Software Reusability and Efficiency

A SCIENTIFIC AND TECHNOLOGICAL STUDY

undertaken by Parallab, Bergen Center for Computational Science,
University of Bergen (Norway) for the ENACTS network

SECTORAL REPORT

FINAL VERSION - APRIL 2004

http://www.enacts.org

Contents

1 The ENACTS Project: Background and Introduction

1.1 Presentation of the ENACTS project

ENACTS is a Co-operation Network in the 'Improving Human Potential Access to Research Infrastructures' Programme. This Infrastructure Co-operation Network brings together High Performance Computing (HPC) Large Scale Facilities (LSF) and key user groups. The aim is to evaluate future trends in the way that computational science will be performed and the pan-European implications. As part of the Network's remit, it runs a Round Table to monitor and advise the operation of the LSFs.

This co-operation network brings together many of the key players from around Europe who offer a rich diversity of High Performance Computing (HPC) systems and services. The strategy involves close co-operation at a pan-European level - to review service provision and distil best-practice, to monitor users' changing requirements for value-added services, and to track technological advances. In HPC the key developments are in the area of Grid computing and are driven by large US programmes. We urgently need to evaluate the status and likely impacts of these technologies in Europe whose goal is to set up European Grid computing. This Grid is a 'virtual infrastructure' in which each researcher, regardless of nationality or geographical location, has access to the best resources and can conduct collaborative research with top quality scientific and technological support.

ENACTS provides participants with a co-operative structure within which to review the impact of Grid computing technologies, enabling them to formulate a strategy for increasing the quantity and quality of access provided.

1.1.1 Scope and membership

The scope of the ENACTS network is computational science: the HPC infrastructures which enable it and the researchers, primarily in the area of physical science, who use it. Three of the participants (EPCC, CINECA and CESCA-CEPBA) are LSFs that provided Transnational Access in HPC under the HCM and TMR programmes, and more recently under the I3 HPC-EUROPA project [33]. Under the HCM and TMR programme, they were joined by UiB/Parallab and the associated Bergen Computational Physics Laboratory (BCPL). The LSFs have provided access to hundreds of European researchers in a very wide range of disciplines and are thus well placed to understand the needs of academic and industrial researchers. The other ten ENACTS members are drawn from a range of European organisations with the aim of including representation from interested user groups and also by centres in economically less favoured regions. Their input ensures that the network's strategy is guided by user needs and relevant to smaller start-up centres and larger more established facilities.

Centre	Role	Skills/Interests
EPCC	LSF	Particle physics, materials sciences
ICCC Ltd	User	Optimisation techniques, control engineering
UNI-C	LSF	Statistical computing, bio-informatics, multimedia
CSC	User	Physics, meteorology, chemistry, bio-sciences
ENS-L	Society	Computational condensed matter physics, chemistry
FORTH	User	Computer science, computational physics, chemistry
TCD	User	Particle physics, pharmaceuticals
CINECA	LSF	Meteorology, VR
CSCISM	User	Molecular sciences
UiB	LSF	Computational physics, geophysics, bio-informatics
PSNC	User	Computer science, networking
UPC	LSF	Meteorology, computer science
NSC	User	Meteorology, CFD, engineering
ETH-Zürich	LSF	Computer science, physics

Table 1: *ENACTS participants by role and skills.*

1.1.2 Workplan

The principal objective of the ENACTS project is to enable the formation of a pan-European HPC metacentre. Achieving this will require both capital investment and a careful study of the software and support implications for users and HPC centres. The latter is the core objective of this study. The project is organised in two phases. A set of six studies of key enabling technologies has been undertaken during the first phase:

- Grid service requirements (EPCC, PSNC)
- the roadmap for HPC (NSC, CSCISM)
- Grid enabling technologies (ETH-Zurich, Forth)
- data management and assimilation (CINECA, TCD)
- distance learning and support (ICCC, UNI-C) and
- software efficiency and reusability (UPC, UiB).

This report covers the sixth study of the first phase.

In the second phase, ENACTS partners will undertake a demonstrator of the usefulness and problems of Grid computing and its Europe-wide implementation. Prior to the practical test, the Network will undertake a user-needs survey and assessment of how centres' current operating procedures would have to change with the advent of Grid Computing. As before, these studies will involve only a small number of participants, but with the results disseminated to all:

- European metacentre demonstrator: to determine what changes are required to be implemented by HPC centres to align them with a Grid-centric computing environment. This will include a demonstrator project to determine the practicality of

running production codes in a computational Grid and migrating the data appropriately (EPCC, UiB, TCD);

- user survey: to assess needs and expectations of user groups across Europe (CINECA, CSC);
- dissemination: initiation of a general dissemination activity targeted on creating awareness of the results of the sectoral reports and their implications amongst user groups and HPC centres (UPC, UNI-C, ENS-L).

Up-to-date information on ENACTS can be found on `http://www.enacts.org`

1.2 Study of Software Efficiency and Reusability

The objective of this study is to determine the implications for new, and more importantly, existing programs of using a pan-European computing metacentre. Specifically, the study will look at how portability between facilities can be ensured and how this will affect program efficiency.

Challenges and Potential Benefits. The uptake of computer simulation by new groups and the constant quest for greater efficiency of computer utilisation means that new techniques and approaches are always required. The study will address emerging software technologies that address these challenges and how they affect the efficiency and reuse of existing applications. This study aims to review the current position. The study has two beneficiaries: the users who will be able to work more effectively, and the computer centres which will be able to give better advice and produce more research for their capital investments.

Workplan. The principal deliverable from this activity is this Sectoral Report, commissioned by ENACTS, which will enable the participants to pool their knowledge, to investigate new developments in this area and to come up with recommendations to enhance software efficiency, portability, and reusability. This Sectoral Report is made freely available via the World Wide Web.

There were five workpackages (WP1-WP5) in this project, totalling 5.5 staff-months of effort:

• **WP1:** A survey of computer science techniques/methods and how they meet researchers' applications needs. It will consider established and emerging software standards for portability. Examples include MPI, OpenMP, Java, XML. In addition, it will consider computer programming languages and their recent extensions, high-level metalanguages, libraries for efficiency, and emerging new application areas. The deliverable is a report that is produced with input from other participants.

• **WP2:** A Sectoral Report which will distil the key messages from WP1. This deliverable is a report which will be made available via the WWW and whose contents will be disseminated.

• **WP3:** ENACTS aims to make the findings of the Sectoral Report widely available as quickly and cost-effectively as possible. This will be accomplished through a WWW

version on the ENACTS website and by encouraging participants to publicise it through conference talks, etc.

• **WP4:** This is the Final Report which is a deliverable for the EC and for the ENACTS General Meeting.

• **WP5:** Project Management.

The Participating Institution: Parallab, University of Bergen (UiB), Norway.

Parallab is the High Performance Computing Laboratory of the University of Bergen. Parallab is organized as a unit in the Bergen Center for Computational Science (BCCS), which is a section of UNIFOB, the University of Bergen's company for externally funded research. The Laboratory operates closely with various departments of the Faculty for Mathematics and Natural Sciences and in particular the Department of Informatics.

Parallab's staff pursues cross-disciplinary research and carries out projects with industry and other branches of science. The staff works with applications as well as basic research tailored to parallel and distributed processing, with a special emphasis on knowledge transfer to bridge the gap between computer science and computational sciences as well as the gaps between academic research and industrial needs.

Parallab has a track record in parallel computation that dates back to 1985. Parallab participated in various Framework III, IV, and V projects, including FRONTIER, SISCI, PARASOL, EUROGRID (and ENACTS). In recent years, Parallab has developed considerable interest and knowledge in grid-enabling technologies. Parallab is partner in the new Framework VI projects EGEE and HPC-EUROPA.

Parallab also operates the supercomputer facilities of the University of Bergen and provides support to the users of these facilities. Parallab is one of the four nodes in the Norwegian HPC infrastructure that is funded in part by the Research Council of Norway and in part by in-kind contributions from the participating partners. As such, Parallab has considerable interest in helping the user community of the Norwegian facilities in developing portable and efficient software.

The main author of this document was Jacko Koster with contributions from Csaba Anderlik and Jan-Frode Myklebust. Useful feedback to an earlier draft of this document was provided by various people active in the ENACTS consortium. We mention in particular Arto Teras (CSC), Satu Torikka (CSC), Nello Nellari (CSCS), Jean-Christophe Desplat (EPCC), and Giovanni Erbacci (CINECA).

1.2.1 Management

The Management Committee performed quality assurance on the project deliverables and accepted them on behalf of ENACTS. The Final Report is presented to the EC and to the General Meeting.

The Study Coordinator agreed to a detailed project and dissemination plan with the ENACTS Network Coordinator.

1.2.2 Scope of the study

During the study, we became aware of the difference in the way various software developers and development groups approach software reuse. For an academic researcher or research group, software reuse is typically the use of software that is in public domain, open source, or otherwise freely available to the researcher in some form. Reuse of software is often a way to acquire competence and improve and build upon methods, techniques and algorithms (and the software itself) that are developed elsewhere by colleagues or other research groups. Whether or not to reuse software is often a psychological and/or individual issue. In an industrial environment on the other hand, software reuse includes different concepts. Obviously, commercial considerations (like licensing, development cost estimates, market potential) play an important role, but also other issues like quality assurance, reliability, maintainability, and the support organization of the software are more critical. In all cases, software reuse aims at exploiting previously acquired competence and to reduce the development cost of new applications.

In the report, we primarily address the HPC user community that is connected to the ENACTS consortium, and this is mostly academic. However, we touch upon commercial aspects of code reuse as well.

The concept of reuse and efficiency of software is closely related to many other concepts in software engineering, including (but not limited to) the following:

- software architecture
- end-user and application programming interfaces
- software life cycle (design, prototyping, implementation, verification, testing, and maintenance)
- software quality, reliability, fault-tolerance
- software development environments

It is beyond the scope of this study to review all these in detail. There is a vast amount of methodologies and literature and on-going research in these areas. However, during the study we found it impossible to avoid addressing some of the aspects related to these concepts. For the writing of this sectoral report, we have tried to stay as close as possible to the original objectives of the study (efficiency and reusability) and minimized the inclusion of other software engineering concepts.

Software reusability in community-led initiatives deals a lot with the design of standards for protocols and languages. We have not attempted to describe recent and emerging language standards in detail. Many of these are still under standardization and therefore subject to change in the near future. Describing the temporary status of evolving standards in detail would make this report obsolete in a relatively short time.

2 Software Reusability

In this chapter, we discuss some general concepts related to software reuse and software efficiency. At first glance these may appear to be unrelated and to some extent they are. However, efficiency and reusability of software have always been key concepts for the HPC user community and HPC centres. Both concepts were traditionally mostly concerned with specific hardware architectures (typically those available within an HPC centre). With the emergence in recent years of new methodologies that enable large-scale distributed heterogeneous computing across administrative domains and country boundaries, the concepts of software efficiency and software reusability need to be reviewed.

2.1 General considerations

Software reusability is a broad term and it is easy to get confused about it. Some may say that software reuse is common, while others complain about the lack of it. However, describing a list of different definitions (or interpretations) of software reusability and then proposing a new one is not useful in the context this work. In this section, we will restrict the discussion to characterizing some of the main elements of software reuse that we found in the literature, and we identify some of the issues that arise from these elements. We discuss why reuse of software is necessary, why it does not always happen in practice and we describe some approaches and requirements to achieve and maximize the reuse of software. For the material in this chapter we have used several sources, in particular [59, 79, 85].

Software reuse can be defined as all activity that has using previously written software (in the form of specification, design, code, or related texts) as its goal. Reuse is a matching between new and old contexts. Whenever matching succeeds partially or completely, reuse is possible. Software reuse is not only about the re-application of a software component in another context or environment, it is also about the reuse of *knowledge*. This knowledge can include mathematical, engineering, and any other competencies that were needed for the construction of the reused software. Software reuse is getting considerable attention and research in methodologies, tools and techniques to promote reuse is prolific (see e.g., [71]).

The general opinion is that software reuse is good and that one should try to incorporate reuse of software in the development process where possible. Two basic reasons for software reuse are:

- **Increased productivity.** If the emphasis is on production speed and minimizing cost/effort, then being able to plug-in a collection of ready-made parts will be of great help. There is a reduction in the need for software design, implementation, testing, and documentation.
- **Increased software quality.** If the emphasis is on quality, then it will be advantageous if one has access to well-designed, well-tested, and well-documented code that has a proven correct and well-understood behaviour.

Using prefabricated parts will thus lead to higher productivity and to high-quality code. The comparison with electronics and product-line technology, where reuse of (hardware) devices is common practice, is easy to make [55].

Software reuse can result in substantial savings in the development costs as well as in end-products that are relatively small in size and low in complexity.

In order to increase productivity and quality, a programmer or (virtual) organization ideally constructs a piece of software once, verifies that it functions correctly and efficiently, and then reuses it as many times as possible in a variety of applications. In an environment where large-scale (international) software projects are carried out and multiple development projects exist at the same time, careful collaboration is needed to consider the possibilities for reuse.

The major considerations for software reuse are obviously not only technical, but psychological and economical as well. Everybody agrees with the noble purposes of reuse enhancing quality and productivity but if not properly planned, the effort involved can often be greater than expected or (seemingly) more costly than can be afforded.

If one wishes to reuse external (third-party) libraries and tools, chances remain that the resulting gains in productivity will not be dramatic. The libraries need to be the subject of study and experimentation so as to familiarize software developers with them. Considering the costs of this familiarization process, it may very well be that this required study turns out to be more expensive than developing the libraries themselves ('in-house').

Reuse of software that is developed 'in-house' is typically quite beneficial. From this the larger profits will come, since such libraries will often not create domain-specific problems and there is likely to be on-site competent staff available that can provide support and training for the software. Capturing this knowledge in a library will save time and effort later on.

Nevertheless, despite the above considerations, reuse of software happens all the time. Especially if one considers that writing programs in a high-level language involves reusing language constructs and definitions, standard libraries, low-level program generators (compilers), the operating system, etc. Appropriate questions are therefore: What forms of reuse are there? What is it that we are reusing? How can we promote reuse in all its forms?

Software reusability is the quality of a piece of software that enables it to be used again in another application, be it partial, modified or complete. In other words, software reusability is a measure of the ease with which previously acquired concepts and objects can be used in new contexts [81]. **Reusable software** is software (specification, design, code, etc.) made with reuse in mind. Reusable software may have been extracted from software that was not written with reuse in mind, but this is not often the case.

Before software can be reused, it has to be written. Planned reuse involves substantial changes in the way software is developed. Extra effort is needed to develop, test, and document reusable software and dedicated steps and time must be introduced in the development process to investigate opportunities to reuse existing components. Planned reusable software may therefore need upfront significantly extra investments and time to

create while only at a later stage benefits can be reaped. In an opportunistic approach to software reuse (i.e., unplanned reuse), individuals reuse components if they happen to know of their existence and if these components happen to fit. Salvaging code through opportunistic (unplanned) code grabbing is not a thing to relish. Such activity is often bound to take a lot of time before a well-documented high-quality piece of software is obtained. Unplanned reuse is therefore a process which (hopefully) produces reusable software, but compared to planned reuse should be considered a coincidence.

In principle, any newly written piece of software should be considered a candidate for reuse. This means that in principle every piece of software produced should be made reusable. This, obviously, does not happen. Many programs are written on the spur of the moment, and readability, reliability, and documentation suffer from a speedy approach and time constraints. Even if time is not a critical factor, many programmers still prefer to 'hack something first and improve later' rather than do 'serious programming' from the start.

Reverse engineering may be needed before software can be reused when the documentation produced during the development of the software is inadequate. Hence, in such cases it becomes necessary to reconstruct the documentation from the code to understand how the code works, and also to facilitate both corrective and enhancement tasks to the software.

A different picture exists for commercial projects. Here, with the aid of investments, enforced methodologies and so on, a considerable part of the software development cycle is spent on providing documentation, specifications and interfaces, formal designs, and well-annotated code. An important role here is played by the client, who may have a large influence on the result and the way a project is run.

Software reuse is closely related to software architecture. A software architecture provides a context for the development of reusable building blocks. Conversely, a software architecture provides a skeleton into which new building blocks can be incorporated and it allows implementations of existing building blocks to be replaced by new implementations. Attention to architectural issues is one of the prime criteria for successful software reuse.

A style of software development, emphasizing component reuse within an architectural framework, is known as *component-based software engineering* (CBSE). CBSE produces a set of reusable components that can be combined to obtain a high level of reuse while developing members of an application family. Typically, one first performs a domain analysis and then a layered modular architecture is defined that specifies key layers and core components, key subsystems and interaction mechanisms. High-level specifications and interfaces are defined for 'pluggable' components [74]. Developments in interface technology such as provided by middleware interface description languages (e.g. XML) provide additional leverage for CBSE. In a component-based technology, reuse is achieved by (partly) composing a new larger system from existing basic components or smaller systems. The building blocks used are copied from existing bases. Retrieval of suitable components is a major issue here.

Modern software engineering and programming practices stress **modularity**, especially at the software architecture level. Simply stated, an application is modular when its parts can easily be identified and replaced. Modularization (or component-based design) in software architecture and implementation is perhaps the most basic technique for building software with (new or from old) reusable parts. However, a major issue is to split up software systems into modules in such a way that reuse of one or more of these modules is best served.

Modularization has other advantages. Modularization allows separate and parallel development, it replaces the task of proving the correctness of the overall system by simpler smaller tasks of proving parts of the system, and above all it reduces the problem of maintenance.

Overall, modularity leads to better programs, and - as a bonus - it promotes reuse. Probably one of the first acknowledged steps in software engineering towards reuse was building well-documented software modules. The concept of a module can be found in almost any software system, e.g., subroutines and libraries. All modern programming languages and programming environments have a module concept incorporated in their design.

Reusable software is obtained by keeping various software engineering concepts in mind. Some of these we briefly summarize below.

- **Structured programming** concerns the adherence to a strict discipline of a few well understood control structures. The basic idea is to give careful thought to the structure of software as early as possible in the software development cycle. Its aim is to obtain modular software that is relatively easy to construct, easy to prove correct and maintain, by keeping the structure clean and simple.
- **Information hiding** concerns the separation of concerns. The underlying idea is to make those parts of a code invisible that are of no concern to the user or environment, making the total software package a set or hierarchy of black boxes, each performing a specified function. Key concepts here are data abstraction and specification.
- **Grouping related software pieces** into modules. The idea is to group logically related parts (data and operations) to ease construction and maintenance. Key concept here is functional abstraction.
- **Standardization of naming and interfaces** is an important condition for software reuse. This allows the glueing of software components to construct a larger software system. (Obviously, the quality of the 'glue' has a great impact on the reliability of the end product.)
- **Platform-independence** is crucial to enable software to be used on hardware platforms and in software environments other than the ones on which it has been developed. Machine-specific software instructions (usually low-level) are typically used to achieve efficient software implementations, but they prohibit the reuse of software in heterogeneous environments.

In a *black-box* reuse approach, components are reused as-is, that is, they are not modified in any way to fit the application in which they are being integrated. Often, the person

reusing the component does not know about the internals of the component. Commercial software systems that are constructed by using mostly (or only) readily available black-box software components, be it in the public domain, open source, or commercial, are also known as commercial off-the-shelf (COTS) software systems [66]. Quality, security and legal aspects of COTS-based software are critical issues. In a white-box approach, components can be modified before they are reused. White-box reuse is often done in combination with an opportunistic (ad-hoc) approach to software reuse.

Documentation is obviously of vital importance to the chances of success for a software module. The formality of the methodology for implementation will not always be decisive. A software module's interface and internal workings should be clear by means of a thorough specification, but otherwise by means of documentation and (ultimately) by availability of the source code.

A generalization of making ready-to-be-used software available for reuse, is the reuse of software pieces in which certain details are left open. *Templates* or *skeletons* are 'unfinished' components. By *instantiating* them, i.e., by filling in the unspecified parts (holes), a usable and reusable component results. A simple example is a procedure for sorting a sequence (array) of elements in which the relational operator (function) used to compare elements is provided to the procedure by means of a parameter. A template gets more general as more and more details are not specified. However, there is a danger that there is a price to be paid in software efficiency and time needed to obtain a working application.

Software quality criteria that are closely related to software reuse are

- software portability: the effort needed to transfer a software program from one hardware platform (or software platform) to another.
- software flexibility: the effort required to modify an operational program to accommodate changing needs or a changing environment.
- software interoperability: the effort required to couple one software system to another.

If one of these criteria is not achieved for some piece of software, then the potential for reuse of the software will be limited.

Reuse is not restricted to ready-to-be-used software (source code, libraries, binaries), but is a valid concept for other types of product as well, including design, objects, specifications, text, and so on. There is a strong trend in recent years to capture reusable design in formal specifications of protocols, interfaces, and languages rather than in actual source codes. (However, most often some reference or prototype source codes exist, especially during the development and standardization process of the reusable components.) The advantages of such more abstract approach are clear. First, the specification can be made independent of any programming language, system software, and hardware. This allows for multiple implementations of the specification that together cover a wide variety of heterogeneous environments. Second, developers and vendors can take such specification and build implementations that are tuned for efficiency (performance) on their hardware and software platforms.

