9th Advanced School on PARALLEL COMPUTING

Hybrid MPI+OpenMP Profiling

Paride Dagna – p.dagna@cineca.it
SuperComputing Applications and Innovation Department

CINECA

February 11 - 15, 2013

Introduction

A serial or parallel program is normally composed by a large number of procedures.

To optimize and parallelize a complex code is fundamental to find out the parts where most of time is spent.

Moreover is very important to understand the graph of computation and the dependencies and correlations between the different sections of the code.

For a good scalability in **parallel programs**, it's necessary to have a good load and communication balancing between processes.

To **discover** the **hotspots** and the **bottlenecks** of a code and find out the **best optimization and parallelization strategy** the programmer can follow two common methods:

- Manual instrumentation inserting timing and collecting functions (difficult)
- Automatic profiling using **profilers** (easier and very powerful)

Measuring execution time

Both C/C++ and Fortran programmers are used to instrument the code with timing and printing functions to measure and collect or visualize the time spent in critical or computationally intensive code' sections.

- **Fortran77**
  - etime(),dtime()
- **Fortran90**
  - cputime(), system_clock(), date_and_time()
- **C/C++**
  - clock()

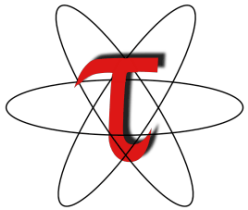
In this kind of operations it must be taken into account of:

- Intrusiveness
- Granularity
- Reliability
- Overhead

Very difficult task for third party complex codes

Profilers

There are many versions of commercial profilers, developed by manufacturers of compilers and specialized software house. In addition there are **free profilers**, as those resulting from the GNU, TAU or Scalasca project.



Tau Performance System
- University of Oregon

Intel® VTune™ Amplifier

Scalasca
-Research Centre Juelich

The Portland Group PGPROF

GNU gprof

OPT

PerfSuite
– National Center for Supercomputing Applications

Profilers

- Profilers allow the programmer to obtain very useful information on the various parts of a code with basically two levels of profiling:

- **Subroutine/Function level**
  - Timing at routine/function level, graph of computation flow
  - less intrusive
  - Near realistic execution time

- **Construct/instruction/statement level**
  - capability to profile each instrumented statement
  - more intrusive
  - very accurate timing information
  - longer profiling execution time

TAU Tuning and Analysis Utilities

- TAU Performance System® is a portable profiling and tracing toolkit for performance analysis of serial and parallel programs written in Fortran, C, C++, Java, and Python.

  www.cs.uoregon.edu/research/tau

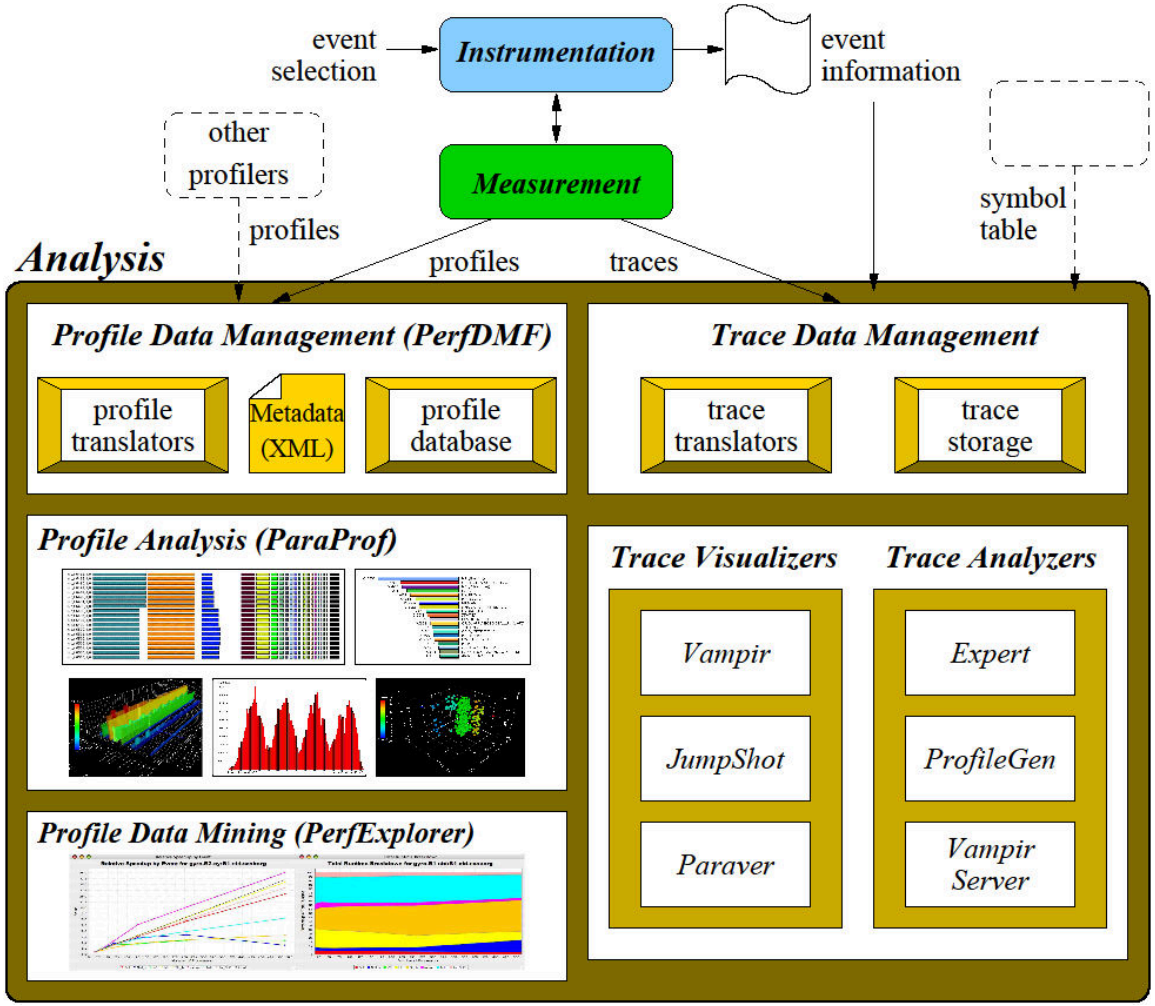
- 12+ years of project in which are currently involved:
  - University of Oregon Performance Research Lab
  - LANL Advanced Computing Laboratory
  - Research Centre Julich at ZAM, Germany

- TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks and statements of serial and shared or distributed memory parallel codes

- It's portable on all architectures

- Provides powerful and user friendly graphic tools for result analysis

TAU Architecture



TAU Installation and configuration

- During the installation phase TAU requires different configurations flags depending on the kind of code to be analyzed.

GNU/XL	Load compiler modules with “module load”
GNU/XL	Configuration Flags
Serial	`configure -prefix=/path_to_destination_folder -pdt=/path_to_pdt -c++=g++ -cc=gcc -fortran=gfortran`
MPI	`configure -prefix=/path_to_destination_folder -mpi -mpiinc=/path_to_mpi_include_dirs -mpilib=/path_to_mpi_lib -pdt=/path_to_pdt -c++=g++ -cc=gcc -fortran=gfortran`
OpenMP	`configure -prefix=/path_to_destination_folder -pdt=/path_to_pdt -openmp -opari -opari_region -opari_construct -c++=g++ -cc=gcc -fortran=gfortran`
MPI+OpenMP	`configure -prefix=/path_to_destination_folder -mpi -openmp -mpiinc=/path_to_mpi_include_dirs -mpilib=/path_to_mpi_lib -pdt=/path_to_pdt -opari_region -opari_construct -c++=g++ -cc=gcc -fortran=gfortran`

- After configuration TAU can be easily installed with:
  - `make install`

TAU - Introduction

- TAU provides three different methods to track the performance of your application.
- The simplest way is to use TAU with dynamic instrumentation based on pre-charged libraries

Dynamic instrumentation

- **Doesn't** requires to recompile the executable
- Instrumentation is achieved at **run-time** through library pre-loading
- Dynamic instrumentation include tracking MPI, io, memory, cuda, opencl library calls. MPI instrumentation is included by default, the others are enabled by command-line options to tau_exec.

  – Serial code

  ```
  %> tau_exec -io ./a.out
  ```

  – Parallel MPI code

  ```
  %> mpirun -np 4 tau_exec -io ./a.out
  ```

  – Parallel MPI + OpenMP code

  ```
  %> mpirun -x OMP_NUM_THREADS=2 -np 4 tau_exec -io ./a.out
  ```

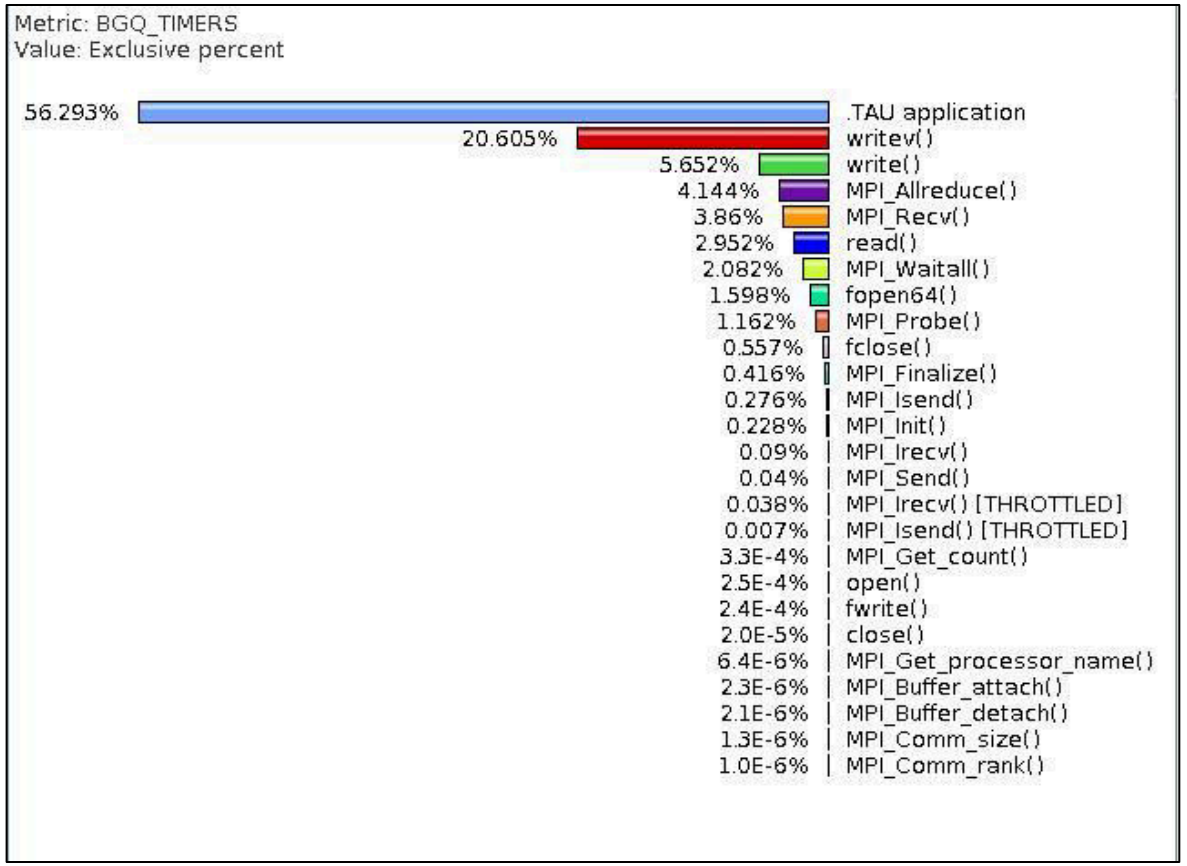
TAU – Dynamic instrumentation on BG/Q

MPI + I/O Dynamic Profiling

```
#!/bin/bash

# @ job_type = bluegene
# @ bg_connectivity = MESH
# @ wall_clock_limit = 24:00:00
# @ notification = never
# @ bg_size = 256
# @ bg_rotate = FALSE
# @ job_name =job_name
# @ initialdir = .
# @ account_no = you_account_number
# @ error = $(job_name)_$(jobid).err
# @ output = $(job_name)_$(jobid).out
# @ queue

export TOTAL_MPI_PROCESSES=1024
export TASK_PER_NODE=16

export LD_LIBRARY_PATH=/path_to_tau_dynamic_library:$LD_LIBRARY_PATH
LD_AUDIT=/path_to_TAU_auditor_library/libTAU-dl-auditor.so
export LD_BIND_NOW=1
EXEC=/path_to_executable/executable

runjob  –envs "LD_PRELOAD=$LD_PRELOAD:/path_to_io_wrap_library/libTAUiowrap.so
:/path_to_tau_dynamic_library/libTAU.so:/path_to_TAU_preloading_library/libTAU-preload.so" --np
$TOTAL_MPI_PROCESSES --ranks-per-node $TASK_PER_NODE --env-all : $EXEC
```

TAU – Dynamic instrumentation on BG/Q

MPI + I/O Dynamic Profiling



TAU - Compiler based instrumentation

- For more detailed profiles, TAU provides two means to compile your application with TAU: through your compiler or through source transformation using PDT.
- **It's necessary** to recompile the application, **static instrumentation** at compile time

