Final Report
Enabling Science at the Petascale: From Binary Systems and Stellar Core Collapse To Gamma-Ray Bursts

Peter Diener[1,2], Frank Löffler[1], Jian Tao[1], and Steven R. Brandt[1,3]

[1] *Center for Computation and Technology, Louisiana State University, Baton Rouge, LA 70803, USA*

[2] *Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803, USA and*

[3] *Division of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803, USA*

Understanding the physics of a Gamma-Ray Burst (GRB) demands truly petascale multi-physics simulations on the largest and most modern High Performance Computing (HPC) systems. To take full advantage of such systems extreme requirements are placed on the computational infrastructure on which such codes are based. In this report we describe the work within the Cactus framework to enable such demanding simulations on Blue Waters.

I. INTRODUCTION

The overall topic of our PRAC project is to perform full 3D evolutions of core-collapse supernovae and of hyper-massive magnetized neutron star remnants of neutron star mergers. Such simulations require evolution of the spacetime itself (as described by Einstein's theory of general relativity), fully general relativistic magnetohydrodynamics and neutrino transport. We base our code on the Cactus [1, 3] computational framework in order to take advantage of the interoperability of modules. For example the spacetime evolution code developed originally for binary black hole simulations could be coupled fairly straightforwardly to a more recently developed magnetohydrodynamics module. In addition Cactus provides advanced Adaptive Mesh Refinement (AMR) and multi-patch capabilities (through Carpet [2, 5, 6]) that are crucial for such simulations.

However, we also realized that there where several improvements that could be made to the Cactus/Carpet infrastructure in order to improve performance and scalability of the code. The first two (as detailed in section II) regards Carpet directly while the last one could enable the use of the GPU nodes on Blue Waters. In this report on the work done during the Blue Waters PRAC subaward we will first describe the challenges we faced (section II), then the work we did to overcome the challenges (section III) and show some results of the improvements (section IV). Finally in section V we will comment on the future directions for further improvements.

II. CODE CHALLENGES

Our simulations are based on Cactus: a framework for solving partial differential equations (PDEs) on a computational domain using a combination of adaptive mesh refinement (AMR) and multi-patch techniques with the computational domain decomposed among MPI processes. In Cactus, the details of the AMR and multi-patch algorithm (processor decomposition, memory allocation and data communications) is performed by Carpet.

Based on numerous performance profiling experiments we identified three areas where we expected that modifications to data structures and algorithms could lead to overall performance and

scalability improvements. In this section we briefly describe the existing data structures and algorithms and their short comings. In section III we will then describe the implemented changes, improvements and additions to our code base and in section IV we will show some results of the improvements.

A. Regridding data structures

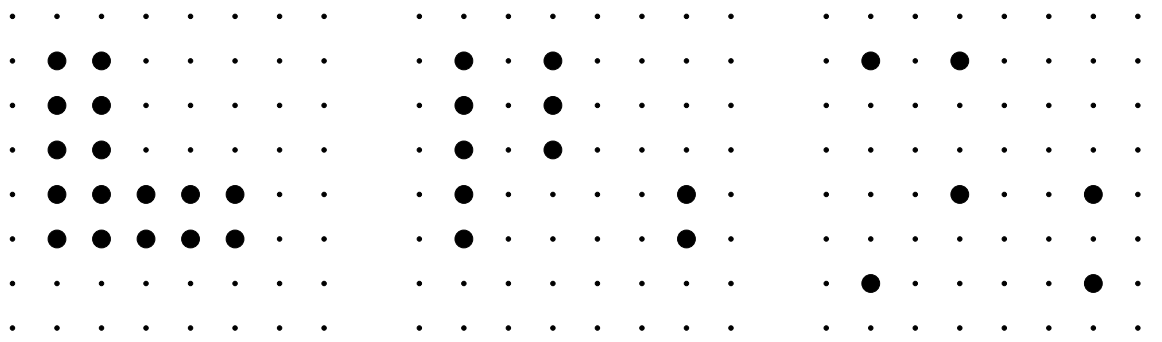
Whenever the grid structure needs to change (as dictated by the physics), the grid has to be redistributed among the MPI processes, data copied when necessary and the new communication schedule determined. Thus, one of the most prominent challenges we are facing at scale is managing the grid structure and its domain decomposition. The number of objects that need to be tracked grows linearly with the number of MPI processes used, while their possible interactions (which region needs to exchange information with which other region) grows quadratically. The goal is to achieve this with at most linear cost, leading to a constant time if parallelized over all processes. With the current data structures, this cost, though inconsequential if using up to about 1,000 MPI processes, begins to dominate the regridding time (i.e. the time required to adapt the grid structure and re-balance the domain decomposition) for larger numbers of MPI processes. These issues are due to the existing data structures for storing this information and the accompanying order n^2 algorithms.