2.2 Non-technical aspects of software reuse

Although a lot of software reuse is happening and many techniques exist that facilitate the creation and publication of reusable software, it is not always as prolific as one would like. Software engineering is not only concerned with technical aspects but with people and other environmental aspects as well. By being embedded in a society and being controlled by human beings, software engineering is influenced by these.

Non-technical issues are intertwined with technical ones. Arguments for or against reuse vary with the role someone plays in the software development process.

Economics plays an important role in making decisions about software reuse. As mentioned earlier, a reusable program requires changes in the way software is developed, and these changes are often not for free. The production of reusable software typically requires an extra initial investment which only pays off after a certain period of time. In a commercial environment, such extra investments require the approval from the management and this is subject to the market situation, etc. Software becomes a *capital* good, with higher initial costs coupled to profit returns over a longer period.

Code licensing is an example of an economical factor that influences software reuse. When proprietary reusable components are used in constructing a system, there are costs and legal issues concerning intellectual property, copyright, and scope of use.

The **management** also plays a key role in decisions about and success of software reuse. The management is largely responsible for planning the software development process. If reuse is not a clear objective from the start, then reuse will remain accidental. Programmers will tinker with the code and may only realize at a late stage that there are similarities between other or successive problems. Such unplanned approach is known as code scavenging or code salvaging.

A **support** structure needs to be put in place that is responsive to users' problem reports and that is able to repair deficiencies in a swift manner. Software (albeit commercial or not) is not likely to be reused by others if there do not exist any recognized contact points for support. It is not acceptable that the success and reputation of a commercial application is hindered by the software quality of embedded third-party software. In a commercial environment where the performance of the end-result is a major worry, the management may require the inspection of the source code of third-party reusable software to assure that the specifications meet the requirements. This is an argument against black-box reuse.

The **psychology** of the programmer is another aspect in software reuse. Software reuse means that a programmer has to adapt, incorporate, and work with software written by other programmers. A major stumbling block is whether the programmer is willing to do so. Writing the code yourself can often be more fun and/or be a sign of craftsmanship especially if it is in a field that is of high interest to the programmer.

Programmers are human, and therefore sensitive to feelings of distaste. A programmer may have a negative attitude towards software he/she is not familiar with. Many reasons for redesigning and rewriting software start with "I don't like the way that ...". These can

come from feelings of dislike or incomprehension. Programming styles for one are very personal and if a code (or documentation) creates a feeling of unease, the programmer may quickly decide to reject it or spend significant time on reworking it. Programmers often prefer 'to take a look inside'. Unfortunately, reusable software that is available in an unreadable (blackbox) format prohibits inspection of the code which could help in understanding the behaviour of the overall system and in the tracking of problems, especially when the provided documentation is insufficient.

Repeated failures to reuse software may discourage and disappoint a programmer. Also programmers must be aware that relatively new components (which may have shown and known problems) will only get better if they are used and their bugs -if found- reported.

Creating a positive attitude towards software reuse can largely be achieved by training and emphasizing the merits of software reuse. Organizing seminars with 'hands-on' training sessions to let programmers get actively acquainted with the software is a good way of boosting reuse.

Of course, there also exist many valid technical reasons why a programmer does not want to reuse software. These include for example algorithmic issues, performance issues, incompatibility of programming languages, language-specific internals (e.g., inappropriate data types), software that is not stand-alone (dependencies on other software may prohibit reuse), and software that is written with a different environment, purpose, or application domain in mind,

Users often have a close and continuous interaction with their software applications, especially if it involves visualization of data. The widespread use of graphical user interfaces illustrates that many aspects of application development and reuse are rooted in areas other than mathematics or algorithms. Increasingly more often, software designers must pay attention to the psychological aspects of human-machine interaction.

2.3 The Waterfall Model revisited

In order to determine the points where software reuse influences and appears in the software development life cycle, it is necessary to first draw a picture of that cycle. Many approaches to software development target a specific range of application domains and process constraints. We use the general, classic, and simple Waterfall Model as example in this section. See Figure 1. Many other methodologies exist to design and write software efficiently, e.g., eXtreme Programming (XP), the Spiral Process Model (SPM), the Incremental Process Model (IPM), the Unified Modeling Language (UML), and the Rational Unified Process (RUP). However, treatment of all these alternatives is beyond the scope of this document. We refer to [59] and the references cited therein.

The Waterfall Model distinguishes five steps:

- **Domain analysis.** A study will have to be made of the problem-domain, highlighting where problems may arise, where to find information on algorithms that may be needed, etc. This phase is also used to understand the nature of the project and

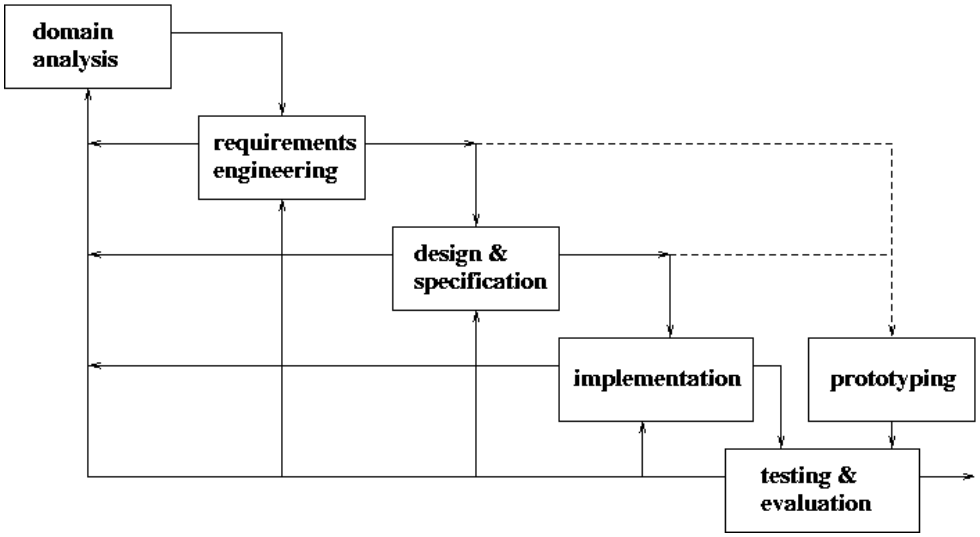

Figure 1: *The Waterfall Model [79].*

understand in broad terms what the customer wants. The general magnitude of the available time, funding, and personnel should be understood.

- **Requirements analysis.** The stakeholder(s), customer, and all other parties who have an interest in the product are interviewed (repeatedly) to obtain a usable and working description of what kind of product is needed. A typical component here is also the overall planning of the project.
- **Design and (technical) specification.** Working out a design for a part (or the whole) of the product, and reworking it into a specification for the implementation.
- **Implementation.** The actual implementation of the product has to be made.
- **System testing and evaluation.** The emerging product (and individual parts of it) have to be tested thoroughly with respect to the requirements specification. The final product is then delivered to the customer (acceptance). From then on, the product enters in maintenance mode which includes repairs and enhancements.

The Waterfall Model is quite general. Many refinements and modifications to the model have been proposed to suit specific application domains. Usually, design, specification, implementation, and testing overlap in their 'moments of truth', especially when the overall software project can be decomposed in several smaller but relatively independent software tasks.

The overall initial plan will undergo repeated refinements and modifications as more becomes known about the requirements and the design. In addition, experiences and findings at each step of the model can lead to partial, or even complete, restarts at a higher point, in order to steer the development away from observed shortcomings, or towards desirable refinements.

The practice of *prototyping* aims at making a (partial, simplified, or high-level) implementation at an early stage in order to find in the final product possible causes for restarts

sooner. Prototyping is also used to give the end-user an idea of what the resulting product will be like and to see if the requirements need adjustment. As a result, it is quite well possible that a customer poses new requirements during the software development process, not in the least because he/she realizes based on these intermediate results what the possibilities of modern automation are.

Reuse of software becomes particularly important when a restart is made while part of the software implementation is already completed or under final testing. It would be very bad if the development process requires repeated starts from scratch that invalidate large parts of the software. Preferably, most software parts that were (nearly) finalized, remain unaffected by such restart, or require minimal modifications. This puts significant requirements on the design and specification phase.

Advanced software engineering tools exist these days to guide the software developer through the development cycle, thereby providing guidelines and smooth transitions between development phases. Reducing the effort spent in a step can for example be achieved by *generating* or retrieving the documents or code for that step. For example, tools may be used to automatically generate the implementation from a specification, and entire modules may be retrieved from a library. Reuse can be seen as skipping a step or phase in the development process.

The detection of the potential for software reuse within the same project can itself be a major cause for rewinding part of the development process, especially if such reuse will short-cut part of the original development plan. Such short-cuts are typically ad-hoc and unplanned and a thorough design and specification of the software system should have avoided this.

The detection of the potential for software reuse for other (future or co-existing) projects can also be a cause for rewinding part of the on-going process. In commercial or time-critical development projects, this is a non-trivial economic issue, since this may lead to extra investments and delays that may not pay off at a later period of time.

Finally, we note that it is not only the reuse of the software or specifications that one should strive for. Reuse of information obtained during the domain analysis and design phases should be pursued as well, especially if there is potential for related follow-up projects. In addition, during the development of a large-scale software project, a lot of thought goes into the overall design (on paper) of the software. Reusing this design will result in productivity gains as well and avoid the repetition of mistakes in future development projects. Besides design information, any other form of knowledge gathered during software construction can potentially be reused. This can include for example software frameworks for automated testing. There exists a vast amount of literature and on-going research on reuse at all stages of the development cycle, but we consider this to be outside the scope of this study.

2.4 Making reusable software available

In this section we briefly review some of the popular tools that exist to make software available for wider use. Making software available happens at various stages of the development process. We distinguish between specification, libraries, and generation.

2.4.1 Specifications

Simply stated, a **specification** is a protocol or language that defines a set or hierarchy of basic components, each of which has an interface that details what to expect of it: what valid requests (input) are, and how it responds (output). Basic components can for example be data types, subroutines and objects (modules). The choice of abstraction at the interfaces is important and has a large impact on the reuse and acceptance of the software/specification.

In order to understand fully the behaviour of scientific programs it is important for users to be able to distinguish between what a specification prescribes and what it does not.

Defining a thorough specification is a hard task. A specification must be unambiguous. For each valid input to a component, the output must be well-defined and documented. Specifications are typically accompanied by tutorials, checkers, and examples.

A specification in itself is in general not an interesting object to retrieve. A software implementation of the specification is typically desired as well.

Documentation that accompanies a specification should include an overview of the design considerations. These have to focus on the problems encountered and choices made so as to allow software engineers to compare their situation with the one described in the documents. In addition, examples, test results, and component measurements should be included. Performance results (actual or theoretical) are often important criteria for HPC users to adopt software, especially if there exists software with similar functionality. In case the software is not black-box software, the documentation should aid re-users in adapting the components, and it should aid maintenance persons in the problem tracing and solving process. Finally, documents should include some installation guides, trouble shooting, FAQs, etc.

2.4.2 Libraries

Practically anything worth reusing can be made (publicly) available in a **library**. Code libraries have probably been around longest, and they are an accepted part of the software development process. Most libraries consist of source code or pre-compiled components. A library's API (application programming interface) is the means by which an application developer accesses and uses the parts of a library. Often, not all components of a library need be visible to the application developer, and then only the 'public' components or members are made accessible to the outside world.

Libraries can be developed in a hierarchical manner, that is, one library can make use of

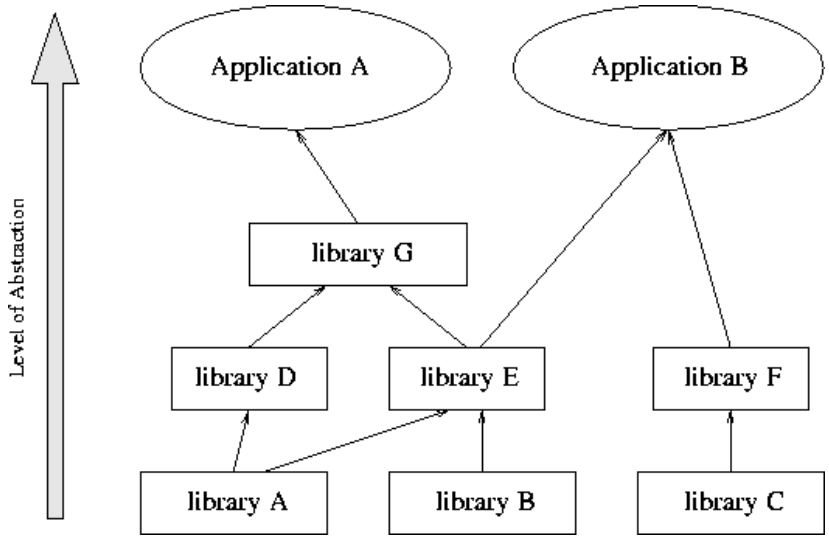

Figure 2: *Example of hierarchical library use.*

functionality that is defined is another library. In addition, a library can be used in more than one other library (reuse of code). This is illustrated in Figure 2. Section 4.1 includes examples of well-known software libraries for numerical applications. Another advantage of a library is that it can be tested and debugged independently of the overall application.

An obvious problem with a library is how to identify it (or part of it) when searched for. There are various approaches to the retrieval problem, ranging from text/keyword search up to using algebraic specifications. At least as important as the code is the specification and documentation of the library and its components. These present a problem to the library builder: although there are standards for the structure of a code library (usually determined by the program which links the several parts of a program to form an executable) such standards are hard to find for specification and documentation (manuals, on-line manual pages, hypertext links, etc.)

A large success rate can also be achieved with program skeletons (templates) that provide users with a rough framework for their program. The need for these skeletons is felt by the developers of programming environments, but they are mostly provided in the form of example programs that come with the libraries and manuals. Example programs often prompt users to start code-grabbing, remove the application specific part of the example program, and reduce it to the actual skeleton wanted. This can easily lead to programs that ”work, but don’t ask me why”, or contain hard-to-uncover bugs because of un-understood dependencies. This is unfortunate but practically inevitable as this is the consequence of an uncontrolled development process.

2.4.3 Generators

Code generation is the process by which some semi-automated tool - the code generator - is used to turn higher level (more abstract or even graphical) input into lower level (less

abstract) output. The output of the code generator often needs further processing before it is complete.

Automatic code generation is enabled by **generative programming**. A main goal of generative programming is to replace manual search, adaptation, implementation, and assembly of components with on-demand automatic generation of these components. Generative programming focuses on maximizing the automation of application development: given a system specification, generators use a set of *reusable* components to generate a (lower-level) concrete system (implementation).

Generative programming is about designing and implementing software modules which can be combined to generate specialized and highly optimized systems with specific requirements. The goals are manifold: (i) reduce the conceptual gap between program code and domain concepts, (ii) achieve high reusability and adaptability, (iii) simplify managing many variants of a component, (iv) increase efficiency both in code size and code execution time, and (v) increase productivity.

Generative programming introduces the notion of a domain-specific language. This is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power in a particular problem domain.

Compilers for programming languages like Fortran, C, C++, and Java, are some of the best developed code generators in existence. They not only generate executable files that can be executed by a machine but also evaluate large parts of the higher-level source code and try to reduce it to a more efficient implementation by taking the (hardware and software) context into consideration. In doing so, the actual programming language can be aimed more at the problem domain, while the task of finding an efficient implementation is moved to the compiler.

Code generators are very powerful tools that facilitate software reuse. When tailored to the application domain, they may be able to -automatically- retrieve and adapt code as well as documents. Examples are the so-called 'experts' or 'wizards' in development environments for microcomputer systems, which produce working skeletons for the program based on an interactively built specification. Other examples of generative technology are automatic configuration management, preprocessing (e.g., macro-expansion), and meta-programming.

Unfortunately, developing application generators and component generators is hard, as it requires being knowledgeable and experienced both in language design and compiler development. Recent developments such as XML-based technologies and template metaprogramming made generative programming more accessible to developers and thus revived the interest in generative programming.

2.5 Impact of Open Source

Open source [42, 60] presents one of the most interesting and influential trends in the software community over the past decade. Many software developers and organizations consider open source as a way to provide greater flexibility in their development prac-

tices, jump-start their development efforts by reusing existing code, and get access to a much broader group of users or market. As a result of the activity of the open source movement, there is now a vast amount of reusable software available that developers can use. There are many open source products that have become a basic building block in the software development process. Examples include EMACS, various GNU software, and scripting languages as Perl, PHP, and Tcl. Open source projects often focus around popular products such as Linux.

The open source approach is perhaps today's most successful form of large scale software reuse. Open source software is available for many activities, is often supported on many platforms, and can be used in many academic and industrial (business) domains.

The public image of an open source project is often that of a chaotic series of actions that by some miracle results in software that compiles and executes. The reality is however different. The open source movement has evolved toward a well-defined development process geared to unique requirements: coordination of a large number of code developers across the Internet bound together by common goals and interests. Open source approaches to software development have proven that complex, mission critical software can be developed by distributed teams of developers.

The open source movement appears to be a diverse collection of different ideas, knowledge, techniques, and solutions. However, all activities are motivated by the recognition of the benefits of broader and more open access to the software development process. Elements of the open source development process include the broadening the notion of a project team, greater collaboration across geographically dispersed teams enabled by the Internet, frequent releases of software builds, and the creation of publicly available source code for adaptation and reuse.

The open source approach is often a planned strategy to augment a popular free product, to take advantage of a wide informal network of developers and users, and to minimize the use of internal development resources. Besides the tangible result in the form of software, there are two other important aspects to open source [60]:

- **the business model**. By 'giving away' part of the intellectual property, one gets in return access to a much larger market or user base. However, before a company uses an open source model for the development of some product, it will have to weigh several factors. These include for example the potential that the software becomes a key platform for others, the impact on the revenue model, identified market-need, and gaining momentum among a community of developers and users.
- **the development process**. Characteristics include the encouragement of a wide community of users, testers and developers, frequent and vigorous communication with developers, frequent software releases, strong project coordination, responsiveness, and transparency in the development process.

Many people believe that open source is mostly a licensing issue, but there is more to it:

- release of open source code broadens its user base and encourages third-party participation; it also leverages the skills of a wider base of talented software engineers

- open source breaks the monopoly on software products and development by a few large software vendors
- open source enhances code reuse
- open source allows frequent releases of software systems in line with users expectations for greater responsiveness and shorter development cycle times
- open source relates to ethical aspects of software that are concerned with freedom of use and participation.

The open source operating system Linux has gained a large momentum over the past ten years. Much of this was due to its similarity to proprietary UNIX-based operating systems (e.g., IBM/AIX, SGI/IRIX, HP/HP-UX) but also to the perceived lack of reliability and speed of the Microsoft Windows platform and to the failure of other vendors to offer low-priced alternatives. As a result, many commercial organizations are now looking to deploy solutions to a target platform containing a number of open source products. They are attracted by the low cost of powerful server farms composed of commodity off-the-shelf hardware (clusters of PCs or small SMPs) running open source software.

In a commercial setting, licensing is a major consideration whether or not to use open source software. Open source licenses are typically straightforward, but their implications on any systems reusing that software can be profound. According to many open source licenses, the system reusing the software must also be considered open source and be subject to the same licensing agreements (see for example the GNU General Public License - GPL [42]). Software development with a commercial potential or objective is therefore at risk from infringements of any such open source licensing clauses.

Software reliability and security is another major consideration whether or not to use open source software. These considerations are often motivated by concerns on the quality of the software, how it was developed, and in particular what are the origins of the software (or parts of it). It is worthwhile to note that if the open source software has a large user base and active developer community, then this software is likely to be better tested and examined than any proprietary commercial code.

The maximal use of open source software and minimal dependency on proprietary technology is nowadays often considered an effective step to ensure the portability of an application. An unknown parameter for proprietary software is often when the closed-source library or tool or code generator or network protocol will no longer be supported ('end-of-lifed'), or when the interface will be altered in a backward-incompatible way. With open source code, one has always access to the source code. The interfaces of the leading-edge version of the open source software may change, but having access to the source code facilitates the modification and porting of the application.

Finally, we mention that SourceForge [17] is an internet site for software specifically designed to support collaborative development. It is an archive as well as a free development-hosting service. At the time of writing this document, it is undoubtedly one of the largest repositories of open source activity in the world. For more information on open source, we refer to [11, 30, 42].

3 Standards

If there were no standards in the software community, we would soon notice. Standards make an enormous contribution to many aspects of software development, research, and business, although that contribution is often unknown or invisible. When a software product meets one's expectations, this is often taken for granted. We are usually unaware of the role played by standards in raising levels of quality, reliability, security, efficiency, portability, interoperability, and interchangeability - as well as in providing such benefits at an economical cost.

Standardization is usually achieved through consensus by delegations representing all the stakeholders concerned: users, industry, commerce, government, and other interest groups. They agree on specifications and criteria to be applied consistently, for example in the manufacture and supply of products, and in the terminology and provision of services. In this way, international standards provide a reference framework, or a common technological language, between suppliers (vendors) and their customers (users) which greatly facilitates trade and the transfer of technology.

A **standard** is a specification for a specific problem domain that is widely accepted by a broad (international) community. There are additional requirements that include (but are not limited to) the following.