- TAU provides these scripts to instrument and compile Fortran, C,and C++ programs respectively:
  - `tau_f90.sh`
  - `tau_cc.sh`
  - `tau_cxx.sh`
- Compiler based instrumentation needs the following steps:
  - Environment configuration
  - Code recompiling
  - Execution
  - Result analysis

TAU Compiler based instrumentation

1. Environment configuration:

```
%>export TAU_MAKEFILE=[path to tau]/[arch]/lib/[makefile]
%>export TAU_OPTIONS='-optCompInst -optRevert'
```

   - Optional:
   - `%>export PROFILEDIR = [path to directory with result]`

2. Code recompiling:

```
%>tau_cc.sh source_code.c
```

3. Execution:
   - **To enable callpath creation:**
   - `%>export TAU_CALLPATH=1`
   - `%>export TAU_CALLPATH_DEPTH=30`
   - **To enable MPI message statistics**
   - `%>export TAU_TRACK_MESSAGE=1`

TAU Compiler based instrumentation - Execution on BG/Q

1. Environment configuration:

```
%> module load bgq-xl
%> module load tau/2.21.4
%>export TAU_MAKEFILE=[path to
   tau]/[arch]/lib/[makefile]
```

2. Code recompiling:

```
%>tau_cc.sh source_code.c
```

3. Execution:
   - `%>llsubmit launch_script.ll`

```
#!/bin/bash

# @ job_type = bluegene
# @ bg_connectivity = MESH
# @ wall_clock_limit = 24:00:00
# @ notification = never
# @ bg_size = 64
   ...   ... ...   ...

export TOTAL_MPI_PROCESSES=512
export TASK_PER_NODE=16

export TAU_CALLPATH=1
export TAU_CALLPATH_DEPTH=30
export TAU_COMM_MATRIX=1
export TAU_TRACK_MESSAGE=1

EXEC=/path_to_executable/executable
runjob --np $TOTAL_MPI_PROCESSES
      --ranks-per-node $TASK_PER_NODE --env-all : $EXEC
```

TAU - environment variables

Environment Variable	Default	Description
TAU_PROFILE	0	Set to 1 to have TAU profile your code
TAU_CALLPATH	0	When set to 1 TAU will generate call-path data. Use with TAU_CALLPATH_DEPTH.
TAU_TRACK_MEMORY_LEAKS	0	Set to 1 for tracking of memory leaks (to be used with tau_exec –memory)
TAU_TRACK_HEAP or TAU_TRACK_HEADROOM	0	Setting to 1 turns on tracking heap memory/headroom at routine entry & exit using context events (e.g., Heap at Entry: main=>foo=>bar)
TAU_CALLPATH_DEPTH	2	Callapath depth. 0 No callapath. 1 flat profile
TAU_SYNCHRONIZE_CLOCKS	1	When set TAU will correct for any time discrepancies between nodes because of their CPU clock lag.
TAU_COMM_MATRIX	0	If set to 1 generate MPI communication matrix data.
TAU_THROTTLE	1	If set to 1 enables the runtime throttling of events that are lightweight
TAU_THROTTLE_NUMCALLS	100000	Set the maximum number of calls that will be profiled for any function when TAU_THROTTLE is enabled
TAU_THROTTLE_PERCALL	10	Set the minimum inclusive time a function has to have to be instrumented when TAU_THROTTLE is enabled.

TAU_OPTIONS

- Optional parameters for TAU_OPTIONS: [tau_compiler.sh -help]
  - -optVerbose — Vebose debugging
  - -optCompInst — Compiler based instrumentation
  - -optNoCompInst — No Compiler based instrumentation
  - -optDetectMemoryLeaks — Debug memory allocations/de-allocations
  - -optPreProcess — Fortran preprocessing before code instrumentation
  - -optTauSelectFile="" — Selective file for the tau_instrumentor

Result analysis

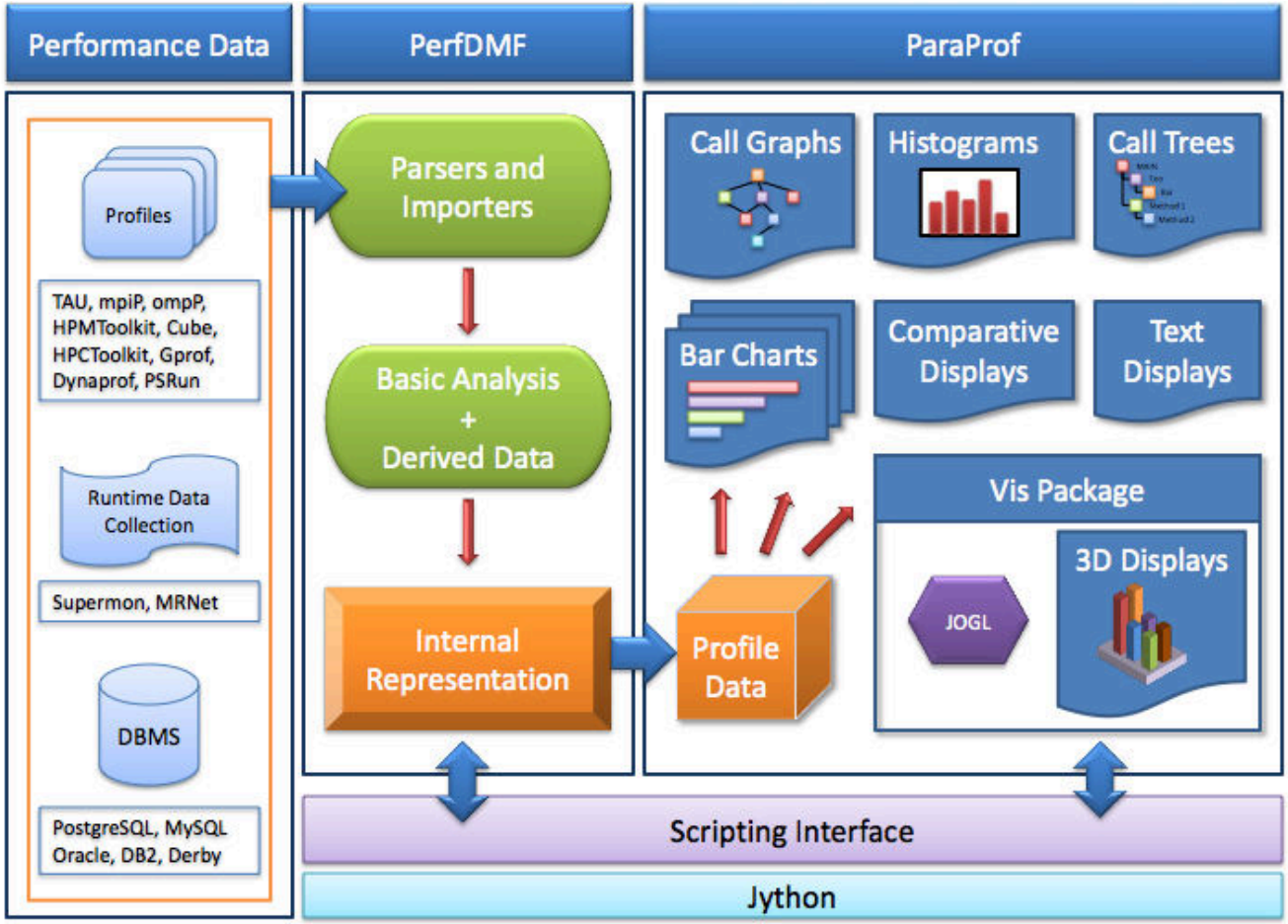
- At the end of a run, a code instrumented with TAU produces a series of files "`profile.x.x.x`" containing the profiling information.

- TAU provides two tools for profiling analysis :
  - `pprof` command line, useful for a quick view summary of TAU performance
  - `Paraprof` with a sophisticated GUI allows very detailed and powerful analysis

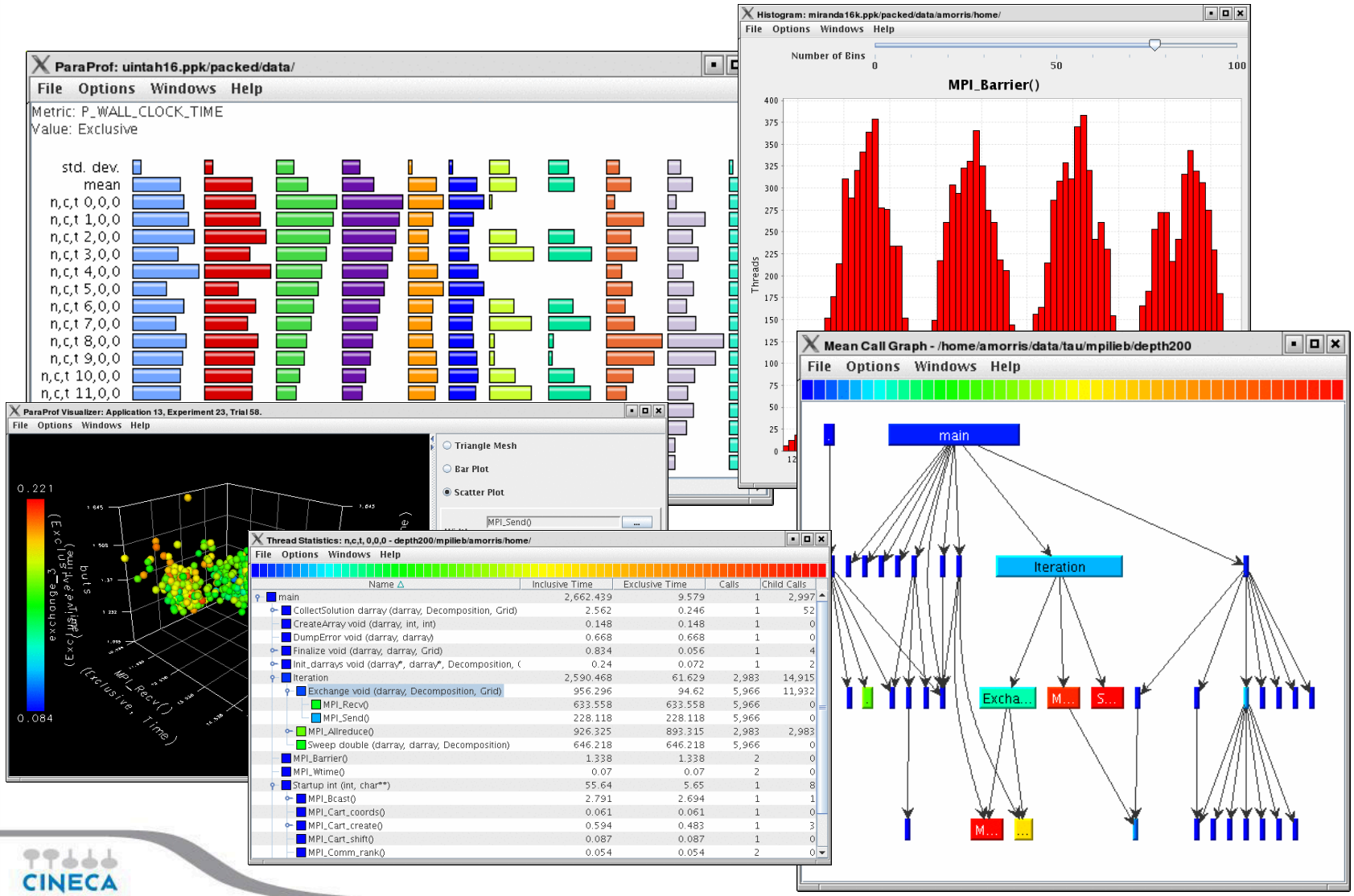
- **Usage:** `pprof [-c|-b|-m|-t|-e|-i|-v] [-r] [-s] [-n num] [-f filename] [-p] [-l] [-d] [node numbers]`

```
-a : Show all location information available
-c : Sort according to number of Calls
-b : Sort according to number of suBroutines called by a function
-m : Sort according to Milliseconds (exclusive time total)
-t : Sort according to Total milliseconds (inclusive time total)  (default)
-e : Sort according to Exclusive time per call (msec/call)
-i : Sort according to Inclusive time per call (total msec/call)
-v : Sort according to Standard Deviation (excl usec)
-r : Reverse sorting order
-s : print only Summary profile information
-n <num> : print only first <num> number of functions
-f filename : specify full path and Filename without node ids
-p : suPpress conversion to hh:mm:ss:mmm format
-l : List all functions and exit
```

Result analysis: paraprof



Paraprof


ParaProf: uintah16.ppk/packed/data/
File Options Windows Help
Metric: P_WALL_CLOCK_TIME
Value: Exclusive
std. dev.
mean
n,c,t 0,0,0
n,c,t 1,0,0
n,c,t 2,0,0
n,c,t 3,0,0
n,c,t 4,0,0
n,c,t 5,0,0
n,c,t 6,0,0
n,c,t 7,0,0
n,c,t 8,0,0
n,c,t 9,0,0
n,c,t 10,0,0
n,c,t 11,0,0
Histogram: miranda16k.ppk/packed/data/amorris/home/
File Options Windows Help
Number of Bins
0
50
100
MPI_Barrier()
Threads
ParaProf Visualizer: Application 13, Experiment 23, Trial 58.
File Options Windows Help
Triangle Mesh
Bar Plot
Scatter Plot
MPI_Send()
0.221
0.084
Thread Statistics: n,c,t, 0,0,0 - depth200/mpilieb/amorris/home/
File Options Windows Help
Name
Inclusive Time
Exclusive Time
Calls
Child Calls
main 2,662.439 9.579 1 2,997
CollectSolution darray (darray, Decomposition, Grid) 2.562 0.246 1 52
CreateArray void (darray, int, int) 0.148 0.148 1 0
DumpError void (darray, darray) 0.668 0.668 1 0
Finalize void (darray, darray, Grid) 0.834 0.056 1 4
Init_darrays void (darray*, darray*, Decomposition, 0.24 0.072 1 2
Iteration 2,590.468 61.629 2,983 14,915
Exchange void (darray, Decomposition, Grid) 956.296 94.62 5,966 11,932
MPI_Recv() 633.558 633.558 5,966 0
MPI_Send() 228.118 228.118 5,966 0
MPI_Allreduce() 926.325 893.315 2,983 2,983
Sweep double (darray, darray, Decomposition) 646.218 646.218 5,966 0
MPI_Barrier() 1.338 1.338 2 0
MPI_Wtime() 0.07 0.07 2 0
Startup int (int, char**) 55.64 5.65 1 8
MPI_Bcast() 2.791 2.694 1 1
MPI_Cart_coords() 0.061 0.061 1 0
MPI_Cart_create() 0.594 0.483 1 3
MPI_Cart_shift() 0.087 0.087 1 0
MPI_Comm_rank() 0.054 0.054 2 0
Mean Call Graph - /home/amorris/data/tau/mpilieb/depth200
File Options Windows Help
main
Iteration
Excha...
M...
S...
M...

