FUJITSU

shaping tomorrow with you

Getting the best performance from massively parallel computer

June 6th, 2013

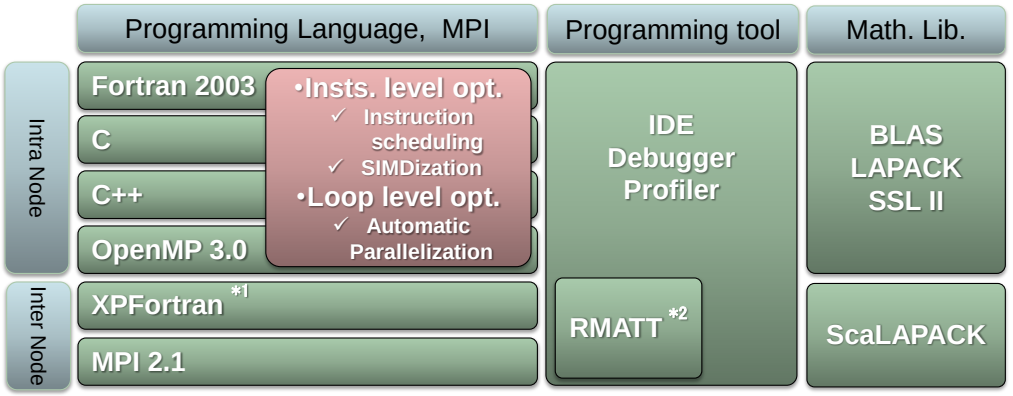
Takashi Aoki

Next Generation Technical Computing Unit

Fujitsu Limited

Agenda

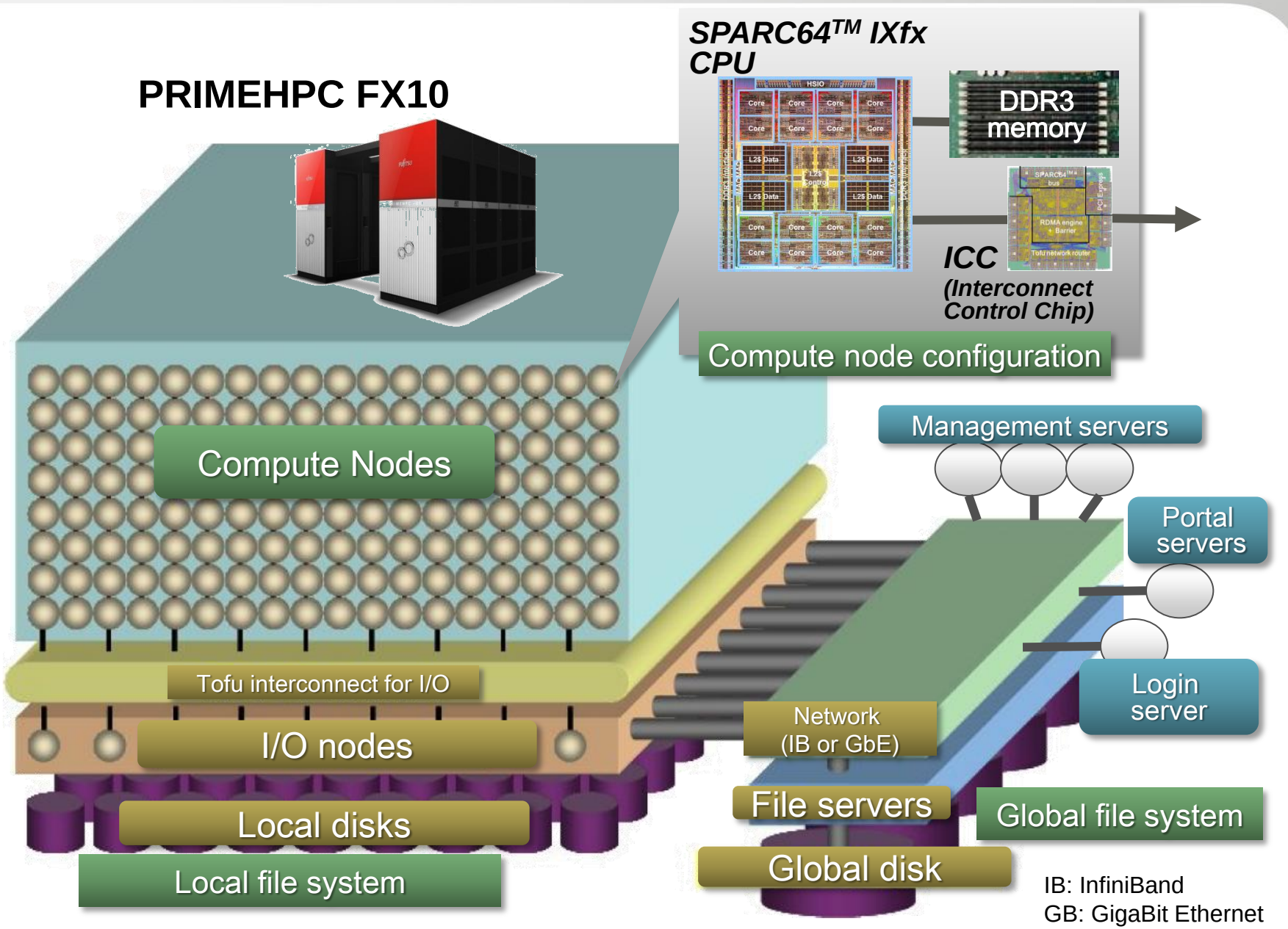
- Second generation petascale supercomputer PRIMEHPC FX10
- Tuning techniques for PRIMEHPC FX10



*1: eXtended Parallel Fortran (Distributed Parallel Fortran)
*2: Rank Map Automatic Tuning Tool

PRIMEHPC FX10 System Configuration

FUJITSU

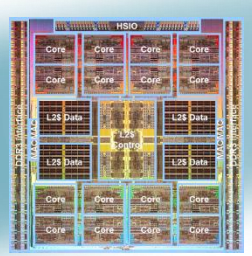


FX10 System H/W Specifications

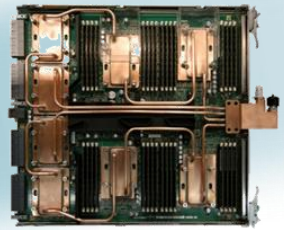
FUJITSU

PRIMEHPC FX10 H/W Specifications		
CPU	Name	*SPARC64™ IXfx*
	Performance	236.5GFlops@1.848GHz
Node	Configuration	1 CPU / Node
	Memory capacity	32, 64 GB
Rack	Performance/rack	22.7 TFlops
System (4 ~1024 racks)	No. of compute node	384 to 98,304
	Performance	90.8TFlops to 23.2PFlops
	Memory	12 TB to 6 PB

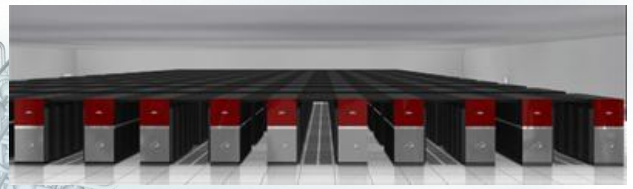
- ***SPARC64™ IXfx CPU***
  - 16 cores/socket
  - 236.5 GFlops



- ***System rack***
  - 96 compute nodes
  - 6 I/O nodes
  - With optional water cooling exhaust unit



- ***System board***
  - 4 nodes (4 CPUs)



- ***System***
  - Max. 23.2 PFlops
  - Max. 1,024 racks
  - Max. 98,304 CPUs

Comparison of System H/W Specifications

FUJITSU

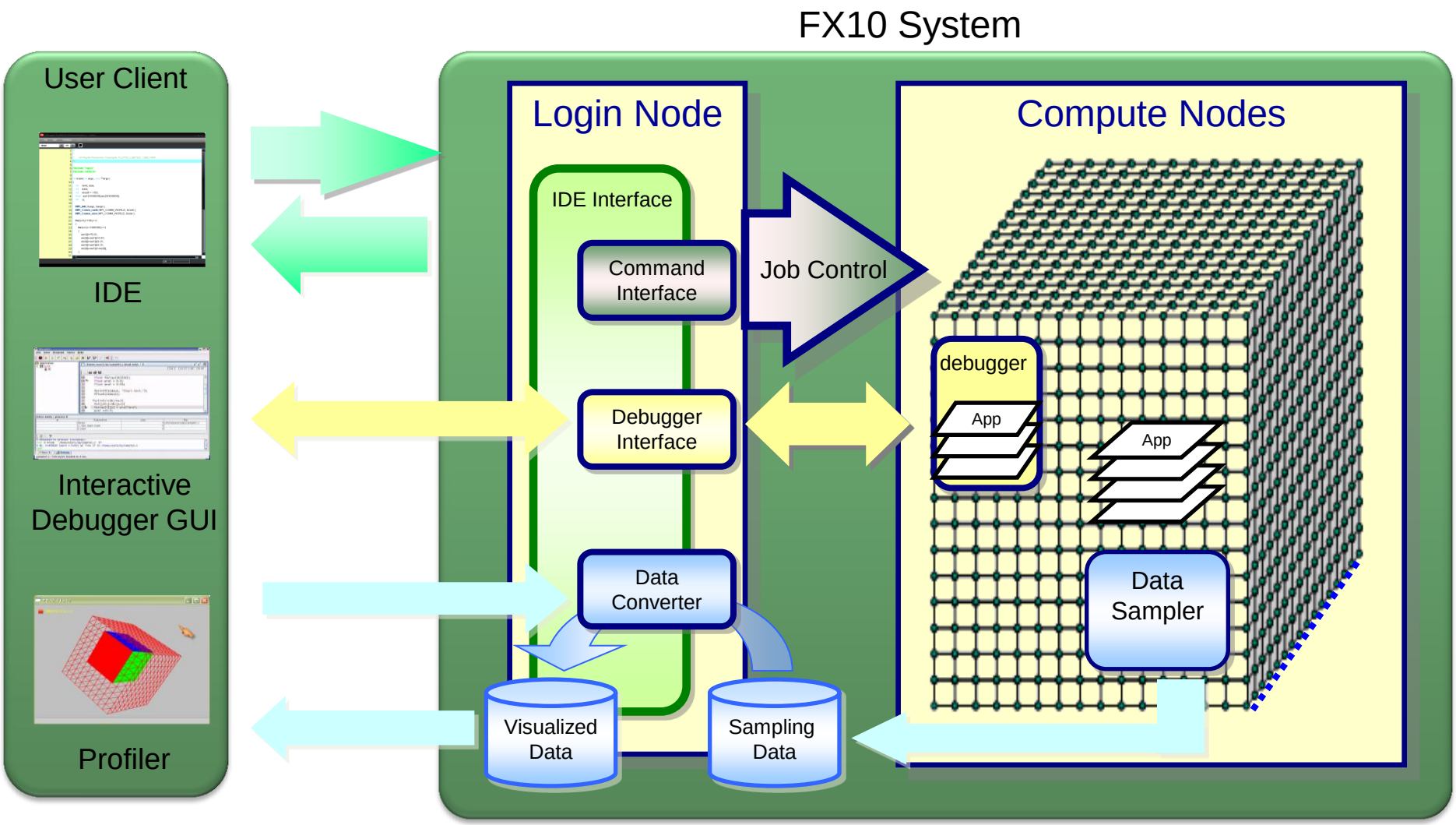
		K computer	FX10
CPU	Name	*SPARC64™ VIIIfx*	*SPARC64™ IXfx*
	Performance	128GFlops@2GHz	236.5GFlops@1.848GHz
	Architecture	SPARC V9 + HPC-ACE extension	←
	Cache configuration	L1(I) Cache:32KB/core, L1(D) Cache:32KB/core	←
		L2 Cache: 6MB(shared)	L2 Cache: 12MB(shared)
	No. of cores/socket	8	16
	Memory band width	64 GB/s.	85 GB/s.
Node	Configuration	1 CPU / Node	←
	Memory capacity	16 GB	32, 64 GB
System board	Node/system board	4 Nodes	←
Rack	System board/rack	24 System boards	←
	Performance/rack	12.3 TFlops	22.7 TFlops

Comparison of System H/W Specifications (cont.)

