The Cactus Framework and Toolkit: Design and Applications

Tom Goodale[1], Gabrielle Allen[1], Gerd Lanfermann[1], Joan Massó[2], Thomas Radke[1], Edward Seidel[1], and John Shalf[3]

[1] Max-Planck-Institut für Gravitationsphysik, Albert-Einstein-Institut,
Am Mühlenberg 1, 14476 Golm, Germany
{goodale, allen, lanfer, tradke, eseidel}@aei.mpg.de
[2] Departament de Física, Universitat de les Illes Balears
E-07071 Palma de Mallorca, Spain
jmasso@gridsystems.com
[3] Lawrence Berkeley National Laboratory,
Berkeley, CA
jshalf@lbl.gov

Abstract. We describe Cactus, a framework for building a variety of computing applications in science and engineering, including astrophysics, relativity and chemical engineering. We first motivate by example the need for such frameworks to support multi-platform, high performance applications across diverse communities. We then describe the design of the latest release of Cactus (Version 4.0) a complete rewrite of earlier versions, which enables highly modular, multi-language, parallel applications to be developed by single researchers and large collaborations alike. Making extensive use of abstractions, we detail how we are able to provide the latest advances in computational science, such as interchangeable parallel data distribution and high performance IO layers, while hiding most details of the underlying computational libraries from the application developer. We survey how Cactus 4.0 is being used by various application communities, and describe how it will also enable these applications to run on the computational *Grids* of the near future.

1 Application Frameworks in Scientific Computing

Virtually all areas of science and engineering, as well as an increasing number of other fields, are turning to computational science to provide crucial tools to further their disciplines. The increasing power of computers offers unprecedented ability to solve complex equations, simulate natural and man-made complex processes, and visualise data, as well as providing novel possibilities such as new forms of art and entertainment. As computational power advances rapidly, computational tools, libraries, and computing paradigms themselves also advance. In such an environment, even experienced computational scientists and engineers can easily find themselves falling behind the pace of change, while they redesign and rework their codes to support the next computer architecture. This

rapid pace of change makes the introduction of computational tools even more difficult in non-traditional areas, e.g., social arts and sciences, where they may potentially have the most dramatic impact.

On top of this rapidly changing background of computation, research and engineering communities find themselves more and more dependent on each other as they struggle to pull together expertise from diverse disciplines to solve problems of increasing complexity, e.g., simulating a supernova or the Earth's climate. This creates a newfound need for *collaborative* computational science, where different groups and communities are able to co-develop code and tools, share or interchange modules and expertise, with the confidence that everything will work. As these multidisciplinary communities develop ever more complex applications, with potentially hundreds or thousands of modules and parameters, the potential for introducing errors, or for incorrectly using modules or setting parameters, also increases. Hence a software architecture that supports a wide range of automated consistency checking and multi-architecture verification becomes essential infrastructure to support these efforts.

Application developers and users also need to exploit the latest computational technologies without being exposed to the raw details of their implementation. This is not only because of the time needed to learn new technologies, or because their primary interest is in what the application itself can do, and not how it does it, but also because these technologies change rapidly; e.g. time invested in learning one technology for message passing is largely wasted when a developer has to learn the next. What the developer really needs is an *abstracted* view of the operations needed for data distribution, message passing, parallel IO, scheduling, or operations on the Grid [1] (see section 16.5), and not the particular implementation, e.g., a particular flavour of MPI, PVM, or some future replacement. The abstract concept of passing data between processors does not change over time, even though a specific library to do it will. A properly designed *application framework* with suitable abstractions could allow new or alternate technologies, providing similar operations, to be swapped in under the application itself. There is a great need for developing *future-proof* applications, that will run transparently on today's laptops as well as tomorrow's Grid.

At the same time, many application developers, while recognising the need for such frameworks, are reluctant to use "black box", sealed packages over which they have little control, and into which they cannot peer. Not only do some application developers actually want to see (perhaps some) details of different layers of the framework, in some cases they would like to be able to extend them to add functionality needed for their particular application. Such transparency is nearly impossible to provide through closed or proprietary frameworks without resorting to excessively complex plug-in SDKs. For this reason, freely available, *open source* tools are preferred by many communities (see, for example, Linux!). Source code is available for all to see, to improve, and to extend; these improvements propagate throughout the communities that share open source tools.

The idea of open source tools, once accepted by one segment of a community, tends to be contagious. Seeing the benefits, application developers making use

of such open tools are often inclined to make their own modules freely available as well, not only in the computational science areas, but also for modules specific to research disciplines. In the computational science disciplines, modular components that carry out specific tasks are increasingly able to inter-operate. Developing application components for, or in, one framework makes it easier to use them with another framework. This is a particular goal of the "Common Component Architecture" [2], for example. As another example, we describe below application modules developed for numerical relativity and astrophysics; a growing number of scientists in this domain make use of the Cactus framework [3], and many of them make their application specific modules available to their community. Not only does this encourage others to share their codes, it raises the overall quality; knowing modules will become public increases the robustness of the code, the accompanying documentation, and the modularity.

For all of these reasons, and more that we touch on below, open, modular, application frameworks that can both hide computational details, and enable communities to work together to develop portable applications needed to solve the complex problems in their disciplines, are becoming more and more important in modern computational science. This will become particularly true as the Grid, described below, becomes a reality [1]. "Grid-enabling frameworks" will be crucial if application scientists and engineers are to make use of new computational paradigms promised by the Grid [4].

2 Cactus as an Application Framework

We have motivated various general reasons why application frameworks are needed, and in the rest of this paper we describe a specific example. The programming framework *Cactus* [5,6], developed and used over a number of years, was designed and written specifically to enable scientists and engineers to perform the large scale simulations needed for their science. From the outset, Cactus has followed two fundamental tenets: respecting user needs and embracing new technologies. The framework and its associated modules must be driven from the beginning by user requirements. This has been achieved by developing, supporting, and listening to a large user base. Among these needs are ease of use, portability, the abilities to support large and geographically diverse collaborations, and to handle enormous computing resources, visualisation, file IO, and data management. It must also support the inclusion of legacy code, as well as a range of programming languages. It is essential that any living framework be able to incorporate new and developing, cutting edge computation technologies and infrastructure, with minimal or no disruption to its user base. Cactus is now associated with many computational science research projects, particularly in visualisation, data management, and the emerging field of Grid computing [7].

These varied needs for frameworks of some kind have long been recognised, and about a decade ago the US NSF created a funding program for "Grand Challenges" in computational science. The idea was to bring computer and computational scientists together with scientists in specific disciplines to create the

software needed to exploit large scale computing facilities to solve complex problems. This turned out to be a rather difficult task; in order to appreciate the evolution of attempts to tackle this problem, and the maturity of current solutions, it is instructive to review a bit of history before plunging directly into the present design of Cactus 4.0.

Like many of these NSF Grand Challenge projects, a particular project called the "Black Hole Grand Challenge Alliance", aimed to develop a "Supertoolkit" for the numerical relativity community to solve Einstein's equations for the study collisions of black holes. The thought was that, as these equations are so complex, with equations of many types (e.g., hyperbolic, elliptic, and even parabolic), a properly designed framework developed to solve them would also be a powerful tool in many fields of science and engineering. (This has turned out to be true!) A centrepiece of this toolkit was, in it simplest description, a parallelisation software known as DAGH [8] (which has now evolved into a package called GrACE [9]), that would allow the physicist to write serial Fortran code that could be easily parallelised by using the DAGH library.

At the same time, a series of independent parallel codes were developed to solve Einstein's equations, with the expectation that they would later make use of DAGH for parallelism. However, some of these codes, notably the "G-code" [10], took on a life of their own, and became mini-frameworks (in Fortran!) themselves, evolving into workhorses for different collaborations within the project. As the codes and collaborations grew, it became clear that the original designs of these codes needed major revision. A prototype framework, naturally called "The Framework" [11], was designed and developed by Paul Walker at the Max Planck Institute for Gravitational Physics (AEI), with the idea that different physics codes could be plugged in, like the G-code and others, and that it would be an interface to DAGH to provide parallelism.

Learning from this experiment, Paul Walker and Joan Masso began to develop Cactus in January 1997. Cactus 1.0 was a re-working of the Framework for uni-grid applications using a simple parallel uni-grid driver as a temporary replacement for DAGH. In Cactus there were two types of object, the "flesh", a core component providing parameter parsing, parallelisation, and IO, and "thorns" which are optional modules compiled in. In October 1997 Cactus 2.0 was released, providing for the first time the ability to dynamically pick which routines were run at run-time, and some ability to order them, this ability removed the necessity of modifying the flesh to support new thorns. In these early versions of Cactus all thorns received all variables defined for a simulation; Cactus 3.0 removed this necessity, further enhancing the modularity.

