Final Report on Tech-X Phase II SBIR project “SECAD – a Schema-based Environment for Configuring, Analyzing and Documenting Integrated Fusion Simulations”

Grant No DE-FG02-08ER85155

Tech-X Corporation

Principal Investigator Svetlana Shasharina (sveta@txcorp.com)

Date of Report: 16/5/12

Grant is supported by the DOE Office of Advanced Scientific Computing Research

1 Table of Contents

1	Table of Contents	1
2	List of Tasks in Phase II	2
3	Summary of Task 1	2
3.1	<i>Dakota Integration and Use in SWIM (IPS)</i>	<i>2</i>
3.2	<i>Dakota Integration and Use in FACETS</i>	<i>3</i>
4	Summary of Task 2	6
4.1	<i>Local Composer</i>	<i>6</i>
4.2	<i>Remote Composer</i>	<i>9</i>
4.2.1	<i>TxSsh Library</i>	<i>9</i>
4.2.2	<i>Remote Execution in Composer Toolkit</i>	<i>10</i>
4.2.3	<i>Remote Visualization in Composer Toolkit</i>	<i>13</i>
5	Summary of Task 3	13
5.1	<i>Initial instrumentation and changes in TAU</i>	<i>14</i>
5.2	<i>TAU Profile Analysis of CSWIM/IPS</i>	<i>16</i>
5.3	<i>Overhead Analysis of IPS Framework Design</i>	<i>18</i>
5.4	<i>Analysis of Toric</i>	<i>20</i>

2 List of Tasks in Phase II

Task 1: Implement Parameter and Optimization Studies for Integrated Fusion Simulations (SWIM and FACETS)

Task 2: Develop GUIs for FACETS, SWIM and DAKOTA

Task 3: Develop Performance Instrumentation.

The main goal of the project was to create a graphical user interfaces for two fusion frameworks SWIM (<http://cswim.org/>) and FACETS (<https://ice.txcorp.com/facets>), developed in two independent SciDAC projects. The only deviation from the tasks was not to integrate DAKOTA tool into the GUI as DAKOTA supports its own GUI now (Jaguar, http://dakota.sandia.gov/release_notes.html).

3 Summary of Task 1

3.1 Dakota Integration and Use in SWIM (IPS)

In accordance with concurrent IPS workflow, we created the following scripts:

- dakota_wrapper.sh, which parses the configuration file, finds IPS_ROOT and DAKOTA_ROOTS and exports them as environment variables, extracts parameter names and values from the DAKOTA configuration file for the parameter study, creates working directories, creates IPS components input files for each parameter value and invokes “dakota” executable.
- Helper Python scripts for making the above script work
- Python script for the batch submission of Dakota studies, franklin_4_debug_mcmd_sveta:

```
#!/bin/bash
#PBS -A m876
#PBS -N model_sim_run
#PBS -m e
#PBS -j oe
#PBS -l walltime=0:3:00
#PBS -l mppwidth=4
#PBS -q debug
#PBS -S /bin/bash
cd $PBS_O_WORKDIR
umask=0222
$HOME/projects/ips/bin/ips --
config=/scratch/scratchdirs/sveta/code_camp/model_sim_mcmd_sveta.conf
--platform=$HOME/projects/ips/franklin.conf --log=py.log --debug
```

This scripts is now submitted as follows: qsub franklin_4_debug_mcmd_sveta and successfully runs a parameter study for 4 values of the initial temperature in the center for the model IPS components.

The SWIM team (in particular Dr. Wael Elwasif) modified our scripts and performed

multiple DAKOTA studies performed outside of SECAD.

3.2 Dakota Integration and Use in FACETS

The DAKOTA toolkit has been used for several physics studies in the area of magnetically confined plasmas. In particular, DAKOTA has been used in the validation study of particle transport models in the plasma edge of the DIII-D tokamak discharges. The particle transport in this region has significant impact on the structure of the H-mode pedestal that determines the confinement of tokamak plasmas. In addition, the pedestal provides the boundary conditions for the hot core tokamak region and determines the stability properties of the plasma edge. That's why it is important to elucidate the primary mechanisms that are responsible for the pedestal structure in order to optimize the tokamak performance, avoid disruptions and large scale instabilities such as NTM and ELMs.