B. Load balancing algorithm

In addition to the inefficient data structures and algorithms for handling the decomposed domain (see section II A) we have also observed significant deficiencies with the algorithm used to split the grid structure among processors. In Carpet each refinement level is split among all MPI processes. Each refinement level can consist of multiple maps that potentially can have very different sizes. The current algorithm has two main problems. Firstly, for the case of splitting n maps (each map has the shape of a rectangular box) on P processes, each map is assigned an integer number of processors P_i in such a way that $P_1 + P_2 + \ldots + P_n = P$ and $P_i/P \approx N_i/N$ where N_i is the number of grid-points on each map and $N_1 + N_2 + \ldots + N_n = N$. In most cases this leads to some load imbalance. Secondly, splitting each map n_i, across the assigned processors p_i is done using a recursive algorithm where at each recursion level the box is split in two along the longest dimension and each new box assigned an integer number of processes chosen in such a way that the load imbalance between the two boxes are minimized. This continues until each box is assigned 1 MPI process. Using this algorithm it can happen that a seemingly good choice for a split at an early recursion level can lead to very large imbalances at a later level, whereas a slightly worse choice at an early level would allow for a much better split at a later level. In some cases (at large MPI process counts) load imbalances up to 20% has been seen, leading to some processors being idle 1/5 of the runtime.

C. CaKernel

As graphics processing units (GPUs) become more and more prominent on high performance computing (HPC) systems, we started adding Cactus support for GPU programming with the CaKernel project. GPU programming is complicated by the fact that the programmer has to explicitly handle the copying of data between the CPU and GPU and launching computational kernels on the GPU. The main idea behind the project is to take advantage of the modularity of



(a) L-shaped region (b) x-derivative of this region (c) xy-derivative of this region

FIG. 1: An L-shaped region and its derivative. The xy-derivative consists only of the "key points" of the shape

.

Cactus to hide this complexity as much as possible. Therefore, in the CaKernel project, we have created a Cactus-aware GPU programming infrastructure allows Cactus, at runtime, to keep track of which variables are stored on the GPU and which are stored on the CPU as well as to control when data needs to be transferred. This provides a depth of awareness about data placement that goes beyond what is possible to describe using simple annotations such as those available e.g. by OpenACC. Additionally, the construction of the actual computational kernels have been simplified through the use of Kranc's automatic code generation feature that takes advantage of optimized macro templates for stencil operations. The goal of this part of the project was to make CaKernel robust enough for inclusion into our production simulation.

III. CODE IMPROVEMENTS

In this section we in turn describe describe how we addressed each of the challenges described in section II: 1) improvements to the data structures for regridding in Carpet, 2) an improved load balancing algorithm and 3) the development of abstractions in the Cactus framework (CaKernel) to enable automatic code generation for GPUs.

A. Regridding data structures

In order to improve on the performance and scaling of the regridding data structures and algorithms we implemented completely new data structures and algorithms based on storing discrete derivatives of bounding box sets. This is a convenient way of storing bonding box sets for regions of arbitrary shape defined on a uniform grid and is illustrated in figure 1. Here we show, as an example, an L-shaped region and its x- and xy-derivatives. The main idea is that a discrete derivative reduces the number of elements needed to describe the region. As can be seen from the figure an L-shaped region in 2 dimensions can be described by just 6 points, regardless of how many points are in the set. The full set can be reconstructed from the derivative using the unique anti-derivative. When additionally the derivative bounding box sets are stored in a tree structure, all set operations needed can be implemented with log-linear cost. For more details see [4].

