

Grid Aware Parallelizing Algorithms

Thomas Dramlitsch,* Gabrielle Allen,* Ed Seidel*[†]

April 8, 2002

Abstract

Running tightly coupled parallel MPI applications in a real grid environment using distributed MPI implementations [1, 2] can, in principle, make better and more flexible use of computational resources, but for most parallel applications it has a major downside: The performance of such codes tends to be very poor. Most often the natural characteristics of realworld grids are responsible for this behavior, for example rapidly changing and unpredictable network quality, heterogeneity and load imbalance. Worse, in most cases the parallelized code itself does not take this grid nature into account, having been parallelized for single machine execution. We have already shown that it is possible to improve the efficiency of such codes by manually adapting the parallel layout to the given grid environment [3]. Based on these results we now present a new set of algorithms for parallel codes running within the Cactus framework [4], which *automatically* adapt to the grid environment. We show how these algorithms can be implemented in an abstract and flexible manner so that a large family of application codes can benefit without modifications, or even any knowledge, of the underlying parallel layer or computational resources. The algorithms determine the optimal processor topology and optimal load balance, minimize communication over the WAN and apply latency hiding and compression techniques where needed. Those adaptations, which as part of an MPI driver layer are hidden from both the Cactus end user and application programmer, are performed *automatically* and *dynamically* during code execution in order to attempt to meet the requirements of the changing nature of the Grid. Largescale test runs with real world applications have shown performance improvements of 200-300%, importantly with no changes required in the application code itself.

*Max Planck Institute for Gravitational Physics, Golm, Germany

[†]National Center for Supercomputing Applications, Illinois, USA

1 Introduction

The increasing maturity of grid computing is opening a door to many new opportunities and possibilities for how application codes can better make use of the resources available to them [5, 6]. New application driven projects, such as GridLab, GriPhyn and DataGrid [7] are working together with infrastructure developers to develop technologies such as schedulers, monitoring and information systems, resource brokers, data management schemes and user portals. Such technologies promise to enable the users of today's applications with increased accessibility to resources, data and information, simplifying many of the difficulties which currently restrict supercomputing to the domain of a few hardy scientists and engineers.

Even more exciting is the inspiration grid computing is providing towards the development of fundamentally new scenarios for applications running in a grid environment. The European GridLab project will provide generic APIs and tools which general applications can easily make use of to, among many other scenarios, migrate entire simulations between machines, spawn subtasks to less loaded or less expensive resources, and perform intelligent parameter surveys. New collaborative tools will provide access to, information about, and visualization of simulations to whole groups of researchers while the codes are running, providing interactive control and monitoring. New messaging interfaces will inform users about important events and problems with their jobs wherever they are located, using emails, mobile phones, PDAs, and web browsers.

One of the key requirements for these new applications is that they no longer interact with resources as a set of individual machines, networks and data devices, but instead view all these components as one large virtual supercomputer. The grid services which the applications use should have the intelligence to make use of the most appropriate components or sub-components of this virtual machine, considering factors such as code requirements, performance and cost.

In this paper we describe techniques which enable a small piece of this vision, allowing tightly coupled MPI applications to run efficiently across multiple computers, connected together by arbitrary networks. This work has three driving motivations: Firstly, today's large scale applications often require more resources than are available on a single machine. Making use of larger virtual machines will enable otherwise impossible simulations. Second, even when an application can be run on a multiple processors of a single machine, it may still be beneficial to use different components from the virtual supercomputer avoiding long or expensive queues to obtain a quicker or cheaper solution. Finally, the development of the flexible, adaptive and dynamic techniques described here is a step towards the goal of enabling applications to be able to take full advantage of the emerging grid software.

2 Distributed Simulations

For some years, it has been possible to run parallel codes across more than one supercomputer or cluster (see, e.g., [8]). However, even with the introduction of grid enabled MPI implementations such as [1, 2], most tightly coupled, communication intensive codes will achieve very poor performance [9] once the transfer of data across a WAN is introduced.

The main reason for this suboptimal performance, aside from load imbalance resulting from processor inhomogeneity, is the fact that between *some* processors there is a network with a comparatively high latency and low bandwidth. Typical real-world numbers, even for high performance research networks, are a latency of 20 to 200ms for a wide area network link and a bandwidth of between 0.5 and 5 MB/s. For a supercomputer or tightly coupled cluster those numbers are 2 to 4 orders of magnitude better, namely around $10\mu s$ and 120MB/s respectively. Since many parallel simulation codes need data synchronization across processors at each iteration, it is clear that unless special care is taken, a slow network between any *one* pair of processors is enough to arbitrarily degrade the code's efficiency.

The family of codes we have considered in this paper are finite differencing codes that compute functions discretized on a regular grid. This is typical of many codes in astrophysics, relativity, fluid dynamics, etc. The communication scheme of those codes arises from the need to synchronize data, that is, to send and receive the discrete function values at "grid-points" with a neighbored processor. The number of neighbors and also the amount of data to send grows linearly with the dimension. Similar considerations are also true for codes using unstructured meshes, particle methods etc.

In a previous paper [3] we have shown that it is possible to execute parallel codes in a distributed grid environment and obtain a high efficiency (between 60% - 88%, see the paper for problems of defining speedup and efficiency in a heterogeneous setup). The code was executed across four supercomputers which were geographically distributed across the United States. These demonstration runs were part of the "Distributed Terascale Facility" proposal (DTF), which was later funded by the NSF to build a nationwide "Tera Grid" [10]. The work described in [3] was awarded a Gordon Bell Prize at Supercomputing 2001 in Denver.

