Domain-Specific Metacomputing for Computational Science:

Achieving Specificity Through Abstraction

Oral Comprehensive Exam Position Paper
September 1997

By Steven T. Hackstadt

Department of Computer & Information Science
University of Oregon, Eugene, OR 97403

Contents

CHAPTER 1

The “Big” Deal with Computational Science

The most incomprehensible thing about the world is that it is comprehensible.

- Albert Einstein
(1879-1955)

SCIENCE CARES little about philosophical conundrums—even if proposed by one of its greatest contributors. Incomprehensible or not, science is in a relentless pursuit of comprehensive knowledge, constantly striving to characterize the unknown, understand the mysterious, and explain the obvious. Mankind’s scientific knowledge, while relentlessly pursued, is far from comprehensive even though in almost every discipline, scientists have at their disposal a vast array of technology to assist them. But science and technology, while often equated, have a complex relationship, for it is often the case that science results in technology that is of essential use to other science. And while technology certainly has broad implications and applications outside of science, the self-perpetuating relationship between the two is central to mankind’s relentless pursuit of the incomprehensible.

Computational Science Research: An Evolution of Computer Science

Without question, one of the greatest technological achievements of this century has been the computer. With simulation now an adjunct to experimentation and theory as paradigms of science [NCO96], scientists increasingly need to use the computational power of high performance computers to work on very large problems. To the scientist, though, computers are only as good as the problems they can solve or address. The area of computational science is primarily concerned with facilitating scientists’ ability to solve or simulate large scientific problems with computers.

Recent advances in computing techniques and technology, coupled with new models for scientific phenomena, are fueling a revolution in the way science and engineering are performed. The impact of this revolution is still in its infancy, though, in part because access to high performance computers is relatively scarce and because few people (among scientists, in particular) possess the skills to use such machines [NSF93]. Thus, computational science may be described as a combination of computer science and traditional physical sciences, with the expected side-effect that scientists become more computer-savvy and computer scientists become more science-savvy. In the computer science community, two trends have been instrumental in motivating the focus on computational science research.

First, during the past century, society has benefited so significantly from scientific achievements that the nature of science is, as Rice declares, shifting from “science for science’s sake, to science for society’s sake” [Rice95]. That is, government is increasingly expected to ensure that the research it funds with taxpayers’ dollars will generate useful results. In this atmosphere, social and government concerns for health, education, security, environment, economics, etc. are becoming the motivating factors in funding decisions.

Second, computer science is a relatively new “science.” As such, it has been building its own foundations. Rice argues that those foundations are now laid, and that computer science must now become more “outward looking” and challenge itself with solving real problems [Rice95].

These trends are having a profound effect on the nature of computer science. The new focus on computational science will contribute greatly to the benefit that computer science will have on society in the next century.

Solving “Big” Problems

Computational science spans a broad range of scientific areas, including computational fluid dynamics, geology, material science, mathematics, mechanics and structural analysis, molecular modeling and quantum chemistry, and physics. Problems represented by these areas are “big,” as characterized across several dimensions. Problem size, simulation complexity, and required execution time each impact the computational requirements of a given application. The following application descriptions exemplify these requirements.

> **Mechanics and Structural Analysis.** One application of supercomputers in this area is to analyze automobile crashworthiness. Models consisting of 100,000 to 250,000 unknowns that require 20 hours of Cray C90 computer time to solve often prove too coarse to generate accurate predictions, requiring verification by actual vehicle crash tests [NSF93]. Indeed, computational requirements are often so large that resolution and consideration of important physical phenomena are omitted in exchange for faster results.

> **Molecular Modeling and Quantum Chemistry.** Computer models are used to understand the effects of carcinogens on the structure of DNA. In one simulation, it can take a Cray Y-MP nearly six days to model the interactions of 3,500 water molecules and 16 sodium ions for just 200 trillionths of a second [NSF93]. Hence, problems are often so complex that even the briefest simulation can take an inordinate amount of time to complete.

> **Computational Fluid Dynamics.** Numerical equations of mass, momentum, and energy simulate the evolution of fluids. Such a simulation would typically take place in three dimensions, each divided into, say, 1,000 cells. If each cell requires 100 floating point operations per time step and the simulation were to last 25,000 time steps, the entire simulation would require 2.5 quadrillion (2.5 x 10^{15}) floating point operations. And if a scientist requires the results in two hours, the computing system must sustain 300 billion (3 x 10^{11}) floating point operations per second (commonly expressed as 300 gigaflops) [NSF93]. Thus, applications with strict time requirements can rapidly drive up the performance requirements of the computing hardware.

Many problems in these areas and others have been designated Grand Challenges, which are computation-intensive, “fundamental problems in science and engineering with broad economic and scientific impact whose solution can be advanced by applying [high performance computing and communications] technologies” [NCO96].

A major question facing application and computer scientists is how best to apply such technologies to these problems. The computer science community has long envisioned *problem-solving environments (PSEs)*. Gallopoulos, Houstis, and Rice [GHR94] describe a PSE as “a computer system that provides all the computational facilities necessary to solve a target class of problems... [including] advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods....” While the technology of problem-solving environments holds promise for well-defined, well-understood problems with standardized solutions, it does not adequately address the leading-edge, experimental nature of most Grand Challenge problems [GHR94]. Even so, the general concept of a comprehensive computational environment is appealing as a means of improving scientists’ access to high performance computing systems.

More recent work [CDHH96, BRRM95] has taken a different approach. Rather than provide a specific computational capability that potentially spans many computational domains (*e.g.,* solving partial differential equations), *domain-specific environments (DSEs)* seek to provide comprehensive computational abilities within a single domain. That is, a DSE seeks to address “requirements that are unique to [a particular] application domain” through a collaboration between application and computer scientists [CDHH96]. While their approaches are orthogonal to each other, DSEs and PSEs share many of the same motivations and characteristics. Central to both technologies is achieving high performance through the use of parallel computing systems.

Pancake points out that while parallel computing has been around in various forms since the mid-1970s and current systems are “undeniably powerful,” the large-scale transition to parallel computing has been slow because of a lack of software support [Panc91]. The relatively recent appearance of PSE and DSE software that supports parallel computing capabilities with the scientist in mind at first appears to corroborate this claim. But it is slightly more complicated than that because today’s proponents of PSEs claim that while attempts to build PSEs in the 1960s and 1970s were thwarted because “technology could not yet support PSEs in computational science,” today’s “high-performance computers combined with better understanding of computing and computational science have put PSEs well within our reach” [GHR94]. Essentially, each blames the other. That is, parallel systems were not adopted because adequate software support was not available, but those software environments could not be developed, in part, because machines did not offer enough performance. We are simply considering different sides of the same coin, implying that achieving good application performance requires both robust software and high performance machines. Pancake explains the nature of that coin further [Panc91]:

> The audience for high-performance computing is not the computer science community, but scientists, engineers, and other technical programmers whose computational requirements exceed the capacities of even our fastest sequential machines. They have turned to parallel processing because their problems are too big, or their time constraints too pressing, for conventional architectures. As Ken Neves of Boeing Computer Services puts it: “Nobody wants parallelism. What we want is performance.”

Delivering “Big” Performance

As computer systems offer more performance potential, they become more appealing to scientists with “big” problems to solve. According to the United States Department of Energy, if current trends in high performance computing (HPC) continue, sustainable teraflop performance (that is, a trillion floating point operations per second) will be achieved by the year 2002, but machines capable of sustaining hundreds of teraflops will not appear until roughly 2025 [DOE96]. With computational science and Grand Challenge problems already rapidly approaching teraflop-level performance, the U.S. Government has launched the Accelerated Strategic Computing Initiative (ASCI) to address the “full system, full physics” problems surrounding virtual testing and prototyping capabilities for nuclear stockpile stewardship [DOE96]. This program has stimulated U.S. high performance computer manufacturers to create more powerful machines. A teraflop system is already in place at Sandia National Laboratory, and systems capable of more than three teraflops each will be sited at Lawrence Livermore and Los Alamos National Laboratories within the next two years. Sustainable performance of hundreds of teraflops is expected by 2003, bucking current trends by more than twenty years.

For many computational science problems, a machine is rarely too powerful. Being able to perform longer simulations on larger problem sizes at higher resolutions is certainly not considered a bad thing among scientists. The primary issue, though, is cost-effectiveness. Together, the first three ASCI systems will cost a total of about $250 million.[1] Obviously, cost-effectiveness is not the primary goal of the ASCI program. But, cost-effective performance is now an important goal for the business-oriented computing market. And cost-effectiveness is a motivating factor behind other trends in high performance computing.

In particular, researchers feel there are additional ways of significantly improving application performance. For example, by using existing resources more efficiently and increasing access to spare computational cycles, researchers claim that tremendous amounts of computing power can be harnessed [BP94, GNW95, SKS92, ZWZD92]. Headed in this direction is the area of *heterogeneous network computing*, which has as its goal the use of a collection of autonomous computers to solve one or more computational tasks concurrently ([Esha96], p. 4). Work in metacomputing extends this notion in both scale and overall system cohesiveness. These areas are discussed in considerable detail later. The most successful approach will undoubtedly be the one that can offer cost-effective increases in performance and simultaneously take the greatest advantage of existing, stand-alone, high performance systems. In fact, the ASCI program recognizes this and is ultimately looking to university research to develop the long-term solutions needed to bridge the gap between the individual teraflop systems being built today and the hundred-teraflop system expected by 2003.

Building “Big” Software

Unfortunately, raw machine performance figures are, for all intents and purposes, unattainable by real applications.[2] The input/output, synchronization, and communication costs of parallel and distributed programs can degrade real application performance significantly. Consequently, achieving “big” performance requires more than just bigger, faster computers. Concerted efforts at improving the performance of application codes, algorithmic kernels, and system software are also required. The ASCI project, for example, expects a tenfold performance increase from improvements in system software alone. Similar improvements in the application codes themselves are also expected [DOE96].

Constructing the software systems for the ASCI platforms is challenging enough; tuning and optimizing them for performance is even harder; but uniting the three teraflop systems (and others) into a cohesive metacomputing environment eventually capable of sustaining hundreds of teraflops is no less than a monumental undertaking. Such a system must support application development, debugging,

1. According to corporate press releases, the first three systems delivered as part of the ASCI program cost $46 million (Intel), $93 million (IBM), and $110 million (SGI/Cray), respectively; the entire ASCI budget is nearly $1 billion over the next ten years.
2. The ASCI performance requirements are based on benchmark results rather than theoretical machine performance; even so, the performance of real applications will likely be less (and in some cases significantly so) than the benchmark performance figures.

performance evaluation and tuning, archival data access, and a host of other application- and programmer-oriented services—all with performance being paramount [SDA96]. This is "big" software. And while ASCI may be at one extreme of the high performance computing spectrum, there are numerous other research labs, universities, and companies seeking more moderate increases in performance simply by improving the utilization of their existing computational resources in a similar manner.

Big software must be designed and built in a methodical, organized manner. However, among the parallel and distributed computing research community, software has largely been built on a very ad hoc basis [Panc91, Panc94]. Furthermore, the tools and environments that have been produced, with a few notable exceptions, have gone largely unused by programmers and scientists. Finally, history has shown that industry simply can not be relied upon to produce high-quality, useful tools and environments for parallel and distributed systems [Panc94], in part because the market is just too small compared to the massive personal computer software market. Pancake claims that just one in fifty research tools and one in twenty commercial tools can be deemed "successful."[3]

What, then, is going to happen as researchers begin extending the computational software environment to include, for instance, metacomputing? Fortunately, the groups carrying out the preliminary efforts in this area appear to have at least some understanding of the more general software design and development issues posed by these "bigger" systems [FK96, GW96]. Grimshaw recognized this point in the early days of his work in this area [GNW95]:

> The issue is not whether metasystems will be developed; clearly they will. Rather, the question is whether they will come about by design and in a coherent, seamless system—or painfully and in an ad hoc manner by patching together congeries of independently developed systems, each with different objectives, design philosophies, and computation models.

True to those words, current implementations appear to be emphasizing, to varying degrees, the use of "frameworks" [AM94, DGMS93, Sund96] and "toolkits" [FK96], "object-oriented" design [GW96, ABCH95], and building software from "components" [BWFS96].

Interestingly, these terms and concepts are more often seen in the field of software engineering than high performance computing. While the techniques of software

3. Pancake defines "success" based on the "frequency and significance of tool use" [Panc94].

engineering are often embraced by business and industry, other areas of computer science have seemed less inclined to adopt them. Perhaps this is changing as the HPC community is increasingly faced with building “big” software like that required for metacomputing environments. Indeed, this represents an interesting research challenge.

Software engineering has a fair body of work that could benefit the HPC community in this regard. For example, software composition studies how software can be built as a collection of individual components [AB96, GAO95, NM95]. Other work explores issues of interoperability among independent pieces of software [Heil95, Mano95, PSW91, Purt94]. Object-oriented approaches have been highly touted as improving productivity [MN96]. Many of these topics are encompassed in the emerging area of *software architecture* [SDKR95, SG96].

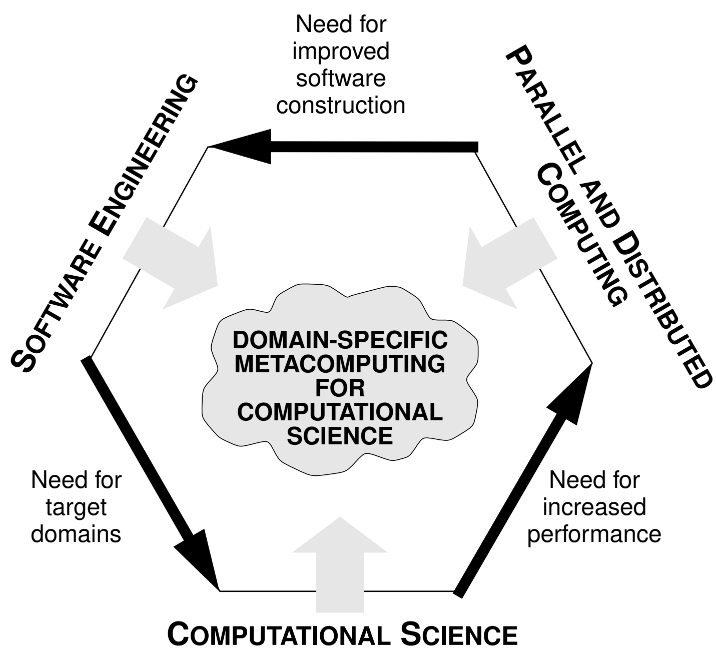
According to Shaw and Garlan [SG96], software architecture “involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.” One area in particular, *domain-specific software architectures (DSSAs)*, may hold particular promise in constructing metacomputing support for computational science problems [HPLM95, TTC95]. The core idea of this area is to tailor the organizational structure of the software to a particular family (or “domain”) of applications. While the topic of software architecture is resumed later, it is important to note at this point that a fundamental requirement for a domain-specific software architecture is, of course, a domain. That is, DSSAs apply to classes of applications that share concepts, terminology, data types, computational structures, etc. It is not surprising, then, that DSSAs potentially have much in common with domain-specific environments. Their commonality lies in a pervasive consideration of the nature of the problems to be solved by the software environment. While DSSAs focus on an appropriate design and development methodology, DSEs tend to focus on meeting functional, nonfunctional, and implementational requirements. As we discuss later, a combined approach holds particular promise.

The “Big” Picture: A Research Challenge

The previous sections have sketched a unifying path through the areas of computational science, parallel and distributed computing, and software engineering. Along this path, a variety of “technologies” that hold promise in addressing the requirements of “big” computational science problems have been mentioned. The nature of this path and the dependencies among the technologies encountered along it are central to the focus of this paper. Figure 1 depicts these relationships.

Starting with the area of computational science, we discover domains like computational fluid dynamics and material science with large problems that require increased performance to be solved or simulated. In search of performance, we look to the area of parallel and distributed computing. Evolving technologies like scalable clustered computing, heterogeneous network computing, and metacomputing can potentially offer substantial, cost-effective increases in application performance. Building software for these domains and computing systems requires improvements over the existing, ad hoc methods of software design and construction used in this community. Looking to software engineering, we find emerging technologies in the area of software architecture that might aid in this endeavor. By defining the software in terms of components and interactions, we are better able to consider its overall structure and topology. But defining a completely generic model might constrain our ability to meet other nonfunctional requirements such as performance or resource utilization. As a result, restricting software development to specific domains is often advantageous, which leads us back to computational science, an area replete with problem domains in search of software solutions.

FIGURE 1. Domain-specific metacomputing for computational science is defined as the confluence of three major areas of computer science.



Collectively, the technologies from each of these areas constitute a new, crosscutting area which we term ***domain-specific metacomputing for computational science***. Each technology seems to contribute to the common goal of high performance computing for computational science, though no one technology fully addresses the scope of the problem. Thus, these three foundational areas—computational science, parallel and distributed computing, and software engineering—yield a number of technologies that appear promising to the area of domain-specific metacomputing for computational science. To better understand this new area, we must identify the area’s basic requirements, determine how the emerging technologies may best address those requirements, and explain how those technologies have evolved and converged to form the area of domain-specific metacomputing for computational science.

CHAPTER 2

Characterizing Domain-Specific Metacomputing for Computational Science

THE AREA of domain-specific metacomputing for computational science crosscuts computational science, parallel and distributed computing, and software engineering. Part of the goal of this paper is to show how select technologies from each of these areas can be used to address certain requirements of this new area. We have already informally introduced and motivated those requirements and briefly mentioned the technologies. The goals of this chapter are to describe more explicitly what those requirements are and the relationship they have to the technologies. This chapter provides the context within which later ones will describe more formally how the area of domain-specific metacomputing for computational science is grounded in three core areas of computer science. In short, we seek to characterize the area of domain-specific metacomputing for computational science.

It should be noted that to improve readability, we occasionally shorten the verbose title "domain-specific metacomputing for computational science" to just "domain-specific metacomputing," "metacomputing for computational science," or just "computational science metacomputing." A shortened title is used only where it sufficiently suggests the aspects of the area most relevant to the discussion at that point.

Requirements

The three requirements to be presented are high performance heterogeneous computing, software design and development, and domain-specificity.

High Performance Heterogeneous Computing

The first requirement of metacomputing for computational science, high performance heterogeneous computing, arises from three different perspectives: limited financial resources to address increasing performance requirements, a desire to increase the utilization of existing resources, and the need to support multiple types of parallelism within a single application.

A large number of computational science problems take the form of simulations, now a respected companion to scientific theory and experimentation [NCO96]. In general, simulations have four dimensions by which they can be characterized: size, resolution, complexity, and length. For example, a simulation of ocean currents might consider a cubic kilometer (size) of ocean at a time, where one of these cube is decomposed into a large collection of small, 1-meter cubes (resolution). The simulation might consider gravitational effects and water temperature, but neglect surface temperature and windspeed (complexity). Finally, when the simulation is finished, it might result in a portrayal of ocean currents over a period of one hour (length).

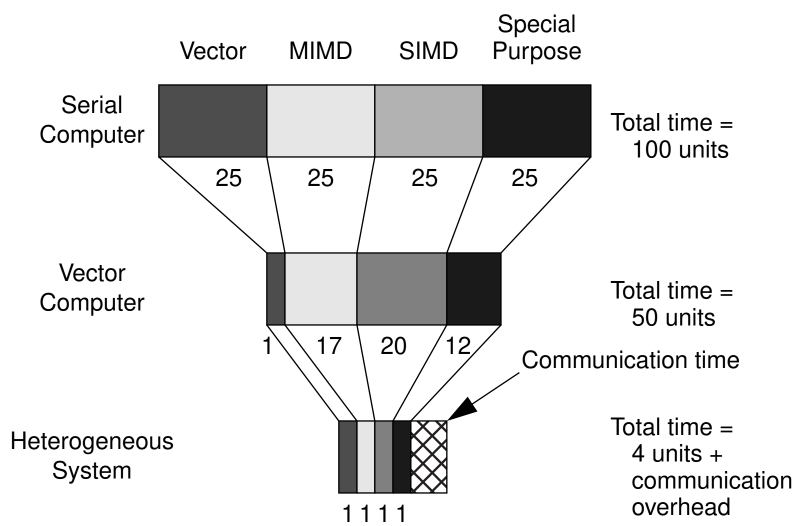
Each of these parameters has a direct effect on the performance of the simulation as a whole. Increasing size, resolution, complexity, or length automatically increases the amount of work that needs to be done. As described in Chapter 1, computational requirements for these problems are, in fact, rapidly increasing. Faced with this situation, three courses of action come to mind.

The first approach is simply to make scientists wait longer and longer for the results of their simulations by not making any improvements to the computational environment they use; the second course of action is to buy new, more powerful supercomputer systems and to port the applications or simulations to them; and the third solution is to better utilize as many of the computational resources available as possible. Of course, a fourth course of action is to buy new machines in addition to harnessing unused, existing computational power. The ASCI program [DOE96] is doing this in a staggered manner, first investing in new machines, and eventually requiring their use in a collective manner.

It is safe to assume that for a large number of problems, the first option is not viable. Furthermore, repeatedly resorting to the second option results in tremendous expense and a collection of different (*i.e.,* heterogeneous) machines, while simultaneously failing to improve the versatility of the system as a whole [KPSW93]. However, as Notkin *et al.* [NHSS87] point out, "heterogeneity is often unavoidable... as evolving needs and resources lead to the acquisition... of diverse hardware...." Eventually, the hardware investment and the cumulative performance potential become large enough that a more general solution is necessary, a solution that makes better and more flexible use of the available resources.

Consider, for example, that many problems exhibit more than one type of parallelism. The image understanding Grand Challenge problem described in [KPSW93] consists of coarse-grained, MIMD-like tasks at a high level with fine-grained, SIMD-like operations applied at a low level. Alternately, different subtasks of a problem may be better suited to different machine architectures [SDA96]. Consider an application, as depicted in Figure 2, that has four distinct phases, each exhibiting a certain type of parallelism. Let us assume the entire execution time on a serial

FIGURE 2. The hypothetical execution on various systems of a code with multiple types of embedded parallelism. (Figure adapted from Khokhar *et al.* [KPSW93].)

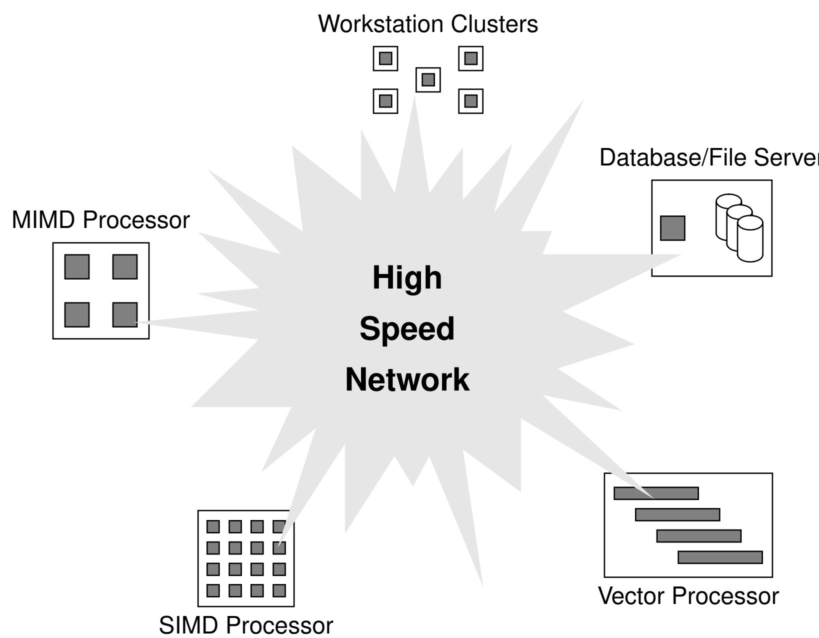


computer to be 100 units. When the code is executed on a (homogeneous) vector processor, the time required for the vector phase is, of course, drastically reduced, but the times for the other phases see only moderate improvements. If a heterogeneous system consisting of a vector machine, a MIMD system, a SIMD computer, and a special-purpose parallel machine is available, we could potentially optimize the total execution time of the code by executing each phase on the most appropriate machine. The trade-off, as noted in the figure, is that we incur a certain amount of communication overhead as we move data between the independent systems. The amount of that communication, of course, can vary widely. The important point is that a given homogeneous system (*i.e.,* one or more machines of the same type), by definition, can not necessarily address the diverse requirements of complex computational science problems.

Hence, heterogeneous computing is desirable from financial, utilitarian, and technical perspectives. A depiction of a possible heterogeneous computing environment is shown in Figure 3, and Khokhar *et al.* [KPSW93] offer the following definition.

> Heterogeneous computing is the well-orchestrated and coordinated effective use of a suite of diverse high-performance machines (including parallel machines) to provide superspeed processing for computationally demanding tasks with diverse computing needs.

FIGURE 3. A heterogeneous computing environment consisting of vector and parallel processors, servers, and workstation clusters. The components are connected by a high-speed networking infrastructure.



The solutions to supporting heterogeneous computing reside in both hardware and software. In terms of hardware, a collection of different computing systems is obviously required. In addition, most researchers agree that a high-speed network connecting the machines is also fundamental [KPSW93, SC92, BM95]. According to Khokhar *et al.* [KPSW93], such a network must have bandwidth capabilities on the order of 1 gigabit/sec to better match computation and communication speeds.

With respect to software, the needs are more diverse. Siegel, Dietz, and Antonio [SDA96] survey the software support needed for heterogeneous computing. They identify several areas requiring software solutions, including code development and generation, debugging and performance tools, operating system support for task manipulation, intermachine data transport, performance monitoring, and administration. In addition, a recent workshop [BM95] concluded that defining a "performance-oriented interface layer between application programs and target systems" could significantly advance the area. Such a layer would enable a similarly broad range of services, including dynamic scheduling, application queries about system state, prediction and measurement, and monitoring and checkpointing. Many of these topics will be explored in more detail in the next chapter. The next section, however, contains a higher-level view of the software development requirements for domain-specific metacomputing.

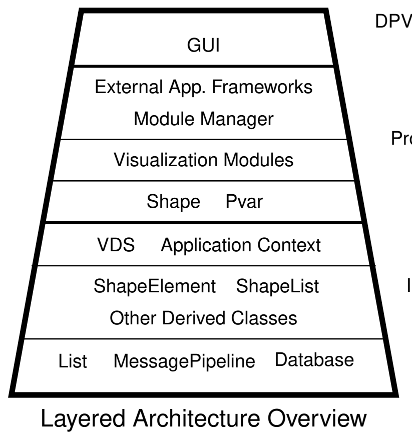
Software Design and Development

Page through the software and tool environments sections of some recent parallel and distributed computing conference proceedings, and many of the papers have a couple features in common. Almost inevitably, the papers have some kind of diagram of the architecture of the proposed tool or environment. Such diagrams typically consist of labelled boxes connected by lines or arrows; occasionally some text is included to explain what the symbols mean. The other common feature of these papers is a prose description of the structure or architecture of the system. Examples of such diagrams and descriptions are recreated in Figure 4.

Indeed, diagrams and descriptions like those in Figure 4 are common in many areas of computer science. And despite being almost completely informal, they can be surprisingly effective. But the labels assigned to the features of a diagram are typically unique to the particular system being described. Furthermore, the terms for general architectural patterns are casually defined (if at all) and are used differently by different authors. Shaw and Garlan [SG96] characterize the situation as follows:

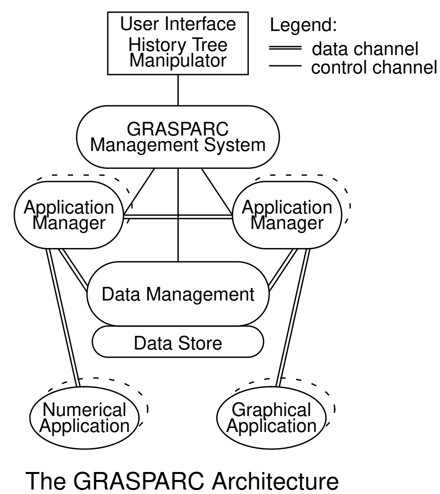
> Unfortunately, diagrams and descriptions are highly ambiguous. At best they rely on common intuitions and past experience to have any meaning at all.

FIGURE 4. The “architectures” of tools and environments are often depicted as box-and-line diagrams which are occasionally accompanied by some descriptive text. These examples each describe visualization tools for parallel computing environments.



“The visualization side of DPV is designed using a layered software architecture....”

Figure and text adapted from Wagner and Bergeron [WB95].



“The next step has been to define a basic system architecture, comprising a framework into which different components or tools can be inserted.”

“The architecture allows the components to be distributed across different processors....”

Figure and text adapted from Brodlie *et al.* [BPWB93].

As we will soon see, the implications of this statement have a profound effect on providing metacomputing support for computational science.

Because the performance demands of computational science problems are so great, collaborations between scientists and expert parallel programmers are essential. As mentioned earlier, a main goal of computational science is to train “hybrid” scientists with expert knowledge in both areas. But, until computational scientists are more numerous, personal collaborations between these previously disjoint communities will be required to get the best application performance. When faced with building entire metacomputing environments for computational science, the success of these collaborations is even more crucial.

In light of Shaw and Garlan's claim that the diagrams and descriptions so commonly used to communicate about software rely on "common intuitions and past experience to have any meaning at all," we must be concerned with the ability of such collaborations to produce robust and useful software. To what extent do physical scientists and computer scientists have the required common intuitions and past experiences to understand each other with respect to this goal?

