Compiler flags for Optimization

GCC compiler

Tata Consultancy Services

30 April 2013

Speed-space tradeoff

- Some forms of optimization are able to both increase the speed of execution, and reduce the size of the executable
- Other types of optimization may produce faster code at the expense of increasing the executable size, or vice versa.
- *Loop unrolling*

```
for(i=0; i<8; i++) {
  y[i] = i;
}
```

Speed-space tradeoff

- *After optimization*

```
y[0] = 0;
y[1] = 1;
y[2] = 2;
y[3] = 3;
y[4] = 4;
y[5] = 5;
y[6] = 6;
y[7] = 7;
```

- Reduces the execution speed of the executabe but increases the executable size

Speed-space tradeoff

- *Common subexpression elimination*

```
x = cos(v)*(1+sin(u/2)) + sin(w)*(1-sin(u/2));
```

- *After optimization*

```
t = sin(u/2);
x = cos(v)*(1+t) + sin(w)*(1-t);
```

- Improves the performance of the executable but does not increase the size

Speed-space tradeoff

- *Function inlining*

```
double sq(double sq) {
 return x*x; }
```

```
for(i=0; i<1000000; i++) {
 sum+=sq(i+0.5); }
```

- *After optimization*

```
for(i=0; i<1000000; i++) {
 double t = (i+0.5);
 sum+=t*t;
}
```

Scheduling

- Lowest level of optimization done by the compiler
- Optimization w.r.t the order of execution of individual instructions in the CPU, taking into account CPU characteristics such as pipelining.
- Improves speed without increasing the executable size
- But requires more time and more memory for compilation (due to complexity)

Optimization levels in *gcc*

- *gcc* provides a no.of general optimization levels, *0-3*, as well as individual options for specific optimizations.
- **-O0** (default)
  - no optimization done by the compiler.
  - best to use when debugging.
- **-O1**
  - turns on the most common forms of optimization which do not have *speed-space* tradeoffs.
    i.e. with this optimization, the executable will be smaller and faster compared to that with *-O0*.
  - Compiling with *-O1* generally takes less time than that with *-O0* due to reduced amount of data that needs to be processed after optimizations.

Optimization levels in *gcc*

- **-O2**
  - turns on further optimizations without any *speed-space* tradeoff. These include *instruction scheduling*.
  - compiler will take more time and more memory to compile compared to *-O1*.
  - maximum optimization without increase in executable size.
  - best to use for deployment.

- **-O3**
  - turns on more expensive optimizations such as function inlining.
  - *speed-space* tradeoff not guaranteed.
  - It may increase the speed of the executable, but also increase the size.

Optimization levels in *gcc*

- **-funroll-loops**
  - turns on *loop unrolling*.
  - independent of the previous optimizations.
  - will increase the size of the executable.
- **-Os**
  - selects optimizations which reduce the size of the executable.
  - for systems constrained by memory or disk space.
  - in some cases, may run faster due to better cache usage.
- **-ftree-vectorize**
  - enables vectorization.
  - enabled by default at *O3*.

Example programs

```
1  #include <stdio.h>
2
3  double powern (double d, int n)
4  {
5   double x = 1.0;
6   unsigned j;
7   for (j = 1; j <= n; j++)
8    x *= d;
9   return x;
10 }
11
12 int main ()
13 {
14  double sum = 0.0;
15  int i;
16  for (i = 1; i <= 100000000; i++)
17  {
18   sum += powern (i, i % 5);
19  }
20  printf ("sum = %g\n", sum);
21  return 0;
22 }
```

Example programs

```c
#include <stdlib.h>
#include <sys/time.h>
#define N 1000000

int main()
{
 struct timeval start, end, diff;
 int a[N], b[N], c[N];
 int i;
 for(i=0;i<N;i++) {
  a[i] = i;
  b[i] = i+2;
 }

 gettimeofday(&start, NULL);
 for(i=0;i<N;i++)
  c[i] = a[i] + b[i];
 gettimeofday(&end, NULL);

 long long sum=0;
 for(i=0;i<N;i++) {
  sum = sum + c[i];
 }
 printf("%lld\n",sum);
 timersub(&end, &start, &diff);
 printf("Time elapsed in adding the two vectors = %ll
 return 0;
}
```

TATA CONSULTANCY SERVICES

References

- `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
- Brian Gough, *An Introduction to GCC, Chapter 6*.

GNU debugger
GDB
with Multithreaded applications

Tata Consultancy Services

30 April 2013

Agenda

- Introduction to GDB
- Using GDB: Basics
- Using GDB with multithreaded applications
- Examples

Introduction

- *gdb* stands for 'GNU debugger'.
- supports both C and C++.
- Enable *gdb* support by using the *-g* flag while compiling

Compiling C code

prompt$> gcc -g -other_flags source_file -o executable

Basics

- Starting *gdb*: type *gdb* command at the prompt

```
prompt$> gdb [executable]
```

- A prompt like the below one appears

```
(gdb)
```

- Loading a file

```
(gdb) file executable
```

- Running the executable to be debugged

```
(gdb) run
```

Basics

- For help with any command

```
(gdb) help [command]
```

- Setting breakpoints

```
(gdb) break sourcecode.c:6
```

- Breaking at a particular function

```
(gdb) break func_name
```

- Conditinal break

```
(gdb) break  func_name condition
```

Basics

- To continue execution till next break point

```
(gdb) continue
```

- Single stepping: executing only the next line of code

```
(gdb) step
```

- Single stepping without stepping into a subroutine

```
(gdb) next
```

- To list the breakpoints already defined

```
(gdb) info breakpoints
```

Basics

- To delete a breakpoint

```
(gdb) delete breakpoint
```

- To print the value of a variable

```
(gdb) print var
```

- Watching variables: execution breaks whenever value of the variable changes

```
(gdb) watch var
```

Basics

- Trace of function calls: listing the stack frames

```
(gdb) backtrace
```

- To switch to different stack frame

```
(gdb) frame frame_num
```

- Listing the variables in current frame

```
(gdb) info locals
```

Multithreaded applications

- *gdb* supports multithreaded applications (pthreads, openmp).
- Automatically displays message when new threads are created.
- To list all the threads

```
(gdb) info threads
```

- *gdb* attaches a thread number to each thread. The current thread is shown with a *.
- To switch to a different thread

```
(gdb) thread thread_num
```

Multithreaded applications

- To break on a particular thread

```
(gdb) break line_no thread thread_num
```

- Stopping on a particular thread stops all the other threads also.
- To apply a command to multiple threads

```
(gdb) thread apply all [command]
(gdb) thread apply 2 3 [command]
```

- To watch a variable on a particular thread

```
(gdb) watch var thread thread_num
```

TATA CONSULTANCY SERVICES

TATA

Valgrind

April 30, 2013

Outline :

- Introduction
- Tools
- Using Valgrind
- Memcheck
- Cachegrind
- Valgrind with MPI
- Lab

Introduction:

- The Valgrind is a tool suite for debugging and profiling memory
- The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs.
- Valgrind translates the program into a Intermediate Representation then instrument the trace points for debugging.

Valgrind Tools

- **memcheck** is a tool for fine grained memory checking
- **cachegrind** provides the details regarding of instructions executed and cache misses incurred during execution of program
- **callgrind** adds call graph tracing to cachegrind. It can be used to get call counts and inclusive cost for each call happening in program
- **helgrind** spots potential race conditions in your program that use the POSIX pthreads threading primitives. Problems like timing-dependent crashes, deadlocks and other misbehaviour are captured by this tool
- **massif** is a heap profiler. It measures heap memory usages of program

Using Valgrind

- Compile with -g, avoid using optimization flags (-O2 -O3):

```
gcc -g filename.c -o run.out
```

- Run with valgrind

```
valgrind --tool=tool_name <tool_args> ./run.out <program args >
```

- Multithreaded programs

```
valgrind --trace-children=yes --tool=tool_name <tool_args> ./run.out
<program args >
```

- Attaching Debugger

```
valgrind --tool=tool_name <tool_args> --db-attach=yes ./run.out
<program args >
```

Memcheck: Error Check

Use of uninitialised values

An uninitialised value use error is reported when program uses a value which has not been initialised .

- Local variables in procedures which have not been initialized
- The contents of malloc'd blocks, before you write something there.
- In C++, the new operator is a wrapper round malloc

```
int main()
{
        int x;
        printf ("x = %d\n", x);
 }
```

In above code snippet, the undefined value is used inside the printf()

Memcheck: Error Check .....cont....

Overlapping source and destination blocks

Memcheck checks If program copies some data from one memory block to another that have overlapping address. memcpy(), strcpy(), strncpy(), strcat(), strncat() are some functions that can lead such errors.

When a block is freed with an inappropriate deallocation function

In C++ it's important to deallocate memory in a way compatible with how it was allocated.

- If allocated with malloc, calloc, realloc, valloc or memalign, free must be used
- If allocated with new[], must be deallocated with delete[]
- If allocated with new, must be deallocated with delete

Memcheck: Error Check .....cont....

Illegal frees

Memcheck keeps track of the blocks allocated by program with malloc/new, so it can know exactly whether or not the argument to free/delete is legitimate or not.

```
int main()
{
        int *array;
        array= (int *)malloc(100) ;
        free(array);
        free(array);
        return 0;
 }
```

In above code snippet, the array variable has been freed twice

Memcheck: Error Check .....cont....

Illegal read / Illegal write errors

This error occurs when program reads or writes memory at a place which was not allocated

```
int main()
{
        int *array;
        array= (int *)malloc(100) ;
        array[101]=999;
        free(array);
        return 0;
 }
```

In above code snippet, program trying to access the memory of 102th location

Memcheck: Error Check .....cont....

Passing system call parameters with inadequate read/write permissions

Memcheck checks all parameters to system calls. If a system call needs to read from a buffer provided by program, Memcheck checks that the entire buffer is addressible and has valid data..

```
#include <stdlib.h>
 #include <unistd.h>

 int main( void ) {
                        char* arr = malloc(10);
                        (void) write( 1, arr, 10 );
                        return 0;
 }
```

In above code snippet, the arr variable has junk data

CacheGrind :

Cachegrind performs cache and branching profiling. A Cachegrind profile run measures the number of cache misses and branch mispredictions performed by an application.