In the first study, the DAKOTA toolkit has been used together with the FACETS code to test the paleoclassical particle transport. The paleoclassical model is a new model that explains the enhanced thermal and particle transport in the near separatrix region by the diffusion of magnetic field lines that is induced by the plasma resistivity. The least square method and gradient-based unconstrained optimization technique from the DAKOTA toolkit has been used to determine the effective diffusivities that are needed in addition to the paleoclassical effective diffusivities in order to reproduce the experimental profiles. The hypothesis that the paleoclassical transport is the dominant transport mechanism in the plasma edge was considered to be proved if the additional effective diffusivity is less the paleoclassical at the mid-pedestal location.

Four DIII-D discharges have been analyzed. Fig. 1 shows the experimental and simulated electron density profiles, and the electron diffusive and convective particle fluxes for the DIII-D discharge 132017 as functions of the square root of the normalized toroidal flux, ρ . For all four DIII-D discharges, the additional effective diffusivities are found to be comparable or larger than the paleoclassical diffusivities. It has been concluded that the paleoclassical transport is not the major contributor to the particle transport in the plasma edge of DIII-D discharges.

The DAKOTA toolkit has been also used in a number of parameter scan studies. One of these studies has an objective to find plasma parameter range, where prediction from several anomalous transport models begin to diverge. While the agreement of different transport models with experimental data was approximately the same for the existing tokamaks, the models diverge when they are used for the prediction of plasma profiles in ITER. Two recently developed transport models, TGLF and MMM7.1, have been tested in this research. Both models describe the drift-wave anomalous transport and take into account similar physical effects, but they use different physical assumptions. In order to understand the ranges of plasma parameters where these assumptions affect the predictions of anomalous transport, a systematic comparison was necessary. In this next study, a set of analytical equilibria has been generated using the TOQ equilibrium solver and the DAKOTA toolkit. These equilibria represent a wide range of basic plasma parameters such a plasma densities, temperatures, currents, and plasma shaping including elongation and triangularity. The plasma profiles for these equilibria were evolved in the

integrated whole-device modeling FACETS code. It has been found that the predictions from different transport models are smallest for the existing tokamaks such as DIII-D. The differences are maximized for an intermediate plasma regime between DIII-D and ITER.

Figure 1. The experimental and computed electron density profiles for the DIII-D discharge 132017 are shown on

the left panel. The corresponding electron particle diffusive, convective and total fluxes as function of square root of the normalized toroidal flux are shown on the lower panel.

Another parameter study that utilizes the DAKOTA toolkit was the ETG verification study for several DIII-D discharges. Following the experimental scenario, plasma parameters for the H-mode pedestal has been modified in the way so that the critical ETG parameter

$$\eta = \frac{\partial T_e}{\partial n_e}$$

changes in the experiment relevant ranges. Here, The corresponding plasma parameters has been passed to the equilibrium solver TOQ using DAKOTA. Set of equilibria that corresponds to different values of η has been generated and used in the FACETS code to verify the predictions for ETG thermal diffusivities from the GLF23, TGLF, Horton-Jenko, and GYRO modules. Fig. 2 shows the effective diffusivities computed using the GLF23, TGLF, and Horton-Jenko models for two limiting cases that correspond to smallest and largest values of η . It has been found the predictions for the ETG thresholds agree reasonable well for different models. However, the stiffness of the ETG models is found to be very different. The GLF23 model is found to be less stiff for the ETG thermal transport among the models tested in this research; and Horton-Jenko model is found to be the most stiff model.

Similar computation protocol has been used in other DAKOTA parameter studies such as neoclassical bootstrap current verification study.

Figure 3. Effective diffusivities computed with the GLF23, TGLF and Horton-Jenko models for two different equilibria that correspond to two cases with smallest and largest values of η .

4 Summary of Task 2

4.1 Local Composer

The code that we develop in this project is called Composer (Fig. 4). Composer is split into FACETS Composer and IPS Composer, with the IPS being the name of the framework of SWIM. The main goal of Composer is to provide a GUI for the workflows associated with running high-performance simulations with fusion SciDACS FACETS and SWIM.

The Composer Toolkit

User Interface Foundations for
High Performance Computing Applications

Fig 4. Composer: a GUI for running HPC simulations.

