SAMRAI Timing & Instrumentation

Intro

SAMRAI provides a timing package to assist in performance analysis and to guide the user about which routines are incurring the largest percentage of computation time. The timers report basic statistics about the time spent in individual routines and can be turned on or off in the input file. The timers can report exclusive time, provide a calling tree hierarchy, and give estimated statistics on the overhead they take up in the program. The package also contains links to the VAMPIR parallel performance analysis package for visual representations of the computation and communication patterns of the application.

Instrumenting an application with the SAMRAI timing package involves wrapping timers around numerical kernels. A number of commonly used library components already have timers added and invoking these timers is simply a matter of specifying them through the input file. The first section of this chapter discusses how to invoke existing timers. The second section discusses how to add timers to your source code . Finally, techniques for analyzing performance on multiple processors using the VAMPIR and Tau instrumentation packages is discussed in the third section. An appendix is attached which lists a catalog of timers, and what they are timing, included in the library.

Invoking Timers

Management of the different timers in the library is performed through the *TimerManager* class. All timers are, by default, turned off. To turn them on, you must add an instance of TimerManager to your application. In main.C, add invocation of TimerManager:

```
TimerManager::createManager(input_db->getDatabase("TimerManager"));
```

Note that creation of the manager expects an entry from the input database. The input entries include a number of options for recording and printing timer values, and a list of timers to turn on. The options for recording and printing timers are as follows, and their default settings if not reset in the input, are as follows:

Input file format:

```
TimerManager{

   print_exclusive = FALSE
```

Specifies whether to track and print exclusive time. Exclusive time is measured by turning off the parent timer when a nested timer is called. The parent timer is turned back on when the nested timer is exited (Fig 1). Thus, the exclusive time is time spent *exclusively* in the routine. This option should be used with some discretion because the cost of maintaining the timer stack to record exclusive time is between four and seven times more expensive than simply turning a timer on and off. For this reason, it is false by default.

```
   print_total = TRUE
```

Specifies whether to track and print total (i.e. non-nested) time. This is the least expensive way to time parts of the code, with each occurrence of a start/stop operation incurring approximately 10 millionth of a second.

```
   print_wall = TRUE
```

Print wallclock time.

`print_user = FALSE`
: Print user time.

`print_sys = FALSE`
: Print system time.

`print_processor = TRUE`
: Prints time measured on individual processors.

`print_summed = FALSE`
: Prints time summed across all processors.

`print_max = FALSE`
: Prints maximum time measured across all processors, and the processor ID that incurred it.

`print_concurrent = FALSE`
: Prints a nested calling tree. For each timer, it prints the names of the timers that were nested within.

`print_percentage = TRUE`
: Prints percentages of the overall run time along with the measured times.

`print_timer_overhead = FALSE`
: Prints number of times a timer start/stop sequence was accessed, and the estimated overhead associated. Prints the total overhead from all timers as a percentage of the total run time and prints a warning if this is greater than 5%.

`print_threshold = 0.25`
: Specifies a threshold setting for which timers whose percentage of time is less than the threshold are not printed. That is, any timers that incur less than (print_threshold)% of the run time are not printed. Useful for preventing huge volumes of output if a lot of timers are being called. To disable completely, set it to zero.

`init_from_restart = FALSE`
: Timer values are written to restart with all the other restart information. One has the option to invoke them from restart so that timing information can be maintained over a series of runs. Set to true to initialize timers with values read from restart.

`timer_list = “pkg1::*::*”, “pkg2::classA::*”, “pkg3::classB::timer”`
: List of timers to be invoked. The timers can be listed individually or the entries may contain wildcards to turn on an entire set of timers in a specified package or class. Specifically, one can use the following formats in the timer list:

`package::*::*`	- turns on all timers in package.
`package::class::*`	- turns on all timers in class.
`*::class::*`	- turns on all timers in class
`class::*`	- turns on all timers in class
`class`	- turns on all timers in class
`package::class::timer`	- turns on specific timer in the package::class
`*::class::timer`	- turns on specific timer in the class
`class::timer`	- turns on specific timer in the class