Three Levels of the cache:

- L1
- L2 /L3

Two type of caches

- Data
- Instruction

```
valgrind  --tool=cachegrind <program name><program args >
```

Sample output of cachegrind:

```
==18232== I   refs:      3,489,832,585
==18232== I1  misses:              972
==18232== L2i misses:              970
==18232== I1  miss rate:          0.00%
==18232== L2i miss rate:          0.00%
==18232==
==18232== D   refs:      2,415,979,570  (2,147,529,619 rd + 268,449,951 wr)
==18232== D1  misses:       33,556,600  (   16,779,049 rd +  16,777,551 wr)
==18232== L2d misses:       33,556,305  (   16,778,795 rd +  16,777,510 wr)
==18232== D1  miss rate:           1.3% (          0.7%   +         6.2%  )
==18232== L2d miss rate:           1.3% (          0.7%   +         6.2%  )
==18232==
==18232== L2 refs:          33,557,572  (   16,780,021 rd +  16,777,551 wr)
==18232== L2 misses:        33,557,275  (   16,779,765 rd +  16,777,510 wr)
==18232== L2 miss rate:            0.5% (          0.2%   +         6.2%  )
```

Valgrind with MPI

- Configure mpi to use valgrind

```
./configure --prefix=/path/to/openmpi --enable-debug  --enable-memchecker
--with-valgrind=/path/to/valgrind
```

- Compile the code

```
mpicc  -g  -o sample.out sample.c
```

- Run the program with valgrind

```
mpirun –np 2 valgrind <memcheck options> ./sample.out <program args>
```

Valgrind with MPI.....cont.....

- Memchecker is implemented on the basis of Memcheck tool from Valgrind, so it takes all the advantages from it.

- it checks all reads and writes of memory, and intercepts calls to malloc/new/free/delete.

- Memchecker is able to detect the user buffer errors in both Non-blocking and One-sided communications for the mpi

Valgrind with MPI.....cont.....

- Accessing buffer under control of non-blocking communication:

```
int buf;
MPI_Irecv(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
 buf = 4711;
MPI_Wait (&req, &status);
```

- Wrong input parameters, e.g. wrongly sized send buffers:

```
char *send_buffer;
send_buffer = malloc(5);
 memset(send_buffer, 0, 5);
MPI_Send(send_buffer, 10, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
```

Valgrind with MPI.....cont.....

- Usage of the uninitialized MPI_Status field in MPI_ERROR structure: (The MPI-1 standard defines the MPI ERROR-field to be undefined for single-completion calls such as MPI Wait or MPI Test)

```
MPI_Wait(&request, &status);
if (status.MPI_ERROR != MPI_SUCCESS)
return ERROR;
```

- Accessing window under control of one-sided communication:

```
MPI_Get(A, 10, MPI_INT, 1, 0, 1, MPI_INT, win);
 A[0] = 4711;
MPI_Win_fence(0, win);
```

Suppressing Error:

Specify pattern to not to be reported as alarm from valgrind:

```
mpirun -np 2 valgrind  –suppressions=$PREFIX/share/openmpi/openmpi-valgrind.supp
```

References:

- http://icl.cs.utk.edu/open-mpi/faq/?category=debugging
- http://valgrind.org/docs/manual/manual.html

Lab:

- Run Sample Programs and check various Memcheck Errors

1. Memory leaks in sample pro
2. Illegal Read/Write
3. Uninitialized values
4. Illegal free

- Check the cachegrind output for sample program

1. Various cache sizes?
2. Which cache is missed most?
3. It is data miss or branching?
4. Speed up gained on optimizing for the cache?

Valgrind path = /opt/software/hpcapps/Tools/valgrind-3.8.1/bin/valgrind

Profiling Tools : Scalasca

Parallelization and Optimization Group
TATA Consultancy Services, SahyadriPark Pune, India

May 5, 2013

Agenda

1. Introduction to Scalasca
2. Phases in Scalasca
3. Analyzing Reports

Introduction to Scalasca:

Scalasca

1. SCalable Analysis of LArge SCale Applications
2. Capable of measuring and analyzing parallel program behavior during execution
3. Supports measurement and analysis of the MPI, OpenMP and Hybrid MPI+OpenMP
4. Supports HPC applications written in C, C++ and Fortran

Introduction to Scalasca:

Scalasca

1. SCalable Analysis of LArge SCale Applications
2. Capable of measuring and analyzing parallel program behavior during execution
3. Supports measurement and analysis of the MPI, OpenMP and Hybrid MPI+OpenMP
4. Supports HPC applications written in C, C++ and Fortran

Usage :

1. scalasca command with appropriate arguments.

Phases in Scalasca :

Use of Scalasca involves three phases:

1. Program Instrumentation
2. Execution Measurement Collection and Analysis
3. Analysis Report Examination

Phase 1 : Program Instrumentation

1. Used to make measurements

Phase 1 : Program Instrumentation

1. Used to make measurements
2. This phase is also called as skin

Phase 1 : Program Instrumentation

1. Used to make measurements
2. This phase is also called as skin

Usage :

- scalasca -instrument (or skin) compilation-commands [compiler flags]

Phase 1 : Program Instrumentation

1. Used to make measurements
2. This phase is also called as skin

Usage :

- scalasca -instrument (or skin) compilation-commands [compiler flags]

Example :

- scalasca -instrument mpicc mpi_send.c -o mpi_send or
- skin mpicc mpi_send.c -o mpi_send

Phase 2 : Execution Measurement Collection and Analysis

Phase 2 : Execution Measurement Collection and Analysis

1. This phase collects the run time information and stores in epik_XX_XX_XX folder used in third phase for analysis

Phase 2 : Execution Measurement Collection and Analysis

1. This phase collects the run time information and stores in epik_XX_XX_XX folder used in third phase for analysis
2. This phase is also called as scan

Phase 2 : Execution Measurement Collection and Analysis

1. This phase collects the run time information and stores in epik_XX_XX_XX folder used in third phase for analysis
2. This phase is also called as scan

The folder epik_XX_XX_XX contains :

Phase 2 : Execution Measurement Collection and Analysis

1. This phase collects the run time information and stores in epik_XX_XX_XX folder used in third phase for analysis
2. This phase is also called as scan

The folder epik_XX_XX_XX contains :

- epik.conf contains analysis measurement configuration details
- epik.log contains execution time logs
- epik.path contains execution path
- epitome.cube contains measurement information

Phase 2 : Execution Measurement Collection and Analysis

1. This phase collects the run time information and stores in epik_XX_XX_XX folder used in third phase for analysis
2. This phase is also called as scan

The folder epik_XX_XX_XX contains :

- epik.conf contains analysis measurement configuration details
- epik.log contains execution time logs
- epik.path contains execution path
- epitome.cube contains measurement information

Usage :

- scalasca -analyze (or scan) execute_command [compiler flags] executable [args]

Phase 2 : Execution Measurement Collection and Analysis

1. This phase collects the run time information and stores in epik_XX_XX_XX folder used in third phase for analysis
2. This phase is also called as scan

The folder epik_XX_XX_XX contains :

- epik.conf contains analysis measurement configuration details
- epik.log contains execution time logs
- epik.path contains execution path
- epitome.cube contains measurement information

Usage :

- scalasca -analyze (or scan) execute_command [compiler flags] executable [args]

Example :

- scalasca -analyze mpirun -np 4 ./mpi_send or
- scan mpirun -np 4 ./mpi_send

Phase 3 : Analysis Report Examination

Phase 3 : Analysis Report Examination

1. Uses epik_XX_XX_XX folder contents to generate reports.
2. Two ways to generate reports

Phase 3 : Analysis Report Examination

1. Uses epik_XX_XX_XX folder contents to generate reports.
2. Two ways to generate reports
   - Textual Report
   - Graphical User Interface

Phase 3 : Analysis Report Examination : Textual

Phase 3 : Analysis Report Examination : Textual

Uses epitome.cube file

Report contains following regions and are classified into

- ANY : Aggregate of all regions
- MPI : Pure MPI library functions
- OMP : Pure OpenMP functions/regions
- COM : Combined user regions calling OpenMP/MPI, directly or indirectly
- USR : User regions with purely local computation not containing MPI or OpenMP.

Phase 3 : Analysis Report Examination : Textual

Uses epitome.cube file

Report contains following regions and are classified into

- ANY : Aggregate of all regions
- MPI : Pure MPI library functions
- OMP : Pure OpenMP functions/regions
- COM : Combined user regions calling OpenMP/MPI, directly or indirectly
- USR : User regions with purely local computation not containing MPI or OpenMP.

Usage :

- cube3_score epik_XX_XX_XX/epitome.cube

Phase 3 : Analysis Report Examination : Textual

Uses epitome.cube file

Report contains following regions and are classified into

- ANY : Aggregate of all regions
- MPI : Pure MPI library functions
- OMP : Pure OpenMP functions/regions
- COM : Combined user regions calling OpenMP/MPI, directly or indirectly
- USR : User regions with purely local computation not containing MPI or OpenMP.

