Runtime Analysis Tools for Parallel Scientific Applications

Oleg Korobkin
Center for Computation & Technology, LSU
Baton Rouge, LA, USA
korobkin@cct.lsu.edu

Gabrielle Allen/ Steven R. Brandt
Center for Computation & Technology, LSU
Baton Rouge, LA, USA

Eloisa Bentivegna
Max-Planck-Institute for Gravitational Physics
Potsdam, Germany

Peter Diener/ Jinghua Ge/ Frank Löffler
Center for Computation & Technology, LSU
Baton Rouge, LA, USA

Erik Schnetter
Perimeter Institute for Theoretical Physics
Waterloo, Canada;
Center for Computation & Technology, LSU
Baton Rouge, LA, USA

Jian Tao
Center for Computation & Technology, LSU
Baton Rouge, LA, USA

ABSTRACT

This paper describes the Alpaca runtime tools. These tools leverage the component infrastructure of the Cactus Framework in a novel way to enable runtime steering, monitoring, and interactive control of a simulation. Simulation data can be observed graphically, or by inspecting values of variables. When GPUs are available, images can be generated using volume ray casting on the live data. In response to observed error conditions or automatic triggers, users can pause the simulation to modify or repair data, or change runtime parameters. In this paper we describe the design of our implementation of these features and illustrate their value with three use cases.

Keywords

Software/Program Verification, Frameworks, Runtime Visualization

1. INTRODUCTION

Modern scientific codes can scale to thousands of nodes, and debugging such applications at scale can be challenging even with commercial tools. Licenses may limit node counts, most tools cause code to run more slowly, and fail to understand the high-level relation between data on different nodes or within an adaptive mesh hierarchy. Component-based software engineering frameworks offer a unique opportunity to develop interactive tools that can address this issue by taking advantage of their knowledge of data distribution and synchronization mechanisms used by the framework to steer and analyze a running simulation. Such tools can be then applied to many different scientific applications using a framework, and general techniques can be easily reused across frameworks.

In a future vision of increased standards between frameworks and increased adoption by scientific code developers and users, this methodology would provide new possibilities for high level, integrated debugging and optimization of codes. Here we report on one aspect of the Alpaca project (tools for Application Level Performance And Correctness Analysis) [18, 10] that enables simple and powerful debugging and interaction capabilities within the Cactus framework.

Cactus [12, 1] is a component based software framework that makes it easy to write code that runs in parallel and scales to thousands of cores. The modular and open-source nature of Cactus facilitates collaborative code development between different groups. Cactus originated in the academic research community, where it has been developed and used over many years by a large international collaboration of physicists and computational scientists studying black hole collisions. Its use has now spread to include many other fields, e.g. relativistic astrophysics (see figure 1), computational fluid dynamics, coastal and environmental modeling, cosmology, and quantum gravity.

Cactus is highly portable; applications developed on standard workstations or laptops can be seamlessly run on clusters or supercomputers. Cactus also provides easy access to many cutting edge software technologies developed in the academic research community such as HDF5 parallel file I/O and the PETSc scientific library.

Providing a simplified interface that makes it easy to access advanced features and scale an application's performance to thousands of cores, however, is not the only challenge to productivity for a framework. The cost of development time may be even more critical to the success of a project, and maintaining a reasonable degree of interactivity and simple portable debugging may be crucial.

Interactivity of large-scale simulations is hard to achieve for two main reasons: first, the data is usually distributed over many compute nodes, and reading and writing it from disk or communicating it over the network can be very costly; second, the simulation carefully synchronizes the interaction of many computing nodes, and introducing new steps into

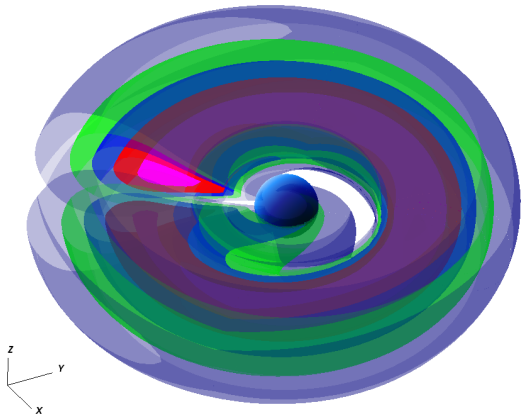

Figure 1: The visualization of fluid density contours and an apparent horizon of a dynamical black hole from one of the Cactus simulations of relativistic accretion disks.

