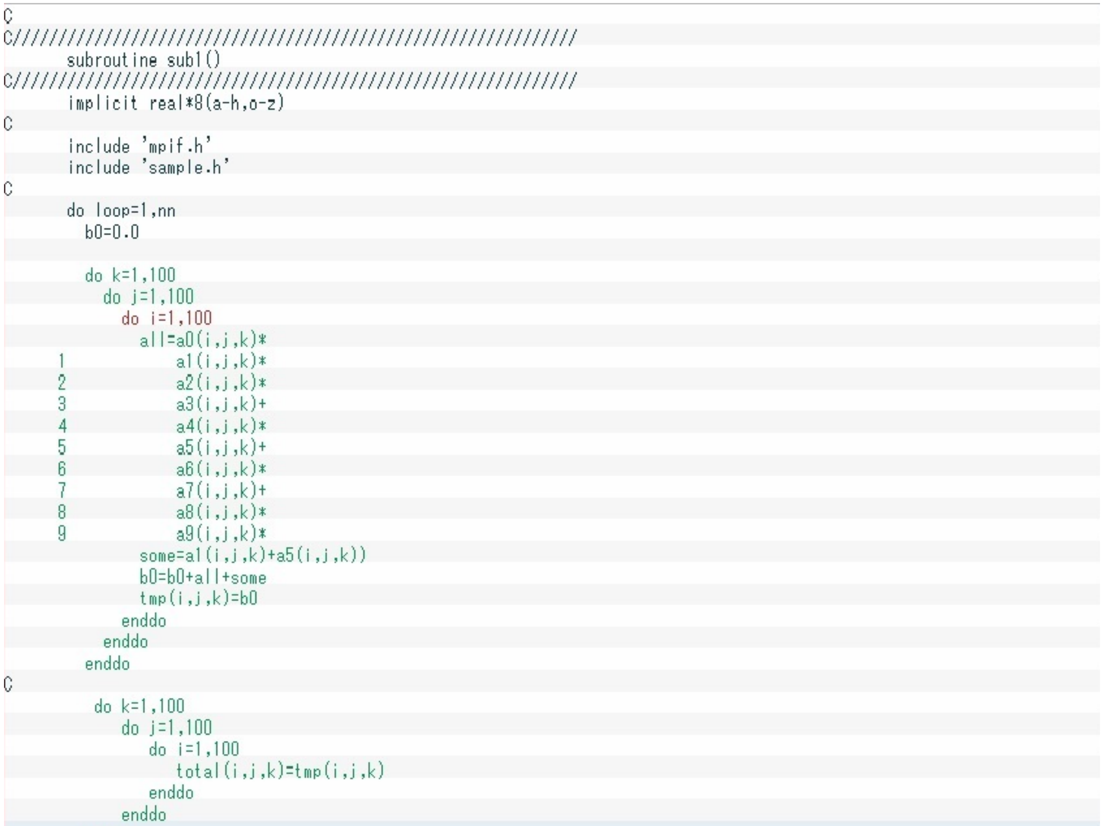
210	4516
211	504262
212	99927
213	

Line information display area

Source View: (B) Source Code Area

- This area displays the target source code.
- Thread parallelization parts appear in green, and high-cost parts appear in red.

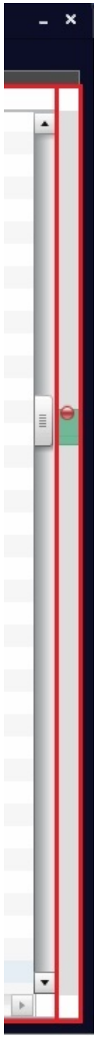
Source code area



```
C
C///////////////////////////////////////////////////////////////
       subroutine sub1()
C///////////////////////////////////////////////////////////////
       implicit real*8(a-h,o-z)
C
       include 'mpif.h'
       include 'sample.h'
C
       do loop=1,nn
         b0=0.0

         do k=1,100
           do j=1,100
             do i=1,100
               all=a0(i,j,k)*
      1            a1(i,j,k)*
      2            a2(i,j,k)*
      3            a3(i,j,k)+
      4            a4(i,j,k)*
      5            a5(i,j,k)+
      6            a6(i,j,k)*
      7            a7(i,j,k)+
      8            a8(i,j,k)*
      9            a9(i,j,k)*
               some=a1(i,j,k)+a5(i,j,k))
               b0=b0+all+some
               tmp(i,j,k)=b0
             enddo
           enddo
         enddo
C
          do k=1,100
             do j=1,100
                do i=1,100
                   total(i,j,k)=tmp(i,j,k)
                enddo
             enddo
```

Source View: (C) Jump Map

- This map shows the colored parts (high-cost places and thread parallelization places) of the (A) and (B) areas.
- You can jump to the intended line in the text area by clicking in the jump map.

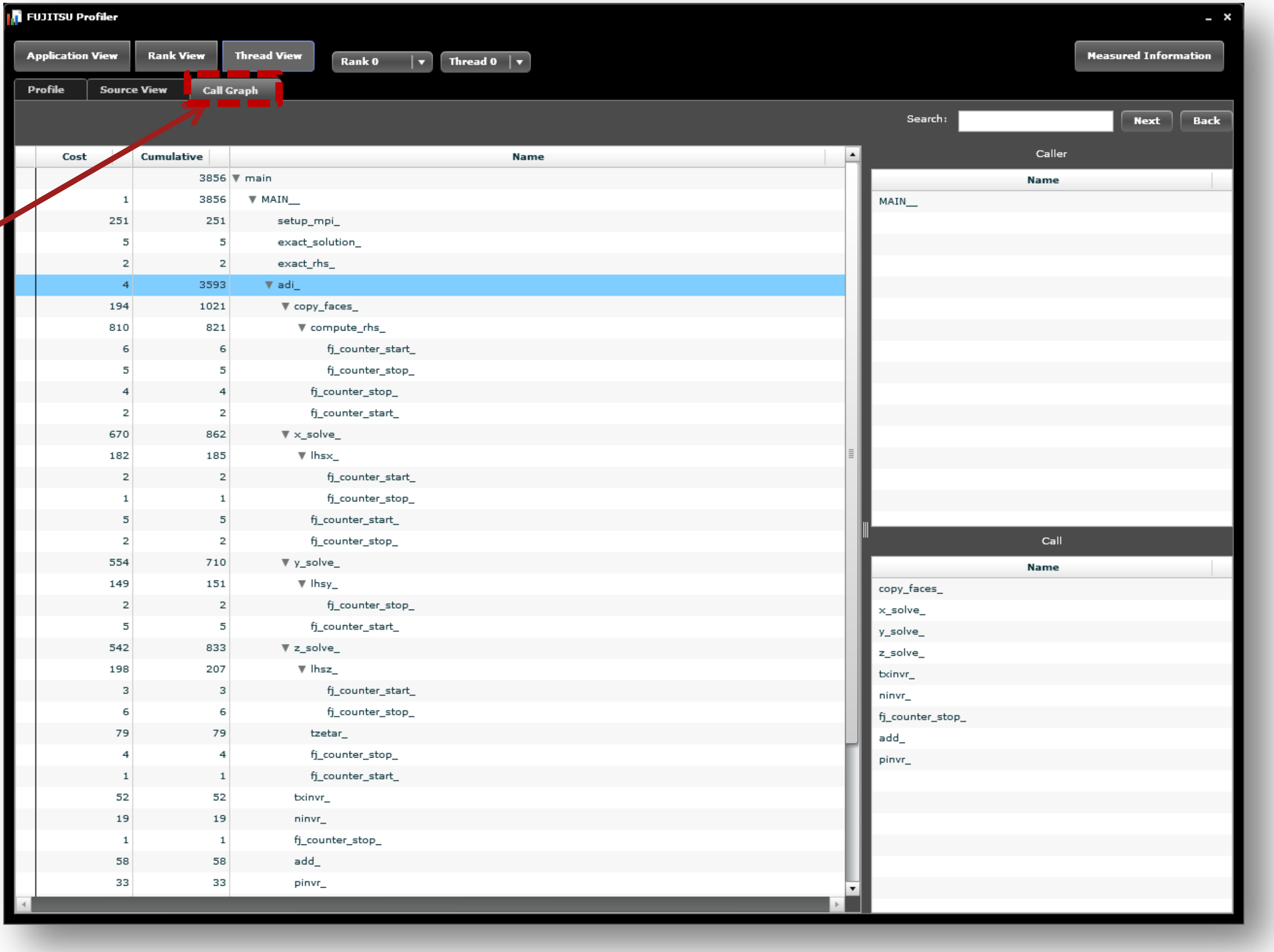


Jump map

Call Graph

Call Graph

- Thread View has the Call Graph tab, which displays the procedure call relationship in the form of a tree.



Cost	Cumulative	Name
	3856	▼ main
1	3856	▼ MAIN__
251	251	setup_mpi_
5	5	exact_solution_
2	2	exact_rhs_
4	3593	▼ adi_
194	1021	▼ copy_faces_
810	821	▼ compute_rhs_
6	6	fj_counter_start_
5	5	fj_counter_stop_
4	4	fj_counter_stop_
2	2	fj_counter_start_
670	862	▼ x_solve_
182	185	▼ lhsx_
2	2	fj_counter_start_
1	1	fj_counter_stop_
5	5	fj_counter_start_
2	2	fj_counter_stop_
554	710	▼ y_solve_
149	151	▼ lhsy_
2	2	fj_counter_stop_
5	5	fj_counter_start_
542	833	▼ z_solve_
198	207	▼ lhsz_
3	3	fj_counter_start_
6	6	fj_counter_stop_
79	79	tzetar_
4	4	fj_counter_stop_
1	1	fj_counter_start_
52	52	txinvr_
19	19	ninvr_
1	1	fj_counter_stop_
58	58	add_
33	33	pinvr_

Caller

Name
MAIN__

Call

Name
copy_faces_
x_solve_
y_solve_
z_solve_
txinvr_
ninvr_
fj_counter_stop_
add_
pinvr_

Call Graph tab

Profiler Use Examples

- General Use Examples
- PA Information Example
- MPI Information Example

Profiler Use Examples

- This section describes the options specified in the following cases.
  - General use examples:
    - Checking high-cost parts (high-cost parts at up to N places)
    - Checking the load balance
    - Checking the cost distribution that includes a call relationship
  - PA information example:
    - Obtaining MFLOPS values and memory throughput values
  - MPI information example:
    - Obtaining the ratio of MPI functions used across an entire program

The subsequent pages describe information collection command examples and visualization examples.

✓ To use the profilers, specify the required options to output information from profiler visualization based on collected profiling data.

General Use Examples

- Checking High-cost Parts
- Checking the Load Balance
- Checking Cost Distribution
- Accuracy of sampling (fipp)

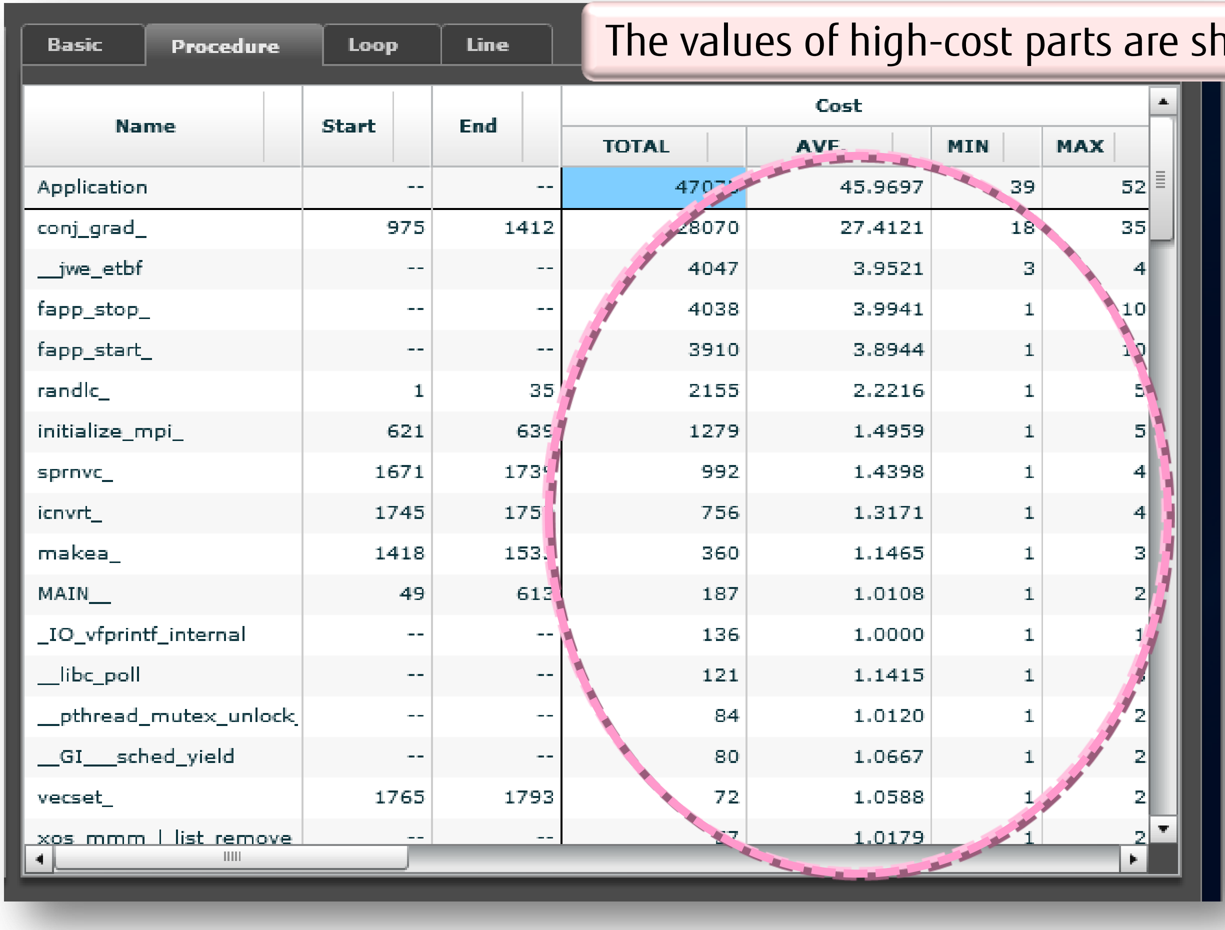
General Use Examples: Checking High-cost Parts

