

General Purpose Timing Library (GPTL)

A tool for characterizing parallel and serial
application performance

Jim Rosinski



Outline

- Existing tools
- Motivation
- API and usage examples
- PAPI interface
- Compiler-based auto-profiling
- MPI auto-profiling (uses PMPI layer)
- Usage examples
- Future work



Existing tools

- Gprof
- PAPI(ex)
- Fpmpi
- Tau
- Vampir
- Craypat

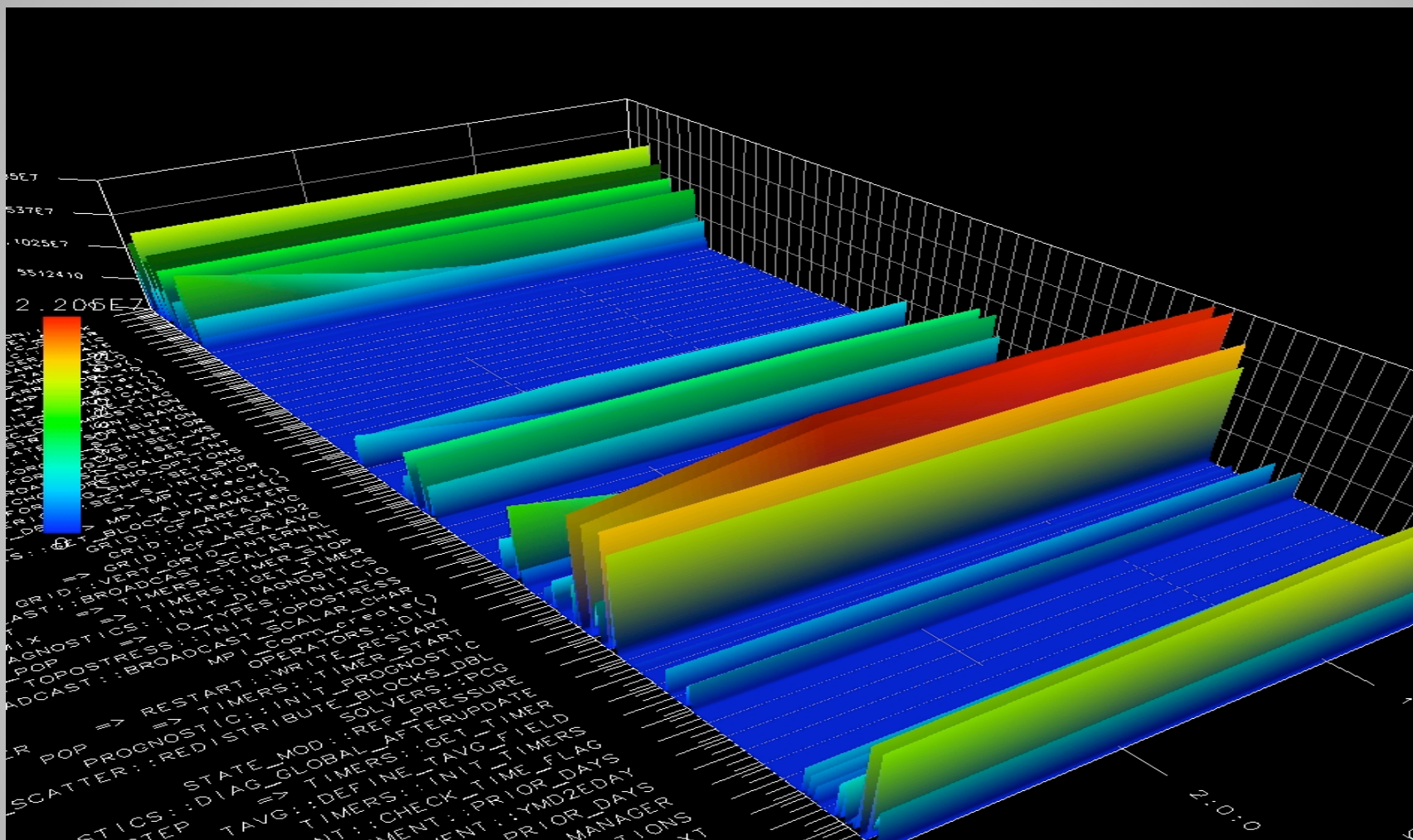


hpcprof

```
1342 0.1% do j=this_block%jb,this_block%je
1343 do i=this_block%ib,this_block%ie
1344 3.0% AX(i,j,bid) = A0 (i ,j ,bid)*X(i ,j ,bid) + &
1345 AN (i ,j ,bid)*X(i ,j+1,bid) + &
1346 AN (i ,j-1,bid)*X(i ,j-1,bid) + &
1347 AE (i ,j ,bid)*X(i+1,j ,bid) + &
1348 AE (i-1,j ,bid)*X(i-1,j ,bid) + &
1349 ANE(i ,j ,bid)*X(i+1,j+1,bid) + &
1350 ANE(i ,j-1,bid)*X(i+1,j-1,bid) + &
1351 ANE(i-1,j ,bid)*X(i-1,j+1,bid) + &
1352 ANE(i-1,j-1,bid)*X(i-1,j-1,bid) ^
```



TAU



Why use GPTL?

- Open source
 - Portable – runs on all UNIX-like Operating Systems
- Easy to use
 - Simple manual instrumentation
 - Compiler-based auto-instrumentation provides automatic dynamic call-tree generation
 - PMPI interface generates automatic MPI stats
- OK to mix manual and automatic instrumentation
- Thread-safe, provides info on multiple threads



Why use GPTL (cont'd) ?

- Assesses its own memory and wallclock overhead
- Utilities provided to summarize results across MPI tasks
- Free, already exists as a module on ORNL XT4/XT5
- Simplified interface to PAPI
- Derived events based on PAPI events (e.g. computational intensity)



Motivation

- Needed something to simplify, for an arbitrary number of regions to be timed:

```
time = 0;
for (i = 0; i < 10; i++) {
    gettimeofday (tp1,0);
    compute ();
    gettimeofday (tp2,0);