This is not to say that these two communities *should* have any common intuitions and past experiences. But it does strongly suggest that a common "domain of discourse" [ASWB95] is needed if research collaborations to support metacomputing for computational science are to be successful. The software engineering community has so far relied on folklore, informal models, and unproven theories about software architectures [SG96]. However, even among that community there is a perceived need for more formal definitions and representations of software architecture. Compared to the relatively new community of scientists and computer scientists faced with building domain-specific metacomputing environments together, software engineers are a tightly-knit group. Accordingly, the benefits of improved software design and development techniques to these software projects seem both more imperative and more promising.

Improved software design and development support must include several features. For example, it has long been desirable among software engineers to build systems from so-called "reusable parts," but doing so has proved very difficult [AB96, GAO95, NM95]. Garlan, Allen, and Ockerbloom [GAO95] contend this is because of "mismatches" in the "assumptions a reusable part makes about the structure of the application in which [it] is to appear." Nonetheless, using reusable parts to instantiate different domain-specific metacomputing environments built from a common collection of basic parts is a desirable goal. In addition, building systems from "frameworks" is complementary to using reusable parts. A framework essentially defines a class of applications (or environments) sharing a common architecture and built from a set of generic components [NM95]. Still another goal is to facilitate a high degree of interoperability between the components of a metacomputing environment. Heiler's work [Heil95] seeks to ensure that "a common understanding of the meaning of the requested services and data" exists between the requester and provider. Manola [Mano95] essentially advocates that the use of domain-specific knowledge is central to achieving interoperability. Finally, Purtilo, Snodgrass, and Wolf [PSW91] describe a “software bus,” an abstract software layer through which heterogeneous application (components) exchange data and control.

One of the most primitive manifestations of many of these features is the software library. But, as Rice [Rice96] explains, the software library “still requires a level of expertise beyond the background and skills of the average scientist and engineer....”—a case of lacking common intuitions and past experiences. Furthermore, a software library is subject to the “mismatches” described by Garlan *et al.* [GAO95]. A higher-level, more comprehensive methodology that simultaneously facilitates the collaboration between scientists and computer scientists is required.

Domain-Specificity

The last general requirement of domain-specific metacomputing for computational science is domain-specificity itself. Some view a domain-specific approach to software as a “compromise” position [TTC95]. That is, while not purporting to be a fully general approach, it does avoid the poor software reuse that results from “point solutions” [FGNS96, TTC95] (a particularly prevalent problem in the parallel and distributed computing community). Thus, a domain-specific approach seeks to build families of related systems.

Domain-specificity is characterized by a deep consideration in the design and development of a software system for the context and type of problems to be addressed. While domain-specificity is a somewhat more qualitative requirement than the others mentioned so far, the reasons for it are no less apparent. For example, Springmeyer, Blattner, and Max [SBM92] express its importance with respect to software functionality:

> Domain specific knowledge plays an important role in improving the functionality of software. A designer can apply knowledge of how domain activities are actually practiced to improve the effectiveness and usability of software tools....

Domain-specificity also offers a way to improve the collaboration between scientists and computer scientists, as expressed by Taylor, Tracz, and Coglianese [TTC95]:

> The domain-specific approach affords greater opportunity for user involvement, analyst/developer cooperation, and concern for manageability, productivity, and cost-effectiveness.

Central to the requirement of a domain-specific approach is the development of a "domain model," the goal of which is to standardize the terminology of the problem domain and the descriptions of specific problems to be solved in the domain. Taylor *et al.* [TTC95] claim that domain models "enable effective communication between the developers of a system and those procuring it." In other words, a domain model begins to build up the common intuitions required for effective collaboration between scientists and computer scientists.

Orienting software development around particular domains has the desirable effect of limiting the scope of the software so that a general solution is neither required or attempted. Simultaneously, it increases software reuse, improves software functionality, and facilitates the collaboration between scientists and computer scientists.

In summary, domain-specific metacomputing for computational science has three main requirements: high performance heterogeneous computing, a software design and development methodology, and means for achieving domain-specificity. These requirements and the preceding discussion begin to characterize more formally the area of domain-specific metacomputing for computational science. But requirements are made explicit so that they may be addressed. The next section continues the characterization of the area by describing three core technologies and how they address the requirements just presented.

Technologies

The purpose of this section is to identify the enabling technologies for domain-specific metacomputing for computational science and to explain how they address the requirements set forth in the previous section. Each technology primarily addresses one requirement and has a secondary correspondence to a another requirement. Figure 5 depicts these relationships and will be explained in the following sections. Each technology will be described in extensive detail in the next chapters; this chapter serves only to outline the technologies and their relationships to the area requirements.

Metacomputing

The first core technology that addresses metacomputing for computational science is metacomputing itself. The literature proposes a variety of definitions for metacomputing. The presence of multiple definitions is indicative of the technology's

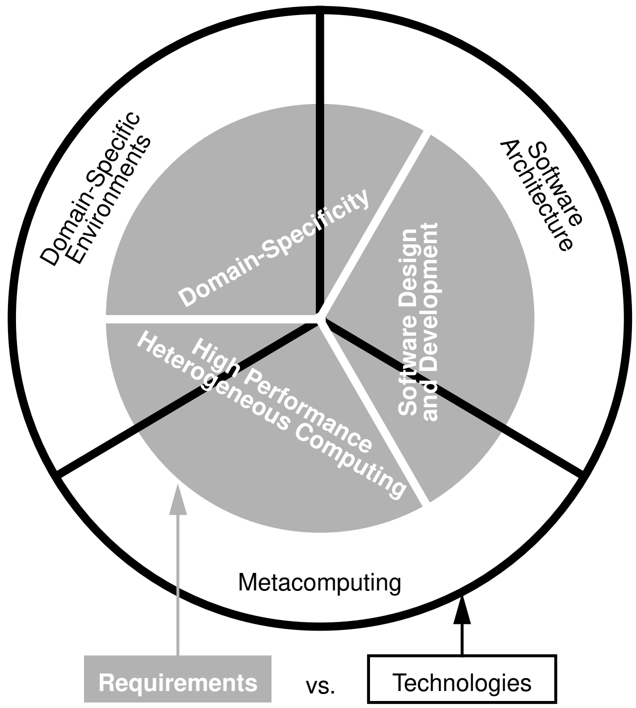


FIGURE 5. The requirements and technologies for domain-specific metacomputing for computational science.

relative youthfulness. Consequently, the definition proposed here is a combination of those found in the literature [KPSW93, SC92, FK96]:

> A *metacomputing environment* is a program execution environment which, primarily through software, supports and simplifies the coordinated use of heterogeneous computing systems, high-speed networks, and other computational resources, possibly located at different geographic sites and in different administrative domains.

Khokhar *et al.* [KPSW93] restrict metacomputing to "computations exhibiting coarse-grained heterogeneity in terms of embedded parallelism." At the other extreme, Smarr and Catlett [SC92] define it very generally as "a network of heterogeneous, computational resources linked by software in such a way that they can be used as easily as a personal computer." Somewhere between these is the definition by Foster and Kesselman [FK96], which defines metacomputers as "execution environments in which high-speed networks are used to connect supercomputers, databases, scientific instruments, and advanced display devices, perhaps located at geographically distributed sites." Certain characteristics of each of these definitions are reflected in the definition we propose, the key aspects of which are that (1) metacomputing is largely a software solution, (2) diverse, distributed computational resources are supported, and (3) the environment is well-integrated and simplifies access to the available resources.

In addition to the obvious goal of improved performance, metacomputing seeks to increase accessibility to supercomputing capabilities and to enable unique computing capabilities not otherwise possible [FK96]. These two goals roughly correspond to the requirement of more cost-effective and flexible heterogeneous computing. To that end, metacomputing primarily addresses the requirement of high performance heterogeneous computing. Metacomputing and heterogeneous computing share an emphasis on utilizing heterogeneous hardware resources such as workstations and parallel/vector supercomputing systems, but metacomputing broadens this somewhat by including other resources like database servers, scientific instruments, graphical displays. In this regard, metacomputing addresses the hardware requirements associated with heterogeneous computing.

But metacomputing is concerned with more than just hardware. In fact, metacomputing emphasizes software as a key, if not the central, part of the solution by focusing on the construction of robust, comprehensive, higher-level environments that support computation and tool construction. For example, Foster and Kesselman [FK96] employ a “toolkit” approach which attempts to integrate a collection of modules, each of which supports higher-level services through well-defined interfaces. Grimshaw and Wulf [GW96], on the other hand, are building an object-oriented class hierarchy and runtime environment to better facilitate customization, extension, and replacement of system functionality by users. The POOMA framework [ABCH95] uses an object-oriented approach to address portability and retargetability requirements. Finally, both POOMA and POET [AM94] utilize frameworks and an object-orientation to facilitate the mapping between a given physical phenomenon and the algorithms and data structures used to model it. While the best approach may not be immediately evident, such efforts are (or soon will be) addressing the software requirements for heterogeneous computing. More generally, these efforts are trying to improve the software design and development process for metacomputing capabilities.

Thus, metacomputing technology primarily applies to the requirement of high performance heterogeneous computing, but also shares a secondary focus on software design and development needs. This relationship is illustrated in Figure 5 in which the area of metacomputing is overlapped mostly by high performance heterogeneous computing and to a lesser extent by software design and development.

Software Architecture

The area of software architecture attempts to formalize the description of the structure and topology of a software system, and to "clarify structural and semantic differences among components and interactions" [SG96]. According to Garlan and Shaw [SG96], software architecture "found its roots in diagrams and informal prose," like those discussed previously and appearing in Figure 4. They arrive at this claim after identifying three design levels for software: architecture, code, and executable. They argue that the code and execution levels are now well-understood as evidenced by the evolution of higher-level languages from machine language, symbolic assemblers, and macro processors. They contend that the architecture level, however, is currently understood mostly in an intuitive manner and lacks uniform syntax and semantics to reason about the diagrams and accompanying prose. Software architecture, as an area, seeks to improve both the understanding and the precision of the architecture level of software design.

A good starting place for software architecture is to consider the variety of organizational styles that software exhibits, such as client-server, pipe-filter, object-oriented, or dataflow. Garlan and Shaw [SG96] make the following observation:

> Systems often exhibit an overall style that follows a well-recognized, though informal, idiomatic pattern.... These styles differ both in the kinds of components they use and in the way those components interact with each other.

They define a common framework within which they can compare the different styles. The framework consists of components (*e.g.,* clients, servers, filters, databases), connectors (*e.g.,* procedure calls, events, protocols, pipes), and a set of constraints that determine how they can be combined. The unique aspect of this model is that it treats the interactions between components (*i.e.,* the connectors) at the same level as the components themselves. At least part of the goal of this technique is to avoid the "mismatches" between components of software [GAO95]. If one abstractly views a metacomputing environments as a collection of interoperating entities (which may be objects, modules, tools, machines, or applications depending on the design and implementation of the particular system), the potential benefits of a design methodology that models both the components of the system as well as the interactions between them become apparent.

The technology of software architecture has been most successfully applied in building domain-specific software architectures (DSSAs) [HPLM95, TTC95]. A DSSA provides an organizational structure for a family of applications (*e.g.,* avionics, mobile robotics, or user interfaces) such that new applications or products can often be created very easily or even automatically [SG96]. DSSAs provide a software engineering methodology that is tailored to a particular domain. In this way, it partially addresses how domain-specificity can be manifested within a software system. With respect to what areas may be amenable to domain-specific architectures, Taylor *et al.* [TTC95] claim

> [The] existence of a large amount of code that is typically successfully scavenged is an indicator that the domain may be mature enough for attempts to regularize it and apply the DSSA approach. Similarly, well-developed and well-used libraries are indications of mature domains.

Clearly, many computational science domains (*e.g.,* computational fluid dynamics, mechanics and structural analysis, etc.) fit these requirements, as evidenced by the large number of scientific software libraries available. Thus, in theory, DSSAs could be created for these areas. Taylor *et al.* [TTC95] outline a five-step process for engineering a DSSA which will be discussed in more detail later. Suffice it to say here that the process includes a high degree of interaction between "domain experts, systems analysts, software engineers, and potential customers" [TTC95]. That interaction is largely facilitated by agreeing upon the terminology, concepts, and requirements of the domain. In other words, domain-specific architectures can aid in addressing the requirement of domain-specificity.

Surprisingly, these and related ideas have not gone completely unexplored in the parallel and distributed computing community. For example, Cuny *et al.* [CDHH96] propose the idea of "domain-specific environments" based on the contention that "solving a particular computational science problem... involves a combination of several technologies with a domain-specific purpose." Their approach emphasizes the collaboration between application scientists and computer scientists. With respect to technology, they seek solutions exhibiting programmability, extensibility, and interoperability to better facilitate the experimental nature of leading-edge scientific applications. In other work, Armstrong and Macfarlane's POET system [AM94] "captures the basic communication paths and the structural aspects of the computational algorithm that are necessary to implement a parallel version of particular scientific problem classes." Concern for application design and a variation of domain-specificity is evident in their approach. In addition, Anglano, Schopf, Wolski, and Berman [ASWB95] propose an abstract, graphical notation for describing heterogeneous applications. Their notation and accompanying annotations are well-defined and precise. But more importantly, this notation acts as a "domain of discourse" between scientists and computer scientists. In essence, they provide a primitive domain-specific software architecture for heterogeneous computing applications.

In summary, the technology of software architecture provides ideas and concepts that can improve the overall software design and development process. In addition, the sub-area of domain-specific software architectures offers methods that could be useful for achieving domain-specificity within a metacomputing environment for computational science applications. The few manifestations of these concepts within the parallel and distributed computing community are of particular interest and will be discussed in more detail later.

Domain-Specific Environments

The last technology central to domain-specific metacomputing for computational science is domain-specific environments. As described earlier, domain-specific environments (DSEs) and problem-solving environments (PSEs) have a unique relationship. Whereas a PSE provides a specific computational capability (*e.g.,* solving partial differential equations) that has potential application in many domains (*e.g.,* structural mechanics, chemical engineering), a DSE seeks to address all of the computational requirements within a single domain. Their relationship is an orthogonal one, as illustrated in Figure 6.

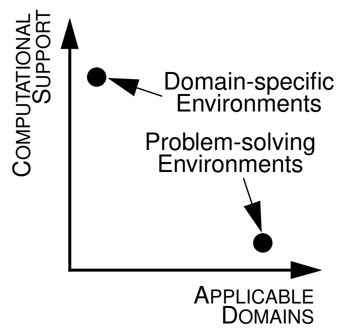


FIGURE 6. Problem-solving environments provide comprehensive support for a single computational service that is applicable across many domains. Domain-specific environments pick a single domain and may provide a collection of diverse computational support.

The main problem in applying PSEs, as originally defined by Gallopoulos *et al.* [GHR94], to large computational science problems, particularly those designated as Grand Challenges, is that the methods and solution techniques are not necessarily well-understood and standardized yet. Similarly, experimental applications typically consist of several component phases that may not be well-integrated. For example, an environment to support seismic tomography [CDHH96] consists of seven distinct steps, each with its own domain-specific requirements. Recently, however, the term “problem-solving environment” has been applied more generally to indicate a well-integrated, comprehensive computational environment. In particular, the ASCI program calls for the development of problem-solving environments that support code development, application execution, and results analysis in a "unified computing and information environment" [DOE96]. This vision of PSEs transcends the original, and bears a greater resemblance to domain-specific environments than to traditional problem-solving environments.

Terminology aside, we consider the creation of domain-specific environments, as characterized below, to be the most relevant example of domain-specific high performance computing support. The GEMS system [BRRM95] seeks broad, comprehensive high performance computing support for the Grand Challenge area of environmental modeling. GEMS is clearly not a problem-solving environment in the original sense of the term; it does not provide a specific computational capability applicable to many domains. Rather, it provides a comprehensive set of capabilities targeted to the domain of environmental modeling, including data management, analysis, visualization, and performance monitoring. The GEMS system was developed through an intense collaboration between software developers and domain experts. The development team concluded that "to produce high-quality software the organizational approach must encourage coordination between the developers [computer scientists] and clients [application scientists] throughout the entire process" [BRRM95]. The work by Cuny *et al.* [CDHH96] has created a similar environment for seismic tomography through a similar collaborative process. Their work focuses on addressing the evolving requirements of experimental scientists as well as building systems in the face of ever-changing computational technology. They conclude that to build DSEs, researchers needs to "develop methods, tools, and infrastructure that utilize capabilities of programmability, extensibility, and interoperability" [CDHH96]. These characteristics are central to the unique role that domain-specific environments play in supporting experimental science. In particular, as scientists develop new ideas about how to solve their problems, new requirements for the DSE arise. Later, we will show in more detail how programmability, extensibility, and interoperability are central to addressing this need.

Domain-specific environments provide the best technology for achieving domain-specificity in the functionality of domain-specific metacomputing environments for computational science. DSEs make two major contributions in this regard. First, they advocate an intense collaboration between computer and application scientists so that the resulting software systems not only address the scientists' needs but do so in a meaningful and useful way. Second, DSEs reveal promising ideas and methods for integrating high performance heterogeneous computing into the experimental process. While this and other sections focus primarily on how problem-solving environments relate to domain-specific environments, we in no way intend to neglect the independent successes of problem-solving environments. Their comprehensive environments, transparent access to high performance computing, and attention to good software engineering practices address aspects of each of the requirements previously identified. The topic of PSEs is raised in the remaining chapters where relevant

In conclusion, when the three technologies of metacomputing, software architecture, and domain-specific environments are examined collectively, they fully address the requirements of high performance heterogeneous computing, software design and development, and domain-specificity. Furthermore, the unique, overlapping manner in which each pair of technologies fully addresses a particular requirement suggests the completeness of these requirements and technologies for domain-specific metacomputing for computational science and positions the area as one with many research challenges.

Conclusion

The focus of this chapter has been on establishing a relationship between the requirements and technologies for domain-specific metacomputing for computational science. While the focus of this paper is primarily on the motivation, characterization, and foundations of this area, it also provides a "case study," if you will, of the more general problem in science—and society—of how technology is understood, applied, integrated, and used.

As we have seen in this chapter, it is often the case that a given technology does not fully address a single requirement. Furthermore, a single technology may have varying degrees of relevance to multiple requirements. Finally, it may take the combination of multiple technologies to address a single requirement. In the case of computational science metacomputing, the collection of technologies and requirements form an interlocking cycle. In general, one can not expect such a nice pattern to arise. And the identification of such patterns, as in this case, may take considerable thought and organization as well as a deep understanding of the requirements and technologies involved.

The next three chapters describe the technologies of metacomputing, software architecture, and domain-specific environments in more detail. Chapter 3 explores the foundations of metacomputing in parallel and distributed computing. Then, supporting research in the areas of software engineering and computational science is described in Chapters 4 and 5, respectively. We conclude in Chapter by synthesizing domain-specific metacomputing for computational science as a research problem and speculating on how it might be pursued.

CHAPTER 3

Foundations in Parallel and Distributed Computing

THE FOUNDATIONS of domain-specific metacomputing for computational science are in parallel and distributed computing. This chapter seeks to motivate the development of metacomputing technology, to identify its key challenges, and to evaluate the major contributions in this area. Our comments primarily focus on metacomputing from the systems and performance perspectives, though we also discuss software organization and implementation methodologies where applicable.

The Convergence of Parallel and Distributed Computing

Parallel computing and distributed computing have, in the past, been mostly distinct areas of research and development. Parallel computing, characterized by the development of closed hardware systems containing multiple processing units, the software and tools for using such systems, and efficient parallel algorithms, has had the exclusive goal of performance. Distributed computing, on the other hand, has been primarily concerned with the sharing of resources among a set of interconnected computers for purposes of performance, reliability, scalability, and security ([CDK94], p. 30). Whereas parallel computing has targeted computationally-intense, numeric applications, distributed computing has focused more on interactive, multi-user computing environments that support a broad range of computing activities.

The advent of high-speed networks and more ubiquitous parallel computing, however, is driving a convergence of parallel and distributed computing, making it increasingly difficult to distinguish between the two. For example, as network speeds increase, clusters of single-processor workstations and personal computers have become a viable platform for parallel computing. Conversely, multiprocessing is evolving from an esoteric technology for achieving *high* performance on expensive, high-end machines to a more wide-spread, mainstream technology for achieving *better* performance in distributed computing environments. An excellent example of this convergence is the rising popularity of shared-memory multiprocessors (SMPs) for general purpose computing. While they are clearly *parallel* computers, SMPs are commonly used as server machines in distributed systems ([CDK94], p. 45-46). The modest parallelism of most SMPs allows these servers to respond simultaneously to multiple service requests, reducing the likelihood that the service becomes a bottleneck. Furthermore, SMP's support of shared-memory allows the efficient implementation of interprocess communication and processor allocation on such machines. While advantageous for distributed systems, shared-memory multiprocessors also have tremendous performance potential. In fact, the 3-teraflop system solicited by the Accelerated Strategic Computing Initiative (ASCI) program will be a cluster of SMP machines [Wood96, LLNL97].

While parallel and distributed computing each maintain their own, unique research directions, their convergence gives rise to many new research challenges. One of these challenges, metacomputing, is the focus of this chapter. Foster and Kesselman [FK96] effectively describe some of the relationships between metacomputing and parallel and distributed computing:

> Metacomputers have much in common with both distributed and parallel systems, yet also differ from these two architectures in important ways. Like a distributed system, a networked supercomputer must integrate resources of widely varying capabilities, connected by potentially unreliable networks and often located in different administrative domains. However, the need for high performance can require programming models and interfaces radically different from those used in distributed systems. As in parallel computing, metacomputing applications often need to schedule communications carefully to meet performance requirements. However, the heterogeneous and dynamic nature of metacomputing systems limits the applicability of parallel computing tools and techniques.

Thus, while metacomputing can certainly build on the foundations of parallel and distributed computing, significant advances in infrastructure, methods, and tools are still required. Because our agenda is somewhat broad, it is beyond the scope of this work to explore all of the research challenges posed by metacomputing. We seek to reveal and describe the important role that metacomputing can play in computational science. To this end, we briefly provide some foundations for the area and then survey representative examples of work most relevant to domain-specific metacomputing for computational science. This is followed by discussions of some specific parallel and distributed computing issues.

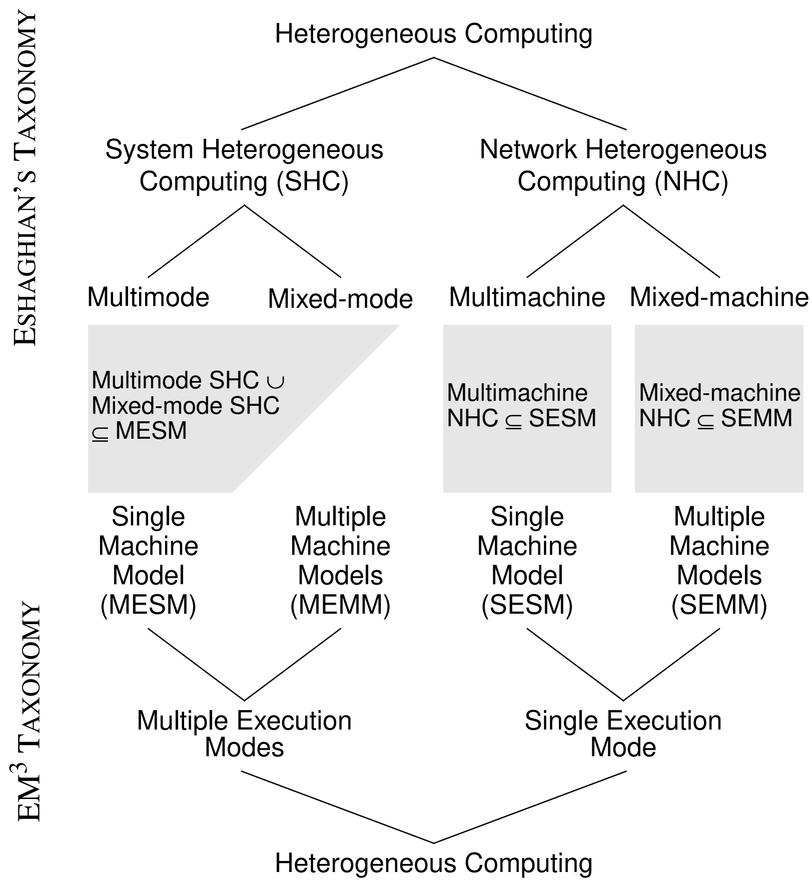
Heterogeneous Computing

Heterogeneous computing (HC) exists at the convergence of parallel and distributed computing and provides many of the foundations of metacomputing. As defined previously, heterogeneous computing seeks to coordinate the execution of a collection of diverse computers to execute computationally demanding tasks. HC inherits parallel computing's concern for performance, but it must also exhibit a strong consideration for certain distributed systems issues. For example, if the proper task allocation for the program in Figure 2 in Chapter 2 is not made, little or no performance gain is realized.

To better distinguish between the different types of HC, various characterizations and taxonomies have been proposed. Heterogeneity can manifest itself in many ways. *Temporal heterogeneity* occurs when a system can execute in different execution modes (*e.g.,* SIMD, MIMD) at different times, while *spatial heterogeneity* occurs when different machines in a system execute in different modes at the same time [ETM95]. In an attempt to capture the distinction between these two types of heterogeneity, Eshaghian ([Esha96], p. 2-4) proposes the taxonomy in the upper part of Figure 7. In *system heterogeneous computing (SHC)*, a single multiprocessor computer executes one or more tasks in both SIMD and MIMD modes. In multimode SHC, SIMD and MIMD execution can take place simultaneously, but in mixed-mode SHC, the execution mode of the whole machine is switched between SIMD and MIMD. SHC machines tend to be specialized systems (*e.g.,* the Image Understanding Architecture ([Esha96], p. 67-97)) and are not intended for general purpose numerical computation. On the other hand, *network heterogeneous computing (NHC)* consists of a collection of interconnected machines that can execute one more tasks concurrently. When all the machines in the system are identical, it is called multimachine NHC. When at least one of the machines is different, mixed-machine NHC occurs.

Unfortunately, Eshaghian's classification is not complete. For example, how does one classify a heterogeneous system that consists of several identical workstations and a single multimode computer that supports simultaneous SIMD and MIMD

FIGURE 7. The relationship between Eshaghian's taxonomy and the EM³ taxonomy of heterogeneous computing



computations? The problem is that the classifications within SHC and NHC consider different features. In particular, NHC considers machine type, but neglects the number of execution modes supported. SHC does the exact opposite.

The EM^3 taxonomy [ETM95] offers an improvement over Eshaghian's approach by independently considering two dimensions of a heterogeneous computing system: *execution modes* and *machine models*. Execution mode represents the type of parallelism supported by a machine (*e.g.,* vector, SIMD, MIMD), while machine model considers the architecture and performance of the machines in the HC system. In the spirit of Flynn's Taxonomy ([HP90], p. 572), each dimension is classified as either single or multiple, yielding the categories shown in the lower part of Figure

7. To improve the comparison with Eshaghian's taxonomy, the EM^3 taxonomy has been arranged as an inverted tree (instead of a simple 2x2 grid).

In the EM^3 taxonomy, SESM systems consist of a single machine architecture which support a single execution mode. SESM systems include uniprocessors as well as parallel or distributed systems that support a single execution mode and are composed of identical processors or machines. SEMM-class systems include, for example, networks of different workstations that each support the same execution mode. Finally, MESM and MEMM systems consist of machines that individually can support multiple modes of parallelism. MESM-class systems consist of a single machine type, while MEMM-class systems are composed of different types of computers.

The relationships between the Eshaghian and EM^3 taxonomies is shown in the middle of Figure 7. While Eshaghian's taxonomy does identify how multiple execution modes are manifested within a single machine (*i.e.,* multimode or mixed-mode), the resulting classes of machines do not fully address the range of possibilities in a heterogeneous system. The Eshaghian taxonomy makes two (implicit) assumptions that limit its usefulness. First, it assumes that if a HC system is composed of a single machine, that machine must exhibit multiple execution modes. Second, it assumes that if multiple machines are used in a HC system, none of the individual machines are capable of multiple execution modes. The first assumption is at least partially valid in that without it, non-heterogeneous systems are possible (*e.g.,* a single workstation or a shared-memory multiprocessor). However, the second assumption is clearly limiting in that it certainly omits valid heterogeneous systems. In particular, MEMM-class machines do not have an obvious classification in Eshaghian's taxonomy.

Yet, MEMM-class systems (*i.e.,* heterogeneous systems that support multiple execution modes on multiple machine types) represent the most flexible metacomputing environment for computational science. Metacomputing is less concerned with specialized architectures capable of supporting multiple modes of parallelism. Such machines are usually research prototypes ([Esha96], p. 34-43) and require non-standard, specialized programming languages ([Esha96], p. 43-49). These characteristics are not consistent with the goals of metacomputing. Rather, metacomputing seeks to support multiple forms of embedded parallelism by allocating tasks to an appropriate machine if available. Furthermore, support of a unified programming model is simplified when the machines that make up the heterogeneous system support standard programming languages, system libraries, and tools.

Metacomputing Challenges

Earlier, metacomputing was distinguished from heterogeneous computing as providing larger-scale and more cohesive computational support. Thus, metacomputing faces all of the same challenges as heterogeneous computing. While general consideration of heterogeneity dates back to 1987 [NHSS87] and specific methods (*e.g.,* remote procedure call) for accommodating heterogeneity date back much further, it was not until the early 1990s that the potential advantages of heterogeneous computing came to light. Many of those advantages have already been discussed in earlier chapters.