Cactus became the major workhorse for a collaboration of numerical relativists spanning several large projects in Europe and the US. This community requires very large scale, parallel computational power, and makes use of computational resources of all types, from laptops for development to heavy-duty production simulations on virtually every supercomputing architecture, including Linux clusters, "conventional" supercomputers (e.g., SGI Origins or IBM

SP systems), and Japanese vector parallel machines. Hence complete portability and very efficient parallel computation and IO are crucial for its users.

Lessons learnt through its heavy use in this community led to the development of a still more modular and refined version of Cactus, Cactus 4.0, now the flesh contains almost no functionality; anything can be implemented as a thorn, not just physics routines, but all computational science layers. These computational science thorns implement a certain function, say message passing, hydrodynamics evolution, or parallel IO, and are interchangeable; any thorn that implementing a given function can be interchanged with any other one. For example, different "driver layer" thorns (see section 10) that provide message passing, can be implemented and interchanged, and can be written using any message passing library; properly written applications that make use of Cactus can use any of them without modification. Entire software packages, such as GrACE for parallelism, PETSc [12] for linear solvers, and others, can be made available for Cactus users through thorns that are designed to connect them. Hence, Cactus should be thought of as a Framework for conveniently connecting modules or packages with different functionality, that may have been developed completely outside of Cactus. In principle, Cactus can accommodate virtually any packages developed for many purposes, and it is very powerful. Moreover, while earlier versions of Cactus were limited in the number of architectures they supported (T3E, IRIX, OSF, Linux, NT), the community's need for portability lead to a complete redesign of the make system to enhance portability and reduce compilation times, and now Cactus compiles on practically any architecture used for scientific computing, and the make system, which is described in section 6 is designed to make porting to new architectures as easy as possible. These points and others will be developed in depth in the sections below.

Cactus 4.0 has been developed as a very general programming framework, written to support virtually any discipline. It has been developed through a long history of increasingly modular and powerful frameworks, always aimed at supporting existing communities. Hence it is by now a very mature, powerful, and most importantly, widely *used* framework; its design has been completely derived from the needs of its users, through both careful planning and much trial, error, and redesign. The first official release of the Cactus Computational Toolkit [13], which builds on the Cactus flesh to provide standard computational infrastructure is released along with Cactus 4.0, as well as application specific toolkits. The success of the Cactus approach is borne out by the wide use of Cactus not only by the numerical relativity community in five continents, but also its emerging adoption in other fields such as climate modelling, chemical engineering, fusion modelling, and astrophysics. In the next sections, we detail the design and capabilities of Cactus 4.0.

Note that Cactus is not an application in itself, but a development environment in which an application can be developed and run. A frequent misinterpretation is that one "runs Cactus", this is broadly equivalent to stating that one "runs perl"; it is more correct to say that an application is run within the Cactus framework.

3 Design Criteria for Cactus 4.0

In reviewing previous versions of Cactus and other codes while designing the new Cactus framework, it became clear that the new framework had to meet some wide ranging design criteria.

It must be able to run on a wide range of different machines, so requires a flexible and modular make system. Moreover people must be able to use the same source tree to build executables for different architectures or with different options on the same architecture without having to make copies of the source tree. The make system must be able to detect the specific features of the machine and compilers in use, but must also allow people to add configuration information that cannot otherwise be auto-detected in a transparent and concise way – without having to learn new programming languages.

Similarly it should be easy to add new modules, and as much functionality as possible should be delegated to modules rather than the core. Data associated with individual modules should exist in separate name spaces to allow independently developed modules to co-exist without conflict. Functionally equivalent modules should be inter-changeable without other modules which use or depend on them noticing the exchange; many such modules should be able to be compiled into the executable at once and the desired one picked at program startup.

A core requirement is to provide transparent support for parallelism. The framework should provide abstractions for distributed arrays, data-parallelism, domain decomposition and synchronisation primitives. Furthermore, the primitives used to manage the parallelism must be minimal and posed at a high enough level of abstraction so as not to be tedious to use or to expose system architecture or implementation dependencies. The framework must be able to trivially build applications that work in uniprocessor, SMP, or MPP environments without code modification or placement of explicit conditional statements in their code to account for each of these execution environments. If the programmer wishes to build a serial implementation on a multiprocessor platform for testing purposes, the framework must support that build-time choice.

Another important class of modules is those responsible for input and output. There are many different file formats in use, and the framework itself should not dictate which is used. Nor should other modules be unnecessarily tied to any particular IO module – as new IO modules are developed people should be able to access their functionality without modification to existing code.

In past versions of Cactus it was possible to set input parameters to values which were meaningless or, in the case of keyword parameters, to values not recognised by any module. This is obviously undesirable and should be picked up when the parameter file is read.

There are many frameworks (for a review, see [14]), however most of them require a new code to be written to fit within the framework, and often restrict the languages which can be used. One design criteria was to be able to support legacy codes, and, in particular, to support the large number of codes and programmers who use FORTRAN 77.

Another criterion was to be able to allow applications using the framework to utilise features such as parallelism, adaptive mesh refinement (AMR) and out of core computation without having to modify their code, as long as their code is structured appropriately.

All the above requirements can be summed up in saying that Cactus must be portable and modular, should abstract the interfaces to lower-level infrastructure as much as possible, but be easy to use and capable of supporting legacy codes. By themselves these requirements would lead to something with cleaner interfaces than older versions of Cactus, however we also chose to design the framework to be as flexible and future proof as possible. The framework should allow the modules to provide access to new technologies, such as those to provide collaborative working, or facilities emerging from the Grid community, while at the same time the framework should not come to depend upon such technologies. This list is by no means a complete list of requirements, however these are the more important ones coming out of our experiences with earlier versions of cactus and our desire for the framework to be as future-proof as possible.

4 Structure

The code is structured as a core – the 'flesh' – and modules which are referred to as 'thorns'.

The flesh is independent of all thorns and provides the main program, which parses the parameters and activates the appropriate thorns, passing control to thorns as required. It contains utility routines which may be used by thorns to determine information about variables, which thorns are compiled in or active, or perform non-thorn-specific tasks. By itself the flesh does very little apart from move memory around, to do any computational task the user must compile in thorns and activate them at run-time.

Thorns are organised into logical units referred to as 'arrangements'. Once the code is built these have no further meaning – they are used to group thorns into collections on disk according to function or developer or source.

A thorn is the basic working module within Cactus. All user-supplied code goes into thorns, which are, by and large, independent of each other. Thorns communicate with each other via calls to the flesh API, plus, more rarely, custom APIs of other thorns.

The connection from a thorn to the flesh or to other thorns is specified in configuration files which are parsed at compile time and used to generate glue code which encapsulates the external appearance of a thorn.

When the code is built a separate build tree, referred to as a 'configuration' is created for each distinct combination of architecture and configuration options, e.g. optimisation or debugging flags, compilation with MPI and selection of MPI implementation, etc. Associated with each configuration is a list of thorns which are actually to be compiled into the resulting executable, this is referred to as a 'thornlist'.

At run time the executable reads a parameter file which details which thorns are to be active, rather than merely compiled in, and specifies values for the control parameters for these thorns. Non-active thorns have no effect on the code execution. The main program flow is shown in figure 1.

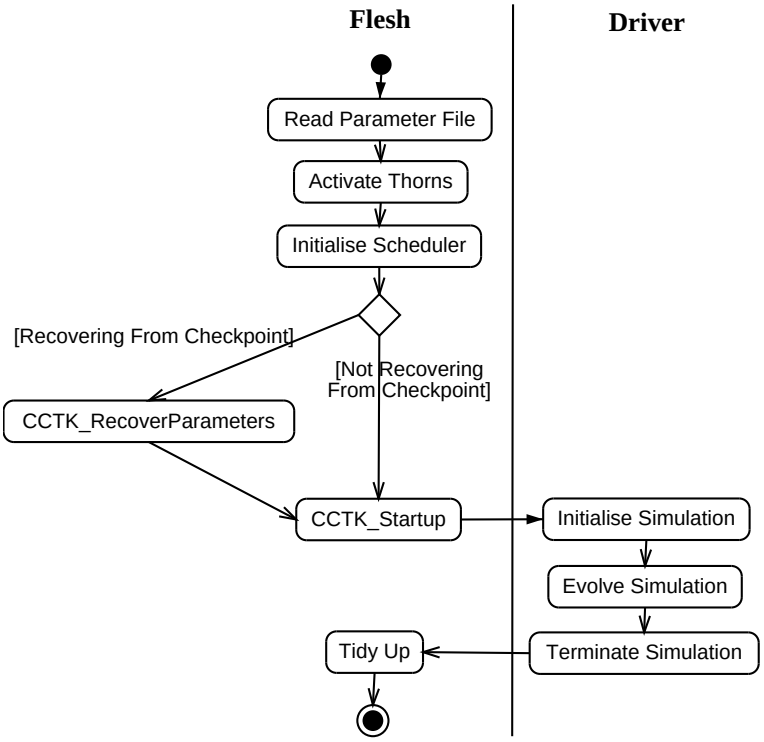

Fig. 1. The main flow of control in the Cactus Framework. The flesh initialises the code, then hands control to the driver thorn (see section 10). The actions in the driver swimlane are detailed in figures 2, 3, and 4.