FUJITSU

		K computer	FX10
Interconnect	Topology	6D Mesh/Torus	←
	Performance	5GB/s x2 (bi-directional)	←
	No. of link per node	10	←
	Additional features	H/W barrier, reduction	←
		no external switch box	←
Cooling	CPU, ICC(interconnect chip), DDCON	Direct water cooling	←
	Other parts	Air cooling	Air cooling + Exhaust air water cooling unit (Optional)

Programming Environment



Tools for high performance computing

FUJITSU

- Hardware
  - Massively parallel supercomputer
    - SPARC64™ IXfx
    - Tofu interconnect
- Software
  - Parallel compiler
  - PA(Performance Analysis) information
  - Low jitter Operating System
  - Distributed File System

Tuning Techniques for FX10

FUJITSU

- Parallel programing style
  - Hybrid parallel
- Scalar tuning
- Parallel tuning
  - False sharing
  - Load imbalance

Parallel Programming for Busy Researchers

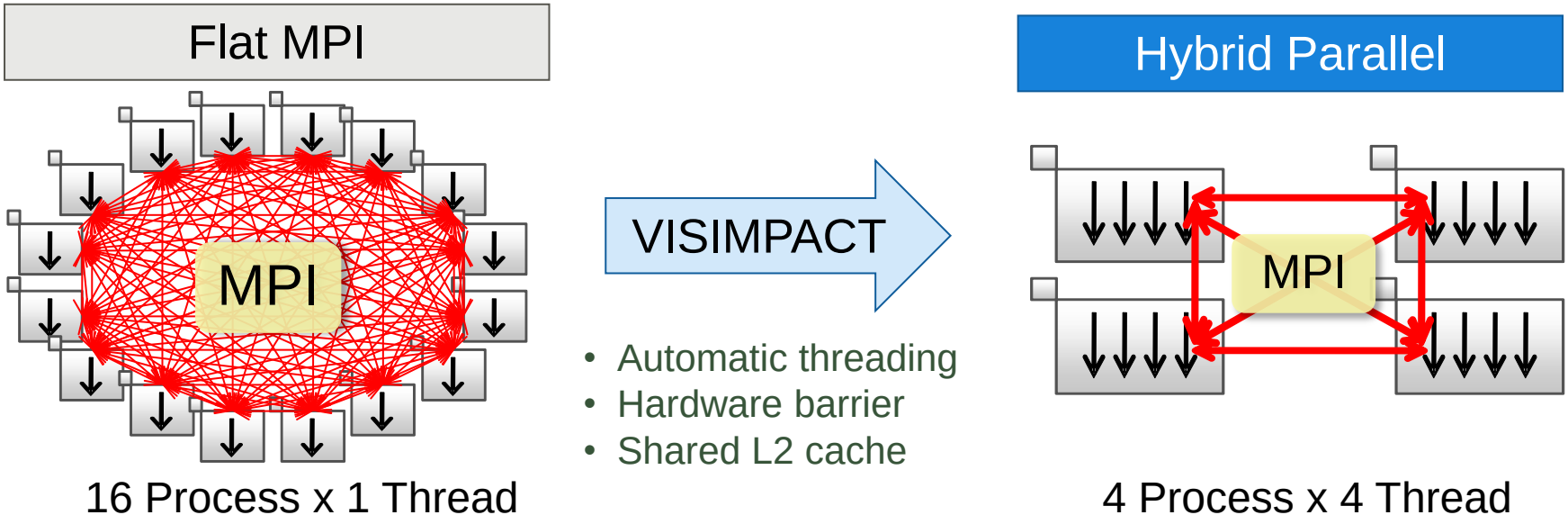
FUJITSU

- Large number of parallelism for large scale systems
  - Large number processes need large memory and overhead
    - Hybrid thread-process programming to reduce number of processes
  - Hybrid parallel programming is annoying for programmers
- Even for multi-threading, the coarser grain the better
  - Procedure level or outer loop parallelism is desired
  - Little opportunity for such coarse grain parallelism
  - System support for “fine grain” parallelism is required
- VISIMPACT solves these problems

Hybrid Parallel

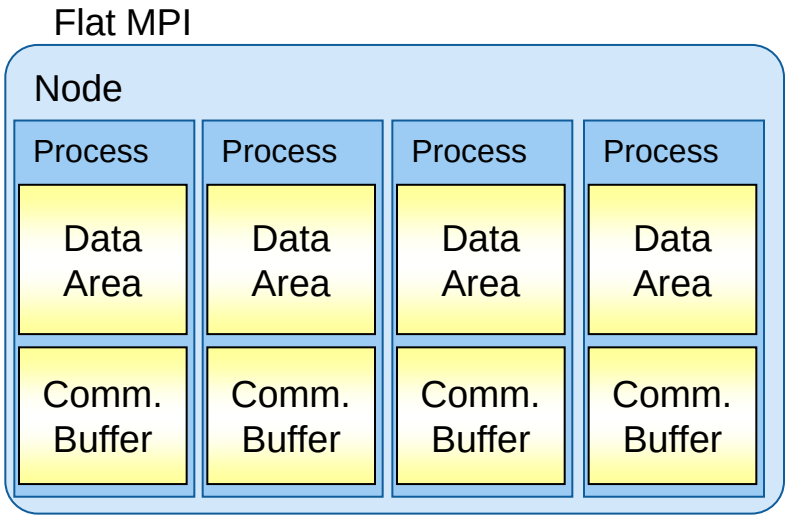
FUJITSU

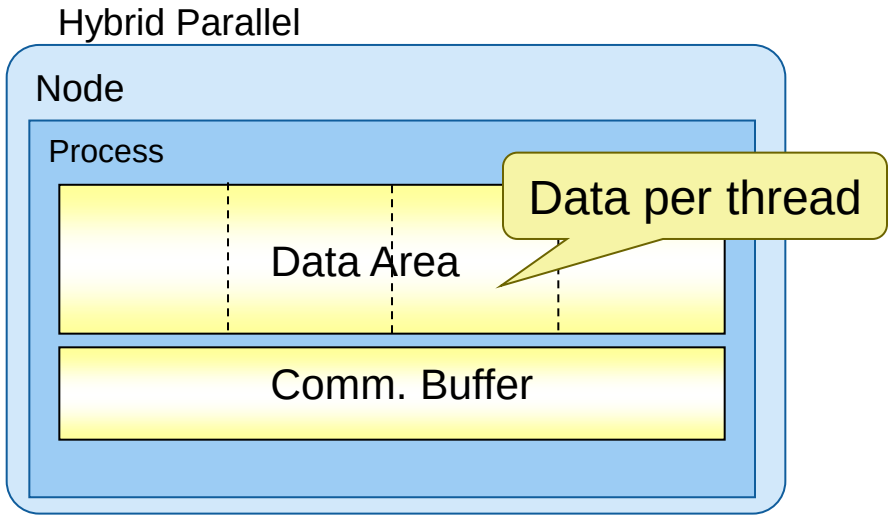
- Hybrid Parallel vs. Flat MPI
  - Hybrid Parallel: MPI parallel between CPUs
    Thread parallel inside CPU (between cores)
  - Flat MPI: MPI parallel between cores
- VISIMPACT (Virtual Single Processor by Integrated Multi-core Parallel Architecture)
  - Mechanism that treats multiple cores as one CPU through automatic parallelization
    - Hardware mechanisms to support hybrid parallel
    - Software tools to realize hybrid parallel automatically



Merits and Drawbacks of Flat MPI and Hybrid Parallel

	Flat MPI	Hybrid Parallel
Merits	Program portability	Reduced memory usage Performance ・less process = less MPI message trans. time ・thread performance can be improved by VISIMPACT
Drawbacks	Need memory ・communication buffer ・large page fragmentation MPI message passing time increase	Need two level parallelization



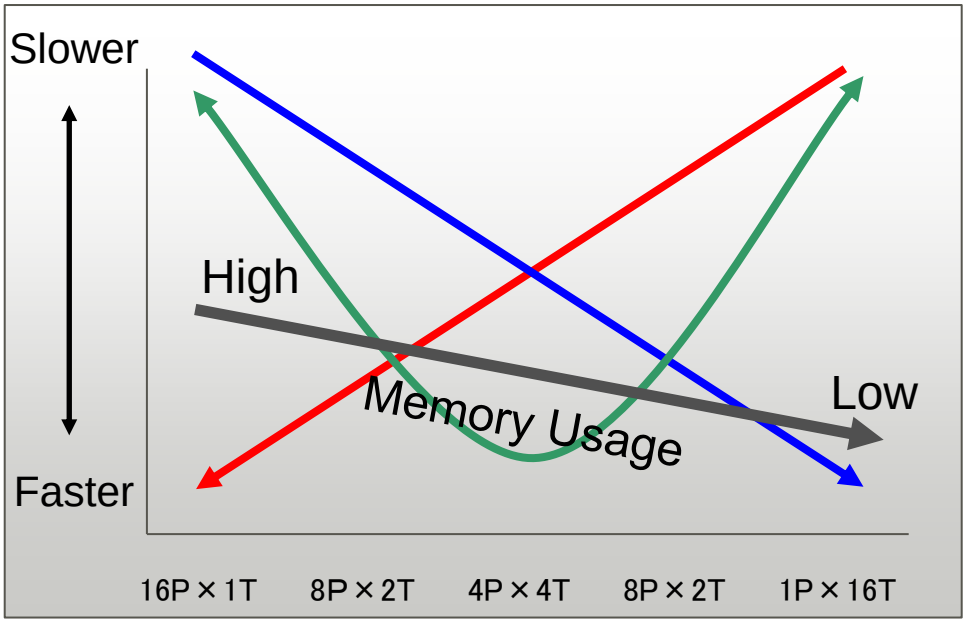


Characteristics of Hybrid Parallel

FUJITSU

- Optimum number of processes and threads depends on application characteristics.
- For example, thread parallel scalability, MPI communication ratio and process load imbalance are involved.

Conceptual Image:
performance change of different combination of process and thread

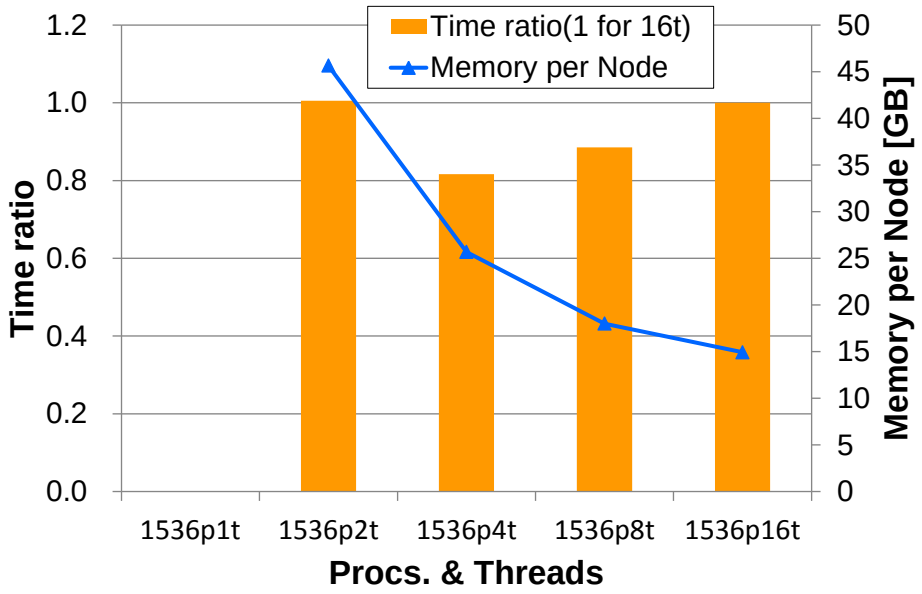


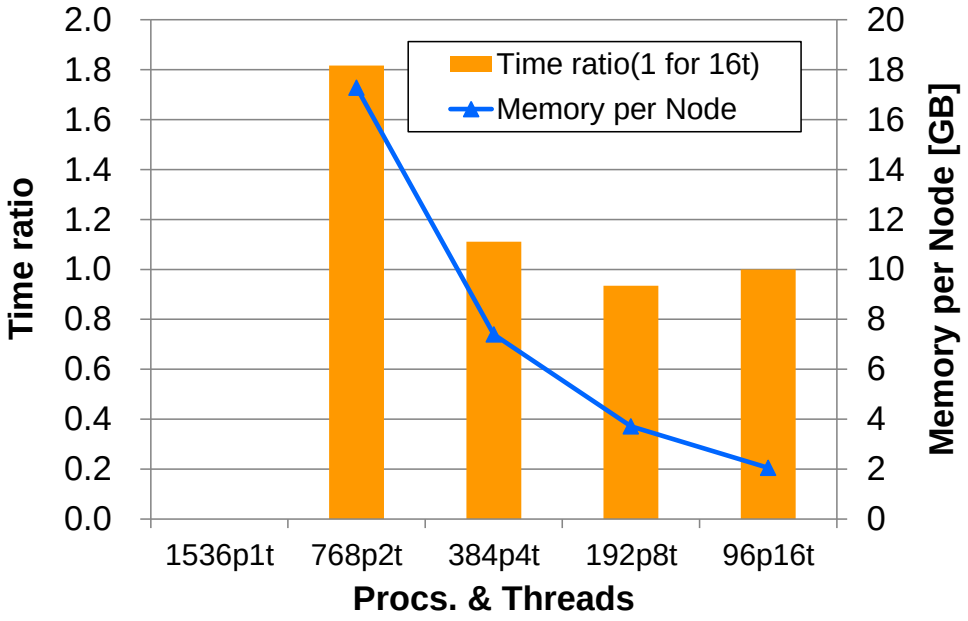
Application A
- mostly processed in parallel
- has a lot of communication
- has a big load imbalance
- has a high L2 cache reuse rate

Application B
- mostly processed in serial

Application C
- has a characteristic between A and B

Impact of Hybrid Parallel : example 1 & 2





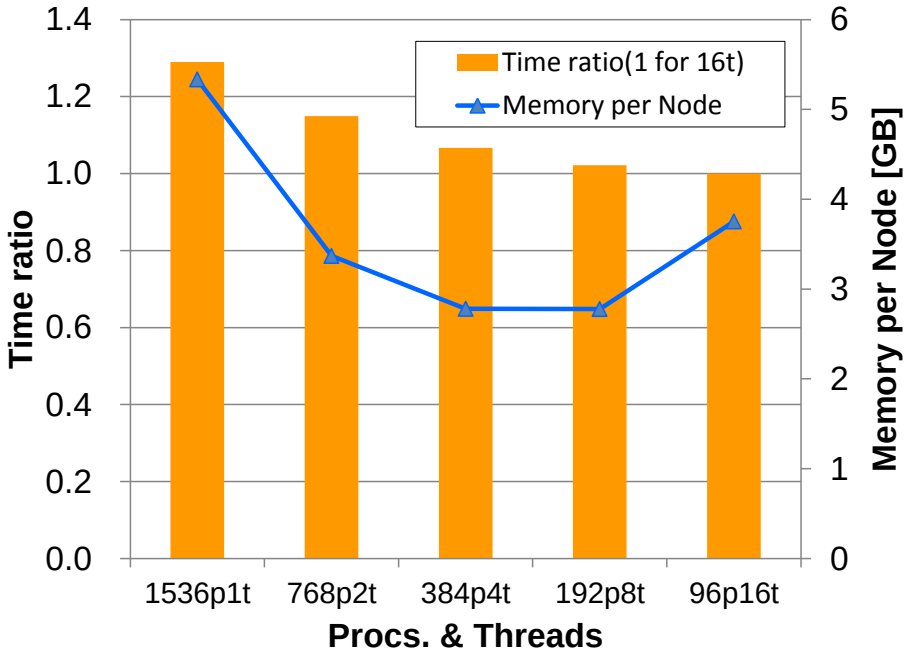
Application A

- MPI + automatic parallelize
- Meteorology application
- Flat MPI (1536 processes-1 thread) doesn't work due to memory limit
- Granularity of a thread is small