But along with those advantages come several challenges. Khokhar *et al.* [KPSW93] identify several that are part of the process of heterogeneous computing. First is algorithm design. Heterogeneous computing opens up unique opportunities for algorithms. Designers must consider the types of machines available in a given heterogeneous system, alternate solution methods to application subproblems, and the cost of network communication. Next, code-type profiling attempts to identify the types of parallelism (*e.g.,* vectorizable, SIMD/MIMD parallel, scalar, etc.) exhibited by the phases of an application and to estimate the execution time of each. Once profiled, analytical benchmarking indicates which machines are most appropriate to each phase of the code. Following this, the code must be partitioned and mapped onto the heterogeneous system. Matching code types to machines and dealing with code and data conversions among different machines complicate these tasks. An appropriate selection of machines must then be made. (A given application doesn't necessarily use all the available machines.) Scheduling of the application modules takes place at several levels (*e.g.,* whole system, jobs, active sets, and processes). During execution, issues of synchronization must be addressed, including coordination between the senders and receivers of messages and controlling access to shared objects. Attention must be given to interconnection requirements so that computation and communication speeds are matched. Finally, programming environments must support portable and parallel languages, cross-parallel compilers, parallel debuggers, configuration management, and performance evaluation.

Siegel *et al.* [SDA96] identify a similar three-stage process for building applications for a heterogeneous system. The process begins by comparing the computational requirements of the application with the machines available in the HC system. Next, task profiling and analytical benchmarking break the application into homogeneous subtasks (with respect to their computational requirements) and quantifies how each of the machines may perform on the tasks. In the third stage, execution times for each task are derived, including consideration of potential communication costs, system load, and network traffic. The tasks are then assigned to machines according to an execution schedule. In the last stage, the application is executed. As the system is monitored, rescheduling or relocating certain tasks may be necessary.

These characterizations of heterogeneous computing are intended to act as a backdrop for the remainder of this chapter as the major contributions to metacomputing are surveyed. In addition, we revisit some of these topics when we discuss particular issues and challenges for metacomputing later in this chapter.

Metacomputing Research

The First Metacomputer: The NCSA Metacomputer

The notion of metacomputing was first popularized by Smarr and Catlett in a 1992 article [SC92]. Oddly enough, their article appeared among a collection of work appearing at the ACM SIGGRAPH ’92 Annual International Conference on Computer Graphics and Interactive Techniques. What, one may ask, does metacomputing have to do with computer graphics? Smarr and Catlett were primarily concerned with the use of scientific visualization in networked environments for purposes of computational steering and remote collaboration. The conference showcased a collection of projects representing the “state of the art in networked visual computational science” [Hart92]. Among those systems was the NCSA metacomputer.

System Description. Smarr and Catlett envisioned a metacomputer that could be used as easily as a personal computer (PC) and supported this vision with a simple analogy. One can think of a PC as a “minimetacomputer” consisting of a general purpose processor, a floating-point processor, an I/O “computer,” audio and graphics chips, etc. “Like the metacomputer, the minimetacomputer is a heterogeneous environment of computing engines connected by communication links” [SC92]. Khokhar *et al.* [KPSW93] take a similar view of heterogeneous computing in general, also noting the development and subsequent widespread use of specialized processors for I/O and floating-point processing. Woodward [Wood96], too, suggests a macro view of clustered SMPs where a single SMP and its shared memory play the roles of a CPU and a CPU cache, respectively. The replication of such views is certainly suggestive of the evolution of a new level in the computing hierarchy.

Suppose that for a certain application, a user requires a desktop workstation, a remote supercomputer, a mainframe supporting mass storage, and a specialized graphics terminal. Smarr and Catlett [SC92] explain the relevance of metacomputing to such an environment:

> Some users have worked in this environment for the past decade, using adhoc, custom solutions, providing specific capabilities at best, in most cases moving data and porting applications by hand from machine to machine. The goal of building a metacomputer is elimination of the drudgery involved in carrying out a project on such a diverse collection of computer systems.

Smarr and Catlett identify three stages in building metacomputer systems. The first stage involves the interconnection of hardware resources by high-speed networks and the software integration required to make the environment seamless (*e.g.,* distributed file systems, user access to remote resources). The next stage focuses on splitting up application codes so they can run on heterogeneous systems—again, primarily a software solution. Finally, Smarr and Catlett call for access to a "transparent national network" that addresses issues of administration, file systems, security, and accounting [SC92].

Smarr and Catlett also identify three general capabilities supported by metacomputing. As mentioned earlier, theoretical simulation allows scientists to perform numerical experiments in the artificial world of a high-performance computer system unconstrained by time and space. Metacomputing can also support access to and control of scientific instruments and sensors, enabling near real-time observation and numerical analysis. Finally, data navigation in a metacomputing environment supports remote access to large data stores (created in part by the previous two capabilities) and the computational power necessary to present and explore them interactively. Similar application areas are the targets of other metacomputing work [FK96, FT96, GW96] and distinguish metacomputing from heterogeneous computing.

Evaluation. Smarr and Catlett can be largely credited with first promoting the idea of metacomputing. They describe one of the first real metacomputing environments, which was built at the National Center for Supercomputing Applications (NCSA) [SC92]. The NCSA metacomputer was truly diverse, integrating massively parallel Thinking Machine CM-2 and CM-5 machines, vector processors like Cray-2, Cray Y-MP, and Convex systems, and superscalar IBM RS/6000 and Silicon Graphics VGX multiprocessors. The systems were interconnected by a gigabit local area network (LAN) with access to file servers, tape robots, and RAID systems. In addition, high-performance visualization and other multimedia capabilities were supported. In fact, this reveals one of metacomputing's distinguishing factors. Whereas heterogeneous computing integrates a collection of diverse computers, metacomputing environments seek more comprehensive computational support by providing access to file systems, mass storage, visualization, and other I/O devices. The NCSA metacomputer was one of the first to exemplify this notion.

Furthermore, the NCSA metacomputer was instrumental in both identifying the importance of software support for metacomputing and providing an example of such support. But NCSA built their software environment primarily from existing software technology like Parallel Virtual Machine (PVM) [DGMS93, Sund96]. PVM is discussed in more detail in the next section, but as earlier chapters predicted, the use of existing software technology proved difficult. As a result, current metacomputing projects appear to be less ambitious in this regard [FGNS96, FK96]. Some projects originally had the intention of using existing technology [GNW95], but eventually abandoned that goal altogether [GW96]. Even the original developers of PVM have found it necessary to build higher-level software to better support appropriate metacomputing abstractions [DGMS93, BDGM96, Sund96]. We explore this issue in more detail as we survey each of the projects below.

Metacomputing From Existing Technology: PVM and HeNCE

Parallel Virtual Machine, better known simply as PVM, is an important exception to the trend of unadopted tool technology in the parallel and distributed computing community [Panc94]. PVM is largely responsible for bringing network-based parallel computing to the masses. That is, PVM was one of the first software systems that allowed a heterogeneous collection of machines, including a simple collection of workstations, to work together on a single computational task. Moreover, it is a relatively small system that is easily installed by the user [DGMS93]. Originally released in 1991, PVM has been consistently updated and improved.

System Description. PVM is essentially a message-passing system. An application consists of several tasks, each of which can exploit the machine architecture best suited to its solution. (An example of this approach appeared earlier in Figure 2.) Each user creates their own virtual machine from the available computational and network resources. Logically, this virtual machine is a single, large distributed memory computer, and the tasks running on the virtual machine must communicate with each other through messages with explicitly typed data. PVM handles all data format conversions that may be required between two machines that use different data representations. In this way, PVM supports heterogeneity at three different levels: application, machine, and network [DGMS93].

With respect to application-level heterogeneity, tasks can be assigned to computational resources in one of three ways. In the transparent mode, PVM attempts to locate tasks on the most appropriate machine. In architecture-dependent mode, the user assigns an architecture type to a task and relies on PVM to find an appropriate machine of that type to execute the task. Finally, in machine-specific mode the user actually assigns particular machines to each task. PVM routines also support process initiation and termination as well as various communication and synchronization primitives (*e.g.,* broadcast, barrier, rendezvous).

Thus, PVM provides many of the low-level mechanisms necessary for heterogeneous computing, but it falls considerably short of supporting the vision of metacomputing for computational science. The developers, of course, recognize this. In an effort to "make network computing accessible to scientists and engineers," they are constructing the Heterogeneous Network Computing Environment (HeNCE) on top of PVM.

The goal of HeNCE is to simplify the tasks of "writing, compiling, running, debugging, and analyzing programs on a heterogeneous network" [BDGM96]. HeNCE consists of several tools: compose, configuration, build, execute, and trace. Each of these is briefly described here. Using the *compose* tool, the user specifies the parallelism in their application as a graph, where nodes represent subroutines and arcs represent control and data flow. The *configuration* tool is used to create the virtual parallel machine by indicating which hosts are to participate, with what priority, and for which tasks. The *build* tool generates the parallel program, compiles it for each node type in the virtual machine, and installs the executables on each machine. The *execute* tool starts up PVM and runs the program, allocating tasks per the information given to the configuration tool. Finally, the *trace* tool provides some rudimentary visualizations of network status and the graph representation of the parallel program.

Obviously, the environment has fairly limited capabilities in terms of the range of parallel programs it can create, and for the most part, the successful application of HeNCE to experimental computational science problems seems unlikely. Nonetheless, the developers have done something fairly unique by creating a higher-level abstraction to parallel computing in a heterogeneous environment. The programmer (scientist?) need only write the core subroutines for a computation or simulation. The rest of the application logic is specified through the graph abstraction, hiding the underlying PVM calls and much of the program's control structure implementation from the user.

Evaluation. As mentioned earlier, PVM was among the first software enabling any form of network computing. In part, it can be credited with establishing network-based computing as a valid alternative to dedicated supercomputers. It remains a practical, immediately available solution that targets small- to medium-scale heterogeneous systems typically consisting of less than one hundred hosts. In many ways, PVM has revealed the limitations that must be addressed by larger-scale systems (like those described below). For example, PVM requires the user to manage message-passing explicitly, to know what machines are available and what their characteristics are, and to have login privileges on each of them.

PVM places an emphasis on practicality and on providing useful technology now. That spirit is shared by HeNCE, which clearly can not solve all application development problems for PVM, though it may provide an incremental improvement for a certain class of programmers. To their credit, the developers of PVM and HeNCE do not claim to provide metacomputing capabilities. Including their work in this discussion is, perhaps, to overstate their goals. Nonetheless, PVM is representative of a software framework that is successfully following the evolution of high performance computing. More recent work is extending PVM to support threads, client-server computing, remote procedure call, agent-based computing, and web-based computing [Sund96]. In general, PVM's interface has become a de facto standard for message-passing and, ironically enough, will likely be supported by future metacomputing systems [GW96].

A Metacomputing Toolkit: The Globus Project

In the discussion so far, a number of implicit issues have been raised that are important to metacomputing. Among those are *resource location*, *resource allocation*, and *authentication*. Resource location is the determination of what computational and network resources are available in the heterogeneous computing environment. Resource allocation is the assignment of machines to tasks. And authentication is the verification of access privileges to a given machine.

In the NCSA metacomputer, all of these were addressed directly by the user. The resource set was fixed; resource allocation was done manually for each application; and access to each resource was assumed. PVM and HeNCE provide an incremental improvement in the area of resource allocation by supporting transparent and architecture-dependent task assignments, but resource location and authentication are still assumed. The limited functionality in these areas is, in some cases, acceptable for small-scale heterogeneous systems, but for systems with more than ten or twenty hosts, keeping track of what each host is named, which hosts are operable at any given time, and the login names and passwords for each host becomes burdensome. In general, we refer to these issues as part of the configuration problem.

The issues identified by Khokhar *et al.* [KPSW93] and Siegel *et al.* [SDA96] described earlier do not necessarily dictate how such requirements should be addressed. They simply claim that such issues must be addressed. The vision of supporting metacomputing environments with access to hundreds or even thousands of machines certainly implies that the user can not be relied upon to address issues of configuration. To provide such support is the responsibility of, indeed the very reason for, a full-scale metacomputing environment.

The Globus project is one of two national-scale metacomputing projects. The other project, Legion, is discussed later. Together, these two projects provide the best examples of metacomputing environments, though both are still under active development at the time of this writing. Even so, much of the design and some implementation has been described in the literature. We present an overview of these projects as well as comments about their strengths and weaknesses. Additional observations about these projects are made as we discuss specific metacomputing issues and challenges later.

System Description. The goal of Globus is to confront directly the problems of configuration and performance optimization by first creating a "metacomputing infrastructure toolkit" that provides a set of basic capabilities and interfaces in these areas [FK96]. These components define a "metacomputing abstract machine" upon which a range of higher level services, tools, and applications may be built. By definition, their approach results in a software layer best classified as *middleware*, leaving the creation of higher-level services to future work. Central to Globus is consideration of the potentially unreliable nature of metacomputing environments. To this end, they view applications that run in metacomputing environments as possessing the ability to configure themselves to a given execution environment and then to adapt to subsequent changes in that environment. But clearly, the functionality to accomplish this is not the application programmer's responsibility; it must originate from within the metacomputing environment itself. Later, we discuss one component system working toward this goal.

Foster and Kesselman [FK96] make five general observations about future metacomputing systems, including Globus. Even though most experiments have been on small-scale environments with the largest consisting of machines at 17 different sites [FGNS96], environments must support both *scale and selection*. When widely deployed, metacomputing technology will support access to hundreds or thousands of machines; users will want to select from those machines based on criteria such as connectivity, cost, security, and reliability. Metacomputers must handle *heterogeneity at multiple levels*, including physical devices, network characteristics, system software, and scheduling and usage policies. Whereas traditional applications could make assumptions about machine and network characteristics, metacomputer applications must be able to handle the *unpredictable structure* of diverse and/or dynamically constructed metacomputing environments. Similarly, *dynamic and unpredictable behavior* may be caused by sharing of metacomputing environments, making guaranteed quality of service and exclusive, predictable access to resources difficult. Finally, security and authentication are complicated by metacomputing resources residing in *multiple administrative domains*. Foster and Kesselman point out that all of these issues have a common requirement of real-time information about the system structure and state. A metacomputing system must use that information to make configuration decisions and be notified when the information changes. Meeting these real-time system information requirements is one of the challenges we address below.

The Globus toolkit is comprised of a set of modules. Each module defines an interface through which higher-level services gain access to the module's functionality. Currently, six modules have been conceived: resource location and allocation, communications, unified resource information service, authentication interface, process creation, and data access [FK96]. Additional details about some of these modules appear below.

Evaluation. It is difficult to evaluate a system that is still under active development. Consequently, our comments here relate more to the design and implementation approach being used by Globus. This is still constructive, though, since we have identified software design and development methodologies as a key contribution of metacomputing technology.

Compared to PVM, the Globus project is taking a very different approach to the metacomputing problem. PVM and HeNCE are concerned with extending an existing system originally intended for a somewhat different purpose (*i.e.,* computing in small- to medium-scale heterogeneous environments). Globus, on the other hand, is building a "metacomputing toolkit" and addressing the metacomputing problem in a bottom-up manner by developing low-level mechanisms and support that can be used to implement higher-level services [FK96]. The Globus project uses existing technology where applicable, extends it when practical, and builds their own services when necessary. Most low-level services appear to require new implementations, though some mid- and higher-level services will likely be more compatible with existing systems.

A notable aspect of Globus that distinguishes it from other projects is the desire to let potential metacomputing applications drive the identification of, and solutions to, technical problems. Through the use of targeted testbeds (one of which is discussed later), the Globus project is hoping to improve their ability to predict the requirements of future metacomputing applications.

What remains to be seen is whether the low-level toolkit components currently being built will be able to address these yet unknown requirements. This reveals one of the potential problems with a bottom-up approach: the resulting system may not be sufficiently general. This is particularly important since Globus is trying to build just that—a general metacomputing environment. In the next section, we see how the Globus approach compares to the design of the Legion metacomputing system.

Distributed Object Metacomputing: The Legion Project

The object-oriented computing paradigm promotes such characteristics as modularity, abstraction, reuse, and programming-in-the-large [RT96]. In many cases, it can reduce the amount of work that a programmer has to do and result in increased productivity [MN96]. The object-oriented paradigm has more recently been extended to the realm of distributed applications in hopes of thwarting the so-called "software crisis" [Cox90].

The rapid increase in internetworked computers during the 1990s has resulted in challenging new requirements for information processing in the areas of interconnection and interoperability [NWM93]. In response to this challenge, distributed object systems such as CORBA [OMG95] and DCE [OG96] have emerged, and object-orientation has proven itself a desirable means of managing the complexity of large-scale distributed systems [NWM93]. However, most of these systems do not target high-performance computing and do not support parallel programming [GW96]. In an effort to bring the advantages of distributed objects to metacomputing, the Legion project is attempting to build an infrastructure capable of supporting high-performance access to millions of hosts based on a solid, object-oriented conceptual model [GW96].

System Description . The Legion vision is nothing less than grandiose, and it forces one to consider seriously how future metacomputing environments will *actually* manifest themselves and be used. Grimshaw and Wulf [GW96] describe Legion as follows:

> Our vision of Legion is of a system consisting of millions of hosts and trillions of objects co-existing in a loose confederation tied together with high-speed links....
>
> It is Legion's responsibility to support the abstraction presented to the user, to transparently schedule application components on processors, manage data migration, caching, transfer, and coercion, detect and manage faults, and ensure that the user's data and physical resources are adequately protected.

One may argue that the Legion vision of computing is actually a fairly simple conceptual extension to the current state of internetworking. With millions of computers connected to the global Internet, it is not a big conceptual leap to imagine those computers participating in a global pool of computational resources. But, as we have argued previously, this relatively small conceptual jump requires enormous advances in software technology. The Legion project recognizes these requirements perhaps more than any other metacomputing project, though this was not always the case. Legion started as an effort to *extend* the Mentat parallel programming language to a "campus-wide virtual computer" [GNW95]. Now, the researchers are designing "from the ground up so that the resulting system has a clean coherent architecture, rather than a patchwork of modifications based on a solution for a different problem" [GW96]. This realization does not bode well if the developers of PVM and HeNCE have any real intentions of supporting large-scale metacomputing.

In this spirit, the Legion object model is guided by the philosophy that the developers can not design a system that satisfies the needs of every user. To this end, the Legion team is not attempting to provide specific solutions to all of metacomputing's problems. Rather, "users should be able, whenever possible, to select both the kind and level of functionality, and make their own trade-offs between function and cost" [GW96]. The goal of Legion, then, is primarily to specify functionality, not implementation. As the developers state, "the [Legion] core will consist of extensible, replaceable components" [GW96]. Legion provides base implementations of core functionality, but users can replace them at will. Object-oriented classes and inheritance provide a natural means of supporting this goal [LG96].

As mentioned above, a key challenge for Legion is to deliver high performance. Performance has not been a primary objective for distributed object systems, and this is part of the reason that the Legion object system was (re)designed from scratch—so that performance considerations could influence the development of the system. The authors point to two methods for achieving high performance. First, load-distributing [SKS92] and load-sharing [ZWZD92] systems can be used to exploit resources throughout the metacomputing system. Second, Legion supports parallel execution. As a project emerging from the parallel processing community, parallelism is a major focus of the project. Four means of achieving parallelism are available: wrapping existing parallel codes in Legion object wrappers, supporting parallel method invocation and object management, exposing the Legion run-time interface to parallel language and toolkit builders, and by supporting popular message-passing APIs like PVM.

Evaluation. The goal of the Legion project is to provide a cohesive middleware system for metacomputing environments that supports a wide variety of tools, languages, and computation models, while simultaneously allowing diverse security, fault-tolerance, replication, resource management, and scheduling policies. The project is very large-scale in its goals and broad in its vision of future computing environments.

The primary strengths of the Legion system follow a common theme that spans from high-level design to low-level implementation. At the highest level, Legion's greatest strength is its philosophy toward addressing the metacomputing problem. At the heart of that philosophy is the tenet that they "cannot design a system that will satisfy every user's needs" [GW96]. Adherence to this principle will result in a system that can grow with the relatively young area of metacomputing. Along with this philosophy, the group has identified several constraints that restrict how the higher-level design can be mapped to lower-level implementations of the system. These include not replacing host operating systems, not legislating changes to interconnection networks, and not requiring Legion to run with special privileges. Like the Legion philosophy, these constraints go a long way toward creating a flexible and usable system. At the next level, the Legion group has adopted a framework approach to implementing their system. A framework specifies the interfaces to core components of the system and may provide default components that can be easily replaced. Finally, Legion takes an object-oriented approach to the actual implementation of the system. Object-orientation provides excellent mechanisms for abstracting machine-specific, operating system-specific, and network-specific characteristics of the computational resources within the metacomputing system. Simultaneously, it provides a robust model on which new components and services may be built. In summary, the main strength of Legion is a thorough appreciation of the need for a flexible and adaptive system. This appreciation is present from the high-level philosophy guiding the project all the way down to the low-level implementation of its objects.

Ironically, the primary weaknesses of Legion can be derived from the very properties that give it strengths. Consider the Legion philosophy that the project cannot possibly build a system to satisfy every user's needs. The Legion builders have actually stated that "the Legion project cannot... build all of 'Legion'" [Grim96]. In some ways, this is a fundamental flaw—albeit an unavoidable one perhaps—in the Legion project. That is, as they seek to maximize the generality and flexibility of the system, they simultaneously minimize what they can actually implement and even specify in the form of interfaces, which in turn complicates the user's ability to customize the system to their own needs. Similarly, in the same way that they can not anticipate every user's needs, the constraints they have identified will likely prove insufficient as the field of metacomputing evolves. While object orientation has some clear advantages, it also has some potential disadvantages. With respect to usability, much of the system is left up to other developers and/or users to implement. Are users willing to write their own security modules? Their own scheduling objects? With respect to performance, it is unclear how much overhead object representations and interactions impose on Legion applications. Similarly, delivering a high-performance global name space is a very challenging problem. Finally, it remains to be seen whether metacomputing will thrive at the scale envisioned by the developers. Do we need support literally capable of managing millions of hosts at once? Or will metacomputing practically be applied in a more modest fashion? In summary, the weaknesses of the Legion system are derived from its extremely general approach and broad scale.

In comparing Globus [FK96] and Legion, two major differences are apparent. First, with respect to determining the requirements of metacomputing, the Globus project recognizes the importance of testbeds as a means of discovering and learning about metacomputing. Legion, on the other hand, is more concerned with building a totally flexible system that maximizes its ability to adapt to future requirements, whatever those may be. In design, Globus is taking a bottom-up approach and trying to use, or at least incorporate, existing technology when possible. Legion's object-oriented view of metacomputing is naturally top-down, and almost all functionality must be reimplemented for that environment.

Clearly, the decisions that the Legion team have made in the design and implementation of their system have trade-offs. This is not surprising. Legion is attempting to tackle a very large problem, and it has already made significant contributions to the consideration, design, and implementation of metacomputing software infrastructure. It has also spurred research into several more refined and focused areas like resource discovery and scheduling.

Issues and Challenges

From the comments above, it should be clear that metacomputing is a young area. The software that is currently available to support metacomputing-like functionality (*e.g.,* PVM and HeNCE) falls short in many respects. At the same time, those projects that are attempting to address the metacomputing problem more completely (*e.g.,* Globus and Legion) are still under active development. Nonetheless, in examining the current state of metacomputing, we have touched upon several more specific issues and challenges that metacomputing must address. Many of those issues are being addressed by other research groups and have close ties to more general research in parallel and distributed computing. This section discusses some of these efforts. We begin with a brief description of a metacomputing testbed. This is followed by sections discussing real-time system information, application scheduling, and global name space.

A Metacomputing Testbed

Both Legion and Globus have recognized that they cannot fully predict the potential applications and requirements of future metacomputing environments. Interestingly, each group has responded in a different way. Legion is starting from scratch with a completely object-oriented approach to facilitate replacing and/or extending default implementations with customized ones. In this way, unpredicted functionality can be more easily incorporated into the system.

The Globus team of researchers has taken a very different approach. To improve their ability to predict the future, they are creating a series of large-scale testbeds. From these, they hope to learn more about the applications, technical problems, and open issues that must be addressed so that the area can continue to evolve. The first of these testbeds, the I-WAY, was created in 1995 and focused on exploring the types of applications that might be deployed in a metacomputing environment.[1]

The I-WAY. The Information-Wide Area Year, or I-WAY, was a metacomputing testbed "in which innovative high-performance and geographically distributed applications could be deployed" [FGNS96]. The organizers hoped that by focusing on applications, the I-WAY would reveal the "critical technical problems" that must

1. The second testbed, Globus Ubiquitous Supercomputing Testbed (GUSTO), was scheduled to begin in late 1996, but no literature on the project has yet appeared. Foster and Kesselman [FK96] suggest that this testbed will focus more on the computer science problems of metacomputing, such as authentication, scheduling, communication, and information.

be solved and allow researchers to “gain insights into the suitability of different candidate solutions” [FGNS96].

Essentially, the I-WAY was a huge experiment in metacomputing. The subjects of the experiment were the more-than-60 applications selected by competitive proposal. These applications fell into three categories: immersive virtual environments coupled with remote supercomputers, databases, or scientific instruments; multiple, geographically-distributed supercomputers coupled together; and virtual environments coupled with other virtual environments [FGNS96]. This experiment was conducted on a wide range of equipment, including high-end display and virtual reality devices, mass storage systems, specialized scientific instruments, and at least seven different types of supercomputers at 17 different sites in North America. The network connecting all of these components was also heterogeneous, using different switching and networking technologies [FGNS96].

A unique aspect of the I-WAY experiment is the use of I-WAY Point of Presence (I-POP) machines at each participating site. An I-POP machine is basically a workstation front-end to a computational resource. By using the same machine running the same operating system, I-POP machines provided a “uniform environment” for management and security of I-WAY resources [FGNS96]. Each I-POP machine also ran a suite of software tools called I-Soft that supported scheduling, security, parallel programming, and a distributed file system. It should be noted, however, that this support was at a relatively low level compared to that currently envisioned by the Globus project; in many cases, I-Soft was the first attempt at providing such functionality.

Discussion. The authors point to several successes of the I-WAY, including the novel idea of point-of-presence machines and preliminary steps toward an integrated software environment for metacomputing. However, the I-WAY also illustrated the acute nature of certain metacomputing problems. For example, the use of point-of-presence machines is clearly not a scalable solution. Perhaps for a small number of well-coordinated sites, this approach works. But on a larger scale, imposing specific standards on the machine and operating system required to interface with the metacomputing system is both unreasonable and difficult to enforce. We suggest that the need for I-POP machines will be obviated by advances in software technology for metacomputing.

The I-WAY environment was also constrained in many ways. For instance, primitive scheduling software limited applications to executing only on predefined, disjoint subsets of I-WAY computers. Similarly, with respect to security, each user had to have a separate account on each site to which access was required. As we have suggested, these limitations are largely due to the experimental nature of this environment. The authors explicitly recognize this and state that none of the user-oriented requirements were "fully addressed in the I-WAY software environment..." [FGNS96]. More importantly, they recognize the important role of software engineering, claiming that "system components that are typically developed in isolation must be more tightly integrated if performance, reliability, and usability goals are to be achieved" [FGNS96].

Real-Time System Information

As motivated by Foster and Kesselman [FK96], the effective use of a metacomputer is largely dependent on being able to know the state of the system resources. This information can include network activity, available network interfaces, processor characteristics, and authentication mechanisms. The information plays an important role in nearly every stage of metacomputing. For example, it indicates which machines are currently operable; it assists in determining which machines may be used to execute an application; it helps in creating a performance efficient schedule for executing the application; and it can be used to determine if rescheduling the application or relocating a task is necessary.

Providing current information only addresses part of the problem, though. Foster and Kesselman [FK96] note that high-performance applications have traditionally been developed for a specific type of system—or even a specific system—with well-known characteristics. Similarly, scheduling disciplines for distributed systems typically assume exclusive access to processors and networks [FK96]. The shared nature of metacomputing environments effectively invalidates both of these assumptions. It is not necessarily known which types of systems will be used to run a given application (or parts of an application) at a given time, and applications encounter contention for network and processor resources from other programs. Therefore, the load and availability of the resources—and hence, the performance that can be delivered to an application—varies over time [Wols96, WSP97]. In order to improve task assignment and scheduling decisions, a prediction of the future system state is also desirable.

Network Weather Service. Wolski [Wols96] describes a "distributed service that dynamically forecasts the performance various networked resources can deliver to an application." His system, the Network Weather Service (NWS), is being built to operate as a component within a metacomputing environment like Globus. NWS itself follows a modular design with the main components being the sensory, forecasting, and reporting subsystems. Together, these components must sense the performance of the metacomputing resources, forecast the future performance of each resource, and disseminate the forecast to higher-level services [Wols96, WSP97].

Network performance and CPU availability sensors exist as a server process that executes on each machine in the metacomputer. These servers periodically conduct communication experiments with one another to determine latency and throughput values. Similarly, each server regularly determines how much CPU time is available to applications. An internal database records these readings locally on each machine.

NWS supports a framework within which any number of forecasting methods can be used. Wolski describes eleven different predictive methods in his initial work, ranging from a simple running average to a more sophisticated autoregressive model [Wols96]. Each predictor uses a history of previously sampled data to forecast future values. However, experiments have shown that different predictors work better at different times [BWFS96]. To assist in choosing the best predictor, NWS records the error between the forecasts for each predictor and the sampled data. NWS actually tracks both the mean square prediction error and the mean percentage prediction error and for each, reports the predictor with the lowest error. Wolski notes that it is unclear which error indicator ultimately yields the best predictions and that in general, "the fitness of each forecasting technique may be application-specific" [Wols96]. Rather than dictating a single solution, NWS leaves it up to the higher-level service to decide which values to use.

