Università di Pisa
Dipartimento di Informatica

Technical Report: TR-02-10

Advanced Parallel and Distributed Programming Environments

P. D'Ambra[a], M. Danelutto[b], D. di Serafino[c a], M. Lapegna[d a]
[a] CPS–CNR, Naples, Italy.
[b] Dept. of Computer Science, University of Pisa
[c] Dept. of Mathematics, Second University of Naples, Caserta, Italy
[d] Depat. of Mathematics and its Applications,
University "Federico II", Naples, Italy

July 1, 2002

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Advanced Parallel and Distributed Programming Environments

P. D'Ambra[a], M. Danelutto[b], D. di Serafino[c a], M. Lapegna[d a]
[a] Center for Research on Parallel Computing and Supercomputers (CPS) - CNR, Complesso Monte S. Angelo, Via Cintia, I-80126 Naples, Italy.
[b] Department of Computer Science, University of Pisa, Corso Italia, 40, I-56125 Pisa, Italy
[c] Department of Mathematics, Second University of Naples, Via Vivaldi, 43, I-81100 Caserta, Italy
[d] Department of Mathematics and its Applications, University of Naples "Federico II", Complesso Monte S. Angelo, Via Cintia, I-80126 Naples, Italy

July 1, 2002

Abstract

In this paper we provide a view of the design and development activity concerning advanced environments for parallel and distributed computing. We start from assessing the main issues driving this research track, in the areas of hardware and software technology and of applications. Then, we identify some key concepts, that can be considered as common guidelines and goals in the development of modern advanced environments, and we come up with a "classification" of these environments into two main classes: Programming Environments and Problems Solving Environments. Both classes are widely discussed, in light of the key concepts previously outlined, and several examples are provided, in order to give a picture of the current status and trends.

Key words: Parallel and Distributed Computing, Programming Environments, Problem Solving Environments

[0]This work has been supported by the ASI-PQE2000 Programme *Development of Applications for Earth Observation with High-Performance Computing Systems and Tools*, and by the CNR Agenzia 2000 Programme *An Environment for the Development of Multi-platform and Multi-language High-Performance Applications based on the Object Model and on Structured Parallel Programming.* Prelimiary version of a paper submitted to "Parallel Computing"

1 Introduction

This paper is aimed at discussing advanced environments for parallel and distributed computing. With the term "advanced environments" we mean both hardware and software resources arranged in such a way that they can be used to develop high-performance software for large-scale science and engineering applications. However, focus here is on the software ensemble providing such kind of environments, rather than on the hardware resources needed to support them. Therefore, the focus will be on advanced environments intended as *software architectures* for current high-end computers, including SMP, MPP and distributed/Grid-aware computer architectures.

There is no standard definition of software architecture, although this concept has deep roots in the context of Software Engineering. More than a definition of software architecture can be found in the literature [1]; here we quote the following ones:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.* [2]

and

> *... the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.* [3]

Note that we report both definitions, since each shows a different interesting aspect of a software architecture. The former explicitly refers to the externally visible properties of software components, intended as their services and behaviors, and hence stresses the importance of abstracting specific functionalities from the entire system by means of clear interfaces. The latter definition underlines the need of specifying a clear methodology for the design and the evolution of software components, that are closely connected to the advances of basic research in computing/computer science and technology.

The development of high-performance parallel and distributed applications requires a combined use of many software tools, that can be arranged in multiple layers. Usually, a base layer implements basic mechanisms, such as interprocessor and inter-system communication and memory hierarchy access, a top layer provides the users with functionalities that can be exploited to develop (hopefully) efficient, scalable and performant applications on the target architecture at hand, and one or more intermediate layers are built on lower ones and provides services to upper layers. The inter-layer and intra-layer interactions among software components should be realized through well-defined APIs, in order to achieve goals such as reliability, maintainability, extensibility, reuse, and so on.

In this paper we try to draw the main requirements, aspects and paradigms of modern advanced environments for parallel and distributed computing, in light of the concept of software architecture previously outlined. In Section 2 we point out the main issues driving advanced environments research and development. In Section 3 we assess our view of the directions where the research on advanced environments moves to and we identify two main typologies of advanced environments. Such typologies are discussed in Section 4 and 5, respectively, giving a picture of the last decade and outlining current trends.

2 Driving forces for advanced environments

Three areas can be identified where continuous advances and improvements drive or influence the development of environments to support parallel and distributed computing. They are *hardware technology*, *base software* and *applications.*

Hardware technology

Recent improvements in hardware technology and architecture concern basic components of parallel and distributed systems: *processors, memory hierarchies* and *networks.*

Processors, and, in particular, the commodity-off-the-shelf (COTS) processors used to develop most of current Top500 machines [4], reached impressive peak performance values. Memory hierarchies, including on-chip level one caches, and off-chip dedicated bus secondary caches, also deliver high bandwidth, in particular the bandwidth necessary to properly feed a typical 2Ghz super-pipeline processor with the data needed to keep its pipelines filled. Interprocessor networking technology has become more and more aggressive. More recent interconnection technology, based on the adoption of multiprocessor architectures with dedicated communication processor on each node, allowed complex, high-performance, interconnection network to be designed [5, 6]. Furthermore, the standard network technology reached Gbps with latencies accounting in the field of micro or submicro seconds [7], giving great impulse to cluster and high-performance distributed computing. Such technology, in particular, also exploits possibilities offered by optical and radio interconnection media.

Technologies have been or are currently being developed that also allow faster and faster CPUs to be implemented. These technologies either affect circuit manufacturing techniques (Heterojunction Bipolar Transistors, Quantum-effect devices, Rapid-Single-Flux-Quantum circuits and non-electronic DNA computing approach), or they affect the CPU architecture itself (Processor In Memory and multithreaded architectures, as an example) [8, 9, 10].

Base software technology

In the field of base software technology, we individuate four different aspects with impact on the design of advanced environments: *network management software, compiler technology, parallelism exploitation tools* and *software engineering techniques.*

The evolution of networking technologies and the increasing possibility of connecting a variety of different standing and mobile devices, over short and long distances, require new protocols, services and tools that cannot be provided by operating systems. Operating systems still include the basic mechanisms needed to handle network operations, but policies and high-level mechanisms have moved to *middleware*. Examples can be found in software architectures for heterogeneous and distributed computing, such as CORBA [11], where a software bus is implemented on the top of plain send/receive packet protocols handled by operating system, in order to interconnect mixed-language software components for execution in heterogeneous environments. In the more recent Grid Computing literature, the term middleware identifies a software layer that provides all the network services needed to support a set of applications in distributed, dynamic and cross-organizational environments. In the last decade many research efforts have been devoted to the development of Grid middleware [12, 13, 14, 15, 16]; today, the most widely used middleware toolkit is Globus, which implements basic services for building Grids and Grid-enabled applications, according to a layered model proposed to define the fundamental structure of a computational Grid [17].

In the meanwhile, research and application have introduced a novel concept: *peer-to-peer* computing, i.e. the ability to install cooperative processes on a network that do not rely on hierarchical, client-server relationships to reach their common goal [18]. Peer-to-peer technology has been impressively demonstrated with a software that has nothing to do with the application fields we discuss here, namely Napster. Napster allows file collection and exchange to be realized between a completely distributed community of users. The techniques used by Napster are now being considered to be used in the design of advanced, high-performance, parallel and distributed computing environments [19].