Usage :

- cube3_score epik_XX_XX_XX/epitome.cube

Example :

- cube3_score epik_mpi_send_4_sum/epitome.cube

Phase 3 : Analysis Report Examination : Textual

Phase 3 : Analysis Report Examination : Textual

Sample Report :

Phase 3 : Analysis Report Examination : Textual

Sample Report :

flt	type	max_tbc	time	%	region
	ANY	3186	11.64	100.00	(summary) ALL
	MPI	3114	10.11	86.79	(summary) MPI
	COM	24	0.85	7.33	(summary) COM
	USR	0	0.00	0.00	(summary) USR
	MPI	2100	0.01	0.13	MPI_Send
	MPI	966	0.19	1.63	MPI_Recv
	EPK	48	0.68	5.88	TRACING
	COM	24	0.85	7.33	main
	MPI	24	9.88	84.83	MPI_Init
	MPI	24	0.02	0.20	MPI_Finalize

Figure: Textual Sample Report

- total_tbc : Aggregate trace size
- max_tbc : Largest process trace

Phase 3 : Analysis Report Examination : Graphical

1. Uses CUBE Viewer
2. This phase is also called as square

Phase 3 : Analysis Report Examination : Graphical

1. Uses CUBE Viewer
2. This phase is also called as square

Usage :

- scalasca -examine (or square) execute_command [compiler flags] executable [args]

Phase 3 : Analysis Report Examination : Graphical

1. Uses CUBE Viewer
2. This phase is also called as square

Usage :

- scalasca -examine (or square) execute_command [compiler flags] executable [args]

Example :

- scalasca -examine epik_mpi_send_4_sum or
- square epik_mpi_send_4_sum

Phase 3 : Analysis Report Examination : Graphical

Sample Report : Matrix Multiplication of size 512 x 512 using MPI

Phase 3 : Analysis Report Examination : Graphical

Sample Report : Matrix Multiplication of size 512 x 512 using MPI

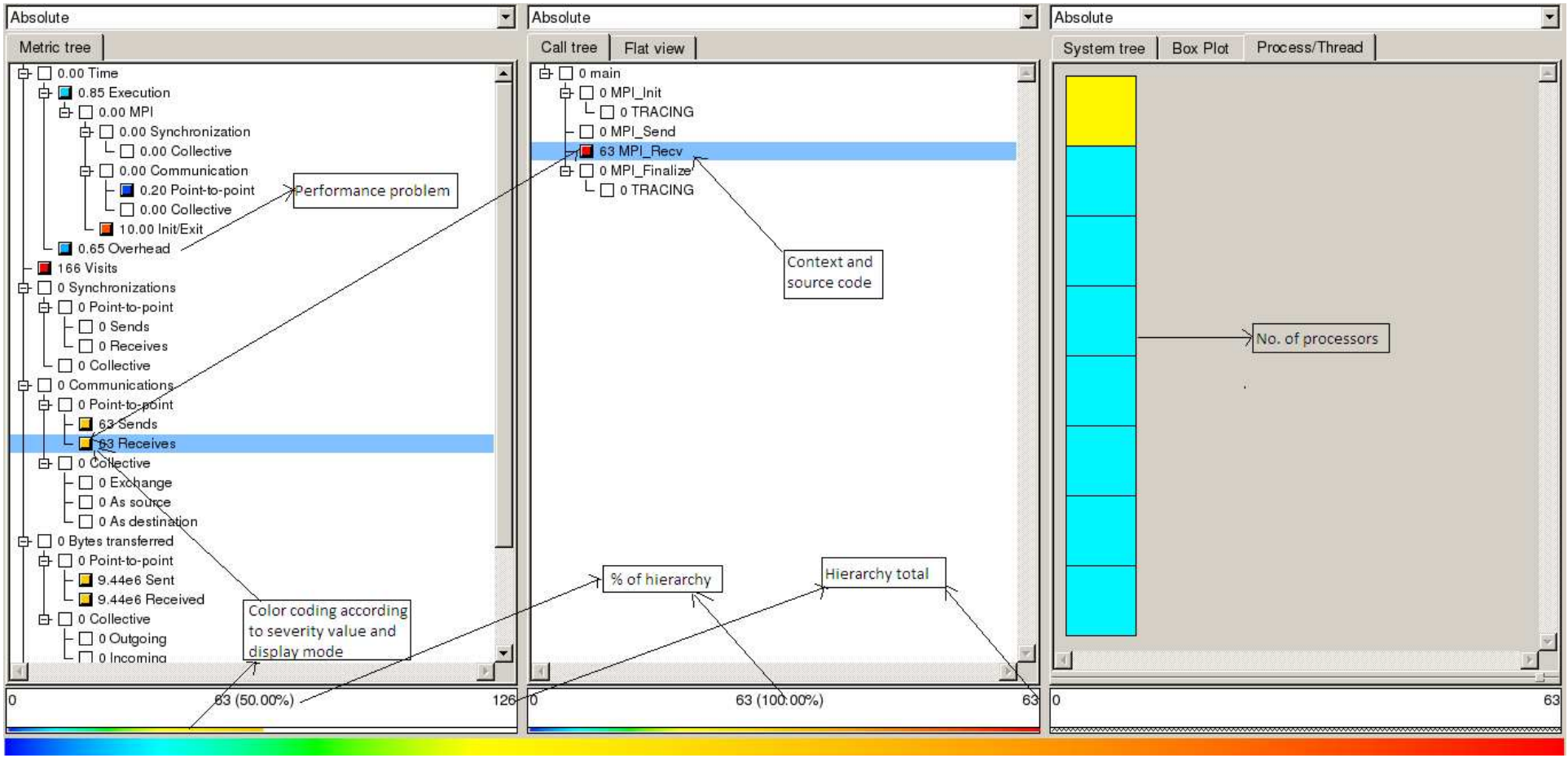


Figure: Using 8 Processors

Phase 3 : Analysis Report Examination : Graphical

Sample Report : Matrix Multiplication of size 512 x 512 using OpenMP

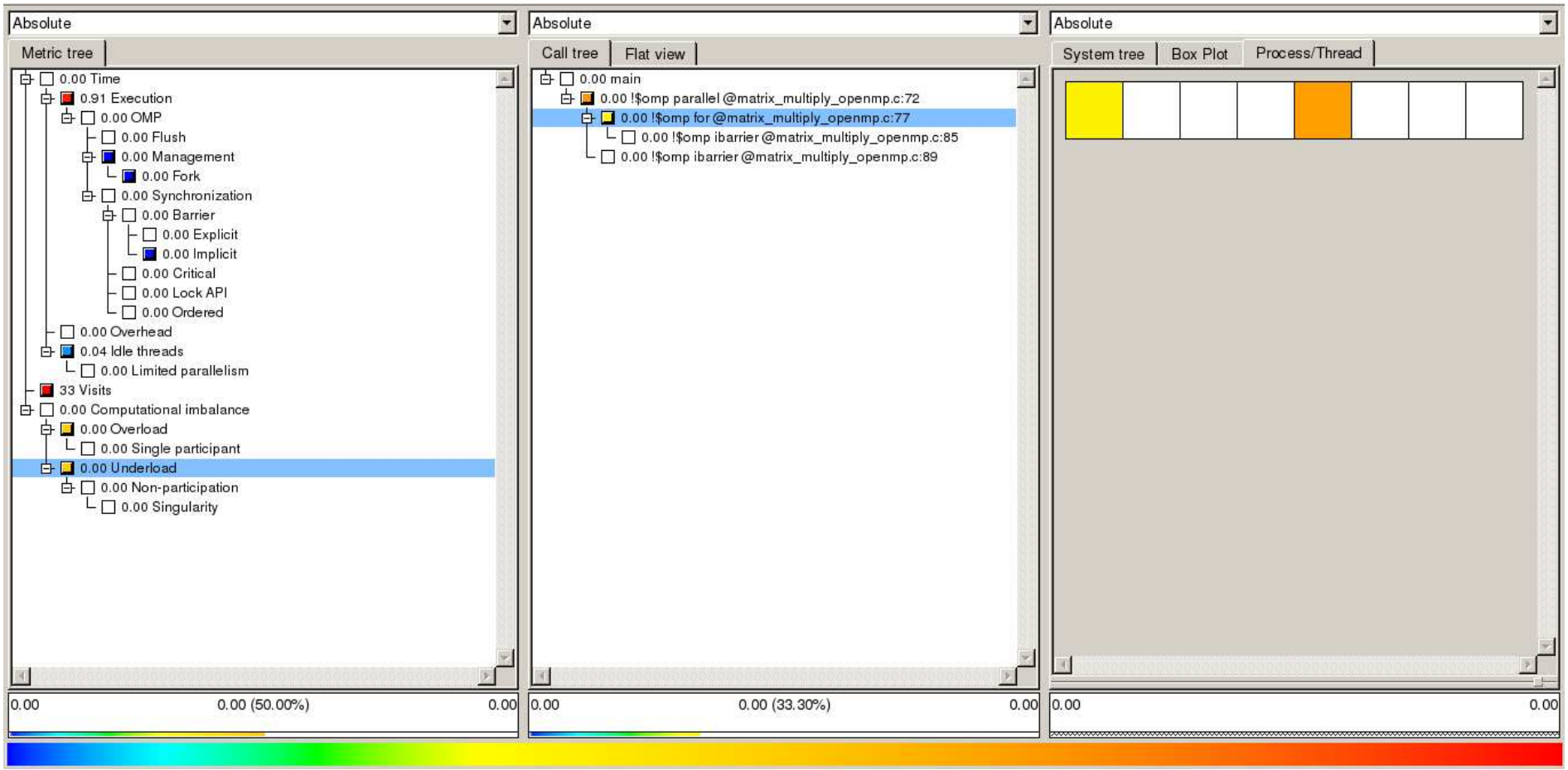


Figure: Using 8 Threads

Analyzing Reports

Analyzing Reports

Simple MPI_Send and MPI_Recv Example

Analyzing Reports

Simple MPI_Send and MPI_Recv Example

```
if ( rank == 0 )
   dest = 1;
   outmsg = 100;
   MPI_Send(&outmsg,1,MPI_DOUBLE,dest,tag,xxx);

else if ( rank == 1 )
   source = 0;
   MPI_Recv(&inmsg,1,MPI_DOUBLE,source,tag,xxx,xxx);
```

Analyzing Reports

Simple MPI_Send and MPI_Recv Example

```
if ( rank == 0 )
   dest = 1;
   outmsg = 100;
   MPI_Send(&outmsg,1,MPI_DOUBLE,dest,tag,xxx);

else if ( rank == 1 )
   source = 0;
   MPI_Recv(&inmsg,1,MPI_DOUBLE,source,tag,xxx,xxx);
```

