Distributed Computations in a Dynamic, Heterogeneous Grid Environment

Dissertation

vorgelegt von
Thomas Dramlitsch

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
– Dr. rer. nat. –

geschrieben am
Max-Planck-Institut für
Gravitationsphysik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Universität Potsdam

Gutachter:
Prof. Dr. B. Schnor
Prof. Dr. E. Seidel
Prof. Dr. I. Foster

Potsdam, im November 2002

Acknowledgments

There are many people who contributed with either their personal or scientific support to the development of this thesis and this is the place I want to express my gratitude.

On both the scientific and the personal side I like to thank my advisor Professor Ed Seidel who guided me over many years through the fields of physics and computational science and – what is most important – gave me outstanding opportunities for my scientific and personal development. I think only very few students can claim to have such a great supervisor.

I would like to express my thankfulness to Professor Bettina Schnor, who made it possible for me to become a PhD student in computational science. I want to thank her very much for the uncomplicated and straight way of dealing with various issues as well as for all the fruitful discussions and advices that significantly influenced this thesis.

I wish to thank Professor Ian Foster for being my third supervisor and supporting my visit at the Argonne National Labs where I learned much about distributed computing.

I like to acknowledge the head of the Cactus team, Dr. Gabrielle Allen, who helped me in many respects during the whole time.

Thanks to Gerd Lanfermann, for many useful and constructive discussions and for helping to make the PhD years a fun time.

Also, I guess I owe many people from the Cactus team a lot of beers for any kind of help, hints and discussions. I would like to thank all the former and current members of the group, including Thomas Radke, Tom Goodale, Michael Russel, Andre Merzky, Kashif Rasul, David Rideout, Jason Novotny and Werner Benger.

Apart from the Cactus team I wish to thank the people from the Albert Einstein Insitute who helped me in various ways, especially Prof. Bernard Schutz, Dr. Friedbert Kaspar, Dirk Oede, Oliver Wehrens and Andreas Donath.

I want to thank Matei Ripeanu from the University of Chicago as well as the Mpich-G2 developers Brian Toonen and Nick Karonis for the interesting and exhausting but also fun time at the Argonne Labs.

I like to thank John Towns from NCSA for giving me the unique opportunity to participate in

the preparation of the DTF proposal and letting me play around with the NPACI supercomputers. These experiences gave major impulses to the development of this work.

On the personal side, my gratefulness goes at first and foremost to Simone. I want to thank her for her love, patience and personal support throughout the years, but also for many ineresting and long discussions about science and life, which had major impacts on my scientific thinking and also on this work.

Finally I want to thank my parents who never stopped supporting me and who I owe a lot.

This work was supported by the Max-Planck-Institute for Gravitational Physics. Computational resources and technical support have been provided by the National Computational Science Alliance (NCSA), San Diego Supercomputing Center (SDSC), Argonne National Labs (ANL), National Partnership for Advanced Computational Infrastructure (NPACI), Washington University of St. Louis (WUStl) and the GridLab Project.

Deutsche Zusammenfassung

Die immer dichtere und schnellere Vernetzung von Rechnern und Rechenzentren über Hochgeschwindigkeitsnetzwerke ermöglicht eine neue Art des wissenschaftlich verteilten Rechnens, bei der geographisch weit auseinanderliegende Rechenkapazitäten zu einer Gesamtheit zusammengefasst werden können. Dieser so entstehende virtuelle Superrechner, der selbst aus mehreren Grossrechnern besteht, kann dazu genutzt werden Probleme zu berechnen, für die die einzelnen Grossrechner zu klein sind. Die Probleme, die numerisch mit heutigen Rechenkapazitäten nicht lösbar sind, erstrecken sich durch sämtliche Gebiete der heutigen Wissenschaft, angefangen von Astrophysik, Molekülphysik, Bioinformatik, Meteorologie, bis hin zur Zahlentheorie und Fluiddynamik um nur einige Gebiete zu nennen.

Je nach Art der Problemstellung und des Lösungsverfahrens gestalten sich solche "Meta-Berechnungen" mehr oder weniger schwierig. Allgemein kann man sagen, dass solche Berechnungen um so schwerer und auch um so uneffizienter werden, je mehr Kommunikation zwischen den einzelnen Prozessen (oder Prozessoren) herrscht. Dies ist dadurch begründet, dass die Bandbreiten bzw. Latenzzeiten zwischen zwei Prozessoren auf demselben Grossrechner oder Cluster um zwei bis vier Grössenordnungen höher bzw. niedriger liegen als zwischen Prozessoren, welche hunderte von Kilometern entfernt liegen.

Dennoch bricht nunmehr eine Zeit an, in der es möglich ist Berechnungen auf solch virtuellen Supercomputern auch mit kommunikationsintensiven Programmen durchzuführen. Eine grosse Klasse von kommunikations- und berechnungsintensiven Programmen ist diejenige, die die Lösung von Differentialgleichungen mithilfe von finiten Differenzen zum Inhalt hat. Gerade diese Klasse von Programmen und deren Betrieb in einem virtuellen Superrechner wird in dieser vorliegenden Dissertation behandelt. Methoden zur effizienteren Durchführung von solch verteilten Berechnungen werden entwickelt, analysiert und implementiert. Der Schwerpunkt liegt darin vorhandene, klassische Parallelisierungsalgorithmen zu analysieren und so zu erweitern, dass sie vorhandene Informationen (z.B. verfügbar durch das Globus Toolkit) über Maschinen und Netzwerke zur effizienteren Parallelisierung nutzen. Soweit wir wissen werden solche Zusatzinformationen kaum in relevanten Programmen genutzt, da der Grossteil aller Parallelisierungsalgorithmen implizit für die Ausführung auf Grossrechnern oder Clustern entwickelt wurde.

Es sei noch angemerkt, dass die Leistungen aus der vorliegenden Dissertation, besonders Kapitel 7, mit dem *Gordon Bell Prize* ausgezeichnet wurden. Ebenso sei angemerkt, dass zum Aufbau der weltgrössten Grid Infrastrukur – des US TeraGrid – die Ergebnisse aus Kapitel 5 benutzt wurden [66, 20].

Nachdem wir im **ersten Kapitel** eine historische Zusammenfassung über dieses Gebiet geben (wobei "historisch" nicht mehr als 7-10 Jahre bedeutet) und die ökonomische und wissenschaftliche Notwendigkeit dieser Aufgabe erläutern, stellen wir in **Kapitel 2** klassische Parallelisierungsalgorithmen vor sowie in **Kapitel 3** unsere Werkzeuge (d.h. Software), mit denen wir solche verteilten Rechnungen durchführen wollen. Bevor wir uns im **Kapitel 5** ins Abenteuer einer grossskalierten Berechnung mit vier Grossrechnern stürzen, versuchen wir zunächst im **Kapitel 4** die Effektivität solcher Berechnungen aufgrund eines einfachen Modells vorherzusagen. Die Vielfalt der Probleme auf die wir in Kapitel 5 gestossen sind und die Erfahrungen, die wir dort gesammelt haben analysieren wir in **Kapitel 6** und schlagen neue Lösungswege vor. In **Kapitel 7** gehen wir noch weiter und versuchen durch dynamische und adaptive Techniken der Tatsache Rechnung zu tragen, dass sich die Intensität der charakteristischen Eigenschaften der Probleme (z.B. Bandbreite, verschiedene numerische Algorithmen etc.) auch noch während der Programmlaufzeit ändern kann. Schliesslich – in **Kapitel 8** – implementieren wir die Strategien, die wir für sinnvoll halten, und zeigen in **Kapitel 9** in welcher Weise sie unsere Ausgangssituation verbessert haben. Nachdem wir in **Kapitel 10** kurz andere Projekte auf diesem Gebiet vorstellen, schliessen wir mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf die Zukunft in **Kapitel 11** unsere Arbeit ab.

Abstract

In order to face the rapidly increasing need for computational resources of various scientific and engineering applications one has to think of new ways to make more efficient use of the worlds current computational resources. In this respect, the growing speed of wide area networks made a new kind of distributed computing possible: Metacomputing or (distributed) Grid computing. This is a rather new and uncharted field in computational science. The rapidly increasing speed of networks even outperforms the average increase of processor speed: Processor speeds double on average each 18 month whereas network bandwidths double every 9 months. Due to this development of local and wide area networks Grid computing will certainly play a key role in the future of parallel computing.

This type of distributed computing, however, distinguishes from the traditional parallel computing in many ways since it has to deal with many problems not occurring in classical parallel computing. Those problems are for example heterogeneity, authentication and slow networks to mention only a few. Some of those problems, e.g. the allocation of distributed resources along with the providing of information about these resources to the application have been already attacked by the Globus software.

Unfortunately, as far as we know, hardly any application or middle-ware software takes advantage of this information, since most parallelizing algorithms for finite differencing codes are implicitly designed for single supercomputer or cluster execution. We show that although it is possible to apply classical parallelizing algorithms in a Grid environment, in most cases the observed efficiency of the executed code is very poor.

In this work we are closing this gap. In our thesis, we will

- show that an execution of classical parallel codes in Grid environments is possible but very slow
- analyze this situation of bad performance, nail down bottlenecks in communication, remove unnecessary overhead and other reasons for low performance
- develop new and advanced algorithms for parallelisation that are aware of a Grid environment in order to generelize the traditional parallelization schemes
- implement and test these new methods, replace and compare with the classical ones
- introduce dynamic strategies that automatically adapt the running code to the nature of the underlying Grid environment

The higher the performance one can achieve for a single application by manual tuning for a Grid environment, the lower the chance that those changes are widely applicable to other programs. In our analysis as well as in our implementation we tried to keep the balance between high performance and generality. None of our changes directly affect code on the application level which makes our algorithms applicable to a whole class of real world applications.

The implementation of our work is done within the Cactus framework using the Globus toolkit, since we think that these are the most reliable and advanced programming frameworks for supporting computations in Grid environments. On the other hand, however, we tried to be as general as possible, i.e. all methods and algorithms discussed in this thesis are independent of Cactus or Globus.

We start with a short introduction to Grid computing and explain its benefits for science and economics. Furthermore, basic terms and notions that are relevant for this thesis are explained and extended from standard parallel computing to the usage in Grid environments. In **Chapter 2** we give an overview about what we call "traditional parallel computing", in **Chapter 3** we analyze the Grid awareness of MPI implementations, present some smaller scaled test runs and end with a description of general problems. **Chapter 4** deals with a simple model to predict the performance of a distributed run in a Grid environment while in **Chapter 5** we present a first large scaled distributed run in a real Grid environment where we manually applied novel techniques to improve the efficiency of the running code. These experiences can also be found

in the award winning paper *Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus* (presented at the Supercomuting Conference 2001). Since we have learned much from those runs we will analyze and develop in **Chapter 6** algorithms that automatically achieve the configuration we adjusted manually in Chapter 5. Since some environment characteristics (e.g. bandwidth or the used application code algorithm) may change in time we also developed *dynamic strategies* to adapt to the current Grid environment, which we are investigating in **Chapter 7**, including adaptive compression, adaptive selection of ghost zone sizes and dynamical load balancing. These two Chapters form the content of the paper *Grid Aware Paralellizing Algorithms*. In the following **Chapter 8** we discuss briefly the implementation of previously analyzed algorithms within the Cactus framework using the Globus toolkit, followed by a presentation of the achieved results in **Capter 9**, a brief presentation of related work in this area in **Capter 10** and a discussion of possible future directions in this field in the last **Chapter 11**.

The achievements in this thesis, in particular those in Chapter 7 were awarded the *Gordon Bell Prize* in the special category. Furthermore, our experiments in Chapter 5 significantly supported the building and funding of the TeraGrid.

Contents

List of Figures

Chapter 1

Grid Computing: Introduction, History and Survey

1.1 Definitions: Metacomputing, Grid Computing

Grid computing is a very young and rather uncharted field in computational science [27]. It has been a buzzword for the last two or three years along with terms like Metacomputing (coined by Larry Smarr), distributed computing and others. Up to know we used these terms in a quite undefined manner. This may reflect the fact that those terms are quite loosely defined throughout the literature.

For the purposes of this thesis we will define the following to avoid confusion. The *execution of parallel codes across more than one supercomputer or cluster, connected by a LAN or a WAN* will be referred to as *Metacomputing*[1]. We regard *Grid computing* as a more general term, subsuming Metacomputing amongst many other notions (like co-scheduling, resource brokering, migration, remote visualization, peer to peer networks etc.).

In this thesis we are dealing with Metacomputing in the above sense. However, due to some reasons the word Metacomputing is more and more replaced by the general word "Grid computing" in current literature. Thus, except for this overview chapter, we will follow this way and will from now on speak about "Grid computing" but we will mean "Metacomputing". In cases where we mean Grid computing in its most general sense we will mark it as such.

In this introduction we give a brief overview and explain the importance of this field in computational and other sciences.

1.2 Motivation

One goal of distributed computing is to numerically solve large problems which cannot be solved by a single CPU or even by hand. Those problems are mainly related to physics (astrophysics, fluid dynamics, quantum mechanics), chemistry (chemical reaction flows, physical chemistry), engineering (crash tests, air plane simulations) or biology/genome analysis [47].

In many cases it is true that the bigger the scale of the problem, the larger the computational resource has to be in order to solve it. Like in the rest of this thesis we are using examples of computational astrophysics (i.e. numerical relativity) as representatives for problems in computa-

[1] *greek* meta = between

tional sciences. A typical problem in numerical relativity consists of calculating the time evolution of celestial bodies, which can be neutron stars, boson stars, black holes, galaxies or even our solar system. The requirements for a satisfactory computation of even one of those objects is immense in terms of CPU power and memory.

Example A boson star calculation on a three dimensional computational grid[2] of the size of 128^3 grid points needs 2 GBytes of main memory. Doubling the resolution in every dimension increases the memory need by a factor 8. A calculation of this problem on a 2048^3 grid therefore needs 8 TBytes of main memory. The worlds current largest computer, the ASCI[3] White [8] at the Lawrence Livermore National Laboratory in Livermore has 6 TBytes of main memory. While a computational grid consisting of 2048^3 grid points may still not be accurate enough for a one or two body problem in numerical relativity, it is ridiculously coarse for the computation of real interesting and important things like the time evolution of our solar system.

This example shows the relation between the needs of computational sciences and what current computational resources can handle. It is for sure that some time in the future there will be a computer which satisfies our present needs, but in the meantime we can try to work around this problem, and this is one point where Grid computing comes into place.

Grid computing can help to attack the problem of non-availability of big resources. One basic idea is to coalesce resources at different administrative sites which can then form a bigger *virtual machine* or a *virtual supercomputer*. Another but similar idea is to collect *idle* resources which do not necessary have to be located at computational centers but may comprize private computational resources like PCs at home. A famous project which set this idea into production is the seti@home project [62] which allows any home PC connected to the internet to help analyze astrophysical data, quasi as a screen saver.

But Grid computing does not only mean co-allocation of resources. In a more general sense it is about a uniform treatment of distributed resources. By this we mean the fact that a code would be able to run on different resources, and it would be irrelevant on which resource it is running on. A "Grid-scheduler" (or "meta-scheduler") would then decide which resource is to be used and the code could even run sequentially on one resources after another if a single resource cannot meet the conditions about how long the code wants to run, which is called migration. Migration and relevant issues are studied by Lanfermann [42, 43].

In order to illustrate the above ideas better we describe a scenario of what an advanced stage of a Grid environment could look like.

A scientific user is interested in solving a problem in computational science, let it be physics, chemistry, biology, genetics or others. He/She writes a simple code to solve a very downsized problem of the same kind on his/her laptop. This simple single processor sequential code will now be plugged into a Grid computing framework (like the Cactus code) and committed into some software repository. From a *portal* the user can specify the problem size he/she wants to compute and the adequate amount of resources. The portal then contacts a resource broker, checks out the code from the repository, compiles it on the given resources and submits it to the local queue systems. After the run the user will be notified and the results are automatically sent to him/her. The information where the code was being run is unknown (or let's say it will be irrelevant) to the user, like the information whether the code ran on one or more resources simultaneously, had to migrate in the meantime or similar.

[2] Here, "grid" means a "computational grid", a descrete set of points forming a grid. We distinguish it from the other "Grid" by using lowercase spelling

[3] Advanced Simulation and Computing or Accelerated Strategic Computing Initiative

All the ideas described in this section come along with many infrastructural and technical problems. The details of those problems strongly depend on the nature of the application code, in particular on the communication behavior. Codes like finite difference codes are usually very communication intensive. This is the class of codes we are dealing with in this work.

1.2.1 Why Grid Computing?

Up to now we have described what Grid computing should be able to do, but in fact we are still far away from this scenario. This thesis forms one link in the chain of enabling codes to be run in a Grid environment. But still the question we need to answer is the following: Even *if* we have codes that can run more or less efficiently in Grid environments, i.e. codes that can run simultaneously on more resources, can migrate, and even *if* we can uniformly treat a set of distributed resources using resource brokers or scheduler – does Grid computing give us any benefit in terms of time and/or money?

One could argue that the amount of resources will not be increased by Grid computing and if everybody can use any resource, nobody gets any benefit.

But the answer to the above question is yes and the simple reason is that we can say that we *make better use* of the given resources. One can show that under simple assumptions – even taking into account that a job which is split to run distributed across two or more resources runs less efficient – the *average response time* (ART) which is the time between job submission and getting the results can be significantly decreased in such Grid environments [21].

Metacomputing, i.e. the coalescence of multiple supercomputers or clusters to a virtual "mega-computer" thus helps us in three basic scenarios:

- If the resources required for a calculation are not present at a single place or not accessible by the user
- If the resources needed are present at a single place and accessible but not accessible within a certain time
- If the resources needed are present and accessible within a certain time but too expensive

The last item will become important once Grid computing will be in the "production" phase.

1.3 Short history of Metacomputing

Attempts to execute codes across geographically separated computers dates back to the early 90's. In order to give the reader a idea on difficulties to run codes in a real Grid environment efficiently – or run them at all – we review a few milestones from back in 1995 until today, starting with the famous I-WAY experiment.

1.3.1 The I-WAY Experiment

The I-WAY (Information Wide Area Year) was a project in 1995 aimed at installing a high performance ATM[4] testbed between major supercomputing sites in the US. In order to demonstrate the concept of Grid computing (in the broader sense) many Grid experiments on the I-WAY testbed

[4] Asynchronous Transfer Mode

have been performed. One major experiment attempted to run a large scale scientific code as a distributed simulation which was to be executed across many testbed sites. This experiment was demonstrated at the supercomputing conference '95 in San Diego [65].

The code to run was a parallel code using the message passing interface (MPI [69]). The most known MPI implementation, Mpich, developed at the Argonne National Labs was in principle capable for heterogeneous setups as it is using sockets and TCP connections (besides shared memory) for communication. However, it was not developed for parallel codes executed at this order of magnitude.

Therefore researchers at the Argonne Labs implemented a special communication layer for the Mpich implementation they called nexus [29]. Nexus is a communication protocol for uniformly supporting intra- and inter-machine communication. Similar to future experiments of this kind the setup consisted of an MPI application code on top of a Grid-aware MPI implementation [24].

After a long period of preparation (half a year and more) including thousands of person hours of programmers, system administrators, site managers and PIs succeeded in performing first test runs of this distributed code across multiple sites, including remote visualisation. However, a stable demonstration of a running code between the sites (including mainly Argonne, NCSA, San Diego and Pittsburgh) was not possible to present during the Supercomputing Conference. The main reasons for this as seen from today seem to be the heterogeneity of the setup along with a (still) slow network and problems which occur only at larger scales (limited number of file descriptors, limited number of open sockets etc.) The codes that were used in this setup were not advanced and tested enough to handle difficulties at that scale.

1.3.2 Supercomputing 98

The I-WAY experiment described in the previous section helped to gain much experience in this field and gave major impacts for the development of software which specifically deals with problems in Grid computing: Globus [26]. The *Globus Metacomputing Toolkit* was designed to provide basic infrastructure information about Grid resources for applications running on top of it. It provides its own IO system along with a huge collection of APIs. From a layer's point of view it runs underneath all software that is Grid aware and therefore also underneath distributed MPI implementations. Such an MPI implementation has also been written and distributed along with the Mpich package. Its name was Mpich-G. It used nexus as a communication layer and runs much more stable then any previous distributed MPI implementation.

Globus is not thought to be a user level application although it provides features like a uniform interface to batch systems (LSF, loadleveler, NQE, PBS etc.) and a security back-end which is used by applications for authentication (called Grid-security-infrastructure, GSI), like `ssh` and `ftp`. For more details about it see next section.

Three years after the I-WAY experiment this advanced and improved Mpich version was used to make a new attempt to run a tightly coupled code across various geographically distributed sites. The main three sites involved were the computing center of the Max-Planck-Institute in Garching/Germany, the Konrad Zuse Institute in Berlin (ZIB) and the San Diego Supercomputing Center (SDSC) each of them having T3Es deployed to be used in this demonstration [12].

Since the network between these three sites was rather slow, dedicated lines between Berlin and San Diego and between Munich and Berlin have been booked. After again investing several months of person hours to get software and dedicated lines in place it was possible to demonstrate a distributed run on 32+32 processors between Munich and Berlin and for a shorter period a

distributed run across all three sites using 32+32+32=96 processors. It was originally planed to have more than 32 processors per site involved, but the setup did not allow such configuration. The reason for this is unknown up to now.

The efficiency (see Section 2.1) was very poor in the two machine case and extremely poor in the three machine case (sometimes below 10%).

Going beyond the I-WAY experiment 1995 this experiment showed that meta-computing across more than one or two supercomputers is definitely possible, but difficult to realize and still very inefficient.

In this thesis an efficient demonstration of large scale distributed computing using tightly coupled finite difference codes is demonstrated for the first time and will be discussed in the next chapters.

1.3.3 Metacomputing Experiments at SC99

The series of metacomputing experiments continued at SC99 in Portland. This time, researchers from Germany and the United Kingdom used machines from Japan, Germany, UK and the United States, comprising a Hitachi SR8000 and three Cray T3Es. The network setup between these machines had latencies ranging from 20 to 80 milliseconds and bandwidths' ranging from 0.5 to 1.6 Mbit/s.

Three application codes were deployed for this experiment: A code performing analyzis of pulsar data, a computational fluid dynamics code and a code for molecular dynamics. We take a closer look at the latter one, since this is the class of codes we are dealing with in this thesis (namely a finite difference code using domain decomposition methods and data exchange at the boundaries of each processor).

The major problem when executing this code was, as expected, the slow communication speed between machines in terms of latency and bandwidth. The researchers tackled this problem by implementing smart tricks like latency hiding techniques and overcomeing bandwidth problems by reducing the amount of data to be exchanged.

Latency hiding was implemented by a *partial overlap of computation and communication* and also a *partial overlap of communication in different directions* (we discuss this in Section 6.4 in more detail). The amount of data to send was reduced by checking each data exchange for its actual relevance (i.e. could this data also be calculated locally). By this, the total amount of data exchanged could be reduced by up to 60%. More details of these experiments can be found in [54].

However, all these changes were done *on the application level*, using a specific knowledge about the scientific application and also specific knowledges about the science itself.

If codes have to be changed in this way for Grid environments it is clear that these optimisations remain singular events. This thesis will show that optimizations can be achieved *without* changing the application or without any specific knowledge about it.

1.3.4 Globus and OGSA

As outlined in the last Section, past Metacomputing experiments led to the development of the Globus toolkit. This software was under rapid development in the last years. In the year 1997 the Globus toolkit was released in version 1.0. In the following we will briefly introduce its main components.

GSI Grid Security Infrastructure. GSI is based on on public key encryption, X.509 certificates,

and the Secure Sockets Layer (SSL) communication protocol. This includes a "single sign-on" for users to the Grid, secure communication and more.

Gatekeeper This is a deamon process running on every computer that offers Globus services. It uses GSI for authentication and GRAM for job submission.

GRAM Globus Resource Allocation Manager. Used for submitting and monitoring remote jobs. It comprises a client API (e.g. to start a job) as well as a "jobmanager" that runs as a service on the remote machine. The jobmanager exchanges information with the client about the job status (pending, active, running, done, failed). It is invoked by the gatekeeper (see above). According to different queue systems like pbs, lsf etc. the appropriate jobmanagers are called jobmanager-lsf, jobmanager-pbs etc.

DUROC Dynamically Updated Resource Co-allocator. As for GRAM, it consists of a client part and a server running on the remote machines. DUROC is used for multi-machine requests. It coordinates the startup (i.e. through a so-called DUROC_Barrier) and enables inter machine communication. For the submission of each job it uses GRAM.

MDS Monitoring and Discovery Service. An LDAP[5] based service that helps quering and updating information about various resources in the Grid.

GASS Globus Access to Secondary Storage. This is a simple API that provides file access services. On the command line level and in conjunction with GRAM it can be used to stage executables or access outputfiles from the remote machine.

The Globus toolkit verion 2.0 (referred to as GT2) was released early 2002. While staying compatible with the previous version it improved a lot regarding the installation/maintaining process. Also, many features have been added regarding data management issues, amongst other improvements.

The persisting problem of today's implementations of Grid environments like the DOE Science Grid [67], GriPhyn [68] and others is the non-interoperability of such "non-standard" implementations. Most of those Grids have been built as a patchwork of various protocols and standards and can be regarded as singular projects since they provide no standard interfaces for talking to other Grids.

However, an already existing effort to define standars for services is currently under strong development: Web Services. A Web Service can be regarded as a standardized interface to a collection of network accessible operations. The basis of Web Services form XML[6] documents that are transported using protocols like SOAP[7] which are sitting on top of lower level transport protocols like http and ftp. Furthermore, there are standards for the description of a service (using e.g. WSDL[8]) and mechanisms for service discovery and registration (WSIL[9], UDDI[10]).

The Globus Toolkit can, in principle, be regarded as a software that provides *Grid Services* (like GRAM, MDS etc.). This is where Web Services come into play. The Globus Toolkit as of version 3 will adopt many mechanisms of Web Services (like WDSL etc.) to make Grid services

[5] Lightweight Directory Access Protocol
[6] eXtensable Markup Language
[7] Simple Object Access Protocol
[8] Web Service Description Language
[9] Web Services Inspection Language
[10] Universal Description, Discovery and Integration

more standardized and inter-operable. The result of this effort will be the *Open Grid Service Architecture* (OGSA) [28] that will come along with the GT3.

1.4 Other Grid Projects

Our thesis deals with a small part of the whole Grid effort, namely with distributed computations in a Grid environment. In this section we cannot describe all projects dealing with Grid computing in the broader sense, we will just introduce a few major ones.

1.4.1 Legion and Condor

When speaking of frameworks that focus on supporting and building Grid infrastructures in a very general sense, the *Legion* project must be mentioned. Legion is often referred to as *an Operating System for the Grid* [35]. Operating systems are user (and developer) interfaces to hardware and low level software (like file systems, device drivers etc.). Key purposes of an operating system are to unify the access to the underlying hardware and hide unneccessary complexity (e.g. different file systems, different hard disk manufacturers) to the user or developer.

Following this picture Legion has been designed to provide a uniform access to various Grid resources (machines, disk space, networks) and hide unneccessary details (heterogeneous networks and machines) to the user. Thus, the user can log on to a *virtual machine* where the operating system is Legion and the underlying hardware can be a broad collection of Grid resources.

The services Legion offers are ranging from process creation and control, interprocess communication, file system to security and resource management.

Rather than trying to harness any kind of computational resources in the Grid, *Condor* focusses on idle workstations within a local area network [11]. A Condor job may travel and migrate through a network of workstations and will always be executed on a currently idle workstation. This migration and scheduling is completely managed by Condor. Instead of linking a code to the standard C library, the user links against a Condor version of the C library that allows system calls to be executed remotely. Only a "shadow" process runs on the local machine, acting as an agent for the remotely executed program.

Condor-G

Condor-G is the marriage of technologies from the Condor project and the Globus project [30]. In fact, both frameworks have similar objectives but whereas Globus coallocates resources across different domains, Condor coallocates resources within one administrative domain (e.g. the pool of workstations in an institute). Both Condor and Globus provide mechanisms for job submission. Within Condor-G, job submission is possible from either side: Condor can be used to submit to Globus resources and a Condor pool of workstations may appear as a Globus resource (represented by a jobmanager-condor).

1.4.2 Commercial Grid Projects

Since the purpose of Grid computing is not a solely academic one, commercial vendors of hard and software are developing their own software for creating Grid environments. One well known product is the SUN Grid Engine (SGE) [70] that allows the effective sharing of resources (mainly CPU cycles, memory, storage) within one organization.

IBM did not start a Grid effort on its own. It rather supports the installation and usage of the Globus Toolkit for IBM computers and offers better documentation and commercial support. IBM is also strongly involved in the development and implementation process of OGSA (see Section 1.3.4).

Compaq (now Hewlett Packard) follows a similar way. Rather than developing their own Grid software from scratch, they support the Globus software and participate in Grid projects like GridLab [34].

1.4.3 More Aspects of Grid Computing

There are certainly many more aspects of Grid computing then the ones described in this chapter. One broad field which belongs to Grid computing is *peer to peer computing* (p2p). p2p computing became extremely popular with the file sharing program *napster*.

As opposed to many other Grid projects that focus on high performance computing (i.e. processor power) p2p computing focuses on sharing storage space. Unlike other services (like web services using a client/server model) this can be done in a way where every computer is both a client and server, all computers are treated equally (hence peer to peer).

The sharing and co-allocation of mass storage systems in geographically distributed Grid environments is addressed by e.g. the European *Data Grid Project* [61].

In this long list of various aspects of Grid computing we close this section with mentioning things like *remote visualisation* and *spawning*. Remote visualisation describes the visualisation of scientific data without moving large datasets to the local machine. Spawning describes the submission of jobs that are created by currently running jobs (e.g. for data analysis; since this can be done independently of the current calculation, it can be submitted to another resource).

1.5 Grid Computing and this Thesis

As can be seen from our brief overview, Grid computing is a very wide field comprising many disciplines in scientific computing. This thesis can certainly cover only a small area of the whole Grid effort which is roughly illustrated in Figure 1.1

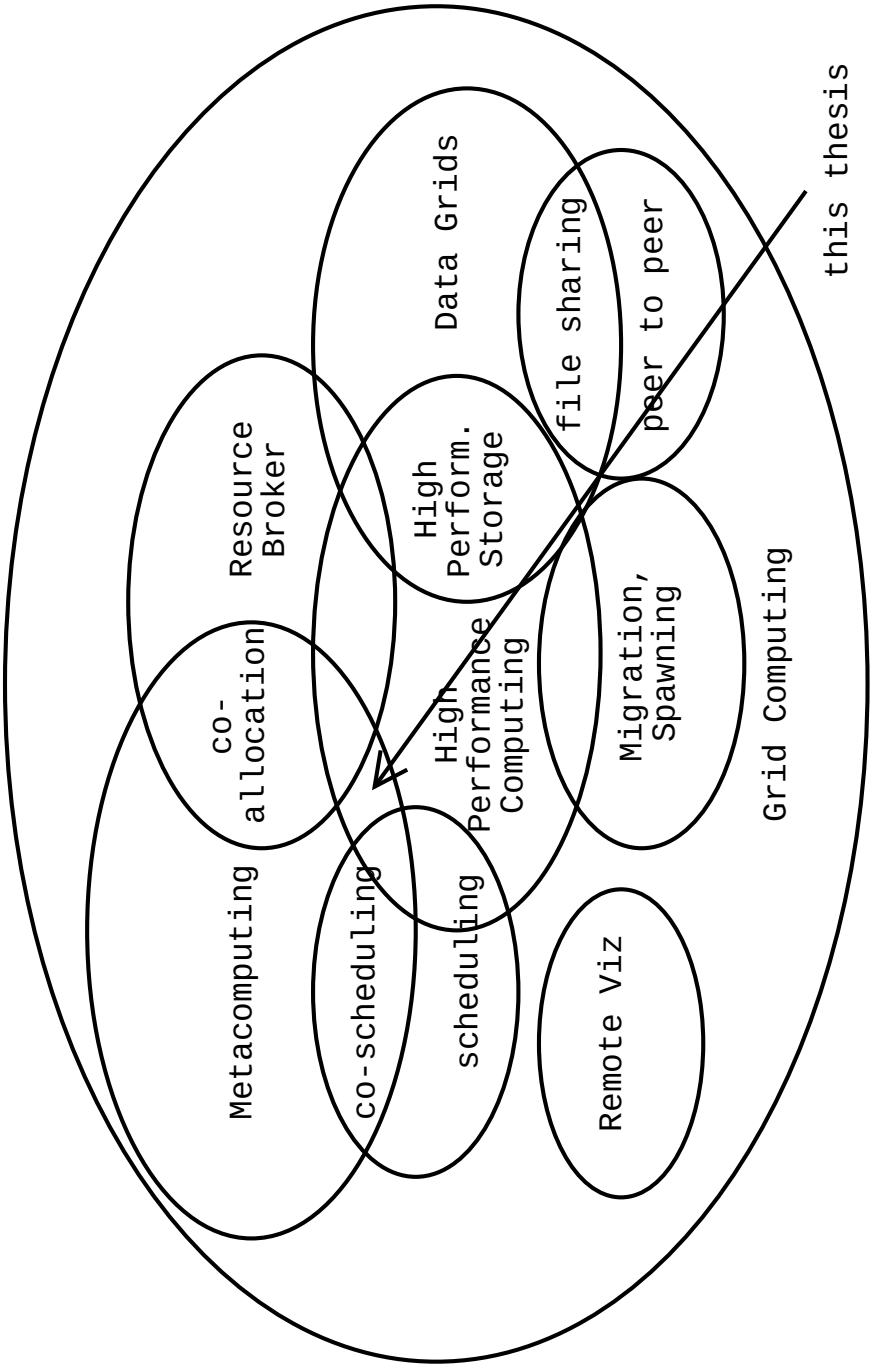

Figure 1.1: **An overview of Grid computing and this Thesis.** The entire Grid effort comprises many areas in computer science. As outlined in this chapter, this thesis covers only a small fraction of it.

Chapter 2

Classical Parallel Computing

In this section we review basic notions of parallel computing for finite differencing codes on a regular (cuboidical) computational grid[1]. We will introduce notions like ghost zones, domain decomposition, processor topologies etc. which are important throughout this thesis. Some of those notions can also be found in [23].

2.1 Performance, Speedup and Efficiency

In this chapter we will define some basic terms used in parallel computing, which are used throughout this work.

Definition

Floprate F Number of floating point operations per second of an executed parallel code.

Speedup S(n) The speedup on n processors is defined as the ratio between the floprate F_1 on one processor and the floprate $F_n$on n processors (even if the execution on one processor is not practical):

$$S(n) = \frac{F_n}{F_1}$$

Efficiency Speedup divided by the number of processors:

$$E = \frac{S(n)}{n}.$$

For a more detailed discussion of these terms see [38].

According to *Amdahl's Law* (see e.g. [23]) each program can be divided into a sequential and a parallel part. By definition, the sequential part cannot be parallelized and as such prevents the parallel code from achieving the maximal possible speedup (i.e. speedup of n on n processors). However, there are other reasons why a code cannot reach the maximal speedup, namely when the processors need to exchange data.
For the efficiency the following holds:
Assume that a given code has a neglectably small sequential part but the processors do spend time

[1] Again, note the lowercase spelling, denoting a mesh

with data communication. When T_{comp} denotes pure computation time of a code, i.e. the time where the CPUs are performing pure computations and T_{tot} is the elapsed wall clock time, the efficiency is given by

$$E = \frac{T_{comp}}{T_{tot}}. \tag{2.1}$$

For the proof one only has to take into account that the floprate is (per definition) proportional to T_{comp} and for the single processor case we assume $T_{comp} = T_{tot}$.

While the definition of the floprate can be smoothly extended to heterogeneous Grid environments we already encounter problems when trying to extend the notion of speedup.

Example.

In a heterogeneous setup, we have 4 processors running the code at a single processor performance of 100Mflop/s and 4 processors running at 250Mflop/s. The measured floprate for a single distributed calculation across all 8 processors is measured as 1Gflop/s. How big is the speedup? One could calculate it as 1000Mflop/s divided by 250Mflops yielding 4 – or one could even divide 1000Mflop/s by 100Mflop/s and thus achieve a speedup of 10 on 8 processors!

One clearly needs a more general definition of speedup then the one above. One finds more generalized definitions for speedup throughout the literature [19, 44], but for us the following is the most appropriate.

Definition.

Given k types of processors in a distributed run, each one occurring n_k times. Let F_k be the floprates of executing the code solely on the k's type of processor. We define the *generalized speedup* $S(n)$ as

$$S(n) = \frac{F_{tot}}{\sum_{i=1}^{k} \frac{n_k}{n} F_k} \tag{2.2}$$

where $n = \sum_{i=1}^{k} n_K$ is the total number of processors and F_{tot} the observed total floprate of the executed code. We claim that the following is true:

(1) For the single machine case where we only deal with one type of processor this new definition falls back to the previous one.

(2) The speedup $S(n)$ is always smaller or equal n.

(3) For a "perfectly scaling" code the speedup is n.

All statements can be easily verified if one assumes that the total floprate F_{tot} of the executed code cannot exceed the sum of all single CPU performances, i.e.

$$F_{tot} \leq \sum_{i=1}^{k} n_i F_i.$$

For our example above we obtain a speedup of 1000/175=5,7 which is a sensible (and unique) number. With this general definition of speedup we can proceed to generalize the notion of efficiency for Grid environments.

Definition.

