MPI Tuning with Intel® Trace Analyzer and Intel® Trace Collector

Intel Software College

Agenda

Introduction

Basic Usage

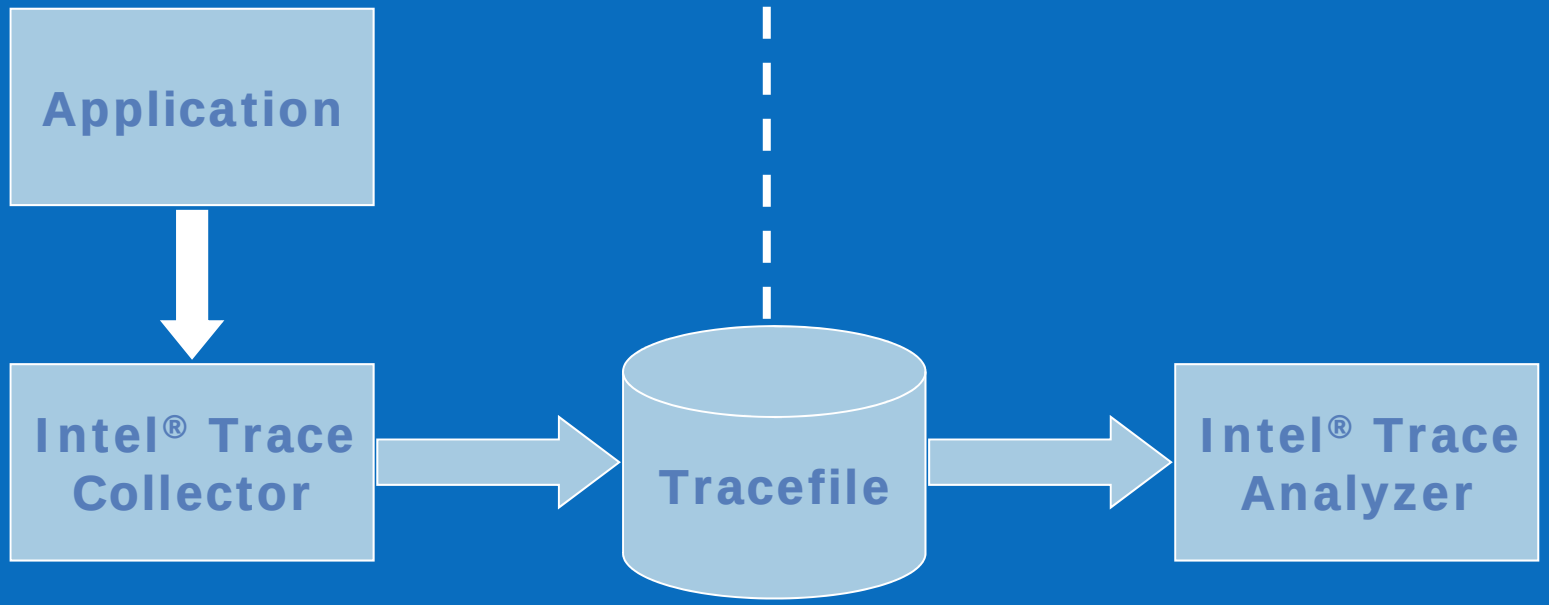
Focusing on the relevant

Detecting problems

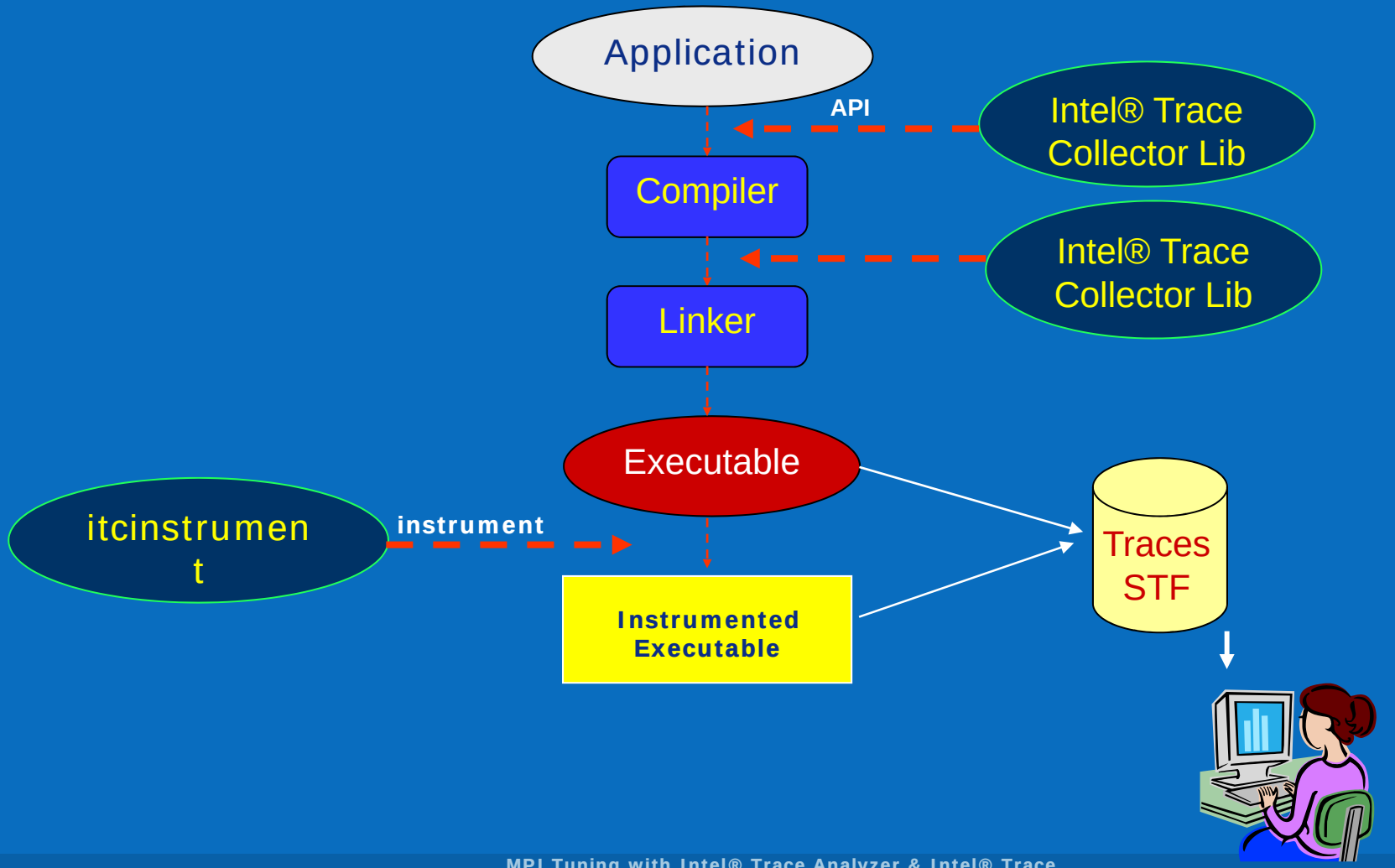
Introduction – What is Tracing?

- Record program execution
  - Program events such as function enter/exit, communication
- 1:1 protocol of the actual program execution
  - Sampling gathers statistical information
- Accurate data
- Easily get loads of data

Trace Universe



Components and Interaction



Supported Programming Models

MPI: Message Passing

Languages: C/C++, Java*, Fortran

Multithreading: explicit threads - implicit threads (OpenMP)

Hybrid Models: MPI + OpenMP

Supported Environments

Intel® Xeon® Processor

Intel 64 Bit Xeon Processor

Intel Itanium® Processor

Intel MPI, MPICH, LAM-MPI

Linux (RHEL, RHAS, SLES)

SGI* Altix*

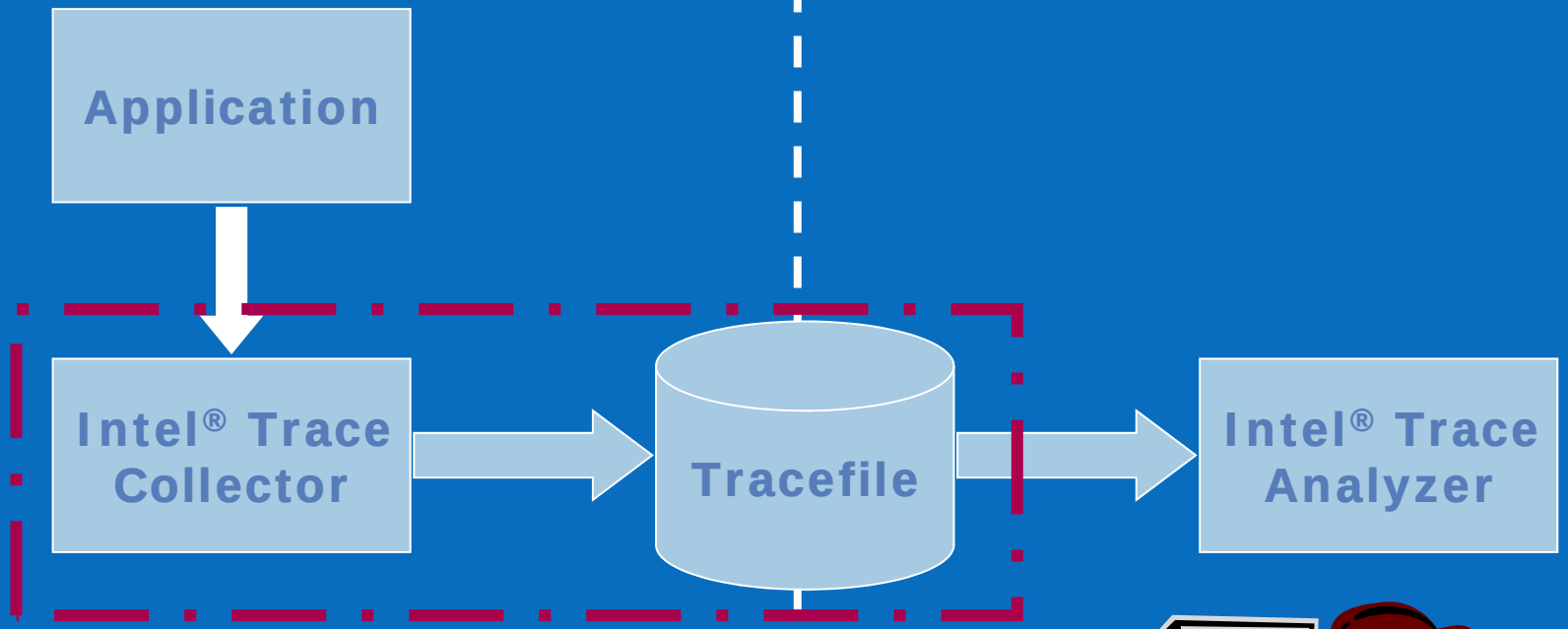
Microsoft* Windows* (Analyzer Only)

Introduction – Intel® Trace Collector

Key features

Usage

Trace Universe – Intel® Trace Collector



Intel® Trace Collector - Overview

Event based approach

Low impact on application performance

Provides API to instrument user code

Trace optimized program runs

Analyzes communication layer (default)

Event based approach

Event = time stamp + thread ID + description

Function entry/exit

Messages

Collective operations

Counter samples

Strengths of Event-based Tracing

Predict detailed program behavior

Record exact sequence of program states – keep timing consistent

Collect information about exchange of messages: at what times and in which order

An event-based approach is able to detect temporal dependencies!

Key Features

Catch all MPI events

Strong configuration mechanism
- Filters, settings, features

Automatic source-code references

Instrumentation
- Rich API
- Binary instrumentation (itcinstrument)
- Compiler based (beta)

Fail-safe version

How to use Intel® Trace Collector

Step 1: Generate instrumented binary re–link your application:

```
mpicc –shlib tst.o $(LFLAGS) -lVT –lmpi –ldwarf –lelf –lnsl
–lm –o tst
```

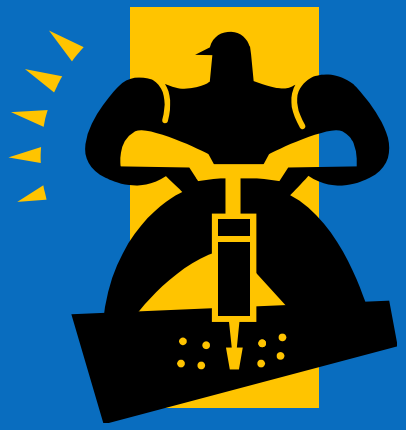
Step 2: Produce tracefile

run the instrumented binary for a representative amount of time (reduce initialization influences) on representative data (no corner cases)

Step 3: (optional): Instrument binary

```
itcinstrument –input lm –output lm_instr
```

Lab 1 - Set up Environment



Get hands-on package

Edit make_base to reflect local configuration

SETUP ENV

Get hands-on package

Package name:
/home/public/itac-bcor2005.tgz

Contents:

- make_base
- Makefile
  - task01
    - **README**
    - **Description**
    - **source files**
  - task02
  - task03
  - ...

Edit make_base to reflect local configuration

```
CC              = [path to MPI compiler wrapper]
MPI_ROOT        = [path to MPI implementation]
VT_ROOT         = [path to  Intel® Trace Collector]
RUN_CMD         = [command to run mpi exec's]
```

Example:

```
CC              = mpiicc
MPI_ROOT        = /opt/intel/mpi/2.0
VT_ROOT         = /opt/intel/itc6
RUN_CMD         = /opt/intel/mpi/2.0/bin64/mpiexec -n
```

SETUP ENV

Intel® Trace Collector installation is in [ITC]: /opt/intel/itc6

- csh: source [ITC]/sourceme.csh
- sh: sh [ITC]/sourceme.sh

Intel® Trace Analyzer installation is in [ITA]: **/opt/intel/ita6**

- csh:

```
set path = ( . [ITA]/bin [ITC]/bin $path)
```

- bash:

```
export PATH=$PATH:[ITA]/bin:[ITC]/itc6/bin
```

- start: traceanalyzer

Lab 2 – Trace File Generation

Objective: How to generate a tracefile.