    delta = tp2.tv_sec - tp1.tv_sec +
            1.e6*(tp2.tv_usec - tp1.tv_usec);
    time += delta;
}
printf ("compute took %g seconds\n", time);
```



Solution

```
GPTLstart ("total");  
for (i = 0; i < 10; i++) {  
    GPTLstart ("compute");  
    compute ();  
    GPTLstop ("compute");  
    ...  
}  
GPTLstop ("total");  
GPTLpr_file ("timing.results");
```



Results

- Output file timing.results will contain:

	Called	Wallclock
total	1	3.983
compute	10	3.877



Fortran interface

- Identical to C except for case-insensitivity

```
include 'gpt1.inc'  
ret = gpt1start ('total')  
do i=0,9  
  ret = gpt1start ('compute')  
  call compute ()  
  ret = gpt1stop ('compute')  
  ...  
end do  
ret = gpt1stop ('total')  
ret = gpt1pr_file ('timing.results')
```



API

```
#include <gptl.h>
...
GPTLsetoption (GPTLoverhead, 0); // Don't print overhead
GPTLsetoption (PAPI_FP_OPS, 1); // Enable a PAPI counter
GPTLsetutr (GPTLnanotime);      // Better wallclock timer
...
GPTLinitialize ();              // Once per process
GPTLstart ("total");            // Start a timer
GPTLstart ("compute");          // Start another timer
compute ();                      // Do work
GPTLstop ("compute");           // Stop a timer
...
GPTLstop ("total");             // Stop a timer
GPTLpr (iam);                   // Print results
GPTLpr_file (filename);         // Print results
```



Available underlying timing routines

```
GPTLsetutr (GPTLgettimeofday); // default
GPTLsetutr (GPTLnanotime); // x86
GPTLsetutr (GPTLmpitime); // MPI_Wtime
GPTLsetutr (GPTLclock_gettime); // clock_gettime
GPTLsetutr (GPTLpapitime); // PAPI_get_real_usec
```

- Fastest and most accurate is GPTLnanotime (x86 only)
- Most ubiquitous is GPTLgettimeofday