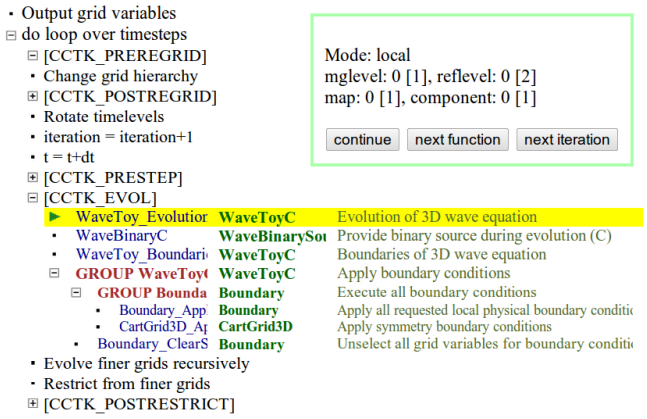

Figure 2: Cactus schedule tree as presented by the Alpaca web server. The top-level nodes are determined by the Cactus driver implementing a particular time-stepping algorithm (e.g. Berger-Oliger adaptive mesh refinement); below this come the schedule groups and schedule items defined by the individual thorns. As this tree can become quite large in production simulations with several hundred leaves, the tree nodes can be expanded and hidden interactively.

the control flow could add significant overhead.

In this work we stress the runtime tools because often it is only at runtime that one can get a good integrated view of the code and the interaction between all of its parts, in particular in a component framework setting where the end user chooses at run time which components are actually enabled. Further, we believe that runtime debugging and analysis that are deeply integrated with the application will be the most effective, and possible the only feasible, mechanism for the future as we move towards ultrascale architectures and increasing complex application codes.

2. THE *CACTUS* FRAMEWORK

The Cactus computational toolkit is divided into the *flesh* (core infrastructure) and *thorns* (components or modules). The flesh parses and manages parameter files, activates thorns, coordinates their execution by constructing a *schedule tree*, and provides a global data structure called a *grid function*. Figure 2 shows part of a schedule tree for an example simulation.

Thorns can also provide higher level infrastructure, allowing the array of services available within the Cactus Framework to be grown in a customizable way. There are *driver thorns* which handle parallelism, memory management, load balancing and I/O. Currently there are two widely used driver thorns available: PUGH for uniform meshes, and Carpet [20, 19, 4] for adaptive mesh refinement (AMR). Other driver thorns for Cactus are available for unstructured grids and alternative adaptive mesh refinement libraries.

With the exception of the driver, thorns are generally stateless. They are passed variables by the flesh and operate on them. This provides a robust model which is easy to test, verify, and parallelize.

The *schedule tree* provides a sequential workflow for the subtasks of a Cactus application. The flesh provides a core set of schedule bins for such functions as startup, initialization, evolution steps, analysis, shutdown, etc. Some bins execute only once (e.g. startup, shutdown), others execute regularly (e.g. evolution). Individual thorns may insert new items into the schedule tree using *schedule groups*. This mechanism enables combining independently developed thorns at run time.

Schedule groups can be scheduled at more than one point in the tree in standard bins or in other groups. Whether individual routines run or not, or whether they run after other routines, can be controlled through parameters. Because of this flexibility, the full tree required to manage a scientific simulation can be quite large and complex. Thorns don't call each other directly, but instead are called when scheduled by the flesh. This makes it possible for our interactive debugging tool (which is itself a set of thorns) to step through time bins in a manner similar to that in which a debugger (e.g. gdb) steps through function calls.

Grid functions are distributed data structures. They are regular arrays of one or more dimensions. Driver thorns manage the partitioning of these arrays to obtain parallelism and also provide mesh refinement. Each time a thorn is invoked to operate on a grid function, it is passed the information about the grid including dimensions and resolution. Data stored on grid functions is synchronized by the driver, usually through a customizable number of ghost zones, but also e.g. via prolongation and restriction operations.

As these data structures are simple and standardized, it is straightforward to make their contents available to our Cactus-aware interaction and debugging tool.

3. RUNTIME APPLICATION INTERFACE

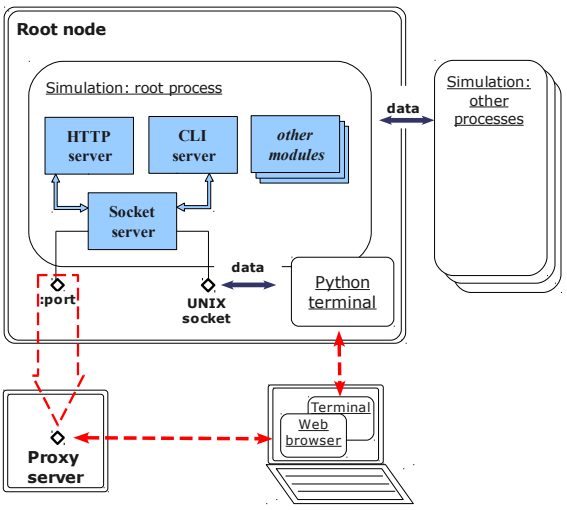

Figure 3: Workflow of the runtime application interface. The simulation contains a socket server that allows outside connections to e.g. a web server or a command line interface (CLI). The simulations' root process also handles communication with other processes that may run on other compute nodes.