- **Completeness.** The functionality provided by the standard must cover the most urgent needs and requirements of the problem domain that is targeted.
- **Simplicity.** The definition of a standard must be easy to understand for the target community. The larger and more complex a standard, the longer it will take for the community to agree on its definition and the longer it will take to get wide support for it.
- **Consistency.** The standard must have a uniform design and use uniform notations throughout.
- **Flexibility.** The standard must be able to accommodate future changes in requirements with minimal modifications needed.
- **Acceptance.** Ideally, the standard is the product of a community-led initiative that includes the major 'market' players, such as the user community, government, and industrial and business organizations, in particular hardware vendors (that provide efficient machine-specific implementations). All parties involved in the standardization process must agree upon the contents of the standard.
- **Platform independence**, so that implementations are possible for a variety of architectures (software and hardware). This will help getting the standard accepted among a wider community.

Standardization is often a long (multi-year) and tedious iterative process, that includes domain analysis, public hearings, intermediate check-points, etc. It involves a wide or selected community of users, testers and developers, frequent and vigorous communication with developers. An *open* standard is not a well-defined concept, but typically involves a

fully transparent standardization process, a wide community of academic users, and does not pose restrictive copyrights or usage restrictions on the end result.

A developer of standards (be it an individual or an international network) should aim for technical excellence but be well aware that what is known and needed today may be insufficient to solve tomorrow's problems. A standardization body should therefore strive to build a standard that can evolve, without disrupting what already works. The principles of simplicity, modularity, compatibility, and extensibility of designs apply here. In addition, the standard developer must be responsive to signals of discontent from the user community.

The International Organization for Standardization (ISO) [34] is the world's largest developer of standards. Although ISO's principal activity is the development of technical standards, ISO standards also have important economic (and social) repercussions. ISO is a network of the national standards institutes of 148 countries, on the basis of one member per country with a Central Secretariat in Geneva, Switzerland, that coordinates the system.

Several other standardization bodies exist. Some of these we will mention later in this report when we address standards that are emerging or important for the HPC user community and HPC centres.

3.1 Programming languages

A programming language is a tool in the software development cycle. It provides useful abstractions, allowing the code to be written using higher level constructs than those available in pure machine code. Languages come in various shapes and sizes and most have been designed with a specific purpose or problem domain in mind. Even 'general purpose' languages are usually designed according to certain principles which have effect on their usefulness.

Standardization of a programming language is a way to fight dialects and enable cross-platform portability of source code (but not binary code). The major programming languages in high performance computing Fortran, C, C++, and Java have a detailed definition that is either published as an ISO standard or is a specification with international acceptance. The major hardware vendors support these standards with usually no or relatively minor deviations.

From the moment of conception, programming languages change. If a new version of a language definition is not backward compatible with previous versions, then continued reusability of old libraries is not ensured.

An additional problem of (frequent) language evolution exists. Even if the language is supposed to be portable across different platforms, it cannot always be guaranteed that the available compilers will support the latest changes in the language definition. Web browsers have long been notorious for displaying HTML incorrectly. This can often be attributed to errors in the HTML but also to the browser supporting an old version of HTML. The experiences with C and C++ show that the biggest growth in the user

community comes when a reasonable amount of platforms is supported with the same standard.

Portability problems with source code across platforms occur where the standard explicitly leaves something unspecified or where components with machine-specific functionality are invoked at run-time (e.g., timings or binary data files). Major portability problems seem to occur in those cases where the author of a program makes the assumption that other compiler implementations follow the same decisions as those made for the implementation he/she is using.

Obviously, all languages have strengths and weaknesses and support for reuse is not always present to the degree wanted. With the emergence of abstractions and modularity, support for reuse across programs and projects, at first an added bonus, became a goal in itself for new languages.

The UNIX operating system and its variants support a wider variety of application languages than does any other single operating system. One reason for this is the wide use of UNIX as a research and teaching platform. Another reason is the fact that matching application design with the proper implementation language(s) can make an immense difference in the developer's productivity. Therefore, the UNIX tradition encourages the design of domain-specific programming languages and what are now called *scripting languages* - those designed specifically to glue together applications and tools.

3.1.1 Fortran

We begin this section with a brief history of the evolution of the Fortran programming standard since it is exemplary for the way many other standards have evolved and it is illustrative for the problems that are also nowadays still encountered to get a new standard completed and accepted [80].

In the 1950s it became increasingly apparent that assembly programming (as was common in those days) was a tedious process, if only because of the length of time and complexity required to write and test a program. These ever recurring difficulties motivated a team led by John Backus of IBM to develop one of the earliest definitions of the high-level programming language Fortran. The development of Fortran was revolutionary as well as innovative. The aim was to develop a language that was simple, would allow programs to be written faster, but would nonetheless be as efficient in execution as assembly programs. Fortran was accepted rapidly by the software developer community and its use spread quickly as it fulfilled a real need. Inevitably, dialects of the language developed, which led to problems in porting programs between computers, and so in 1966, after four years of work, the American Standards Association (later the American National Standards Institute, ANSI) published the first ever standard for a computer programming language, now known as Fortran 66.

The proliferation of dialects remained after publication of Fortran 66, because many of the available compilers did not adhere to the standard and several features which were essential to large-scale programs were not covered by the standard. To overcome porting

problems between different machines, this initially resulted in the use of pre-processors. These were programs that read the source code of some well-defined extended dialect of Fortran and produced a code in standard Fortran 66. This preprocessing was a mixed blessing since it allowed dialects of Fortran to co-exist with Fortran 66. In addition, the generated pre-processed Fortran was often difficult to understand.

The above difficulties were largely resolved by the publication of a new standard in 1978, known as Fortran 77. It took considerable time before compilers became available and therefore the two standards co-existed for considerable time. By the mid 1980s, however, the changeover to Fortran 77 was in full swing. In the eighties, Fortran had gotten the company of several other programming languages, each with their own strengths and weaknesses and targeting specific types of applications. In order to bring the Fortran up-to-date with features of modern language design which were present in languages like Pascal, Ada, C, and C++, to standardize new and popular vendor extensions, and to answer to the requests from the Fortran user community, the language was brought up-to-date in 1991 (after a tedious 10-year process) with the publication of the Fortran 90 standard (ISO/IEC 1539:1991 and ANSI X3.198-1992).

Fortran 90 introduced several new features beneficial for producing reusable, reliable, maintainable, and portable scientific programs. While user-defined types, array-syntax and dynamic memory capabilities are well known beneficial features of Fortran 90, the language supports many other modern software concepts, included object-oriented programming. Although Fortran 90 is not an object-oriented language, it can directly support important concepts of such languages including abstract data types, encapsulation, function overloading, and classes. Fortran 90 is *backward compatible* with Fortran 77, and thus new concepts can be introduced into existing Fortran 77 programs in a controlled manner. This allowed experienced Fortran 77 programmers to modernise their software, making it easier to understand, modify, share, and extend based on the benefits modern programming principles provide.

The Fortran 95 standard (ISO/IEC 1539-1:1997) continued the evolutionary model introduced in Fortran 90 by deleting several of the features marked as obsolescent in Fortran 90 and identifying a few new obsolescent features. Fortran 95 is seen as a relatively minor enhancement of Fortran 90 and introduces a small number of new features, together with the incorporation of the corrections to various defects which were found in Fortran 90.

Fortran stresses arithmetic. Fortran's strength has always been in the area of numerical, scientific, engineering, and technical applications, and the language is known for the availability of extensive numerical libraries.

Despite being perceived by many as being not up-to-date with modern programming standards, it is still the language of choice for many computational scientists. The reasons for this are manifold. First, the Fortran language is simple (yet powerful for its purpose) and it takes the inexperienced programmer little effort to master the basics in Fortran. Second, the Fortan language has been very popular in the past four decades and the scientific community has a truly vast investment in Fortran codes, with many programs still in frequent/production use and under continuous development and some of which of 100,000 lines or more.

The ISO Fortran Standards Technical Committee WG5 [38] is the working group committee responsible for the development and maintenance of Fortran language standards. WG5 has recently finished a major revision of the Fortran language (termed Fortran 2003).

3.1.2 C

The C programming language also has its standard references. The first widely-accepted version of the language is these days still referred to as 'Kernighan & Ritchie C' [77] as opposed to the current standard 'ANSI-C' (ISO/IEC 9899) adopted by ISO in 1999.

C is basically a systems programming language. As such it is famous for its ability to handle and manipulate pointers (a variable that contains a memory address and can point to any data object in the memory space of a program). Unfortunately, its also famous for its cryptic notation. C code *can* be non-portable, and the ability to write (efficient) machine-specific code is considered by many one of the strengths of C (including the standardization committee). Major problems in the portability of C programs usually center around the interface to the operating system.

A significant contribution of C to software reuse is its standard preprocessor. This preprocessor provides the programmer with a macro facility, compile-time source file inclusion and compile-time code selection. Separate source files, to be included using this preprocessor, are generally used for declaring constants (as macros) and function prototypes of routines not located in the source to-be-compiled. Also, system dependent source configuration is typically handled by macros. Effective use of the preprocessor greatly enhances the portability of C programs across different hardware and software environments.

C's ability to handle not only pointers, but also mixed pointer-integer arithmetic in a portable way, has certainly drawn attention. In spite of the obvious dangers, the language definition allows the programmer to use this feature without the need to be aware of the particular machine's architecture. Pointer arithmetic in itself is a feature that should only be used as a last resort, but the way in which it has been incorporated in C shows the attention that has been given to code portability.

3.1.3 Object-oriented programming

Object-orientation is one of the most popular buzzwords of the past decade. However, the approach taken by object-oriented languages is widely considered to provide the best support for code reuse. Object-orientation offers an alternative paradigm to 'traditional' languages. Traditional languages can be regarded as describing a collection of processes in the sense that data objects (or references to objects) are given to procedures that can manipulate them. Object-oriented languages on the other hand work with a collection of objects in which the data is hidden and only procedures declared as belonging to that type of object are allowed to operate on it.

The inheritance and polymorphism features of object-oriented languages provide opportunities for constructing reusable parts. They promote the emergence of abstract classes

that may serve as a starting point from which more specific components are derived. Object-oriented types (or 'object classes') form a hierarchy, where basic types contain basic properties and are then extended to form new types. The new types are said to *inherit* the components of the old types. These newly defined types are therefore refinements (subtypes) of the basic types (supertypes). In traditional languages, such a super- and subtype notion can usually only be found with ordinal types and their subranges.

Objects are a collection of operations that share a state. The variables representing that state are called the instance variables and the operations are called the methods. An object is an entity whose state (i.e., the values of its instance variables) is (partially) hidden from the outside world and whose interface consists of a set of methods that form the only means by which the object's instance variables can be accessed or modified. A class is an object's type. All objects from the same class have the same set of instance variables and methods available. Inheritance is the mechanism that allows classes to be subclasses of other classes, where subclasses offer a superset of instance variables and methods.

The run-time efficiency of object-oriented programs is often considered poor compared to that of non-object-oriented languages. Partly this is based on prejudice, but it is clear that compilers for object-oriented languages are more restricted in how they can optimize codes due to the relatively complex language definitions.

3.1.4 C++

The C++ programming language is based on the C language. Around 1995, an ISO committee was formed to standardize C++. The new standard was ratified in 1998 and is officially known as ISO/IEC 14882-1998. The design goals for C++ [84] were the creation of a high performance object-oriented language in which almost all ANSI-C conforming programs will work without much conversion. The major extensions offered by C++ are object classes with multiple inheritance, templates and exceptions. Templates allow functions and type definitions to be a parameterized skeleton. C++ allows overloading of functions and operators and (user-defined) implicit conversion functions. These greatly enhance the expressive power of C++ (compared to C) at the cost of a reduced external visibility of the semantics.

C++ does not require each data object to belong to an object class, since it retains all features of C. This makes C++ a versatile object-oriented language designed for high performance at the cost of relative 'low-levelledness'.

3.1.5 Java

The programming language Java has an internationally accepted specification whose evolution is in the hands of the Java Community Process (JCP) [36]. The JCP fosters the evolution of Java in cooperation with the international Java developer community, and is overseen by the Project Management Office (PMO) of Sun Microsystems. The specification of Java has not been standardized by any international standards organization.

Proposals for changes to the Java specification can be submitted as so-called Java Specification Requests (JSRs). Expert groups are appointed to consider each JSR, and draft specifications are published and voted upon. Public participation is encouraged and individuals can comment on public drafts.

The object-oriented language Java offers a number of potential benefits as a language for high performance computing, especially in the context of distributed heterogeneous computing and the emerging Grid. Java offers for example a higher level of platform independence (portability) than the traditional HPC programming languages which is important in heterogeneous computing environments and in areas where the lifetime of application codes often exceeds that of the machines.

Java's portability is primarily due to the compilation and execution mechanism. Java source code is compiled using the Java compiler to generate byte code that is platform neutral. This byte code is compiled for the Java Virtual machine (JVM) rather than for a specific target hardware. The JVM provides the extra level of abstraction since it is a software layer implemented on top of the hardware and operating system. Java byte code therefore runs on any system that supports a JVM. Besides the Java byte code, the Java source code itself is also very portable since the language specification ensures that no platform dependent aspects exist.

Another feature of Java that makes it interesting for the Grid is that it provides mechanisms for carrying out so-called Remote Procedure Calls (RPCs). Remote Method Invocation (RMI) enables an application running in one JVM to make method calls to objects located in another JVM, The Java Interface Definition Language (IDL) allows distributed objects to interoperate. The strength of these built-in mechanisms lies in the fact that they offer support for distributed computing that hides differences in connections and communication protocols between distributed and heterogeneous resources. Java also provides support for standard TCP communication (sockets).

Java has a number of features that supports the writing of bug-free code. These include for example strong type-checking, run-time array-bound checking, lack of pointers, garbage collection (to automatically free redundant objects), and exception handling. In addition, Java also has a number of security features that restrict what an untrusted code can do/access on the system ('sandbox' restrictions) or verify that the code comes from a trusted source (using digital signatures).

The Java Grande Forum [35] is a community initiative led by Sun and the Northeast Parallel Architecture Center (NPAC). The forum involves vendors and academics from the US and Europe that aim to promote the use of Java for large computational and memory intensive applications (so-called 'Grande' applications). Grande applications include computational science and engineering codes, as well as large scale database applications and business and financial models. The forum's aim is to propose modifications to the Java language to make it more suitable for the development of Grande codes, and to promote this use in the community. The forum consists of a number of Working Groups that address a number of concerns that surround Java. These include primarily

- numerical issues (floating-point reproducibility, lack of complex numbers, lack of

full support for the IEEE 754 standard, lack of multidimensional arrays),
- benchmarking and performance issues,
- (lack of) support for parallel programming paradigms.

With respect to the last point, JOMP [26] is a research project whose goal is to define and implement an OpenMP-like set of directives and library routines for shared-memory parallel programming in Java. (See Section 4.3 for OpenMP.) Java has a native threads model, but a directive system like OpenMP is considerably easier to use, less error prone and allows compatibility to be maintained with a sequential version.

Benchmark studies carried out by EPCC as part of the UKHEC project show that the performance of Java is still an issue of concern. However, for certain applications, recent Java implementations have become rather competitive compared to the more traditional HPC implementations in C and Fortran (see [75, 83] and the references therein).

3.2 Scripting languages

Besides programming languages, a code developer often makes use of **scripts**. Scripts are often used as *macros* that replace a series of application or system commands by one command. A number of scripting languages have gained immense popularity in the last decade. These include Perl [44], Python [45], Tcl [47], and the many variants of the Berkeley UNIX Bourne shell (sh).

There is not an exact delineation between programming languages and scripting languages. However, scripting languages are typically interpreted (i.e., scripts execute without preliminary compilation) or compiled, whereas programming languages are usually only compiled. Programming languages are usually strongly typed, which means that variable types must be defined at compile-time. This has the advantage that many programming mistakes can be caught before execution. Scripting languages are typically weakly typed, that is, command arguments do not have a pre-defined type, but the type is determined at runtime.

Many of the major scripting languages (Perl, Tcl, Python, etc.) have outgrown their origins and are now standalone general-purpose programming languages of considerable power. The naming tends to obscure the similarities with other languages that are not usually considered to be a scripting language. Modern scripting languages have features that are traditionally associated with programming languages. These include for example functions, static scoping, object-oriented facilities and libraries of utility objects or functions. Scripting languages typically have good string-handling primitives. Scripting languages are usually portable across architectures.

Since scripting languages are often either interpreted or compiled (to an abstract byte code as opposed to real machine instructions), the use of scripts often comes with a penalty in performance. That is usually not an issue in a user interface but is not tolerable in a program kernel loop that is to be executed millions of times. Programming languages are therefore deemed more suitable for implementing computationally intensive algorithms or other time-critical functionality.

Nevertheless, scripting languages have found their way into scientific computing. Scripts are often used to glue together existing programs thereby enabling the reuse of existing functionality. Scripting languages allow for the wrapping of software written in different programming languages so that one obtains a bundling of efficient subroutines, along with some interface code, into a shared library. This library can then be loaded at runtime, and accessed from the scripting language. For example, scalable, parallel solvers can be implemented in Fortran, C or C++ and wrapped into a shared library, and Python is then launched as a parallel (MPI) application that loads the necessary functionality at runtime.

3.3 Middleware

Middleware is nowadays often used to provide a seamless bridge between the simple, standard programming interfaces and desktop scientific computing environments that dominate the work of computational scientists and the rich supply of services supported by the emerging Grid and Web services.

In the object-oriented paradigm, it is often stated that two objects, a *client* and a *server*, have agreed upon some contract, whereby the client object is allowed to request certain services from the server object [85]. A natural step is to isolate the contents of this contract, or interface, from both the client and server.

The purpose of the interface is to be independent of platform, operating system, language, and network protocol. This provides a layer which promotes reuse across platforms, operating systems, and programming languages. A layer with such properties is known as **middleware** in distributed computing.

The Common Object Request Broker Architecture (CORBA, [22]) is one of the earliest examples of middleware. CORBA has been developed by the Object Management Group (OMG, [40]), a multi-vendor effort to standardize object-oriented distributed computing. JavaBeans and Microsoft's .COM are similar solutions to the problem of connecting components with a very long wire.

CORBA interfaces are expressed in IDL, the Interface Definition Language, whose syntax resembles that of C++. The interface construct collects operations that form a natural group. Typically, the operations of one abstract data type constitute such a group. For each operation, the result type is given, followed by the operation names, and finally the parameters. CORBA tools handle the actual communication between objects that provide services and objects that request them.

3.4 Protocols, Markup languages, and Meta languages

Software interoperability is fundamental to such topics as development based on components, reuse, and infrastructure for distributed or network-based systems. As a result, a variety of (often partial) approaches to interoperability have appeared, each offering a potential solution to interoperability problems. Some of the most popular and most promising protocols and languages that appeared are the focus of this section.

The World Wide Web Consortium (W3C, [8]) both develops and promotes standard technologies for the Web. W3C is an international consortium with approximately 450 member organizations and a full-time staff of more than 70 people who contribute to the development of W3C specifications and software. W3C members send engineers to work in W3C Working Groups together with W3C technical staff (the W3C Team) to produce technical specifications for Web technologies.

W3C is a vendor-neutral organization, promotes interoperability by designing and promoting open (non-proprietary) computer languages and protocols that avoid the market fragmentation of the past. This is achieved through industry consensus and encouraging an open forum for discussion. W3C makes a strong commitment to interaction and accountability with the public. W3C invites the Web community to participate on public mailing lists and to interact with the W3C Team at workshops and conferences.

W3C produces what are known as 'Recommendations'. These are specifications, developed by W3C working groups, and then reviewed by members of the consortium. A W3C Recommendation indicates that consensus has been reached among the Consortium Members that a specification is appropriate for widespread use.

Another international standardization body is the Internet Engineering Task Force (IETF, [5]). IETF is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. As in W3C, the actual technical work of the IETF is done in working groups that are organized by topic into several areas (e.g., routing, transport, security, internet, etc.)

Below, we summarize some major standards in Internet and Web protocols and languages that are in use or emerging and that facilitate software (and application) interoperability and software reuse in distributed environments. They provide important capabilities for glueing together stand-alone applications in a platform-neutral manner.

We restrict the summary to languages that are of increasing importance to the HPC user community that is active or has interest in distributed computing technologies. The summary illustrates the hierarchy among some of the standards. Since most of the described languages are emerging or not yet adopted by many computational scientists, we provide some details so that the reader can get some feel for the syntax and semantics of the languages.

• **HTTP - Hypertext Transfer Protocol -** is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use since 1990. The first version of HTTP (HTTP/0.9) was a simple protocol for raw data transfer across the Internet. Its successor HTTP/1.0 improved the protocol by allowing messages to be in the format of MIME-like messages, containing meta-information about the data transferred and modifiers on the request/response semantics. HTTP/1.1 subsequently took into consideration hierarchical proxies, caching, the need for persistent connections, and virtual hosts.

HTTP is a request/response protocol. A client sends a request to the server in the form of a request method, URI/URL, and protocol version, followed by a MIME-like message

containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code (200 = success, 400 = Bad Request, 404 = Not Found, etc), followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content.

Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection between the user agent and the origin server. A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel. A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or part of the message, and forwarding the reformatted request toward the server identified by the URI. A gateway is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol. A tunnel acts as a relay point between two connections without changing the messages; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages.