5 Code Management and Distribution

Since the code is developed and maintained by a geographically dispersed team of people, a well-defined code management system is necessary. The Cactus development follows clearly defined coding guidelines and procedures which are set out in the Cactus Maintainers' Guide [15].

A central component of these procedures is the use of CVS [16] to manage revisions in the flesh and the thorns. The tree-like file layout structure is purposely designed to enable each thorn or arrangement to belong to entirely independent

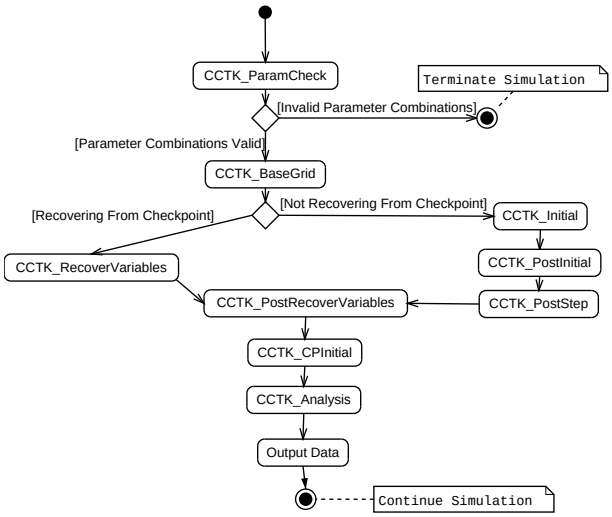

Fig. 2. Initialisation action diagram (corresponding to the *initialise simulation* action in figure 1). All activities prefixed with "CCTK_" are schedule bins (see section 9)

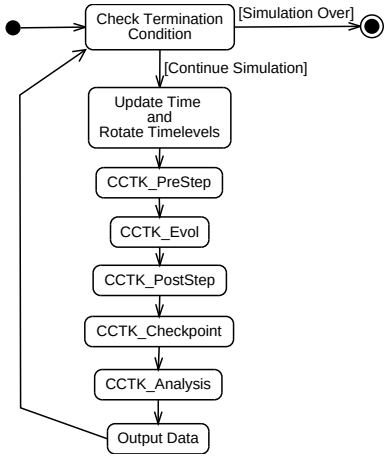

Fig. 3. Evolution action diagram (corresponding to the *evolve simulation* action in figure 1). All activities prefixed with "CCTK_" are schedule bins (see section 9)

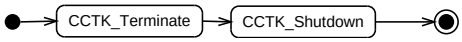

Fig. 4. Termination action diagram (corresponding to the *terminate simulation* action in figure 1). All activities prefixed with "CCTK_" are schedule bins (see section 9)

CVS repositories with their own access control, branching policies. The Cactus flesh provides simplified interfaces to checked-out and assembled together these thorns into a working application.

The primary code distribution method is via CVS, for which the several wrapper scripts are provided to automate downloading the flesh and user-defined lists of thorns. Thorns are not only maintained and distributed at the central Cactus repository, but by different authors who want to make their thorns available to others through their own CVS repositories. The Cactus download scripts can take a thornlist with some embedded comments and retrieve thorns from all the appropriate CVS repositories. Additionally compressed tar files of the flesh and thorns can be downloaded from the Cactus web-site.

Documentation, including an FAQ, Users and Maintainers Guide, as well as Thorn Guides for different collections of thorns, are distributed with the Cactus flesh and thorns, as well as from the Cactus Home Page [6,15,13].

6 A Portable and Modular Make System

The Cactus make system is a two stage process. First of all a "configuration" is created, encapsulating machine-specific information and also user defined choices; the user may define many configurations, each for specific purposes, e.g. for use on different architectures, to use particular parallel libraries, to include debugging information, to be compiled with specific compilers, etc.

Once a configuration has been created, a thornlist is created for the configuration, and the flesh and thorns for this configuration are built. If the thornlist or source files for a thorn are modified, the appropriate files for this configuration and its executable may be rebuilt.

GNU Make is used to avoid incompatibilities in the make program across architectures. This is freely available on a large number of platforms.

The Cactus make system has shown itself to work on a large number of Unix platforms, including SGI IRIX, IBM AIX (e.g. SP systems), Linux (IA32, IA64, Alpha, PowerPC, IPAQ, Playstation), Apple OSX, Compaq OSF, Cray Unicos (T3E, SV1, etc), Sun Solaris, Hitachi SR800, Fujitsu VPP, OpenBSD, and NEC SX-5; in principle Cactus should compile on any UNIX-like platform with at most minimal effort. To date the only non-Unix platform used has been Windows, where the Cygwin [17] environment is used to provide a Unix-like environment; the resulting code is, however, independent of Cygwin if non-Cygwin compilers are used.

Note that unlike a library or system utility, Cactus is not "installed". Currently each user has a copy of the sources and generates their own configurations based upon their individual needs and preferences.

6.1 Creating a Configuration

In order to support as wide a range of computers as possible, GNU Autoconf [18] is used to detect specific features of compilers and libraries. However not all such things can be easily detected, and it was deemed important that in those cases a user compiling on a previously unsupported architecture should not need to learn detailed shell programming and M4 to extend the detection system. To this end, architecture specific information, such as flags necessary for full optimisation, or debugging, or warnings, or which libraries need to be linked against for the Fortran compiler, is stored in 'known-architecture' files, which are simple shell scripts which are sourced by the master configuration script once it has detected the operating system. These files are simple in format, and new architectures can be supported merely by placing a new file in the directory holding these files, with a name corresponding to the operating system name returned by the 'config.sub' script invoked by Autoconf.

A similar system is used to configure for optional packages such as MPI, pthreads or HDF5. To add a new optional package with its own configuration options, a user may just create a directory in a specific place, and add a script file in this place which may add extra definitions to the master include file or add extra information to the make system. This is heavily used to support the various different flavours of available MPI – native MPI on SGI, Cray and IBM AIX machines, or MPICH, LAM, HPVM, MPIPro on these or other systems.

When creating a configuration the user may pass instructions to the configuration scripts to override detection or defaults. For example the user may choose to use a specific C or Fortran compiler, or choose to configure the code for parallel operation by configuring with MPI, or specific the user may want this configuration have different debugging or optimisation flags. The user may also pass a specific architecture name to the configuration system in order to cross-compile for a different architecture; in this case the known-architecture file for the target platform must contain certain details which the configure script would otherwise detect.

6.2 Building a Configuration

Once a configuration has been created the user must pick a set of thorns to build. If the user doesn't provide one, the make system will search for thorns and generate a thornlist and give the user the option to edit it. The thornlist is a plain text file and may be edited with any text editor.

Once there is a thornlist associated with a configuration, the CST, which is described in detail in section 8.2, is run to validate the thorns and provide bindings between the flesh and the thorns, and then the code is built based upon

the flesh source and the sources of all thorns in the thornlist. The flesh and each thorn are compiled into libraries which are linked to form the final executable.

In individual thorns the thorn writer has the option of using their own Makefile or of using a simplified system whereby they list the files and subdirectories which should be compiled, and a standard make file is used for that thorn. The vast majority of thorn writers use the latter approach, which enables them to develop with minimal knowledge of make file syntax or operation. More advanced users may also similarly write make rules specifying dependencies between source files in their thorn.

Since the object files for each configuration are kept in different directories from the original source files, and a user may be building several configurations simultaneously from one source tree, care has to be taken with the working directory of the compilation, especially when Fortran 90 modules or C++ templates are used. This is currently solved by setting the working directory of any compilation as a specific scratch directory in the configuration-specific build area.

Currently each thorn's object files are packaged into libraries which are then linked together to form the final executable. In future versions of Cactus it is planned that these libraries may be dynamically loadable, on platforms which allow this, thus allowing new thorns providing new capabilities, in particular data analysis, to be loaded at run-time in long running simulations.

At build time the user may pass options to the make system to echo all commands to the screen or to switch on compiler warnings. However no option which would change the final executable is allowed at this stage, on the principle that the files stored with the configuration should completely define the way the executable is produced.

6.3 Managing Configurations

The configuration is the basic unit with which an end-user operates. The Cactus framework provides many options to manage a configuration, some examples are: building an executable (this option checks the thornlist and dependencies of the executable and its sources, rebuilds any object files which are out of date, and re-links the final executable if necessary as described above); running a test suite (see section 13); removing object files or stored dependency information to force a clean rebuild or save disk space; or producing a document describing all the thorns in the configuration's thornlist.

