TAU:591C Final Report

Project Team

Sibin Mohan, Salil Pant, Harini Ramaprasad

{ smohan, smpant, hramapr }@unity.ncsu.edu

***Our Project**:*

- *Installation of Tuning and Analysis Utilities (TAU), a portable profiling and tracing toolkit, on the i32/Linux platform.*
- *Evaluation using ParBenCCh (from the ASCI suite of benchmarks).*

Project Topic – TAU

Solved issues:

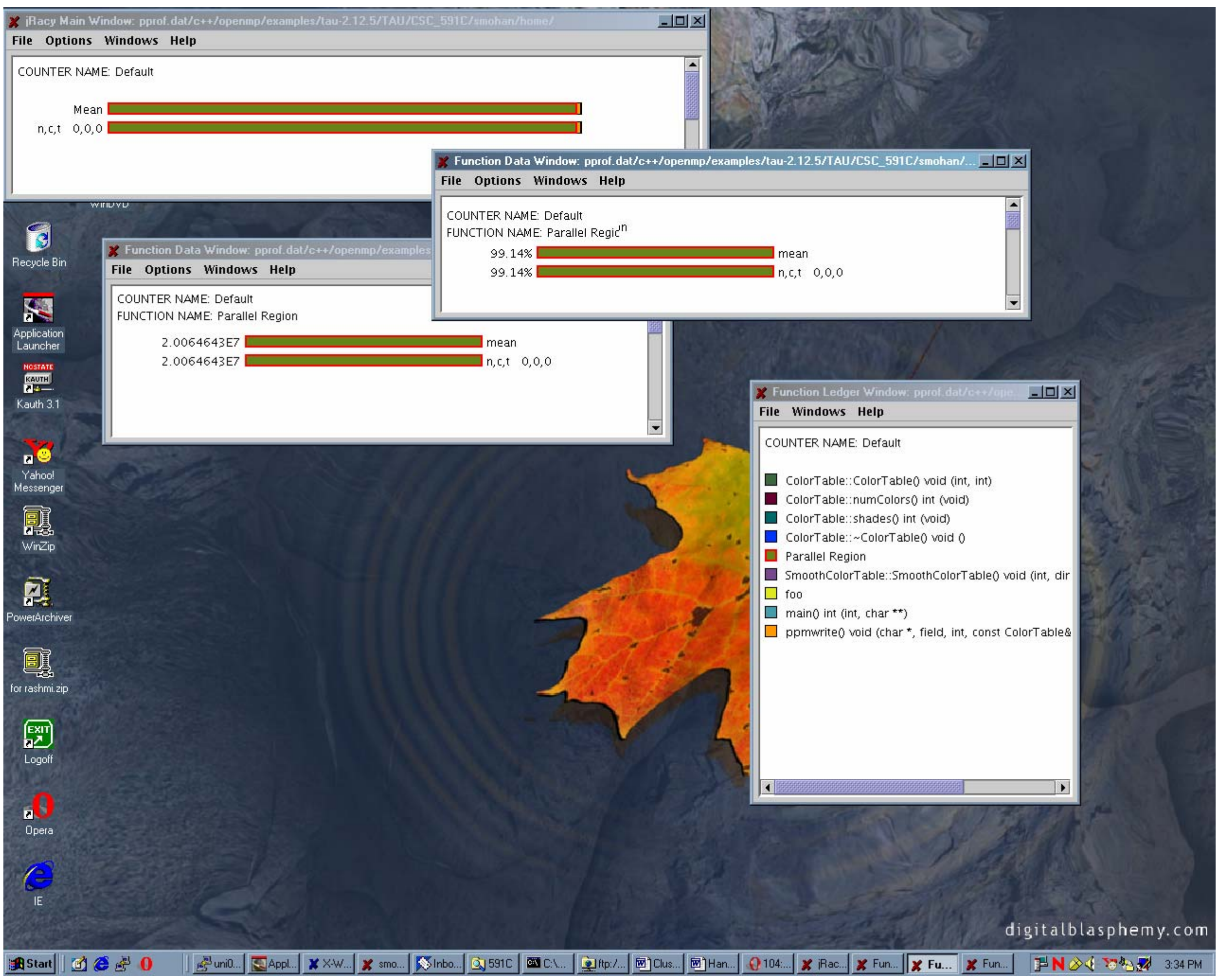
- *We configured TAU to work with OpenMP and OpenMP-MPI hybrid programs. In order to get it to work, we had to make the following changes to the makefile:*
    - *Include the library libompstub.a*
    - *Include the library libgm.so*