Discussion. Wolski's Network Weather Service [Wols96] has a couple shortcomings. First, the right for a NWS server to conduct an experiment is controlled by a single token that is passed from server to server at a rate determined by an external administrative client. Controlling this rate controls the periodicity of communication measurements. While this may be advantageous as a means of minimizing the impact of NWS upon the computational environment, it does not scale to large-scale metacomputing environments. Second, an interesting question about NWS concerns the use of more computationally expensive prediction methods. One might expect a trade-off between the resource consumption of a predictor and the accuracy of the forecast. But there is no guarantee that more computation will actually yield a better prediction. NWS supports a means of dynamically choosing the best predictive method, but in order to do this, each method must be computed. So, one may pay the price of a more expensive method, but get no return on that investment. Practically, there is little NWS can do to solve this problem. At best, careful consideration of which predictors are actually computed is necessary.

In their overview of load distributing, Shivaratri, Krueger, and Singhal [SKS92] identify the main components of load-distributing algorithms. Among these is the *information policy*, which decides when, from where, and what state information is to be collected. While NWS is not a load-distributing system per se, it may ultimately be used as the information policy component of such a system in a metacomputing environment. Shivaratri *et al.* conclude that periodic policies, like NWS, often result in fruitless work when system load is high because the benefits from load distributing in general are minimal when all processors in the system are fully loaded [SKS92]. Consuming resources to collect information under these conditions can actually worsen the situation. Even though NWS is initially targeting the scheduling problem, a similar argument applies.

To counter such problems, the NetSolve system by Casanova and Dongarra [CD96] prefers to "take the risk of having a wrong estimate than to pay the cost for getting a constantly accurate one." This state-change-driven information policy [SKS92] only broadcasts information when it has "significantly changed" [CD96]. This approach avoids repeatedly reporting values that don't change, which in turn reduces the computation and communication load imposed by the monitoring system.

Clearly, both approaches have desirable and undesirable characteristics. The work by Zhou, Wang, Zheng, and Delisle [ZWZD92] seeks to leverage the strengths of both approaches while simultaneously avoiding their weaknesses. Their load sharing system, Utopia, targets large-scale distributed systems consisting of hundreds or thousands of hosts. Zhou *et al.* [ZWZD92] make several observations about large-scale distributed systems. First of all, they note that such systems are typically structured as clusters of machines, each used by a particular group of people. Furthermore, the machines within such clusters usually share resources extensively. Finally, they claim the following [ZWZD92]:

> It is highly unlikely that all the hosts in the system are needed for receiving tasks from far away (in terms of network distance and delay). To achieve optimal performance on a host, typically only other hosts in its local cluster and a select number of remote, powerful hosts, which we call *widely-sharable hosts* (be they ones with fast CPU, large memory, high I/O bandwidth, or special hardware/software), are needed.

Thus, within a cluster, Utopia employs a centralized, periodic information policy where a server on a single resource acts as a master, receiving system information from all of the other servers. This information is then used to place tasks within that cluster. However, between clusters, a different information policy is used. In this case, system information collected within each "target cluster" (*i.e.,* a cluster possibly receiving tasks from another cluster) is sent to the “source cluster” (*i.e.,* the cluster looking to place a collection of tasks) so that a placement decision can be made. “Virtual clusters” of the “widely-sharable hosts” mentioned above “make it possible to share load on widely dispersed hosts in a large scale system, without ‘information pollution’ and undue overhead” [ZWZD92]. This distributed, selective policy is what gives the system desirable scalability properties.

Providing comprehensive, reliable information within clusters and simultaneously avoiding flooding the network with mostly useless information appears to strike an appropriate balance for large-scale systems. It should be noted, however, that while Wolski’s work could benefit in terms of scalability from these results, Utopia does not provide any predictive support like Wolski’s system.

In general, these types of services for metacomputing environment illustrate the tension between providing specific, useful functionality and remaining sufficiently general and flexible enough to address the dynamic nature of the environment. Even though Wolski provides a general framework capable of supporting almost any forecasting method, he still makes certain assumptions about how the system operates that may limit the system’s applicability and performance. Should the NWS information policy, for example, also be a framework that is controllable by a third party? Or is it sufficient just to improve the scalability of the information policy?

The former would certainly result in a more general tool, but would the responsibilities (*i.e.,* the interfaces) for using the system subsequently become too complex? And while the latter appears to address the problem of scalability, might there be other scenarios that demand still a different information policy? There are no obvious answers to these question.[2] While we generally advocate the use of frameworks to address many of the requirements of domain-specific metacomputing for computational science, the extent to which such techniques should be applied remains an open question. This relationship will be explored in more detail in Chapter .

Application Scheduling

It should be clear by now that an important client for metacomputing system state information is a scheduling tool. As Berman *et al.* [BWFS96] state, “despite the performance potential that distributed systems offer for resource-intensive parallel

2. More recently, Wolski [WSP97] has proposed an idea similar to the hierarchical information policy used by Zhou *et al.* [ZWZD92] in the Utopia system.

applications, actually achieving the user's performance goals can be difficult." Indeed, scheduling is central to achieving high application performance on a metacomputer.

However, performance is often user- and application-specific. Furthermore, a system's notion of performance is different than that of the application. For example, whereas a system scheduler might seek to maximize processor utilization or minimize load imbalance, an application-level scheduler might try to optimize some other measure of performance like execution time, speedup, or cost. As Berman and Wolski [BW96] note, "distinct users will attempt to optimize their usage of [the] same metacomputing resources for different performance criteria at the same time."

Application-Level Scheduling. Despite these issues, application programmers have been creating performance-efficient schedules for their heterogeneous parallel and distributed applications for many years. These efforts have largely been "individual and unrelated" despite commonalities in the techniques used [BW96]. To assist and enhance this process in a metacomputing environment, Berman *et al.* propose application-level schedulers (AppLeS) [BW96, BWFS96]. Application-level schedulers represent the "generalizable structure" of the otherwise intuitive and experiential practice of custom application scheduling on heterogeneous distributed systems [BW96]. In other words, Berman *et al.* seek to provide a framework within which metacomputing scheduling can be supported.

The primary difference between application-level schedulers and other scheduling environments is that "everything about the system is experienced from the point of view of the application" [BWFS96]. For example, given a fixed metacomputing system, if the candidate resources for an application are lightly loaded, then the system as a whole appears lightly loaded. This can result in two applications (with different resource requirements) making completely different judgments about the state of the system on which they are going to run. Such a situation does not occur in system-level schedulers.

An AppLeS scheduler consists of several components that correspond to the scheduling process. The Resource Selector considers different combinations of machines for executing the application; the Planner develops possible schedules for each combination of machines; the Performance Estimator evaluates the predicted performance of each schedule in terms of the user's performance metric; and the Actuator executes the best schedule using the target resource management system (*e.g.,* Globus or Legion). All of these components are managed by the Coordinator and have access to an Information Pool that consists of performance models, user preferences and constraints, an application description, and the Network Weather Service (described earlier). In the spirit of Legion, AppLeS' components are meant to be replaceable. For example, custom performance models, planning algorithms, or resource selectors could be used in place of default implementations.

AppLeS essentially generates a "customized scheduler for each application" [BWFS96]. Moreover, since it is at the application-level, the scheduler is essentially an "integrated extension of the program being scheduled" [BWFS96]. That is, the customized scheduler and the application become part of the same execution instance. Berman *et al.* note that these techniques differ from much of the work described in the scheduling literature. In addition, the AppLeS system distinguishes between application-specific, system-specific, and dynamic information used during the scheduling process. It is in this way that Berman *et al.* have parameterized the general scheduling problem [BWFS96].

Discussion. Since AppLeS operates at the application level, it is able to consider and provide information more meaningful to the application programmer. In fact, Berman *et al.* have taken this to something of an extreme, requesting a wealth of information from the user. Some of this information, such as cost constraints and performance objectives, is critical to making appropriate scheduling decisions. But much of the information is excessively detailed, requiring the programmer to indicate, for example, how many megabytes of data is required at input and generated at output. The user must also indicate the number and size of data structures, amount of computation and communication per data structure, the communication patterns exhibited by the application, and information about data conversions [BWFS96]. If a user is willing to collect all of that information, it is arguable that with just a little more effort they could manually create a significantly better custom schedule. Mechanisms for determining such information automatically would be far more desirable.

Even though preliminary results are promising, to evaluate AppLeS based solely on the performance of the schedules produced by the prototype system is to miss the larger contribution the project is making to the area of metacomputing. The primary contribution that AppLeS makes is to provide a framework for scheduling on heterogeneous distributed systems. This framework will eventually support any number of scheduling algorithms, including ones provided by the user. It supports access to different types of information (*e.g.,* application, system, and dynamic) relevant to the scheduling process. And it works with metacomputing resource managers like Globus [FK96] and Legion [GW96]. In the same way that a comparison between the Network Weather Service and a particular predictive method would be meaningless, a rigorous comparison between AppLeS and specific scheduling algorithms is also irrelevant.

But this is not to say that performance is irrelevant. What we are suggesting, though, is that the importance of performance is not necessarily absolute. In light of the increasing complexity of both applications and the systems that run them, achieving high performance is often at odds with other important criteria like portability, extensibility, ease of use, and generality.

Consider, for example, the NetSolve system by Casanova and Dongarra [CD96]. The system places great emphasis on ease of use and flexibility as it supports interfaces to programming languages like C and Fortran, analysis packages like MATLAB, interactive command shell, and the World Wide Web. The authors also claim that NetSolve can use any scientific linear algebra package available on the platforms on which it is installed. Of course, performance is also important, but even the authors admit to using a "simple theoretical model" that is "less accurate and always optimistic..." [CD96].

Contrast with this the attempts by Mechoso, Farrara, and Spahr [MFS94] to achieve superlinear speedup running a climate model in a heterogeneous, distributed computing environment. Performance is the exclusive goal in this work, and the authors go to great lengths to achieve it, carefully optimizing the task decomposition, overlapping computation and communication as much as possible, and manually creating custom schedules for their codes.

On the other hand, problem-specific environments are often capable of achieving high performance simultaneously with usability, flexibility, and other more subjective criteria. But, by definition, these environments lack the generality that an environment such as AppLeS attempts to support. For example, Hui *et al.* [HCYH94] provide a vertically integrated environment for scheduling solutions to partial differential equations on networks of workstations. Their environment is interactive, portable, and scalable, yet it also provides highly-tuned, novel methods for achieving maximal performance. They are able to achieve better performance, in part, by using knowledge about the domain. At the cost of being problem-specific (*i.e.,* the environment can only be used to solve time-dependent PDEs [HCYH94]), they have been able to achieve a desirable combination of the other criteria.

This trade-off between performance and other nonfunctional requirements (*e.g.,* extensibility, generality, portability) is a theme we will return to later. AppLeS is a good example of a system that tries to balance these varied concerns. It does so through a frameworks-based approach that is less concerned with providing specific functionality that meets certain requirements, and more concerned with providing an infrastructure within which a range of functionality can be instantiated.

Global Name Space

A metacomputing environment made up of hundreds or thousands of hosts requires an efficient, general, and usable means for addressing the hosts and other entities (*e.g.,* distributed objects, remote processes, files, users, etc.) in the system. This capability, generally known as *naming*, is a fundamental requirement of distributed systems ([CDK94], p. 254). Coulouris *et al.* point out that since users, programmers, and system administrators all regularly refer to the components and services of a distributed system, names must be "readable by and meaningful to humans" ([CDK94], p. 255). For example, IP addresses (*e.g.,* 128.223.8.39) provide a low-level name space for computer hosts (or, more precisely, for network interfaces), but are not general enough to handle all of the other objects mentioned above. Furthermore, IP addresses are awkward and difficult for most humans to remember. The use of logical IP names (*e.g.,* foo.bar.edu) that are mapped into IP addresses is an improvement in this regard, and they also provide a degree of transparency. That is, a given IP name can, over time, refer to several different machines. But, name lookup operations can have a significant impact on system performance [CM89].

A system which translates a *name* into the attributes of a resource or object is called a *name service* ([CDK94], p. 253). One of the most widely used name services today is the Internet Domain Name System (DNS). DNS is a distributed service that responds primarily to queries for translating IP names into IP addresses. The DNS database is distributed across a collection of servers, and data is widely replicated and cached to address problems of scale ([CDK94], p. 271). DNS supports a *name space* that is tree-structured and partitioned both organizationally (*e.g.,* .com, .edu, and .gov suffixes) and geographically (*e.g.,* .us, .uk, and .fr suffixes).

While DNS provides a good example of a robust name service, it does not fully address the needs of a metacomputing environment. In particular, DNS only assists in locating physical hosts; metacomputing environments contain many other entities that require names. Thus, support of a global name space for metacomputing environments is an important consideration, yet surprisingly little effort has gone into this area. Many projects are adopting the now-common naming system used on the World Wide Web known as Uniform Resource Locators (URLs). URLs define a name space that is more generic than DNS (although, it uses DNS in many cases). This section proceeds by describing in more detail the name space defined by URLs. This is followed by a broader discussion of global name spaces and URL-based naming schemes as used in metacomputing projects.

Uniform Resource Locators (URLs). Since 1990, Uniform Resource Locators (URLs) have been the primary means of addressing—or, naming—information on the World Wide Web [BMM94]. A URL is a textual name composed of two main parts: the *scheme* and the *scheme-specific part*. These two parts are delineated by a colon. The scheme typically specifies a network protocol, the most common of which is Hypertext Transfer Protocol (HTTP). The scheme-specific part differs for each scheme. In the case of HTTP, it typically indicates the IP name of a web server and the directory path to a specific web document on that server. Many other schemes are supported. For example, the “file” scheme typically takes a directory path to a local file; the “mailto” scheme looks for an email address in the scheme-specific part; and the “news” scheme takes the name of an USENET newsgroup [BMM94]. New schemes are easily added to this system. For example, schemes for television (”tv”) and telephone (”phone”) are under consideration [Zigm96].

The URL system is layered upon other name services and network protocols. For many schemes (*e.g.,* http, news, ftp, telnet), DNS must be consulted to translate an IP name into an address. Once the target host is known, the transaction is carried out according to the designated protocol. For instance, the URL <ftp://foo.bar.edu/pub/readme.txt> first results in a DNS lookup of “foo.bar.edu.” Once the network address is known, a connection to that machine using standard File Transfer Protocol (FTP) ensues. This is followed by a FTP request for the file named “readme.txt” located in directory “pub.”

This functionality is largely consistent with the role of a name service. As Coulouris *et al.* ([CDK94], p. 256) point out, the ability to *unify* the resources of different servers and services under a single naming scheme is a major motivation for keeping the naming process separate from other services. URLs provide this capability and further introduce the notion of a “universal set of names” [Bern94]. Berners-Lee defines a member of this set as a Universal Resource Identifier (URI); a URL is then more specifically defined as a URI “which expresses an address which maps onto an access algorithm using network protocols” [Bern94]. More recently, attempts to define Uniform Resource Names (URNs) would result in a name space that is complementary to URLs. URNs would provide a name space (and resolution protocols) for persistent object names [SM94]. Whereas URLs point to specific documents on a specific web server, a URN would have to be “resolved” (much like an IP name) to determine where the desired document resides. That is, URNs would provide a layer of abstraction over relatively low-level URLs.

Discussion. Given the wide acceptance of URLs in the context of the World Wide Web and the emphasis that metacomputing places on internetworking, it is not surprising that several metacomputing projects are attempting to leverage the ubiquity of URLs. The focus of this discussion will be to compare URL-based naming schemes for metacomputing environments with other possibilities. In particular, URLs will be compared to the global name service proposed by Cheriton and Mann [CM89].

Notable among the metacomputing systems currently using URLs in some form are Globus [FK96] and Atlas [BBB96]. The Atlas metacomputing project [BBB96] uses URLs to support a global file system. Atlas integrates a runtime library for scheduling and communication with Java, a programming language that has extended the limited fetch-and-display capabilities of web browsers to include the ability to perform computations. While few details are available, Atlas supports a global file system by providing special versions of standard Unix file I/O routines like fopen() that accept URLs as file names. The URLs are then translated into either local or remote file system accesses. The preliminary work only supports a read-only file system, but a coherent read/write version is planned for the future [BBB96]. The use of Java and URLs suggests a possible future integration with a web-based environment.

As we have described earlier, the Globus project has identified several component modules. Currently, the communication module exists as a system called Nexus [FKT96, FGKT97]. Nexus is a multithreaded communications library that supports an abstract communication model based on nodes, contexts, threads, communication links, and remote service requests [FKT96]. Contexts and nodes roughly correspond to processes and processors, respectively, and a context consists of potentially many threads. In Nexus, communications (*i.e.,* remote service requests) occur between contexts. Higher-level services can manipulate these abstractions through the Globus communications interface.

For two contexts (*i.e.,* multithreaded processes) to communicate, one of them must “attach” to the other [FGKT97, FT96]. The name space of Nexus contexts is accessed through Uniform Resource Locators (URLs). Thus, the Nexus URL <x-nexus://foo.bar.edu:1234/> indicates that there is a Nexus server running on host “foo.bar.edu” and listening on port 1234. Nexus URLs define a global name space for Nexus contexts. While the translation of these names is performed by Nexus itself, by standardizing on URLs to define this name space, Nexus functionality can more easily be integrated into web-based environments. In fact, in an effort to support “ubiquitous supercomputing” similar to the “universal access” supported by the World Wide Web, Foster and Tuecke [FT96] are integrating Nexus with Java. The Atlas project mentioned earlier has a similar goal. Nexus’ robust support for multiple communication methods greatly extends Java’s limited capabilities in this area and creates the possibility of using Java for high-performance, heterogeneous computations [FT96].

In addition, Globus uses a separate URL scheme in its remote file and data access module [FK96]. URLs for remote file access are of the form <x-rio://foo.bar.edu:5678/home/user/file.ext> where “x-rio” is the name of the scheme, “foo.bar.edu:5678” indicates the host and port number for a remote file server, and the rest of the URL is the directory path to a file (on the remote host). So, Globus uses two separate URL schemes to support one global name space for contexts and another global name space for files.

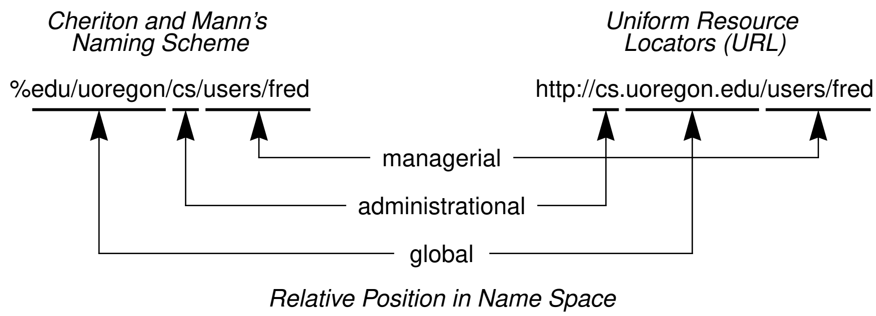
Thus, Nexus and Atlas demonstrate that a primary reason for using URLs is the prospect of increasing ubiquity by integrating metacomputing into the World Wide Web environment. But if any metacomputing project seeks ubiquity, it is the Legion project [GW96]. Yet, Legion is not embracing the URL mechanism to support its name space(s). As we have seen, Globus uses separate URL schemes for inter-context communication and for remote (parallel) file system access via the respective Globus modules; Atlas uses URLs to support a global file system and a completely separate (runtime) mechanism for interprocess communication. But Legion, with its pervasive object-orientation, is providing a significantly more unified approach where everything—processes and files alike—are objects. Grimshaw and Wulf [GW96] note that files are simply objects that reside on a disk. To this end, the Legion project is building its own global file system that will support a high-performance, object-based interface to files.

We stated previously that a key goal of naming is unification ([CM89], p. 256). Through its object-oriented approach, Legion essentially supports an even higher degree of unification than URLs can. That is, whereas Nexus URLs clearly distinguish the type of object being accessed (*i.e.,* by the scheme and syntax of the URL), Legion names may not. One interacts with a Legion object in the same way (*i.e.,* by invoking methods) whether it is bound to a file, a process, another user, or something else in the system. While the available methods may differ across object types, the syntax for naming them does not.

Legion’s ability to provide a higher degree of name unification over URLs is a product of its object-orientation. Object-orientation facilitates the creation of a global name space through its properties of abstraction, inheritance, and encapsulation. URLs, as mentioned above, do not provide these capabilities, and efforts to provide more abstract name services (*e.g.,* URNs) are underway [SM94]. The URL system has other shortcomings which can be revealed through a comparison with the naming system proposed by Cheriton and Mann [CM89].

Cheriton and Mann [CM89] propose a decentralized, global naming service that attempts to provide both performance and fault tolerance. The naming architecture consists of three levels: global, administrational, and managerial. As we will see, these levels have a desirable correspondence to the organization of metacomputing environments that span multiple administrative domains. The global level represents the organizations and groups of organizations that are covered by the naming service. Next, entries at the administrational level are owned and managed by a particular organization. Finally, managerial entries correspond to actual objects (*e.g.,* directories, files, processes, people, etc.) and are called "object managers" [CM89]. Names in this system follow a simple, consistent syntax. Figure 8 contains an example of this syntax, labels the levels in the naming hierarchy, and shows the (approximate) corresponding URL.

FIGURE 8. A comparison between the name spaces supported by Cheriton and Mann's naming service and commonly used Uniform Resource Locators (URLs).



Immediately, an advantage of this scheme (and a shortcoming of URLs) is revealed. In particular, it provides a higher-level abstraction to naming. Whereas URLs contain the names of explicit host machines, this naming scheme abstracts away from particular machines and creates a logical, top-down approach to naming. A related advantage is that the naming syntax is consistent at all levels of the hierarchy. URLs (under the http and several other schemes) are designated in two different formats (*i.e.,* the hostname element is represented in dot notation while directory paths are in path notation). Finally, URLs also include a separate designation for "protocol" (*e.g.,* http, ftp, etc.) Because Cheriton and Mann's global naming system is object-oriented, protocol designation is not necessary (making it similar to Legion in that respect); object managers support a common, method-based interface for requesting the information they manage [CM89].[3]

3. A full comparison between Cheriton and Mann's scheme and URLs is somewhat irrelevant since URLs define only a name *space* and Cheriton and Mann describe a complete name *service*. The comparisons we have made rely primarily on the nature of the name space defined by Cheriton and Mann.

In summary, the potentially dynamic nature of a large metacomputing system requires a consistent and robust global naming service. Comparing the URL system with Cheriton and Mann's system reveals some of the potential problems that metacomputing implementers may encounter. Efforts to augment the URL naming system with more persistent, abstract names (*e.g.,* URNs) may someday provide metacomputing with a reasonable compromise between achieving ubiquity and the quality, transparency, and consistency of more robust naming systems.

Conclusion

The convergence of parallel and distributed computing has created many new, challenging research areas. Many of these challenges exist in the area of heterogeneous computing. To understand those challenges is to understand many of the challenges of metacomputing. Perhaps more revealing, though, is a survey of past and current efforts. The NCSA metacomputing effort was clearly instrumental in defining the vision of metacomputing and actually building an early system. That system, though, lacked the extensive software support that is now widely recognized as being required. Current efforts at creating that support are varied. While the PVM and HeNCE effort is building on proven, existing technology, the Legion project is creating a completely new system from the ground up. In between these two, Globus is using a modular approach that incorporates existing technology where applicable, and creates new functionality when necessary. Regardless of the methodology, a survey of these systems reveals a number of challenges, many of which are being addressed by the metacomputing community. Building metacomputing testbeds, collecting real-time system information, scheduling applications, and supporting a global name space are just a few of the challenges metacomputing faces. These specific challenges often have firm roots in parallel and distributed computing. By comparing the efforts of the metacomputing community with other projects and related research, we gain insight into the nature of metacomputing and the extent of the challenges it faces.

CHAPTER 4

Supportive Research In Software Engineering

CERTAIN RESEARCH topics in software engineering have particular relevance to domain-specific metacomputing. The goal of this chapter is not a comprehensive review of software engineering—a major undertaking by itself—but rather, to identify topics of research that may contribute to the area of domain-specific metacomputing for computational science. Earlier chapters provided some motivations as to how and why software engineering may contribute; we begin by reviewing those. This is followed by a brief look at how software engineering is currently applied in the parallel and distributed computing community. The last two sections focus on the software design and development problem from two perspective: user and developer. For the user (*i.e.,* the application scientist), we explore how object-orientation facilitates application development. For the developer (*i.e.,* the builder of a metacomputing system), we discuss how software architectures—and domain-specific software architectures, in particular—provide guidance for constructing more useful systems for application scientists.

The Software Crisis in Parallel Computing

The "software crisis" of the 1960s led to the first use of the term *software engineering*. Since then, and despite numerous advances in approaches, tools, and education, that crisis is still with us today ([Somm89], p. 3). Today's software crisis challenges the software engineer to produce high quality software in a cost-effective manner. Sommerville proposes four criteria for high quality software ([Somm89], p. 4):

Maintainability: Software changes should not result in undue costs.

Reliability: Software that fails or produces faulty results is not useful.

Efficiency: Software should not waste system resources.

Usability: Software user interfaces should accommodate their intended users.

The extent to which each criterion is individually addressed has a direct impact on the cost of software. But these criteria are interrelated, which makes minimizing overall cost and maximizing individual criteria difficult. For example, incorporating a better user interface may reduce efficiency. Subsequently attempting to improve efficiency, though, may make the software more difficult to change, thus decreasing maintainability. In general, maintainability, reliability, efficiency, usability, and other similar nonfunctional software requirements represent trade-offs for software engineers. It is achieving a balance among such criteria that makes producing high-quality software so challenging.

Software engineering is composed of many subdisciplines. Ramamoorthy and Tsai [RT96] identify seven of them: development process, requirements engineering, design, programming languages, testing, reliability and safety, and maintenance. Of these subdisciplines, four are of particular relevance to this chapter:

Development Process. As applications grow larger and their domains become more complex, the software development process must keep pace. To this end, object-oriented techniques enable modularity, abstraction, reuse, and programming-in-the-large. Software development traditionally ignores the end users' views. More effective software systems combine a developer's technical knowledge with end users' domain knowledge.

Requirements Engineering. End user needs and knowledge are recorded so they may be used during system development. Software prototypes and simulations help users and developers to understand development problems and possible solutions.

Design. Object orientation plays a key role in the area of design. Object-oriented systems are loosely coupled but highly cohesive, with design activities carried out at two levels: the high level focuses on the system architecture and decomposition, while the low level focuses on algorithm, data, and code design.

Programming Languages. The evolution from low-level machine code and assembly languages to high-level languages like Fortran, C, and object-oriented languages is characterized by increases in level of abstraction and modularity. These characteristics facilitate development, improve maintainability, and reduce costs.

Chapter 2 suggests several ways in which software engineering could help realize the goals of domain-specific metacomputing. In building “big” metacomputing software environments, like those described in Chapter 3, it is often desirable to reuse existing software (*e.g.,* communication packages, numerical libraries) and simultaneously support a high degree of flexibility and extensibility so that unpredicted applications of metacomputing technology can be supported in the future. Conversely, the potentially diverse resources and capabilities of metacomputing environments may greatly complicate the creation of the applications which ultimately use them. Thus, from the area of software engineering, domain-specific metacomputing may potentially benefit from research in software design and development, software reuse, and interoperability.

However, in the parallel and distributed computing community, techniques from software engineering have not been widely embraced. Application and tool development has been carried out in a largely ad hoc manner, resulting in what some consider to be yet another crisis [Panc91, Panc94]. Chandy identifies four unique aspects of parallel software that have contributed to this state of affairs [Chan94].

Rate of Change. The relatively rapid emergence of parallel computing demands that a large number of parallel applications be developed in a relatively short amount of time. (Software costs)

Correctness. Nondeterminacy and multiple threads of control make formal reasoning and debugging difficult. (Reliability)

Performance. Architectural diversity and the range of factors that can affect performance (*e.g.,* communication latency, data distribution) complicates optimization and efficiency. (Efficiency)

Absence of Standards. Inconsistent and inadequate tools hinder productivity, and the lack of hardware standards and only a few software standards limits the development of CASE tools. (Maintainability)

Parallel computing's rapid rate of change results in high software costs and makes software development in this area a risky endeavor. Chandy's other observations are largely consistent with Sommerville's criteria for well-engineered software ([Somm89], p. 4). In particular, correctness has direct bearing on the reliability of software; performance and efficiency are inextricably linked; and an absence of standards certainly complicates maintainability. Usability, the fourth of Sommerville's criteria, is, unfortunately, often an afterthought in light of the other daunting challenges faced by parallel software developers [Panc91]. Each of these challenges ultimately affects the productivity of the parallel tool or application developer. Yet, to date, just two relatively simple forms of "software engineering" have been applied.

Primitive Forms of Software Engineering

The most common forms of "software engineering" employed within the parallel and distributed computing community are *code scavenging* and *software libraries*. Both of these techniques facilitate the reuse of programs and code, though at a relatively low level [TTC95].

Code scavenging occurs when pieces of source code are copied from one application for use in another. While common, code scavenging is less a software engineering "technique" and more a natural by-product of informal code development. A software library, however, is a compiled collection of specific procedures and data structures that supports some particular functionality and is accessible through a procedural interface.

Software libraries can support a wide range of functionality. One of the most prevalent types, mathematical software libraries, have been around since the 1960s [Rice96]. Examples include LINPACK, LAPACK and ScaLAPACK [DW95] for linear algebra; ODEPACK [Hind83ode] for differential equations; and FFTPACK [Swar82] for performing fast Fourier transforms. Software libraries for specific scientific areas also exist. For example, CFDLIB solves a wide range of computational fluid dynamics problems [John96]. These types of libraries most commonly support the application developer by providing pre-packaged, generic, and widely applicable functionality. However, the tool developer also benefits from libraries that support tasks like constructing and managing user interfaces (*e.g.,* XForms [ZO97]), sharing and communicating data with other processes (*e.g.,* Nexus [FKT96]), and interacting with databases (*e.g.,* POSTGRES [YC95]).