The DTF runs required a lot of effort. Many communication and domain decomposition details had to be adjusted manually for the environment used (both machines and network). Without such manual tuning very poor efficiency ($< 20\%$) was seen. These adjustments affected primarily the processor topology, the domain decomposition, compression and the choice of the number of ghostzones (choosing multiple ghostzones is a technique to overcome latency problems, see Section 6). These experiences led us to develop algorithms to implement these adjustments, and their automatic tuning, in the parallel driver of the code. This paper describes these algorithms in detail, along with their generic implementation, and presents results for real world applications.

Other authors have also shown that optimizing applications for wide area execution is beneficial [11, 12]. Here we emphasize however that the optimizations described here are performed solely in the *driver level* of the general Cactus Code, and as such can be immediately exploited by entire families of applications running within the Cactus framework. Further, the well defined driver interface and modularity employed in Cactus means that this grid aware driver can also be employed by independent simulation codes.

2.1 Considerations for Efficient Grid Performance

A wish list for smart algorithms for parallel codes running in a grid environment should contain the following considerations:

load balance: Divide the whole domain into subdomains for each processor such that the heterogeneous setup is accounted for (in particular different processor speeds). No processor should need to wait for any other, no processor should hold up others.

processor topology: Minimize communication over the WAN by calculating the optimal processor topology according to the number of processors on each machine and the shape of the computational grid.

latency hiding: The driver code should optimally coalesce groups of smaller messages into larger ones when needed. This deals with latencies in wide area networks (in our case implemented using multiple ghostzones, as discussed below).

apply selective compression: The code should optimally compress data which will be sent over the WAN. Since compression rates and times differ for different kinds of data functions, each data function should be considered separately. For example, periodic, wave-like functions compress much faster and better than irregular, noisy functions. Intra machine communication should usually not require data compression.

dynamically adaptive: These issues in principle all depend on properties which can change with time: compression rates, compression times (calculated functions change in time) and latency effects (frequency of synchronization can change with time), load on a processor and therefore its effective power, etc. This requires the code to continuously adapt and determine the best configuration.

dimension independent: The code should be dimension-independent, i.e. it should be able to deal with 2-dimensional discretized functions as well as with n-dimensional and should calculate the optimal configuration for each dimension.

single-machine as a special case: The implementation of the new algorithm should generalize the previous algorithm in the sense that it contains the special case where the number of machines is one and should then automatically reduce to a classical parallelizing scheme without compression/latency techniques.

minimize overhead: Since additional code will be executed along with the numerical calculations in order to make adaptations on the fly it will introduce additional computational and communication overhead. The goal here is to minimize in particular the communication overhead, i.e. exchange information between processors only if and where really needed.

properly interact with other grid-technologies: The design should be general enough in order to smoothly interact with other grid-technologies (like nomadic migration [13] or similar).

application does not change: The application itself, i.e. the numerical code (written in Fortran, C, or C++) should not need to be affected and can remain unchanged.

All these issues are normally not addressed in codes which have been written for single machine execution. The rest of this paper will describe how we have implemented the above items, and show how these techniques improved efficiency significantly for real world applications.

We would like to make a special remark about the last point above, that the application does not need to be changed. In order to have grid technologies widely adopted by applications, it is crucial to enable existing codes to exploit grid technologies with as little change as possible; otherwise, grids will simply not be used! We show below that for a certain class of applications that use the Cactus framework, this is possible, and with excellent results. However, we want to emphasize that in the future, we do expect that application developers will want to rethink their algorithms, and write new generations of applications that are specifically tailored to run in a grid environment. The GridLab project aims to provide a "Grid Application Toolkit" that will provide monitoring tools to see, for example, which parts of an application perform well on a WAN, to enable existing applications to be retooled to better take advantage of grids, and to allow entirely new families of applications, with new grid-aware algorithms, to be developed.

3 Processor Topology

3.1 Single Machine Case

In the cases considered, the calculations are carried out on a regular, topologically cartesian computational grid in n dimensions. The data on this computational grid is

distributed across processors according to some *processor topology*. This distribution is characterized by an n-tuple of numbers k_1, k_2, \dots, k_n , where k_i is the number of processors used in the i^{th} dimension, and we have $\prod_{i=1}^d k_i = NP$ where NP is the total number of processors. For a three dimensional grid using 32 processors the topology could be $4 \times 2 \times 4$ or $16 \times 2 \times 1$ or even $32 \times 1 \times 1$, i.e., 32 processors in a straight line along the first dimension.

The particular choice of the topology strongly influences the communication behavior of those codes. A two dimensional grid, which is stretched in the y-direction (e.g. 64×128 gridpoints) can be divided up on two processors with topology 1×2 or 2×1 . Since the communication consists of sending/receiving the “faces” of the sliced domain, the latter processor topology turns out to be much more communication intensive: 128 gridpoints would be sent/received every iteration in this case, compared with only 64 points in the former case. Both decompositions will give the same numerical result.

The amount of communication is therefore determined by the size of the faces. For a single machine, our driver code does the following: It computes **all** possible processor decompositions and sums up the sizes of the communication-faces. The topology with the minimal total “face size” is chosen. One might think that this “brute-force” method could become very slow for large numbers of processors and dimensions, but this is not true in our implementation. For example the optimal processor topology for the 6-dimensional grid $128 \times 64 \times 76 \times 96 \times 32 \times 128$ decomposed on 2048 processors (yielding a processor topology of $8 \times 4 \times 4 \times 4 \times 1 \times 4$) takes 0.3 seconds on a EV6 alpha processor.