}

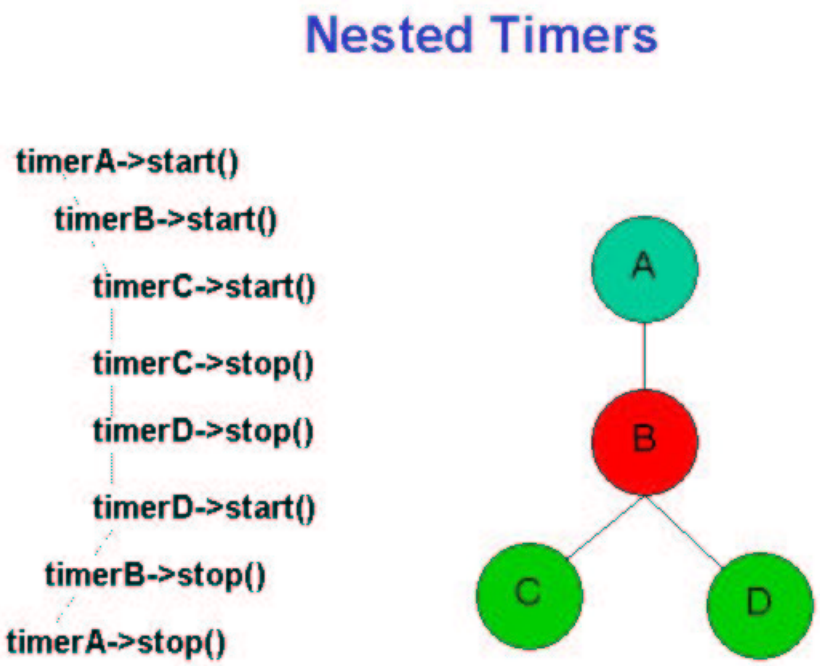


Figure 1 Exclusive time illustration

Timers can be printed at any point in the code by invoking the TimerManager's *print* function:

```
TimerManager::getManager()->print(pout);
```

The only argument required is the preferred output stream (e.g. pout, plog, or perr). The values printed are controlled by input file entries discussed above.

Adding Timers to New Code

Any code built with SAMRAI can utilize functionality of the TimerManager for timers in new code. However, the TimerManager assumes a certain naming and invocation format for each of the timers it manages and *these formats should be adhered to for the manager to work properly*.

The following is an example of how to invoke a timer in a piece of code:

```
static tbox_Pointer<tbox_Timer> t_method_name =
   tbox_TimerManager::getManager()->
   getTimer("package::class::methodName()");

t_method_name->start();

   perform function methodName()…

t_method_name->stop();
```

The primary features to note in the above invocation are discussed below:

`static`: We make a static pointer to the timer so that the TimerManager only has to search through its lists for the timer once, the first time the timer is invoked. The timer is has been requested in the input, maintained by the TimerManager, this call returns a pointer to the appropriate timer object. If the timer was not requested, the manager returns a pointer to its so-called "null" timer. The null timer is as a special case for which all calls to start/stop simply drop through without recording the time. This provides the capability of hardwiring a timer in the code but preserving the capability to turn it on or off at will.

`t_method_name:` Several timers may occur in the same region of the code, and it is sometimes difficult to discern pointers to the different timers. For this reason, we adopt a naming convention that the pointer to the timer begins with "t_" to designate it as a pointer to a timer object, followed by the "method_name" to designate the particular timer. This naming format is not a requirement for the TimerManager so user code can adopt whatever pointer name is most desirable. But for library code, this convention should be used to maintain consistency.

`package::class::methodName()`: All timers should be named in a format with two "::", as in package::class::timer. Use of this format is expected by the parsing routines in the manager that allow wildcard entries. That is, use of this format allows the capability to enter "package::*::*" in the input file and have package::class::timer be one of the timers turned on. If you do not use this format, the parser may not work properly.