Example

```
#include<stdio.h>

double add3(double x){
        return x+3;}

double mysum(double *a, int n){
double sum=0.0;
for(int i=0;i<n;i++)
        sum+=a[i]+add3(a[i]);
return sum;
}

double init(double *a,int n){
double res;
for (int i=0;i<n;i++) a[i]=double(i);
res=mysum(a,n);
return res;
}

int main(){
double res,mysum;
int n=30000;
double a[n];

for (int i=0;i<n;i++){
        res=init(a,n);
}
printf("Result %f\n",res);
return 0;}
```

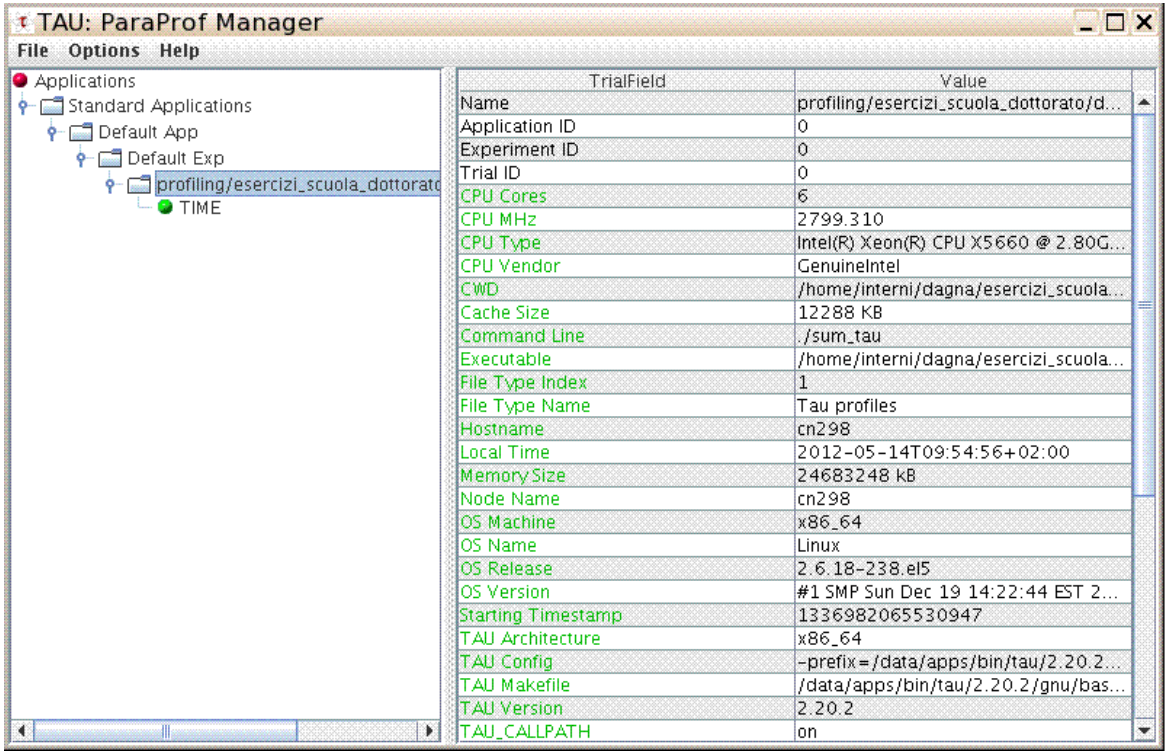
Pprof

`pprof` output:

```
%> pprof

Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           3     3:20.342           1           1  200342511 .TAU
   application
100.0           4     3:20.338           1       30000  200338851 main
100.0       2,344     3:20.334       30000       30000       6678 init
 98.8    1:40.824     3:17.989       30000       9E+08       6600 mysum
 48.5    1:37.164     1:37.164       9E+08           0          0 add3
```

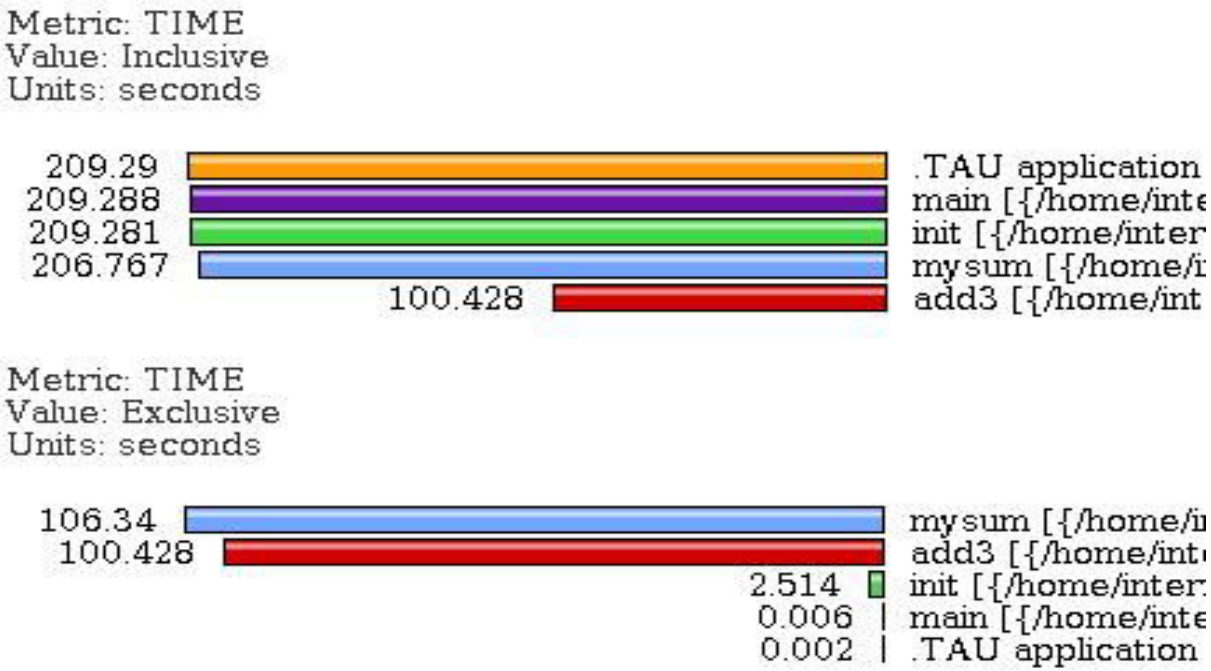
Paraprof Manager Window

`paraprof` output:



This window is used to manage profile data. The user can upload/download profile data, edit meta-data,launch visual displays, export data, derive new metrics, etc.

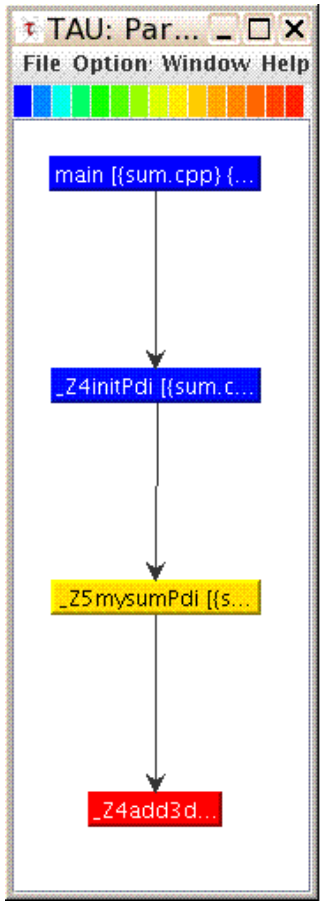
Thread bar chart



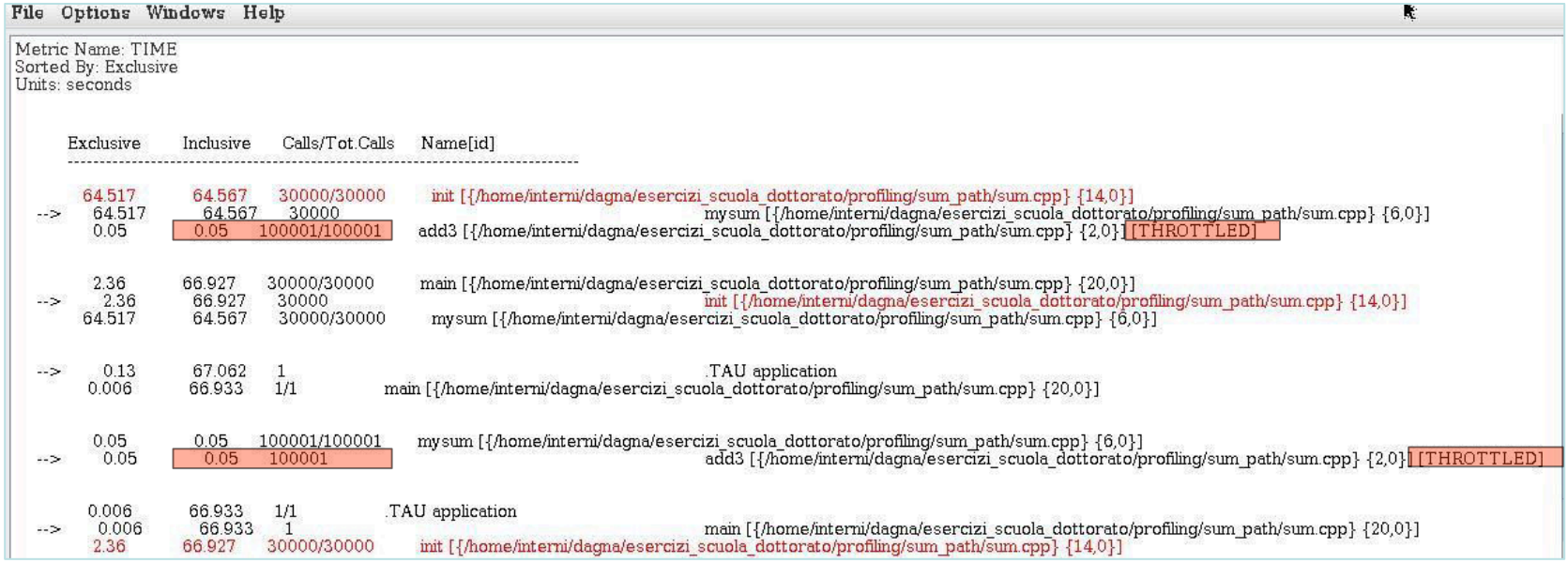
This display graphs each function on a particular thread for comparison. The metric, units, and sort order can be changed from the **Options** menu.

Call Graph

- This display shows callpath data in a graph using two metrics, one determines the width, the other the color.
- The full name of the function as well as the two values (color and width) are displayed in a tooltip when hovering over a box.
- By clicking on a box, the actual ancestors and descendants for that function and their paths (arrows) will be highlighted with blue.
- This allows you to see which functions are called by which other functions since the interplay of multiple paths may obscure it.



Thread Call Path Relations Window



For example “mysum” is called from “init” 30000 times for a total of 64.5 seconds and calls “add3” function 30000 times.