3.2 Multi Machine Case

The above procedure is optimal for a homogeneous setup – assuming an identical network quality between all processors. In the multi machine case, we deal with another picture: We have fast communication networks between most processors and slow network between *some* processors. Typically, as pointed out in the introduction, the network quality is of 2-4 orders of magnitude worse in terms of latency and bandwidth.

In our model we restrict ourselves to cases where the number of processors on each machine are *divisible*. This is usually the case, since most queues on big systems allow a maximum processor number which is a power of two.

In order to minimize the communication over the wide area network we always pick the smallest faces to be the ones where communications go across the wide area network. In a n-dimensional cuboid, these are the faces which are orthogonal to the *longest* dimension, i.e. the dimension with the largest number of grid points. This means in particular, that the machines always have to be “lined up” in the longest dimension of the computational grid.

In order to solve this general problem the code needs to have all the basic infor-

mation at hand, leading to the following question: How does the driver code know how many processors are being used on how many machines. The local `MPI_Comm_size()` function only returns the total number of processors. We thus need to exploit higher level information services, in our case Globus, or in particular it's resource coallocating package DUROC [14].

3.3 Globus and DUROC

One basic idea behind the Globus Metacomputing Toolkit is to provide infrastructure information about the grid environment, even dynamically in a running code. The part of Globus which provides this information is called DUROC: *Dynamically updated resource coallocator*. This software contains an API which provides exactly the information we need, namely the *job-layout*. It allows us to obtain the information about how many processors are used on how many machines even before `MPI_Init()` is called (We are mainly using the `globus_duroc_runtime_inter_subjob_structure()` function call). Using DUROC terminology, the number of processors on one machine is called a *subjob* of a distributed job.

The road map for constructing the optimal processor topology is the following:

- (1) Obtain the job-layout using the DUROC API.
- (2) Find out which dimension contains the largest number of gridpoints.
- (3) Divide up the grid in this longest dimension across the machines into imaginary subgrids for each subjob. The more processors a subjob has, the bigger it's subgrid.
- (4) Find the subjob with the smallest number of processors.
- (5) Use the algorithm from Section 3.1 to find the optimal processor topology for this *smallest* subjob in it's sub-domain.
- (6) Set the global number of processors for all but the longest dimension according to the topology in the smallest subjob.
- (7) To get the number of processors for the longest dimension divide the total number of processors successively by the number of processors in each other dimension.
- (8) The overall processor topology is now determined for the whole job.

The implementation of this algorithm is straightforward. The input parameters are the size of the global grid, the number of machines, the number of processors on each machine and the dimension. The output then consists of the number of processors in each dimension for each machine, namely the processor topology.

3.4 Example

For an example of the new topology algorithm in practice, consider a three dimensional computational grid with the shape $128 \times 256 \times 64$, and a machine setup with four machines containing 64, 128, 128 and 256 processors. Applying the topology algorithm in Section 3.3: **(1)** DUROC reports the job-layout, namely that there are 4 machines ordered with 64, 128, 128 and 256 processors respectively, **(2)** The longest dimension is the second, with 256 grid points, **(3)** For the purpose of finding the topology, the grid point distribution across the four machines in this second (longest) dimension is taken to be 28:57:57:114, **(4)** The first subjob is the smallest with 64 processors, **(5)** The subdomain for this subjob is $128 \times 28 \times 64$, applying the single machine algorithm of Section 3.1 the processor topology on this subjob is $8 \times 2 \times 4$, **(6)** The global topology is then $8 \times X \times 4$, where the number X of processors in the second direction is still to be found, **(7)** This number X is then the total number of processors $64 + 128 + 128 + 256 = 576$ divided by the number of processors in other directions than the second, $X = 576 / (8 \times 4) = 18$, **(8)** The global topology is now determined to $8 \times 18 \times 4$.

Classical algorithms would not consider machine topologies and maybe not even the shape of the grid. Usual input parameters would be the total number of processors and the dimension of the problem. A previous algorithm we used for single machine execution would try to distribute the numbers of processors equally across the dimensions, and gives a decomposition of $8 \times 8 \times 9$, instead of the $8 \times 18 \times 4$ obtained above.

To compare the old and new methods of calculating processor topology, we can calculate the amount of data sent across the wide area network at each iteration, assuming that every grid point sends 8 bytes (1 floating point number) to it's neighbor.

The old algorithm (always) lines up the machines in the highest dimension, and one slice through the third dimension contains 128×256 gridpoints. The amount of data sent across the WAN is then $128 \times 256 \times 8 \times 3 \approx 0.8\text{MB}$ (multiplied by 3 for 3 wide area connections between 4 machines). In fact it is even more than this since this decomposition does not respect machine boundaries which means that the boundaries between machines are not planes but may contain "edges" (see Figure 1).

For the topology provided by the new algorithm, a slice through the second dimension contains 64×128 gridpoints and since there are now by construction no "edges" the amount of data which has to be sent across the WAN is $64 \times 128 \times 8 \times 3 \approx 0.2\text{MB}$, a factor four less than the 0.8MB with the old algorithm.