This directory contains **cpi.c** which is a simple example program.

Re-link (or compile and link) with Intel® Trace Collector library.

Run again – check for console message on 'writing tracefile'.

Open tracefile with Intel® Trace Analyzer.

```
[traceanalyzer cpi_itc.stf]
```

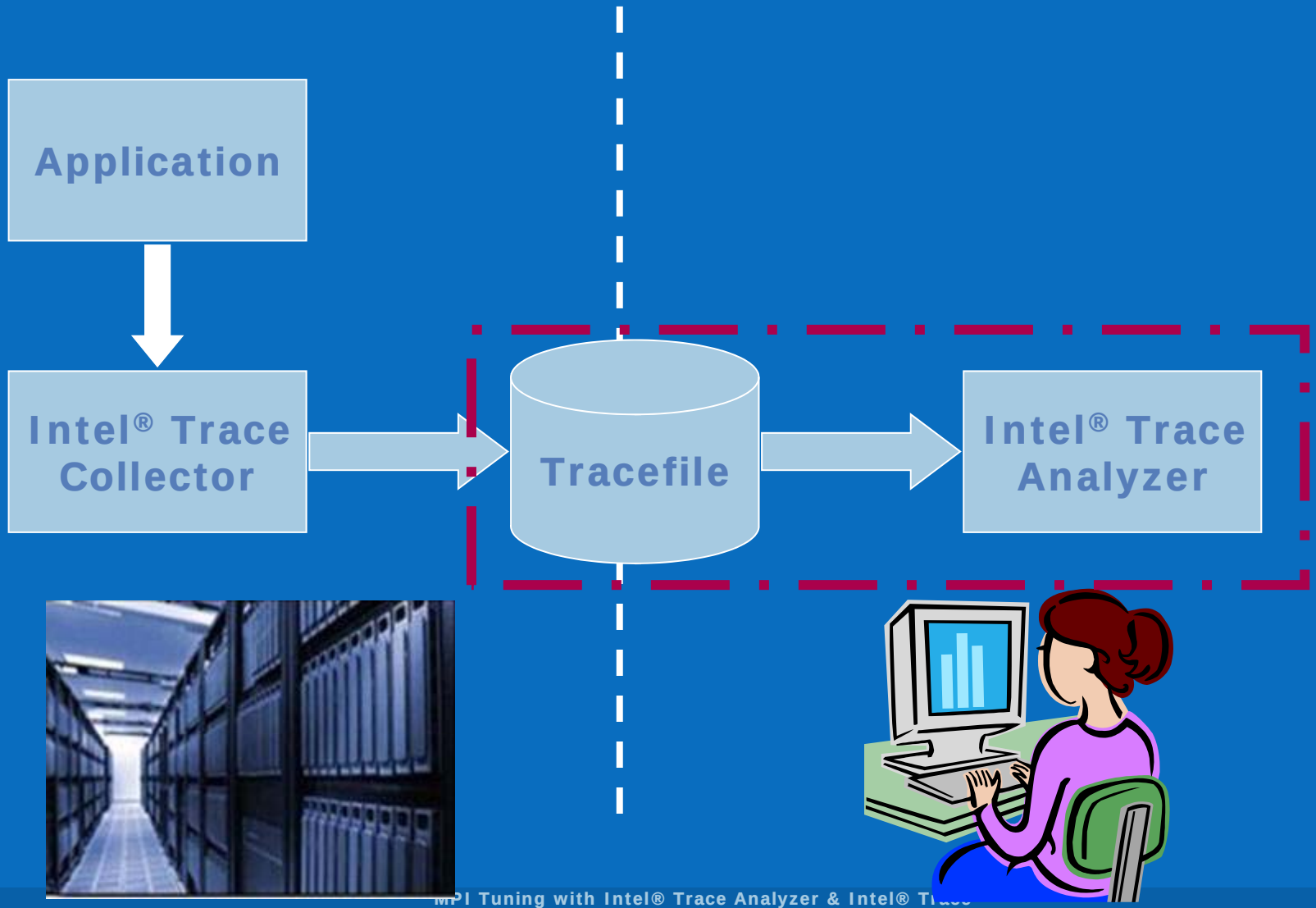
Lab 3 – Trace File Generation

Introduction – Intel® Trace Analyzer

Key features

Usage

Trace Universe - Intel® Trace Analyzer



Intel® Trace Analyzer - Overview

Enables the user to quickly focus at the appropriate level of detail Offline trace analysis (postmortem)

Use of hierarchical techniques to address scalability in function, time and processor–space

High–performance graphics, excellent zooming and filtering

Views and Charts - I

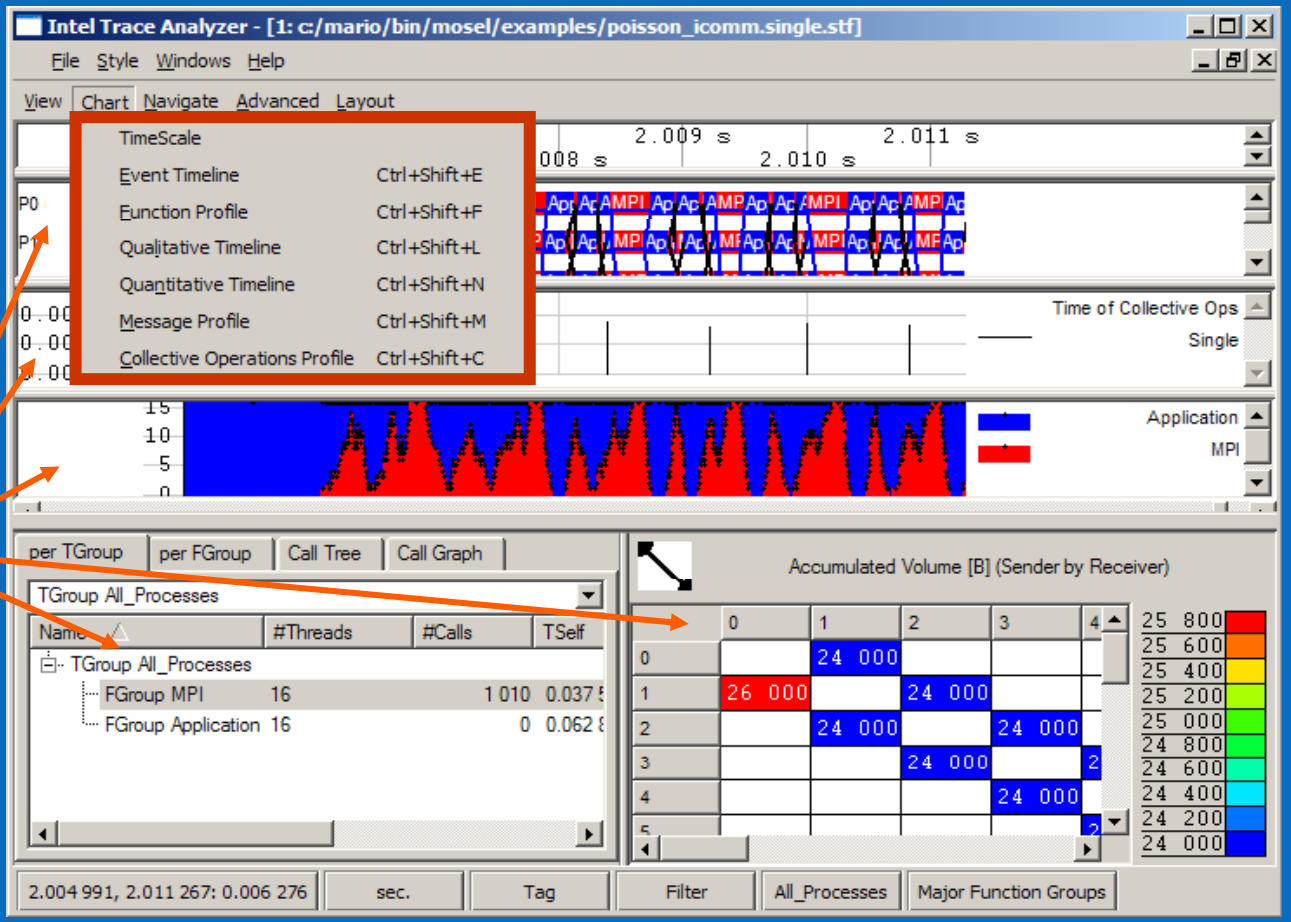
What's a Chart?

What Charts are provided?

What's a View?

What's the purpose of a View?

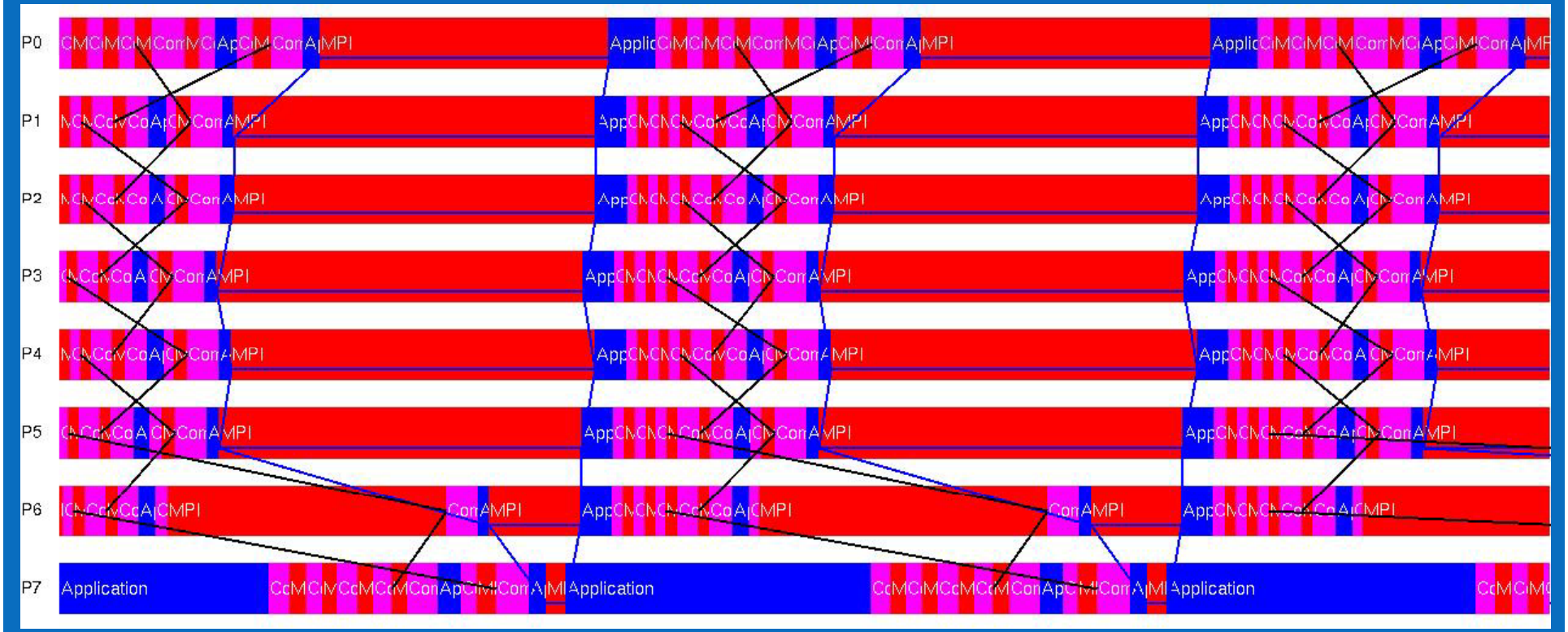
Chart



Chart

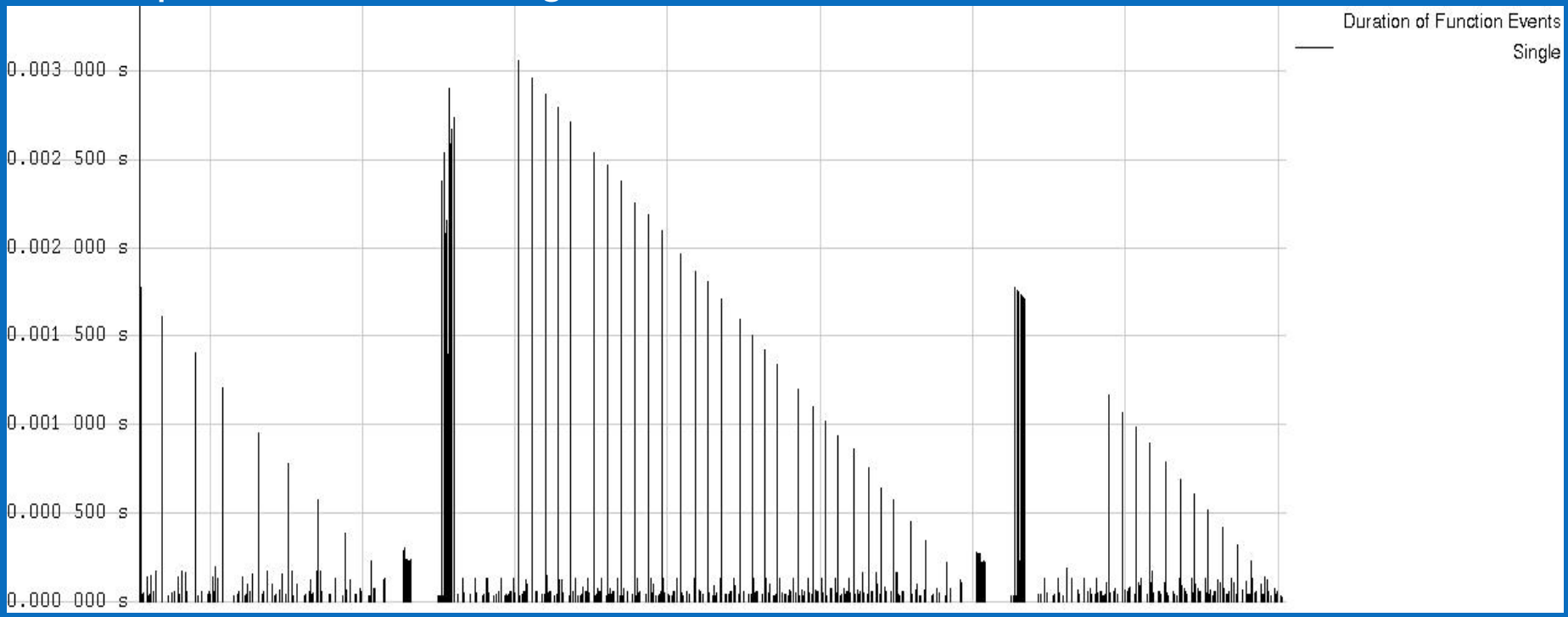
A Chart is a numerical or graphical diagram

