Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus

Gabrielle Allen[*]

Thomas Dramlitsch[*] Ian Foster[†‡] Nicholas T. Karonis[§†]

Matei Ripeanu[‡] Edward Seidel[*] Brian Toonen[†]

Abstract

Improvements in the performance of processors and networks make it both feasible and interesting to treat collections of workstations, servers, clusters, and supercomputers as integrated computational resources, or Grids. However, the highly heterogeneous and dynamic nature of such Grids can make application development difficult. Here we describe an architecture and prototype implementation for a Grid-enabled computational framework based on Cactus, the MPICH-G2 Grid-enabled message-passing library, and a variety of specialized features to support efficient execution in Grid environments. We have used this framework to perform record-setting computations in numerical relativity, running across four supercomputers and achieving scaling of 88% (1140 CPU's) and 63% (1500 CPUs). The problem size we were able to compute was about five times larger than any other previous run. Further, we introduce and demonstrate adaptive methods that automatically adjust computational parameters during run time, to increase dramatically the efficiency of a distributed Grid simulation, without modification of the application and without any knowledge of the underlying network connecting the distributed computers.

[*] Max Planck Institute for Gravitational Physics, Golm, Germany

[†] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL

[‡] Department of Computer Science, The University of Chicago, Chicago, IL

[§] Department of Computer Science, Northern Illinois University, DeKalb, IL

1 Introduction

A continued rapid evolution in both the sophistication of numerical simulation techniques and the acceptance of these techniques by scientists and engineers fostered rapid increase in demand for computing cycles. Particularly at the high end of the scale, the supercomputer centers capable of supporting the most realistic simulation studies are all grossly oversubscribed. And while commodity clusters are emerging as a promising lower-cost alternative to tightly integrated supercomputers, they seem only to be spurring further demand.

Simultaneously, capabilities of both "low-end" computers and commodity networks are increasing rapidly, to the point where a typical research or engineering institution will soon include large numbers of gigaflop/s workstations connected by gigabit/s networks, in addition to the usual collection of high-end servers and clusters. It thus becomes feasible and indeed interesting to think of the high-end computing environment as an integrated "computational Grid" [11] rather than a set of disjoint point sources. For this Grid to be widely useful, one must be able to design applications that are flexible enough to exploit various ensembles of workstations, servers, commodity clusters, and true supercomputers, matching application requirements and characteristics with Grid resources.

The need for such distributed techniques should be clear: the effective exploitation of Grid computing environments could increase dramatically the accessibility and scale of large-scale simulation. Simulations designed to include enough detailed knowledge of the physics, chemistry, engineering, and so forth, that aim to accurately represent some natural or manmade process, (e.g., collisions of neutron stars or airflow around a space shuttle) are usually both memory and throughput bound. Many fields in science and engineering (e.g., relativity, astrophysics, cosmology, turbulence, heart modeling, rocket engine design) have computational needs ultimately orders of magnitude beyond the capabilities of present machines. Indeed, the largest supercomputers typically have too little memory to carry out realistic simulations in such areas, or even if the memory are large enough for some problems, the computers are rarely available to be dedicated to a single simulation. If one had applications that could effectively harness computers at multiple sites, however, one could both attack larger problems than currently can be tackled on the largest machines and could dramatically speed throughput by enabling simulations that might saturate a large existing machine to be carried out immediately on multiple machines.

The development of such Grid-enabled applications presents a significant challenge, however, because of the high degree of heterogeneity and dynamic behavior (in architecture, mechanisms, and performance) encountered in Grid environments.

While techniques for dealing with some of these difficulties are known and have been applied successfully in some situations [3, 17, 18, 4, 19, 25], their use remains difficult, particularly for complex applications.

A promising approach to application development in such environments is to develop *Grid-enabled computational frameworks* that implement the advanced techniques referred to above, hide irrelevant complexities, and present application developers with familiar abstractions.

In the rest of this paper we present such a framework, building on two major software systems, namely, the Cactus simulation framework [2, 1] and the Globus-based MPICH-G2 implementation of MPI. By integrating these two systems and extending them to incorporate various advanced techniques for Grid execution, we have created a general framework that helps different applications become Grid enabled. We call this framework Cactus-G. Cactus-G uses standard Cactus infrastructure routines (thorns) that have been made Grid aware and uses the MPICH-G2 message-passing layer for communications. The particular scientific application we use to test this environment is, contrary to many distributed computing experiments, a *production* simulation code, used regularly to study gravitational waves from colliding black holes, that has itself not been specially modified for the Grid. The simulations are *tightly coupled*, requiring many communications on each time step. We demonstrate that even for such a demanding application, the environment and techniques we developed are quite effective. We stress that Cactus-G could be used for a large class of other applications as well.

2 The Computational Grid Environment

The computational Grid environment that we seek to harness in this work differs in many respects from "standard" parallel computing environments:

- A parallel computer is usually fairly homogeneous. In contrast, a Grid may incorporate nodes with different processor types, memory sizes, and so forth.

- A parallel computer typically has a dedicated, optimized, high bisection bandwidth communications network with a generally fixed topology. In contrast, a Grid may have a highly heterogeneous and unbalanced communication network, comprising a mix of different intramachine networks and a variety of Internet connections whose bandwidth and latency characteristics may vary greatly in time and space.

- A parallel computer typically has a stable configuration. In contrast, resource availability in a Grid can vary over time and space.

- A parallel computer typically runs a single operating system and provides basic utilities such as file system and resource manager. In contrast, a Grid environment, being a collection of single vendor machines, integrates potentially radically different operating systems and utilities.

A conventional parallel program will not run efficiently—if at all—in such a Grid environment. Even assuming that appropriate resources can be selected and reserved, and that we can reduce sheer complexity by using advanced Grid services [10, 12, 16, 26], the cited characteristics of the Grid lead to high communication latencies, low bandwidths, and unequal processor powers, which together mean that overall performance tends to be poor. Many specialized techniques can be used to overcome these problems, including the following:

1. *Irregular data distributions.* We can avoid load imbalances by using irregular data distributions that optimize overall performance. In computing these data distributions, we need information about the application itself, the target computers, and the networks that connect these computers [22, 20].

2. *Grid-aware communication schedules.* We can schedule communication (and computation) so as to maximize overlap between computation and communication, for example, by computing on the interior of a region while exchanging boundary data; we can group communications so as to increase message sizes; or we can organize data distributions or use dedicated communication processors so as to manage the number of processors that engage in intermachine communication.