■Checking high-cost parts

Information collection command example

```
fipp -C -d data a.out
```

Visualization example



Name	Start	End	Cost			
			TOTAL	AVE.	MIN	MAX
Application	--	--	4707[illegible]	45.9697	39	52
conj_grad_	975	1412	28070	27.4121	18	35
__jwe_etbf	--	--	4047	3.9521	3	4
fapp_stop_	--	--	4038	3.9941	1	10
fapp_start_	--	--	3910	3.8944	1	10
randlc_	1	35	2155	2.2216	1	5
initialize_mpi_	621	639	1279	1.4959	1	5
sprnvc_	1671	1739	992	1.4398	1	4
icnvrt_	1745	175[illegible]	756	1.3171	1	4
makea_	1418	153[illegible]	360	1.1465	1	3
MAIN__	49	613	187	1.0108	1	2
_IO_vfprintf_internal	--	--	136	1.0000	1	1
__libc_poll	--	--	121	1.1415	1	[illegible]
__pthread_mutex_unlock	--	--	84	1.0120	1	2
__GI___sched_yield	--	--	80	1.0667	1	2
vecset_	1765	1793	72	1.0588	1	2
xos_mmm_l_list_remove	--	--	77	1.0179	1	2

The values of high-cost parts are shown here.

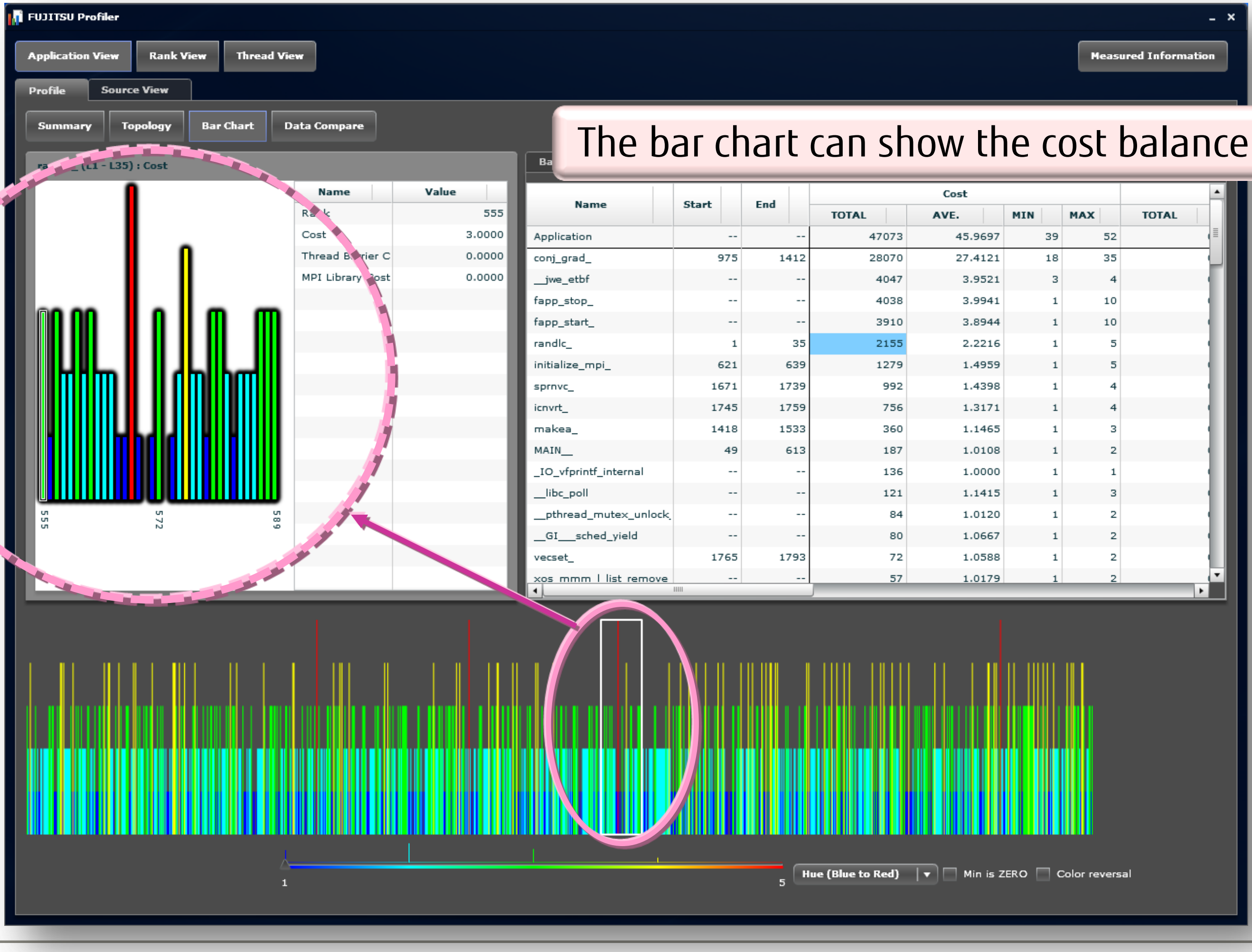
General Use Examples: Checking the Load Balance

■Checking the load balance

Information collection command example

```
fipp -C -d data a.out
```

Visualization example



The bar chart can show the cost balance.

Name	Value
Ra k	555
Cost	3.0000
Thread B rier C	0.0000
MPI Library ost	0.0000

Name	Start	End	Cost				
			TOTAL	AVE.	MIN	MAX	TOTAL
Application	--	--	47073	45.9697	39	52	
conj_grad_	975	1412	28070	27.4121	18	35	
__jwe_etbf	--	--	4047	3.9521	3	4	
fapp_stop_	--	--	4038	3.9941	1	10	
fapp_start_	--	--	3910	3.8944	1	10	
randlc_	1	35	2155	2.2216	1	5	
initialize_mpi_	621	639	1279	1.4959	1	5	
sprnvc_	1671	1739	992	1.4398	1	4	
icnvrt_	1745	1759	756	1.3171	1	4	
makea_	1418	1533	360	1.1465	1	3	
MAIN__	49	613	187	1.0108	1	2	
_IO_vfprintf_internal	--	--	136	1.0000	1	1	
__libc_poll	--	--	121	1.1415	1	3	
__pthread_mutex_unlock_	--	--	84	1.0120	1	2	
__GI___sched_yield	--	--	80	1.0667	1	2	
vecset_	1765	1793	72	1.0588	1	2	
xos_mmm_l_list_remove	--	--	57	1.0179	1	2	

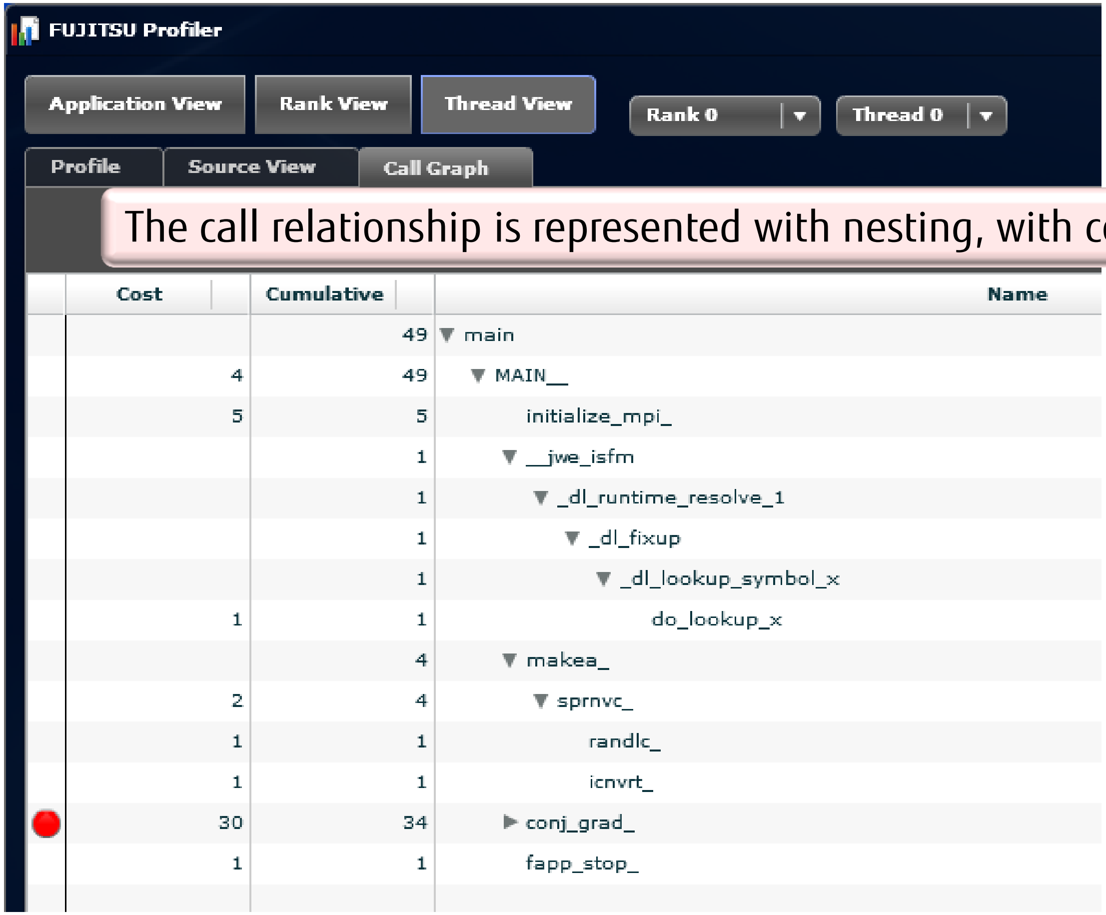
General Use Examples: Checking Cost Distribution

■Checking cost distribution with a call relationship

Information collection command example

fipp -C -Icall -d data a.out

Visualization example



The call relationship is represented with nesting, with costs displayed by call path.