One other feature that is not required by the TimerManager but is something we found to be a useful standard is the format of the method name. If the timer is timing an entire method, we specify the name like above as "methodName()". If the timer is placed around a specific part of the method, say a synchronization call, we name the timer something like "methodName()_sync". This distinguishes it as a piece of a certain method. If a timer is placed around a call in the class that is made in several methods, implying that its pointer is a data member of the class, we simply name the timer like a variable, such as "fill_data". The point of this formatting is to distinguish the different parts of the class that are being timed.

A number of timers have already been added to the library. See Appendix A for a catalog of the timer names and function they time.

Using VAMPIR

VAMPIR is a useful tool for analyzing performance of an application on a reasonable (i.e. < 16) number of processors. VAMPIR works by placing "phase markers" around what the user deems are important parts of the code. Most applications require the user to go in by hand to add these phase markers in order to use VAMPIR. In SAMRAI, however, we have instrumented the timing class to automatically invoke a vampir trace for every start and stop of a timer. Thus, adding a timer to the code will not only generate timing statistics at the end of the run, it will also allow analysis using the VAMPIR tool.

VAMPIR tracing is left off by default. This document discusses how to invoke it, and provides some details on how to use it to analyze application performance.

Configuring SAMRAI with VAMPIR

VAMPIR is installed and working at Livermore only on blue pacific. Thus, it is the default location known to the configure script. When compiling on this machine, add the --with-VAMPIR option to the configure line. If you happen to be running on a machine other than blue pacific and know where VAMPIR is installed, you may specify the directory in the configure.

Blue: `configure … --with-vampir`
Other Machines: `configure … --with-vampir=<dir>`

VAMPIR works with code compiled in optimized mode. That is, it is OK to configure with both `–enable-opt` and `–with-vampir`.

Once VAMPIR is compiled with the code, tracing will be invoked every time you run the code. Due to license issues, there are a few environment variables you must explicitly set or the code will not run. I usually put these in a little script, which I invoke whenever I start doing a run with code compiled with VAMPIR:

```
setenv PAL_ROOT /usr/local/kppp
setenv PAL_LICENSEFILE $PAL_ROOT/license.dat
set path=($PAL_ROOT/bin $path)
```

Once these environment variables are set, run the code. It will generate a trace file called `<exec>.stf`, where `<exec>` designates the name of your executable. The trace file contains statistics about code performance. It can be viewed graphically using the VAMPIR browser, invoked by the command `vampir`.