Timelines: Event Timeline



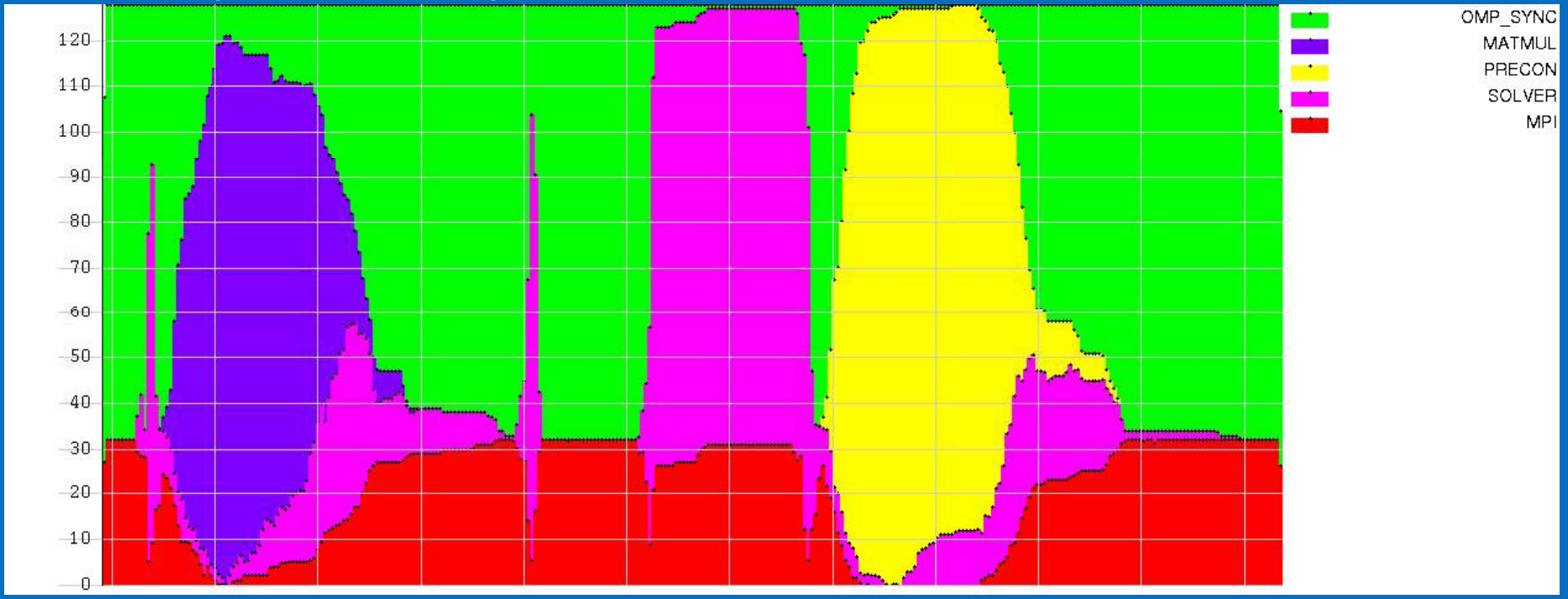
Timelines: Qualitative Timeline

Find patterns and irregularities



Timelines: Quantitative Timeline

Get impression on parallelism and load balance



Profiles: Flat Function Profile

Statistics about functions

Flat Profile | Load Balance | Call Tree | Call Graph

Group All_Threads

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Threads					
PRECON	678.787 445 s		678.787 445 s	49 536	0.013 703 s
OMP_SYNC	580.473 344 s		580.473 344 s	296 320	0.001 959 s
MATMUL	410.463 131 s		410.463 131 s	49 280	0.008 329 s
SOLVER	328.400 819 s		2 169.146 934 s	128	2.565 631 s
User_Code	149.746 154 s		2 383.561 817 s	128	1.169 892 s
MPI_Bcast	94.227 914 s		94.227 914 s	37 248	0.002 530 s
ASSEMBLY	43.822 701 s		43.822 701 s	32	1.369 459 s
MPI_Barrier	24.222 499 s		24.222 499 s	49 312	0.000 491 s
MPI_Reduce	23.807 645 s		23.807 645 s	37 184	0.000 640 s
MPI_Waitall	17.607 615 s		17.607 615 s	49 472	0.000 356 s
MPI_Comm_dup	11.756 564 s		11.756 564 s	64	0.183 696 s
MPI_Isend	7.838 689 s		7.838 689 s	145 324	0.000 054 s
MPI_Wtime	7.490 313 s		7.490 313 s	136 192	0.000 055 s
MPI_Irecv	4.909 197 s		4.909 197 s	145 324	0.000 034 s
MPI_Finalize	0.006 288 s		0.006 288 s	32	0.000 197 s
MPI_Comm_size	0.001 205 s		0.001 205 s	64	0.000 019 s
MPI_Comm_rank	0.000 293 s		0.000 293 s	32	0.000 009 s

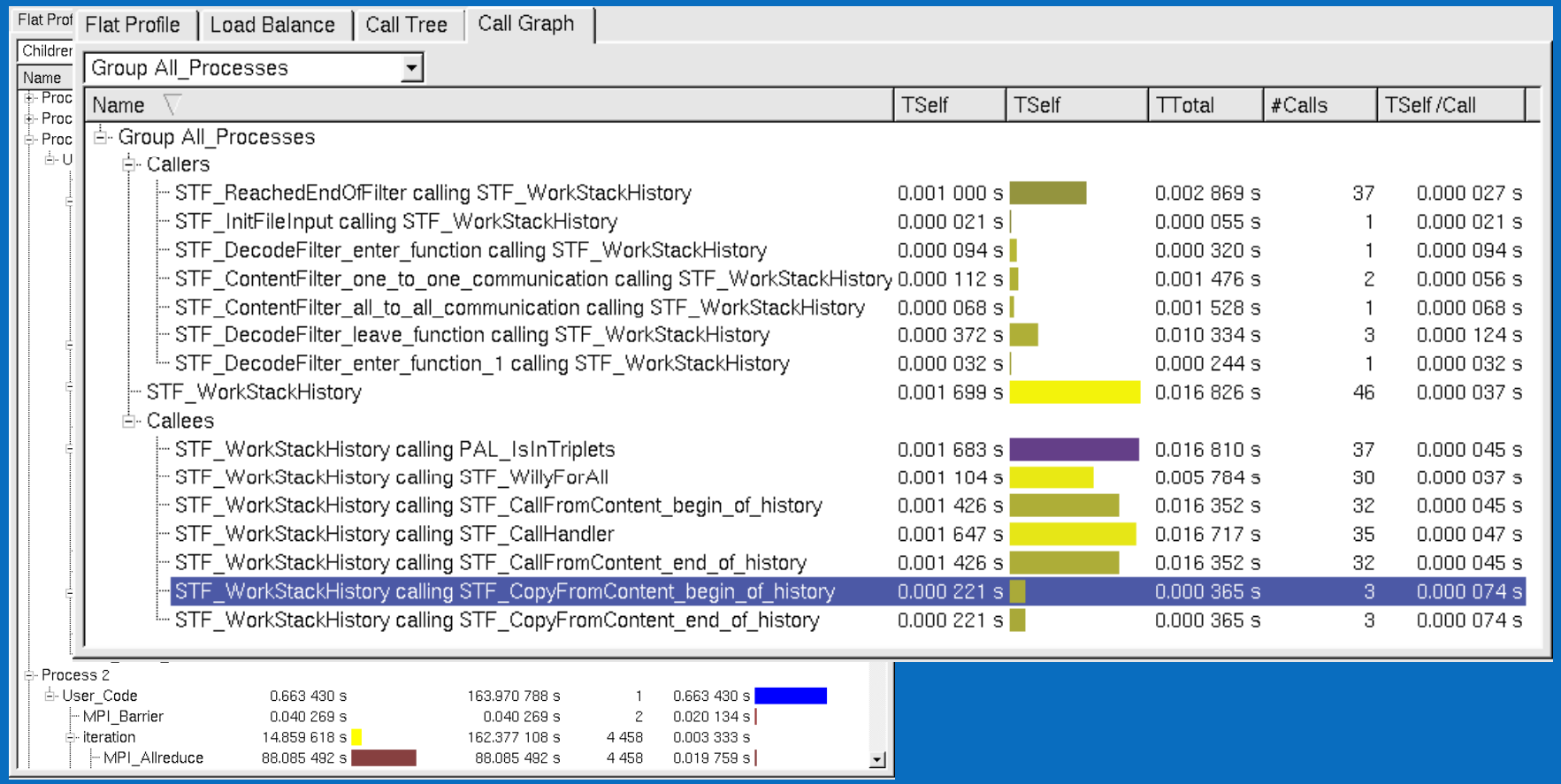
Flat Profile | Load Balance | Call Tree | Call Graph

Children of Group All_Threads

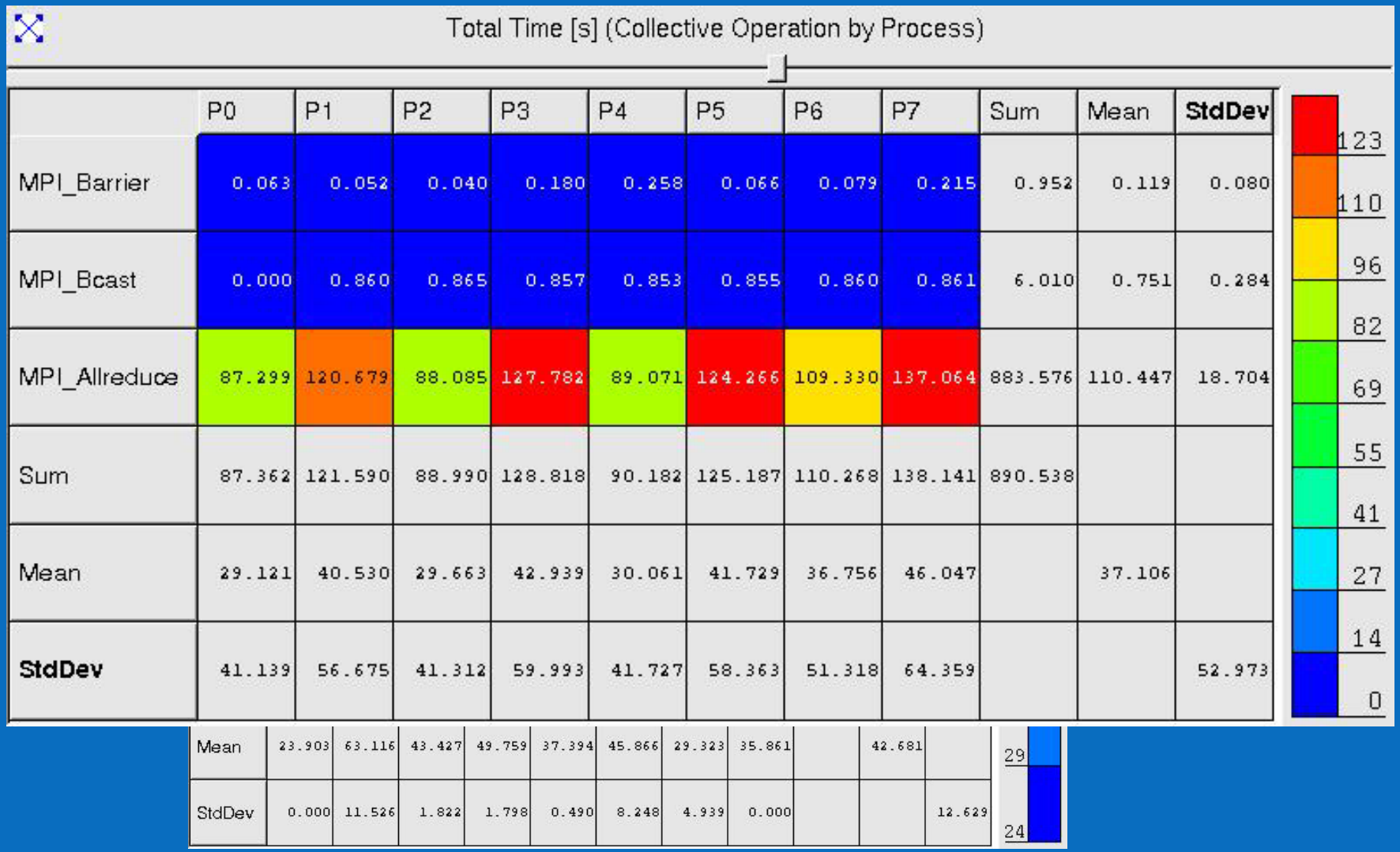
Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
MPI_Comm_dup					
MPI_Waitall					
Process 31 Thread 0	0.013 338 s		0.013 338 s	1 546	0.000 501 s
Process 24 Thread 0	0.801 994 s		0.801 994 s	1 546	0.000 519 s
Process 28 Thread 0	0.756 392 s		0.756 392 s	1 546	0.000 489 s
Process 23 Thread 0	0.721 329 s		0.721 329 s	1 546	0.000 467 s
Process 27 Thread 0	0.711 207 s		0.711 207 s	1 546	0.000 460 s
Process 7 Thread 0	0.643 754 s		0.643 754 s	1 546	0.000 416 s
Process 15 Thread 0	0.637 547 s		0.637 547 s	1 546	0.000 412 s
Process 16 Thread 0	0.628 403 s		0.628 403 s	1 546	0.000 406 s
Process 0 Thread 0	0.610 254 s		0.610 254 s	1 546	0.000 395 s
Process 8 Thread 0	0.598 698 s		0.598 698 s	1 546	0.000 387 s
Process 4 Thread 0	0.594 556 s		0.594 556 s	1 546	0.000 385 s
Process 20 Thread 0	0.575 368 s		0.575 368 s	1 546	0.000 372 s
Process 25 Thread 0	0.573 404 s		0.573 404 s	1 546	0.000 371 s
Process 26 Thread 0	0.571 285 s		0.571 285 s	1 546	0.000 370 s
Process 11 Thread 0	0.555 121 s		0.555 121 s	1 546	0.000 359 s
Process 30 Thread 0	0.547 251 s		0.547 251 s	1 546	0.000 354 s
Process 29 Thread 0	0.547 177 s		0.547 177 s	1 546	0.000 354 s
Process 3 Thread 0	0.540 298 s		0.540 298 s	1 546	0.000 349 s
Process 19 Thread 0	0.510 765 s		0.510 765 s	1 546	0.000 330 s
Process 2 Thread 0	0.495 491 s		0.495 491 s	1 546	0.000 320 s
Process 12 Thread 0	0.485 023 s		0.485 023 s	1 546	0.000 314 s
Process 5 Thread 0	0.480 013 s		0.480 013 s	1 546	0.000 310 s
Process 21 Thread 0	0.474 150 s		0.474 150 s	1 546	0.000 307 s
Process 6 Thread 0	0.466 212 s		0.466 212 s	1 546	0.000 302 s
Process 18 Thread 0	0.452 495 s		0.452 495 s	1 546	0.000 293 s
Process 1 Thread 0	0.448 999 s		0.448 999 s	1 546	0.000 290 s
Process 13 Thread 0	0.392 865 s		0.392 865 s	1 546	0.000 254 s
Process 22 Thread 0	0.387 010 s		0.387 010 s	1 546	0.000 250 s
Process 14 Thread 0	0.377 664 s		0.377 664 s	1 546	0.000 244 s
Process 17 Thread 0	0.377 174 s		0.377 174 s	1 546	0.000 244 s
Process 10 Thread 0	0.374 776 s		0.374 776 s	1 546	0.000 242 s
Process 9 Thread 0	0.357 603 s		0.357 603 s	1 546	0.000 231 s
MPI_Irecv					
Process 19 Thread 0	0.245 502 s		0.245 502 s	6 184	0.000 040 s
Process 17 Thread 0	0.243 382 s		0.243 382 s	6 184	0.000 039 s