The runtime application interface is implemented as a set of Cactus thorns which are themselves components of the framework, easy to include into an existing Cactus application, and able to interact with other Cactus thorns. The runtime interface thorns provide custom APIs which can be used by other thorns to register new interface functions or browser webpages.

The currently implemented runtime interface consists of a secure HTTPS web server and a Command Line Interface (CLI) server, and both are using a unified socket interface for runtime communication. Figure 3 illustrates the workflow between different parts of the runtime interface modules. For a parallel simulation on a cluster, the runtime interface runs only on the root process. If necessary, the data from other threads or processes is transferred to the root process of the simulation using regular MPI communication.

During startup, the socket server creates a network socket for the web server and/or a UNIX domain socket for the CLI server. When the simulation is running, the socket server redirects incoming requests to corresponding request handlers. The same socket server handles the run-time state of the simulation (running, paused, etc.) and provides an internal representation for the commands which control the simulation or retrieve data. This design allows us to combine the capabilities of several user interfaces, in this case both HTTPS and CLI, into a single session. Figure 3 gives a graphical overview of the Alpaca runtime analysis interfaces.

3.1 Secure web server

An easy to use and secure user interface is one of the key components of this infrastructure. For many years, the Cactus framework has included a (non-secure) web server using the HTTP protocol. This version is e.g. used on the Cactus website [1] as a live demonstration of a scalar wave equation simulator. This version is very portable, and is also included in the Einstein Toolkit [2].

We now developed a secure HTTPS server that provides an enhanced user interface in the form of interactive web pages [17, 10]. Below is the list the capabilities of this web server:

- As described in Sec. 2 above, in the Cactus framework the partial ordering of functions introduced by individual thorns is determined at run time on basis of a set of rules defined by the thorns. The web server displays this schedule tree in the form of a nested list with expandable nodes; see Fig. 2. We note that a typical production simulation contains hundreds of scheduled functions, and this view by itself helps users check the correctness of the schedule order.

- The web server allows *pausing* the simulation and *single-stepping* through individual function calls in the schedule. This capability combined with runtime access to the internal data of the simulation is essential for locating the source of errors. We note that the schedule tree as available in Cactus encapsulates high-level information about the structure of the overall code, and a traditional debugger unaware of the software framework could not provide this functionality.

- A *simulation control* page allows to pause the simulation if it receives a warning higher than certain level. The simulation can also be paused by other thorns using an API. This is analogous to a breakpoint in a conventional debugger, but unlike regular debuggers which can only create a breakpoint conditional on rather simple expressions, this API allows expressions at a higher level involving e.g. calling framework APIs or involving parallel operations. For example, a binary black hole simulation can pause whenever a common horizon is detected.

- For simulations which use adaptive mesh refinement (AMR), *Carpet* is required as driver thorn. The grid hierarchy as maintained by Carpet [20, 19, 4] consists of one or more (curvilinear) maps, where each map possibly contains multiple levels of refinement; each level can be split across multiple processors. Carpet implements the Berger-Oliger AMR algorithm, where each coarse level time step is followed by two fine level time steps, repeated recursively over all active levels. In practice, this can result in a highly non-trivial execution pattern.

  The schedule tree also displays the refinement level in the grid hierarchy at which the next scheduled call will be made. The web interface thus allows the user to determine that each scheduled function is called at the right level in the Berger-Oliger algorithm. In combination with access to run-time data, this allows a close monitoring of the changes in the simulation state introduced by each call to a scheduled function.

- In the Cactus framework, each simulation is controlled by a set of parameters that are set by the user via a

so-called parameter file. Parameters can control virtually any aspect of the simulation, ranging from e.g. choosing the output frequency of certain variables to applying a completely different evolution method or using a different set of evolution equations. Certain parameters are *steerable*, which means that they can be modified at run time. The web server allows viewing and setting such simulation parameters. Each thorn defines its own set of parameters and is free to use the `steerable` keyword to make a specific parameter modifiable at runtime.