4 Load Balancing

For codes which are executed in a real metacomputing environment a proper load balancing is often the key to high efficiency. The machine setup used in [3] used 480 R10000 processors and 1020 Power-3 processors to calculate a set of differential

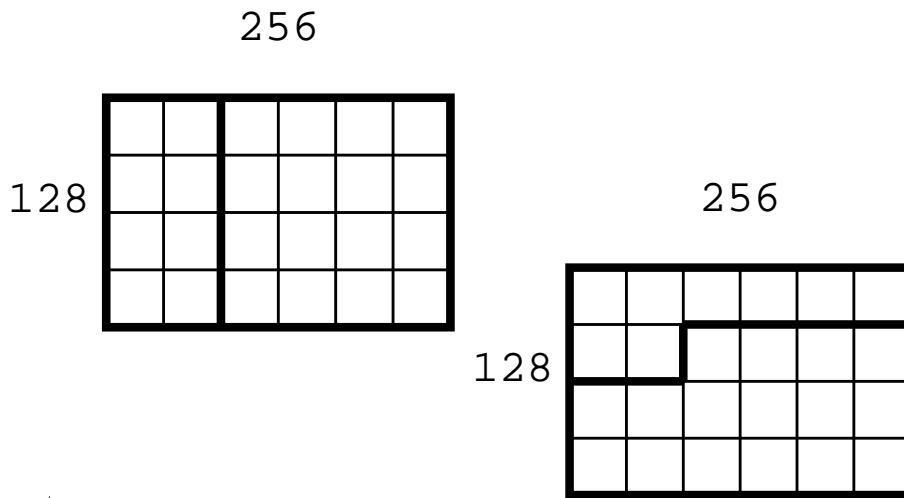


Figure 1: A simple example of processor topology with a two-dimensional grid. Our example shows a computational grid of 128×256 gridpoints. Computing this problem on two machines with 8 and 16 processors respectively our algorithms would yield a processor topology of 4×6 , represented in the upper left diagram. A standard single machine algorithm however, even if it yields the same processor topology will most likely end up in a very ineffective processor distribution due to the lack of grid environment information as can be seen in the right diagram. The thick line represents the machine-boundary in both cases. The longer this line the more data has to be sent across slow inter-machine network connections. Assuming one byte per gridpoint we have to send $256 + (128/4) = 288$ bytes per iteration in the right case and only 128 bytes in the left case. Moreover we have to deal with a machine-boundary which does not have a planar form. This would prevent us from using the latency-avoiding techniques described in Section 6. This is a very simple two dimensional example, but the algorithm always provides the optimal distribution for an arbitrary number of machines and computational grids in arbitrarily high dimensions.

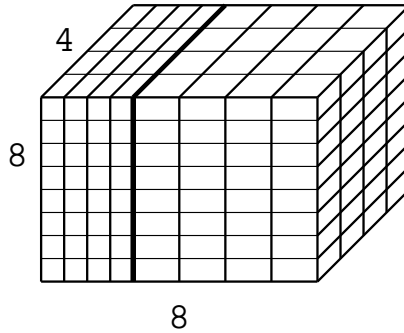


Figure 2: The domain decomposition of a three dimensional computational grid with load $320 \times 320 \times 160$. There are 256 processors distributed equally on two machines. The processor topology is calculated to be $8 \times 8 \times 4$. Since the second machine is about twice as fast as the first, it is assigned twice as many gridpoints. The thick line marks the machine boundary.

equations on a three dimensional computational grid. Since the Power-3 processors are about twice as fast as the R10000s for floating point performance, an equally distributed load would cause 1020 processors to idle half of their time. Then, since they are representing $4/5$ of the total computational power, the total efficiency cannot exceed $3/5 = 60\%$ even for the ideal case of no communication or parallelization overhead.

4.1 Domain Decomposition

Section 3 illustrated how the processors are arranged, i.e. which processor has which neighbor. We have not yet specified how large the part of the computational domain assigned to each processor is. That is, we need to construct a suitable *domain decomposition*. We have one single MPI job consisting of one or more subjobs. The total number of processors is the sum of all processors in the subjobs and we assume *one type of processor per subjob*. From Section 3 we know that the processor topology is calculated in such a way that each subjob-set of processors forms an n-dimensional cuboid and that neighboring subjobs are always located in the dimension with the highest number of gridpoints (the “longest” dimension). Figure 2 illustrates this for the case of three dimensions.

Taking these facts into account, the domain decomposition is calculated in the following manner: The entire computational domain is divided into slabs (see Figure 2) of processors in the longest dimension. Each slab is uniquely located on one machine or subjob. The local processor speed now determines the “thickness” of the slab in the longest dimension.

This can be compared with the situation where n painters have to paint a wall with all painters finishing at the same time, but where each painter works at a different speed. The way of dividing up the area A of the wall between the painters is the following. The speed S_n of every painter and his part of the wall A_n are related by

$$\frac{A_k}{A_j} = \frac{S_k}{S_j}$$

Since the sum $\sum_{k=1}^n A_k = A$ and the speeds S_k are known the above system can be solved for every A_k .

The distribution of gridpoints across the machines is calculated with the same principle. Each subjob runs a test loop to measure floating point performance. The speeds of the machines (or subjobs) are the inverse of the time which taken, measured by the `getrusage()` call.

4.2 Example

We illustrate domain decomposition with an example run across 256 processors. As in the DTF case, we performed distributed runs between SDCS and NCSA. We used two machines, an Origin 2000 with R12000 processors and 'Blue Horizon' an IBM SP equipped with Power3-processors. The global three dimensional grid size used was $320 \times 320 \times 160$. The processor topology was calculated to be $8 \times 8 \times 4$, the subjobs being lined up in the first dimension. The benchmark floating point loop took 2.41 seconds on the IBM SP and 4.40 seconds on the Origin subjob. The resulting domain decomposition then contained 41×42 grid points on every processor in the y and z directions and the gridpoint distribution across processors in the x direction looked like: $28 + 28 + 28 + 26 + 52 + 52 + 52 + 54$ which means that Blue Horizon calculates with almost twice as much data as the Origin. Of course, since the number of gridpoints must be an integer, the number of gridpoints on the last processor of the subjob changes slightly (i.e. 26 and 54).