Application B

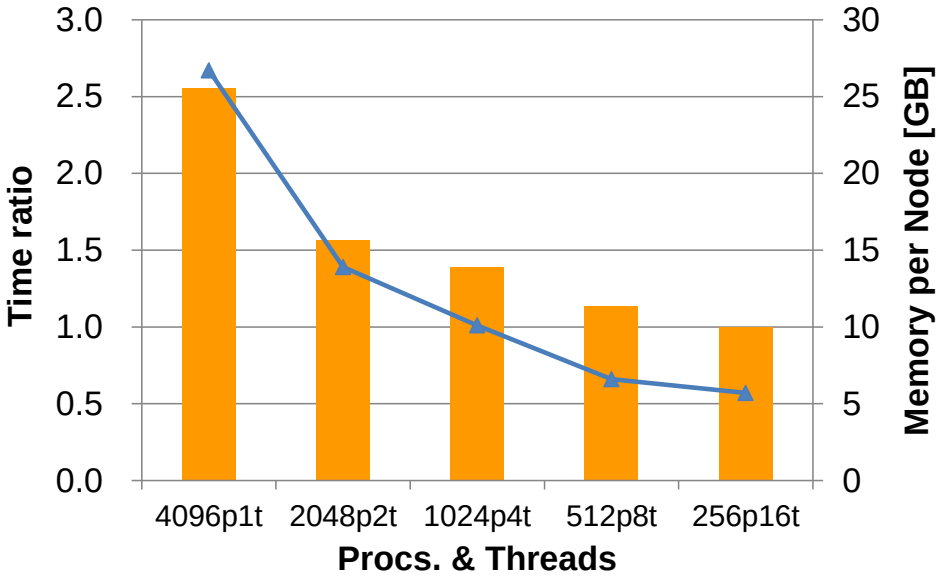
- MPI + automatic parallelize + OpenMP
- Meteorology application
- Flat MPI (1536 processes-1 thread) doesn't work due to memory limit
- Communication cost is 15%
- 72% of the process is done by thread parallel

Impact of Hybrid Parallel : example 3 & 4



Application C

- MPI + OpenMP
- Meteorology application
- Load imbalance is eased by hybrid parallel
- Communication cost is 15%
- 87% of the process is done by thread parallel



Application D

- MPI + automatic parallelize
- NPB.CG benchmark
- Load imbalance is eased by hybrid parallel
- Communication cost is 20%~30%
- 92% of the process is done by thread parallel

Application Tuning Cycle and Tools

FUJITSU

Job Information

Profiler

Vampir-trace

RMATT

Tofu-PA

MPI Tuning

Execution

Overall Tuning

CPU Tuning

Profiler

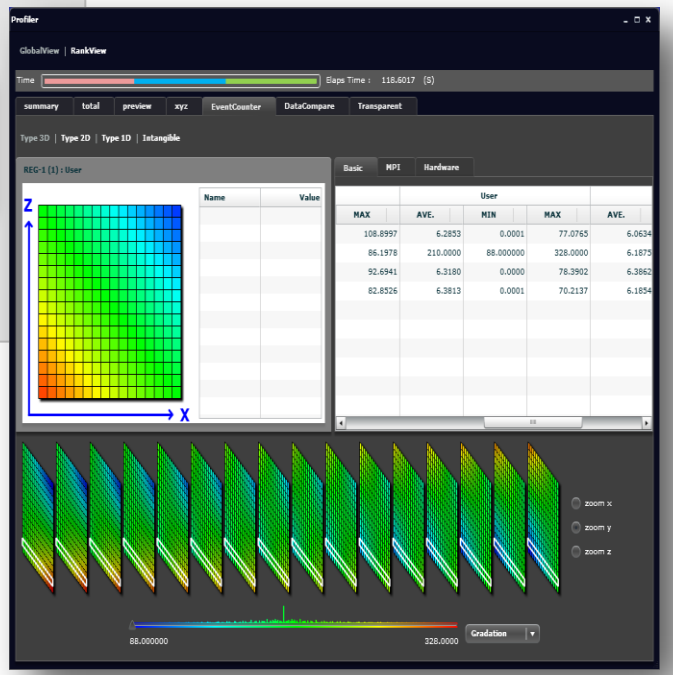
Vampir-trace

PAPI

FX10 Specific Tools

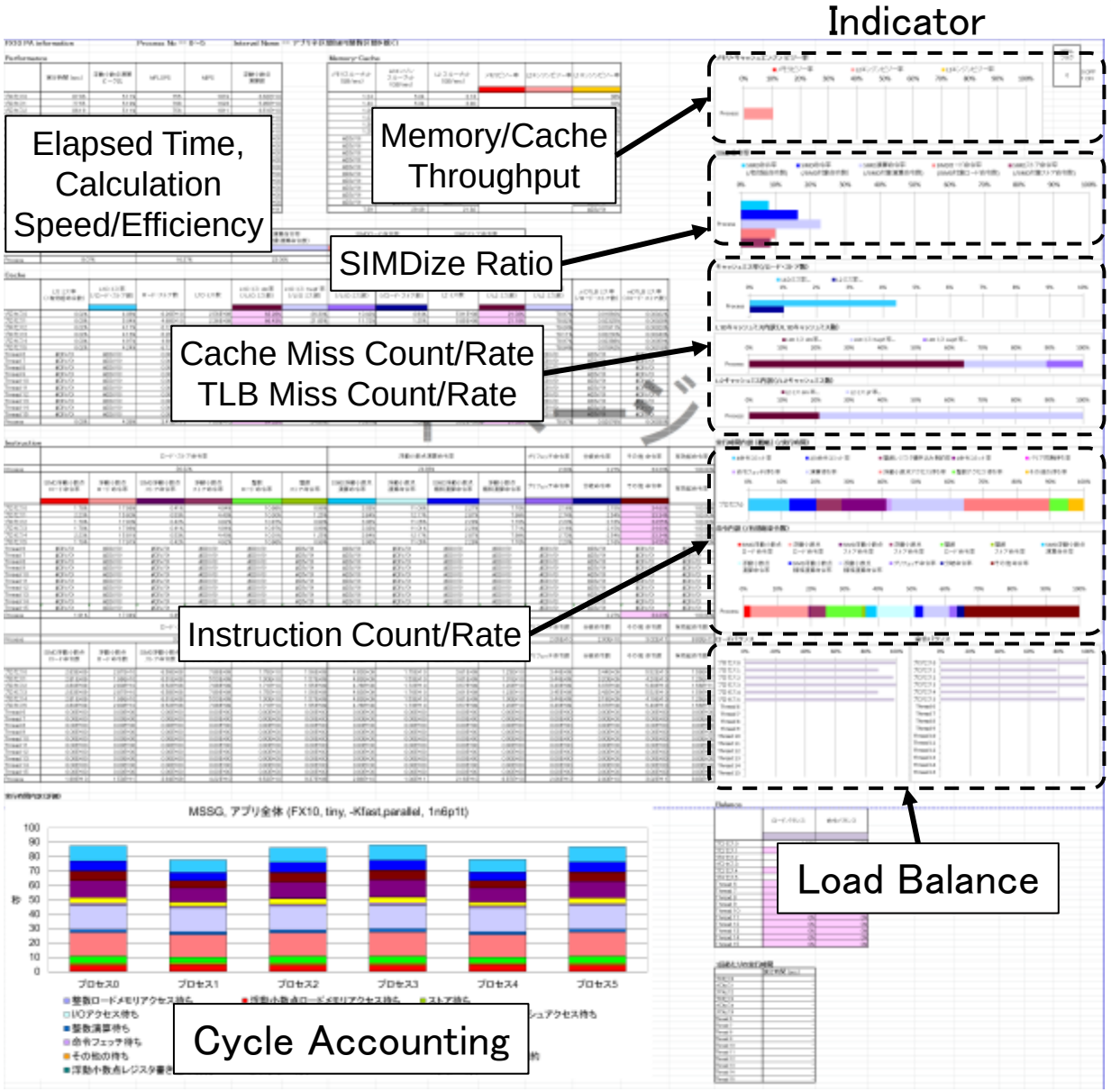
Open Source Tools

Profiler snapshot



PA(Performance Analysis) reports

- Performance Analysis reports:
  - elapsed time
  - calculation speed(FLOPS)
  - cache/memory access statistical information
  - Instruction count
  - Load balance
  - Cycle accounting
- Cycle Accounting data:
  - performance bottleneck identification
  - systematic performance tuning

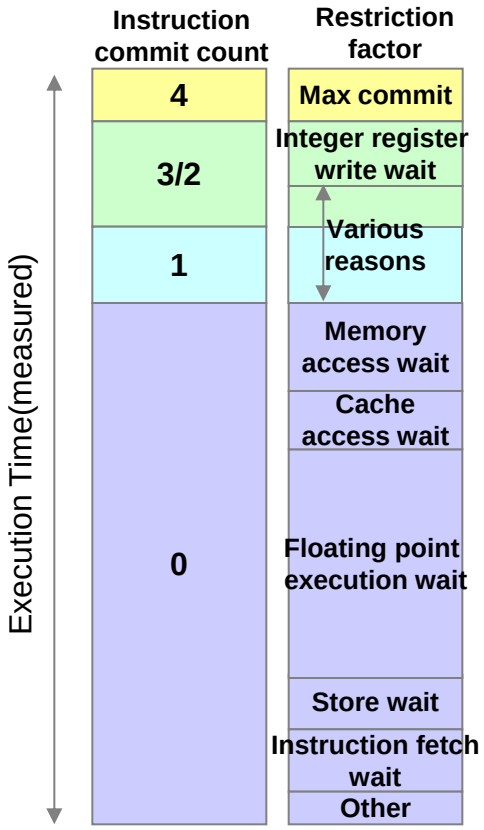


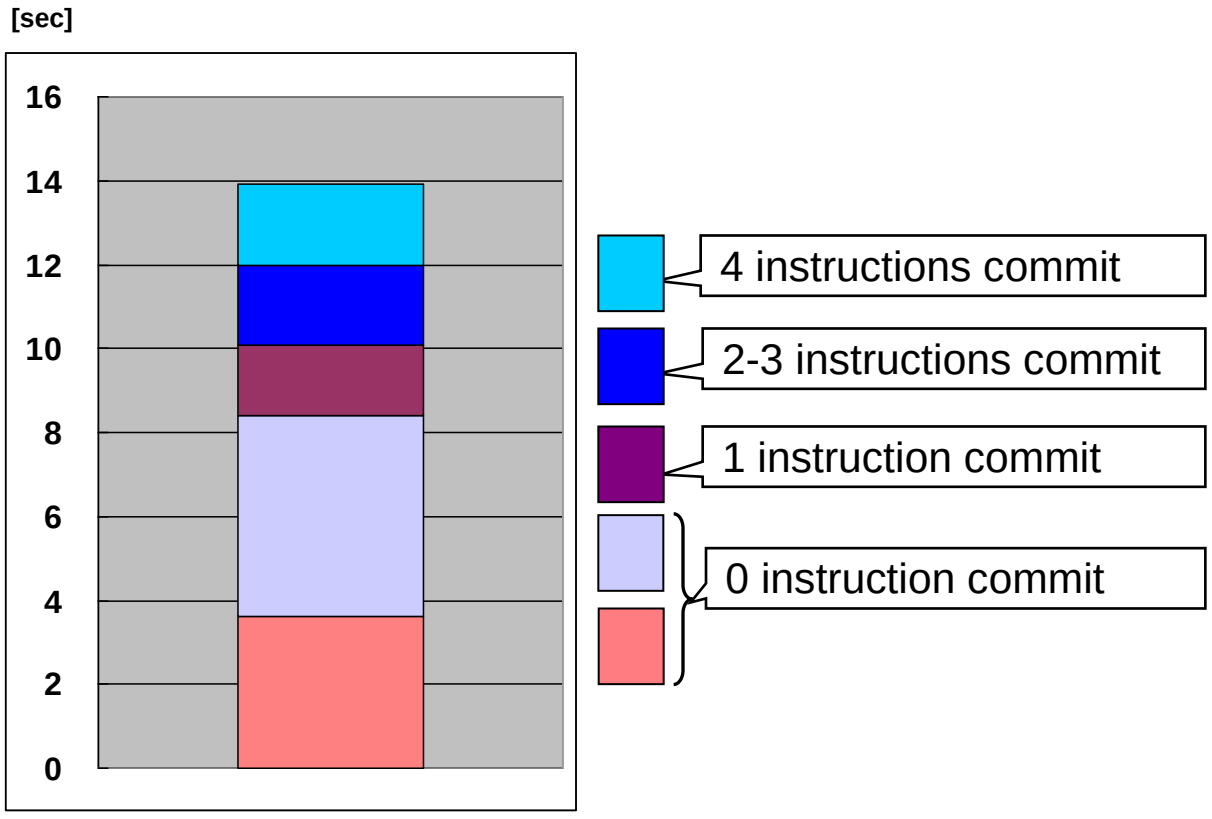
Cycle Accounting

FUJITSU

- Cycle Accounting is a technique to analyse performance bottleneck
  - SPARC64™ IXfx can measure a lot of performance analysis events
  - Summarize the execution time for each instruction commit count