We split the workflow associated with a single HPC run performed by FACETS and IPS into 5 steps:

- Create (creating of the area on the disc space and obtaining all files that are needed for the run, including all input for simulation components, scripts and data file generated by other codes or by experiments). Once the runspace gets created, the user obtains the master file for editing. The master file determines the workflow and has extension .ctkwf (see Fig. 5).
- Setup (preprocessing, including generating core meshes, data interpolation, expanding formulas in input file, format transformations)
- Run (for example, mpiexec or batch command, with some postprocessing that is not optional). While run is performed, the standard out is exposed to the user.
- Analyze (user-dependent data analysis and data transformation). This step can happen many times after and before the visualization step and is optional.
- Visualize (application specific visualization using embedded VisIt, see Fig. 6). The default viz brought up to the user has the expected dimensionality, expected lineouts and allow time slider with the movie if many steps are present.

This steps are defined in a file with .ctkwf extension. If they are not defined, or the file is not present, than each Composer flavor performs a default workflow. For FACETS it consists of running fluxgrid to generate core mesh, executing the facets version specified in shell or python script (both are supported), transforming all generated data in HDF5 files (as some legacy components do not use HDF5 natively) and bring the default 2D visualization with the

Figure 5. The master file gets opened after the create step. All other steps are available on the left.

symmetric lineout in the viz panel. For IPS it consists of running ips executable with the .conf file. No default viz is performed at the moment although we have verified that most of the NetCDF files generated by IPS can be imported into VisIt and are visible in Composer.

Fig. 6. Visualization window of FACETS Composer. It provides default 3D, 2D and 1D (lineout) views that can be manipulated through the menus.

4.2 Remote Composer

In addition to the above capabilities we implement a simplified GUI for editing input files and also “remoting” capability that allows to use ssh tunnel and run Composer locally but the codes themselves remotely.

4.2.1 TxSSH LIBRARY

Remote execution is performed by the TxSsh library, which is used as svn extern in Composer toolkit. TxSsh wraps NetSieben SSH Library underneath, which is a Secure Shell client software written in C++ utilizing Botan library for it's cryptographic functions. The TxSsh API provides simple and easy interface to create ssh connections supporting both private key and password based authentication. Allows for easy methods to create Sftp clients to perform operations like get, put, ls, cd, mkdir, execute commands on remotely, etc.

Code sample to download a remote file:

```
TxSsh* ssh = new TxSsh();
std::string hostName = "iter.txcorp.com", userName = "abc", password = "abc";

// Connect using password
```

```
if (ssh->connectWithPassword(hostName, userName, password, 0, 20)) {
    std::cerr << "Error occurred while connecting to " << hostName << std::endl;
} else {
    std::cout << "Successfully connected to " << hostName << std::endl << std::endl;
}

// Get Sftp client
TxSftp* sftp = new TxSftp(ssh);

// Get remote file
std::string remFile = "build.out", locFile = "buildLoc.out";
if (sftp->get(remFile, locFile)) {
    std::cerr << "Error occurred while getting remote file " << remFile
        << " from " << hostName << std::endl;
} else {
    std::cout << "Successfully downloaded remote file " << remFile
        << " from " << hostName << std::endl << std::endl;
}
```

4.2.2 REMOTE EXECUTION IN COMPOSER TOOLKIT

Remote operations in Composer Toolkit involve connecting to a remote host securely, listing files, uploading and downloading files, and executing commands, on the remote host. All these operations are easily provided by the TxSsh library.

Fig. 7 shows the setup for using Composer Toolkit for remote execution. The Composer for the specific engine (Vorpal, Facets, Nautilus, etc.) will be run locally. The UI of the composer provides interfaces (Fig. 8) for entering host information. It is possible to create multiple host profiles. All the host profiles are saved for future Composer sessions except the password for security reasons. The actual engines will have to be installed on the remote host, and the appropriate paths need to be provided to the composer in the host settings (Fig. 9).

A remote workspace can be opened (Figs. 10, 11) using the UI. The necessary input files will be copied onto a remote workspace automatically and the engines will be run on the remote host using the TxSsh library through the composer user interface.

Currently, the remote operations supported are connecting to a remote host using the Host Profiles Settings, Opening Remote Runspaces which involve listing of remote files, downloading the remote file and displaying it.

Fig. 7. Setup of remote execution in Composer.

Fig. 8: Host settings for entering user login information.

Fig. 9: Host settings for entering required paths to commands.

Fig. 10. Opening remote runspace.

Fig. 11. Displaying remote runspace.