- *We ran an example program that uses OpenMP with TAU. The program involves calculations to generate the Mandelbrot set.*

    - *The results obtained from pprof are shown below.*

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
---
%Time Exclusive  Inclusive  #Call  #Subrs  Inclusive  Name
            msec total msec                  usec/call
---------------------------------------------------------------------------------------
---
100.0     0.505     20,238      1       5   20238593 main() int (int, char **)
 99.1    20,064     20,065      1     800   20065216 Parallel Region
  0.9       172        172      1       2     172817 ppmwrite() void (char *, field,
int, const ColorTable&)
  0.0     0.573      0.573    800       0          1 foo
  0.0     0.038      0.038      1       0         38 ColorTable::~ColorTable() void
()
  0.0     0.013      0.013      1       0         13 ColorTable::ColorTable() void
(int, int)
  0.0     0.004      0.004      1       0          4
SmoothColorTable::SmoothColorTable() void (int, direction, base, base, base)
  0.0     0.002      0.002      1       0          2 ColorTable::numColors() int
(void)
  0.0     0.001      0.001      1       0          1 ColorTable::shades() int (void)
```

- *A snap shot of jRacy outputs for the same program is shown below.*



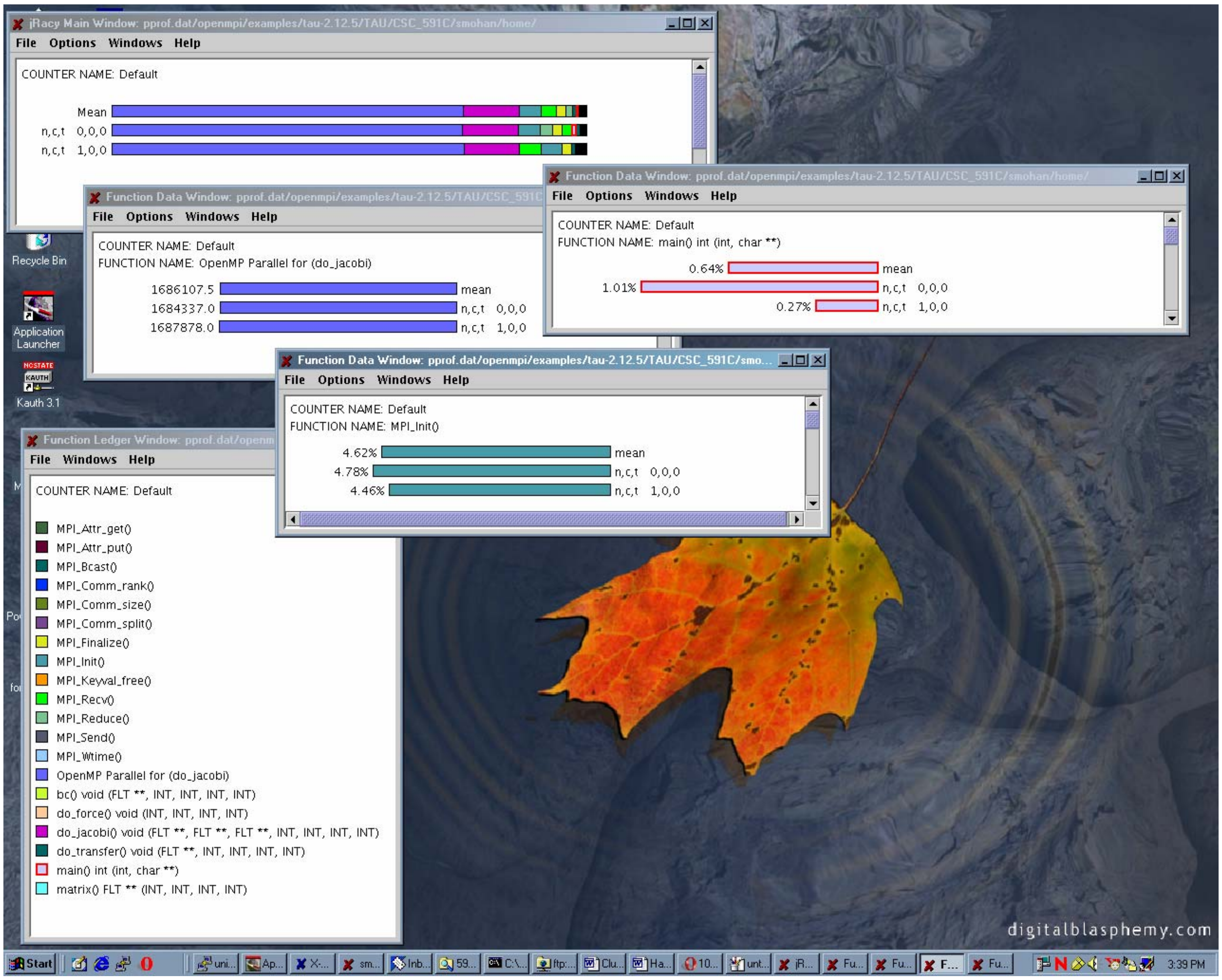
- *We ran an example program to test TAU for hybrid OpenMP-MPI. The program used is one that solves 2nd Stommel Model of Ocean Circulation using a Five-Point stencil and Jacobi iteration.*

    - *The results obtained from pprof for this program are shown below.*