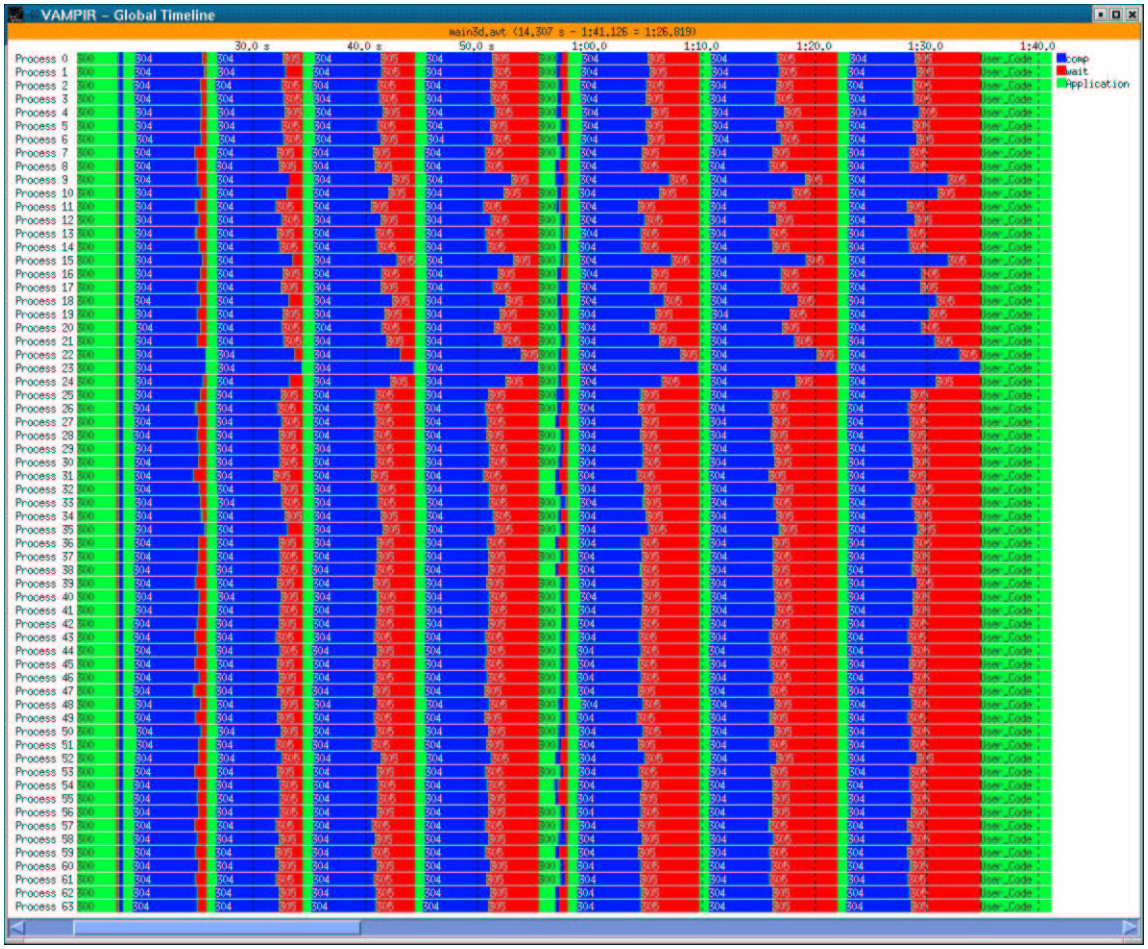


Figure 2 – VAMPIR Trace analysis on 64 processors

Using TAU

Tau is a tool developed at the University of Oregon to analyze code performance. The acronym stands for Tuning and Analysis Utilities – see *http://www.cs.uoregon.edu/research/paracomp/tau/*. Unlike VAMPIR which generates traces, Tau profiles an application. That is, it does not generate traces for every MPI call or call to a new method but instead records time for each of these operations and displays these at the end. Tau is capable of doing automated whole-code analysis, placing timers at the beginning and end of every method of every class. However, we have integrated Tau with the SAMRAI timers to make the integration simple. The only requirement to invoke Tau is to request it when you configure your version of SAMRAI.

Tau has the advantage over VAMPIR that it is freely-available and, unlike tracing which can incur significant overhead, its overhead is quite small. The Tau team, particularly Sameer Suresh, is very responsive and helpful and fixes most problems within a day of being reported.

This discussion is broken into two sub-sections. The first section discusses the minimal steps required to link to Tau. The second gives a brief tutorial on different functions that we have found useful. Appendix B summarizes the steps required to download and install Tau (we maintain a copy of Tau locally so this section is primarily intended for developers who need to update the installation and off-site users who wish to install versions on their own systems).

Configuring SAMRAI with Tau

Tau may be explicitly configured with SAMRAI by adding the `--with-tau` flag pointing to the tau Makefile. You must explicitly point to the version of tau that is compatible with the OS/compiler with which you are configuring SAMRAI:

I386-linux:	`configure --with-tau=/usr/casc/samrai/tau/tau-2.12/i386_linux/lib/Makefile.tau-linuxtimers-mpi`
gps:	`configure –-with-tau=/usr/local/tools/tau/alpha/lib/Makefile.tau-kcc`
frost:	`configure –-with-tau=/usr/casc/samrai/tau/tau-2.12/rs6000/lib/Makefile.tau-mpi-kcc`

If Tau becomes used on a regular basis, we will clean up the process so that the SAMRAI configure operation will automatically pull in the correct tau Makefile, but for now the appropriate version must be explicitly given in the configure line.