From the compiler technology point of view, we can see that new compilers afford more and more complex and effective tasks, leading to code with high performance, both in terms of sequential code execution (single processor resource optimization) and in terms of parallel/distributed computation (communication optimizations, scheduling and load-balancing optimizations, etc.). As an example, commodity processors nowadays implement "parallel instructions" (e.g. I32/64 SIMD extensions) that exploit the large number of functional units/ALUs present in the CPU itself. This has led to the introduction of suitable compiling algorithms able to extract (limited amounts of) SIMD parallelism out of plain sequential code [20]. As a further example, the wide diffusion of explicitly data-parallel languages such as HPF has led to the development of compiling techniques that optimize different aspects interrelated with parallel program execution, such as data distribution or communication optimization [21].

Parallelism exploitation techniques also improved. While in the past parallel applications often exploited a single kind of parallelism, i.e. data or control/task parallelism, nowadays advantages of exploiting both forms of parallelism within the same application have been demonstrated [22, 23, 24, 25]. Therefore, compiling technology exploiting "mix" of parallelism exploitation patterns has been developed, either based on the concept of template [26] or on macro data flow [27]. Such techniques apply different algorithms to get rid of data distribution or task mapping and scheduling.

Last but not least, software engineering research led to the development of a set of techniques that are having a deep impact on the whole program development process, both in the field of sequential programs and in that concerning parallel and distributed code development. In particular, *components* [28], *design patterns* [29] and all the related software design and development techniques become more and more mature and can be usefully exploited in the design of both high-performance software and environments supporting software development. We come back to this point later, in Sections 3 and 4.

Applications

A key role in the development of advanced environments for parallel and distributed computing is played by the applications, not only in the scientific fields which traditionally use high-performance computing, but also in public services, government policies and industry.

Scientists always wish to investigate increasingly complex problems with an increasing level of detail, requiring more and more computing power and sophisticated algorithms and software. The complexity and scale of scientific Grand Challenges justify the large investments towards petaflops machines, exhibiting parallelism at many different levels, and stress the importance of software environments allowing to effectively exploit such parallelism in order to get acceptable percentages of the peak performance, in a reliable and predictable way [30, 31]. Furthermore, those applications are usually multidisciplinary and require collaboration among different teams of scientists having a deep knowledge only of their own specific domains. Therefore, advanced simulations are characterized by the interaction and the composition of many simulation kernels and subsystems, often developed for different hardware/software platforms, with different tools and for various purposes.

On the other hand, it has been recently recognized that fields such as Medicine, Environment, Crise Management, etc. [10] can benefit from high-performance computing and networking, since emerging applications in those fields require the combined use of hardware, software, datasets, instruments and skills not available at a single site, but distributed geographically and among different organizations. Finally, an increasing number of industrial users looks at parallel and distributed computing as a key technology to reduce time-to-market and to increase competitiveness. In both cases, aspects such as rapid prototyping, fast code development, software restructuring and reuse, and interoperability, are even more central than in the development of high-end scientific applications.

The advances in hardware and base software and the varied requirements of the applications provide guidelines, methodologies and tools for the development of effective advanced environments for parallel and distributed computing. On the other hand, all the aspects previously outlined show the difficulties concerning the development of standard software infrastructures that are able to provide adequate answers to all needs.

3 Key concepts and trends

Once recognized the main areas influencing the design and development of environments for parallel and distributed computing, we wish to outline some general but key issues naturally emerging from advances and changes in those areas, and to analyze the typologies of modern advanced environments, both existing and under development.

Two fundamental concepts are software *integration* and *interoperability*, that refer to the growing need of assembling software modules or subsystems into an operational system and of doing such composition easily and reliably, possibly in a "plug-and-play" fashion [32]. Strictly connected is the concept of *reuse* of existing software, expressing the need of preserving years of research and development. To satisfy these needs many issues must be addressed: the definition of interoperability standards, including common software abstractions and interfaces and standard languages for describing them, models and mechanisms for linking different software units and transferring data between them, consistent schemes for memory management and error handling, and so on.

Answers to the problem of integration and interoperability can be found in the *component* programming model, that can be considered as an evolution of the object-oriented model, in order to overcome the complexity of composition and reuse of heterogeneous cross-project software, providing also seamless access to distributed software resources. Shortly, a component can be defined as an independent software unit, encapsulating a set of functionalities and having a well defined and published interface, that allows it to be composed with other components, according to the rules of a component architecture.

Component standards and implementations, e.g. OMG CORBA [11], Microsoft DCOM [33], Sun Java Beans and Enterprise Java Beans [34],were initially developed by the business world, that recognized their importance. However, they do not support basic needs of high-performance computing, such as the abstraction needed by parallel programming and the performance. A large effort is currently devoted to defining a standard component architecture for high-performance computing in the context of the Common Component Architecture (CCA) Forum [35].

Applications are even more complex and multidisciplinary, hardware, software, data and skills are distributed among different sites and organizations, and networking technologies and infrastructures make it possible to share and aggregate them. *Network-based computing* is therefore a central concept in the development of advanced computing environments. In this context, software is increasingly regarded as a service to be provided on demand, rather than a product to be obtained, installed and updated; hence, the concept of network-enabled computational servers is emerging, that implement the remote computing paradigm. Furthermore, distributed resources must not only be accessed, but must also be located and selected as much transparently as possible, and used in a reliable and secure way. Therefore, middleware, either special-purpose or general-purpose one, is even more important in building infrastructures that ease the development of applications in a distributed setting, and an intense research activity is carried out in this field (see, for example, [16]).

In this scenario, a significant role should be played by repositories of software interfaces, implementations and documentation, searchable by both human and machine clients. Mobile agents are recognized as a possible solution for resource discovering and monitoring, and expert assistants and knowledge discovery in database techniques are considered useful for the selection of the most appropriate software or software/machine pair [36, 37].

As already observed, *performance* is still a fundamental feature, and achieving it is made much more difficult by the variety and complexity of current hardware and software platforms. A challenging goal is to obtain software with portable performance. In this context, compiler technology and parallelism exploitation tools play a central role, as well as techniques for the development of self-adaptive, latency-tolerant, parameterized or performance-annotated codes [38].

Finally, an important design principle of advanced environments is *ease of use*, where the term "ease" is obviously related to the expertise of the target users. Among the other things, this requires languages and or graphical user interfaces (GUIs) close to the users' knowledge domain, for design, implementation composition, execution and analysis of applications, and other tools such as debuggers and profilers.

All the above concepts can be considered suitable common goals in the design of modern environments for parallel and distributed computing. However, existing implementations or proposals of such environments usually address different topics with different emphasis.

Depending on both the amount of programming effort required to develop satisfactory applications and on the application field addressed, we basically recognize two main typologies of environments: *Programming Environments (PEs)* and *Problem Solving Environments (PSEs)*.

With the generic term *Programming Environments* we denote the environments that basically provide all the tools needed to design, code and debug parallel and/or distributed applications, according to a given programming model or language. Programmers using a PE to develop a parallel application must know very well the features of the PE in order to be able to fully exploit its potentialities. In addition, they must have knowledge of the specific application field, in order to suitably exploit the PE features in the target application code. However, depending on the programming abstraction level provided by PE, the kind of knowledge required to the user may vary a lot. PEs just supporting a conjunction of plain sequential languages with common communication libraries (e.g. C or C++ with MPI [39] or PVM [40]) require a consistent knowledge to develop an efficient application, even in case that complex, effective debugging and optimizing tools are provided. On the other hand, PE based on design patterns or skeletons (see Section 4) already provide suitable, efficient frameworks that can be easily used to implement new applications, requiring a moderate knowledge of the base mechanisms used to implement parallel/distributed application features. PEs generally succeed in providing tools that allow very high-performance applications to be developed. This is due both to the fact that the mechanisms used in PEs are usually state-of-the-art, very efficient tools, and to the fact that a programmer is allowed to proceed with performance debugging within all the levels of the application.