```
FUNCTION SUMMARY (total):
---------------------------------------------------------------------------------
---
%Time Exclusive    Inclusive    #Call      #Subrs  Inclusive Name
           msec   total msec                        usec/call
---------------------------------------------------------------------------------
---
100.0        29        4,550        2        6050    2275313 main() int (int, char
**)
 85.9       537        3,909     2000      800000       1955 do_jacobi() void (FLT
**, FLT **, FLT **, INT, INT, INT, INT)
 74.1     3,372        3,372   800000           0          4 OpenMP Parallel for
(do_jacobi)
  4.8        41          218     2000       16000        109 do_transfer() void (FLT
**, INT, INT, INT, INT)
  4.6       210          210        2          16     105126 MPI_Init()
  3.4       154          154     8000           0         19 MPI_Recv()
  2.1        97           97        2           8      48588 MPI_Finalize()
  1.5        67           67     2000           0         34 MPI_Reduce()
  0.5        22           22     8000           0          3 MPI_Send()
  0.2        10           10       16           0        685 MPI_Bcast()
  0.1         4            4        2           0       2382 do_force() void (INT,
INT, INT, INT)
  0.1         2            2        4           0        619 MPI_Comm_split()
  0.0     0.236        0.236        4           0         59 MPI_Wtime()
  0.0     0.064        0.064        6           0         11 matrix() FLT ** (INT,
INT, INT, INT)
  0.0     0.013        0.013        2           0          6 bc() void (FLT **, INT,
INT, INT, INT)
  0.0     0.012        0.012        8           0          2 MPI_Attr_get()
  0.0      0.01         0.01        8           0          1 MPI_Attr_put()
  0.0     0.007        0.007        8           0          1 MPI_Keyval_free()
  0.0     0.004        0.004        6           0          1 MPI_Comm_size()
  0.0     0.003        0.003        6           0          0 MPI_Comm_rank()
```

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------
---
%Time Exclusive    Inclusive    #Call      #Subrs  Inclusive Name
           msec   total msec                        usec/call
---------------------------------------------------------------------------------
---
100.0        14        2,275        1        3025    2275313 main() int (int, char
**)
 85.9       268        1,954     1000      400000       1955 do_jacobi() void (FLT
**, FLT **, FLT **, INT, INT, INT, INT)
 74.1     1,686        1,686   400000           0          4 OpenMP Parallel for
(do_jacobi)
  4.8        20          109     1000        8000        109 do_transfer() void (FLT
**, INT, INT, INT, INT)
  4.6       105          105        1           8     105126 MPI_Init()
  3.4        77           77     4000           0         19 MPI_Recv()
  2.1        48           48        1           4      48588 MPI_Finalize()
  1.5        33           33     1000           0         34 MPI_Reduce()
  0.5        11           11     4000           0          3 MPI_Send()
  0.2         5            5        8           0        685 MPI_Bcast()
  0.1         2            2        1           0       2382 do_force() void (INT,
INT, INT, INT)
  0.1         1            1        2           0        619 MPI_Comm_split()
  0.0     0.118        0.118        2           0         59 MPI_Wtime()
  0.0     0.032        0.032        3           0         11 matrix() FLT ** (INT,
INT, INT, INT)
  0.0    0.0065       0.0065        1           0          6 bc() void (FLT **, INT,
INT, INT, INT)
  0.0     0.006        0.006        4           0          2 MPI_Attr_get()
  0.0     0.005        0.005        4           0          1 MPI_Attr_put()
  0.0    0.0035       0.0035        4           0          1 MPI_Keyval_free()
  0.0     0.002        0.002        3           0          1 MPI_Comm_size()
  0.0    0.0015       0.0015        3           0          0 MPI_Comm_rank()
```