Profiles: Call-Tree and Call-Graph

Function statistics including calling hierarchy



Communication Profiles



View

Helps navigating through the trace data and keep orientation

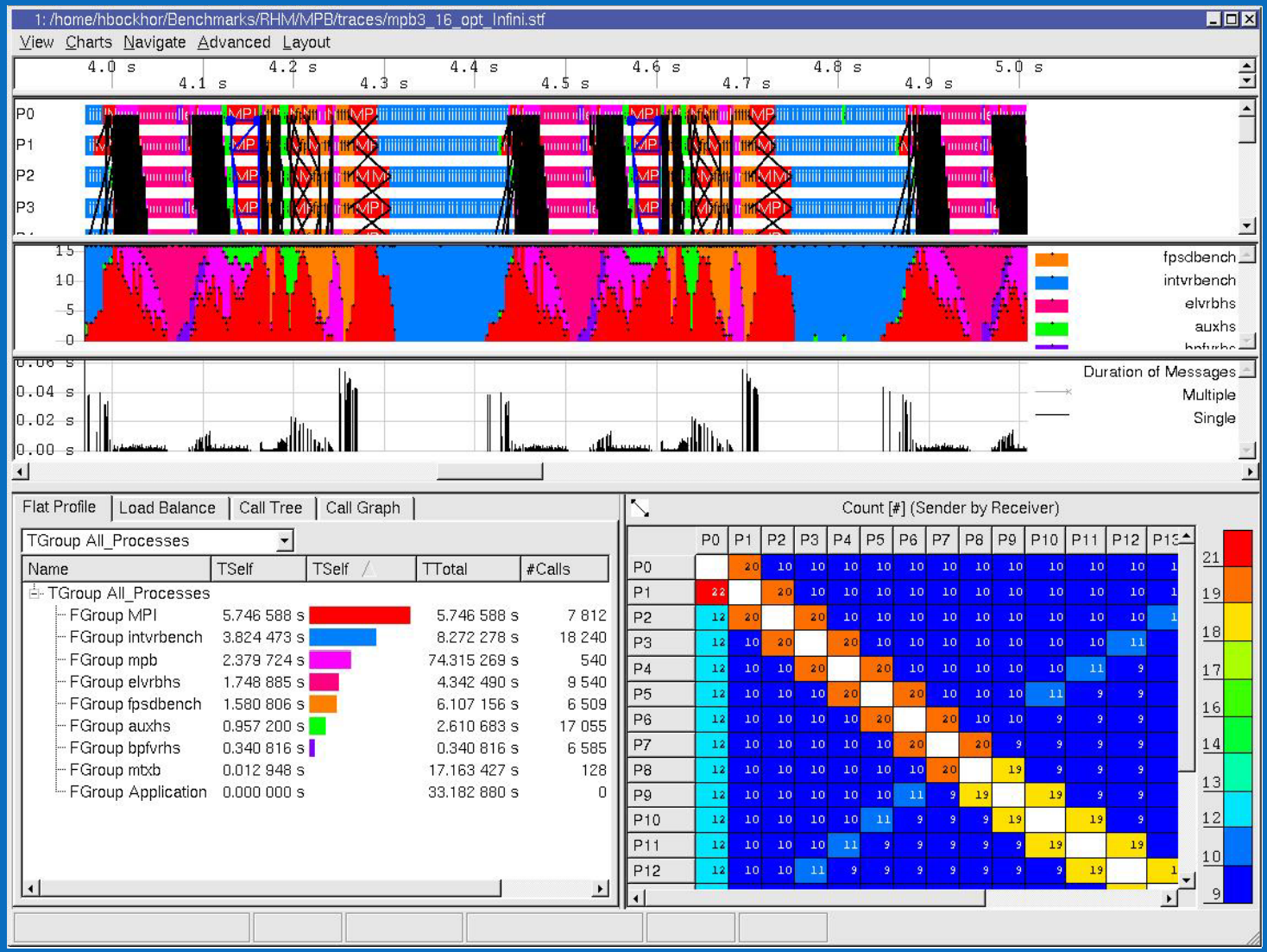
Every View can contain several Charts

A View on a file is defined by a triplet of
- time-span
- set of threads
- set of functions

All Charts follow changes to View (e.g. zooming)

Timelines are correctly aligned along time

View - zooming



Flexibility of Views

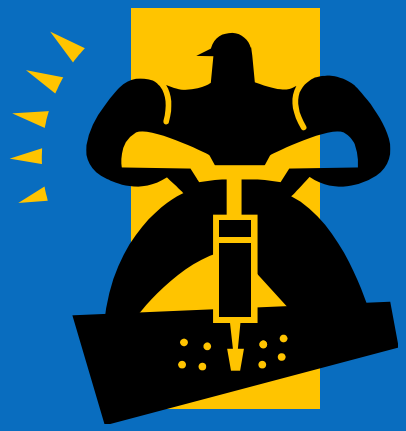
Several Views can be opened (on the same or on different files)

Location, orientation and size of charts can easily be changed

Entire Views can and individual charts can be cloned and closed

Individual charts can be cloned in own View

Lab 4 – Views and Charts



Charting and Viewing

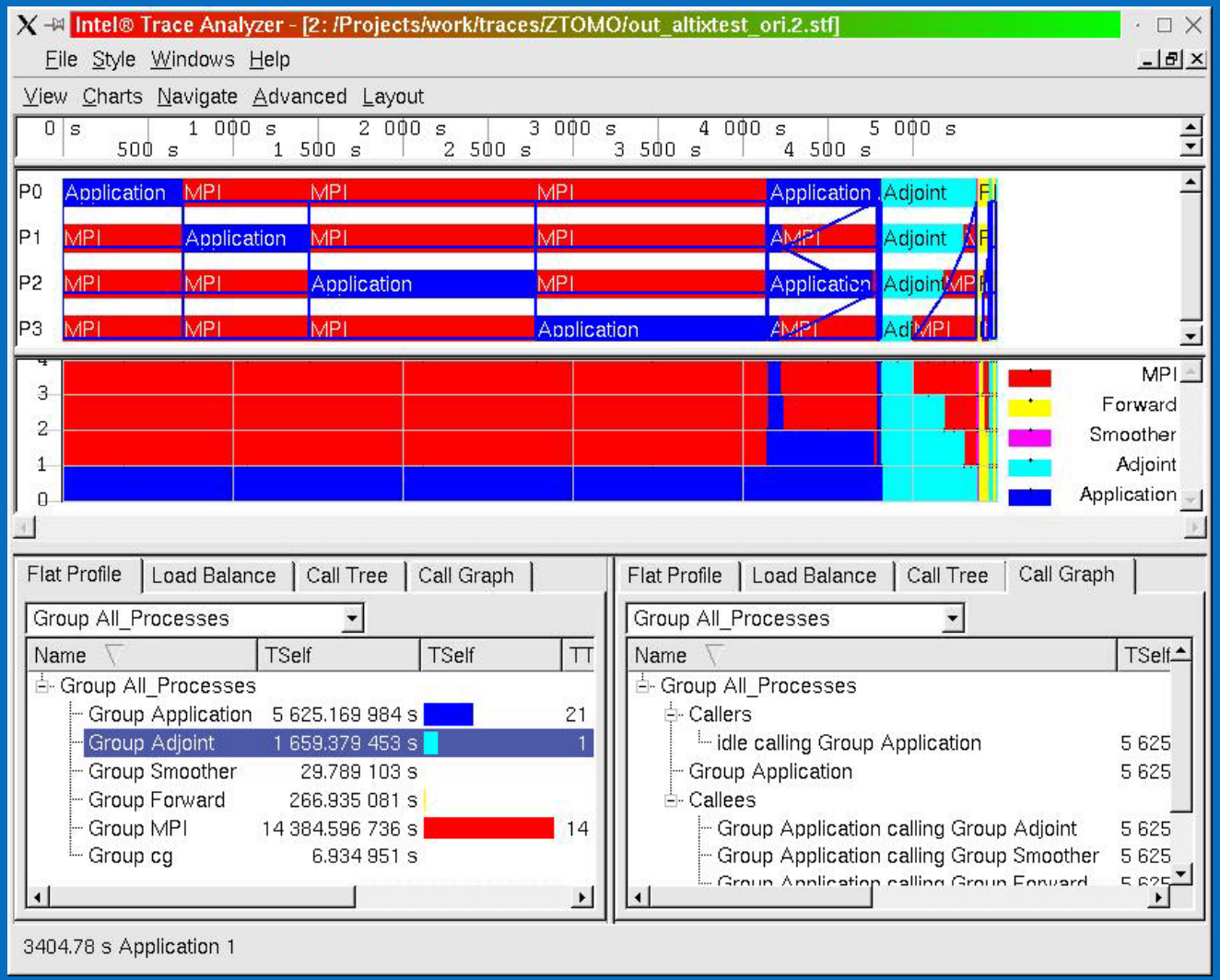
Start with

`traces/ztomo/out_altixtest_ori.2.stf`

Use Chart menu to open

- EventTimeline
- Quantitative Timeline
- Function Profile (Flat Profile)
- Function Profile (CallGraph)

Lab 4 - Results



Charting and Viewing

Start with

`traces/ztomo/out_altixtest_ori.2.stf`

Use Chart menu to open

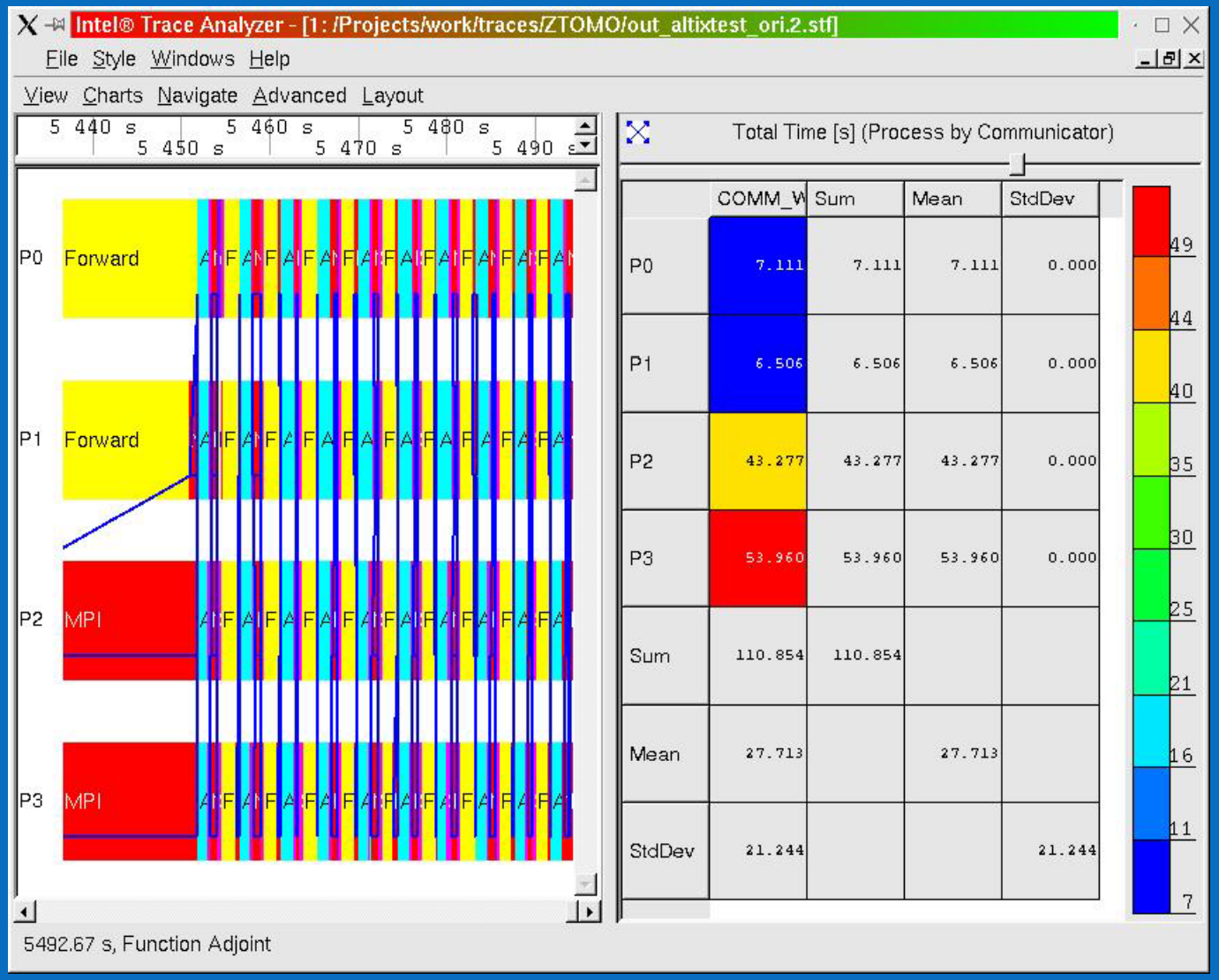
- EventTimeline (zoom to very end (left))
- Collective Operation Profile