But, as Chandy [Chan94] describes, software libraries have several limitations with respect to parallel computing. First, libraries are language- and architecture-specific. It is not possible to develop libraries for every combination of parallel programming language and parallel architecture. Second, software libraries for parallel computing carry a risk of commitment because so few standards exist. The chance that a given parallel language and architecture may not be supported in the future is much greater than in sequential computing. Finally, composition and data mapping complicate the creation of parallel software libraries. In sequential programming, a single data space and a single type of functional composition simplifies the issues of composition and data mapping. But in parallel programming, library procedures may have certain expectations for data layout (*e.g.,* with respect to the distribution among processors) but also be expected to cooperate in parallel execution.

In addition, other researchers [TBSS93] expose an inherent limitation on the scalability of software libraries, pointing to a phenomenon called "feature combinatorics" as the primary cause. As software libraries achieve broader use, a broader range of features must be supported. "The implementor of the component library must laboriously enumerate every permutation of feature selections" or possibly fail to meet the needs of a particular application [TBSS93].

Finally, Rice [Rice96] comments on the generally usability of software libraries:

> Although the software library provides some form of abstraction and a facility for reusing software parts, it still requires a level of expertise beyond the background and skills of the average scientist and engineer....

Thus, for parallel and distributed computing (and ultimately, metacomputing) software libraries pose difficulties for both the developers and potential users. This suggests that software libraries, as a mechanism for software reuse, may not be adequate. In the remainder of this chapter, additional software engineering techniques that may benefit software design and development of, and within, metacomputing environments are presented.

We do not intend to single out a "best" solution, but rather we seek to identify the potential advantages and disadvantages of each approach, recognizing that for a given situation, the best approach depends upon a number of criteria, including the application, the programmer, project goals, and functional and nonfunctional requirements.

Object Orientation

Proponents of an object-oriented approach to software development claim many advantages, including data abstraction, reuse, extensibility, and flexibility ([CABD94], p. 7). Data abstraction occurs since the implementation of an object is hidden behind the interface (methods) through which other objects access functionality. Reusability is facilitated because objects encapsulate both methods and data structures into a single entity that can more easily be applied in different contexts. Extensibility is supported through object inheritance and sub-classing.

While the potential benefits are numerous, object technology has not been as widely adopted as expected [Panc95]. Certainly, part of the reason for this has been the relatively steep learning curve incurred by programmers switching from other programming paradigms [Panc95, FT95]. Other factors include the requirements of new tools, languages, metrics, and software development processes [FT95]. Norton *et al.* [NSD95] summarize the apprehension among scientific programmers:

> Although valuable progress continues, until these methods become commonplace, as demonstrated by supercomputer manufacturer support and standards committees, most developers may remain apprehensive about adopting new languages. Thus, the future of scientific programming will depend on *establishing standards* and *recognizing educational trends* in software design.

From this standpoint, we still consider object oriented computing a "technology" that may potentially benefit metacomputing for computational science. But, we must note that compared to the other technologies examined in this chapter, object-oriented computing is more advanced, more thoroughly researched, and more widely applied.

Attempts to apply object-oriented technology within the parallel and distributed computing communities have followed three primary techniques. The most basic approach is to use an object-oriented language (typically C++) combined with a standard message-passing library (such as MPI). In this case, the application developer must manage parallelism explicitly. Among those attempts to support more general, portable, and automatic object-oriented parallel computing (*i.e.,* object-oriented parallel languages), a common theme has emerged: parallelism is supported through specialized class hierarchies and/or language extensions that interface with complex runtime systems supporting task and/or data parallelism. Norton *et al.* [NSD95] identify several examples: ACT++, C**, Charm++, Compositional C++, Concert, Concurrent Aggregates, Concurrent C++, COOL, DC++, DCE++, HPC++, Mentat, Parallel C++, pC++, POOL-T, and POOMA. In this case, the application developer adopts one of these language systems (and the abstractions it supports) to implement their application. Finally, higher-level programming and problem-solving environments built on an object-oriented foundation support a range of features to assist scientists and/or application developers. The following sections briefly describe examples of each approach to integrating object-oriented technology with parallel computing.

Object-Oriented Languages And Message Passing Libraries

The most rudimentary means of parallel object-oriented computing is to use an object-oriented language such as C++ and a message passing library. An example of this technique is the work by Norton *et al.* [NSD95]. They describe their experience porting a Fortran 77 plasma particle in cell (PIC) simulation code to C++ and Fortran 90 (which also supports some object-oriented concepts). They seek a high-performance, cross-platform solution capable of executing on Intel Paragon, IBM SP1/SP2, and Cray T3D distributed memory parallel computers, while simultaneously taking advantage of the improved design, development, and maintenance characteristics of the object-oriented paradigm.

Two requirements are immediately evident. First, the target parallel architecture must have a C++ compiler available. Second, a message passing library such as PVM or MPI that is compatible with the compiler must also exist for the target architecture. Once these basic requirements have been met, it is up to the programmer to carry out the design and implementation of the application. This includes designing and implementing the required class hierarchy (usually from scratch), managing decomposition and concurrency, and dynamically balancing processor loads at runtime (if necessary). Of course, with respect to metacomputing, all of the shortcomings of basic message-passing libraries (*e.g.,* PVM) as discussed in Chapter 3 also apply. In addition, though, this low-level approach to parallel object-oriented computing has other limitations.

For example, a main reason for adopting the object-oriented paradigm is reuse. But as Pancake [Panc95] notes, it is rarely known what functionality may actually be amenable to reuse:

> Typically, it is only after an [object-oriented] application is complete that the developers understand which objects might have broader use. Those objects then must be restructured through a process known as *generalization*, that in turn may require revisiting several earlier stages of object design....

Norton *et al.* [NSD95] confirm this through the refinement of their particle simulation class hierarchy:

> Unfortunately, hierarchies are nearly impossible to design correctly on the first attempt. Moreover, when the design is poorly organized, it is difficult to modify it without triggering something close to a complete redesign.

While they claim to have reused much of their code during the refinement process, they recognize that "if the new class hierarchy cannot be defined with clean interfaces, the best approach is to redesign it from the beginning" [NSD95].

In addition to limiting reuse, a low-level parallel object-oriented approach also limits its code portability, especially where machine-specific message passing libraries and compilers are involved. Norton *et al.* [NSD95] use different communication libraries and compilers for each architecture.[1] This resulted in numerous problems, including five months of lost development time from compiler inconsistencies alone.

But perhaps the largest barrier preventing wide-spread adoption of C++ (and other object-oriented languages) in the parallel computing community is performance. Norton *et al.* [NSD95] report C++ execution times that are 53-110% *slower* than same-sized problems implemented in Fortran 77, depending on the architecture and message-passing library used. Even for their sequential computations, C++ performs about twice as slow as Fortran 77. This loss of performance is partly due to memory overhead and data access costs. Also, whereas Fortran allows arrays to be passed directly as message-passing parameters, C++ requires the use of intermediate buffers [NSD95]. In general, object-oriented computing has overheads associated with it that decrease performance, or at least make optimization more difficult.

Similarly, efforts to develop efficient computational kernels and components for one compiler/library/machine combination do not necessarily translate into efficient execution on other platforms. For example, Norton *et al.* [NSD95] observe that "designing efficient and portable C++ code is difficult due to differences in

1. The Intel Paragon used the NX message-passing library and the GNU g++ and Intel C++ compilers; the IBM machines used the MPL communications library and the IBM xlC compiler; and the T3D used a modified version of PVM and Cray C++ [NSD95].

compiler implementations." In addition, they designed and built a separate class that supported a virtual parallel machine. This class encapsulated all the machine- and library-specific details needed for parallel execution. While clearly a good example of how the object-oriented paradigm can be used to enhance portability, the reusability and generality of that class is subject to the problems previously described. Such a class could be applied to a broad range of other applications, but to do so might require significant modification to the original implementation.

Norton *et al.* also identify the major design improvements over Fortran 77 (and similar languages). An object-oriented approach allows program classes to correspond directly to the "physical and computational constructs" of the simulation [NSD95]. They also note that in comparison to Fortran 77, object-orientation "provides a programming perspective that reflects the problem domain" [NSD95]. Thus, object-orientation holds particular promise as a means of facilitating domain-specificity. We explore this topic in more detail later.

Object-oriented languages provide a good example of the trade-offs associated with nonfunctional requirements. In terms of Sommerville's criteria for well-engineered software ([Somm89], p. 4), object-oriented languages facilitate maintainability as well as some degree of reusability, but efficiency (or more specifically, performance) suffers. In the context of parallel scientific computing, reliability and performance are typically the most critical nonfunctional requirements. Does this mean object-oriented languages have no place in computational science? Certainly not. What is needed are more compelling reasons for using object-oriented languages. That is, the nonfunctional benefits of object-oriented languages must be brought to bear on the open challenges in domain-specific metacomputing for computational science.

Parallel Object-Oriented Class Hierarchies

An eventual outgrowth of the approach taken by Norton *et al.* [NSD95] is a general framework within which different types of parallel particle simulation codes may be deployed. Indeed, the development of parallel object-oriented class hierarchies can result in improved reuse. Furthermore, hierarchies focused on a particular domain may provide an effective means for achieving domain-specificity. As we have shown, however, the creation of such a framework requires careful consideration in the design and implementation of the constituent classes. Thus, for the developer, all of the challenges of the lower-level approach remain since essentially the same task is being carried out. But the application developer who can adopt such a framework gets all the advantages of the end product without incurring the cost of developing it. The disadvantages to the end-user are (1) a possible lack of familiarity with the hierarchy's implementation which could hinder major changes, and (2) encountering the situation where the framework provides most of the desired functionality, but is missing one or more application-critical features or components, thus preventing adoption of the framework as a whole. A framework designed in conjunction with application scientists could avoid these problems. Essentially, this approach decouples the design and development of a class hierarchy from its actual use.

Parallel Object-Oriented Methods and Applications. The Parallel Object-Oriented Methods and Applications (POOMA) Framework attempts to support a flexible environment for scientific programs that can exploit data-parallel semantics [RHCA97]. In fact, POOMA was originally conceived to support the same domain (*i.e.,* particle in cell simulations) as the work by Norton *et al.* [NSD95]. Not surprisingly, there are major similarities in the object-oriented abstractions (appropriate to PIC simulations) each system supports. But whereas Norton *et al.* construct a class hierarchy as a side-effect of porting an application from Fortran 77, POOMA seeks first to construct a comprehensive class hierarchy that can subsequently be used by a range of applications in the given domain.

POOMA is a comprehensive C++ class hierarchy for the data-parallel development of classes of scientific applications. POOMA actually targets several related domains, including plasma physics, molecular dynamics, computational fluid dynamics, rheological flow, vortex simulations, porous media, medical imaging, and material science [ABCH95, RHCA97]. This is accomplished through a five-layer class hierarchy and a variety of data types appropriate to the application areas. In addition, POOMA provides a portable, high-performance, serial/parallel programming model.

The POOMA system provides an example of a framework, which is discussed in more detail in Chapter . Reynders *et al.* [RHCA97] explain the positive implications of this approach:

> Computer scientists and algorithm specialists can focus on the lower realms of the FrameWork, optimizing computational kernels and message-passing techniques without having to consider the application being constructed. Meanwhile, application scientists can construct numerical models with objects in the upper leaves of the FrameWork, without knowing their implementation details.

This separation is possible because of POOMA's object-orientation. Parallelism and application science are encapsulated within different objects, helping to prevent the interlacing of message-passing commands and computational algorithms within the application code [RHCA97]. Not only does this improve the understandability of application code, it keeps orthogonal aspects of the hierarchy code separate from one another. This, in turn, improves the generality and extensibility of the hierarchy.

This separation manifests itself in the five layers of the class hierarchy. At the highest level, the Application Layer represents "abstractions directly relevant to application domains" [RHCA97]. The Components Layer contains the building blocks from which the Application Layer is constructed, including solvers, FFTs, and particle operations. This layer represents reusable components that can be used to compose applications [ABCH95]. Next, the Global Layer defines the abstract data types (fields, particles, matrices, etc.) that are used by the Component and Application Layers to create domain-specific structures. The Parallel Abstract Layer implements the key abstractions of parallel simulation and programming, such as data layout, communication, and load balancing. Finally, the Local Layer contains node-local instances of Global Layer data structures.

POOMA is a language target for the Accelerated Strategic Computing Initiative (ASCI) described earlier [DOE96]. Thus, its operation in larger-scale, heterogeneous environments is of concern. Reynders *et al.* [RHCA97] indicate that they are investigating how POOMA might support more coarse-grained, task-parallelism. POOMA already provides abstract representations for key metacomputing requirements like load balancing, data distribution, and communication. It remains to be seen whether the POOMA researchers will expand these capabilities to provide a comprehensive metacomputing programming environment, or whether they will make attempts to interface with external objects, tools, and resource managers like Globus and Legion and have POOMA operate as a component within a larger environment.

In summary, object-oriented class hierarchies still require substantial programming efforts to create real applications. The reusable abstractions are available, but the science (in the form of algorithms) still has to be expressed. On the other hand, perhaps this approach strikes an effective balance between allowing a high-degree of software reuse while still retaining full programmability of the application.

Object-Oriented Systems For Parallel Computation

We describe an *object-oriented system for parallel computation* as a software system built with object-oriented technology and which supports or facilitates higher-level access to parallel computing capabilities. The object-oriented nature of the system's implementation may be present in varying degrees in the user's experience with the system, and the level of access to parallel computing capabilities is assumed to be something higher than that required when using a predefined class hierarchy. We describe only one example in this section, but other systems also fit this description. For example, the problem-solving environment PDELab [WHRC94] (Chapter 5) and the program archetypes concept [Chan94] are other examples of object-oriented systems for parallel computation.

Parallel Object-Oriented Environment and Toolkit. Armstrong and Macfarlane [AM94] describe the Parallel Object-Oriented Environment and Toolkit (POET). At first glance, POET has several striking similarities to POOMA [ABCH95, RHCA97]: the researchers describe it as a framework; it is implemented in C++; it provides domain-specific abstractions useful for a class of problems; and parallelism and data movement is encapsulated within objects. Indeed, at first glance, POET appears to be yet another parallel object-oriented class hierarchy.

But Armstrong and Macfarlane are careful to distinguish their work in this regard. For example, unlike most who use the term, they define their use of the term *framework*: “an object-oriented style of programming where a pre-existing environment provides a top-level object or objects within which all others are nested” [AM94]. While we regard the term much more generally (*e.g.,* we do not limit it to an object-oriented style of programming), this definition notably excludes C++ (and hence, POOMA) as a framework because “the user must provide a main program that is itself not an object” [AM94]. Even though it is written in C++, the POET toolkit provides a top-level object from which the user “will instantiate, modify, and ‘frame’ together objects provided by the toolkit to create a numerical application” [AM94].

In addition, whereas class hierarchies act as components within a larger calculation or simulation, POET provides a top level (application structure) and a bottom level (parallelism, data movement, and communication) between which the user code is inserted. “Put succinctly: a class library is designed to be driven by user code while a frame-based system [like POET] is designed to drive the user code” [AM94]. Thus, similar to what might be done when using an user interface construction kit, the programmer/scientist fills in “stubs” in template objects [AM94]. POET provides the main control structure (like the user interface event loop that handles keyboard and mouse events) and also manages low-level details such as communication and load-balancing.

POET “frames” are essentially domain-specific abstractions that capture “the basic communication linkages that are necessary to implement a parallel version of particular scientific problem classes...” [AM94].

A specific framed object represents a complete algorithm. The stubs within an object can be replaced by user-defined, science-specific objects or methods. (In addition to C++, POET supports callbacks to C or Fortran routines.) Then, POET uses such objects as it solves the problem. Parallelism and other artifacts of the computational environment are encapsulated within the lower levels of the framework.

POET is fundamentally a template-based approach, except that it supports multiple languages [AM94]. Its object-oriented basis provides the encapsulation of high-level program structure and low-level implementation details, and also allows user code to be inserted in between. Creation of domain-specific POET frames is currently done on an individual basis; the environment does not support this process. (Work in the area of domain-specific software architectures addresses this issue and is discussed later in this chapter.)

What initially appears to be a subtle change in the perspective of control (*i.e.,* who provides the application control structure) is actually a significant paradigm shift that represents support for parallel computation at a higher-level than class hierarchies. In this way, POET is similar to the HeNCE/PVM environment (discussed in Chapter 3), in which scientists expressed algorithms using a graph-based paradigm and only filled in code for an application's core procedures and functions. Creation of the program control structure (*i.e.,* the top-level) and the underlying calls to PVM (*i.e.,* the low-level) were generated automatically. The main problem with HeNCE was the potentially limited set of applications it could generate. With respect to providing multiple frames, POET is an improvement. Within a particular frame, however, POET is actually more restrictive than HeNCE. But that is the very point: by limiting the programmer/scientist to providing only the algorithmic support unique to the science they are pursuing, they can—for a class of problems—be more productive. As Armstrong and Macfarlane state, "the frame-based approach is useful because *it is* restrictive" [AM94]. In other words, POET achieves of a high degree of (domain-)specificity through (frame-based) abstraction.

Carried to an extreme, this approach may ultimately result in something very similar to a problem-solving environment (PSE), such as PDELab [WHRC94]. A problem-solving environment similarly provides the top-level structure and low-level implementation details for solving the problem, and may allow user-defined functionality to be inserted in between. In a more limited case, the user might only be allowed to specify values for parameterized objects or algorithms. PSEs go beyond a system like POET, though, in that they typically try to emulate the entire problem-solving process (*e.g.,* from initial brainstorming to interpretation of results) [WHRC94]. As we have mentioned previously, though, PSEs transcend domain-specificity by focusing exclusively on a single problem. Nonetheless, the goal is still *to facilitate high-level access* to high performance computing for solving specific types of problems. As in the case of PDELab, these systems are often built on an object-oriented foundation which extends, to varying degrees, into the user's interaction with the system.

In summary, object-orientation is a major software engineering technology that can assist in supporting access to parallel and distributed computing capabilities. Furthermore, parallel object-oriented computing can take on a range of forms, from an object-oriented language combined with a message passing library to parallel class hierarchies to object-oriented systems that facilitate high-level access to parallel computing capabilities. Indeed, the Legion metacomputing project [GW96] recognizes the potential benefits of this approach as its researchers are constructing a completely object-oriented metacomputing environment. Object-orientation provides some fundamental mechanisms for achieving software reuse which, in turn, can result in increased productivity (by creating less work for the application scientist [MN96]). Furthermore, properties of the object-oriented paradigm facilitate the creation of domain-specific abstractions such as application data structures with clear relationships to the physical phenomena being modeled.

Software Architecture

In this section, we shift our attention from the application scientist's view of software development to the challenges faced by the developers of the systems used by application scientists. The application of object-orientation technology to parallel computing reveals an important theme in using high performance computing to solve computational science problems:

> *To increase scientists' productivity when trying to use high performance computing, they must be able to interact with the computing environment in a way, and at a level, that is meaningful to them.*

For parallel class hierarchies and object-oriented systems for parallel computing, developers make efforts to tailor the systems to specific scientific or computational domains. We can observe the evolution of this trend in the systems previously described.

As we move from parallel class hierarchies (*e.g.,* POOMA [RHCA97]), to programmatic object-oriented systems for parallel computing (*e.g.,* POET [AM94]), to object-oriented problem-solving environments (*e.g.,* PDELab [WHRC94]), increasing efforts to specialize and refine the end user's view of the system are made. In POOMA, for example, the focus is on creating meaningful objects and methods that the application scientist can use in developing code. POET extends this by providing computational templates which begin to limit the amount of code required from the user. Finally, PSEs like PDELab need virtually no code from the user and present an encompassing environment exclusively built to solve a single type of problem.

The construction of these systems demands an increasing amount of effort to determine, design, and implement the domain-specific concepts to be supported. Recent work in software engineering attempts to address this problem. The area of software architecture seeks to represent and reason about the structure and topology of software systems. Its motivations and goals were covered in Chapter 2. Our goal in this section is to explore how the area may apply to parallel and distributed computing as well as domain-specific metacomputing.

Software architecture is concerned with the variety of organizational styles for constructing software that have emerged over time. Systems like POOMA [ABCH95, RHCA97] and POET [AM94] clearly exhibit the style of an *object-oriented system*. We naturally use this and similar terms throughout our discussion of object-orientation. Indeed, as we argued earlier, software architecture has evolved from intuition, diagrams, and informal prose—a sort of folklore, if you will, that has been built up over several years. A similar term, *layered system*, is also prevalent in parallel and distributed systems, particularly in the context of networked communications ([SG96], p. 25). A layered system is organized in a hierarchical manner such that each layer provides services to the one above. Despite it being built with object-oriented methods, the PDELab problem-solving environment is primarily built as a "layered architecture" [WHRC94]. Hence, programming style does not necessarily imply architectural style. That POOMA and POET exhibit an object-oriented style results from the manner in which the systems are constructed, not because of the languages in which they are implemented. (Though, in this case, an object-oriented language greatly facilitates the use of an object-oriented architectural style.)

The use of architectural styles is one technique for software reuse. If common software architectures are identified and formalized, then perhaps reusable, modular, style-specific components can be developed. But software engineering has not yet advanced to this point, and the large-scale reuse of generic software is an elusive goal [Chan94, GAO95, HPLM95, SG96, TTC95]. Garlan *et al.* [GAO95] suggest many possible reasons for this phenomenon. Part of the problem is clearly that software designed with reuse in mind is hard to find or simply does not exist. Even among so-called reusable software, low-level interoperability issues such as programming languages, operating systems, and machine platforms often prevent pieces from fitting together. But Garlan *et al.* [GAO95] point to a higher-level, more pervasive problem. Components often make assumptions about the structure of the applications in which they are to appear. These assumptions lead to *architectural mismatches* between components and the applications trying to use them.

The assumptions concern a variety of software construction issues [GAO95]. For example, components often make assumptions about what part of the software holds the main thread of control. If two components both demand it, an architectural mismatch occurs. Similarly, components make assumptions about the format of the data on which they operate. Other assumptions involve application protocols. For instance, if one component uses an event-based model for sharing messages and another one uses procedure calls, the application attempting to integrate these is forced to resolve the difference. The consequences of these assumptions include excessive code size, poor performance, the need to modify external packages, the need to reinvent existing functionality, and an error-prone construction process [GAO95].

Domain-Specific Software Architectures

Indeed, despite promising research [AB96, GAO95, NM95, SDKR95], the *general* problem of software reuse may not be solved for some time. Furthermore, the use of software architectures based on generic software styles only addresses one aspect of the problem revealed by the application of object-oriented technology to parallel computing. That is, software styles do very little in the way of capturing domain-specific representations and integrating them into a software system.

To this end, researchers are exploring an alternative approach: *domain-specific software architectures (DSSAs)*. The ultimate goal of DSSAs is the same as the other technologies discussed in this section: reusability. But DSSAs do not attempt to provide *general* software reuse. Rather, the goal of DSSAs is to facilitate software reuse within classes of applications. By agreeing on certain domain-specific software characteristics, the conflicting assumptions of software components can be eliminated, thus avoiding the architectural mismatches that inhibit reuse. Thus, focusing on particular domains has the effect of reducing the software development problem space, allowing effective software solutions to be found more easily.

However, despite promising DSSA efforts, so far computational science has not been a primary target. Current DSSA domains include avionics, command and control, and vehicle-management ([SG96], p. 32).[2] For example, Hayes-Roth *et al.*

[HPLM95] describe a DSSA for adaptive intelligent systems (*e.g.,* robotics, monitoring systems). They identify the three components of a DSSA: a *reference architecture* that describes the common framework of computation for the domain of interest; a *component library* of reusable "chunks of domain expertise," and an *application configuration method* for choosing and assembling the components required to build a specific application. A full analysis of their system is beyond the scope of this paper. Rather, we focus on a distinguishing feature of DSSAs that is also pertinent to the needs described above: domain modeling.

Domain Modeling

With respect to parallel computing and, more generally, metacomputing, we contend that the only means of improving access to these technologies is to deliver them in domain-specific concepts. We assume that *application scientists* are scientists first and programmers second. Thus, allowing them to program in familiar terms improves their ability to do science and reduces the struggle with high performance computer programming concepts.

To this end, research on the development of *domain models* offers methodologies for collecting, organizing, and applying domain-specific concepts and knowledge during the software construction process. A domain model serves several purposes. For example, domain models provide a standard terminology for describing problems within the domain, and they reveal useful abstractions and patterns of composition. Domain models also establish high-level constraints that software must satisfy [Bato94]. According to Might [Migh95], a domain model also indicates what "functions are being performed and what data, information, and entities are flowing among those functions." The concept of a domain is relative. Domains can vary in breadth, depth, level of abstraction, form, and representation. However, the single, common characteristic among all domain models is that they "represent the functions and flows (or behavior) in the domain of interest" [Migh95].

Taylor *et al.* [TTC95] propose five phases of *domain engineering*— that is, the creation (or evolution) of domain models and their associated software architectures. In the first phase, interviews with users (or domain experts) reveal the contents of the domain and the needs of users in that domain. The second phase attempts to produce a dictionary and thesaurus of domain-specific terminology and to distinguish between essential and optional features. Design and implementation con-

2. The practical reason that DSSA research has not yet targeted computational science domains is that funding has largely originated from the Defense Advanced Research Projects Agency (DARPA), whose focus is naturally on defense-related application areas [Trac94].

straints are developed in the third stage. Whereas the first two stages primarily answered questions about *what* the domain is, the third stages begins to explore *how* domain knowledge will manifest itself. The fourth phase addresses the design and analysis of the domain-specific software architecture through an iterative process that is applied at successively lower levels of abstractions until the desired detail is achieved. During the final step, the DSSA is populated with reusable software components. From the preceding description, it is clear that developing a domain model and software architecture is a complex and time consuming process. The DSSA for adaptive intelligent systems developed by Hayes-Roth *et al.* [HPLM95] reportedly took several person-years to design, implement, test, debug, and document. But Hayes-Roth *et al.* also note that there is only marginal additional costs for developing software that, in addition to meeting its functional objectives, is reusable. Furthermore, the additional cost must be weighed against the cost-savings of subsequent reuse.

Domain-Specific Software Architectures for Computational Science Problems

As stated earlier, DSSA technology has rarely been applied to computational science problems. One of the distinguishing characteristics of these problems is their experimental nature. In many cases, this nature makes it impossible to form complete specifications of desired functionality, requirements, constraints, or problem description. In fact, this proves to be one of the limiting factors in successfully applying problem-solving environments to computational science problems. What are the prospects, then, of applying DSSA technology to these same problems? This section briefly speculates on the answer to that question.

First, domain-specific software architectures and problem-solving environments are fundamentally different. DSSAs primarily address software development concerns whereas PSEs address issues of software use and usability. The connection between the two is that by applying DSSAs to computational science, someday it may be possible to build software that *can* address software use and usability issues for this type of problem.

Second, by definition, DSSAs (unlike PSEs) do not result in “point solutions” [TTC95]. In fact, Might [Migh95] claims that a key function of domain models is “to systematically study the impact of alternative strategies and policies.” Much of the experimentation in computational science involves just that—alternative strategies and policies for simulating, parameterizing, and analyzing computational models of physical phenomena. Even though Might targets software development supporting business-oriented processes (in the healthcare industry, in particular), his comments apply equally well to computational scientists in search of a scientific result, but lacking an obvious solution for attaining it.

Third, as we describe above, domains are relative. In particular, domains vary in depth and in level of detail and abstraction. A reasonable assumption is that the quality of a domain has direct bearing on the quality of resultant domain-specific software architectures, not to mention the quality of applications derived from that architecture. In many cases, though, while the science being conducted is experimental, the computational problems are not. The computational problems may change and evolve over time, but they do so within a common set of terminology, concepts, assumptions, and goals. They occur within a common domain.

Thus, computational science problems appear amenable to domain-specific software architectures. Anglano, Schopf, Wolski, and Berman [ASWB95] describe a system for hierarchically describing and characterizing heterogeneous, computational science applications that execute on high performance systems. Their system, Zoom, is *not* a domain-specific software architecture, but it does have several similarities to domain modeling and software architecture. We briefly explore the differences and similarities below.

There are three major properties that distinguish Zoom from a DSSA for computational science metacomputing. The first major difference is that Zoom does not explicitly target computational science domains. Instead, Zoom addresses the somewhat broader and differently focused “domain” of heterogeneous applications. In this way, Zoom does not directly facilitate the integration of domain knowledge into software and tool systems. Second, given its domain, Zoom is naturally concerned primarily with performance. As a result, Zoom characterizes applications at a lower level than what might be envisioned for domain-specific metacomputing. The reason for this is that the applications targeted by Zoom do not assume the existence of a computational infrastructure that handles requirements like resource allocation, scheduling, and monitoring. Finally, Zoom is primarily a mechanism for representing and reasoning about applications. Thus, unlike a software architecture, it does not support the subsequent instantiation of those applications.