	Cost	Cumulative	Name
		49	▼ main
	4	49	▼ MAIN__
	5	5	initialize_mpi_
		1	▼ __jwe_isfm
		1	▼ _dl_runtime_resolve_1
		1	▼ _dl_fixup
		1	▼ _dl_lookup_symbol_x
	1	1	do_lookup_x
		4	▼ makea_
	2	4	▼ sprnvc_
	1	1	randlc_
	1	1	icnvrt_
●	30	34	▶ conj_grad_
	1	1	fapp_stop_

Accuracy of sampling (fipp)

- Following phenomena may occur if the execution time of 1 invocation of a procedure is less than sampling interval time(the execution times for one occurrence are 150 microseconds) that can be specified by fipp.
  - Time measured by timer routine may differ from time measured by fipp.
  - A load imnalance may occur between threads, or barrier time may increase.(Those are different from results by precision PA)

In case that the execution time of 1 invocation of a procedure is less than sampling interval time that can be specified by fipp, correct data may not be obtained by fipp.
(In such case, please use precision PA)

PA Information Example

PA Information Example: MFLOPS Values and Memory Throughput Values

- Checking MFLOPS values (operation performance) and memory throughput values (memory access performance)

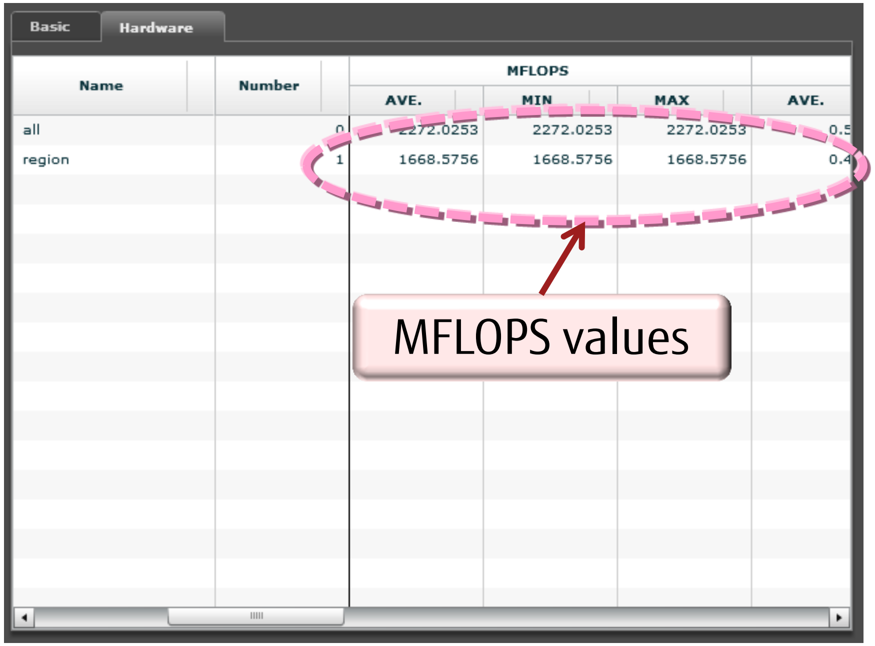
Information collection command example

MFLOPS: **fipp -Ihwm -d data -C a.out**

Memory throughput:

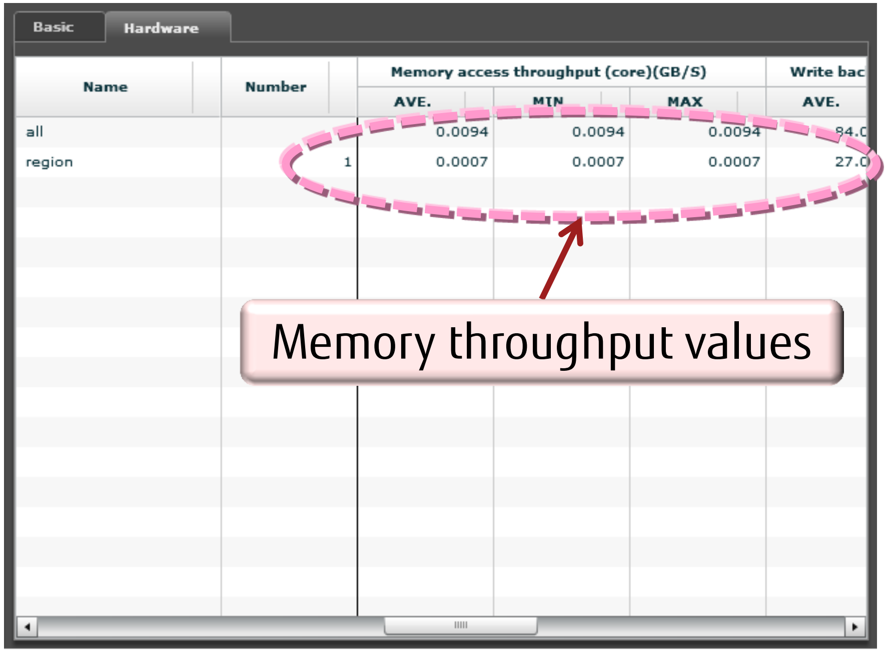
fipp -Ihwm -Hevent=MEM_access -d data -C a.out

Visualization example



Name	Number	MFLOPS			
		AVE.	MIN	MAX	AVE.
all	0	2272.0253	2272.0253	2272.0253	0.5
region	1	1668.5756	1668.5756	1668.5756	0.4

MFLOPS values



Name	Number	Memory access throughput (core)(GB/S)			Write bac
		AVE.	MIN	MAX	AVE.
all		0.0094	0.0094	0.0094	84.0
region	1	0.0007	0.0007	0.0007	27.0

Memory throughput values

MPI Information Example

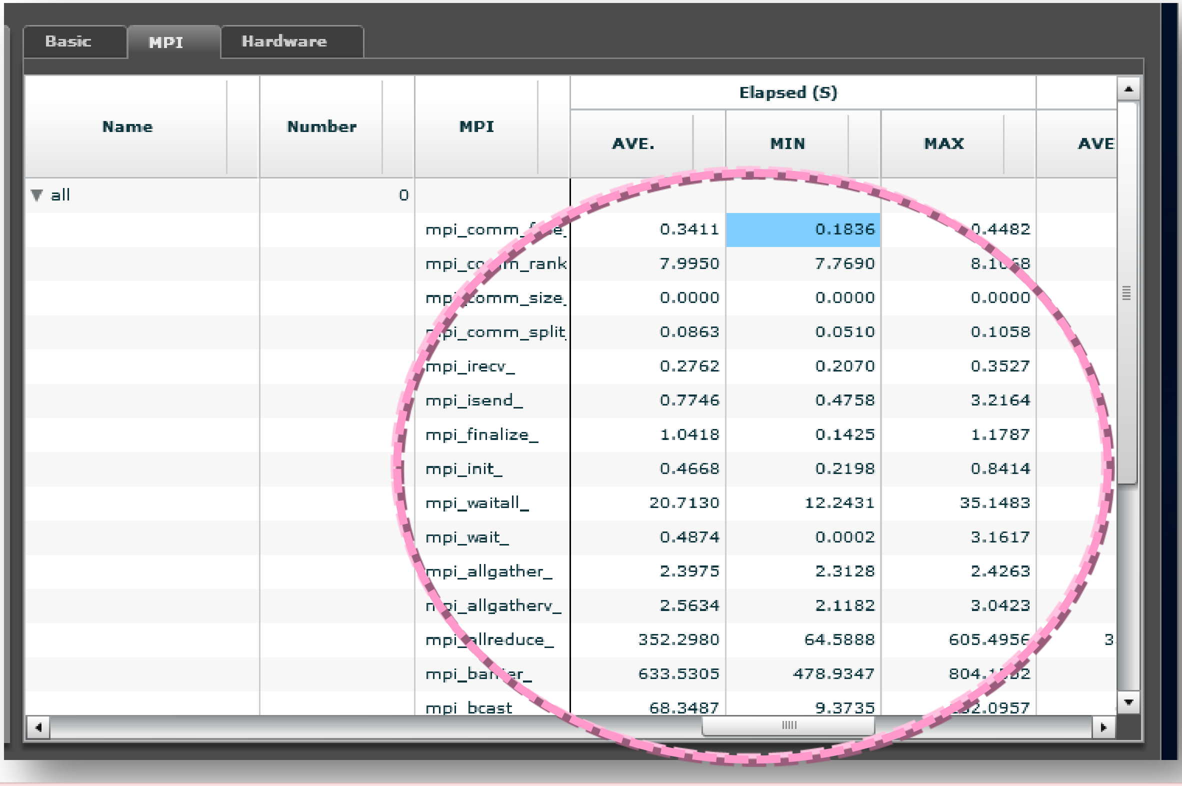
MPI Information Example: Ratio of MPI Functions

- Checking the ratio of MPI functions used across an entire program

Information collection command example

fapp -Impi -d data -C mpiexec a.out

Visualization example



Name	Number	MPI	Elapsed (S)			
			AVE.	MIN	MAX	AVE
▼ all	0					
		mpi_comm_free_	0.3411	0.1836	0.4482	
		mpi_comm_rank	7.9950	7.7690	8.1[illegible]58	
		mpi_comm_size_	0.0000	0.0000	0.0000	
		mpi_comm_split	0.0863	0.0510	0.1058	
		mpi_irecv_	0.2762	0.2070	0.3527	
		mpi_isend_	0.7746	0.4758	3.2164	
		mpi_finalize_	1.0418	0.1425	1.1787	
		mpi_init_	0.4668	0.2198	0.8414	
		mpi_waitall_	20.7130	12.2431	35.1483	
		mpi_wait_	0.4874	0.0002	3.1617	
		mpi_allgather_	2.3975	2.3128	2.4263	
		mpi_allgatherv_	2.5634	2.1182	3.0423	
		mpi_allreduce_	352.2980	64.5888	605.4956	3
		mpi_barrier_	633.5305	478.9347	804.1[illegible]2	
		mpi_bcast	68.3487	9.3735	[illegible]2.0957	

The total values (elapsed time and number of MPI calls) and average message length across all processes are displayed for each MPI function.

Tuning Examples

- Tuning Procedure
- Sequential Tuning
- MPI Tuning

Tuning Examples: Tuning Procedure

- This section describes the tuning procedure using fipp/fapp.

1. Identifying a high-cost loop

- Collecting information with the instant profiler (fipp)
- Text output of visualization and basic information
- GUI output of visualization and basic information

2. Detailed analysis of a high-cost loop

- Inserting the fapp information collection routines into a source file
- Collecting information with the advanced profiler (fapp)
- Text output of visualization and detailed information
- GUI output of visualization and detailed information

3. Tuning work

- Source tuning, option tuning, and optimization control line tuning

4. Verifying tuning results

- Collecting information and checking results with the advanced profiler (fapp)

The subsequent pages describe examples of <sequential tuning> and <MPI tuning>.

Sequential Tuning

- Identifying a High-cost Loop
- Detailed Analysis of a High-cost Loop
- Tuning Work
- Verifying Tuning Results

Sequential Tuning: Identifying a High-cost Loop

<Sequential tuning example>

Collecting information with the instant profiler (fipp)

Information collection command example

```
fipp -C –Ihwm -d prof_fipp  a.out
```

- *-C* : Gives an instruction to collect instant profiling data.
- *-Ihwm* : Gives an instruction to collect hardware monitor information.
  (By default, cost information is collected.)
- *-d* : Specifies the instant profiling data name
  (name of the directory storing instant profiling data files).

Sequential Tuning: Identifying a High-cost Loop

FUJITSU

Text output of visualization and basic information

Visualization command example

This is executed on a login node.

```
fipppx -A –Icpu,hwm -d prof_fipp –o cost.txt
```

- *-A* : Gives an instruction to output instant profiler information.
- *-Icpu,hwm* : Gives an instruction to output cost information and hardware monitor information.
- *-d* : Specifies the instant profiling data name (name of the directory storing instant profiling data files).
- *-o* : Gives an instruction on the output destination of instant profiler information.

- The default of *-I* is *-Icpu*. *cpu* means to output cost information.