HTTP communication usually takes place over TCP/IP connections. A lot of the performance work on HTTP has been to improve the interactions between HTTP and TCP.

• **SGML - Standardized Generalized Markup Language -** is a system for defining markup languages. Authors mark up their documents by representing structural, presentational, and semantic information alongside content. HTML is one example of a markup language (see below). Each markup language defined in SGML is called an *SGML application*. An SGML application is generally characterized by:

- A **declaration** that specifies which characters and delimiters may appear in the application.
- A **document type definition** (DTD) that defines the syntax of markup constructs. The DTD may include additional definitions.
- A specification that describes the semantics to be ascribed to the markup. This specification imposes syntax restrictions that cannot be expressed within the DTD.
- Document instances containing data (content) and markup. Each instance contains a reference to the DTD to be used to interpret it.

• **HTML - HyperText Markup Language -** is the *lingua franca* for publishing hypertext on the World Wide Web. It is a non-proprietary format based on SGML and can be created and processed by a wide range of tools, from simple plain text editors to sophisticated WYSIWYG authoring tools. The most recent specification of HTML (4.01) was recommended by W3C in December 1999.

HTML documents have a *predefined* structure and make use of *predefined* tags ('markups') to structure text into headings, paragraphs, lists, hypertext links etc. Figure 3 shows an example of an HTML document.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
     "http://www.w3.org/TR/html4/strict.dtd">
<html>
   <head>
      <title>My first HTML document</title>
   </head>
   <body>
      <p>Hello J-C!
   </body>
</html>
```

Figure 3: *An example of an HTML document. An HTML document is divided into a head section (between the tags <head> and </head>) and a body (between the tags <body> and </body>). The title of the document appears in the head (along with other information about the document), and the content of the document appears in the body. The body in this example contains just one paragraph, marked up with the tag <p>.*

Related to the web and HTML are Uniform Resource Identifiers (URIs, better known as URLs). URLs are short strings that identify resources in the web: documents, images, downloadable files, electronic mailboxes, and other resources. They make resources available under a variety of naming schemes and access methods such as HTTP, FTP, and Internet mail addressable in the same simple way.

• **JavaScript** gives HTML designers a programming tool with a language with a relatively simple syntax. It was designed to add interactivity to HTML pages. JavaScript is a fairly universal extension to HTML that can enhance the user experience through event handling and client-side execution, while extending a Web developer's control over the client's browser. JavaScript can put dynamic text into an HTML page, react to events (page loading, mouse clicks, window opening, etc.), and it is often used to validate data (e.g., validate form data on the client side before it is submitted to a server).

JavaScript was developed by Netscape and is the most popular scripting language on the internet. JavaScript is a lightweight language that consists of lines of executable computer code that are embedded directly in HTML pages. JavaScript is an interpreted language (i.e., scripts execute without preliminary compilation) and is supported by all major web browsers. JavaScript is a 'safe' language, one that cannot access any system controls or hardware on a user's computer.

JavaScript is very different from Java but has some object-oriented features and several programming structures that are similar. JavaScript contains a much smaller and simpler set of commands than Java.

• **XML - eXtensible Markup Language -** [49] is a simple, flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. In fact, XML has become the universal format for structured documents and data on the Web. It allows one to define markup formats when HTML is not a good fit.

XML is a markup language much like HTML. However, XML is a complement to HTML and not a replacement for HTML. It is beyond the scope of this report to explain in detail the differences between XML and HTML, but simply stated, HTML is designed to mark the structural elements that describe the look, feel, and action of a web document (important for the display of data), whereas XML is designed to describe data.

XML was designed to store, carry, and exchange data. XML is a cross-platform, software and hardware independent tool for transmitting, sharing, and storing information. Computer systems and databases contain data in incompatible formats. One of the most time-consuming challenges for developers has been to exchange data between such systems over the Internet. Converting the data to XML can greatly reduce this complexity and create data that can be read by many different types of applications.

A strength of XML is that it makes data available to more users. Remote clients and applications can access XML files as data sources, like they are accessing databases. Data can be made available to all kinds of 'reading machines' (agents). Another strength of XML is that it enables the separation of data from HTML, that is, data is stored outside HTML files and can be stored in separate XML files. Hence, one can concentrate on using HTML for data layout and display, and be sure that changes in the underlying data will not require any changes to the HTML.

A big difference with HTML is that in XML tags are *not predefined*. XML allows the author to define new tags and a new document structure. Figure 4 shows an example of an XML document.

The data model for XML is simple. XML provides no more than a baseline on which more complex models can be built. The main structure of an XML document is tree-like, and most of the lexical structure is devoted to defining that tree, but there is also a way to make connections between arbitrary nodes in a tree. For example, Figure 5 shows a root node with three children, and one child has a link to another child.

Related to XML are the Document Object Model (DOM) and the Simple API for XML (SAX) that define powerful interfaces, properties and methods to manipulate XML documents. We will not detail these further.

• **BinX** is an example of a language and tool that is based on XML. With BinX, one can describe the layout and contents of binary files. Numerous large files of binary data store a wealth of computational science information and many existing applications make use of such files. Large datasets are output from various applications or are gathered and recorded by sensing devices (e.g., particle physics and astronomy). The sharing of binary data over the Web is however difficult, because there is no universal binary data format available. BinX defines an annotation language for representing meta-data for binary streams, and specifies a data model to describe the data types and structures contained in binary data files. BinX can for example describe the content, structure and physical layout (endian-ness, blocksize) of binary files.

BinX is being developed by edikt [25], based at the National e-Science Centre at the University of Edinburgh. edikt aims to construct novel software tools that underpin the linking, management and interpretation of the vast amounts of data available on global

```
<?xml version="1.0" encoding="ISO-8859-1"?>
   <catalog>
      <cd>
         <TITLE>La Revancha del Tango</TITLE>
         <ARTIST>Gotan project</ARTIST>
         <COUNTRY>France</COUNTRY>
         <COMPANY>ya basta</COMPANY>
         <YEAR>2001</YEAR>
      </cd>
      <cd>
         <TITLE>Never mind the Bollocks</TITLE>
         <ARTIST>The Sex Pistols</ARTIST>
         <COUNTRY>UK</COUNTRY>
         <COMPANY>Virgin Records</COMPANY>
         <YEAR>1977</YEAR>
      </cd>
   </catalog>
```

Figure 4: *An example of an XML document for a CD catalogue. The first line in the document - the XML declaration - defines the XML version and the character encoding used in the document. In this case the document conforms to the 1.0 specification of XML and uses the ISO-8859-1 (Latin-1/West European) character set. The next line describes the root element of the document* <catalog>*, the next 4 lines describe a child element of the root* <cd> *that on its turn has five child elements (*<TITLE>*,* <ARTIST>*,* <COUNTRY>*,* <COMPANY>*, and* <YEAR>*). The last line defines the end of the root element.*

networks.

The Data Format Definition Language (DFDL, [12]) working group in GGF grew out of BinX. DFDL is an XML-based language to delineate the structure and semantic content of all type files. BinX is a subset of DFDL, restricted to binary files and supporting limited semantics. Related work to BinX are the Binary Format Description language (BFD, [9]), the Hierarchical Data Format (HDF, HDF5, [10, 14]), the External Data Representation (XDR, [82]), and the Resource Description Framework [78].

For other references to data management tools, we refer to the Sectoral Report on *Data Management in HPC* that was released earlier in the ENACTS network [52].

• **XHTML - eXtensible HyperText Markup Language -** is a family of current and future document types and modules that reproduce, subset, and extend HTML, reformulated in XML. XHTML document types are all XML-based. XHTML is the successor of HTML. Some of the expected benefits of XHTML include reduced authoring costs, an improved match to database and workflow applications, a modular solution to the increasingly disparate capabilities of browsers, and the ability to cleanly integrate HTML with XML applications. XHTML 1.0 is the first major change to HTML since HTML 4.01.

According to [69, 53], a **service** is ”a network-enabled entity that provides a specific capability. [...] A service is defined in terms of the protocol one uses to interact with it and the behavior expected in response to various protocol message exchanges (i.e., service = protocol + behavior).”

```
<p>
<q id="x7">The first q</q>
<q id="x8">The second q</q>
<q href="#x7">The third q</q>
</p>
```

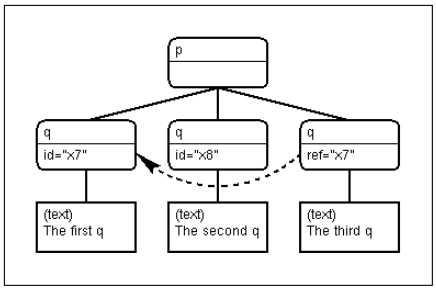

Figure 5: *Example of XML data tree representation [8]. The third* `q`*-tag has an* `href` *attribute and it points to an element with an* `id`*. In this case the first* `q`*-tag has an* `id` *with the same value as the* `href` *(minus the #), so the third* `q`*-tag points to the first.*

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

Figure 6: *A SOAP Example. A* `GetStockPrice` *request is sent to a server (left panel). The request has a* `StockName` *parameter. A* `Price` *parameter will be returned in the response (right panel). The namespace for the function is defined in the "*`http://www.stock.org/stock`*" address.*

• **SOAP - Simple Object Access Protocol -** is a simple XML-based protocol that provides a means of messaging between a service *provider* and a service *requestor*. More generally, SOAP is a communication protocol for sending messages via Internet between applications running on different operating systems, with different technologies and programming languages.

SOAP provides a simple mechanism that defines a remote procedure call (RPC) convention and a messaging convention. SOAP is independent of the underlying transport protocol: SOAP messages can be carried on HTTP, FTP, and the like. Traditionally, RPCs represented a compatibility and security problem and firewalls and proxy servers would normally block this kind of traffic. SOAP provides an alternative way to communicate between applications (via HTTP), because HTTP is supported by all Internet browsers and servers.

A **Web service** "describes an important emerging distributed computing paradigm with focus on simple, Internet-based standards (e.g., eXtensible Markup Language: XML [...]) to address heterogeneous distributed computing. Web services define a technique for describing software components to be accessed, methods for accessing these components, and discovery methods that enable the identification of relevant service providers." [68, 53]

Web services have generated considerable excitement in the global computing industry recently [62]. For end-users, the vast increase in available services can imply either more freedom or more chaos. For developers, Web services pose significant theoretical and engineering challenges as they need to determine how to leverage emerging technologies to automate individual applications based on cross-organizational, heterogeneous software components.

Web services describe an important emerging distributed computing paradigm that differs from other approaches such as CORBA and Java RMI in its focus on simple Internet-based standards to address heterogeneous distributed computing. Web services are programming language-, programming model-, and system software-neutral. One novel aspect of Web services is that they disregard (or eliminate) the traditional separation between local and global applications, and therefore pose significant challenges related to for example security, accounting, billing, and workflow/load workload management.

In a Web services context, software is increasingly more considered as a service. In a demand-led software market, businesses can assemble systems from component Web services to address particular requirements. Developers will offer applications as a federation of Web services that are only linked at the point of execution, rather than developing software components and binding them together to form a single, rigid application.

Web services can roughly be defined as web-based applications composed of coarse-grained functions accessed through the Internet. From a technical perspective, Web services are a standardized way of integrating web-based applications using open standards including XML, SOAP, WDSL, and UDDI. XML structures the message, SOAP transfers the message, WSDL describes the available services, and UDDI lists them. XML describes both the nature of each self-contained function and the data that flows to and from it.

Web services must conform to a particular specification format to ensure that each service function will easily integrate with others to create a complete (business) process. This interoperability allows the publication, discovery, and aggregation of a range of Web services through the Internet to more easily create innovative applications.

Several Web services have been competing to become the platform of choice for Web services developers. Although Web services technology is platform neutral, each camp provides specific development tools and APIs. These include Microsoft's .NET and Sun Microsystems' J2EE. J2EE is an open standard, but there are several J2EE-based development environments proposed, including IBM WebSphere.

Over the years, the goal of achieving software interoperability across computer networks has remained elusive. Standards such as CORBA and DCOM have met with marginal success. Web services may just be an evolutionary step toward software interoperability instead of a revolutionary jump. However, the XML/SOAP approach offers some significant advantages. Web services use document-style messages that offer flexibility and pervasiveness. With the strong support that major market players are providing, Web services may achieve what CORBA and DCOM could not.

• **WSDL - Web Services Description Language -** is an XML-based language for *de-*

```
<definitions>
<types>
   definition of types........
</types>
<message>
   definition of a message....
</message>
<portType>
   definition of a port.......
</portType>
<binding>
   definition of a binding....
</binding>
</definitions>
```

Figure 7: *The main structure of a WSDL document.*

scribing Web services and how to access them. Web services are a set of *endpoints* operating on messages containing either document-oriented (messaging) or RPC payloads. Service interfaces are defined abstractly in terms of message structures and sequences of simple message exchanges (operations), bound to a network protocol and data-encoding format.

An WSDL document specifies the location of the service and the operations (or methods) the service exposes. Figure 7 shows the main structure of a WSDL document. The Web service is defined using four major elements:

- <portType> defines a web service, the operations that can be performed, and the messages that are involved. This is the most important WSDL element.
- <message> defines the data elements of an operation.
- <types> defines the data types used by the web service. XML Schema syntax is used for this (for maximum platform neutrality).
- <binding> defines the message format and protocol details for each port used by the web service.

A WSDL document can also contain other elements, like extension elements and a service element that makes it possible to group together the definitions of several web services in one single WSDL document. Figure 8 shows a more concrete example of a Web service.

• **UDDI - Universal Description, Discovery and Integration -** is a *directory service* for storing information ('listings') about Web services. UDDI is based on many of the earlier mentioned standards, such as XML, HTTP protocols. In particular, UDDI uses WSDL to describe interfaces to Web services. Cross-platform programming features are addressed by adopting SOAP.