Problem Solving Environments provide a set of user-friendly mechanisms and tools that allow to "build up" an application, within a specific application domain, by gluing together, with an intuitive compositional model and using some kind of problem-oriented language, different building blocks. Such building blocks range from libraries and application codes, to tools for I/O, data visualization and analysis, and interactive steering. In a distributed/Grid setting, they usually also include tools for data set online access, resource management, interconnection and monitoring tools. Therefore, PSEs enable applications to be developed without requiring the users to explicitly deal with most of the details related to solution algorithms and their implementations, to resource discovery, allocation and use, etc. The user of a PSE, therefore, must only have an appropriate knowledge of the specific application domain, plus some limited knowledge concerning the compositional model exported by the environment. In this context, performance is still important, but it is not the primary goal; indeed, a small loss in performance is acceptable to achieve a "more productive" application development environment.

Although the properties discussed above separate PEs from PSEs, different "contaminations" can be observed, that move positive features of each kind of environment on the other side. On one hand, higher and higher level PEs provide the programmer with abstractions of computation patterns that are closer and closer to the components typical of PSEs, while preserving general purpose-ness (e.g. PEs based on both skeleton and design pattern programming models). On the other hand, the compositional models implemented by PSEs include more and more general patterns; indeed, a current trend is designing PSE infrastructures that are application-independent, and then customizing them for a specific problem. Furthermore, the standardization of the component concept allows components developed for one PSE to be used in other PSEs, making PSEs more and more general purpose. Last but not least, performances demonstrated by software developed using PSEs are becoming closer to those of software developed using PEs, while the time spent in developing applications with PEs is becoming closer to the usually lower time required to develop applications using a PSE.

A discussion on PEs and PSEs, with focus on the current status and perspectives, is carried out in the next two sections.

4 Programming Environments

Programming Environments have been traditionally developed as a set ot tools built on top of a given existing programming language, and of a particular library providing the basic parallel and/or distributed mechanisms. To a closer look, these PE do not exist "as a whole" but, instead, machine vendors as well as user communities build them by gluing together software components coming from different places or experiences. The base programming languages and models used turn out to be very well assessed. This allows to reuse, within a PE, lots of existing codes and libraries. However, the designers and implementors of PE tools must cope with all the features of the supported languages, in order to guarantee that integration of existing code in the parallel/distributed framework effectively and efficiently works.

As an example of traditional PE, we consider the one built around the *C* language and the *MPI* communication library. This PE usually runs on top of Linux/Unix workstation clusters and networks, as well as of widely used MPPs. It can include existing tools, such as editors with specialized editing modes able to handle C as well as MPI code (e.g. *emacs*), debuggers that can deal with multithreaded code and can be run in multiple instances to get rid of multiprocess code (*gdb, ddd, totalview* and the alike), profilers (*prof, gprof*), proper versioning software (*rcs, cvs*), etc. The programming model provided to the user is basically SPMD, which is definitely a well-known programming model. Other programming models are possible (the MPI-2 standard supplies also APIs to handle dynamic process generation), but nevertheless the most widely used way to write working and efficient MPI programs is SPMD. The C language is also a well-known (although definitely not a modern) programming language. Despite this fact, and despite the high effectiveness demonstrated by the single PE components, the development of a working and efficient parallel application using such PE is a really hard task. The programmer must know very well the SPMD programming model as well as the details needed to set up an SPMD process network, including those details needed to set up communication/synchronization channels between processes, otherwise he cannot succeed in writing a working parallel code.

Therefore, although very efficient applications can be developed using the C/MPI PE, and have actually been developed, nobody claims it is user friendly. On the other hand, code reuse can be easily achieved not only concerning C code, but also FORTRAN or C++ code, which, in particular, can be reused exploiting the common object code format of the Unix architecture. Furthermore, C offers bindings to APIs of commonly used middleware, such as CORBA, that can be exploited in parts of MPI C code, to access external functionalities or to provide parallel application functionalities to the external world, thus achieving a very rough level of interoperability.

The situation just described can be easily generalized to other PEs: *C++/ACE*, i.e. C++ used in conjunction with the ACE cooperative library [41], *HPF/MPI*, i.e. High Performance Fortran used to exploit data parallelism with MPI calls to express task parallelism forms that are not primitive within HPF, and so on.

The basic nature of "software collection" of these PEs can be better understood taking into account how different products can be used to provide some of the PE components. As an example, the C/MPI PE can exploit MPICH [42] as well as LAM [43] software to provide MPI compliant communications, C compilers can come either from machine vendors (Intel, IBM, etc.) or from the open source community (GNU, Cygnus), etc. There are also "organized software collections" that can be viewed as full PEs. A notable example, may be the only one, is the Linux Beowulf distribution. But in this case too the PE is just a collection of packages built starting from existing assessed software components coming from different sources.

While such traditional PEs were being developed, research concentrated on the design of parallel and distributed programming environments that could relieve the programmer from most of the parallelism exploitation details, while preserving his ability to handle the qualitative aspects of parallelism exploitation. This happened in three main flavors, with each flavour usually "inheriting" useful features from the others: *algorithmic skeletons*, *design patterns* and *coordination languages*.

The original skeleton idea was introduced by Cole in the last 80's. According to Cole, an algorithmical skeleton is a known reusable parallelism exploitation pattern. Cole originally proposed a fairly small set of skeletons that can be used to drive parallel application development. Starting from this idea, some PEs were developed, that provided the user with skeletons modeling different parallelism exploitation patterns. P^3L is an example of such kind of PEs. It was designed in early 90's within a joint "academic" project between HP and University of Pisa [44] and successivley refined into an industrial product named *SkIE* [45]. In $\mathrm{P}^3\mathrm{L}$ every skeleton can be instantiated providing the sequential code needed to model sequential parts of an application. Skeletons can also be nested in order to obtain more complex parallelism exploitation patterns. In any case, the work needed to generate parallel code out of skeleton code was mostly automated by $\mathrm{P}^3\mathrm{L}$ tools. Besides $\mathrm{P}^3\mathrm{L}$, notable examples of skeleton-based PEs developed in the 90's come from Imperial College [46, 47, 48, 49], the Basel group [50] and the Alberta group [51]. In late 90's a lot of research work was dedicated to skeletons by different research groups, leading to the development of different prototype skeleton-based PEs [52, 53, 54, 55, 56, 27].

By using skeleton-based PEs, programmers wrote simple and concise code, just modeling the qualitative aspects of parallelism exploitation. Then, either the skeleton compiling tools or their runtime support managed to handle all the quantitative and "mechanic" aspects of parallelism exploitation. However, two major problems affected skeleton-based PEs and avoided them to be used, but within a small group of researchers and parallel application developers. On one hand, most skeleton based programming systems (including $\mathrm{P}^3\mathrm{L}$) only provided a fixed set of skeletons. Although these sets covered the most common parallelism exploitation patterns, software developers often required slightly different skeletons, and there was no chance either to extend the language or to use, into the sequential code, alternative mechanisms (e.g. MPI or even sockets) without interfering with the global parallel semantics. On the other hand, skeleton PEs were usually provided with completely new languages and sometimes they did not allow dusty-deck code to be used as skeleton parameters.