4.3 Computation / Communication overlap

In the DTF setup [3] it was observed that an additional performance gain could be achieved when a lower workload was given to the processors located at the wide area network boundary. By this we achieve a partial computation / communication overlap since processors deep inside a machine can keep doing calculations while other processors on the WAN boundary are waiting for data. In the DTF setup, we decided (without theoretical justification) to give those processors 20% less workload and redistribute this on the other processors of that subjob.

The new parallelizing algorithm now includes this overlap automatically. An example can be seen in Figure 3. We implemented a load rebalance of 20%. In future

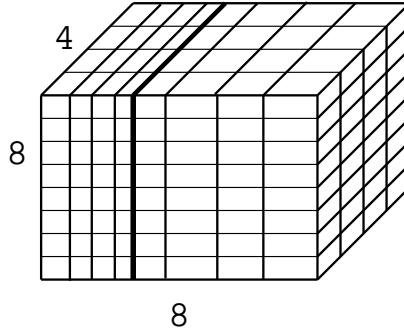


Figure 3: A slightly corrected domain decomposition of the one in Figure 2. The processors sitting at the WAN boundary (the thick line) get less workload than the others in the same subjob. By this we achieve a partial overlap of computation and communication.

versions of our code this number will be estimated using network characteristics of the wide area network interconnect which will be determined by our driver code in terms of latency and bandwidth.

5 Compression

One common way to work around network bandwidth problems is to use data compression. In the DTF setup [3] we made the following simple choice: all data which was communicated in the direction of the compressed. Since the machines had been manually lined up in this direction, this meant that all TCP traffic was compressed. Unfortunately this was not a very satisfactory implementation since *all* communicated data in this direction was compressed, including messages across intra-machine processor boundaries with relatively high bandwidth.

In the new implementation, compression is only applied where it provides an advantage. That is, where the time taken to compress, send and uncompress is less than simply sending the uncompressed data. We are working with floating point data from discretized functions whose values change with time, and the potential advantage of compression depends strongly on characteristics of the data.

The achieved compression rates can vary strongly, depending on the application. We observed a whole spectrum, from compressing data down to less than 10% of the original size to rates of about 70%. The compression method we are using is the

“deflate” algorithm which is implemented within the zlib package [16]. It is a lossless compression method which represents a mixture between the well known Huffman coding and the Lempel-Ziv (LZ77) algorithm. Floating point data may be difficult to compress, and different compressions algorithms may provide additional benefit over what we have seen to date.

The fundamental object in the Cactus Code is a *group* of functions with usually have very similar properties (often the components of a vector or tensor), and we apply compression in a groupwise manner.

Unlike the choice of the number of ghostzones (see next section) the choice of whether or not to compress data is a *local* decision. This means that every processor can decide independently if it wants to send compressed or uncompressed data. Even the receiving processor does not have to know explicitly: It knows the size of the uncompressed data. If the size of the arrived data is smaller than the expected size then it assumes compression.

The question about whether compression gives a performance benefit or not depends in principle on the ratio between the speed of the processor and the network bandwidth, i.e. is the time needed for compressing and decompressing shorter than the time one saves by sending a compressed message? Experience shows that this is definitely *not* the case within supercomputers and clusters with a high speed interconnect, where the bandwidth between nodes approaches 100MB/s. This is the reason why the code never tries to compress data within one cluster or supercomputer. This is possible since from Section 3 we exactly know where WAN communications occur.

The procedure here is the following: At a given iteration the driver code switches the compression mode (on or off) and monitors the communication times for some iterations (the user can choose the length of this monitoring interval, default is 20). This is being compared to the times before switching the mode and depending on the success this mode is maintained or not. This whole procedure is repeated user defined intervals (the default is 300 iterations) since in a dynamic grid environment many things can change, like network bandwidth, the compression rate of the functions or the number of functions to synchronize.

6 Ghostzones

Ghostzones are a common technique used in parallelizing finite differencing codes. They are buffers which contain data from neighboring processors which allow all interior grid points on a processor to be updated. After each iteration this data has to be synchronized, which means it has to be refilled with data from the neighbored processor. Depending on the *stencil size* of the numerical algorithm the depth of the ghostzone can be one or more points.

The crucial point in our considerations is the fact that if the ghostzone size is twice as big as the stencil size, the synchronization of grid functions (that is, send/receive op-

erations) only needs to be applied half as frequently. This kind of “message-coalescence” allows us to overcome latency effects. In [3] we used a ghostzone size of 10 in the highest dimension. This allowed us to exchange data over the wide area network only every 10th iteration, reduce the latency by a factor of 10.

Increasing the ghostzone size however incurs a forfeit – we are trading network latency at the expense of additional memory and CPU (additional ghostzones require more memory, and must also be evolved at each iteration).

Whereas the compression choice was seen to be a *local* consideration, the decision of whether or not to change the ghostzone size is *global* in nature. The number of ghostzones needs to be consistent from processor to processor. In particular they need to have the same depth in the $n-1$ dimensional plane they form (like e.g. the thick line in figure 2). In our model the machine boundaries always form a $n-1$ dimensional plane, and we need to change the ghostzones consistently on this plane.

The goal again is to automatically and dynamically select the optimal number of ghostzones while the code is running. No *a priori* assumptions about the quality of the network should be assumed. We proceed as in the case of compression. The ghostzone size is successively changed (in steps of the stencil size of the numerical algorithm), and the impact on code performance is measured and archived. Once there is no significant steps of the stencil size and measure the performance benefit or loss. This performance improvement the code settles down to the optimal number of ghostzones.