Overview



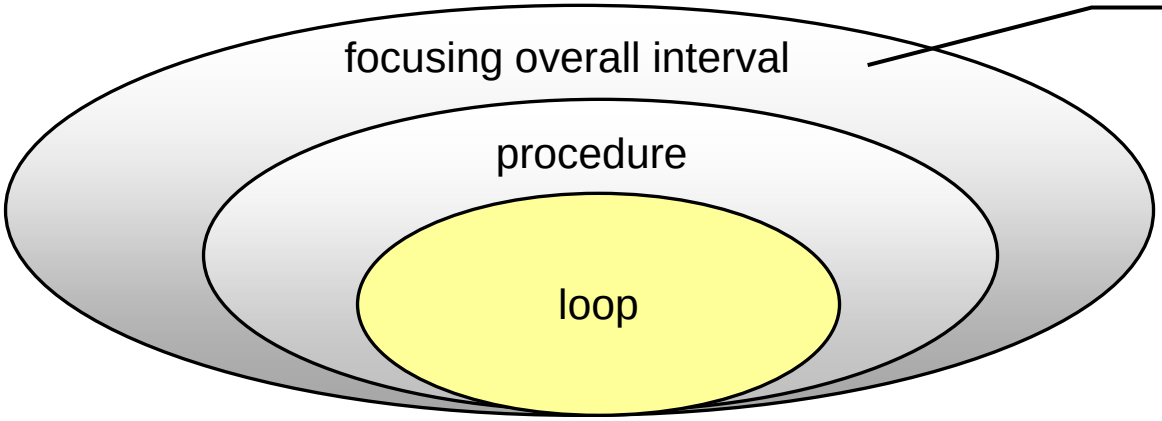


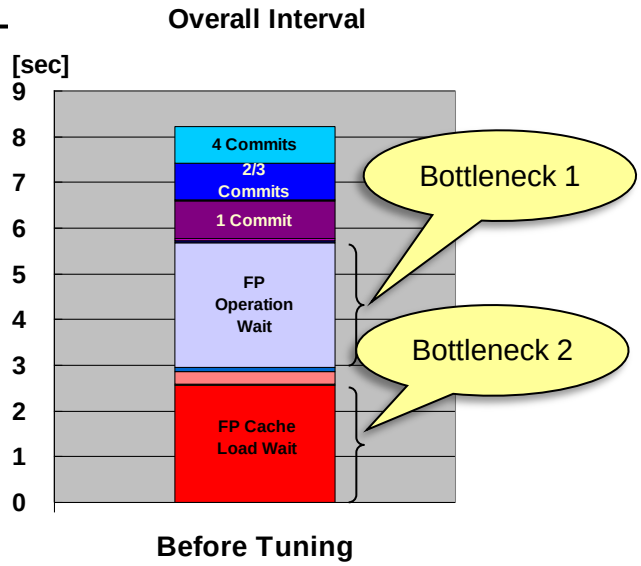
1-4 instruction(s) commit: time to execute N instruction(s) in one machine cycle
0 instruction commit: stall time due to some reason

FUJITSU

- Understanding the bottleneck

The bottleneck of a focusing interval (exclude input, output and communication) could be estimated from PA information of the overall interval





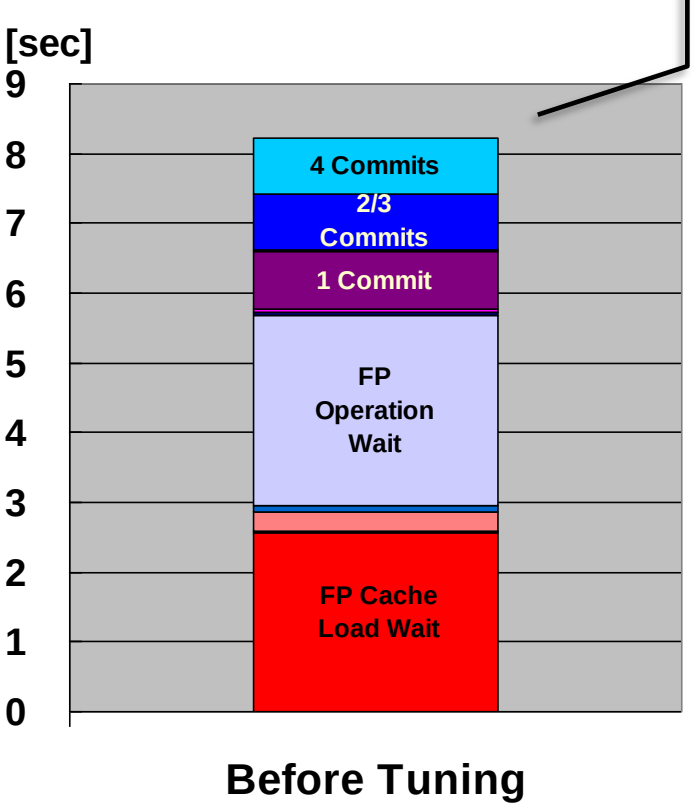
- Utilization for tuning

By breaking down to a loop level and capture the PA information of the loop, you can find out what you can do to improve the bottleneck or how far you can improve the performance

Hot spot break down

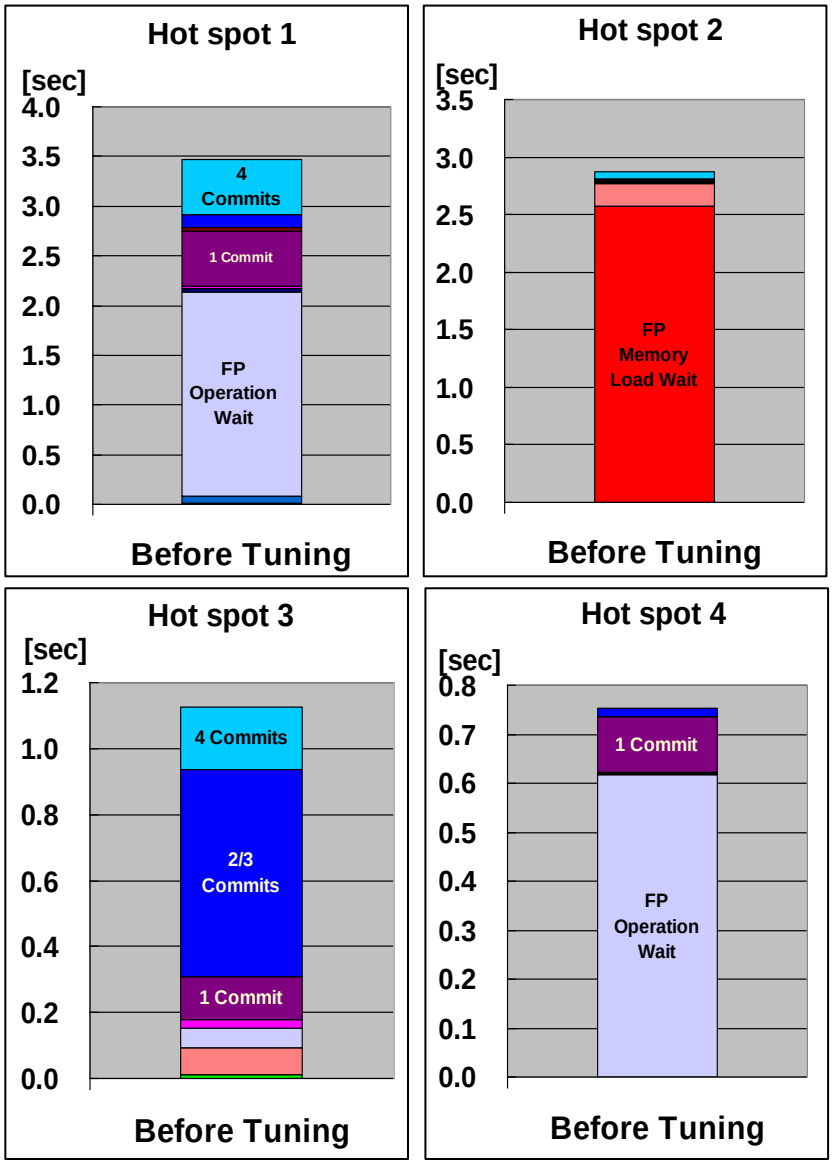
Possible to check the bottleneck level

PA graph of a whole measured interval



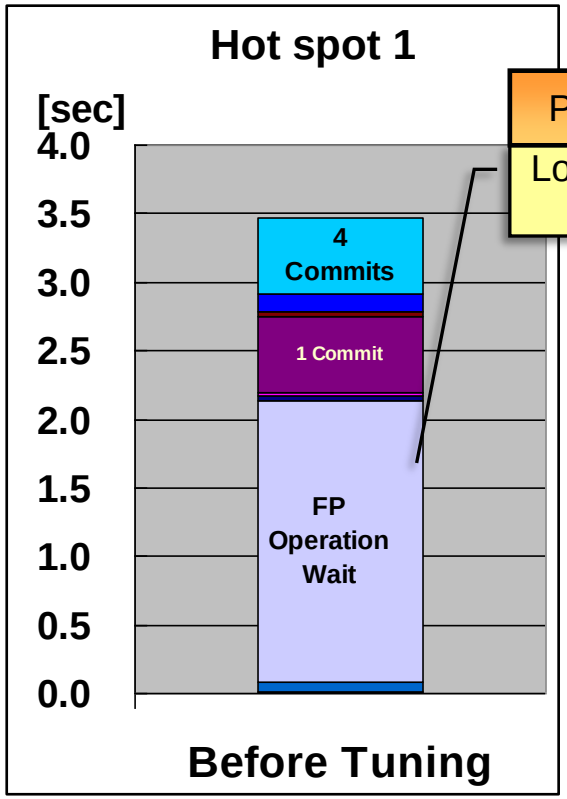
Breakdown to hot spot intervals

PA graph of different hot spots



Analyse and Diagnose(Hot spot 1: if-statement in a loop)

PA graph



Phenomenon

Long Operation Wait

Source List

```
                !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH       : 24
                <<<      a: 12, b: 12
                <<< Loop-information  End >>>
152   2  p  6s            do i=1,n1
153   3  p  6m             if (p(i) > 0.0) then
154   3  p  6s              b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
155   3             &           (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
156   3             &           (c8 + a(i)*c9))))))))
157   3  p  6v             endif
158   2  p  6v            enddo
159   1          !$omp enddo
```

Analysis

No software pipelining or SIMDizing due to inner loop if-statement

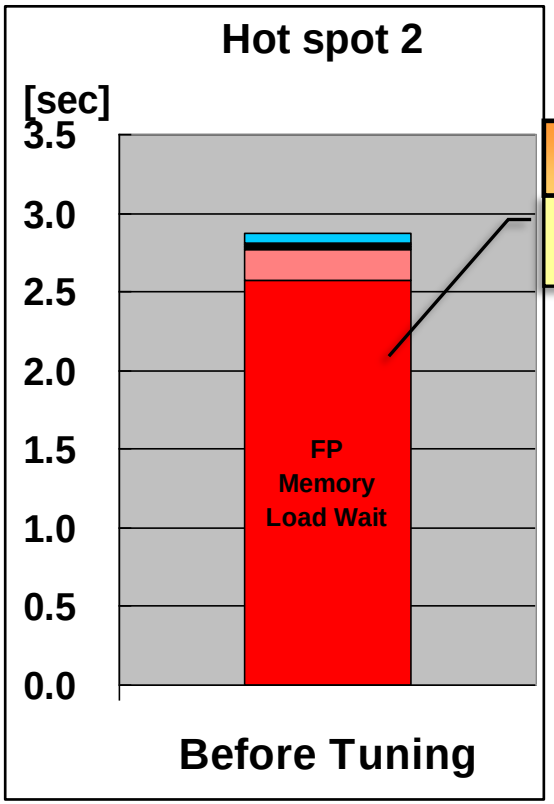
Diagnosis

Tuning required
-> Eliminate the inner loop if-statement

Analyse and Diagnose(Hot spot 2: Stride Access)

FUJITSU

PA graph



Phenomenon

Long Memory Access Wait

Source List

```
176    1          !$omp do
                     do j=1,n2
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD
                <<<    SOFTWARE PIPELINING
                <<< Loop-information  End >>>
178    3  p  6v          do i=1,n1
179    3  p  6v            b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
180    3                &          (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
181    3                &          (c8 + a(j,i)*c9))))))))
182    3  p  6v          enddo
183    2  p             enddo
184    1          !$omp enddo
```

Analysis

The accessing pattern of array ‘b’ is sequent but that of ‘a’ is stride. This reduces the cache utilization rate.

L1D miss rate	L2 miss rate
53.01%	53.04%

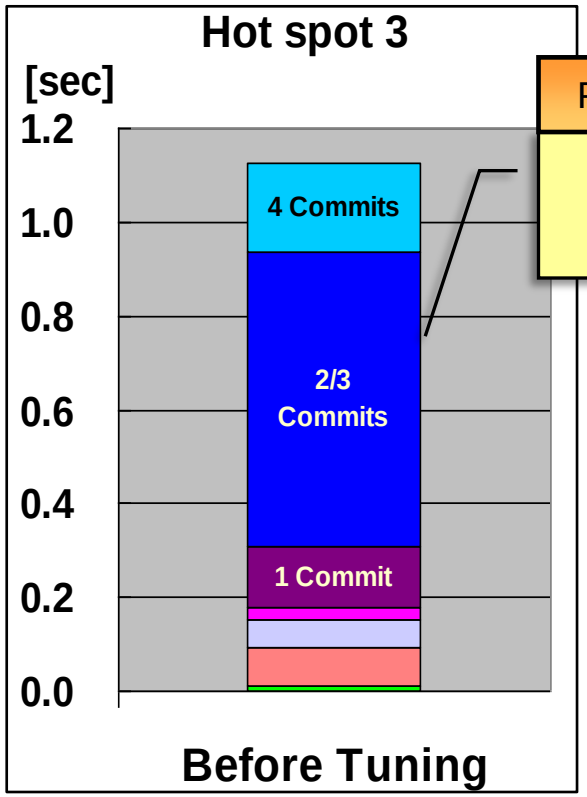
Diagnosis

Tuning required
-> Improve cache utilization rate of array ‘a’

Analyze and Diagnose(Hot spot 3: Ideal Operation)

FUJITSU

PA graph



Source List

```
                !$omp do
                   p-information Start >>>
                   PTIMIZATION]
                   MD
                <<<    SOFTWARE PIPELINING
                <<< Loop-information  End >>>
202   2  p  6v          do i=1,n1
203   2  p  6v            b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
204   2                &       (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
205   2                &       (c8 + a(i)*c9))))))))
206   2  p  6v          enddo
207   1             !$omp enddo
```