This notwithstanding, skeleton based PEs demonstrated to be able to match performance values achieved by hand-written code, while guaranteeing shorted design and implementation times (in [57] performance results concerning ray tracers, OCR and other numerical applications are reported, while in [58] the results of an experiment aimed at measuring the effort required to write parallel applications with skeletons vs. that required using traditional PEs are discussed).

Coordination languages play a different role. They have been developed to solve the problem of having different software components to interact to perform complex tasks. *Linda* [59] is a notable example of something which is universally recognized to be a coordination tool. Linda provides a global *tuple* space along with simple operations working on the tuple space (in - reads and consumes a tuple, out - place a tuple in the tuple space, read - read a tuple, and eval - evaluate a tuple, a shortcut for starting new processes). It is not a real programming language. Instead, it has to be intended as a library to be linked to code written in any existing sequential programming languages, and to be used to make different processes to cooperate in order to achieve some common goal.

Coordination languages and tools immediately demonstrated that users can actually exploit them to implement a small number of interaction and cooperation patterns in many applications. In late 90's, it was understood that it would have been better to have more complex operations in a coordination framework, i.e. not only the basic operations needed to make existing different processes to cooperate by synchronizing and exchanging messages, but also the complex mechanisms used to make a set of processes to cooperate in a structured way. Therefore, the research activity on coordination languages focused on the development of environments that could provide the users with primitive mechanisms, interpreted as well as compiled, directly implementing the most typical parallelism exploitation patterns, in the same way skeletons do within the skeleton-based languages.

Differently from plain skeletons, however, a major attention has been posed on the fact that processes interacting through these patterns may come from different environments: they may be written in different languages or they may also autonomously use communication libraries, such as MPI and PVM, or operating system mechanisms, such as sockets and threads. Therefore, coordination languages have been defined that allow different sequential or parallel processes to be integrated into the same parallel application via the coordination patterns.

Eventually, a third research line come to the scene, in the framework of advanced environments for parallel and distributed computing. In late 90's, the software engineering community find out that it was convenient to formalize somehow the structuring mechanisms used when writing sequential code, mainly in the object-oriented (OO) framework. This lead to the definition of the design pattern concept. A design pattern is a well-known pattern that can be adopted to write code matching given constraints. As an example, the "factory" pattern allows all those situations where a basic set of operations may be provided using different implementations to be modeled. The initial focus of the design pattern community was on the discovery and formalization of the whole set of patterns used in the OO framework to write efficient and maintainable programs. Soon after, a community of researchers involved in OO program development and in the design pattern stuff, started to be interested in parallel and distributed computations. The step moving patterns to parallel and distributed computing was natural. Therefore, some groups started to define parallel design patterns, aimed at modeling the most common parallel programming techniques. This led to two consequences: on one hand, most of the interaction (communication/synchronization) patterns used in the skeleton and coordination languages framework have been modeled by proper design patterns; on the other hand, programming environments have been built that provide the user with parametric runnable implementations of parallel design patterns, in such a way that those patterns take care of all the details involved in the exploitation of parallelism, in the same way skeletons do in the skeleton framework.

A notable example of PE based on design patterns is CO_2P_3S, coming from the Alberta University [25, 60]. $\mathrm{CO_2P_3S}$ provides the user with a set of parallel design patterns which are implemented in a framework and can be used to model the parallel behaviour of an application. The user simply subclasses a pattern and provides the hook methods used to model the application specific code. Different levels of interaction are assumed in $\mathrm{CO_2P_3S}$: application developers may just use patterns, but may also intervene on the implementation layer framework to make performance tuning. Other groups are working on parallel design patterns, either in the perspective of providing users with design-pattern-based PEs [61], or to better assess parallel design pattern concepts [62, 63]. One more example of PE inheriting concepts from the OO world (although not directly from the design patterns) is *POOMA* [64], which uses C++ templates to implement high-level parallel features (basically a parallel/distributed array data type).

Last but not least, the component model came to the high-performance parallel and distributed computing scene, and affected the development of advanced environments, exploiting somehow the experiences from skeletons, design patterns and coordination languages. As previously noted, the component technology has been developed mainly with the aim of providing easy and affordable software composition and reuse. Some widely used software packages (e.g. window managers) are based on the concept of component; components are parametrized and reused in multiple places in these packages, thus boosting the software development process. Concerning advanced parallel and distributed PEs, the component technology has been adopted to provide a feature that was not formerly present, namely interoperability. On one hand, by making parallel application to look like standard components (i.e. objects in a CORBA framework), the parallel application code become usable outside its developing environment, even to those users that are not familiar with parallelism. On the other hand, by allowing to import and use external components in a parallel or distributed application prototype, the development of new applications can be performed faster, and better code reuse capabilities can be provided to the final user. CORBA itself has been used as the basic component mechanism in different projects aimed at providing advanced parallel PEs [65, 66]. Currently, efforts are carried out to develop frameworks that implement the specifications given by the CCA Forum. Notable examples are *CCAFFEINE* [67], for developing SPMD parallel applications, and *XCAT* [68], for building Grid applications.

Altogether, skeletons, design patterns and coordination languages provided suitable ways to overcome a main problem in developing parallel and distributed applications with classical communication libraries, middleware or operating system mechanisms: writing all the low-level code needed to set up an effective application, i.e. the code for process network setup, mapping and scheduling, communication coding, synchronization handling and so on. New PEs are being designed that include features from the three worlds. As an example, *ASSIST* [69, 70] provides a skeleton-based coordination language that allows components (distributed objects, actually) to be used within the sequential portions of code encapsulated in skeletons. Furthermore, it has been designed in such a way that complete ASSIST programs can be encapsulated within a definite CORBA object, that can be used from (possibily sequential) CORBA compliant applications. ASSIST provides features that are aimed at overcoming some of the problems found in other skeleton-based parallel programming environments. In particular, ASSIST provides methods to define and use arbitrary graphs of concurrent activities (each described by one of the supported skeletons) as well as a new skeleton (the *parmod* one) modelling the more common data-parallel and task-parallel parallelism exploitation patterns. Currently, ASSIST is being developed for homogeneous and heterogeneous workstation clusters. Plans have been made to extend ASSIST in such a way that it can be used on Grids. More generally, the skeleton and design pattern experience is moving towards the Grid community, to match some of the "desiderata" about Grid programming models and tools [71, 72].

However, the users of any existing PE are still required to be somehow aware of the parallel and/or distributed platform they are operating on. Furthermore, they must learn a consistent amount of details concerning the PE in order to be able to use it. This makes a fundamental difference with the PSEs. Although some of the PSEs discussed in the next section also require consistent knowledge related to parallelism exploitation, the original idea of PSE is such that only application-specific knowledge is required to the PSE user in order to be able to develop parallel/distributed applications.

5 Problem Solving Environments

Problem Solving Environments can be considered as a "natural" evolution of the computational approach to the solution of scientific and engineering problems. For many years, this approach consisted in the development of large monolithic codes, usually written by a restricted group of people with a good level of expertise of the specific application domain and a reasonable level of expertise of computing techniques. However, since from the 70's, it was recognized, in the scientific community, that an effective solution of even more complex application problems would have required specialized computing skills, to exploit the advances in hardware, software and algorithmic technologies. Software libraries begun to be developed to solve computational kernels common to different applications, with the final goal of enhancing the quality of the results, while saving a lot of human effort [73, 74]. Later on, PSEs appeared on the scene, to satisfy the increasing need of integrated environments supporting the whole process of development of an application, from problem description to solution analysis, in a specific domain [75].