- The web server provides users with a basic viewport to the distributed state variables of the simulation. The variables associated with different thorns are presented on the web page in the form of a nested expandable list (see Fig. 6 for an example). This list shows the current value for each scalar simulation variable. Grid functions can also be visualized using various image rendering libraries (e.g. gnuplot, jpeg, RuntimeVTK).

3.2 Command Line Interface server

Aside from the web server, we have developed a Command Line Interface (CLI) server that can be launched independently, and which provides a more elementary access to application data. While the web server provides for visual representations of the simulation data, it does not give direct access to these data. Correctness analysis in scientific applications often requires ad-hoc data manipulations. In this sense, a web interface is not flexible enough, while a CLI allows accessing these data from any other application including Python scripts, Matlab, or Mathematica.

To interact with a running simulation, a user needs to log in to the root node of a simulation and connect to a socket in the node's local filesystem that has been set up by the simulation (as illustrated on Fig. 3). In its current version, the CLI server uses local UNIX domain sockets (also known as IPC sockets). While this provides additional security and performance, it has the drawback that collaborators may not be able to log in to the same compute node. We plan to offer (secure) network sockets for this in the future. The CLI server reads data stream from the socket and parses commands which come in rather simple but universal format.

Communication on user side is currently implemented in Python, however any other program which can communicate through an IPC socket (Perl, Lua, Octave/Matlab, Mathematica etc.) could be used to interact with a simulation. We chose Python because it is a high-level object-oriented language, it is available on virtually all HPC systems, and because it has many convenient visualization and analysis modules. A typical session with CLI is presented in Fig. 4.

Below we list the commands and features currently implemented in the CLI server:

- `sim.init()`: Retrieve the data about all thorns, groups, grid functions and parameters and create Python structures which facilitate access to these. The Python structures are organized in a tree, which has three root branches: `thorns`, `vars`, and a `gh` branch (see Fig. 5). The latter contains information about the current grid hierarchy.

- `cctk_*`: Wrappers for the basic functions provided by the Cactus flesh, querying general properties of the simulation. For instance, a command `cctk_NumCompiledThorns` will return the number of all thorns compiled into the current Cactus configuration.

- Cactus parameters in Python conveniently represented as objects with a `value` field implemented as a Python *property*. Whenever a user-defined property of a Python object is accessed, a redefined getter/setter function is called. For example, a command `sim.CarpetIOHDF5.out_dt.value = 1.0` will invoke a method `set_param(...)` with correct set of arguments sending a request to the simulation to change the parameter `out_dt`.

- Cactus grid functions are represented as objects with redefined `getitem/setitem` methods, i.e. the methods which are called whenever an access operator `[]` is used. This simplifies access to the grid functions, so that a user can type e.g. `sim.vars.phi[3,14,15] = -92.65`, where the Python module will automatically assemble request to set the corresponding value.

- A grid hierarchy object `sim.gh` replicates information about the locations and sizes of multiple maps, refinement levels and single processor components of the grids. This information can be updated from the current state in the Carpet driver thorn using the sim.gh.update() command.

- `vtk_*`: the commands which are registered with CLI by the VTK visualization thorn RuntimeVTK (see Sec. 3.3.1 below).

```
$ python
>>> import sim                  # import sim module
>>> sim.init("/tmp/comm_sock") # initialize module
>>> sim.pause()                 # pause simulation
'Paused at CCTK_POSTSTEP in Server::Server_Work'
>>> sim.next()
'Next scheduled call at CCTK_CHECKPOINT is
CarpetIOHDF5::CarpetIOHDF5_EvolutionCheckpoint'
>>> sim.CarpetIOHDF5.out_every.value
10
>>> sim.CarpetIOHDF5.out_criterion.value = "time"
>>> sim.CarpetIOHDF5.out_dt.value = 10.0
>>> sim.vars.phi.vtk_Render2D("wt2.png")
```

Figure 4: An example of runtime communication in Python.

3.3 Visualization

Fast, interactive remote visualization tools are a necessity when examining large datasets, a task that is often associated with code verification. Traditionally, this process entails massive disk read/write operations, which limits the frequency at which the data can be scrutinized; in order to overcome this restriction, in recent years the idea of runtime, on-the-fly visualization of data resident in memory has