Set options via Fortran namelist

- Avoid recoding/recompiling by using Fortran namelist option:

```
call gptlprocess_namelist ('my_namelist', unitno, ret)
```

- Example contents of 'my_namelist':

```
&gptln1  
  utr = 'nanotime'  
  eventlist = 'GPTL_CI', 'PAPI_FP_OPS'  
  print_method = 'full_tree'  
/
```



Threaded example

- GPTL works on threaded codes:

```
ret = gptlstart ('total')           ! Start a timer
!$OMP PARALLEL DO PRIVATE (iter)    ! Threaded loop
do iter=1,nompiter
  ret = gptlstart ('A')             ! Start a timer
  ret = gptlstart ('B')             ! Start another timer
  ret = gptlstart ('C')             ! Start another timer
  call sleep (iter)                 ! Sleep for "iter" seconds
  ret = gptlstop ('C')              ! Stop a timer
  ret = gptlstart ('CC') ↵
  ret = gptlstop ('CC') ↵
  ret = gptlstop ('A') ↵
  ret = gptlstop ('B') ↵
end do
ret = gptlstop ('total') ↵
```



Threaded results

Stats for thread 0:

	Called	Recurse	Wallclock	max	min
total	1	-	2.000	2.000	2.000
A	1	-	1.000	1.000	1.000
B	1	-	1.000	1.000	1.000
C	1	-	1.000	1.000	1.000
CC	1	-	0.000	0.000	0.000
Total calls	= 5				
Total recursive calls	= 0				

Stats for thread 1:

	Called	Recurse	Wallclock	max	min
A	1	-	2.000	2.000	2.000
B	1	-	2.000	2.000	2.000
C	1	-	2.000	2.000	2.000
CC	1	-	0.000	0.000	0.000
Total calls	= 4				
Total recursive calls	= 0				



PAPI details handled by GPTL

- This call:

```
GPTLsetoption (PAPI_FP_OPS, 1);
```

- Implies:

```
PAPI_library_init (PAPI_VER_CURRENT);  
PAPI_thread_init ((unsigned long (*)(void(pthread_self)));  
PAPI_create_eventset (&EventSet[t]);  
PAPI_add_event (EventSet[t], PAPI_FP_OPS));  
PAPI_start (EventSet[t]);
```

- PAPI multiplexing handled automatically, if needed



PAPI details handled by GPTL (cont'd)

- And these subsequent calls:

```
GPTLstart ("timer_name");  
GPTLstop ("timer_name");
```

- automatically invoke:

```
PAPI_read (EventSet[t], counters);
```

- GPTLstop also automatically computes:

```
sum[n] += counters[n] - countersprv[n];
```



Derived events

- Computational Intensity:

```
if (GPTLsetoption (GPTL_CI, 1) != 0);      // comp. intensity
if (GPTLsetoption (PAPI_FP_OPS, 1) != 0); // FP op count
if (GPTLsetoption (PAPI_L1_DCA, 1) != 0); // L1 dcache accesses
if (GPTLinitialize () != 0);
...
ret = GPTLstart ("millionFPOPS");

for (i = 0; i < 1000000; ++i) {
    arr1[i] = 0.1*arr2[i];
}

ret = GPTLstop ("millionFPOPS");
```

- 2 PAPI events enabled above:

$$\text{GPTL_CI} = \text{PAPI_FP_OPS} / \text{PAPI_L1_DCA}$$


Derived events (cont'd)

- Results:

Stats for thread 0:

	Called	Wallclock	max	min	CI	FP OPS	L1_DCA
millionFPOPS	1	0.006	0.006	0.006	5.00e-01	1.00e+06	2.00e+06

Total calls = 1
Total recursive calls = 0



Auto-instrumentation

- Works with Intel, GNU, Pathscale, and PGI

```
# icc -g -finstrument-functions *.c -lgptl  
# gcc -g -finstrument-functions *.c -lgptl  
# gfortran -g -finstrument-functions *.f90 -lgptl  
# pgcc -g -Minstrument:functions *.c -lgptl
```