The emergence of a wide variety of parallel and distributed computer architectures have stressed the difficulties arising in the development of efficient and reliable software, and have emphasized the role of high-quality software modules in building computational applications, and even more the role of PSEs providing such modules as well as user-friendly mechanisms for their integration and interaction [76, 77]. The advantages of the modular approach have more clearly appeared also in the business world, although monolithic codes are still dominant there; some efforts have been carried out to exploit parallel library software modules into existing industrial application codes [78, 79, 80], and many more efforts have been devoted, as already observed in previous sections, to the development of component architectures and environments, allowing the interoperation of distributed software units, possibly produced by various developers and running on different hardware/software platforms.

The first "clear" definition of PSE was given by Gallopoulos, Houstis and Rice in the 90's [81]:

> *A PSE is a computer system that provides all the computational facilities needed to solve a target class of problems. These features include advanced solution methods, automatic and semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems, so users can run them without specialized knowledge of the underlying computer hardware or software. ...*

The above definition sets the main general features of a PSE and is still able to describe many modern PSEs. However, as observed by Houstis and Rice themselves in 2000 [82], due to rapid advances in high-performance computing and networking technologies, computational modeling is shifting from the single physical component design to the design of a whole physical system, with a large number of components that have different shapes, obey different physical laws and manufacturing constraints, and interact with each other through geometric and physical interfaces. In this scenario, the concept of *Multidisciplinary*, or *Multiphysics*, *PSE (MPSE)*, has emerged. Simply speaking, a MPSE can be defined as *a framework and software kernel for combining PSEs for tailored, flexible, multidisciplinary applications* [36]; furthermore, it is naturally thought as network or Grid-enabled.

More generally, the research activity in the field of PSEs has been so active that several very different environments are available or under development, which can be classified as PSEs, since they exhibit many or all the features reported in the above definitions. Furthermore, they implement these features with different base technologies, architectural solutions, levels of user interaction and goals.

An exhaustive discussion of PSEs is beyond the scope of this paper. Our aim is to sketch the current status of PSEs in the context of parallel and distributed computing, through a few representative projects, for modeling, simulation and desing optimization in scientific and engineering applications. More details on PSEs can be found, for example, in [75, 77].

To give an idea of "traditional" PSEs for parallel computing architectures, we consider the *Portable Extensible Toolkit for Scientific Computation (PETSc)* [83], developed at Argonne National Laboratory, and *Parallel ELLPACK (//ELLPACK)*, developed at Purdue University [84]. Both of them have a quite general scope, i.e. applications modeled by Partial Differential Equations (PDEs), but differs in their architecture and set of tools, thus allowing different levels of user interaction.

In PETSc, data structures and support for distributed matrix/vector management, preconditioners, linear solvers, non-linear solvers and time-stepping routines are organized into a hierarchical architecture, using an object-oriented programming model. The bottom layer of this architecture is implemented on the top of the MPI and BLAS [85, 86, 87] standard interfaces and of the LAPACK library [88], thus ensuring portability on a wide range of machines. Routines from other well-known numerical software libraries, converted to the C language, are included into higher layers. C/C++ and Fortran interfaces are provided at the highest level. PETSc can be used at a beginner level, but it is suited for advanced users wishing to have a detailed control over the solution process. Although PETSc does not have all of the features reported in the definition of PSE, it is widely used by the scientific community, since it makes available a large selection of advanced solvers for main computational kernels arising in scientific simulations, hiding the complexity of algorithms and data structures.

//ELLPACK evolved from the sequential ELLPACK PSE [89], not only by introducing PDE solvers and facilities for parallel processing, but also by including other discretization techniques and an advanced graphical user interface (GUI). //ELLPACK has a software architecture consisting of five main layers [84]. The top one is the above mentioned GUI, which allows to specify the PDE problem, the solution process and various post-processing analyses, with the help of a knowledge-based system, assisting the users in their choices. The specifications are transformed into a high-level PDE-oriented language, which is the second layer of interface of the architecture, and, at the third layer, the //ELLPACK language preprocessor compiles this language into a Fortran program, that calls the library modules needed by the solution process. The fourth layer is the execution environment, which assists users in compiling and running //ELLPACK programs, and is therefore responsible for locating and allocating hardware and software resources and for managing data scatter/gather to/from distributed machines. The fifth, and bottom, layer is composed by several libraries of sequential and parallel PDE solvers, both "native" and "foreign", provided with suitable interfaces that allow them to be composed and to interact with the whole system; the parallel solvers have been implemented on the top of many communication libraries.

Other traditional parallel PSEs could be mentioned here, targeted at different application domains and implementing somewhat different software architectures, but showing essentially the same general features of the previous ones [75]. To give just an example of a PSE with a narrow, i.e. more specific, scope, we cite the *AirShed Modeler*, developed by the University of California-Irvine and the California Institute of Technology, which provides a workbench for simulations with different air quality models and solution algorithms [90]. It is also interesting to mention CAMEL, as an example of parallel PSE based on the cellular automata approach to scientific and engineering modeling [91].

The previous PSEs do not implement the modern concept of network-based computing. Actually, an interesting evolution toward that direction is *WebPDELab* [92], which is an Internet-based client-server implementation of //ELLPACK, and shows a possible approach for building network-enabled computational servers. ¿From the point of view of software architecture, it can be seen as an extension of //ELLPACK, through the addition of further layers addressing issues of network-based computing. The WebPDElab server is accessed from the WebPDELab web site, by using a Java-enabled browser; indeed, the //ELLPACK GUI is made available through a Java-based remote display system, which uses the TCP/IP protocol. A WebPDELab manager, made of CGI scripts, controls all the user-server interactions; it also select the machine with the lightest traffic among the available ones, to run the //ELLPACK software, in a pool of machines available at Purdue University.

A different approach to building a network-enabled computational server is implemented by *NetSolve*, under development at the Innovative Computing Laboratory of University of Tenneesee [93]. In this case, a distributed virtual library for numerical computation is made available to the user by simple APIs for a wide variety of programming languages/environments, like C, Fortran, Java, Matlab and Mathematica. NetSolve can be regarded as a PSE for remote computing and, more generally, for Grid computing. The NetSolve architecture is based on a client-server design, with intelligent agents keeping the state of the whole system. At the top layer, the NetSolve client library is linked in with the user's application, that makes RPC-style calls to NetSolve APIs for specific services. At the bottom layer, there is a set of loosely-connected machines (workstations, SMPs, MPPs, clusters), in a local or non-local, homogeneous or heterogeneous network, running a NetSolve computational server, i.e. a computational resource, that has access to sequential and parallel libraries for scientific computing. The bottom layer, in turn, is built on the "abstract machine" that is provided by middleware and by isingle-machine software. The whole system can be seen as a completely connected graph, in which every agent maintains a database of NetSolve servers along with their hardware/software capabilities and dynamic usage statistics, for using this information to allocate the "best" resource for client requests, balance the load among the servers and keep track of failed ones. It is also worth noting that NetSolve provides the possibility of extending its services by the use of Problem Description Files to generate wrappers to library codes.