Using Tau to Assess Performance

Once you have configured with Tau, run the code in the way you are used to. Depending on the number of processors you run on, the file(s) `profile.[node].[context].[thread]` will be generated . SAMRAI only does node-level parallelism so, unless you invoke threaded code in your application, the context and thread entries should both be zero.

A graphical tool included with Tau to analyze performance is called “jracy”. Invoke it[1] via:

[1] If you plan to use tau frequently, you may wish to add the tau /bin to your path to avoid entering this long line.

```
<tau-dir>/bin/jracy
```

where <tau-dir> corresponds to the location where tau is installed. See the configure options above for the installation directory (<tau-dir> is everything before "/lib/Makefile.tau-…").

Jracy is a Java application and sometimes the default version of java installed on the blue/frost and gps/tc2k systems at LLNL is too dated to run it. Current versions of java exist on these systems but they may not be in your path by default. If you get an error when invoking jracy, try updating your version of java to something more current by setting your path as:

```
blue:   set path=(/usr/java130/bin $path)
gps:    set path=(/usr/opt/java122/bin $path)
```

Jracy will invoke a timeline window showing an aggregate of various timers from largest to smallest time on the different processors, and a mean from all processors. Click on the bars of any of the timers and a window will pop up showing a timeline for only that timer (Fig. 3).

Clicking on the "n,c,t" on the left side of each bar of the timeline will provide a breakdown of time spent in each routine on that particular processor (Fig. 4).