B. Load balancing algorithm

The improvements to the load balancing algorithm were two-fold. Firstly, we overcame the limitation that maps had to be assigned an integer number of MPI processes. That is, we now allow an MPI process to be assigned to handle pieces of 2 different maps. This allows us to better handle the cases of multiple patches of different sizes as well as different refinement regions (on the same refinement level) of different sizes. As an example consider the case of 3 patches, where 2 patches (patches 1 and 2) are the same size and the first patch (patch 0) is twice the size. Splitting this on 3 processors would before lead to one processor having twice as many grid points as the others. In the new scheme processor 0 would get 2/3 of patch 0, processor 1 would get 1/3 of patch 0 and 2/3 of patch 1 while processor 2 would get 1/3 of patch 1 and all of patch 2.

Secondly we modified the recursive load balance algorithm. The new algorithm is similar to the existing one in that, when splitting a rectangular region with N grid points on M processors, it splits a region in the longest direction first into 2 regions; the first region with N_1 points gets assigned M_1 processors and the second with $N_2 = N - N_1$ gets assigned $M_2 = M - M_1$ processors. Here N_1, M_1, N_2 and M_2 are chosen such that the numbers of grid points per processor in each region (N_1/M_1 and N_2/M_2) are as close to each other as possible. Then each of those regions are recursively split into 2 regions again, and this continues until M regions have been created that are each associated with a single processor. However, in order to avoid the case of a choice made at an early recursion level leads to very bad choices at a later lever, the new algorithm explores several different options at each level, evaluating the final load imbalance for the different options once the full decomposition tree have been constructed, and keeping only the best. It is not feasible to explore all options, as this would lead to an exponential number of total options. Instead we explore a parameter adjustable number of options at early recursion levels and a decreasing number at later recursion levels.

C. CaKernel

In this project we have focused mainly on improving the interoperability of Cactus and CaKernel in order to more easily allow developers to implement new (or adapt existing) functionality to make use of GPUs. Cactus already provides the following grid abstractions (common in high level programming frameworks for parallel block-structured HPC applications):

- The *Grid Hierarchy (GH)* represents the distributed, adaptive hierarchy of grids. The abstraction enables application developers to create, operate and destroy hierarchical grid structures. The regridding and partitioning of a grid structure are done automatically whenever necessary. In Cactus, grid operations are handled by a driver thorn which is a special module in Cactus.

- A *Grid Function (GF)* is a distributed data structure that represents the variables in an application. Storage, synchronization, arithmetic, and reduction operations are implemented for the GF by standard thorns. The application developers are responsible for providing proper routines for initialization, boundary updates, etc.

- The *Grid Geometry (GG)* represents the coordinates, bounding boxes, and bounding box lists of the computational domain. Operations on the GG, such as union, intersection, refine, and coarsen are usually implemented in a driver thorn as well.

In addition to this CaKernel now provides the following kernel abstractions:

- A *CaKernel Descriptor* describes one or more numerical kernels, dependencies, such as grid functions and parameters required by the kernel, and grid point relations with its neighbors; the information provided in the descriptor is then used to generate a kernel frame (macros) that performs automatic data fetching, caching and synchronization with the host;
- A *Numerical Kernel* uses kernel-specific auto-generated macros; the function may be generated via other packages (such as Kranc), and operates point-wise;
- The *CaKernel Scheduler* schedules CaKernel launches and other CaKernel functions in exactly the same way as other Cactus functions; data dependencies are evaluated and an optimal strategy for transferring data and performing computation is selected automatically.

To couple together the grid and kernel abstractions we added a Catus component (thorn), Accelerator, that tracks which parts of what grid functions are valid where, and which triggers the necessary host–device copy operations that are provided by other, architecture-specific thorns. By also modifying Kranc to generate CaKernel code, we were then able to use our Kranc script for the BSSN spacetime evolution module McLachlan to generate versions for both the CPU and GPU. Thus we are now able to hide the complexities of writing GPU kernels and scheduling host to device memory transfers from the user who can focus on providing the equations in high level tensorial form.

IV. RESULTS

In this section we present some results showing the performance improvements and new capabilities for each of the three sub projects.