Somewhat similar projects to build remote computing servers are *Ninf* [94], *Nimrod* [95], and *MetaNEOS* [96], but their discussion is beyond the scope of this paper. We spend instead some more words on NetSolve, to show another interesting approach in the extension and evolution of PSEs toward Grid computing. Rather than a PSE, NetSolve is more often "classified" as a middleware enabling the access to remote hardware and software resources. Therefore, it offers a possibility for tying together PSEs and Grid resources. In other words, NetSolve can act as a middle layer between a front-end, such as a PSE, and a Grid back-end, such as Globus. In this context, some efforts have been carried out to integrate NetSolve with the *SCIRun* PSE [97], under continuous development at the SCI Institute of University of Utah [98]. SCIRun has been designed to allow interactive construction, debugging and steering of large-scale scientific computations, via a component-based visual programming model. The SCIRun architecture has been built looking at an application as a dataflow graph of computational modules, linked together in a visual composition environment. Execution is multi-threaded, with each SCIRun module having its own thread of execution and running as soon as it receives all of its own parameters; therefore, task parallelism is allowed, on SMP machines. This point makes difficult to extend SCIRun to a distributed-memory environment. The integration of SCIRun and NetSolve, allows not only to extend the functionality of the PSE with optimized numerical libraries, but also to provide a mechanism to distribute some compute-intensive task to an MPP, without merging the SCIRun multi-threaded environment with the MPP message-passing environment.

The RPC-style mechanism exploited by NetSolve can be used as a possible mechanism supporting the integration of heterogeneous and distributed units into large software systems, in the same way other common communication and cooperation mechanisms can be used. Another approach, also exploiting the agent technology, is that implemented in in the *GasTurbnLab* project, at Purdue University, for developing a MPSE that supports the simulations needed for the design of efficient gas turbine engines [36]. The whole PSE framework is based on a network of computational agents, assuming a network-enabled runtime support environment. Three main architectural layers can be identified: the user interface, the middleware and the computational software infrastructure. A visual composition environment, to construct and wire components in the form of a dataflow graph, is provided as user interface. The Grasshoper distributed agent environment is used as middleware, to facilitate the agent-based computational simulation paradigm; it runs on all the distributed hosts making the hardware infrastructure and manages resource selection, allocation and monitoring through the so-called Database Agents and Resource Agents. The computational software infrastructure, determined by the target class of problems, is made essentially by Computational Fluid Dynamics codes, such as ALE-3D and KIVA-3V, and by //ELLPACK modules, encapsulated within agents, using a multi-layered model, designed to facilitate agent mobility.

A completely different approach to software integration and reuse is based on the component programming model. In this context, several CORBA-based PSEs have been designed. A representative example is the *MDS-PSE* project, under development at Cardiff University and targeted at Molecular Dynamics Simulations (MDS) [99]. In the MDS-PSE architectural design, each component is represented by a well-defined component model specified in XML and is stored in a component repository. A Visual Program Composition Environment enable users to build and edit applications by plugging together components, by inserting components into pre-defined templates or by replacing components in higher-level hierarchical components; a Java-based expert system gives the user some advice on the use of components. An Intelligent Resource Management System assigns to different computational resources the tasks corresponding to the composed application. Either a whole MPI-based legacy code or subsystems of it have been wrapped as CORBA objects with no extensions of OMG's CORBA specification and IDL compiler; a combination of the MPI run-time with the CORBA environment allows to use MPI to manage intra-communications of components, and the CORBA ORB to manage inter-communications of components [100].

A slightly different solution for wrapping MPI legacy codes as CORBA objects has been chosen in the Esprit project *JaCo3*, carried out by many European research and industry partners to develop a PSE for building multi-code simulation applications. In this case, the CORBA object model has been extended by introducing the concept of parallel object and IDL distributed data structures (arrays and sequences) [101], following the approach proposed in *Pardis* [65].

The different solutions to the problem of interoperability and software reuse in a distributed and/or Grid environment naturally raise the problem of having a well-defined standard to be used in developing large high-performance multi-component applications. The CCA Forum is certainly addressing this problem, not only by defining the required specifications, but also working to develop CCA-compliant frameworks, such as the already mentioned CCAFFEINE and XCAT, and CCA-compliant components to be connected inside the frameworks, such as the ones developed at the Argonne National Laboratory [102].

¿From the previous discussion on PSEs for parallel, distributed and Grid computing we see that a great effort is currently carried out to design and build PSEs that can match the current needs of science and technology. Furthermore, an interesting debate on features and requirements of the future generations of PSEs is active, in order to individuate the research issues to be addressed in the medium and long term.

6 Conclusions

In this paper we tried to give a unified view of the research and development concerning modern advanced environments for parallel and distributed computing. We first recognized those issues in hardware and software technology and in applications that significantly affect the design of modern environments. Then, we identified some key concepts that appear to be common guidelines and goals in the development of such environments, although they are addressed with different emphasis and providing different solutions. We grouped all the advanced environments for parallel and distributed computing into two main, very wide and general classes, Programming Environments and Problem Solving Environments, making a distinction which is essentially based on target users and fields of application. A large discussion on both types of environments followed, with many examples of PEs and PSEs, to give a picture of the current status and trends. From this discussion, it appears that a significant evolution took place in the last decade and that the research in this field is currently very active, to match the rapid changes in technology and the new requirements of applications. On the other hand, to make such environments effectively, easily and reliably usable by a wide community, many problems must still be addressed and solved.

Finally, we report the "origins" of this paper. The authors are involved in two ongoing Italian National Research Programmes, funded by the Italian Space Agency (ASI) and by the Italian National Research Council (CNR). These Programmes share the objective of developing an advanced PE for a wide range of parallel and distributed architectures, which provides also numerical software components that can be easily exploited into existing and new high-performance applications. In this context, the authors met several times and discussed their different experiences and points of view, coming to the view presented in this paper.

References

[1] Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/architecture/definitions.html.

[2] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice (Addison-Wesley, 1997).

[3] D. Garlan, D. Perry, Introduction to the Special Issue on Software Architecture, IEEE Transactions on Software Engineering 21 (1995) 269–274.

[4] Top500.org, Top500 supercomputer sites, http://www.top500.org.

[5] QSNet, http://www.quadrics.com/website/pages/02qsn.html.

[6] The Myricom home page, http://www.myrinet.com/.

[7] J. Hsieh, V. Mashayekhi, R. Rooholamini, Architectural and Performance Evaluation of GigaNet and Myrinet Interconnections on Clusters of Small-Scale SMP Servers, in: Proceedings of SuperComputing 2000, 2000.

[8] P. Messina, High-Performance Computers: The Next Generation (Part I), Computers in Physics 11 (1997) 454–466.

[9] P. Messina, High-Performance Computers: The Next Generation (Part II), Computers in Physics 11 (1997) 598–610.

[10] USA Interagency WG on Information Technology Research and Development, Networking and Information Technology Research and Development, 2001, available at http://www.itrd.gov/pubs/blue02.

[11] OMG, the CORBA home page, http://www.corba.org/.

[12] I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, Intl J. Supercomputer Applications 11 (1997) 115–128. See also the Globus Project home page, http://www.globus.org/.

[13] A. Grimshaw, A. Ferrari, F. Knabe, M. Humphrey, Legion: An Operating System for Wide-Area Computing, IEEE Computer 32 (1999) 29–37.

[14] J. Basney, M. Livny, Deploying a High Throughput Computing Cluster, in R. Buyya, ed., Performance Cluster Computing, Vol. 1 (Prentice Hall PTR, 1999). See also the CONDOR Project home page, http://www.cs.wisc.edu/condor/.