7 Code Portability

The previous section has described how the make system is able to configure and make the code on all platforms. However this would be of no use if the code itself were not portable. In order to ensure code portability, the flesh is written in ANSI C. Thorns, however, may be written in C, C++, FORTRAN 77 or Fortran 90. In order to ensure that thorns are platform independent, the configuration script determines various machine-specific details, such as the presence or absence of

certain non-ANSI C functions (thorns are encouraged to use equivalent functions in the Cactus utility libraries (see section 12) which are guaranteed to be there), or the sizes of variables, which vary from platform to platform – in order to avoid problems with this, e.g. in cross-language calls or parallel jobs distributed across heterogeneous platforms, Cactus defines a set of types, such as CCTK_REAL, CCTK_INT, which are guaranteed to have the same size on all platforms and in all languages. This also means that runs can produce exactly the same results on different machines, and checkpoint files created on one machine can be restarted on any other.

8 Modularity

This section describes how thorns identify themselves to the flesh and the make system, and the way in which functionality is encapsulated and name spaces are realised. Two important pieces of terminology must be introduced before preceding. *Grid variables* are variables which can be passed between thorns or routines belonging to the same thorn through the defined flesh interface; this implies it is related to the computational grid rather than being an internal variable of the thorn or one of its routines. An *implementation* is an abstract name for the functionality offered by a thorn; all thorns providing the same implementation expose the same interfaces to the outside world, and only one such thorn may be active at a time.

8.1 CCL

Each thorn provides three specification files written in the Cactus Configuration Language (CCL). These files designate which implementation the thorn provides and the variables that the thorn expects to be available, describe the parameters it uses, and specify routines from the thorn which must be called.

Variable scope may be either private, i.e. visible only to routines from this thorn; protected, i.e. visible to all thorns in the group of friends of this thorn's implementation; or public, i.e. visible to all thorns which inherit from this thorn's implementation.

Grid Variables fall into three categories: Grid Scalars (GSs), which are single numbers (per processor); Grid Functions (GFs), which are distributed arrays with a size fixed to the problem size (all GFs have the same size); and Grid Arrays (GAs) which are similar to GFs but may have any size. Grid Variables must be one of the defined CCTK types (e.g. CCTK_REAL, CCTK_INT, etc), whose size is determined when a configuration is created (see section 6.1). Grid Functions are by default vertex-centred, however they may be staggered along any axis.

Parameter scope may similarly be private, or may be declared to be 'restricted', which means that other thorns may choose to share these parameters. Parameters may be numeric, i.e. integer or floating point, boolean, keywords, i.e. the value is a distinct token from a defined list, or arbitrary strings. The

specification of a parameter includes a specification of the allowed values, i.e. a range of numbers for numeric, a list of keywords for keyword parameters, or a set of regular expressions for string parameters.

Parameters may also be declared to be *steerable*. This means that they are not fixed for the duration of the simulation, but may have their value changed. By default parameters are non-steerable, however a thorn author can declare that a parameter may be steered at all times, or when the code is recovering from a checkpoint. This is an important distinction as, while parameters are provided as local variables to routines for speed of access, a lot of parameters are used to setup data structures or to derive other values, which would thus be unaffected if the parameter value was subsequently changed; by declaring a parameter "steerable" the thorn author is certifying that changing the value of the parameter will make a difference. (Parameters values may only be changed by calling a routine in the flesh which validates the change.)

Routines from the thorn may be scheduled to run at various time bins and relative to the times when other routines from this thorn or other thorns are run. The thorn author may also schedule groups within which routines may be scheduled; these groups are then themselves scheduled at time bins or within schedule groups analogously to routines. The scheduling mechanism is described in more detail in section 9. The scheduling of routines may be made conditional on parameters having certain values by use of `if` statements in the CCL file.

Memory for variables may be allocated throughout the course of the simulation, or allocated just during the execution of a particular scheduled routine or schedule group.

Additionally thorn authors may define specific functions which they provide to other thorns or expect other thorns to provide. This provides an aliasing mechanism whereby many thorns may provide a function which may be called by another thorn with a particular name, with the choice of which one is actually called being deferred until run-time. In the absence of any active thorn providing such a function an error is returned.

8.2 CST

When the executable for any particular configuration is built, the Cactus Specification Tool (CST) is invoked to parse the thornlist and then the individual thorns' CCL files. When the CST has parsed the CCL files it performs some consistency checks. These include: checking that all definitions of a particular implementation are consistent; checking that default values for parameters are within the allowed ranges; and checking that parameters shared from other implementations exist and have the correct types.

Once the CCL files have been verified the CST uses the specifications from these files to create small C files which register information with the flesh about the thorns. This mechanism allows the flesh library to be completely independent of any thorns, while at the same time minimising the amount of machine-generated code. Along with the registration routines the CST generates various macros which are used to provide argument lists to routines called from the

scheduler, as well as wrapper routines to pass arguments to Fortran routines. The parameter specifications are turned into macros which place the parameters as local variables in any routine needing access to parameters. All the macros are generated on a thorn-by-thorn basis and contain only information relevant to that thorn. The schedule specifications are turned into registration routines which register relevant thorn routines with the scheduler.

9 The Scheduling Mechanism

Routines (or schedule groups – for scheduling purposes they are the same) scheduled from thorns may be scheduled *before* or *after* other routines from the same or other thorns, and *while* some condition is true. In order to keep the modularity, routines may be given an alias when they are scheduled, thus allowing all thorns providing the same implementation to schedule their own routine with a common name. Routines may also be scheduled with respect to routines which do not exist, thus allowing scheduling against routines from thorns or implementations which may not be active in all simulations. Additionally the `schedule.ccl` file may include 'if' statements which only register routines with the scheduler if some condition involving parameters is true.

Once all the routines have been registered with the scheduler, the before and after specifications form a directed acyclic graph, and a topological sort is carried out. Currently this is only done once, after all the thorns for this simulation have been activated and their parameters parsed.

The *while* specification allows for a degree of dynamic control for the scheduling, based upon situations in the simulation, and allows looping. A routine may be scheduled to run while a particular integer grid scalar is non-zero. On exit from the routine this variable is checked, and if still true, the routine is run again. This is particularly useful for multi-stage time integration methods, such as the method of lines, which may schedule a schedule group in this manner.

This scheduling mechanism is rule-based as opposed to script-based. There are plans to allow scripting as well; see section 17 for further discussion of this.

10 Memory Management and Parallelisation

In order to allow the use of the same application codes in uni-grid, parallel and adapted-grid modes, each routine called from the scheduler is assigned an n-dimensional block of data to work on; this is the classical distributed computing paradigm for finite difference codes. In addition to the block of data and its size, information about each boundary is given, which specifies if this boundary is a boundary of the actual computational domain, or an internal grid boundary. The routine is passed one block of data for each variable which has storage assigned during this routine (see section 8.2 for argument list generation). The memory layout of arrays is that of Fortran, i.e. fastest-changing index first; in C these appear as one-dimensional arrays and macros are provided to convert an n-dimensional index into a linear offset withing the array.

In order to isolate the flesh from decisions about block sizes, a particular class of thorns, which provide the DRIVER implementation, is used. Drivers are responsible for memory management for grid variables, and for all parallel operations, in response to requests from the scheduler.

The driver is free to allocate memory for variables in whatever way is most appropriate, some possibilities for uni-grid are: having memory for the whole domain in one block, with this block being passed to routines; all the memory in one block, but passing sub-blocks to routines; splitting the domain into sub-domains and processing these one by one, e.g. for out of core computation; or splitting into sub-domains and then processing sub-domains in parallel.

Since the application routines just get such a block of data, data layout may change with time, and indeed will do with adaptive mesh refinement or dynamic load balancing.

As the driver is responsible for memory management, it is also the place which holds the parallelisation logic. Cactus 4.0 supports three basic parallelisation operations: ghost-zone synchronisation between sub-domains; generalised reduction operators (operations which take one or more arrays of data and return one or more scalar values from data across the whole domain); and generalised interpolation operators (operations which take one or more arrays and a set of coordinates and return one or more scalars for each coordinate value).

Synchronisation is called from the scheduler – when a routine or schedule group is scheduled it may list a set of grid variables which should be synchronised upon exit. Routines may also call synchronisation internally, however such routines would fail when more than one block of data is processed by a processor or thread, such as with adaptive mesh refinement or out of core computation, as the routine would not yet have been invoked on the remaining blocks so ghost-zone exchange would be impossible.

Similarly the other operations – reduction and interpolation – being global in nature, should only be performed one per grid, however many sub-grids there are; routines which call these routines must be scheduled as "global". This information is then passed to the driver which should then only invoke such routines once per processor, rather than once per sub-block per processor.

The flesh guarantees the presence of these parallel operations with a function registration mechanism. A thorn may always call for synchronisation, reduction or interpolation, and these are then delegated to functions provided by the driver, or return appropriate error codes if no such function has been provided.