TAU automatically throttles short running functions in an effort to reduce the amount of overhead associated with profiles of such functions, default throttle limit is:

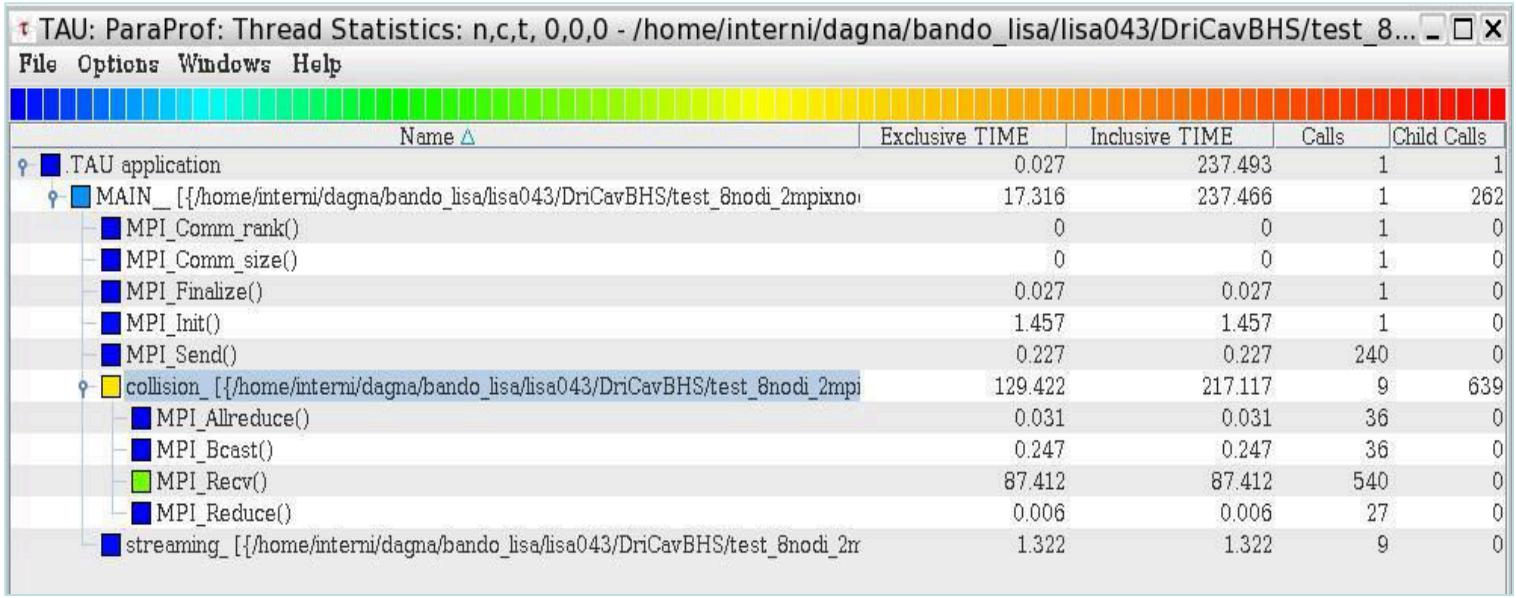
- numcalls> 100000 && usecs/call < 10

To change default settings TAU gives the following environment variables:

- TAU_THROTTLE_NUMCALLS, TAU_THROTTLE_PERCALL

To disable TAU throttle : export TAU_THROTTLE=0

Thread Statistics Table


TAU: ParaProf: Thread Statistics: n,c,t, 0,0,0 - /home/interni/dagna/bando_lisa/lisa043/DriCavBHS/test_8...

File Options Windows Help

Name △	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	0.027	237.493	1	1
MAIN__ [{/home/interni/dagna/bando_lisa/lisa043/DriCavBHS/test_8nodi_2mpixno	17.316	237.466	1	262
MPI_Comm_rank()	0	0	1	0
MPI_Comm_size()	0	0	1	0
MPI_Finalize()	0.027	0.027	1	0
MPI_Init()	1.457	1.457	1	0
MPI_Send()	0.227	0.227	240	0
collision_ [{/home/interni/dagna/bando_lisa/lisa043/DriCavBHS/test_8nodi_2mpi	129.422	217.117	9	639
MPI_Allreduce()	0.031	0.031	36	0
MPI_Bcast()	0.247	0.247	36	0
MPI_Recv()	87.412	87.412	540	0
MPI_Reduce()	0.006	0.006	27	0
streaming_ [{/home/interni/dagna/bando_lisa/lisa043/DriCavBHS/test_8nodi_2r	1.322	1.322	9	0

This display shows the callpath data in a table. Each callpath can be traced from root to leaf by opening each node in the tree view.
A colorscale immediately draws attention to "hot spots" areas that contain highest values.

Tau profiler: parallel codes

TAU provides a lot of tools to analyze OpenMP, MPI or OpenMP + MPI parallel codes.

Profiling the application the user can obtain a lot of useful information which can help to identify the causes of an unexpected low parallel efficiency.

Principal factors which can affect parallel efficiency are:

- load balancing
- communication overhead
- process synchronization
- Latency and bandwidth

Tau profiler: parallel codes

- Configure:

```
%> module load bgq-xl
%> module load tau/2.21.4
%>export TAU_MAKEFILE=[path to tau]/[arch]/lib/[makefile]
%>export TAU_OPTIONS=-optCompInst
```

- Compile:

```
Tau_cc.sh -o executable source.c      (C)
Tau_cxx.sh -o executable source.cpp   (C++)
Tau_f90.sh -o executable source.f90   (Fortran)
```

- Run the application:

```
llsubmit launch_script.ll
```

At the end of simulation, in the working directory or in the path specified with the `PROFILEDIR` variable, the data for the profiler will be saved in files `profile.x.x.x`

Unbalanced load

```
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <cmath>
using namespace std;

# include "mpi.h"
void compute(float * data, int start, int stop){

        for (int i=0;i<1000000;i++){
                for(int j=start;j<stop;j++){
                        data[j]=pow((double)j/(j+4),3.5);}}
}
int main ( int argc, char *argv[] )
 {
  int count;
  float data[24000];
  int dest,i,num_procs,rank,tag;
  MPI::Status status;
  float value[12000];
    MPI::Init ( argc, argv );
    rank = MPI::COMM_WORLD.Get_rank ( );
    if ( rank == 0 )
  {
     num_procs = MPI::COMM_WORLD.Get_size ( );

     cout << "  The number of processes available is " << num_procs << "\n";
  }
```

Unbalanced load

```
if ( rank == 0 )
  {
    tag = 55;
    MPI::COMM_WORLD.Recv ( value,12000, MPI::FLOAT, MPI::ANY_SOURCE, tag,
      status );

    cout << "P:" << rank << " Got data from process " <<
      status.Get_source() << "\n";

    count = status.Get_count ( MPI::FLOAT );

    cout << "P:" << rank << " Got " << count << " elements.\n";

    compute(value,0,12000);
  }
else if ( rank == 1 )
  {
    cout << "\n";
    cout << "P:" << rank << " - setting up data to send to process 0.\n";

    for ( i = 0; i <24000; i++ )
    {
      data[i] = i;
    }
    dest = 0;
    tag = 55;
    MPI::COMM_WORLD.Send ( data, 12000, MPI::FLOAT, dest, tag );
    compute(data,12000,24000);
  }
```

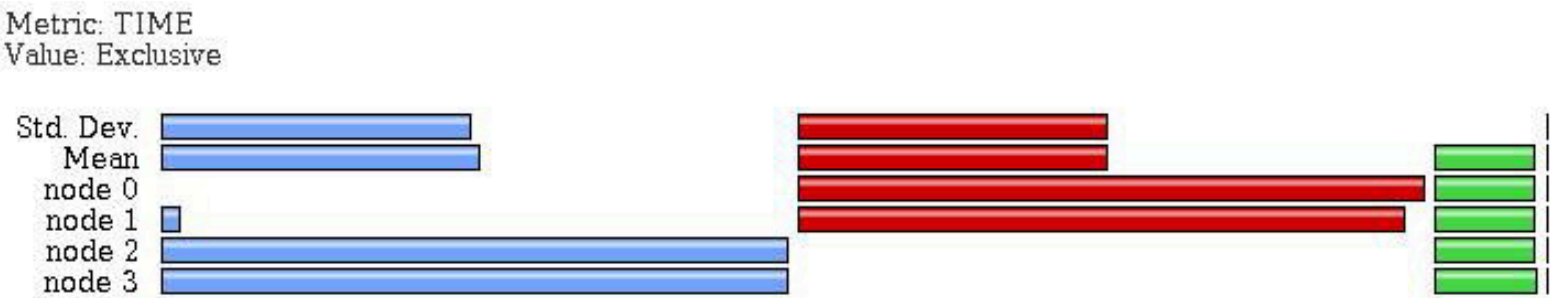
Unbalanced load

```
else
  {
    cout << "\n";
    cout << "P:" << rank << " - MPI has no work for me!\n";
  }
  MPI::Finalize ( );
  if ( rank == 0 )
  {
    cout << "  Normal end of execution.\n";
  }
  return 0;
}
```

Output:

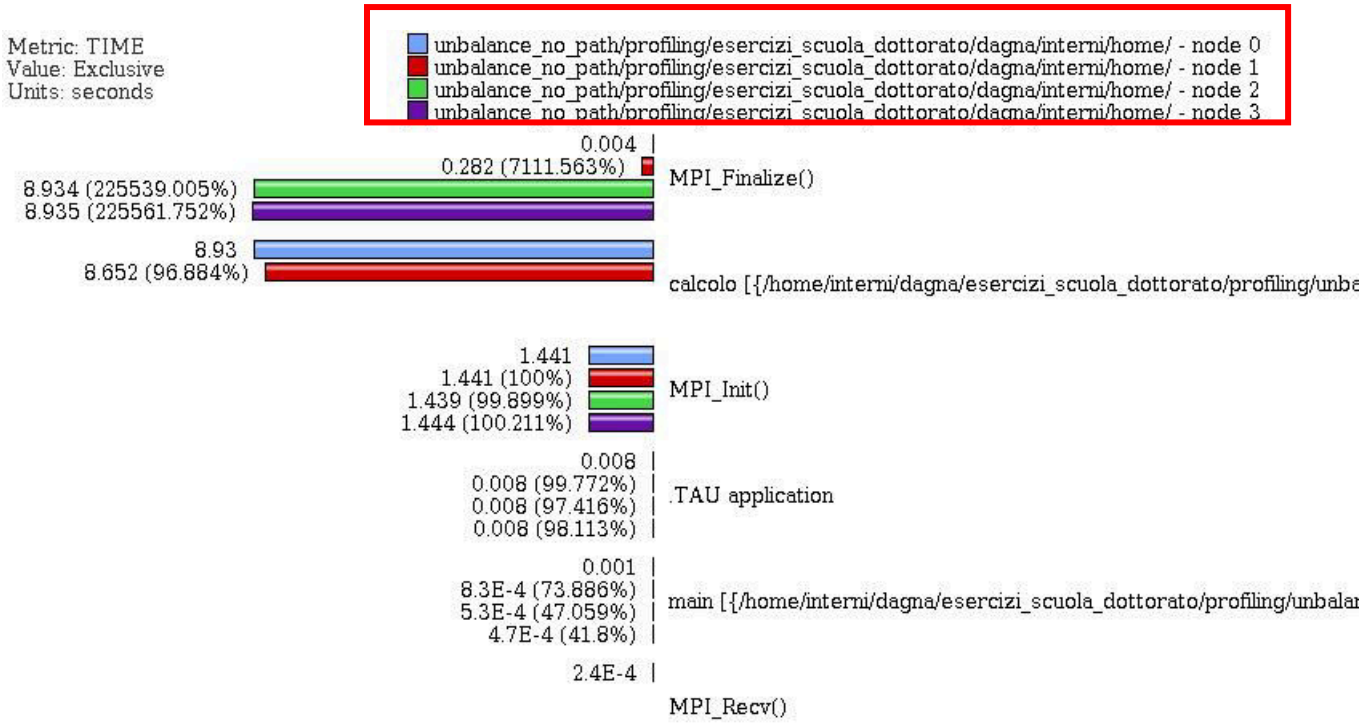
```
The number of processes available is 4
P:0 Got data from process 1
P:0 Got 12000 elements.

P:1 - setting up data to send to process 0.

P:3 - MPI has no work for me!

P:2 - MPI has no work for me!
Normal end of execution.
```