Despite these differences, Zoom has much in common with the DSSA process. At its highest level, Zoom serves as a “domain of discourse” between computer scientists and computational scientists [ASWB95], making it consistent with the first two stages of the DSSA process, as described by Taylor *et al.* [TTC95]. Next, Zoom identifies computational requirements, constraints, and options with respect to implementations, data format and structure conversions, and performance trade-offs. This corresponds to the third phase of the DSSA process. It also echoes Might's [Migh95] claim that domain models should support consideration of different implementation options. Just as Taylor *et al.* [TTC95] describe the fourth phase as an iterative process for defining a domain architecture at successively lower levels of abstraction, Zoom supports a hierarchical representation that reveals increasing amounts of application detail at each of three levels. Zoom also tries to address one of the notable shortcomings of the object-oriented systems described earlier: tool support. At its deepest level, Zoom provides information necessary for program development tools and the "accurate cost models required for optimization and scheduling" [ASWB95]. Anglano *et al.* hope to use Zoom as an interface to a suite of tools for developing and managing heterogeneous computing applications. DSSAs also strive to facilitate the development of tools in addition to applications [HPLM95]. Finally, Anglano *et al.* propose a well-defined, (mostly) graphical notation for describing applications that is similar to the block diagrams used during the DSSA process [TTC95] and in software architecture in general ([SG96], pp. 160-163).

Zoom provides a compelling glimpse into the application of DSSA technology to high performance computing and computational science. But what are the possible advantages to using a DSSA approach over object-oriented class hierarchies and object-oriented systems for parallel computation?

Conclusion

The most obvious benefit of DSSAs is that they define methodologies for integrating domain knowledge into software systems. Object-oriented systems like Norton's plasma simulation [NSD95], POOMA [ABCH95, RHCA97], and POET [AM94] use ad hoc, informal methods for incorporating domain-specific concepts into user applications or their own software. Furthermore, these systems do not address the more challenging problem of supporting computation within a metacomputing environment.

Systems like Legion [GW96] and Globus [FK96] strive to support generic metacomputing capabilities. In addition, interoperability of components within these systems is not reinforced by the careful modeling and analysis encouraged by software architecture. Together, these traits make generic metacomputing systems susceptible to the same poor reusability exhibited by other general software. With respect to domain knowledge, some of the object-oriented systems described earlier allow it to be reflected in application programming languages. However, reflecting that same knowledge in associated tools (*e.g.,* debuggers, performance analyzers, visualizers) is difficult and not always possible using ad hoc methods. Not surprisingly, domain-specific tools are rare. A lack of incorporated domain knowledge in both languages and tools allows the relatively poor usability of parallel and distributed systems to persist.

In summary, domain-specific software architectures address both problems of usability and reusability. Domain modeling offers a well-defined process for collecting, organizing, and applying domain knowledge to the software construction process. And software architecture encourages careful consideration of how software is structured as components and connectors, supporting several nonfunctional software requirements like reusability, interoperability, and extensibility. Ultimately, applying domain-specific software architecture technology to metacomputing for computational science results in applications and tools that appear cognizant of a scientist's domain. Such systems may be similar to domain-specific environments, a topic of the next chapter.

CHAPTER 5

Supportive Research In Computational Science

WHEREAS DOMAIN-SPECIFIC software architectures address the integration of domain knowledge into well-engineered software systems, they do not guarantee that the computational and analytical capabilities of the resultant system actually meet the demands of the scientist. After all, computational science problems are characterized by diverse computational and analytical requirements that evolve over time. DSSAs clearly focus primarily on design issues.

Problem-Solving Environments

Computational scientists are less concerned with how well-designed a software system is than with how well it solves the specific computational problems they have. In this regard, problem-solving environments are touted as a major breakthrough for computational science because they provide highly usable and useful computational capabilities [GHR94, Rice96, WHRC94]. (As we have mentioned earlier, PSEs also emphasize careful software design [GHR94, MCWH95], but that is not the primary concern here.) Gallopoulos *et al.* [GHR94] identify three measures of problem-solving environments: scope, power, and reliability. *Scope* refers to the number of problems that a PSE addresses. Generally speaking, a PSE becomes easier to build as its scope is reduced. The *power* of a PSE measures its ability to actually solve the problems that can be posed to it. As scope increases, the likelihood that some of the problems cannot be solved also increases. Finally, *reliability* refers to a PSE's ability to produce correct solutions. For example, returning a message indicating the PSE's inability to solve the problem is much more desirable than a wrong answer [GHR94].

Problem-solving environments seek depth of support for specific computational functionality. As an example, the PDELab problem-solving environment [WHRC94] specializes in solving partial differential equations (PDEs). PDELab attempts to support the entire spectrum of activities in which scientists might engage while working with PDEs, from brain storming, trial and error reasoning, and simulation to optimization, visualization, and interpretation. PDELab's goal is to *emulate* all of these processes and to *automate* many of them. PDELab is separated into three logical layers, each of which provides support in several areas:

Application Development Framework: specification tools, computational skeletons, knowledge sources, and visual programming tools.

Software Infrastructure: software bus, object-oriented user interfaces, programming environments, language translation systems, graphics display systems, and machine managers.

Algorithms and Systems Infrastructure: numerical libraries, parallel compilers, and expert system engines.

There is little doubt that systems like PDELab hold significant promise for certain scientists. Indeed, many isolated physical phenomena can be modeled by systems of partial differential equations [GHR94, HCYH94, WHRC94]. But increasingly, scientists are interested in coupling two or more distinct models together. For example, in climate modeling, it is desirable to couple unique models of atmospheric and oceanic circulation and chemistry, land surface and sea ice processes, and trace gas biogeochemistry [MABB94]. Each of these models could *individually* be addressable by a unique PSE. But supporting composite and complex processes like climate modeling is beyond both the scope and the power of existing PSEs.

Multi-Component Modeling

The example of climate modeling is indicative of a more general trend in computational science toward whole-system modeling. For example, the Accelerated Strategic Computing Initiative (ASCI) calls for "full-system" and "full-physics" applications for maintaining the nation's nuclear stockpile [DOE96]. *Full-system* refers to all the components (nuclear and non-nuclear) of a weapons system; *full-physics* refers to all the modeling techniques (*e.g.,* physical, chemical, material, and engineering) required for simulation. Similarly, automobile manufacturers desire more robust modeling capabilities. Houstis *et al.* [HRJW95] detail the diverse problem areas encountered in modeling an engine:

> The analysis of an engine involves... thermodynamics (gives the behavior of the gases in the piston-cylinder assemblies), mechanics (gives the kinematic and dynamic behaviors of pistons, links, cranks, etc.), structures (gives the stresses and strains on the parts) and geometries (gives the shape of the components and the structural constraints).

The total design and analysis of an engine requires that solutions to these individual problem areas interact to determine a final solution. Drashansky *et al.* [DJRH97] summarize the evolution of multi-component physical systems:

> Computational modeling will shift from the current single physical component design to entire physical systems with many components that have different shapes, obey different physical laws, and interact with each other through geometric and physical interfaces.

Thus, for the experimental computational scientist, PSEs pose a dilemma. By definition, a PSE provides computational support for a single, well-defined, and well-understood type of problem [GHR94]. The computational scientist, however, may require support for several such problems. Worse yet, some of those problems may be poorly defined and/or poorly understood [BRRM95, CDHH96, HRJW95].

Multidisciplinary Problem-Solving Environments

Recognizing this and having had at least moderate success with prototype problem-solving environments, the PSE community is now pursuing the creation of *multidisciplinary problem-solving environments (MPSEs)* [HRJW95]. As suggested above, an MPSE provides a framework and software kernel through which multiple problem-specific PSEs are brought to bear on complex and composite problems. Houstis *et al.* [HRJW95] identify three general requirements of MPSEs:

- MPSEs must be applicable to a wide variety of problems.
- MPSEs must allow software reuse.

- MPSEs must be able to accommodate “reasonably fast numerical methods.”

MPSEs and PSEs share an emphasis on sound software engineering practices, but MPSEs are particularly concerned with achieving a high degree of software reuse because “without software reuse, it is impractical for anyone to create on his own a large software system for a reasonably complicated application” [HRJW95].

MPSEs also place a greater emphasis on network-based computing than do PSEs. Researchers envision agent-based systems accessible in much the same fashion as the World Wide Web is today. In this scenario, MPSE servers export user interface agents to local desktop computers through which users build and operate MPSE applications. In turn, the MPSE server requests services from a diverse metacomputing environment. In addition to computational hardware, the metacomputing environment contains computational software in the form of traditional problem-solving environments. Each PSE is contained in an *agent wrapper* that interacts with and controls the execution of the PSE to solve components of larger, multidisciplinary problems [DJRH97].

Whereas basic PSEs provide a consistent abstraction to component numerical libraries and modules, MPSEs create yet another level of abstraction where entire PSEs are the objects of composition. While we discuss the issue of abstraction in the next chapter, we simply pose a question here: how many layers of abstraction are necessary? In addition, since PSEs are constructed as complete systems, it is not clear how easily they could be used as components in a larger system. It is widely accepted that reusable components must be designed and implemented with reuse in mind.

The result of this continuing composition of systems is larger, bulkier software. With each component PSE containing on the order of one million lines of code [JDRW96], it remains to be seen if MPSEs ever achieve the ubiquity predicted by their proponents. Currently, their promising vision is far from being readily available. Even basic PSEs are still mostly an emerging technology with very few prototypes in existence. It is not surprising, then, that an early prototype MPSE [JDRW96] contains just one component PSE, which evolved out of the canonical PSE example, PDELab [WHRC94].

Even if efforts to build MPSEs are successful, MPSEs are ultimately destined to the same limitations of PSEs. With each component PSE able to address only a well-defined, well-understood computational process, MPSEs offer no way to break free of that restriction. Ultimately, to support experimental computational science—

where a simulated process may not yet be clearly understood—a different means of support is required. The next section explores one such means.

Domain-Specific Environments

Supporting experimental computational science requires a fundamentally different approach to problem-solving. The *problems* of experimental computational science are much more diverse than those addressed by PSEs. It may be the case that scientists are trying to refine a method of simulation for a particular physical phenomenon. Or, perhaps several different techniques for modeling a biological, ecological, or environmental system are being explored. In other words, the problems of experimental computational science are often exploratory in nature. They may not be well-defined, and they are rarely well-understood. In addition, the problem-solving process may include several loosely connected steps or stages of analysis, each with its own unique requirements.

While this type of exploratory problem-solving may eventually evolve into a methodology that could be encapsulated in a PSE, until that happens, scientists still require the support of a robust computational environment. In fact, the science they conduct today is what ultimately allows their results to be applied more broadly tomorrow.

To this end, we propose and motivate three critical nonfunctional requirements for the *domain-specific environments (DSEs)*[1] required by exploratory computational scientists:

> *Programmability* facilitates rapid prototyping and provides a mechanism by which domain abstractions and user-defined functionality become an integral and essential part of the environment.
>
> *Extensibility* allows the environment to be enhanced with new, unanticipated functionality and problem-solving techniques as the scientific process evolves over time.

1. Our use of the term "domain-specific environment" is consistent with the apparent first application of the term in this way by Cuny *et al.* [CDHH96]. In the same work, Cuny *et al.* also propose these three requirements for DSEs, though little discussion of their general importance to DSEs is presented therein. We attempt to expand on these ideas here and later in this chapter.

> *Interoperability* allows a wide range of tools to be brought to bear on the scientific process and enables the loosely connected components of the environment to work together via well-defined, open programming and communication interfaces.

Each of these requirements address limitations of problem-solving environments and multidisciplinary problem-solving environments. In particular, PSEs are programmable only at the highest levels where they support visual programming and some template-based techniques. In contrast, DSEs provide frameworks of tools and analytical support that can be tailored (albeit at a lower level) to the unique requirements of each phase of the exploratory process.

PSEs provide extensibility by supporting the integration of foreign libraries and systems. Unfortunately, the new functionality is only available in the confines of the problem-solving context supported by the particular PSE. DSEs, however, permit the problem-solving context to expand and contract as necessary by maintaining a looser coupling between components, which, in turn, allows unanticipated functionality to be integrated more easily.

Finally, PSEs attempt to provide a complete set of interoperable tools necessary for supporting the problem-solving process; thus, the need to apply external tools is not a primary consideration. MPSEs create interoperability among several PSEs, but they do not facilitate external tools and systems for the same reason. DSEs, on the other hand, try to recognize that scientists may have pre-existing tools and systems that they desire in an initial DSE implementation. Simultaneously, DSEs recognize that as the scientific process evolves, those tools may be replaced or augmented with others. Being able to use commercial, off-the-shelf software as components is also critical. To this end, DSEs emphasize open and well-defined interfaces as a means of increasing this type of component-level interoperability.

These distinctions reveal a fundamental difference between PSEs and DSEs: *Whereas PSEs attempt to support specific problem-solving methods, DSEs attempt to support the evolution of such methods.* Thus, in many regards, PSEs and DSEs attempt to address very different needs. We contend that domain-specific environments are particularly applicable to exploratory computational science in metacomputing environments. The remainder of this chapter will describe two such environments.

A Domain-Specific Environment for Environmental Modeling

The purpose of the Geographic Environmental Modeling Systems (GEMS) is to provide general environmental modeling capabilities to "policy technicians" in regulatory agencies [BRRM95]. GEMS assists them in developing cost-effective controls for hazardous waste, air pollution, acid rain, and global climate change by providing a comprehensive computational environment which includes transparent access to parallel and distributed computing over high-speed networks; geographically distributed, object-oriented databases; and a robust graphical user interface.

At first glance, GEMS appears to be more like a problem-solving environment: it targets less technical end users, it provides general computational capabilities for a specific problem area, and it provides transparent access to a collection of underlying computational resources. Indeed, GEMS has much in common with the ideals of problem-solving environments. However, a closer examination reveals an overwhelming consistency with the characteristics and objectives of domain-specific environments. This section explores the GEMS system as a representative example of a domain-specific environment.[2]

With respect to problem area, environmental modeling is far from the requisite well-defined, well-understood process suitable to a PSE. In fact, the science behind environmental problems is not widely agreed upon. Consequently, organizations currently use several different physical and chemical models, no one of which is recognized as being superior [BRRM95]. Furthermore, policy makers require extensive "what-if" capabilities to explore the implications of various control strategies. This type of analysis is complicated since air quality laws in the United States are largely "goal-oriented." That is, state and local policy makers can meet federal air quality regulations in a variety of ways [BRRM95]. Hence, environmental modeling (and the subsequent policy making) is largely an exploratory process in the context of uncertain scientific laws.

Design. Prior to the development of GEMS, researchers made several design decisions. Perhaps the most critical of these was the recognition that the design team itself had to be composed of both computer scientists and application scientists [BRRM95]:

> Software engineers by themselves shouldn't try to develop a system with such ill-defined requirements.... [They] require the active participation of domain experts

2. The creators of GEMS, Bruegge *et al.* [BRRM95], do not use the term "domain-specific environment" to describe their system.

> and end users, people who have traditionally been considered "outsiders" in the software development process. At the same time, environmental engineers cannot take risks with the most recent advances in computing technology without seeking to include computer science experts.

What is notable about this statement is the *mutual* need for collaboration. That is, each group (computer scientists and application scientists) views the other as "domain experts." In addition, though, Bruegge *et al.* suggest that "ill-defined requirements" are responsible for creating this need. The lack of requirements demanded that the developers identify a series of design goals.

One of the general goals was to develop an "open" modeling system. Whereas PSEs function more like "black boxes," Bruegge *et al.* [BRRM95] envisioned a "glass box" approach, which echoes the properties of programmability, extensibility, and interoperability:

> Existing modeling systems have always been developed for a specific end-to-end purpose.... If the component parts were open to inspection, while at the same time maintaining internal consistency—hence the term "glass box"—it would be much easier to combine the parts to work as a coherent whole.

In fact, many of the design goals reflect these properties as well as the trade-offs that often arise among nonfunctional requirements. For example, the developers had to balance between making the system *easy to use* and *accessible* to nonexperts and simultaneously providing experienced users with access to sufficient *power* and *flexibility* to which they may be accustomed. A similar trade-off arose as the team considered how to provide robust visualization support:

> The main trade-off here is between providing sufficient power and flexibility to achieve a sophisticated analysis and making the system easy to use for more frequent rudimentary analyses.

In general, the developers wanted GEMS to act as a framework that could integrate a diverse set of known and unknown capabilities. Many of the specific design goals contributing to this vision are consistent with the ideas of domain-specific environments:

- Incorporate existing functionality as system components (interoperability).
- Add new analysis capabilities (programmability, extensibility).
- Incorporate additional data sets (extensibility, interoperability).
- Change underlying model assumptions (programmability).

- Add new and improved computational models (programmability, extensibility).
- Let users choose among different models (programmability).

One of the most unique aspects of GEMS is the intense collaboration between computer scientists and application scientists to build a system that exhibited good software engineering principles and could deliver adequate computational performance, but that was also responsive to user needs and reflected domain knowledge of environmental modeling. The developers adopted a development model based on an object-oriented notation. Similar to the role played by the Zoom system by Anglano *et al.* [ASWB95], this notation served as a "two-way mirror" between the software engineers and environmental modeling experts [BRRM95].

Implementation. The GEMS prototype consists of about 75,000 lines of C++ code—an order of magnitude less than that required by problem-solving environments. As mentioned earlier, Joshi *et al.* [JDRW96] claim PSEs can have as many as one million lines of code. Why is there such a large differential in code size between these two types of environments? The answer is not completely clear, but the DSE properties described earlier offer some possibilities. First, DSEs emphasize interoperability with *existing* systems. A relatively small amount of carefully constructed interface code allows a wide range of functionality to be available through the environment without having to be expressed as part of the system code. Second, programmability offers a similar cost-savings; domain knowledge, tool customization, and a variety of system-wide configurations are expressed outside of the core system code. Finally, the need for extensibility suggests that not all of the anticipated functionality is yet part of the system; hence, the system is operable and usable despite being functionally incomplete. The important observation, though, is that because of the nature of their science, the scientists *may not yet know* what the missing functionality is. A DSE is designed and built to accommodate this evolution. As a result, a preliminary incarnation of a DSE likely contains fewer lines of code.

The implementation of the GEMS framework is based on a collection of five core components: user interface, visualization, execution, monitoring, and data management. The GEMS graphical user interface provides a point-and-click interface to system functionality. Through extensive and iterative prototyping, the GEMS interface reflects the user's view of environmental modeling with tools like *Map View*, *Chemical Lab*, and *Population*. Rather than a bulky and inconvenient afterthought, visualization is an integral part of the interface used for both specification and results analysis.

The execution component facilitates access to the underlying environmental models; it is not the model itself [BRRM95]. The goal is to allow the user to choose and interact with the model. To this end, the execution component uses an event-based system to handle the communication between the computational model and the other components of the framework. GEMS targets a metacomputing environment in which the computationally intense models run on supercomputers or networks of workstations, visualization services utilize graphics displays, databases reside on geographically distributed servers, and the user interface appears on the users workstation. An event-based system provides a loose, yet flexible, coupling among GEMS components. It also limits the amount of GEMS-specific code that must be inserted into the models themselves to make them compatible with the rest of the system [BRRM95].

Earlier, we suggested that one of the reasons DSE code size might be significantly less than PSEs was because DSEs often rely on existing systems. The GEMS visualization capabilities provides a good example of this by using a commercial off-the-shelf visualization system (PV-Wave). The visualization component acts as an interface between the GEMS user interface and the PV-Wave rendering system. Not only does this reduce the code size, it offers better visualization capabilities and in a shorter amount of time than the developers could create on their own. Similar to the visualization component, the GEMS monitoring component integrates a pre-existing distributed monitoring system for identifying and debugging performance bottlenecks.

With respect to data management, the developers were attracted to the rich data model supported by object-oriented database management systems (OODBMS). They adopted this paradigm because it showed “considerable promise in terms of ease of use and... flexibility” [BRRM95]. However, meeting the nonfunctional requirements of usability and flexibility came at the expense of performance:

> The largest difficulty has been the performance penalty incurred with an OODBMS.... [and] we have not yet found a package that adequately meets the requirements of our system in practice.... [The] use of an OODBMS to manage the objects being displayed leads to unacceptable response times for an interactive system.

In summary, the GEMS framework provides a PSE-like user experience but addresses a much broader range of nonfunctional requirements essential to exploratory computational science. Key to accomplishing this is the design of a loosely coupled framework that can incorporate existing systems (interoperability), accommodate new functionality (extensibility), and allow the environment to be tailored to user needs (programmability). As such, GEMS is an excellent example of a domain-specific environment.

A Domain-Specific Environment For Seismic Tomography

The goal of the Tomographic Imaging Environment for Ridge Research and Analysis (TIERRA) project is to provide "a computational environment for tomographic image analysis for marine seismologists studying the formation and structure of volcanic mid-ocean ridges" [CDHH96]. The scientists are concerned with areas of the ocean floor where magma rises up from the Earth's mantle to form volcanoes. To understand the magmatic, hydrothermal, and tectonic processes, the scientists use a method called "seismic tomography" (which is similar to medical CAT scans) to construct 3D models and images of the ocean floor. The models and images are created by extensive computations on seismic wave data collected by seismometers positioned on the seafloor.

The goal of the environment is to support the general problem-solving process that the scientists use in their analysis; the computation of models and images is only one part of this process. As Cuny *et al.* [CDHH96] note, "the full analysis requires extensive, domain-specific input from a geoscientist."

Again, the similarity to PSEs is evident in this description. Cuny *et al.* [CDHH96] recognize this similarity and point to the same important difference proposed earlier: "the leading-edge science applications we are concerned about are exploratory and experimental in nature.... [and] demand domain-specific support that can adapt to changing requirements." Seismic tomography is characterized by a significant amount of interaction with, and intervention by, the scientist. Creating a sea floor model is not merely a task of specifying a few parameters, providing a dataset, and running a computation. The scientist actually plays an integral role in the validation, optimization, and convergence of the final model [CDHH96]. This is reflective of the poorly defined and poorly understood nature of this computational domain, which, in turn, makes it unsuitable as a PSE target.

Since the scientist plays such a central role in the problem-solving process, it is imperative that the environment fully consider their requirements. To this end, TIERRA adopted a collaborative process for design and implementation [CDHH96]:

> The application scientists must establish requirements for problem investigation and evaluate the environment as its implementation proceeds. The computer scien-

tists must fashion available technologies to address the requirements and possibly even develop new infrastructure to complete the environment.

Design. The TIERRA design process employed an informal method for identifying the requirements of the environment. After scientists described their overall problem-solving process, the group partitioned it into a series of seven steps. From each step, a list of domain-specific requirements was generated. The steps in the problem-solving process are diverse and include data processing, verification and validation, and testing and optimization. The corresponding computational requirements include tools and support for data manipulation; sophisticated, interactive visualization; improved performance; and program interaction and steering.

However, not all of these requirements were apparent *ab initio*; in some cases, they evolved over the course of the collaboration. The need for program (computational) steering is such a requirement. Far and away the most critical requirement for the geoscientists at the beginning of the collaboration was performance. Their previous computational support required approximately 6 hours per iteration of the main computation. The large amount of time required to generate models resulted in a decoupled and poorly integrated problem-solving process. The preliminary, trivial parallelization of the main computation, however, reduced the iteration time to about 25 minutes. This performance improvement “allowed more rapid analysis and the consideration of considerably larger data sets” [CDHH96]. The interesting side effect was that as a result of the performance increase, the computational analysis bottleneck shifted from model generation to model evaluation. Suddenly, it became possible to view the problem-solving process not as a “series of isolated computer runs, but as a series of steering operations on a single (long) computation” [CDHH96]. Thus, program interaction and computational steering became a requirement of the evolving environment.

A similar process led to the requirement of more sophisticated, interactive, 3D visualization. Scientists were originally content with simple, 2D plots created offline. However, model representations depicted in three dimensions were more easily understood, and since model evaluation had become more of a bottleneck, the scientists “became more convinced of their utility” [CDHH96].

Implementation. The features of programmability, extensibility, and interoperability are primarily evident in the two main tool components of the TIERRA framework. One component, DAQV (for Distributed Array Query and Visualization) [HM96], handles program interaction and computational steering. The other component, Viz [HHHM96], is a visualization programming system. Both systems embody all three of the desirable DSE features.

DAQV is a software library through which a parallel program with distributed data makes its data available to external tools. Tools, on the other hand, are presented with a logical, global view of program data and can make simple, high-level requests for data. DAQV accesses distributed data through a library of programmable data access functions. In addition, the library of access functions can be extended to perform preprocessing, reductions, or compositions on the distributed data before it is delivered to external tools. Finally, interoperability is achieved by using an open client/server protocol based on standard, socket-based, interprocess communication [CDHH96, HM96]. It is interesting to note that in its original form, DAQV did not support computational steering. However, because of its extensible design and open communication protocol, DAQV was easily extended to support this capability [CDHH96].

For visualization support, TIERRA relies on Viz, a highly programmable and extensible visualization toolkit.[3] Visualizations are created through an interpreted, high-level language that provides access to an object-oriented, 3D graphics library. The robust language features and graphics model accessible through Viz enable a high degree of programmability and extensibility. Interoperability and extensibility are also enhanced by the ability to incorporate and access external libraries through "foreign function interfaces" [CDHH96].

Whereas Bruegge *et al.* [BRRM95] present GEMS as a functionally complete system, TIERRA continues to evolve:

> The requirements for our environment are driven by the scientific discovery process. As we provide increasing support for the activities of the seismologist, the way in which they use the environment—and thus the requirements for the environment—change. Given the experimental nature of our application domain, this evolution will continue.

The researchers point to several future improvements, including new visualizations, better support for data management, improved performance, more flexible and interactive program control, and portability.

In summary, Cuny *et al.* propose the term *domain-specific environment*, which we have also adopted, to describe an environment suitable for exploratory, ill-defined, and poorly understood problems and their associated computational requirements.

3. Since the description of TIERRA by Cuny *et al.* [CDHH96], the TIERRA environment has abandoned use of Viz. Future work will likely utilize easier to use and more stable visualization packages [Harr97]. In terms of nonfunctional requirements, this suggests that programmability and extensibility in the visualization component proved to be less important than ease of use.

However, the goal of their work, as well as that of Bruegge *et al.*, is not to develop a theory of DSEs. To the contrary, these reports are "experiential"; the researchers did not seek general principles, nor did they fully discover any [CDHH96].

Conclusion

The emergence of problem-solving environments, multidisciplinary problem-solving environments, and domain-specific environments strongly suggests a need for improved computational environment support. As Cuny *et al.* [CDHH96] point out, performance used to be the sole concern for application scientists. But now, while performance is still important, that need is more often balanced with a demand for "more robust, integrated support from computational tools of diverse types" [CDHH96]. The tendency for PSEs, MPSEs, and DSEs to support a *range* of problem-solving capabilities is indicative of this trend.

Just as scientists are engaged in different types of research, these environments address different needs. PSEs provide depth of support for a particular well-defined and well-understood computational problem. If the science is expressible as that type of problem, then a scientist can apply a PSE to their work. MPSEs seek to bring several individual PSEs to bear on more complex phenomena. But again, each part of the whole problem must be expressible in a form that can be addressed by a component PSE. Finally, DSEs fill the gap left by PSEs and MPSEs by addressing exploratory computational science problems that lack a precise definition, exhibit several loosely coupled stages of analysis, and evolve over time.

Domain-specific environments, like the very problems they address, are not a well-understood concept; there are no hard and fast rules for design or implementation. However, we identify some general characteristics of DSEs that at least distinguish them from problem-solving environments. In particular, DSEs are characterized by an intense, if mostly informal, design and development collaboration between computer scientists and application scientists that can, if desired, result in a loosely coupled framework capable of high degrees of extensibility, programmability, and interoperability. Collaboration by no means guarantees this result; designers and implementers must make conscious decisions to avoid creating ad hoc systems. DSEs do not take a formal approach to achieving domain-specificity like domain-specific software architectures (DSSAs); rather, domain-specificity results as a natural side-effect of the collaboration between scientist and software engineer. More generally, DSEs do not universally place a great emphasis on software engineering; rather, software engineering techniques are employed where convenient or necessary, and where they may directly result in meeting important nonfunctional requirements (*e.g.,* extensibility, interoperability, portability, etc.). Finally, by way of the collaborative process, DSEs end up addressing the needs of *specific* scientists as opposed to scientists *in general*. It is perhaps this characteristic more than any other that holds particular promise with respect to the usefulness and usability of the resultant environments. We envision a similar approach to building domain-specific metacomputing environments for computational science.

CHAPTER 6 Synthesis

WE SEEK to “synthesize” the area of domain-specific metacomputing for computational science in this chapter. We begin with a broad overview of the preceding chapters. Next, we integrate the component technologies of the area by discussing three key “integrating concepts” common to all of the technologies discussed earlier. These concepts are nonfunctional requirements, software frameworks, and abstraction. Each concept has a direct bearing on a specific stage in the lifecycle of software (*i.e.,* design, implementation, and use). Given this foundation, we speculate on the actual construction and evaluation of domain-specific metacomputing environments and conclude by identifying the key open questions in the area.

Overview

We propose three broad requirements for domain-specific metacomputing for computational science in Chapter 2: high performance heterogeneous computing, software design and development, and domain-specificity. Subsequently, we suggest that three broad, enabling technologies collectively address these requirements: metacomputing, software architecture, and domain-specific environments. The intricate relationship between the requirements and technologies of domain-specific metacomputing for computational science is depicted in Figure 5, where each requirement is addressed in a primary way by one of the technologies and in a secondary way by another technology. Only together can these technologies fully address the needs of this emerging area. Chapters 3, 4, and 5 bear out these relationships in extensive detail. In addition, the chapters present other research that is relevant to domain-specific metacomputing for computational science.

Since performance is so central to a scientist's ability to conduct simulation-based research, we consider parallel and distributed computing to be the foundation of metacomputing for computational science. Chapter 3 examines the major development efforts in metacomputing and also explores several other projects addressing related issues such as real-time system information and application scheduling. The primary goal of the work in metacomputing is achieving high performance in heterogeneous computing environments. Many of the projects, though, also emphasize the software design and development process, employing modular frameworks or object-oriented approaches.