- *The results obtained from jracy for this program are in the next figure. We see that the maximum time is spent in the omp parallel for sections while the next largest is for the calculation of the jacobians.*



Project Topic – ParBenCCh
Solved issues:

- *We installed and completely configured ParBenCCh. This required the following changes to be made to the makefile:*
  - *Include the library libompstub.a*
  - *Include the library libguide.so*

- *Once ParBenCCh was installed, we had to configure the various tests in the suite. The tests in the ParBenCCh suite are briefly described below (information obtained from the README on the website http://www.llnl.gov/asci/purple/benchmarks/limited/parbencch/parbencch.readme.html):*

  - *The Haney Test*

    This test compares the performance of matrix-vector operations in the following settings:
    - *Real matrix multiplication*
    - *Complex matrix multiplication*
    - *Real vector operations to test the cost of overloaded operators for operations on arrays*

  - *The Stepanov Test*

    This test measures the compiler support and performance of expression templates – a C++ template mechanism intended to achieve FORTRAN-like performance.

  - *The OpenMP Test*

    This is a test of OpenMP-style parallel direct and indirect addressing. In this test, a one-dimensional array of doubles whose size is close to the maximum heap size is allocated. The following operations are then performed on this array:
    - *Linear read*
    - *Linear read-write*
    - *Random read*
    - *Random read-write*

  - *Tests for Indirect Addressing*

    The tests in this directory exercise parallel indirect addressing using MPI-based parallelism. There are three related tests that are performed here.

Project Topic – running ParBenCCh tests with TAU

Solved issues:

- We ran the tests in the ParBenCCh suite using TAU. In order to be able to run the tests with TAU, the source code was instrumented according to instructions provided in the TAU installation guide.

- We first had to make the following changes to the makefile for the tests.
  - Include the TAU makefile stub (found in TAU_ROOT /i386_linux/lib directory) in the makefile of each test.
  - Add TAU_INCLUDE and TAU_DEFS in CXXFLAGS, CXXINCLUDE and CFLAGS.
  - Add –ltau, -lpthread and –lstdc++ in LIBS
  - Include the library /opt/papi/lib/libpapi.a
  - Include the TAU library TAU_ROOT/i386_linux/lib/libtau-mpi-pthread-papi-pdt-openmp.a

- The next step was the actual instrumentation of the source code of each test. The process is as follows:
  - Parse the source file using

    cxxparse <source_file>

    to generate a .pdb file.
  - Instrument the source file using

    tau_instrumentor <pdb file> <source file>

- Make and run the instrumented test programs.