Compile and Execute using Scalasca :

- scalasca -instrument (or skin) mpicc mpi_send.c -o mpi_send
- scalasca -analyze (or scan) mpirun -np 4 ./mpi_send
- scalasca -examine (or square) epik_mpi_send_4_sum (For Graphical)
- cube3_score epik_mpi_send_4_sum/epitome.cube (For Textual)

Analyzing Reports : Textual

Analyzing Reports : Textual

```
Reading epik_mpi_send_4_sum/epitome.cube... done.
Estimated aggregate size of event trace (total_tbc): 768 bytes
Estimated size of largest process trace (max_tbc):   218 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
 or reduce requirements using file listing names of USR regions to be filtered.)

flt  type       max_tbc         time       % region
      ANY           218         4.14  100.00 (summary) ALL
      MPI           146         4.08   98.71 (summary) MPI
      COM            24         0.00    0.00 (summary) COM
      USR             0         0.00    0.00 (summary) USR
```

Figure: Textual Report

Analyzing Reports : Textual

Analyzing Reports : Textual

To see individual function call report

- square -s epik_mpi_send_4_sum

```
Reading ./epik_mpi_send_4_sum/summary.cube.gz... done.
Estimated aggregate size of event trace (total_tbc): 768 bytes
Estimated size of largest process trace (max_tbc):   218 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
 or reduce requirements using file listing names of USR regions to be filtered.)

INFO: Score report written to ./epik_mpi_send_4_sum/epik.score
[727901@01HW534197 IUCAA]$ cat epik_mpi_send_4_sum/epik.score
flt  type       max_tbc         time      % region
      ANY           218         4.14  100.00 (summary) ALL
      MPI           146         4.08   98.71 (summary) MPI
      COM            24         0.00    0.00 (summary) COM
      USR             0         0.00    0.00 (summary) USR

      MPI            50         0.00    0.00 MPI_Send
      EPK            48         0.05    1.29 TRACING
      MPI            46         0.00    0.00 MPI_Recv
      COM            24         0.00    0.00 main
      MPI            24         4.08   98.70 MPI_Init
      MPI            24         0.00    0.00 MPI_Comm_size
      MPI            24         0.00    0.00 MPI_Comm_rank
      MPI            24         0.00    0.01 MPI_Finalize
```

Figure: Textual Report - Individual Function Call

Analyzing Reports : Graphical

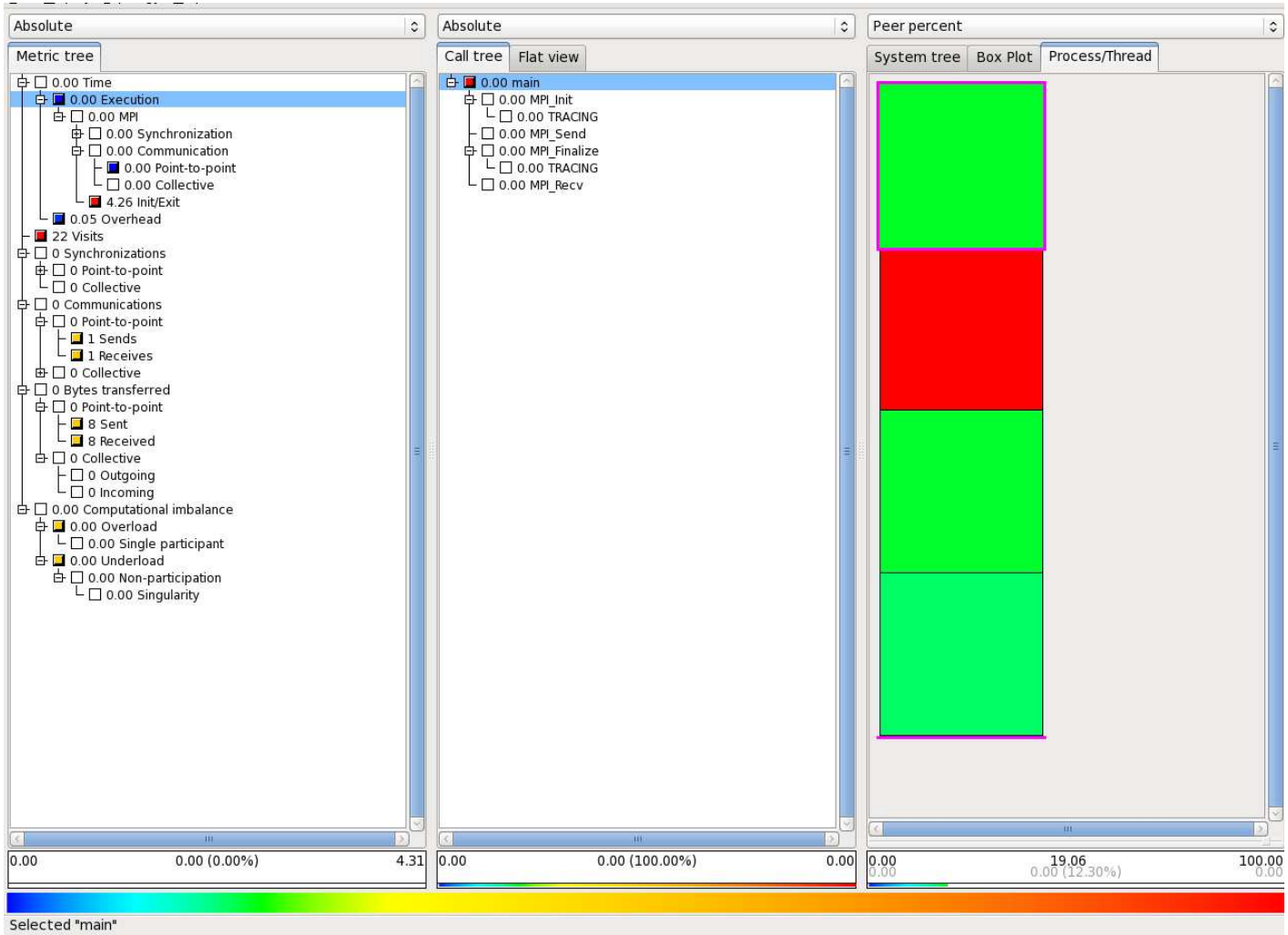


Figure: Graphical User Interface Report - main()

Thank You

Profiling Tools : gprof

Parallelization and Optimization Group
TATA Consultancy Services, SahyadriPark Pune, India

April 29, 2013

Introduction of Profilers:

Why profiling?

1. To understand program behaviour
2. Allows you to learn where your program spent its time
3. Which functions called which other functions while it was executing

Introduction of Profilers:

Why profiling?

1. To understand program behaviour
2. Allows you to learn where your program spent its time
3. Which functions called which other functions while it was executing

Profiler results may contains

1. Event-based
2. Statistical
3. Instrumented
4. Sampling
5. Simulation etc

Introduction of Profilers :

How to do ?

1. Compile and Link your program with profiling enabled

Introduction of Profilers :

How to do ?

1. Compile and Link your program with profiling enabled
2. Execute your program to generate a profile data file

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage :

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg
2. Execute your program. This generates gmon.out executable file required by 'gprof'
   Usage :

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg
2. Execute your program. This generates gmon.out executable file required by 'gprof'
   Usage : ./hello

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg
2. Execute your program. This generates gmon.out executable file required by 'gprof'
   Usage : ./hello
3. Now execute your program along with 'gprof' to generate basic profiler data
   Usage :

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg
2. Execute your program. This generates gmon.out executable file required by 'gprof'
   Usage : ./hello
3. Now execute your program along with 'gprof' to generate basic profiler data
   Usage : gprof ./hello

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg
2. Execute your program. This generates gmon.out executable file required by 'gprof'
   Usage : ./hello
3. Now execute your program along with 'gprof' to generate basic profiler data
   Usage : gprof ./hello *gmon.out*

GNU Profiler (gprof) :

Basic profiler in Linux and comes with OS.
How to Compile?

1. Use -pg flag in addition to your usual options
   Usage : gcc -g hello.c -o hello -pg
2. Execute your program. This generates gmon.out executable file required by 'gprof'
   Usage : ./hello
3. Now execute your program along with 'gprof' to generate basic profiler data
   Usage : gprof ./hello *gmon.out>profiler_data*

GNU Profiler (gprof) :

What is inside profiler_data file?

GNU Profiler (gprof) :

What is inside profiler_data file?
Contains Two Tables

GNU Profiler (gprof) :

What is inside profiler_data file?
Contains Two Tables

1. Flat profile:
   Shows the total amount of time your program spent executing each function

GNU Profiler (gprof) :

What is inside profiler_data file?
Contains Two Tables

1. Flat profile:
   Shows the total amount of time your program spent executing each function
2. Call graph:
   Shows how much time was spent in each function and its children

GNU Profiler Example : Merge Sort

```
#include <stdio.h>
{
...
sort(A, sizeA); // Sort Array A
sort(B, sizeB); // Sort Array B
merge( A, B, sizeA ,sizeB ); //Merge two arrays
...
}
sort() {
//Some Code
}
merge(){
//Some Code
}
```

Merge Sort : Flat Profile

Each sample counts as 0.01 seconds.
no time accumulated

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	2	0.00	0.00	sort
0.00	0.00	0.00	1	0.00	0.00	merge

Merge Sort : Call Graph

```
granularity: each sample hit covers 4 byte(s) no time

index % time    self  children    called     name
                0.00    0.00       2/2          main [6]
[1]      0.0    0.00    0.00         2          sort [1]
-----------------------------------------------------------

                0.00    0.00       1/1          main [6]
[2]      0.0    0.00    0.00         1          merge [2]
-----------------------------------------------------------

Index by function name

   [2] merge                   [1] sort
```