The *generalized efficiency* E is defined as the generalized speedup divided by the number of processors,

$$E = \frac{S(n)}{n}$$

Taking the above example we obtain an efficiency of 71%.

2.2 Finite Differencing Algorithms

The broad class of codes we are dealing with in this thesis are codes using *finite differencing algorithms*. This is the most popular method for solving differential equations from many branches of science like hydrodynamics, astrophysics, chemistry and many others. The type of the differential equation does not matter, it can be of arbitrary order, a partial or an ordinary differential equations (PDE or ODE) [2].

Again, to avoid any possible confusion we will make the following clear. In major parts of this section, whenever we talk about the *grid*, we do not mean a set of computational resources but the *computational grid*, namely the $n-$dimensional set of points on which a smooth function is discretized (the *grid function*). The *Grid*, as a set of computational resources, will be written in uppercase letters.

Each function is represented in memory as an n-dimensional array, where n is the dimension of the function's domain. All algebraic operations on these functions are then represented by operations on the functions discretized points. We distinguish between point wise operations and others. Most operations like adding, multiplying and integrating (in some cases) are point wise operations, which means that the result of the operation for every single point does not depend on the neighboring points in the $n-$dimensional index space. If this would be true for all kind of operations we would call the code to be trivially parallel (and in that case do not need any communication).

However, it is not true for codes using finite differencing. For a function f which is discretized on a one dimensional grid (marking the $x-$direction) the $x-$derivative $\partial_x f$ of the function at the point x_0 would read

$$\partial_x f|_{x_0} = \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x}. \tag{2.3}$$

For a code which is executed on multiple processors it may happen that the point $(x_0 + \Delta x)$ lies on another processor. In this case the value of this point hast to be *sent* from the neighbored processor to the local processor and this is where most parallel overhead comes into place in this class of codes.

In the following, to be able to illustrate things better, we use a two dimensional grid which means that the domain of all (real or complex) functions is two dimensional. Everything we explain and introduce in this thesis is generic $n-$dimensional though.

Given a set of discretized functions we perform calculations on. If we distribute those calculations across several processors we have to divide up the domain, i.e. the set of points. Fig. 2.1 shows an example where we divide up the domain on four processors. This *domain decomposition*

[2] A mathematical theorem states that every differential equation at any order can be rewritten as a system of first order differential equations, and that is what is mostly done in finite difference codes.

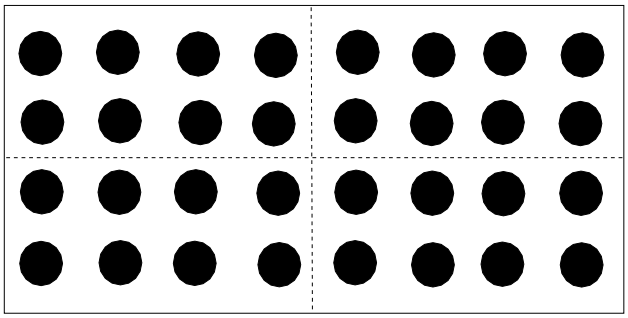

Figure 2.1: This figure illustrates a two dimensional computational grid. Every spot in this picture represents a value of a discretized function in the two dimensional (x,y) space. The dashed lines indicate the domain decomposition for four processors

is not unique. The number of possibilities to distribute the domain across more processors grows with the number of dimensions and the number of processors. A parallelizing algorithm written for a single-machine execution would decompose the workload into *equal parts*, which is good for a homogeneous environment but very inefficient for a Grid environment.

The $n-$tuple of numbers which specify the number of processors per dimension is known as the *processor topology*. In Figure 2.1 where the dashed lines indicate a domain decomposition for four processors we say that we have a 2×2 processor topology. Like the domain decomposition, the processor topology even for the same number of processors is not unique. For four processors one could also think of a 1×4 or 4×1 processor topology.

2.3 Ghost Zones

Points needed for calculations by a neighbored processor have to be send across the connecting network to the local processor which usually happens in an MPI call. Since there may be many of those points there will be many MPI calls. In order to avoid this one creates local buffers and copies all points needed (usually the points of the whole "face") into that buffer which is allocated on the local processor. This buffer is called a *ghost zone* containing *ghost points*, because they appear like real grid points for the application on the local processor but they in fact are not. Figure 2.2 illustrates this. Regular grid points are black whereas ghost points are white. This configuration is good for a finite differencing code using a stencil size of 1 which means that for a proper calculation (update) of a grid point it needs the values of the directly neighboring points in a given direction. A code having a stencil size of two would need two neighboring points in a given direction. This would require a ghost zone which is two points wide. We then speak of a ghost zone size of 2. Whether a code has a stencil size of 1 or more depends solely on the numerical scheme used in the finite differencing code and cannot be influenced by any parallelizing algorithms. Numerical algorithms with higher accuracy tend to have a bigger stencil sizes.

Our illustrations always show two-dimensional discretized functions where the ghost zone is a line of points. For a three dimensional computational grid it would be a plane of points, for a four dimensional grid a cube of points etc.

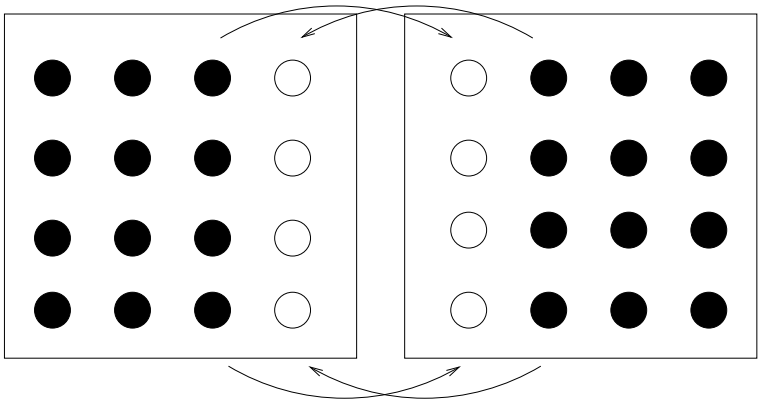

Figure 2.2: The ghost zone concept. A two dimensional grid distributed across two processors. The processor "face" consists of four grid points which are copied into the corresponding ghost zone on the local processor. The finite differencing code can loop through the computational grid on each local processor and properly calculate all real grid points (black) whereas the ghost points cannot be updated. They rather have to be sent from the neighboring processor.

2.4 The Cactus Framework

Writing a code that solves differential equations using domain decomposition methods, initial data solvers, IO and the actual evolution code can be very time consuming. It would be very convenient to have a modular framework where each of the above components (evolution code, driver code, IO code etc) would be represented as a module so that the user could compose his/her numerical code according to his/her needs. Such a framework exists and is called the *Cactus Computational Toolkit* [6]. It has been developed at the Max-Planck-Institute for Gravitational Physics in cooperation with researchers world wide. The development of the Cactus code was also driven by the need of keeping numerical physicists or engineers away from issues like parallel programming, MPI, and other things which are not directly related to numerical sciences. The basic idea behind the framework is to provide an easy but effective way to parallelize Fortran, C or C++ codes written by numerical scientists.

The name Cactus comes from its design where a central flesh represents the core of the software and various "thorns", i.e. the above mentioned modules can be easily plugged in or out according to the needs of the scientist. Since the flesh is a very generic code the thorns can be application codes (finite difference codes written in C, C++ or Fortran) or driver thorns, which are providing parallelizing algorithms, IO, checkpointing, visualization and other features. Since, as said above, thorns can be plugged in and out it is possible to replace the parallelizing driver code with another one without the need to change the application code. On the other end of the spectrum also application thorns can be replaced without changing the driver thorns. In this sense the Cactus code has to be seen as a complete framework, rather than as a simple library.

This opens a variety of possibilities for application programmers and also for driver code developers. Since this thesis is about new Grid aware parallelizing algorithms, the Cactus framework is an ideal playground to quickly test and implement new algorithms with real world applications and immediately compare with old codes. Current applications in the Cactus framework cover different domains like astrophysics, climate models, computational fluid dynamics etc.

2.5 Classical Uni Grid Parallelisation Schemes

The method of solving (partial) differential equations on a regular grid are called unigrid methods. In this Section we will describe these classical unigrid algorithms which are used to parallelize codes for a single machine or cluster execution. The implementation of these algorithms are currently done by Cactus' unigrid parallel driver thorn, which is called PUGH[3]. The way the parallelization is done can be seen as representative for the big class of finite difference codes.

Currently the code supports the parallelization of finite difference codes on a regular grid, i.e. all discretized functions have the shape of an $n-$dimensional cuboid.

PUGH divides up the entire computational grid into subgrids, each located on one processor. The numerical code running on top of Cactus is completely unaware of all decompositions as it always performs loops in every dimension from 1 to nx, ny, nz, ... and so forth. According to the domain decomposition the values for the local nxs are changed by PUGH adequately without explicitly notifying the application code.

One of the goals of this thesis is to generalize classical parallelizing algorithm to include the efficient execution in a heterogeneous and dynamic Grid environment. In order to understand those changes to the driver code we describe in this chapter PUGH's parallelizing algorithm in more detail. Like most other algorithms, it has been originally written for single machine or cluster execution.

More precisely, the assumptions under which PUGH was written are the following:

- all processors are of same kind, in particular they run at the same speed
- all network interconnects between processors are of same kind
- bandwidth and latency between processors are sufficiently high or low respectively
- the network characteristics between processors do not change in time

2.5.1 Processor Topology

For a n-dimensional computational grid recall that the processor topology is an n-tuple of numbers $k_1, k_2, \ldots, k_n$ where $\prod_{j=1}^{n} k_j = N$ where N is the total number of processors.

Since all interconnects are regarded to have the same bandwidth/latency characteristics and to be sufficiently fast, the goal of a topology creating algorithm is to distribute the processors more or less equally across the dimensions.

Thus the input parameters for this algorithm are only the number of processors and the dimension:

[3] stands for Parallel UniGrid Hierachy

```
Algorithm: CreateTopology(nprocs, dim)
{
  remaining = nprocs
  for  (k = 0 to dim)
    {
    procs[k] = IntegerRoot(remaining,k);
    while (remaining not divisible by procs[k])
      procs[k]- -
    remaining /= procs[k]
    }
return procs[k]
}
```

`IntegerRoot()` is a routine which returns the highest integer below a given integer root of an integer.

Example Let the dimension be 3, the number of remaining processors 32. The `InterRoot(32,3)` returns 3 since the third root of 32 is 3.1 and the highest integer below it is 3. As 32 is not divisible by 3, the above algorithm decreases this number until 32 is divisible by it, which in this case is 2. Therefore the first dimension gets 2 processors and for the remaining 2 dimensions 16 processors do remain. `IntegerRoot(16,2)` now returns 4 (as the square root of 16 and since 16 is divisible by 4, the second dimension gets 4 processors leaving again 4 processors for the last dimension, resulting in a processor topology of $2 \times 4 \times 4$.

The next table shows the processor topologies for 3 dimensions for some processor numbers.

N procs	topology
1	$1 \times 1 \times 1$
2	$1 \times 1 \times 2$
4	$1 \times 2 \times 2$
8	$2 \times 2 \times 2$
16	$2 \times 2 \times 4$
32	$2 \times 4 \times 4$
64	$4 \times 4 \times 4$

This table clearly shows the intention of this processor topology generating algorithm: To distribute the number of processors equally across the dimensions.

2.5.2 Processor Ordering

The processor topology in the last section calculated the number of processors per dimension but it did not specify which particular processor – i.e. which particular MPI_Rank, see 3.1 – is exactly located where in the $n-$dimensional topology.

In classical parallel computing all processors are assumed to be of the same kind and thus the particular processor number is not of importance.

To be able to exchange information between processors however, every processor is required to know the rank of all of its neighbors. The fact of knowing all neighbors of all processors however, determines a specific and unique mapping of processor ranks into the processor topology.

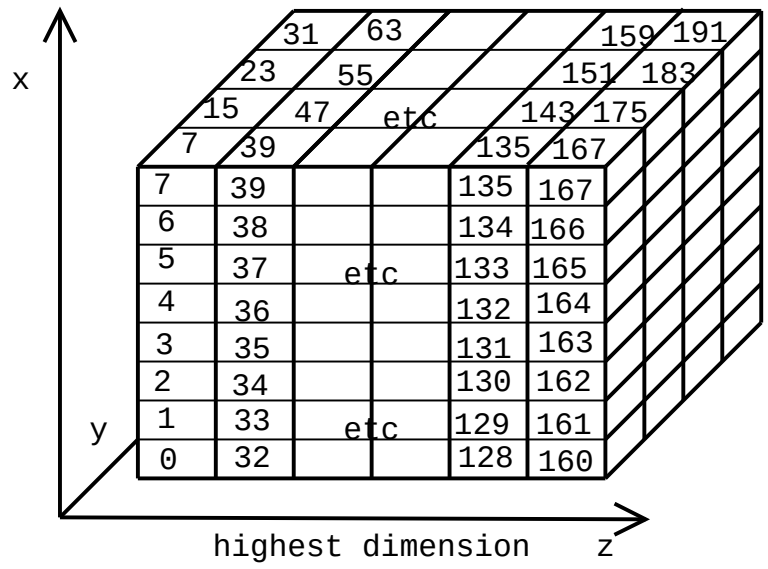

Figure 2.3: The ordering of processor numbers (MPI_Ranks) in a three dimensional setup. The ranks are first counted up in the lowest dimension, here x, then in y and finally in the highest (z) dimension.

The way PUGH distributes the processors is therefore the simplest one can assume, starting with processor 0 as the "first" processor in all dimensions, processor 1 as the second in the first dimension etc. ending with processor N as being the last processor in all dimensions, where N is the total number of processors. Figure 2.3 illustrates this for a three dimensional setup. It is important to emphasize however that rearranging the processors (as long as the neighbor information is regenerated) does not have any significance in standard parallel computing, since every processor is regarded to have identical properties.

2.5.3 Domain Decomposition

After knowing how many processors are located in each dimension one has to distribute the workload across these. As PUGH assumes that every processor runs at the same speed, the global grid size is divided by the number of processors in each dimension. Thus every processor gets the same number of points in each dimension, modulo points which remain because the total number of points per dimension is not divisible by the number of processors in that dimension.

2.5.4 Ghost Zones and Synchronizing

As said in the previous section, the points of the computational grid are equally distributed across the processors. However, as described in Section 2.2 we need to add a set of points to represent the ghost zone. Thus every processor that has a neighbor in a given direction gets an additional row of points. The (initial) ghost zone size is given by the stencil size of the finite differencing code.

The only information the application code has to provide to PUGH is when and which discretized function has to be "synchronized". By this we mean to update the ghost points with real

data from the neighbored processor. The application code then calls a function like `sync(`*`function`*`)` which is provided by a driver code. The way PUGH is synchronizing functions is the following:

```
Algoritm: Synchronize()
{
  int dim; [dimension of the grid]
  for  (k = 0 to dim)
    {
    while (processor has a neighbor in +dim or -dim)
      {
      copy real grid points into a send buffer
      post an MPI_Irecv
      post an MPI_Isend
      }
    post an MPI_Wait for all directions in this dim
    copy arrived data into the ghost zones
    }
}
```

The code loops through all dimensions, checks whether there is a neighbor and posts asynchronous MPI commands to initiate the communication and finalizes it with an `MPI_Wait()` call. Note that the synchronization is done dimension wise and the communications only overlap when sending/recieving data in the positive or negative direction of a given dimension. Overlapping communication in different directions will be discussed in Chapter 6.

2.6 Extensions and Replacements of PUGH

PUGH is a unigrid driver module ("thorn") in Cactus, as described in the last sections. There are several possible branches of extension for this module. One possible way is to replace it with a parallel *fixed mesh refinement* (FMR) or even a parallel *adaptive mesh refinement* (AMR) driver. Instead of using one global grid (that is distributed across the processors) these methods use more levels (hierarchies) of grids with different resolutions which do not cover the whole domain, but rather are form a sort of patches for specific regions. An FMR thorn for Cactus is currently under development ("Carpet") as well as a AMR thorns (using GrACE [53]).

An other possible branch of extension is to make the parallel driver Grid aware. First steps have been taken by allowing a heterogenious grid point distribution in PUGH on a parameter file entry basis [58]. The analysis and implementation of a whole range of Grid aware extensions are the subject of this thesis.

Chapter 3

Performing Distributed Runs

In this section we will perform and analyze some distributed runs using a Grid aware message passing protocol. Before we can do that, we need to choose an adequate communication software that will run underneath the application and driver code level. We will focus on the implementations of the MPI standard and analyze their "Grid awareness". After we have decided to choose an implementation we will use it to perform some sample distributed runs and discuss major problems.

3.1 Message Passing Libraries

There are dozens of possibilities to communicate data among processes or processors. One could use sockets, shared memory operations, TCP/IP to mention just a few. It certainly depends on the underlying hardware which method is the optimal one.

However, a developer of parallel codes does certainly not want to deal with these issues. Instead, he/she wants to use standardized message passing libraries that hide all unneccessary details of underlying communication channels.

The most popular and most used message passing library is the *Message Passing Interface*, MPI. There are many implementations of this library, but its basic functionality is defined by the *MPI Standard* [69].

MPI provides basic communication operations between a pair of processors (point to point operations) like MPI_Send and MPI_Recv (see the above algorithm). For a whole set of processors MPI provides *collective operations*. The set of processors participating in such a collective operation is called a *communicator* in MPI. Processors are numbered, and those numbers are called *ranks*. For a basic introduction into MPI see [69].

Apart from MPI there is another known standard for communication among processors in a parallel code which is called *Parallel Virtual Machine*, PVM [64]. For a discussion about the differences between MPI and PVM see [33].

The way such message passing standards hide hardware communication details to the user, the Cactus framework hides the message passing standard from the "user". Which message passing library is currently used depends solely on the driver thorn loaded.

3.2 Grid aware MPI Implementations

The most known MPI implementation is the Mpich-package [37, 50]. Its development dates back to the early 90's where William Gropp and Ewing Lusk developed a free and portable implementation of the MPI standard. The main two goals of this software and research project are to achieve high performance **and** portability. As commonly known these two notions are often contradictory. The way out of this problem was to create a so called *abstract device interface*, ADI, within the Mpich package. According to various types of low level communication like sockets, shared memory, Gigabit Ethernet, Myrinet, vendor's MPI implementations, nexus etc. different *devices* have been created within the Mpich package to support each of those communication channels. The decision about what device will be selected happens at the time Mpich is installed (compiled). Switching between devices on the fly is not possible.

The following table maps the different Mpich-devices to the underlying physical communication networks.

p4	sockets
shmem	shared memory
gm	Myrinet
gamma	Gigabit Ethernet
globus	nexus
globus2	TCP/IP and vendor's MPI

The gamma device has been developed by the University of Genova to support fast communications on clusters with fast or Gigabit Ethernet interconnect [32, 14]. For Gigabit Ethernet it is able to achieve bandwidths of more than 120MB/s [15], which is close to the physical limit of what a Gigabit Ethernet network can achieve. The gm device is to support Myricom's Myrinet physical interconnect.

In the following we give a brief overview about some Mpich devices and the possibility to use it for Grid computing.

3.2.1 Mpich, p4 and shmem Device

The p4 device uses sockets, i.e. TCP/IP to pass messages between processors. Since this is a very general and portable way of communication it can be installed and used on clusters, supercomputers, collection of workstations and even collection of supercomputers.

As such it can be used for Grid computing. The downside of this general communication channel – as pointed out earlier – is the performance. Since all communications, even on one supercomputer, is done through sockets it will be orders of magnitude slower than other adequate MPI implementations. For example, the vendor's supplied MPI implementation on an Origin 2000 computer can achieve a bandwidth of more than 100MB/s, whereas the TCP bandwidth across a Fast Ethernet interface cannot be faster than the physical Ethernet limit of 12,5 MB/s.

But the Mpich-p4 device is not as unqualified for Grid computing as it now seems. Another device, namely the shared memory device shmem can help to use the p4 device in some Grid-like environment. The shmem device as such cannot be used as a Grid aware implementation since it exclusively performs communications via shared memory. In conjunction with the p4 device, however, it can help to use shared memory for communications between processors within one shared memory machine and to use sockets between machines since it is possible to install Mpich

for the usage of more than one device. In order to achieve this, one has to specify it at the time one configures the Mpich installation. The appropriate line should look like:
`./configure -device=ch_p4 -comm=shared`

This can be used for running distributed jobs across a network of workstation with two or more processors per workstation or even for a distributed run between two Origin 2000 machines (that do have a shared memory). However, there are still problems which are left to the user to solve, like authentication, handling different account names, directories and so forth which is not handled by Mpich.

3.2.2 Mpich, Globus Device

The Globus device was one of the first MPI implementations that was specifically written for the purpose of executing codes in a real Grid environment [25]. It does not directly use sockets but uses the nexus communication library instead, which was also developed by researchers of the Argonne National Labs [29]. Nexus itself uses different communication channels underneath like sockets or vendor supplied communication libraries. The code which wants to use this implementation has to be recompiled and linked also against the Globus libraries.

As the name says, Mpich-G uses Globus. What does this exactly mean, and what is the benefit for the end user? Roughly, it means that the user has all features of the Globus toolkit at his/her disposal. The main and most helpful features are:

- Authentication: Mpich-G uses Globus to authenticate. The user does not have to deal with ssh-keys or rsh or other methods.
- Staging: the user can use Globus to stage executables.
- GASS[1]: can be used to (re-) direct standard output or standard error to the local machine and/or stage executables or parameter files to the remote machine.
- GRAM[2]: is used to provide a uniform access to queue systems, using a standardized language (resource specification language).

Mpich-G performed better than Mpich-p4 [12] but could not get as fast as the native MPI implementation on a supercomputer. This led to the development of a new next generation device which would use vendor's supplied MPI ("VMPI") on a machine where it is installed and TCP/IP between machines. The device which is capable of this is the globus2 device.

3.2.3 Mpich, Globus2 Device

From the performance point of view a Grid aware MPI implementation should always use the fastest communication channel available. It is almost impossible to implement a portable version of such a code, but the Mpich-G2 developers applied the following trick: For wide area network communication, they use TCP/IP. Instead of implementing fast communications for intra-machine or intra-cluster communication on top of various physical connections they simply assumed the existence of an already installed MPI implementation which makes use of fast connections underneath.

[1] Global Access to Secondary Storage
[2] Globus Resource Allocation Manager

By this the selection of the fastest possible way for communication is mostly assured. On commercial supercomputing systems (like SGI's Origin 2000, IBM's SP-2, Cray's T3E etc.) it is almost impossible to have a faster communication than the one performed by the MPI implementation which is supplied by the vendor.

The problem of dealing with another MPI implementation *within another* MPI implementation is a quite subtle one, leading to name clashes and other inconsistencies. The way around it is to rename all MPI calls of a MPI application at compile time in order to be able to call VMPI routines later.

In order to compile an MPI code to be used with Mpich-G2 it is not enough (as for most other MPI implementations) to link against the appropriate libraries at the linking stage. The Mpich distribution traditionally provides socalled `mpicc` scripts that take care about including and linking but their usage was not mandatory. For Mpich-G2 it is. In the following we explain why. For every user level initiated point to point MPI communication within an distributed job Mpich-G2 does the following:

- Check whether the two processes that want to communicate are located on the same machine
- If not, use TCP/IP
- If yes, make a call to the vendor's supplied MPI installation

Let's consider for example an `MPI_Send()` operation called by the user code. Let's assume that the two communication end points are located on the same machine. It is impossible now for Mpich-G2 to make another `MPI_Send()` since that would conflict with the previous (user level) call.

This is where the `mpicc` script comes into place. Rather than simply calling a C or C++ compiler underneath with the appropriate linking and including flags it first goes through the code and replaces every single `MPI_` call by an `MPQ_` call *on the source code level*. Thus the above conflict can be solved.

The downside of this procedure however is that on some systems it is not possible to run configure scripts using the above mentioned mpicc (or `mpiCC` or `mpif90` etc.) as the appropriate compiler (i.e. `./configure --with-cc=mpicc`). On some systems those compilers would produce executables which cannot be run without appropriate `mpirun` commands and as such would cause the configure script to fail.

The first effective and large scale demonstration of this new MPI implementation was done in April 2001 and is part of this thesis. It will be described in detail in Section 5.

3.2.4 PACX

Aside from Mpich-G2 there is another implementation using the same ways of communication in a Grid environment: TCP/IP for inter- and VMPI for intra-machine communication. It is called PACX (PArallel Computer eXtension) [31] and is developed and maintained at the High Performance Computing Center (HLRS) in Stuttgart, Germany.

When a PACX job is run across more than one supercomputer, a daemon is running on each computer as an additional MPI process responsible for the communication across the wide area network. Mpich-G2 does not use this approach, it rather opens a socket for every single communication across the wide area network for every MPI process. The existence of such a "forwarding node" has advantages and downsides. Given that a distributed job is supposed to run across a setup including a supercomputer or cluster that has one front node with a public IP address and

all compute nodes have private IP addresses. Having this additional MPI process running on the front node makes it possible to use this machine in a distributed job. Mpich-G2 would not be able to make use of such a machine. On the other hand, as we will see at the end of Chapter 4, having multiple TCP streams across the same physical network *can* increase the total bandwidth (namely the sum of bandwidths of all streams) by far.

Since PACX does not use Globus, the user has to deal with access to machines, authentication, queue systems etc. himself/herself. It can be done by using remote or secure shell. This way of starting the code seems rather copious and is per se not able to handle the following problem. As an example we consider the Origin cluster at NCSA. The front cluster is modi4.ncsa.uiuc.edu with a known IP address. After submitting the job into queues, however, the code will get started on other machines of this origin cluster (balder, aegir, forseti etc.) whose IP address differs from modi4's IP address and is not known in advance. The question now is: How to contact the job from outside if the IP address is not known in advance? The PACX solution is a so called server mode startup (or a lately implemented Globus version of a distributed startup) where all processors have to contact a central server over which IP address information is exchanged. The IP adress of the server itself has to be know at startup.

3.2.5 LAM

Apart from Mpich another well-known free and portable MPI implementation is LAM[3] [63]. For communications it uses two major methods namely TCP/IP and shared memory. Like Mpich LAM allows setups where those two kinds of communications can be mixed, i.e. it is possible to couple several SMP machines for one distributed MPI job and LAM would use TCP for communication between machines and shared memory for communication between processors on one SMP node. As for Mpich p4, LAM-MPI needs to be specifically configured to use shared memory within an SMP node. The correct configuration line is:
`./configure --with-rpi=usysv`.

3.2.6 Virtual Machine Interface: VMI

The Virtual Machine Interface [52] is more than an MPI implementation. It is rather a *messaging layer* which provides MPI support among other things. It has been developed by Avneesh Pant et al. at the National Center for Supercomputing Applications. VMI has been designed in a very expendable and general manner. Figure 3.1 illustrates the structure of this messaging layer. Drivers for various physical networks like Gigabit Ethernet, Myrinet etc. are provided for the application code that performs communications through one single API. The application code can dynamically load or unload those driver modules. Furthermore, VMI provides data stripping in the case that more than one network is possible to be used and even fail overs (in the case that a Gigabit Ethernet cable or a Myrinet switch gets damaged) and also compression modules can be loaded dynamically. Apart form the direct usage through an API the application can use regular MPI calls which themselves use the features of VMI.

The simultaneous usage of TCP/IP and e.g. Myrinet or Gigabit Ethernet makes an effective usage of VMI for Grid computing possible. Jobs can be started as Globus jobs (taking care about authentication) and VMI specific 'keys' (set as environment variables within a Globus rsl script) can be used to tell each VMI subjob that it belongs to a global and bigger MPI (or VMI) job. All

[3]Local Area Multicomputer

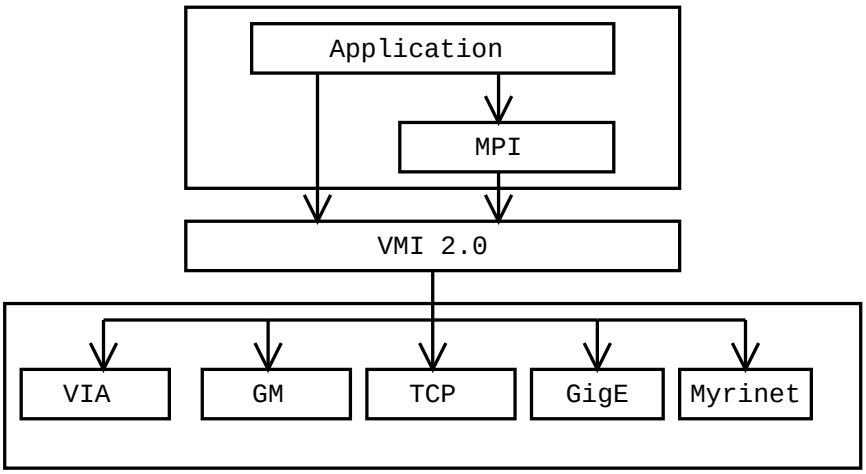

Figure 3.1: The **Virtual Machine Interface** as a multi layered communication library for applications supporting a variety of low level physical communication layers through one interface.

this is coded in a portable way which also supports execution across a heterogeneous setup.

VMI version 2.2 is installed on the titan cluster at NCSA (a 128 node [256 processors] Intel IA64 cluster, the second fastest cluster in the world in November 2001[4]).

3.2.7 PACX versus MPICH-G2

After the short overview of the "Grid awareness" of MPI implementations we have to choose one of those to perform distributed runs. Two of them have been specifically written for supporting high performance execution in a real Grid environment, namely PACX and Mpich-G2. Both variants are designed to support native MPI (VMPI) whenever available and TCP/IP between machines in a wide area network.

Also VMI should be mentioned which provides future pointing technologies and a variety of useful and needed features for distributed Grid computing. However the software, at the time this thesis is written, is still under development or at least in a testing phase. Furthermore, its design applies more to clusters (connected by GigE or Myrinet which VMI can make use of) than to supercomputers where we already have a fast MPI installation. Since clusters may replace supercomputers VMI will certainly play an important role in the future, e.g. in the TeraGrid (see Section 5).

We have chosen to use Mpich-G2 for several reasons. Roughly spoken the usage of Mpich-G2 is more convenient because of all benefits provided by the Globus usage underneath, namely

- authentication
- co-allocation services
- uniform access to queue systems

[4]The worlds fastest cluster according to the Top 500 list as of November 2001 is a self made cluster at the Sandia National Laboratories, USA

- Globus IO

With PACX the user has to handle problems like authentication and co-allocation himself. Mpich-G2 does not have to deal with it, since this is done by Globus. Even the problem of not knowing the IP address of a machine is solved without the need of user interaction.

Let us reconsider the problem from Section 3.2.4. A user wants to run a distributed job across three machines. One machine is a multiprocessor machine with different IP addresses per node, another machine is only the front end machine for another cluster of machines which the job is eventually run on after submitting to a queue system. How can the subjobs contact each other when the IP address of them is not known in advance?

PACX offers the following solution: The user has to start the job in a socalled "server mode" which means that a daemon with a known public IP address is started along with the job. This IP address is then known to the other subjobs and they now can exchange their contact information using this server.

MPICH-G2 solves this problem without any additional user action. When a distributed job is started using `globusrun` Mpich-G2 *can* obtain the IP address of every MPI process using Globus (and Duroc) IO at the stage where all subjobs have started. The way this works is exactly the same as for the PACX server startup case – but it happens automatically. The `globusrun` command namely starts a https server on the local client. The contact information (IP address, port number etc.) for the single Globus subjobs are distributed via environment variables which are set on the machines where the job is finally started. This explains why the `globusrun` job itself must not be killed until the entire job has finished. It also explains (now coming to the major downside of this procedure) why a Globus job cannot be started from a machine or laptop with a private IP address or behind a firewall.

PACX on the other hand does not need to have the complete Globus package installed and deployed at all sites, which has usually to be done by root (it is also possible to install and deploy Globus as a normal user, but experience showed that this is a very unhandy way of using Globus).

3.2.8 Running Parallel Codes with Mpich-G2

To run a distributed job (e.g. using Cactus) using Mpich-G2 we need the following:

- the Cactus source code, available at `http://www.cactuscode.org`
- Mpich-G2 installed for every architecture
- Globus installed for every architecture and deployed at every site one wants to run across

To configure and compile Cactus for distributed runs using Mpich-G2 one has to issue the following:

```
make <myname> MPI=MPICH MPICH_DIR=<mpich-G2-dir> CC=<mpich-G2-dir>/bin/mpicc
LD=<mpich-G2-dir>/bin/mpiCC
```

Cactus and the mpicc compiling scripts take care about including and linking the appropriate Globus libraries and include files (tested for Mpich version 1.2.1 and Globus version 1.1.4). Note that it is indispensable to use the mpicc scripts for every code which calls MPI routines.

After obtaining an executable one has to create a socalled rsl[5] script which specifies the resources one wants to use.

Befor starting the job the user has to get a proxy, i.e. an authentication ticket for the Grid (comparable with a Kerberos ticket). To get this, the user has to type

```
tom@origin~%  grid-proxy-init
Your identity: /C=US/O=Globus/OU=The National Computational Alliance/CN=Thomas Dramlitsch
Enter GRID pass phrase for this identity:
Creating proxy ................................................ Done
Your proxy is valid until Thu Sep 26 00:03:36 2002
```

and finally start the job using

```
tom@origin~%  globusrun -f my_script.rsl
```

A sample script could look like this:

```
+
( &(resourceManagerContact="harpo.wustl.edu:2119/jobmanager-lsf")
(count=8)
(label="subjob washu")
(maxMemory=2000)
(project=paa)
(queue=8pe_4hr)
(jobtype=mpi)
(directory=/u1/thomasd)
(environment=(GLOBUS_DUROC_SUBJOB_INDEX 1))
(executable=/u1/thomasd/cactus_g2)
(arguments="test.par")
(stdout=out)
(stderr=err)
)
(&(resourceManagerContact="modi4.ncsa.uiuc.edu:2119/jobmanager-lsf")
(count=8)
(label="subjob ncsa")
(project=tro)
(jobtype=mpi)
(maxWallTime=60)
(environment=(GLOBUS_DUROC_SUBJOB_INDEX 0))
(directory=/u/ac/tdramlit/Cactus/exe)
(executable=/u/ac/tdramlit/Cactus/exe/cactus_g2)
(arguments="test.par -r")
(stdout=/u/ac/tdramlit/Cactus/exe/out.harpo)
(stderr=/u/ac/tdramlit/Cactus/exe/err.harpo)
)
```

[5] resource specification language

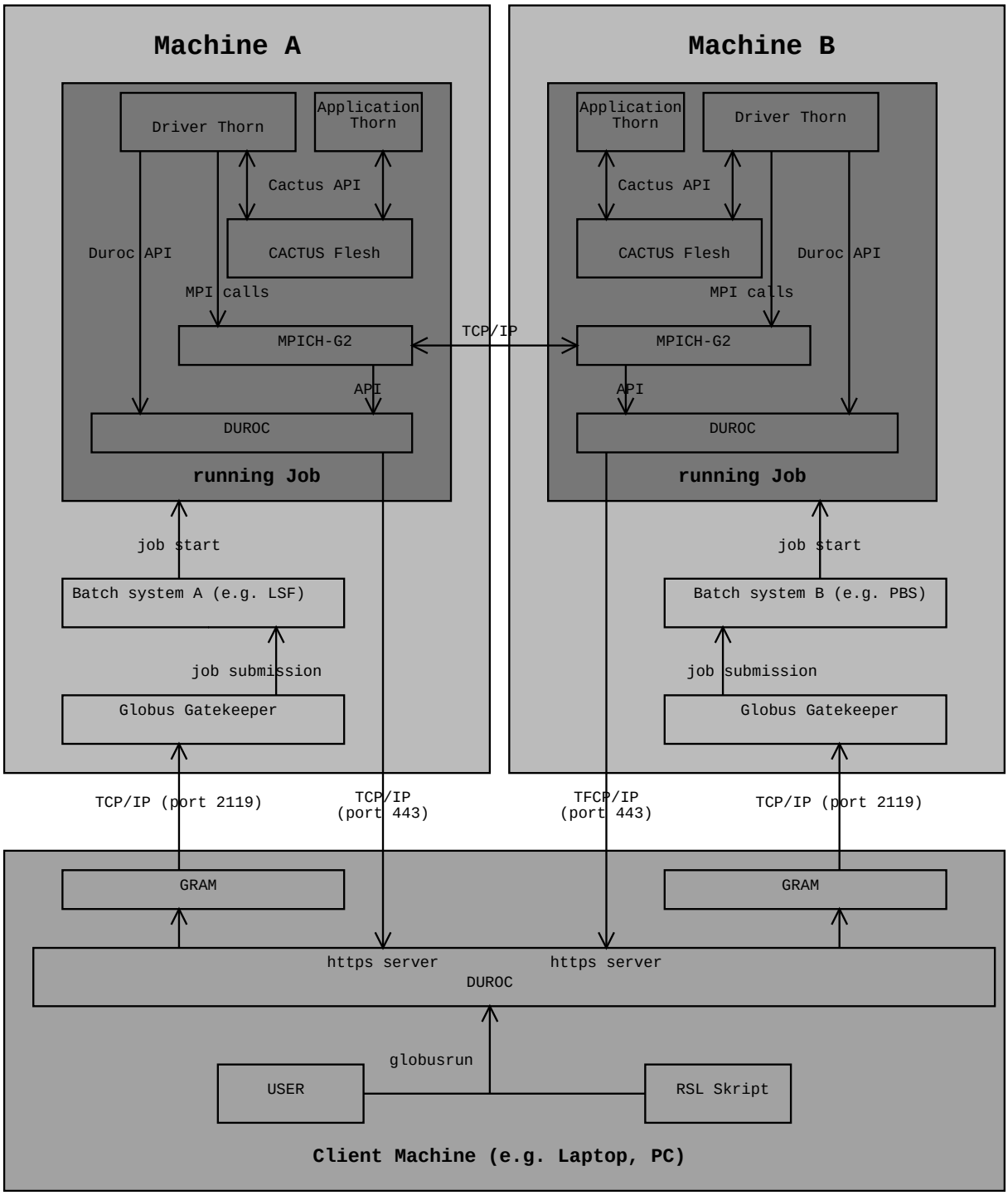

Figure 3.2: The architecture of a distributed run using Cactus, Globus and Mpich-G2. A user must (at the current development stage) have an RSL skript and executables in place. By either typing `globusrun` or call `globus_duroc_control_job_request()` from a running code, a Duroc process gets created that (using Gram) contacts the appropriate gatekeepres at each machine. Furthermore it starts a https server for information interchange with the finally submitted job. Contact information for the https server are transmitted through environment variables. In this architectural picture the implementations of the methods and algorithms discussed in this thesis will take the form of a *driver thorn*.