Sequential Tuning: Identifying a High-cost Loop

Text visualization example

Loops profile

```
 ****************************************************************************
   Thread      0 - loops
 ****************************************************************************

  Cost        %  Operation (S)   Barrier       %   Nest   Kind     Exec  Start  End
 --------------------------------------------------------------------------------
    85   100.0000        8.6466       19    22.3529   --     --     --       --  --  Thread    0
 --------------------------------------------------------------------------------
    55    64.7059        5.5949        0     0.0000    1   DO       AUTO      72   75   sub3._PRL_1_
     5     5.8824        0.5086        5   100.0000    1   DO       SERIAL    34   37   init_
     5     5.8824        0.5086        5   100.0000   --   ARRAY    SERIAL    32   32   init_
     5     5.8824        0.5086        0     0.0000    1   DO       AUTO      57   61   sub2._PRL_1_
     4     4.7059        0.4069        4   100.0000    1   DO       SERIAL    57   61   sub2_
     3     3.5294        0.3052        0     0.0000    1   DO       AUTO      45   49   sub1._PRL_1_
     3     3.5294        0.3052        3   100.0000    1   DO       SERIAL    72   76   sub3_
     2     2.3529        0.2034        2   100.0000    1   DO       SERIAL    45   49   sub1_
```

High cost

Low cost

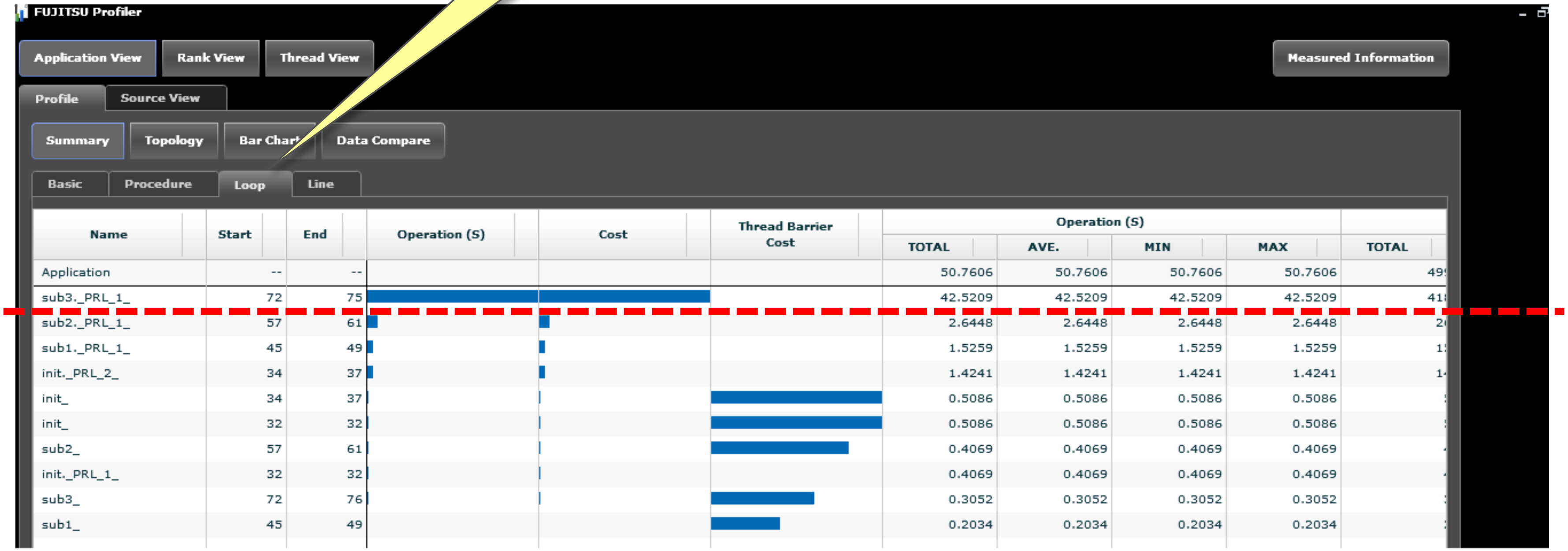
- You can see that the loop from line number 72 to 75 has a high cost of about 64.7%.
- Next, collect data with the advanced profiler to investigate the high-cost loop in detail.

Sequential Tuning: Identifying a High-cost Loop

GUI output of visualization and basic information (high-cost loop)

GUI visualization example

Loop cost



Name	Start	End	Operation (S)	Cost	Thread Barrier Cost	Operation (S)				
						TOTAL	AVE.	MIN	MAX	TOTAL
Application	--	--				50.7606	50.7606	50.7606	50.7606	49
sub3._PRL_1_	72	75				42.5209	42.5209	42.5209	42.5209	41
sub2._PRL_1_	57	61				2.6448	2.6448	2.6448	2.6448	2
sub1._PRL_1_	45	49				1.5259	1.5259	1.5259	1.5259	1
init._PRL_2_	34	37				1.4241	1.4241	1.4241	1.4241	1
init_	34	37				0.5086	0.5086	0.5086	0.5086	
init_	32	32				0.5086	0.5086	0.5086	0.5086	
sub2_	57	61				0.4069	0.4069	0.4069	0.4069	
init._PRL_1_	32	32				0.4069	0.4069	0.4069	0.4069	
sub3_	72	76				0.3052	0.3052	0.3052	0.3052	
sub1_	45	49				0.2034	0.2034	0.2034	0.2034	

- You can see that the loop from line number 72 to 75 has a high cost.
- Next, collect data with the advanced profiler to investigate the high-cost loop in detail.

Sequential Tuning: Detailed Analysis of a High-cost Loop

- Inserting the fapp information collection routines into a source file
  - Insert the advanced profiler routines (fapp_start and fapp_stop) before and after the high-cost part (line number 72 to 75) in the source file.

```
70                call fapp_start("region", 1, 0)
71
             <<< Loop-information Start >>>
             <<<  [PARALLELIZATION]
             <<<    Standard iteration count: 422
             <<<  [OPTIMIZATION]
             <<<    COLLAPSED
             <<<    SIMD(VL: 4)
             <<<    SOFTWARE PIPELINING
             <<< Loop-information  End >>>
72   1  pp   6v     do j = 1 , m
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    COLLAPSED
             <<< Loop-information  End >>>
73   2   p   6        do i = 1 , n
74   2   p   6v          a(i, j) = b(i, j) + c(i, j) + d(i, j)+e(i,j)+f(i,j)+g(i,j)+h(i,j)
75   2   p   6v        enddo
76   1   p            enddo
77
78                call fapp_stop("region", 1, 0)
```

High-cost loop subject to tuning

Sequential Tuning: Detailed Analysis of a High-cost Loop

Collecting information with the advanced profiler (fapp)

Information collection command example

Cache miss rate

```
fapp -C –Ihwm –Hevent=Cache -d prof_fapp  a.out
```

- *-C* : Gives an instruction to collect advanced profiling data.
- *-Ihwm* : Gives an instruction to collect hardware monitor information.
- *-Hevent=Cache* : Gives an instruction to collect cache miss ratios.

  For this event, change the specification depending on the purpose of collection. (Multiple specifications are not allowed.)

  (MEM_access: Memory access status, Instructions _SIMD: Execution instruction details (SIMD), Instructions_NOSIMD: Execution instruction details (NOSIMD), Performance: Instruction execution efficiency, Statistics: CPU core activity status, TLB: TLB miss rate)
- *-d* : Specifies the advanced profiling data name (name of the directory storing advanced profiling data files).

Sequential Tuning: Detailed Analysis of a High-cost Loop

Text output of visualization and detailed information

Visualization command example

Cache miss rate

This is executed on a login node.

```
fapppx -A –Ihwm -d prof_fapp –o mem.txt
```

- *-A* : Gives an instruction to output advanced profiling data.
- *-Ihwm* : Gives an instruction to output hardware monitor information.
- *-d* : Specifies the advanced profiling data name.
- *-o* : Gives an instruction on the output destination of advanced profiler information.

Sequential Tuning: Detailed Analysis of a High-cost Loop

Text visualization example

Performance monitor : Cache

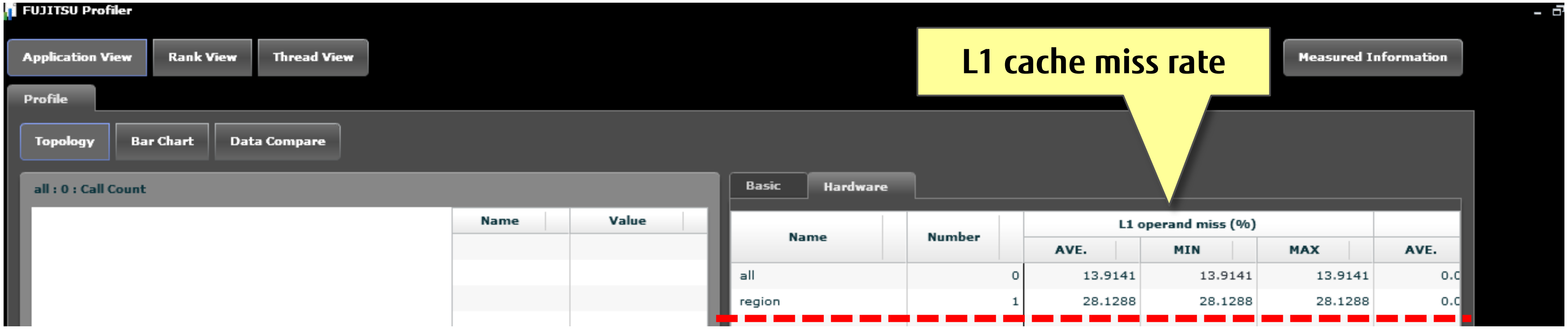
```
***************************************************************************
   Application
 ***************************************************************************

   Kind   Elapsed(s)              Inst    L1I miss(%)    L1D miss(%)
 ----------------------------------------------------------------------
    AVG       9.1036   27116915236          0.1074           13.9141   all 0
    MAX       9.1036   27116915236          0.1074           13.9141
    MIN       9.1036   27116915236          0.1074           13.9141

   Kind   Elapsed(s)             Inst    L1I miss(%)    L1D miss(%)
 ----------------------------------------------------------------------
    AVG       6.0335    9648402545          0.1795          28.1288   region 1
    MAX       6.0335    9648402545          0.1795          28.1288
    MIN       6.0335    9648402545          0.1795          28.1288
```

- You can see that the L1D cache miss rate of an advanced profiler routine, region 1, is about 28%.

Detailed Analysis of a High-cost Loop

GUI output of visualization and detailed information (Cache)

GUI visualization example



L1 cache miss rate

Name	Value

Name	Number	L1 operand miss (%)			
		AVE.	MIN	MAX	AVE.
all	0	13.9141	13.9141	13.9141	0.0
region	1	28.1288	28.1288	28.1288	0.0

- You can see that the L1D cache miss rate of an advanced profiler routine, region 1, is about 28%.

Sequential Tuning : Tuning Work

- Perform tuning based on the profiler results.
  - L1D cache thrashing occurs because each array is located on a 16-KB boundary. Consequently, the following is a frequent event: No instruction commit due to L2 cache for a floating-point load instruction.

```
 66            parameter(n=256,m=256)
 67            real*8  a(n, m),b(n,m),c(n,m),d(n,m),e(n,m),f(n,m),g(n,m),h(n,m)
 68            common /test/a,b,c,d,e,f,g,h
  :
            <<< Loop-information Start >>>
            <<<  [PARALLELIZATION]
            <<<    Standard iteration count: 422
            <<<  [OPTIMIZATION]
            <<<    COLLAPSED
            <<<    SIMD(VL: 4)
            <<<    SOFTWARE PIPELINING
            <<< Loop-information  End >>>
 72  1  pp   6v    do j = 1 , m
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    COLLAPSED
            <<< Loop-information  End >>>
 73  2   p   6      do i = 1 , n
 74  2   p   6v       a(i, j) = b(i, j) + c(i, j) + d(i, j)+e(i,j)+f(i,j)+g(i,j)+h(i,j)
 75  2   p   6v      enddo
 76  1   p         enddo
```

Array size
256 x 256 x 8 B = 32 x 16 KB
(16-KB boundary)

Sequential Tuning : Tuning Work

- As a solution, array merging reduces to reduce the number of streams from eight to two, thereby preventing L1D cache thrashing.

```
70            parameter(n=256,m=256)
71            real*8  abcd(4,n,m),efgh(4,n,m)
72            common /test/abcd,efgh
 :
           <<< Loop-information Start >>>
           <<<  [PARALLELIZATION]
           <<<    Standard iteration count: 2
           <<< Loop-information  End >>>
76  1  pp        do j = 1 , m
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 4)
           <<<    SOFTWARE PIPELINING
           <<< Loop-information  End >>>
77  2   p   4v      do i = 1 , n
78  2   p   4v         abcd(1,i,j)=abcd(2,i,j)+abcd(3,i,j)+abcd(4,i,j)+efgh(1,i,j)+efgh(2,i,j)+efgh(3,i,j)+efgh(4,i,j)
79  2   p   4v      enddo
80  1   p        enddo
```

8 array merging reduces in units of 4.

Sequential Tuning: Verifying Tuning Results

- To check the tuning effect, collect information and check the visualization with the profilers again.