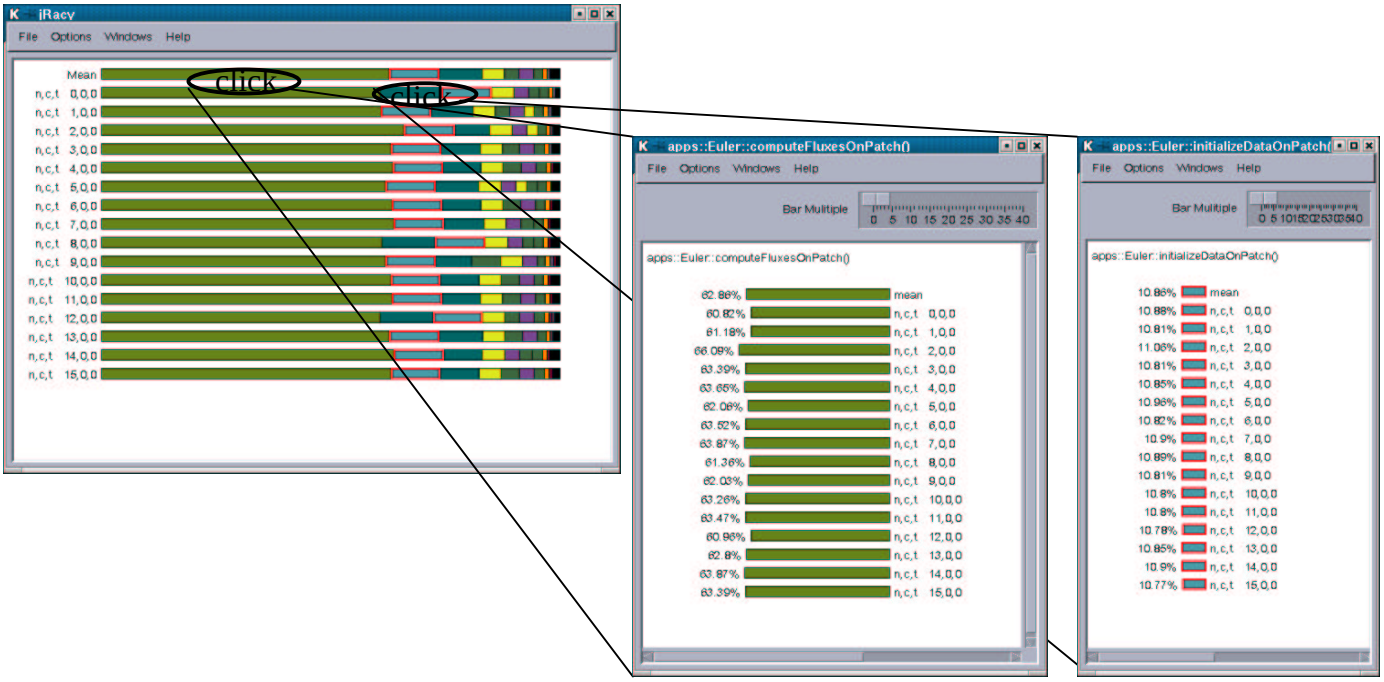
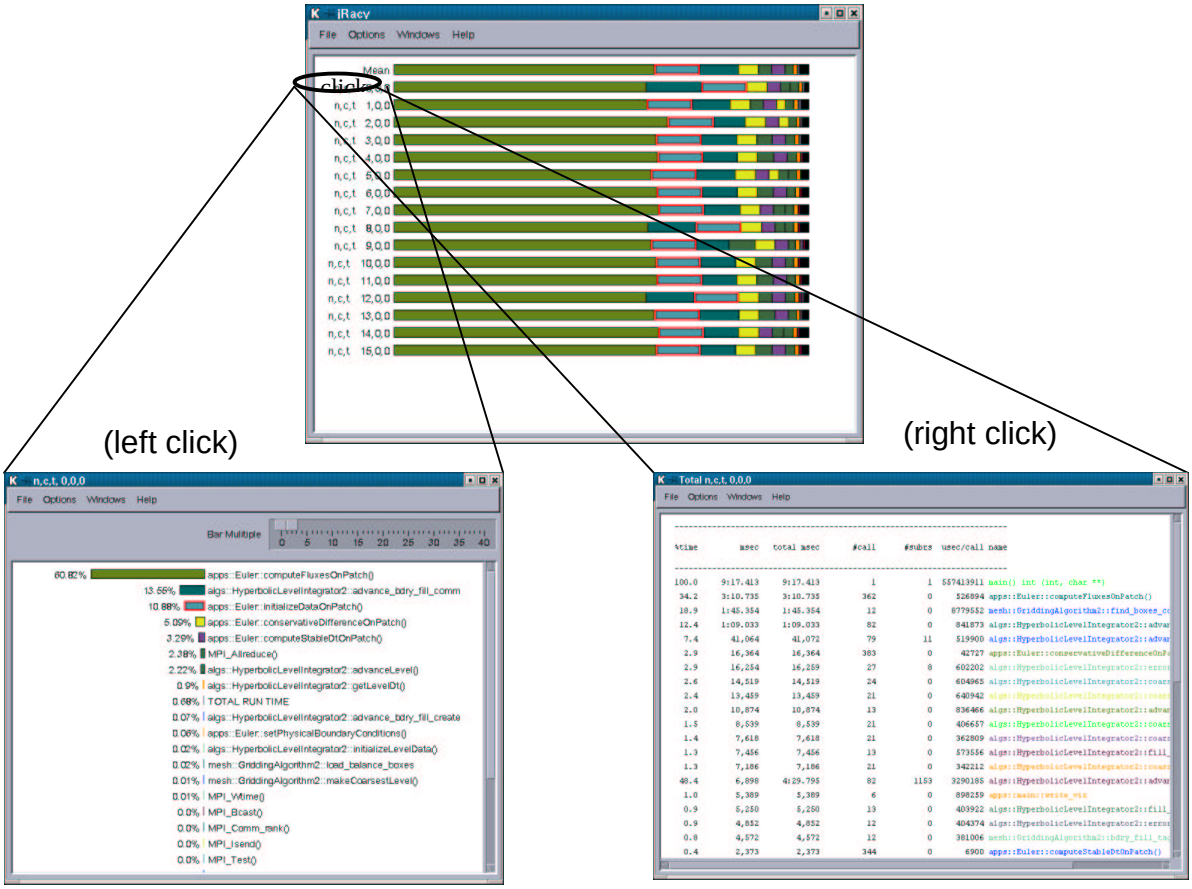


Figure 3 - Main window of tau's *jracy* tool



This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.
Document UCRL-TM-202188.