3. *Redundant computation.* We can perform redundant computation to reduce communication costs. For example, increasing the size of the "ghostzone" region in a finite difference code allows us to increase communication granularity significantly, at the cost of otherwise unnecessary computation.

4. *Protocol tuning.* We can tune a particular protocol based on known characteristics of the application and environment, for example, by selecting TCP window sizes [24]; we can use specialized protocols for wide area communication taking into account application-specific knowledge; or we can compress messages prior to transmission over slow links.

3 The Grid-Enabled Cactus Toolkit

Each of the techniques listed in the preceding section is difficult to apply in an application program. Applying a collection of these techniques in a coordinated fashion, or ultimately dynamically adapting these techniques on the fly to respond to poor performance, can be extremely challenging. The Grid-enabled Cactus-G toolkit represents an attempt to solve these problems. We first describe the Cactus and MPICH-G2 systems, then describe how together they can be used for efficient computing in a Grid environment.

3.1 Cactus and MPICH-G2

Originally developed as a framework for the numerical solution of Einstein's equations [23], Cactus [5, 2, 1] has evolved into a general-purpose, open source problem solving environment that provides a unified modular and parallel computational framework for scientists and engineers. Cactus is widely used in the numerical relativity community and is becoming increasingly adopted by other fields, including astrophysics, chemical engineering, crack propagation, and climate modeling.

The name Cactus comes from its design, which features a central core (or *flesh*) that connects to application modules (or *thorns*) through an extensible interface. Thorns can implement custom-developed scientific or engineering applications, such as computational fluid dynamics, as well as a range of computational capabilities, such as data distribution and checkpointing. An expanding set of Cactus toolkit thorns provides access to many software technologies being developed in the academic research community, such as the Globus Toolkit, as described below; HDF5 parallel file I/O; the PETSc scientific computing library; adaptive mesh refinement; Web interfaces; and advanced visualization tools.

Cactus runs on many architectures, including uniprocessors, clusters, and supercomputers. Parallelism and portability are achieved by hiding features such as the MPI parallel driver layer, I/O system, and calling interface under a simple abstraction API. These layers are themselves implemented as thorns that can be interchanged and called as desired. For example, the abstraction of parallelism allows one to plug in different thorns that implement an MPI-based unigrid domain decomposition, with general ghostzone capabilities, or an adaptive mesh domain decomposer. A properly prepared scientific application thorn will work, without changes, with any of these parallel domain decomposition thorns or with others developed to take advantage of new software or hardware technologies.

The second system that contributes to our framework is MPICH-G2, an MPI implementation designed to exploit heterogeneous collections of computers. MPICH-G2 is a second-generation version of the earlier MPICH-G [9]. Like MPICH-G, MPICH-G2 exploits Globus services [10] for resource discovery, authentication, resource allocation, executable staging, startup, management, and control; it extends MPICH-G by incorporating faster communications and quality of service, among other new features.

What distinguishes MPICH-G2 from other efforts concerned with message passing in heterogeneous environments (PACX [13], MetaMPI, STAMPI [15], IMPI [6], MPIconnect [8]) are its tight integration with the popular MPICH implementation of MPI and its use of Globus mechanisms for resource allocation and security.

3.2 The Grid-Enabled Cactus Application Architecture

The Cactus system defines a set of appropriately layered abstractions and associated libraries, such that irrelevant (from a performance viewpoint) complexities are hidden from higher layers, while performance-critical features are revealed. The design of such a system must inevitably evolve over time as a result of empirical study. However, our experiences to date persuade us that the architecture illustrated in Figure 1 has some attractive properties, as we describe in the following.

At the highest level, we have a *Grid-aware application.* For the user, the abstraction presented is a high-performance numerical simulation. The user controls the behavior of the simulation by specifying initial conditions, resolution, and the like. All details of how this simulation is performed on a heterogeneous collection of computers are encapsulated within the application.

This *Grid-aware application* is built from several layers. At the top of these layers lie the various Cactus *application* thorns, which are used to perform the actual scientific calculation (e.g., solve differential equations from various branches of physics). These thorns can be written in Fortran, C, or C++, but they do not have to be Grid aware. The application programmer only needs to take care to use correct algorithms, differencing schemes, equations, and so forth. Some of these schemes will be better than others in a Grid environment, and hence a Grid-aware set of application thorns will be useful but, in principle, not required. All details of how data is distributed across processors and how communication is done is still hidden from the programmer at this level.

Next are the *Grid-aware infrastructure* thorns, providing all features, layers, and drivers that the application thorns need, namely, parallelism, I/O, Web interfaces, visualization, and many more. These thorns contain all details about communication, data mapping, parallel I/O, and the like. Application thorn writers simply