Thank You

MPI Parallel Environment - MPE

5/3/13

About MPE

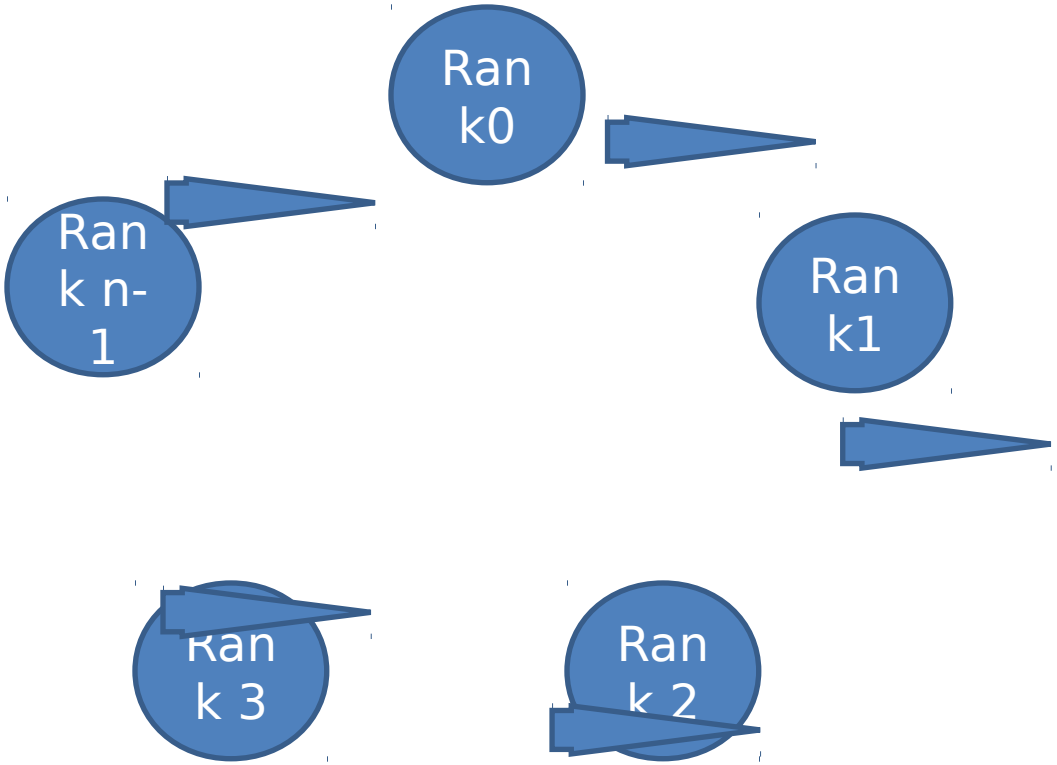
- Tool for performance visualization of MPI programs.
- Set of profiling libraries
- Compiler wrapper (mpecc & mpefc)
- Jumpshot: Logfile viewer
- Work with
  - OpenMPI, LAM/MPI, MPICH-1, MPICH2
  - Commercial MPI implementations available on IBM's AIX, BG/L, BG/P, NEC SX-8, Cray X1E, Cray XT4, HP-MPI
- The latest version is MPE2, mpe2-1.3.0

Using MPE

- mpecc filename.c –o filename –mpilog
- Mpirun –np <n_cpus> ./filename
- Successful execution creates filename.clog2
- Convert it to slog2 format
- clog2TOslog2 filename.clog2
- Creates filename.slog2
- View this log file using jumpshot-4

Examples

- Using blocking Send & Recv
- Using non-blocking Isend & Irecv



Ring example with blocking send/recv

```
inittime = MPI_Wtime();
 if ( taskid == 0 )
{
     MPI_Send(sendbuff,buffsize,MPI_DOUBLE,taskid+1,0,MPI_COMM_WORLD);
     MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,ntasks-
1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
     recvtime = MPI_Wtime();
}
else if( taskid == ntasks-1 )
{
     MPI_Send(sendbuff,buffsize,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
     MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,taskid-
1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
     recvtime = MPI_Wtime();
}
else
{
     MPI_Send(sendbuff,buffsize,MPI_DOUBLE,taskid+1,0,MPI_COMM_WORLD);
     MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,taskid-
1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
     recvtime = MPI_Wtime();
}
     MPI_Barrier(MPI_COMM_WORLD);
     totaltime = MPI_Wtime() - inittime;
```

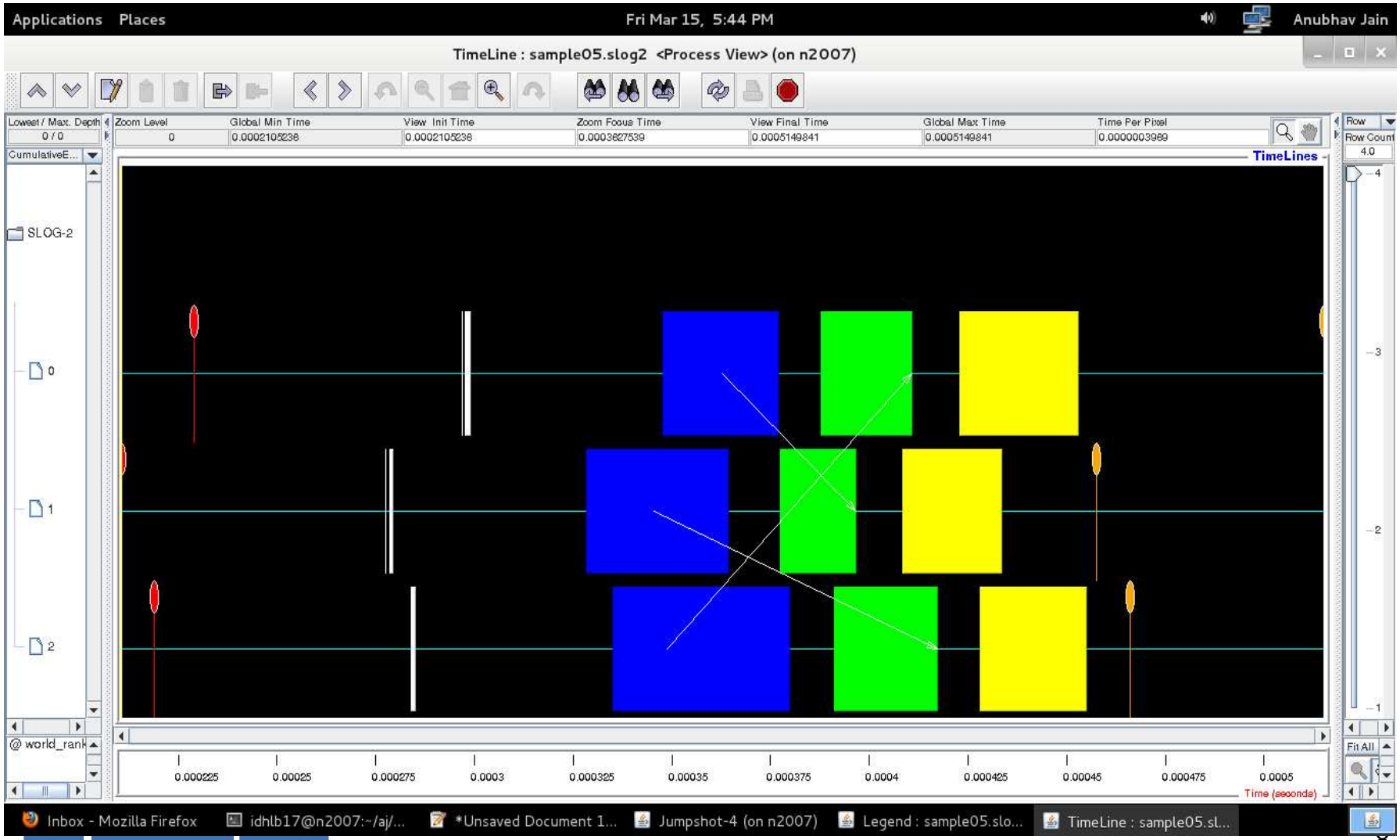
Ring example with non blocking send/recv

```
inittime = MPI_Wtime();
if ( taskid == 0 )
{
 MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,taskid+1,0,MPI_COMM_WORLD,&send_request);
 MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,ntasks-1,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
 recvtime = MPI_Wtime();
}
else if( taskid == ntasks-1 )
{
 MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&send_request);
 MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,taskid-1,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
 recvtime = MPI_Wtime();
}
else
{
 MPI_Isend(sendbuff,buffsize,MPI_DOUBLE,taskid+1,0,MPI_COMM_WORLD,&send_request);
 MPI_Irecv(recvbuff,buffsize,MPI_DOUBLE,taskid-1,MPI_ANY_TAG,MPI_COMM_WORLD,&recv_request);
 recvtime = MPI_Wtime();
}
MPI_Wait(&send_request,&status);
MPI_Wait(&recv_request,&status);
totaltime = MPI_Wtime() - inittime
```