4.2.3 REMOTE VISUALIZATION IN COMPOSER TOOLKIT

Composer Toolkit uses the client/server architecture of VisIt to perform remote visualization. At the Composer level, all visualization operations appear to be happening locally. All remote operations are handled seamlessly by the embedded VisIt engine. For both local and remote, when the run finishes the Composer Toolkit creates a list of data files in the runspace. It then passes this list of files to the embedded VisIt engine to be opened. If a file is remote, we use a special syntax to specify the name of the host profile that should be used to connect to that remote host. For each remote host, we also send host information to VisIt - including such things as host address, ssh port, and path to VisIt. From this point on, Composer does not distinguish between local and remote visualization operations.

5 Summary of Task 3

This work has been performed by Paratools, Inc. (Wyatt Spear, Sameer Shende, and

Allen D. Malony) as a part of subcontract from Tech-X.

5.1 Initial instrumentation and changes in TAU

Our initial goal was to examine the performance of the IPS framework by itself, without the added complexity of executing particular physics applications. However there were a number of technical challenges to overcome before even preliminary performance data collection from the IPS framework was possible.

The primary issue was the Python threading model used by IPS. Although TAU supports analysis of threaded applications in general, at the beginning of this undertaking there had been no prior testing with multi-threaded Python applications. We quickly determined that thread spawning and management activity of IPS was not being captured by TAU. This meant that profile output from child processes was not being differentiated from the parent process. This resulted in performance profiles that only represented the activity of the main process.

In order to address this issue it was necessary to catch the Python threading calls used by IPS. We accomplished this via library interposition. With the aid of the *tau_exec* utility the thread spawn and kill calls were replaced with wrapper calls that initiated the necessary TAU threading controls before the system level thread operations were invoked. Invocation of a simple IPS test with TAU instrumentation *appeared* as follows:

```
tau_exec -XrunTAUsh-pathscales-mpi-pthread-python-pdt python
~/SWIM/trunk/bin/wrapper.py --config=hybrid002_PT.config --
platform=$IPS_ROOT/franklin.conf -debug -log=hy040510_002_pbs.log
```

This change resulted in distinct profile files for the main IPS thread as well as its subordinate threads as shown in Fig. 8.

Fig. 8. Main IPS Python thread profile.

Our next tests invoked IPS with simple demonstration physics code module. These

revealed another technical issue unique to performance analysis with the IPS framework. TAU normally operates on a single application, but IPS invokes multiple applications, each with their own main methods. This could result in conflated profile output being overwritten by the last process to terminate. We solved this by further modifying our thread interposition system to ensure that the profiles from each distinct application are placed in a unique profile output directory.

After addressing these fundamental issues of TAU support for IPS we began collecting data from relevant SWIM Integrated physics codes. We first instrumented and tested these codes as standalone applications. Afterwards, we will collect profile data from the instrumented physics codes as run within the IPS framework. Target physics codes that we have investigated to date include NUBEAM (as shown in Fig. 9) that has been a successful subject of TAU based analysis in the past.

Fig. 9. Example Nubeam Profile Output from Standalone Run.

Direct articulation with application developers has been essential in our continued progress. For example, we are working closely with the developers of the M3DC1 code from PPPL, not only to generate performance data as shown in Fig. 10 but to assist them in conducting their own performance analysis experiments.

Metric: TIME
Value: Exclusive percent

Fig. 10. Example M3DC1 Profile Output from Standalone Run

The performance experiments conducted so far pave the way for more intensive investigation of the SWIM codes. Issues which we will be addressing in future work include selective instrumentation of the various codes, to reduce instrumentation overhead and produce data that focuses on the most salient components of the code. We will also be expanding the use cases and configurations, which we test. In general, a larger battery of variables, such as hardware performance metrics and IO/Memory behavior remain to be examined. These issues must all be approached with significant input from the application developers, to ensure the data we provide is relevant to their needs and actionable in tuning the elements of SWIM. As the volume of performance data increases we will make use of our performance database system to keep it organized and readily accessible. Further improvements to TAU, such as refinement of support for Python threading, will arise from our continued participation in this undertaking.