- View the performance results as text using pprof and a graphical output using jRacy.

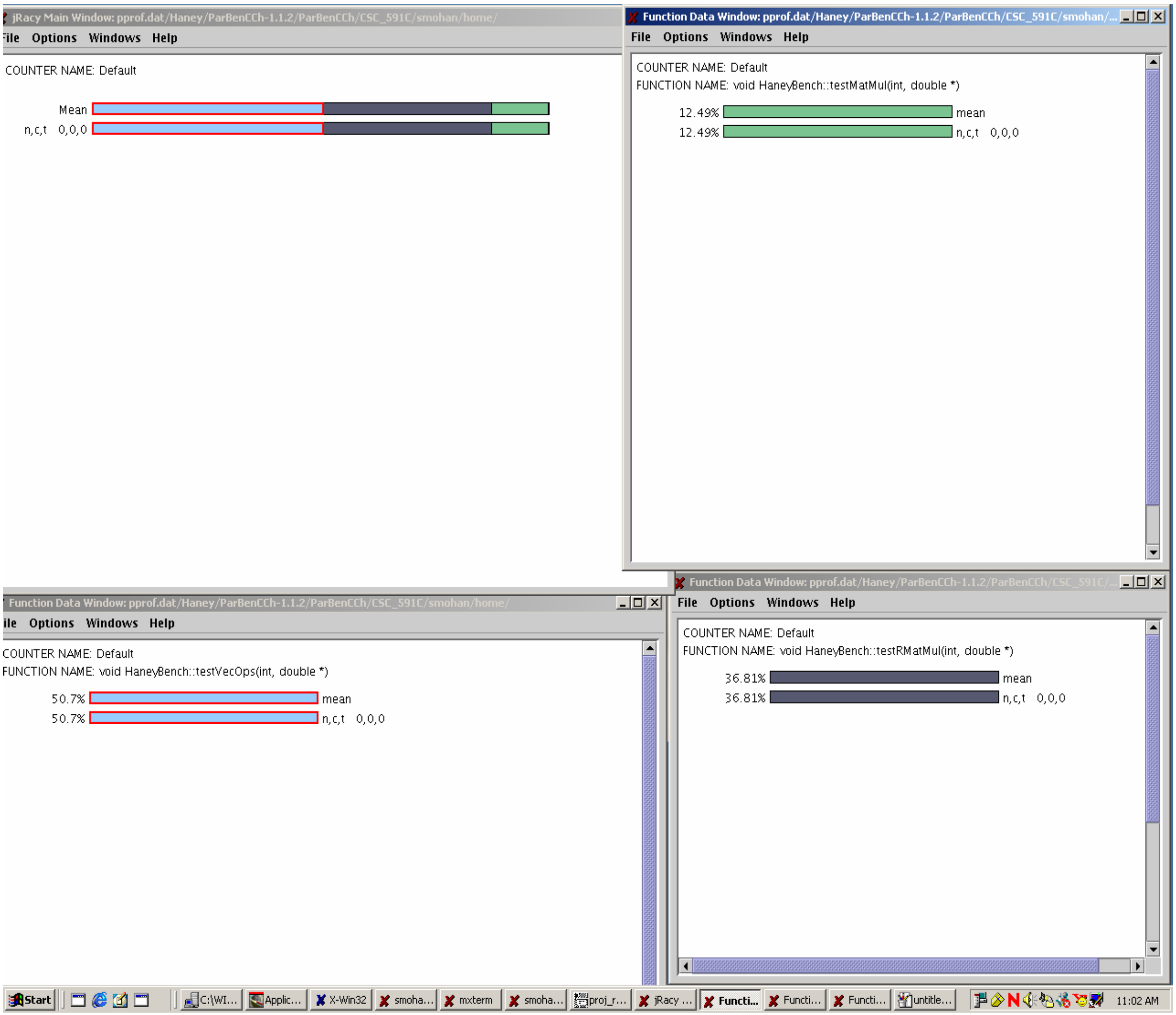
- The results obtained from these tests are shown below.
  - The Haney Test