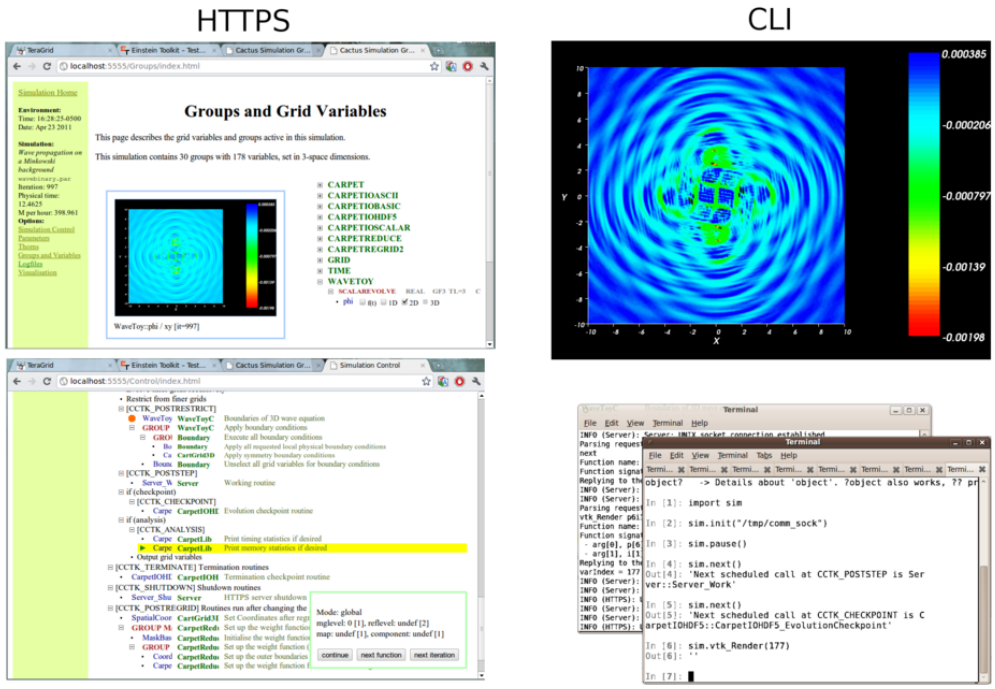

Figure 6: Alpaca runtime analysis interface. An example of 2D VTK plot produced with a command in a Python session, and displayed in a web server.

```
sim
+- thorns
|  +- ADMBase
|  +- Carpet
|  |  +- groups
|  |  |  +- timing       # sample group
|  |  |     +- timing[0] # sample variable
|  |  |     +- timing[1] # another variable
|  |  |
|  |  +- verbose  # sample parameter
|  |     +- type  # parameter type: [BRISK]
|  |     +- range # range of the parameter
|  |     +- desc  # description
|  |     +- value # Python 'property'
|  +- CarpetLib
|  +- LoopControl
|
+- vars  # shortcut bypassing thorns/groups
|  +- ... # grid functions
|
+- gh    # grid hierarchy
   +- ... # information about grid functions
```

Figure 5: Data structures created by the `sim.init()` function containing information about thorns, parameters, and grid functions.

gained momentum, and several visualization packages offer some infrastructure for enabling simulation code to interface with their front-ends and expose their data structures to the visualization methods. For example, the `VisIt` visualization tool [9] offers a `libsim` interface for this task [8].

This idea, however, underestimates the fact that both scientific simulation codes and visualization packages are extremely complex pieces of software, often inspired by different philosophies regarding, for instance, execution control or parallelization.

A natural question is whether it would possible, using the same graphical libraries at the core of many visualization packages, to drive the data visualization from within the simulation code, scheduling the visualization routines at will and doing away with extraneous GUIs.

3.3.1 RuntimeVTK

The Cactus thorn `RuntimeVTK` realizes this concept by implementing a `VTKRender` API (based on structures and functions provided by the VTK library, and available within the framework through an interface thorn `VTK`) which can be called from any point in the simulation code and which provide a visualization of the specified data according to specified directives. This makes a large array of visualization methods that are available in VTK also accessible for use in Cactus simulations. This functionality of the `RuntimeVTK` thorn can also be accessible through the CLI interface. Figure 6 (top right panel) illustrates a 2D slice of a grid function produced at runtime on request from the CLI. The plots produced by RuntimeVTK can be viewed through the web