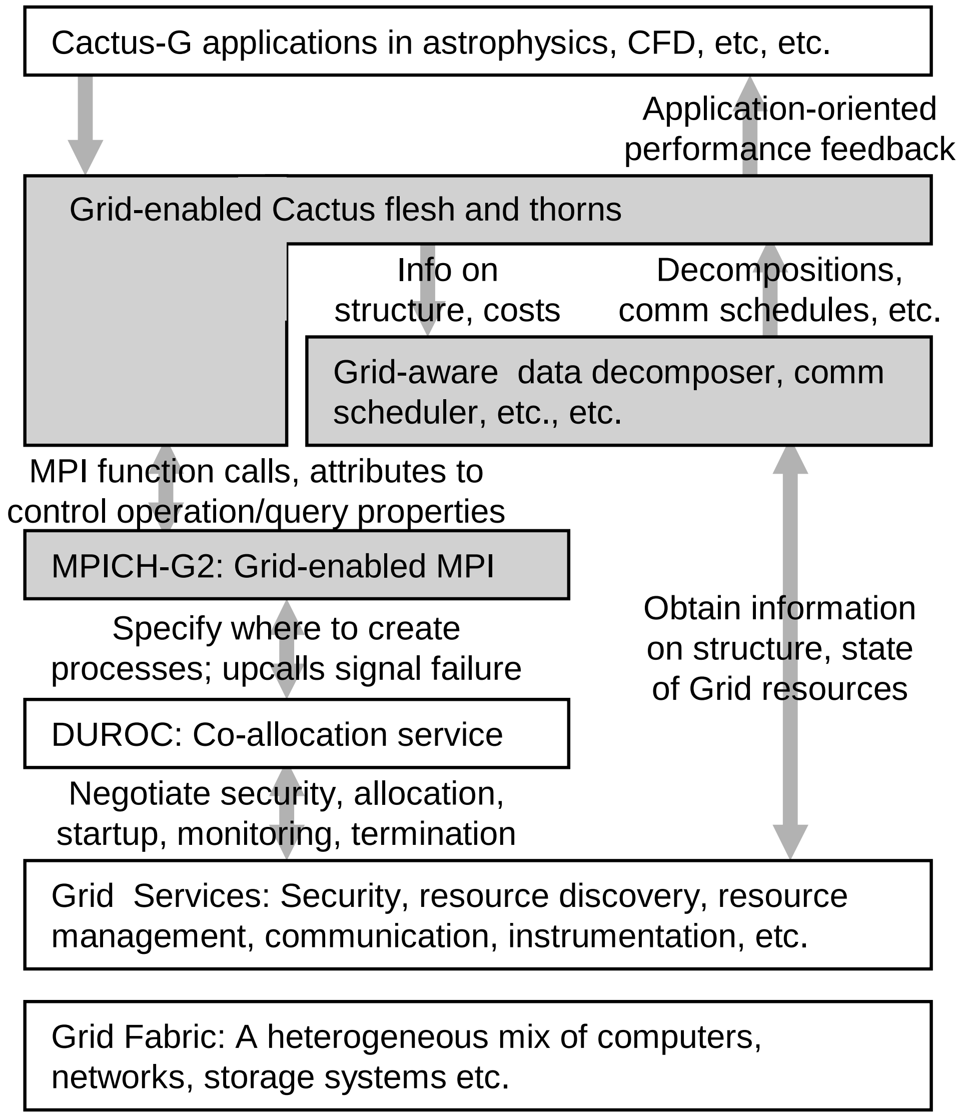
Cactus-G applications in astrophysics, CFD, etc, etc.
Application-oriented performance feedback
Grid-enabled Cactus flesh and thorns
Info on structure, costs
Decompositions, comm schedules, etc.
Grid-aware data decomposer, comm scheduler, etc., etc.
MPI function calls, attributes to control operation/query properties
MPICH-G2: Grid-enabled MPI
Obtain information on structure, state of Grid resources
Specify where to create processes; upcalls signal failure
DUROC: Co-allocation service
Negotiate security, allocation, startup, monitoring, termination
Grid Services: Security, resource discovery, resource management, communication, instrumentation, etc.
Grid Fabric: A heterogeneous mix of computers, networks, storage systems etc.

Figure 1: The Cactus architecture for the Grid, showing the information and control flows between the different layers.

include those thorns into their code (without having to modify any application thorn) depending on their needs (e.g., running single, parallel or multihost jobs). An important thorn in our case is the default Cactus driver thorn (PUGH), which provides MPI-based parallelism. This thorn has been improved and optimized for a heterogeneous Grid environment, containing features for Grid-oriented data distributions, communication schedules, and so forth.

Below this we have a *Grid-enabled communication library*: in our case, MPICH-G2. The abstraction presented is the MPI programming model; the programmer queries structure and state and controls behavior of the underlying implementation via getting and setting, respectively, attribute values associated with MPI communicators. All details of how the MPI programming model is implemented across different resources are encapsulated, including startup, monitoring, and control.

The Grid-enabled MPI implementation makes use of functions provided by a co-allocation library, in our case, the Dynamically Updated Resource Online Co-Allocator (DUROC) [7]. This library abstracts away details relating to how a set of processes are created, monitored, and managed in a coordinated fashion across different computers. The programmer can specify a set of machines on which processes are to be started; the DUROC library manages the acquisition of those resources and the creation of those processes.

Finally, a set of *Grid services* abstract away the myriad complexities of heterogeneous environments. These services provide uniform protocols and APIs for discovering, authenticating with, reserving, starting computation on, and in general managing computational, network, storage, and other resources.

3.3 Present Implementation

We have implemented substantial components of the architecture just described. In particular, we have Grid-aware thorns for Cactus that support flexible data distributions, hence enabling (for example) grid points to be mapped to processors in a heterogeneous system according to their speed and the amount of off-machine communication they have to do. We have incorporated support for message compression, hence trading off compression costs; for reduced communication costs, for variable-sized ghostzone regions, allowing message size to be increased at the cost of some redundant computation; and for a grid-aware communication schedule that allows overlapping of communication and computation. Furthermore, these techniques have been shown to work with MPICH-G2, which supports external management of TCP protocol parameters, the simultaneous use of multiple communication methods, and efficient and secure startup across multiple computers.

We have built a model to evaluate the expected performance in a Grid environment [21]. The model is parameterized with execution environment data (number and performance of processors, network performance) and application data (problem size, ghostzone size, etc.), allowing us to quantify the sensitivity of overall performance to these parameters.

As described in Section 6, we have also recently implemented a prototype *adaptive* mechanism that is capable of varying the compressions and ghostzone parameters during run time, monitoring the effect they have on performance. In this way, performance can be automatically optimized, without needing to know the network characteristics, and without the application itself needing to be modified in any way.

4 Experimental Results: Distributed Terascale Computing

To evaluate real-world applications using Cactus-G on a Grid, we performed a distributed run of a scientific Fortran application across four supercomputers. We emphasize that the application code itself remained unchanged: all adaptations and improvements were carried out within the Cactus infrastructure. In this test, we chose parameters that allowed dramatically improved performance over the standard implementation, but these remained fixed during the test. In Section 6 we introduce *adaptive* mechanisms that seek optimal communication parameters while the code is running, and we demonstrate their effectiveness in test simulations.

4.1 Code Characteristics

Our application is a standard code for solving problems in numerical relativity: the model used here solves Einstein's equations for the evolution of a gravitational wave spacetimes. The techniques we have developed can be used with any application, however, from black holes to chemical engineering. This application exemplifies many large scale computing problems: in three-dimensional Einstein's theory contains dozens of coupled, nonlinear hyperbolic and elliptic PDEs, with thousands of terms, and is both memory and compute bound.

The code uses finite differencing techniques and functions discretized on a regular grid. In the specific case under consideration, there are 57 three-dimensional variables, with approximately 780 floating-point operations per grid point at each iteration. The finite difference method uses a computational stencil of width one to calculate partial derivatives, and normally a ghostzone size of one is used, with six variables needing to be synchronized in each direction at each iteration. The default data type is eight bytes, so that a local processor grid size of N^3 implies sending $N^2 \times$ 6 variables $\times$ ghostzone_size $\times$ 8 bytes in each direction.