Text visualization example

Performance monitor : Cache

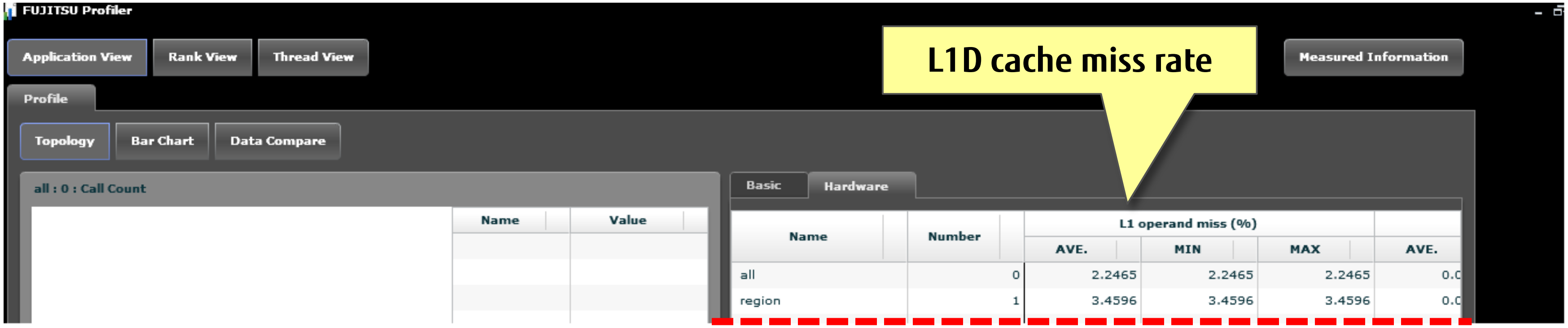
```
 ***************************************************************
    Application
  ***************************************************************

    Kind  Elapsed(s)              Inst   L1I miss(%)   L1D miss(%)
 -------------------------------------------------------------------------
     AVG       4.0285   26906348505       0.1053           2.2465   all 0
     MAX       4.0285   26906348505       0.1053           2.2465
     MIN       4.0285   26906348505       0.1053           2.2465

    Kind  Elapsed(s)             Inst   L1I miss(%)   L1D miss(%)
 ------------------------------------------------------------------------
     AVG       0.8705    9437208662       0.1797           3.4596   region 1
     MAX       0.8705    9437208662       0.1797           3.4596
     MIN       0.8705    9437208662       0.1797           3.4596
```

- You can see that the L1D cache miss rate has improved from about 28% (before improvement) to about 3.46%.

Sequential Tuning: Detailed Analysis of a High-cost Loop

GUI output of visualization and detailed information (**Cache**)

GUI visualization example



L1D cache miss rate

Name	Number	L1 operand miss (%)			
		AVE.	MIN	MAX	AVE.
all	0	2.2465	2.2465	2.2465	0.0
region	1	3.4596	3.4596	3.4596	0.0

- You can see that the L1D cache miss rate has improved from about 28% (before improvement) to about 3.46%.

Sequential Tuning: Verifying Tuning Results

- Check execution times before and after the tuning.

Text visualization example

(Before improvement of target loop)

```
Basic profile

Kind  Elapsed(s)    User(s)   System(s)       Call
------------------------------------------------------
  AVG      5.7754     89.6100      0.0700   25600.0000   region 1
```

Before improvement: 5.7754 sec

(After improvement of target loop)

```
Basic profile

Kind  Elapsed(s)    User(s)   System(s)       Call
------------------------------------------------------
  AVG      0.6079      9.1600      0.0500   25600.0000   region 1
```

After improvement: 0.6079 sec and resulting 9.5-fold improvement

- Summary of tuning results
  - Array merging reduced the number of streams and prevented L1D cache thrashing. The result was an improvement in cache efficiency.
  - A comparison between execution times before and after the improvement shows a resulting 9.5-fold improvement at the target loop.

MPI Tuning

- Cost Information Collection and Simple Analysis
- Detailed Analysis of MPI Information
- Tuning Work
- Verifying Tuning Results

MPI Tuning: Cost Information Collection and Simple Analysis

<MPI tuning example>

- Collecting information with the instant profiler (fipp)
  - Use the fipp and fipppx commands of the instant profiler in the same way as described for sequential tuning.

Text visualization example

Procedures profile

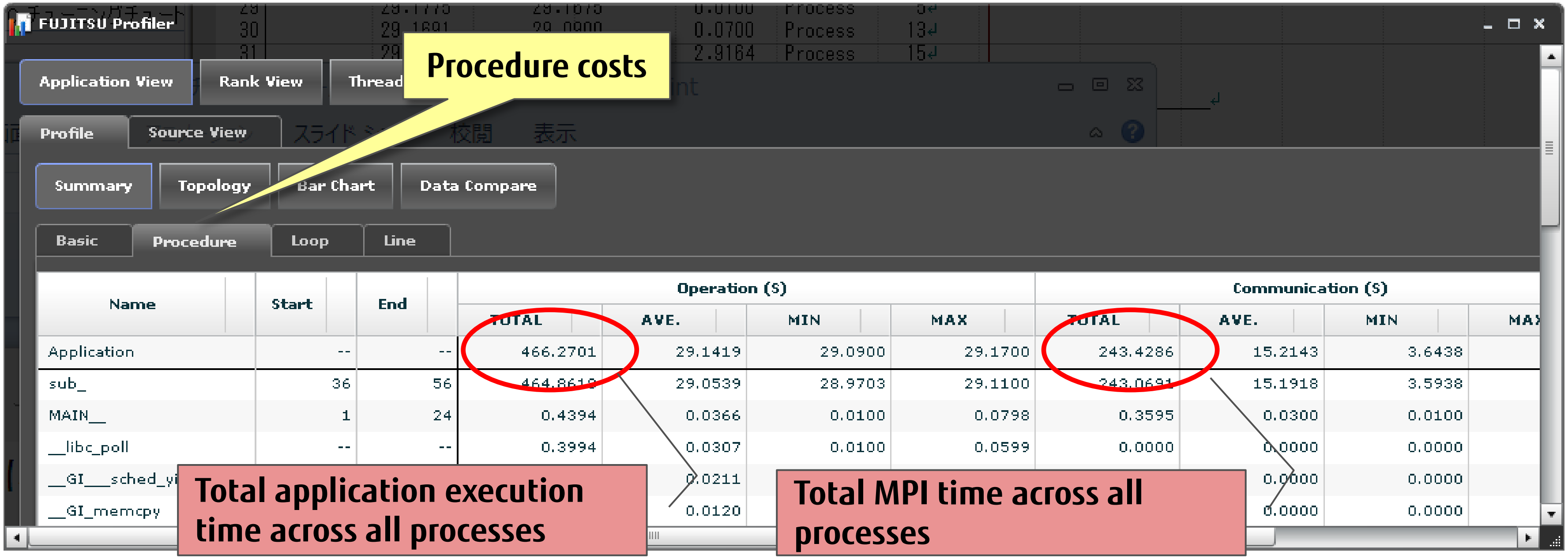
```
*****************************************************************************
  Application - procedures
*****************************************************************************

        Cost        %  Operation (S)   Start    End
----------------------------------------------------------
       46679   100.0000      466.7900      --       --  Application
----------------------------------------------------------
       46538    99.6979      465.3800      36       56  sub_
          44     0.0943        0.4400       1       24  MAIN__
        (snip)

         MPI        %  Communication (S)   Start    End
--------------------------------------------------------------
       24369    52.2055        243.6900      --       --  Application
--------------------------------------------------------------
       24333    52.2863        243.3300      36       56  sub_
```

- You can see that the MPI cost is high since the MPI percentage of the overall application execution time is about 52%.
- Collect data with the advanced profiler to investigate the use conditions of the MPI cost.

Cost Information Collection and Simple Analysis

GUI output of visualization and basic information

GUI visualization example



Procedure costs

Name	Start	End	Operation (S)				Communication (S)			
			TOTAL	AVE.	MIN	MAX	TOTAL	AVE.	MIN	MA
Application	--	--	466.2701	29.1419	29.0900	29.1700	243.4286	15.2143	3.6438	
sub_	36	56	464.8619	29.0539	28.9703	29.1100	243.0691	15.1918	3.5938	
MAIN__	1	24	0.4394	0.0366	0.0100	0.0798	0.3595	0.0300	0.0100	
__libc_poll	--	--	0.3994	0.0307	0.0100	0.0599	0.0000	0.0000	0.0000	
__GI___sched_yi	[illegible]	[illegible]	[illegible]	0.0211	[illegible]	[illegible]	[illegible]	0.0000	0.0000	
__GI_memcpy	[illegible]	[illegible]	[illegible]	0.0120	[illegible]	[illegible]	[illegible]	0.0000	0.0000	

Total application execution time across all processes

Total MPI time across all processes

- Check the total application execution time and total MPI time across all processes, and you can see that the MPI percentage is high at about 52%.
- Collect data with the advanced profiler to investigate the use conditions of the MPI cost.

MPI Tuning: Detailed Analysis of MPI Information

FUJITSU

- Collecting information with the advanced profiler (fapp)

Information collection command example

```
fapp -C –Impi -d prof_fapp mpiexec a.out
```

- *-C* : Gives an instruction to collect advanced profiling data.
- *-Impi* : Gives an instruction to collect MPI information.
- *-d* : Specifies the advanced profiling data name (name of the directory storing advanced profiling data files).

MPI Tuning: Detailed Analysis of MPI Information

- Text output of visualization and detailed information (MPI information)

Visualization command example

```
fapppx -A -d prof_fapp –o mpi.txt
```

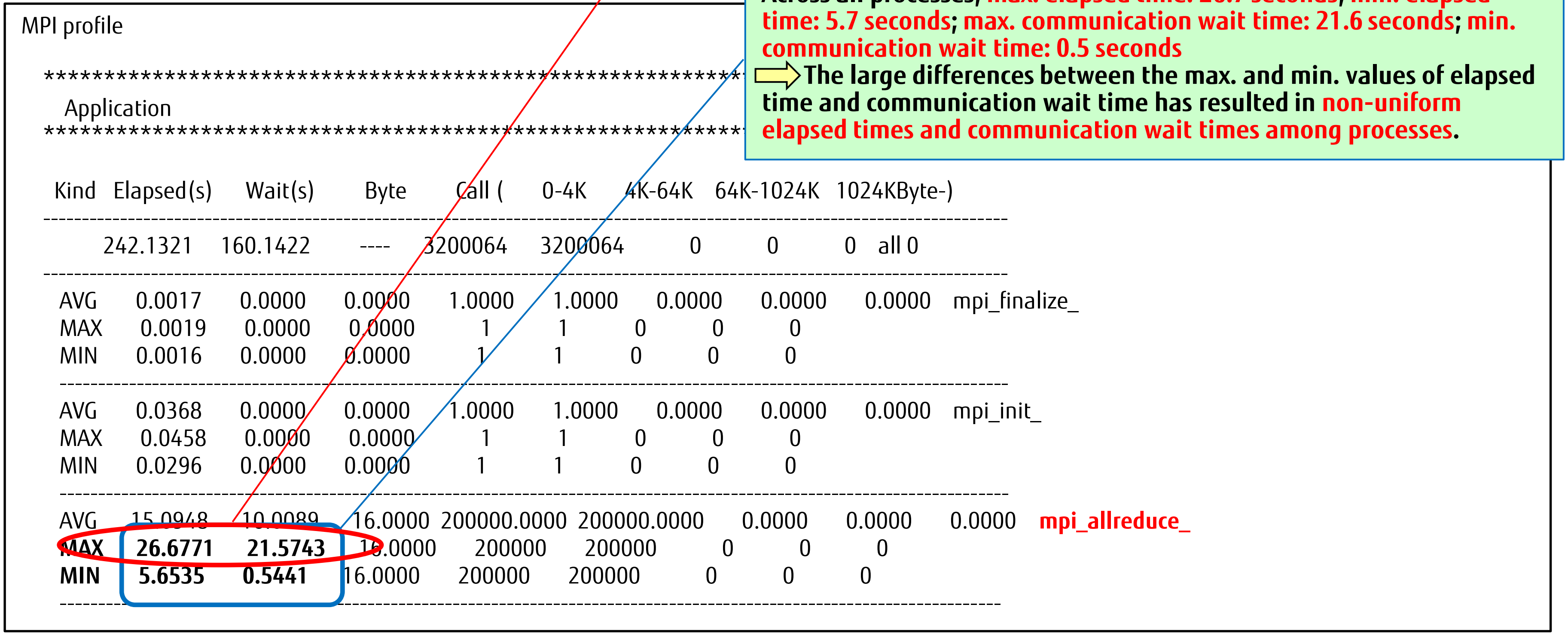
- *-A* : Gives an instruction to output advanced profiler information.
- *-d* : Specifies the advanced profiling data name.
- *-o* : Gives an instruction on the output destination of advanced profiler information.

For an MPI application, the default of *-I* is *-Impi*.
mpi means to output MPI information.