Use Layout menu to get time line to left

Use Context menu in CollOpChart

- Colums to show Communicator
- Rows to show Processes

Lab 4b - Results



End of Module

Grouping and Aggregation

Allow analysis on different levels of detail by aggregating data upon group-definitions

Functions and threads can be grouped hierarchically

- Function Groups and Thread Groups

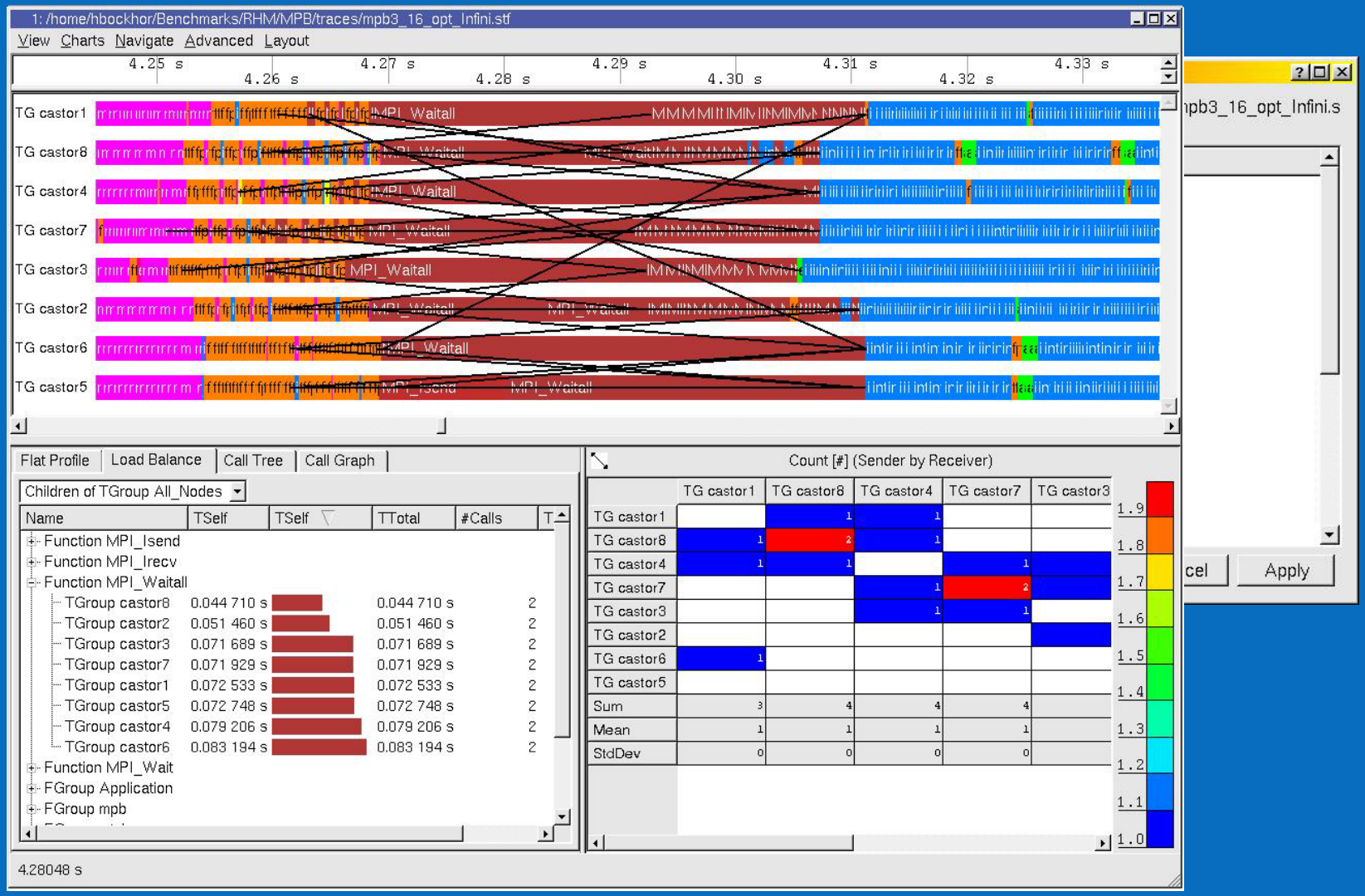
Arbitrary nesting is supported

- Functions/threads on the same level as groups
- User can define his/her own groups

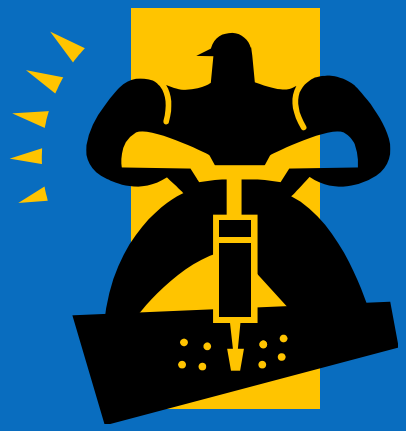
Aggregation is part of View-definition

- All charts in a View adapt to requested grouping
- All charts support aggregation

Aggregation Example



Lab 5 - Aggregation



Two ways to determine time spent not in MPI for instrumented codes

Start with

`traces/ztomo/out_altixtest_ori.2.stf`

Quick

- Select MPI for Function Aggregation

MPI <-> non-MPI

- Create new group
- Drag & Drop MPI
- Select new group for Function Aggregation

Tagging & Filtering

Help concentrating on relevant parts

Avoid getting lost in huge amounts of trace data

Define a set of interesting data

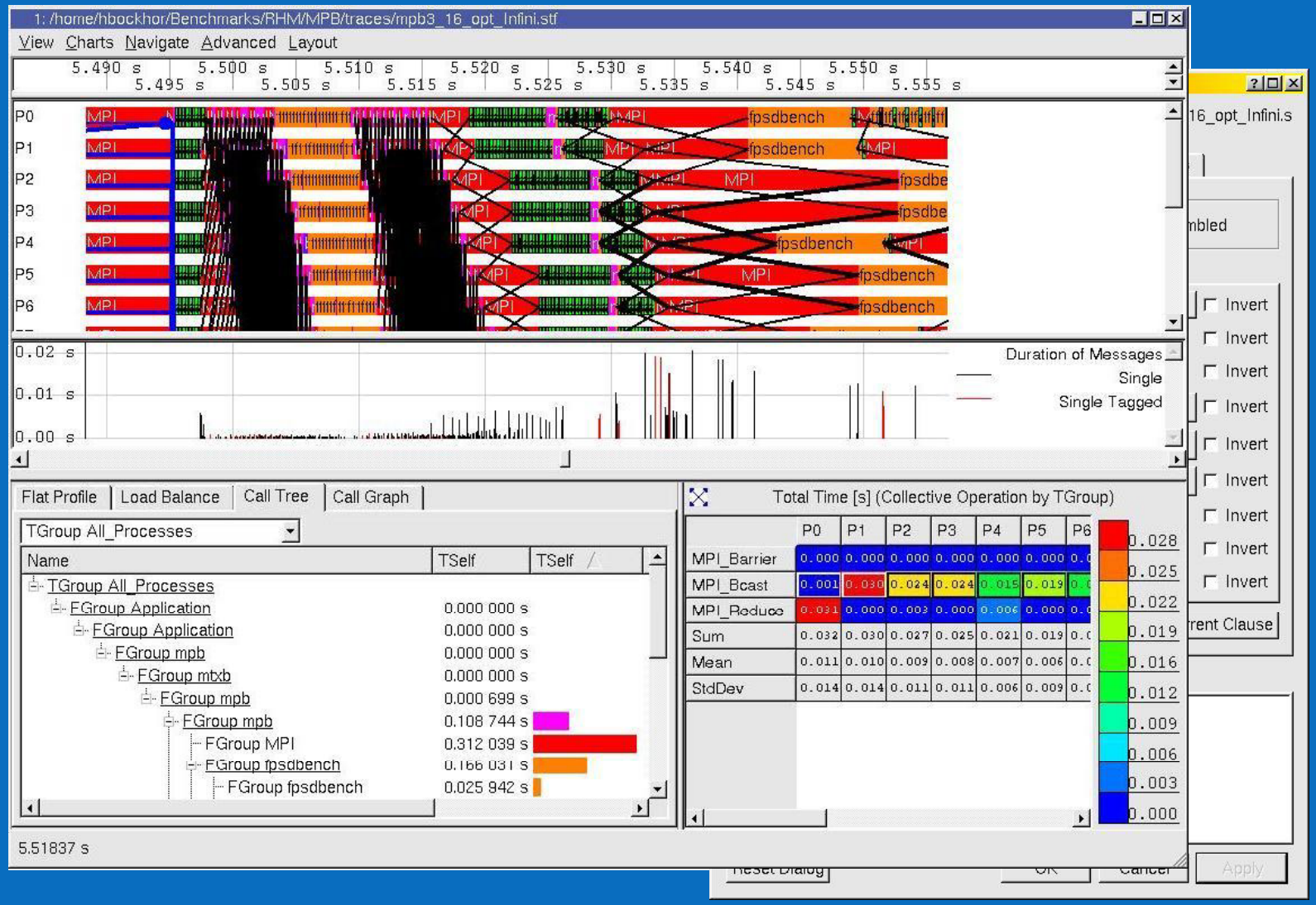
- E.g. all occurrences of function x
- E.g. all messages with tag y on communicator z