Appendix A

Catalog of Available Timers

Algorithms package: (algs::*::*)

TimeRefinementIntegrator (algs::TimeRefinementIntegrator::*)

algs::TimeRefinementIntegrator::initializeHierarchy() – times creation of hierarchy and initialization of data on the levels (initialization data may be invoked from scratch or read from restart).
algs::TimeRefinementIntegrator::advanceHierarchy() – times recursive advance of data as it steps thru each level in the hierarchy.

HyperbolicLevelIntegrator (algs::HyperbolicLevelIntegrator::*)

algs::HyperbolicLevelIntegrator::initializeLevelData() – time taken to set data on the hierarchy when hierarchy is initialized (not from restart).
algs::HyperbolicLevelIntegrator::applyGradientDetector() – time required for application of error estimator, or gradient detector.
algs::HyperbolicLevelIntegrator::advanceLevel() – time integration on the level.
algs::HyperbolicLevelIntegrator::resetHierarchyConfiguration() – time to reset hierarchy after a re-grid.
algs::HyperbolicLevelIntegrator::bdry_fill_comm – time to fill boundaries, using schedule created in the resetHierarchyConfiguration() routine.
algs::HyperbolicLevelIntegrator::fill_new_level_create – time to create schedule used to fill a newly-created level.
algs::HyperbolicLevelIntegrator::fill_new_level_comm – time to communicate data to the new level.
algs::HyperbolicLevelIntegrator::coarsen_fluxsum_create – time to create communication schedule for the fluxsum coarsen operation.
algs::HyperbolicLevelIntegrator::coarsen_fluxsum_comm – time to communicate data in the coarsen fluxsum operation.
algs::HyperbolicLevelIntegrator::coarsen_sync_create – time to generate schedule for the time synchronization step during coarsening.
algs::HyperbolicLevelIntegrator::coarsen_sync_comm – time to communicate data for the time synchronization step.
algs::HyperbolicLevelIntegrator::patch_numerical_routines time to perform numerical routines - compute fluxes and conservative difference – on patches of the level.
algs::HyperbolicLevelIntegrator::sync_initial_create – time to generate schedule for initialization of the time synchronization step.
algs::HyperbolicLevelIntegrator::sync_initial_comm – time to communicate during initialization of time synchronization.
algs::HyperbolicLevelIntegrator::getLevelDt()_sync – time for the MPI min reduction performed in the getLevelDt() function across all processes to determine the timestep. This is mainly a measure of load imbalance. While there is some MPI cost incurred in this operation, studies we have conducted show that the MPI cost is actually quite small compared to load imbalance costs.
algs::HyperbolicLevelIntegrator::advanceLevel()_sync – time for an MPI reduction performed in advanceLevel(). This is a mainly a measure of load imbalance. While there is some MPI cost

incurred in this operation, studies we have conducted show that the MPI cost is actually quite small compared to load imbalance costs.

Mesh package: (mesh::*::*)

GriddingAlgorithm (mesh::GriddingAlgorithm::*)