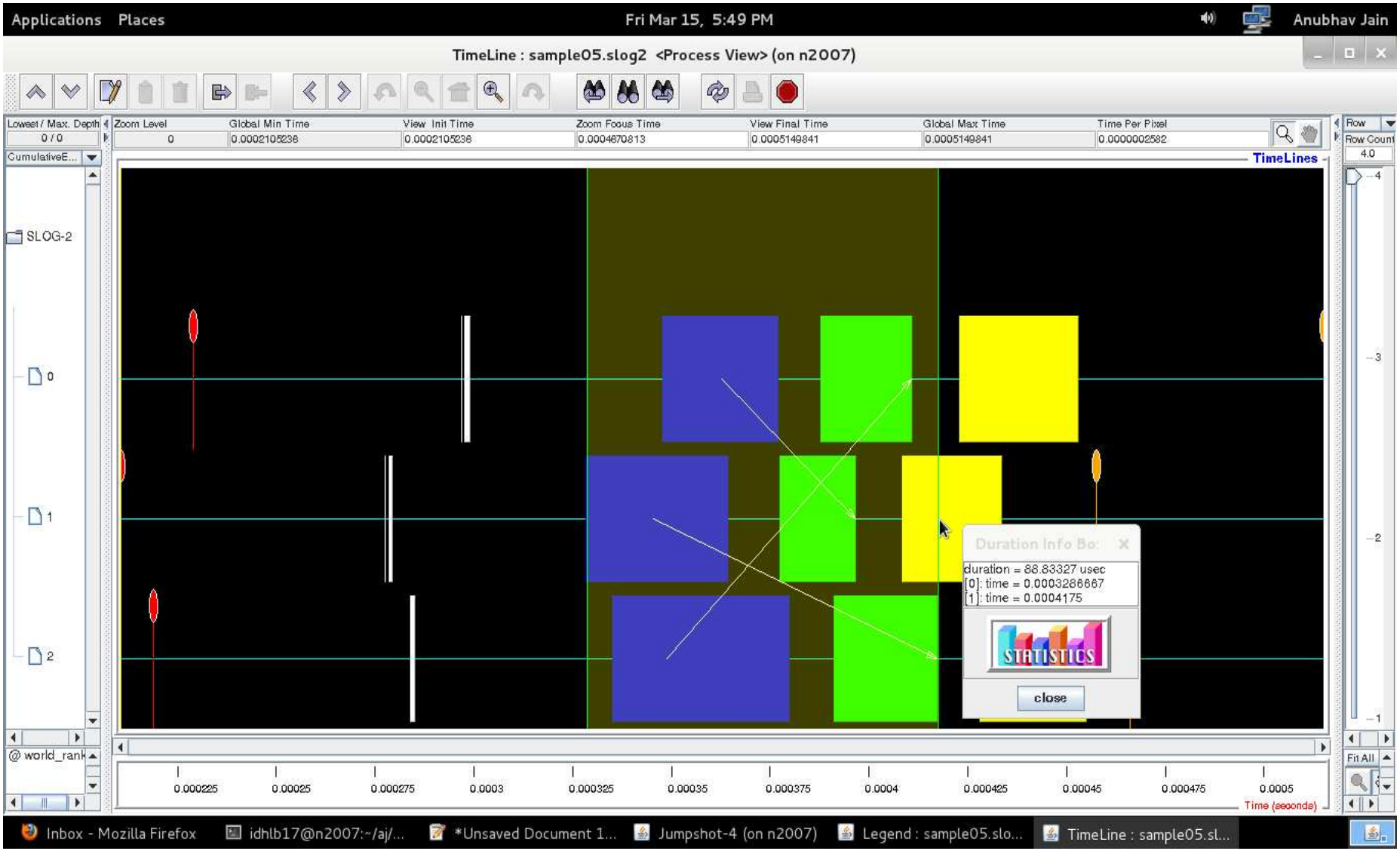
Performance

- MPI Send and Recv
  - Communication time : 0.000141 seconds
- MPI Isend and Irecv
  - Communication time : 0.000092 seconds