[15] R. Wolski, N. Spring, J. Hayes, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, Future Generation Computer Systems 15 (1999) 757–768.

[16] USA NSF Middleware Initiative, http://www.nsf-middleware.org.

[17] I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, Intl J. Supercomputer Applications 15 (2001).

[18] G. Fox, Peer-to-Peer Networks, Computing in Science & Engineering 3 (3) (2001) 75–77.

[19] G. Fox, D. Gannon, Computational Grids, Computing in Science & Engineering 4 (2001) 74–77.

[20] GNU, GCC home page, http://www.gnu.org/software/gcc/gcc.html.

[21] The Portland Group home page, http://www.pgroup.com/index.htm.

[22] P. Au, J. Darlington, M. Ghanem, Y. Guo, H. To, J. Yang, Coordinating heterogeneous parallel computation, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (eds.), Europar '96, 1996, 601–614.

[23] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, P^3L: a Structured High Level Programming Language and its Structured Support, Concurrency Practice and Experience 7 (3) (1995) 225–255.

[24] S. Chakrabarti, J. Demmel, D. Yelick, Modeling the Benefits of Mixed Data and Task Parallelism, Lapack Working Note 97, Tech. Rep. CS-95-289, University of Tennessee (1995), available at http://www.netlib.org/lapack/lawns/.

[25] S. McDonald, D. Szafron, J. Schaeffer, S. Bromling, Generating Parallel Program Frameworks from Parallel Design Patterns, in: A. Bode, T. Ludwing, W. Karl, R. Wismüller (eds.), Euro-Par 2000 Parallel Processing, Lecture Notes in Computer Science, Vol. 1900 (Springer Verlag, 2000) 95–105.

[26] S. Pelagatti, Structured Development of Parallel Programs (Taylor&Francis, 1998).

[27] M. Danelutto, Efficient support for skeletons on workstation clusters, Parallel Processing Letters 11 (1) (2001) 41–56.

[28] C. Szyperski, Component Software: Beyond Object-Oriented Programming (ACM Press, 1998).

[29] E. Gamma, R. Helm, R. Johnson, J. Vissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley, 1994).

[30] J. J. Dongarra, D. W. Walker, The Quest for Petascale Computing, Computing in Science & Engineering 3 (3) (2001) 32–39.

[31] R. Stevens, Applications for PetaFLOPs, PetaFLOPs II Conference (1999), available at http://www.cacr.caltech.edu/pflops2/.

[32] J. Ambrosiano, D. Quinlan, R. A. Armstrong, Software Interoperability, ASCI Technology Prospectus on Simulation and Computational Science 1 (2001) 25–35.

[33] M. Horsmann, M. Kirtland, DCOM Architecture, Microsoft White Paper, 1997, available at http://www.microsoft.com/com/wpaper/.

[34] The Sun Java home page, http://java.sun.com.

[35] Common Component Architecture Forum, http://www.cca-forum.org/.

[36] S. Markus et al., An Agent-Based Netcentric Framework for Multidisciplinary Problem Solving Environments, International Journal of Computational Engineering Science 1 (2000) 33–60. See also GasTurbnLab home page, http://www.cs.purdue.edu/research/cse/gasturbn/.

[37] A. Joshi, N. Ramakrishnan, E.N. Houstis, Multiagent Recommender Systems in Networked Scientific Computing, in: E.N. Houstis, J.R. Rice, E. Gallopoulos, R. Bramley (eds.), Enabling Technologies for Computational Science. Frameworks, Middleware and Environments (Kluwer Academic Publishers, 2000), 213–223.

[38] A. Petitet et al., Numerical Libraries and The Grid, International Journal of High Performance Applications and Supercomputing 15 (2001) 359–374.

[39] M. Snir et. al, MPI: The Complete Reference, 2-volume set (MIT Press, 1998). The MPI standard home page: http://www-unix.mcs.anl.gov/mpi/.

[40] A. Geist et al, PVM - Parallel Virtual Machine (MIT Press, 1994).

[41] The Adaptive Communication Environment (ACE) home page, http://www.cs.wustl.edu/ schmidt/ACE.html.

[42] MPICH home page, http://www-unix.mcs.anl.gov/mpi/mpich/.

[43] LAM-MPI home page, http://www.lam-mpi.org/.

[44] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, M. Vanneschi, A methodology for the development and support of massively parallel programs, Future Generation Computer Systems 8 (1–3) (1992) 205–220.

[45] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi, SkIE: a heterogeneous environment for HPC applications, Parallel Computing 25 (1999) 1827–1852.

[46] J. Darlington, A. J. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, R. L. While, Parallel Programming Using Skeleton Functions, in: M. R. A. Bode, G. Wolf (eds.), PARLE'93 Parallel Architectures and Langauges Europe, Lecture Notes in Computer Science, Vol. 694 (Springer Verlag, 1993).

[47] J. Darlington, Y. Guo, H. W. To, J. Yang, Parallel Skeletons for Structured Composition, in: Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, 1995.

[48] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, M. Kohler, Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages, in: Third Parallel Computing Workshop (PCW'94), Fujitsu Laboratories Ltd., 1994.

[49] J. Darlington, M. Ghanem, H. W. To, Structured Parallel Programming, in: Programming Models for Massively Parallel Computers, IEEE Computer Society Press, 1993.

[50] H. Burkhart, S. Gutzwiller, Steps Towards Reusability and Portability in Parallel Programming, in: K. M. Decker, R. M. Rehmann (eds.), Programming Environments for Massively Parallel Distributed Systems, Birkhauser, 1994, 147–157.

[51] J. Schaeffer, D. Szafron, G. Lobe, I. Parsons, The Enterprise Model for Developing Parallel Applications, IEEE Parallel and Distributed Technology 1 (3) (1993) 85–96.

[52] T. Bratvold, Skeleton-based parallelisation of functional programs, Ph.D. thesis, Heriot-Watt University (1994).

[53] J. Serot, Embodying Parallel Functional Skeletons: An Experimental Implementation on Top of MPI, in: G. Lengauer, Griebl (Ed.), Euro-Par'97 Parallel Processing, Lecture Notes in Computer Science, Vol. 1300 (Springer Verlag, 1997), 629–633.

[54] J. Serot, D. Ginhac, J. Derutin, SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications, in: Proceedings of the 5th International Parallel Computing Technologies Conference (PaCT-99), 1999.

[55] M. Südholt, The transformational derivation of parallel programs using data-distribution algebras and skeletons, Ph.D. thesis, Technische Universität Berlin, 1997, available at http://www.emn.fr/dept_info/perso/sudholt/papers/phd.ps.gz

[56] B. Bacci, S. Gorlatch, C. Lengauer, S. Pelagatti, Skeletons and transformations in an integrated parallel programming environment, 1999, to be presented at *PaCT'99*, St. Petersburg, September 1999.

[57] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, Summarising an experiment in parallel programming language design, in: B. Hertzberger, G. Serazzi (eds.), High-Performance Computing and Networking, Vol. 919, 1995, 8–13.

[58] D. Szafron, J. Schaeffer, Experimentally assessing the Usability of Parallel Programming Systems, in: Programming Environments for Massively Parallel Distributed Systems, Birkhauser, 1994, 203–212.

[59] N. Carriero, D. Gelernter, Linda in context, Communications of the ACM 32 (4) 444–58.