A. Regridding data structures

We have tested the effect of the new regridding data structures on the scaling of Carpet on various super computers. The results are shown in figure 2. As can be seen from the pink (finely dotted) and blue (dotted) curves changing from the old to the new data structures resulted in a dramatic improvement of the scalability of the code. We see a clear deterioration of the scaling when running on more than 16k cores on Blue Waters. This is most likely due to the serial nature of the algorithms.

B. Load balancing algorithm

As all MPI processes have to wait until the slowest one finishes (i.e. the MPI process with the most grid-point), we calculate the load imbalance L defined as

$$L = \frac{\max(n_i) - n_{\text{ideal}}}{n_{\text{ideal}}}. \tag{4.1}$$

Here $\max(n_i)$ is the maximum number of grid-points that have been assigned to any of the MPI processes and $n_{\text{ideal}} = N/M$ is the ideal number of grid-points that should be assigned to an MPI process. Note, that we currently assume that the computational cost of all grid-points are the same. For ideal load balance, $\max(n_i) = n_{\text{ideal}}$ and $L = 0$. In figure 3 we plot L (in percent) as a function of the number of MPI processes for the case of a cubical box with 512^3 grid-points for the old (red) and new (green) algorithms. It is clear that the new algorithm consistently yields

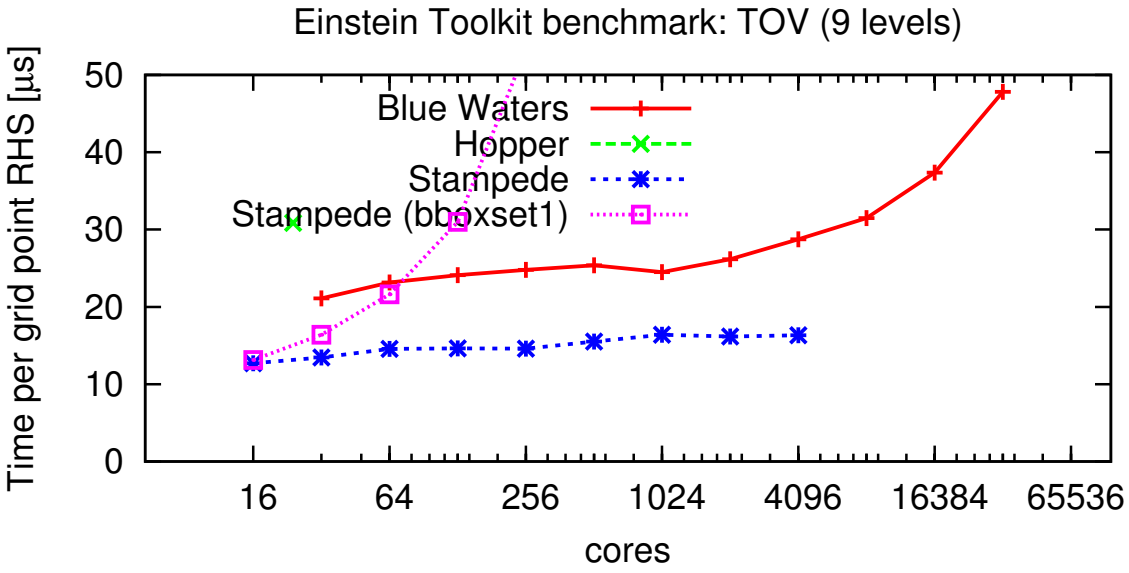


FIG. 2: Weak scaling for a simulation of a TOV star using 9 levels of refinement on Blue Waters, Hopper and Stampede. The performance is given in the time per grid point RHS evaluation (in μs). Perfect scaling would result in a horizontal line. The pink curve for Stampede is for the original regridding data structure (bboxset1), while the new implementation was used for the red (Blue Waters) and blue (Stampede) curves.

smaller load imbalances and in the cases where a perfect load imbalance is possible (2, 4, 8, 16, 32, 64, 128, 256 and 512 MPI processes) it finds it, whereas the old algorithm fails to find the optimial decomposition in several cases. The difference in performance can be even greater in the case of multi-patch runs, where we are better able to handle patches of different sizes.

C. CaKernel

With the improvements and additions to CaKernel we were able to run a demonstration binary black hole simulation where the evolution was done on the GPU and the MPI communication and several analysis routines where done on the CPU. This was done on a 3D uniform Cartesian grid. This demonstrates that the infrastructure are able to keep track of when data need to be copied from host to device and back again and that the integration with the analysis modules running on the CPU were seamless.