- Inserts automatically at function start:

```
__cyg_profile_func_enter (void *this_fn, void *call_site);
```

- And at function exit:

```
__cyg_profile_func_exit (void *this_fn, void *call_site);
```



Auto-instrumentation (cont'd)

- GPTL handles these entry points with:

```
void __cyg_profile_func_enter (void *this_fn,  
                              void *call_site) ↵  
{  
    (void) GPTLstart_instr (this_fn);  
}
```

```
void __cyg_profile_func_exit (void *this_fn,  
                              void *call_site) ↵  
{  
    (void) GPTLstop_instr (this_fn);  
}
```



Auto-instrumentation (cont'd)

- User needs to add only:

```
program main
  ret = gptlsetoption (PAPI_FP_OPS, 1) ¶
  ret = gptlinitialize () ¶
  call do_work ()      ! Lots of embedded subroutines
  call gptlpr (iam) ¶ ! Print results for this MPI task
  stop 0
end program main
```



Raw auto-instrumented output

- Function addresses are printed:

Stats for thread 0:

	Called	Wallclock	max	min	% of	pop	FP_INS
pop	1	290.307	290.307	290.307	100.00	1.61e+09	
80ee040	1	35.855	35.855	35.855	12.35	3.52e+06	
81593b0	1	2.681	2.681	2.681	0.92	5	
8158e60	1	0.050	0.050	0.050	0.02	1	
8104840	1	0.089	0.089	0.089	0.03	25	
* 81571d0	460	0.038	0.001	0.000	0.01	460	
* 8157250	30	0.002	0.000	0.000	0.00	30	
* 81572e0	60	0.005	0.000	0.000	0.00	60	
8065270	1	0.000	0.000	0.000	0.00	1	
80751a0	1	0.012	0.012	0.012	0.00	57	
8158d60	1	0.000	0.000	0.000	0.00	1	
80644b0	1	0.001	0.001	0.001	0.00	1	
80a8890	1	0.026	0.026	0.026	0.01	62289	
80a5740	2	0.006	0.003	0.003	0.00	27538	
80a5e40	2	0.004	0.004	0.000	0.00	61322	
8075e60	1	17.820	17.820	17.820	6.14	2.10e+06	
* 8064e50	536794	6.840	0.000	0.000	2.36	536794	



Converting auto-instrumented output

- To turn addresses back into names:

```
# hex2name.pl [-demangle] <executable> <timing_file>
```

- Uses “nm” to determine entry point names which correspond to addresses



Converted auto-instrumented output (POP)

- Addresses converted to human-readable names:

Stats for thread 0:

	Called	Wallclock	max	min	% of	pop	FP_INS
pop	1	290.307	290.307	290.307	100.00	1.61e+09	
INITIALIZE_POP.in.INITIAL	1	35.855	35.855	35.855	12.35	3.52e+06	
INIT_COMMUNICATE.in.COMMUNICATE	1	2.681	2.681	2.681	0.92	5	
CREATE_OCN_COMMUNICATOR.in.COMM	1	0.050	0.050	0.050	0.02	1	
INIT_IO.in.IO_TYPES	1	0.089	0.089	0.089	0.03	25	
* BROADCAST_SCALAR_INT.in.BROADCA	460	0.038	0.001	0.000	0.01	460	
* BROADCAST_SCALAR_LOG.in.BROADCA	30	0.002	0.000	0.000	0.00	30	
* BROADCAST_SCALAR_CHAR.in.BROADC	60	0.005	0.000	0.000	0.00	60	
INIT_CONSTANTS.in.CONSTANTS	1	0.000	0.000	0.000	0.00	1	
INIT_DOMAIN_BLOCKS.in.DOMAIN	1	0.012	0.012	0.012	0.00	57	
GET_NUM_PROCS.in.COMMUNICATE	1	0.000	0.000	0.000	0.00	1	
CREATE_BLOCKS.in.BLOCKS	1	0.001	0.001	0.001	0.00	1	
INIT_GRID1.in.GRID	1	0.026	0.026	0.026	0.01	62289	
HORIZ_GRID_INTERNAL.in.GRID	2	0.006	0.003	0.003	0.00	27538	
TOPOGRAPHY_INTERNAL.in.GRID	2	0.004	0.004	0.000	0.00	61322	
INIT_DOMAIN_DISTRIBUTION.in.DOM	1	17.820	17.820	17.820	6.14	2.10e+06	
* GET_BLOCK.in.BLOCKS	536794	6.840	0.000	0.000	2.36	536794	