4.2 Experimental Configuration: Machines and Network

Our testbed included four supercomputers at two sites: the National Center for Supercomputing Applications (NCSA) in Champaign-Urbana, Illinois, and the San Diego Supercomputing Center (SDSC) in California. This set already includes all major challenges and problems: heterogeneity, unstable and/or slow network, load balancing, multiple sites, multiple authentication schemes, multiple schedulers, and the like.

The machines involved in this run were a 1024-CPU IBM Power-SP at SDSC (*Blue Horizon*) and a 256-CPU (*Balder*) SGI Origin2000 and two 128-CPU (*Forseti1* and *Aegir*) SGI Origin2000 systems at NCSA. The Origins each have two 250 MHz R10000 CPUs per node, and Blue Horizon has eight 350 MHz Power-3 CPUs per node. The machines could allocate 512 MB of memory per CPU. Single-processor performance of our application on the two platforms was measured (using `perfex` on Irix and `hpmcount` on AIX) to be 306 MFlop/s on the SDSC SP2 and 168 MFlops/s on the NCSA Origins.

All machines had a high-speed interprocessor interconnection switch providing O(200 MB/s) intramachine communication bandwidth. Intermachine communication performance varied. Machines at NCSA were locally connected using (almost dedicated, no competing traffic) Gigabit Ethernets achieving up to 100 MB/s—comparable to intramachine communication performance.

Between SDSC and NCSA, we achieved a measured application-level bandwidth of at most 3 MB/s, as measured by the `mpptest` program [14], over what is actually an OC12 (622 Mb/s) network. We have not yet determined the reason for this low performance. Even with such poor performance, the various techniques described below allowed us achieve efficiency of up to 88% on runs across the two sites for this tightly coupled simulation.

4.3 Processor Topology and Load Balancing

We used a total of 120+120+240=480 CPUs on the NCSA machines and 1020 CPUs at SDSC, giving a total of 1500 CPUs. These processors were organized in a $5 \times 12 \times 25$ three-dimensional logical mesh, which was decomposed over machines in the z (long) dimension such that all x- and y-directional communication occurred

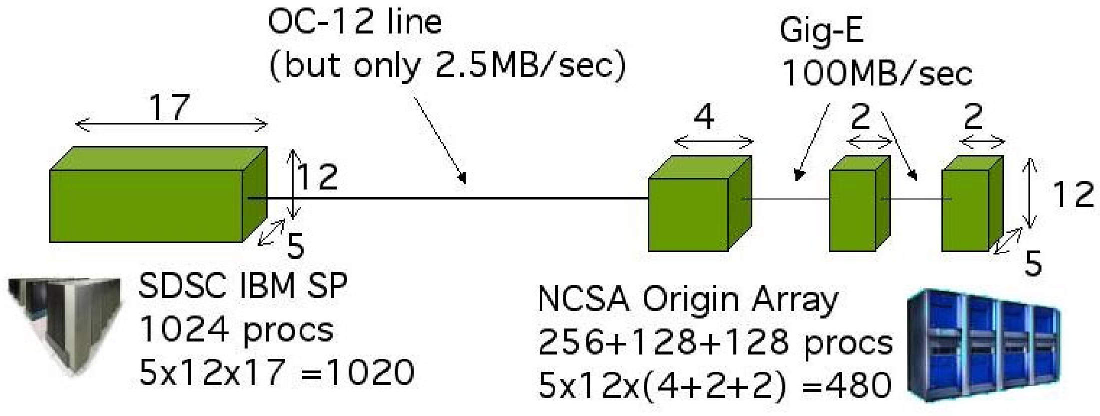

Figure 2: Processor topology across machines: Aegir [$5 \times 12 \times 2 = 120$], Forseti [$5 \times 12 \times 2{=}120$], Balder [$5 \times 12 \times 4{=}240$], and Blue Horizon [$5 \times 12 \times 17{=}1020$].

entirely within a single machine. With this organization, the number of processors communicating over the WAN is $2 \times 5 \times 12 = 120$.

The total number of gridpoints used for the calculation was about 7.2×10^8 ($360 \times 720 \times 3345$ gridpoints). This provided a simulation over five times larger than we have been able to perform previously. Distributing gridpoints across the processors, we account for the fact that the Power-3 processor provides almost twice the performance of the R10000 processor for our application, leading us to choose a load-balanced local grid of approximately $70 \times 60 \times 95$ on each Origin CPU and $70 \times 60 \times 155$ on each SP processor. With this distribution, a single iteration on any processor took around 1.7 seconds.

4.4 Communication Optimizations

In this section we briefly introduce techniques we used to improve application's performance. In particular, we overlapped communications with computation to remove communication bottlenecks; added extra ghostzones to reduce latency effects; and compressed data before transferring it across the WAN to solve bandwidth constraints. In Section 5 we present a detailed costs/benefits analysis of these techniques.

Computation/Communication Overlap. Because of the relatively slow WAN link the 120 "boundary" processors at SDSC and NCSA performing WAN communication will run slower than the others. We addressed this problem by redistributing gridpoints so that the WAN communicating processors had fewer grid points and could overlap communication with computation on the other processors. In this experiment, the internal NCSA processors were given 20% more, and at SDSC 23% more gridpoints than the WAN communicating processors.

Compression. We incorporated into Cactus a thorn that compresses messages prior to transmission. Since our simulation data are smooth, as they should be for well-resolved physics simulation, high compression rates (sometimes as high as 99%) are achieved. With compression in place, the amount of data to be sent is greatly reduced, and latency becomes the bottleneck.

Ghostzones. Larger ghostzones lead to an increase in communication granularity at the cost of replicated computation and increased memory usage. This reduces the number of messages (and hence the costs related to communication latency) while the total amount of data exchanged remains constant.

4.5 Observed Performance

We gauge achieved performance using two metrics: Flop/s rate and efficiency. We compute these based on three factors:

- **Total execution time t_{tot}.** The elapsed wallclock time the code needs to be executed, measured by simply calling MPI_Wtime to get the elapsed time as the code executes. To get a more fine-grained idea, we measure elapsed time every 10 iterations (i.e., after every sync) and take the maximum across all processors.

- **Expected computation time t_{comp}.** Defined as the ideal time needed for the processors to actually do calculations (i.e., floating-point operations), computed from measurements on a single, dedicated processor.

- **Flop count F.** For the optimized code we measured, using hardware counters on various machines, 780 Flops per gridpoint per iteration.