```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>
```

Figure 8: *A simplified excerpt of a WSDL document. The <*`portType`*> element defines 'glossaryTerms' as the name of a port, and 'getTerm' as the name of an operation. The 'getTerm' operation has an input message called 'getTermRequest' and an output message called 'getTermResponse'. The <*`message`*> elements define the parts of each message and the associated data types. In traditional programming terminology, 'glossaryTerms' is a function library, 'getTerm' is a function with 'getTermRequest' as the input parameter and 'getTermResponse' as the return parameter.*

4 Software Efficiency

Computational scientists often find answers to their questions by means of computer simulation to study a physical phenomenon over a period of time. For this, they translate the physical phenomenon into a mathematical description (model). This model can for example be a set of partial differential equations (PDEs). The model is subsequently translated into a family of (possibly nested and hierarchical) numerical methods, each of which consists of one or more numerical algorithms. These algorithms are then translated into a programming language and then further into machine code. The resulting computer program can then be executed on a (super)computer. An additional problem is that computer simulations often require advanced visualization techniques to assist the scientist in understanding the physical behaviour.

The path from modeling the physical phenomenon to machine executable is often a long and time-consuming one. At each step of the chain, there is the danger that a decision has an undesirable (side-)effect that eventually leads to either an inaccurate (useless) or inefficient (slow) computer simulation. In order to avoid such problems, the scientist must acquire a lot of different skills from mathematical modeling to computer program optimization. Since software optimization is the last step in the chain, it often gets less attention. In addition, program optimization is by many researchers considered 'low-level' and not found to be a noble occupation.

However, software optimization and software efficiency often have a dramatic impact on the overall performance (run-time) of the overall application. To write efficient software for solving a computational problem, one must carefully choose and tune the algorithms to keep the number of (symbolic or floating-point) operations as low as possible. Unfortunately, for numerical problems, one must often make a trade-off between a fast algorithm and reduced accuracy. Once the algorithms have been chosen and algorithmic complexity is minimized while keeping a desired level of accuracy, the resulting computer program's structure still must be tuned so that it fully exploits the power of the (super)computer.

Fortunately, there is quite some help available along the described path in the form of *reusable* software. Such software can comprise entire computer simulation packages where the scientist 'only' needs to translate the PDEs into some dedicated high-level language, but it can also be further down the chain in the form of software libraries that contain efficient implementations of algorithms or basic components of algorithms.

A scientist normally prefers to formulate his problem in the highest level computer language available for tackling his problems and leave the generation of the final computer executable as much as possible to *code generators*. Unfortunately, the higher the level of the programming language that the scientist uses, the more levels of software there exist between this language and the hardware, and the more difficult it may get to obtain efficient machine executables.

Things get an extra dimension of complexity for computer simulations that require execution times that are too long to be executed on a (fast) uni-processor computer. *Parallel computer(s)* can then be used with multiple processors sharing the required operations and possibly also the memory (data structures) needed for the simulation. This requires

an arrangement of the computations that allows efficient workload management of the processors and minimal inter-processor communication.

Another scenario is possible where the data sets are so large that they cannot be stored in their entirety on the machine(s) that will perform the simulation. This requires efficient data management, for example to make sure that the input data subsets arrive at (or are streamed to) the machine before the simulation begins.

In the remainder of this section, we will restrict ourselves to the discussion how a general computer architecture looks like and what the basic efficiency problems are at the various levels within this architecture.

To obtain efficient software, one must understand how the computer spends its time while performing the required operations [63]. This in turn requires us to understand how the computer handles data, that is, how computer memories operate. See Figure 9. All modern computers are built with a hierarchical memory design, with a series of very fast, expensive (and therefore small) memory at the top of the hierarchy, down to slow, cheap, and very large memory at the bottom of hierarchy.

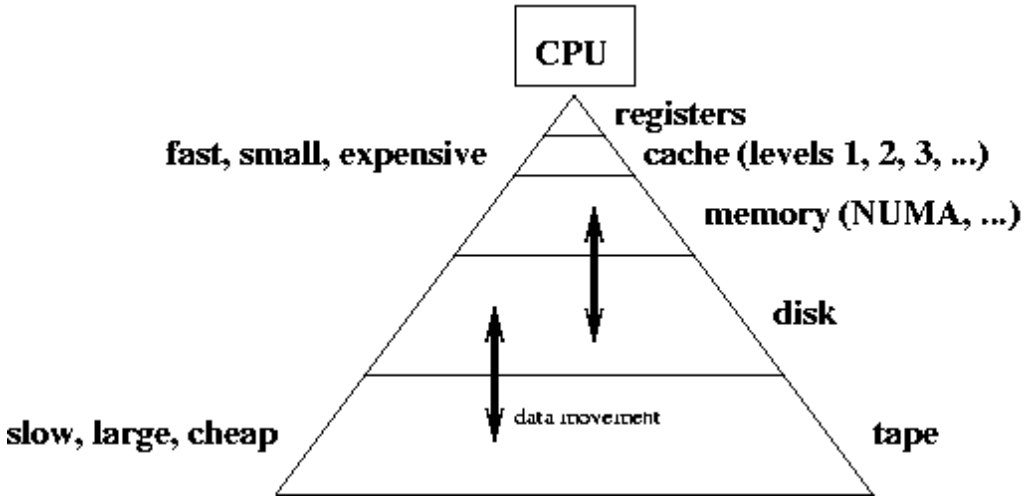

Figure 9: *A general layout of memory hierarchy in a computer.*

Registers typically form the fastest memory and reside close to the Central Processing Unit (CPU, typically on the same chip). Caches are used to hide main memory latency and take advantage of spatial and temporal locality that the codes exhibit. Tape storage is the slowest memory and can be located far away (many kilometers) from the CPU.

Useful arithmetic and symbolic (logical) operations can only be done by the CPU, by using data present in the registers or the (level 1) cache closest to the CPU. Data at one memory level can move to an adjacent level (towards or away from the CPU), for example, between main memory and disk. The speed at which data moves (bandwidth) between levels is high near the top of the hierarchy and orders of magnitude slower near the bottom of the hierarchy. In addition, the speed at which the CPU can perform computations is much higher than the speed at which data is transferred between lower levels in the

hierarchy. Finally, in a multi-user environment (as an HPC system), the levels of memory further away from the CPU are often shared by several users/applications and access times to data that resides in remote levels is typically subject to various external influences and bottlenecks (and therefore more unpredictable).

An ill-designed algorithm may lead to the computer spending most of its time moving data between lower memory levels, while the CPU is waiting for such data to arrive at the register/cache level. In the worst-case, an algorithm will then allow the processor to perform useful operations at a speed that is equal to the speed of the slowest memory that is involved in the operation. Even on modern architectures, the (growing) gap between main memory and CPU performance can significantly limit the execution speed of applications.

A well-designed algorithm may still require significant data movements between lower memory levels, but this movement should be organized such that the CPU at any time is supplied with a stream of data from the cache and the CPU is able to continue to perform useful work most of the time. In order to facilitate this, modern processor and chip designs have various features that enable data to be moved (prefetched) from memory to cache before it is actually needed by the CPU. And also modern compiler technologies offer instructions to perform such prefetch operations. An example of a library that takes into account (timely) data movement between memory levels is given in Section 4.1.

Hence, efficient program execution can only be expected if the codes respect the underlying hierarchical memory design. Unfortunately, today's compilers cannot introduce highly sophisticated 'cache-aware' program transformations and, consequently, much of this optimization effort is still left to the programmer. This is particularly true for numerically intensive codes that occur in almost all science and engineering disciplines; e.g., computational fluid dynamics, computational physics, and mechanical engineering. They are characterized both by a large portion of floating-point operations as well as by the fact that most of their execution time is spent in small computational kernels based on loop nests. Cache performance optimizations can be based on both data layout transformations as well as data access transformations. Such transformations are of particular importance in programs where data (and data structures) are accessed in an irregular or quasi-random manner. Such problems typically arise in the context of the numerical solution of partial differential equations.

Efficient computations become even more complicated on parallel computers. Since a parallel computer has multiple ($p > 1$) processors capable of simultaneously working on the same problem, one may hope to solve any given problem p times faster on such a machine than on a conventional uni-processor machine. However, such perfect parallel efficiency is rarely achieved, even if there are at least p independent subtasks to execute, because of overheads related to coordinating (synchronizing) p processors and the costs of sending data from its (remote) storage location to the processor that needs it. In addition, writing software that enables multiple processors to tackle a problem together is inherently more complex than software for uni-processor machines. It involves strategies for (work)load balancing of the processors, detecting levels of granularity (ratio of communication vs. computation), exploitation of the naturally available parallelism in the computations (data and operation independencies), data coherence, and scalability of

algorithms in time and memory as a function of the number of processors.

Distributed and grid computing (or meta computing) further complicates the topology of the computational infrastructure. Distributed computing allows the seamless coupling of resources and operate these resources as if they were contained in a single machine. Large-scale applications can therefore be executed on such aggregated configuration. In some way, one can consider this by expanding the diagram of Figure 9 with extra levels (of memory). In practice, data may not be stored on the same machine or within the same administrative domain as where the application will execute. This can be regarded as another level of memory in Figure 9 (not depicted) that is obviously located far away from the CPU in terms of access times and latencies. In addition, such memory level is actually remote physically and possibly accessible only through the wide and insecure Internet. This requires a whole new area of protocols to get access to such data. These have to address for example security, resource discovery, and fault detection. These are further discussed in Section 5.

Obviously, such new distributed computing environments have serious implications to the structure and performance of applications that want to exploit the full potential of such distributed topologies. Not only the physical distance but also the many extra protocols (and their overheads) have an impact. Distributed computing technologies should provide efficient tools that enable data to be moved (file staging or streaming) from such remote memory into the local memory or disk before it is needed by the CPU. And subsequently, performance analysis and performance prediction of (coupled) applications in a distributed or grid environment receive considerable attention recently.

Finally, in this section we have assumed that software efficiency is primarily measured in terms of application execution time that is the key optimization parameter. Execution time can be influenced by source code optimization for faster sequential execution (better CPU utilization) and by code parallelization. Some well-known tools for achieving these are listed in the following sections 4.1-4.6. Obviously, besides elapse times, reliability and robustness of software must be measured as well. A fast application is of little value if it (without notice) may fail or produce incorrect results for example caused by programming errors or external influences like hardware faults.

4.1 High performance libraries

A vast amount of tools exist that support the scientist and engineer to develop efficient software. We will not attempt to list them all here, but we illustrate the discussion with an example of a family of *standardized* linear algebra tools that provide both *efficiency* and *reuse* of software.

Many scientific applications that require high performance computer facilities spend a considerable part of their time in solving basic linear algebra problems. In many of these applications, the performance of the linear algebra operations are the main constraint preventing the scientist from modeling more complex problems, which would then more closely match reality. This then dictates an ongoing need for highly efficient routines, since each new generation of increasingly powerful computers have a need for optimized

linear algebra routines. Standard problems in numerical linear algebra include finding the solution to so-called linear systems of equations, least squares problems, eigenvalue problems, and singular value problems (see e.g., [63]).

Since these linear algebra problems play such important role, a lot of effort has been spent in the past two decades in making the solution methods as efficient (and accurate) as possible. This is done in a variety of ways, including the exploitation of the 'sparsity' in the matrix formulations in order to reduce the number of arithmetic operations. Another approach is to identify independencies (partial ordering) in the required operations that allow for the design of parallel solution algorithms in which multiple processors work simultaneously on the overall computation.

Linear algebra is rich in operations which are highly optimizable, in the sense that a highly tuned code may run orders of magnitude faster than a naively coded routine. Unfortunately, such optimizations are typically platform specific, such that an optimization for one computer architecture can actually cause a slow-down on another architecture.

Basic linear algebra operations are so common and careful implementations can have such a dramatic impact on the performance of applications that all major computer manufacturers recognized their importance and provide system-optimized versions of these operations. As a result of a standardization effort in the late 1980s, the Basic Linear Algebra Subroutines (**BLAS**) [64] were defined that define a important set of operations with standard interfaces (name conventions, calling sequences, type definitions, etc.). Reference implementations of the BLAS are freely available from Netlib or the World Wide Web.

Let us examine the BLAS in more detail. Table 2 counts the number of memory references and arithmetic operations by three related BLAS subroutines. For example, the number of memory references needed to implement the `saxpy` operation is $3n + 1$, because we need to read n values of vector x, n values of vector y, and one value from α from (slow) memory to registers, and then write n values of vector y back to memory. There is a total of $2n$ arithmetic (floating-point) operations necessary, i.e., additions and multiplications. The ratio between f and m tells how many operations can be performed per memory reference or in other words, how much useful work we can do compared to moving data. This gives an indication how fast the algorithm potentially can run. In general, algorithms with large f/m ratio are better building blocks for other algorithms.

The BLAS are divided into three levels. The Level 1 BLAS operate on vectors and involve $\mathcal{O}(n)$ operations on $\mathcal{O}(n)$ data. The routines permit efficient implementation on scalar machines, but the ratio of floating-point operations to data movement is too low to achieve effective use of most vector or parallel hardware. The Level 2 BLAS perform matrix-vector operations. These subroutines include matrix-vector multiplication, rank-1 updates of matrices, and solution of triangular systems of equations. Level 2 BLAS typically require $\mathcal{O}(n^2)$ floating-point operations on $\mathcal{O}(n^2)$ data. Although faster than Level 1 BLAS, Level 2 BLAS may still run at a speed well beneath the machines' peak performance. Finally, the Level 3 BLAS perform $\mathcal{O}(n^3)$ floating-point operations on $\mathcal{O}(n^2)$ data and offer the possibility of running at a speed closer to the machine peak performance. The Level 3 BLAS perform matrix-matrix operations, including matrix-matrix multipli-

cation. For a more detailed discussion, we refer to [65]. Table 2 illustrates this hierarchy of BLAS operations: `saxpy` is Level 1 BLAS, `dgemv` is Level 2 BLAS, and `dgemm` is Level 3 BLAS.

BLAS operation	f	m	f/m
`saxpy` : $y = \alpha \cdot x + y$	$2n$	$3n + 1$	$2/3$
`dgemv` : $y = A \cdot x + y$	$2n^2$	$n^2 + 3n$	2
`dgemm` : $C = A \cdot B + C$	$2n^3$	$4n^2$	$n/2$

Table 2: *f is the number of numerical (floating point) operations, m is the number of memory references, for three basic BLAS subroutines. α is a scalar, x and y are vectors of length n and A, B and C are $n \times n$ matrices.*

LAPACK (Linear Algebra PACKage, [54]) provides routines for commonly occuring higher-level linear algebra operations, including the solution of systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. LAPACK routines are written so that as much as possible of the computation is performed by calls to the Level 3 BLAS. Simply stated, the BLAS provides high speed and LAPACK adds functionality, reliability, and accuracy to this. A related library called ScaLAPACK (Scalable LAPACK) is designed for distributed-memory parallel machines.

The **ATLAS** (Automatically Tuned Linear Algebra Software) project is a research effort focusing on applying *empirical* techniques to automatically generate an optimized BLAS library for a given architecture [86]. During the installation process, ATLAS uses code generators in order to provide many different ways of doing a given operation. In addition, ATLAS uses parameterized adaptation that involves parameterizing characteristics which vary from to machine (for example data cache utilization). The ATLAS package includes sophisticated search scripts and timing mechanisms to determine the fastest way to perform the operation for a given architecture. All this allows the package to adapt to a new computer architecture in a matter of hours, and eliminates the need for months or even years of highly-trained professionals' time to produce hand-optimized routines for the new architecture.

Another project that uses empirical techniques is the **FFTW** project to optimize Fast Fourier Transforms [70]. In FFTW, the computation of the transform is accomplished by an *executor* that consists of highly optimized, composable blocks of C code called *codelets*. A codelet is a specialized piece of code that computes part of the transform. The combination of codelets applied by the executor is specified by a special data structure called a *plan*. The plan is determined at runtime, before the computation begins, by a *planner* which uses a dynamic programming algorithm to find a fast composition of codelets. The planner tries to minimize the actual execution time (not the number of floating point operations). The planner measures the run time of many plans and selects the fastest. Plans can also be saved to disk and used at a later time. Codelets are generated automatically by a special-purpose compiler.

PETSc, the Portable, Extensible Toolkit for Scientific Computation, has demonstrated

that the use of modern programming paradigms can ease the development of large-scale scientific application codes in Fortran, C, and C++. The PETSc package is a large suite of data structures and routines which are intended as building blocks for the construction of portable, parallel, efficient application programs for the solution of problems in scientific computation, especially the solution of partial differential equations and related problems.

The PETSc package is predicated on two main principles: data hiding (or data encapsulation) and software layering. This allows several programmers to develop large application code without requiring each programmer to understand both the entire application and the underlying architectural details.

PETSc consists of a variety of libraries each of which manipulates a particular family of objects (for example vectors) and the operations one typically would like to perform on the objects. Some of the PETSc modules deal with index sets, vectors, matrices (generally sparse), distributed arrays (for parallelizing regular grid-based problems), Krylov subspace methods (for the iterative solution schemes), preconditioners, nonlinear solvers, and time stepping schemes (for solving time-dependent nonlinear PDEs).

The libraries are organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem. By using techniques of object-oriented programming, PETSc provides flexibility for users. Each PETSc library consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures. Thus, PETSc provides codes for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables easy comparison and use of different algorithms (for example, to experiment with different Krylov subspace methods, preconditioners, or truncated Newton methods). Hence, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping. The libraries enable customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility, and separates the issues of parallelism from the choice of algorithms. The PETSc infrastructure creates a foundation for building large-scale applications.

The interrelationships among different pieces of PETSc is shown in Figure 10. The PETSc library [58, 56, 57] is built on top of BLAS, LAPACK, SPARSPAK, SPARSEKIT2, etc. and includes interfaces to numerous third-party software.

Besides the above-mentioned numerical libraries that form a core component in the solution of linear algebra components, another programming paradigm provides a much easier-to-use environment but at the cost of some performance. Such environment is for example provided by the commercial system **Matlab** [37]. The Matlab environment provides a simple interactive programming environment where all basic linear algebra entities (matrices, vectors, etc.) are defined and most linear algebra operations are available as built-in functions. This environment make it easy to prototype algorithms and study their behaviour before initiating any time-consuming software implementations.

NetSolve/GridSolve [15] is a client-server system that enables the solution of complex scientific problems remotely. The system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for com-

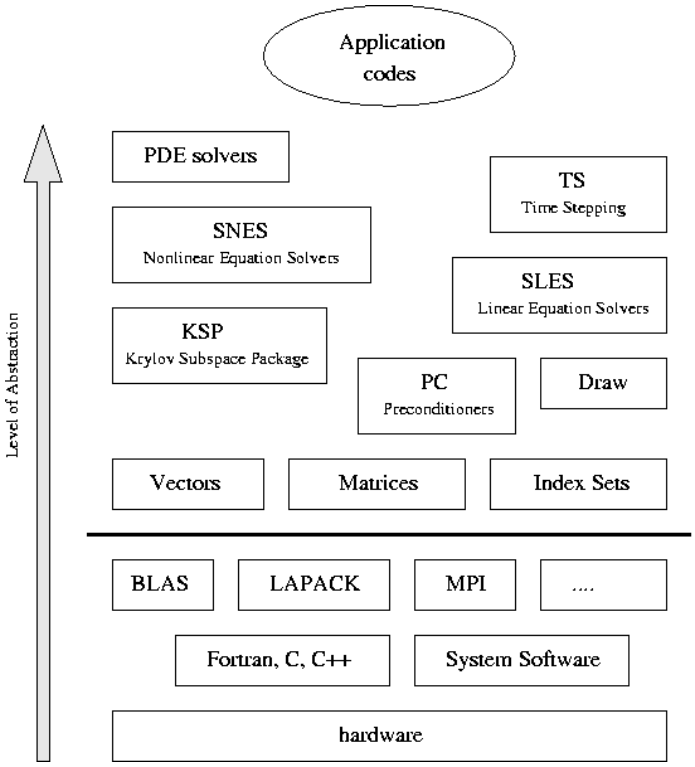

Figure 10: *Organization of the PETSc libraries.*

putational resources on a network, chooses the best one available, solves a problem, and returns the answers to the user. A load-balancing policy is used by the NetSolve system to ensure good performance by enabling the system to use the computational resources available as efficiently as possible.

Finally, we mention that there exist several other popular numerical libraries that provide functionality and performance that is similar to the libraries described in this section. There are hardware vendor specific libraries (e.g., ESSL from IBM), but also third-party libraries (e.g., NAG). Also packages exist that provide functionalities that target specific computational science disciplines.

4.2 MPI

The Message-Passing Interface (MPI, [7]) is a library of functions that is intended for use in application programs to exploit the existence of multiple processors by message-passing. The prime goals for the development of MPI were to provide a standard interface (API) definition for message-passing that allowed source-code portability and implementation for high performance on both massively parallel machines and on workstation clusters.

Various message-passing environments were developed in early 80s. Some were developed for special purpose machines, some were developed for networks of UNIX work-

stations (e.g., P4 from Argonne and PVM from Oak Ridge). The Ohio Supercomputer Center developed a message-passing library called LAM. In 1992, many of the developers agreed to develop a common standard for *message-passing* that would incorporate all the interesting ideas developed so far and build further on them. As a result, the MPI Forum [7] was established as an open group with representatives from many organizations to define and maintain the MPI standard. Amongst them were various hardware vendors like Convex, Cray, IBM, Intel, NEC and Thinking Machines. The first MPI standard, called MPI-1, was completed in May 1994. There was much enthusiasm back then and the first MPI-1 implementations already appeared one year later. The MPI Forum considered corrections and extensions to the MPI-1 standard and released them as the MPI-1.1 and MPI-1.2 versions of the standard. Popular implementations were EPCC's CHIMP, Argonne's MPICH, and Ohio's LAM-MPI [6] (the latter two are still popular today). When the MPI-2 standard was formalized in 1997, much of the initial enthusiasm had evaporated and the first complete implementation of this more advanced standard was released only in November 2002. Many features of MPI-2 are therefore still relatively new to many code developers. Today, most vendors support the MPI-1.1 standard plus some (but not all) of the functionality of the MPI-2 standard.

MPI provides a rich set of collective communication functions, virtual topologies, hooks for debugging and profiling, blocking and non-blocking sends and receives, and support for heterogeneous networks. MPI-2 adds numerous clarifications, specification of portable MPI process startup, new data type manipulation functions and new predefined types, support for dynamic process creation and management, support for one-sided communications, i.e., for the ability to write data directly into other processes' memories, portable high-performance I/O in the form of MPI-IO, and support for C++ and Fortran 90/95.

The library **PACX-MPI** (PArallel Computer eXtension) [4] enables scientists and engineers to run MPI-conforming parallel applications on a computational grid, such as a cluster of (parallel) high-performance computers connected through high-speed networks or even the Internet. Communication between MPI processes internal to a node in the grid is done with the native (vendor) MPI, while communication to other nodes is done via the connecting network using communication daemons, one on each of the nodes in the grid. PACX-MPI supports the full MPI-1.2 and parts of the MPI-2 standard.

MPICH-G2 is a grid-enabled implementation of the MPI-1.1 standard. See Section 5.8.2.

4.3 OpenMP

OpenMP is an industry standard for shared-memory programming that allows parallel programs to be made portable across a wide range of shared-memory platforms [41]. OpenMP aims to be a portable, scalable model that gives parallel programmers a simple and flexible interface for developing parallel applications for all architectures, including UNIX platforms and Windows NT platforms.

The definition of the OpenMP API is governed by the OpenMP Architecture Review Board that includes vendors like HP, IBM, Intel, SGI, and Sun. The standard has been

endorsed by software and application vendors like Absoft, the Portland Group, ANSYS, Fluent, and NAG. The API provides definitions for Fortran, C, and C++ bindings. The first Fortran specification was released late 1997 and the first C and C++ specifications came late 1998. The OpenMP API is now a widely accepted standard for high-level shared-memory programming.

A summary of some of the characteristics of OpenMP and MPI is given in Table 3.

MPI	**OpenMP**
characteristics	**characteristics**
- distributed memory model	- shared-memory model
- explicit communication (subroutine calls)	- implicit communication (threads)
strengths	**strengths**
- data placement automatic	- relatively easy to use
- implicit synchronization (subroutine calls)	- better use of shared-memory environment
- provides portability across distributed and shared memory machines	- both course and fine grain parallelism are effective
weaknesses	**weaknesses**
- development and debugging time consuming	- limited to shared-memory machines
- communication overhead	- data placement may cause contention in NUMA systems
- large message granularity often required to offset latency	
- global operations can be expensive	

Table 3: *Summary of OpenMP and MPI characteristics.*

4.4 Mixed programming

The advent of *clusters of SMP systems* has increased the importance of developing applications that are portable and efficient on such systems. A combination of shared-memory and message-passing parallelization paradigms within the same application (mixed mode programming) may provide a more efficient parallelization strategy than pure MPI.

Using OpenMP over a cluster of SMPs is in general not possible since OpenMP assumes that the underlying hardware supports a single address space. Using MPI across the whole machine is in principle possible but this may incur unnecessary overheads within each SMP node and not exploit the advantages of the single address space. The idea is therefore to use OpenMP on the processors within an SMP node and MPI between the SMP nodes. This creates a hierarchy (two levels) of parallelism.

The advantages are potentially manifold. In a pure MPI implementation that is not cluster-aware, we often can encounter collective operations where all MPI processors send or receive messages at the same time. This may cause contention for network ports or other hardware resoures. It may therefore be better to reduce the number of messages, especially across the relatively slow interconnect between the nodes in the cluster. In mixed mode programming, a good implementation of collective communications minimizes inter-node message traffic, by first (OpenMP) reduction within the nodes and then (MPI) reduction across the nodes. Replication of data may also be reduced in mixed

mode, since one typically needs only one copy of the data per SMP system node and not one copy per processor (as in a pure MPI implementation).

Code development in a mixed programming model is typically harder than for MPI code, and much harder than for OpenMP code. One has to cope with both OpenMP and MPI problematics at the same time. An additional complication is for example to ensure thread-safety of the MPI part of the application (to not corrupt the OpenMP part). In addition, current batch systems have varying amounts of support for mixed mode programs.

The trade-off between number of threads (for the OpenMP part) and number of processors (for the MPI part) is a user responsibility. An advantage with respect to code portability is that the code should function correctly (with conditional compilation) as a stand-alone MPI code (on a distributed-memory machine with single-cpu nodes) and as a stand-alone OpenMP code (on one single SMP node).

Finally, we note that the concept of two-level parallelism in mixed mode programming has similarities with a distributed/grid computing environment where a set of geographically distributed (potentially parallel) machines are connected via the slow Internet. The communication latencies between remote machines will be considerably higher (and bandwidths lower) than those within a node of the grid, and more effort will be needed to further minimize data traffic between remote machines.

4.5 Application performance optimization in grid environments

From a resource provider's perspective, the performance objective for a grid infrastructure is to achieve maximum utilization of the various resources within the grid to achieve maximum throughput [76]. The resources may include but are not limited to CPU cycles, memory, disk space, federated databases, or application processing. Workload balancing and preemptive scheduling may be used to achieve this objective. Applications may be allowed to take advantage of multiple resources by dividing the grid into smaller instances to have the work distributed throughout the grid. The goal is to take advantage of the grid as a whole to improve the application performance. The workload management can make sure that all resources within the grid are actively servicing jobs or requests within the grid.

From a service requester's perspective, the turnaround time of an application running on the grid is important. This can vary depending on the type of grid resource used and the resource provider's quality-of-service agreement including job scheduling priorities. There are several factors that impact the turnaround time of applications run on the grid resources. These include network topology, network speed and latencies, datasets topology and data access delays, application topology (e.g., distribution of concurrent jobs on the grid), proximity of data to the application, contention for resource, and reliability and fault-tolerance in the grid resource and network (e.g., checkpoint and restart capabilities). Optimum application performance is usually achieved by proper tuning and optimization on a particular operating system and hardware configuration. This poses possible issues if an application is simply loaded on a new grid host in a mixed platform environment.

Several functionalities exist that have proven successful in distributed and grid environments to enhance performance and providing means for adequate data management. We mention some of these in the remainder of this section.

Data replication is a well-known technique in distributed and grid environments to optimize access times to data (data locality) and improve fault tolerance. Replication is a mechanism to make one or more copies of data at the source site, transfer the copies to the destination site(s), and keep track of all copies of the data until they cease to exist. Data replication can be done at various levels of granularity, for example at the file level or object level (typically a collection of files). Replication of data is a standard tool in large-scale scientific collaborations with distributed user communities and distributed data sites.

A fundamental building block in data replication is the replica catalogue. This catalogue addresses the need to keep track of multiple physical copies of a data set. This can be done by maintaining a mapping from logical file names to physical locations. Operations on this catalogue are for example creation and deletion, location, listing of contents, and listing of all physical locations of a data set.

Another technique for efficient access to data in the grid is data **caching** to provide temporary storage on the execution nodes and avoid network access during job execution. For safety and performance reasons, data **mirroring** across a set of nodes provides several access points for jobs that need to process this data. Mirroring consists of being able to synchronize data manipulations that occur at different locations. The mirroring can be synchronous or asynchronous (mirroring happens at certain time intervals).

A form of data management that is related to data replication is **file staging**. When a user submits a job to a remote execution host, the job executable must be copied to this host (assuming it's not already cached on that host) as well as any input file that corresponds to that particular job. This transfer must complete before the job can be executed. This process is referred to as file staging. When the job completes, the output files must be retrieved and copied to the location that the user specified in the job definition file.

Secure and efficient data movement by means of standard data transfer and access protocols is a necessary feature in distributed and grid environments. The File Transfer Protocol (FTP) is the most commonly used protocol for data transfer on the Internet. Several protocols have been developed that extend the standard FTP protocol, providing a superset of features that include for example public-key based support, parallel data transfer (host-to-host using multiple TCP streams), striped data transfer (m hosts to n hosts), partial file transfer, reliable and restartable data transfer, and monitoring on-going transfer (for performance).