Multiple parent information

- GPTL prints the number of invocations by each parent:

```
2 INIT_IO.in.IO_TYPES
3 INIT_DOMAIN_BLOCKS.in.DOMAIN
1 INIT_GRID1.in.GRID
5 READ_HORIZ_GRID.in.GRID
2 READ_TOPOGRAPHY.in.GRID
662 INIT_DOMAIN_DISTRIBUTION.in.DOMAIN
2 READ_VERT_GRID.in.GRID
.....
2 INIT_TAVG.in.TAVG
2 INIT_MOORING.in.MOORINGS
932 BROADCAST_SCALAR_INT.in.BROADCAST
```



MPI auto-profiling

- Utilizes PMPI “hooks” to provide stats on MPI primitives
 - Bytes transferred, number of calls, time taken
- Can automatically add and time MPI_Barrier before communication primitives
- No need to call GPTLinitialize() or GPTLpr() if compiler supports iargc() and getarg()
 - Can profile with zero mods to user code
- Very similar to FPMPI



MPI auto-profiling example output

	Called	Recurse	Wallclock	max	min	AVG_MPI_BYTES
MPI_Init_thru_Finalize	1	-	0.055	0.055	0.055	-
MPI_Send	1	-	0.001	0.001	0.001	4.000e+05
MPI_Recv	2	-	0.000	0.000	0.000	4.000e+05
MPI_Ssend	1	-	0.000	0.000	0.000	4.000e+05
MPI_Sendrecv	1	-	0.000	0.000	0.000	8.000e+05
MPI_Irecv	2	-	0.000	0.000	0.000	4.000e+05
MPI_Iprobe	1	-	0.000	0.000	0.000	-
MPI_Test	1	-	0.000	0.000	0.000	-
MPI_Isend	2	-	0.000	0.000	0.000	4.000e+05
MPI_Wait	2	-	0.000	0.000	0.000	-
MPI_Waitall	2	-	0.000	0.000	0.000	-
MPI_Barrier	1	-	0.000	0.000	0.000	-
MPI_Bcast	1	-	0.001	0.001	0.001	4.000e+05
MPI_Allreduce	1	-	0.002	0.002	0.002	8.000e+05
MPI_Gather	1	-	0.005	0.005	0.005	3.200e+06
MPI_Gatherv	1	-	0.002	0.002	0.002	2.800e+06
MPI_Scatter	1	-	0.002	0.002	0.002	8.000e+05
MPI_Scatterv	1	-	0.001	0.001	0.001	3.200e+06
MPI_Alltoall	1	-	0.000	0.000	0.000	5.600e+01
MPI_Alltoallv	1	-	0.000	0.000	0.000	5.600e+01
MPI_Reduce	1	-	0.002	0.002	0.002	4.000e+05
MPI_Allgather	1	-	0.005	0.005	0.005	5.600e+06
MPI_Allgatherv	1	-	0.005	0.005	0.005	5.600e+06



Synchronizing MPI before communication

```
• ret = GPTLsetoption (GPTLsync_mpi, 1);
```

	Called	Wallclock	max	min	AVG_MPI_BYTES
sync_Allreduce	1	0.000	0.000	0.000	-
MPI_Allreduce	1	0.002	0.002	0.002	8.000e+05
sync_Gather	1	0.000	0.000	0.000	-
MPI_Gather	1	0.005	0.005	0.005	3.200e+06
sync_Gatherv	1	0.000	0.000	0.000	-
MPI_Gatherv	1	0.002	0.002	0.002	2.800e+06
sync_Scatter	1	0.000	0.000	0.000	-
MPI_Scatter	1	0.002	0.002	0.002	8.000e+05
sync_Scatterv	1	0.000	0.000	0.000	-
MPI_Scatterv	1	0.001	0.001	0.001	3.200e+06
sync_Alltoall	1	0.000	0.000	0.000	-
MPI_Alltoall	1	0.000	0.000	0.000	5.600e+01