These parallel operations are in principle all that an application programmer needs to think about as far as parallelisation is concerned. Such a programmer is insulated by the scheduler and by these functions of any specific details of the parallelisation layer; it should be immaterial at the application level whether the parallelisation is performed with MPI, PVM, CORBA, threads, SUN-RPC, or any other technology.

Since all driver thorns provide the DRIVER implementation, they are interchangeable, no other thorn should be affected by exchanging one driver for another. Which driver is used is determined by which is activated at run-time.

At the time of writing there are four driver thorns known to the authors: a simple example uni-grid, non-parallel one (`SimpleDriver`); a parallel uni-grid one (`PUGH`); a parallel fixed mesh refinement driver (`Carpet`) ; and a parallel AMR one (`PAGH`). `PUGH` is the most frequently used one, and is contained in the computational toolkit available from the Cactus web site; `Carpet` and `PAGH` have been developed independently.

10.1 `PUGH`

`PUGH` is the only driver currently distributed with the Cactus Computation Toolkit. This driver is MPI based, although it may also be compiled without MPI for single processor operation. In single-processor mode `PUGH` currently creates one block of data per variable and passes that through to routines; there has also been discussion of enhancing this to generate smaller sub-blocks tuned to the cache size to increase performance. In parallel mode `PUGH` decomposes the processors in the n-dimensions such that the smallest dimension has the fewest number of processors, which allows greater cache-efficiency when constructing messages to be passed to other processors, however this may be over-ridden by parameter setting which allow any or all of the dimensions to be set by hand. There are plans to add features to allow the processor topology to be optimised for node-based machines such as the IBM SP or Hitachi SR8000 which consist of small (typically 8 or 16 processor) nodes connected together to produce a larger machine. The load-balancing algorithm in `PUGH` is also customisable, and a modified version of `PUGH` has been used in large scale distributed computing [19].

Along with `PUGH`, the Computational Toolkit provides the auxiliary thorns `PUGHReduce` and `PUGHInterp` to provide reduction and interpolation when `PUGH` is used as a driver, and `PUGHSlab`, which provides the hyperslabbing capability used by the Toolkit's IO thorns.

11 IO

Input and Output are vital to any simulation, and indeed to any framework. In keeping with the overall design of Cactus, the flesh has minimal knowledge of the IO, but provides a mechanism so that thorn authors can call for output of their variables and the appropriate IO routines will be called.

All thorns providing IO routines register themselves with the flesh, saying they provide an *IO method*, which is a unique string identifier. Associated with an IO method are three functions: one to output all variables which need output; one to output a specific variable; and one to determine if a variable requires output. The first two of these routines then have analogues in the flesh which traverses all IO methods calling the appropriate routines, or call the routine corresponding to a specific IO method. Once per iteration the master evolution loop calls the routine to output all variables by all IO methods.

Thorns providing IO methods typically have string parameters which list the variables which should be output, how frequently (i.e. how many iterations between output), and where the output should go. In order to simplify such thorns,

and to provide standards for parameter names and meanings, the computational toolkit contains the IOUtil thorn. This thorn provides various default parameters such as the output directory, the number of iterations between output of variables, various down-sampling parameters and other parameters which may need to be used by thorns providing output of the same data but in different formats. It also provides common utility routines for IO.

The computational toolkit contains thorns which output data to screen, to ASCII output files in either xgraph/ygraph [20], or GNUPlot [21] format, binary output in IEEEIO [22] or HDF5 [23] formats, or as jpegs of 2d slices of datasets. This data may also be streamed to external programs such as visualisation clients instead of or in addition to being written to disk. Streamed visualisation modules exist for Amira [24] and the widely available OpenDX [25] visualisation toolkits.

Since all IO thorns must operate on m-dimensional slabs of the n-d data, which is (section 10) laid out and distributed in a driver dependent manner, there is also a defined interface provided by thorns providing the HYPERSLAB implementation. This interface is driver independent, and allows an IO thorn to get or modify a particular slab of data without being tied to a particular driver.

12 Utilities

The flesh provides sets of utility functions which thorns developers may use. These include common data storage utilities such as hash tables and binary trees; an arithmetical expression parser which may be used to perform calculations on data based upon an expression given as a string; access to regular expression functions (using the GNU regex library if the configure script does not find one); and an interface to create and manipulate string-keyed tables of data.

Apart from these general utilities, the flesh also provides an infrastructure to time the execution of code blocks. This consists of a way to create and manipulate timer objects, each of which has an associated set of clocks, one for each time-gathering function, such as the Unix getrusage call, the wall-clock time or the MPI_Wtime function. These timers and their clocks may be switched on or off or reset as necessary. The scheduler in the flesh uses these functions to provide timing information for all scheduled functions.

13 Test Suites

An essential component of any code is a well-defined way to perform regression tests. This is especially true for a framework, where the final result of a set of operations may depend upon many independently written and maintained modules. Cactus includes an automated test-suite system which allows thorn authors to package parameter files and the results they are expected to produce with their thorns, and then allows the current code to be tested against these results. The test-suite compares numbers in the output files to a specified tolerance, which is generally set to the accuracy which can be expected given different

machines' round-off error and machine precision, but can be over-ridden by the thorn author if this measure is not appropriate.

14 Thorn Computational Infrastructure

The overriding design criterion for Cactus has been to put as little in the flesh as possible, which allows the maximum flexibility for users to develop new functionality or modify existing functionality. The corollary to this is that the flesh by itself is not very useful. Sections 10 and 11 described two sets of infrastructure thorns, the driver and the IO thorns respectively. This section describes some of the other infrastructure which is available in the computational toolkit.

14.1 Interpolation and Reduction

While the flesh provides standard interfaces for generalised interpolation and reduction operations, and guarantees that these functions may always be called, the actual operations need to be done by the driver thorn (see section 10 , or by a closely associated thorn. A thorn providing such an operation registers a function with the flesh, with a unique name associated with the operator, such as "maximum". A thorn which needs to interpolate data in a grid variable, or to perform a reduction on it, then calls the appropriate flesh function, with the name of the operation, and the flesh then delegates the call to the thorn providing the operation. The Computational Toolkit provides several thorns for operations such as maximum, L1 and L2 norms, parallel interpolation, etc.

14.2 Coordinates

A particular grid has a single coordinate system associated with it, out of the many possibilities, and each grid array may be associated with a different coordinate system. To facilitate the use of different coordinate systems we have a thorn providing the COORDBASE implementation, which defines an interface to register and retrieve coordinate information.

This interface allows thorns providing coordinate systems to store information such as the coordinates of the boundaries of the computational grid in physical space, and the coordinates of the interior of the grid – i.e. that part of the grid which is not related to another by a symmetry operation or by a physical boundary condition. The thorn also associates details of how to calculate the coordinate value of a point with the coordinate system; this data may be calculated by origin and delta information, by associating a one-dimensional array with the coordinate direction to hold the coordinate data, or, for full generality, with an n-dimensional GA (or GF) where each point in this GA (or GF) holds the coordinate value for this coordinate for this point.

Based upon this coordinate system information, IO methods may output appropriate coordinate information to their output files or streams.

14.3 Boundary Conditions

Boundary conditions are another basic feature of any simulation. They fall into two broad classes – symmetry and physical boundary conditions. Additionally boundary conditions in both these classes may be either local or global. For example periodic boundary conditions are global symmetry conditions, the Cartoon [26] boundary condition is a symmetry condition but is local, and radiative boundary conditions are physical and local.

If all boundary conditions were local, there would be no problem calling them from within an application routine, however as soon as global conditions are used, calling from a routine suffers the same problems as any parallel operation. Additionally it is undesirable that routines should have to know anything about the symmetries of the grid – they should just know about the physical boundary conditions, and when these are applied the appropriate symmetries should be performed.

In order to address these needs, we have defined a scheme whereby instead of calling the boundary conditions directly, calculation routines merely call a function to indicate that a particular variable requires a particular physical boundary condition, and then the thorn providing this routine schedules a schedule group called ApplyBCs after the calculation routine (schedule groups may be scheduled multiple times, so there is no problem with conflicts between multiple thorns scheduling this group). All symmetry boundary condition and global physical boundary condition thorns schedule a routine in the ApplyBCs group, and the Boundary thorn schedules a routine to apply local physical boundary conditions in the group as well. Thus when the ApplyBCs group is active, each routine scheduled in it examines the list of variables which have been marked as needing boundary conditions applied, and, if it is a symmetry routine applies the symmetry, or if it is a physical boundary condition and the variable requires that physical boundary condition, applies the appropriate physical boundary condition – local physical boundary conditions are registered with the Boundary thorn, which then uses the routine it has scheduled in the group to dispatch variables to the appropriate routine.

This scheme allows new symmetry conditions to be added with no modification of any calculation routine, and allows these routines to work appropriately in the presence of multiple blocks of data per process or thread.