5.2 TAU Profile Analysis of CSWIM/IPS

The ParaTools team performed a performance measurement experiment on the CSWIM/IPS framework. The goal of this experiment was to use TAU to identify and measure the overhead of the Python IPS framework. The experiment was performed on Hopper, a Cray XE6 system at NERSC. The input data and configuration were provided by the CSWIM/IPS team, and consisted of a small test case integrating the simulations TSC, Toric and Nubeam. The experiment was executed on 96 cores. TAU was configured for use with the Pathscale compilers, including pthread and python support.

The ParaTools team checked out the latest IPS code at the time, configured and built IPS using the default Hopper build settings. In order to measure the IPS framework, the tau_exec wrapper was used to preload selected system calls in order to capture when the Python runtime system called fork() and exit(). This method provides TAU the ability to correctly measure each

Python process within the IPS framework. In addition, the TAU support for measuring Python functions was enabled.

In the collected profile, eight IPS component processes were identified and measured. Five of these processes are involved in startup, logging and coordination between the three simulation applications, and the other three represent the simulation applications. As can be seen in Fig. 11, all processes spend some amount of time (~3.8%) in the function “lstat”, which represents filesystem overhead. From this measurement, it would appear that the only significant “overhead” penalty the IPS framework incurs is limited to this time.

As can also be seen in Fig. 11, the processes 7803, 7827 and 7873 are the simulation processes, which spend most of their time in the Python calls “recv” and “waitpid”. Waitpid represents the time spent actually running the simulation, and “recv” represents the time that one simulation waits on one or both of the other processes. It appeared to the ParaTools team that there was a resource imbalance between the processes. That is, the TSC and Nubeam simulations were spending significant time waiting on the Toric simulation, which was confirmed by the IPS developers. However, this test represented only 3 timesteps of a simulation restart, and the IPS team explained that the combination of resource requirements for the simulations is not static,

Fig. 11. IPS Python processes in sample experiment.

but rather changes over time.

5.3 Overhead Analysis of IPS Framework Design

While the previous experiment measured the overhead directly attributable to the IPS infrastructure, the developers also wished to know the overhead due to the overall design. In particular, the developers were concerned about how long the “aprun” command would take to launch the MPI processes on Hopper, a Cray XE6 system at NERSC. The IPS coupled simulation is designed to load multiple, different simulation applications, run them for one iteration, save the results, create new input data for each simulation from the previous iteration’s output results and repeat. Because each of the simulation applications has to serialize data, terminate, prepare new data, and re-initialize, considerable overhead is possible. In particular, each iteration requires a system call from the IPS Python code in order to launch MPI processes. Our goal with this experiment was to measure the total execution time for each of the simulations as well as the time that the IPS framework waits for the simulation to run. The difference in time between these two measurements is the amount of overhead incurred.

In our experiment setup, IPS was profiled like in the previously described experiment. In order to measure the overhead of the system call, the “aprun” command was wrapped with a shell script which timed the actual “aprun” call. In addition, we instrumented one of the simulation applications, Toric. By instrumenting Toric, we were able to measure the initialization and termination overhead (all time spent outside of the main computation loop, which is iterated one time) within a sample application, as well as measure the time spent in the application, excluding the “aprun” overhead. In order to account for possible caching effects, we ran the full IPS simulation for 1, 3, 10 and 20 iterations. In order to account for possible caching effects, we ran the full IPS simulation for 1, 3, 10 and 20 iterations.

The “step” routine in the Python file `rf_ic_toric_mcmd.py` is responsible for setting up, launching, waiting, and cleaning up after the Toric simulation. The full steps are:

1. Copy plasma state files to working directory
2. Get input files
3. Copy machine.inp_XXX files
4. Call Toric prepare_input to generate torica.inpp *
5. Call xeqdsk_setup to generate eqdsk.out file *
6. Launch Toric (launch_task, waitpid) *
7. Call process_output *
8. Merge partial plasma state containing updated IC data
9. Archive output files

The steps marked with an asterisk (*) are those that directly make system calls, including step 6 which makes the call to “aprun” to launch the MPI processes. Because all of these steps are necessary for each call to Toric, we measured the overhead using both the full “step” routine and just step 6 of the “step” routine as our outer measurement.