Unstacked bars



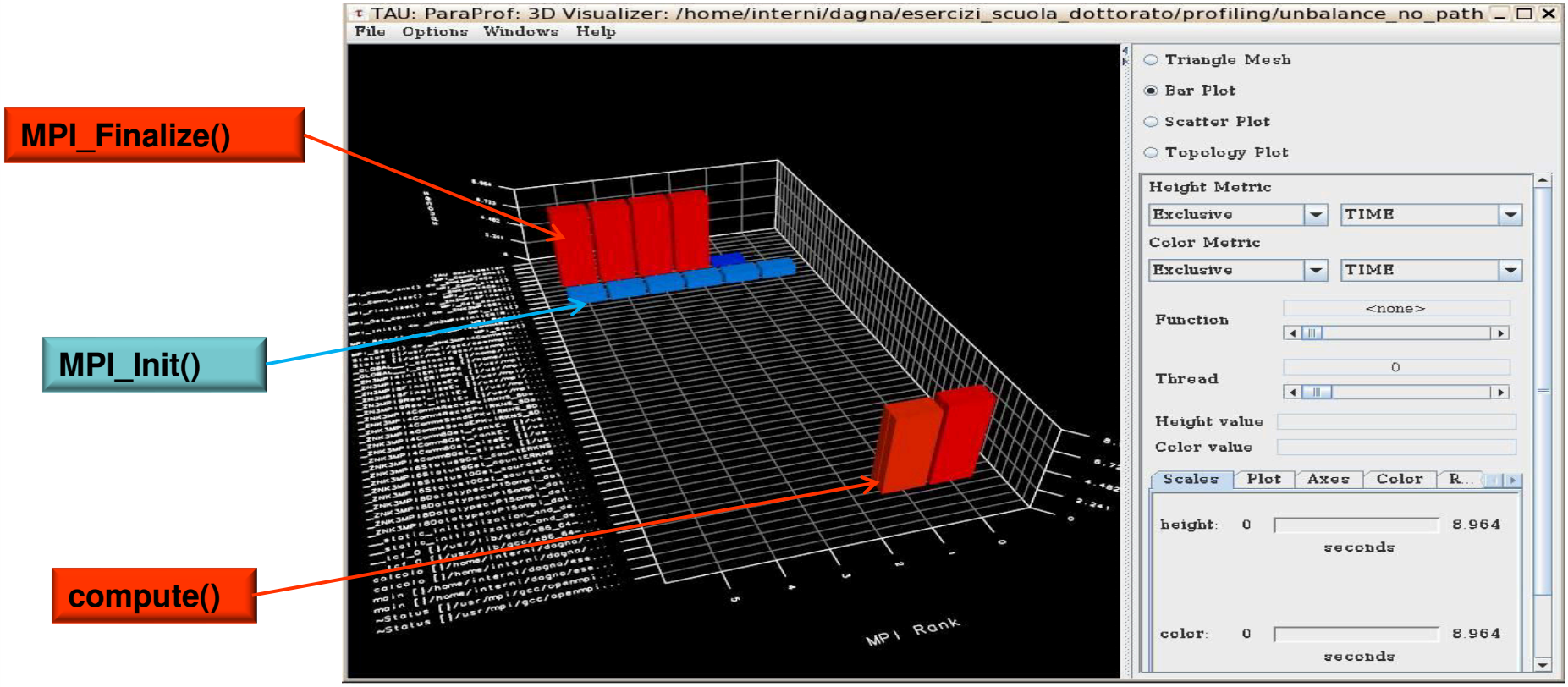
- Very useful to compare individual functions across threads in a global display

Comparison window



- Very useful to compare the behavior of process and threads in all the functions or regions of the code to find load unbalances.

3D Visualizer



- This visualization method shows two metrics for all functions, all threads. The height represents one chosen metric, and the color, another. These are selected from the drop-down boxes on the right.
- To pinpoint a specific value in the plot, move the *Function* and *Thread* sliders to cycle through the available functions/threads.

Balanced load

```
int main ( int argc, char *argv[] )
{
MPI::Init ( argc, argv );
rank = MPI::COMM_WORLD.Get_rank ( );
float data[24000];

  if ( rank == 0 )
  {
    num_procs = MPI::COMM_WORLD.Get_size ( );

    cout << "  The number of processes available is " << num_procs << "\n";
  }
  int subd = 24000/num_procs
  if ( rank!= 0)
  {
    tag = 55;
    MPI::COMM_WORLD.Recv ( data,subd, MPI::FLOAT, MPI::ANY_SOURCE, tag, status );

    cout << "P:" << rank << " Got data from process " <<
      status.Get_source() << "\n";
    count = status.Get_count ( MPI::FLOAT );

    cout << "P:" << rank << " Got " << count << " elements.\n";
    compute(data,rank*subd,rank*subd+subd);
    printf("Done\n");
  }
```

Balanced load

```
else if ( rank == 0 )
  {
    cout << "\n";
    cout << "P:" << rank << " - setting up data to send to processes.\n";

    for ( i = 0; i <24000; i++ )
    {
      data[i] = i;
    }
    tag = 55;
    printf("Done\n");
    for(int el=1;el<num_procs;el++){

        MPI::COMM_WORLD.Send ( &data[subd*el], subd, MPI::FLOAT, el, tag );
        }
    compute(data,0,subd);
  }

  MPI::Finalize ( );

  if ( rank == 0 )
  {
    cout << "  Normal end of execution.\n";
  }

  return 0;
}
```

Balanced load

- Output:

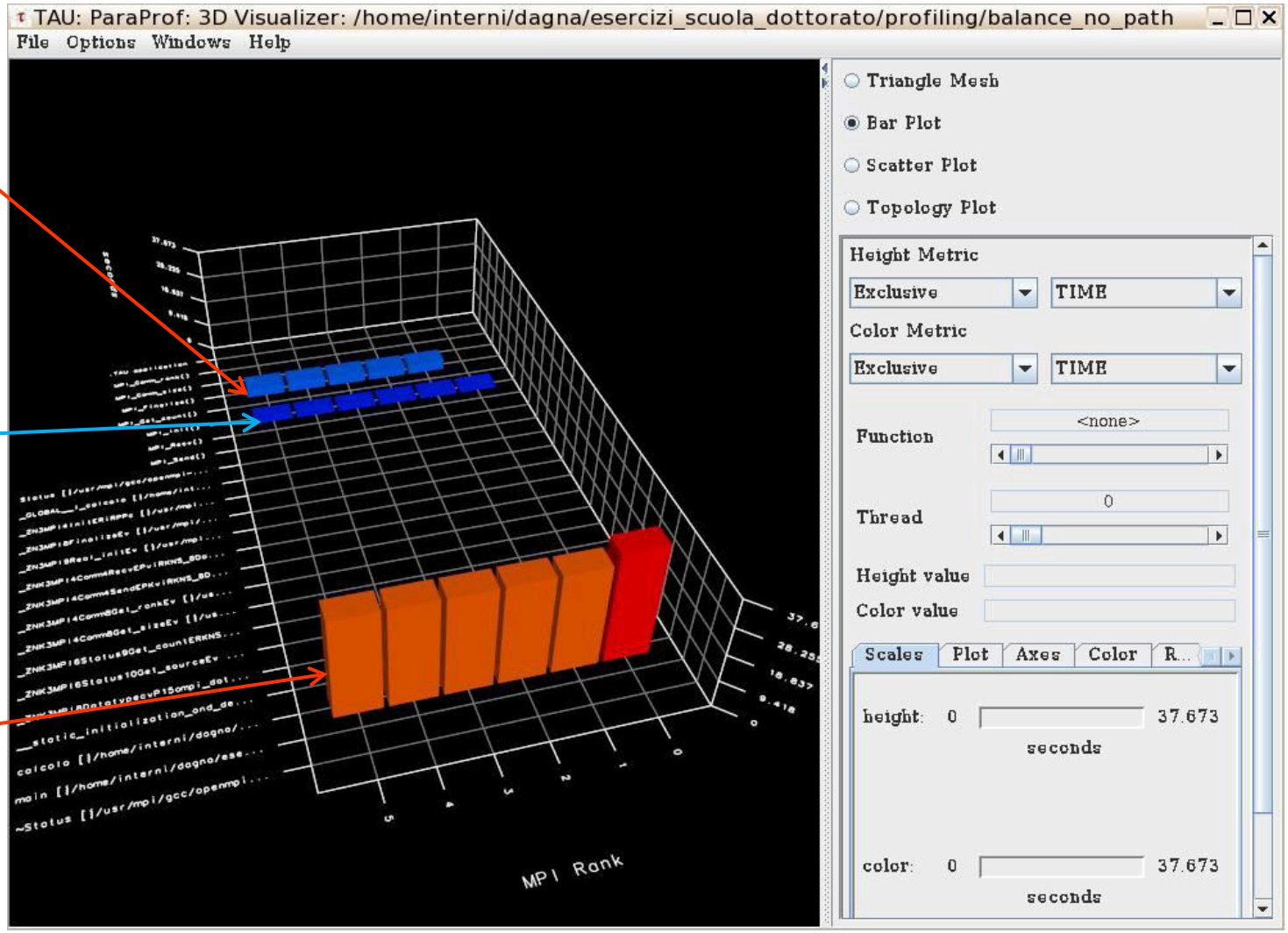
```
 The number of processes available is 6

P:0 - setting up data to send to processes.
Done
P:5 Got data from process 0
P:5 Got 4000 elements.
P:1 Got data from process 0
P:1 Got 4000 elements.
P:2 Got data from process 0
P:2 Got 4000 elements.
P:3 Got data from process 0
P:3 Got 4000 elements.
P:4 Got data from process 0
P:4 Got 4000 elements.
Done
Done
Done
Done
Done
  Normal end of execution.
```

Balanced load

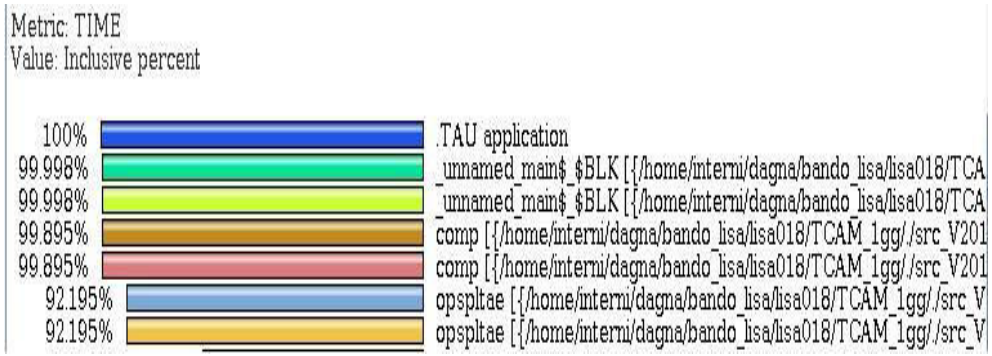
MPI_Finalize()

MPI_Init()

compute()

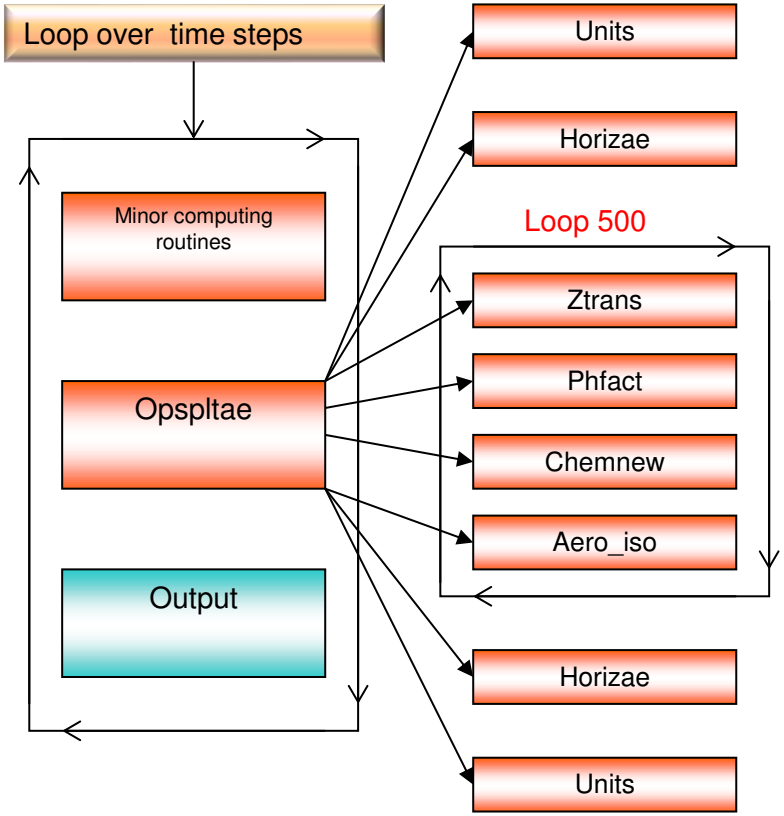


Real Case Air Pollution Model



Metric: TIME
Sorted By: Exclusive
Units: seconds

	Exclusive	Inclusive	Calls/Tot.Calls	
	71.785	3829.47	72/72	comp [{/home/interni/dagna/bando_lisa/lisa018/T
-->	71.785	3829.47	72	opspltae
	0.248	0.248	100001/100001	phfact [{/home/interni/dagna/bando_lisa/lisa0
	2.4E-4	2.4E-4	72/72	newphknew [{/home/interni/dagna/bando_lisa/lisa
	6.123	6.123	288/478	units [{/home/interni/dagna/bando_lisa/lisa018/T
	6.48	2746.714	4419360/4419360	chemnew [{/home/interni/dagna/bando_lis
	7.8E-4	7.8E-4	72/74	datetm [{/home/interni/dagna/bando_lisa/lisa018/
	80.281	452.527	144/144	horizae [{/home/interni/dagna/bando_lisa/lisa01
	33.933	362.447	4419360/4419360	aero_iso [{/home/interni/dagna/bando_lis
	0.021	0.021	35211/100001	relhum [{/home/interni/dagna/bando_lisa/lisa01
	189.604	189.604	1607040/1607040	ztrans [{/home/interni/dagna/bando_lisa/l
	7.8E-4	7.8E-4	864/938	iaddrs [{/home/interni/dagna/bando_lisa/lisa018
	4.2E-5	4.2E-5	72/72	savphknew [{/home/interni/dagna/bando_lisa/lisaC