We define Flop/s rate and efficiency as

$$Flop/s\_rate = \frac{F * number\_of\_gridpoints * number\_of\_iter}{t_{tot}} \quad (1)$$

$$E = \frac{t_{comp}}{t_{tot}}. \quad (2)$$

First we executed a distributed computation across the four supercomputers without the benefit of the compression and ghostzone techniques we developed. We achieved only 42 GFlop/s and a scaling efficiency of only 14%. This is less than can be achieved on Blue Horizon alone. The only gain in this case is problem size, which is five times larger than has ever been run, but the efficiency is poor.

We ran the same simulation but with compression to compensate for bandwidth limitations and with 10 ghostzones to compensate for the latency. We found dramatic improvements, increasing our total Flop/s rate to 249 GFlop/s. As mentioned above, in single-processor runs, our application achieves 168 and 306 MFlop/s on R10000 and 306 Power-3, respectively.

To compute efficiency, we first measured the theoretical "peak speed" of $480 * 168 + 1020 * 306$ MFlop/s = 393 GFlop/s. The actual performance of 249 GFlop/s, divided by the theoretical peak performance of 393 GFlop/s, gives us an efficiency of 63.3%. (An alternative computation, which uses the average rather than the maximum time measured across all processors, gives a higher efficiency of 73.8%.) Note that computing efficiency as $\frac{t_{comp}}{t_{tot}}$ leads to the same value for efficiency.

We also performed a smaller run including only 120+1020=1140 processors, which achieved 292 GFlop/s, with an efficiency of 88%. In this case, it was not clear why the efficiency was better, as the experiments could not be systematically repeated. However, small changes in the domain decomposition, processor topology, and network characteristics at the time of the experiments could easily account for this. The number of CPUs communicating over the WAN remains the same (120). In Section 6 we present dynamic techniques to optimize, or at least improve, performance under such conditions.

Note that the performance numbers quoted here include the redundant calculations performed in the ten ghostzones used for communications, especially in the z-directions for the non-WAN adjacent processors. Although the effective physical problem size was smaller because of these ghostzones, later revisions of the framework will not use them, and the ghostzones will correspond directly to "real" gridpoints. Also, during the first few iterations the observed performance was poor as a result of a slow distributed MPI_Init, and this effect was not included in these results. After all processors left the barrier, the peak performance was achieved. After these experiments were performed, this MPI_Init issue was solved.

5 Ghostzones and Compression

In the preceding section we presented experimental results we obtained from a large distributed calculation. Although those results are good in terms of performance and efficiency, most of the decisions about various parameters, mainly compression and ghostzone size, were done with rough estimates under "battleground" conditions. In this section, completed after the big run described above, we investigate more closely the effect of ghostzone size and compression on performance. Such information will help us know more precisely when, for example, compression makes sense and what is the "optimal ghostzone size".

5.1 Ghostzones

Message transfer costs have two components: bandwidth and latency-related costs. Supercomputers have negligible interprocessor latencies of a few microseconds. For communications across the WANs, however, latencies go up to tens of milliseconds. Since we are using TCP for wide area communication, latency effects are exacerbated by TCP's slow start: TCP might need up to a dozen RTTs to start sending at full speed. As discussed above, to reduce latency-related overhead, we increase the ghostzone size. This reduces the number of messages sent across the network while keeping the total amount of data transferred.

In Figure 3 we illustrate the performance tradeoffs of applying multiple ghostzones to reduce latency costs. Since it is not possible to "steer" the latency of a network connection, we simulated it on a single machine. In a two-processor run on a single Origin we placed a loop before every MPI_Send, which waited a defined time, simulating communication latency. Since the bandwidth was around 90MB/s (intramachine bandwidth on a SGI Origin), we are certain to exclude bandwidth effects.

Figure 3 (left) shows that latency-associated costs can have a significant impact on overall efficiency (when no extra ghostzones are used). In the experiment described above we determine empirically the number of ghostzones that maximize efficiency (Figure 3, right). Increasing the number of ghostzones further does not give any benefit; it only wastes memory resources. For the same latency and application as in the our large-scale run, we found that a ghostzone size of 4 results

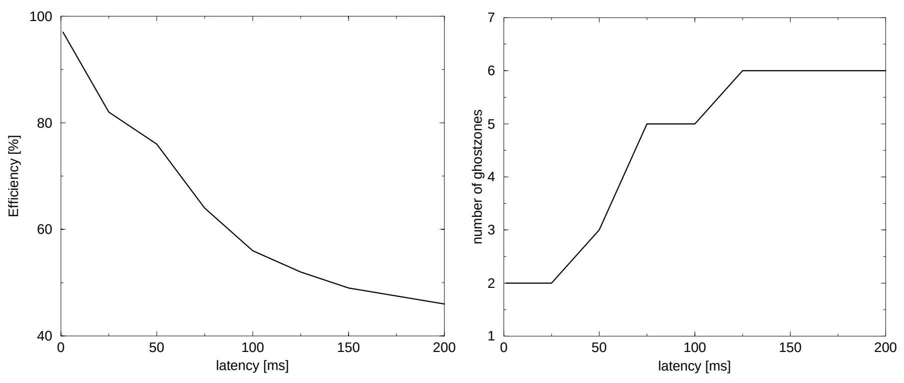

Figure 3: The left graph shows latency effects on efficiency. For our application we simulate different latencies (see text). Efficiency goes rapidly down as latency increases. The values around 75 ms represent approximately the situation between SDSC and NCSA. At that point the efficiency drops to 65%. The right graph shows the "optimal" number of ghostzones plotted against the latency. As expected the number of ghostzones goes up with the latency. According to this figure, the optimal size for a latency of 75 ms is 4.

in maximum efficiency of about 92% (from 65% when using a single ghostzone). In our big run we used a ghostzone size of 10. Although the conditions were different (additional bandwidth problems, compression, etc.) this result indicates that our ghostzone size has been larger than needed. In any case, we stress that ideally one would like to adjust parameters phenomenologically, adapting to the current application problem and network (as shown in Section 6).

5.2 Compression

As mentioned above, one of the major hindrances for achieving a decent performance of a tightly coupled scientific code in a metacomputing environment is low throughput for data transfers over WANs. Since data sent over the WAN has a "regular" structure (numerical floating-point data of smooth functions) we suspected a big benefit in terms of overall efficiency when compressing messages before sending them across the network. In this section we investigate compression rates and speed, and we emphasize the benefits of this technique.