14.4 Elliptic Solvers

The computational toolkit provides a thorn Ell_Base which provides an experimental interface for elliptic solvers. This thorn provides a set of registration functions to allow thorns which solve certain sets of elliptic equations to be accessed via a specified interface; e.g. a thorn wanting to solve Helmholtz's equation makes a call to the function in Ell_Base passing as one argument a well-known name of a particular solver, such as PETSc [12] , which is then invoked to perform the calculation. Thus the actual elliptic solver used may be decided at run-time by a parameter choice, and new elliptic solvers may be used with no change to the application thorn's code.

14.5 Utility Thorns

In addition to necessities such as drivers, IO, coordinates and boundary conditions, there are various general utility thorns. For example we have a thorn which takes datasets and checks for NaNs; this can be either done periodically as set by a parameter, or called from other thorns. On finding a NaN the thorn can issue a warning, report the location of the NaN, stop the code, or any combination of these.

Another thorn, which is under development at the moment, interacts with the Performance API (PAPI) [27] to allow profiling of a simulation. PAPI allows access to hardware performance counters in a platform-independent manner.

15 External Interaction

The most basic way to run a simulation is to start it and then examine the data written to disk during or after the run. A framework, however, is free to provide modules to allow interaction with a running simulation. This may range from just querying the status of the simulation, through visualising data remotely, to steering the control parameters of a simulation based upon user input or predictions and analysis performed by other programs.

15.1 `HTTPD`

The Cactus Computational Toolkit contains a thorn, `HTTPD` (based on an original idea and implementation by Werner Benger), which acts as a web-server. The basic thorn allows users to connect to the running simulation with a web-browser and to examine and, if authenticated, to modify, the values of parameters. An authenticated user may also choose to terminate or pause the simulation; the user may also tell the code to pause at a specific iteration or after a certain amount of simulation time. The ubiquitous nature of web-browsers makes this an almost ideal way to interact with the simulation.

Additional thorns may be used to provide further capabilities to the web server. For example, application thorns can also add their own information pages and provide custom interfaces to their parameters. The toolkit contains the thorn `HTTPDExtra` which provides access to the IO methods within the framework by providing a view port by which users may view two-dimensional slices through their data using IOJpeg, and through which any file known by the simulation may be downloaded. This allows users to examine the data produced by a remotely running simulation on the fly by streaming data to visualisation tools such as OpenDX [25], Amira [24] or GNUPlot [21], see section 15.2

The combination of the ability to pause the simulation, to examine data remotely, and to steer simulation parameters provides a powerful method for scientists to work with their simulations. This combination could be enhanced further to provide a debugging facility.

Currently `HTTPD` only supports un-encrypted HTTP connections, however there are plans to enhance it to use HTTP over TLS [28] [29]/SSL [30] or over GSI [31], providing capabilities for secure authentication and encrypted sessions.

15.2 Remote Visualisation and Data Analysis

As mentioned above, the Cactus Computational Toolkit provides mechanisms to query and analyse data from a running simulation. While it is true that because the simulation may be scheduled in the batch queue at unpredictable times, the scientist will not necessarily be on hand to connect to the simulation at the moment that it is launched, the simulations that create the most egregious difficulties with turnaround times typically run for hours or days. So while the interactive analysis may not necessarily capture the entire run, it will intercept at least part of it. Any on-line analysis of the running job before the data gets archived to tertiary storage offers an opportunity to dramatically reduce the turn-around time for the analysis of the simulation outcome.

Keeping up with the data production rate of these simulations is a tall order, whether the data is streamed directly to a visualisation client or pre-digested data is sent. Earlier pre-Cactus experiments with a circa 1992 CAVE application called the Cosmic Worm demonstrated the benefits of using a parallel isosurfacer located on another host to match this throughput. However, it also pointed out the fact that it doesn't take too many processors limits primarily from the I/O rate are reached. It does no good to perform the isosurface any faster if its performance is dominated by the rate at which data can be delivered to it.

For the first remote visualisation capability in Cactus, it was decided to implement a parallel isosurfacer in-situ with the simulation so that it would use the same domain decomposition and number of processors as the simulation itself. The isosurfacer would only send polygons to a very simple remote visualisation client that runs locally on the user's desktop; offering sometimes orders of magnitude in data reduction with throughput that exactly matched the production rate of the simulation code. This paradigm was further enhanced to include geodesics and other geometric visualisation techniques.

Other ways to reduce the throughput to data clients is to send only subsets of the data. The Cactus IO methods, in particular the `HDF5` method, allow the data to be down-sampled before being written to disk or streamed to a visualisation client. Another way to stream less data is to pick a data-set of a lower dimension from the full data-set, and we have IO methods which will produce 1 or 2-dimensional subsets of three-dimensional data-sets.

These and other remote technologies are being actively developed through a number of projects for example [32,33,34].

16 Applications

16.1 Numerical Methods

Although Cactus was designed to support, or to be extensible to support, different numerical methods, most of the existing infrastructure has been developed around regular structured meshes with a single physical domain and coordinate system, and more specifically for finite difference methods with spatial dimensions of three or less. This is well-suited to many calculations, particularly in the

problem domains for which Cactus was first developed. However there are many other problem domains for which these restrictions pose problems.

Fundamental support for other numerical methods typically involves the development of a standard driver and associated thorns (with possible extensions of CCL features). It is hoped that many of the existing infrastructure thorns, such as those for IO and coordinates, can be developed to support additional methods. For example, the addition of a hyperslabbing thorn for a particular driver should allow for IO thorns to provide for different underlying methods.

With the addition of appropriate drivers (see section 17), and associated driver infrastructure thorns, the Cactus framework can be used with structured, unstructured, regular and irregular meshes, and can implement finite differencing, finite volumes or finite elements, particle and spectral methods, as well as ray tracing, Monte Carlo etc.

16.2 Scalar Wave Equation

The solution of the scalar wave equation using 3D Cartesian coordinates provides a prototypical example of how a wide class of initial value problems can be implemented with finite differencing methods in Cactus. The `CactusWave` arrangement contains thorns which implement this solution in each of the currently supported programming languages, along with additional thorns providing initial data and source terms [13].

16.3 General Relativity

The introduction describes the historical relationship between Cactus and numerical relativity. The requirements of this diverse community for collaboratively developing, running, analysing and visualising large-scale simulations continue to drive the development of Cactus, motivating computational science research into advanced visualisation and parallel IO as well as Grid computing.

The Cactus framework and Computational Toolkit is used as the base environment for numerical relativity codes and projects in nearly two dozen distinct groups worldwide, including the 10-institution EU Network collaboration and others in Europe, as well as groups in the US, Mexico, South Africa, and Japan. In addition, the Cactus Einstein Toolkit provides an infrastructure and common tools for numerical relativity, and groups which choose to follow the same conventions can easily inter-operate and collaborate, sharing any of their own thorns, and testing and verifying thorns from other groups (and many do).

The numerical relativity community's use of Cactus provides an example of how a computational framework can become a focal point for a field. With many different relativity arrangements now freely available, both large groups and individuals can work in this field, concentrating on their physics rather than on computational science. In this environment, smaller groups are more easily able to become involved, as they can build on top of open and existing work of other groups, while concentrating on their particular expertise. For example, an individual of group with expertise in mathematical analysis of gravitational

waves can easily implement thorns to do this, using other groups' initial data and evolution thorns. Such an approach is increasingly becoming a point of entry into this field for groups around the world.

In addition to the leverage and community building aspect of an open framework like Cactus, the abstractions it provides make it possible for a new technology, such as fixed or adaptive mesh refinement, improved IO, new data types, etc, to be added and made available to an entire community with minimal changes to their thorns. Not only does this have obvious benefits to the user communities, it has the added effect of attracting computational science groups to work with Cactus in developing their technologies! They find not only a rich application oriented environment that helps guide development of their tools, but they also find satisfaction in knowing their tools find actual use in other communities.

16.4 Other Fields

Although Cactus began as a project to support the relativity community it is an open framework for many applications in different disciplines. Here we simply list a few of a growing number of active projects of which we are aware at present. As part of an NSF project in the US, the Zeus suite of codes for Newtonian magnetohydrodynamics has been developed into a set of thorns (in the `Zeus` arrangement, which use AMR, and is publicly available. In another discipline, Cactus has been used as a framework to write parallel chemical reactor flow applications, where it was found to be an effective tool for speeding up simulations across multiple processors [35]. Climate modellers at NASA and in the Netherlands have also taken interest in Cactus, and are actively developing thorns for both shallow water models and a coupling of two ocean models [36]. Other application communities prototyping applications in Cactus, or investigating its use as a framework for their applications, include the fusion simulation community, avalanche simulators, and geophysics groups, to name a few.