To address issues of software construction in a more direct way, Chapter 4 seeks out supporting research in software engineering. While ultimately focusing on software architecture and domain-specific software architecture, in particular, the chapter also explores the role that object-oriented technology plays in the parallel and distributed computing community. These techniques, in addition to offering improved methods for software design and development, also facilitate the integration of domain-specificity into software.

Finally, supporting efforts within (or at least targeting) the computational science community to develop computational support environments are discussed in Chapter 5. Two main technologies, problem-solving environments and domain-specific environments, are presented. These environments exhibit different types of domain-specificity, and we ultimately conclude that domain-specific environments are better suited to the challenging and exploratory problems in computational science. In addition to domain-specificity, though, access to and delivery of high performance computing capabilities is central to both types of environments.

The Role of Nonfunctional Requirements in Software Design

We consider parallel and distributed computing to be the foundational area for domain-specific metacomputing for computational science, with software engineering and computational science taking on supportive roles. With performance so central to many scientists' ability to conduct their work, this seems a reasonable enough approach. But actually, this only represents one perspective of domain-specific metacomputing.

In earlier chapters, we refer to *nonfunctional requirements* and their effect on design and implementation decisions. Sommerville ([Somm89], p. 101) describes nonfunctional requirements as restrictions or constraints placed on a software system and notes that nonfunctional requirements typically interact and conflict with the functional requirements of a system. Our view of nonfunctional requirements is consistent with this, if not somewhat more general.

The nonfunctional requirements of a software system are, by nature, often hard to recognize. They are even harder to express in a formal way. Sommerville ([Somm89], p. 88-89, 101-102) indicates that analyzing nonfunctional requirements can be difficult for several reasons: they are often expressed in natural language; the relationship between nonfunctional requirements and functional requirements is not always easy to discern; and, when the relationship is understood, nonfunctional requirements often relate to multiple functional requirements. Furthermore, failing to meet nonfunctional requirements results in what are generally accepted as being the most costly problems to fix once the software is complete. Yet, nonfunctional requirements have received relatively little research attention [MCN92].

Indeed, recognizing, expressing, and analyzing nonfunctional requirements poses a significant challenge to the software designer. In fact, our choice to "base" domain-specific metacomputing for computational science in parallel and distributed computing results from our (implicitly stated) perception that performance is a very critical nonfunctional requirement for this area. To that end, we naturally take a *performance-centric* view of the problem. However, we could equally well take a *software-centric* view and consider the software engineering problems to be the most crucial. Or, we could take a *domain-centric* view and approach the problem primarily from the standpoint of (a particular domain of) computational science.[1] The best approach to a given problem —that is, the perspective a software developer takes toward a system—ultimately depends on their perception of the relative importance of numerous (and possibly unstated) nonfunctional requirements.

A good example of how nonfunctional requirements can affect the design process is seen in how a developer might go about integrating domain-specificity into a meta-

1. Additional viewpoints are undoubtedly possible.

computing environment. Domain-specific software architectures and domain-specific environments each offer a means of addressing this requirement. DSSAs support a formal model for integrating domain-specificity into the fundamental design of software. With an emphasis on careful software design, this approach may be best suited to nonfunctional requirements like reusability and reliability. DSEs, on the other hand, support more informal methods of achieving domain-specificity and place a greater emphasis on the actual development of the underlying software. This tends to result in systems that better satisfy nonfunctional requirements like flexibility, extensibility, and interoperability. Similar scenarios surrounding the requirements of high performance heterogeneous computing (generic metacomputing systems vs. domain-/problem-specific support) and software design and development (informal, convenient use of basic software engineering techniques vs. formal application of software architecture) also exist.

In this way, our organization of requirements and technologies for domain-specific metacomputing, as depicted in Figure 5, creates a "solution space" within which nonfunctional requirements may be used to discriminate between various design and development options. As we have stated earlier, nonfunctional requirements often result in trade-offs, both with respect to other nonfunctional requirements as well as the functional requirements they impact.

There are numerous nonfunctional requirements that may be of interest to domain-specific metacomputing for computational science. Unfortunately, a complete list of nonfunctional requirements does not exist [MCN92], and it is beyond the scope of this work to determine and describe just the ones applicable to domain-specific metacomputing. Several examples of nonfunctional requirements, however, are derivable from our discussion so far: extensibility, flexibility, interoperability, performance, programmability, reliability, reusability, and usability. Sommerville ([Somm89], p. 101) offers a few more common nonfunctional requirements for software: speed, size, robustness, and portability. Mylopoulos *et al.* identify three broad categories of nonfunctional requirements. In addition, for each category, they list several user concerns and associated nonfunctional requirements. Their classification is recreated in Figure 9. In a separate dimension, they also classify nonfunctional requirements with respect to who observes them. For example, nonfunctional requirements like efficiency, correctness, and interoperability are "consumer-oriented"—that is, observable by the user. Other requirements, such as extensibility, reusability, and maintainability are "technically-oriented"—that is, they are apparent to the developer [MCN92].

Thus, nonfunctional requirements ultimately impact the design phase of the software lifecycle, "serving as selection criteria for choosing among myriads of deci-

FIGURE 9. Three categories of nonfunctional requirements showing examples of user concerns and the nonfunctional requirements that address them. (Figure adapted from Mylopoulos *et al.* [MCN92].)

Acquisition	User Concern	Nonfunctional Requirement
Performance— How well does it function?	How well does it utilize a resource? How secure is it? What confidence can be placed in what it does? How well will it perform under adverse conditions? How easy is it to use it?	Efficiency Integrity Reliability Survivability Usability
Design— How valid is the design?	How well does it conform to requirements? How easy is it to repair? How easy is it to verify its performance?	Correctness Maintainability Verifiability
Adaptation— How adaptable is it?	How easy is it to expand or upgrade its capability or performance? How easy is it to change? How easy is it to interfere with another system? How easy is it to transport? How easy is it to convert for use with another application?	Expandability Flexibility Interoperability Portability Reusability

sion" [MCN92]. An explicit consideration of nonfunctional requirements can assist in design decisions, but the method in which this happens can vary. Cuny *et al.* [CDHH96], for example, take a very informal approach in which three nonfunctional requirements (*i.e.,* extensibility, programmability, and interoperability) act as guiding principles in the subsequent design and implementation of the components in their environment. Bruegge *et al.* [BRRM95] pursue a more explicit method, though much more formal approaches, such as those from the area of requirements engineering, also exist.

The Role of Frameworks in Software Implementation

Many of the tools, systems, and approaches discussed in the previous chapters are built as, or claim to support, a *framework* to do one thing or another. Like the informal and inconsistent use of the word "architecture" when describing software structure ([SG96], p. 16), this term, too, is widely used and means very different things to different people. While the word is not always intended to be a precise, well-defined term (that is, it may be used in a vague, descriptive sense), authors often do offer informal, or at least implicit, definitions. Only by sampling the literature can the diverse contexts of use be revealed.

To begin, the term “framework” is often applied to collections of interoperable tools.[2] The software infrastructure that maintains open interfaces, provides tool access and control, and supports functionality common to the entire toolset is called a *framework*:

> Vendors of these tools increasingly support some <u>framework</u>[3] that defines the relationships among tools, whereby tools from different vendors can be intermingled by a user in a plug-and-play manner.... (Rover *et al.* [RMN96])

> We discuss the development of an integrated <u>framework</u> for heterogeneous network computing, in which a collection of interrelated components provide a coherent high-performance computing environment. (Dongarra *et al.* [DGMS93])

> [We] hope to provide a common <u>framework</u> with a graphical user interface from which [users] can access all the data management, analysis, visualization, and computational tools and services they need.... [The system should] serve as a <u>framework</u> tying together the underlying pieces and providing a coherent interface. (Bruegge *et al.* [BRRM95])

Perhaps the most common use of the term is in reference to an object-oriented style of programming in which the amount of work required of the programmer is reduced by the availability of one or more predefined class hierarchies. The programming environment created by such support is often termed a *framework*:

> A <u>framework</u> is a reusable object-oriented analysis and design for an application or subsystem. An application <u>framework</u> provides a template for an entire application.... A subsystem <u>framework</u> provides a set of services that are relatively independent of the rest of the system. (Coleman *et al.* [CABD94])

> A <u>framework</u> provides an integrated, layered system of objects. Each object in the <u>framework</u> is composed of or utilizes objects from lower layers. [It] defines an interface in which the users, who need not be familiar with object-oriented programming, express the fundamental scientific content and/or numerical methods of their problem.... (Reynders *et al.* [RHCA97])

> The common <u>framework</u> that enables a coherent solution to these problems is object-orientation. (Grimshaw and Wulf [GW96])

2. It is in this spirit that we have applied the term in our own research [BHMM94, HM97].
3. Underlines in the excerpts are ours.

Related to its use describing object-oriented class hierarchies, the term "framework" is more specifically applied to systems that support a template-based style of programming. Such a programming environment is also called a *framework*:

> Frames... provide support for the programming of distributed memory machines via a library of basic algorithms, data structures and so-called programming frames (or frameworks). The latter are skeletons with problem dependent parameters to be provided by the user. Frames focus on re-usability and portability as well as on small and easy-to-learn interfaces. (Romke and Silvestre [RS97])
>
> [A framework is] an object-oriented style of programming where a pre-existing environment provides a top-level object or objects within which all others are nested.... It is intended that the user... instantiate, modify, and "frame" together objects... to create a numerical application.... A frame-based system is designed to drive the user code. (Armstrong and Macfarlane [AM94])
>
> Basically, a framework can be seen as a semi-finished program that not only packages some reusable functionality, but also defines a generic software architecture in terms of a set of collaborating, extensible object classes. (Demeyer *et al.* [DMNS97])

The term is also applied to system support for generating or building software components, extensions, or complete environments (as opposed to end-user applications, as in the preceding examples):

> The objective of this work is to design a framework for building computer environments that provide all the computational facilities needed to solve a target class of problems.... (Weerawarana *et al.* [WHRC94])
>
> Amplifying this capacity for architecture reuse, we also provide a framework for developing software components that can be reused and reconfigured easily in a large domain of applications. (Hayes-Roth *et al.* [HPLM95])

Finally, systems which assist in the communication and design of software have also been labeled *frameworks*:

> [We] present... a representational framework that captures the structure of heterogeneous applications and the interactions between their components. (Anglano *et al.* [ASWB95])
>
> Design patterns are frameworks that software engineers can use to communicate the pros and cons of their design decisions to others. (Ramamoorthy and Tsai [RT96])

Clearly, the framework concept is broadly applied. From just the excerpts above, five types of frameworks are pertinent to domain-specific metacomputing for computational science:[4] *tool frameworks*, *object-oriented programming frameworks*, *program template frameworks*, *software construction frameworks*, and *design frameworks*. Why is the concept of a framework so ubiquitous?

Certainly, the excerpts suggest a variety of benefits that can be derived from frameworks, regardless of the particular type being used. These benefits include properties like abstraction, communication, customizability, extensibility, interoperability, productivity, retargetability, and reusability. Indeed, Demeyer *et al.* [DMNS97] point to the important role that frameworks play in constructing open systems that exhibit high degrees of interoperability, distribution, and extensibility. Not coincidentally, many of these characteristics correspond to the nonfunctional requirements of the previous section. To that end, we argue that frameworks *facilitate the implementation* of the nonfunctional requirements most important to the contributing technologies of domain-specific metacomputing for computational science.

First, we already know that a system's nonfunctional requirements and the designer's perception of the software are closely related. Second, each of the possible perspective views toward a domain-specific metacomputing environment—performance-, software-, and domain-centric—has a natural correspondence to one or more of the framework types described above.

For example, a performance-centric perspective would likely result in the use of a tool framework, allowing individual tools and components to be optimized for maximum performance while still participating in a larger computational environment. Examples of this perspective/framework pairing include the Globus metacomputing toolkit [FK96], AppLeS application-level schedulers [BW96], the p2d2 parallel debugger [CH94, Hood96], the DAQV distributed data access system [HM96, HM97], the PVM message-passing library [DGMS93], and the GEMS [BRRM95] and TIERRA [CDHH96] domain-specific environments. In addition, Rover *et al.* [RMN96] consider the differences between "integrated environments" and "toolkits" (both of which are framework-like approaches) as they pertain to tools for high performance computing.

A software-centric view, depending on the nonfunctional requirements motivating it, might utilize a software construction programming framework or a more formal design framework to support the desired software engineering paradigms. Relevant

4. With only a couple exceptions, all of the excerpts about frameworks in this section are taken from sources cited elsewhere in this discussion.

literature to these pairings include Manola's concern for interoperability in distributed object-systems [Mano95], Heiler's work on semantic interoperability [Heil95], a DSSA-based adaptive intelligent system by Hayes-Roth *et al.* [HPLM95], models for composing multiple (software) architectural styles [AB96], issues related to software composition [NM95], and the approach for constructing scalable software libraries put forth by Batory *et al.* [TBSS93].

Finally, a domain-centric system view might focus on the rapid creation of real application codes, making a program template or object-oriented programming framework most applicable. Systems meeting this description include parallel object-oriented plasma simulation by Norton *et al.* [NSD95], POET [AM94], POOMA [ABCH95, RHCA97], the programming frames approach by Romke and Silvestre [RS97], and the Concurrent Archetypes project [Chan94].

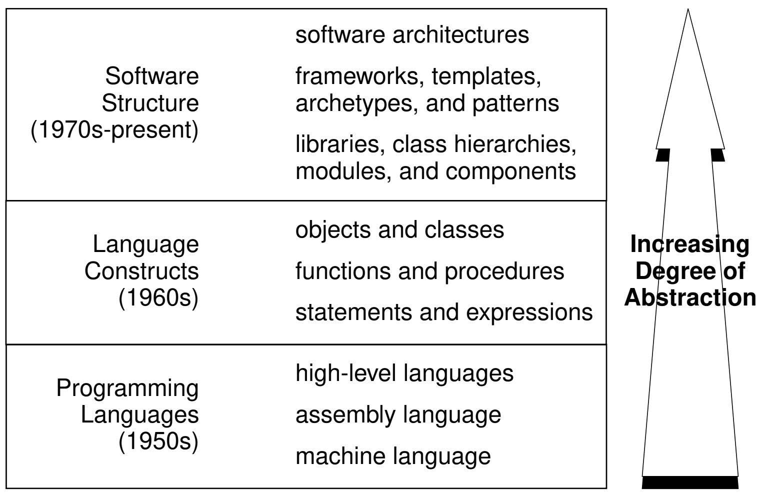
It is most definitely the case that other system views and framework types exist—if not for domain-specific metacomputing, then certainly for other problem areas. The views and types described here are merely derived from the component technologies and requirements of domain-specific metacomputing. In general, we simply propose that frameworks provide a means of implementing nonfunctional requirements and that the best type of framework for the task is a product of those requirements and the developer's perception of the system.

The Role of Abstraction in Software Use

An abstraction *reveals the salient features of an entity and simultaneously hides unnecessary, underlying detail*. Abstraction is a dominant theme in the evolution of computer system hardware, programming languages, and software development. Indeed, many of the systems and concepts discussed in earlier chapters exhibit various types and layers of abstraction.

For example, the Globus metacomputing project [FK96] proposes a "metacomputing abstract machine" to which applications and services may be targeted. This abstraction is layered on the Nexus communication library [FKT96, FGKT97] (among other components), which supports several lower level abstractions (*e.g.*, nodes, contexts, threads, links, etc.) for interprocess communication, data sharing, and remote function invocation. At a higher level, the PVM/HeNCE environment for heterogeneous computing [DGMS93, BDGM96] interacts with the user via a graph-based abstraction to parallel programming, where nodes represent procedures and arcs represent data dependencies and control flow. Object-orientation is often touted for its ability to support abstraction, as in the Legion metacomputing project [GW96], object-oriented parallel computing [NSD95], and the POOMA framework [ABCH95, RHCA97]. As illustrated by these examples, we are particularly interested in the use and purpose of abstraction in the development of software. Figure 10 illustrates the numerous and increasing levels of abstraction for programming and software development that have emerged over time (and continue to do so).[5]

FIGURE 10. The level of abstraction for programming and software development is continually increasing, yielding larger conceptual "building blocks" ([SG96]. p. 12) for constructing software systems.



The figure reveals an interesting trend. Many of the modern developments in programming languages and software engineering have been motivated by the desire to work at a higher conceptual level. Abstraction is both a driving force behind the development of, and a powerful tool for using, the new technologies that allow us to do this. With respect to programming languages, Sebesta defines abstraction as "the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored" ([Sebe93], p. 15). The same definition largely applies to software engineering as a whole. The purpose of abstraction is to preserve access to the most relevant features and make the details which enable them wholly irrelevant. We often distinguish between *interface* and *implementation*; abstraction is what creates this separation.

5. Figure 10 is based on ideas from Shaw and Garlan ([SG96], p. 4, 12-13) and Wiederhold *et al.* [WWC92]. The figure is a composite of their ideas but includes our own slight modifications and extensions.

Computer science is replete with examples of abstraction, which is often motivated by the insight that a repeated, complex operations can be represented symbolically. For example, assembly language first allowed the programmer to use symbolic names in place of machine operation codes and addresses ([SG96], p. 12). Eventually, certain useful patterns of execution emerged like evaluating arithmetic expressions, conditional statements, and loop constructs. These patterns were reflected in early high-level languages like Fortran ([SG96], p. 13). Each of these advances represents an increase in the level of abstraction.

The introduction of high-level languages and abstract data types, however, began to shift the target of abstraction from the language itself to language constructs. Procedures and functions serve a role that is analogous to assembly language commands and language statements for loops: they replace commonly used, unimportant, and/or complex details with a single symbol. Objects and classes refine the technique further, providing a more explicit distinction between interface and implementation.

Finally, as the need for abstraction exceeds what is possible for primitive language constructs to address, attention is shifting to a broader view of overall software structure. Libraries are, of course, a well-established, well-understood abstraction for software construction. Class hierarchies, too, have become a common software abstraction. Libraries and class hierarchies both represent yet larger conceptual building blocks for software developers. But many of the proposed abstractions remain topics of active research. The development of a module-based abstraction, which is to libraries and class hierarchies what objects and classes are to procedures and functions, continues to be refined today despite its origins in the 1970s and continued work throughout the 1980s ([SG96], pp. 13-14). Template-based and archetype abstractions are being applied with limited success (*e.g.,* Armstrong and Macfarlane [AM94], Chandy [Chan94], and Romke and Silvestre [RS97]) though their wide-spread acceptance and use remains at large. Finally, frameworks, while already being applied in a variety of ways (as described in the previous section), have yet to emerge as a well-defined abstraction for software development, though research results are beginning to appear (*e.g.,* Demeyer *et al.* [DMNS97]). Likewise, efforts are being made to promote abstractions within software architecture (*e.g.,* the analysis and use of software styles), but this, too, remains an area of active research.

As the level of abstraction increases, so does the target audience. For example, many of today's "end users" of high performance computing systems could fill in the stubs of a template to instantiate a parallel program (*e.g.,* as in the POET system

[AM94]), but the vast majority of them could certainly not create the assembly code, much less the machine code, to effect the same functionality.

Abstraction is by no means unique to programming and software development. Other areas of computer science also embrace abstraction. Graphical user interface creation tools allow interfaces to be constructed from a palette of predefined "widgets." (XForms by Zhao and Overmars [ZO97] is an example of such a system.) While this type of *programming* is similar to a component-based approach, the user's *interactions* with the system are at the user's level. That is, the environment allows the user to interact with important functionality (*e.g.,* scroll bars, dialog boxes, buttons, etc.) without having to explicitly deal with the low-level details (*e.g.,* windowing library calls, event loops, callbacks, etc.).

Indeed, abstraction can have a significant impact on the usability of software by end users. It is not surprising, then that the concept also thrives in commercially produced software. Scientific visualization environments, for example, support many levels of abstractions. Low level graphics primitives are accessible through higher level visualization abstractions like surfaces, contours, and glyphs. Visualization programs can be created by wiring together preprogrammed modules to create a data flow network that loads, formats, analyzing, and renders user data. Similarly, applications for creating multimedia presentations often employ user interface metaphors that reflect a high degree of abstraction. For example, the user might be able to compose a "cast" of "actors" and write a "script" to "direct" the presentation. Later, "set changes" and "special effects" could be also added. In theory, metaphors like this engage the user and give them a vehicle to understanding the software's functionality. In this way, abstraction functions as a lens through which users can see how software can be used to address their needs. Thus, abstraction ultimately impacts software usability.

Given this, we now consider the role that abstraction plays in domain-specific metacomputing for computational science. In fact, abstraction is applicable in all of the ways we have just described. Clearly, the construction of metacomputing environments already benefits from existing programming and software abstractions. We feel that emerging abstractions like software architecture and frameworks should also be brought to bear on this problem. But an equally important role for abstraction is to improve scientists' access and use of future metacomputing systems. We state domain-specificity as a requirement for this area, but "domain-specificity" is really just a type of abstraction. *Domain-specific abstractions* can be layered on top of metacomputing environments to convey the salient functions and characteristics of the environment (*e.g.,* performance capability) in terms of, and with respect to, the domain in which the scientist works.

Specificity Through Abstraction

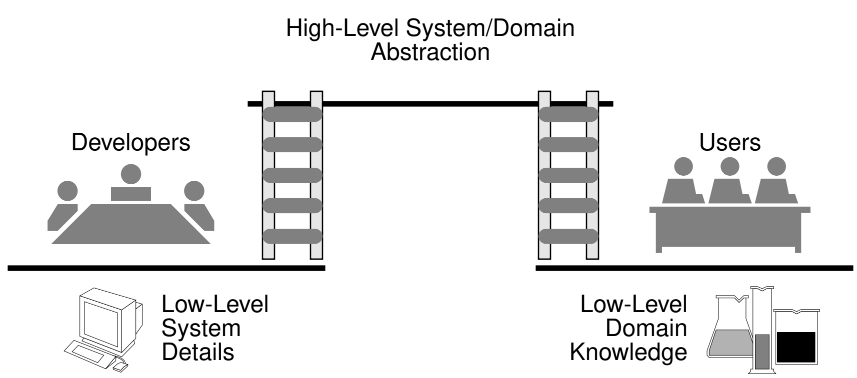
Thus, it is ultimately through a process of *abstraction* that metacomputing can be brought to bear on *specific* domains of computational science: *specificity through abstraction*.

The notion that software can be made more precise, more specific, and ultimately more usable through a process that seeks to replace lower-level details with higher-level concepts—that is, achieving specificity through abstraction—is somewhat counter-intuitive. The key to understanding this is a consideration of the perspectives from which software is viewed.

Developers typically take a bottom-up approach, seeing the underlying mechanisms and specific technologies which make up the software. That view may be performance-centric, software-centric, or even domain-centric, but as *developers* they possess intimate knowledge of the system and understand how to use it.

Users, on the other hand, look at software first for what it can do for them and how it fits into the problems they are trying to solve. Parallel and distributed computing has proven difficult enough for scientists to exploit; metacomputing technology represents an order of magnitude increase in the problem. The capabilities *must* be delivered in a domain-specific way that scientists can actually *use*. A process of abstraction builds up the "domain-specific lens" through which a user—a scientist—can determine how to apply a particular system to their problems. Or, to employ another metaphor, Figure 11 illustrates how abstraction creates a bridge between low-level system detail and low-level domain knowledge.

FIGURE 11. Software abstractions form a bridge between the low level system details that developers understand and the low level domain knowledge that scientists (users) possess.



The work on problem-solving environments, domain-specific environments, and domain-specific software architectures represents only the beginnings of a domain-specific approach to metacomputing for computational science. Just as this chapter attempts to synthesize the main concepts of this area, the technologies themselves must be synthesized to address, in a collective manner, the three broad requirements we propose: high performance heterogeneous computing, software design and development, and domain-specificity. In this section, we briefly speculate on how that implementational synthesis might occur and how candidate environments might be evaluated. The steps in the informal and speculative model we present below, like the sections above, correspond to the lifecycle stages of software design, software implementation, and software use.

Design: Identifying the Domain and Nonfunctional Requirements

Building any kind of domain-specific environment must begin with the identification of the domain to be addressed. Through a collaborative process, both functional and nonfunctional requirements must be identified. This collaboration can be a formal one like that used to build domain-specific software architectures or informal like the process used in the TIERRA domain-specific environment. The nonfunctional requirements and the developer's perspective of the environment, as suggested above, should play a role in determining the best approach. Obviously, existing techniques for software specification, modeling, and requirements engineering could also be applied.[6]

It is important to make nonfunctional requirements explicit at this stage because they (1) often affect multiple functional requirements, (2) play a significant role in implementation strategies, (3) may change over the development (and subsequent use) of the environment, and (4) are usually only evaluated subjectively. Even the best understanding of nonfunctional requirements at the beginning of the development process is not a guarantee that the environment will be a success. Like most software design and development, an iterative process is ultimately required.

Most software engineering approaches to addressing nonfunctional requirements attempt to develop formal definitions and then measure the degree to which a finished software product meets the requirement. Mylopoulos *et al.* [MCN92] propose a technique that is much more amenable to domain-specific metacomputing. In their "process-oriented" approach, nonfunctional requirements are viewed as crite-

6. We do not discuss those techniques here since they are not *unique* aspects of domain-specific metacomputing for computational science.

ria for making design decisions. Thus, rather than evaluating a finished product, they attempt to “rationalize the development process in terms of nonfunctional requirements” [MCN92]. They point out that each design decision affects each nonfunctional requirement in either a positive or negative way. An analysis of the “positive and negative dependencies” can be used to argue that a system meets a given nonfunctional requirement or to explain why it does not. In a broad sense, their approach formalizes that taken by Cuny *et al.* in the development of the TIERRA domain-specific environment.

Implementation: Using Frameworks or Software Architectures

We believe that a frameworks-based approach holds particular promise for implementing systems in this area. Frameworks, as described above, address a range of nonfunctional requirements important to domain-specific metacomputing, and they can also facilitate the creation of appropriate abstractions (see next section). If existing technology is to be used, a framework may improve the ability to integrate it into the environment. If a more formal approach is desired, methodologies from software architecture and domain-specific software architecture should be employed.

The exploratory nature of many computational science problems demands a flexible implementation that can evolve with the problem-solving requirements. The properties of domain-specific environments—extensibility, programmability, and interoperability—should all be targeted as critical nonfunctional requirements for domain-specific metacomputing environments.

However, the challenge in implementing frameworks is achieving an appropriate balance between “*flexibility*—packaging software components that can be reused in as many contexts as possible—and *tailorability*—designing software architectures that are easy to adapt to targeted requirements” [DMNS97]. In this regard, flexibility and tailorability are inherently at odds with each other. A greater degree of flexibility results in more opportunities for adapting the framework to particular needs; however, having additional opportunities for adaptation requires the developer to possess a better understanding of the framework in order to adapt it; and a framework that requires more comprehensive knowledge ends up being less tailorable. We noted a similar problem in the Legion metacomputing project [GW96] where attempts to maximize flexibility impinge on the extent to which developers can actually implement the system. This, in turn, makes it very difficult for others just to grasp *how* the system may be tailored. Demeyer *et al.* [DMNS97] conclude, “although flexibility is, in general, a good thing, too much flexibility can result in decreased tailorability”—yet another trade-off among nonfunctional requirements.

The situation is particularly critical for domain-specific metacomputing since the optimal degree of flexibility almost certainly can never be determined *ab initio*.

Use: Creating Appropriate Abstractions

Ultimately, the adaptation of a framework must involve a consideration of how the end user interacts with the system, including the identification of appropriate metaphors and abstractions. The most likely place for these to be used is in a graphical interface. However, abstractions can also be applied to framework components that are open to replacement, extension, or customization by the user. Again, DSEs provide examples of how, through the use of frameworks, environments can be tailored to meet user needs. Similarly, DSSAs can incorporate domain abstractions into the design of the software itself, perhaps facilitating the extension of that abstraction to how the user interacts with the system.

Another important issue is how the underlying computational resources are (or are not) presented to the user. The literature supports a spectrum of possibilities. Raw message-passing libraries like PVM [DGMS93] make few attempts at abstraction in this regard. But the extension to PVM, HeNCE [BDGM96], does support the configuration and cost estimation of target virtual machines. Other systems, like Legion [GW96] and Globus [FK96], envision supporting a range of resource-level access. At one extreme, users simply submit an application and rely on the system to figure out when and where to run it. At the other extreme, a user could exert full decision-making control over the resources used. The problem, though, is that these approaches are not really abstractions: they fail to communicate in meaningful terms the important features of the system and at the same time conceal the unnecessary details. So, what abstractions are appropriate to accomplish this for metacomputing?

At best, this is a difficult question to answer. First, metacomputing systems are primarily intended to be generic systems that enable a wide range of computing activities. Discovering and defining widely applicable abstractions for these generic systems may not even be tractable, nor is it consistent with the goal of *domain-specific* metacomputing. Second, since current metacomputing systems primarily target application developers and system software programmers, an abstraction relevant to their needs (if one exists at all) is not necessarily an abstraction that will help the end user better understand or interact with the system.

This remains a challenging issue. To begin to understand its implications, one must consider a simple question: *what is meaningful and important to the scientist?* With respect to computational resources, one can imagine that a scientist might have questions like the following:

How long will the application take to run?

How accurate will the results be?

What could be done to make the code run faster (or be more accurate)?

Note what the questions are not:

Does the compiler on node X automatically parallelize the code?

Do nodes X and Y use compatible data representations?