We will not go into the details of the Globus resource specification language which can be found at [57], but we will explain two lines in this script which are important for the internals of distributed computing with Mpich-G2.

The first important specification is `(jobtype=mpi)`. This triggers the usage of the native VMPI implementation on the machine in question. The requirement for this is that Mpich-G2 has been configured and compiled for VMPI usage and the requirement for this is that the Globus installation successfully built "MPI-flavored" libraries. Thus if one wants to use VMPI in a Mpich-G2 job one has to take care of it at the time of the Globus installation.

If there is no VMPI on a machine available, or it cannot be used, the flag has to be set to `(jobtype=multiple)`. This triggers the usage of TCP for communications between CPUs even in one subjob.

The second important specification is the environment variable GLOBUS_DUROC_SUBJOB-_INDEX. Its value has to be an different integer for each subjob starting at 0. This variable specifies how the MPI ranks are distributed across the machines. The subjob with the GLOBUS_DUROC-_SUBJOB_INDEX 0 contains the global MPI rank 0, the other rank numbers are counted along this index, i.e. the subjob with the highest index contains the highest MPI rank.

As mentioned before, we will use Mpich-G2 as a Grid aware MPI implementation for our distributed runs since we think that it is the most capable implementation in respect of performance and portability. Figure 3.2 illustrates the interaction between Cactus, Globus and Mpich-G2.

Up to now we did not discuss *how* all information can be acquired that is neccessary to compile such an rsl script. At the current stage, all information has to be found out manually. This concerns e.g. the type of batch system (e.g. jobmanager-lsf or jobmanager-pbs) the available amount of memory on the machine or if there is a VMPI installation on that machine.

Finding out whether Mpich-G2 has been configured to use VMPI underneath is even harder. To our knowledge, ther is no user level method to find out which intra-machine communication is used by Mpich-G2.

However, several efforts are currently worked on to easy access these informations. As discussed earlyer, Globus comes along with the MDS toolkit, an LDAP based server that collects information about machines and networks. Each Globus enabled machine can register with such a server and publish any kind of information to it (i.e. number of processors, memory, queue system and such).

A related PhD thesis(Lanfermann, 2002 [43]) designs a *Grid Object Description Language* (GODLS) that generelizes and standardizes information about "objects" on the Grid. An object can be anything that "lives on the Grid" e.g. a machine, a file, an operating system, a firewall etc. An Application Information Server handles requests from running codes that look for Grid objects with particular properties.

The above described problem of not knowing wheter to choose `jobtype=multiple` or `jobtype=mpi` are thus solved by the Application Information Server.

3.2.9 Application Codes

The application code which we will run within Cactus is a code that calculates the time evolution of objects in the general theory of relativity (i.e. neutron stars, boson stars, black holes or similar), which is very communication intensive and represents the state of the art in this area. It solves Einstein's field equations of the general theory of relativity. More precisely it calculates the time evolution of a given initial dataset. Exactly this code was used to obtain latest results in this area of research [3, 4]. It uses finite differences to compute the time evolution of the fundamental quantity

in general relativity namely the components of the metric tensor (along with other quantities). The numerical method used here is a Iterative Crank Nicolson scheme using three iterations [16]. The initial data we use in our test runs is a set of "Brill-Lindquist black holes" [1] which is a known analytical solution of Einstein's equations. By an analytical solution we mean a set of discretized functions which represent a known function (e.g. $f = x + y/z$), as opposed to initial data sets that are obtained numerically by solving elliptic or other equations. Those calculations are needed to extract gravitational waveforms of the calculated "cosmic events" in order to compare it with waveforms still to be observed at large gravitational wave detectors [71, 2]. A match between calculated and observed waveforms would finally confirm theory of general relativity. This confirmation of Einstein's theory is a major goal in theoretical physics.

Again, we strengthen the fact that we are not using test codes but real production codes. This is very important since real production codes mostly have different communication and computation characteristics than test or benchmark codes which mostly have simplified characteristics. By focusing on production codes, the methods developed in this thesis can be brought back and immediately used by the science community.

3.2.10 Testbeds and Certification Authorities

The Globus authentication scheme uses X509 certificates to identify a user on the Grid. Those certificates are signed by a certification authority's (CA) private key. To submit a job to a machine where a Globus gatekeeper is running the user needs a local account and a certificate which is signed by a CA which is trusted by the machine. The gatekeeper knows the CA's public key and can therefore verify the users authenticity.

The most known certification authority is the one that is run by the Globus developers themself (ca@globus.org) but there are also others (Alliance certs used for the NPACI sites, DoE certificates etc.).

For a testbed to perform distributed computations – like the ones in this thesis – the user needs

- A local account at every site
- A certificate signed by a certification authority accepted at every site.

A site can be a member of more than one testbed. For example the Origin 2000 at the Albert Einstein Institute accepts Globus as well as Alliance and other certificates. The collection of sites one has access to with a specific certificate can be called a *virtual organization*.

3.2.11 Co-Scheduling

In order to run codes efficiently across more sites they have to start up at the same time. For bigger jobs this can only be achieved when the schedulers at both sites talk to each other. We call this co-scheduling.

The only scheduling system which is used in production environments we currently know of that *could* support a primitive kind of co-scheduling is the Maui scheduler [48] where it is possible to schedule a job at an absolute time, e.g. from 3pm to 6pm on the 21^{st} of June 2002. If the batch system underneath is able to meet this condition the job will be submitted and started at the requested time, if not then the submission did not succeed. Theoretically this is independent of the fact whether this is a multi site job or not (In regular cases this kind of advanced reservation is used to prepare demonstrations or for administrative work on the machine). A multi site request

requiring multiple resources at the same time (e.g. 9am in two days) can be regarded as a primitive form of co-scheduling. Unfortunately this does not properly interact with the Globus toolkit since it does not support the usage of Maui schedulers (more precisely its resource specification language does not contain commands to talk to the Maui scheduler).

Another batch and scheduling system which supports real co-scheduling even across remote locations is the Load Sharing Facility (LSF) as of version 5.0 [46]. However, at the time writing this thesis this software still was not released.

Thus, since currently there are no sophisticated systems which support co-scheduling with Globus we have to restrict ourselves to the submission jobs without co-scheduling, meaning that it can take a long time for both jobs to start. When a Globus job gets started by a queuing system it waits in the so called `duroc_barrier()` routine until every subjob starts up. A job sitting in the `duroc_barrier()` is considered by any queue system as a running job. Thus in the worst case one job may already exceed its computing time while another is still sitting in the queue.

However, if the requested processor number is small, the job might start immediately at all/both sites. This unfortunately restricts us to a rather small processor number. With the current load in queue systems at supercomputing sites like SDSC, NCSA, WashU or others we are able to start distributed runs with at most 8 processors per site on a more or less regular basis.

At some occasions however, like for Supercomputing Conference demos or for TeraGrid preperation runs (Chapter 5) we had the chance of running distributed jobs at a large scale. These singular events helped us a lot to gain experience and on that basis we were able to develop and implement techniques (Chapters 6 and 7) that we than tested at the previously described scale (∼8 processors per site).

3.2.12 Example runs

Our main testbed comprizes the Origin 2000 at the Albert Einstein Institute in Potsdam, the Origin cluster at NCSA, Blue Horizon at SDSC and another Origin 2000 at Washington University in St. Louis (WashU). Recalling the co-scheduling problems described in the last section we found that the chances to have a distributed job starting right away are very high for runs between Potsdam and WashU (having up to 8 processors per site). Figure 3.3 shows the time evolution of the total performance, measured in Mflop/s, of a distributed run across 16 processors, 8 located in Potsdam, 8 in St. Louis. Taking into account that this run would perform at 1.4Gflop/s on one origin it turns out that the performance achieved here is very poor (around 10% efficiency). Other distributed runs we have executed show a very similar performance behavior. We ran exactly the same application code across an origin at NCSA (8 processors) and Blue Horizon at SDSC (8 processors) and achieved a performance in the same order of magnitude, about 200Mflop/s. Given the fact that Blue Horizon is about twice as fast as on origin 2000 (setting 70 and 140 Mflop/s for the single CPU performance of the origin and Blue Horizon respectively) the speedup of this code can be calculated using equation 2.2:

$$S(16) = \frac{200}{\frac{8}{16}70 + \frac{8}{16}140} \approx 2$$

A speedup of 2 on 16 processors is equivalent to a efficiency of 12%. Although this is a very poor number, we emphasize that we did not apply any special methods or enhancements to any part of the software used for this. We also want to emphasize that these runs show that such a distributed calculation with standard software is possible at all.

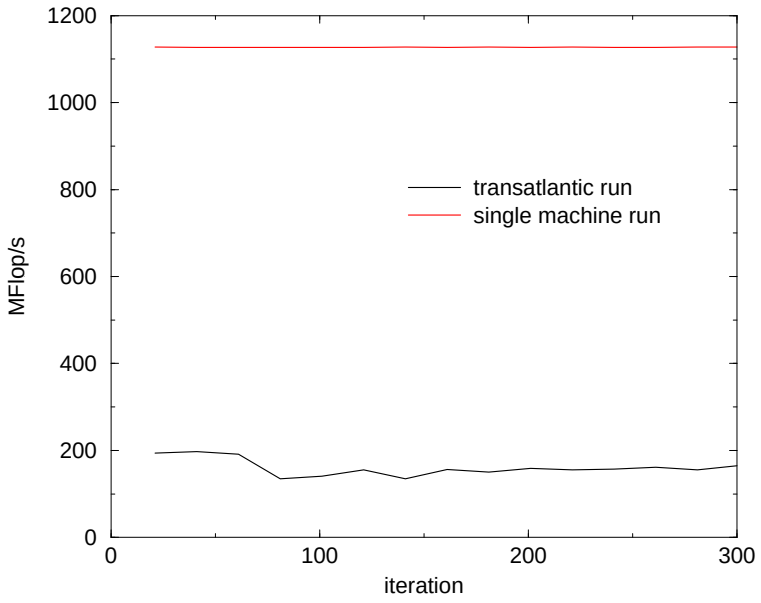

Figure 3.3: The performance of a standard distributed run between origin.aei.mpg.de and chico.washu.edu, both origin 2000 systems. The latency between the two systems is about 65ms with a bandwidth of less than 400kb/s. This graph shows a run across 8+8 processors. If one takes into account that the performance of the same run on one processor would be about 1.4Gflop/s we achieve an efficiency of about 11% for this distributed run of about 160 Mflop/s (and an efficiency of about 80% for the single machine run).

But we also see why distributed computing in a real heterogeneous Grid environment does not belong to a scientists daily work: Apart form the practical problems described in the next section the resulting performance does not justify the execution of codes in such an environment. Given the efficiency of around 10% in the previously discussed runs it would be much more convenient and efficient (as well as cheaper, faster) to run it on two processors of a local cluster with fast interconnects.

It is a major goal of this thesis however to investigate possibilities and techniques to improve the efficiency of such codes to bring distributed Grid computing closer to a scientists daily work.

3.3 Practical Problems in Grid Computing

In this section we very briefly describe general and practical problems when one wants to perform distributed runs in heterogeneous environments.

Although one has to emphasize that the Globus toolkit addresses and solves many of those structural problems there are still many practical and structural problems left which need to be solved and are not addressed in this thesis. These problems include:

Lack of a common file system. In classical parallel computing on clusters or supercomputers we have several file systems which can be accessed from every node and processor. On supercomputers we mostly have a local fast data disk for storing runtime data and other mounted file systems which can be commonly accessed. Many applications however require a common file system for e.g. parallel IO, checkpointing, recombining etc. Another very related problem is that due to the

lack of a common file system we have

Multiple software trees. Since we don't have a common file space we have to maintain several copies of the same software tree on each machine. Currently we don't know of any tool which is specifically designed to consistently maintain multiple copies of such a software tree. Similar tools like CVS, rdist etc. do exist but turn out to be unsuitable for this problem. For example, every little change which is made to some line in some source code would have to be committed to CVS and then checked out on the other machines. Even if we successfully manage to keep a consistent copy of our software on all machines it happens often that one forgets to compile on one machine leading to another inconsistency. What we need here is a software that checks consistency and recompiles on the "make" level, i.e. we need a kind of *distributed make*, which is currently addressed by Feider [22].

Executables. From the above said it follows that it can easily occur that the executables in question do not match for some reason. There is in principle no way to properly verify that all executables on all sites are "the same" (since it may be a different operating system). This phenomenon is completely unknown to classical parallel computing. In our test runs it happened very often that the executables did not "match", which results in the fact that the code simply 'hangs'.

Parameter files. The same goes for parameter files. Users change them quite often and this again leads very easily to inconsistent parameters within the run. As being text files the consistency of those can be checked.

Debugging. Debugging parallel programs is harder than debugging serial codes and debugging programs in a Grid environment is even harder. While for the first problem a lot of software for parallel debugging exists, there is currently no software to support "Grid-debugging". The only solution we know up to now is to simply attach to a running process using a standard debugger like gdb or dbx (if one has access to the machine the code is actually executed).

Precisions and pointers. Different machines may have different ways of representing integers, pointers or floating point numbers (as already mentioned in 6.1). For example a pointer on a 64 bit architecture is 8 bytes long while a pointer on a 32 bit architecture is only 4 bytes long. We experienced many problems when running a mixed 64 and 32 bit code across two architectures. Also one has to be very careful when sending integers in a Grid computing environment since they may have different size on different systems (e.g. 8 bytes on a T3E while 4 bytes on an SGI origin).

Firewalls and private networks. Firewalls are the biggest enemy of Grid computing but they do not completely prevent it. Mpich-G2 and Globus can be restricted to use a certain port range. Due to security but also due to a lack of public IP addresses it often happens that only the front node of a cluster has a public IP address while the compute nodes have a private one. Performing distributed runs with such a system would require to install a "forwarding node" on the front node where all communication with remote sites goes across.

All those and many other reasons may lead to the fact that a code runs in parallel on one machine or cluster but does not properly run in a real Grid environment. These issues need to be resolved, but are beyond the scope of this thesis.

3.4 Summary

In this chapter we shortly introduced MPI as a message passing library standard. For the purpose of Grid computing runs we analyzed which implementation of the MPI standard fits best our needs. After choosing Mpich-G2 as the best candidate to perform distributed runs in a Grid environment

we performed some first test runs using real world applications. We saw that runs are possible, though very low performing. Also, we experienced a couple of other problems related to executing codes in a Grid environment, which are not addressed by this thesis.

Chapter 4

Performance Analysis of Distributed Computing

4.1 Motivation

In the last section we described how distributed Grid computing across multiple supercomputers can be done. More important is the question whether this gives us any benefit at all, in other words how good is the efficiency of the running code. To answer this question we now build and apply a model which can predict the performance of a code (that is executed in such a Grid environment) in advance. By Grid environment we mean – more specifically – a collection of supercomputers (or clusters). The model used in this Chapter is very similar to the one developed by Ripeanu [59]. However, it is not the goal of this Chapter to develop a new theoretical model, but rather use existing models and current codes to learn and understand performance issues of distributed runs in order be able to develop techniques for performance improvement on that basis.

Our motivation basically comprizes of the following issues. First, a predicted low performance keeps the user from entering this territory of configureing and running those distributed jobs. Second, modeling is informative and helps us to understand better all performance related issues on all levels of hard and software. Third, having a (even rough) model we are able to identify issues that are needed for an easy and effective execution of distributed codes on the Grid. These information are important for future investments, i.e. what special characteristics should machines, operating systems and networks have in order to support Grid computing.

4.2 Single Processor Performance

In this study we do not model the single processor performance. Of course it can be modeled using many input parameters like local grid size, array sizes, cache size, cache line size, processor speed etc. But this is not our goal. Instead, we simply use the single processor performance as an input parameter to predict the Grid environment performance.

4.3 Single Machine Performance

For the class of codes which run within the Cactus framework we start with a simple model.

We introduce a simple model to predict the performance of a typical application running within

the Cactus framework. This model makes the following assumptions:

- the workload on each CPU is exactly the same
- the code spends time solely with pure computation or communication
- network characteristics between pairs of processors are identical
- all processors finish their work at the same time
- all processors are dedicated (no competition with other processes)

Under those assumptions it is easy to compute the efficiency of an example code with known characteristics. We choose an optimized version (application-level optimized) of a standard code from numerical relativity ("benchADM"). We keep the problem size per processor constant and run the code on different numbers of processors (1,2,4,8,16,32) on an origin 2000 system (modi4.ncsa.uiuc.edu).

From the assumptions above and using (2.1) we calculate the efficiency for running the code on different processor numbers by dividing the time for a single CPU run T_{single} by the (wall clock) time for an $n-$CPU run T_{total}.

In order to *predict* the efficiency for those runs we only have to know the pure computation T_{comp} time which is simply given by the single processor performance which again is given as an input parameter. The communication time T_{comm} is what we need to calculate.

To be able to do this we need to know the interconnect characteristics of the origin 2000 systems. We found a bandwidth of 127MB/s and a latency of $10\mu s$ between processors using the mpptest code [36].

Our single CPU code has a computational grid size of 60^3, performs 10 iterations and needs 12.86 seconds for it. We emphasize that the single CPU version of our code (i.e. the Cactus code) does *not* contain any communication or parallelization overhead. This trick is commonly used to improve the efficiency of a parallel code: Introducing parallelization overhead even to the single processor code increases the numerator in equation 2.1 and thus the efficiency of the parallel executed code.

The n CPU versions also have a local subgrid size of 60^3 per processor (ghost zones are omitted in this model, see our assumptions) and the communication pattern corresponds to the one described in sections 2.3 and 2.4.

The set of points which has to be sent to the neighbored processor contains in our case 60×60 points. We send a group of six functions per iteration containing floating point data. The size of a double precision floating point variable is 8 bytes long.

For a two processor run we thus have to send $60 \times 60 \times 6 \times 8 \times 10 = 1728$ kb in total. To compute T_{comm} we have to take the latency of $10\mu s$ and the bandwidth of 127MB/s into account:

$$T_{comm} = \frac{1.728\mathrm{M}B}{127\mathrm{M}B/s} + 10 \cdot 10\mu\mathrm{s}$$

$$= 0.013606\mathrm{s} + 0.00010\mathrm{s} = 0.0137\mathrm{s}$$

yielding an efficiency of

$$Eff = \frac{T_{comp}}{T_{tot}} = \frac{12.86 - 0.0137}{12.86} = 99.98\%.$$

For more processors we have to take into account that the number of neighbors increases and along with this the amount of data to exchange. The table below shows the results of our calculations.

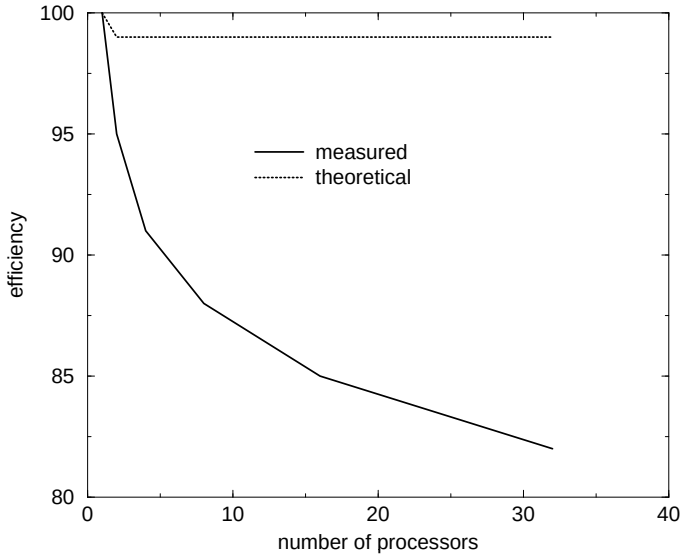

Figure 4.1: The efficiency of an executed parallel code on a SGI origin 2000 computer system. The dotted line shows the efficiency as calculated by our model whereas the solid line represents the real performance as we have measured it.

N procs	topology	efficiency
1	$1 \times 1 \times 1$	100 %
2	$1 \times 1 \times 2$	99.9
4	$1 \times 2 \times 2$	99.7
8	$2 \times 2 \times 2$	99.6
16	$2 \times 2 \times 4$	99.5
32	$2 \times 4 \times 4$	99.4
64	$4 \times 4 \times 4$	99.3
128	$4 \times 4 \times 8$	99.3
256	$4 \times 8 \times 8$	99.3

Since we assume that all connections between processors are identical the growing number of processors does neither influence the bandwidth nor the latency. Only the number of neighbors changes with higher processor number and reaches its maximum at 64 processors where it can happen that one processor has to send data to all its six neighbors. Then, according to our model the efficiency cannot get worse than in the 64 processor case, namely 99,3%.

Let's compare those numbers to the ones we have measured on a 48 CPU origin 2000 at NCSA (modi4). Figure 4.1 shows the big mismatch between reality and prediction. We repeated the measurements 10 times to obtain a statistical error of about 2% for each value which means that we must have made a systematical error in our model. We now have to go step by step through our assumptions and check whether and how this assumption is close to reality.

1. **Identical workload on CPUs** is true for our runs because we specifically instructed our code to maintain a constant load per processor (normally one specifies the global problem size rather than the local one and in that case the load among processors is not exactly the same) apart from

the fact that processors located on the boundary of the computational grid have one fewer ghost zones. For a 60^3 sized subgrid however one additional ghost zone increases the workload by 1.6%.
2. **The code solely computes or communicates**. This is also true for our code since it is not doing any time consuming IO or other things (like collective operations, file output etc.).
3. **Network characteristics between pairs of processors are identical**. This can be tested at least to a certain degree. We obtained a bandwidth of 127MB/s between two processors on the origin. We can also measure the *bi-sectional* bandwidth including more CPUs (The bi-sectional bandwidth includes the simultanious data transmission in both directions). As before we are using the mpptest program [36] which is coming along with the Mpich distribution. We run the code on 32 processors telling it to perform pingpong tests between the first and the last 16 processors simultaneously. However, we still obtain a bandwidth of about 124MB/s.
4. **All processors are used dedicated**. We ran our code in a busy environment so this point is definitely not true. In the system we used (Origin 2000) any user can log in and out to every node and start jobs anytime. Even if loging in is not permitted the operating system may swap processes from node to node and such disturb the computation. Combined with the fact that we assume that
5. **all processors finish their work at the same time** we at least know one reason why we are not observing the predicted efficiency.

However, it is possible to check whether point 4. and 5. is responsible for the observed discrepancy. We will redo the runs and use a *dedicated machine*. Furthermore we can tell the operating system not to swap MPI processes during execution (on Origin 2000 systems this is possible by setting appropriate environment variables, e.g. `MPI_DSM_MUSTRUN`).

We used the so called dedicated queues on the origin cluster system at NCSA. We ran the code on 64 CPUs and measured an efficiency of more than 97%. We did not perform this run 10 or more times to estimate the statistical error but if we simply assume a 2% error taken from our previous measurements we see that the error intervals of the theoretical and measured efficiency are overlapping.

To summarize we can claim that our parallel model is useful if applied to *dediccated systems*.

4.4 Grid Environment Performance

As for the single machine performance model we need to make assumptions about the Grid environment. For now, we assume the following:

- The code we are dealing with is **load balanced**. We assume – similar to the single machine case – that a neglectably short time is spent with catching up with other processors. We thus either deal with identical supercomputers or with a perfectly balanced code.

- **Networks are ideal**. We neither share the network with other users nor it changes its characteristics in time (In later Chapters, we will consider this effect seperately, allowing networks to vary in time). If a supercomputer in a Grid computing setup is connected to more than one other supercomputer the network traffic does not interact. Also, the total bandwidth from one supercomputer to another is independent of the number of TCP connections.

- The **latency** time for sending a message over wide area networks with in the Grid computing environment is half of the IP round trip time (RTT). The RTT can be measured using the

UNIX `ping` command. By this we also neglect all overhead due to buffer allocation, buffer copy and similar.

- We allow **ghost zone sizes** for intra- and inter-machine communication to differ. In regular cases the ghost zone size for intra machine communication is the same as the stencil size of the numerical code.
- the **time a processor needs to catch up to others** is not taken into consideration.
- For the single machine performance we deploy our model from the previous section.

Having said this, we proceed the same way as in the single machine case namely by calculating the computation and communication time using known input parameters (like bandwidth between machines etc).

Let n be the total number of processors, g the ghost zones size for inter machine communication (divided by the stencil size), t_s the sequential time needed for one iteration on a given CPU, L the latency between machines (in seconds), s_n the number of synchronizations per iteration called by the application code and D the total amount of data which is send from one machine to another. Let further be B the bandwidth between machines.

If our setup includes more than one machine it is clear that the *slowest* wide area network link determines the efficiency of the distributed calculation.

Since we know that efficiency is according to 2.1

$$E = \frac{T_{comp}}{T_{tot}} = \frac{T_{comp}}{T_{comp} + T_{comm}} \tag{4.1}$$

We need to calculate T_{comp} and T_{comm}. According to our assumptions we get

$$T_{comp} = t_s \tag{4.2}$$

and

$$T_{comm} = \frac{s_n L}{g} + \frac{D}{B}. \tag{4.3}$$

Note that

- B and L are determined by the wide area network
- s_n is determined by the application code
- D is determined by the application code and the driver code
- and g is determined by the driver code.

For the overall efficiency we therefore get

$$E = \frac{t_s}{t_s + \frac{s_n L}{g} + \frac{D}{B}}. \tag{4.4}$$

The number of processors or machines does not occur in this formula but it is hidden in D, the total amount of data send over the WAN. Unfortunately we have to calculate it from case to case manually since it depends on the processor topology and the machine topology which again depends on the driver code.

4.5 Example Runs

We now take a look at the distributed run we performed in Section 3.2.12. We used a real production code for our runs which is used in current numerical sciences. It synchronizes 7 variables three times per iteration which sets $s_n = 21$. For a single CPU execution we measured 2.31 seconds per iteration (using a grid size which corresponds to the subgrid size per processor in a distributed run) which makes $t_s = 2.31$.

In order to calculate D we need to know how many grid functions each variable contains, the processor topology and the global grid size. It turns out that the code synchronizes 19 grid functions (grouped into 7 variables).

The global grid size was the three dimensional grid $120 \times 60 \times 60$, the processor topology $2 \times 2 \times 4$. The total amount of grid points sent with each grid function over the network is thus 120×60 and the total amount of data is

$$D = \underbrace{120 \times 60}_{\text{gridpoints}} \times \underbrace{8}_{\text{double precision}} \times \underbrace{19}_{\text{number of gridfunctions}} \times \underbrace{3}_{\text{per iteration}} = 3283200\text{bytes}. \tag{4.5}$$

We measured the TCP bandwidth B with a classical ftp transfer and got transfer rates around 320kb/s. The latency L as measured by the `ping` program is about 65ms. The communication time therefore is given by

$$T_{comm} = 21 \times 0.065\text{s} + \frac{3283200\text{bytes}}{320000 \frac{\text{bytes}}{\text{s}}} = 11.62\text{s}. \tag{4.6}$$

For the efficiency we finally get

$$E = \frac{T_{comp}}{T_{comp} + T_{comm}} = \frac{2.31}{2.31 + 11.62} = 0.165 \approx 17\%. \tag{4.7}$$

With this number we are quite close at the measured efficiency of 9% in Section 3.2.12. However, to see if this is just a coincidence we need to calculate the error intervals for both the measured and the predicted efficiency value.

Since we permanently monitor the performance of the running code we may calculate the relative error of the floprate out of the values from Figure 3.3. We get an error of about 3% for the mean value of the performance, i.e. (162 ± 5)Mflop/s. Assuming the single CPU performance of 112Mflop/s is error free we thus obtain a measured Efficiency of

$$E_M = (9 \pm 0.3)\%. \tag{4.8}$$

We now want to calculate the error of the theoretically predicted value. In order to calculate the relative error of the theoretical efficiency E_T we need to sum up the relative errors of the bandwidth and latency (assuming an error of 0 for the other quantities).

For a single stream we can estimate the statistical error for the bandwidth between the two end points to 320 ± 20 MB/s which corresponds to a relative error of about 6%. Adding the latency error of about 2% we end up with 8% so we can say about our theoretical efficiency

$$E_T = (16.5 \pm 1.3)\%. \tag{4.9}$$

So we get a theoretical efficiency of (16.5 ± 1.3) % as opposed to a measured one of $(9 \pm 0.3)\%$. The error intervals do not overlap which means that our model is *not* predicting the right value at all. Therefore, we – again – have to review our initial assumptions and check their validity.

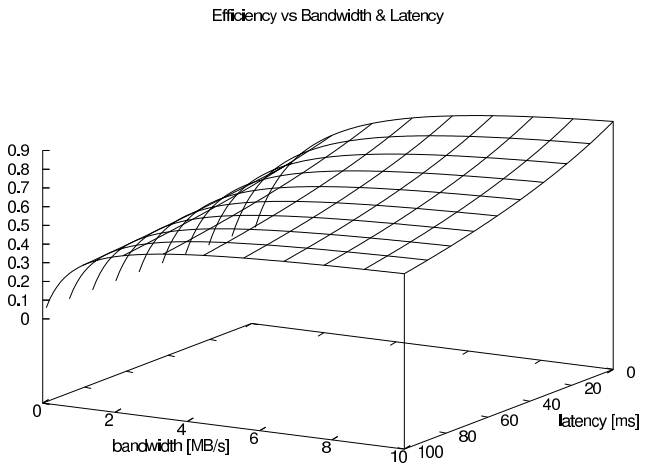

Figure 4.2: Theoretical Efficiency according to our model depending on the bandwidth and latency. A latency of less than 20 milliseconds combined with a bandwidth of more than 8MB/s already can result in an efficiency of more than 85%. Since the usage of multiple ghost zones decreases the latency in our model (i.e. a ghost size of 2 halfes the latency) one achieves significantly better performance improvements at high bandwidth rates when using multiple ghost zones than at low bandwidth rates.

Load balanced code. We used a regular domain decomposition across a homogeneous setup (both were equipped with the same processor type, MIPS R1000). The processors in the middle of the computational grid have one additional grid point (because of two ghost zones) but we still think this is neglectable.

Time for processors to catch up is not taken into consideration here but surely plays an important role. One cannot expect that all communications take exactly the same time. Since one processor cannot continue to perform calculations without its neighbors data, the slowest processor (or the slowest connection) holds up the entire calculation. We think that this fact has an important impact on the efficiency so that the predicted efficiency should be much lower!

Networks are ideal. Today's wide area networks are of course everything else than ideal. We measured the bandwidth between the two supercomputers using single streams. However today's physical networks do not necessarily share the bandwidth for multiple links between the same pair of IP addresses. The above calculation used (due to Mpich-G2) 8 simultaneous streams.

We measured the bandwidth of 8 simultaneous streams (due to firewall issues, we had to use ssh with no encryption) between origin.aei.mpg.de and harpo.wustl.edu. Surprisingly it turns out that each stream still achieved a throughput of about 290MB/s resulting in an overall transfer rate of 2.3MB/s. Putting this value into our model we get an theoretical efficiency of 45%.

Unfortunately this is not true for all links. Rerunning the same test across a completely different network (i.e. between uranus.haiti.cs.uni-potsdam.de and origin.aei.mpg.de) we rather see a more familiar behavior: A single stream has a throughput rate of 1,35MB/s and four simultaneous streams have about 360kb/s each. As a conclusion one can say that the behavior of multiple TCP streams strongly depends on the type of physical networks underneath, maybe the number of hops and other things that hardly be predicted.

The reason we observe a value of 9% which is about a factor 4 lower than 45% (taking the multiple stream behaviour into account) this must be – as in the single machine case – due to the fact that processors are not dedicated. However, these hold-up times are hard to predict.

4.6 Summary and Conclusion

In this chapter we have tried to predict the performance of a parallel code ran in a Grid environment by using simple input parameters like bandwidth, latencies, single processor speed etc. However, we saw that already in the single machine case the theoretically predicted value deviated strongly from the measured value. After setting up a better (or say more artificial) environment, e.g. a dedicated machine with specific operating system flags and environment variables, we obtained values which are close to the predicted one.

For the Grid environment performance we calculated the communication time using a simple model and saw that the predicted value is rather close to the measured one. An error calculation however showed that this match is just a coincidence. Reviewing our assumptions we found that multiple TCP streams can result in a full range of bandwidths' depending on the type of physical networks underneath.

Thus the prediction of the performance of a distributed code in a general Grid requires two things: The detailed understanding of all physical wide area networks involved and dedicated machines.

Modeling the performance is a useful and informative way to get closer to the core of the problem of low effeciency of codes ran in a grid environment.

Since it is clear, however, that it is hardly possible to fully understand all details of networks, operating systems and hardware, we have to conclude that modeling the performance of a parallel code in a Grid envrionment cannot predict the exact value. We thus need a more phenomenological treatment for future experiments.

Chapter 5

Large Scale Distributed Computing

After having described the general setup of distributed computations with Cactus and Globus, analyzed the performance behavior and other things we now describe an example run which we performed in April 2001. Since this run is – according to our knowledge – the biggest distributed run of a tightly coupled code ever performed in a real Grid environment. In future chapters we will take what we learned from this exeriment and on that basis we will analyze and design new methods and techniques for parallel computing. This is one reason why we will describe this experiment in detail. Furthermore these experiments were used to justify the funding of the DTF proposal.

5.1 DTF – Distributed Tera Flop Computing

Since the late nineties Grid computing was realized to be a forward-looking technology, many proposals have been written and funded to build a suitable Grid infrastructure backbone consisting of software and hardware components in many parts of the world.

One of the most prominent proposals to date was the so called DTF proposal which was submitted in may 2001 to the National Science Foundation (NSF). The proposal was written by major supercomputing sites in the United States that are members of the Partnership for Advanced Computational Infrastructure (PACI), which comprise the biggest supercomputing centers in the US like the San Diego Supercomputing Center (SDSC), the National Center for Supercomputing Applications (NCSA), California Institute of Technology (CalTech), Argonne National Labs (ANL) and others. It was funded for more than $50 million a few months later in 2001. This made it possible to build a national wide (and unique in the world) infrastructure to support execution of distributed applications on high-speed networks and systems which geographically cover major parts of the US. It consists mainly of exceptionally high speed wide area network connections and is now called the *TeraGrid* [66]. Figure 5.1 shows the planned network infrastructure of the TeraGrid which at the time of writing this thesis was still not ready for use.

As for every project of this kind, the public funding has to be justified. We were invited to perform experiments to show the feasibility of such wide area distributed computation and the results of our work are presented in this chapter.

Figure 5.1: **The TeraGrid.** This probably world wide unique Grid infrastructure – mainly consisting of networks and high speed clusters – is currently being build to support distributed execution of codes in the teraflop range. The results obtained in this chapter of the thesis were used to justify to support the funding of the TeraGrid.

5.2 General Setup

Since the distributed runs described in this chapter were done for the preparation of the TeraGrid one can see the location of the machines used here by taking a look at figure 5.1. The back bone of the TeraGrid network architecture will be upgraded to a 40 Gbit/s link (!) form the University of Illinois (UIUC) to the San Diego Supercomputing Center (SDSC) also connecting the Argonne National Labs and the California Institute of Technology, among others. Our test runs were done using machines at NCSA and SDSC.

5.2.1 Machines

We had four supercomputers at our disposal. We begin with the description of the smallest one, an Origin 2000 equipped with 128 R10000 CPUs running at 250MHz, followed by another one of exactly the same kind and yet another one of the same kind but equipped with twice as many processors. This set of machines is located at the NCSA. In addition to this a fourth machine, Blue Horizon, located in San Diego was deployed for our test runs. This IBM SP-2 computer was equipped with more than 1100 Power-3 processors running at 375 MHz and thus we had the chance to perform a distributed run across more than 1500 processors, see Figure 5.2.