Combine several filters:
Intersection, Union, Complement

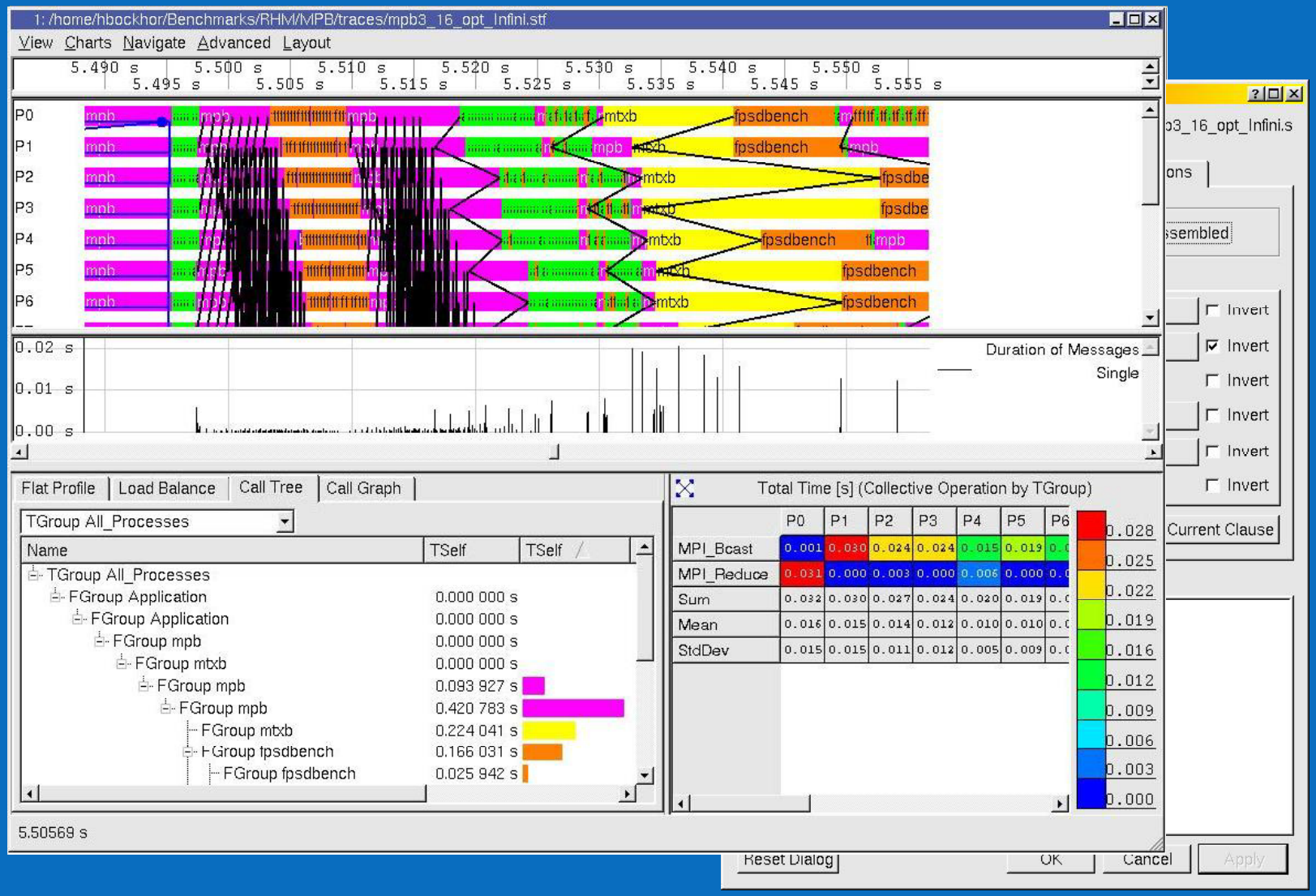
Apply it

- Tagging: Highlight messages
- Filtering: Suppress all non-matching events

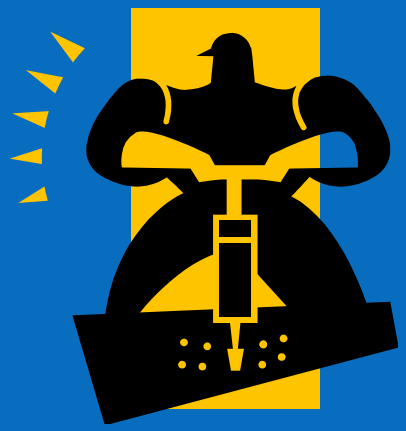
Tagging Example



Filtering Example



Lab 6 – Tagging and Filtering



Find threads performing a task

Start with

`traces/ita/ita_fgtl_threaded.stf`

Search for `FuncProfAnalyzer::*`

Load Balance & Event Timeline

Tagging

- Advanced|Tagging
  - Select `Function` tab and `Custom`
  - Type `FuncProfAnalyer` and press `ok`

Suppress GUI idle time

Start with

`traces/ita/ita_fgtl_threaded.stf`

Assumption: GUI is idle time

Use Filter (Function, “GUI”, invert)

Detecting Problems

Link to material

Verify important hypotheses

Slide demo

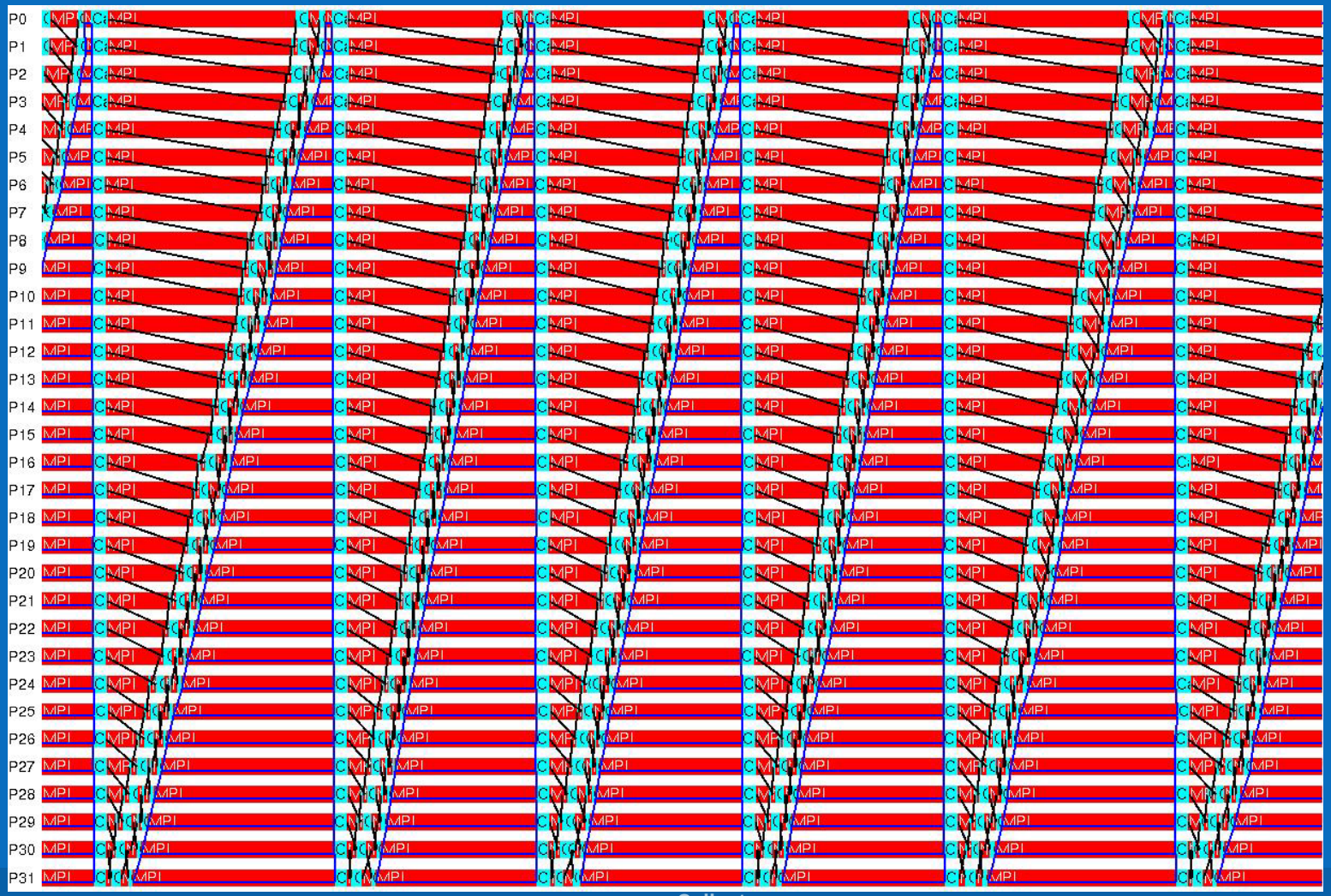
- Live demo

Understanding your code

Parallel Poisson Solver

Example of intuitive parallelization with disadvantageous communication pattern

Partial Trace of Poisson Solver



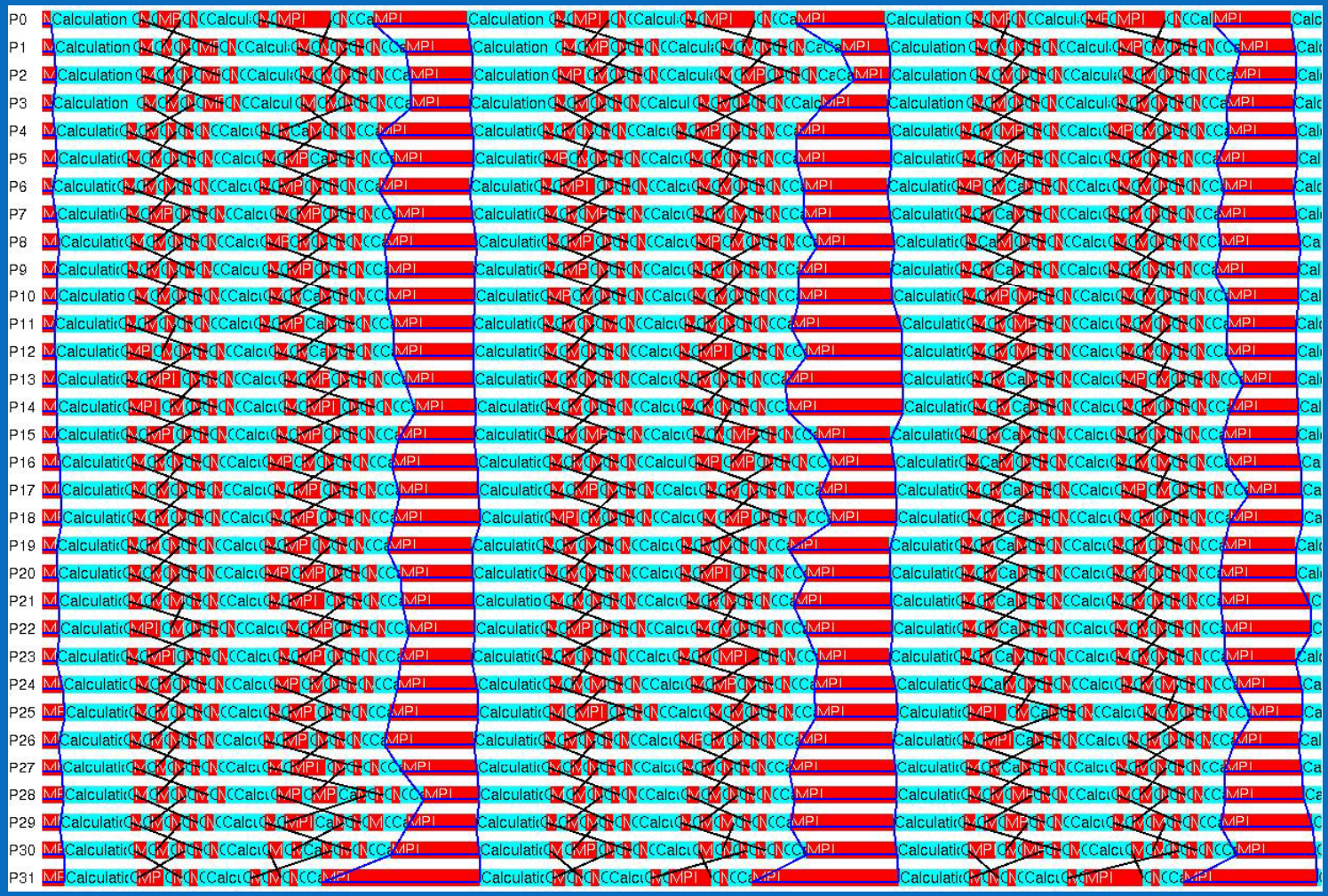
Understanding the problem

Blocking border exchange

P_n has blocks until communication between P_{n+1} and P_{n+2} was completed

Solution: Non blocking communication

Non-Blocking Version



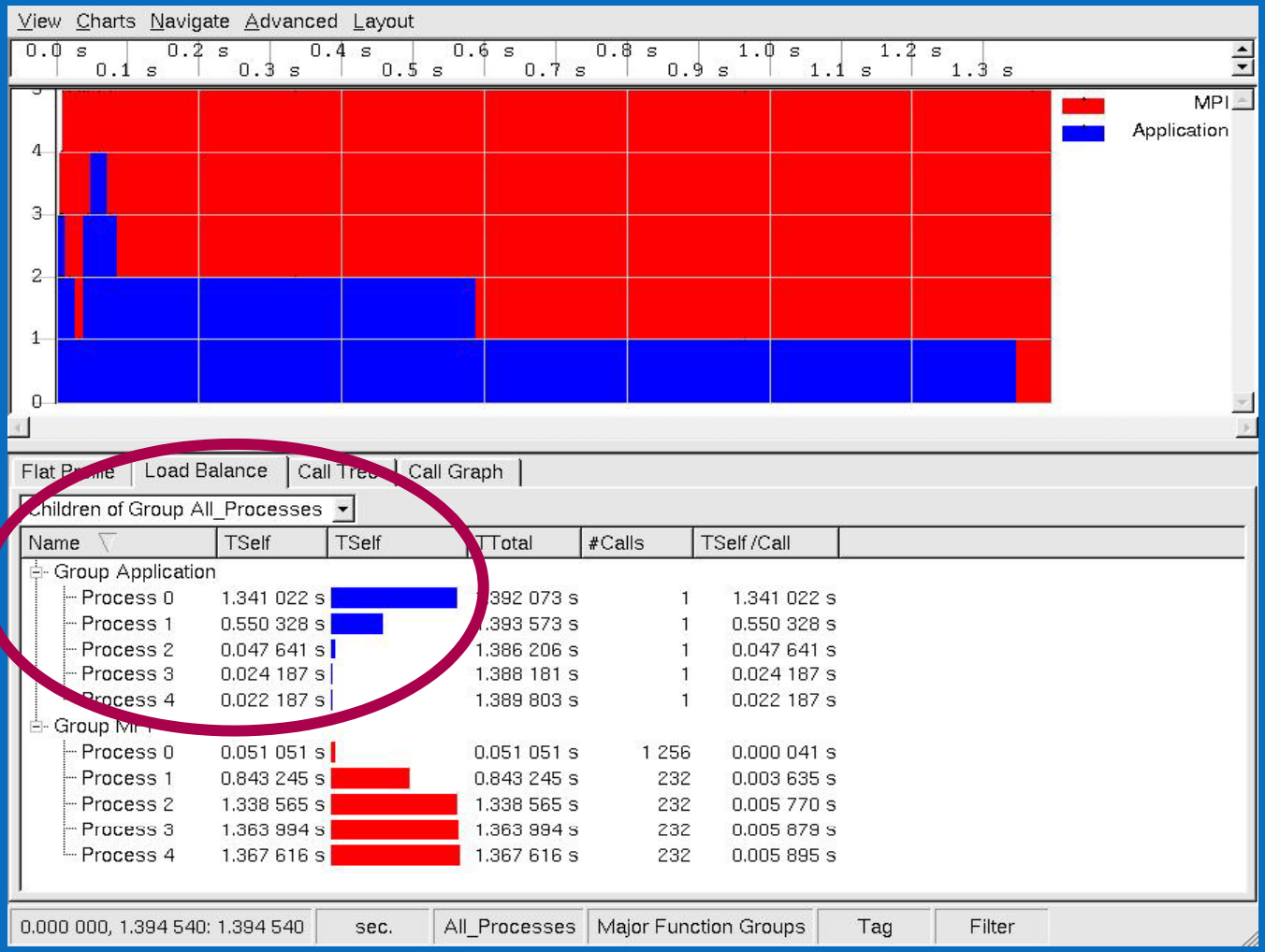
Detecting Load Imbalance

Mandelbrot set (MPI-tutorial)