What is the communication latency between node X and node Y

Theoretically, this suggests that the abstraction best suited to scientists could be extremely simple. As we stated early on, “nobody wants parallelism” [Panc91]; what scientists and engineers do want is performance. If performance is, in fact, the single most meaningful and important concept to what they are trying to accomplish, then the abstraction through which they access and interact with computational resources could be trivial. However, the underlying details and support for that abstraction tell a different story. The requirements to create such a simple abstraction far exceed current capabilities to automate processes like program analysis and decomposition, resource selection, system configuration, scheduling, monitoring, and optimization. Thus, creation of the “best” abstractions for domain-specific metacomputing may not be possible for some time. Meanwhile, the area has the potential to “push the envelope” of current technology while trying to achieve that goal.

Evaluation

The nature of domain-specific metacomputing environments for computational science makes them difficult to evaluate in the same way other software systems are. The process of software testing and evaluation has long been split into two categories ([Somm89], p. 406):

Validation: *Are we building the right product?*

Verification: *Are we building the product right?*

Validation attempts to make sure that the implementation of the software does what the user wants it to do, and *verification* ensures that the software meets the specifications and requirements set out for the system. While similar, the concepts have at least one distinct difference: verification can be carried out during implementation, but validation can only be applied afterwards.

This creates an interesting situation for domain-specific metacomputing environments. Verification, in its purest form, is complicated by the fact that complete specifications for such environments are rarely feasible. Thus, how can developers know if they are building the product correctly if they do not have a specification for what "correct" is?

Part of the reason complete specifications are so difficult is that these environments are intended to evolve with the scientist's process of exploration and discovery. In other words, domain-specific environments effectively assume that validation—prior to actual use of the system by the end user—is impossible. That is, how can developers build the correct product if the whole point of the product is to change over the course of its use?

What this suggests is that some form of "meta-level" evaluation must take place. If the point of a system is to change and evolve in light of emergent requirements and problem-solving needs, then what software features can be examined that might yield insight in this regard? In other words, can some higher-order aspect of the software be evaluated to determine if it does (or will) address this need? We already have a term for these characteristics: nonfunctional requirements.

That domain-specific metacomputing environments for computational science must place so much emphasis on nonfunctional requirements clearly complicates the evaluation process. Methods for guaranteeing the fulfillment of nonfunctional requirements are not yet widely used (or known) ([Somm89], p. 101-103). This relegates the evaluation of domain-specific metacomputing environments to be a largely subjective process. Rather than asking, "Is the environment portable?" Evaluators must ask, "Is the environment portable *enough*?" And they must continue to ask such questions throughout the life of the environment because the scientist's need for portability might change. The answers to such questions are subjective, dependent on the particular user's needs at a given time.

It is largely for this reason that collaboration is such a critical component of the domain-specific metacomputing process. That is, formal methods for designing, implementing, and evaluating domain-specific metacomputing environments for computational science do not yet exist. Collaborative approaches play an important role in the development of these techniques. In this way, the area can be viewed as a driver of advancements in several key areas of computer science, including high performance computing, software engineering, and especially computational science.

Open Problems

Throughout the course of this and previous chapters, several open problems for the area of domain-specific metacomputing for computational science have been revealed. This section consolidates and summarizes these issues by posing them as questions and, where possible, briefly speculating on answers and possible research directions. Consistent with other sections in this chapter, we organize our discussion according to the software lifecycle stages of design, implementation, and use.

Design

> *How are the important features of computational science domains extracted?*
>
> *How are nonfunctional requirements determined?*
>
> *How are domain-knowledge and nonfunctional requirements integrated into software implementation and software use?*

Informal collaborations have had promising results, though more well-defined and precise methods exist in the areas of domain modeling [Bato94, Migh95] and domain-specific software architecture [TTC95, Trac94]. Ultimately, the scientist understands the domain, and the software engineer understands the software. Through a collaborative process, developers must educate users about the trade-offs associated with various nonfunctional requirements. And scientists must educate developers about what is needed out of the environment. A large body of research in the software engineering and user interface communities addresses collaborative computing [ACM91] and participatory design [ACM93]. Adaptations and/or extensions to these ideas could be applied to these types of problems. The "process-oriented" approach to using nonfunctional requirements by Mylopoulos *et al.* [MCN92] offers a promising direction, but must be made more accessible to the average software developer.

> *Should formal software engineering methods for specification, requirements engineering, and software testing be applied to this area? If so, how?*

A single answer to this question is not in the spirit of domain-specific metacomputing. The question should be posed on a case-by-case basis with any decision subject to at least the nonfunctional requirements of the environment. The application of more formal methods must ultimately be balanced with expected benefits of doing so. Domain-specific metacomputing is unique in many ways. It is unclear whether existing formal methods are well-suited to it.

Implementation

How is a software architecture or framework type chosen?

How are frameworks built?

As Shaw and Garlan point out, picking the best architectural style for a given problem (or domain) is an open problem ([SG96], p. 17). Similarly, general methodologies for choosing among the various types of frameworks we identify are largely missing [DMNS97]. And while researchers in the area of software architecture are developing formalisms and tools to assist in applying the technology, the implementation of frameworks remains more art than science. Unfortunately, frameworks appear to have more immediate and practical application to domain-specific metacomputing. A general treatment of frameworks-based computing (as opposed to treatments of specific types of frameworks) is necessary.

How are properties like extensibility, programmability, and interoperability actually "implemented"?

Essentially, the question exposes the problem of how nonfunctional requirements can be satisfied in an implementation. The work in domain-specific environments [CDHH96] offers examples of how specific nonfunctional requirements can be satisfied in a particular environment. But this does not offer any general understanding of the problem. A traditional software engineering approach would define "metrics" for measuring such properties ([Somm89], pp. 101, 292), but this is notoriously difficult for nonfunctional requirements. A more formal treatment of how to identify and measure nonfunctional requirements would go a long way toward determining how best to implement them. Mylopoulos *et al.* [MCN92] offer a promising approach to the problem, but it is not a broadly applicable solution.

How is access to computational resources to be improved and automated so that both performance and ease of use are maximized?

This is the proverbial “Holy Grail” of parallel and distributed computing. The performance capability is clearly there; it’s just incredibly difficult to deliver it in a nice package. A vast amount of work is required to address this open problem. While some projects (*e.g.,* Legion [GW96]) envision easy-to-use, desktop access to a worldwide metacomputer, that vision is simply not possible until some major breakthroughs in high performance computing are made. Not only must problems of performance and usability be addressed, but issues of administration, security, and other socio-technical problems require similar concerted efforts. Relatively little work is being pursued in these areas.

Use

> *How are appropriate domain-specific abstractions for metacomputing systems identified and built?*

> *How can high-level abstractions be built so that they simultaneously support the process of scientific exploration?*

As suggested in the section on open problems in design, collaboration is central to identifying appropriate abstractions for computational science domains. But these questions get at how the underlying heterogeneous computing environments of domain-specific metacomputers is actually used by the scientist. We suggeste that a very simple, performance-oriented abstraction might be appropriate where the metacomputing environment is a black box that always delivers maximum performance; delivering that abstraction, though, is currently not possible. A degree of interaction in the form of application description, resource selection, and performance goals is the least that is required. Efforts to build such metacomputer abstractions for specific domains of computational science is an important research goal.

A potential problem with high-level abstractions, though, is that they can lock the user into a particular way of thinking. This may not be conducive to the processes of exploration and experimentation required for scientific advancement. Achieving the optimal balance between abstraction and flexibility, with respect to how the user interacts with a system, pushes software engineering and the study of user interfaces to the limit.

> *How do you evaluate instances of domain-specific metacomputing environments?*

The simple answer to this question is to build the environment; let (or make) the scientists use it; and then decide if it is successful or not. Of course, this is not always a feasible or economical approach—it certainly is not a formal one—to evaluation. But currently, it is representative of the state-of-the-art in evaluating these types of systems. Furthermore, it is largely consistent with the ways of experimental computer science. For this type of work, theory and formalism alone do not—and may never—suffice. The insight, experience, and knowledge that is derived from experimentation is extremely beneficial. Efforts to combine theory and practice may result in improved methods, but attempting to replace the process of experimentation in which computer scientists engage would not be productive.

Conclusion

The emerging area of domain-specific metacomputing for computational science stands to revolutionize the way in which scientists conduct their computer-based research. A promising collection of technologies already exists and need only be organized and applied in a collective manner to begin this revolution. Numerous pitfalls and problems must be confronted, but this is, to a large extent, reflective of the methods of experimental computer science. Experimental computer scientists must continually confront the development of larger and larger software systems—and on an experimental basis. That is, the scale of experimentation is growing. And with that growth, researchers are confronted with new challenges of software design, implementation, use, and evaluation. Like the very domain scientists for which they construct systems, experimental computer scientists are ultimately engaged in their own exploratory process of experimentation and discovery. It is only through this process that revolutions like domain-specific metacomputing for computational science can take place.

References

[AB96] A. Abd-Allah and B. Boehm, *Models for Composing Heterogeneous Software Architectures*, University of Southern California Computer Science Department Technical Report USC-CSE-96-505, 1996.

[ASWB95] C. Anglano, J. Schopf, R. Wolski, and F. Berman, *Zoom: A Hierarchical Representation for Heterogeneous Applications*, University of California San Diego, Department of Computer Science and Engineering Technical Report CS95-451, January 1995.

[AM94] R. Armstrong and J. Macfarlane, *The Use of Frameworks for Scientific Computation in a Parallel Distributed Environment*, Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing, San Francisco, CA, August 1994, pp. 15-25.

[ACM91] Association for Computing Machinery (ACM), *Communications of the ACM*, issue on “Collaborative Computing,” ACM Press, New York, NY, Vol. 34, No. 12, December 1991.

[ACM93] Association for Computing Machinery (ACM), *Communications of the ACM*, issue on “Participatory Design,” ACM Press, New York, NY, Vol. 36, No. 4, June 1993.

[ABCH95] S. Atlas, S. Banerjee, J. Cummings, P. Hinker, M. Srikant, J. Reynders, and M. Tholburn, *POOMA: A High Performance Distributed Simulation Environment for Scientific Applications*, Proceedings of Supercomputing '95, San Diego, CA, December 1995.

[BBB96] J. Baldeschweiler, R. Blumofe, and E. Brewer, *ATLAS: An Infrastructure for Global Computing*, Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, Connemara, Ireland, September 1996.

[Bato94] D. Batory, *Products of Domain Models*, Proceedings of the ARPA Domain Modeling Workshop, George Mason University, September 1994.

[BDGM96] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam, *Tools for Heterogeneous Network Computing*, Proceedings of the SIAM Conference on Parallel Computing, 1993.

[BM95] F. Berman and R. Moore, *Heterogeneous Computing Environments*, Working Group 9 Report from the Proceedings of the 2nd Pasadena Workshop on System Software and Tools for High Performance Computing Environments, T. Sterling, P. Messina, and J. Pool, eds., available at <http://cesdis.gsfc.nasa.gov/PAS2/>, January 1995.

[BW96] F. Berman and R. Wolski, *Scheduling from the Perspective of the Application*, Proceedings of the High Performance Distributed Computing Conference, Syracuse, NY, August 1996.

[BWFS96] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, *Application-Level Scheduling on Distributed Heterogeneous Networks (Technical Paper)*, Proceedings of Supercomputing '96, Pittsburgh, PA, November 1996.

[Bern94] T. Berners-Lee, *RFC 1630: Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*, June 1994. See also <http://ds.internic.net/rfc/rfc1630.txt> and <http://www.w3.org/Addressing/>.

[BMM94] T. Berners-Lee, L. Masinster, and M. McCahill, eds., *RFC 1738: Uniform Resource Locators (URL)*, December 1994. See also <http://www.w3.org/Addressing/> and <http://ds.internic.net/rfc/rfc1738.txt>.

[BP94] R. Blumofe and D. Park, *Scheduling Large-Scale Parallel Computations on Networks of Workstations*, Proceedings of the 3rd International Symposium on High Performance Distributed Computing, San Francisco, CA, August 1994.

[BPWB93] K. Brodlie, A. Poon, H. Wright, L. Brankin, G. Banecki, and A. Gay, *GRASPARC - A Problem Solving Environment Integrating Computation and Visualization*, Proceedings of IEEE Visualization '93, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 102-109.

[BHMM94] D. Brown, S. Hackstadt, A. Malony, and B. Mohr, *Program Analysis Environments for Parallel Language Systems: The TAU Environment*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Townsend, TN, May 1994, pp. 162-171.

[BRRM95] B. Bruegge, E. Riedel, A. Russell, and G. McRae, *Developing GEMS: An Environmental Modeling System*, IEEE Computational Science and Engineering, Vol. 2, No. 3, Fall 1995, pp. 55-68.

[CD96] H. Casanova and J. Dongarra, *Netsolve: A Network Server for Solving Computational Science Problems*, Proceedings of Supercomputing '96, Pittsburgh, PA, November 1996.

[Chan94] K. Chandy, *Concurrent Program Archetypes*, Proceedings of the 8th International Parallel Processing Symposium, Cancun, Mexico, April 1994.

[CH94] D. Cheng and R. Hood, *A Portable Debugger for Parallel and Distributed Programs*, Proceedings of Supercomputing '94, Washington, D.C., November 1994, pp. 723-732.

[CM89] D. Cheriton and T. Mann, *Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance*, ACM Transactions on Computer Systems, Vol. 7, No. 22, May 1989, pp. 147-183.

[CABD94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 2nd Edition, Addison-Wesley, Inc., New York, NY, 1994.

[Cox90] B. Cox, *Planning The Software Industrial Revolution*, IEEE Software, Vol. 7, No. 6, November 1990, pp. 25-33.

[CDHH96] J. Cuny, R. Dunn, S. Hackstadt, C. Harrop, H. Hersey, A. Malony, and D. Toomey, *Building Domain-Specific Environments for Computational Science: A Case Study in Seismic Tomography*, International Journal of Supercomputing Applications and High Performance Computing, Vol. 11, No. 3, Fall 1997.

[DMNS97] S. Demeyer, T. Meijler, O. Nierstrasz, and P. Steyaert, *Design Guidelines for Tailorable Frameworks*, submitted to Communications of the ACM, ACM Press, New York, NY, October 1997 (issue on "Object-Oriented Frameworks").

[DGMS93] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, *Integrated PVM Framework Supports Heterogeneous Network Computing*, Computer in Physics, Vol. 7, No. 2, April 1993, pp. 166-175.

[DW95] J. Dongarra and D. Walker, *Software Libraries for Linear Algebra Computations on High Performance Computers*, SIAM Review, Vol. 37, No. 2, June 1995. See also <http://www.netlib.org/>.

[DJRH97] T. Drashansky, A. Joshi, J. Rice, E. Houstis, and S. Weerawarana, *A MultiAgent Environment for MPSEs*, (CD-ROM) Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[ETM95] I. Ekmecic, I. Tartalja, and V. Milutinovic, *EM^3: A Taxonomy of Heterogeneous Computing Systems*, IEEE Computer, Vol. 28, No. 12, December 1995, pp. 68-70.

[Esha96] M. Eshagian, ed., *Heterogeneous Computing*, Artech House, Inc., Norwood, MA, 1996.

[FT95] M. Fayad and W. Tsai, *Object-Oriented Experiences*, Communications of the ACM, Vol. 38, No. 10, October 1995, pp. 50-53.

[FGKT97] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke, *Managing Multiple Communication Methods in High-Performance Networked Computing Systems*, Journal of Parallel and Distributed Computing, Vol. 40, No. 1, January 1997, pp. 35-48.

[FGNS96] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke, *Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment*, Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing, Syracuse, New York, August 1996.

[FK96] I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Lyon, France, August 1996.

[FKT96] I. Foster, C. Kesselman, and S. Tuecke, *The Nexus Approach to Integrating Multithreading and Communication*, Journal of Parallel and Distributed Computing, Vol. 37, No. 1, August 1996, pp. 70-82.

[FT96] I. Foster and S. Tuecke, *Enabling Technologies for Web-Based Ubiquitous Supercomputing*, Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing, Syracuse, New York, August 1996.

[GHR94] E. Gallopoulos, E. Houstis, and J. Rice, *Computer as Thinker/Doer: Problem-Solving Environments for Computational Science*, IEEE Computational Science and Engineering, Vol. 1, No. 2, Summer 1994, pp. 11-23.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom, *Architectural Mismatch, or Why It's Hard to Build Systems Out of Existing Parts*, Proceedings of the 17th International Conference on Software Engineering (ICSE-17), Seattle, WA, April 1995.

[Grim96] A. Grimshaw, *LEGION: Supporting Diversity and Performance in Wide-Area Metasystems*, presentation at the Workshop on Software Tools for High Performance Computing Systems, Chatham, MA, October, 1996.

[GNW95] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf, *Campus-Wide Computing: Results Using Legion at the University of Virginia*, University of Virginia Computer Science Department Technical Report CS-95-19, March 1995.

[GW96] A. Grimshaw and W. Wulf, *Legion - A View from 50,000 Feet*, Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing, Syracuse, New York, August 1996.

[HM96] S. Hackstadt and A. Malony, *Distributed Array Query and Visualization for High Performance Fortran*, Proceedings of Euro-Par '96, Lyon, France, August 1996, pp. 55-63.

[HM97] S. Hackstadt and A. Malony, *DAQV: Distributed Array Query and Visualization Framework*, Journal of Theoretical Computer Science, special issue on Parallel Computing (to appear).

[Harr97] C. Harrop, personal communication, Department of Computer and Information Science, University of Oregon, June 1997.

[Hart92] J. Hart, *Introduction: Visualization in Networked Environments*, Communications of the ACM, Vol. 35, No. 6, June 1992, p. 43.

[HPLM95] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic, *A Domain-Specific Software Architecture for Adaptive Intelligent Systems*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 288-301.

[Heil95] S. Heiler, *Semantic Interoperability*, ACM Computing Surveys, Vol. 27, No. 2, June 1995, pp. 271-273.

[HHHM96] H. Hersey, S. Hackstadt, L. Hansen, and A. Malony, *Viz: A Visualization Programming System*, University of Oregon, Department of Computer and Information Science, Technical Report CIS-TR-96-05, April 1996.

[Hind83] A. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers*, Scientific Computing, R. Stepleman *et al.* (eds.), North-Holland, Amsterdam, 1983 (Vol. 1 of IMACS Transactions on Scientific Computation), pp. 55-64.

[Hood96] R. Hood, *The p2d2 Project: Building a Portable Distributed Debugger*, Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96), Philadelphia, PA, May 1996.

[HRJW95] E. Houstis, J. Rice, A. Joshi, S. Weerawarana, E. Sacks, V. Rego, N. Wang, C. Takoudis, A. Sameh, and E. Gallopoulos, *MPSE: Multidisciplinary Problem Solving Environments*, Purdue University, Department of Computer Sciences, Technical Report CSD-TR-95-047, 1995.

[HCYH94] C. Hui, G. Chan, M. Yuen, M. Hamdi, and I. Ahmad, *Solving Partial Differential Equations on a Network of Workstations*, Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing, San Francisco, CA, August 1994, pp. 194-201.

[John96] N. Johnson, *The Legacy and Future of CFD at Los Alamos*, Proceedings of 1996 Canadian CFD Conference, Ottawa, Canada, June 1996. See also <http://gnarly.lanl.gov/>.

[JDRW96] A. Joshi, T. Drashansky, J. Rice, S. Weerawarana, and E. Houstis, *Multi-Agent Simulation of Complex Heterogeneous Models in Scientific Computing*, Purdue University, Department of Computer Science, Technical Report 96-025, 1996.

[KPSW93] A. Khokhar, V. Prasanna, M. Shaaban, and C. Wang, *Heterogeneous Computing: Challenges and Opportunities*, IEEE Computer, Vol. 26, No. 6, June 1993, pp. 18-27.

[LLNL97] Lawrence Livermore National Laboratory, *Accelerated Strategic Computing Initiative*, brochures from Supercomputing '96, available at <http://www.llnl.gov/asci/overview/>, May 1997.

[LG96] M. Lewis and A. Grimshaw, *Using Dynamic Coonfigurability to Support Object-Oriented Programming Languages and Systems in Legion*, University of Virginia Computer Science Department Technical Report CS-96-19, December 1996.

[Mano95] F. Manola, *Interoperability Issues in Large-Scale Distributed Object Systems*, ACM Computing Surveys, Vol. 27, No. 2, June 1995, pp. 268-270.

[MCWH95] S. Markus, A. Catlin, S. Weerawarana, and E. Houstis, *On the Software Engineering of Multi-Platform Parallel/Distributed Software*, Proceedings of the First International Conference on Neural, Parallel, and Scientific Computation, 1995.

[MFS94] C. Mechoso, J. Farrara, and J. Spahr, *Running a Climate Model in a Heterogeneous, Distributed Computer Environment*, Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing, San Francisco, CA, August 1994, pp. 79-84.

[Migh95] R. Might, *Domain Models: What are They? How are They Used?*, George Mason University, unpublished. See also <http://www.owego.com/dssa/>.

[MABB94] A. Mirin, J. Ambrosiano, J. Bolstad, A. Bourgeois, J. Brown, B. Chan, W. Dannevik, P. Duffy, P. Eltgroth, C. Matarazzo, and M. Wehner, *Climate System Modeling Using a Domain and Task Decomposition Message-Passing Approach*, Computer Physics Communications, Vol. 84, 1994, pp. 278-296.

[MN96] S. Moser and O. Nierstrasz, *The Effect of Object-Oriented Frameworks on Developer Productivity*, IEEE Computer, Vol. 29, No. 9, September 1996, pp. 45-51.

[MCN92] J. Mylopoulos, L. Chung, and B. Nixon, *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach*, IEEE Transactions on Software Engineering, Vol. 18, No. 6, June 1992, pp. 483-497.

[NCO96] National Coordination Office for High Performance Computing and Communications, *High Performance Computing and Communications: Foundation for America's Information Future (FY 1996 Blue Book)*, S. Howe, ed., National Coordination Office, 1996. See also <http://www.hpcc.gov/>.

[NSF93] National Science Foundation, *From Desktop to Teraflop: Exploiting The U.S. Lead In High Performance Computing*, Report NSB 93-205, NSF Blue Ribbon Panel on High Performance Computing, chaired by Prof. Lewis Branscomb, August 1993.

[NWM93] J. Nicol, C. Wilkes, and F. Manola, *Object Orientation in Heterogeneous Distributed Computing Systems*, IEEE Computer, Vol. 26, No. 6, June 1993, pp. 57-67.

[NM95] O. Nierstrasz and T. Meijler, *Research Directions in Software Composition*, ACM Computing Surveys, Vol. 27, No. 2, June 1995, pp. 262-264.

[NSD95] C. Norton, B. Szymanski, and V. Decyk, *Object-Oriented Parallel Computation For Plasma Simulation*, Communications of the ACM, Vol. 38, No. 10, October 1995, pp. 88-100.

[NHSS87] D. Notkin, N. Hutchinson, J. Sanislo, and M. Schwartz, *Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity*, Communications of the ACM, Vol. 30, No. 2, February 1987, pp. 132-140.

[OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, Document ptc/96-03-04, July 1995. See also <http://www.omg.org/>.

[OG96] Open Group, *Distributed Computing Environment Overview*, Document TOG-DCE-PD-1296, 1996. See also <http://www.opengroup.org/>.

[Panc91] C. Pancake, *Software Support for Parallel Computing: Where Are We Headed?*, Communications of the ACM, Vol. 34, No. 11, November 1991, pp. 52-64.

[Panc94] C. Pancake, *A Collaborative Effort In Parallel Tool Design*, Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing, Townsend, TN, May 1994, pp. 112-118.

[Panc95] C. Pancake, *The Promise And The Cost of Object Technology: A Five Year Forecast*, Communications of the ACM, Vol. 38, No. 10, October 1995, pp. 32-49.

[HP90] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[Purt94] J. Purtilo, *The Polylith Software Bus*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 16, No. 1, January 1994, pp. 151-174.

[PSW91] J. Purtilo, R. Snodgrass, and A. Wolf, *Software Bus Organization: Reference Model and Comparison of Two Existing Systems*, DARPA Module Interconnection Formalism Working Group, Technical Report 8, University of Arizona Computer Science Department, November 1991.

[RT96] C. Ramamoorthy and W. Tsai, *Advances in Software Engineering*, IEEE Computer, Vol. 29, No. 10, October 1996, pp. 47-58.

[RHCA97] J. Reynders *et al.*, *POOMA: A Framework for Scientific Simulations on Parallel Architectures*, available from <http://www.acl.lanl.gov/PoomaFramework/>, January 1997.

[Rice95] J. Rice, *Computational Science and the Future of Computing Research*, IEEE Computational Science and Engineering, Vol. 2, No. 4, Winter 1995, pp. 35-41.

[Rice96] J. Rice, *Scalable Scientific Software Libraries and Problem Solving Environments*, Purdue University, Department of Computer Sciences, Technical Report CSD-TR-96-001, January 1996.

[RS97] T. Romke and J. Silvestre, *Programming Frames for the Efficient Use of Parallel Systems*, University of Paderborn, Paderborn Center for Parallel Computing, Technical Report TR-001-97, 1997.

[RMN96] D. Rover, A. Malony, and G. Nutt, *Summary of Working Group on Integrated Environments Vs. Toolkits*, Debugging and Performance Tuning for Parallel Computing Systems, M. Simmons, A. Hayes, J. Brown, and D. Reed, eds., IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 371-389.

[Sebe93] R. Sebesta, *Concepts of Programming Languages*, 2nd Edition, Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.

[SDA96] H. Seigel, H. Dietz, and J. Antonio, *Software Support for Heterogeneous Computing*, ACM Computing Surveys, Vol. 28, No. 1, March 1996, pp. 237-239.

[SDKR95] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 314-335.

[SG96] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Pretice-Hall, Inc., Upper Saddle River, NJ, 1996 (242 pages).

[SKS92] N. Shivaratri, P. Krueger, and M. Singhal, *Load Distributing for Locally Distributed Systems*, IEEE Computer, Vol. 25, No. 12, December 1992, pp. 33-44.

[SC92] L. Smarr and C. Catlett, *Metacomputing*, Communications of the ACM, Vol. 35, No. 6, June 1992, pp. 44-52.

[SM94] K. Sollins and L. Masinster, *RFC 1737: Functional Requirements for Uniform Resource Names*, December 1994. See also <http://ds.internic.net/rfc/rfc1737.txt>.

[Somm89] I. Sommerville, *Software Engineering*, 3rd Edition, Addison-Wesley, Inc., New Yory, NY, 1989.

[SBM92] R. Springmeyer, M. Blattner, and N. Max, *A Characterization of the Scientific Data Analysis Process*, Proceedings of IEEE Visualization '92, Boston, MA, October 1992, pp. 235-242.

[Sund96] V. Sunderam, *Heterogeneous Network Computing: The Next Generation*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Lyon, France, August 1996.

[Swar82] P. Swarztrauber, *Vectorizing the FFTs*, Parallel Computations, G. Rodrigue, ed., Academic Press, 1982, pp. 51-83.

[TTC95] R. Taylor, W. Tracz, and L. Coglianese, *Software Development Using Domain-Specific Software Architectures*, ACM SIGSOFT Software Engineering Notes, Vol. 20, No. 5, December 1995, pp. 27-37.

[TBSS93] J. Thomas, D. Batory, V. Singhal, and M. Sirkin, *A Scalable Approach To Software Libraries*, Proceedings of the Sixth Annual Workshop on Software Reuse, Owego, New York, November 1993.

[Trac94] W. Tracz, ed., *Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)*, ACM Software Engineering Notes, Vol. 19, No. 2, April 1994, pp. 52-56. See also <http://www.owego.com/dssa/faq/faq.html>.

[DOE96] United States Department of Energy Defense Programs, *Accelerated Strategic Computing Initiative Program Plan (Draft)*, September 1996. See also <http://www.llnl.gov/asci-alliances/>.

[WB95] T. Wagner and R. D. Bergeron, *Data-Parallel Program Visualization*, Proceedings of IEEE Visualization 95, November 1995, pp. 224-231.

[WHRC94] S. Weerawarana, E. Houstis, J. Rice, A. Catlin, C. Crabill, C. Chui, and S. Markus, *PDELab: An Object-Oriented Framework for Building Problem Solving Environments for PDE Based Applications*, Proceedings of the 2nd Annual Object-Oriented Numerics Conference, 1994. Also available as Purdue University Department of Computer Science Technical Report CSD-TR-94-021, 1994.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri, *Toward Megaprogramming*, Communications of the ACM, Vol. 35, No. 11, November 1992, pp. 89-99.

[Wols96] R. Wolski, *Dynamically Forecasting Network Performance Using the Network Weather Service*, University of California San Diego, Computer Science and Engineering Department, Technical Report TR-CS96-494, November 1996.

[WSP97] R. Wolski, N. Spring, and C. Peterson, *Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service*, University of California San Diego, Department of Computer Science, Technical Report TR-CS97-540, May 1997.

[Wood96] P. Woodward, *Perspectives on Supercomputing: Three Decades of Change*, IEEE Computer, Vol. 29, No. 10, October 1996, pp. 99-111.

[YC95] A. Yu and J. Chen, *The POSTGRES95 User Manual*, Version 1.0, September 1995. See also <http://www.postgresql.org/>.

[ZO97] T. Zhao and M. Overmars, *Forms Library: A Graphical User Interface Toolkit for X*, Version 0.86, March 1997. See also <http://bragg.phys.uwm.edu/xforms/> and <http://www.westworld.com/~dau/xforms/>.

[ZWZD92] S. Zhou, J. Wang, X. Zheng, and P. Delisle, *UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, University of Toronto, Computer Systems Research Institute, Technical Report CSRI-257, Toronto, Canada, April 1992.

[Zigm96] D. Zigmond, *Uniform Resource Locators for Television and Telephony*, Internet Draft, November 1996. See also <http://www.ics.uci.edu/pub/ietf/uri/>.