Phenomenon

Long instruction commit
Most of them are plural

Analysis

Both SIMDize ratio and SIMD multiply add instruction ratio are high

SIMDize ratio	SIMD multiply add instruction ratio
97.25%	79.53%

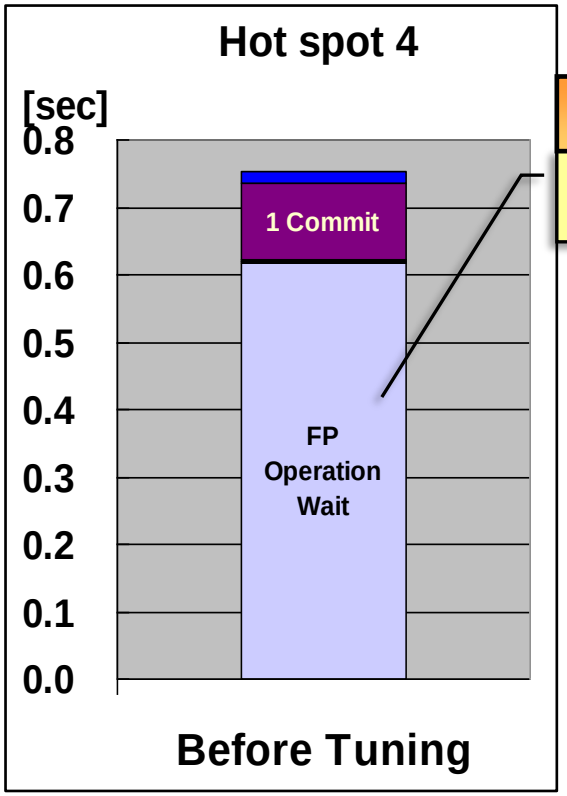
Diagnosis

Tuning NOT required
-> Instruction level parallelization is highly done

analyse and Diagnose(Hot spot 4: Data Dependency)

FUJITSU

PA graph



Source List

Phenomenon

Long Operation Wait

Analysis

Statement ‘a(i)=a(i-1)’ makes data dependency between iteration. This causes no software pipelining and no SIMDizing.

```
224   1          !$omp do
225   2  p  6s        do i=2,n1
226   2  p  6s        a(i) = c0 + a(i-1)*(c1 + a(i-1)*(c2 + a(i-1)*(c3 + a(i-1)*
227   2             &      (c4 + a(i-1)*(c5 + a(i-1)*(c6 + a(i-1)*(c7 + a(i-1)*
228   2             &      (c8 + a(i-1)*c9))))))))
229   2  p  6s        enddo
230   1          !$omp enddo
```

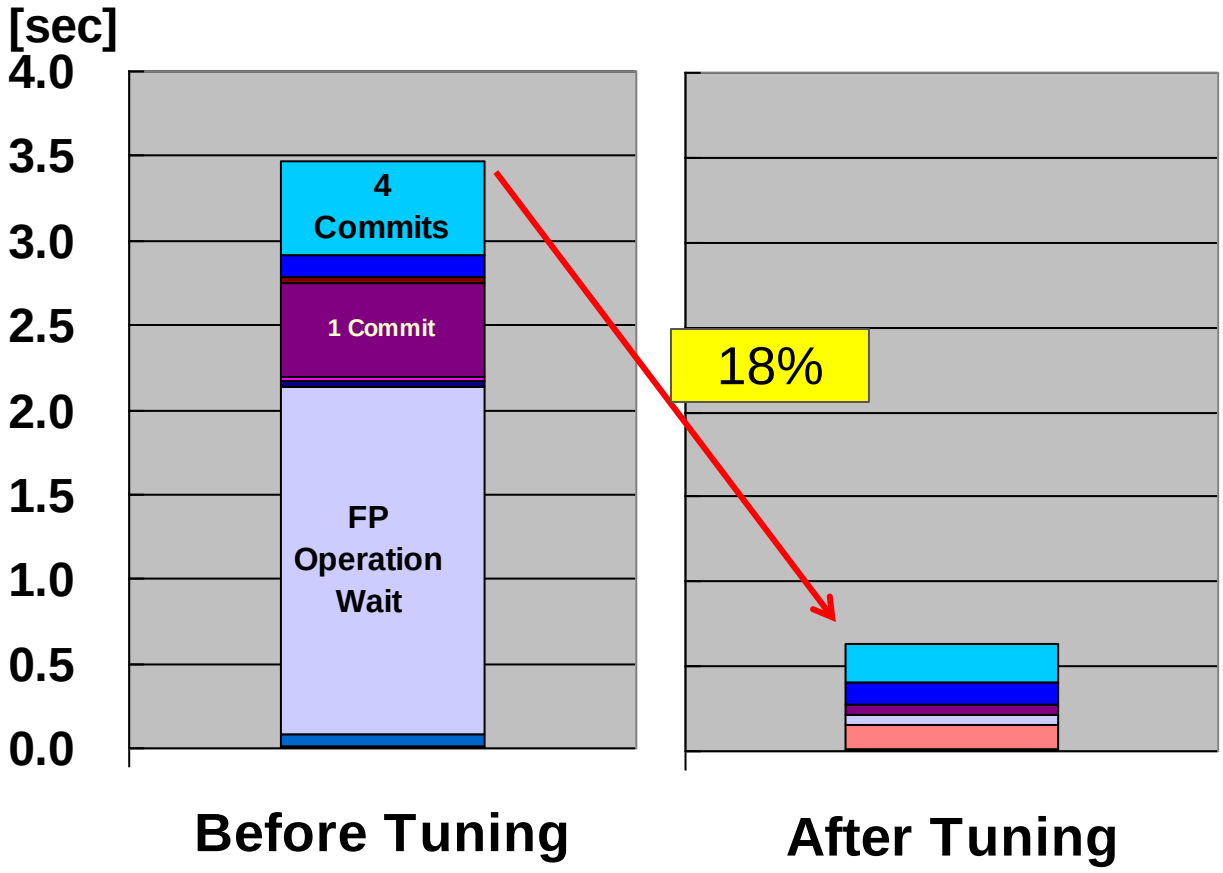
Diagnosis

No way to tune
-> Need to change the algorithm

Tuning Result (Hot spot 1): if-statement in a loop

FUJITSU

- Hot spot 1: use mask instruction instead of if-statement



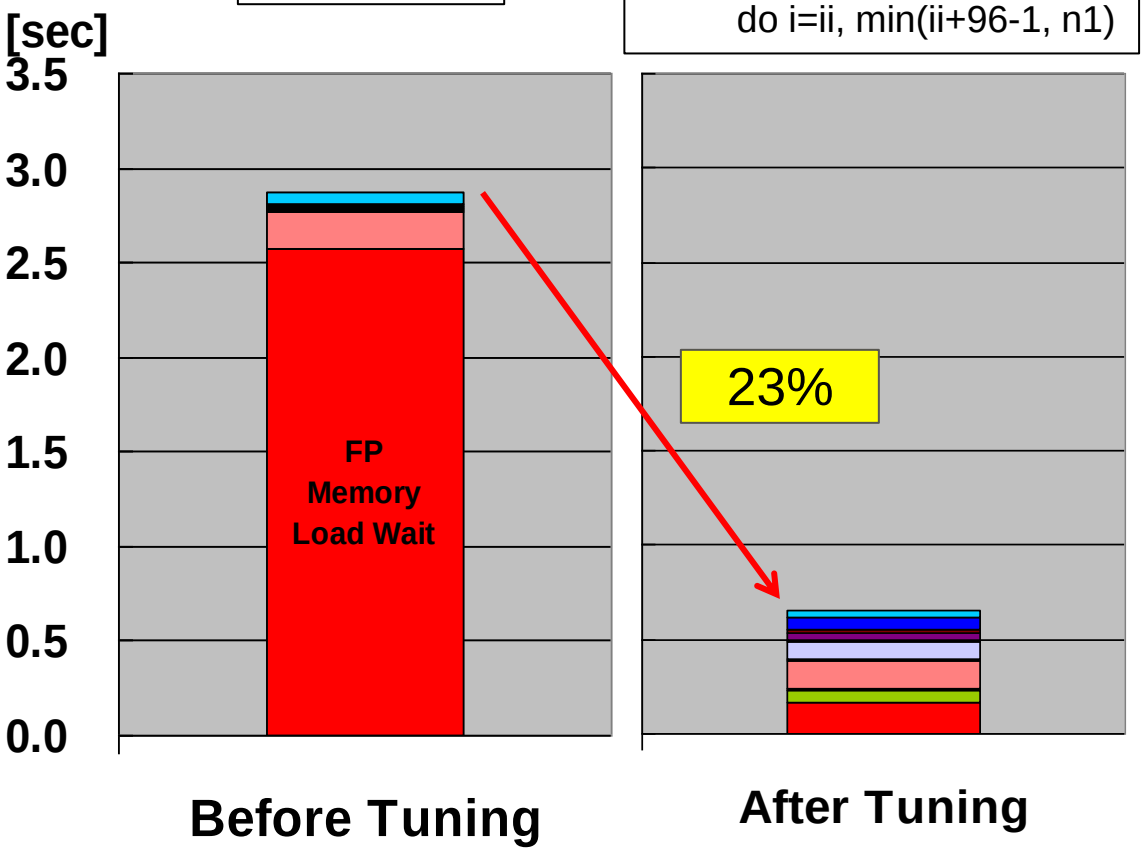
	Execution time(sec)	FP op. peak ratio	SIMD inst. ratio (/all inst.)	SIMD inst. ratio (/SIMDizable inst.)	Number of inst.
Before Tuning	3.467	9.90%	0.00%	0.00%	9.46E+10
After Tuning	0.631	60.11%	87.79%	99.98%	3.79E+10

Tuning Result (Hot spot 2): Stride Access

- Apply loop blocking (divide into blocks which fit the cache size)

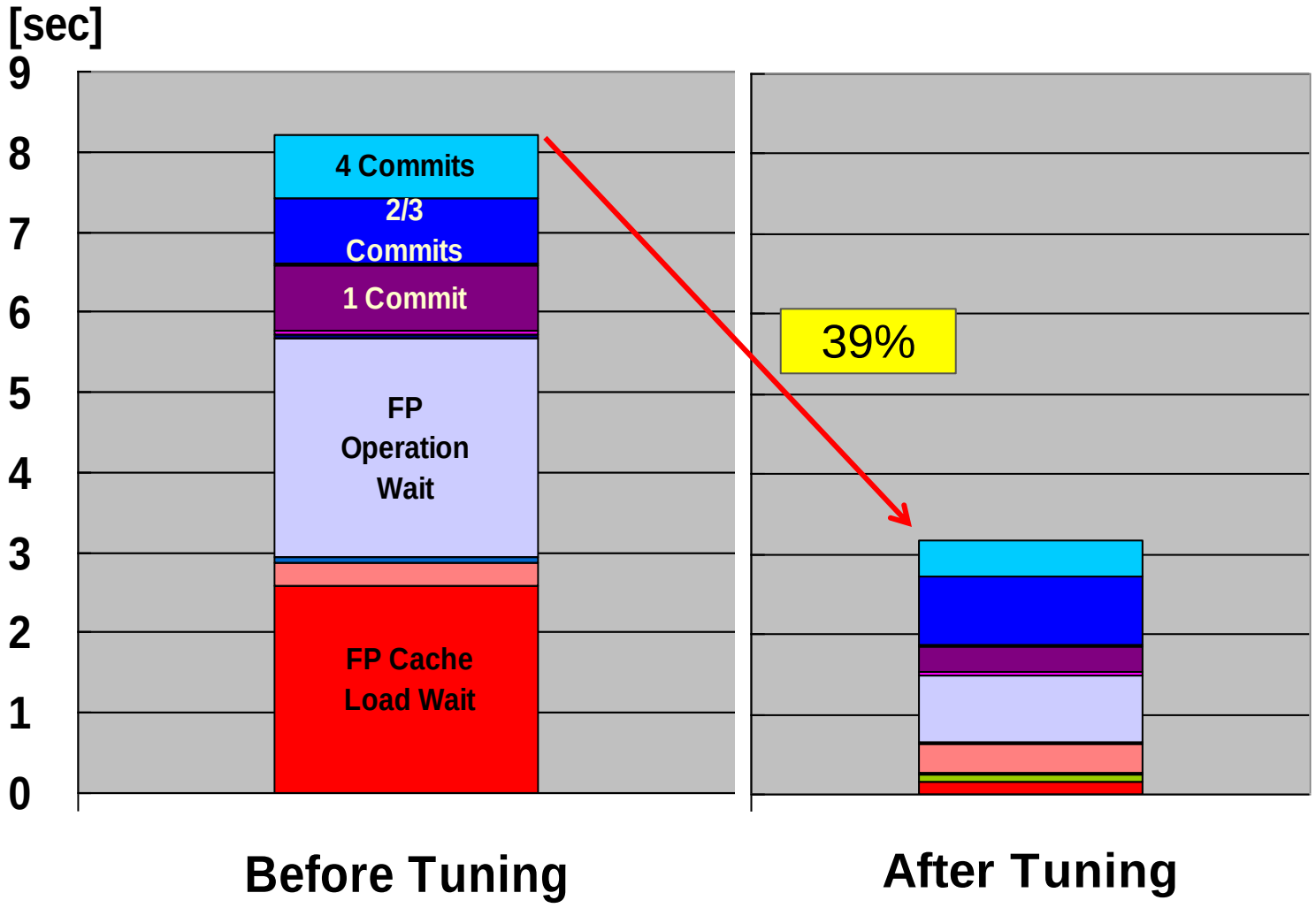
```
do j=1, n2
  do i=1, n1
```

```
do jj=1, n2, 16
  do ii=1, n1, 96
    do j=jj, min(jj+16-1, n2)
      do i=ii, min(ii+96-1, n1)
```



	Execution time(sec)	FP op. peak ratio	L1D miss ratio	L2 miss ratio
Before Tuning	2.874	3.07%	53.01%	53.04%
After Tuning	0.658	13.30%	6.69%	4.29%

Tuning Result (Overall)