We have studied this issue extensively, by applying a standard Unix compression routine (`compress()` routine of `libz` library) to a vast array of floating-point data coming from simulations of black holes, gravitational waves, and other Cactus applications. We measured compression characteristics for different resolutions and domain sizes.

For the simulations performed in the big run, we found compression ratios of more than 95%. In nearly all cases studied, we found compression to be surprisingly effective, usually much better than 50%. We attribute this to the fact that data being simulated should be smooth. If a sufficiently fine grid is used to resolve the data, neighboring data values will differ little from each other. This is normally the case; and even with shocks common in CFD, we expect large changes in the data fields to limited to the shock itself. In any case, we have seen favorable compression ratios achieved for nearly all systems tested.

We stress that the compression achieved will change during the evolution of the system being simulated. During an evolution, smoothness of data will change, and the effect of compression will change. Hence, we would like to have adaptive algorithms that adjust compression parameters on the fly. In Section 6 we demonstrate a crude but effective example of such an algorithm.

Although the compression ratios we have seen so far give rise to the speculation that compression is always helpful, we must also consider the compression time.

We define total communication time as

$$t_{total} = t_{compr} + \frac{msg\_size \times compr\_factor}{bandwidth}$$

to reflect its two components the compression time t_{cmpr} and the (reduced) message transfer time. For a given message size of 200 KB (as in the big run) in Figure 4 we plot the communication times against the bandwidth. The thick solid line represents the communication times without compression. For a specific Einstein equation application under study, a network bandwidth of 2 MB/s is already good enough not to need compression for a single black hole simulation, while for a wave pulse *evolved by the same application* it still makes sense even with bandwidth of 10 MB/s. This analysis shows that the results are problem dependent, even within a single application that can evolve different initial data sets. Hence, adaptive algorithms that can adjust to local network and application characteristics are valuable. We treat this subject in the next section.

6 Adaptive Strategies

As discussed in the preceding section an adequate choice of the number of ghostzones as well as the decision about whether to compress data substantially influences overall performance. Furthermore, as real network latency and bandwidth characteristics

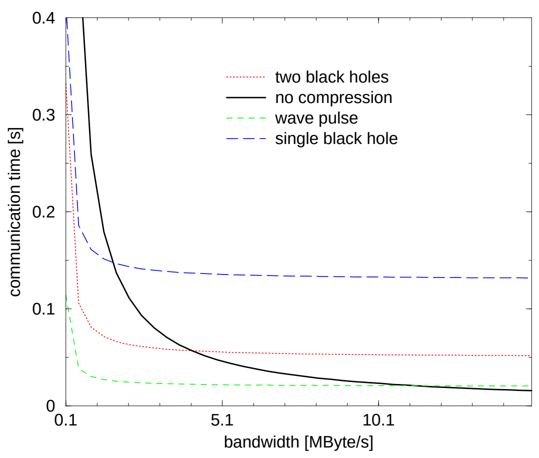

Figure 4: **Communication time depending on bandwidth and compression.** The solid line represents the communication time when no compression occurs. The intersections of this line with the other lines mark the point where compression does not make sense anymore since the time to compress is higher than one saves due to reduced message size.

may change dramatically over time, a good choice of compression/ghostzone parameters at one instant may become a poor choice at a later time. Even sophisticated models to predict optimal parameters, given detailed knowledge of network characteristics, would have to be rerun every time the network characteristics change. Hence, we need a phenomenological, adaptive mechanism that can adjust ghostzone, compression, and other communication parameters *on the fly* to optimize code performance *without any knowledge* of theoretical bandwidth and latency characteristics of a given network. Such knowledge is usually going to be poorly correlated with actual network performance at any given time.

In this section we introduce a simple set of methods, which we have implemented and tested in Cactus, that can improve the performance of a distributed simulation without any *a priori* knowledge of either the application or the type or performance of the underlying network characteristics. Furthermore, the application running in Cactus does not need to have any knowledge of these methods or network characteristics, and hence neither does the scientist or engineer running the code. The distributed application begins with a certain set of default communication parameters (e.g., in this case, the number of ghostzones and compression state), and these are adjusted *automatically* until the code becomes "optimally" efficient. In what follows, we give examples where the efficiency of a distributed Einstein equation simulation improved from about 68% to 85%, without any hand tuning, user intervention, and any assumptions about or knowledge of the underlying network being used.

In our big experiment described earlier, we chose a ghostzone size of 10, and we applied compression in the z-direction. This decision was based on a few preliminary tests and a bit of guesswork at the time of the experiment. As we further develop our adaptive algorithms, we will test them on such large-scale simulations. In the rest of this section we present an encouraging initial assessment of these algorithms, using only two processors to prove the concepts.

6.1 Adaptive Compression

We implement runtime-adaptive compression as follows. Every processor decides *independently* whether it wants to send compressed messages in a certain direction. The peer processor assumes that if the sizes of arriving messages are smaller than expected, they are compressed. To decide whether to compress, each processor uses a local estimate of efficiency. Our first attempt to provide an adaptive algorithm for compression is crude but effective: after a number of iterations the Cactus code changes the compression state (for now, this means simply switching on/off) and monitors the efficiency. If the efficiency improves, this choice remains; otherwise the old choice is used. To get to a reliable decision, the efficiency is always monitored and averaged over N iterations. In test cases where the compression rate and network bandwidth took extreme values (intramachine network, cross-Atlantic network, etc.), the code always made the “right” decision. This routine can be called every 500 timesteps, since network quality in a wide area Internet can change rapidly and numerical data can get noisy (uncompressible). Figure 5 shows an example of the improvement in performance after compression was automatically turned on.