<u>The results for the Haney Benchmark on pprof::</u>

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
---
%Time  Exclusive    Inclusive  #Call #Subrs  Inclusive Name
            msec   total msec                  usec/call
---------------------------------------------------------------------------------------
---
100.0          1    4:30.521      1      8  270521253 int main(int, char **)
 36.7   1:39.394    1:39.394     28      0    3549819 void
HaneyBench::testRMatMul(int, double *)
 12.6     33,983      33,983     28    336    1213684 void
HaneyBench::testMatMul(int, double *)
  0.0      0.373       0.373      1     84        373 void
HaneyBench::writeLabels(std::ostream &)
  0.0      0.001        0.06      1     28         60 void HaneyBench::finalize()
  0.0      0.034       0.034     84      0          0 ComplexArray3
&ComplexArray3::ComplexArray3(Integer, Integer, Integer, Boolean)
  0.0      0.029       0.029     84      0          0 ComplexArray4
&ComplexArray4::ComplexArray4(Integer, Integer,Integer,Integer, Boolean)
  0.0      0.002       0.002      1      1          2 HaneyBench
&HaneyBench::HaneyBench(int, int, int)
  0.0      0.001       0.001      1     28          1 void
HaneyBench::initialize(int, char **)
  0.0          0           0      1     84          0 void
HaneyBench::runBenchmark()
  0.0          0           0      1      0          0 void
HaneyBench::setArraySize(int)
  0.0          0           0     28      0          0 void
HaneyBench::testVecOps(int, double *)
  0.0          0           0      1      0          0 void
HaneyBench::writeToFile(std::ostream &)