5.2.2 Network

When we use the word *network* we consider every physical communication layer that we used in this setup to perform MPI communications. The range of latencies and bandwidths values for a

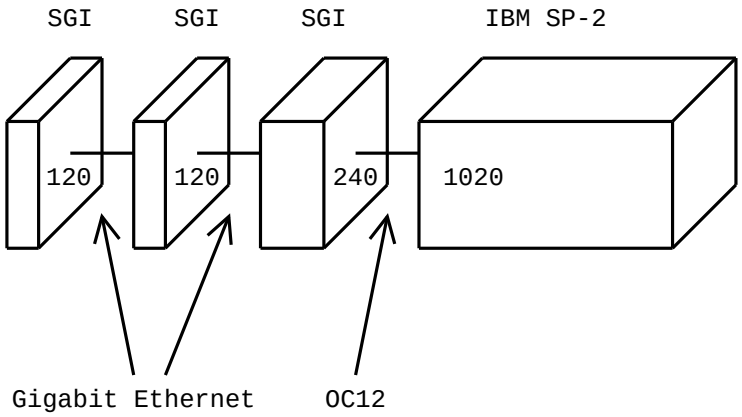

Figure 5.2: The machine topology in the DTF demonstration runs. We had four machines of different size and type connected by different types of networks namely OC12 and Gigabit Ethernet (GigE). The theoretical limit for the bandwidth of the physical network is 125MB/s for GigE and 622Mbit/s for OC12. While for GigE we achieved almost 100MB/s we only had 2.5MB/s real application bandwidth for the OC12 network.

single distributed MPI run across the whole setup includes up to four orders of magnitude!

Let us begin with the description of the slowest network namely the wide area network between NCSA and SDSC. Although this is physically an OC12 Network having a theoretical bandwidth of 622 Mb/s at the physical transport layer, we performed some bandwidth tests using the mpptest program [36]. For a single TCP connection we only achieved a bisectional bandwidth of about 2.5 MB/s. We did not measure the TCP performance for parallel streams. The latency between the two supercomputing sites was determined to be 36ms.

The next faster level of networks we are using in this test run is the local area network between machines at NCSA. The local physical network between the origins consists of an Fast Ethernet (FE) and Gigabit Ethernet (GE) network. Since the TCP implementation under IRIX and the network card drivers are fairly fast, we could achieve an MPI performance of almost 100MB/s between machines at NCSA using GE. Even faster than this is the network between two nodes on Blue Horizon which – after a switch upgrade in January 2001 [13] increased to more than 400MB/s with a latency of 17 μs, both measured in MPI communication. The latency for intra machine communication on the origins is even below 10 μs and the bandwidth is around 120 MB/s [51]. See the summary in the following table.

Network type	bandwidth	latency
Intra machine Blue Horizon	420MB/s	17 μs
Intra machine origin 2000	120MB/s	10 μs
GigE between origins	98MB/s	??
OC12 between NCSA and SDSC	2,5MB/s	72ms

5.3 Code Characteristics

The application code we used to run across this huge setup was a numerical Fortran code computing the time evolution of gravitational fields in the general theory of Relativity. The evolution code

calculated the equations in the socalled *ADM* formulation (see [49]) using the "staggered leapfrog" method [55]. This code has been used as the standard code for problems in numerical relativity throughout decades. In principle we could take any other numerical code running in the Cactus framework. However, we would like to mention that this code has been optimized on the application level. This means in particular that the synchronizations (and thus the communication) has been reduced to a bare minimum. Note that it is solely determined by the application programmer how many synchronizations have to be done. This code synchronizes six grid functions of double precision floating point data and as such sends $8 \times 6 = 48$ bytes per ghost point to the neighboring processor. It also has been optimized for single processor performance by rewriting the Fortran code to make better use of the memory cache.

5.4 Remarks about General Problems

In this big setup we are already confronted with most major problems in Grid computing:

- multiple authentication
- co-allocation
- co-scheduling
- variety of networks in different quality
- different operating systems
- processors running at different speed

The Globus and Mpich-G2 packages offer solutions to some of the above problems like e.g. the first two items whereas co-scheduling is a still open problem. Currently, in our setup, co-scheduling was done by picking up the phone and talking to the side administrators of all sites involved.

As we will see in the next chapter the performance of a distributed run across this setup yields a very low performance. It is a major goal of this chapter to learn from this experiment where the communication bottlenecks are located.

5.5 Observed Performance

As outlined in Section 2.1 we will use the terms speedup and efficiency to measure performance using equation 2.2. We have two different types of processors to deal with and in order to calculate the speedup and efficiency we need the single floprate of the two types as well as the total floprate.

In order to get the single floprates F_k we scaled down the problem size to fit on one CPU. Then we used hardware counters on both systems, i.e. `hpmcount` on Blue Horizon and `perfex` on the SGI systems. We measured 168 on the R10000 processors and 306 on the Power-3, respectively[1].

[1] These floprates include the so called "multiply-adds": It was found that an operation which contained a multiplication followed by an addition was (by some hardware counters) counted as one floating point operation (but is clearly two operations). The `perfex` tool on IRIX systems does not include those multiply-adds while the `hpmcount` tool on AIX systems does. It even counts the number of those operations so that we could extrapolate the real floprate for the SGI systems as well.

Measuring the total floprate of a distributed running code is harder. We did this by using timers printing out the duration for the last 10 iterations to a local file measured in wall clock time, using the `MPI_Wtime()` call. Since we know the characteristics of our code we know that the number of floating point operations increases linearly with the problem size. We therefore can calculate the number of flops *per grid point* and therefore the total number of executed floating point operations for an arbitrary large computational grid. We calculated that the code executes 780 flops per grid point per iteration. The total floprate of the running code could then be measured as *gridsize* $\times 780 \times$ 10 iterations divided by the time needed for 10 iterations.

For a grid consisting of $nx \times ny \times nz$ =842.940.000 points we therefore have $842.940.000 \times 780 \times 10 = 6.574$ Tflops to be executed. As said above, each processor took the time for every 10 iterations. The current floprate for those 10 iterations was then calculated as 6.574 Tflops divided by the *maximum* of each processors measured time which was 155 seconds in the best case. The floprate was thus determined to

$$F = \frac{6.574 \cdot 10^{12}\text{flops}}{155\text{s}} = 42\text{Gflop/s}.$$

The "expected" floprate can be calculated as the sum of the single processor floprates on all machines namely $480 * 168 + 1020 * 306$ Mflop/s = 393 Gflop/s and hence we obtain an efficiency of

$$E = \frac{42}{393} = 11\%.$$

Note that this Floprate can be achieved on the 256 CPU SGI alone!

5.6 Manual Adjustments

Since the above quoted efficiency is less than satisfactory we will now try to improve the efficiency by *manually adjusting* parameters on the driver level of our code (*without* modifying the application code).

5.6.1 Processor Topology

As we have seen in Section 2.5.1 classical parallelizing codes like PUGH try to distribute processors equally among the dimensions. For 1500 processors the topology would yield $10 \times 10 \times 15$ for three dimensions. This can or cannot be an adequate processor topology since it depends on the grid size and shape where wide area network communications take place. As we will see later (in Section 6.2.2) we need to "line up" all machines in the dimension where the computational grid has the most number of points. Since, again, PUGH *always* (unintentionally) lines up the machine in the highest dimension we *chose* the computational grid to have its longest dimension to be the z-dimension. In this case we were lucky: The z-dimension already had the most number of grid points. However, the shape and size of the computational grid is solely defined by the scientists needs.

Thus we chose a computational grid stretched in the z-direction. We also found (without any theoretical calculations that would justify this choice) that a processor topology of $5 \times 12 \times 25$ seems to be adequate for this computational grid size and machine topology. This is possible in PUGH by specifying the desired processor topology in the parameter file which is read at startup time.

For this topology we now can calculate the amount of data to be sent over the wide area network. Since our global grid size was chosen to be 350×720 in the $x-y$-plane we had a local grid size of 70×70 in the $x-y$-plane on each processor. Since we know that every grid point is sending 6×8 bytes per iteration we have to send $5 \times 12 \times 70 \times 70 \times 6 \times 8 \approx 14$MB across the WAN. Assuming the maximal possible bandwidth of an OC12 as being 77MB/s and 1.7 seconds for one iteration we theoretically cannot expect the code's efficiency to be better than 90% since it takes 0.18 seconds for communication.

5.6.2 Load Balancing

From test runs on single processors we saw that the IBM Power-3 processor is about twice as fast as the R1000 processor, so we had to rebalance our code in a way that the local grid size in x and y direction was the same on each processor but differed in the z-direction. We chose to give 135 grid points in the z-direction to a 'Blue Horizon' processor whereas 75 grid points to a Origin processor. As in the processor topology case this is possible by specifying it in the parameter file at startup[2].

In addition to that we also chose to give all processors that are involved in wide area network communication *less* work than their "colleagues". By this we achieve a partial overlap of computation and communication since those processors (they are $2 \times 5 \times 12 = 120$ to be exact) can send their data before others have finished their work and thus are performing communication while others keep computing. We gave 60 grid points (instead of 75) to the SGI processors at the WAN boundary and 100 (instead of 135) to the Power-3 processors, i.e. roughly about 20% in both cases. Again, this choice is completely empirical, without theoretical justification. In the following chapters we will design algorithms that automatically choose the "right" numbers.

5.6.3 Ghost Zones

Recalling the way the synchronization between processors is done (see Section 2.3) one can apply the following trick: For a code which has a stencil size of 1 we increase the ghost zone size to e.g. 10. By this choice we need to send 10 times more data, but in our current setup, we need to do it only every 10^{th} iteration. In other words, instead of sending 10 small messages we *coalesce* them into a single big one and send it every 10 iterations (and we do not change the total amount of data). The advantage of this procedure is the following. In our setup we have to deal with a latency of about $36ms$ which adds up to 0.36 seconds in 10 iteration. Sending one big message however we only have to deal with a 10 times smaller latency, i.e. 0.036 seconds. Apart from network latency one also saves time used for buffer copying, memory allocation (leading to cache misses) etc.

By doing this we are trading the latency of sending a message against CPU power: The additional ghost zones have to be calculated by the local CPU.

We now have only 120 processors that are involved in wide area network communication and thus we should give them more ghost zones. Unfortunately, the current implementation of our code only allows a consistent ghost zone size in each dimension. There is, however, no numerical or other reason which forbids different ghost sizes on different processors even in the same dimension or even on the same processor. Thus we could only increase the ghost zone size for the entire dimension. Since we knew that all TCP traffic will occur in the z-direction of our computational grid we chose to increase the number of ghost zones in the highest dimension to 10. However, this also means that we have 10 ghost zones at places where it is not needed (e.g. in intra-machine communications

[2]The implementation of this parameter file based loadbalancing was done by Matei Ripeanu

where the latency is about 10 μs). This can obviously be improved with more and better work, which we will discuss in the next chapters. Again, this number is not justified by any theoretical study, it is entirely empirical.

5.6.4 Compression

Another trick we applied before sending data over the wide are network was to *compress* the data. We used the `compress()` routine which is shipped with zlib version 1.1.3 [18] to compress the send buffer and `decompress()` to decompress it on the receiver side. The compression factor and the time needed to compress strongly depends on the type of data we are compressing, i.e. the type and shape of the floating point functions. We expect noisy and irregular functions to compress much worse than smooth, regular functions. Our code calculated the evolution of gravitational waves, which have a smooth and also periodic nature and therefore the compression rate was very high (better than 1/10 of the original size).

Like in the ghost zones case our code was not sophisticated enough to apply compression only at places where needed, so we applied it to any communication going on in the z-direction. This involves compression/decompression of data which is sent over very fast intra-machine networks (e.g. 420MB/s within blue horizon) and as such cannot considered to be optimal. We will resolv this issue in coming chapters.

5.7 Observed Performance II

After manually applying all techniques described in the last sections we had the opportunity of running the code a few more times. It happens that our improvements drastically improved the efficiency of our distributed run. We can quote efficiency numbers of 88% on 1140 (involving two machines, blue horizon (1020) and aegir (120)) CPUs and 63% on 1500 CPUs.

Of course it would be a very interesting scientific question which single technique exactly led to which performance improvement. Unfortunately, due to the lack of time and we were not able to perform single large scaled runs with every change in the driver layer separately.

5.8 Discussion of Results

The set of DTF runs has shown us two basic things. We now know that (a) large scale distributed computing is actually *possible*. Remember that all previous attempts (known to us) to run tightly coupled MPI codes across several supercomputers involved a far smaller setup. Furthermore we learned that (b) manual tuning, not necessarily on the application code level, can improve the performance of such a code drastically.

Up to now we did not exactly specify what we mean by “manual tuning”. By this we mean any change applied to the code using the knowledge of the setup *before* starting the code, mostly achieved by setting parameters in the parameterfile.

It is clear that all those improvements and adjustments are **not** the scientist’s responsibility. It rather would be ideal if all techniques which have been applied in the DTF run would be made by the driver code *automatically*, without user interaction.

Setting the adequate processor topology, ghost zone, load balance in the parameter file is a very error-prone process. For an n-machine run we need to have n consistent parameter files. Even

within one parameter file it is very cumbersome (as one can imagine) to specify the exact load for each of the 1500 processors.

This however leads us to the question about the exact values we specified for various adjustments (e.g. 10 for the number of ghost zones). Recall that we manually set those values on a gut level but this is not possible from within a running code. How do we solve that problem? One possibility would be to *calculate* the optimal ghost zone size using measured input parameters like bandwidth and latency. However, since the performance models we have at hand are too imprecise we rather choose the following method: The running driver code should simply *test* possible values while monitoring the performance and stick to the value which gives the maximum performance. This procedure has even an additional advantage: Since the network quality is not independent in time but rather can change during code execution we do need such *dynamic* techniques to adapt a running code efficiently to the runtime environment.

From the things said above we should now be able to formulate a wish list for a new smart algorithm for parallel codes running in a Grid environment that therefore should contain the following considerations:

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

processor topology: Minimize communication over the wide area network 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 and where needed. This deals with latencies in wide area networks (in our case implemented using multiple ghost zones, as discussed before).

apply selective compression: The code should optimally compress data which will be sent over slow communication channels. 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, competing traffic may change the network bandwidth), 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 algorithms should generalize the previous algorithms 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/load balancing 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 e.g. migration [43] 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.

communication overlap: If one processor waits for data from a specific direction it could still keep doing sensible work like initiate other communications, copy buffers, do calculations or similar.

In the next chapter we will analyze and investigate the possibilities of implementing algorithms which fulfill the requirements of our list.

Chapter 6

Adaptive Strategies for Performance Improvement

The last chapter has shown what kind of problems one faces when running tightly coupled parallel codes in a real Grid environment. The performance of such codes gets very poor unless one specifically tries to adapt the code to the environment. Three major reasons can be blamed for bad performance:

- load imbalance
- high latency
- low bandwidth
- non-dedicated environment

While we can't deal with the last issue, in this chapter we will try to analyze the situation and suggest algorithms and concepts to improve the performance by trying to reduce the effects of the first three items listed above. The first item can be treated by redistributing all data according to processor speeds to find an optimal (but static) load balance. High latency often leads to problems when message sizes are small and the number of messages is large, so we should always aim at achieving the opposite namely having few but bigger messages in the case of high latency. If the bandwidth is low we have to try to send as little data as possible. As mentioned before the amount of data which has to be exchanged between machines is not uniquely determined by the application. As already seen in the last chapter, one method to reduce the amount of data (send in a specific direction) is to choose an adequate processor topology.

Our code therefore has to *adapt* itself to the environment, i.e. to the machines, processor power and networks. As opposed to the next chapter, we will discuss how the code *initially* adapts to the environment. Dynamic strategies, where the code keeps adapting throughout the evolution will be discussed in the next chapter.

But before we begin to discuss processor topologies we like to say a few words about testsuites.

6.1 Testsuites

Our goal is to analyze and improve all algorithms on the *driver level* of a parallel code, not on the application level. This does not only mean that the application remains unchanged but it also

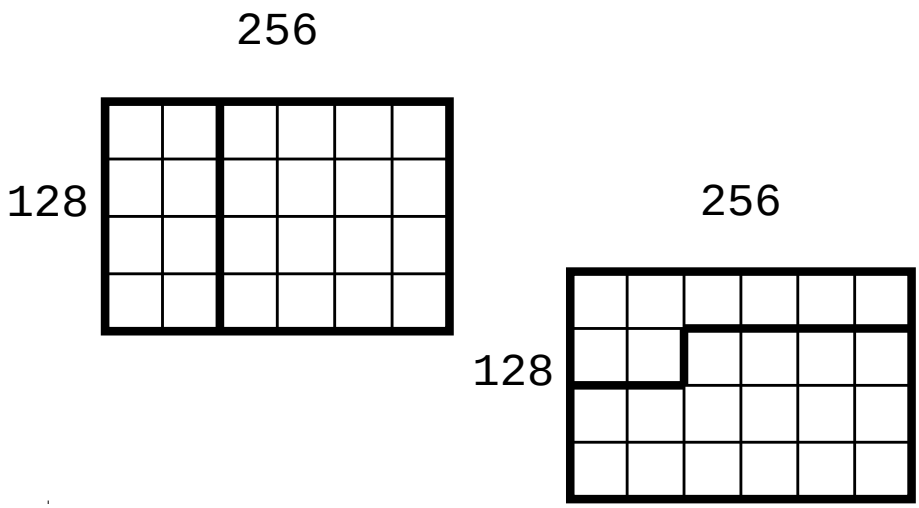

Figure 6.1: A simple example of a two dimensional computational grid distributed on two machines having 8+16 processors. The thick line marks the border between machines. The longer that line, the more data has to be sent over a possibly slow network.

means that the results of the calculations of the application code need to remain *exactly* the same as before. All our changes to the driver code *must not change the numerical results*. They should be the same no matter if the code is run on a single CPU on a laptop, cluster, supercomputer or on multiple supercomputers.

For this purpose many parallel applications come along with a so called testsuite to check the correctness of the results. This testsuite compares stored data of previous runs (which it assumes to be correct) with data just produced in a (e.g. multi-machine) run.

Unfortunately when dealing with floating point numbers, the results cannot be *exactly* the same on different machines. The reason for this is that different architectures may have different formats to represent floating point data in memory. Since every floating point operation produces roundoff errors the result of a floating point operation can differ on different architectures although the input data was identical.

An appropriate testsuite therefore compares the results omitting the last two digits in a floating point number. We then say that the numbers are the same *up to machine precision*.

Although all issues discussed in this and the following chapters are treated in a implementation independent manner we anticipate a realisation within the Cactus framework and as such every time we replace driver routines we run a set of testsuites (coming along with different application thorns) to make sure that our changes/improvements did not affect the numerical results.

6.2 Processor Topology

Figure 6.1 gives a simple idea about the situation we are facing. It is our goal to reduce the amount of data which is sent over the wide area network. Looking at Figure 6.1 we see that this turns into finding the processor topology where the "thick line" in Figure 6.1 gets a minimal length. This "line" is a plane-like object in the three dimensional case, a solid body in the four dimensional case etc.

Thus we have to find an algorithm which tries to minimize the communication by minimizing the size of the "faces" between processors (i.e. lines in the 2D case, planes in the 3D case, cuboids in the 4D case etc.). Before we do this we have to take one step back and take a look at single machine topologies to have a basis for multi-machine topologies.

6.2.1 Single Machine Topology

Problem. A three dimensional computational grid of size $256 \times 512 \times 128$ has to be divided up on 128 processors in a way that the communication is minimized, i.e. the sum of the areas of the planes between processors become minimal.

The solution to this problem is quite straightforward and can be solved by the following (known) algorithm:

```
Algorithm: Create3DTopology(nprocs, dim, gridsize[])
{
  /* nprocs: total number of processors
  dim: dimension of the grid
  gridsize: array containing the number of points in each dim */
  for (i = 1 to gridsize_x)
    for (j = 1 to gridsize_y)
      for (k = 1 to gridsize_z)
        if (i*j*k == nprocs)
          compute sum of areas between procs
          if area < minarea
            minarea = area ; save i,j,k
}
```

This procedure represents a so called "brute-force" method: It simply loops through *all* decompositions, checks if the product equals the given processor number and calculates the areas of the processor faces, always recording the topology associated with the smallest area.

Example

For a three dimensional computational grid of size $128 \times 32 \times 64$ decomposed on 64 CPUs the above algorithm yields a processor topology of $8 \times 2 \times 4$. Recall that the old algorithm from Section 2.5.1 would calculate a topology of $4 \times 4 \times 4$. One clearly sees that our new algorithm puts more processors to longer dimensions which was our goal.

This algorithm works fine, but since we cannot restrict ourselves to three dimensional computational grids we have to extend the above algorithm to arbitrary dimensions. Using this brute force method it means that we have to loop through *all* possible decompositions in N dimensions.

Therefore we propose to replace the three `for` loops with a general loop through a N-dimensional cuboid. We implemented this, and as a result we get a code which does in principle what we want.

Unfortunately, this code gets very slow with increasing processor numbers and dimensions. For processor numbers > 1000 and dimensions > 4 the calculation can take hours, which makes the code unusable.

Thus we need to profile our code to see what the most time consuming part is. For a brute-force algorithm it is not surprising that the code spends most of its time to loop through topologies that are no real decompositions (its product does not yield the total number of processors) rather than

with calculating the areas once a valid topology is found. For example, if the total number of processors is 8 and the dimension is 4, our code loops through all decompositions:

```
1 1 1 1
2 1 1 1
3 1 1 1
4 1 1 1
5 1 1 1
6 1 1 1
7 1 1 1
8 1 1 1
      ⋮
1 2 1 1
2 2 1 1
3 2 1 1
4 2 1 1
5 2 1 1
      ⋮
8 8 8 6
8 8 8 7
8 8 8 8
```

which is too many. We rather have to think of a loop that automatically stops increasing the processor number of the current dimension if the current product of CPUs in all dimensions is bigger than the total number of processors, meaning that the loop should look at least like

```
1 1 1 1
2 1 1 1
3 1 1 1
4 1 1 1
5 1 1 1
      ⋮
1 2 1 1
2 2 1 1
3 2 1 1
4 2 1 1
1 3 1 1
2 3 1 1
```

$$\vdots$$

$$\begin{array}{cccc} 1 & 1 & 1 & 4 \\ 2 & 1 & 1 & 4 \\ 1 & 2 & 1 & 4 \\ 1 & 1 & 2 & 4 \\ 1 & 1 & 1 & 5 \\ 1 & 1 & 1 & 6 \\ 1 & 1 & 1 & 7 \\ 1 & 1 & 1 & 8. \end{array}$$

As can be seen this loop skips "really stupid" decompositions like $8 \times 8 \times 2 \times 6$ for 8 processors in total as opposed to our previous version. This significantly decreases the combinations to check for. Therefore we have to find a way to generate such a loop. To achieve this, we developed the following algorithm:

Algorithm: *CreateNDTopology(nprocs, dim, gridsize)*

```
{
int topol[dim];    /* the decomposition */
  while (loop)
    {
      product = ∏_{k=1}^{dim} topol[k]
      if (product == nprocs)
        {
          calculate area
          if (area < minarea)
            minarea = area; save topol[]
        }

      for(k=0;k<=dim;k++)
        {
          topol[k]++
          product = ∏_{k=1}^{dim} topol[k]
          if (product > nprocs)
            topol[k]=1 ; continue
          else
            break
        }
      product = ∏_{k=1}^{dim} topol[k]
      if (product == 1)
        exit loop /* we arrived at 1 × 1 × ... × 1 × 1 again! */
    }
}
```

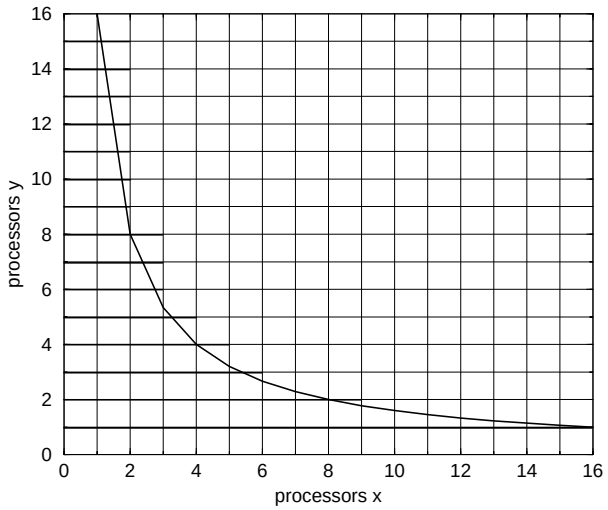

Figure 6.2: This picture is to illustrate how our algorithm loops through possible decompositions in two dimensions for 16 processors. The allowed decompositions are the ones which hit the curve $y = 16/x$. The black vertical lines represent the way of our loop which always crosses the curve and therefore no allowed decomposition is omitted. The figure also shows how the loop is reduced as compared to the initial brute force method.

The `topol[]` array represents the possible decompositions. The algorithm starts with an infinite loop. First we calculate the product of all indices to check if this is a decomposition. If yes, we calculate the $n-1$ dimensional "volume" of the processor faces. If this is smaller than the current minimum the index-array will be saved. Now we increase the first index of our `topol[]` array by 1 and again calculate the product. If the product is bigger than the total number of processors we reset this index to 1 and continue with the next index. Otherwise the loop continues. If the product becomes 1 again it means that we already looped through all possible combinations and ended at $1 \times 1 \times 1 ... 1 \times 1$ again.

Now it remains to investigate whether our algorithm really gives us the optimal result. By *optimal* we mean the decomposition where the sum of areas of the processor boundaries will be minimal. In order to do this we recapitulate again the idea of creating the loop over all dimensions. Figure 6.2 illustrates in a two dimensional example all possible and not possible decompositions for 16 processors. All allowed decompositions are the ones that are located on the curve $y = 128/x$. Instead of looping through all possible pairs of numbers in two dimensions our algorithm loops just as far as we cross the curve in Figure 6.2. For this example, it shows that our algorithm does not omit any allowed decompositions.

Theorem
Our Algorithm from page 60 gives the optimal processor topology.

Proof
We proof it by induction. Our graphical example (see again Figure 6.2) showes that the first claim is true for $D = 2$ (or alternatively we may even consider that our assumption holds for $D = 1$).

Suppose that the algorithm finds the optimal processor topology for an arbitrary dimension D. This means that it loops through all possible decompositions, starting from $< N_1, 1_2, \ldots, 1_D >$ to $< 1_1, 1_2, \ldots, N_D >$ and finds the optimal one, (e.g. $< n_1, n_2, \ldots, n_D >$). Suppose now that we deal with $D+1$ dimensions. The above decomposition, as being the optimal one in D dimensions could be also interpreted as the optimal one in $D+1$ dimensions where the last dimension is fixed to $n_{D+1} = 1$. Now, note that the last dimension gets increased when the decomposition looks like $< 1_1, 1_2, \ldots, N_D, 1_{D+1} >$. Now the algorithm finds again the optimal decompostion in $D+1$ dimensions, this time under the condition that the last dimension is kept fixed to $n_{D+1} = 2$. This is being repeated N times. The restriction that the last dimension is always fixed to some value cancels now, since the loop has already gone through all values. It follows that if the algorithm finds the optimal processor topology in D dimensions, it also finds it in $D+1$ dimensions. Since we know that it is true for $D = 2$ we proofed it for an arbitrary dimension by induction.

■

This algorithm is fast enough for our purposes. For example, to calculate the optimal processor topology for a six dimensional grid on 2500 processors it takes 0.8 seconds on a Pentium-III processor.

Another interesting question is the scaling behavior of our algorithm. If N is the total number of processors and D the dimension then the pure brute force method clearly scales like $O(N^D)$.

For the scaling behavior of our new algorithm we have to take into consideration that the number of elements the loop goes through is exactly the number of elements under the curve in Figure 6.2. For the two dimensional case it is quite easy to calculate. If N is the total number of processors then the curve is parametrized by

$$f(x) = \frac{N}{x}. \tag{6.1}$$

The area A under the curve is then given by the integral

$$A = \int_1^N f(x) = ln(x)|_1^N = ln(N). \tag{6.2}$$

For the same problem in D dimensions we have to calculate the $D-1$ dimensional integral of the function implicitly given by

$$N = \prod_{k=1}^{D} x_k,$$

where x_k is the number of processors in the k^{th} dimension. Solving for x_1 we get

$$f(x_1) = \frac{N}{\prod_{k=2}^{D} x_k} \tag{6.3}$$

and thus we have to calculate the integral

$$\int_{<1,\ldots,1>}^{<N,\ldots,N>} \frac{N}{\prod_{k=2}^{D-1} x_k} d^D x = (ln(N))^{D-1}, \tag{6.4}$$

using Fubini's theorem. Our algorithm thus scales like $O(ln(N)^{D-1})$.

■

Summary The goal of this subsection was to develop a processor topology algorithm to reduce the communication in single machine execution. We started by taking a known 3D-algorithm and extended it to n-dimensions. After learning that the n-dimensional version performes very slow (due to the brute-force nature of the original 3D algorithm) we analyzed bottlenecks and developed a better approach. Our final algorithm scales like $O(ln(N)^{D-1})$, as opposed to $O(N^D)$ for the original 3D algorithm (where D is the dimension and N the total number of processors).

6.2.2 Multi machine topology

Since we now have an algorithm that always minimizes the communication between processors in a single machine setup we have to take a look at our original problem again. Looking at figure 6.1 we observe the following: Even if we have minimized the area of the processor "faces" it does not prevent us from ending up with the situation illustrated in the right picture of figure 6.1. This is not surprising, since we did not take the machine layout into account, namely which processor is located on which machine, how many processors does a machine have, how many machines do we have in total etc.

Before solving the problem for the multi-machine case we first need to obtain this information. On the MPI level we only have the possibility of calling `MPI_Comm_size` which only reports the total number of processors (on all machines). We thus have to exploit higher (or lower?) level information services to get this information. However, since this problem is clearly depending on the particular implementation (or the particular service) we will discuss this in Chapter 8.

Assuming we have this information, how do we create a topology which fits to the actual machine layout? We know that we *first* have to minimize the wide area network communication and *then* the intra machine communication. Taking again a look at Figure 6.1 we realize that the smallest faces – slices through an $n-$dimensional cuboid with the smallest n-1-dimensional area – are always the ones which are orthogonal to the *longest* dimension of the computational grid. By longest dimension we mean the dimension with the most number of grid points.

The proof of this fact is quite easy since the area of such a slice is the product of all grid points in all dimensions but the one it is orthogonal to. Since we chose the longest dimension to be the one orthogonal to the slice any other (orthogonal) slice would have a bigger area. Thus we need to "line up" the machines in the longest dimension.

The "inner" topology of a subjob should be determined using our single machine algorithm we have developed. We don't want any "edges" between processors belonging to different machines like in Figure 6.3. Since we rather want the machine boundaries to be "flat" (to minimize its area) it follows that the number of processors in all but the longest dimension (the one the machines are "lined up" in) should be the same for all subjobs (machines) to maintain consistency. Furthermore it follows that the *smallest* subjob, i.e. the machine with the fewest processors determines the overall topology. In order to calculate the global processor topology we therefore first have to calculate the inner topology of the smallest subjob. For constructing an optimal processor topology in the multi-machine case we propose the following Grid aware processor topology algorithm ("GAPTA"), which consists of the following steps:

(1) Obtain the job-layout, i.e. how many processors are distributed on how many machines.

(2) Find out which dimension contains the largest number of grid points.

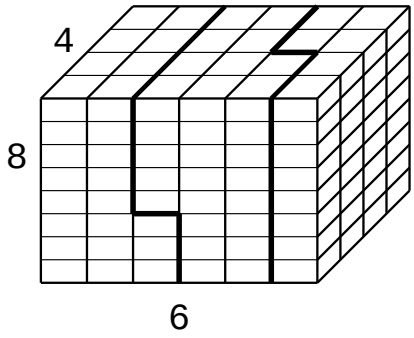

Figure 6.3: This picture illustrates the situation when classical parallelizing algorithms are used for a multi machine code execution. This situation will also occur if the number of processors on each subjob are not multiples of the smallest subjob. The thick line marks again the wide area network boundary. The code could basically run without problems in such a configuration but the amount of communication increases as compared to a "regular" processor number and we will not be able to apply adapted ghost zone sizes for intra- and inter-machine communication (see next chapter).

(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 its subgrid.

(4) Find the subjob with the smallest number of processors.

(5) Use the algorithm from Section 6.2.1 to find the optimal processor topology for this *smallest* subjob in its 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 entire job.

The implementation of this algorithm is straightforward. As input we need 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 namely the processor topology.
Example. For a three dimensional grid of the shape $128 \times 256 \times 64$ we calculate the processor topology when running on 4 machines using 64,128,128 and 256 processors using

a) the classical algorithm from the last section and Section 2.5.1

b) the full Grid aware topology algorithm we just developed.

For all three algorithms we will additionally calculate the amount of data which will be sent across the wide area network every time a function will be synchronized.

The algorithm from Section 2.5.1 does not take the machine topology and not even the shape of the grid into account. The only input parameters are the total number of processors and the

dimension (2D, 3D, 4D... etc.) of the problem. It gives us a decomposition of $8 \times 8 \times 9$ trying to distribute the numbers of processors equally across the dimensions. If we calculate the amount of data sent across the wide area network every iteration we assume that every grid point sends 10×8 bytes (10 functions consisting of floating point numbers) to its neighbor. Since this algorithm (always) lines up the machines in the highest dimension and one slice through the third dimension contains 128×256 grid points the amount of data sent across the WAN can be estimated to $128 \times 256 \times 8 \times 10 \times 3 \approx 8$MB ($\times$ 3 for 3 wide area connections between 4 machines). In fact it is even more since this decomposition does not respect machine boundaries which means that the boundaries between machines are not planes but may contain edges (see Figure 6.3).

Our single machine algorithm from the last section won't do any better. It yields a topology of $9 \times 16 \times 4$ taking the shape of the global grid into account, but not the machine topology.

We now apply GAPTA according to the outline previously discussed:

(1) We have a job-layout consisting of 4 machines ordered with 64, 128, 128 and 256 processors respectively.

(2) The longest dimension (the dimension with the most number of grid points) is the second (containing 256 grid points).

(3) Those 256 grid points have to be divided up across the machines according to the number of processors in each machine. Thus the grid point distribution across the four machines in this second (longest) dimension is taken to be 28:57:57:114 (according to 64:128:128:256 processors).

(4) The first subjob is the smallest with 64 processors.

(5) The subdomain for this subjob is $128 \times 28 \times 64$ (using the subgrid size from **(3)**). Applying the single machine algorithm of Section 6.2.1 the processor topology on this subjob is determined to be $8 \times 2 \times 4$.

(6) Since we said that the number of processors in all but the longest dimension have to be the same on all machines, the global topology then must be $8 \times X \times 4$, 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 product of the number of processors in all other directions namely $X = 576/(8 \times 4) = 18$.

(8) The global topology is now determined to $8 \times 18 \times 4$.

Thus our algorithm gives us a topology of $8 \times 18 \times 4$ and chooses to line up the machines in the second dimension. A slice through the second dimension contains 64×128 grid points and since there are no "edges" the amount of data which has to be send across the WAN is $64 \times 128 \times 8 \times 10 \times 3 \approx 2$ MB as compared to 8MB when using the other algorithms.

6.2.3 Processor Ordering

As opposed to considerations for classical parallel computing (like in Section 2.5.1) the mapping of processor ranks into the processor topology is of particular importance. Our goal was to "line up" the machines in the longest dimension. But what does it mean in practice? It means that every processor is aware or "knows" which processor is located to his left, right, top, bottom etc.

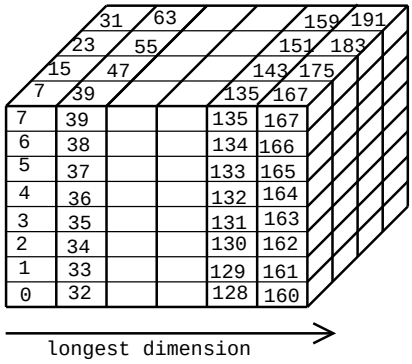

Figure 6.4: **Numbering of MPI ranks in a distributed job** when using Mpich-G2. Since we want to line up the machines in the longest dimension and we know in which manner Mpich-G2 constructs the global ranks we have to distribute the ranks in a way that counts the longest dimension last.

Since within MPI processors are uniquely identified by their MPI rank it means that we need to specifically map the MPI ranks into their place in the n-dimensional topology. This information then needs to be passed to other driver routines which are setting up coordinate grid functions and other things depending on the particular topology. We stress again that within a modular framework the numerical Fortran code does *not* need to be aware of processor topologies, ordering or similar.

Before we try to do this we need to find out how the *local* MPI_Ranks on each machine are mapped to the *global* MPI_Ranks in the Grid computing environment. Unfortunately this is implementation dependent. However, since we decided to use Mpich-G2 we will describe (and later implement) how Mpich-G2 assigns MPI ranks within a global job. Recall that Mpich-G2 uses (or is able to use) the local VMPI[1] and as such all processors within a subjob have their own 'local' MPI_COMM_WORLDS and therefore 'local' MPI ranks. It is Mpich-G2's job to hide these local ranks and "worlds" and create a global MPI_COMM_WORLD. For this purpose it uses the `$GLOBUS_DUROC_SUBJOB_INDEX` environment variable to perform this mapping. The processor with the global rank 0 is the one with the local rank 0 and the subjob index 0. If the subjob with the index 0 has N processors the global $N + 1$ rank is the processor with the local rank 0 in the job with the index 1 and so forth.

This again leads us to the question of how to order the machines themselves. For example, if a distributed setup includes three machines A, B and C with fast wide area networks between (A and B) and (B and C) but a slow network between (A and C) it makes sense to have a machine order of the type A-B-C and not C-A-B. With the information we just learned about Mpich-G2 it turns out to that the user needs to assign the `$GLOBUS_DUROC_SUBJOB_INDEX` to every machine. In our case it would be A:0, B:1 and C:2. However, as said above this maybe completely different in different implementations.