MPI Tuning: Detailed Analysis of MPI Information

FUJITSU

Text visualization example



```
MPI profile

 ****************************************************************
   Application
 ****************************************************************

  Kind  Elapsed(s)    Wait(s)      Byte      Call (     0-4K     4K-64K   64K-1024K  1024KByte-)
 ---------------------------------------------------------------------------------------------
        242.1321    160.1422      ----     3200064    3200064        0         0         0   all 0
 ---------------------------------------------------------------------------------------------
   AVG      0.0017      0.0000      0.0000      1.0000      1.0000      0.0000      0.0000      0.0000   mpi_finalize_
   MAX      0.0019      0.0000      0.0000           1           1           0           0           0
   MIN      0.0016      0.0000      0.0000           1           1           0           0           0
   ------------------------------------------------------------------------------------------
   AVG      0.0368      0.0000      0.0000      1.0000      1.0000      0.0000      0.0000      0.0000   mpi_init_
   MAX      0.0458      0.0000      0.0000           1           1           0           0           0
   MIN      0.0296      0.0000      0.0000           1           1           0           0           0
   ------------------------------------------------------------------------------------------
   AVG     15.0948     10.0089     16.0000  200000.0000  200000.0000      0.0000      0.0000      0.0000   mpi_allreduce_
   MAX     26.6771     21.5743     16.0000       200000       200000           0           0           0
   MIN      5.6535      0.5441     16.0000       200000       200000           0           0           0
   ------------------------------------------------------------------------------------------
```

The communication wait time of 21.6 seconds accounts for a large percentage of the elapsed time of 26.7 seconds.

Across all processes, max. elapsed time: 26.7 seconds; min. elapsed time: 5.7 seconds; max. communication wait time: 21.6 seconds; min. communication wait time: 0.5 seconds
⇨The large differences between the max. and min. values of elapsed time and communication wait time has resulted in non-uniform elapsed times and communication wait times among processes.

- **MPI_Allreduce has the highest cost among MPI functions.**
- **The communication wait time of MPI_Allreduce accounts for a large percentage of the elapsed time.**
- **You can also see that the large differences between the maximum and minimum values of elapsed time and latency has resulted in non-uniform elapsed times and latencies among processes.**
- **Next, check the cost information for the MPI functions of each process in the same text data to check the MPI_Allreduce status in each process.**

MPI Tuning: Detailed Analysis of MPI Information

Text visualization example

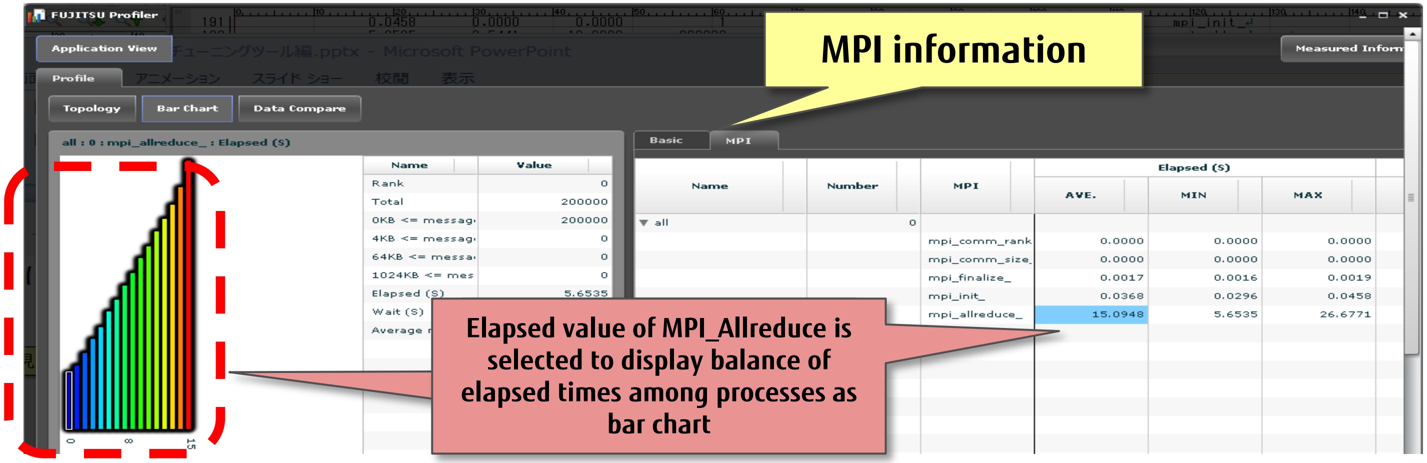
```
MPI profile
  ***************************************************************************
    Application
  ***************************************************************************


   (snip)

  ***************************************************************************
    Process 0
  ***************************************************************************
       Elapsed(s)    Wait(s)      Byte      Call (     0-4K     4K-64K   64K-1024K  1024KByte-)
          5.6535      0.5441    16.0000    200000    200000          0          0          0  mpi_allreduce_

   (snip)

  ***************************************************************************
    Process 15
  ***************************************************************************
       Elapsed(s)    Wait(s)      Byte      Call (     0-4K     4K-64K   64K-1024K  1024KByte-)
         26.6771     21.5743    16.0000    200000    200000          0          0          0  mpi_allreduce_
```

- **Both the elapsed time and communication wait time of process 0 are short, while both the elapsed time and communication wait time of process 15 are long.**
- **A load imbalance has occurred between the processes. Process 15 is waiting for process 0 to finish operation and execute MPI_Allreduce.**

MPI Tuning: Detailed Analysis of MPI Information

- Example of GUI output of visualization and detailed information (MPI information)

GUI visualization example



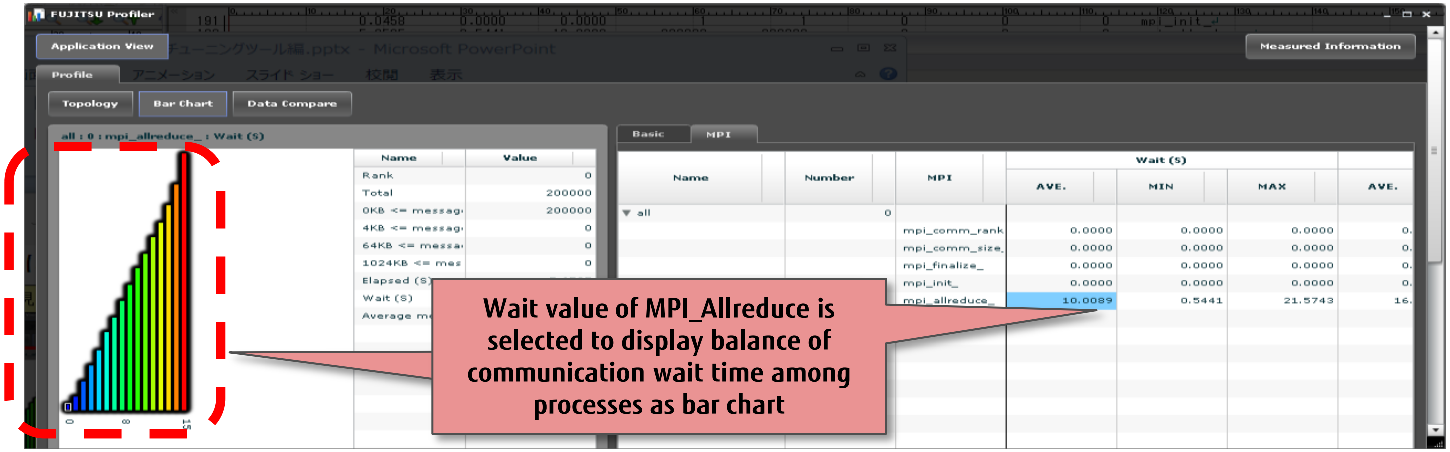
all : 0 : mpi_allreduce_ : Elapsed (S)

Name	Value
Rank	0
Total	200000
0KB <= messag	200000
4KB <= messag	0
64KB <= messa	0
1024KB <= mes	0
Elapsed (S)	5.6535
Wait (S)	
Average r	

Name	Number	MPI	Elapsed (S)		
			AVE.	MIN	MAX
▼ all	0				
		mpi_comm_rank	0.0000	0.0000	0.0000
		mpi_comm_size_	0.0000	0.0000	0.0000
		mpi_finalize_	0.0017	0.0016	0.0019
		mpi_init_	0.0368	0.0296	0.0458
		mpi_allreduce_	15.0948	5.6535	26.6771

MPI information

Elapsed value of MPI_Allreduce is selected to display balance of elapsed times among processes as bar chart



all : 0 : mpi_allreduce_ : Wait (S)

Name	Value
Rank	0
Total	200000
0KB <= messag	200000
4KB <= messag	0
64KB <= messa	0
1024KB <= mes	0
Elapsed (S)	
Wait (S)	
Average m	

Name	Number	MPI	Wait (S)			
			AVE.	MIN	MAX	AVE.
▼ all	0					
		mpi_comm_rank	0.0000	0.0000	0.0000	0.
		mpi_comm_size_	0.0000	0.0000	0.0000	0.
		mpi_finalize_	0.0000	0.0000	0.0000	0.
		mpi_init_	0.0000	0.0000	0.0000	0.
		mpi_allreduce_	10.0089	0.5441	21.5743	16.

Wait value of MPI_Allreduce is selected to display balance of communication wait time among processes as bar chart

- From the bar charts showing the balance between elapsed times and balance between communication wait time, you can see that the elapsed time and communication wait time of MPI_Allreduce vary among processes.

MPI Tuning: Tuning Work

- Perform tuning based on the profiler results.
  - The elapsed time and communication wait time of an MPI function vary among processes, so consider how the operation load imbalance among the processes can be resolved.
  - After checking the source, you will see that a triangular loop (*) before the high-cost MPI function has been divided into blocks for process parallelization processing. Consequently, the amount of operation calculation varies among processes.

```
subroutine sub(ista, iend)
 include 'mpif.h'
 integer*8 i,j,n
 parameter(n=1024)
 real*8 a(n+1,n),b(n+1,n),c(n+1,n)
 integer ista, iend
 integer ierr
 common a,b,c

  do j=ista, iend
   do i=j, n
    a(i,j)=b(i,j)/c(i,j)
   enddo
  enddo

 call MPI_Allreduce(a(i-1,j), a(i,j), 1, MPI_REAL8,
&      MPI_MAX, MPI_COMM_WORLD, ierr)

 end
```

Triangular loop
(Initial value of inner loop is control variable of outer loop)

High-cost MPI function

*** Triangular loop**
A loop in which the initial value of an inner loop is determined by the control variable of an outer loop. Dividing this loop into blocks causes a load imbalance to occur.

MPI Tuning: Tuning Work

- To improve the imbalance in the amount of calculation among processes, change the method of data division for a triangular loop from block division to cyclic division.
- For cyclic division in a triangular loop, correct the initial and final values and specify the incremental value of the control variable.

```
subroutine sub(isize, irank)
    include 'mpif.h'
    integer*8 i,j,n
    parameter(n=1024)
    real*8 a(n+1,n),b(n+1,n),c(n+1,n)
    integer ierr, isize, irank
    common a,b,c

      do j=irank+1, n, isize
        do i=j, n
          a(i,j)=b(i,j)/c(i,j)
        enddo
      enddo

     call MPI_Allreduce(a(i-1,j), a(i,j), 1, MPI_REAL8,
   &       MPI_MAX, MPI_COMM_WORLD, ierr)

    end
```

Correct the initial value according to the MPI process number, and revise the final value to the last array element number. Add the number of MPI processes as the incremental value.

MPI Tuning: Verifying Tuning Results

- To check the tuning effect, collect information and check the visualization with the profilers again.