Real Case Air Pollution Model

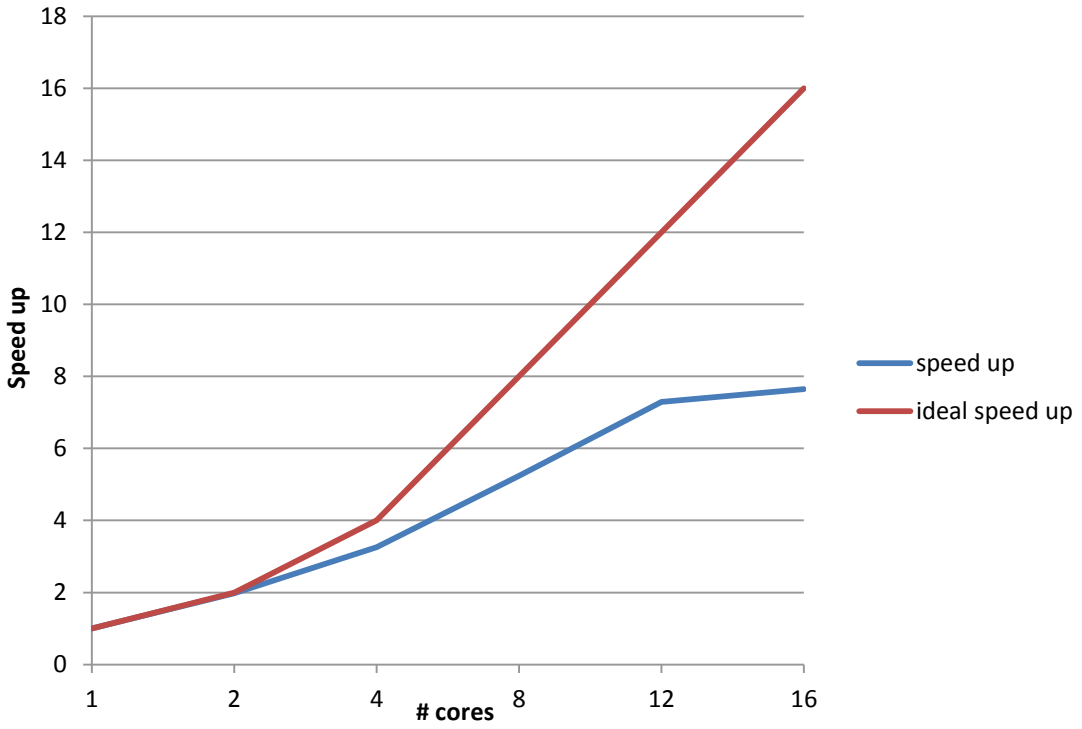
Amdahl law

Theoretical speedup

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

P=0.93 ➔ S(N)=14

Real speedup = 7.6 ☹



Let's check communication and load balncing !!

Real Case Air Pollution Model

Master process

opspltae [{/home/interni/	3.855	451.743	72	700,772
MPI_Bcast()	6.751	6.751	648	0
MPI_Comm_rank()	0	0	72	0
MPI_Comm_size()	0	0	72	0
MPI_Recv()	142.179	142.179	792	0
aero_iso [{/home/inter	2.079	32.924	237,600	475,200
calcola_elementi [{/hor	0	0	72	0
chemnew [{/home/inte	0.375	160.998	237,600	237,600
copia_vettori_in [{/hon	3.888	3.888	792	0
datetm [{/home/intern	0.001	0.001	72	0
horizae [{/home/intern	7.755	82.626	144	73,584
MPI_Bcast()	16.155	16.155	432	0
MPI_Comm_rank()	0	0	144	0
MPI_Comm_size()	0	0	144	0
MPI_Recv()	15.138	15.138	4,752	0
blcuvs [{/home/inte	6.884	6.884	15,840	0
blcuvsae [{/home/i	21.517	21.517	15,840	0
copiax_caein [{/hor	9.146	9.146	792	0
copiax_cin [{/home,	2.754	2.754	792	0
copiay_caein [{/hor	2.422	2.422	1,584	0
copiay_cin [{/home,	0.758	0.758	1,584	0
diffvs [{/home/inter	0.099	0.099	31,680	0
iaddrs [{/home/interni	0.001	0.001	864	0

Slave processes

opspltae [{/home/interni/	5.961	460.322	72	1,036,220
MPI_Bcast()	21.115	21.115	648	0
MPI_Comm_rank()	0	0	72	0
MPI_Comm_size()	0	0	72	0
MPI_Send()	0.191	0.191	72	0
aero_iso [{/home/inte	3.243	41.528	380,160	760,320
chemnew [{/home/inte	0.606	268.726	380,160	380,160
copia_vettori_out [{/h	0.465	0.465	72	0
datetm [{/home/interr	0.001	0.001	72	0
horizae [{/home/inter	10.553	95.75	144	83,952
MPI_Bcast()	30.98	30.98	432	0
MPI_Comm_rank()	0	0	144	0
MPI_Comm_size()	0	0	144	0
MPI_Send()	21.505	21.505	432	0
blcuvs [{/home/inte	7.722	7.722	20,592	0
blcuvsae [{/home/i	23.975	23.975	20,592	0
copiax_caeout [{/h	0.402	0.402	72	0
copiax_cout [{/hom	0.12	0.12	72	0
copiay_caeout [{/h	0.288	0.288	144	0
copiay_cout [{/hom	0.064	0.064	144	0
diffvs [{/home/inte	0.14	0.14	41,184	0
iaddrs [{/home/interni	0.001	0.001	864	0

Communication issues

Load balancing issues

The imbalance of computational load causes an overhead in the MPI directives due to long synchronization times dramatically reducing the scalability

TAU - Hybrid MPI + OpenMP

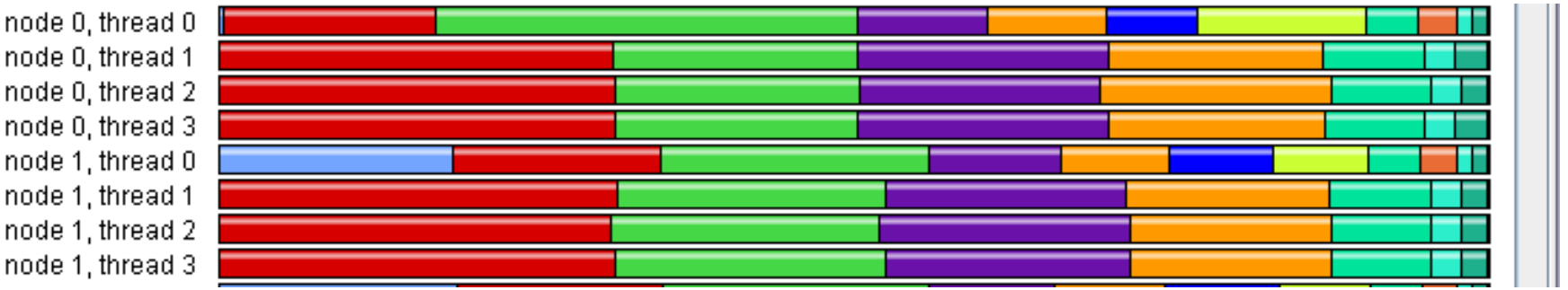
- Profiling of hybrid MPI + OpenMP applications with TAU is as easy as profiling of serial or parallel MPI codes.
- Standard procedure
  - Configure (select correct tau makefile)
    - `Makefile.tau-mpi-pdt-openmp`
  - Compile
  - Run

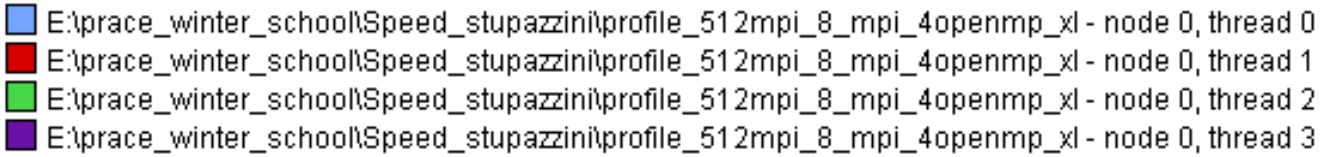
- At the end of simulation files profile.x.x.x will be produced, one for each MPI process and OpenMP thread.

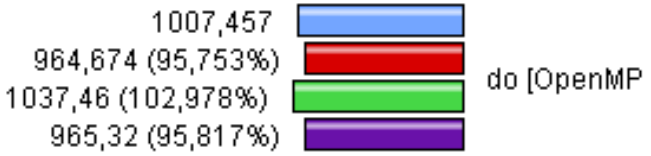
```
profile.0.0.0
profile.0.0.1
profile.0.0.2
profile.0.0.3
profile.1.0.0
profile.1.0.1
profile.1.0.2
profile.1.0.3
profile.2.0.0
profile.2.0.1
profile.2.0.2
profile.2.0.3
profile.3.0.0
profile.3.0.1
profile.3.0.2
profile.3.0.3
```

TAU - Hybrid MPI + OpenMP

- All the TAU analysis tools provided for serial and pure MPI codes are already available for MPI processes and OpenMP threads.

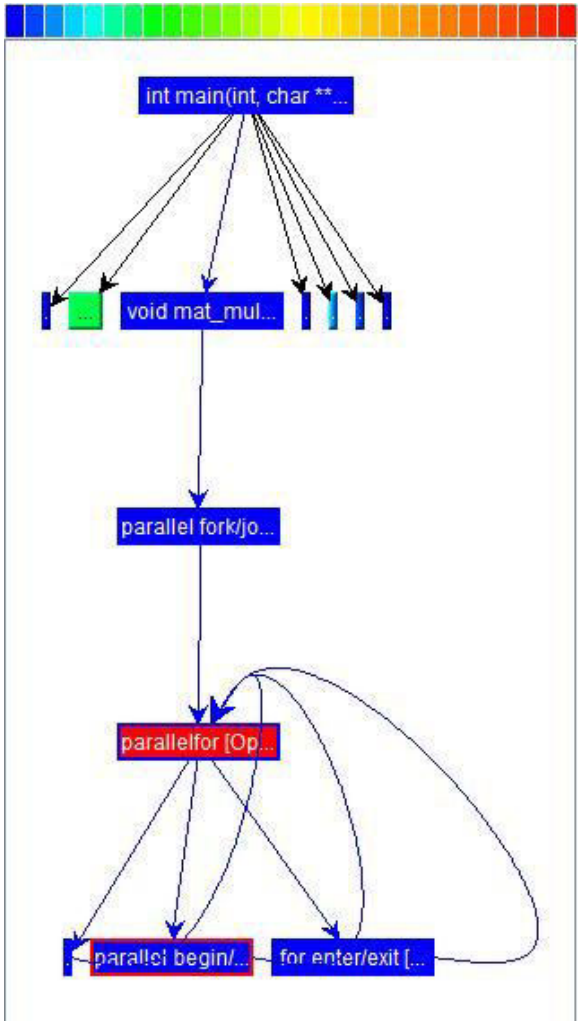






TAU - Hybrid MPI + OpenMP

- It's possible to highlight the callpath and the callpath relations tabel for each specific thread.



Metric Name: BGQ_TIMERS
Sorted By: Exclusive
Units: seconds

	Exclusive	Inclusive	Calls/Tot.Calls	Name[id]
	2,262	2,262	1/1	int main(int, char **) [{mpi_mat_mul.cpp} {58,1
-->	2,262	2,262	1	void mat_mul_2(int, int) C [{mpi_mat_mul.cpp} {
	0,077	0,077	1/1	int main(int, char **) [{mpi_mat_mul.cpp} {58,1
-->	0,077	0,077	1	MPI_Finalize()

TAU Instrumentation API

- Using the specific API with TAU it's possible to obtain a very detailed profiling of your code.
- Code instrumentation based on the API can be done authomatically or manually. With manual code instrumentation the programmer can establish exactly **which sections** are to be profiled and **how**.
- TAU API is available for C++, C and Fortran77/90/95 codes and is portable among different platforms and compilers.
- To use the API at the beginning of each source to be profiled must be present the line: #include<TAU.h>
- Most important API capabilities:
  - **Routines profiling**
  - **Blocks or lines profiling**
  - **Heap-memory tracing**

TAU Instrumentation API

- Configuration and Initialization:
  - At the beginning of each instrumented source file, include the header "TAU.h"

```
TAU_PROFILE_INIT(argc, argv);
TAU_PROFILE_SET_NODE(myNode);
```

- Class funcitions and methods (C++ only):

```
TAU_PROFILE(name, type, group);
```

- User-defined timing

```
TAU_PROFILE_TIMER(timer, name, type, group);
TAU_PROFILE_START(timer);
TAU_PROFILE_STOP(timer);
```

- Heap-memory tracing:

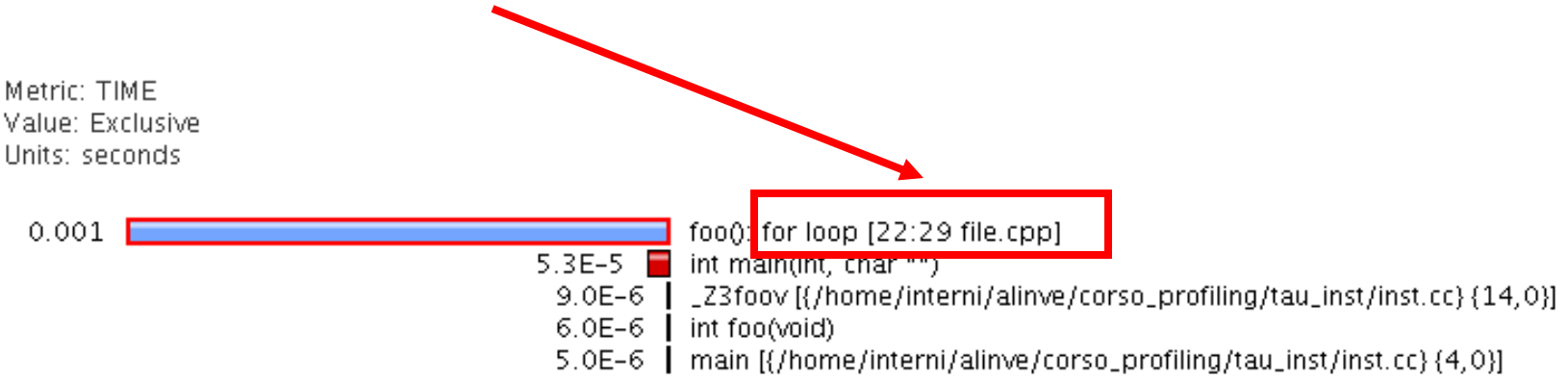
```
TAU_TRACK_MEMORY();
TAU_SET_INTERRUPT_INTERVAL(seconds);
```

C++ example

```
#include <TAU.h>

int foo();
int main(int argc, char **argv)
{
TAU_PROFILE("int main(int, char **)","", TAU_DEFAULT);
TAU_PROFILE_INIT(argc, argv);
TAU_PROFILE_SET_NODE(0); /* just for serial programs */
int cond=foo();
return 0;
}
int foo()
{
int N=100000;
double a[N];
int cond=0;
TAU_PROFILE("int foo(void)","", TAU_DEFAULT); // routine level profiling foo()
TAU_PROFILE_TIMER(t,"foo(): for loop", "[22:29 file.cpp]", TAU_USER);
TAU_PROFILE_START(t);
for(int i = 0; i < N ; i++){
a[i]=i/2;
if (i%2 ==0) cond=0;
else cond=1;
}
TAU_PROFILE_STOP(t);
if (cond==1) return 25;
else return 15;}
```

Example

With manual instrumentation using the API we can see detailed statistic information on a specific block of code

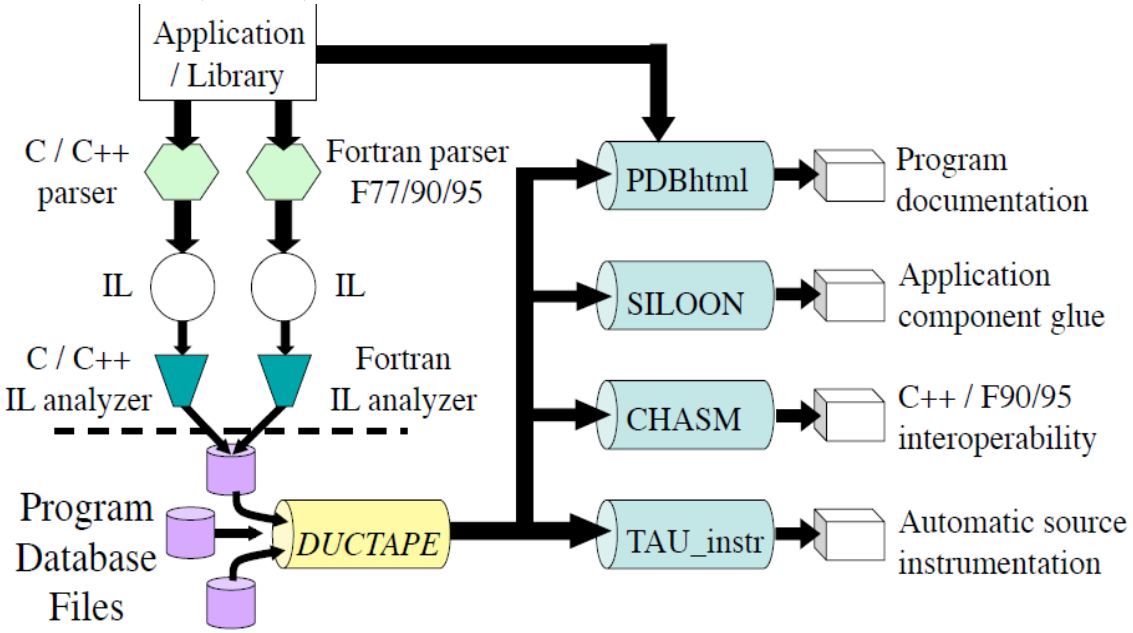


Fortran Example

```
PROGRAM SUM_OF_CUBES
integer profiler(2)
save profiler
INTEGER :: H, T, U
call TAU_PROFILE_INIT()
call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
call TAU_PROFILE_START(profiler)
call TAU_PROFILE_SET_NODE(0)
! This program prints all 3-digit numbers that
! equal the sum of the cubes of their digits.
DO H = 1, 9
DO T = 0, 9
DO U = 0, 9
IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
PRINT "(3I1)", H, T, U
ENDIF
END DO
END DO
END DO
call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
```

TAU source instrumentation with PDT

- Sometimes, for complex routines manual source instrumentation can become a long and error prone task.
- With TAU, instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT).



TAU source instrumentation with PDT

TAU and PDT howto:

- Parse the source code to produce the .pdb file:
  - `cxxparse file.cpp  C++`
  - `cparse  file.c  C`
  - `f95parse file.f90  Fortran`
- Instrument the program:
  - `tau_instrumentor file.pdb file.cpp -o file.inst.cpp -f select.tau`
- Complile:
  - `tau_compiler.sh file.inst.cpp -o file.exe`

TAU source instrumentation with PDT

- The ”-f” flag associated to the command “tau_instrumentator” allows you to customize the instrumentation of a program by using a selective instrumentation file. This instrumentation file is used to manually control which parts of the application are profiled and how they are profiled.
- Selective instrumentation file can contain the following sections:

1. Routines exclusion/inclusion list:

```
BEGIN_EXCLUDE_LIST / END_EXCLUDE_LIST
BEGIN_INCLUDE_LIST / END_INCLUDE_LIST
```

2. Files exclusion/inclusion list:

```
BEGIN_FILE_EXCLUDE_LIST / END_FILE_EXCLUDE_LIST
BEGIN_FILE_INCLUDE_LIST / END_FILE_INCLUDE_LIST
```

3. More detailed instrumentation specifics:

```
BEGIN_INSTRUMENT_SECTION / END_INSTRUMENT_SECTION
```

TAU source instrumentation with PDT

In a `BEGIN_INSTRUMENT_SECTION/END_INSTRUMENT_SECTION` block it's possible to specify the profiling of:

- Cycles

```
loops file="filename.cpp" routine="routinename"
```

- Memory

```
memory file="filename.f90" routine="routinename"
```

- I/O with dimension of read/write data

```
io file="foo.f90" routine="routinename"
```

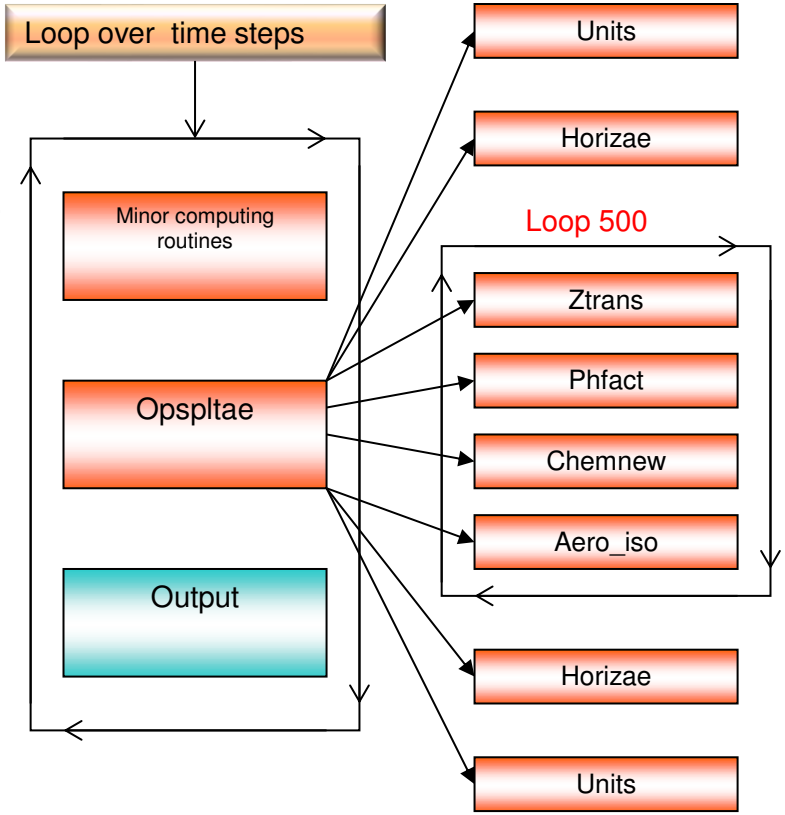
- Static and dynamic timers

```
static/dynamic timer name="name" file="filename.c"
  line=17 to line=23
```

TAU with PDT
Real Case Air Pollution Model

Instrumentation file : instrument_rules.txt

```
-----------------------------------------
BEGIN_FILE_INCLUDE_LIST
opspltae.f
chemnew.f
horizae.f
ztrans.f
END_FILE_INCLUDE_LIST


BEGIN_INSTRUMENT_SECTION

loops file="opspltae.f" routine="OPSPLTAE"
loops file="chemnew.f" routine="CHEMNEW"
loops file="horizae.f" routine="HORIZAE"
loops file="ztrans.f" routine="ZTRANS"
io file="wrout1.f" routine="WROUT1"
dynamic timer name="dyn_timer" file="opspltae.f" line=183 to line=189
END_INSTRUMENT_SECTION
-----------------------------------------
```



TAU with PDT
Real Case Air Pollution Model

Routine opspltae: Loop 500, TAU automatic instrumentation

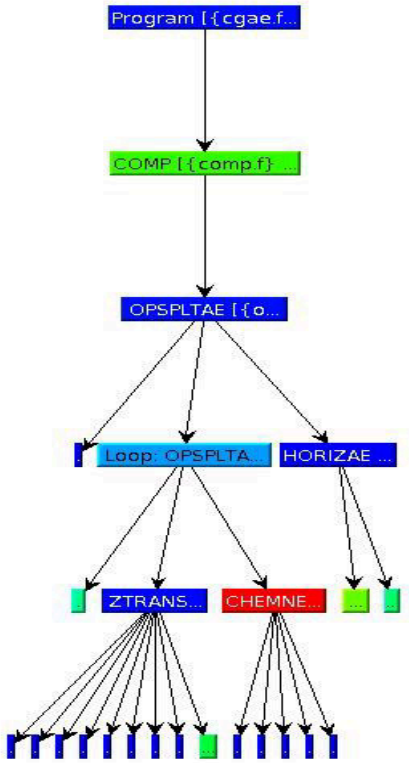
```
call TAU_PROFILE_TIMER(profiler, 'OPSPLTAE [{opspltae.f} {2,18}]')
call TAU_PROFILE_START(profiler)
call TAU_PROFILE_TIMER(t_131, ' Loop: OPSPLTAE [{opspltae.f} {131,7}-{143,12}]')
call TAU_PROFILE_TIMER(t_195, ' Loop: OPSPLTAE [{opspltae.f} {195,10}-{203,17}]')
call TAU_PROFILE_TIMER(t_247, ' Loop: OPSPLTAE [{opspltae.f} {247,7}-{592,14}]')
call TAU_PROFILE_TIMER(t_597, ' Loop: OPSPLTAE [{opspltae.f} {597,10}-{605,17}]')
call TAU_PROFILE_TIMER(t_639, ' Loop: OPSPLTAE [{opspltae.f} {639,10}-{647,17}]')
iugrid= iaddrs('UGRID   ',1,1,1,1,1)
                ……..
 call TAU_PROFILE_START(t_247)
   do 500 i=2,nxm1
      do 500 j=2,nym1
      ……..
      ……..
    500   continue
call TAU_PROFILE_STOP(t_247)
```