16.5 Grid Computing

Last but not least, we turn to Cactus as a framework, not only for traditional applications, but also for developing applications that may run in a Grid environment. The Grid is an exciting new development in computing, promising to seamlessly connect computational resources across an organisation or across the world, and to enable new kinds of computational processes of unprecedented scales. As network speeds increase, the effective "distance" between computing devices decreases. By harnessing PCs, compute servers, file servers, handhelds, etc, distributed across many locations, but connected by ever better networks, dynamically configurable virtual computers will be created for many uses [1].

However, even if the networks, resources, and infrastructure are all functioning perfectly, the Grid presents both new challenges and new possibilities for applications. Although the applications of today may (or may well not) actually run on a Grid, in order to run efficiently, or more importantly, to take advantage of *new* classes of computational processes that will be possible in a Grid world,

applications must be retooled, or built anew. And yet, as we stressed in the introduction, most application developers and users want to focus on their scientific or engineering discipline, and not on the underlying computational technology.

For the same reasons that Cactus has been an effective tool for building applications that take advantage of advanced computational science capabilities on many different computer architectures, while hiding many details from the user, it has also been one of the leading sources of early Grid applications. Its flexible and portable architecture make it a good match for the varying needs of Grid applications. In particular, it has traditionally been used for many remote and distributed computing demonstrations [37,38,39,19]. Because of the abstractions available in Cactus, it has been possible to modify the various data distribution, message passing, and IO layers so they function efficiently in a distributed environment, without modifying the applications themselves [19]. For example, it has been possible to take very complex, production code for solving Einstein's equations which is coupled to GR hydrodynamics to simulate colliding neutron stars, and distributed across multiple supercomputers on different continents, while it is remote visualised and controlled from yet another location.

This is just the beginning of what can be done on future Grids. We imagine a world where applications are able autonomously to seek out resources, and respond to both their own changing needs and the changing characteristics of the Grid itself, as networks and computational resources change with time. Applications will be able not only to parallelise across different machines on the Grid, but will also be able spawn tasks, migrate from site to site to find more appropriate resources for their current task, acquire additional resources, move files, notify users of new events, etc [4]. Early prototypes of this kind of capability have already been developed in Cactus; the "Cactus Worm" demonstration of Supercomputing 2000 was a Cactus application (any application inserted in the Cactus framework would do) that was able to: run on any given site, make use of the existing Grid infrastructure to move itself to a new site on a Grid, register itself with a central tracker so its users could monitor or control it, and then move again if desired [40]. This demonstration of new types of computation in a Grid environment foreshadows a much more complex world of the future.

However, the underlying Grid technology is quite complex, and varies from site to site. Even if an application programmer learns all the details of one Grid technology, and implement specific code to take advantage of it, the application would likely lose some degree of portability, crucial in the Grid world. Not only does one wish to have applications that run on any major computing site, with different versions of Grid infrastructure, but also one wants them to run on local laptops or mobile devices, which may have no Grid infrastructure at all!

Learning from the Cactus Framework, one solution is to develop a Grid Application Toolkit (GAT), that abstracts various Grid services for the application developer, while inter-operating with any particular implementation. The GridLab project [41], is currently underway to do just this. The goal is to develop a toolkit for *all applications*, whether they are in the Cactus framework or not. The GAT will provide an abstracted view of Grid services of various kinds to the

user/developer, and will dynamically discover which services are actually available that provide the desired functionality (e.g., "find resource", "move file", "spawn job", etc). It is far beyond the scope of this article to go into further detail, but we refer the reader [4,42,43] for more detail.

17 Future Plans

Cactus 4.0 is a great step from previous versions of Cactus, providing much more modularity and flexibility. However there are numerous shortcomings, and many things are planned for future versions.

Support for unstructured meshes is intended, which opens up the use of Cactus for finite volume and finite element calculations. We have had many discussions with communities, e.g. aerospace and earthquake simulation, which use these methods, and have well-developed plans on how to support such meshes.

Another plan we have is to enable more complicated scenarios, consider the following: (*i*) Many CFD calculations also use multi-block grids – grids which are themselves made up of many sub-grids, some being structured and some unstructured, e.g. simulation of flow past an aircraft wing may use an overall structured mesh, with a small unstructured grid block around the wing and its stores; (*ii*) Climate modelling simulations have many distinct physical domains, such as sea, icebergs, land and air, each of which has its own physical model, and then interaction on the boundaries; (*iii*) Some astrophysical simulations require several coordinate patches to cover a surface or volume.

These are all sub-classes of what we refer to as "multi-model". While all can be done currently within Cactus using Grid Arrays, this is not the most natural way to provide this functionality, and requires more communication and coordination between individual module writers than is desirable. We have defined a specification to enable any combination of these types of scenarios and intend to enable this functionality in a future release.

Currently all Grid Functions or Grid Arrays are distributed according to their dimension across all processors. This is the most common need, however there are many situations where one would like to define a variable which only lives on the boundary of a grid. This can be faked at the moment by defining a GA with one lower dimension than GFs and ignoring it on internal processors, however even then the distribution of this GA may not correspond to the distribution of a GF on that face. To solve this problem we intend to introduce a new class of grid objects, sub-GFs, which will have the appropriate distribution.

Another class of methods not currently supported are particle methods, such as Particle-In-Cell (PIC) and Smoothed-Particle-Hydrodynamics (SPH). Both these methods require a small amount of additional communication infrastructure to enable exchange of particle information between different processors.

Currently thorns can only be activated when the parameter file is read, and not during the course of the simulation. An obvious enhancement is to enable thorn activation, and even de-activation, at any point in the program execution.

This would allow the code to act as a compute server, receiving requests for simulations from external sources, and, if thorns were dynamically loadable libraries as opposed to statically linked, would allow new functionality to be incorporated at will into long-running simulations.

Another enhancement would be to allow scripting as an alternative to the current scheduling mechanism. The current mechanism allows thorns to interoperate and for simulations to be performed with the logic of when things happen encapsulated in the schedule CCL file; other frameworks do the same thing by providing a scripting interface, which gives more complete control of the flow of execution, at the expense of the user needing to know more of the internals. Both schemes have advantages and disadvantages. In the future we would like to allow users to script operations using Perl, Python, or other scripting languages.

Currently Cactus only allows thorns in C, C++, and Fortran. Addition of other languages is fairly straightforward as long as there exists a method to map between the language's data and C data. We plan to add support for thorns written in Perl, Python and Java in the future; this will be done by defining an interface which would allow a thorn to inform the make system and the CST about the mappings and how to produce object files from a language's source files, thus allowing support for any language to be added by writing an appropriate thorn.

In section 6 it was stated that Cactus is not installed. In the future we would like to support the installation of the Cactus flesh and the Computational Toolkit arrangements in a central location, with other arrangements installed elsewhere, for example in users' home directories, thus providing a consistent checked out version of Cactus across all users on a machine. In principle it would also be possible for configurations to be centralised to some extent – the object files and libraries associated with the Computational Toolkit for a particular configuration could be provided by the system administrator, and only the object files and libraries of user-local arrangements would then need to be compiled by the user. However, given the dynamic nature of code development it is not clear that such centralised configurations would be practicable.

All the above require modifications to the flesh. However the primary method to add functionality is by adding new thorns.

One basic feature intended for the future is to develop a specification for a set of parallel operation, the Cactus Communication Infrastructure (CCI) which will abstract the most commonly used parallel operations, and provide a set of thorns which implement this infrastructure using common communication libraries such as MPI, PVM, CORBA, SUN-RPC, etc. This would then allow driver thorns themselves to be built independently of the underlying parallel infrastructure, and greatly increase the number of parallel operations available to other thorns.

Another set of useful functionality would be to develop remote debuggers and profilers based upon the current `HTTPD` thorn and the expression parser. Indeed HTTP is far from being the only possible way to interact with a remote code, and thorns could be written to present allow interaction with a simulation via a

SQL or LDAP interface, or as a web-service; there is already work on steering directly from visualisation clients using the streamed HDF5 interface [44].

Multi-grid methods are commonly used to solve elliptic equations, however all current methods of putting elliptic solvers into Cactus require knowledge of the driver or at least of the parallelisation technology. It should be possible to develop a set of thorns which use the scheduler in a driver and parallel-layer independent manner to solve elliptic problems.

The current set of thorns allows IO in many formats, however it is straightforward to add other formats, and there is at least one group with plans to write a NetCDF IO thorn. Additionally the infrastructure for IO makes it plausible to write IO methods that do transformations on the data first, e.g. spectral decomposition, before writing them out to disk or a stream; such methods would need to call routines from lower-level IO methods, and we have designed an interface which presents lower-level IO methods as stream-equivalents, thus insulating the higher-level IO methods from knowledge of specific lower-level IO methods.