6.2 Adaptive Selection of Ghostzone Sizes

Unlike compression, implementing an adaptive algorithm to dynamically modify ghostzone sizes is rather difficult and complex to code for several reasons. First, ghostzone sizes cannot be chosen independently. If a processor decides to change its ghostsize in one direction, then it at least has to inform its neighbor in that direction, which has to modify its ghostzone size in turn. Changing the ghostzone size also requires a memory reallocation of all involved gridfunctions and data. If the ghostzone size increases, it has to be filled with valid data, which has to be sent

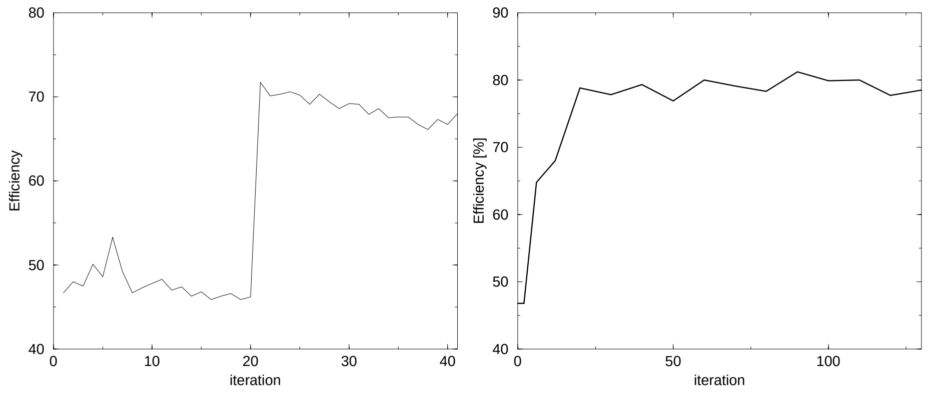

Figure 5: Adaptive Techniques. On the left we show the effect of compression on message sizes for the same code we ran in our big experiment, but on just two workstations connected by a fast Ethernet LAN and with a problem size of 80^3 per machine. On the right, we show how the adaptive ghostzone technique increases efficiency for a test latency of 75 ms. The code starts with a ghostsize of 1, resulting in less than 50% efficiency. After the third iteration the ghostsize changes to 2, improving the performance. The process continues until maximum efficiency is achieved. Changing the ghostzone number can be time consuming and has been excluded in the analysis.

by the neighboring processor. Further, although at present there are only two states of compression (on/off), there is more than one possible ghostzone size.

We have tested an implementation of our adaptive ghostzone technique. Initially, the code starts with a ghostzone size of 1, as would be the norm in most codes. After some iterations it increases to 2 and compares efficiency. If efficiency improves, the ghostzone size keeps increasing. If the performance goes down or stays about the same (this means here $\pm 20\%$) the ghostsize decrements by one and (for now) sticks to that value. Another possibility is that if the *neighboring* processor measures that the installation of an extra ghostzone does not significantly improve efficiency, it sends a message to its neighbor, and both finish increasing the ghostzone size.

In Fig. 5 we illustrate the effect of adaptive ghostzones for a test with a fixed latency of 75 ms, simulated on a single Origin2000, by placing a time loop before every MPI_Send in the code as described above. The application again involves solving Einstein's equations but with an iterative Crank-Nicholson evolution scheme used in black hole calculations, which performs 9 syncs per time step and is hence more sensitive to latency effects. (We stress that Einstein application and adaptive techniques have no knowledge of each other.) At the beginning the efficiency starts with a poor value of under 50%. Our algorithm increases successively the ghostzone size to a value of 4, achieving an efficiency of more than 80%.

Next, we examine the effect of varying latencies in a controlled environment. We again use a grid-unaware numerical relativity application code similar to that used above, and run our tests on two R10000 processors located on the same machine but communicating over a 100 Mbit Fast Ethernet interface. The bandwidth of this interface was determined to be about 30 MB/s (measured by `mpptest`), and the latency was below 1 ms, much lower than the latencies we simulate in these experiments.

Figure 6 shows two curves. The dotted line, similar to the one in Fig. 3, shows that efficiency goes down with latency almost linearly when no adaptive mechanisms are used. The solid line shows achieved efficiency when using adaptive strategies applied. Although this curve also tends to go down with increasing latency, the difference from the original one is remarkable. Note that the efficiency is higher in the adpative ghostzone case than the fixed one even at “zero latency”. This is because, even with the internal latency of an Origin2000, “two ghostzones are better than one” for this application, an unexpected result. Figure 6 (right) shows the ghostsize our algorithm picked for the appropriate latency. Naturally, it gets higher as latency increases.

Finally, we move from test cases to a real numerical relativity simulation run across two machines, with both adaptive compression and ghostzones enabled. The

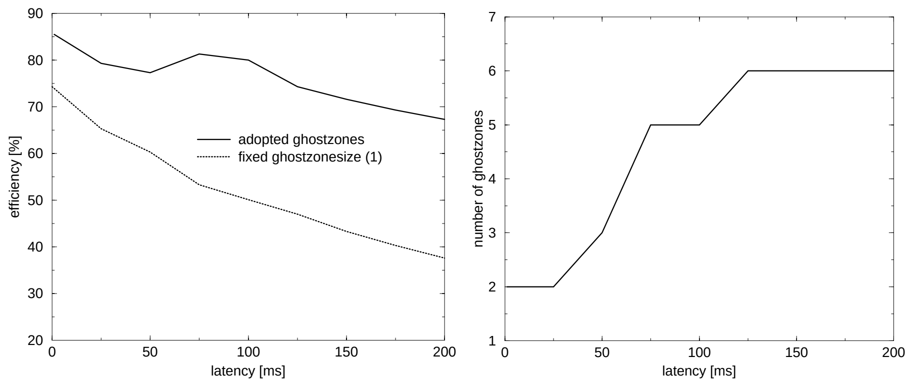

Figure 6: Adaptive ghostzones. Left: efficiency achieved using fixed ghostsizes (dotted line) and with adaptive ghostsizes (solid line). While the former drops relatively fast the latter achieves an efficiency of more than 80% even at high latencies. (right) The ghostzone size chosen by our algorithm, which increases with increasing latency.

code initially starts with one ghostzone and no compression. After a hundred iterations it increases the ghostsize to two and therefore communicates only every second iteration. As we shown in Fig. 7, since this code sends many small messages per iteration, it gives an improvement. After another 200 iterations the code increases the size to 3, then to 4 and 5. At the ghostsize of 5 it decides that the improvement is not significant and switches back to 4, where it settles down. Some 100 iterations later compression is switched on, which boosts the performance to almost 85%. Without those adaptations the efficiency would stay where it was at the beginning, below 70%.

7 Summary and Future Plans

The heterogeneous, dynamic, and unreliable nature of Grid environments has major implications for application development. In this paper, we describe the architecture and current prototype status of a Grid-enabling computational framework, Cactus, designed both to hide irrelevant Grid-related complexities and to present application developers with familiar abstractions. As a first step we have developed Grid-aware infrastructure routines implementing advanced techniques that are completely transparent to the user. These routines will be part of the standard Cactus distribution, making Grid computing trivially available to any Cactus application and user.