Our original goal was to line up the machines along the dimension with the most number of grid points. With the above knowledge at hand we are now able to perform the processor mapping. Let $< N_1, N_2, \ldots, N_n >$ be the given processor topology for N processors in n dimensions. The

[1] Vendor's MPI, or native MPI

naivest way of processor ordering (used in PUGH) would be the following: Rank 0 would be the first processor in all dimensions. Rank 1 would be the second processor in the first dimension (and still the first in the other dimensions), rank N_1 would be the first processor in the first and third dimension and the second processor in the second dimension and so forth, ending with the $(N-1)^{\text{th}}$ processor to be last processor in all dimensions. Combined with our knowledge about how Mpich-G2 distributes the global processor ranks we find that this way of mapping the ranks into the topology always lines the machines up in the *highest* dimension. This is not what we want since this would require the computational grid to always have the most number of grid points in the highest dimension.

In order to achieve our goal we rather use the following algorithm:

```
Algorithm: Compose(dim,topol[],longest_dim,rank)
{
int idim, position[];
for(idim = 0; idim < dim; idim++)
  {
    if(idim == longest_dim and longest_dim != dim-1)
      idim++;    /* omit the longest dimenstion for now */
    position[idim] = rank modulo topol[idim];
    rank /= topol[idim];
  }
if(longest_dim != dim-1)
  {
    position[longest_dim] = rank modulo topol[longest_dim];
    rank /= topol[longest_dim];
  }
return position[];
}
```

As input parameters this algorithm needs aside from the dimension `dim` and the given processor topology `topol[]` the `rank` of a processor and returns the `position[]` of this rank in the $n-$dimensional processor topology. The result is illustrated by Figure 6.4. By this algorithm it is always guaranteed that the wide area connections are located between the right groups of processors.

6.2.4 Discussion

The algorithms developed in this section compute an optimal processor topology for a finite differencing code which is run on multiple machines, a *Grid aware processor topology algorithm* (GAPTA). Optimal means that the amount of data exchange is simultaneously minimized for the wide area network and the intra-machine communication. For "regular" processor numbers it leads to a configuration which significantly decreases the traffic across the wide area network between machines. By "regular" processor numbers we mean in particular that the numbers of processors on each machine should be divisible. More precisely we mean that the number of processors in every subjob needs to be divisible by the number of processors in the smallest subjob. Otherwise our algorithms would not work and we would have to fall back to the classical scheme again. Figure 6.3 shows

what may happen in this case. The boundaries between machines do not form a flat plane anymore but rather an *irregular* area. Roughly we see several ways to deal with this situation:

- abandon processors on some machines to make the numbers divisible
- setup a trivial topology: line up all processors in the longest dimension
- do not allow such a job layout
- do nothing

In order to give up processors on some machines one would have to develop an algorithm that calculates which machine should abandon which processor in order to keep this loss of resources minimal. This information would need to be propagated to all other routines which is not feasible in a modular framework since the total number of processors cannot be changed during runtime on the MPI level.

The "trivial" topology can always be achieved easily and looks e.g. for a five dimensional computational grid like $1 \times 1 \times N \times 1 \times 1$ where the longest dimension would be the third. The advantage here is that this topology is *always* possible. The disadvantage is however a very ineffective intra-machine topology which, in the wost case could maximize the intra-machine communication (and as such the redundant computation of ghost zones).

The option to simply not allow such a configuration is in our opinion the one to choose. The question arises whether our code has to deal with such a problem at all or does it rather has to be solved at some higher level, either by the user or by a portal or resource broker or similar. Currently we feel that the divisibility of the processor numbers is not too restricting since most queue systems on most supercomputers or clusters allow a maximum processor number which is a power of two and as such is always divisible by another (smaller) power of two.

The option to do nothing is also valid as long as we do not try to specify a different number of ghost zones for inter- and intra-machine communication (see chapter 2.3). As illustrated by Figure 6.3 the machine boundary is not a plane and thus the ghost zone size cannot be changed consistently without changing it also for intra-machine communication where it is not needed.

6.3 Domain decomposition, Load Balance

Regular domain decomposition methods of computational grids are extensively studied in the single machine or cluster case (see e.g. [56])

For codes which are executed in a real Grid computing environment however, the requirements for a proper load balancing and domain decomposition are quite different and are often the key to achieve a high efficiency. The machine setup in [7] used 480 R10000 processors and 1020 Power-3 processors to calculate a set of differential equations on a three dimensional computational grid. Given the fact that the Power-3 processors are about as twice as fast in respect of floating point performance an equally distributed load would cause 1020 processors to idle half of their time. But since they represent 4/5 of the computational power, the total efficiency would not exceed $3/5 = 60\%$ even for an ideal case of no communication and parallelization overhead.

In Section 6.2.2 we have seen how the processors are arranged, i.e. which processor has which neighbor, but we did not specify how big the part of the computational domain for each processor is. In order to construct a suitable *domain decomposition* we first want to analyze the situation. We have one single MPI job which consists of one or more subjobs. The total number of processors is

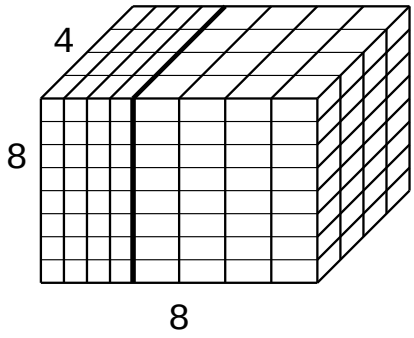

Figure 6.5: The domain decomposition of a three dimensional computational grid for a $320 \times 320 \times 160$ grid. We have 256 processors distributed equally on two machines. The processor topology here is calculated to $8 \times 8 \times 4$. Since the second machine is about twice as fast as the first, it gets twice as much grid points. The thick line again marks the machine boundary.

the sum of all processors in the subjobs. Further we assume that we have *one type of processor per subjob*. From Section 6.2.2 we know that the processor topology is done in a way that each subjob-set of processors forms an n-dimensional cuboid and that the neighbored subjob is always located in the dimension with the highest number of grid points (the "longest" dimension). Figure 6.5 illustrates this in three dimensions.

Taking these facts into account, the way of calculating the adequate domain decomposition is the following: The entire computational domain can be divided into slabs (see Figure 6.5) 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 a situation where n painters have to paint a wall but every painter works at a different speed, and all painters have to finish at the same time. The way of assigning parts of the wall's area A of adequate sizes to the painters is the following. The speed S_k, S_j of any pair of painters and their parts of the wall A_k, A_j should naturally be proportional:

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

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 same way we are calculating the distribution of grid points across the machines. 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 is taken by the `getrusage()` call.

Example: Consider a distributed run across 256 processors and two machines [Here, we already anticipate an implementation of most algorithms we discussed up to now. For a more detailed discussion of implementation issues see Chapter 8.], between SDSC and NCSA. We use two machines, an Origin 2000 with R10000 processors and 'blue horizon' an IBM SP equipped with Power3-processors. The global three dimensional grid size is $320 \times 320 \times 160$. The processor topology was calculated to be $8 \times 8 \times 4$, the subjobs are lined up in the first dimension. The floating point loop takes 2.41 seconds on the IBM SP and 4.40 seconds on the Origin subjob. The domain decomposition then contains 41×42 grid points on every processor in the y and z directions and the grid

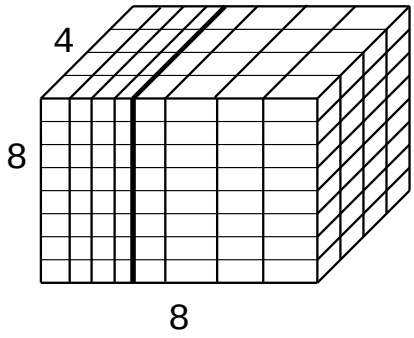

Figure 6.6: **The final domain decomposition.** This picture is similar to Figure 6.5 but has got a small correction in respect of the distribution of the workload. The slab of processors which are located at the wide area network boundary gets less workload than its colleagues in the inner regions of a machine. Thus they can already exchange data while the others are still computing

point distribution across the processors in the x direction looks like: 28+28+28+26+52+52+52+54 which means that blue horizon calculates about twice as much data than the Origin. Of course, since the number of grid points always has to be integer, it changes slightly the number of grid points on the last processor of the subjob (i.e. 26 and 54).

The domain decomposition for a distributed run as illustrated in Figure 6.5 seems to be "almost perfect". But still, since we are not living in a perfect world we have to take communication issues into account. It is obvious that only the processors that are located at the wide area network boundary are hold up by a slow connection and thus hold up others.

It thus would be a good idea to let those processors without wide area network connections still keep doing computations while the others are waiting for the data from the other machines. This can be achieved by giving the inner processors a slightly bigger workload than the WAN processors, see Figure 6.6. By this we already achieve a partial *overlap of communication and computation*. In our big distributed run [7] (see also Section 5) we intuitively gave those processors 20% less work and redistributed it to the others.

To estimate the amount of reduced workload for those inner boundary processors we unfortunately need data which we do not have at our disposal in the initial phase of our code where we set up all those things described here like load balance and topology. We would need to know the total amount of communication time per iteration but this is solely determined by the applications running within Cactus. We don't even know what the pure computation time for one iteration is since we have no information about which code is calculating what (using which algorithm) within the evolution loop. In order to be able to deal with this kind of problems we need to deploy *adaptive and dynamical* techniques which will be discussed in chapter 7.

For a first simple implementation (see Chapter 8) we code the amount of work to be redistributed to a fixed value of 20%.

6.4 Overlap of Communications in Different Dimensions

Let us recall how the updating of the ghost zones (the synchronization of grid functions) is done. As described in Section 2.5.4 the data communication happens sequentially using a loop through all dimensions. [For the following it is important to distinguish *dimension* and *direction*. Every dimension has two directions (call it positive and negative).] An asynchronous MPI_Isend and MPI_Irecv call is posted in every direction of a given dimension and finally a MPI_Wait is posted for both directions of the given dimension. Once the data in both directions has arrived it is copied from the receive buffers into the ghost zones (by a `memcpy()` call or similar).

This algorithm clearly treats all dimensions and directions identically. At this point it is quite natural to break this loop and treat different directions according to the nature of the underlying physical network. From our topology setup procedure we now *know* if there could be a slow network (e.g. a WAN) in a given direction or not. Let's call these directions WAN-directions.

For example we could replace the algorithm in 2.5.4 with the following, improved one:

```
Algorithm: SynchronizeII()
{
  if (one direction is a WAN-direction)
    {
      copy the data into buffers for this WAN-direction
      post an MPI_Isend and Irecv for this WAN-direction
      for (all other directions)
        post Isends and Irecvs
      for (all directions)
        {
          if (MPI_Test > 0 in this dir)
            copy data from buffers into the ghosts
        }
    }
  else
    use old algorithm form Section 2.5.4
}
```

As opposed to the classical algorithm this one first tries to identify whether there is a connection with a WAN in some direction (from what we have said in the last sections follows that it can't be more than 2). If such a direction exists the code sends the data in this direction first using an asynchronous MPI call and also posts an MPI_Irecv for the expected data from the neighbor in the WAN-direction. Without waiting for that data it now loops through all remaining directions posting again asynchronous sends and receives. After this is done the code permanently tests all directions for incoming data and copies it from the receive-buffer into the ghost zones if it has arrived. If there is no WAN-direction, the old algorithm can be used.

It is quite obvious that this procedure can save time as compared to the old one since intra- and inter-machine communication do overlap. In principle we now would have to analyze or at least estimate the performance benefit we gain by making this change to our code. For this we first implemented a test version of this idea. However, since we always deal with real production applications, we have to perform a small reality check. Specifically we mean that we need to run this

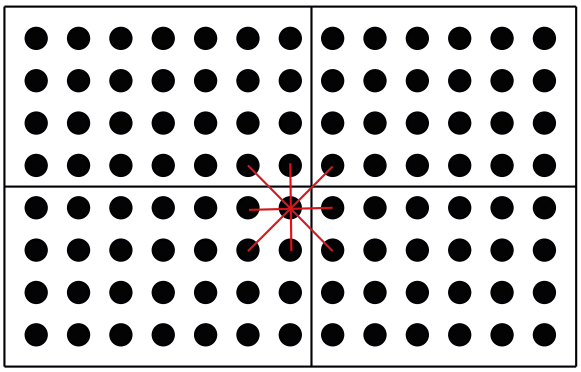

Figure 6.7: This figure shows a simple two dimensional computational grid divided up on four processors. The code has a stencil size of 1. The stencil is illustrated as a set of red lines in this picture. In order to update the grid point in the middle of the stencil it needs the values of all points which are covered by the stencil, which in this case also includes the diagonal points.

algorithm within the testsuites of those applications. Unfortunately we realize that some testsuites fail!

In order to understand why they fail we should take a look at Figure 6.7. The point (in the middle of the stencil in this figure) needs the information of *all* neighboring points and as such also the ones in the diagonal. This happens e.g. in finite differencing codes that need to take a cross derivative like

$$\frac{\partial^2 f}{\partial x \partial y}.$$

Looking again at Figure 6.7 one realizes that those points in the diagonal are located on processors which do not "face" each other. The question now is how the information of that point propagates to the other one since our synchronizing mechanism only exchanges information between processors that do face each other.

The answer is that this point gets synced "for free" if and only if the communications in (here) x and y direction *do not overlap*. To get a clearer idea of what we mean we illustrate this in Figure 6.8 in more detail.

The point in the upper right corner of processor 0 needs to be synced with the ghost point belonging to the processor to its right and to the processor above it. Suppose that we first synchronize in the x direction. In that case the upper right point of processor 0 updates the corresponding ghost point on its right neighbor. Once this information has arrived we synchronize all processors in the y direction. It is now important to understand that this previously updated ghost point on processor 1 now itself updates a ghost point on processor 2. This freshly updated ghost point now corresponds to the real grid point from the processor 0 which lies diagonal to processor 2. In the case that we don't wait for a synchronization in one direction to be finished but do these communications in parallel that diagonal ghost point never gets updated.

There are now several possibilities to proceed

- check if the application code has a stencil which uses points in the diagonal. Overlap communications in different directions if not, otherwise don't overlap
- overlap all communications and in addition send the "corner" point to its diagonal neighbor

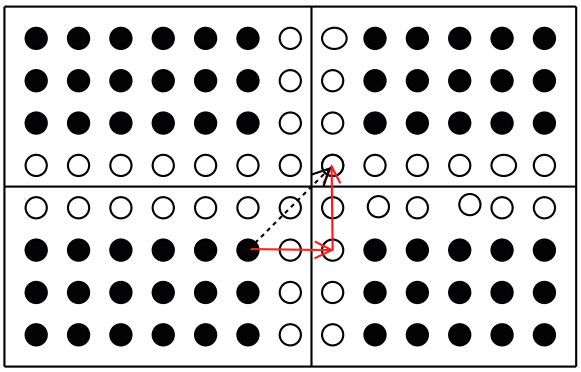

Figure 6.8: As shown in Figure 6.7 one also has to synchronize a ghost point in the "diagonal" direction. There are two possible ways of transferring this information. One way is to synchronize processors in the non-diagonal direction *sequentially* as indicated by the red arrows. The other way, indicated by the dotted black arrow is to synchronize this point directly which means to initiate a communication with that processor which lies in the diagonal line.

in a separate MPI call

- don't do overlapping

About the first possibility: It seems quite hard, if not impossible to check whether a running application code would need this point or not. At least to our understanding there is no way to check for that.

The second possibility is – as opposed to the first – possible to implement. Unfortunately things might get complicated in higher dimensions. In two dimensions there is one corner point that has to be send separately to one processor. In three dimensions we have a row of points which need to be sent to three processors. In four dimension we have a plane of corner points which need to be sent to five processors. This would involve five more WAN communications (in the Mpich-G2 and PACX case using TCP/IP) sending a rather small amount of data. When taking possibly high latencies into account this procedure seems not feasible apart from the fact that the implementation is not trivial.

What remains is the third possibility not to overlap the communications in different dimensions. For now it is the option we choose but compared to our old method from Section 2.5.4 we can still maintain full overlapping of communications in different directions in the same dimension. Recall that the communications itself (i.e. the MPI_Isend/Irecv calls) are already overlapping in Section 2.5.4, but the sequential MPI_Wait operations prevent the code from taking advantage of it.

Thus the only thing which could be implemented is to replace the MPI_Waits with a MPI_Test loop across both directions of a dimension which yields the following

```
Algorithm: SynchronizeIII()
{
  for (all dimensions)
    {
      for (all directions in this dim)
        post an MPI_Isend and Irecv
      for (all directions in this dim)
        {
          if (MPI_Test[dir] > 0 )
            copy data from buffer [dir] into ghost zones
        }
    }
}
```

This algorithm can be seen as a compromise between our old algorithm from Section 2.5.4 and the fully dimension-overlapping algorithm from the last section. The communications in different directions in one and the same dimension now fully overlap. If one direction is a WAN direction while the other is not, the data of the latter one naturally arrives first. Thus the code can copy the data from the receive buffer into the ghost zones while the other side arrives. This algorithm passes the testsuites.

6.5 Overlap of Computation and Communication

In Section 6.3 we already studied a possibility to achieve an overlap between computation and communication. This overlap was an overlap involving several processors. Here we examine possibilities how to achieve such overlap on the same processor. The requirement is simply that the processor should calculate, i.e. loop through (parts of) the computational grid while data is being exchanged with neighbored processors at the same time.

In principle there are several methods to implement this and here we discuss two of them.

6.5.1 Method #1

Let us recall what the structure of a finite difference code running within the Cactus framework is. On every local processor we have one or more n-dimensional arrays which represent values of discretized functions. In order to calculate the evolution of such a function (to update the grid points) the application code performs an $n-$dimensional loop through that array and after that it requests a synchronization of all functions it took derivatives from.

```
Evolution_routine(nx,ny,nz)
{
  do  i=2,nx-1
    do j=2,ny-1
      do k=2,nz-1
        array1(i,j,k) = (array2(i,j+1,k)-array2(i,j,k))/2 + ...
      end do
    end do
  end do
  synchronize(array1, array2,..)
}
```

The above routine represents a typical evolution routine in numerical sciences. The idea is now to achieve full overlap of communication and computation by splitting the (Fortran) loop into smaller subloops. Ghost data which is already calculated can be sent to the neighbored processor, without waiting for it and the loop can proceed to calculate the next chunk. This can be achieved by changing the application in the following way: Instead of looping from 1 to nx (or from 2 to nx-1 etc.) all applications should loop from `from[1]` to a `to[1]` where `from[]` and `to[]` are indices representing the local sizes of the grid on each processor.

The routine would then look like:

```
Evolution_routine(from[],to[])
{
  do  i=from[0]+1,to[1]-1
    do j=from[1]+1,to[1]-1
      do k=from[2]+1,to[2]-1
        array1(i,j,k) = (array2(i,j+1,k)-array2(i,j,k))/2 + ...
      end do
    end do
  end do
  synchronize(array1, array2,..)
}
```

Suppose a slow network connection underlies all communications in the y-direction. To perform the overlap, we need to split the main evolution loop into several blocks. The number of blocks (*nblocks*) can be left as an external parameter. Thus, the main evolution loop could look like:

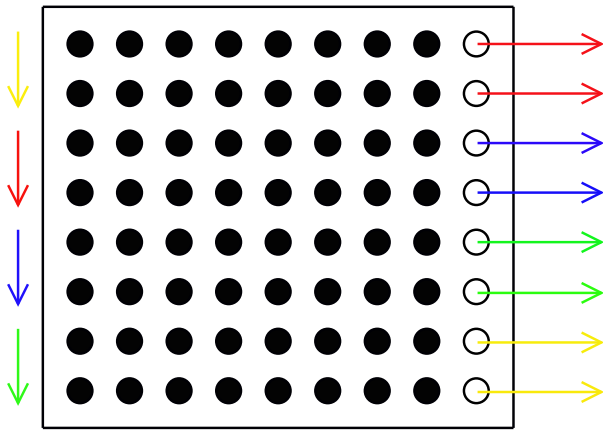

Figure 6.9: One method to achieve an overlap between computation and communication. The horizontal arrows indicate communication while the vertical ones in the same color represent the part of the computational loop which is performed at the same time. Since the data is only needed in the next iteration the communications for these data rows can overlap.

```
Evolution_loop(int nblocks)
{
  ind dim; [dimension of the grid]
  int split;
  for(i=0;i¡dim;i++)
    from[i] = 0; /* to[]=nx,ny,nz etc.. */
  split=to[1]/nblocks
  to[1]=split;
  for(i=0;i<nblocks;i++)
  {
    Evolution_routine(from[],to[]);
    from[1] = to[1]+1
    to[1]+=split
  }
}
```

Note that the `synchronize()` function will now only send data according to the `from[]` and `to[]` arrays. It does not wait for the sends/receives to complete until the data is needed (i.e. in the next iteration).

Figure 6.9 illustrates this procedure. While sending one set of already computed ghost points we can already calculate the next set by which we achieve a full overlap of computation and communication.

Discussion. Up to here it seems that the proposed overlap of computation and communication is quite feasible and easy to implement. Unfortunately this method of overlapping has a major downside and also the implementation is not that easy. In Section 5.6.3 we already brought forward the argument that coalescing small messages into bigger ones helps to hide the latency of the wide area network from the application code. If we implement the method discussed here then we are

practically doing the opposite: We are cutting messages into smaller pieces and thus increase the effect of high latency. However, this may or may not lead to a decreased communication time. It depends on whether the evolution of the single parts of the grid takes longer than the latency of the network. If yes, this procedure is beneficial, if not we lose efficiency. Apart form this we introduce more overhead by introducing more buffer copying and thus increasing the chance for cache misses. Furthermore we have to investigate what requirements the Fortran, C, or C++ code has to fulfill in order to interact smoothly with this technique and not causing its testsuite to fail.

Assume an application routine which looks like this:

```
Evolution routine (Fortran, C, or C++)
{
  do  i=1,nx
    do j=1,ny
      do k=1,nz
        function1(i,j,k) = ...
      end do
    end do
  end do
  synchronize(function1, function2,..)
  do  i=1,nx
    do j=1,ny
      do k=1,nz
        function1(i,j,k) = ...
      end do
    end do
  end do
}
```

Our goal is still to implement all discussed techniques and algorithms within a modular framework (like Cactus) and as such we do not change the application code. For the above example it would not be possible to implement our overlapping technique in this way since the correct values of *function*1 and *function*2 are needed for the second loop but they did not arrive yet since did not wait for them in the previous synchronization call.

However, there is one way to get it all working by *requiring* the application modules only to have one n-dimensional loop per scheduled evolution bin. Every additional loop which is performed to calculate variables which need the ghost zones has to be scheduled in an additional evolution bin such that our overlapping technique can work.

Summary. The technique described above requires application codes to be rewritten and as such does not seem to be worth to implement. Furthermore it is not clear whether this overlap leads to a performance benefit or not since it depends on the ratio of computing time and network latency.

6.5.2 Method #2

The last method introduced had the disadvantage that latency effects may get increased since bigger messages are divided up into smaller ones. A way out of this is the next method which will be introduced in the following. It does not split messages but nevertheless achieves an overlap

between computation and communication.

Suppose that the applications are not calculating their loops from 1 to nx but from a field `from[]` to another index field `to[]` like in the previously discussed method. Suppose further that a slow network connection is associated with communications in the z-direction. The parallel code could then set `from[0]=from[1]=1, from[2]`=nz-ghost_z where ghost_z is the ghost zone size in the z-direction. Further we set `to[0]`=nx, `to[1]`=ny, `to[2]`=nz. If now the evolution routine is called with those parameters it loops through all ghost points and calculates the updated values which could be sent to the neighbored processor using an asynchronous send. Then one could set `from[0]=from[1]=1, from[2]=1` and `to[0]`=nx, `to[1]`=ny, `to[2]`=nz-ghost_z and call again the evolution loop which would compute the rest of the array while data with the neighbored processor is exchanged.

Although this method does not have the disadvantage of increasing latency effects it still requires the application code to be coded in a special manner as already needed in method #1. While not impossible, it still would require a lot of work to adapt all existing applications to be able to use this technique.

6.6 More Methods to Improve Performance

The methods discussed in this section deal with the reduction of the amount of data to communicate.

6.6.1 Using Single Precision

One way of immediately reducing the amount of all communications by a factor two is to use single precision. Recall that numerical codes are mostly exchanging floating point data. As we know a floating point representation can be in single or double precision (according to the number of the so called 'fraction' and 'exponent' bits in the IEEE standard for floating point representation) where single precision uses only 4 bytes as opposed to 8 bytes for double precision.

However, performing numerical calculations in single precision may or may not be a suitable choice for the numerical problem. For numerical relativity for example the usage of double precision is indispensable. A simulation of colliding neutron stars in single precision would not be possible because the stars would disappear in the noise (due to the low precision) before they would collide.

6.6.2 Omitting Messages

Bal et al. [10] suggest to simply *omit* messages. We know for sure that this will break our testsuites, but going one step down, we could ask what weaker conditions we could require the code to fulfill. The answer to this question is quite difficult as we don't want to (and cannot) deal with issues on the application level.

However, we emphasize that for many branches of numerical sciences it is hard enough to obtain a code that can perform a long time stable calculation. Omitting messages is not a very helpful option and no numerical scientist would seriously consider doing it.

Moreover, the way to measure and rate this deviation differs from application to application and would even require to (manually) compare the run with a – previously done – fully synchronized one.

These contemplations already show us that it will be virtually impossible to implement an automatic decision mechanism on the driver level of a parallel code that switches message dropping

on/off.

It thus remains the possibility to manually specify a parameter where the scientist can decide whether he/she allowes message dropping in a distributed run. This seems to be the most sensible way (if at all) since only the scientist itself can decide about the trade off between faster code execution and precision.

Chapter 7

Dynamic Strategies for Performance Improvement

In the last chapter we investigated various possibilities, algorithms and techniques to deal with Grid computing related problems like load imbalance, different network qualities between processors and similar. However, the nature of the Grid is not only a distributed and heterogeneous one but also a dynamic one.

The term dynamic not only refers to the fast changing quality of networks but also to many other building blocks of the Grid. Machines can come up or shut down, the load on the machines can change in time, scheduling systems can change their configuration and even the running code itself can change its algorithms, change the type of data which is calculated and thus also the requirements in respect of computational and communicational resources needed.

An important goal of an application running in a Grid environment is to be able to deal with all those changes and adapt itself as well as possible to the changing environment. In this thesis we can only deal with a small fraction of all possible changes a running code could adapt to.

Since our goal is to achieve a high performance we now discuss techniques to deal with changing network properties like latency and bandwidth as well as changing loads on machines while a distributed code is running. Changes of the Grid that affect queuing systems or even entire machines (coming up or shutting down) are treated in related works [42, 43].

Another important point is that the application scientists does not need to know any of these details – the application itself should adapt autonomously to changing conditions.

In this chapter we discuss various techniques we have developed for real codes and production computing environments to address the issues of changing conditions and automatic adaption.

As seen in Chapter 5 there are possibilities to deal with a low bandwidth and a high latency: Compression and the usage of multiple ghost zones.

7.1 Compression

We have learned from Section 5.6.4 that data compression prior to send operations may speed up the communication. This gave us an enormous performance benefit since the compression factor of the data we used in those runs was very high. In the general case this may or may not be an adequate option, since it strongly depends on the compression ratio of the dataset and the time it takes to compress. If the time it takes to compress a chunk of data is smaller than the time one saves by sending a smaller package then compression makes sense. The two most important

notions are thus compression time and compression rate (or factor).

7.1.1 Compression Algorithms

Both compression time and compression rate depend crucially on the compression algorithm which is being used. Among all compressing algorithms nowadays the JPEG[1] algorithm seems to be the most known. It is used to compress two dimensional graphical data. In a slightly modified version this algorithm is also used to compress video and sound data where the resulting format is called MPEG[2] (e.g. MPEG layer 3, better known as mp3).

Looking at the compression ratios by comparing e.g. the sizes of an uncompressed image format (e.g. GIF[3]) with the size of the corresponding JPEG image or by comparing the size of a mp3 file with the size of the same file on an audio CD one realizes compression factors of 10 and more.

However, for our purposes these algorithms are not really suitable. Apart from the fact that they specifically have been developed to compress graphics and sound data (and we deal with floating point data) they are unsuitable because they belong to so called *lossy* algorithms which means that compressing and uncompressing a dataset does not necessarily reproduce the original, uncompressed dataset. For some scientific problems, however, lossy compressoin may be adequate (e.g. visualisation), but for solving PDEs it is unacceptable.

Therefore we rather need a *lossless* compression algorithm which is specifically developed to compress floating point data. However, as far as we know, no such code exists up to now, but there is is a whole range of codes supporting lossless compression such as `zip, bzip2, arj, gzip, compress` and others. They all use variants of either the *Huffman* or the *Lempel-Ziv* algorithm. For implementation reasons, however (without comparing the properties of the above algorithms) we decide to choose the `deflate()` algorithm from the `zlib` [18] package. Mainly there are two reasons for this: First, to achieve a short compression time we need a code that compresses data which lies in memory. Second, since we want to use it in Grid computing, we need a portable code. Both conditions are met by this package which we already used in Chapter 5.

However, our techniques are rather independent of the compression library used. In future implementations we may use other compression schemes.

7.1.2 Compression Time vs Compression Factor

The goal of this section is to predict under which circumstances compression makes sense. It is not possible however to calculate this for the most general case since compression time as well as the factor depend on a rich variety of things like the local processor speed, the type of data, the amount of data, the algorithm and other things. Thus we will try to make a good choice by selecting "typical" and fixed values for the above quantities. We choose three typical datasets in numerical relativity: a black hole, a wave pulse, and a two-black-hole dataset. All datasets are solutions of Einstein's equations. We also choose a fixed grid and ghost zone size as well as the `deflate` algorithm introduced in the last section. For the processor type we choose an Intel Pentium II and the IBM Power-3.

We use a simple formula to calculate the time for data transmission:

$$t_{total} = t_{cmpr} + \frac{\text{message\_size} \times \text{compr\_factor}}{\text{bandwidth}} \tag{7.1}$$

[1] Joint Photographic Experts Group
[2] Motion Picture Experts Group
[3] Graphics Interchange Format

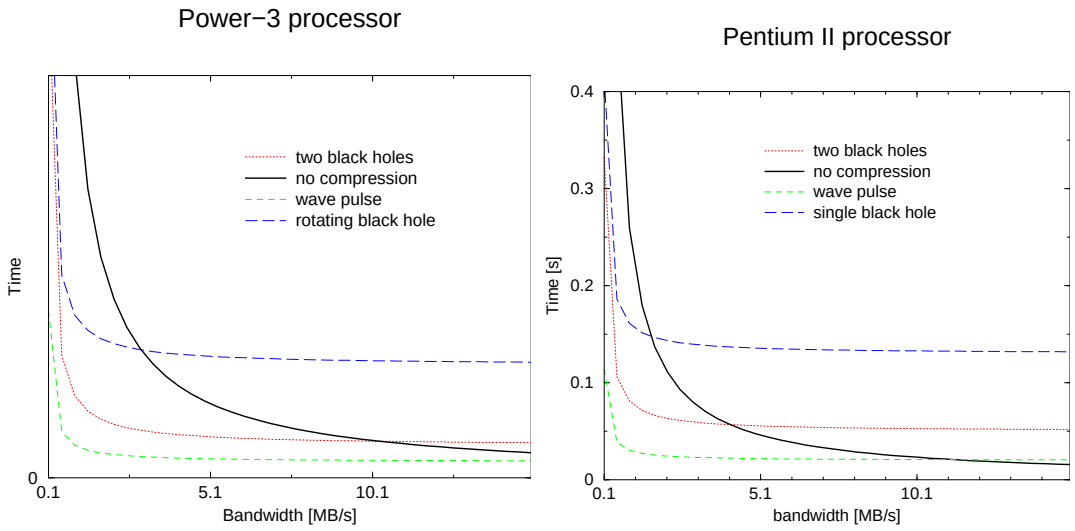

Figure 7.1: **When does compression make sense?** The total communication time is plotted on the y-axis, while we vary the bandwidth on the x-axis. The thick solid line represents communication times without compression, which naturally is the same for any kind of dataset. The point where this line crosses another one marks the point where compression does not make sense any more because it takes more time to compress than one saves during the transmission.

where t_{compr} is the time needed for compression. If there is no compression, t_{cmpr} is zero while compr_factor is one. For the message size a typical value of $\sim 235kb$, (as in the DTF run) is used. Latency is not taken into account because our goal is not to calculate absolute values for the communication time, we rather compare communication time for our three datasets with and without compression. Note that latency would just be an addend in eq. 7.1, which "cancels out". For a fixed dataset, data size and algorithm we have a fixed compression ratio (for our datasets, the compression ratio did not significantly change with the message size, not shown here). In this case the communication time solely depends on the bandwidth. Plotting the time against the bandwidth in Figure 7.1 we immediately can read off the critical bandwidth, under which compression still makes sense. In formula 7.1 it appears like this point depends on the message size, but in principle it does not because the compression time grows almost linearly with the message size (not shown here). Conclusion: Compression makes better sense at low bandwidth rates, high processor speeds and high compression ratios.

7.1.3 Overlap of Compression and Communication

The last section implicitly used the fact that the data transmission and the compression operation do not overlap. We discussed a similar situation in Section 6.5 where computation and communication overlapped. The procedure would be very similar. We cut the whole dataset into pieces, and start compressing the first chunk. While posting an asynchronous send for it we compress the second one and so forth. The question whether this gives us a performance benefit or not strongly depends on the relation between compression time and bandwidth. It would only make sense to use this option in the case that compressing a chunk takes less time than sending it, otherwise we just increase latency effects.

7.1.4 Compression of Incremental Data

In the case that given floating point datasets do not compress very well (e.g. the data is very noisy), i.e. it does not give us a benefit, one could apply the following trick: Like in backup procedures of storage systems one could – instead of sending the dataset itself – send the *difference* between the current dataset and the one in the last synchronization process. Without compression the amount of data remains the same, but if the evolution of the dataset is sufficiently slow (which should be the case for high resolved datasets) this incremental dataset should be close to zero and as such should compress very well.

However, in some cases incremental data is sent even without explicit implementation in the driver code. It happens on the application level! Suppose our numerical code solves a system of first order partial differential equations of the following kind:

$$\dot{\gamma}(x,t) = K(x,t) \tag{7.2}$$

$$\dot{K}(x,t) = \frac{\partial}{\partial x}\gamma(x,t) \tag{7.3}$$

(these could be Einstein's field equations in a one dimensional form). A spatial derivative is taken from the function γ so this function is clearly the one which needs to be synchronized, i.e. parts of its data have to be exchanged through the ghost zones. Suppose a discretization scheme does the following

$$\gamma(t+\Delta t,x) = \Delta t K(x,t) + \gamma(t,x) \tag{7.4}$$

$$K(t+\Delta t,x) = \Delta t \frac{\gamma(x+\Delta x)-\gamma(x)}{\Delta x} + K(x,t) \tag{7.5}$$

and updates equation (7.4) on the even time steps (iterations) while equation (7.5) on the odd time steps (we call this scheme "staggered leapfrog", the odd time steps are called the staggered time steps). The application code now has two choices: It may synchronize γ *before* updating equation (7.5) **or** K *before* updating equation (7.4). Some codes choose to do the latter. Recall that K is the time derivative of γ so that we achieve an exchange of incremental data on the application level.

This phenomenon is very interesting since the functions γ and K may have completely different shapes and as such a totally different compression behavior[4].

It is hard, if not impossible to predict which function compresses better. The only solution is to simply test, on the fly during code execution which method (incremental or normal exchange) gives better compression rates and as such better performance.

7.1.5 An Adaptive Compression Algorithm

As the previous subsection has shown, compression ratios and times can vary as much as network quality, depending strongly on the type of data, processor speed, resolution etc. It almost seems to be impossible to make an intelligent decision about compressing or not for this variety of cases.

Taking into account however that the compression times and ratios do not change very fast in time (for most of our test datasets) one could monitor the code performance, test wise switch compression on/off, keep monitoring the performance for an iteration interval yet to be specified and compare with the previously achieved performance. If it results in a performance benefit, leave

[4]This leads us to the question about whether there is a minimal information which needs to be exchanged

compression state as is, otherwise revert to the old state. Of course this only makes sense under the assumption that the compression ratios and times as well as the network bandwidth do not change within this monitoring interval.

To introduce compression the synchronizing algorithm from chapter 6.4 would then look like:

```
Algorithm: SynchronizeIV()
{
  for (all dimensions)
    {
      for (all directions in this dim)
        if (compression=yes)
          compress Sendbuffer
        post an MPI_Isend and Irecv
      for (all directions in this dim)
        {
          if (MPI_Test[dir] > 0 )
            {
              Size = MPI_GetCount
              if (Size > uncompressed Size)
                uncompress receive buffer
              copy data from buffer [dir] into ghost zones
            }
        }
    }
}
```

It is worthwhile to note that the peer processor does *not* have to know whether a message comes in compressed or not. The processor on the receive side simply posts an asynchronous MPI_Irecv with the known size of the uncompressed message. After the message arrived it compares the received message size with the expected one. If it is smaller the processor can assume compression and decompress the message.

From our investigation in the previous section, especially from figure 7.1 we know that for very fast intra-machine connections of up to 100MB/s compression almost never makes sense. We thus never try to compress data over those connections and restrict ourselves to wide area network connections. This is possible since – as outlined in Section 6.2.2 we know where those connections occur and where not.