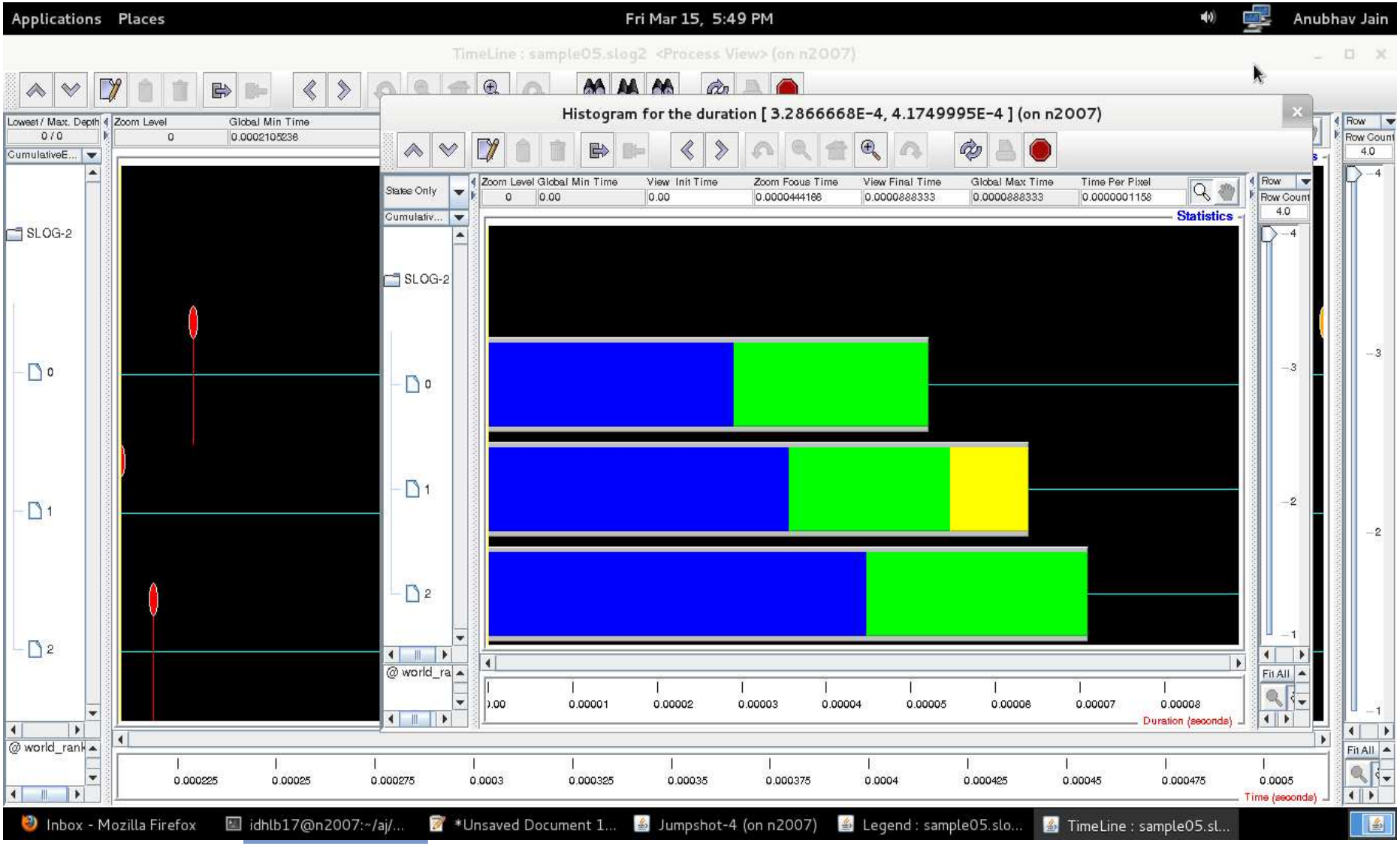
Visualizing with Jumpshot



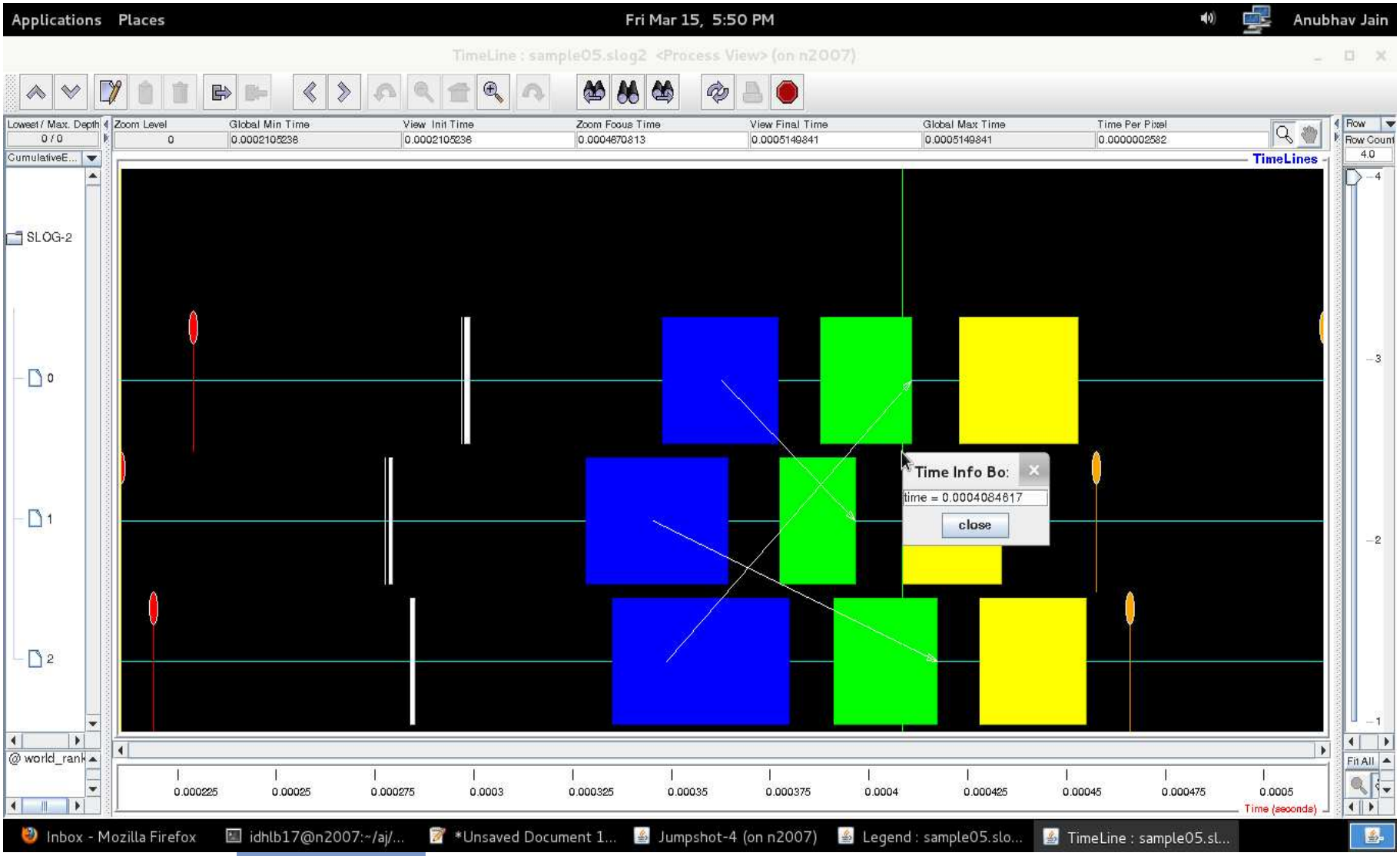
Statistics of chosen range



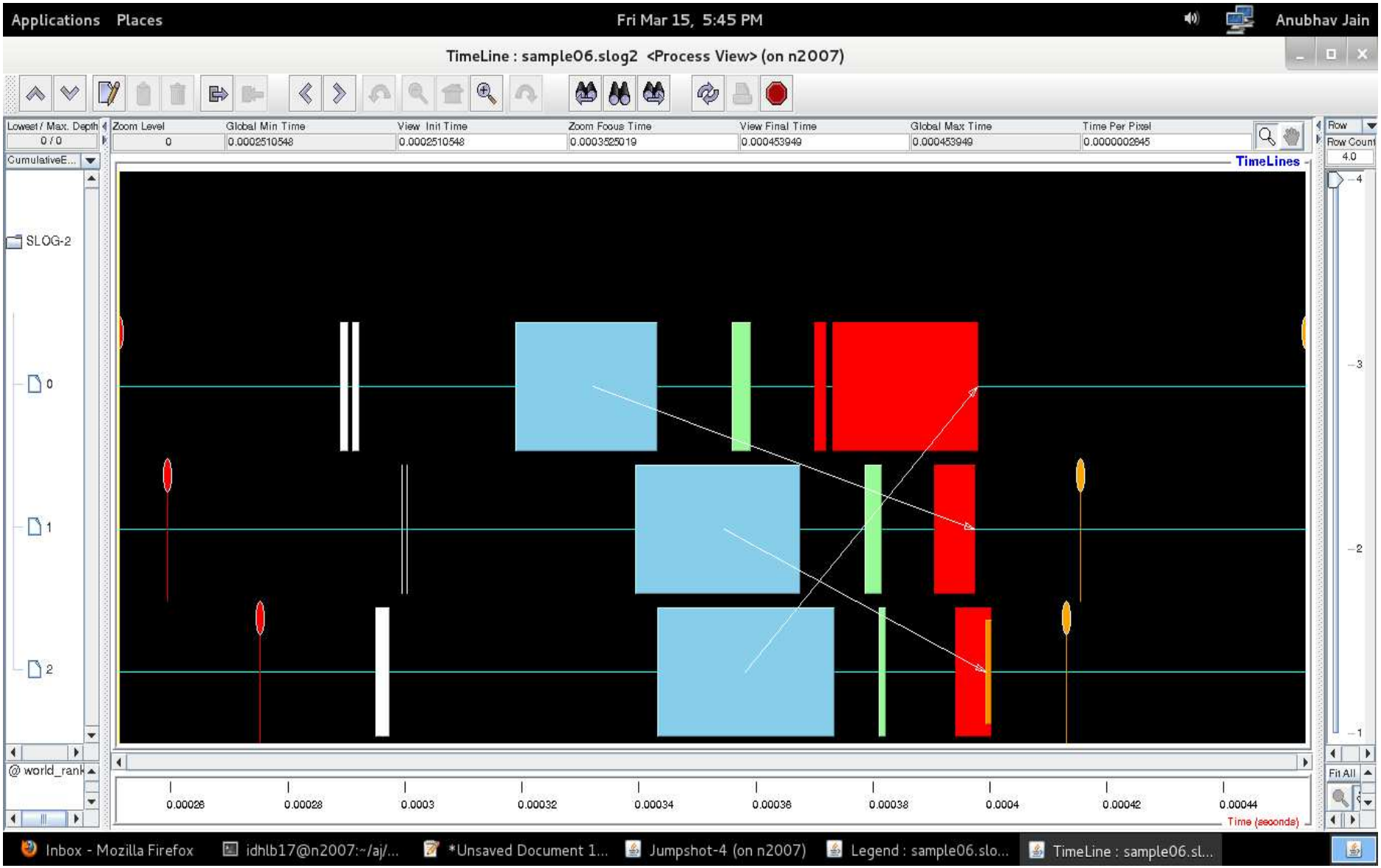
Histogram


Applications Places
Fri Mar 15, 5:49 PM
Anubhav Jain
Histogram for the duration [3.2866668E-4, 4.1749995E-4] (on n2007)
Statistics
Duration (seconds)
Time (seconds)
Inbox - Mozilla Firefox
idhlb17@n2007:~/aj/...
*Unsaved Document 1...
Jumpshot-4 (on n2007)
Legend : sample05.slo...
TimeLine : sample05.sl...

Time line

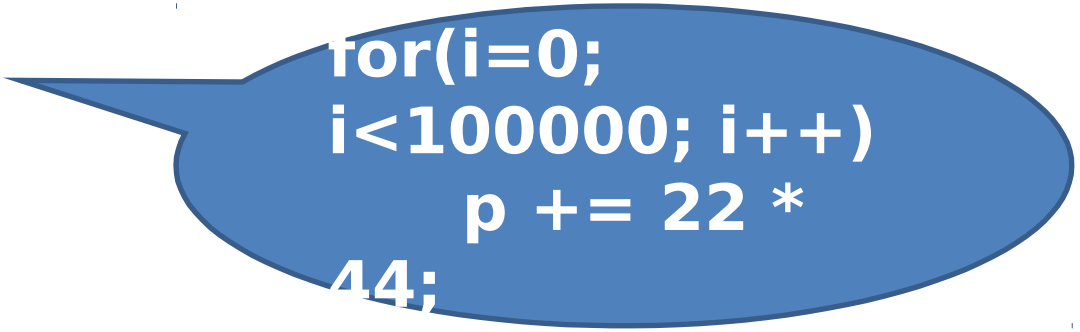

Applications Places
Fri Mar 15, 5:50 PM
Anubhav Jain
TimeLine : sample05.slog2 <Process View> (on n2007)
Lowest / Max. Depth
Zoom Level
Global Min Time
View Init Time
Zoom Focus Time
View Final Time
Global Max Time
Time Per Pixel
TimeLines
SLOG-2
Time Info Bo:
time = 0.0004084617
close
Row
Row Count
Fit All
0.000225
0.00025
0.000275
0.0003
0.000325
0.00035
0.000375
0.0004
0.000425
0.00045
0.000475
0.0005
Time (seconds)
Inbox - Mozilla Firefox
idhlb17@n2007:~/aj/...
*Unsaved Document 1...
Jumpshot-4 (on n2007)
Legend : sample05.slo...
TimeLine : sample05.sl...

Non blocking Send/Recv



Let’s add computation in between Send and Recv

Blocking send recv



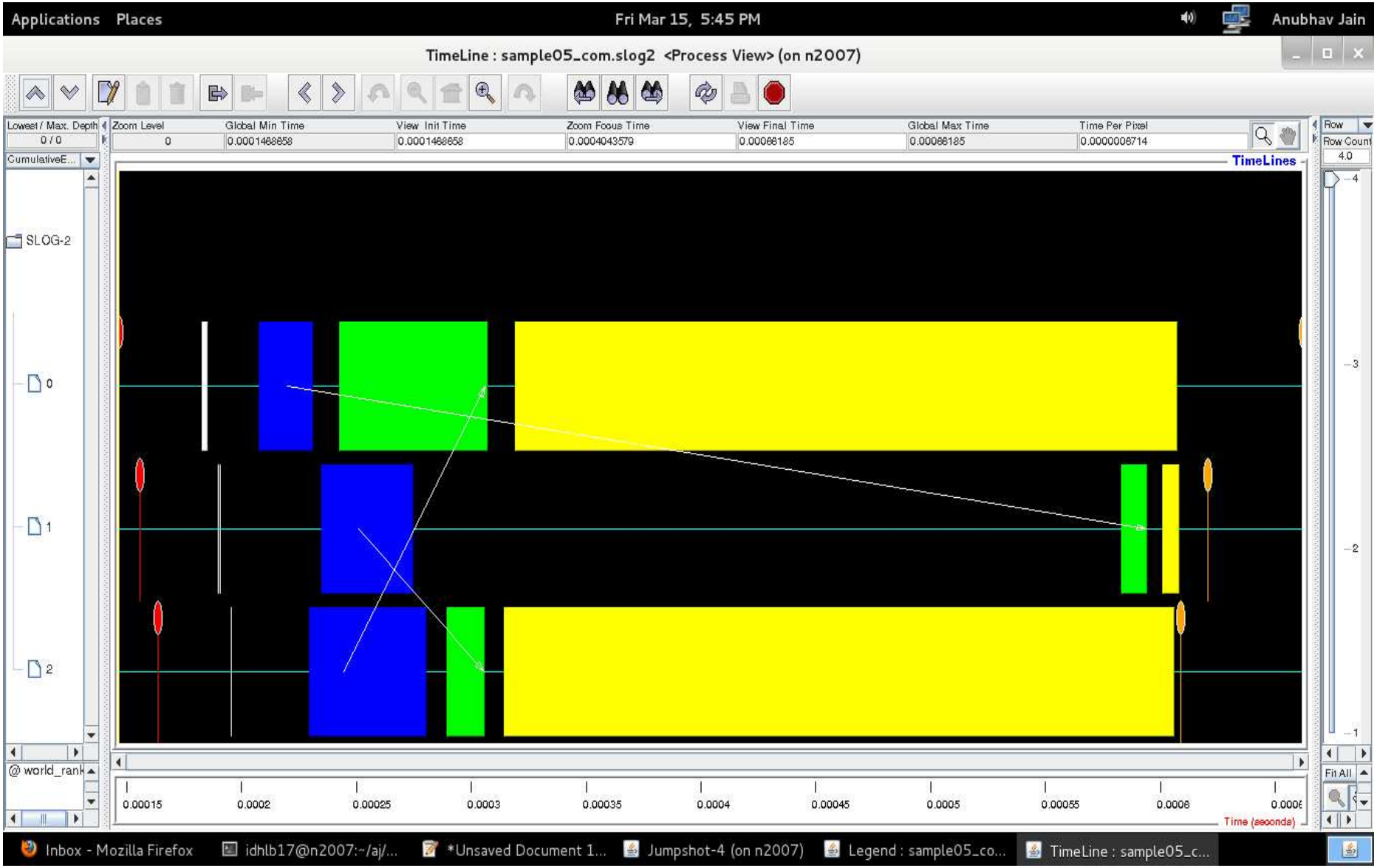

```
inittime = MPI_Wtime();
else
{
     MPI_Send(sendbuff,buffsize,MPI_DOUBLE,taskid+1,0,MPI_COMM_WORLD);
     for(I = 0; I < 100000; i++)
          p += 22*44;
     MPI_Recv(recvbuff,buffsize,MPI_DOUBLE,taskid-
1,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
     recvtime = MPI_Wtime();
}
     MPI_Barrier(MPI_COMM_WORLD);
     totaltime = MPI_Wtime() - inittime;
```