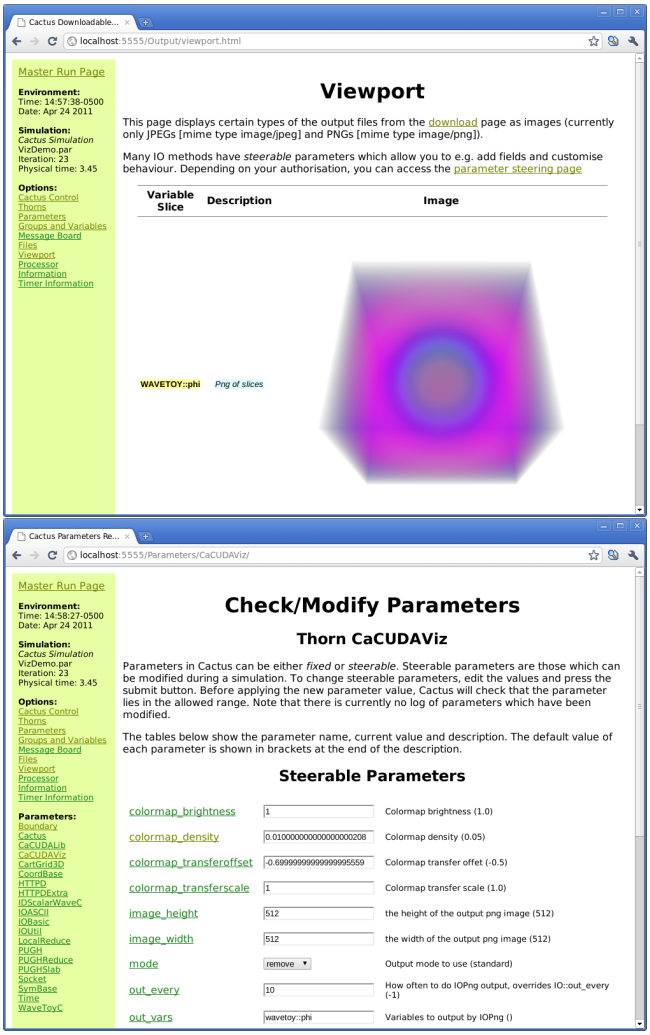

Figure 7: The volume rendering of a scalar wave spreading from the center. The volume rendering was performed using CUDA on a GPU, while rendering parameters are steerable at runtime via the Cactus web server.

server, as shown on the top left panel of Fig. 6.

3.3.2 CaCUDAViz

In addition to RuntimeVTK, a synchronous volume rendering thorn called CaCUDAViz has been developed to leverage the computational power of CPU/GPU hybrid systems. This thorn is implemented with Cactus's built-in callback I/O mechanism. Each I/O method has a unique name, and the flesh provides I/O functions operating on all registered I/O methods. This callback approach is very flexible and makes it possible for scientists to develop their favorite I/O methods and contribute them to the community.

High-quality direct volume rendering [11] with multi-dimensional transfer function [13] can reveal rich features inside the whole volume dataset. Comparing to 2D plots such as slicing, 3D volume rendering can provide an intuitive impression of the overall structure of the datasets, which is crucial to user visual analysis, and also for comparison to observed image data.

The 3D volume rendering is implemented using NVIDIA GPGPU programming CUDA API [5]. The use of CUDA takes advantage of the general parallel computing architecture provided by modern GPUs. Hardware-accelerated volume ray casting [15] implemented in OpenGL [6] has been used widely in direct volume rendering, usually with performance two magnitudes better then software-based ray casting. The same algorithm can be implemented in CUDA in a straightforward manner. Another advantage of using CUDA is that CUDA is designed to run on GPUs without the need of an X Server, which is not supported by default in some Teragrid GPU platforms such as NCSA Lincoln. We show an example of such a visualization in Figure 7.

The image composition stage of parallel volume rendering requires data communication [16]. An existing composition library called Ice-T [3] (Image Composition Engine for Tiles) is used by some visualization softwares such as VisIt [9] and Paraview [7]. The strategy of the CaCUDAViz thorn is to explore a composition algorithm which can achieve better efficiency on hybrid computing platforms. Currently, image composition of the CaCUDAViz thorn is under development, based on the scheduled-linear [21] algorithm.

4. USE CASES

Below we give examples where the Alpaca tools are used in real-world application scenarios to detect and correct high-level problems in simulation codes or in simulations. In these cases, the Alpaca tools greatly simplify these tasks both by an improved user interface and by making high-level information about the simulation directly accessible to the user.

4.1 Case 1: Instabilities in multi-block MHD simulations

Some of the authors of this paper participate in an international collaboration that is currently developing a new three-dimensional general relativistic magneto-hydrodynamics (MHD) code on arbitrary curvilinear multi-block grids, called *Thor* [22, 14]. This code is designed to address a wide class of astrophysical problems that involve an ideally conducting plasma with magnetic fields. The simulation domain is represented as a set of curvilinear overlapping blocks, with interpolation boundary conditions on the block interfaces. This code is based on the Cactus framework and the Carpet driver.

It goes without saying that designing, implementing, and testing such a code requires expertise from many different fields of science, and is a highly non-trivial task involving a significant amount of time from a large group of people. Perhaps not surprisingly, even after an initial prototype of a complex code exists, there remains a large amount of debugging work involving testing the code in various scenarios, comparing it to analytically known solutions or against results from other codes. We note that often, unfortunately, development and testing of a 3D core already requires a parallel HPC system. Below we describe an example where one of us used the Alpaca tools to help our development process.