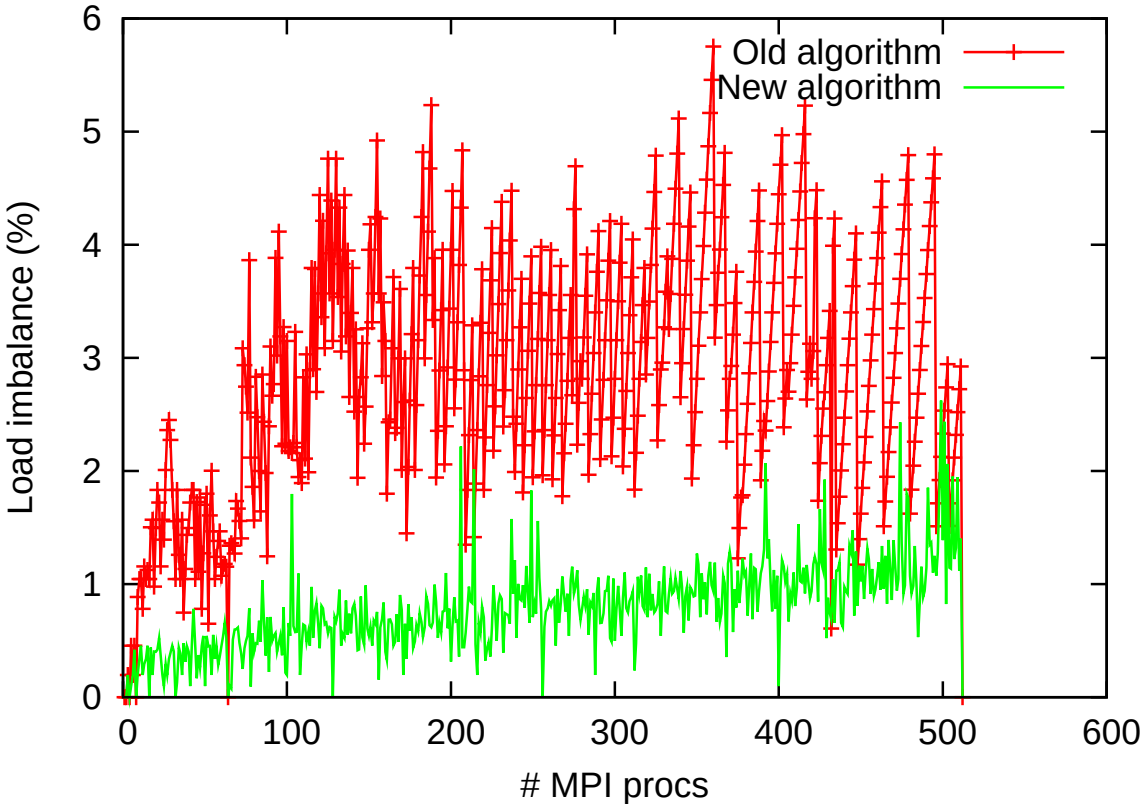


FIG. 3: A comparison of the load imbalance between the old and new algorithm for a 512^3 box split among different numbers of MPI processes up to 512. A perfect split is possible for 2, 4, 8, 16, 32, 64, 128, 256 and 512 MPI proceses. The new algorithm recovers the perfect split for all these cases and in general achieves significantly better load imbalance for most processor counts.

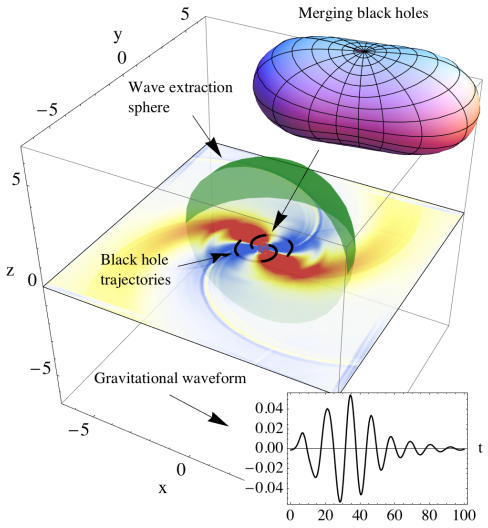


FIG. 4: The visualization of a binary black hole system.