4.6 Tools to monitor software efficiency

Optimizing the performance of parallel applications on homogeneous parallel systems remains a challenging research problem, with open questions related to analysis of for example data-parallel codes, characterization of I/O behaviour, and management of data and

memory locality. Heterogeneous grid applications pose even more daunting performance problems.

While performance measurements reveal the most promising among a variety of optimization alternatives for a given code fragment on a certain architecture, they do not reveal the reasons why this is the most promising alternative. Additional simulations can give a wealth of additional information on the code's access behaviour, and therefore allow for an easier explanation of unexpected effects by starting from an ideal model. So-called **profiling** techniques have been developed for this purpose as well as techniques that are based on modern performance hardware counters.

Profiling data has to be presented in a meaningful and summarized fashion in order to facilitate the recognition of performance bottlenecks in the code. Thus, visualization techniques and appropriate tools play an important role in application profiling and performance analysis.

Profiling a grid application requires performance measurement at many levels [67] (Chapter 15). Performance data must be correlated across levels and presented in ways that match the user's semantic model (i.e., application, network, or storage-centric). Effective grid performance tools should provide high-level performance data that can be correlated with application source code and they should support interactive understanding of lower semantic levels, allowing users to explore both the proximate and underlying causes of poor performance.

A priori prediction of the location of performance bottlenecks or their causes is often not possible. Application performance is often a combination of issues related to processor and network hardware, protocols and run-time systems, resource management policies, compiler transformations, and application behaviour. This makes performance optimization often a posteriori process that relies on performance measurement and correlation at multiple system levels.

An added complication in wide area computational grids is that performance measurements may not repeatable. Most often performance analysis is an iterative process in which applications and systems are assumed largely static and have reproducible performance behaviour. Grid environments are inherently more dynamic in nature and neither available resources nor application behaviour may be reproducible. A more practical way to deal with such dynamic environments is to instrument the application with real-time performance visualization and adaptive control of application behaviour and resource policies.

In the remainder of this section, we mention a few tools that are of interest for tuning, optimization, and benchmarking of software on parallel and grid architectures. *Performance analysis tools* are intended to provide insight in potential performance bottlenecks.

PAPI [16], the Performance API, specifies a standard application programming interface for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count *events*, occurrences of specific signals related to the processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code

to the underlying architecture. This correlation has a variety of uses in performance analysis including hand tuning, compiler optimization, debugging, benchmarking, monitoring and performance modeling.

A popular and portable parallel programming profiling tool is **Vampir** (Visualization and Analysis of MPI Programs) [43]. In its simplest form, the tool visualizes the run-time behaviour of an MPI code, by generating a profile of all MPI subroutine calls and messages. Useful basic features are timelines, where message traffic is visualized between processors over time, and activity pie-charts that depict the relative amounts of time each processor spends in MPI calls compared with the rest of the computations. More sophisticated analysis of a program is allowed, but this typically requires manual changes (additions) to the source code. A grid-enabled version of Vampir (MetaVampir) was developed as part of the DAMIEN project (see Section 5.8.3).

Paraver, developed by ENACTS partner CEPBA, is another very useful tool for the profiling and run-time analyis of parallel programs [21]. The tool includes much of the Vampir functionality but has several other useful features. A strong feature of this tool is that it allows the detailed profiling of OpenMP programs, MPI programs, mixed OpenMP/MPI programs, Additionally, it can report performance through hardware counters and report system activity.

Finally, we mention that there exist other tools. **SvPablo** [18], for example, is a portable performance instrumentation and analysis toolkit.

5 Grid and Distributed Environments

An increasingly popular method to solve various types of scientific and business problems is to harness a set of (geographically) distributed computational resources to exploit their aggregate power. This mode of using computers is usually known as **distributed computing**. Distributed computing was introduced in the 1970s when computers were first linked by networks, with the ARPAnet being the first of these. Distributed computing scaled to a global level with the maturing of the Internet in the nineties. Several projects proved that the concept works extremely well even better than many experts had anticipated. The SETI@home project is probably the most successful distributed computing project in history. SETI stands for ”Search for Extraterrestrial Intelligence”, and the aim is to search for radio signal fluctuations that may indicate a sign of intelligent life from space. The signals are collected by the Arecibo Radio Telescope in Puerto Rico. The project originally received far more terabytes of data every day than its assigned computers could process. So the project directors turned to volunteers, inviting individuals to download the SETI@home software to donate the idle processing time on their computers to the project. The project originated at the University of California at Berkeley and was launched in May 1999. Since then, more than two million people have installed the SETI@home software. The project conclusively proved that distributed computing could accelerate computing project results while managing project costs.

By the late 1990s, the Internet was adequately equipped to move vast amounts of data between HPC systems, and efforts were initiated to link together the (national) infrastructures of high performance computational and data storage resources together into a general computational utility ’grid’.

Grid computing is a form of distributed computing that involves coordinating and sharing computing, application, data, storage, or network resources across dynamic and geographically dispersed organizations. The Grid refers to an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations. Grid applications often involve large amounts of data and/or computing and often require secure resource sharing across organizational boundaries, and are thus not easily handled by today’s Internet and Web infrastructures.

Grid technologies promise to change the way organizations tackle complex computational problems. However, the vision of large scale resource sharing is not yet a reality in many areas. Grid computing is an evolving area of computing, where standards and technology are still being developed to enable this new paradigm.

The term ”Grid computing” suggests a paradigm that is similar to an electric power grid: a variety of resources contribute power into a shared ”pool” for many consumers to access on an as-needed basis. This ideal is still a few years off. However, there are key efforts underway to define standards that would allow the easy pooling and sharing of all computing resources, including cycles, storage, and data in a way that can promote mass adoption of grid computing.

Two papers that provide overviews of Grid computing are ’Anatomy of the Grid’ [69],

which defines Grid computing, proposes a Grid architecture, and discusses relationships between Grid technologies and other contemporary technologies; and 'Physiology of the Grid' [68], which describes how Grid mechanisms can implement a service-oriented architecture, explains how Grid functionality can be incorporated into a Web Services framework, and illustrates how this architecture can be applied within commercial computing as a basis for distributed system integration. An older but more extensive reference is the book 'The Grid: Blueprint for a New Computing Infrastructure' [67] edited by Foster and Kesselman (1999).

Major issues concerned with grid computing are resource discovery, resource allocation and resource management, fault-tolerance, security and access control (resource and data ownership), scalability, flexibility, and performance. Various organizations have developed mechanisms that attempt to address these issues, each with their own perspectives of how to resolve them.

The massive growth of the Internet has given rise to a wide variety of protocols, libraries, and applications allowing various resources to talk to one another. Developing distributed applications has become increasingly difficult when faced with the growing number of technologies from which to choose. This situation is further complicated by the growing number of technical issues that need to be addressed when attempting to develop an application.

The Global Grid Forum (GGF) [28] is a community-initiated forum of thousands of individuals from industry and research leading the global standardization effort for grid computing. GGF's primary objectives are to *promote and support the development, deployment, and implementation of Grid technologies and applications via the creation and documentation of "best practices" - technical specifications, user experiences, and implementation guidelines*. GGF adopted an open process for development of Agreements and Specifications. GFF uses a 'Recommendations' process modelled after the internet standards process of the IETF, and it uses persistent, reviewed document series to document the work of the community and to facilitate the development of Grid standards. GGF aims to be a forum for information and exchange and collaboration to compare experiences and approaches, to span vertical layers including people who research, build, and use grids and grid component software.

GGF spans a wide variety of technology 'areas' in which well-defined projects are pursued. The work of GGF is performed within various working groups (WG) and research groups (RG). A working group is generally focused on a very specific technology or issue with the intention to develop one or more specific documents aimed generally at providing specifications, guidelines or recommendations. A research group is often longer-term focused, intending to explore an area where it may be premature to develop specifications. The GGF technical areas are (February 2004):

- APME : Applications and Programming (2 WGs and 10 RGs)
- ARCH : Architecture (4 WGs and 4 RGs)
- DATA : Data (6 WGs and 2 RGs)
- ISP : Information Systems and Performance (4 WGs and 2 RGs)

- P2P : Peer-to-Peer (2 RGs)
- SEC : Grid Security (5 WGs and 2 RGs)
- SRM : Scheduling and Resource Management (7 WGs and 1 RGs)

One of the challenges that came with the grid is the use of the Web for exchange of data between administrative domains (see also Section 3.4). The Web is a collaborative medium for browsing and viewing, but most importantly for interacting and information exchange. To promote a more collaborative environment, the web must be built so that it can be *trusted*. It must offer confidentiality. This drives much of standardization work around digital signatures, annotation mechanisms, group authoring, versioning, etc.

Challenges in the design of distributed computing architectures are in the following areas:

• **Security.** The first technical challenge encountered in harnessing multiple resources consists of authentication and authorization. In this case, it is assumed that authorization – permission to access a system – is determined by the systems administrator and subject to local site policy. Authentication, however, is the means by which one establishes identity to the remote site. Generally, sites have their own policies regarding how one can authenticate to a resource. Such policies might include using security services such as SSH (Secure Shell) or Kerberos or perhaps even allowing plain-text passwords (i.e., no secure authentication required).

The decision to adopt a particular distributed computing framework may be highly dependent on the security requirements of the site resources one plans on using.

• **Accessing Remote Resources.** Once authenticated, the user will be launching an application. Based on the application, and possibly on other parameters provided by the user, the next step is to identify the available and appropriate resources to utilize within the grid. A *resource broker* takes incoming job requests and allocates system resources in an efficient and fair manner. In typical client/server applications, the server is responsible for spawning processes and allocating machine resources such as memory, processors, and bandwidth. One may want to use several resources simultaneously which represents a difficult scheduling problem because one needs to coordinate resource usage between multiple sites, each operating under its own policy. The level of complexity of a resource broker largely depends on the applications. For example, a large-scale distributed simulation may require resource brokers at each site to coordinate the allocation of multiple processors at a given time (called advance reservation), as well as providing for a certain amount of network bandwidth for the visualization.

• **Resource Discovery.** One of the central problems facing distributed application developers is keeping track of the resources being used and allowing for dynamic resource discovery. Rather than specifying the names of the machines one wants to utilize, ideally one would like to specify the application requirements (computing power, network performance, or installed software) that one needs to run. An information or naming system could compare requirements with its dynamic database of resources and find the machines most appropriate for one's needs.

• **Scheduler.** Once the resources have been identified, the next step is to schedule the individual jobs to run on them. If a specific resource is needed or different jobs need to run

concurrently within the application (for example with inter-process communication), then a job scheduler is needed to coordinate the execution of the jobs. A grid environment can have multiple levels of schedulers. A compute cluster can have its own scheduler to help manage the compute nodes it contains. This cluster is one node in the grid environment. A higher level scheduler (sometimes called a meta scheduler) might then be used to schedule work to be done on the various nodes in the grid. Scheduling jobs for load balancing is an important function in the grid.

• **Data Management.** In a distributed environment, data that must be moved or made accessible to the grid nodes where an application will execute. Therefore, secure and reliable methods must exist for moving files and data to various nodes within the grid. Secure communication and encryption of data across the network must be ensured.

• **Fault Detection / Tolerance.** An ever-changing (dynamic) grid environment requires some way of knowing which resources may be down and which one can use. Providing basic fault detection can pave the way for the application developers to incorporate dynamic reconfiguration in a distributed application or checkpointing/restarting in the case where one of the resources required in a computation goes down.

At the time of writing this document, there are hundreds of grid related (or grid-like) projects underway world-wide [28]. These vary from small scale specialized projects (e.g., AccessGrid, NetSolve) to large scale specialized projects (e.g., SETI@HOME) and from small scale general projects (e.g., PBS, Condor, IBM's LoadLeveler, and LSF) to large scale general projects (e.g., Legion, Globus, UNICORE).

5.1 Application-enablement considerations

When designing grid-enabled applications, a number of concerns must be taken into consideration. For example, the application designer needs to be aware of the security administration and implications. If the grid is a heterogeneous multi-platform environment, the underlying security mechanisms of the various platforms may not always be consistent. For instance, the security mechanisms for reading, writing, and execution on UNIX or Linux-based systems differ from those on a Microsoft environment. If such security is a concern, the application developer should take into account the possible platforms on which the application may execute (by informing the resource broker).

There are also several considerations for application architecture, design, and deployment related to resource management. The application must ensure that the appropriate target resources are used. This requires for example that the application accurately specifies the required environment (operating system, processor, speed, memory, and so on). Obviously, the less specific these requirements are, the better the chance that the available resources are available. If an application includes multiple jobs, their interdependencies must be well understood. Also file staging of input and output data must be dealt with in the resource specification.

Application enablement considerations also consist with respect to data management. Data management in a grid environment is often concerned with optimal use of the limited

storage space, networking bandwidth, and computing resources. Data issues that need to be considered are for example [76]:

- **Dataset sizes.** For large datasets, it is not practical and may be impossible to move the data to the system where the job will actually run. Using data replication or otherwise copying a subset of the entire dataset to the target system may provide a solution.
- **Geographically distributed users, data, computing and storage resources.** If the grid is geographically distributed with limited network connection speeds, one must take into account design considerations around slow or limited data access.
- **Data transfer over wide-area networks.** One must take into account the security, reliability, and performance issues when moving data across the Internet or other wide area network. The application must have built-in logic to handle situations when the data access may be slow or prevented.
- **Scheduling of data transfers.** Data transfers must be scheduled so that the data is at the appropriate location at the time that it is needed.
- **Data replica selection.** In more advanced situations where multiple copies of data exist in the grid, one may want to include features in the application that handle selecting the appropriate *replica*, that is, select a dataset that contains the data that is needed, while also providing the performance requirements that may exist.

Application enablement considerations exist with respect to the brokering and scheduling mechanisms. It is important to understand how resources are discovered and allocated. It may be up to the application to identify its resource requirements to the broker so that the broker can ensure that the proper and appropriate resources are allocated to the application. In an environment with multiple schedulers, such as with meta schedulers, the complexities of coordinating concurrent jobs, or ensuring certain jobs execute at a specific time, can become a complicated matter.

Inter-process communication between jobs on the grid typically increases complexity. When communicating over wide-area networks, applications may encounter network congestion that severely impacts the performance of the application. However, in many applications, it often cannot be avoided. Understanding the inter-process communication mechanisms that are available and minimizing the effect of failed or slowed communications can help ensure the overall efficiency and reliability of the application.

Finally, ease-of-use and user requirements are main criteria for the success of an application. Once the grid application has been developed, it will be accessed through a *portal*. A grid portal may be Web page interface that provides easy access to the application while hiding many of the complex issues listed above. The portal provides user authentication, job submission, job monitoring, and results of the job. As with any user interface, there are trade-offs between ease-of-use and the ability for advanced usage. Also job steering mechanisms like suspend/resume and cancelling of a job or more advanced interactive access mechanisms could be required by the end-user.

5.2 Criteria for grid applications

Not all applications lend themselves to successful or cost-effective deployment on a grid. Several attributes of an application or its requirements may inhibit the application from being a good candidate for a grid environment. In [76], a grid qualification scheme with 35 criteria is presented to evaluate whether an application is suitable for grid enabling and deployment. Also data influences many aspects of application design and deployment and determines whether a planned grid application can provide the expected benefits over any other data solution. Some applications that are difficult, require extensive work effort, or cannot be grid-enabled are listed below.

- Applications with high inter-dependencies between the jobs, which expose complex job flow management and cause high rates of inter-process communication. Applications with high inter-process communication between concurrent jobs without high speed interconnect.
- Applications that depend on data provided real-time by external (uncontrolled) sources in an environment with strict job scheduling policies.
- Applications with strong system environment dependencies for the jobs (specific operating system, libraries, etc.).
- Applications with jobs that use unsupported network protocols may be prohibited to perform their tasks due to firewall rules.

The above list is incomplete and depends on the exact specifications of the available resources and infrastructure. Issues like temporary data storage, data type conformity across nodes within the network, the number of software licenses available, or the degree of complexity of the job flow can be solved, but have to be addressed up front in order to create a grid application with reasonable performance characteristics.

The following criteria are related to data when deciding whether an application is a good candidate for a grid [76].

- **Individual/separated data per job.** Ideally, each single job in the application receives a well-defined set of input data and has an equally well-defined place to leave its intermediate and final computing results. Interdependencies between jobs in a grid application that require frequent synchronization of data should be avoided.
- **Shared data access.** One must consider whether there are any restrictions on the access of databases, files, or other data stores for either read or write. Potential data access conflicts in which jobs try to update the same data records at the same time should obviously be avoided.
- **Locking.** Read/write locking is as important in a grid context as it is in any other concurrency situation with databases and other data sources. Read only locks for keeping accuracy of data sets should be used where possible.
- **Temporary data spaces.** Temporary data spaces are often needed and the forms and amount of temporary data space should be considered.

- **Data volume** The amount of input, intermediate, and output data within a grid application is important relative to the total storage capacity in the grid. A large data volume may require that the data sets are split. A suitable case for a grid-enabled application is when the application can be split into several jobs that are able to work independently on a disjunct subset of the input data and produces its own output data.
- **Time-sensitive data.** Data may have a limited lifetime, meaning its values are only valid during a defined time period. The jobs in a grid application have to reflect this in order to operate with valid data when executing. Especially when techniques like data replication and data caching is used, the validity of data needs to be assured.
- **Data encryption.** The subjects to consider are authentication, access control, data integrity, data confidentiality, and key management. For a grid application this can for example be addressed via a Public Key Infrastructure (PKI) or via the Grid Security Infrastructure (GSI) as supported by Globus [29].

Grid environments enable several new classes of applications or give a significant boost to certain existing classes of applications. These include [29]:

- **Smart instruments**: Advanced scientific instruments, such as electron microscopes, particle accelerators, and wind tunnels, coupled with remote supercomputers, users, and databases, to enable interactive rather than batch use, online comparisons with previous runs, and collaborative data analysis.
- **Teraflop desktops**: Chemical modeling, symbolic algebra, and other packages that transfer computationally intensive operations to more capable remote resources.
- **Collaborative engineering** (teleimmersion): High-bandwidth access to shared virtual spaces that support interactive manipulation of shared datasets and steering of sophisticated simulations, for collaborative design of complex systems.
- **Distributed supercomputing**: Ultra-large virtual supercomputers constructed to solve problems too large to fit on any single computer. Ultimately, the grid links massive computational resources at high performance computing centres and experimental facilities.
- **Parameter studies**: Rapid, large-scale parametric studies, in which a single program is run many times in order to explore a multidimensional parameter space.

Ultimately, the grid aims at moving the current bottleneck out of the hardware and back to the human mind.

For a listing of projects related to actual grid testbeds and applications, we refer to the Sectoral Report on *Grid Enabling Technologies* that was released earlier in the ENACTS project [50].

The following sections highlight the design architecture of some of the more popular choices of middleware such as CORBA, Globus, Legion, and OGSA. Some advantages and disadvantages are discussed for each design architecture.

5.3 TCP/IP

Although TCP/IP does not provide a framework to address the various challenges raised in developing applications on the Grid, it is discussed briefly because it is used by most, if not all, of the other distributed computing frameworks.

TCP/IP is a family of network protocols that has gained general acceptance over nearly twenty years. It underpins most of the tools used everyday to browse web pages, download files, etc. The development of TCP/IP created the notion of *layered* protocols. The three most general protocols are IP, UDP, and TCP.

The Internet Protocol (IP) is the most basic of the three. IP provides the minimum requirements to route a message and move it from one machine to another over a network interface.

The User Datagram Protocol (UDP) provides is a basic transport protocol built on top of IP and provides more functionality. UDP does not guarantee that messages will reach their final destinations, and therefore UDP is used for certain applications like streaming of audio and video where (accidental) packet loss recovery is not as important.

The Transmission Control Protocol (TCP) is also layered on top of IP and provides more reliability by ensuring that packets that are lost are retransmitted and reach their destination in the order they were sent. TCP addresses transport issues like handshaking mechanisms to establish a connection between two machines, flow control capabilities, congestion control, message polling (how long to wait for an incoming packet), and retransmission of lost or corrupted data.

TCP/IP specifies a sockets API that lets developers write their own network code to transmit data using the TCP or UDP protocol. Servers can be designed that process requests from clients and return updated information.

Unfortunately, sockets programming is too low-level for developers not intimately familiar with the fine details of TCP/IP and network programming. For this reason, developers typically rely on distributed computing frameworks, or middleware, discussed below to more effectively enable application development.

5.4 Middleware

Middleware provides a software framework that attempts to overcome many of the limitations of using TCP/IP for communication, as well as support advanced capabilities and services for distributed applications. Middleware provides advanced architecture that sits on top of TCP/IP and hides various communications issues and enhanced functionality. It also provides:

- more robust communications libraries that provide the ability for clients and servers to communicate via remote method invocations or remote procedure calls. This allows clients and servers to specify remote tasks besides just sending data.
- advanced architectures for addressing other technical challenges that face the design

and development of a distributed application. The most common issues are security, resource management, resource discovery, and fault tolerance in a heterogeneous environment (described earlier).

The object-oriented approach of CORBA was already addressed briefly in Section 3.3. CORBA defines an Interface Definition Language (IDL) and an API that enable client/server interaction within a particular Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships between objects. Client and server applications (objects) are specified using IDL and then compiled into the stub (or skeleton) interfaces (the actual code to describe the object's methods). IDL makes it possible for both Legion and Globus to *interoperate* with CORBA. Developers can therefore take advantage of the various services Legion and Globus offer while still working within the CORBA model.

5.5 Legion/Avaki

Legion [23] shares with CORBA its object-based approach. Legion is an object-based metasystem designed to connect various computers and networks and provide the illusion of a single system. Unlike CORBA, Legion is also intended for high-performance computing users.

Legion is a monolithic distributed object-oriented grid computing environment that is extensible both by developers and users. Legion lives on top of a user's operating system, negotiating between that computer's resources and whichever resources or applications are required. Legion handles resources scheduling and security issues. A user-controlled naming system called *context space* is used, so that users can easily monitor and use objects. A Legion resource maintains Core Objects, such as the host and vault objects used to represent processors and persistent storage respectively. Using Legion's IDL, code can be written in any of the supported languages Java, C, C++, Fortran, and the CORBA IDL.

Legion is currently being commercialized by Avaki [20].

5.6 Globus

Globus [29] is a community-based, open architecture, open source set of services and software libraries that addresses issues of security, information discovery, resource management, data management, communication, fault detection and portability. It is used at hundreds of sites and by dozens of major projects worldwide. The design goals of the Globus project are to enable the construction of computational grids that provide pervasive, dependable and persistent access to high performance distributed computing resources, regardless of the geographical distribution of resources and users.

In contrast to the more monolithic architecture of Legion, the Globus architecture follows a strictly layered and modular structure termed the Globus Toolkit. The Globus Toolkit

provides a framework for the use of multiple technology components to provide progressive layers of abstraction and extensibility to both the Globus system and to Globus users. Globus Toolkit version 2 (GT2), like Legion, focuses on the requirements of the high-performance computing spectrum. The toolkit consists of a number of components including:

- Grid Resource Allocation Management (GRAM) for secure, reliable, service creation and management,
- Meta Directory Service (MDS) for information monitoring and discovery,
- Data Management for high-performance, secure, reliable data transfer and intelligent data replication,
- Grid Security Infrastructure (GSI) for sign on, delegation, and credential mapping.

The modules act as a middle layer between resources and services, and Globus applications that utilize the toolkit. The toolkit represents an environment abstraction that provides a solution for applications to utilize resources made available by sites on the Grid. Globus can easily be extended to support a wide variety of technologies. Conversely, Globus modules an also be used separately from Globus to provide customized services and for grid-enabling existing applications.

Systems with Globus deployed are, like Legion, capable of secure authentication and job scheduling. Unlike Legion or CORBA that define an object-oriented architecture, the Globus framework provides C libraries for addressing individual areas like security, remote job submission, file staging, and resource discovery. In Globus, applications can take advantage of (only) the libraries they need without adopting an entire architecture to create a distributed application. Globus also offers a number of higher-level tools to allow users to stage files, execute programs remotely, and inquire about other resources.

The Globus Toolkit includes a set of service components collectively referred to as the Globus Resource Allocation Manager (GRAM). GRAM facilitates the use of remote systems by providing a single standard interface for requesting and using remote system resources for the execution of jobs. The most common use (and the best supported use) of GRAM is remote job submission, monitoring, and control. This is typically used to support distributed computing applications. GRAM is the lowest level of the Globus resource management architecture. GRAM does not provide scheduling or resource brokering capabilities and does not provide accounting and billing features, but it provides some basic mechanisms to run existing applications unmodified through the Grid.

Job requests are prepared using the Resource Specification Language (RSL). RSL is a structured language by which resource requirements and parameters can be outlined by a user. All job submission requests are described in an RSL string that includes information such as the executable file, its parameters, information about redirection of stdin, stdout, and stderr, and so on.

There are three commands available for the user to submit a job (clients for GRAM): globus-job-run, globus-job-submit, globusrun. Globusrun is the executable which gives

access to the full power of the RSL. Globus-job-run and globus-job-submit are both shell script wrappers around globusrun.

When a job is submitted, a request is sent to the gatekeeper of the remote computer. The gatekeeper handles the request and creates a job manager for the job. The job manager starts and monitors the remote program, communicating state changes back to the user on the local machine. When the remote application terminates, normally or by failing, the job manager terminates as well.

Below is an example RSL job request for execution of the BLAST program in an AIX environment.