The algorithm to apply adaptive compression then looks like the one described below. Note that two basic parameters are free to choose: The *monitoring interval*, i.e. the time (or iteration-) interval the code is accumulating performance data. This interval should not be too short since our algorithm is designed to react to systematical changes in performance, not statistical changes, that occur every iteration. The other free parameter is the interval at which the whole algorithm is repeated. This depends on empirical data about how fast network performance may change. We think that a value of 200 is a good place to start. The algorithm below will be called every iteration by the main evolution routine.

```
Algorithm: Adaptive_compression(int iteration))
{
  int mon_int /* monitor interval */
  int rep_int /* interval the algorithm is repeated */
  int k
  /* compression is off initially */
  k = (iteration modulo rep_int) / mon_int
  switch(k)
  {
    case 0:
      start_timer(time_prev)
      break;
    case 1:
      stop_timer(time_prev)
      break;
    case 2:
      change_compressio_flag()
      start_timer(time_now)
      break;
    case 3:
      stop_timer(time_now)
      if(time_prev < time_now)
        compression_flag = 0; /* no compression */
      else
        compression_flag = 1; /* compression */
      break;
  }
}
```

After mon_int iterations the performance benefit of compression is checked again. However, as we have seen in the previous Section it strongly depends on the type of data whether compression makes sense or not. Since within one numerical calculation more than one type of data (more than one discretized function) it would be much more advantageous to run the above algorithm separately across every synchronized dataset.

Additionally, the algorithm should test incremental compression and the overlap of compression and communication. For each dataset we have to find out the best combination of compression/overlap/incremental or no compression and stick to the configuration that brings the best performance.

It is worthwhile to mention that (unlike the choice about whether to compress at all or not) in the incremental compression and overlap case the peer processor needs to know whether the data is e.g. an incremental data or not. This leads to additional communication and a more complicated code, that we did not implement within this thesis.

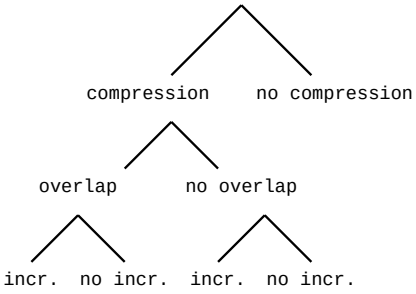

Figure 7.2: From all possibilities using compression a smart algorithm should pick the one with maximal performance.

7.2 Dynamically Applying the Adaption

We saw in the last section that the procedure we propose includes two free parameters that are statically set (or read in from a parameter file) initially. For example, we choose to repeat this algorithm every 300 iterations. We may claim that we then are able to dynamically react to network bandwidth changes, but we always have to wait to the next 200th iteration.

It should be possible, however, to constantly monitor the bandwidth and therefore immediately react on a sudden change. To do this, we have all data we need: We know how many bytes we send across the network and we measure how long it takes, thus we permanently know the current bandwidth. Then, we don't repeat things every 200 iterations but rather every time the bandwidth drops or increases significantly. However this again introduces a bunch of new parameters to choose: Over how many iterations we want to average the bandwidth (there will be some noise) and what is the threshold number (e.g. in percent) that makes a bandwidth change "significant"?

7.3 Performance Monitoring

In order to be able to implement the procedure described in the last section a history of performance numbers has to be maintained and based on these statistics, the driver code should be able to make decisions on whether or not changing the compression state.

One choice for those performance numbers could be the total wall clock time spent in the communication routine measured by e.g. `MPI_Wtime()`. Unfortunately this is not representative for the entire code. It may well happen that our routine spends less time in communication but the overall performance of the code gets worse.

This problem can be solved by deploying a more professional solution, namely to use the software package PAPI [45] to obtain detailed performance statistics for the running code on the fly. PAPI provides a machine independent Performance API (hence the name), which allows the driver to obtain the required metrics. The main quantity we are interested in is the current floprate. All decisions made during runtime to improve the performance should be based on this number. However, we still don't always do this, even if PAPI is installed. This is because we then could not try compression of all datasets simultaneously, since the overall effect may cancel out. In this case

we thus stick to measure the communication time for each dataset.

General remark about performance monitoring. Due to the dynamic nature of wide area networks the performance of a code may change rapidly, even if there is no adaption (see e.g. Figure 3.3). It thus may happen that this "natural noise" influences the decision of the adaptive elements of the driver code in a wrong way. This is a serious problem. We observed that this problem gets better with higher number of processors (when the data stream saturates the network). In order to get rid of this noise we need to monitor and average the performance over a longer time interval. This however limits adaption, which is supposed to react to changes in the environment. To keep this balance is a still unresolved issue. Currently the lentgh of the interval can be set as a parameter in our implementation.

7.4 Adaptive Latency Hiding Algorithm

We have seen that increasing the ghost zone size reduces latency effects. Sending 50 messages per iteration at a latency of 50ms yields an additional latency overhead of 2,5 secons per iteration. Having 5 ghost zones reduces this time to half a second.

In Section 5 we increased the ghost zone size to 10 in order to minimize these latency effects. This number may have been too high as its choice was not based on theoretical considerations.

In this Section we introduce an algorithm that tries to determine the "optimal" ghost zone size for a given application.

The code that we have run in the DTF performed one send operation per iteration. Current codes used for latest scientific calculations perform up to 50 synchronization operations per iteration. Calculating the optimal ghost zone size for a given environment and a given code seems to be even much more demanding than to predict the benefit of compression.

Thus, following the ideas of the last sections it seems natural to proceed the same way, i.e. to change the ghost zone size during code execution and settle down to a number where the performance is maximized.

However, there are several differences compared to adaptive compression:

(1) Compression can be either on or off. The ghost zone size however covers a one dimensional parameter range and as such it will take more time and overhead to find out the optimal number.

(2) The decision about the number of ghost zones cannot be made processor-wise. The peer processor always have to have the same number of ghost zones.

(3) Furthermore the number of ghost zones has to be consistent across the whole *plane* (see Figure 7.3).

(4) Changing the number of ghost zones introduces additional communication overhead since the ghost points have to be filled with data from the neighbored processor.

(5) Increasing the number of ghost zones leads to the need of more memory and increased CPU usage.

One of the requirements for algorithms and implementations to find the optimal number of ghost zones is that the communication produced by the change of ghost zones has to be as little as possible, namely the communication produced by keeping consistent ghost sizes and the communication

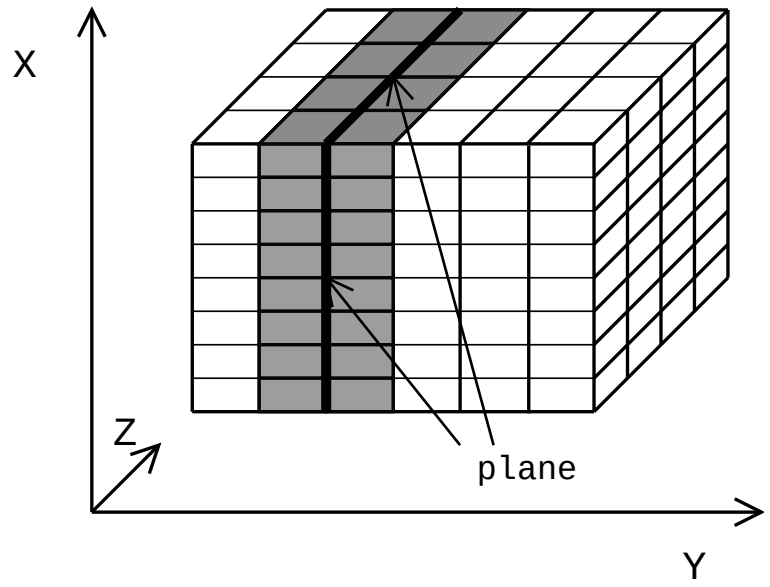

Figure 7.3: Ghost zones have to be consistent across the plane they form (thick line). In this example all grey marked processors have to have the same ghost zone number in positive y-direction (and as such have to exchange additional information in order to make consistent decisions about changing their ghost zone size.

produced by filling new ghost zones with data from the neighbored processor [This is even needed for functions which are normally not involved in synchronization processes to keep the number of ghost zones consistent through all discretized functions]. As pointed out in the above list additional communication is needed to keep the ghost zone size consistent in the whole plane. This overhead should also be minimal.

Since the network quality and also the code characteristics can change in time the adaptation should not be called only once at the beginning. It rather should try to continuously adapt itself to the environment. However, since we know that a communication overhead is involved in this adaptation process we don't want to call it too often. The length of the period should thus be limited by the ratio between the benefit this adaptation provides and the additional communication on the other hand. We leave it as a parameter and the user can specify him/herself how often the adaptation code should be called.

The following algorithm iteratively increases the number of ghostzones until a performance improvement is no longer achieved. As for our adaptive compression algorithm, we set the monitoring interval M to be a free parameter, as well as the number of iterations R at which this procedure is repeated.

(1) code starts up

(2) start timer

(3) code runs for M iterations

(4) stop timer

(5) increase ghost size by one stencil size

(6) start timer

(7) code runs for M iterations

(8) stop timer

(9) compare performance;

(10) if improved goto **(5)**

(11) if worse decrease ghost size by one stencil size and finish adaptation

This algorithm could be called again after R iterations with the difference that it tries to go *down* with the ghost zone size instead of up. After R more iterations, it again tries to go up and so forth.

For our test runs (see Chapter 9) we used the values $R = 300$ and $M = 20$.

Again, we stress that all these techniques do not change the application code or affect the application scientists work in any way (well, except that the execution should be faster).

7.4.1 Discussion

We have proposed two similar algorithms to adapt a running code to the Grid environment. Keeping overlap and incremental compression aside, the first algorithm tries to find out the best compression state (on/off) on a per processor and per dataset basis. The adaptation can therefore be very quick and effective since the set of parameters contains two elements (on and off) and no significant computation or communication overhead is involved.

Using the same algorithm for finding the optimal ghost zone size can take much longer since the parameter space is an endless discrete set of points. Moreover it causes additional communication overhead and as such should not be called too often. However, we will see that for our testruns this communication overhead is neglectable (less than 2% of the total execution time, see next Section).

7.5 Dynamical Load Balancing

7.5.1 Motivation

There can be many reasons to implement a dynamic way of balancing the load between processors and/or machines even for finite difference codes on a regular computational grid. It is unlikely that – when running in regular batch systems – the running job will have to share CPUs with other processes, but the requirements of the code itself can change. This can occur when the evolution code switches to another algorithm, calls an analysis routine or even uses adaptive meshes and nested grids.

Another reason for doing it could also be that the static load balancing fails for some reason. Imagine the floating point test loop which is used to achieve the initial static load balance is written in C but the main evolution routine is written in Fortran or maybe C++. Suppose further that the code has been compiled with different optimization flags for C and Fortran compilers on all machines that are involved in a distributed run [it is user's responsibility to choose the correct optimization flags for various compilers on various machines]. If on machine A Fortran optimization

flags are used while C optimization flags are not, and on machine B it occurs to be the other way around, the initial load balancing routine will clearly fail. There is no (easy) way to check if the main evolution routine is written in Fortran, C or C++ and apart from that it can change in time, as explained above.

Apart from pure CPU power related issues we already used load rebalancing for another purpose, namely in Section 6.5 where we gave CPUs which were involved in wide area network communication less workload, not because they are slower but because they are busy with communication. The rebalance, as outlined in Section 6.5 was hard coded to be 20% less but a proper rebalancing mechanism needs to know the exact value and that can be only obtained by dynamically monitoring the communication behavior.

7.5.2 Basic Requirements

In order to be able to implement a proper dynamical load balancing algorithm we have to investigate the following questions.

How to find out an imbalance. This item rather refers to the question how we want to *define* a load imbalance. In a parallel code processors never exactly finish at the same time.

How to rebalance. This question has similar characteristics as the issue about how increase the ghost zone size. This cannot happen on a per processor basis since also here a consistency has to be preserved.

How big is the overhead. Again this situation is similar to the scenario of adapting ghost zones dynamically. The overhead will certainly depend on how strong the imbalance will be since data has to be copied around the place.

How often to rebalance. This depends on the previous items.

The first item is closely related to the second one. We certainly do not want – nor are able to – take care of imbalances between every single pair of processors. The coarsest assumption is (as in Section 6.3) to assume that the processor speed is the same within one subjob so that the load balancing always would look like in Figure 6.5. However, as we already have seen in Section 6.5 it makes sense to remap the workload even within one subjob, achieving a domain decomposition which gives processors even within one subjob different workloads (see Figure 6.6). For the purpose of dynamical load balancing we want to stick to this "slab-wise" balancing.

7.5.3 Monitoring

The question about how to define a load imbalance is closely related to the question about which performance numbers we can access during runtime. Within a modular code framework it is possible to obtain detailed timing information. However, they may not be detailed enough to base e.g. load balancing decisions upon them. For example it is a priori *not* possible to exactly determine the total time a modular code spends with communication. The reason for this is that it is unknown which lower leveled communication protocol is used. Some of them cause the processor to be in a "spin-wait" state when waiting for data and thus accumulating CPU time. The behavior of the processor during communication can even change within one MPI implementation, e.g. Mpich-G2 (it uses different 'flavors', see [40]).

Our monitoring strategy will assume the following:

(1) Most communications will be initiated by our driver code. Communications initiated by other modules will be regarded as neglectable.

(2) We will measure the time a whole processor slab waits for his left and right neighbor in the highest dimension. Since we don't know the underlying communication code we will always use wall clock time.

(3) The difference between the total wall clock time for one iteration and the communication time in our driver code will be assumed to be a measure for the processor speed.

(4) The ratio between the above quoted times is regarded as the efficiency of the current slab (see eq. 2.1). If the efficiency differs more than a certain value (still to be chosen) between any pair of slabs, a load rebalance will be initiated.

The "slab wise" instrumentation of the code is necessary to achieve domain decompositions like in Figure 6.6, which was achieved at that time by a static and hard coded method which does not necessarily meet the requirements of the Grid environment. If we design a proper dynamical load balancing scheme, a domain decomposition similar to Figure 6.6 should be achieved automatically.

7.5.4 Strategy for Dynamical Load Balancing

Since we now defined a set of numbers to operate with we can design a dynamical load balancing scheme. Under the assumptions of the last Section we calculate the efficiency of every slab by dividing computation time by total run time. If there is a difference of more than say 10% between a pair of slabs we will rebalance.

The rebalancing algorithm will first – similar to what we have done in Section 6.3 – calculate the "processor speeds" which are represented as the reciprocal value of the computation time. Then for each pair of neighboring slabs the difference of the communication times (i.e. the time one slab waited for the other) is calculated. According to this time and the processor speed it can be calculated how many grid points have to be "shifted" to the faster processor in order to achieve a good load balance between this pair of slabs.

We will simulate this algorithm with a simple Perl script which begins with an initial equally distributed workload of 100 grid points and fixed processor speeds. From those speeds it can calculate the computation time and also the time difference to the neighboring slab. The simulation calculates the grid point distribution for a one dimensional grid. However, as pointed out before, we only redistribute the workload in the longest and as such this can be regarded as representative for arbitrary dimensions.

The results of this simulation can be seen in the following table. The first number represents the processor speed (which is now fixed in time). The higher this number the faster the processor (e.g. MHz). The next rows are the initial grid point distribution, the evolution time in seconds, assuming that the processors need 10000/speed seconds per grid point, the time every processor waited for its right neighbor and the resulting load balance. Communication times between slabs are not taken into account. Shifting the grid points between pairs of slabs does not necessary mean that the total amount of grid points remains constant. We calculate the number of grid points which is missing or which is supernumerary and distributes this number across the slabs according to the local speeds (i.e. in the same manner as in Section 6.3).

For a proper load balancing under these conditions our algorithm should yield a grid point distribution which is directly proportional to the processor speeds. The simulation of this algorithm

shows that this is far away of being true:

Processors	I	II	III	IV	V	VI
Processor speed	200	130	100	130	100	200
Initial grid Size	100	100	100	100	100	100
Evolution in seconds	5000	7692	10000	7692	10000	5000
Wait time in seconds	-2692	-2307	2307	-2307	5000	0
New grid size	144	123	72	123	45	90

This is not surprising since we only tried to achieve a proper load balancing between single pairs of slabs and as such it does not give us a perfect load balancing for the whole configuration. However, if we repeat this algorithm n times, where n is the number of slabs we get the following picture:

Processors	I	II	III	IV	V	VI
Processor speed	200	130	100	130	100	200
Initial grid distribution	100	100	100	100	100	100
Evolution in seconds	5000	7692	10000	7692	10000	5000
Wait time in seconds	-2692	-2307	2307	-2307	5000	0
Current grid distribution	144	123	72	123	45	90
wait time in seconds	-2307	2307	-2307	5000	0	0
current grid distribution	196	97	98	62	48	96
wait time in seconds	2307	-2307	5000	0	0	0
current grid distribution	165	137	55	72	55	111
wait time in seconds	-2307	5000	0	0	0	0
current grid distribution	216	75	58	75	58	116
wait time in seconds	5000	0	0	0	0	0
current grid distribution	139	90	69	90	69	139
wait time in seconds	0	0	0	0	0	0

Since the algorithm begins to redistribute the load of the slab with the highest index it is clear that one has to apply it n times in order to achieve a proper load balancing between every subjob. Looking at the final grid point distribution we unfortunately realize that four grid points are missing. We will discuss that later. Applying this algorithm n times does not mean that we have to redistribute the real data n times. We rather redistribute the load virtually in every step and keep all needed values (like processor speed, communication times etc.) constant. After we have done this n times we redistribute the load, including all communication costs.

7.5.5 Dynamical Load Balancing and Communication

Following the scenario in Section 6.5 we want to see what happens if we introduce a *slow network connection* between a pair of slabs. In our above example using 6 slabs we do this by constantly adding 300 seconds wait time between the third and the fourth slab. After applying our algorithm six times we end up with the following grid point distribution:

Processors	I	II	III	IV	V	VI
Processor speed	200	130	100	130	100	200
Initial grid Distribution	100	100	100	100	100	100
Final grid Distribution	150	97	55	71	75	150
Distr. without communication	139	90	69	90	69	139
wait time in seconds	0	0	0	0	0	0

We can see that our algorithm nicely distributes the workload such that still no processor (slab) has to wait for its neighbor. It happens in a way that we already anticipated in Section 6.5 namely that the processor slabs that are involved in inter machine communication get a smaller workload!

7.5.6 Summary and Discussion

We have designed an algorithm for a dynamical load rebalancing and we have shown that it always leads to a load distribution where no processor slab has to wait for another. Apart from imbalances which are caused by different computation times (caused by different processor speeds, different computations or similar) it can also cure imbalances caused by slow communication channels. As such it achieves a partial overlap of computation and communication. Load imbalance is here defined as an imbalance between slabs of processors and therefore this algorithm cannot resolve imbalances within one slab. Furthermore this algorithm is based upon reliable performance numbers, e.g. what amount of time does the code spend with computing and what amount of time does it spend in communication. In a modular framework those numbers are not necessarily available.

Although the algorithm tries to keep the total number of grid points constant it happens that due to roundoff errors a small number of grid points (small means fewer than the number of slabs) are missing or are supernumerary. Since the constancy of the number of grid points is essential they have to be distributed across the slabs and therefore lead to a deviation from a perfect load balance.

7.6 Putting it all Together

From experimental physics we know that one always has to vary only one parameter when performing measurements. In our case it means that we cannot perform adaptive compression and adaptive selection of ghost zone sizes at the same time as both decisions (compression on/off and number of ghost zones) are based on the resulting performance. Since load rebalance also changes the performance all three adaptive techniques have to be applied successively. Load balancing also takes communication into account and thus it should not be based on non-optimal communication schemes (wrong number of ghost zones, no compression etc.), which means that this should be called *after* a compression and ghost zone adaption have been performed.

Thus we decided to choose an overall adaptation scheme which could look like the following:

- Code starts up
- Apply initial (static) load balance algorithm
- Apply adaptive compression algorithm from Section 7.1.5
- Apply adaptive latency hiding algorithm from Section 7.4 finish adaptation

- Apply dynamical load balancing from Section 7.5
- Repeat everything every R iterations

A still open question is how often this procedure can or have to be repeated. R =300 iterations could be adequate for some applications or environments but not for all. We leave it as a parameter for now.

7.7 Summary and Discussion

This section dealt with adaptive and dynamic strategies to improve the efficiency of a running distributed code in a real Grid environment. The assumptions we made are very general. The algorithms of this chapter (together with the static ones of previous chapters) can deal with a mixture of slow and fast inter-processor connections within one distributed run as well as with a mixture of fast and slow processors. Both, the network quality and the speed of the processor (trigged by e.g. using different algorithms) can even change during runtime since our algorithms are applied again and again.

All techniques and algorithms discussed in this Chapter can be implemented in a way that the application code does is not affected. Moreover the application scientist does not need to know any of the underlying mechanisms.

The slow and fast network mixture as well as the mixture of different processor types cannot be too complex though. We always try to achieve a job layout as discussed in Section 6.2.2. For other layouts we currently do not have a suitable algorithm for adaptive load balance and ghost zone sizes (our adaptive compression method can be realized in every topology). We also assume that we always have access to any kind of performance numbers within a running code. As pointed out earlier this could be difficult within a modular framework.

The changing ghost zone sizes as well as rebalancing has to cause communication overhead – regardless of the implementation – as information has to be exchanged between processors and machines.

It thus remains to see if the benefits of the algorithms we designed in this chapter outnumber the additional communication (and computation) overhead. This is implementation dependent and will be treated in the following two chapters, where we implemented most of the methods described here.

Chapter 8

Implementation within a Modular Framework using Cactus and Globus

After we have extensively discussed methods and algorithms which help us to achieve a decent performance for distributed finite differencing codes in a Grid environment we now have to implement them into a suitable framework. Most ideas were discussed in a very general way which did not involve any special restriction to any underlying software. All issues treated in Chapters 6 and 7 are solely restricted to the problem of parallelizing finite difference codes on a regular grid (apart from Section 6.2.3, where we discussed the processor ordering of Mpich-G2). However, based also on the considerations in Chapter 2 we now want to discuss (and perform) a possible implementation of the most algorithms discussed in the last chapters.

8.1 Implemented Techniques/Methods

More precisely we implemented the Grid aware topology generating method "GAPTA" (from Section 6.2.1 and Section 6.2.2), the statical loadbalancing (Section 6.3 and Section 6.5), adaptive compression (Section 7.1) as well as the adaptive ghostzone algorithm (Section 7.4).

Furthermore, we implemented a test version of a code that performs dynamical loadbalancing according to our design in Section 7.5. It is a test version because it (at the current stage) changes the numerical results, so can't be used in production runs. For details see next Chapter.

8.2 Why Cactus and Globus?

One major goal of a proper implementation is that the implemented code should be widely applicable. Concerning the choice of Cactus and Globus as higher-ranking software frameworks, there are two basic sub-issues that were driving our choice towards these two toolkits. First, software projects in the Grid Computing field cannot be singular projects. Grid Computing is essentially about sharing resources and as such any software that claims to run in Grid environments must take care about other existing software and about de-facto standards concernig hardware and software. Also, we want to avoid redundant work by avoiding the implementation of techniques and services that already are implemented. In this respect it is important that we base our work on the Globus toolkit. The Globus toolkit laid a solid foundation of low level Grid services that became a world wide de-facto standard. Ignoring Globus would on one hand increase the implementation work and

also reduce the applicability of this work by far.

The other issue that is important for achieving a high applicability is the requirement that it can be applied to existing applications. This means in particular, that existing applications should not be rewritten. We thus need a parallel computing framework that supports a proper and transparent distinction of application code and parallel driver code. To our knowledge only the Cactus framework meets these requirements. Moreover, many scientific applications are already running within this framework. These applications are by no means test codes or demonstration codes. They are production codes used to produce latest scientific results [4]. These production codes can immediately take advantage of all techniques and methods we have implemented.

8.3 More Requirements for an Implementation

Following the arguments of the last section, we will list a few more items that we think are important for implementation. Unlike many publications in computer science where the ideas and methods that are being designed in papers or theses are only partly implemented or run only on very few platforms (or even just on one, namely the own laptop) it is a major goal of this thesis to make all methods, techniques and algorithms that were developed in the last chapters available to the numerical science community.

In particular we pay special attention to the following issues

Portability Almost needless to mention because of the heterogeneous nature of the Grid. In order to achieve this (combined with performance issues) we choose C for implementation and in particular we strictly stick to the ANSI standard (C++ supports object oriented programming but the resulting code is often platform and even compiler dependent).

Make use of current Grid resources Since many testbeds are being deployed nowadays ([34, 5]) the implementation should be able to make use of it.

Extendability Since the code which will be developed does not cover all possibilities and not even all algorithms discussed in this thesis, it should be written in a general way and provide hooks for possible future extensions.

Modularity For testing and evaluating it would be convenient to quickly replace the new routines with old ones or change the application running on top. In order to achieve this the code should be implemented within a modular framework.

8.4 Cactus Thorn Concept, Function Overloading

In Sections 2.4 and 2.5 we already introduced Cactus as a framework for finite difference codes. The modules in this code which can be plugged in and out are called thorns. Cactus differentiates between driver thorns and application thorns. Driver thorns take care of parallelization, output, visualization, streaming, monitoring etc. The Cactus "flesh" itself provides a core API for the application thorns which can be *overloaded* by any application thorn. It is thus possible to replace one driver thorn with another without changing the application as long as it provides the same API.

The "standard" thorn which provides and overloads basic functions for parallelization is called PUGH which provides function synchronization, processor topology calculations, load distribution and all other features connected with parallelization issues.

Following this concept we wrote a thorn, called PUGH_G[1] which overloads, provides or extends most parallelization functions formerly provided by PUGH.

8.5 Thorn PUGH_G

Our thorn has defined C structures that hold all information about parallelism: the global size of the grid, the local sizes of the grid (on each processor), the processor topology, the ghostzones, the domain decomposition etc. The application code does not have any knowledge about parallelism, except the fact that it is required to provide information when and which grid function has to be synchronized. This is done by a `CCTK_SyncGroup()` call that is overloaded by PUGH_G, along with other overloaded functions.

8.6 Implementation Issues

The implementation of the single machine topology from Section 6.2.1 is straightforward and gives a very fast code. For example, to calculate the optimal processor topology for a six dimensional grid it takes 0.3 seconds on an EV6/7 alpha processor.

For the multi-machine topology in Section 6.2.2 however we need to exploit higher level information services to get the information about the job- or machine-layout, namely how many processors are deployed on how many machines. Without this information we cannot proceed. It is the kind of information which is provided by Duroc [17].

Duroc works below the MPI level (which means that Mpich-G2 uses Duroc, if it is installed) which in principle means that it is active and can be used even before MPI_Init(). The task of Duroc is to submit multi-request jobs to the machines (using GRAM) and continuously check the status of those subjobs. Its API even offers the possibility to dynamically submit or kill subjobs out of a running code. This can be used for jobs to spawn off subtasks to or even completely migrate to other machines. Furthermore it offers send/receive operations for communication among processors and machines independent of the MPI communication. For our purposes it is enough to query this information at the time the parallel setup of a distributed computation is done. For this Duroc offers a library call named `globus_duroc_runtime_inter_subjob_structure()`. It returns an allocated array containing the number of machines the job is currently running on.

Unfortunately this routine cannot be called on every processor. Duroc defines ranks for the processors in each subjob starting from 0 to the number of processors in this subjob. Only one processor per subjob can call the above library function, namely the one with the Duroc subjob rank 0.

Together with the `globus_duroc_runtime_intra_subjob_size()` which gives us the number of processors on the calling subjob and Duroc provided send/receive calls we can distribute this information to every processor so that finally the job layout is known to every processor. By job layout we mean an array containing the number of processors per machine (or per subjob respectively).

[1] G stands for Grid

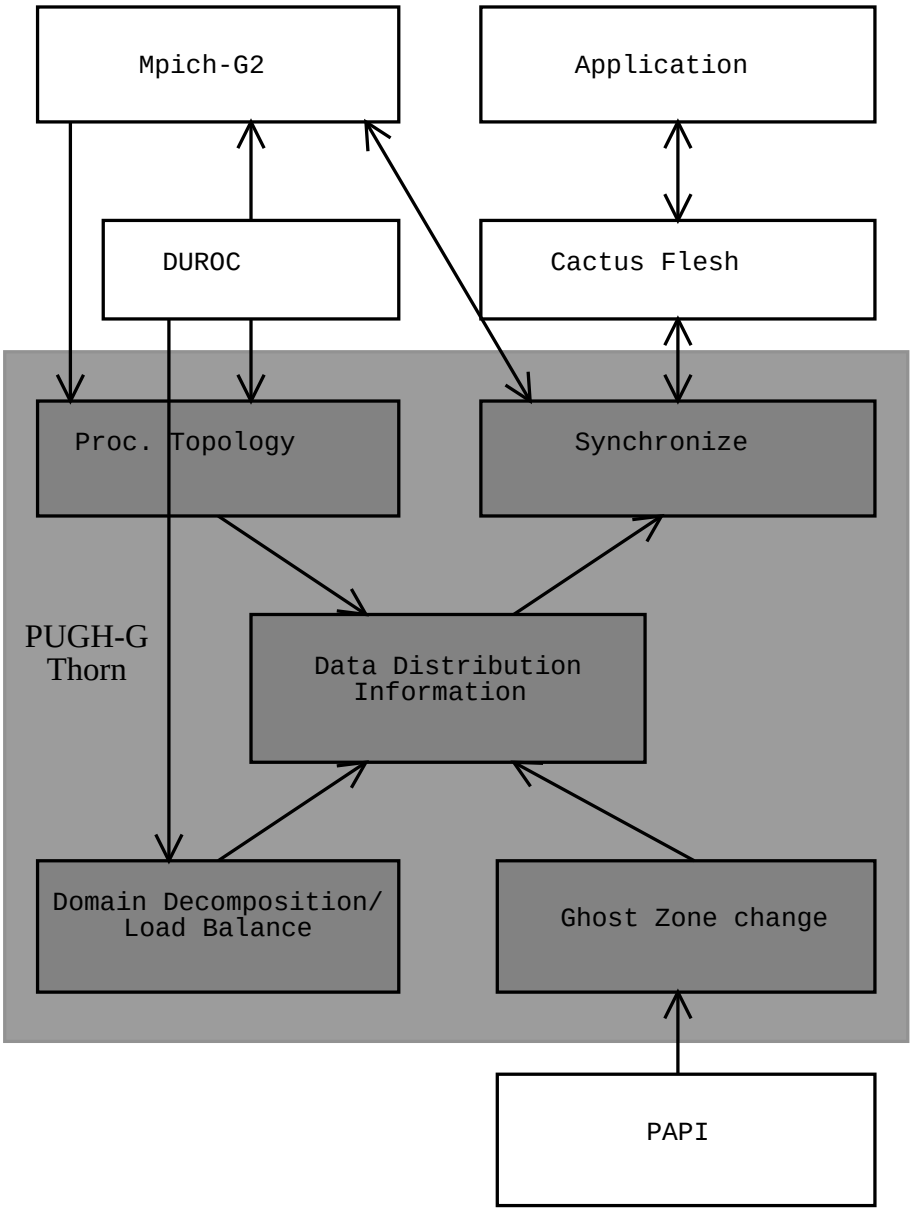

Figure 8.1: **The architecture of thorn PUGH_G** All information about parallelization is stored in a central structure. The information mainly depends on the actual processor topology that is calculated initially, the domain decomposition and the number of ghostzones. It is solely used by the synchronizing routine. Duroc provides basic information about the machine topology to Mpich-G2 and the topolgy routine. The routine for dynamically changing the ghostzone size uses PAPI, if installed. Arrows represent the flow of information.

We have also implemented a routine for static load balancing following the method and assumptions in Section 6.3. For this we measured the elapsed CPU (not wall clock) time for the following routine:

```
double x=3.1415926,y=3.1415926;
double z=0;
for (i=0.0 ; i < 50000000.0 ; i+= 1.0)
  { z+=x*y+(i/1000);}
```

One interesting pitfall here is that a loop like this as such cannot be used as a benchmark if the value of `z` is not used later on as "smart" compilers with high optimization flags turned on simply remove this loop and the measured CPU time (taken by the `getrusage()` call) is (almost) zero. We thus have to take the value of `z` as a return value or print it out to `/dev/null`.

To implement adaptive compression we followed the methods introduced in Section 7.1.5 and used the zlib library as in Section 5.6.4. As mentioned in that Section a numerical calculation deals often with a variety of datasets which can have very different compression properties. The Cactus framework coalesces datasets (functions) with similar properties into groups. Those groups can be for instance components of a vector or tensor. Our implementation of the adaptive compression algorithm monitors the performance change for the compression of every single group which makes the algorithm more sensitive.

The most tricky part of the implementation of the discussed methods in this thesis is given by the adaptive ghost zone algorithm. Simply changing the local subgrid size on each processor is by far not enough. The data has to be kept consistent. Within Cactus, all grid functions are bundled into groups and allocated in one chunk the procedure for changing the ghost zone size of one slab is implemented as follows:

- loop through all allocated grid functions
- store the data of the function in a buffer
- change the local subgrid size according to the new size of the ghost zones
- copy the data back from the buffer into the newly allocated array
- if the ghost zone size increased, fill the zone with data from the neighbored processor

While the copying of the old data into the new array is quite fast (i.e. it takes far less time than the time needed for one evolution iteration) the time consuming part is to fill the new ghost zones with real data from the neighbored processor if the new ghost size is bigger than the old one, see Figure 8.2. The procedure to refill the ghost zone does not have to be coded since it exactly corresponds to the procedure of synchronizing functions for which we already have routines. However, even functions that did not need communication (like coordinate functions or similar) have now to be synchronized since all allocated grid functions need proper data in the ghost zone area.

8.6.1 Performance Monitoring Issues

As already pointed out in Section 7.3 we have two basic choices to monitor the performance of a running code. An always possible choice is to measure the wall clock time of communication and

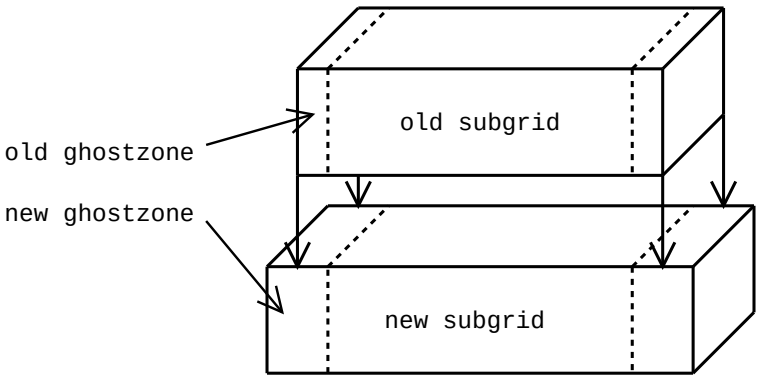

Figure 8.2: **Dynamical change of ghost zone sizes.** In the most general case the ghost zones on one processor can change in both directions. The saved values of the grid function have to be copied into the right place in the newly allocated array.

for this we use the `MPI_Wtime()` call, assuming that the code spends the rest of its time with useful computation.

Another much better choice is the usage of PAPI. It offers various performance monitoring functions ranging from the amount of cache misses, performed integer operations, elapsed rtime to the amount of performed floating point operations (flops) which we are interested in. We use the `PAPI_flops()` call to get the current floprate of the code (= number of floating point operations since the last call of this function divided by the elapsed wall clock time).

For both methods – wall clock time and floprate – we need to exchange the information among processors in order to keep consistency for the decision of the ghost zone size. Also, we want to minimize the additional communication associated with the exchange of this information. The information needs always to be consistent across a slab of processors as shown in Figure 7.3. To minimize the overhead we created new MPI communicators containing exactly one such slab (see. Figure 8.3). We did not average performance numbers across an entire subjob since one subjob, in the most general case, can be connected to two other subjobs involving two – maybe totally different – wide area network connections. Every time we want to average performance numbers we call an `MPI_Reduce()` on the communicator of the slab and thus keep this communication fast. Unfortunately those performance numbers also have to be averaged between two slabs on the same wide area connection. In order to also keep this communication fast we restrict this averaging to one processor on each slab. Note that at the time of performing this implementation Mpich-G2 coming with version 1.2.1 of the Mpich package did not support topology aware collective operations ("collops") on the MPI level. Later versions are to support this, among other features [39].

A last remark concerns a typical problem of a completely heterogeneous setup. We have written our code so that PAPI is being used if PAPI is installed on the system. A problem occurs when it is installed and can be used on *some* systems, not on all. This requires the implementation to perform additional steps in order to make sure to maintain consistency. We decided to use PAPI for the performance monitoring if at least *one* machine per wide area network has PAPI installed. In this case we only use those performance numbers.

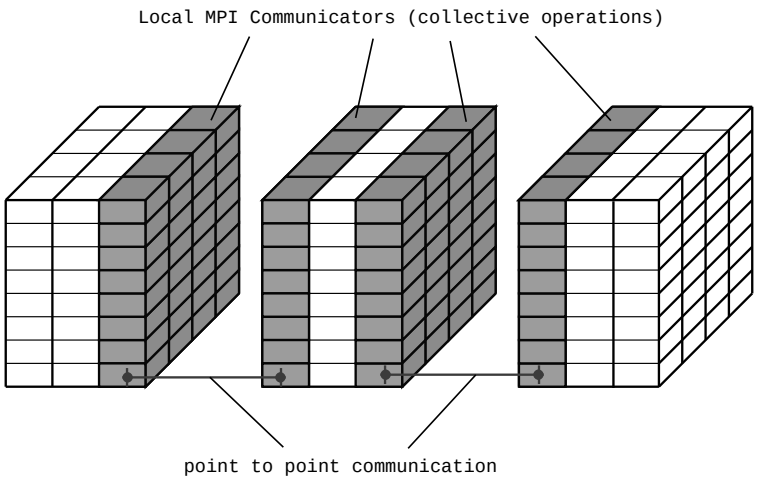

Figure 8.3: **Topology Aware Communication.** The exchange of information about performance only affects the pair of slabs communicating across the WAN with each other. In order to achieve a topology aware scheme for those communication we created MPI communicators for each slab (grey colored in this picture) and use MPI collective operations within it. The communication between the slabs is done through one pair of processors using MPI point to point operations.