- `mpitutorial.tar.gz`

Example of intuitive parallelization with huge load imbalance

Mandelbrot Example Trace



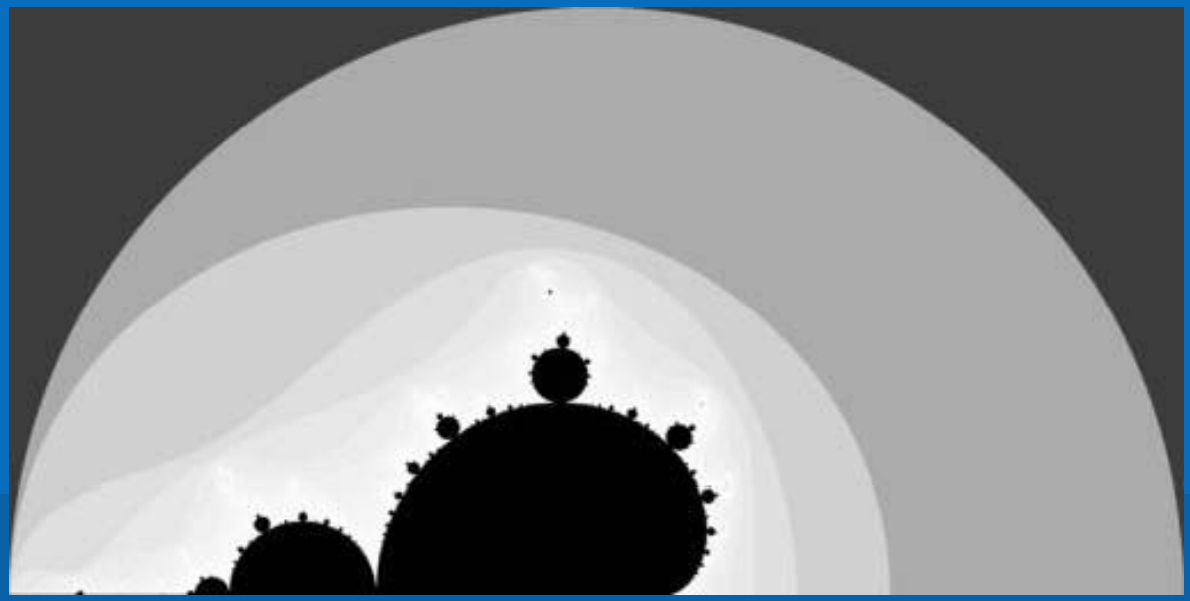
Understanding Load Imbalance

Static partitioning assigns contiguous blocks of rows

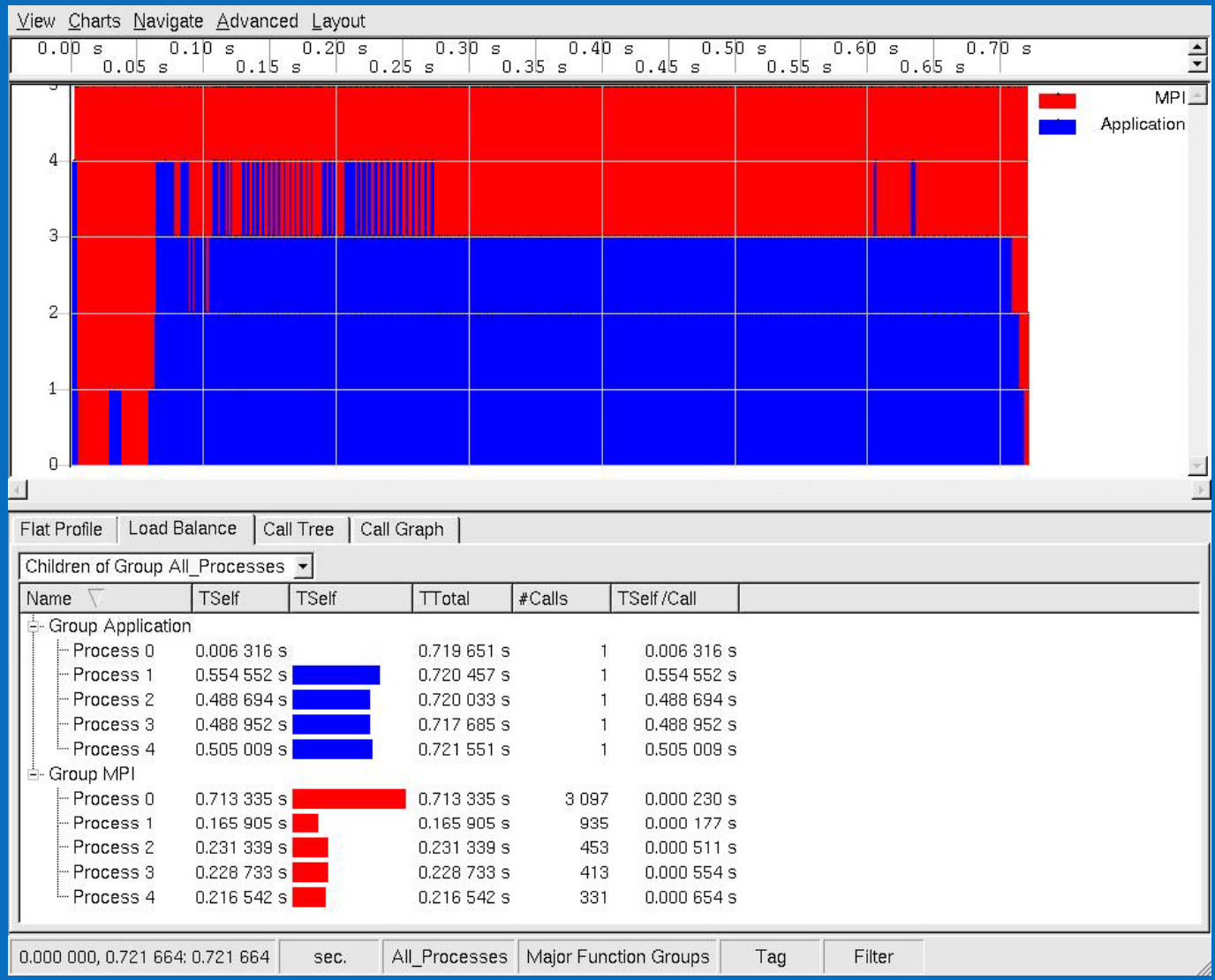
- Workload changes gradually with row-index!

What is the/a solution?

- Use modulo operator
- Use dynamic master/slave approach



Understanding Load Imbalance



Investigating Scalability - Amdahl's Law

Non-parallel sections limit the possible speedup

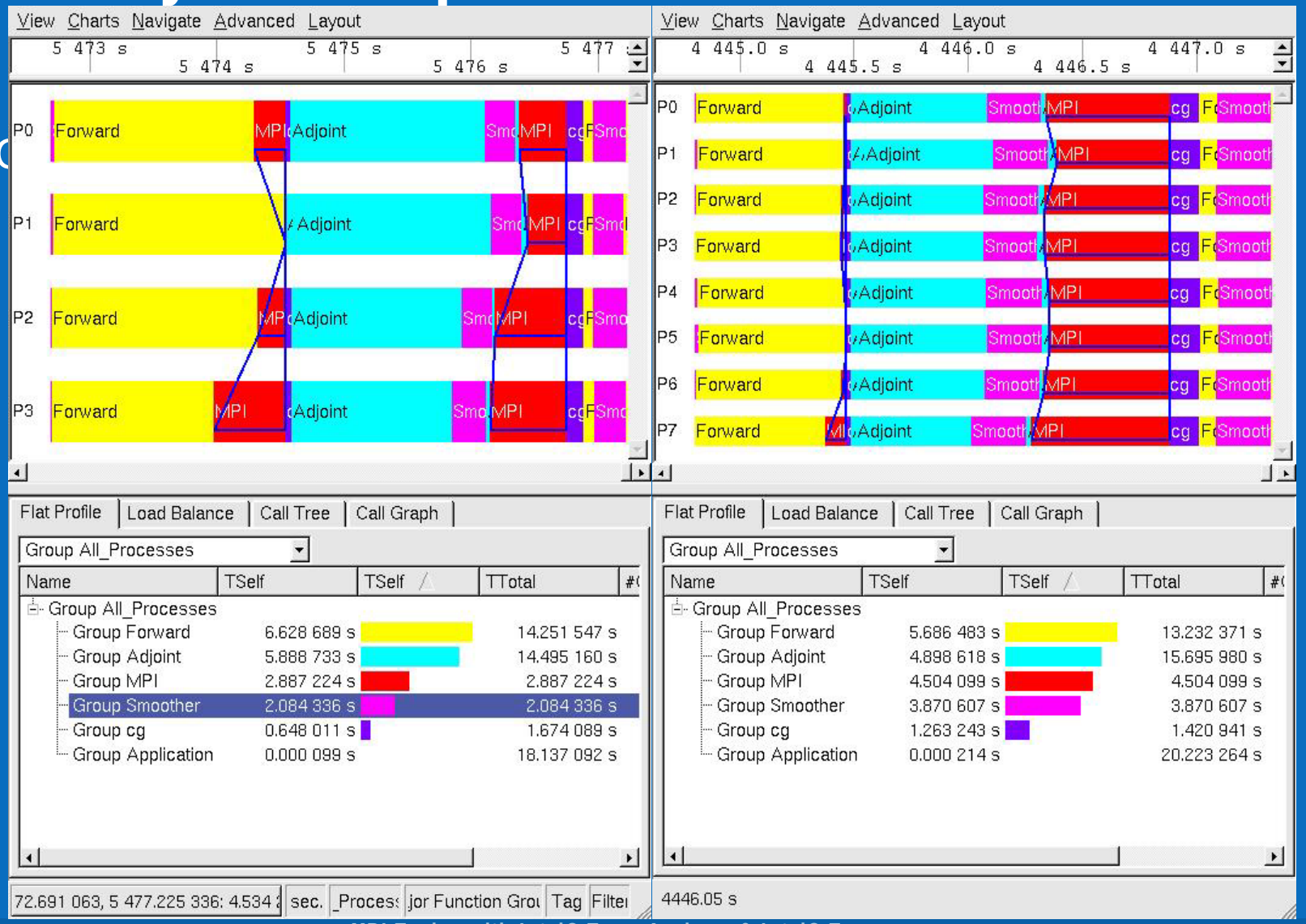
Amdahl's Law: If F is the fraction of a calculation that is sequential, and (1-F) is the fraction that can be parallelized, then the maximum speedup that can be achieved by using P processors is 1/(F+(1-F)/P)

2 runs with different number of processes:

- If sum of time spent in function increases with increasing number or processes
  => non-parallel

Scalability - Example

ZTOMC



Detecting Communication Inefficiency

Message Profile

Use minimum transfer rate

Thread Groups and Aggregation

- All_nodes group
- Intra <-> extra node

Compare with HW expectation

Determining Overhead of Modules

Modules can contain more than one function

Use Function Aggregation to determine the module's overhead

Trace of Intel® Trace Analyzer

- Does ZLib produce significant overhead?

Focusing on the relevant

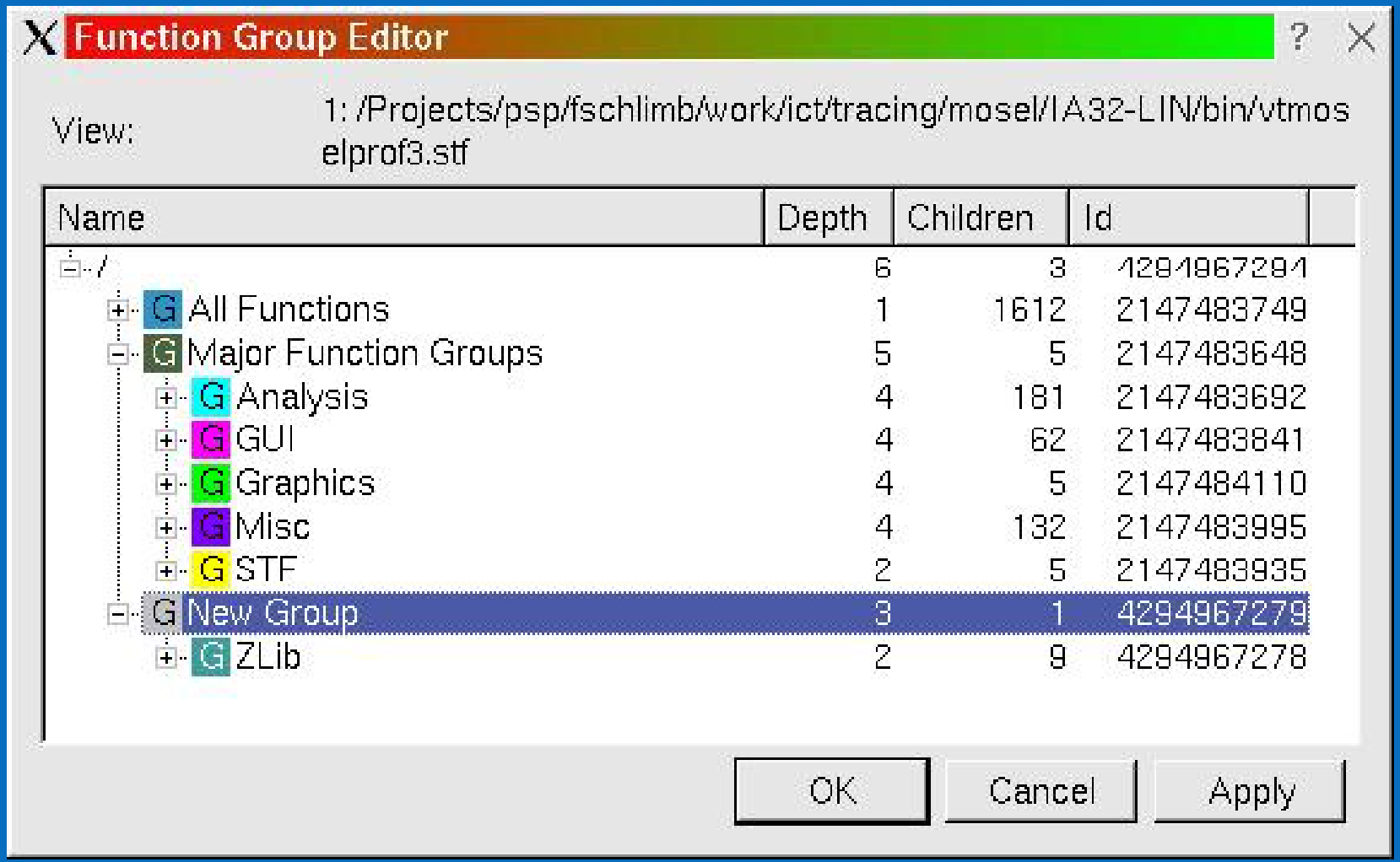
Goal: determine overhead of ZLib in the Intel Trace Analyzer