TAU TIMER Initialization

TAU Loop 500 instrumentation

TAU Loop 500 end instrumentation

TAU with PDT
Real Case Air Pollution Model



File Options Windows Help

Name △	Exclusive TIME	Inclusive TIME	Calls	Child Cal...
Program [{cgae.f} {111.7}]	0.652	1.913.031	1	2
COMP [{comp.f} {2.18}]	304.466	1.912.365	1	72
OPSPLTAE [{opspltae.f} {2.18}]	5.645	1.607.9	72	288
HORIZAE [{horizae.f} {2.18}]	0	219.192	72	144
Loop: OPSPLTAE [{opspltae.f} {131.7}-{143.12}]	0.185	0.185	72	0
Loop: OPSPLTAE [{opspltae.f} {247.7}-{592.14}]	63.181	1.165.318	72	10.445.760
AERO_ISO [{aero_iso.f} {2.18}]	171.098	171.098	4.419.360	0
CHEMNEW [{chemnew.f} {2.18}]	742.535	742.763	4.419.360	500.005
Loop: CHEMNEW [{chemnew.f} {64.7}-{66.14}]	0.033	0.033	100.001	0
Loop: CHEMNEW [{chemnew.f} {81.7}-{92.14}]	0.045	0.045	100.001	0
Loop: CHEMNEW [{chemnew.f} {103.7}-{106.14}]	0.025	0.025	100.001	0
Loop: CHEMNEW [{chemnew.f} {124.7}-{134.14}]	0.095	0.095	100.001	0
Loop: CHEMNEW [{chemnew.f} {145.7}-{148.14}]	0.029	0.029	100.001	0
ZTRANS [{ztrans.f} {2.18}]	2.333	188.276	1.607.040	2.407.048
Loop: ZTRANS [{ztrans.f} {101.7}-{104.14}]	0.038	0.038	100.001	0
Loop: ZTRANS [{ztrans.f} {108.7}-{114.14}]	0.031	0.031	100.001	0
Loop: ZTRANS [{ztrans.f} {131.7}-{134.14}]	0.028	0.028	100.001	0
Loop: ZTRANS [{ztrans.f} {137.7}-{145.14}]	0.034	0.034	100.001	0
Loop: ZTRANS [{ztrans.f} {156.7}-{160.14}]	0.028	0.028	100.001	0
Loop: ZTRANS [{ztrans.f} {203.7}-{206.14}]	0.026	0.026	100.001	0
Loop: ZTRANS [{ztrans.f} {209.7}-{222.14}]	0.027	0.027	100.001	0
Loop: ZTRANS [{ztrans.f} {236.7}-{247.14}]	0.04	0.04	100.001	0
Loop: ZTRANS [{ztrans.f} {253.7}-{351.14}]	185.692	185.692	1.607.040	0
dyn_timer [0]	0	2.467	1	1
dyn_timer [1]	0	3.262	1	1
dyn_timer [2]	0	3.255	1	1
dyn_timer [3]	0	3.209	1	1
dyn_timer [4]	0	3.215	1	1
dyn_timer [5]	0	3.21	1	1
dyn_timer [6]	0	3.224	1	1

Profiling time with default routine level compiler based instrumentation : 4192 sec
Profiling time with PDT and selective instrumentation : **1913 sec**
Execution time without profiling overhead: **1875 sec**

TAU: Memory Profiling C/C++

TAU can evaluate the following memory events:

- how much heap memory is currently used
- how much a program can grow (or how much headroom it has) before it runs out of free memory on the heap
- Memory leaks (C/C++)

TAU gives two main functions to evaluate memory:

- TAU_TRACK_MEMORY()
- TAU_TRACK_MEMORY_HERE()

Example:

```
#include<TAU.h>
int main(int argc, char **argv) {

   TAU_TRACK_MEMORY();
   sleep(12);
   double *x = new double[1024];
   sleep(12);
return 0; }
```

TAU: Memory Profiling Fortran

To profile memory usage in Fortran 90 use TAU's ability to selectively instrument a program. The option -optTauSelectFile=<file> for tau_compilier.sh let you specify a selective instrumentation file which defines regions of the source code to instrument.

To begin memory profiling, state which file/routines to profile by typing:

```
BEGIN_INSTRUMENT_SECTION
memory file="source.f90" routine="routine_name"
END_INSTRUMENT_SECTION
```

Memory Profile in Fortran gives you these three metrics:

- Total size of memory for each malloc and free in the source code
- The callpath for each occurrence of malloc or free
- A list of all variable that were not deallocated in the source code.

TAU: Memory leak Profiling

```
#include <stdio.h>
#include <malloc.h>
int bar(int value) { printf("Inside bar: %d\n", value);
 int *x;
 if (value > 5)
{ printf("looks like it came here from g!\n");
x = (int *) malloc(sizeof(int) * value);
x[2]= 2;
if (value > 15) free(x);
}
else {
printf("looks like it came here from foo!\n");
 x = (int *) malloc(sizeof(int) * 45);
 x[23]= 2;
free(x);}
return 0;}
 int g(int value) { printf("Inside g: %d\n", value);
 return bar(value); }
 int foo(int value) { printf("Inside f: %d\n", value);
 if (value > 5) g(value);
else bar(value);
return 0; }
```

TAU: Memory leak Profiling

```
int main(int argc, char **argv)
{ int *x; int *y;
foo(12);
foo(20);
foo(2);
foo(13);
}
```

To allow memory leak checking source code must be compiled using this TAU option:

```
export TAU_OPTIONS='-optDetectMemoryLeaks'
```

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
         2         52         48         50          2  MEMORY LEAK! malloc size <file=test2.cc, line=14> : g =>
                                                                                    bar
         1         80         80         80          0  free size <file=test2.cc, line=17>
         1         80         80         80          0  free size <file=test2.cc, line=17> : g => bar
         1        180        180        180          0  free size <file=test2.cc, line=24>
         1        180        180        180          0  free size <file=test2.cc, line=24> : foo => bar
         3         80         48         60      14.24  malloc size <file=test2.cc, line=14>
         3         80         48         60      14.24  malloc size <file=test2.cc, line=14> : g => bar
         1        180        180        180          0  malloc size <file=test2.cc, line=22>
         1        180        180        180          0  malloc size <file=test2.cc, line=22> : foo => bar
---------------------------------------------------------------------------------------
```

TAU and PAPI
Hardware Counter Measurements

- **How to**
  - Before compiling configure TAU with the flag `-papi=/path_to_papi_dir`
  - On BG/Q just load the module with : `module load tau/2.21.4`
  - Set `TAU_MAKEFILE` environment variable:
    `export TAU_MAKEFILE $TAU/Makefile.tau-gnu-papi-mpi-openmp-pdt`

  – Compile with TAU wrappers:
  – `tau_cc.sh example.cc -o my_exe`

  – Select hardware counters needed:

    - `export TAU_METRICS=GET_TIME_OF_DAY:PAPI_FP_INS:PAPI_L1_DCM`

TAU and PAPI
Hardware Counter Measurements

- Run the program

```
llsubmit launch_script.ll
```

- At the end of run a folder for each selected hardware counter will be created in the working directory

```
MULTI__GET_TIME_OF_DAY
MULTI__PAPI_FP_OPS
MULTI__PAPI_L1_DCM
```

- To analize results you can simply use `paraprof` gui.

PAPI events

Counter/Event Name	Meaning
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_L3_TCM	Level 3 cache misses
PAPI_FPU_IDL	Cycles floating point units are idle
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_STL_ICY	Cycles with no instruction issue
PAPI_HW_INT	Hardware interrupts
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_TOT_INS	Instructions completed
PAPI_FP_INS	Floating point instructions
PAPI_BR_INS	Branch instructions

Counter/Event Name	Meaning
PAPI_VEC_INS	Vector/SIMD instructions
PAPI_RES_STL	Cycles stalled on any resource
PAPI_TOT_CYC	Total cycles
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_L2_ICH	Level 2 instruction cache hits
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L2_ICA	Level 2 instruction cache accesses
PAPI_L1_ICR	Level 1 instruction cache reads
PAPI_L2_TCA	Level 2 total cache accesses
PAPI_L3_TCR	Level 3 total cache reads
PAPI_FML_INS	Floating point multiply instructions
PAPI_FAD_INS	Floating point add instructions (Also includes subtract instructions)
PAPI_FDV_INS	Floating point divide instructions (Counts both divide and square root instructions)
PAPI_FSQ_INS	Floating point square root instructions (Counts both divide and square root instructions)
PAPI_FP_OPS	Floating point operations

Example

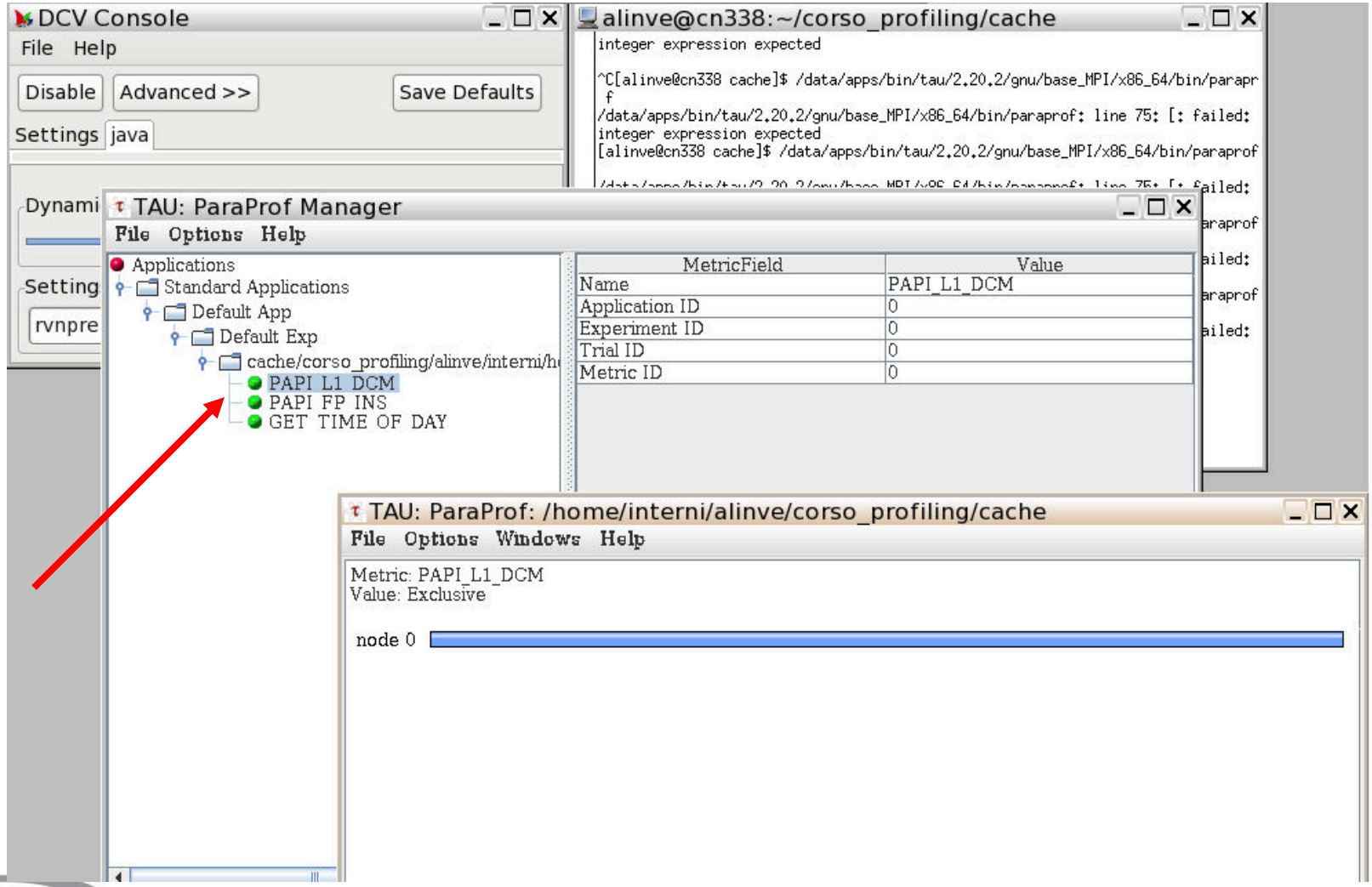
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define nn (2048)
double a[nn][nn], b[nn][nn], c[nn][nn]; /** matrici**/
int main()
{
  int k, i, j, ii, jj;
 float time1, time2, dub_time,somma;
  /* initialize matrix *7
  time1 = clock();
  for (j = 0; j < nn; j++)
    {
    for (i = 0; i < nn; i++)
      {
      a[j][i] = ((double)rand())/((double)RAND_MAX);
      b[j][i] = ((double)rand())/((double)RAND_MAX);
      c[j][i] = 0.0L;
      }
    }
  time2 = clock();
  dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
  printf("Tempo impiegato per inizializzare \n");
  printf("Tempo ----------------> %f \n", dub_time);
  time1 = clock();
for (i = 0; i < nn; i++)
        for (k = 0; k < nn; k++)
            for (j = 0; j < nn; j ++)
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
  time2 = clock();
  dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
  printf("================================\n");}
```

Option 2

```
for (j = 0; j< nn; j++)
      for (k = 0; k < nn; k++)
          for (i = 0; i < nn; i ++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Option 1

TAU: Paraprof



TAU - PAPI : Cache miss

Time (sec)		
Dimension	Option 1	Option 2
512	1.9	3.46
1024	10.42	19.45
2048	77.23	182.91

L1 Cache Misses		
Dimension	Option 1	Option 2
512	1.6938 E7	2.7585 E8
1024	1.3531 E8	2.2164 E9
2048	1.1339 E9	1.826 E10

MFlops		
Dimension	Option 1	Option 2
512	141.28	77.58
1024	206.09	110.41
2048	222.42	93.92