Performance

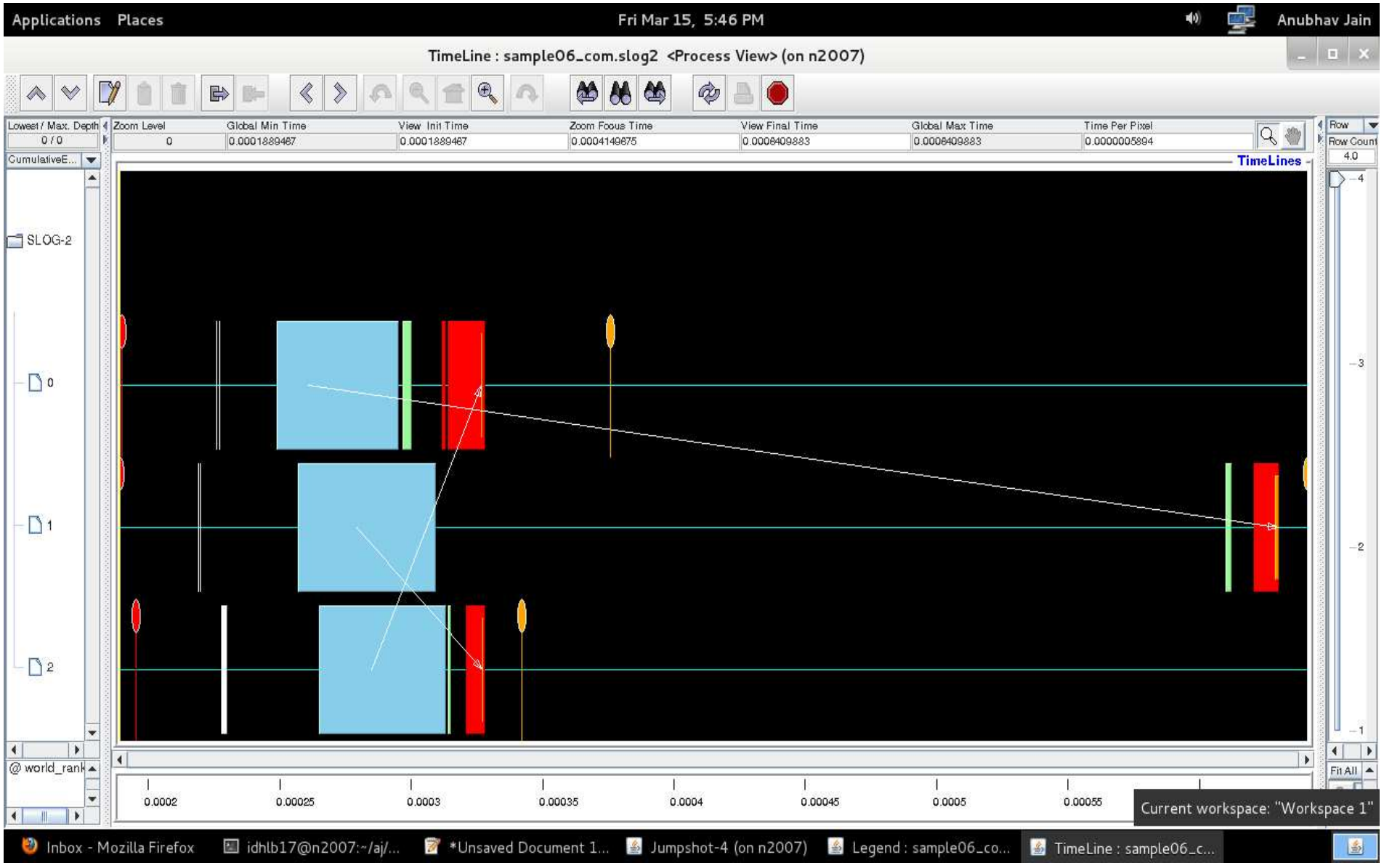
- MPI Send and Recv
- Time : 0.000409 seconds

- MPI Isend and Irecv
- Time : 0.000087 seconds

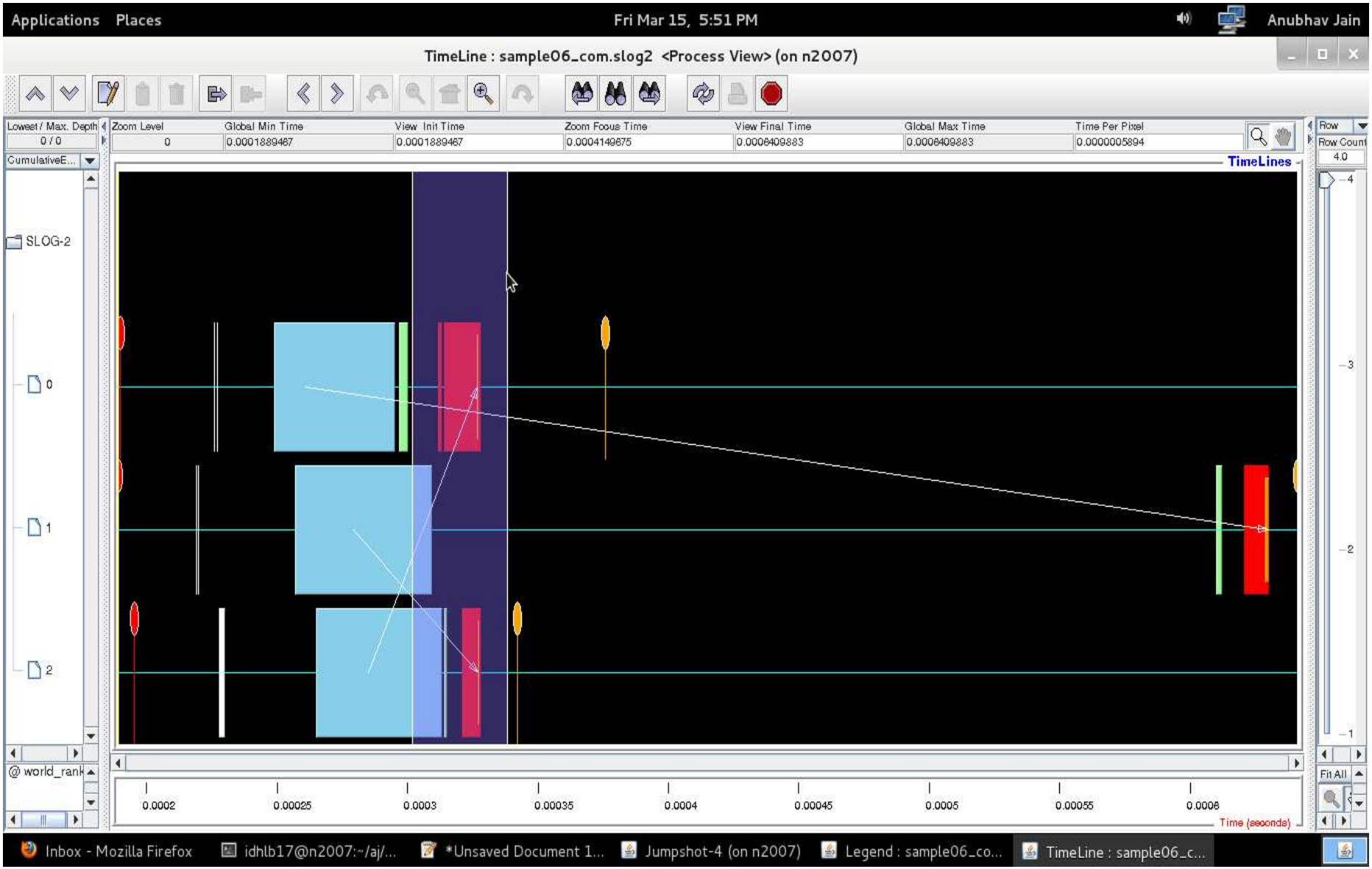
Blocking send recv with computation



Non blocking send recv with computation



Time window


Applications Places
Fri Mar 15, 5:51 PM
Anubhav Jain
TimeLine : sample06_com.slog2 <Process View> (on n2007)
Lowest / Max. Depth
0 / 0
Zoom Level
0
Global Min Time
0.0001889467
View Init Time
0.0001889467
Zoom Focus Time
0.0004149875
View Final Time
0.0006409883
Global Max Time
0.0006409883
Time Per Pixel
0.0000005894
TimeLines
SLOG-2
0
1
2
@ world_rank
Row
Row Count
4.0
Fit All
0.0002
0.00025
0.0003
0.00035
0.0004
0.00045
0.0005
0.00055
0.0006
Time (seconds)
Inbox - Mozilla Firefox
idhlb17@n2007:~/aj/...
*Unsaved Document 1...
Jumpshot-4 (on n2007)
Legend : sample06_co...
TimeLine : sample06_c...

Thank You !