mesh::GriddingAlgorithm::makeCoarsestLevel() – time to construct coarsest level in hierarchy.
mesh::GriddingAlgorithm::makeFinerLevel() – time to construct finer level from coarser.
mesh::GriddingAlgorithm::regridAllFinerLevels() – time to do error estimation, generate and load balance boxes on new level, and regrid all levels in the hierarchy finer than the current one.
mesh::GriddingAlgorithm::regridFinerLevel() – time to regrid just one finer level.
mesh::GriddingAlgorithm::setTagsOnLevel() – time to set the error tags on the level.
mesh::GriddingAlgorithm::bufferTagsOnLevel() – time to add a buffer layer around the tags on the level.
mesh::GriddingAlgorithm::findRefinementBoxes() – time to construct refinement boxes from the buffered tags.
mesh::GriddingAlgorithm::findProperNestingBoxes() – once refinement boxes are constructed, must determine proper nesting to insure valid interpolations.
mesh::GriddingAlgorithm::remove_intersections_make_finer – time required to remove box intersections (call to BoxList::removeIntersections()) within makeFinerLevel().
mesh::GriddingAlgorithm::remove_intersections_regrid_all – time required to remove box intersections within regridAllFinerLevels().
mesh::GriddingAlgorithm::remove_intersections_find_proper – time required to remove box intersections within findProperNestingBoxes().
mesh::GriddingAlgorithm::intersect_boxes_find_proper – time required to determine box intersections (call to BoxList::intersectBoxes()) within findProperNestingBoxes.
mesh::GriddingAlgorithm::intersect_boxes_find_refinement – time to determine box intersections within findRefinementBoxes().
mesh::GriddingAlgorithm::find_boxes_containing_tags – time in the box generator strategy to determine tagged cells.
mesh::GriddingAlgorithm::load_balance_boxes – time to load balance the boxes once they have been generated.
mesh::GriddingAlgorithm::make_new_level – time to generate a new patch level (call to PatchHierarchy::makeNewPatchLevel() routine).
mesh::GriddingAlgorithm::bdry_fill_tags_create – time to build communication schedule for communication of tags.
mesh::GriddingAlgorithm::bdry_fill_tags_comm – time to communicate tag data.
.

Transfer package: (xfer::*::*)

CoarsenSchedule (xfer::CoarsenSchedule::*)

xfer::CoarsenSchedule::coarsenData() – time to perform communication during coarsening.

RefineSchedule (xfer::RefineSchedule::*)

xfer::RefineSchedule::fillData() – time to perform communication during refine operation.
xfer::RefineSchedule::generate_comm_schedule – time to build a schedule between patches on a patch level that have like refinement. For example, to exchange data between patches on the same level.

xfer::RefineSchedule::finish_schedule_const – time to build a schedule between patches that have different refinement. For example, to move data from a coarser to a finer level.

Applications package: (apps::*::*)

Euler (apps::Euler::*)

apps::Euler::initializeDataOnPatch() – time to set data on patch when started at time zero.
apps::Euler::computeStableDtOnPatch() – time to step thru data on the patch, computing timestep.
apps::Euler::computeFluxesOnPatch() – time to compute fluxes by performing the flux calculation routine (either Corner-Transport-Upwind scheme of Colella or scheme by Trangenstein).
apps::Euler::conservativeDifferenceOnPatch() – time to apply conservative difference, once fluxes have been calculated.
apps::Euler::setPhysicalBoundaryConditions() – time to apply boundary conditions.
apps::Euler::findErrorCells() – time to apply error detector and tag cells.

Appendix B

Installation of Tau and PDT

Installing Tau

1. Download the latest release of Tau from http://www.acl.lanl.gov/tau
2. Uncompress and go into the directory tau-x.x.xx. Configure using the following:

```
KCC: configure –c++=KCC
   –mpiinc=<mpidir>/include –mpilib=<mpidir>/lib
   –pdt=<pdtdir>/pdtoolkit-2.x

g++: configure –c++=g++ -cc=gcc
   –mpiinc=<mpidir>/include –mpilib=<mpidir>/lib
   –pdt=<pdtdir>/pdtoolkit-2.x
```

   The italicized arguments listed above are optional:

   -mpiinc -mpilib: If the location of MPI is specified, Tau will compile an MPI "wrapper" that tracks message traffic in your application. If SAMRAI was compiled without MPI, there is no need to configure Tau with MPI.

   -pdt: Specifies location of PDT installation – only necessary if using the PDT to do automatic instrumentation of source files.

3. Run **`make install`**

Installing PDT

The Program Database Toolkit (PDT) is used for doing whole-code instrumentation. It is a more heavyweight instrumentation option and may be non-trivial to configure but may be useful for codes with little instrumentation by SAMRAI timers. The steps below describe how to install it.

1. Download the Program Database Toolkit (PDT) from http://www.acl.lanl.gov/pdtoolkit
2. Uncompress and go into the directory `pdtoolkit-2.x`. Configure using the following:

```
KCC:   configure –KAI
g++:   configure -GNU
```

3. Run **`make`** followed by **`make install`**