Create appropriate Function Group(s)

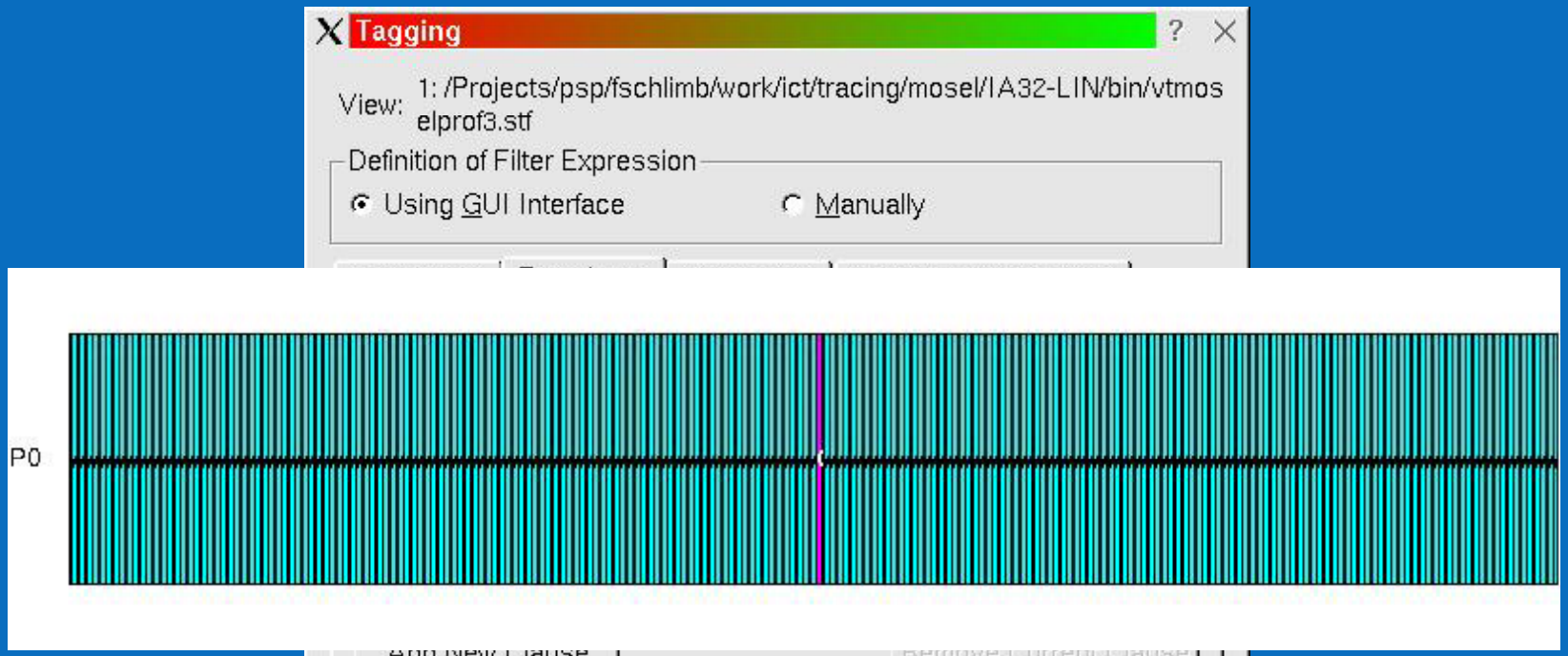
Use Tagging to zoom to representing interval

Use Function Profile

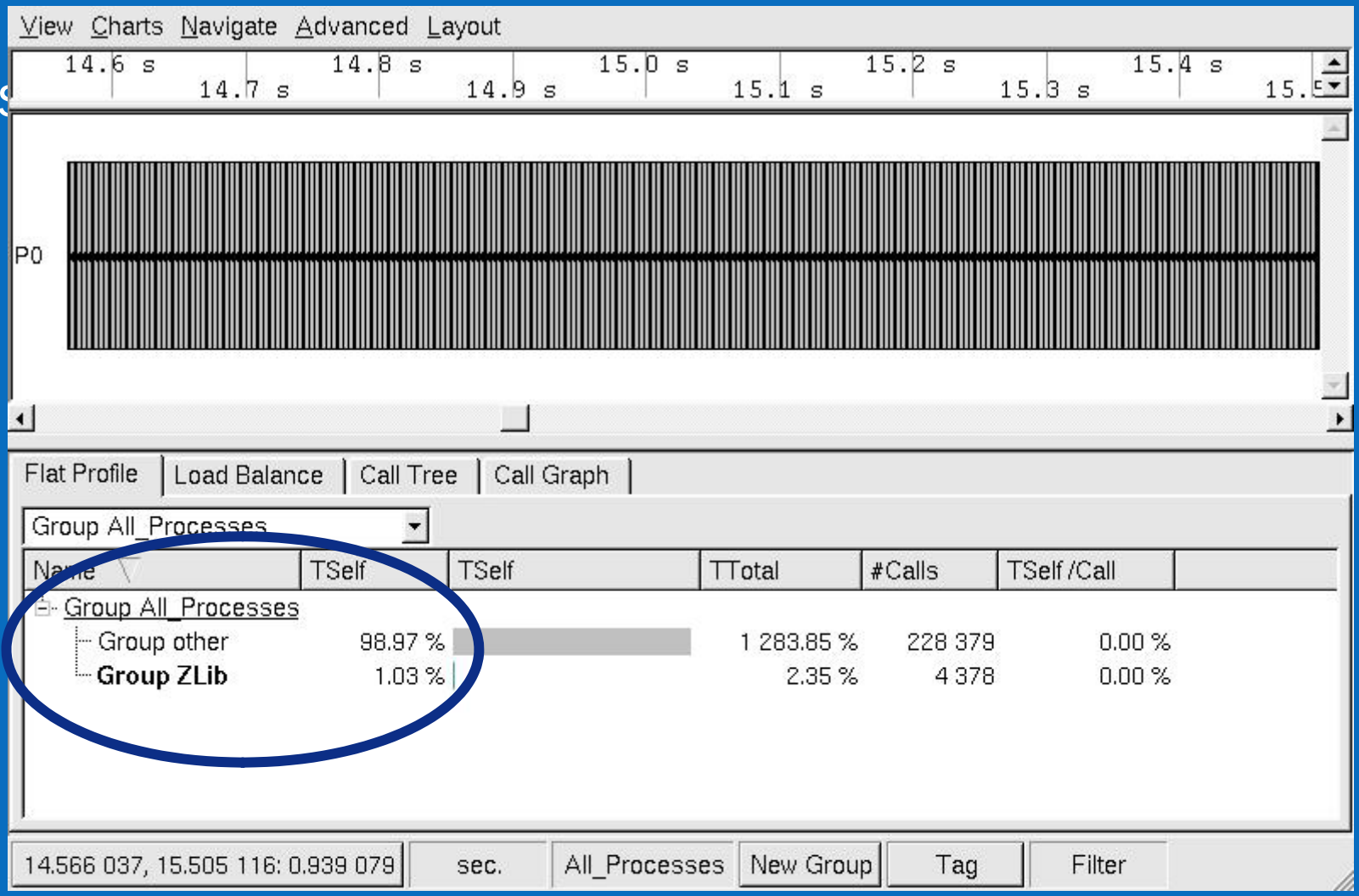
Step 1: Creating Group(s)



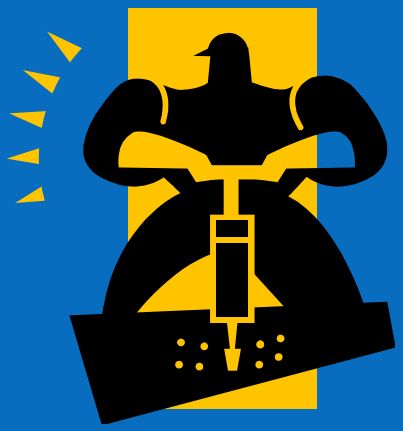
Step 2: Tagging and zooming



Step 3: Determine Overhead



Lab 7 -



Who was calling?

Start with

`traces/ita/ita_fgtl_threaded.stf`

Which caller of STF_WorkStackHistory caused the most costly calls to it?

Function Profile

- Advanced|Function Aggregation->All_Functions
- Search&Find STF_WorkStackHistory
- Show it in Call Graph
- Max TimeSelf/Call of Callers group

Instrumenting Functions

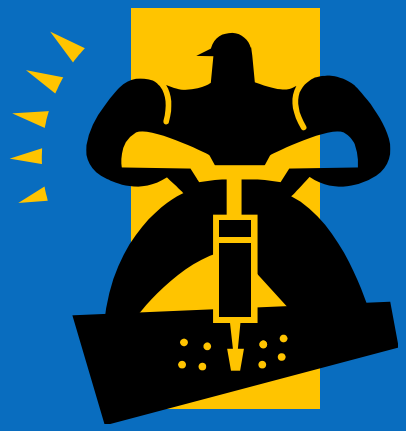
Generic API to instrument functions

Hierarchical grouping (classes)

Insert probes to

- declare function
- enter/leave the function

Lab 8 -



Lab 9 – Basic use of API

Objective: Use of Intel® Trace Collector API to instrument a function

Edit cpi.c to instrument function 'f' as of group 'Application'

- Use `VT_classdef()` and `VT_funcdef()` to declare class/function
- Use `VT_enter()` and `VT_leave()`, to record function begin and end

Compile-link-run and view tracefile in Intel® Trace Analyzer

- Find events 'f' in the timeline, in the function profile and in the call tree display

ITC API: Instrument a function

Allocate a handle for a class name (optionally)

```
int VT_classdef( const char * classname,
                 int * classhandle )
```

classname: name of the class

returns handle (int) referencing this class

Allocate a handle for a state

```
int VT_funcdef( const char * symname,
                int classhandle,
                int * statehandle )
```

symname: name of the symbol

classhandle: handle of the class (see above)

returns handle (int) referencing this function

Solution – Use of API

```
#include <VT.h>
[...]
int application_class;

double f( double a ){
    static int f_state = 0;
    if( f_state == 0 ) {
      VT_funcdef( "f", application_class, &f_state );
    }
    VT_enter( f_state, VT_NOSCL );
    [...]
    VT_leave( VT_NOSCL );
    return (result);
}
int main( int argc, char *argv[])
    [...]
    MPI_Init(&argc,&argv);
    VT_classdef( "Application", &application_class );
```

Lab 10 – Selective Tracing

Objective: Switch tracing off and on.

Edit **cpi.c** to instrument function 'f'.

- use
  `VT_traceoff(), VT_traceon()`
  do not record `MPI_Reduce()`

```
VT_traceoff();
MPI_Reduce(&mypi, &pi, 1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
VT_traceon();
```

Lab 10 – Selective Tracing

Objective: Use of a VT_CONFIG configuration file

Compile-link-run-tracefile and check with Intel® Trace Analyzer.

- check timeline display for TRACE_OFF region where **MPI_Reduce** had been removed.
- Create a file '**my_vt_config**' with the following line in it SYMBOL **MPI_Bcast OFF**
- Set environment variable **VT_CONFIG** to refer to complete path of '`pwd`/my_vt_config'
- Check timeline or statistics … no more data of **MPI_Bcast**

Lab 11 – Instrumentation

Objective: Use of Intel® Trace Collector to instrument a binary

*Note: shared MPI libraries *MUST* be available*

Currently only available for IA32

Instrument binary:

```
itcinstrument --input ./cpi_vt --mpi /opt/intel/mpi_2.0/lib --
   profile --output ./cpi_vt_instr
```

Compile-link - Run the instrumented binary and check tracefile with Intel® Analyzer

Lab 12 – Compare applications

Objective: Analyze and compare solver application with Intel® Trace Analyzer

The laplace equation is solved by two different iterative methods

- Using Jacobi iteration method
- Using Successive Over Relaxation

Open both tracefiles in traceanalyzer (in subdirectory ./traces) - use Window|Tile to compare the traces.

Which solver method seems to be best regarding:

- Amount of MPI?
- Load balance?
- Performance?

Backup

Lab 12 – Create STF single file

Objective: Learn about stftool, use single trace file to simplify transfer of storage

Convert set of stf (structured trace format) files to single file:

```
stftool cpi_itc.stf --logfile-format stfsingle --convert
cpi_itc.single.stf
```

Note: Can degrade performance of ITA

Note: ITC config allows creation of single-STF

Lab 13 – Statistics trace file

Objective: How to generate a tracefile with only statistic information

Generate file my_vt_config

- `# enable statistics gathering`
- `STATISTICS ON`
- `#no need to gather trace data`
- `PROCESS 0:N OFF`

Set environment variable and build example

- `> setenv VT_CONFIG $PWD/my_vt_config`

Run example and look at stats-file cpi_itc.prot

- `> stftool cpi_itc.stf –-print-statistics`


intel®
Leap ahead™