Fig. 12 shows the measured overhead of just the “aprun” call (the “Toric Time (diff)” value). The time is somewhat variable, but seems to have a minimum of 2.724 seconds to launch

MPI on 48 processes. When the time spent in initialization and finalization are removed from the application measurement time, as measured by the “Main Loop” time, the overhead has a minimum value of 4 seconds. The ParaTools team is of the opinion that for higher processor counts, this overhead to launch the MPI processes will increase as an absolute value. If the

Iterations	1	3	10	20
IPS Wait	53.905	55.356	54.019	53.961
Toric Time (mean)	51.181	50.801	50.804	50.755
Toric Time (min)	-	50.567	50.425	50.367
Toric Time (max)	-	51.225	51.724	51.653
Toric Time (stdev)	-	0.368	0.395	0.327
Toric Time (diff)	2.724	4.556	3.215	3.206
Toric Time (%)	5.054%	8.230%	5.952%	5.940%
Main Loop (mean)	49.882	49.911	49.891	49.890
Main Loop (min)	-	49.904	49.848	49.789
Main Loop (max)	-	49.923	49.924	50.125
Main Loop (stdev)	-	0.010	0.025	0.089
Main Loop (diff)	4.024	5.445	4.128	4.071
Main Loop (%)	7.464%	9.8369%	7.642%	7.544%
MPI_Init + MPI_Finalize	1.287	0.8713	0.783	0.807

Fig. 12. "IPS Wait" is defined as the time spent in the "waitpid" system call to aprun. The "Toric Time" is defined as the time spent in the Toric application, as measured by TAU. The "Main Loop" time is defined as the time spent in the computation loop.

overhead penalty is incurred on every iteration, that time will be significant.

Fig. 13 shows the same measurement, but from a higher-level outer timer. The outer timer in Figure 4 is the “step” function, including all initialization and finalization in the IPS Python wrapper, not just the system call to “aprun”. When viewed from this perspective, the overhead increases to at least 5.48 seconds at the application level, and 6.35 seconds at the computation loop level. Again, this overhead penalty is incurred on every iteration, and is expected to increase for larger processor counts. It could potentially increase for larger problem sizes, as larger input files and output files would require more I/O time.

Iterations	1	3	10	20
IPS Wait	57.039	57.734	56.322	56.235
Toric Time (mean)	51.181	50.801	50.804	50.755
Toric Time (min)	-	50.567	50.425	50.367
Toric Time (max)	-	51.225	51.724	51.653
Toric Time (stdev)	-	0.368	0.395	0.327
Toric Time (diff)	5.858	6.933	5.518	5.479
Toric Time (%)	10.270%	12.009%	9.798%	9.744%
Main Loop (mean)	49.882	49.911	49.891	49.890
Main Loop (min)	-	49.904	49.848	49.789
Main Loop (max)	-	49.923	49.924	50.125
Main Loop (stdev)	-	0.010	0.025	0.089
Main Loop (diff)	7.157	7.823	6.431	6.345
Main Loop (%)	12.548%	13.550%	11.419%	11.282%
MPI_Init + MPI_Finalize	1.287	0.871	0.783	0.807

Fig. 13. In this figure, "IPS Wait" is defined as the total time spent in the "step" function, including setting up input files and cleaning up output files.

5.4 Analysis of Toric

ParaTools also collected performance profiles for one of the simulations used in IPS, Toric. Toric was compiled with the TAU auto-instrumentor and performance profiles including hardware counter data were collected from a series of standalone executions of the application on 96 MPI ranks using 12 cores on the NERSC's Hopper system. In addition to time profiles (Fig. 14) hardware counter data collected included level 1 data cache misses, floating point operations and total instructions. Caching behavior was determined to be not immediately relevant to the application's performance. Floating point operations and instructions were used in conjunction with the wallclock time based profile data to determine floating point operations per second (Fig. 15) and total instructions per second within instrumented code blocks.

Preliminary examination of the performance data revealed that the powtwd and ibweld modules have a relatively low FLOPS rate despite comprising a significant portion of execution time, averaged over all cores. This finding suggests that these modules comprise a performance bottleneck, since they are expected to be performing floating point intensive operations. Further analysis of Toric may include in depth examination of the activity of the powtwd and ibweld modules as well as examination of communication patterns to evaluate communication overhead and possible load imbalance issues.

Metric: TIME
Value: Exclusive
Units: seconds

Fig. 14. Mean seconds spent in top 8 Toric routines.

Metric: (PAPI_FP_INS / TIME)
Value: Exclusive
Units: Derived metric shown in seconds format
Sorted By: Exclusive (TIME)

Fig. 15. Mean seconds spent in top 8 Toric routines.