  0.0          0           0      1      1          0 void
HaneyBench::~HaneyBench()
```

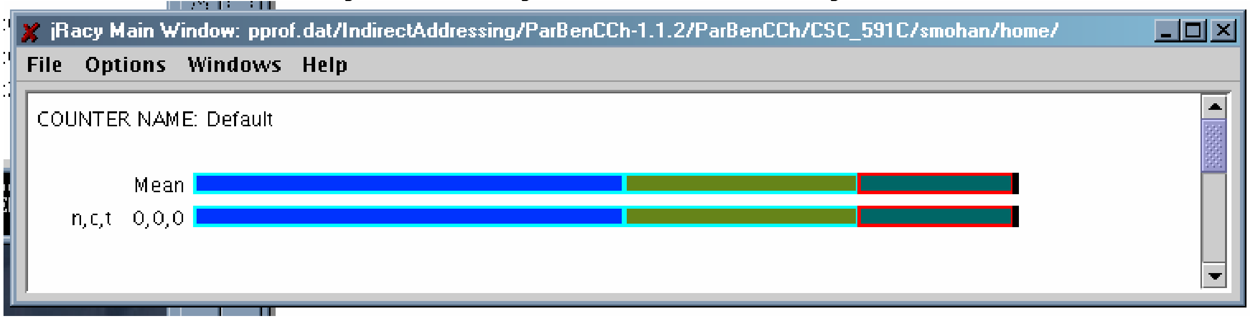
The above results show that the maximum number of calls were made to the constructors of the ComplexArray3 and ComplexArray4 classes. The most time was spent in the HaneyBench::testRMatMul() and HaneyBench::testMatMul() methods of the HaneyBench class.

- *jRacy outputs for the program are shown below.*

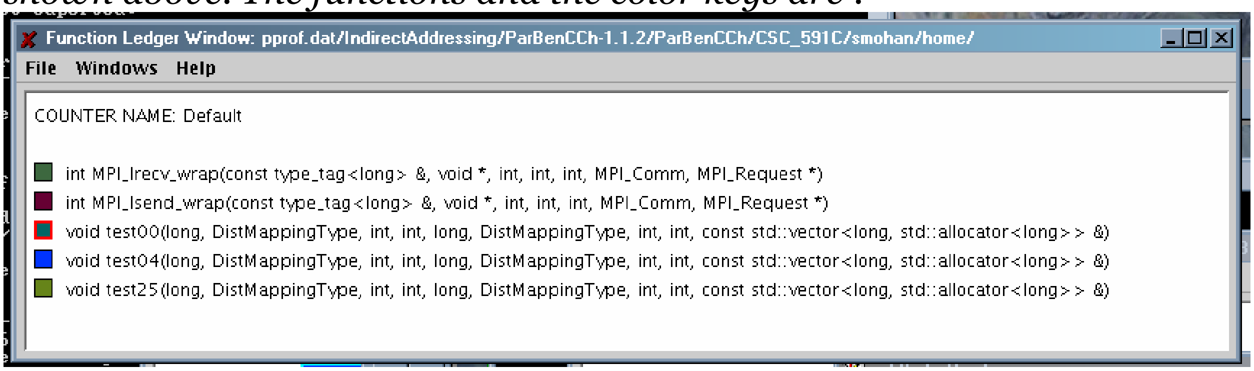


- *The Stepanov Test*
    - *Instrumentation for this test was causing segmentation faults. The Non-instrumented program was executing normally though.*
- *The Indirect Addressing Tests*

The results obtained from TAU for this test are as follows :

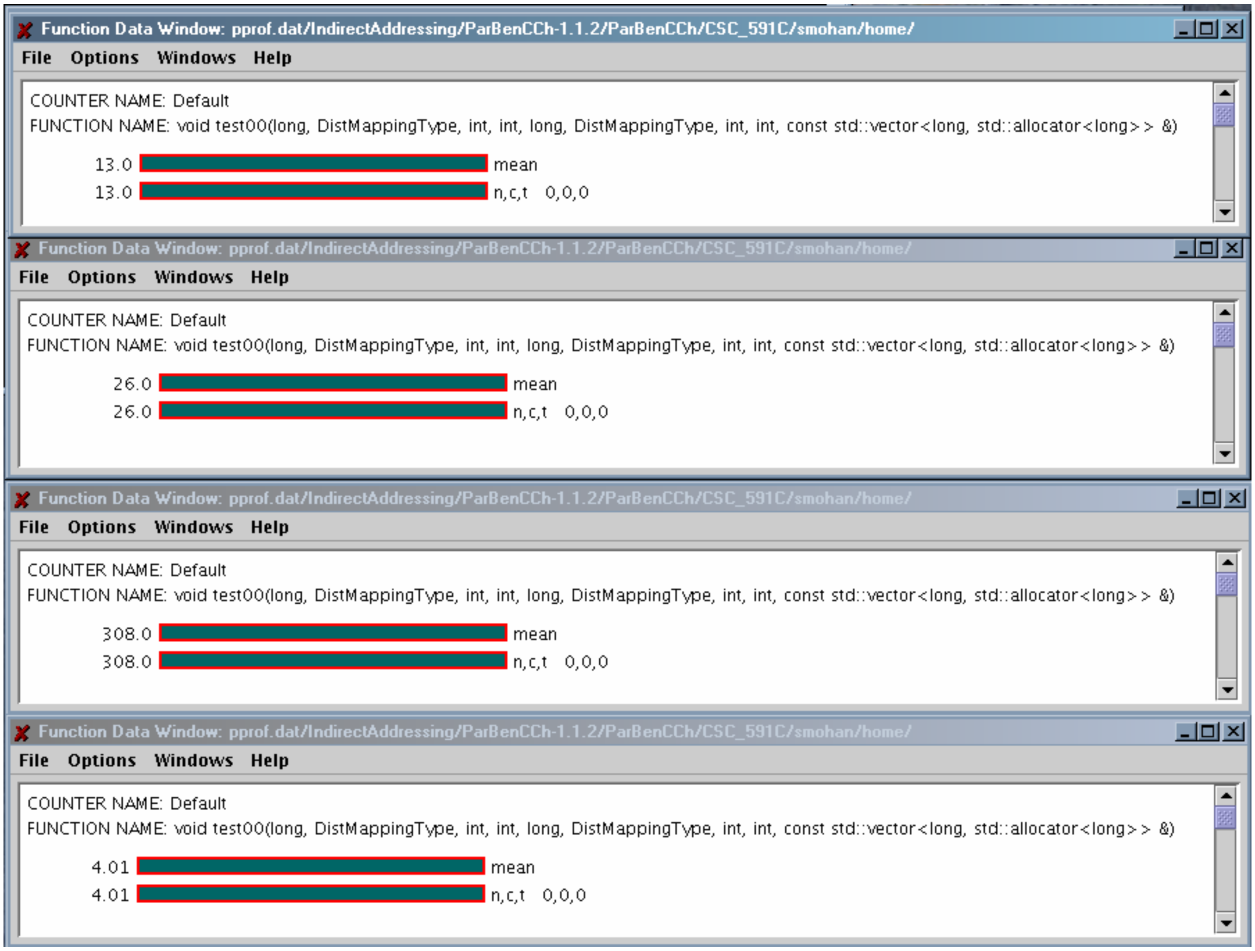


This shows the percentage of time spent in various functions are shown above. The functions and the color keys are :



the MPI_Irecv_wrap() and MPI_send_wrap() are wrappers in the benchmark for calling MPI functions.

Various statistics obtained for this program are (number of calls, number of subroutine calls, per-call-value and time spent inclusive of profiling, in milliseconds) :



[where n – node, c – context, t – thread]

These results are for the test00() function. It shows that this particular function was called 13 times, called other subroutines 26 times, took 308.0 microseconds for each invocation, and also that it took totally 4.01 milliseconds to execute all of it's invocations.

We also see that this particular function was 3rd in the most time consuming functions in this particular test (as seen from the very first image).

The pprof output for the same :