Since changing the number of ghostzones needs to be consistent across sets of processors, any individual processor cannot decide to change autonomously. Instead it must exchange performance data with all other processors on it’s slab to make a consistent decision. Moreover, it has to exchange this information with the processors on the opposite side of the wide area network since they also need to use the same ghostzone size.

All those extra communications are implemented such that this overhead is minimized. A new MPI_Comm_Group for processors on the slab is created, and only the upper left processor of this slab exchanges ghostzone information with it’s neighbor on the other side of the WAN. This is a simple implementation of a *topology aware* collective operation on the driver code level. Efforts of implementing this on the MPI level are a current research topic [15].

This leads us to another difference between adaptive compression and adaptive ghostzones: Increasing ghostzone sizes during code execution introduces additional communication overhead since each new ghostzone point has to be filled with data from the neighboring processor.

7 Performance Monitoring

The adaptive methods described in the last two sections follow the same pattern. A history of performance numbers is maintained and based on these statistics, the driver

is able to make decisions on whether or not, and in which direction, to change the communication infrastructure.

One important statistic is the communication time for data exchange between processors, measured as the wallclock time. The smaller the communication time, the better the chances for high performance.

We also use the software package, PAPI [17, 18] to obtain detailed performance statistics for the running code on the fly. PAPI provides a machine independent performance API, which allows the driver to obtain the required metrics. The main quantity we are interested in is the current floprate.

For example, when the ghostzone size is changed, the new floprate is compared with the previous one. If it improved the ghostzone size is increased until no significant improvement is observed.

8 Results

A major motivation for distributed parallel simulations is to achieve high performance for large scale simulations. In this chapter we present results from example runs which demonstrate how a code's performance can be drastically improved using the new algorithms described in this paper. The test runs used here are examples of real world applications, executed in real grid environments. In particular, we used the same computationally intensive module sets and parameter files from Cactus which are used in largescale production runs by Numerical Relativists to simulate the physics of black holes.

8.1 Single Machine Tests

The first test is a 16 processor run on an Origin 2000 machine, using two subjobs on the same machine, each with 8 processors. As for real distributed runs the communication within a subjob is very fast, with communications between subjobs slower across a 100MBit fast Ethernet interface (MPICH-G2 routes communications between subjobs across TCP connections even if they are located on the same machine).

Figure 4 compares runs using the old and new algorithms. One can see that both runs start at about the same floprate, around 1100 MFlops/s. The automatic adaptation then switches on in the new algorithms, monitoring the communication performance seen data which is sent over the network interface is compressed. Since the local interface is quite fast (more than 40MB/s to the local machine) the additional CPU time needed for compressing and decompressing actually leads to a performance loss (at around iteration 45) as the diagram shows. The decision routines of our driver code recognize this degradation, and switch off data compression. It is quite surprising that even in this case with a single machine, increasing the ghostzone size improved the efficiency.

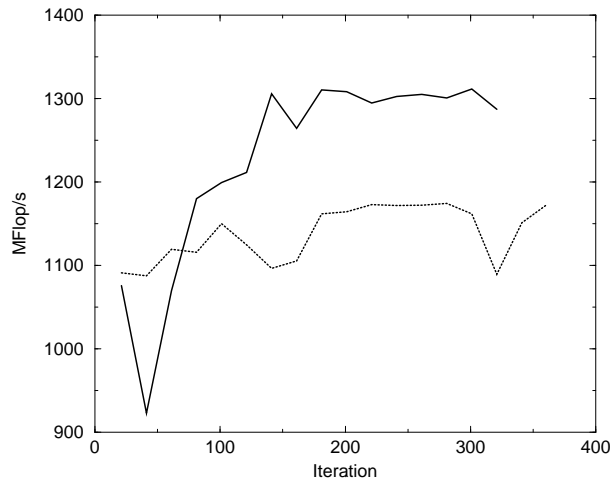


Figure 4: Single processor run with two subjobs: The upper curve shows a run using the new algorithms and the lower curve shows a standard run. Although in this case compression does not provide a performance benefit, increasing the number of ghostzones results in the performance improving by 20%. In this run, the initial ghostzone size was one, and the final ghostzone size was three.

8.2 Transatlantic Runs

Having proved the logic of our new algorithms with test runs on a single machine, we move to results from tests in a real distributed grid environment. The setup now comprises two Origins on opposite sides of the Atlantic Ocean. The first machine is located in St.Louis (USA), and the second in Potsdam (Germany). The two machines are built from different speed processors. As in the previous example, the code applied all the new techniques discussed in this paper, rearranging processors according to job layout, load balancing, overlap of computation and communication and compression application and ghostzone adaptation during the run. Figure 5 clearly shows how these techniques improve performance. Using an improved processor topology and a properly balanced workload the new code starts with a much better performance right from the start. After the application of data compression and increasing the ghostzone size the floprate of the application improved once again yielding a total performance gain over the old code of a factor 3-4.