```
&
(* test run: if "yes", only submits RSL without actual job start *)
(dryRun=no)
(* some environment variables, to be used by the job *)
(environment=(AIXTHREAD_SCOPE "S") (MALLOCMULTIHEAP "true") (SPINLOOPTIME "500"))
(* the main executable file *)
(executable=/usr/local/blast/blastall)
(* the arguments for the executable above *)
(arguments=-p blastp -d /scratch/blast-run/nr -b 5 -v 5)
(* file to be staged in before the execution, using globus-gass for this example *)
(stdin = https://tre.ii.uib.no:37929/home/csaba/seq1)
(* file to be staged out after the execution, using gridftp in this case *)
(stdout= gsiftp://hydra.ii.uib.no/ndgf/blastout)
(* user-specified job name *)
(jobName=blast-test)
(* specific project under which the job will run*)
(project=parallab)
(* maximal CPU time required for the job, minutes for PBS*)
(CpuTime=64)
(* memory required for the job, Mbytes *)
(maxMemory=200)
(* time required by the job*)
(maxTime=300)
```

The Globus Toolkit provides two methods for accessing distributed data. As a data transfer protocol, it provides the Grid File Transfer Protocol (GridFTP), which is a common protocol independent of the underlying architectures and provides high-speed data transfer functionality. It includes features such as GSI security for both control and data channels, parallel transfers, striped transfers, partial file transfers, and other (third-party) transfer functionality. The other method, Globus Access to Secondary Storage (GASS), allows applications to use standard file I/O interfaces (open, read/write, close) for distributed/remote access.

The successor to GT2, Globus Toolkit 3 (GT3) was released in 2003 and is OGSA-compliant (see next section) but backward compatible by supporting the APIs from GT2. The successor of GT3 is already planned (GT4). GT4 will most probably adopt a so-called Web Services Resource Framework (WSRF) whose development has a large momentum currently.

5.7 Open Grid Services Architecture (OGSA)

Globus Toolkit Version 2 (GT2) led the way in developing the middleware necessary to construct a Grid. It contains the key concepts, but in a somewhat less general and flexible form. It was recognized that GT2 contained a number of deficiencies, especially in the field of protocol use (lacking standard means of authorization, invocation, termination, error propagation, etc.). In addition, there was significant lack of other functionality, for example for databases and workflow, and 'virtualization' of end systems.

The Open Grid Services Architecture (OGSA) [31, 69, 68] is a GGF initiative that was started in 2001 to provide a standard interface for a service-based architecture for the Grid. OGSA attempts to align Grid technologies with Web services technologies to capitalize on desirable Web services, such as service description and discovery, automatic generation of client and server code from server descriptions, binding of service descriptions to interoperable network protocols, and compatibility with emerging higher-level open standards, services and tools.

Besides OGSA, there exists the Open Grid Services Infrastructure (OGSI), that refers to the base infrastructure on which OGSA is built. OGSI addresses detailed specifications of the interfaces that a service must implement in order to fit into the OGSA framework. At its core is the 'Grid Service Specification', which defines the standard interfaces and behaviours of a Grid service, building on a Web services base. OGSI uses WSDL to achieve self-describing, discoverable and interoperable protocols.

A Grid Service is a Web Service that conforms to a set of conventions that provide for the controlled, fault resilient and secure management of stateful services. OGSA addresses architectural issues related to the requirements and interrelationships of Grid Services. In this way, OGSA further refines the Web Services model to focus on the features required by Grid infrastructures and applications.

GT3 implements OGSI and in addition provides some OGSI-compliant Grid services that correspond to GT2. For instance, GT3 includes an OGSI-compliant version of the execution service known as GRAM. GT3 provides the same services as GT2, as well as extensions to the Globus protocol suite and grid service development tools. The implementations provided in GT2 were designed before OGSA/OGSI, whereas many implementations provided in GT3 are OGSI-compliant. All major capabilities in GT2 (data transfer, replica management, job submission and control, secure communication, access to system information) are available in GT3.

It is beyond the scope of this study to describe the OGSA architecture and corresponding OGSI infrastructure in detail. We refer to [31, 69, 68] for further reading. However, we briefly mention the merits of OGSA that are of interest to this study.

OGSA provides an *architecture* in the sense that it provides a well-defined set of basic interfaces from which larger and more interesting systems can be built for commercial as well as scientific applications.

OGSA is *open* is the sense that it communicates extensibility, vendor neutrality, and interoperable protocols, and commitment to a community standardization process.

OGSA uses technology developed for Web services technologies that aim to provide platform-independent and language independent means to communicate between different hosts. OGSA incorporates specifications defined within the W3C and IETF and other standards organizations.

In OGSA all components are 'virtualized'. By providing a core set of consistent interfaces from which all Grid services are implemented, OGSA facilitates the construction of hierarchical, high-order services that can be treated in a uniform way across layers of abstraction. Such virtualization enables mapping of multiple logical resource instances onto the same physical resource, composition of services regardless of implementation, and management of resources based on composition from lower-level resources.

The OGSA Working Group in the GGF Architecture area aims at achieving an integrated approach to future OGSA service development via the documentation of requirements, functionality, priorities, and interrelationships for OGSA services. The progress of the OGSA is encapsulated in the 'Grid Services Specification' which reached version 1.0 at the GGF7 meeting held in Tokyo, Japan (March 2003). OGSA-related GGF groups include:

- Open Grid Services Architecture Working Group (OGSA-WG)
- Open Grid Services Infrastructure Working Group (OGSI-WG)
- Open Grid Service Architecture Security Working Group (OGSA-SEC-WG)
- Database Access and Integration Services Working Group (DAIS-WG)

The OGSA Data Access and Integration (DAI) project [39] aims to provide an mechanism allowing data resources such as databases to become part of the OGSA Grid architecture. This allows other OGSA-compliant components to interact and access these 'grid-enabled' data resources. The OGSA-DAI project constructs middleware that allows access and integration of data from separate data resources that can be accessed via a common interface regardless of the underlying technology. The OGSA-DAI provides primitive services on which higher level services can be constructed that provide for example data federation and distributed query processing capabilities. OGSA-DAI is already part of the Grid Technology Repository [13].

5.8 Application reuse and development in a grid environment

A question that should be addressed in this report is how existing HPC applications can be integrated and executed in a grid environment. In a traditional setting, a scientist establishes a (secure) connection to log directly onto the HPC facility (or its front-end machine) of his/her choice. The scientist then uses the full functionality provided by the shell access to install, compile and submit applications to the batch/queue system that manages the work load of the machine.

In a grid environment, the situation is different. There the scientist has to interact with the grid middleware. The scientist typically uses some interface (portal) that delivers the

users' job description in a secure way to the middleware. The middleware will then transfer the job to the specified target machine or use a resource brokering mechanism to decide which machine (or machines) suits the job best. This choice may for example depend on availability of compute nodes in the grid, characteristics of the job, and commercial considerations (pricing). The choice may also be determined by the job description itself that could request specific machine properties.

The middleware subsequently ensures that any program executables and data are transferred to the machine before the application starts executing. Any (intermediate) output from the application is then sent back via the middleware to the user's machine or stored as specified by the job description.

We do not attempt to answer the question which grid middleware is best suited for a particular application. The choice of middleware is not always a parameter in application development, but rather decided by for example the developer's working environment and local system administrator security policies.

In the remainder of this section, we briefly discuss how this is dealt with in two modern grid software models, UNICORE and Globus. We mention some aspects in the grid software models that are of interest from a reusability and efficiency point of view. We do not attempt to list the specific advantages and disadvantages of different technologies. For a detailed description of the UNICORE architecture and its features, we refer to [48] and for Globus to [29]. We also refer to the Sectoral Reports *Grid Service Requirements* and *Grid Enabling Technologies* that were released earlier in the ENACTS project [50, 51]. Besides the grid software models mentioned in these reports, there exist other application frameworks that allow existing (legacy) codes to be used in a multi-platform environment. We note in particular the Cactus framework [73].

5.8.1 UNiform Interface to COmputer REsources (UNICORE)

The design goals of the UNICORE architecture [48] are to provide a uniform interface for job preparation and control, to provide seamless and secure access to supercomputing resources, to preserve local site administrative autonomy over local resources, and to reuse as much as possible existing and emerging HPC technologies.

The UNICORE architecture [48] consists of three logical layers: user level, UNICORE server level, and batch-subsystem level. At the user layer, UNICORE gathers users grid resource requirements and code modules through an authenticated web browser, and authenticates and verifies access for the user. The UNICORE server layer matches a set of UNICORE resources to the users job requirements and dispatches the users job to the selected resources. The batch-subsystem level consists of computing resources that are accessible by a batch scheduling system.

From an end-user's perspective, UNICORE is based on the creation of so-called Abstract Job Objects (AJO) that are sent between the client and execution servers. The client has a *plugin interface*, a software module that extends dynamically the (UNICORE) middleware and provides the desired end-user functionality. The plug-in interface (Figure 11)

basically consists of two components: a task container that prepares the AJO (dealing with security aspects, resource brokering, etc.) and a graphical component that is the GUI for the container. This GUI can contain general components (File Import/Export panel, Resource management, Broker) as well as application specific components. Figure 11 shows a view of a Gaussian plugin.

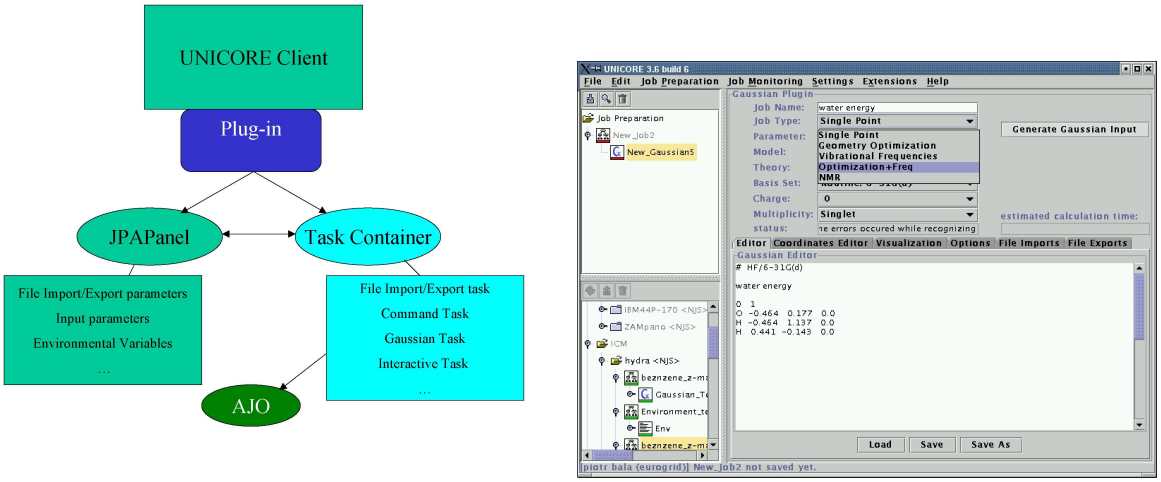

Figure 11: UNICORE plug-in interface and Gaussian GUI.

An application need not be aware of the grid. This facilitates the reuse of traditional sequential or parallel applications. An application is running somewhere on a remote target machine in the grid. The application (or job) is given assigned a home directory on the target machine and the plugin specifies where the input files to the application should be stored. The application accesses files locally on that machine. Those files are moved around in the grid using the UNICORE infrastructure.

The AJO prepared by the plug-in is sent to a computing resource (called Virtual Site, shortened Vsite), which may be a single computer, an SMP machine or a cluster of machines. A Vsite is managed by a single NJS. The NJS 'incarnates' the AJO, according to its incarnation database, into a batch script which then will be executed on the target resource.

This structure is reflected in the structure of an UNICOREpro job. A job is modelled as a directed acyclic graph of tasks and sub-jobs which may contain other tasks and subjobs. On the user level, a task cannot be divided into smaller execution parts. All tasks of a job or sub-job are executed on the same Vsite. However, a subjob (i.e., its tasks) may be executed on a different Vsite than the main job.

AJOs are signed with the users private key and contain information on the requested resources (software resources, runtime, memory, storage, etc.) and file imports/exports (source and target locations).

The development of UNICORE plugins is *not* straightforward for computational scientists that are not familiar with the UNICORE infrastructure, Java, etc. The developers of

UNICORE anticipated this problem and provided standard plugins: scriptPlugin, commandPlugin, compilePlugin.jar, autoupdatePlugin (to check the job status) and the smallServicePlugin (mainly intended as an example for plugin developers). These provide most of the functionalities that are needed to execute basic tasks on the UNICORE Grid.

In addition, a number of application-specific plugins have been developed within several projects (in particular EUROGRID [27]), like CPMD, Fluent, MSC.Nastran, Star CD, VampirTrace, Gaussian98, and Dyna 3D. In addition, there were functionalities added for interactive access (including SSH functionality within UNICORE), resource brokering, and application coupling using CORBA.

If one has some experience with the UNICORE system and Java, it is typically not very hard to write application plugins. UNICORE provides a full set of APIs needed to implement the part of the plugin that manages the communication with the Grid.

5.8.2 Globus

Globus Toolkit 2.4 (see also Section 5.6) provides access to Grid services through particular frameworks, the so-called Commodity Grid (CoG) Kits. The aim of the CoG kits is to enable the *reuse* of commodity technologies in combination with Grid technologies. CoG Kits integrate basic Grid services into existing application development frameworks, environments, and languages (these are similar to the UNICORE plugin APIs). The combination aims to reduce the threshold for acceptance and deployment of Grid services in existing infrastructures and to enhance the functionality, maintenance, and deployment of newly developed Grid services. Presently, there are kits available for Java (Java CoG Kit), Python (pyGlobus), Corba (Corba CoG Kit), Perl (Perl CoG) and Matlab.

The Globus Toolkit, CoGs, and appropriate application development tools form a Grid Computing Environment (GCE) capable of supporting collaborative development of grid applications. In context with the Globus initiative, various frameworks for collaborative and special industry solutions, as well as a grid services flow language are being worked on. For details and recent activities on Application Development Environments (ADEs) for grids at Globus, we refer to [29].

The effort needed to port a Globus-aware application from a 'classic' existing application depends on the complexity of the application but requires at least a good knowledge of the Globus Toolkit and preferably some scripting (Perl, Python) or Java programming skills. In short, to use Globus, developers must either write a Globus aware application from scratch or modify their application to replace UNIX system calls with Globus aware system calls.

A feature of Globus is that its capabilities can often be incorporated into an existing application *incrementally*, producing a series of intermediate increasingly 'grid-enabled' versions. For example, a grid-enabled distributed interactive simulation application, SF-Express, was developed at Caltech as follows [29]:

- Integration of resource management services to simplify program startup on multiple computers, avoiding separate logins and scheduler commands to each system.

- Introduction of the Globus library called DUROC to coordinate program startup across multiple sites, addressing certain fault detection and recovery strategies.
- Use of Globus data access services to stage executables and configuration files to the remote systems, and to direct output streams to the originating location.

MPICH-G2 is a grid-enabled implementation of the MPI-1.1 standard ([19], see also Section 4.2). That is, using services from the Globus Toolkit (e.g., job startup, security), MPICH-G2 allows one to couple multiple machines, potentially of different architectures, to run 'standard' MPI applications. MPICH-G2 automatically converts data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for inter-machine messaging and (where available) vendor-supplied MPI for intra-machine messaging.

MPICH-G2 enables certain applications to run in a grid environment with minor (or even without) modifications. One important class of applications is those that are *distributed by nature*, for example, remote visualization applications in which computationally intensive work producing visualization output is performed at one location, perhaps as an MPI application running on some massively parallel processor, and the images are displayed on a remote high-end (e.g., IDesk, CAVE) device. For such problems, MPICH-G2 allows one to use MPI as your programming model. A second class of applications is those that are *distributed by design* where one has access to multiple computers, perhaps at multiple sites connected across a WAN, and one wishes to couple these computers into a computational grid.

It is relatively straightforward to use MPICH-G2 for applications in which all the processes are *on the same side* of a firewall. However, if one wants to run MPICH-G2 applications where processes are *on opposite sides* of a firewall, then one needs to accommodate for that. The two issues that arise are job control (e.g., start-up, monitoring, and termination) and TCP messaging during execution. Since MPICH-G2 uses Globus services for both, using MPICH-G2 through firewalls is really an issue of using Globus through firewalls.

Other examples of Globus related application development tools and environments are the Grid Application Development Software (GrADS, [1]), the IBM Grid Toolbox [3], and the Grid Application Framework for Java (GAF4J, [2]).

Finally, for a more thorough treatment of enabling applications for grid computing in Globus, we refer to [76].

5.8.3 DAMIEN

The goal of the European project DAMIEN (Distributed Applications and Middleware for Industrial use of European Networks, 2001-2003) [32] was to develop a toolset for Grid computing, based partially on existing and widely accepted tools in HPC and partially on developing new tools. All tools together form an environment which aims to support application developers and end-users in a novel way on the Grid.

The communication library used is PACX-MPI (see Section 4.2). For the coupling of several applications, the Mesh-based parallel Code Coupling Interface (MpCCI) [46] is employed. For debugging and performance analysis of a grid-enabled scientific application, the tool MetaVampir [61] is used. Both MpCCI and MetaVampir were ported to PACX-MPI for this purpose.

For applications in production mode, Dimemas [72] is used. Dimemas is a tool for performance prediction, which can estimate the behaviour of an application based on an initial tracefile. Based on this file, the user can modify certain parameters of the run, e.g., the quality of the communication network or the distribution of the processes on the machines. Dimemas then simulates the behaviour of the application with modified parameters.

6 Summary and Recommendations

In this report, we have reviewed some of the aspects related to efficiency and reusability in modern and emerging software technologies for computational science. We hope that the document has been useful to the reader in a variety of ways, for example