8.6.2 Usage on Clusters

The design as well as the implementation of the ideas and algorithms of this thesis have been developed and written to specifically support efficient execution of parallel codes on multiple supercomputers. In conjunction with the Globus toolkit our code localizes communication bottlenecks (in terms of wide area network connections) and tries to apply techniques to deal with high latency and low bandwidth to those bottlenecks. This only works under the condition that there are only *some* of those bottlenecks.

Having said this it is obvious that this code cannot result in a very high performance benefit when executed on a slow connection cluster where we have communication bottlenecks all over the place. However, if the cluster has more processors on one node one can do the following.

Suppose the cluster has 2 CPUs per node. We can assume that the communication between those CPUs is fast, whereas the rest is slow (meaning at least one order of magnitude). Having N processors in a run we thus can setup a topology of $1 \times 2 \times N/2$ where the last dimension is the longest one. Then we can try to apply our adaptive techniques to every communication but the communication between processors on one node (see Figure 8.4). The downside of this method however is the fact that the overall topology is always the above one which does not necessarily minimize the overall communication. It may also happen that the total amount of communication across TCP/IP is bigger than the amount of shared memory communication which cannot be considered as beneficial.

8.7 Restrictions and Limitations

Although we have implemented our techniques in a very general way we made some basic assumptions about the Grid environment. In this section we review and discuss briefly the restrictions

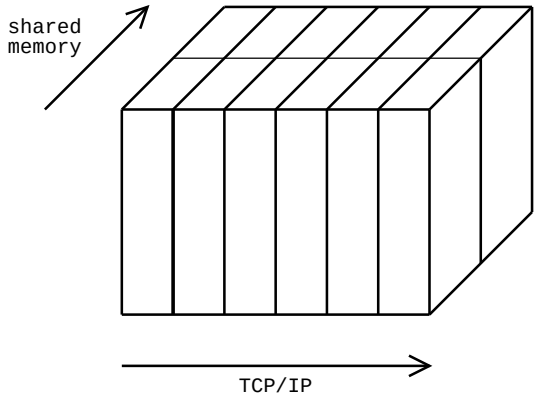

Figure 8.4: **Possible Topology on two Node SMPs.** When trying to apply the methods developed in this thesis to a cluster of two node SMP machines one could think of the above topology. Compression and latency hiding techniques then would be applied to all communications between the nodes (using TCP).

and limitations of our implementation. Most of it has been already hinted in previous Sections and Chapters.

- **Only Globus and Mpich-G2 supported.** Currently our software runs solely on top of these packages. However, we presently rely on the Globus toolkit primarily at one point, namely when we determine the job-layout. Mpich-G2 is only implicitly assumed in order to be able to arrange the machine order.

- **One type of processor per subjob.** Our implementation expects to have the same type of processors throughout the subjob as every CPU in the same slab is given the same load. Of course this could be refined in future but would also enormously complicate the general setup. Due to the fact that supercomputing systems are quite expensive they can be bought piece wise. This often leads to a heterogenous processor equipment in a single supercomputer. Vendors try to resolve this problem by shipping batch systems that can specifically request a processor type. In conjunction with our code this would lead to several subjobs within one supercomputer. The trade off which is made here is load balance against communication speed as the different processor types would communicate through TCP/IP.

- **Divisible number of processors per subjob.** We already extensively discussed that issue in Section 6.2.4. The processor number on subjob X should always be a multiple of the processor number on subjob Y (or vice versa). Since we mostly deal with processor numbers that are a power of two we do not feel that this is a major restriction.

- **Cache effects are not considered.** The single processor performance of a code is usually very cache sensitive. In our case this may lead to a strong dependency of the performance on the *shape* and *size* of the local subgrid on each processor (this was also found in [59]). Changing ghost zone sizes also changes the local subgrid and as such may effect the local performance. This in turn may lead to decisions that are not related to communication issues.

- **No adaptive meshes.** At the current stage our code does not support adaptive mesh refinement. While not impossible, it may significantly increase the complexity of this problem and as such

we leave it for future research.

- **Intra machine communications are considered neglectable.** We implicitly used this assumption throughout our studies. Our code has been designed to deal with *a few* slow communication channels within one distributed run. This is done under the assumption that all other communication channels are neglectably fast. It is thus not convenient to use our implementation on a loosely coupled set of workstations (see previous Section).

- **Networks characteristics change only slow.** Another implicit assumption which is being made is that the network characteristics do not change during two intervals our code is monitoring the performance. If this is not the case, wrong decisions are taken by our code. The monitoring interval however, can be manually tuned in our implementation [as a parameter file entry].

Chapter 9

Results

After having implemented various ideas of previous chapters we now will investigate in detail what effect our techniques have on performance improvement.

9.1 General Remarks

In this section we will comment on the way we tested our code. Originally, we designed it for large scale distributed runs as demonstrated in Chapter 5. However, we did not have 4 dedicated supercomputers at hand for our test runs. At the current stage there is no way to smoothly co-schedule runs across two sites. We strongly believe that this will be possible in the future, but for now it is a major drawback to test our code for large scale calculations.

9.1.1 Firewalls

Many sites begin to build firewalls around their HPC facilities. Without having at least one open TCP port window, distributed runs are impossible. Many firewall sites do have this window, they allow any connection within a certain range, e.g. between port numbers 10000-10100. The Globus toolkit and also Mpich-G2 allow to specify this allowed TCP port range via an environment variable. However, if all involved sites have different allowed port ranges, things become difficult to manage, although distributed runs work in principle.

9.1.2 Software Versions

Having two independent executables in place that are to form a unity in a distributed run can be problematic. They are built from software on various levels (e.g. operating system, Globus, Mpich, Cactus framework, thorns). Some software can be installed in user space, some not (e.g. operating system include files and libraries, Globus gatekeeper services etc.). Since the Globus software undergoes a rapid development it happens that different sites have different versions of Globus installations making it difficult to support distributed computing. There are two possible solutions: Installing parts of the Globus software in user space to ensure a consistent software tree or – if we are dealing with the identical platforms – compiling and linking statically on one platform and shifting the executable to the other.

9.1.3 Reference Testbeds

From the restrictions described above it follows that to test our code for real distributed runs we have to restrict ourselves to the cases where we have working executables and our code starts up on all machines immediately. By this we are in most cases restricted to two machines and a small number of processors (meaning about 8 processors or less per machine).

In order to test our implementation in different setups, we chose the following pair of machines to execute our testruns on:

9.1.4 Setup A

`modi4.ncsa.uiuc.edu` ⟷ `origin.aei.mpg.de` This two-machine-testbed has the following properties. The bandwidth between these is about 350kb/s and the latency is about 60ms. They are supercomputers having the same operating system but are equipped with different processors. While the `origin.aei.mpg.de` has 32 R10000 processors running at 195MHz, the other site is equipped with processors running at 250MHz. Note that `modi4.ncsa.uiuc.edu` is only the front machine to a whole cluster of Origin 2000 supercomputers. At the NCSA site we submit our jobs to the LSF queue system (using the Globus jobmanager-lsf) whereas on the other site we don't use a batchsystem but rather submit using the so-called fork-gatekeeper (or Globus jobmanager-fork).

Both systems have a vendor's supplied MPI version installed and Mpich-G2 is configured to use this native MPI for any MPI data transmission within one machine. Both sites have PAPI installations in place, so that some of our decision routines are based on PAPI hardware counters (i.e. the current floprate). We were able to start up to 8+8 processors for a distributed run.

9.1.5 Setup B

`tg32-0.ncsa.uiuc.edu` ⟷ `fs0.das2.cs.vu.nl` This testbed exhibits different properties (as compared to the testbed above). First, these machines are not supercomputers but (linux) clusters having the Portable Batch System installed (PBS). They both have a front node (listed above) with a Globus jobmanager-pbs running on it. Jobs are submitted to this front node and distributed to the compute notes by PBS. As opposed to the Origin systems there are no vendor's supplied MPI installations which means that the communication among the cluster nodes is using TCP/IP (compare Section 3.2.3 on page 23).

The cluster interconnect consists of a Fast Ethernet network. Another difference between this and our first testbed is the fact that there are no hardware counters that could be used by the PAPI interface in order to get runtime performance data. Our decision routines are in this case solely based on pure wall clock time. The cluster on the NCSA site had 4 nodes which we had at our disposal. We could start distributed runs up to 4+8 processors across the sites.

9.1.6 Reproducability and Bandwidth

Since it is the purpose of this chapter to study the performance improvement of various techniques we discussed in this thesis, we performed a series of runs with different configurations in order to see the possible impact of various techniques. However, since the runs are not done all at the same time, but rather in a time range of days or even weeks, one cannot expect that the same run always shows the same performance. This is, as we have shown in previous chapters, not only depending on the actual bandwidth (competing traffic, bandwidth per stream etc.) but also on the issue "how dedicated" the system is.

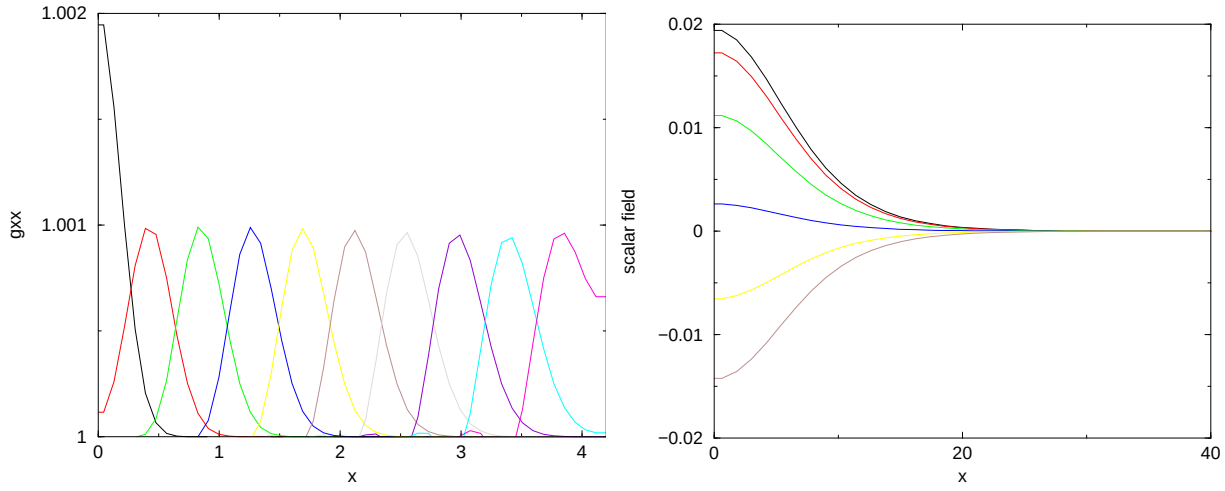

Figure 9.1: **Field dynamics of two applications.** For testing our code we use two different applications, having different computation and communication patterns as well as different dynamics, type of datasets and other characteristics (like memory demand, time for one iteration etc). Both applicaiions however are finite differencing codes solving differential equations on a regular mesh.

However, we measured the bandwidth between sites using regular ftp transfers. While we did not specifically look at the actual value of the bandwidth we tried to exclude bandwidth effects on comparison runs by measuring the bandwidth immediately after and before the run. We did not see a significant bandwidth change during our comparison runs.

9.1.7 Applications

As we stated earlier, our code can be applied to any application running within the Cactus framework. We picked two different applications with different computation and communication patterns. One application calculates the time evolution of a gravitational field, the other calculates the time evolution of a massive scalar field. Figure 9.1 shows how the dynamics of those fields look. Both codes solve a system of partial differential equations, but are using totally different algorithms. The scalar field code for example synchronizes 12 grid functions per iteration, the gravitational field code 27. We executed the gravitational field application on setup A and the scalar field code on setup B.

9.2 Topology

Here we discuss the influence of the processor topology on the performance of distributed runs.

9.2.1 Setup A

8+8 processor run: Let's first discuss the NCSA-AEI run. A very often used shape of the computational grid is the so-called "bitant" mode: The x-dimension is twice as big as the other dimensions. As a result our code creates a topology that produces only half of the communication

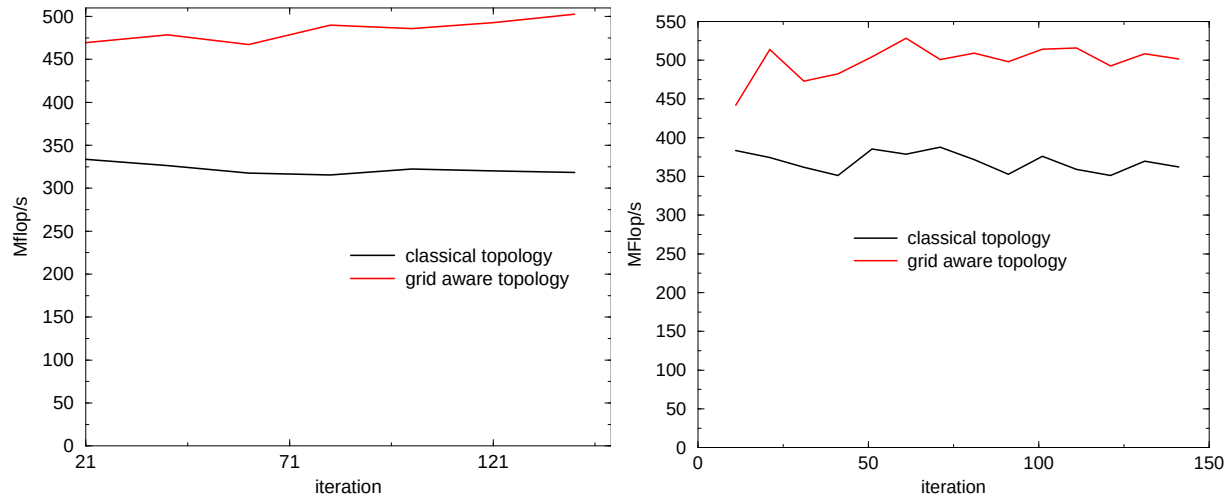

Figure 9.2: **The influence of an adequate processor topology on the performance**. While the upper curve shows the floprate of a run with a topology created by our algorithm, the lower curve shows the same run using a classical non Grid aware topology scheme. The right diagram was a 8+8 run while the left diagram shows a 6+12 processor run. Both runs were executed across two Origin 2000 systems, calculating the evolution of a black hole. Figure 9.3 explains why we gain a performance improvement here: The total amount of communicated data is twice as big.

during the run as compared to a non-Grid aware algorithm (see right diagram in Figure 9.2). The performance improvement we gained by this is about 50% in terms of floprate.

12+6 processor run: We also performed a 12+6 run between those two sites. While our code generates a $3 \times 3 \times 2$ topology, the classical code generates a $2 \times 3 \times 3$ topology. Our code produces plane machine boundaries, while the machine boundaries created by the classical code are irregular and as such generate much more traffic, see Figure 9.3. The performance improvement in this case is documented in the left diagram of Figure 9.2.

9.2.2 Setup B

For the testruns between the GridLab and TeraGrid clusters we did not have PAPI at hand[1]. Our code therefore used wall clock time as a decision base and we use the total execution time as the appropriate metric to measure the benefit of our techniques. Since the total execution time is just measured as one number per run, we performed several (usually three) runs and compared the average time of our different setups. Moreover, we calculated the standard error of the mean value to verify that the differences are significant; The execution time dropped by 22%, see Figure 9.4.

As for the other application it means that our Grid aware topology reduces the amount of data transfers between machines.

There might be *special cases*, however, that lead classical topology algorithms to the same result as our Grid aware algorithm (GAPTA). But even if we deal with a cubical grid and have the same number of processors at each site we do not need to end up with a satisfying topology. There are

[1]PAPI requires special patches to be applied to the linux kernel

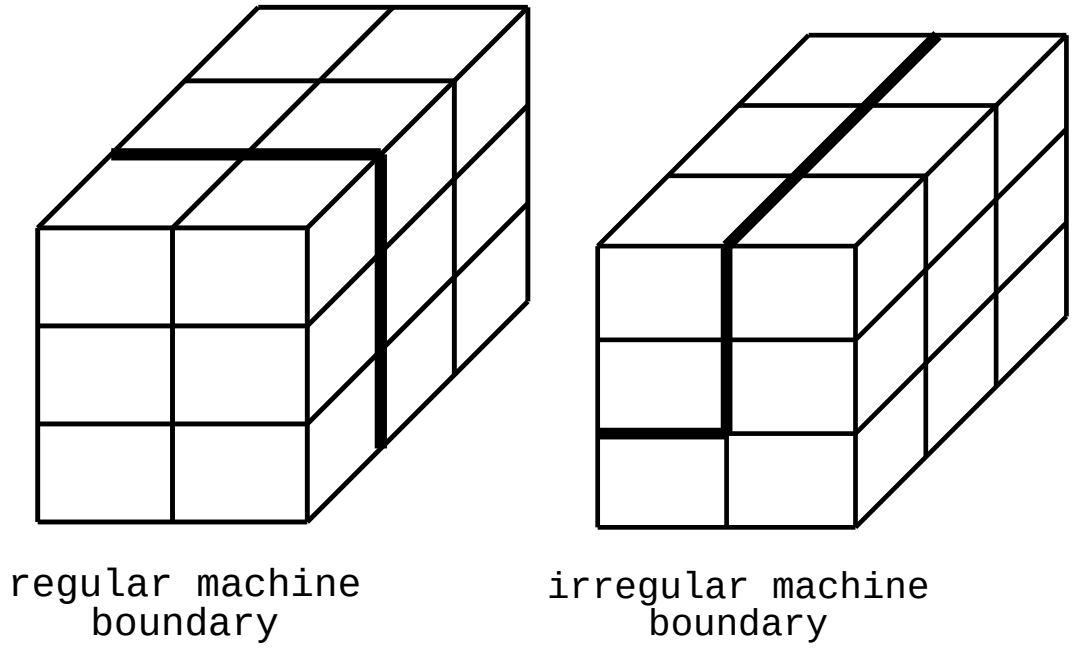

Figure 9.3: These figures represent the topologies for a 12+6 processor run between two machines. The thick line marks the machine boundaries. Our Grid aware algorithm (GAPTA) creates regular and plane machine boundaries as seen in the left figure. The topology in the right figure, as calculated by the classical algorithm, yields an irregular machine boundary which generates about twice as much traffic. The amount of exchanged data is proportional to the area that is surrounded by the thick line in this figure. Taking the same load on each processor into account, it is evident that this area is twice as large in the right case that is produced by the traditional algorithm.

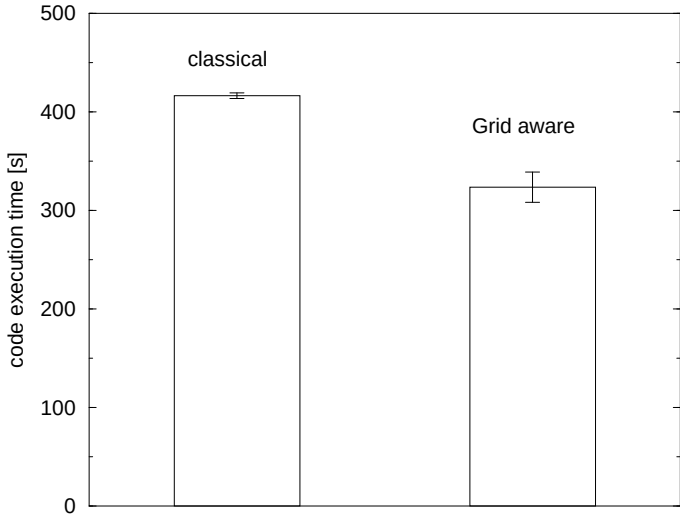

Figure 9.4: **Effect of an adequate topology on the total execution time of the distributed code.** This figure shows the execution time of the scalar field application with and without our Grid aware topology algorithm. The reduction of the execution time is more than 20%. The error intervals of the mean value of our runs (see diagram) do not overlap which means that we can consider this difference as significant.

additional constraints on the number of processors (and the dimensions) so that these cases can be regarded as exceptional. For example a code with a cubical grid executed on two machines with the same number of processors would have the same processor topology with both algorithms. Doubling the number of processor on one machine or adding another machine or doubling the length of one dimension of the grid (mesh) leads immediately to significant differences.

9.3 Loadbalancing

Our statical loadbalancing algorithm, as discussed in Section 6.3 (on page 67) measures the processor speed on the local machine and compares it with the speeds of the other machines. According to those numbers it performs a statical rebalance of the workload before the actual calculations begin.

In both our testbeds we deal with machines that are equipped with processors and other hardware that runs at a different speed. However, in setup B we deal with processors that run at the same speed so that our techniques do not give us any improvement. We focus on setup A.

The Origin 2000 in Potsdam has processors on board that run at 195 MHz while the ones at NCSA run at 250MHz. Our algorithm identified the heterogeneity of the setup:

```
INFO (PUGH_G): Running distributed across 2 machines.
INFO (PUGH_G): Processor distribution across machines is: 8 + 8
INFO (PUGH_G): testing floating point speed on local processor ....
INFO (PUGH_G): floating-point routine took: 8.72 seconds
INFO (PUGH_G): PAPI: floating-point routine made 28.7 MFlop/s
INFO (PUGH_G): subjob 0 needed 8.72 seconds
INFO (PUGH_G): subjob 1 needed 6.80 seconds
INFO (PUGH_G): grid-point distribution per subjob in the longest dimension: 88  112
INFO (PUGH_G): grid-point distribution in the longest dimension: 52  35  44  69
```

The workload between the two machines is distributed according the local processor speed. Recall that in our design and from the experiences we made in Chapter 5 we decided to give processors at the machine boundary less work than other processors. In this case our algorihmt yields the above shown distribution: `52 35 44 69`. Recall that traditional algorithms would give us `50 50 50 50`.

The rebalance of the load thus can give us an additional performance benefit. Figure 9.5 shows that this is really true. Even if the performance of the single runs is quite noisy, the performance gain by rebalancing is significant as can be seen in this figure.

For the scalar field application and the cluster testbed we do not expect any performance benefit, since the two clusters have processors on board that do run at the same speed. Our initial load balancing algorithm divides up the load to equal parts between the machines as it would be done by an algorithm that is not aware of different processor speeds.

9.4 Compression

The benefit of compression strongly depends on the type (smoothness, periodicity) of data that is exchanged between sites. In order to illustrate this we show the influence of data compression on the performance for different datasets. In our first application, we calculated the evolution of

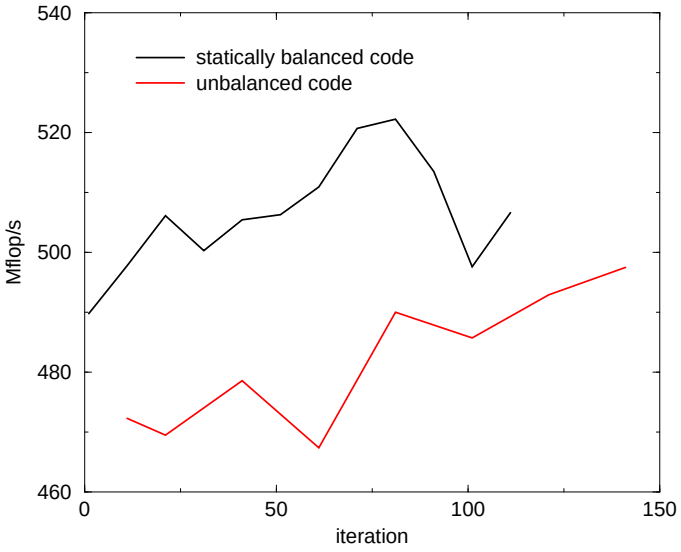

Figure 9.5: The influence of a proper loadbalancing on the floprate of a running code between two systems that run at different speed even if the speed difference is "only" 195MHz as compared to 250MHz. The performance is rather noisy in both cases but the difference is significant. Our test loop took 8.72 seconds on one machine and 6.80 seconds on the other yielding a grid point distribution of 88 to 122 in the longest dimension (instead of 100 to 100).

plane gravitational waves, this dataset compresses very well, i.e. the performance increases almost by a factor two. However, the compression ratio depends on many things (where are the machine boundaries, what is the resolution, the grid size etc.) and may even change in time. We want to show the effect of compression in two other cases, namely in the case of a black hole (where compression does not seem to have any effect, see Figure 9.6) and the evolution of a massive scalar field, where compression has a moderate effect (see Figure 9.7). The scalar field code synchronizes four functions in total. In two cases the compression did not change the communication time, in one case it became worse, and in one case it improved. Thus, the alorithmt decided to maintain compression for one function out of four. By this, the communication time was reduced by 25%.

In the case of our scalar field application, compression led to a moderate reduction of the total execution time by about 30%.

9.5 Ghostzones

According to our two different setups, the decision about the number of ghostzones is based on two different metrics. For setup B (the Origin-Origin setup using PAPI) the current floprate is the basis for decision, for the second one it is the pure communication time.

9.5.1 Scalar Field Simulation

Our code consistently chose a ghostzone size of 4 in three runs. As for compression and topology issues, we can claim that this, again, reduced the computation time by a significant amount, see Figure 9.8.

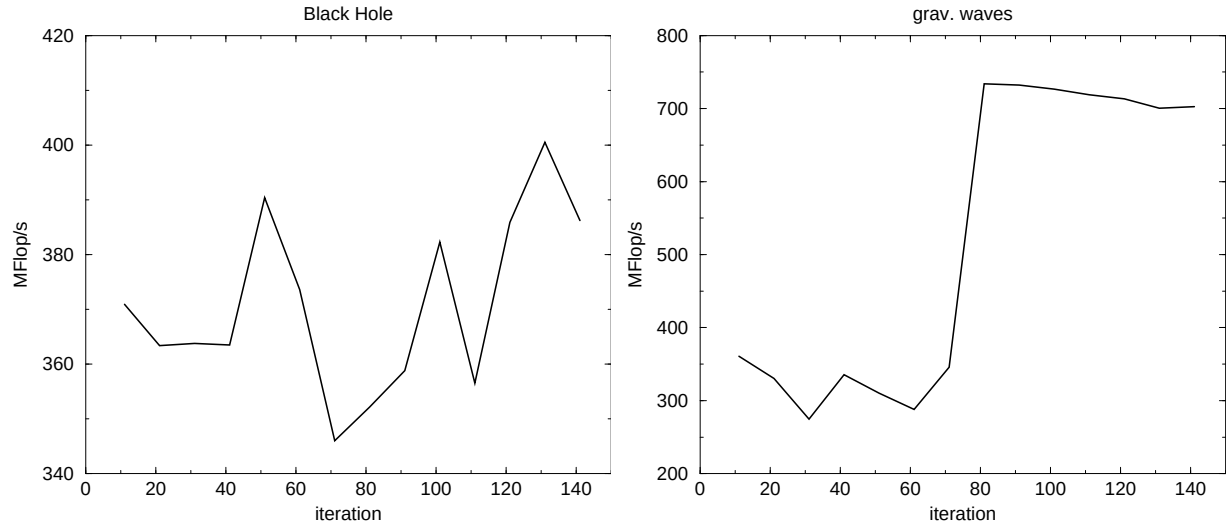

Figure 9.6: **Effect of compression on a 8+8 black hole (left) and 8+8 gravitational wave run**. In both figures we see the performance of the numerical algorithm that calculates the evolution of gravitational fields, but uses different initial datasets. The code evolves up to iteration 70, then compression is turned on. In the case of gravitational waves (right) it improves the performance almost by a factor 2. In the left case (black holes) compression led to a reduction of communication time by about 12% (not seen here). We see a slight performance improvement after interation 70. However, this benefit seems to get lost in the noise so we can't claim that this improvement is significant.

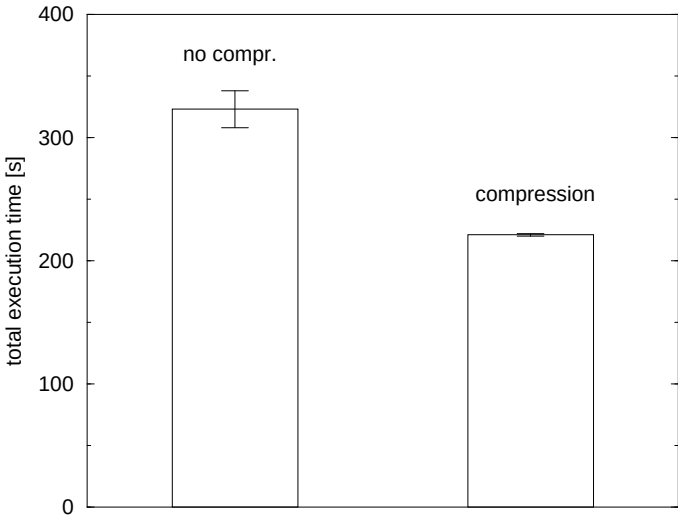

Figure 9.7: **Comparison of compression vs. no compression** for the scalar field application running across two clusters (8+12 processors). The bandwidth was about 480kbyte/s. Switching on compression prior to data transfer leads to a reduction of the execution time. Even since compression led to a reduction of communication time for two out of four datasets, the overall effect on the execution time is a reduction by 35%.

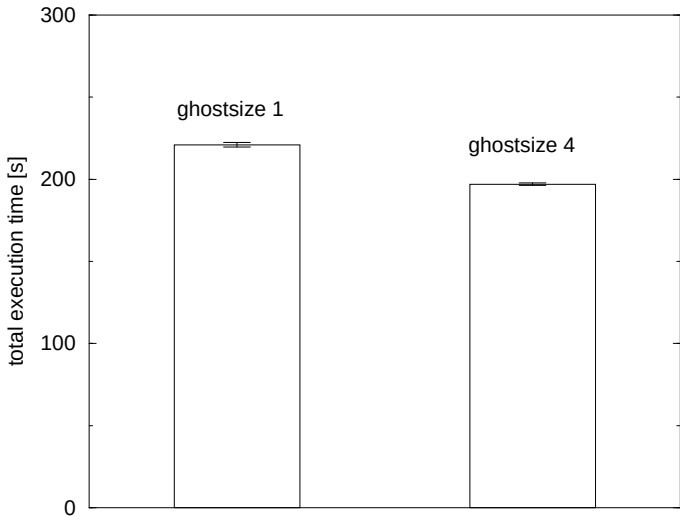

Figure 9.8: The total execution time of the scalar field code in setup B with different number of ghostzones. Our algorithm consistently chose a ghostsize of 4 for this particular setup. The total execution time reduced by 10%. The chosen ghostzone proved to be consistent in this series of testruns.

9.5.2 Gravitational Field Code

For the black hole application the decision base is the current floprate, since we are able to use performance counters on the systems we are running on. As disscussed earlier, our algorithm increases successively the ghost zone size at the machine boundaries until no significant performance improvement (in terms of floprate) is achieved. However, we cannot claim that the choice of an adequate ghost zone size follows a consistent pattern. Figure 9.9 for example shows two identical runs with different datasets, performed at different times. In the left diagram, where we see a big effect of compression, our algorithm chose a ghostsize of 12, which is too big. One can ask why the ghost size keeps increasing when the overall performance goes slightly down between iteration 100 and 200. The answer is that the decision base for the choice of ghost zone sizes is the performance measured solely at processors at the machine boundary, not the overall performance (this is needed in cases where there are more machines involved and as such we could have different sizes of ghostzones between machines).

The right image in Figure 9.9 demonstrates what may happen in a real Grid environment: Around iteration 150 the performance significantly drops by more than 100 MFlop/s. We don't know the reason for this. It may be the result of a sudden drop of bandwidth (somebody started an ftp transfer) or a temporary overload of the machine, swapping of the operating system or other things.

9.6 Reaching Single Machine Performance

We now compare the total execution time or floprate respectively to the execution time or floprate that we obtain at single machine (or single cluster) execution. Figure 9.10 compares the performance improvements of all techniques we have applied. The total execution time is about half as long as for the "standard" run. The performance of the run with all methods applied is very close to the

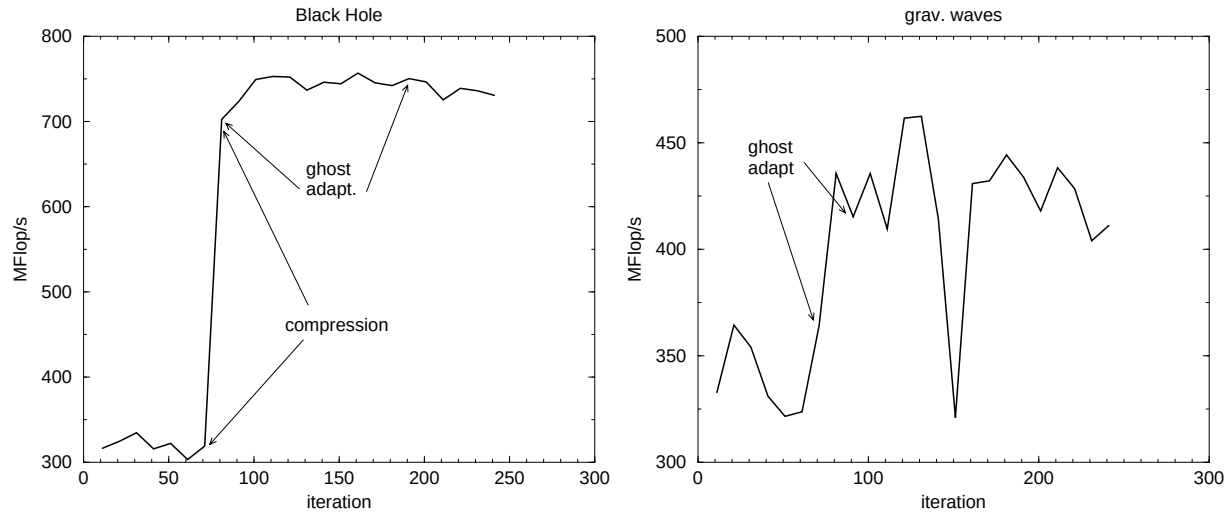

Figure 9.9: Two examples of the same application code in the same setup, but with different initial datasets. We see a black hole (right) and a plane gravitational wave evolution code (left) across the 2-Origin-setup (8+8 processors). After switching on compression, the ghostzone size increases to 2 in the right case and to 12 (!) in the left case.

single cluster performance using 12 processors. Compared to the single cluster execution (set to 100% efficiency) we can quote efficiencies of 40% for the standard run, 52% for a run including the new topology, 76% including compression and 86% including all techniques.

We also performed another run between origin.aei.mpg.de and harpo.wustl.edu. The application code was the same as for the other Origin-Origin runs, we used 8+8 processors in total. Figure 9.11 compares the performance with the "standard" performance (achieved by the standard code) and the performance from single machine execution. Due to the heterogeneous setup (i.e. the two machines ran at different speed) we performed single machine runs on both machines and took the average performance of both in order to be able to compare against the performance of the distributed run.

9.6.1 Single Machine Tests

To test the logic of our implementation it could be sufficient to run a test code in an environment which simulates a Grid environment in the following sense. Recall that `MPICH-G2` uses vendor's MPI routines within one subjob and TCP/IP for the communication between subjobs. A simple test environment would therefore comprise two subjobs running on the same machine where our code meets the "real" Grid situation of mixed slow and fast communication since the communication between subjobs goes across (in this case) a 100Mbit Fast Ethernet interface using TCP/IP.

Our test here comprises thus a 16 processor run on an Origin 2000 machine, using two subjobs on the same machine, each with 8 processors.

Figure 9.12 compares runs using the old and new algorithms. One can see that both runs start at about the same floprate, around 1100 Mflops/s, as seen in Figure 9.12. While the performance of the code running with conventional static parallelizing algorithms stays more or less constant (within some noise) our adaption algorithm first tries to switch on data compression between the two

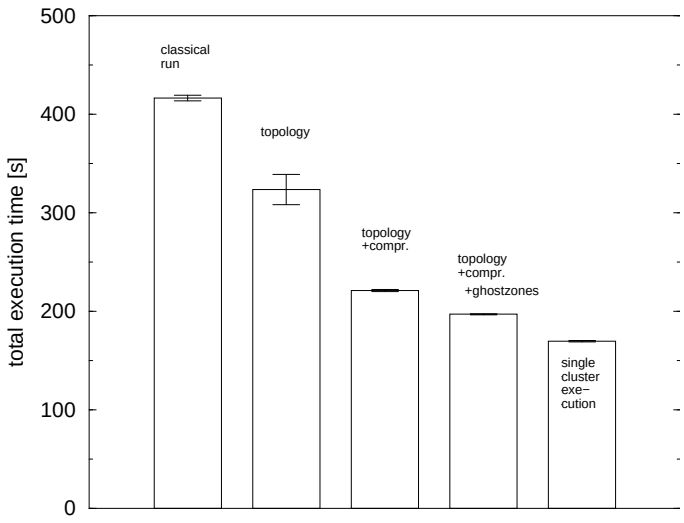

Figure 9.10: In this diagram we compare the performance improvements we achieve with our techniques with single machine or single cluster execution. For the first four bars we executed the scalar field code distributed across two clusters in the US and the Netherlands with 4+8=12 processors. For the single cluster run we executed that code on the latter cluster using 12 processors. Assuming a 100% efficiency for the single cluster run, the distributed run using traditional parallelizing algorithms runs at 40% efficiency whereas the code using our algorithm reaches 86% efficiency.