The application code executed in this run required 9 different groups of functions to be synchronized. The adaptive compression algorithm applied found that not every group of functions led to a performance increase with the application of compression. In the run shown, the algorithm chose to apply compression to only 5 out of the 9 function groups. This number of course can change while the code is being executed

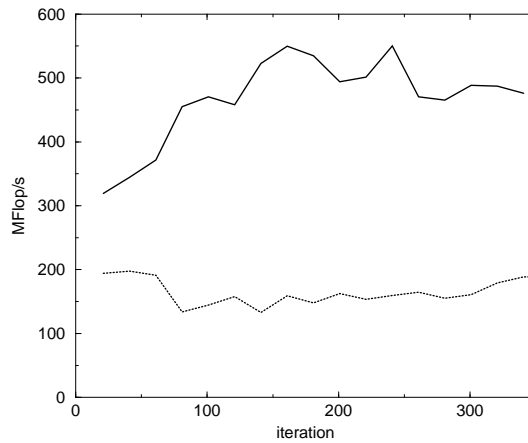


Figure 5: The change in the floprate with iteration number for a transatlantic run using 8 + 8 processors on two Origin 2000s in the USA and Germany. The lower curve represents the floprate of a “standard” distributed run, and the upper curve shows the performance improvement when applying our new algorithms. The application code is exactly the same in both cases.

according to the actual network performance and changes to the characteristics of the data. The ghostzone size to which the code adapted to was found to be 3 or 4 for this set of test runs.

8.3 Large Scale Distributed Runs

The last results presented here use a similar setup to that used for the DTF runs, running between Origin 2000 machines at NCSA in Illinois, USA and a SP2 machine at SDSC in California, USA. For the previous DTF runs we used a manually tuned code, with optimizations needing to be specially applied at the application level. The results here, as in the previous sections, use the new techniques which are all implemented in the driver layer, the application code required no changes.

This particular setup and application code were used in demonstrations at Supercomputing 2001 in Denver. The results of these test runs can be seen in Figure 8.3. The left diagram shows the efficiency of a large scaled distributed run between NCSA and SDSC running on 256 processors while the right one plots the floprate of a smaller run. During these runs, a ghostzone size of 3 was shown to be optimal.

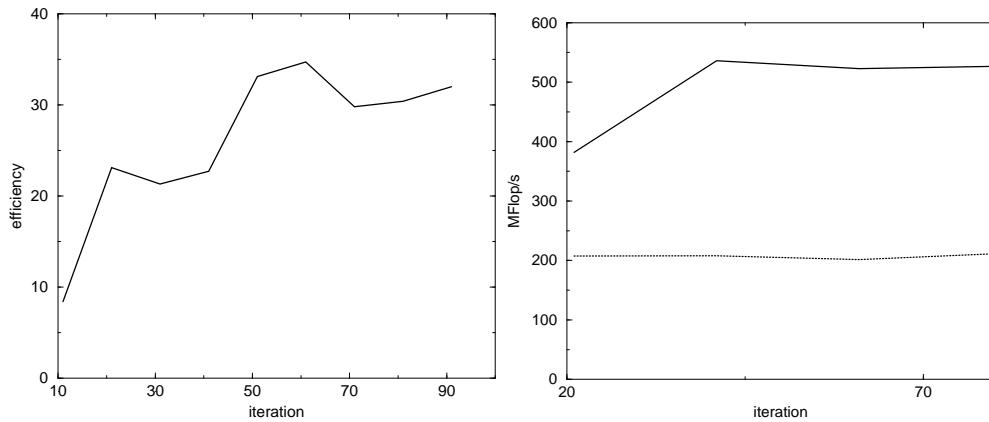


Figure 6: The efficiency of a larger scaled distributed runs across 256 processors on two different sites. The picture here is very similar to our previous example of a transatlantic run as seen in the right diagram which shows the Floprate of a smaller run (16 CPUs) between SDSC and NCSA with and without our algorithms.

9 Discussion and Summary

We have shown that rewriting a parallel driver for multi-machine execution can in many cases lead to an large performance benefit for computations carried out on in a distributed environment. Our algorithms include special techniques which reduce the limitations of network latency and bandwidth. The algorithms are also adaptive, efficiency is automatically improved with time as the computation runs without needing user intervention. The adaptation also allows a computation to deal with changing characteristics of the various networks connecting the machines, on the fly, during the computation. The techniques were demonstrated to work well for a family of production finite difference simulation codes.

In a modular framework such as Cactus, developing such a grid-aware, adaptive parallel driver layer can have enormous benefits. In principle, the application itself, inserted into Cactus, does not require any modification to run in a distributed grid environment. In practice, using realworld environments, we have shown that some families of applications can not only be run without modification, but also run quite efficiently. However, it is certainly the case that other types of applications will not perform as well, and would not benefit from the techniques we have developed.

Although implemented in a generic way, various assumptions have been made about the environment. For example, although we expect different machines to have different processor characteristics, and can automatically load balance for this, at present we assume all processors on a given machine to have the same characteristics. This can obviously be refined in the future, and will also be addressed by scheduling

systems.

Our dynamic, runtime adaption algorithms are based on the premise that adjustments in the communications parameters can be made quickly enough to respond to the changing network characteristics. For example, it is possible that the network quality of a wide area network can change much faster than the monitoring intervals of our code which then leads to a wrong result of the adaptation. One method for more effectively choosing when to try adaptation, would be to interface with a network monitoring service, such as NWS [19]. The calculation could then adapt, as it starts up, to the network and code characteristics at that time, and could then wait for instructions to adapt again. The instructions coming both from the network monitoring service, signaling changes in the used network, and the driver which monitors changes in the characteristics and data of the simulation itself.

Our techniques also introduce some overheads. For example the latency reducing techniques of changing of ghostzones during code execution introduces an additional communication overhead. Our test runs however, which were quite short as compared to real production runs, the time needed for the extra communication was about 1% of the total execution time.

We have also dealt with cases up to now where the calculation itself does not change over time. Although we load balance automatically at the initialization phase, giving more work to more powerful processors, some types of calculations will change with time, depending on the algorithms, physics models, etc. For example, adaptive mesh refinement techniques will automatically increase or decrease simulation resolution in various spatial-temporal regions of a simulation. This changing resolution could produce dramatically changing local workloads as the computation proceeds. Our adaptive tools will need to be modified to account for such behavior in the future.