One of our first code tests was to reproduce a constant and uniform magnetic field in a uniform-density plasma on a curvilinear multiblock system. Since the coordinates are curvilinear, this is a non-trivial problem, as the field representation on the computational grid is neither constant in magnitude nor in direction.

In this particular case, we observed an instability of unknown origin that would begin to develop right from the start of the time evolution, resulting in an unbounded growth of the magnetic field – clearly an error. This instability only appeared in those cases where curvilinear multiblock systems were used. We could also identify this growth in the right-hand sides of the densitized magnetic field variables, which should remain zero in the ideal case.

To investigate this issue, we started a test simulation (in parallel) and activated the Alpaca web server. Single-stepping through the Cactus schedule tree and examining temporary grid functions, we were able to spot the individual step which introduced the large errors in the right-hand sides. In this particular case, they appeared after the external boundary conditions were applied. Using the parameter steering ability, we switched to a different kind of external boundary condition and monitored the right-hand sides as they were computed at the next iteration. Parameter steering allowed us to continue with the already running simulation, without needing to restart the simulation to try a new boundary condition.

Because this switch of boundary conditions appeared to correct the problem, we investigated the boundary condition routine further. By visually examining the wrong result, we came to the hypothesis that the problem was caused by an extrapolation using the wrong coordinate system – using a block-local coordinate system instead of the global coordinate system. At this point, we discussed the problem and our analysis on our mailing list.

In this case, the Alpaca tools allowed us to quickly narrow down the issue interactively, without having to restart the simulation multiple times, and without having to output many intermediate variables to disk (aka "printf debugging", although we actually use HDF5 as file format). This saves people time in addition to supercomputer allocation time. Without such interactive tools, a researcher needs to submit and run a new simulation each time a new hypothesis needs to be tested.

4.2 Case 2: Occasional *NaNs* during large-scale GR hydrodynamics simulations

In this second use case, we look at the class of failures which happen rarely and which are difficult to reproduce. General relativistic (GR) hydrodynamics codes typically employ two different sets of variables, *conserved variables* (mass density, momentum, total energy) and *primitive variables* (another definition of density, velocity, and internal energy). During evolution, one needs to convert the conserved to the primitive variables, in Cactus terms this process is called `con2prim` which requires a multidimensional root-finding for every grid point, and which causes a substantial amount of headache among developers and users. In most cases, failures in a simulation manifest themselves as a `con2prim` failure. In many cases, a `con2prim` failure is actually harmless, and one needs to recover gracefully. Each simulation code contains a large, implicit amount of expertise for how to deal with such failures.

In our code, `con2prim` failures lead to NaNs (Not a Number, the IEEE representation for results of "impossible" arithmetic operations such as square roots of negative numbers). `con2prim` failures can be inconsequential e.g. if they occur in vacuum, where they will be overwritten after the time step, or they can also occur inside a black hole, where matter cannot be observed from the outside and result is not important. However, if a `con2prim` failure is not inconsequential, then NaNs very quickly propagate away from an initial single point of failure, and the simulation has to be terminated. Cactus contains a tool, called NaNChecker, which monitors the simulation, and either issues a warning or terminates the simulation whenever NaN is encountered.

The Alpaca tools provide the facility to automatically pause the simulation if such a warning is issued. This allows the user to inspect the current state of grid functions, including temporary grid functions that are not normally output. If the user decides that the NaNs are not critical to the physical result, they can be reset to a finite value either manually or using a special cleanup function. After this, the simulation can continue.

In this case, the Alpaca tools do not treat the root of the issue, but instead allows one to treat the symptoms if the condition is not critical. While this is obviously not appropriate for production simulations, it can be most useful during development runs.

4.3 Case 3: Correcting simulation parameter errors

Sometimes it so happens that a large production simulation is submitted to a queuing system, and while the simulation is running, an error in its setup is detected. There is a large class of setup errors that do not affect the physical results of the simulation, but which nevertheless can make the simulation result unusable. It is in this case annoying to have to stop and restart the simulation, particularly if the simulation has been waiting in a queuing system for many days, or if the simulation has already used a significant amount of CPU time. One can argue that being careful can avoid such situations, but one can also argue that simulations are designed by humans, and being able to correct certain errors after the fact makes a system easier to use. Examples of such errors include choosing too frequent output, slowing the simulation down, or outputting too many / not enough variables, or choosing a wrong checkpointing interval.