Timer	Percentage of total evolution time
Inter-process synchronization	39%
RHS advection	13%
RHS evaluations	12%
Wait	11%
RHS derivatives	6%
Compute Psi4	5%
Multipolar decomposition	3%
File output	3%
BH tracking	3%
Time integrator data copy	2%
Horizon search	2%
Boundary condition	1%

TABLE I: The timer breakdown for the binary black hole simulation. Routines in bold face (37%) are executed on the GPU.

In figure 4 we show the numerical simulation domain. On the $x-y$ plane we project the Ψ_4 variable which represents gravitational waves. The black hole trajectories are shown as black curves near the center of the grid; they end when the black holes merge into a single black hole located at the center. The sphere on which the multipolar decomposition of the gravitational waves is performed is also shown. In the insets, we show (a) the time evolution of the (dominant) $\ell = 2, m = 2$ mode of the gravitational radiation computed on the sphere at $r = 4M$, and (b) the (highly distorted) shape of the common apparent horizon formed when the two individual black holes merge.

Table I shows a break-down of the total run time of the BBH simulation. The routines labeled in bold face run on the GPU. The times measured are averaged across all processes. The *Wait* timer measures the time processes wait on each other before an inter-processor synchronization. This encapsulates the variance across processes for the non-communicating routines. We see that the inter-process synchronization is a significant portion (39%) of the total run time. One reason for this is that the large number of ghost zones (5) needed for 8th order stencils require transmitting a large amount of data.

V. FUTURE WORK

During the project we have made considerable progress in all three sub projects.

Of the three, the improvements to the bounding box data structure has had the most immediate impact as they have been incorporated in both the development and release version of Carpet and are used routinely in production runs.

The new load balancing algorithm, though promising, suffers from the drawback that it is significantly more time consuming than the original. This is due to the fact that many more different ways of splitting the domain have to be in investigated before the best option can be chosen and in addition the algorithm is implemented serially. Thus if regridding happens too frequently, the overhead of finding a good split can offset the advantages of having a better load balance leading to longer run times. Therefore, though the algorithm has been implemented in Carpet, the current recommendation is to only use it for fixed mesh refinement runs, where the

longer time initially spent decomposing the grid does not matter. After the project finished, we have started looking at ways of parallelizing the algorithm using C++ 11 futures as a first step in getting at least a shared memory parallel version.

Though we made good progress during the project, the CaKernel developments are not yet ready for use in production mode. In a core collapse simulation the majority of the computational time is spent in the magnetohydrodynamics and neutrino transfer rather than in the space time evolution. Hence, limited benefits would result from using CaKernel for such simulations until Kranc versions of the corresponding Cactus modules have been created. Also, we would like to refine the CaKernel infrastructure to be able to automatically split and merge computational kernels to better use the different memory systems on a GPU in order to obtain the best floating points performance possible. This work is ongoing.

---

[1] *Cactus computational toolkit home page*, `http://www.cactuscode.org/`.

[2] *Mesh refinement with Carpet*, `http://www.carpetcode.org/`.

[3] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf, *The Cactus framework and toolkit: Design and applications.*, High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference, Porto, Portugal, June 26-28, 2002 (Berlin), Springer, 2003, pp. 197–227.

[4] Erik Schnetter, *Performance and optimization abstractions for large scale heterogeneous systems in the cactus/chemora framework*, CoRR **abs/1308.1343** (2013), URL `http://arxiv.org/abs/1308.1343`.

[5] Erik Schnetter, Peter Diener, Nils Dorband, and Manuel Tiglio, *A multi-block infrastructure for three-dimensional time-dependent numerical relativity*, Class. Quantum Grav. **23** (2006), S553–S578, eprint gr-qc/0602104, URL `http://stacks.iop.org/CQG/23/S553`.

[6] Erik Schnetter, Scott H. Hawley, and Ian Hawke, *Evolutions in 3D numerical relativity using fixed mesh refinement*, Class. Quantum Grav. **21** (2004), no. 6, 1465–1488, eprint gr-qc/0310042.