```
[smohan@os07 IndirectAddressing]$ 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           10           10          13          26        844 void test04(long,
DistMappingType, int, int, long, DistMappingType, int, int, const std::vector<long,
std::allocator<long>> &)
 54.4            5            5          13           0        459 void test25(long,
DistMappingType, int, int, long, DistMappingType, int, int, const std::vector<long,
std::allocator<long>> &)
 36.5            3            4          13          26        308 void test00(long,
DistMappingType, int, int, long, DistMappingType, int, int, const std::vector<long,
std::allocator<long>> &)
  0.4        0.044        0.044          26           0          2 int
MPI_Isend_wrap(const type_tag<long> &, void *, int, int, int, MPI_Comm, MPI_Request
*)
  0.2        0.026        0.026          26           0          1 int
MPI_Irecv_wrap(const type_tag<long> &, void *, int, int, int, MPI_Comm, MPI_Request
*)
```

Problems encountered and Changes required in TAU/ParBenCCh

We faced the following problems and we have listed what we believe should be improvements to the Profiler/Benchmark.

- *Instrumentation of programs – This was not an easy task, and involved a lot of intricate changes to the Makefiles, as well as source code, on our part.*
- *The libraries and paths were not being set and had to be often set manually, with the full path.*
- *Many coding bugs still exist in the Benchmark which made not only instrumentation difficult, but also just running the benchmark as is.*
- *The sample programs provided along with TAU also had problems in their source code and Makefiles – for eg. : the OpenMP sample provided wouldn't compile as is – there were problems in the Makefile which had to be corrected, even before the instrumentation.*
- *The automatic source-code instrumentor, tau_instrument, would often produce incorrect C++/C output files and put instrumentation code in the wrong places which would*

- *In certain cases, when comprehensive instrumentation was requested (for eg. : instrumentation of all source files in the Haney Test), Segmentation faults would occur, and hence the program/profiling would crash.*
- *In certain cases, (for Eg. : Stepanov test) any form of automatic instrumentation would cause Segmentation faults. Hence we were unable to obtain profile data for the same.*