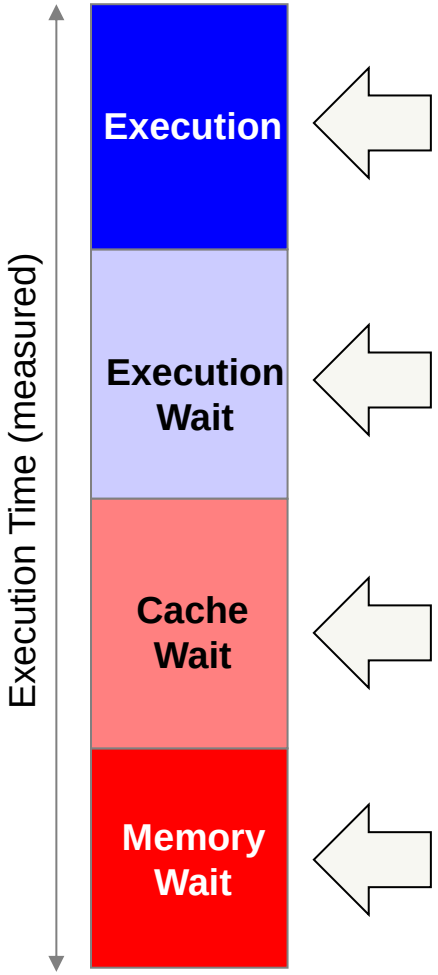


Scalar Tuning Technique

FUJITSU

- Chose tuning methods according to the cycle accounting result

Breakdown of Execution Time



Major Tuning Technique

- Instruction Reduction
  - SIMDize
  - Common subexpression elimination
- Instruction Scheduling/Software pipelining
  - Masked execution of ‘if’ statement
  - Loop unrolling
  - Loop fission
- Efficient L1 cache use
  - Padding, Loop fission, Array merge
- L2 cache latency hiding
  - L1 cache prefetch(stride/list access)
- Efficient L2 cache use
  - Loop blocking
  - Outer loop fusion, Outer loop unrolling
- Memory latency hiding
  - L2 cache prefetch(stride/list access)

Scalar Tuning Techniques

FUJITSU

Criteria	Technique	Speed Up Example
Execution	mask instruction	x1.49
	loop peeling	x2.01
	explicit data dependency hint	x1.46 x2.63
Data	loop interchange	x3.74
	loop fusion	x1.56
	loop fission	x2.69
	array merge	x2.98
	array index interchange	x2.99
	array data padding	x2.84

False Sharing

FUJITSU

Example of 4 threads

Source Program

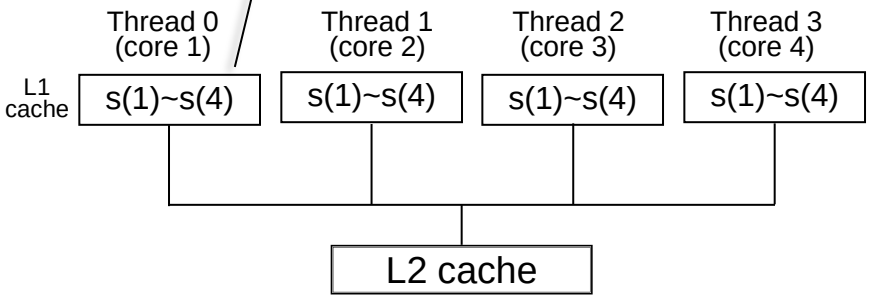
```
1                    subroutine sub(s,a,b,ni,nj)
2                    real*8 a(ni,nj),b(ni,nj)
3                    real*8 s(nj)
4
5   1  pp              do j = 1, nj
6   1  p                s(j)=0.0
7   2  p  8v             do i = 1, ni
8   2  p  8v               s(j)=s(j)+a(i,j)*b(i,j)
9   2  p  8v             end do
10  1  p              end do
11
12                   end
```

nj=4
ni=2000

◆ Initial State

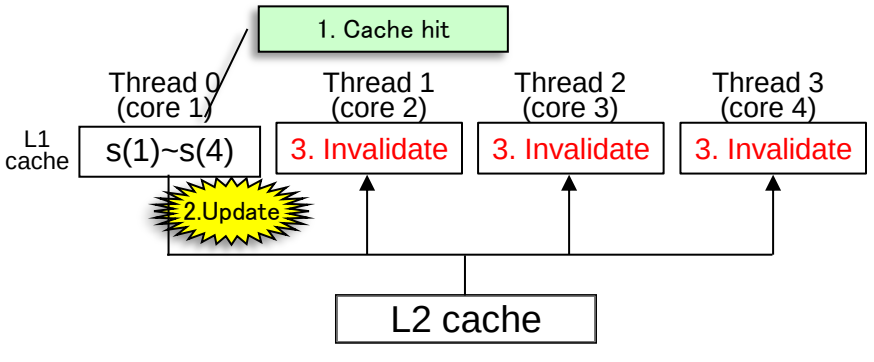
Cache holds the data by line size

Each thread reads same cache line containing s(1)~s(4)



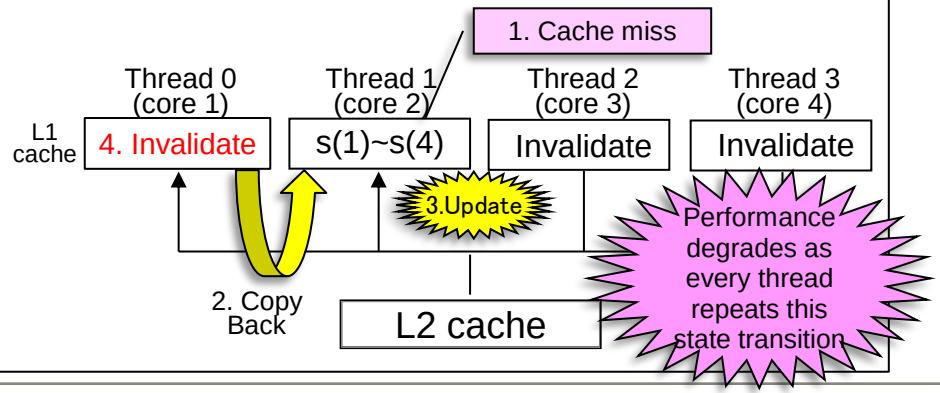
◆ Thread 0 updates s(1)

1. Cache hit
2. Thread 0 completes s(1) update
3. Invalidate the cache lines of thread 1-3 to keep the data coherent



◆ Thread 1 updates s(2)

1. Cache miss
2. Copy back cache line from thread 0 to thread 1
3. Thread 1 completes s(2) update
4. Invalidate the cache line of thread 0 to keep the data coherent



False sharing outcome (before tuning)

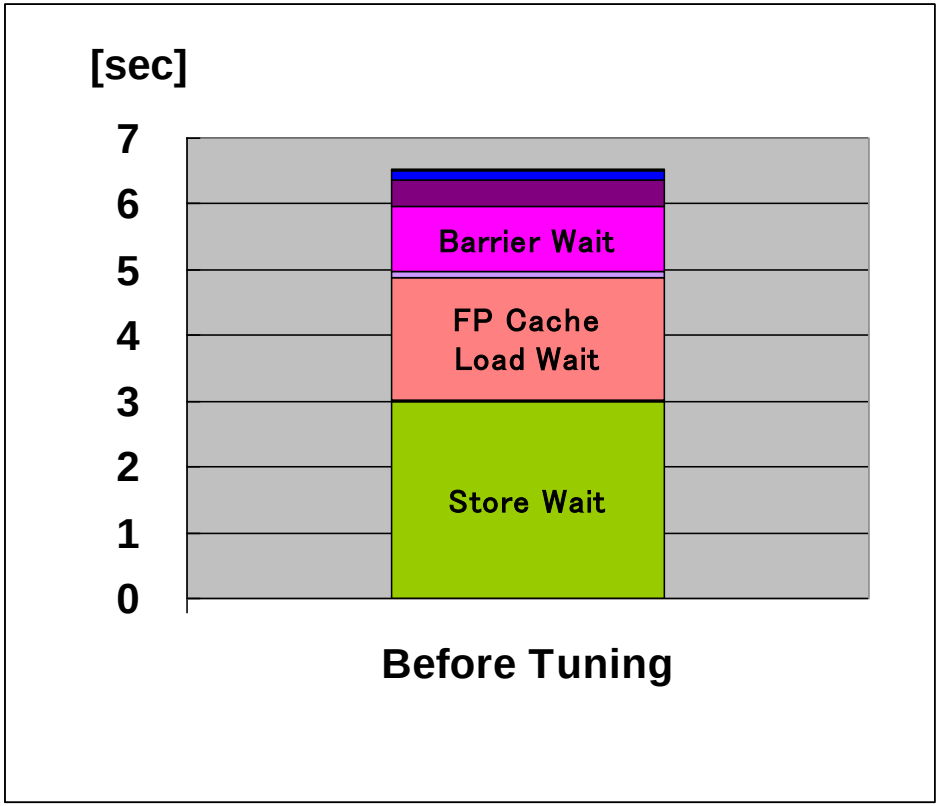
- Because the parallelized index ‘j’ runs from 1 to 8 (too small), every threads share the same cache line of array ‘a’.
  This causes a false sharing.
  PA data shows a lot of data access wait occurred.

Source code before tuning

```
20                subroutine sub()
21                integer*8 i,j,n
22                parameter(n=30000)
23                parameter(m=8)
24                real*8 a(m,n),b(n,m)
25                common /com/a,b
26
                  <<< Loop-information Start >>>
                  <<< [PARALLELIZATION]
                  <<<   Standard iteration count: 2
                  <<< Loop-information  End >>>
27    1  pp           do j=1,m
                  <<< Loop-information Start >>>
                  <<< [OPTIMIZATION]
                  <<<   SIMD
                  <<<   SOFTWARE PIPELINING
                  <<< Loop-information  End >>>
28    2  p   8v         do i=1,n
29    2  p   8v           a(j,i)=b(i,j)
30    2  p   8v         enddo
31    1  p            enddo
32
33  End
```

parallelized index runs only 8

False sharing occurs here



	L1D miss ratio
Before tuning	29.53%

False sharing tuned

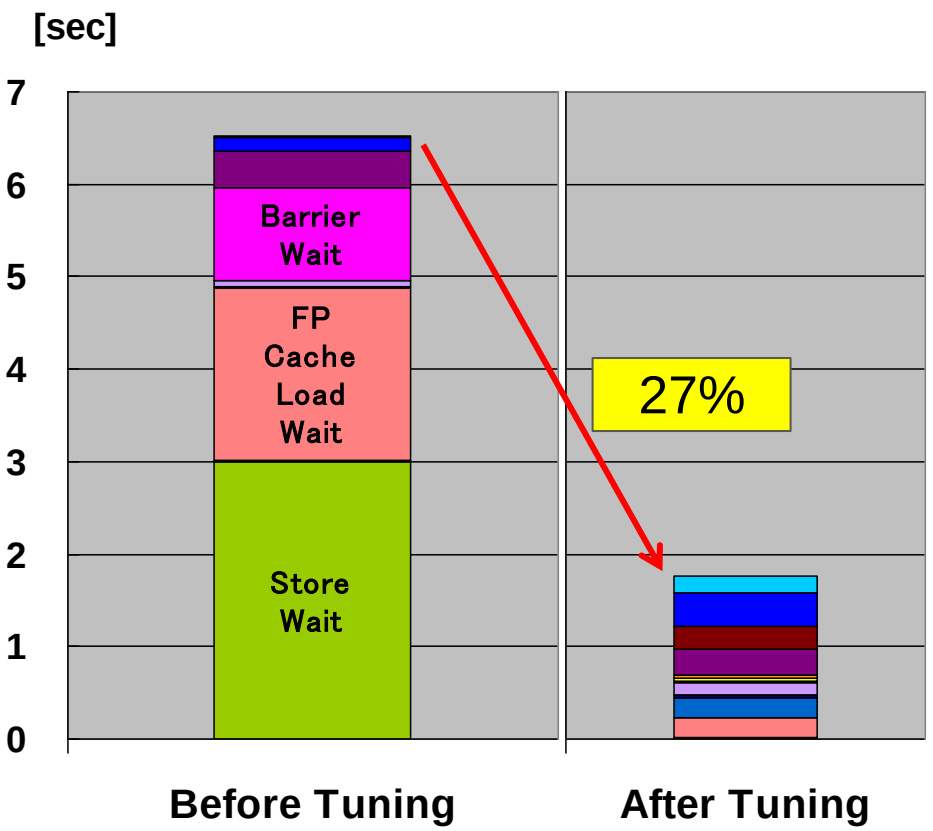
- By doing loop interchange, the false sharing can be avoided.
  This reduces L1 cache miss and improve the data access wait.

Source code after tuning (source level tuning)

```
20                subroutine sub()
21                integer*8 i,j,n
22                parameter(n=30000)
23                parameter(m=8)
24                real*8 a(m,n),b(n,m)
25                common /com/a,b
26
              <<< Loop-information Start >>>
              <<< [PARALLELIZATION]
              <<<   Standard iteration count: 2
              <<< [OPTIMIZATION]
              <<<   PREFETCH      : 4
              <<<     b: 2, a: 2
              <<< Loop-information  End >>>
27    1  pp                 do i=1,n
              <<< Loop-information Start >>>
              <<< [OPTIMIZATION]
              <<<   SIMD
              <<<   SOFTWARE PIPELINING
              <<< Loop-information  End >>>
28    2  p   8v               do j=1,m
29    2  p   8v                 a(j,i)=b(i,j)
30    2  p   8v               enddo
31    1  p                  enddo
32
33    End
```

Parallelize the second index by loop interchange

Avoid false sharing



	L1D miss ratio
After tuning	7.89%

Triangular loop

- Triangular loop is a loop that has an inner loop initial index value driven by the outer loop index variable. If you divide this loop into blocks and run in parallel, you will get a load imbalance.