Text visualization example

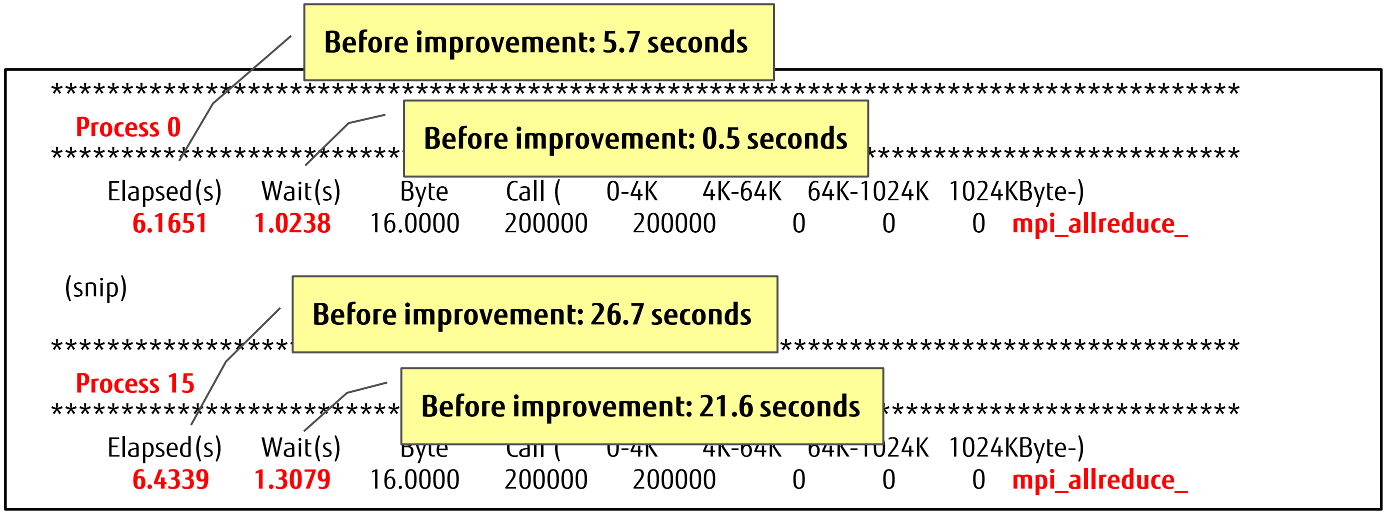
MPI profile

```
 *******************************************************************************
     Application
   *******************************************************************************

     Kind   Elapsed(s)     Wait(s)       Byte            Call (      0-4K      4K-64K    64K-1024K   1024KByte-)
   ---------------------------------------------------------------------------------------------------------------
             100.9875     18.3668       ----         3200064      3200064          0            0            0   all 0
   ---------------------------------------------------------------------------------------------------------------
      AVG       0.0019     0.0000       0.0000          1.0000       1.0000      0.0000       0.0000       0.0000   mpi_finalize_
      MAX        0.0021     0.0000       0.0000              1            1           0            0            0
      MIN       0.0017     0.0000       0.0000              1            1           0            0            0
     -------------------------------------------------------------------------------------------------------------
      AVG       0.0400     0.0000       0.0000          1.0000       1.0000      0.0000       0.0000       0.0000   mpi_init_
      MAX        0.0460     0.0000       0.0000              1            1           0            0            0
      MIN       0.0309     0.0000       0.0000              1            1           0            0            0
     -------------------------------------------------------------------------------------------------------------
      AVG       6.2698     1.1479      16.0000  200000.0000  200000.0000      0.0000       0.0000       0.0000   mpi_allreduce_
      MAX       6.8456     1.7255      16.0000       200000       200000           0            0            0
      MIN       5.7971     0.6905      16.0000       200000       200000           0            0            0
     -------------------------------------------------------------------------------------------------------------
```

- You can see that the elapsed time (maximum value) of the MPI_Allreduce function has improved from 26.7 seconds (before improvement) to 6.8 seconds. Also, the communication wait time (maximum value) has improved from 21.6 seconds (before improvement) to 1.7 seconds (after improvement).

Verifying Tuning Results

Text visualization example



Before improvement: 5.7 seconds

Before improvement: 0.5 seconds

Before improvement: 26.7 seconds

Before improvement: 21.6 seconds

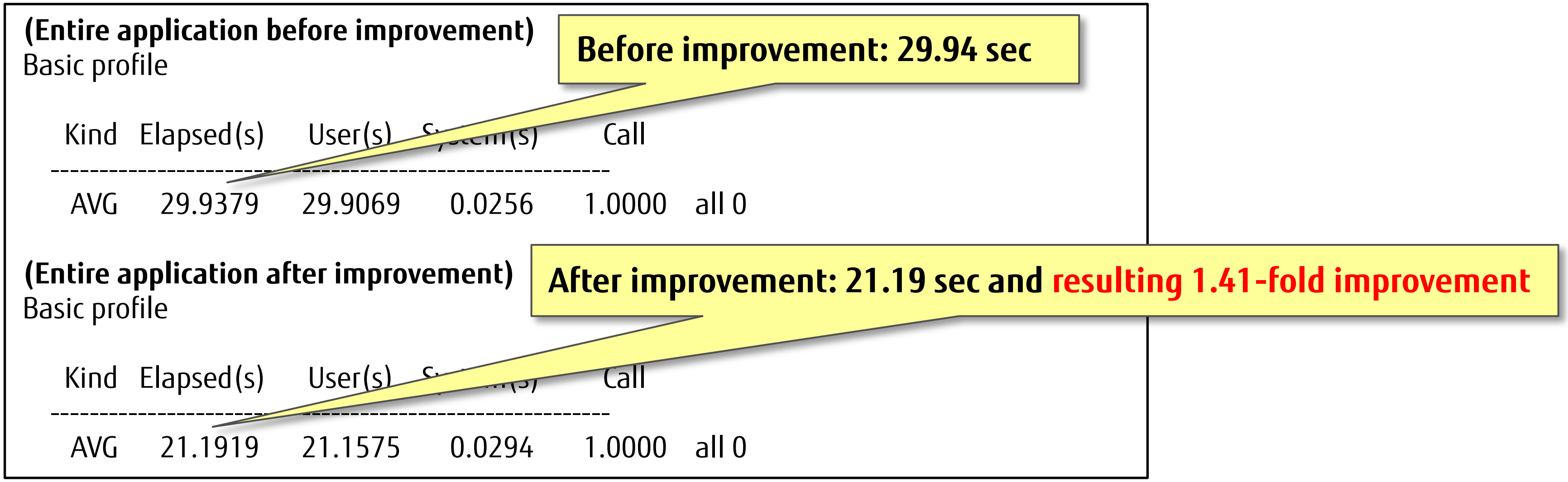
```
*******************************************************************************
  Process 0
*******************************************************************************
      Elapsed(s)    Wait(s)      Byte       Call (     0-4K      4K-64K    64K-1024K   1024KByte-)
        6.1651      1.0238     16.0000     200000     200000          0          0          0   mpi_allreduce_

 (snip)

*******************************************************************************
  Process 15
*******************************************************************************
      Elapsed(s)    Wait(s)      Byte       Call (     0-4K      4K-64K    64K-1024K   1024KByte-)
        6.4339      1.3079     16.0000     200000     200000          0          0          0   mpi_allreduce_
```

- Before improvement, the large differences in the elapsed time and communication wait time between processes 0 and 15 are due to the effect of operation load imbalance.
- After tuning, the improvement in operation load imbalance reduces variations in the elapsed time and communication wait time between the processes.

MPI Tuning: Verifying Tuning Results

- Check the execution times before tuning and after improvement.

Profiler output example



```
(Entire application before improvement)
Basic profile

  Kind   Elapsed(s)     User(s)    System(s)       Call
 -----------------------------------------------------------
   AVG      29.9379     29.9069       0.0256      1.0000   all 0

(Entire application after improvement)
Basic profile

  Kind   Elapsed(s)     User(s)    System(s)       Call
 -----------------------------------------------------------
   AVG      21.1919     21.1575       0.0294      1.0000   all 0
```

Before improvement: 29.94 sec

After improvement: 21.19 sec and resulting 1.41-fold improvement

- Summary of tuning results
  - The method of division for a triangular loop that causes a communication wait for an MPI function was changed from block division to cyclic division. The result was an improvement in load imbalance between processes.
  - A comparison between execution times before and after the improvement shows a resulting 1.41-fold improvement in the entire application.

Precision PA Visibility Function (Excel Format)

- Data Collection (Execution)
- Data Analysis
- Aspects of Excel Sheets

Overview

- The precision PA visibility function (Excel format) can analyze data collected in a certain format by importing it to an Excel sheet, and display the results in graphs and tables.

  - This analysis requires that the fapp command be executed eleven times. The reason for that requirement is that hardware counter information of that amount is needed. Consequently, the execution time may differ slightly. Those differences may have the effect of negative numerical values for some information.

  - The operation of the profiler follows the procedure below.
    1. Determine a measurement section, and insert information collection routines in a program.
    2. Compile the source code.
    3. Collect data.
    4. Convert data.
    5. Analyze data by using the Excel sheet.

Data Collection (Execution)

- Measurement Section Specification
- Compilation
- Information Collection by fapp
- Information Collection Command Example

* For details, see the *Profiler Usage Guide "3.2.4 Compilation"*.

Data Collection (Execution) (1)

- Inserting information collection routines for a measurement section
    - Determine a measurement section and section name for a program.
    - Insert information collection routines (start_collection/stop_collection) for the measurement target section.

Specify the section name in an argument.

```
program
  call start_collection("region1")
  call sub1()
  call sub2()
  call stop_collection("region1")
end
```

Measurement section enclosed

(Precision PA information in this section is collected.)

- Compilation
    - Program compilation requires a tool library option (-Ntl_trt), but since the option is specified by default, users do not need to pay special attention to it.

Data Collection (Execution) (2)

- Information collection by fapp
  - Use the fapp command to collect data.
  - Execute the fapp command a total of eleven times.
  - The following fapp command options must be specified.
    - -C : Gives an instruction to output advanced profiler information.
    - -Hpa=*no* : Outputs in the precise PA format or measurement event number specification.
    - -d *profiling_data* : Specifies a profiling data name.
  - **Options at the data collection time**
    - –C –d Fapp_pa -Hpa=*n* a.out

Specify a number between 1 and 11 in n.

1st time: -Hpa=1, 2nd time: -Hpa=2, 3rd time: -Hpa=3, 4th time: -Hpa=4, 5th time: -Hpa=5,
6th time: -Hpa=6, 7th time: -Hpa=7, 8th time: -Hpa=8, 9th time: -Hpa=9, 10th time: -Hpa=10,
11th time: -Hpa=11

Data Collection (Execution) (3)

Information collection command example

The following example shows the information collection command executed eleven times.
The fapp command is executed on a compute node.

```
#PA1
fapp –C –d pa1 -Hpa=1 a.out
#PA2
fapp –C –d pa2 -Hpa=2 a.out
#PA3
fapp –C –d pa3 -Hpa=3 a.out
#PA4
fapp –C –d pa4 -Hpa=4 a.out
#PA5
fapp –C –d pa5 -Hpa=5 a.out
#PA6
fapp –C –d pa6 -Hpa=6 a.out
#PA7
fapp –C –d pa7 -Hpa=7 a.out
#PA8
fapp –C –d pa8 -Hpa=8 a.out
#PA9
fapp –C –d pa9 -Hpa=9 a.out
#PA10
fapp –C –d pa10 -Hpa=10 a.out
#PA11
fapp –C –d pa11 -Hpa=11 a.out
```

Data Analysis

- Information Output by fapppx
- Information Output Command Example
- Operations in Excel

Data Analysis (1)

- Information output by fapppx
  - The data output by fapppx must be converted to CSV format before analysis on an Excel sheet.
  - The following fapppx options must be specified.
    - -A: Gives an instruction to output advanced profiler information.
    - -tcsv: Outputs collected information as a CSV file.
    - -H *hardmon* : Specifies output in the format for hardware monitor information (precision PA).
    - -d *profiling_data* : Specifies the profiling data to be analyzed.
  - The precision PA information file to be output for importing tabulated results must have the following name:
    - **output_prof_1.csv to output_prof_11.csv**

Data Analysis (2)

Information output command example

Collected data is converted into CSV data.
The fapppx command is executed on a login node at the front end.

```
#PA1
fapppx –A –d pa1 -o output_prof_1.csv –tcsv -Hpa
#PA2
fapppx –A –d pa2 -o output_prof_2.csv –tcsv -Hpa
#PA3
fapppx –A –d pa3 -o output_prof_3.csv –tcsv -Hpa
#PA4
fapppx –A –d pa4 –o output_prof_4.csv –tcsv -Hpa
#PA5
fapppx –A –d pa5 -o output_prof_5.csv –tcsv -Hpa
#PA6
fapppx –A –d pa6 –o output_prof_6.csv –tcsv -Hpa
#PA7
fapppx –A –d pa7 –o output_prof_7.csv –tcsv –Hpa
#PA8
fapppx –A –d pa8 –o output_prof_8.csv –tcsv -Hpa
#PA9
fapppx –A –d pa9 -o output_prof_9.csv –tcsv -Hpa
#PA10
fapppx –A –d pa10 –o output_prof_10.csv –tcsv -Hpa
#PA11
fapppx –A –d pa11 –o output_prof_11.csv –tcsv –Hpa
```

Data Analysis (3)

Operations in Excel

- Arranging data

  Prepare an Excel worksheet for importing data.

  Place each CSV file (output_prof_1.csv to output_prof_11.csv) output by the fapppx command in the same folder as the above Excel worksheet.

- Double-click the Excel sheet to start Excel, and a macro runs automatically to start reading the CSV file.
- Removing the security warning

  This Excel sheet uses a macro. Therefore, if macros are disabled in your security settings, enable macros.