- to understand some of the major factors that impede software efficiency and successful software reuse in traditional HPC environments and in distributed and grid environments
- to recognize the relationship between software reuse and various other software engineering concepts and techniques
- to recognize the complexity related to software efficiency in modern (multi-level) hardware and software technologies
- to understand the importance of software standards and software interoperability and their impact on many facets of software design and application development and on computational science and engineering in general

One objective of this study was to put forward recommendations for the establishment of a pan-European HPC centre with respect to software efficiency and reusability. We believe the following conclusions and recommendations are valid.

1. Moving large numbers of application codes between heterogeneous HPC systems or to new software environments is only feasible if well-accepted standards, languages and tools are available. In a pan-European HPC consortium, this will be facilitated if the user work environment is standardized in some form and best practices for software development and job execution are established. Potentially troublesome hardware-specific or site-specific dependencies should obviously be eliminated or be dealt with in a transparent manner.

2. Traditionally, data management connected to a numerical simulation was limited to the use of the programming bindings provided. The application typically ran on only one architecture and the local file systems would ensure that I/O to the application could be achieved by using the primitive bindings provided by programming languages like C and Fortran. The archival, retrieval, and exploration of remote (off-site) data is of growing importance as computer systems enable the rapid generation of extremely large, complex, and remote data sets. Increasing the ease and efficiency of data transfer and access will greatly enhance the amount of science that can be performed on HPC facilities, and allows for inter-disciplinary sharing of data. An HPC centre or consortium should therefore promote best practices in portable data management towards its user community and provide appropriate advice and training.

3. Tools are important in enhancing the efficiency of the software engineering process and of the software itself, and permit rapid code debugging, performance profiling and analysis, and subsequent software re-engineering. HPC centres should actively keep users up-to-date with the latest information on emerging tools, monitor their development and

report on those that are deemed most effective and useful for the user community. Promising tools should be evaluated for robustness, functionality, and performance, and their applicability in real-life applications assessed. A tool that is considered mature enough and provides added value should be promoted to the user community of the HPC consortium by means of providing adequate training material.

4. An HPC user may not rapidly move onto the grid if this requires significant changes to his/her existing application. Ideally such move must be seamless and require little software revisions. Getting users onto the grid will require that end-user interfaces towards the grid middleware are simple or are similar to what the user uses in a traditional HPC environment. The computational scientist need not have to go through a lengthy learning process before he/she feels comfortable using grid technology. Ideally, the sheer technological complexity of the grid should be hidden to the user as much as possible. The HPC centres will have to make a serious effort in achieving this by choosing the right middleware technologies, provide friendly end-user interfaces (portals) to this middleware, and provide adequate training and support for the new methodologies.

An emerging problem in grid computing is the sheer amount of portals that are being developed world-wide. A grid portal may be constructed as a Web page interface to provide easy access to grid applications and provides user authentication, job submission, job monitoring, and results of the job. Many grid projects are developing their own interfaces to software packages that have a well-defined interface (e.g., the chemistry application Gaussian). There appears to be quite some overlapping (and also incompatible) activity in this context that could be avoided by merging portal development projects and reusing previously developed (generic) portal framework software.

5. Standards for grid services, web services are rapidly evolving. This is good in the sense that it shows a significant progress in (and need for) the methodology. On the other hand, it makes it harder for a user of the grid (not necessarily a code developer) to keep up-to-date with the latest developments and keep the application 'alive'. Adapting to emerging web and grid standards may require regular or frequent changes to the grid-enabled application. This is undesirable for production software. An additional complication is that HPC centres typically do not support the same (recent) version of a rapidly evolving standard or tool. Moreover, different (e.g., vendor) implementations of evolving standards may not all support the latest changes to the standard definition. Regular synchronization between the HPC centres is needed in this context to reduce this kind of portability problems.

6. The use of open standards should be promoted actively. Open standards have several advantages, see Sections 2.5 and 3. Also the contribution by users or user groups to the standardization process of open standards should be promoted. Open standards are not only used in the area of high performance computing or grid computing, but also in many other areas of science, like for example visualization (OpenGL),

7. The Web has become the user interface of global business, and Web services now offer a strong foundation for software interoperability through the core open standards of XML, SOAP, WSDL, and UDDI. Models and applications that make use of this huge potential are just beginning to emerge.

It can be expected that Web services will play an increasingly more important role in integrating new initiatives in HPC consortia. Web services standards permit application-to-application interoperability, but the coordination of a set of Web services working towards a common end is still an open issue. Several XML-based protocols are under standardization to target specific needs of businesses and application domains. These standards clearly illustrate the momentum behind the Web services computing community.

The large majority of HPC user community is not currently familiar with Web services. We therefore strongly recommend that HPC centres provide good training programs for their users to get acquainted at an early stage with the basic concepts and tools in Web services and show the users the impact these services can have on their research activity.

8. Many of the new technologies require knowledge of programming languages like Java and C++, scripting languages like Perl, middlewares like Globus, and XML-based standards. Several of these are fairly new and evolving. Computational science groups and (senior) scientists that traditionally have used HPC in their activities are not always familiar with these recent languages and technologies. This lack of familiarity hinders the uptake of these new technologies by traditional HPC user groups. It is therefore important that HPC centres not only provide adequate training programs on the latest technologies but also on the basic components (and concepts) that underpin these technologies.

9. In its early days, grid computing was often thought of as a *computational* grid, an infrastructure that combines a set of distributed computing resources into one big computing resource on which one can run a large scale application that solves a computational problem that is too large to fit on any single machine. However, application efficiency on such a grid remains an issue. The interconnect between the individual resources can be slow, faulty, and insecure, and hence, the efficiency and reliability of the overall distributed application may not be what one would like. Moreover, these limitations in the interconnect may lead to under-utilization of the individual (but expensive) resources.
These days, one can see a shift towards grid computing as being a *data* grid, an infrastructure for distributed data management that is transparent to the user and application. Such infrastructure greatly facilitates the reusability of existing applications. For example, a data grid allows the physical location of the data to be decoupled from the physical location of the application. The application need not be aware of this since the run time environment will make sure that the data will be transferred to a place where the application expects the data to be. The data grid provides mechanisms that form the glue between remote applications, devices that generate data, and data bases, and thus enables the creation of smart coupled applications.
Another emerging form of grid computing is the *collaborative* grid. Such a grid enables the creation of virtual organizations in which remote research groups can perform joint research and share data.

10. The use of grid technologies will eventually lead to more remote (distant) collaborations. It is therefore essential that mechanisms are in place for code maintenance by a large research group with multiple programmers modifying a single code. It is our experience that many scientists have little knowledge of the wealth of tools available to assist in this. An HPC centre or consortium should be committed to provide advice and training in

code development and maintenance; application of these techniques leads directly to increased productivity and enhanced code portability. This will become even more apparent when grid-enabling technology will have become mature enough to become an accepted tool for remote (global) collaboration.

Finally, we note that several newly established European projects in Framework VI address some of the issues addressed in this report. These include HPC-EUROPA [33] and DEISA [24]. The interested reader is encouraged to follow the developments in these projects. We also refer to the Sectoral Reports that have already appeared in the ENACTS project. These are available from the ENACTS home page `http://www.enacts.org`

References

[1] *http://nhse2.cs.rise.edu/grads/*, Grid Application Development Software (GrADS).

[2] *http://www.alphaworks.ibm.com/tech/GAF4J*, Grid Application Framework for Java (GAF4J).

[3] *http://www.alphaworks.ibm.com/tech/gridtoolbox*, IBM Grid Toolbox.

[4] `http://www.hlrs.de/organization/pds/projects/pacx-mpi/`, PArallel Computer eXtension (PACX-MPI).

[5] `http://www.ietf.org/`, The Internet Engineering Task Force (IETF).

[6] `http://www.lam-mpi.org`, LAM-MPI.

[7] `http://www.mcs.anl.gov/mpi/index.html` *and* `http://www.mpi-forum.org/docs/docs.html`, Message-Passing Interface (MPI).

[8] `http://www.w3.org/` *and* `http://www.w3schools.com/`, W3C and W3C web building tutorials.

[9] `http://collaboratory.emsl.pnl.gov/sam/bfd/`, Binary Format Description language (BFD).

[10] `http://daac.gsfc.nasa.gov/REFERENCE_DOCS/HDF/gdaac_hdf.html`, Goddard DAAC's Hierarchical Data Format (HDF).

[11] `http://en.wikipedia.org/`, Wikipedia. The Free Encyclopedia.

[12] `http://forge.gridforum.org/projects/dfdl-wg/`, DFDL Working Group.

[13] `http://gtr.globus.org/`, Grid Technology Repository.

[14] `http://hdf.ncsa.uiuc.edu/HDF5/`, HDF5.

[15] `http://icl.cs.utk.edu/netsolve/index.html`, Netsolve/GridSolve.

[16] `http://icl.cs.utk.edu/papi/`, Performance API (PAPI).

[17] `http://soureforge.net/`, SourceForge.

[18] `http://www-pablo.cs.uiuc.edu/`, SvPablo.

[19] `http://www3.niu.edu/mpi/`, MPICH-G2.

[20] `http://www.avaki.com`, Avaki.

[21] `http://www.cepba.upc.es/paraver/`, Paraver.

[22] `http://www.corba.org/`, Common Object Request Broker Architecture (CORBA).

[23] `http://www.cs.virginia.edu/ legion/`, Legion.

[24] `http://www.deisa.org`, DEISA.

[25] http://www.edikt.org/, E-science Data, Information and Knowledge Transformation (edikt).

[26] http://www.epcc.ed.ac.uk/research/jomp/, JOMP.

[27] http://www.eurogrid.org/, EUROGRID.

[28] http://www.ggf.org/, Global Grid Forum (GGF).

[29] http://www.globus.org/, Globus.

[30] http://www.gnu.org/, GNU Operating System - Free Software Foundation.

[31] http://www.gridforum.org/ogsi-wg/, Open Grid Services Architecture (OGSA).

[32] http://www.hlrs.de/organization/pds/projects/damien, Distributed Applications and Middleware for Industrial use of European Networks (DAMIEN).

[33] http://www.hpc-europa.org, HPC-EUROPA.

[34] http://www.iso.ch/, The International Organization for Standardization (ISO).

[35] http://www.javagrande.org/, Java Grande Forum.

[36] http://www.jcp.org, Java Community Process.

[37] http://www.mathworks.com/, The Mathworks Inc. (MATLAB).

[38] http://www.nag.co.uk/sc22wg5, ISO Fortran Standards Technical Committee Working Group 5.

[39] http://www.ogsadai.org/, OGSA Data Access and Integration (OGSA DAI).

[40] http://www.omg.org/, Object Management Group (OMG).

[41] http://www.openmp.org, OpenMP.

[42] http://www.opensource.org/, Open Source.

[43] http://www.pallas.com/e/products/vampir/, Vampir/Vampirtrace.

[44] http://www.perl.org/, Perl.

[45] http://www.python.org/, Python.

[46] http://www.scai.fraunhofer.de/222.html, Mesh-based parallel Code Coupling Interface (MpCCI).

[47] http://www.tcl.tk/, Tcl.

[48] http://www.unicore.org/, UNICORE.

[49] http://www.w3.org/xml/, eXtensible Markup Language (XML).

[50] *ENACTS - Grid Enabling Technologies*, Sectoral Report, December 2002. A joint scientific and technological study undertaken by FORTH and the Swiss Center for Scientific Computing for the ENACTS network. http://www.enacts.org.

[51] *ENACTS - Grid Service Requirements*, Sectoral Report, January 2002. A joint scientific and technological study undertaken by EPCC and the Poznan Supercomputing Centre for the ENACTS network. `http://www.enacts.org`.

[52] *ENACTS - Data Management in HPC*, Sectoral Report, October 2003. A joint scientific and technological study undertaken by CINECA and TCD for the ENACTS network. `http://www.enacts.org`.

[53] G. ALLEN, D. ANGULO, T. GOODALE, T. KIELMANN, A. MERZKY, J. NABRZYSKY, J. PUKACKI, M. RUSSELL, T. RADKE, E. SEIDEL, J. SHALF, AND I. TAYLOR, *GridLab: Enabling Applications on the Grid*, in GRID Computing - GRID 2002, M. Parashar, ed., Baltimore, MD, USA, November 2002, Springer, pp. 39–45. Lectures Notes in Computer Science 2536.

[54] E. ANDERSON, Z. BAI, C. BISCHOF, S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. D. CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' Guide, Third Edition*, SIAM, September 1999.

[55] C. ATKINSON AND D. MUTHIG, *Enhancing software reusability through product line technology*, in Software Reuse: Methods, Techniques, and Tools. 7th Int. Conf. (ICSR-7), C. Gacek, ed., Austin, TX, USA, April 2002, Springer, pp. 93–108. Lectures Notes in Computer Science 2319.

[56] S. BALAY, K. BUSCHELMAN, W. D. GROPP, V. EIJKHOUT, D. KAUSHIK, M. KNEPLEY, L. C. MCINNES, B. F. SMITH, AND H. ZHANG, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 2.2.0, Argonne National Laboratory, February 2004.

[57] S. BALAY, K. BUSCHELMAN, W. D. GROPP, D. KAUSHIK, M. KNEPLEY, L. C. MCINNES, B. F. SMITH, AND H. ZHANG, *PETSc home page*. http://www.mcs.anl.gov/petsc, 2001.

[58] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[59] E. J. BRAUDE, *Software Engineering: An Object-Oriented Perspective*, John Wiley & Sons, Inc., 2001. ISBN 0-471-32208-3.

[60] A. W. BROWN AND G. BOOCH, *Reusing open-source software and practices: The impact of open-source on commercial vendors*, in Software Reuse: Methods, Techniques, and Tools. 7th Int. Conf. (ICSR-7), C. Gacek, ed., Austin, TX, USA, April 2002, Springer, pp. 123–136. Lectures Notes in Computer Science 2319.

[61] H. BRUNST, M. WINKLER, W. E. NAGEL, AND H.-C. HOPPE, *Performance optimization for large scale computing: the scalable vampir approach*, in Computational Science ICCS 2001, San Francisco, Springer, 2001, pp. 751–760. Lecture Notes in Computer Science 2074 Part II.

[62] J.-Y. CHUNG, K.-J. LIN, AND R. G. MATHIEU, *Web services computing: Advancing software interoperability*, IEEE Computer, (2004). `http://www.computer.org/computer/homepage/1003/GEI/`.

[63] J. W. DEMMEL, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, USA, 1997.

[64] J. DONGARRA, J. DU CROZ, I. DUFF, AND S. HAMMARLING, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 1–17.

[65] J. J. DONGARRA, I. S. DUFF, D. C. SORENSEN, AND H. A. VAN DER VORST, *Solving linear systems on vector and shared memory computers*, SIAM, Philadelphia, USA, 1991.

[66] H. ERDOGMUS AND T. WENG, eds., *COTS-Based Software Systems*, Springer, Ottawa, Canada, February 2003. Lectures Notes in Computer Science 2580.

[67] I. FOSTER AND C. K. (EDS.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

[68] I. FOSTER, C. KESSELMAN, J. M. NICK, AND S. TUECKE, *The Physiology of the Grid. An Open Grid Services Architecture for Distributed Systems Integration*. Version 6/22/2002, `http://www.gridforum.org/ogsi-wg/`, 2002.

[69] I. FOSTER, C. KESSELMAN, AND S. TUECKE, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Int. J. of Supercomputer Applications, 15 (2001), pp. 200–222. `www.globus.org/research/papers/anatomy.pdf`.

[70] M. FRIGO AND S. G. JOHNSON, *FFTW: An adaptive software architecture for the FFT*, in Proc. 1998 Int. Conf. Acoustics Speech and Signal Processing (ICASSP), vol. 3, IEEE, 1998, pp. 1381–1384. `http://www.fftw.org/`.

[71] C. GACEK, ed., *Software Reuse: Methods, Techniques, and Tools*, Springer, 7th Int. Conf. (ICSR7), Austin, TX, USA, April 2002. Lectures Notes in Computer Science 2319.

[72] S. GIRONA, J. LABARTA, AND R. M. BADIA, *Validation of Dimemas communication model for MPI collective communications*, in 7th EurPVM/MPI, Balatonfüred, Lake Balaton, Hungary, September 2000.

[73] T. GOODALE, G. ALLEN, G. LANFERMANN, J. MASSÓ, T. RADKE, E. SEIDEL, AND J. SHALF, *The Cactus Framework and Toolkit: Design and Applications*, in High Performance Computing for Computational Science - VECPAR 2002, Porto, Portugal, June 2002, Springer, pp. 197–227. Lectures Notes in Computer Science 2565.

[74] M. GRISS AND R. KESSLER, *Achieving the Promise of Reuse with Agent Components*, in Software Engineering for Large-Scale Multi-Agent Systems, Springer, January 2003, pp. 139–147. Lectures Notes in Computer Science 2603.

[75] J. HEIN, *Java Grande language comparison benchmarks on the IBM p690 system*, tech. rep., Edinburgh Parallel Computing Centre, The University of Edinburgh. Technical Report of the UKHEC Collaboration, `www.ukhec.ac.uk`.

[76] B. JACOB, L. FERREIRA, N. BIEBERSTEIN, C. GILZEAN, J.-Y. GIRARD, R. STRACHOWSKI, AND S. YU, *Enabling Applications for Grid Computing with Globus*, IBM International Technical Support Organization, 2003. IBM Redbooks SG246936.

[77] B. B. KERNIGHAN AND D. RITCHIE, *The C Programming Language*, Prentice-Hall, 1988.

[78] O. LASSILA AND R. R. SWICK, *Resource Description Framework (RDF) model and syntax specification*, tech. rep., W3C, 1999. Recommendation, 22 February 1999, `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[79] B. LAVERMAN, *Supporting software reusability with polymorphic types*, PhD thesis, the Netherlands, 1995.

[80] M. METCALF AND J. REID, *Fortran 95 explained*, Oxford University Press, 1996. ISBN 0-19-851888-8.

[81] J. F. PETERS AND W. PEDRYCZ, *Software Engineering: An Engineerng Approach*, John Wiley & Sons, Inc., 2000. ISBN 0-471-18964-2.

[82] T. SGOUROS, *DODS user guide, Version 1.11, May 22, 2003*, tech. rep., 2003. `http://www.unidata.ucar.edu/packages/dods/user/guide-html/guide.html`.

[83] L. A. SMITH AND J. M. BULL, *Java for High Performance Computing*, tech. rep., Edinburgh Parallel Computing Centre, The University of Edinburgh. Technical Report of the UKHEC Collaboration, `http://www.ukhec.ac.uk`.

[84] B. STROUSTRUP, *The C++ Programming Language (2nd edition)*, Addison-Wesley, 1991.

[85] H. VAN VLIET, *Software Engineering: Principles and Practice*, John Wiley & Sons, Ltd., second ed., 2000. ISBN 0-471-97508-7.

[86] R. C. WHALEY, A. PETITET, AND J. J. DONGARRA, *Automated Empirical Optimization of Software and the ATLAS Project*, Tech. Rep. UT-CS-00-448, Dept. of Computer Science, Univ. of Tennessee, Knoxville, 2000. Linear Algebra Working Notes 147.