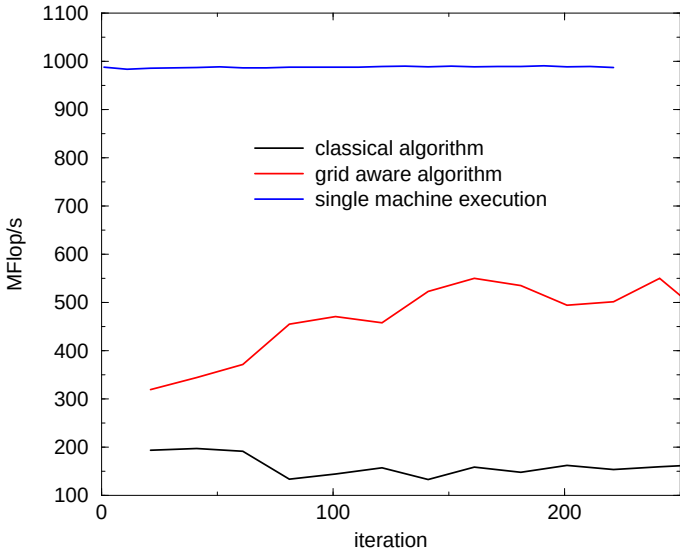

Figure 9.11: This figure compares another two transatlantic runs, this time between origin.aei.mpg.de and harpo.wustl.edu. Due to a proper loadbalancing, the code starts up at a performance rate that is already 50% better than the standard run. Applying compression and adaptive ghost zones (up to iteration 120) again improves the performance up to 550 MFlop/s. The performance of the single machine execution is also shown.

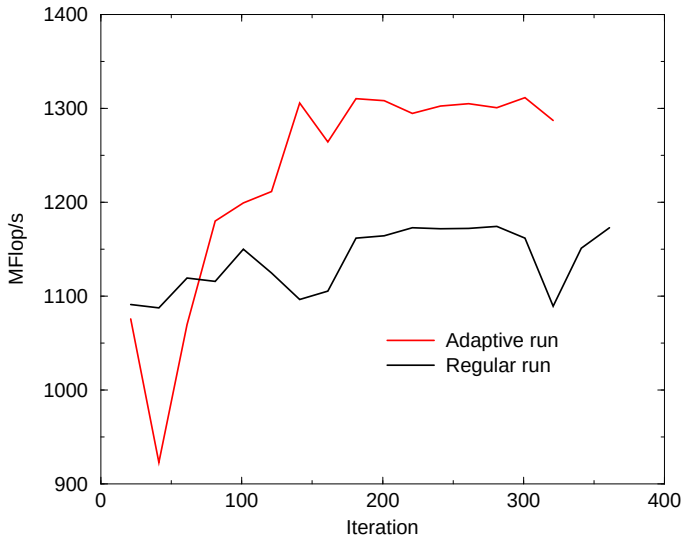

Figure 9.12: Single processor run with two subjobs: The upper curve shows a run using the new algorithms and the lower curve shows a standard run. Although in this case compression does not provide a performance benefit, increasing the number of ghost zones results in a performance improvement by 20%. In this run, the initial ghost zone size was one, and the final ghost zone size was three. Since latency is not a factor here (we have a latency of about 1ms for connections across the local Fast Ethernet interface), we suspect cache effects to account for this.

subjobs. TCP/IP performance tests across the local interface showed us that more than 40MB/s is possible, which is quite fast (i.e. faster than the critical bandwidth, compare Figure 7.1), and so we see (at around iteration 45 in Figure 9.12) a significant performance loss since the additional CPU time needed for compressing and decompressing dominates over the benefit of smaller data packages. The decision routines of our driver code recognize this degradation and switch off data compression and the performance of the code returns to the original value. Then our algorithm starts to increase the ghost zone size between the two subjobs. It is surprising that this leads to a significant increase of performance since the latency in this case is around 1ms. This speedup could be also influenced by cache effects since extra ghostzones change the local sub grid size.

9.6.2 Large Scale Distributed Runs

The problem of the runs of the last sections was that we couldn't test our codes and algorithms in an environment similar to the one in Chapter 5.

The results presented in this section use a similar setup to that used in Section 5, running between Origin 2000 machines at NCSA in Illinois, USA and a SP2 machine at SDSC in California, USA. For those previous DTF runs we used a manually tuned code, with optimizations specially applied at the application level, which we described in detail in Chapter 5.

The results of these test runs can be seen in Figure 9.13. The left figure shows the efficiency change in time of a 256 processor run between NCSA and SDSC. To create this graph we assumed that the Cactus code solely performs all communications within our driver thorn (we did not execute comparison runs with the standard code). The right figure shows a 16 processor run namely the same run we performed in the last two subsections (but across different machines). Comparing this

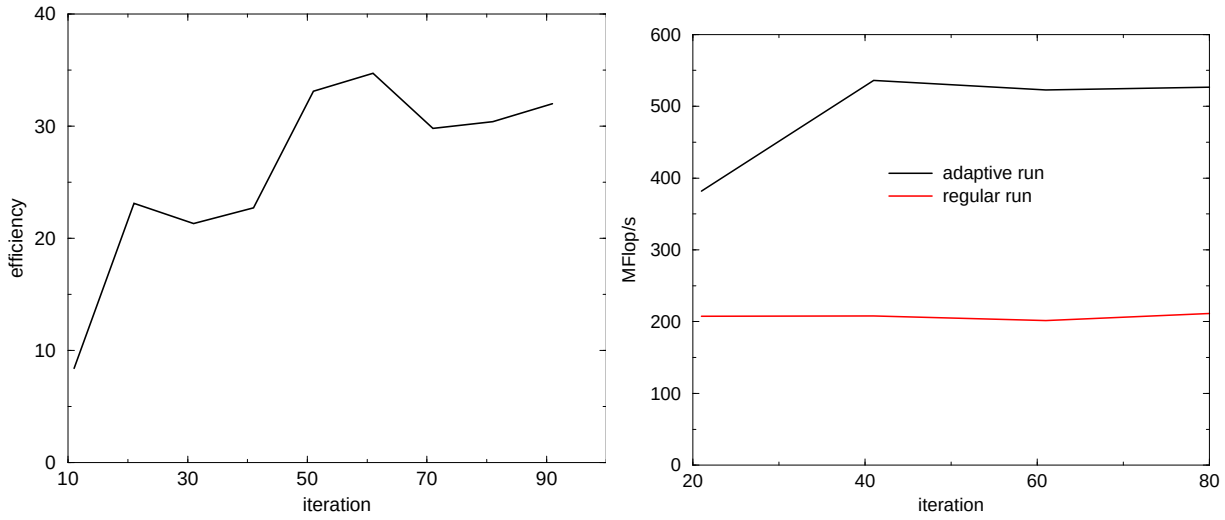

Figure 9.13: The efficiency of a larger scaled distributed run across 256 processors on two different sites. The picture is very similar to our previous example of a transatlantic run as seen in the right diagram which shows the Floprate of a smaller run (16 CPUs) between SDSC and NCSA with and without all algorithms we implemented. Due to a proper loadbalancing and better topology the adaptive code starts with a much better performance at the beginning (around 400 MFlop/s as compared to around 200 MFlop/s).

Figure with Figure 9.11 we see that the initial performance difference of the old and new code is bigger here. The reason for this is that the Power-3 processor (used on the SDSC side) is about twice as fast as the R10000 processor and thus a proper load balancing has a much bigger effect.

9.6.3 Cluster Runs

We also implemented a test code of an algorithm we designed in Section 8.6.2 (on page 101) which supports the execution on a collection of workstations with two CPUs as discussed in Section 8.6.2. As mentioned in Section 3.2.1 the Mpich package supports a mixed shared memory and TCP communication when properly configured (i.e. having compiled in both devices shmem and p4). However, at the time we conducted these experiments the current Mpich version 1.2.3 has some (known) problems with this mixture which makes it unuseable for our test runs.

The LAM package [63] can also be configured to support shmem as well as TCP communication within one execution[2].

Our test runs were done using a collection of dual-xeon workstations with a 100Mbit/s full duplex (switched) network. Test runs showed that we can increase the efficiency from about 50% to about 75% mainly due to compression. The ghost zone adaption algorithm, surprisingly, always yielded a size of 1.

[2]Note it has to be configured with `--with-rpi=sysv`

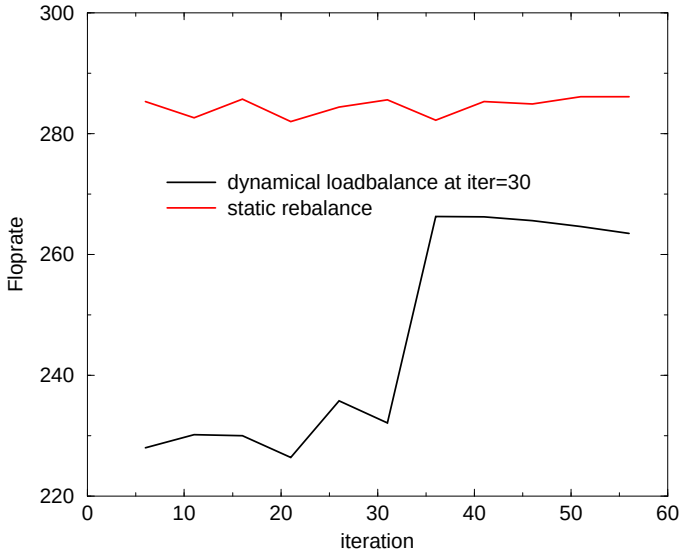

Figure 9.14: These two curves show a run with (a) statical loadbalance (upper curve) and (b) dynamical rebalance at iteration=30 (lower curve). As discussed in the text, the intensity of the dynamical rebalance depends on a parameter. If the value of this parameter is not optimal, dynamical loadbalance achieves worse results than static loadbalance.

9.7 Experiments with Dynamical Loadbalancing

Implementing a fully dynamical loadbalance code means mainly two things: First, it means to implement a code that calculates the new loadbalance according to Section 7.5 and secondly one needs a code that actually shifts data accordingly from processor to processor. For this thesis, we only implemented the first part. For applications that calculate e.g. N-body problems this might be easy, for finite differencing codes the implementaiton is rather difficult. Nevertheless, it can be implemented in a way that does not affect the application.

Note that our statical loadbalancing algorithm does *not* need to shift workload from processor to processor since at the stage the balancing is done no numerical data has been calculated yet.

Our major motivation for dynamical loadbalance is to see if our statical loadbalancing is appropriate enough, since it is based on testloops, not on the performance of the actual application. Also, we introduced a slight disbalance between processors at the machine boundary and others to overlap communication and computation (see Section 6.5). We expected this to happen automatically in our dynamical loadbalancing strategy from Section 7.5.

From our experiments we now see that in real runs with production applications the following problem can occur. If a code runs below 50% efficiency, our algorithm can lead to negative loads, especially for the processors at the machine boundary: Since our algorithm tries to compensate the waiting time for a processor by lesser workload negative workloads occur if the waiting time is longer than its computation time.

One way to work around this is to introduce a parameter that steers the intensity of the rebalance. This could be for example a parameter p that reduces the local processor speed, that we use in our calculations to rebalance. For example if we want our algorithm to compensate only

half of the waiting time by computation time we would set the parameter to $p = 1/2$.

The question whether the dynamical loadbalance verifies our assumptions that we made for statical loadbalancing strongly depends on this parameter. The value of this parameter that leads to similar results as statical loadbalancing however depends on other things, like global grid size, total number of processors, shape of the grid and maybe other factors. Up to now, we don't know how to estimate or even calculate the "optimal" value for this parameter. If this value is not optimal, static loadbalancing might even be better than dynamic, as shown in Figure 9.14.

In that run we used a computational grid that is stretched in some (e.g. the second) dimension, having 200 grid points and ran it across a homogeneous setup (using 4+4 processors) but with slow connections between the machines (350kb/s bandwidth, 60ms latency). After collecting runtime data for about 30 iterations, our dynamical load balancing algorithm chose to distribute the gridpoints along the second dimension as follows: `32 27 23 19 20 22 26 31` (Note that an unbalanced code would look like `25 25 25 25 25 25 25 25`). We see that this is already very close to the distribution we simulated in the table on page 92. However, this distribution strongly depends on the choice of the parameter described above.

Overhead Introduced by Dynamical Adaptions Up to now we did not investigate how big the introduced overhead of all adaptation techniques is. Recall that all performance numbers of the last sections did *not* take the communication overhead into account that is produced by our code. Since Cactus has build-in timers for every scheduled routine (measuring wall clock time as well as CPU time) we can easily read off the produced overhead by reading off the elapsed wall clock time of our scheduled routine.

In the cluster tests from the last Section for example it did not take more than 1.6 seconds which is less than 1% of the total execution time. Also for the other runs we observed an overhead of the same order of magnitude (around 1% of the total run time).

For the transatlantic runs we have a similar relation. In 200 iterations the time needed for pure adaption was 48 seconds out of a total time of 2157 seconds which corresponds to 2%. The adaption finished at iteration 120. Assuming that the evolution goes over several thousand iterations (like in real production runs) and the process repeats every 500 iterations the average overhead of the adaption becomes neglectably small.

9.8 Discussion

In the previous Sections we discussed examples of test runs where our code significantly improved the performance. However, our code can not guarantee a performance improvement in any case.

Multi machine processor topology algorithm We can prove that our algorithm for creating a multi-machine topology (see Section 6.2.2) always minimizes the total amount of data which is being transferred over the wide are network. We cannot guarantee that it also minimizes the transfer time. Why could this happen? In Chapter 4 we saw that multiple streams may increase the total bandwidth between machines. Our algorithm however, after minimizing the inter machine data transfer, minimizes the amount of intra machine data exchange instead of *maximizing* the number of TCP/IP streams between machines. There is a priori no reason why this could not be implemented, except for the fact that it depends on the distributed MPI implementation whether multiple streams are used; Mpich-G used a forwarding node (i.e. only one stream) while Mpich-G2 does not. We leave it for future projects.

Static load balancing The weakness of static load balancing has already been discussed in our Section about dynamical loadbalancing. Initial test loops may not be representative for the application and the overlap of computation and communication cannot be optimal with a fixed value.

Compression Our adaptive algorithm only changes the compression state if and only if our counters report a performance improvement and as such can, in principle, never impair the performance. However, the monitored performance metric may have improved due to a sudden bandwidth increase. Whether this effect can be avoided by continuously monitoring the bandwidth remains questionable. However, since we periodically apply our techniques a "wrong" decision can be reverted in the next adaption process.

Multiple ghostzones The same holds true multiple ghostzones, although we believe that the optimal number of ghost zones mainly depends on latency issues, that do not change that quick. However, since our overall performance is our decision base for the number of ghostzones, the decision made here can be also "wrong".

Dynamical loadbalancing As we have seen, there are certain limits for dynamical loadbalancing, especially when one wants to achieve an overlap between computation and communication. The intensity of rebalancing can be steered by a parameter, that has to be set to an appropriate value from case to case. We believe that an appropriate value leads to a significant performance improvement due to a balanced code, but the value has to be found automatically, since it can't be the user's responsability. However, at this stage, we don't know of any way to do that.

Cluster Runs We have shown that even on a loosely coupled set of workstations one can achieve a performance improvement when one has e.g. two CPUs on one node using fast communication (like shared memory) among them. However, our implementation is not general enough to support this kind of topology on an arbitrary set of two (or more) processor PCs. The problem is to *identify* this configuration at startup time. Recall that for the major class of our runs we use the Globus toolkit to identify the machine topology which is not possible in this case. Either we install Globus on every single node (which is not practical) or configure a smart batch system on top of the cluster to work with Globus (and LAM or Mpich) in a way that each subjob is exactly located on one node if this is possible at all.

9.9 Summary and Conclusion

We have seen that distributed runs executed in real Grid environments face a variety of issues that lead to low performance of the executed code. The design and implementation of our algorithms attacked many of those issues. We could show that for many cases the performance of distributed codes could be much improved, sometimes even close to single machine performance.

It is hard to predict which of our discussed technique has what effect on a running code. Our autonomic routines however apply techniques in a way that the performance cannot be impaired: If a technique caused a performance drop, the decision is reverted.

We believe that in a real Grid environment whose properties change in an unpredictive manner this autonomic way of adapting a running code is the best method to achieve optimal performance.

Chapter 10

Related Work

In this chapter we will sketch the work of other authors which is related to our work. Related means that those studies also deal with the execution of codes in a distributed Grid environment.

As already outlined in the introduction there are many research projects that are dealing with the development of software for Grid computing on all levels.

However, we don't know of any effort to try to optimize the execution at the level *between* communication layer and application. We will introduce two projects that focus on optimizations on the application level and the message passing layer.

10.1 The Albatross Project

The albatross project was initiated by dutch computing centers to study the execution of parallel codes on wide area network clusters. The testbed comprised four clusters distributed across the Netherlands where the number of nodes ranged from 24 to 128. The clusters were connected by a (dedicated) 6Mbit/s line [9].

The goal of this project was to develop programming models to support efficient execution of codes on this testbed.

Apart from aplication level optimizations, a special MPI library MagPIe [41] that is based on Mpich and supports effective collective operations within such a distributed cluster environment was developed.

The MagPIe library focussed on optimized collective MPI operations, like `MPI_Bcast, MPI_Reduce, MPI_Gather` etc.

As mentioned in the introduction, existing parallel applications have been optimized for wide area cluster execution [10]. The authors could show that by *manual tuning* many parallel applications can improve their performance by far.

This coincides with our experiences in Chapter 5. However, our manual tunings in Chapter 5 did not effect the numerical result. The researchers in the Albatross project simply dropped some messages that were supposed to be sent over the WAN (we also discussed this in Section 6.6.2). Moreover, those changes were done *on the application level* for *specific* applications.

Our approach in this thesis is different. We apply changes for efficient execution in general Grid environments to *a whole class* of applications and *without* changing the application itself.

The researchers in the Albatross project applied Grid-awareness to the MPI and to the application level. In this thesis we took another way: We made the code *between* the MPI and the application level Grid aware.

Making an MPI implementation Grid aware has the advantage that it can be used by a huge class of codes. The major downside, however, is that the performance benefit will be rather low since too little is known about the communication patterns that are finally used.

Making a specific application Grid-aware has the advantage of achieving a high performance since the communication patterns are well known. Unfortunately this has to be done for any application seperately.

In this thesis we tried to keep the balance between those two extremes. By making all code *between* those two levels Grid aware which achieve a fairly good performance of a whole class of parallel codes.

10.2 Metacomputing at MRCCS and HLRS

As already described in Section 1.3.3, the Manchester Research Center for Computational Science (MRCCS) and the HochLeistungsRechenzentrum Stuttgart (HLRS) worked together on an efficient execution of parallel codes in a Grid environment. On the message passing level, a special MPI implementation, PACX was developed, which we introduced in Section 3.2.4.

Also, manual optimizations were done at some sample applications codes from computational fluid dynamics and data processing, including latency hiding techniques and computation/communication overlap. As for the Albatross project, those optimizations were done manually and specifically depending on the application code.

Chapter 11

Conclustions and Directions for Future Research

In order to recognize the rapid development in the field of Grid computing let us compare the situation in 1998 (at the Supercomputing Conference) with the possibilities we have today.

11.1 From Supercomputing 98 to now

At SC 98 we saw that Metacomputing is possible but hard to realize and inefficient in performance. It took many months and person hours to prepare a single run, including machine setup, network setup, manual adaption of software components on all levels and other things. The resulting distributed run was done over a highly homogeneous setup (solely T3Es) and did not use more than 32+32+32 processors with an extremely low efficiency.

In April/May 2000 we could improve this situation by far. In only three weeks of work, involving only a few (< 3) persons it was possible to run a benchmark optimized application code *efficiently* across 1500 processors and a heterogeneous setup.

At SC 2001 we were able to run an *non-optimized production code* across 256 processors in a heterogeneous setup that automatically adapts itself to the environment and dynamically increases its efficiency by a factor 3-4. There was (except for compiling, parameter file preparation etc.) almost *no preparation time*.

11.2 Contribution of this Thesis

The rapid development described in the last section was possible due to many improvements of software packages on all levels. The main representatives of those software packages on those different levels are Globus, Mpich-G2 and Cactus. This thesis mainly analyzed and implemented "Grid-awareness" of software between the MPI and the application level, as extensively discussed in Chapters 6 and 7. The major results and contributions of this thesis can be summarized as follows.

Large scale Grid computing is possible The last major attempt (before writing this thesis) to run a tightly coupled scientific code across multiple supercomputers was done at SC 1998 (see Section 1.3.2). Recall that the total number of processors did not exceed 96 and in most attempts was only 64. In Section 5 we demonstrated for the first time that there is *virtually*

no limit to the number of processors and machines and even to the heterogeneity of the setup. To our knowledge this was the biggest distributed execution of a parallel code ever done.

Large scale Grid computing can be made efficient We also demonstrated that by manual tuning (load balance, compression etc.) such a code can be executed with a decent performance.

Optimal performance can be automatically achieved In Section 8 we implemented a code that applies many performance optimizing techniques (like load balancing, compression, optimal processor topologies etc.) according to information about network and machines.

No user interaction is required for efficient execution Our implementation applies many techniques that are completely hidden to the end user. We get all information that is needed to apply our techniques from services like Globus or from runtime measurements of processor speed, network speed etc.

Besides this, we also consequently *use* and *test* software that has been developed for Grid computing in real application scenarios. Due to our consequent usage of available Grid software we also gave useful feedback (especially in Chapter 5) for the development of those packages and also to the development of hardware infrastructures like the Teragrid.

11.3 Future Directions and Scenarios

Although we have taken an important step in this thesis, 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.

11.3.1 Algorithms and Implementations

We discussed and analyzed far more techniques and algorithms than we implemented in the end. Some of those are not worth to be implemented for several (previously discussed) reasons.

The general principle should be to move from static algorithms to dynamic ones. The most urgent issue seems to be the implementation of our dynamical load balancing algorithm. Also, the design and implementation of an algorithm that maximizes the number of streams between machines will surely give a further performance improvement.

11.3.2 Networks and Testbeds

Besides reliable software components, fast networks and functional testbeds are required for efficient Grid computing. Since the importance of Grid computing has been recognized by federal institutions in Europe and the United States, millions of Dollars and Euros have been invested to build testbeds and testbed infrastructures.

In the United States the TeraGrid is currently about to be installed. Although still not fully functional it will provide a total computational power of more than 10 Tflop/s distributed across major supercomputing sites in the country. The network back bone connecting most sites will be

a 40Gbit/s link. Recall that this bandwidth is already **higher** than the memory bandwidth of a today's common memory chip (e.g. DRDRAM, having a peak bandwidth of about 10GBit/s).

However, in times where the regular bandwidth of wide area networks catches up to the memory bandwidth this picture will dramatically change. Scenarios that today appear wild and crazy will then belong to the daily work of numerical and computational scientists.

It is the goal of todays fundamental research to be prepared for those days. It is also one major goal of this thesis that could only cover a minor part of this entire enterprise.

Another important project which helps us to explore the capability of Grid computing and prepare things for the future is the GridLab [34] project. It is the biggest Grid project in Europe funded by the European Commission. Various supercomuting centers spread across entire Europe (from Hungary to Wales) are building a testbed and software components that will fully meet all conditions of production Grid computing in the near future.

The GridLab testbed at the current stage already provides fairly advanced features like

- Uniform access to a variety of resources. Every single user can access any resource just using his/her X500 certificate. The "access" comprises `ssh`, `ftp`, and a uniform access to batch systems.

- Centralized accounts. The user only needs to apply for an account at one central institution. The distribution of the account and account information happens automatically.

- Dynamically updated testbed status information. All testbed machines continuously report their status and configuration (including hardware, queues and batch system status) to one or more `ldap` servers. Those servers can be queried by any program or application.

11.3.3 Users and Portals

The above described testbed infrastructure offers all information needed by applications and lower level software components to be fully functional in a dynamic Grid environment.

However, the end user does not want to care about the details of a Grid environment at all. The requirements for a user that wants to run his/her applications in a real Grid environment should be minimal. More precisely, running a Grid application should not be more complicated than running a parallel application on a single parallel system. This is still not the case by far. Up to now, most of the following issues are still a manual process: Finding resources, co-allocate them, get the software on it, configure with the appropriate (machine specific) flags, compile, write scripts for submission etc.

Even if software components for all the above problems are currently under development (in the GridLab project and elsewhere) the user does not want to get familiar to a whole bunch of software packages.

We thus need software that provides all benefits of Grid enviroments to the end user but – in principle – fully takes over and automatize the above described issues.

Users need a *single access point* to log in to that uses all those other components but hides unrelevant details from the user. The development of such a *Portal* is a major effort within the GridLab project [60].

We now see that we are quite close to the scenario we have described in Section 1.2. We also see that the realisation of such a scenario cannot be done by a single person or group but only by the whole computational science *and* user community in a global and collaborative way where

finally many software building blocks will fit consistently together to form the GRID. In this thesis we have studied and implemented one of those building blocks.

Bibliography

[1] A. Abrahams and R. Price. Black-hole collisions from brill-lindquist initial data: Predictions of perturbation theory. *Phys. Rev. D*, 53:1972–1976, 1996.

[2] A. A. Abramovici, W. Althouse, R. P. Drever, Y. Gursel, S. Kawamura, F. Raab, D. Shoemaker, L. Sievers, R. Spero, K. S. Thorne, R. Vogt, R. Weiss, S. Whitcomb, and M. Zuker. LIGO: The laser interferometer gravitational-wave observatory. *Science*, 256:325–333, 1992.

[3] M. Alcubierre, G. Allen, B. Brügmann, T. Dramlitsch, J.A. Font, P. Papadopoulos, E. Seidel, N. Stergioulas, W.-M. Suen, R. Takahashi, and M. Tobias. Towards a stable numerical evolution of strongly gravitating systems: The conformal treatments. 1999. gr-qc/0003071.

[4] Miguel Alcubierre, Bernd Brügmann, Peter Diener, Michael Koppitz, Denis Pollney, Edward Seidel, and Ryoji Takahashi. Gauge conditions for long-term numerical black hole evolutions without excision. 2002. gr-qc/0206072.

[5] G. Allen, G. Aloisio, Z. Balaton, M. Cafaro, T. Dramlitsch, J. Gehring, T. Goodale, P. Kacsuk, A. Keller, H.P. Kersken, T. Kielmann, H. Knipp, G. Lanfermann, B. Ludwiczak, L. Matyska, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, A. Reinefeld, M. Ruda, F. Schintke, E. Seidel, A. Streit, F. Szalai, K. Verstoep, and W. Ziegler. Early experiences with the EGrid testbed. In *IEEE/ACM International Symposium on Cluster Computing and the Grid CCGrid2001*, pages 130–127, 2001. URL citeseer.nj.nec.com/379615.html.

[6] 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. URL http://www.Cactuscode.org.

[7] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel, and Brian Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. 2001. URL http://www.cactuscode.org/Papers/GordonBell_2001.ps.gz.

[8] ASCI White home page. URL http://www.llnl.gov/asci/.

[9] H. Bal, A. Plaat, T. Kielmann, J. Maassen, R. van Nieuwpoort, and R. Veldema. Parallel computing on wide-area clusters: the albatross project, 1999. URL citeseer.nj.nec.com/bal99parallel.html.

[10] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, Rutger, and F. H. Hofman. Optimizing parallel applications for wide-area clusters. Technical Report IR-430, Vrije Universiteit, Amsterdam, 1998.

[11] J. Basney, M. Livny, and T.Tannenbaum. High throughput computing with Condor. *HPCU News*, 1(2), June 1997.

[12] Werner Benger, Ian Foster, Jason Novotny, Edward Seidel, John Shalf, Warren Smith, and Paul Walker. Numerical Relativity in a Distributed Environment. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[13] Blue Horizon News. URL http://www.npaci.edu/online/v5.1/colony.html.

[14] G. Chiola and G. Ciaccio. Porting MPICH ADI on GAMMA with Flow Control. In *in proceedings of MWPP'99 (1999 Midwest Workshop on Parallel Processing)*, Kent, Ohio, August 11-13 1999. URL ftp://ftp.disi.unige.it/pub/project/GAMMA/mwpp99.ps.gz.

[15] G. Ciaccio, M. Ehlert, and B. Schnor. Exploiting Gigabit Ethernet Capacity for Cluster Applications. In *To appear in the Proceedings of the Workshop on High-Speed Local Networks (HSLN) 2002 within the 27th IEEE Conference on Local Computer Networks (LCN)*, 2002.

[16] J. Crank and P. Nicholson. A practical method for numerical evaluation solutions of partial differential equations of the heat conduction type. In *Proceedings of the Cambridge Phil. Society*, number 43, pages 50–67, 1947.

[17] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–??, 1998. URL citeseer.nj.nec.com/czajkowski97resource.html.

[18] L. Deutsch. Zlib compressed data format specification version 3, 1996. URL http://www.gzip.org/zlib.

[19] V. Donaldson, F. Berman, and R. Paturi. Program speedup in a heterogeneous computing network, 1994.

[20] DTF:. Location of the full npaci dtf proposal. URL http://www.ncsa.uiuc.edu/News/Access/Releases/PublicTeraGrid.pdf.

[21] Carsten Ernemann, Volker Hamscher, Ramin Yahyapour, and Achim Streit. On effects of machine configurations on parallel job scheduling in computational grids. URL citeseer.nj.nec.com/528018.html.

[22] H. Feider. Gridmake. grosser beleg, universität potsdam. URL http://www.cs.uni-potsdam.de/~schnor/potsdam/Research/Grid/grid_make.html.

[23] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. URL http://www.mcs.anl.gov/dbpp/.

[24] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high-performance distributed computing experiment. pages 562–571. IEEE Computer Society Press, 1996. URL citeseer.nj.nec.com/foster96software.html.

[25] I. Foster and N. Karonis. A grid-enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of SC'98*. ACM Press, 1998. URL citeseer.nj.nec.com/foster98gridenabled.html.

[26] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2): 115–128, Summer 1997. URL ftp://ftp.globus.org/pub/globus/papers/globus.ps.gz.

[27] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. MORGAN-KAUFMANN, 1998.

[28] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Service Architecture for distributed systems integration, June 2002. URL http://www.globus.org/ogsa/.

[29] I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994. URL ftp.globus.org/pub/globus/papers/nexus_paper_ps.pdf.

[30] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Journal of Cluster Computing*, 5:237–246, 2002.

[31] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogeneous computing environment. In *PVM/MPI*, pages 180–187, 1998. URL citeseer.nj.nec.com/gabriel98distributed.html.

[32] Gamma Project home page. URL http://www.disi.unige.it/project/gamma/mpigamma/.

[33] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996. URL unlser1.unl.csi.cuny.edu/~archives/postscripts/parallel/pvm/pvmvsmpi.ps.

[34] GridLab Home Page. URL http://www.gridlab.org.

[35] A. Grimshaw and W. Wulf. The Legion vision of a world-wide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[36] W. Gropp and E. Lusk. Reproducable measurements of MPI performance characteristics. In *Proceedings of the PVMMPI Meeting*, 1999. URL http://www-unix.mcs.anl.gov/~gropp/papers.html.

[37] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. In *Preprint MCS-P567-0296*, Argonne National Labs, 1996. URL ftp://info.mcs.anl.gov/pub/tech_reports/reports/P567.ps.Z.

[38] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, New York, 1998.

[39] N. Karonis, B. de Supinski, I. Foster, W. Gropp, W. Lusk, and S. Lacour. A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids. 2002.

[40] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. In *To appear*, 2002. URL http://www.globus.org/mpi.

[41] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. Mpi's reduction operations in clustered wide area systems, 1999. URL citeseer.nj.nec.com/kielmann99mpis.html.

[42] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic migration: A new tool for dynamic grid computing. In *Submitted to Proceedings of Tenth IEEE International Symposium on High Performance Distributed Computing, HPDC-10, San Francisco*, 2001. URL http://www.cactuscode.org/Papers/HPDC10_2001_Worm.ps.gz.

[43] Gerd Lanfermann. *Nomadic Migration – A Service Environment for Autonomic Computing on the Grid*. PhD thesis, University of Potsdam, November appears 2003.

[44] H. Langendörfer and B. Schnor. *Verteilte Systeme*. Hanser, München, 1994.

[45] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis, using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, August 2001. URL http://icl.cs.utk.edu/projects/papi.

[46] LSF 5.0 Info page. URL http://www.platform.com/products/wm/LSF/index.asp.

[47] T. Ludwig, M. Lindermeier, A. Stamatakis, and G. Rackl. Tool Environments in CORBA-based Medical High Performance Computing. In *Proceedings of the 6th International Conference on Parallel Computing Technologies (PaCT-2001)*, 2001.

[48] MAUI home page. URL http://mauischeduler.sourceforge.net.

[49] C. W. Misner, K. S. Thorne, and J. A. Wheeler. *Gravitation*. W. H. Freeman, San Francisco, 1973.

[50] Mpich home page. URL http://www-unix.mcs.anl.gov/mpi/mpich.

[51] Origin 2000 Technical Data. URL http://www.sgi.com/origin/2000/datasheet.html.

[52] S. Pakin and A. Pant. Vmi 2.0: A dynamically reconfigurable messaging layer for availability, usability, and management, 2002. URL http://vmi.ncsa.uiuc.edu/Publications/san1.pdf.

[53] Manish Parashar and James C. Browne. On partitioning dynamic adaptive grid hierarchies. In *HICSS (1)*, pages 604–613, 1996. URL citeseer.nj.nec.com/parashar96partitioning.html.

[54] Stephen Pickles, Fumie Costen, John Brooke, Edgar Gabriel, Matthias Müller, Michael Resch, and Stephen Ord. The problems and the solutions of the metacomputing experiment in sc99. In *HPCN Europe*, pages 22–31, 2000. URL http://www.csar.man.ac.uk/staff/brooke/./hpcn2000.ps.gz.

[55] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, England, 1986.

[56] M. Prieto, I. Llorente, and F. Tirado. A review of regular domain partitioning, 2000.

[57] Resource Specification Language. URL http://www.globus.org/gram/rsl.html.

[58] Matei Ripeanu. Issues of running large scientific applications in a distributed environment. Master's thesis, University of Chicago, 2000.

[59] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Performance predictions for a numerical relativity package in Grid environments. 15(4):375–387, November 2001. ISSN 1094-3420.

[60] M. Russell, G. Allen, G. Daues, I. Foster, T. Goodale, E. Seidel, J. Novotny, J. Shalf a nd W.M. Suen, and G. Von Laszewski. The astrophysics simulation collaboratory: A science portal enabling community software development. In *Submitted to Proceedings of Tenth IEEE International Symposium on High Performance Distributed Computing, HPDC-10, San Francisco*, 2001. URL http://www.cactuscode.org/Papers/HPDC10_2001_Portal.ps.gz.

[61] Ben Segal. Grid computing: The european data project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, October 2002. URL http://web.datagrid.cnr.it/pls/portal30/docs/1442.DOC.

[62] Seti@home page. URL http://setiathome.ssl.berkeley.edu/index.html.

[63] Jeffrey M. Squyres, Andrew Lumsdaine, William L. George, John G. Hagedorn, and Judith E. Devaney. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings, MPIDC'2000*, March 2000. URL http://www.lam-mpi.org/.

[64] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990. URL www.hensa.ac.uk/parallel/environments/pvm3/emory-vss/pvmsystem.ps.Z.

[65] Supercomputing 95 home page. URL http://www.supercomp.org/sc95/web/SC95/.

[66] Tera Grid home page. URL http://www.teragrid.org.

[67] The Department of Energy Science Grid. URL http://doesciencegrid.org/.

[68] The Grid Physics Network. URL http://www.griphyn.org.

[69] The MPI standard. URL http://www.mpi-forum.org/.

[70] The Sun Grid Engine. URL http://wwws.sun.com/software/gridware/.

[71] B. Willke, P. Aufmuth, C. Aulbert, S. Babak, R. Balasubramanian, B.W. Barr, S. Berukoff, S. Bose, G. Cagnoli, M. Casey, D. Churches, D. Clubley, C.N. Colacino, D. Crooks, C. Cutler, D. Danzmann, R. Davis, E. Elliffe, C. Fallnich, A. Freise, S. Gossler, A. Grant, H. Grote, G. Heinzel, A. Hepstonstall, M. Heurs, J. Hough, K. Kawabe, K. Kötter, V. Leonhardt, H. Lück, M. Malec, P. McNamara, S. McIntosh, K. Mossavi, S. Mohanty, S. Mukherjee, S. Nagano, G.P. Newton, B.J. Owen, D. Palmer, M.A. Papa, M.V. Plissi, V. Quetschke, D.I. Robertson, N.A. Robertson, S. Rowan, A. Rüdiger, B.S. Sathyaprakash, R. Schilling, B.F. Schutz, R. Senior, A.M. Sintes, K.D. Skeldon, P. Sneddon, F. Stief, K.A. Strain, I. Taylor, C.I. Torrie, A. Vecchio, H. Ward, U. Weiland, H. Welling, P. Williams, W. Winkler, and I. Zawischa. The geo 600 gravitational wave detector. *Class. Quantum Grav.*, (19):1377–1387, April 2002.