We also present the results of large-scale experiments (using 1500 processors

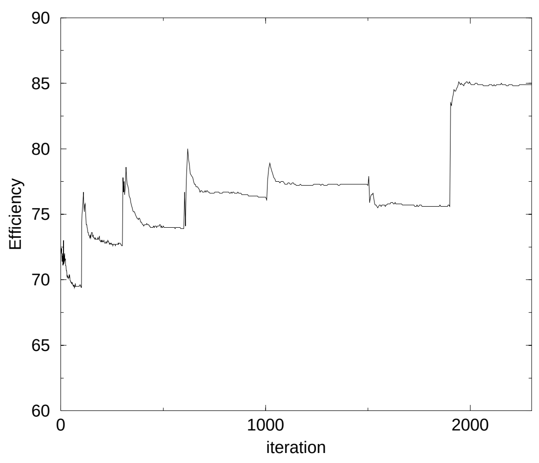

Figure 7: Automatic performance improvement. Ghostzones start to increase as the evolution of the code goes along. At a later stage, compression is automatically switched on. This gives us an performance gain of more than 15% in this case. Note that on a single supercomputer the efficiency of this code would be 99%.

across four separate machines) in numerical relativity that demonstrate convincingly that the Cactus framework is able to support efficient execution in a heterogeneous Grid. Introducing special communication and latency reducing techniques, we were able to increase the efficiency of a transcontinental distributed simulation between NCSA and SDSC more than fivefold, from a low of 14% without techniques, to over 80% with them. These techniques were independent of the application code itself, and many application codes could immediately take advantage of these techniques.

We then introduced and demonstrated *on-the-fly* runtime adaptive techniques and showed that under many conditions these can increase dramatically the efficiency of distributed simulations by hiding latencies and reducing bandwidth requirements. In a simple implementation, we showed that the algorithms can increase efficiency for different applications that have no knowledge of the fact they are running in a distributed environment. Further, the adaptive techniques themselves are *phenonmenological* and assume no knowledge of the underlying networks connecting the distributed resources.

A future goal is to improve and fully implement the adaptive algorithms presented here for large scale distributed applications, such as the record-breaking numerical relativity simulations we demonstrated. Our tests indicate that our adaptive techniques should *automatically* find communication parameters that optimize performance, such as those that were found by hand to increase the efficiency of the "big run" from 14% to over 80%. We also plan to extend this work to include load balancing, which also needs to become dynamic. Here we hope to exploit techniques being developed by other participants in the Grid Application Development Software (GrADS) project.

Acknowledgments

It is a pleasure to acknowledge the support of the staff at NCSA and SDSC, who made these experiments possible, especially Jon Dugan, Wayne Louis Hoyenga, Doru Marcusiu, Tony Rimovsky, John Towns at NCSA, and Phil Andrews, Giri Chukkapali, Larry Diegel, Eva Hocks, Keith Thompson at SDSC. Also, we would like to thank Tom Goodale, Gerd Lanfermann, and Thomas Radke from the Cactus Team.

This material is based in part upon work supported by the National Science Foundation under Grant No. 9975020.

References

[1] G. Allen, W. Benger, C. Hege, J. Massó, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Solving einstein's equations on supercomputers. *IEEE Computer*, 32(12), 1999.

[2] G. Allen, T. Goodale, and E. Seidel. The cactus computational collaboratory: Enabling technologies for relativistic astrophysics, and a toolkit for solving pdes by communities in science and engineering. In *7th Symposium on the Frontiers of Massively Parallel Computation-Frontiers 99*, New York, 1999. IEEE.

[3] F. Berman. High-performance schedulers. In *[11]*, pages 279–309.

[4] S. Brunett, D. Davis, T. Gottschalk, P. Messina, and C. Kesselman. Implementing distributed synthetic forces simulations in metacomputing environments. In *Proceedings of the Heterogeneous Computing Workshop*, pages 29–42. IEEE Computer Society Press, 1998.

[5] *http://www.cactuscode.org.*

[6] IMPI Steering Committee. IMPI - interoperable message-passing interface, 1998. `http://impi.nist.gov/IMPI/`.

[7] K. Czajkowski, I. Foster, and C. Kesselman. Co-allocation services for computational grids. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1999.

[8] G. E. Fagg, K. S. London, and J. J. Dongarra. MPI_Connect managing heterogeneous MPI applications inter operation and process control. In V. Alexandrov and J. Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 93–96. Springer, 1998. 5th European PVM/MPI Users' Group Meeting.

[9] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98*. ACM Press, 1998.

[10] I. Foster and C. Kesselman. Globus: A toolkit-based grid architecture. In *[11]*, pages 259–278.

[11] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

[13] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogenous computing environment. In *Proc. EuroPVMMPI'98*. 1998.

[14] W. Gropp and E. Lusk. Reproducible measurements of mpi performance characteristics. `http://www-unix.mcs.anl.gov/g̃ropp/papers.htm`.

[15] T. Kimura and H. Takemiya. Local area metacomputing for multidisciplinary problems: A case study for fluid/structure coupled simulation. In *Proc. Intl. Conf. on Supercomputing*, pages 145–156. 1998.

[16] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[17] Paul Messina. Distributed supercomputing applications. In *[11]*, pages 55–73.

[18] J. Nieplocha and R. Harrison. Shared memory NUMA programming on the I-WAY. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 432–441. IEEE Computer Society Press, 1996.

[19] P. M. Papadopoulos and G. A. Geist. Wide-area ATM networking for large-scale MPPS. In *SIAM conference on Parallel Processing and Scientific Computing*, 1997.

[20] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. Technical Report CS-TR-3055, Department of Computer Science, University of Maryland, 1993.

[21] M. Ripeanu, A. Iamnitchi, and I. Foster. Cactus application: Performance predictions in grid environments. *In proceedings of EuroPar 2001 Conference, LNCS 2150*, 2001.

[22] J. Saltz and M. Chen. Automated problem mapping: The crystal runtime system. In *Proceedings of the Second Hypercube Microprocessors Conference*, Knoxville, TN, September 1986.

[23] E. Seidel and W. Suen. Numerical relativity as a tool for computational astrophysics. *J. Comp. Appl. Math.*, 109(1-2):493–525, 1999.

[24] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. *Computer Communication Review*, 28(4), 1998.

[25] T. Sheehan, W. Shelton, T. Pratt, P. Papadopoulos, P. LoCascio, and T. Dunigan. Locally self consistent multiple scattering method in a geographically distributed linked MPP environment. *Parallel Computing*, 24, 1998.

[26] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, Portland, Oregon, 1997. IEEE Press.