We have implemented various simple decision making algorithms that try to make only those changes that lead to an increase in efficiency, but as we have mentioned they can be fooled, especially in cases where grid conditions change rapidly. In the future they can be made to be much more intelligent, making use of services such as NWS for more information, or checkpointing if the network becomes unusable.

Hence, although we have taken an important step, both by demonstrating that wide area distributed computing can indeed be performed efficiently in a flexible framework like Cactus, supporting multiple simulation codes with little or no changes, and allowing the computation to adapt to the current grid environment, there is much work to be done to enhance and improve both the algorithms and their implementation. Not only that, but there is much to be done before such techniques will be widely used by the various applications communities. For example, the acquisition of multiple resources is still very difficult, and will require co-schedulers to be in place across many sites to be useful. Further, the use of portals, that will enable the average user to simply request some abstract need, for example by clicking a web button, will need to be further developed and deployed.

Even once such portals, co-schedulers, resources brokers become routinely deployed, much more work remains to accommodate the vast spectrum of applications that can potentially run on the grid. To further enhance the efficiency of even well suited codes, like those used in this paper, and to address the cases where algorithms need to be reconsidered or redeveloped for a grid environment, we are developing a Grid Application Toolkit in the GridLab project [7] that will allow application developers to custom build their applications to better exploit grid services. When the grid environment fully matures, and when such a Grid Application Toolkit becomes available, with effective monitoring tools to show application developers how their algorithms and codes are performing, we will finally reach a stage where grids can be used to enable the new and innovative applications people are envisioning.

10 Acknowledgments

The ideas to develop this code arose from highly collaborative efforts between the Albert Einstein Institute (AEI), Argonne National Labs (ANL), the National Center for Supercomputing Applications (NCSA) and also the SanDiego Supercomputing Center (SDSC). In particular, we thank our Gordon Bell award colleagues Ian Foster, Matei Ripeanu, Brian Toonen, and Nick Karonis, as well as Tom Goodale, Thomas Radke, and John Shalf, among others, for many ideas that contributed to this work. We would like to especially thank Ian Foster (ANL), John Towns (NCSA) and Phil Andrews (SDCS) for providing computing resources and technical support.

References

- [1] *A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems*, I. Foster and N. Karonis, Proc. Supercomputing 98 (SC98), Orlando, FL, November 1998.
- [2] *Distributed computing in a heterogeneous computing environment*, E. Gabriel, M. Resch, T. Beisel, R. Keller, EuroPVMMPI'98 Liverpool/UK, 1998.
- [3] *Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus*, G. Allen, T. Damlitsch, I. Foster, N.T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen, Proc. SC2001, Denver, CO, November 10-16, 2001.
- [4] Cactus Code home page: <http://www.cactuscode.org>
- [5] *Scientific Computing on the Grid*, G. Allen, E. Seidel, J. Shalf, Byte, Spring 2002, <http://www.byte.com/download/BYTE02.zip>
- [6] *Cactus Grid Computing: Review of Current Development*, G. Allen, W. Benger, T. Damlitsch, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke and E. Seidel, Euro-Par 2001: Parallel Processing, Proceedings of 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, R. Sakellariou, J. Keane, J. Gurd, L. Freeman (Eds.), Springer, 2001.
- [7] GridLab home page: <http://www.gridlab.org/>,
GriPhyn home page: <http://www.griphyn.org>,
DataGrid home page: <http://www.eu-datagrid.org>.
- [8] *Distributing Spacetime: Computing and Visualizing Einstein's Gravitational Waves across the Metacenter*, R. Gjertsen, J. Massó, M. Nardulli, E. Seidel, J. Shalf and D. Weber, Forefronts, Volume 11, Number 3, 1996
- [9] *Cactus Application: Performance Predictions in Grid Environments*, M. Ripeanu, A. Iamnitchi and I. Foster, Euro-Par 2001: Parallel Processing, Proceedings of 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, R. Sakellariou, J. Keane, J. Gurd, L. Freeman (Eds.), Springer, 807-816, 2001.
- [10] Tera Grid home page: <http://www.teragrid.org>
- [11] *Optimizing Parallel Applications for Wide-Area Clusters*, H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, Rutger, F. H. Hofman, Technical Report IR-430, Vrije Universiteit, Amsterdam, 1998.

- [12] *Parallel Computing on Wide-Area Clusters: the Albatross Project*, Henri E. Bal, Aske Plaat, Thilo Kielmann, Jason Maassen, Rob van Nieuwpoort, Ronald Veldema. <http://www.cs.vu.nl/albatross>.
- [13] *The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment*, G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel and J. Shalf, International Journal of High Performance Computing Applications, Volume 15, Number 4, 2001.
- [14] *A Resource Management Architecture for Metacomputing Systems*, K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pp. 62-82, 1998.
- [15] *Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance*, N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, Fourteenth International Parallel and Distributed Processing Symposium (IPDPS '00), pp 377-384, Cancun, Mexico, May 1-5, 2000.
- [16] Zlib home page: <http://www.gzip.org/zlib>
- [17] *End-user Tools for Application Performance Analysis, Using Hardware Counters*, K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, T. Spencer, Presented at International Conference on Parallel and Distributed Computing Systems, August 2001.
- [18] PAPI home page: <http://icl.cs.utk.edu/projects/papi>
- [19] *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, Rich Wolski, Neil Spring, and Jim Hayes, Journal of Future Generation Computing Systems, 1998