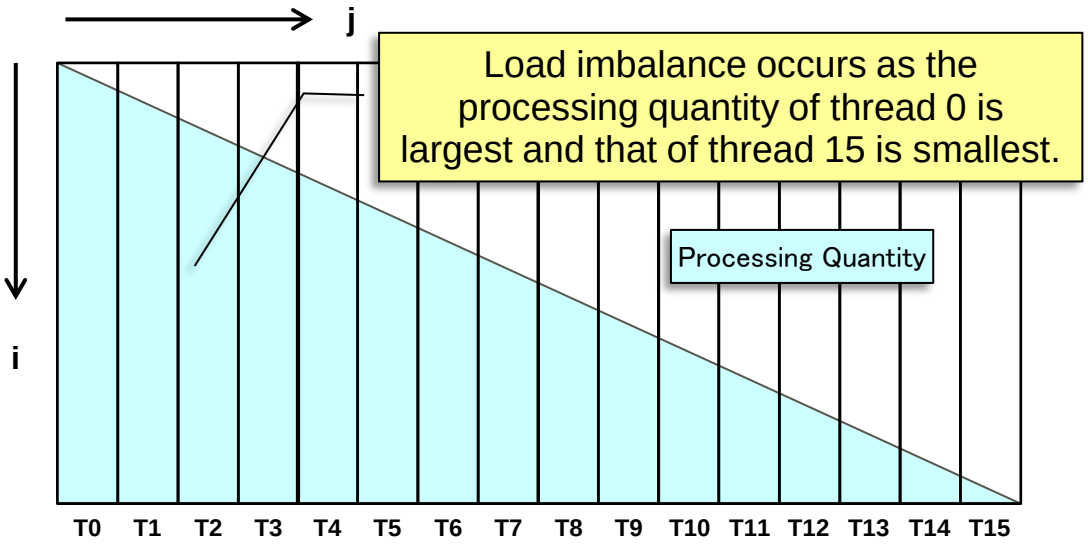
Example

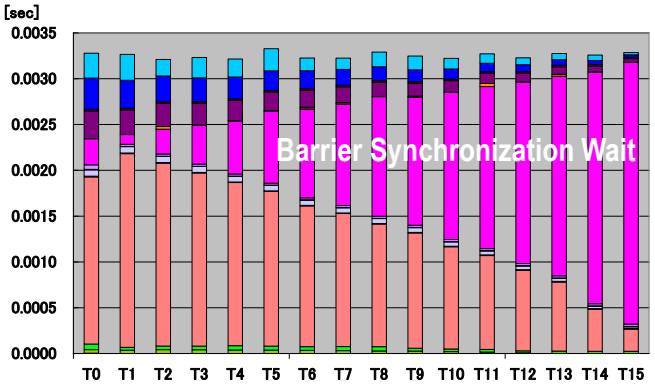
```
subroutine sub()
    integer*8 i,j,n
    parameter(n=512)
    real*8 a(n+1,n),b(n+1,n),c(n+1,n)
    common a,b,c

!$omp parallel do
      do j=1,n
        do i=j,n
          a(i,j)=b(i,j)+c(i,j)
        enddo
      enddo

    end
```

The initial index value of inner loop is determined by the outer loop variable.



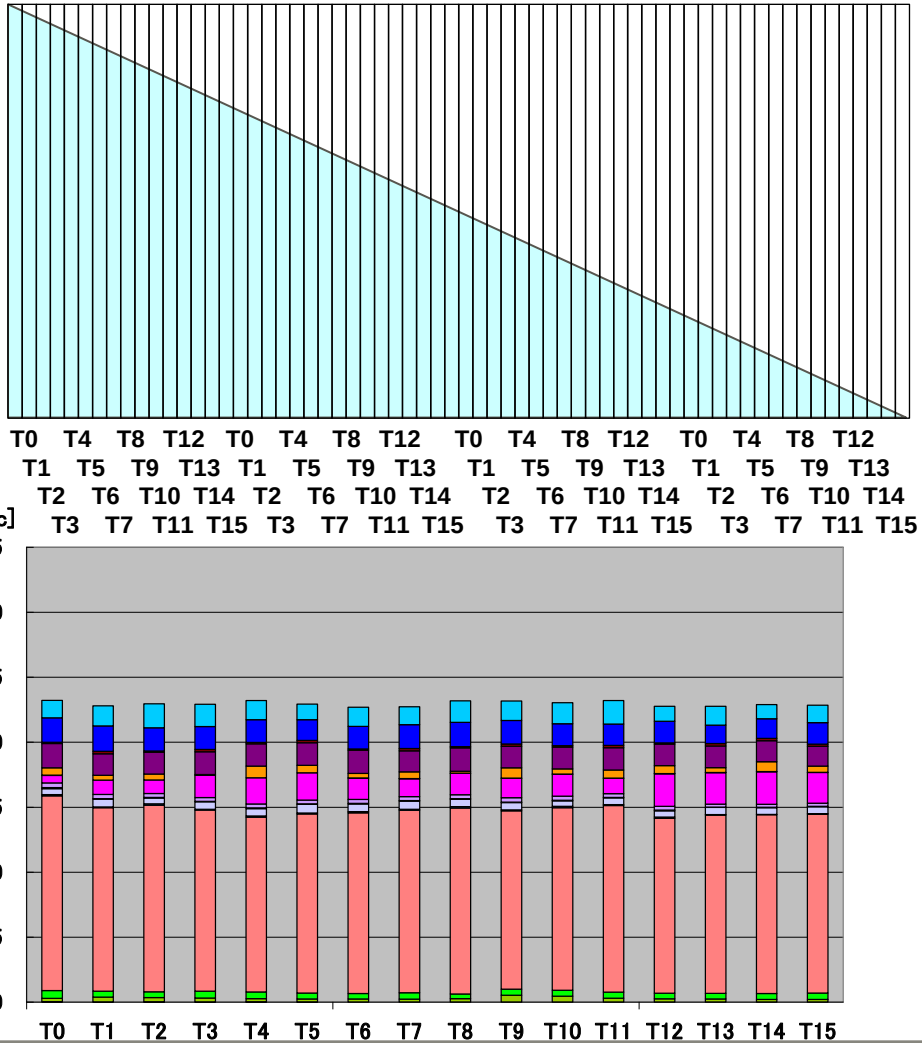


Triangular loop load imbalance tuning

- By adding openMP directive, schedule(static, 1), loop size becomes small and loops are assigned to each thread in cyclic manner. This assignes almost same job quantity to each thread and reduces load imbalance.

Modified code

```
28                      subroutine sub()
29                      integer*8 i,j,n
30                      parameter(n=512)
31                      real*8 a(n+1,n),b(n+1,n),c(n+1,n)
32                      common a,b,c
33
34                !$omp parallel do schedule(static,1)
35    1   p               do j=1,n
36    2   p   8v            do i=j,n
37    2   p   8v              a(i,j)=b(i,j)+c(i,j)
38    2   p   8v            enddo
39    1   p               enddo
40
41                      end
```



Loop with if-statement

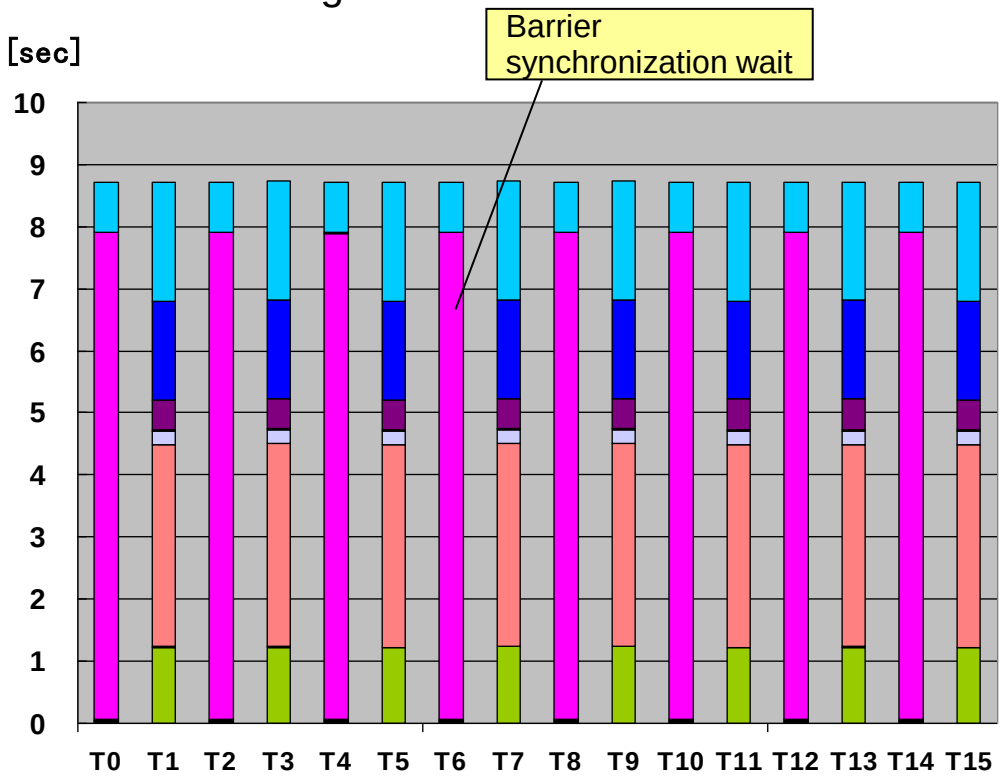
- When the processing quantity of each thread is different, say the loop contains an if-statement, load imbalance can NOT be resolved by a static cyclic divide.

Example

```
 1                 subroutine sub(a,b,s,n,m)
 2                  real a(n),b(n),s
 3                 !$omp parallel do schedule(static,1)
 4    1   p           do j=1,n
 5    2   p             if( mod(j,2) .eq. 0 ) then
 6    3   p   8v          do i=1,m
 7    3   p   8v            a(i) = a(i)*b(i)*s
 8    3   p   8v          enddo
 9    2   p             endif
10    1   p           enddo
11                end subroutine sub
 :
21                 program main
22                  parameter(n=1000000)
23                  parameter(m=100000)
24                  real a(n),b(n)
25                  call init(a,b,n)
26                  call sub(a,b,2.0,n,m)
27                end program main
```

Only odd threads execute 'then' clause

Before Tuning



Using dynamic scheduling to reduce load imbalance

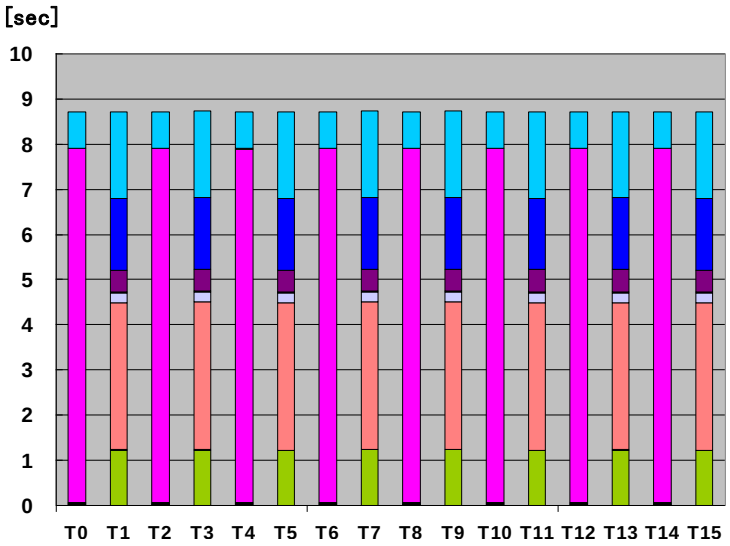
FUJITSU

- By changing the thread schedule method to dynamic, a thread which finishes its execution earlier can execute the next iteration. This reduces the load imbalance.

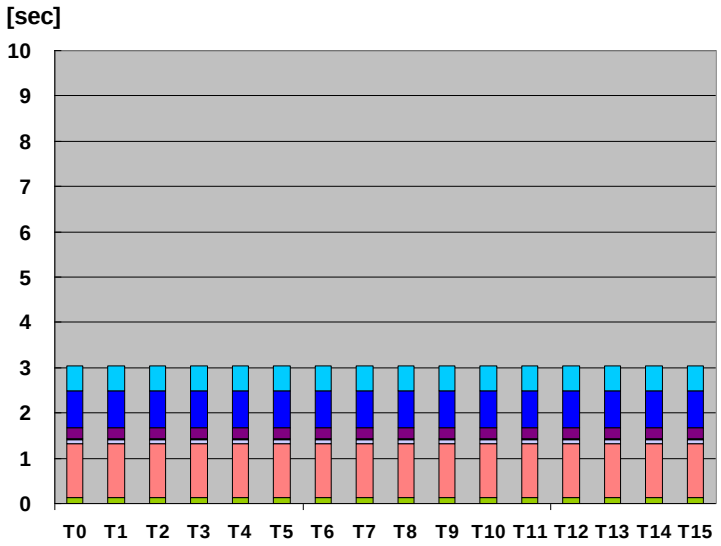
Modified code

```
 1                 subroutine sub(a,b,s,n,m)
 2                   real a(n),b(n),s
 3                 !$omp parallel do schedule(dynamic,1)
 4    1   p            do j=1,n
 5    2   p              if( mod(j,2) .eq. 0 ) then
 6    3   p   8v           do i=1,m
 7    3   p   8v             a(i) = a(i)*b(i)*s
 8    3   p   8v           enddo
 9    2   p              endif
10    1   p            enddo
11                  end subroutine sub
 :
21                  program main
22                    parameter(n=1000000)
23                    parameter(m=100000)
24                    real a(n),b(n)
25                    call init(a,b,n)
26                    call sub(a,b,2.0,n,m)
27                  end program main
```

Before Tuning



After Tuning



Choosing a right parallelize loop

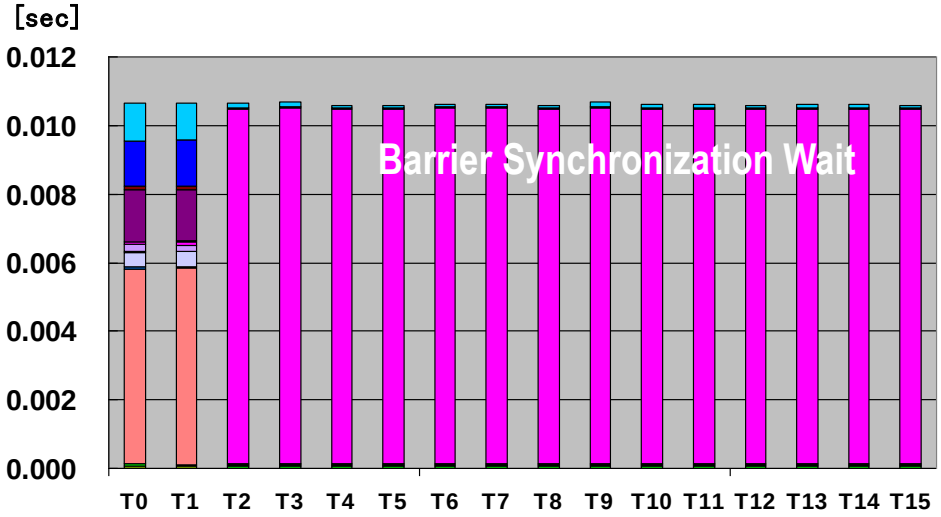
FUJITSU

- If the iteration count is too small to parallelize, a load imbalance occurs.