Examining load imbalance

```
GPTLstart ("total");
GPTLstart ("sleep(iam)"); // "iam" is the process rank
sleep (iam);             // Do some load-imbalanced work
GPTLstop ("sleep(iam)");

// Synchronize (MPI_Bcast), then do load-balanced work

GPTLstart ("sleep(1)");
MPI_Bcast (&ret, 1, MPI_INT, commsize - 1, MPI_COMM_WORLD);
sleep (1);                // load-balanced work
GPTLstop ("sleep(1)");
GPTLstop ("total");
```



Results

Process 0:

	Called	Recurse	Wallclock	TOT_INS	e6 / sec
total	1	-	3.983	1.24e+09	311.55
sleep(iam)	1	-	0.000	968	161.33
sleep(1)	1	-	3.982	1.24e+09	311.56

Process 3:

	Called	Recurse	Wallclock	TOT_INS	e6 / sec
total	1	-	4.027	6863	0.00
sleep(iam)	1	-	3.017	1233	0.00
sleep(1)	1	-	1.010	1592	0.00



Examining load imbalance (cont'd)

```
GPTLstart ("total");
GPTLstart ("sleep(iam)");
sleep (iam);          // Do some load-imbalanced work
GPTLstop  ("sleep(iam)");

// Now, add a timed barrier before the synchronization

if (barriersync) {
    GPTLstart ("barriersync");
    MPI_Barrier (MPI_COMM_WORLD);
    GPTLstop  ("barriersync");
}
GPTLstart ("sleep(1)");
MPI_Bcast (&ret, 1, MPI_INT, commsize - 1, MPI_COMM_WORLD);
sleep (1);          // load-balanced work
GPTLstop  ("sleep(1)");
GPTLstop  ("total");
```



Results (barriersync = .true.)

Process 0:

	Called	Recurse	Wallclock	TOT_INS	e6 / sec
total	1	-	3.997	1.19e+09	297.52
sleep(iam)	1	-	0.000	969	138.43
barriersync	1	-	2.986	1.19e+09	398.27
sleep(1)	1	-	1.011	1416	0.00

Process 3:

	Called	Recurse	Wallclock	TOT_INS	e6 / sec
total	1	-	4.016	11723	0.00
sleep(iam)	1	-	3.006	1235	0.00
barriersync	1	-	0.000	3059	61.18
sleep(1)	1	-	1.010	1595	0.00



Aggregating across MPI tasks

- To compute stats across all threads and tasks:

```
# parsegptlout.pl [-c column] <region_name>
```

- E.g. for the POP run earlier:

```
# parsegptlout.pl -c 1 ELLIPTIC_SOLVERS
```

```
Found 192 calls across 192 tasks and 1 threads per task  
192 of a possible 192 tasks and threads had entries for ELLIPTIC_SOLVERS  
Heading is Wallclock  
Max    = 78.257 on thread 0 task 128  
Min    = 76.746 on thread 0 task 35  
Mean   = 77.161  
Total  = 14814.932
```



Utility functions

- Print memory usage stats:

```
GPTLprint_memusage ("testbasics after allocating 100 MB");
```

```
testbasics after allocating 100 MB size=140.2 MB rss=117.5 MB share=0.9 MB  
text=2.6 MB datastack=0.0 MB
```

- Retrieve wallclock, usr, sys timestamps to user code:

```
GPTLstamp (&wallclock, &usr, &sys);
```



Future work

- More derived events
- Support for more MPI primitives
- XML-based output for better visual display
- True per-parent call stats



Download & documentation

- www.burningserver.net/rosinski/gptl

www.burningserver.net/rosinski/gptl

- On ORNL xt4/xt5:
 - module load gptl
 - module load gptl_pmpi