The Alpaca tools allow users to modify parameters at run time, using either the graphical web interface or the Python-based command line interface. Additionally, there are cases where the simulation itself notices a problem in its setup and can then notify the user to correct the settings, optionally pausing until the user has done so. We note that such manual babysitting of simulations may not be appropriate in many cases, but it can be very useful either for development runs, or for "hero runs" that require manual supervision anyway.

5. CONCLUSION AND FUTURE WORK

While the capabilities of these tools are already significant, there is still room for development. The ability to repair data and steer simulations can help debug and produce better results are important, but it makes a simulation potentially more difficult to reproduce. Adding a logging and replay capability to the interactive tools will make problems identified clearer and the results of simulations that had undergone steering more relevant and useful.

We have shown the potential advantages for building a system for interactive steering and/or debugging into the infrastructure of the Cactus component based software framework. The static workflow that is built-in to Cactus provides a natural way to step through the execution of an application, and its simple distributed data structures make it possible to visualize and monitor variables on the fly.

Because these tools build on the infrastructure of the framework, they require little in the way of software support or other services, only a way to independently connect to the node running the simulation. These tools are thus ultimately portable in a way that regular debugging solutions are not, running on all platforms where a simulation can run, and have access to capabilities that are intrinsic to the framework itself.

6. ACKNOWLEDGMENTS

We thank our colleague Andrei Hutanu for his input and many helpful discusssions regarding remote, interactive visualization. We also thank our collaborators in the Cactus, the numerical relativity, and the CFD groups at the CCT.

This work is supported by NSF awards 0721915 *Alpaca*, 0725070 *Blue Waters*, 0904015 *CIGR*, 0905046 and 0941653 *PetaCactus*, and 0947825 *EAVIV*. Computing time is provided by LONI allocation "cactus" and by TeraGrid allocation TG-MCA02N014.

G. Allen acknowledges that this material is based upon work supported while serving at the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] Cactus computational framework. `http://www.cactuscode.org/`.

[2] Einstein Toolkit for numerical relativity. `http://einsteintoolkit.org/`.

[3] The image composition engine for tiles (icet). `http://www.cs.unm.edu/~kmorel/IceT`.

[4] Mesh refinement with Carpet. `http://www.carpetcode.org/`.

[5] Nvidia cuda library documentation, 3.2 beta. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/online/index.html`.

[6] Opengl®, the industry's foundation for high performance graphics. `http://www.opengl.org`.

[7] Paraview: Open source scientific visualization. `http://www.paraview.org/`.

[8] VisIt simulation control interface. `http://www.visitusers.org/index.php?title=Simulation_Control_Interface`.

[9] The VisIt visualization tool. `https://wci.llnl.gov/codes/visit/`.

[10] E. Bentivegna, G. Allen, O. Korobkin, and E. Schnetter. Ensuring correctness at the application level: A software framework approach. In *Proceedings of the 2009 Workshop on Component Based High Performance Computing*, 2009.

[11] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM New York, NY, USA, 2001.

[12] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer.

[13] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[14] O. Korobkin, E. B. Abdikamalov, E. Schnetter, N. Stergioulas, and B. Zink. Stability of general-relativistic accretion disks. *Phys. Rev. D*, 83:043007, 2011.

[15] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. *IEEE Visualization, 2003. VIS 2003*, pages 287–292, 2003.

[16] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, 1994.

[17] E. Schnetter, G. Allen, T. Goodale, and M. Tyagi. Alpaca: Cactus tools for application level performance and correctness analyais. Technical Report CCT-TR-2008-2, Louisiana State University, 2008.

[18] E. Schnetter, G. Allen, T. Goodale, and M. Tyagi. Alpaca: Cactus tools for application level performance and correctness analysis. Technical Report CCT-TR-2008-2, Louisiana State University, 2008.

[19] E. Schnetter, P. Diener, N. Dorband, and M. Tiglio. A multi-block infrastructure for three-dimensional time-dependent numerical relativity. *Class. Quantum Grav.*, 23:S553–S578, 2006.

[20] E. Schnetter, S. H. Hawley, and I. Hawke. Evolutions in 3D numerical relativity using fixed mesh refinement. *Class. Quantum Grav.*, 21(6):1465–1488, 21 March 2004.

[21] A. Stompel, K. Ma, E. Lum, J. Ahrens, and J. Patchett. SLIC: scheduled linear image compositing for parallel volume rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 6. IEEE Computer Society, 2003.

[22] B. Zink, E. Schnetter, and M. Tiglio. Multi-patch methods in general relativistic astrophysics – I. Hydrodynamical flows on fixed backgrounds. *Phys. Rev. D*, 77:103015, 2008.