18 Acknowledgements

Cactus has evolved and been developed over many years, with contributions from a great many people. Originally an outgrowth of work in the NCSA numerical relativity group in the early 1990's, the first framework with the name Cactus was developed at the Max Planck Institute for Gravitational Physics by Paul Walker and Joan Massó. Its creation and further development was influenced greatly, and directly contributed to, by many friends, especially Werner Benger, Bernd Brügmann, Thomas Dramlitsch, Mark Miller, Manish Parashar, David Rideout, Matei Ripeanu, Erik Schnetter, Jonathan Thornburg, Malcolm Tobias and the numerical relativity groups at the Albert Einstein Institute and Washington University. It has also been generously supported by the MPG, NCSA, Intel, SGI, Sun, and Compaq, and by grants from the DFN-Verein (TiKSL, GriKSL), Microsoft, NASA, and the US NSF grant PHY-9979985 (ASC).

References

1. Foster, I.: (2002). The Grid: A New Infrastructure for 21st Century Science. *Physics Today*, **Febrary**.
2. Component Component Architecture (CCA) Home Page `http://www.cca-forum.org`.
3. Cactus Numerical Relativity Community `http://www.cactuscode.org/Community/Relativity.html`.
4. Allen, G., Seidel, E., and Shalf, J.: (2002). Scientific Computing on the Grid. *Byte*, **Spring**:24–32.
5. Cactus: http://www.cactuscode.org.
6. *Cactus Users Guide* `http://www.cactuscode.org/Guides/Stable/UsersGuide/UsersGuideStable.pdf`.

7. Allen, G., Benger, W., Dramlitsch, T., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., and Seidel, E.: (2001). Cactus Grid Computing: Review of Current Development. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Europar 2001: Parallel Processing, Proceedings of 7th International Conference Manchester, UK August 28-31, 2001*. Springer. `http://www.cactuscode.org/Papers/Europar01.ps.gz`.
8. Parashar, M. and Browne, J. C.: (2000). *IMA Volume on Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, chapter System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement, pages 1–18. Springer-Verlag.
9. Grid Adaptive Computational Engine (GrACE) `http://www.caip.rutgers.edu/~parashar/TASSL/Projects/GrACE/`.
10. Anninos, P., Camarda, K., Massó, J., Seidel, E., Suen, W.-M., and Towns, J.: (1995). Three-dimensional numerical relativity: The evolution of black holes. *Phys. Rev. D*, **52**(4):2059–2082.
11. Walker, P.: (1998). *Horizons, Hyperbolic Systems, and Inner Boundary Conditions in Numerical Relativity*. Ph.D. thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois.
12. Balay, S., Gropp, W., McInnes, L. C., and Smith, B.: (1998). PETSc- The Portable, Extensible Toolkit for Scientific Compuation. `http://www.mcs.anl.gov/petsc/`.
13. *Cactus Computational Toolkit Thorn Guide* `http://www.cactuscode.org/Guides/Stable/ThornGuide/ThornGuideStable.pdf`.
14. TASC: Software Engineering Support of the Third Round of Scientific Grand Challenge Investigations: Task 4, Review of Current Frameworks `http://sdcd.gsfc.nasa.gov/ESS/esmf_tasc/t400.html`.
15. *Cactus Maintainers Guide* `http://www.cactuscode.org/Guides/Stable/MaintGuide/MaintGuideStable.pdf`.
16. Concurrent Versions System (CVS) Home Page `http://www.cvshome.org/`.
17. Cygnus Solutions (Cygwin) Home Page: `http://www.cygwin.com`.
18. The Autoconf Home Page `http://www.gnu.org/software/autoconf/autoconf.html`.
19. Allen, G., Dramlitsch, T., Foster, I., Karonis, N., Ripeanu, M., Seidel, E., and Toonen, B.: (2001). Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Proceedings of Supercomputing 2001, Denver, USA*. Http://www.cactuscode.org/Papers/GordonBell_2001.ps.gz.
20. The xgraph and ygraph Home Pages `http://jean-luc.aei-potsdam.mpg.de/Codes/xgraph`, `http://www.aei.mpg.de/~pollney/ygraph`.
21. GNUPlot Home Page `http://www.gnuplot.org`.
22. IEEEIO Home Page `http://zeus.ncsa.uiuc.edu/~jshalf/FlexIO/`.
23. Hierarchical Data Format Version 5 (HDF5) Home Page `http://hdf.ncsa.uiuc.edu/HDF5`.
24. *Amira - Users Guide and Reference Manual* and *AmiraDev - Programmers Guide,* Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) and Indeed - Visual Concepts, Berlin, `http://amira.zib.de`.
25. Open DX Home Page `http://www.opendx.org`.
26. Alcubierre, M., Brandt, S., Brügmann, B., Holz, D., Seidel, E., Takahashi, R., and Thornburg, J.: (2001). Symmetry without Symmetry: Numerical Simulation of Axisymmetric Systems using Cartesian Grids. *Int. J. Mod. Phys. D*, **10**(3):273–289. URL http://ejournals.worldscientific.com.sg/ijmpd/10/1003/S0218271801000834.html.

27. The Performance API (PAPI) Home Page `http://icl.cs.utk.edu/projects/papi/`.
28. Dierks, T. and Allen, C.: The TLS Protocol Version 1.0: RFC 2246 January 1999.
29. Rescorla, E.: HTTP Over TLS: RFC 2818 May 2000.
30. Frier, A., Karlton, P., and Kocher, P.: The SSL 3.0 Protocol. Netscape Communications Corp. November 18, 1996.
31. Foster, I., Kesselman, C., Tsudik, G., and Tuecke, S.: (1998). A Security Architecture for Computational Grids. In *Proceedings of 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92.
32. DFN Gigabit Project *"Tele-Immersion: Collision of Black Holes"* (TIKSL) Home Page `http://www.zib.de/Visual/projects/TIKSL`.
33. DFN-Verein Project *"Development of Grid Based Simulation and Visualization Techniques"* (GRIKSL) Home Page `http://www.griksl.org`.
34. Shalf, J. and Bethel, E. W.: Cactus and Visapult: An Ultra-High Performance Grid-Distributed Visualization Architecture Using Connectionless Protocols. *Submitted to IEEE Computer Graphics and Animation*.
35. Camarda, K., He, Y., and Bishop, K. A.: (2001). A Parallel Chemical Reactor Simulation Using Cactus. In *Proceedings of Linux Clusters: The HPC Revolution, NCSA 2001*.
36. Dijkstra, H. A., Oksuzoglu, H., Wubs, F. W., and Botta, F. F.: (2001). A Fully Implicit Model of the three-dimensional thermohaline ocean circulation. *Journal of Computational Physics*, **173**:685715.
37. Allen, G., Goodale, T., Massó, J., and Seidel, E.: (1999). The Cactus Computational Toolkit and Using Distributed Computing to Collide Neutron Stars. In *Proceedings of Eighth IEEE International Symposium on High Performance Distributed Computing, HPDC-8, Redondo Beach, 1999*. IEEE Press.
38. Allen, G., Benger, W., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., and Shalf, J.: (2000). The Cactus Code: A Problem Solving Environment for the Grid. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing, HPDC-9, August 1-4 2000, Pittsburgh*, pages 253–260. IEEE Press. `http://www.cactuscode.org/Papers/HPDC9_2000.ps.gz`.
39. Benger, W., Foster, I., Novotny, J., Seidel, E., Shalf, J., Smith, W., and Walker, P.: (March 1999). Numerical Relativity in a Distributed Environment. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. `http://www.zib.de/visual/projects/TIKSL/Papers/numrel_dist.ps`.
40. Allen, G., Dramlitsch, T., Goodale, T., Lanfermann, G., Radke, T., Seidel, E., Kielmann, T., Verstoep, K., Balaton, Z., Kacsuk, P., Szalai, F., Gehring, J., Keller, A., Streit, A., Matyska, L., Ruda, M., Krenek, A., Frese, H., Knipp, H., Merzky, A., Reinefeld, A., Schintke, F., Ludwiczak, B., Nabrzyski, J., Pukacki, J., Kersken, H.-P., and Russell, M.: (2001). Early Experiences with the EGrid Testbed. In *IEEE International Symposium on Cluster Computing and the Grid, Brisbane, Australia, May 16-18 2001*. Available at `http://www.cactuscode.org/Papers/CCGrid_2001.pdf.gz`.
41. GridLab: A Grid Application Toolkit and Testbed Project Home Page: `http://www.gridlab.org`.
42. Allen, G., Goodale, T., Russell, M., Seidel, E., and Shalf, J.: (2003). *Grid Computing: Making the Global Infrastructure a Reality*, chapter Classifying and Enabling Grid Applications. Wiley.
43. Seidel, E., Allen, G., Merzky, A., and Nabrzyski, J.: (2002). GridLab — A Grid Application Toolkit and Testbed. *Future Generation Computer Systems*, **18**:1143–1153.

44. Allen, G., Benger, W., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., and Shalf, J.: (2001). Cactus Tools for Grid Applications. *Cluster Computing*, **4**:179–188. `http://www.cactuscode.org/Papers/CactusTools.ps.gz`.