Example

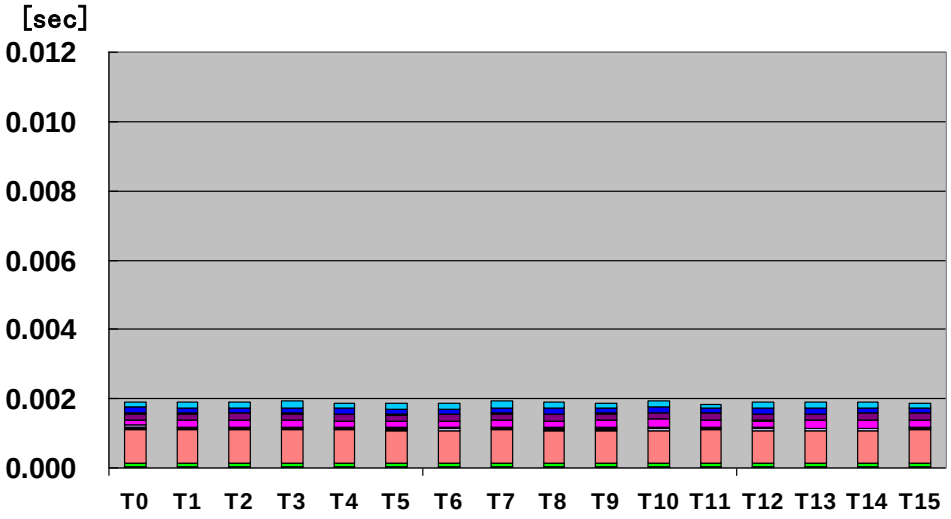
```
34   1  pp        do k=1,l
35   2  p          do j=1,m
36   3  p  8v        do i=1,n
37   3  p  8v          a(i,j,k)=b(i,j,k)+c(i,j,k)
38   3  p  8v        enddo
39   2  p          enddo
40   1  p        enddo
```

l=2
m=256
n=256



Modified code

```
33              !ocl serial
34   1          do k=1,l
35   1          !ocl parallel
36   2  pp        do j=1,m
37   3  p  8v      do i=1,n
38   3  p  8v        a(i,j,k)=b(i,j,k)+c(i,j,k)
39   3  p  8v      enddo
40   2  p        enddo
41   1          enddo
```



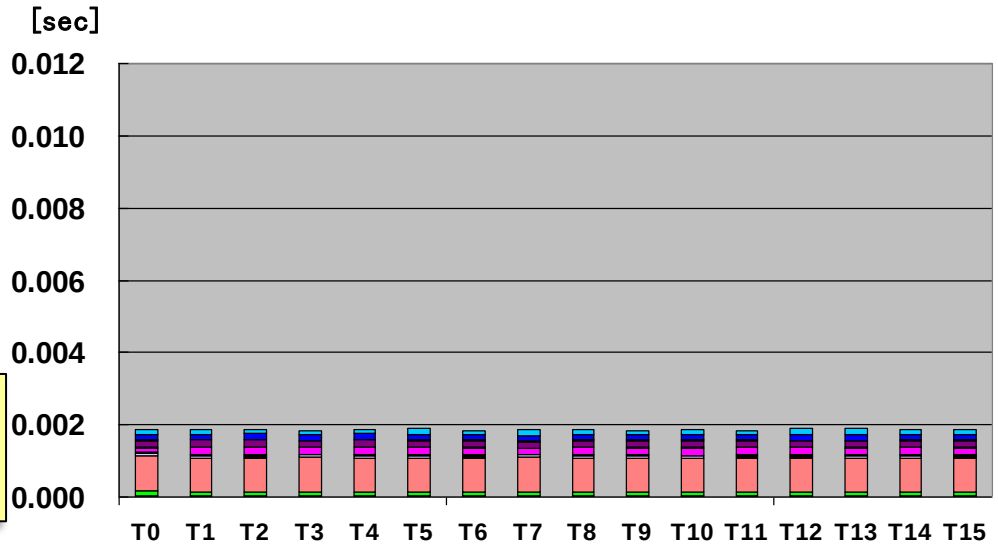
Compiler option to choose a right iteration

- By using a compiler option –Kdynamic_iteration, an appropriate iteration is chosen at runtime and the load imbalance is reduced.

Example

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
34    1  pp          do k=1,l
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 4
                <<< Loop-information  End >>>
35    2  pp            do j=1,m
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 728
                <<<  [OPTIMIZATION]
                <<<    SIMD
                <<<    SOFTWARE PIPELINING
                <<< Loop-information  End >>>
36    3  pp  8v           do i=1,n
37    3   p  8v             a(i,j,k)=b(i,j,k)+c(i,j,k)
38    3   p  8v           enddo
39    2   p             enddo
40    1   p           enddo
41
42              end
```

l=2
m=256
n=256

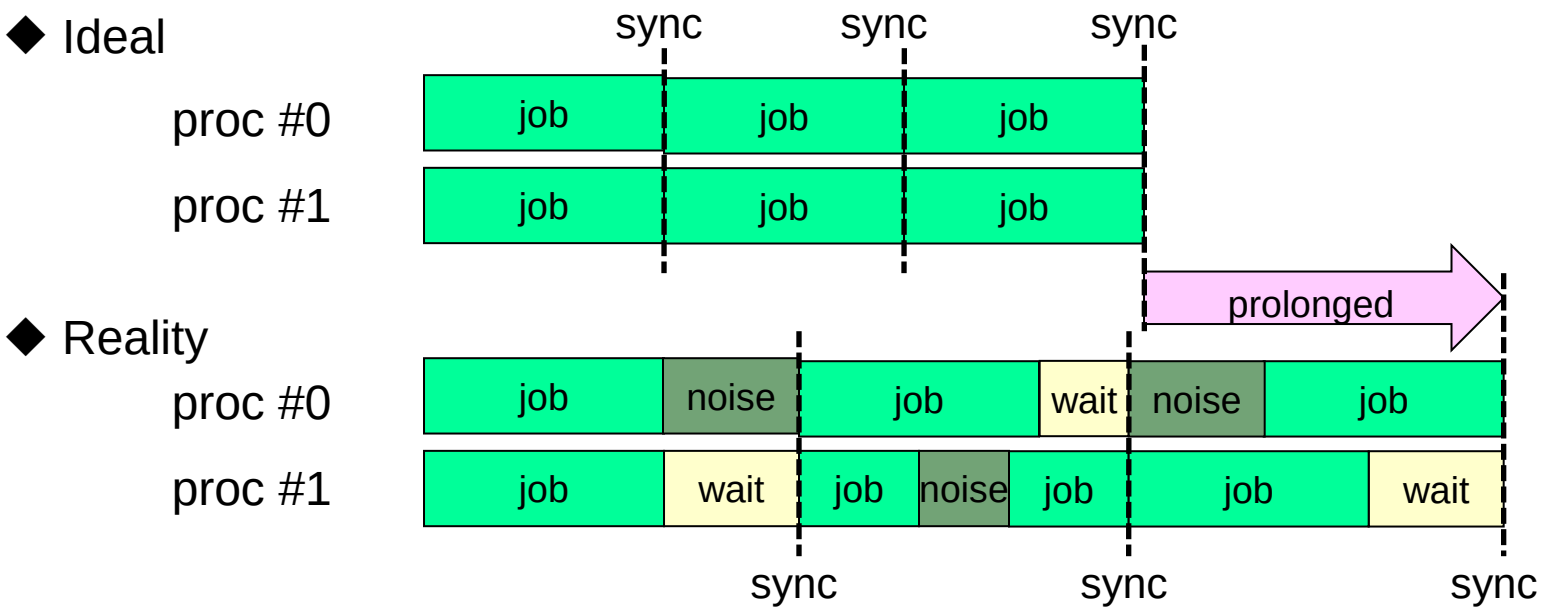


Though it tries to execute the outer loop in parallel at first, the iteration count ‘k’, which is 2, is too small to execute in parallel. So the inner loop, whose count is 256, is executed in parallel instead.

OS jitter problem for parallel processing

FUJITSU

- Due to synchronization, each computation is prolonged to the duration of the slowest process.
- Even if the job size of each process is exactly the same, OS interferes the application and time varies



- OS tuning and hybrid parallel can reduce OS jitter.

OS jitter measured

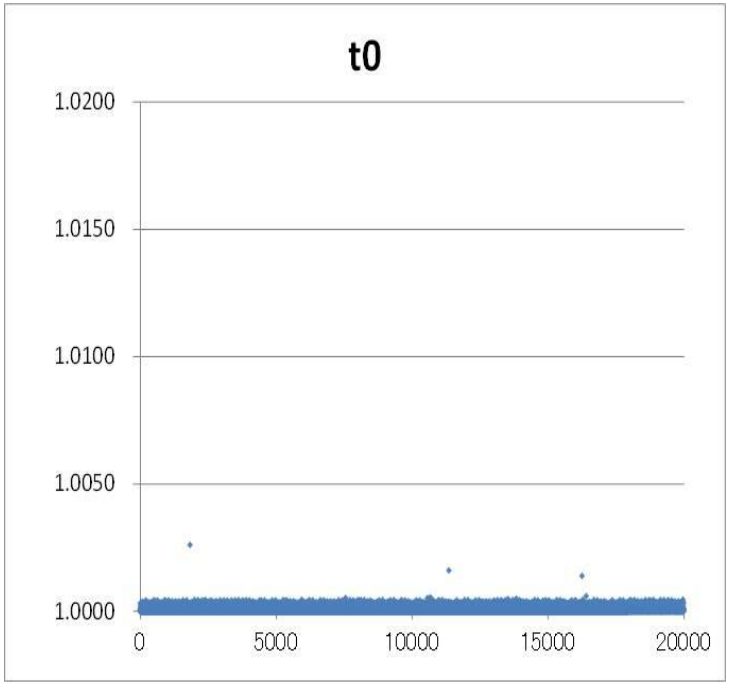
FUJITSU

- OS jitter (noise) measured using a program called FWQ developed by Lawrence Livermore National Laboratory. https://asc.llnl.gov/sequoia/benchmarks/

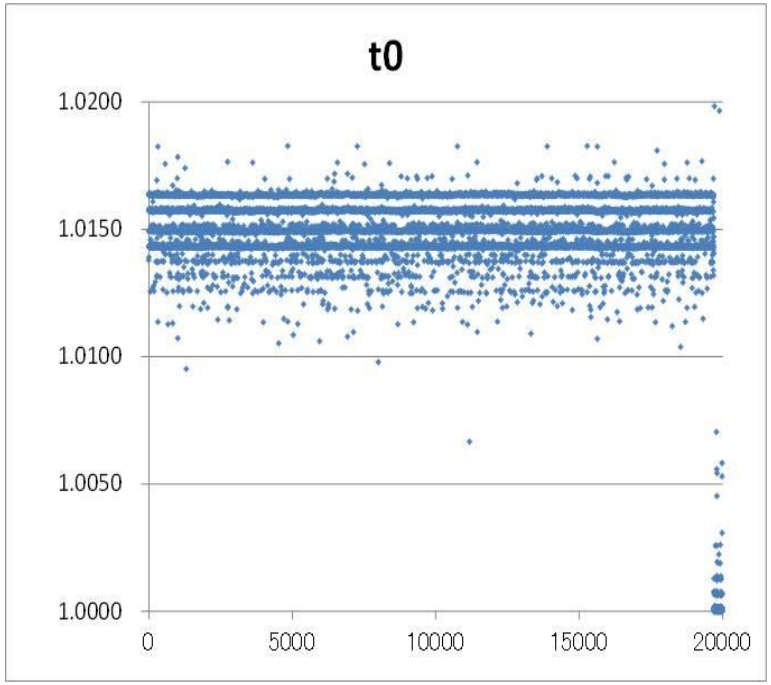
  t_fwq –w 18 –n 20000 –t 16

  -w: workload
  -n: repeat time
  -t: number of thread

PRIMEHPC FX10



x86

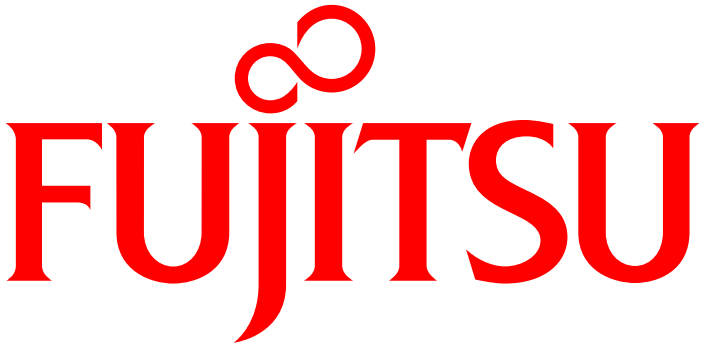


Machine	PRIMEHPC FX10	PC Cluster
Mean noise ratio	0. 589E-04	0. 154E-01
Longest noise length(usec)	29.3	644.0

Summary

FUJITSU

- Fujitsu's supercomputer, PRIMEHPC FX10, as well as K computer consists of high performance multi-core CPUs
- There are bunch of scalar tuning techniques to make each process faster
- We recommend to program applications in hybrid parallel manner to get a better performance
- By checking performance analysis information, you can find bottlenecks
- Some parallel tuning can be done using open MP directives
- Operating system could be an obstacle to get higher performance



shaping tomorrow with you