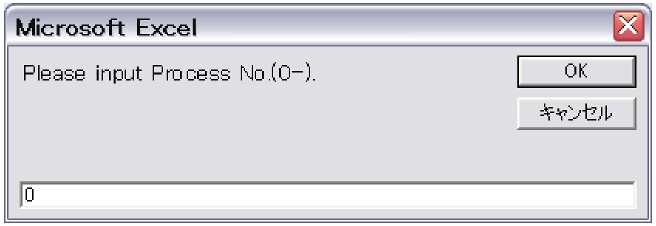
  If macros cannot run in Excel, change the Excel macro security level.

  The operations to enable macros vary depending on the Excel version.

Data Analysis (4)

Specifying a process number

A process number specification dialog box appears automatically when the macro starts. Specify the process number of the process to analyze.

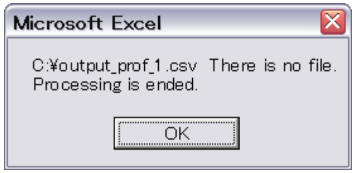


Microsoft Excel

Please input Process No.(0–).

0

Note: If the folder does not contain the target file (output_prof_1.csv to output_prof_11.csv) or if the specified process number is wrong, processing stops and Excel quits.



Microsoft Excel

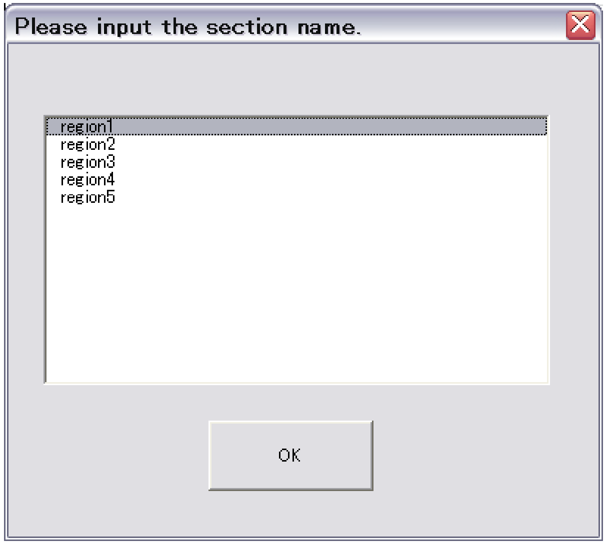
C:¥output_prof_1.csv There is no file.
Processing is ended.

Microsoft Excel

There is no specified process.

Specifying a section name (measurement section)

A region name specification dialog box appears when there are no process specification errors. Specify the name of the section to analyze.



Please input the section name.

Data Analysis (5)

- Generating an Excel sheet

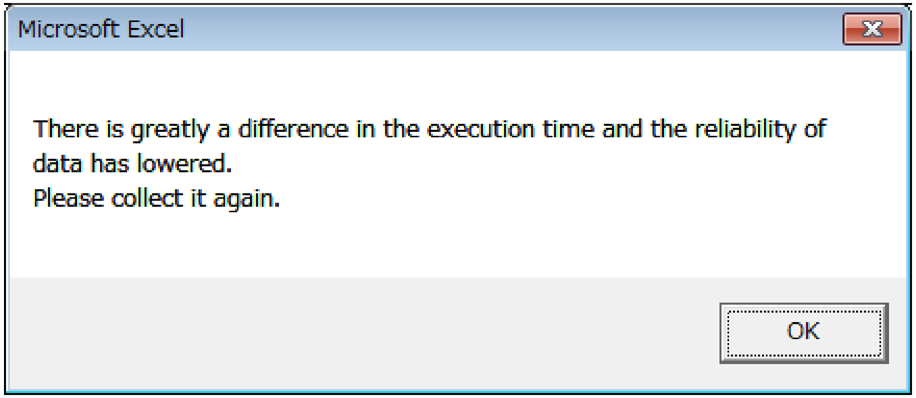
  Excel sheet generation begins when there are no measurement section specification errors.

- Notice:

  Data is collected eleven times, and the resulting data contains some differences. These differences appear at the bottom of the second Excel sheet.

  If the execution time differs from the first execution time by 5% or more (shown as 95% or less or 105% or more), a warning dialog box appears.

  If the difference is large and high precision is required, you are recommended to check this execution time difference and collect data again in order to use data with insignificant differences.



There is greatly a difference in the execution time and the reliability of data has lowered.
Please collect it again.

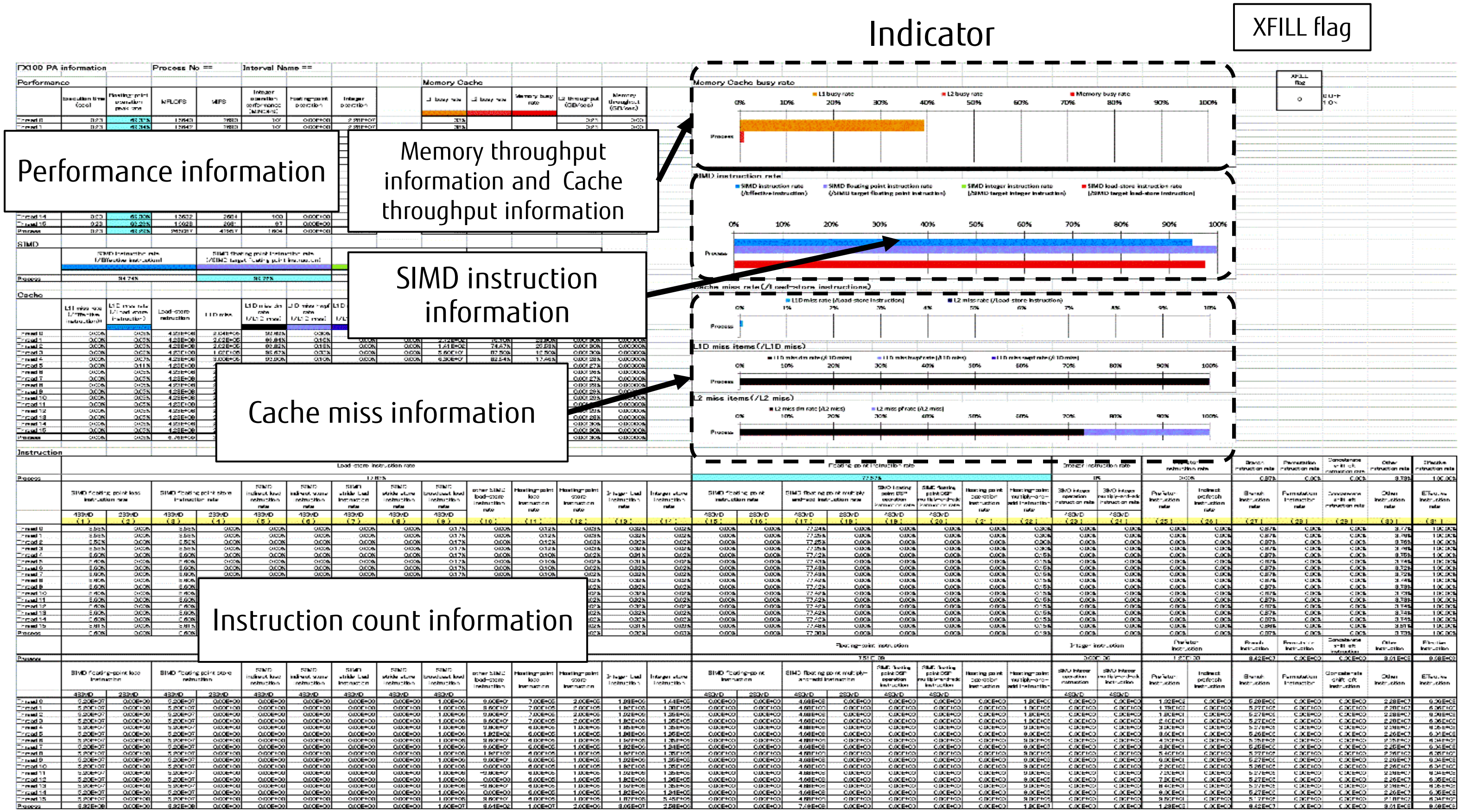
Aspects of Excel Sheets

- Tabulated Result Example

Aspects of Excel Sheets

Tabulated result example (1)

- The tabulated results are output to an Excel sheet (two A4 pages when printed).
- The following information is tabulated on the first page.



Indicator

XFILL flag

Performance information

Memory throughput information and Cache throughput information

SIMD instruction information

Cache miss information

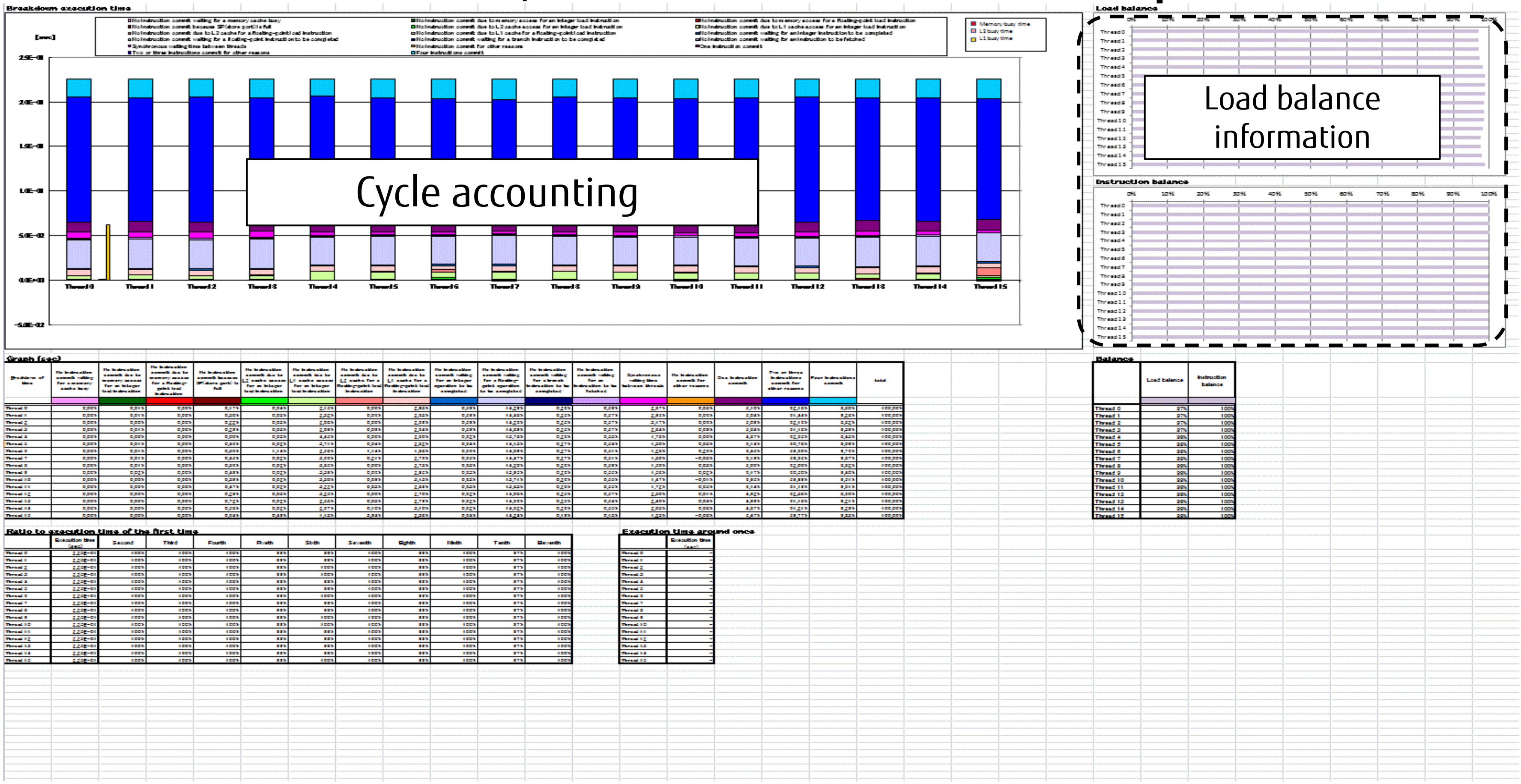
Instruction count information

Aspects of Excel Sheets

Tabulated result example (2)

- The second pages contains a graph and the tabulated time information from which the graph was generated.

*** For details on the contents, see "PA Information Lists" in "Chapter 6 PA Event."**



Revision History

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

FUJITSU

shaping tomorrow with you