[60] S. McDonald, J. Anvik, D. Szafron, J. Schaeffer, S. Bromling, K. Tan, From Patterns to Frameworks to Parallel Programs, to appear on "Parallel Computing"

[61] B. L. Massingill, T. G. Mattson, B. A. Sanders, A Pattern Language for Parallel Application Programs, in: Bode, Ludwig, Karl, Wismuller (eds.), Euro-Par 2000 Parallel Processing, Lecture Notes in Computer Science, Vol. 1900 (Springer Verlag, 2000) 678–681.

[62] B. L. Massingill, Experiments with Program Parallelization Using Archetypes and Stepwise Refinement, Tech. Rep. TR 98-012, Univeristy of Florida, CISE (1998).

[63] B. L. Massingill, T. G. Mattson, B. A. Sanders, A Pattern Language for Parallel Application Languages, Tech. Rep. TR 99-022, Univeristy of Florida, CISE (1999).

[64] The POOMA home page, http://www.acl.lanl.gov/pooma/.

[65] K. Keahey, D. Gannon, PARDIS: a CORBA-based Architecture for Application-Level Parallel Distributed Computation, in: Proceedings of Supercomputing 97, 1997, available at http://www.supercomp.org/sc97/proceedings/TECH/KEAHEY/INDEX.HTM.

[66] P. Beaugendre, T. Priol, C. René, Cobra: a CORBA-compliant programming environment for high-performance computing, Tech. Rep. PI 1141, INRIA, 1998, available at http://www.irisa.fr/EXTERNE/bibli/pi/1141/1141.html.

[67] B.A. Allan et al., The CCA Core Specification in a Distributed Memory SPMD Framework, submitted to Concurrency and Computation - Practice & Experience, available at http://www.cca-forum.org/old/ccafe03a/index.html.

[68] XCAT home page, http://www.extreme.indiana.edu/xcat/.

[69] M. Vanneschi, ASSIST: an Environment for Parallel and Distributed Portable Applications , Tech. Rep. TR-02-07, Dept. Computer Science, University of Pisa, Italy, 2002, available at http://www.di.unipi.it/ricerca/TR.

[70] M. Vanneschi, The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications , to appear on "Parallel Computing".

[71] Global Grid Forum, Application Programming Models home page, http://www.gridforum.org/7_APM/APS.htm.

[72] The GrADS home page, http://hipersoft.cs.rice.edu/grads/.

[73] J.R. Rice, Mathematical Software (Academic Press, 1971).

[74] W.J. Cody, Observations on the Mathematical Software Effort, in: W.R. Cowell (ed.), Sources and Development of Mathematical Software (Prentice-Hall, 1984).

[75] Problem Solving Environments home page, http://www-cgi.cs.purdue.edu/cgi-bin/acc/pses.cgi/.

[76] D. di Serafino, L. Maddalena, P. Messina, A. Murli, Some Perspectives on High-Performance Mathematical Software, in: R. De Leone, A. Murli, P.M. Pardalos, G. Toraldo (eds.), High Performance Algorithms and Software in Nonlinear Optimization (Kluwer Academic Publishers, 1998) 1–23.

[77] E.N. Houstis, J.R. Rice, E. Gallopoulos, R. Bramley (eds.), Enabling Technologies for Computational Science. Frameworks, Middleware and Environments (Kluwer Academic Publishers, 2000).

[78] D. di Serafino, L. Maddalena, A. Murli, PINEAPL: A European Project to Develop a Parallel Numerical Library for Industrial Applications, in: C. Lengauer, M. Griebl, S. Gorlatch (eds.) Euro-Par'97 Parallel Processing, Lecture Notes in Computer Science, Vol. 1300 (Springer, 1997) 1333–1339.

[79] I. de Bono, D. di Serafino, E. Ducloux, Using a General-Purpose Numerical Library to Parallelize an Industrial Application: Design of High-Performance Lasers, in: D. Pritchard, J. Reeve (eds.) Euro-Par'98 Parallel Processing, Lecture Notes in Computer Science, Vol. 1470 (Springer Verlag, 1998) 812–820.

[80] L. Arnone, P. D'Ambra, S. Filippone, A Parallel Version of KIVA-3 based on General-Purpose Numerical Software and its Use in Two-stroke Engine Applications, International Journal of Computer Research 10 (2001), Special Issue on Industrial Applications of Parallel Computing, 31–46.

[81] E. Gallopoulos, E. Houstis, J.R. Rice, Computer as Thinker/Doer: Problem-Solving Environments for Computational Science, IEEE Computational Science & Engineering 1 (2) (1994) 11–23.

[82] E.N. Houstis, J.R. Rice, Future Problem Solving Environments for Computational Science, 2000, available at http://www.webpdelab.org/doc.html.

[83] PETSc home page, http://www.mcs.anl.gov/petsc.

[84] E.N. Houstis at al., Parallel ELLPACK: a Problem Solving Environment for PDE Based Applications on Multicomputer Platforms, ACM Trans. Math. Soft. 24 (1998) 30–73. See also Parallel ELLPACK home page, http://www.cs.purdue.edu/research/cse/pellpack/.

[85] C.L. Lawson, R.J. Hanson, D. Kincaid, F.T. Krogh, Basic Linear Algebra Subprograms for FORTRAN Usage, ACM Trans. Math. Soft. 5 (1979) 308–323.

[86] J.J. Dongarra, J. Du Croz, S. Hammarling, R.J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Trans. Math. Soft. 14 (1988) 1–17.

[87] J.J. Dongarra, J. Du Croz, I.S. Duff, S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft. 16 (1990) 1–17.

[88] E. Anderson, LAPACK Users' Guide, Third Edition (SIAM, 1999).

[89] J.R. Rice, R.F. Boisvert, Solving Elliptic Problems Using ELLPACK, (Springer Verlag, 1984).

[90] D. Dabdub, K.M. Chandy, T.T. Hewett, Managing Specificity and Generality: Tailoring General Archetypal PSEs to Specific Users, in: E.N. Houstis, J.R. Rice, E. Gallopoulos, R. Bramley (eds.), Enabling Technologies for Computational Science. Frameworks, Middleware and Environments (Kluwer Academic Publishers, 2000) 65–77.

[91] G. Spezzano, D. Talia, S. Di Gregorio, A Parallel Cellular Tool for Interactive Modeling and Simulation, IEEE Computational Science & Engineering 3 (3) (1996) 33–43.

[92] E.N. Houstis, A.C. Catlin, N. Dhanjani, The WebPDELab Server: a Problem Solving Environment for PDE-based Applications, to appear. See also WebPDELab home page, http://www.webpdelab.org/

[93] D. Arnold et al., Users' Guide to NetSolve V1.4, Tech. Rep. CS-01-467, Department of Computer Science, University of Tennessee, 2001. See also NetSolve home page, http://icl.cs.utk.edu/netsolve/.

[94] Ninf home page, http://ninf.apgrid.org/.

[95] Nimrod home page, http://www.csse.monash.edu.au/˜davida/nimrod.html/.

[96] MetaNEOS home page, http://www-unix.mcs.anl.gov/metaneos/.

[97] M. Miller, C. Moulding, J. Dongarra, C. Johnson, Grid-Enabling Problem Solving Environments: a Case Study of SCIRun and NetSolve, in: Proceedings of High Performance Computing Symposium 2001 - Grand Challenges in Computer Simulation (HPC 2001), High Performance Simulation Environments, 2001, Seattle, Washington, 98–103.

[98] SCIRun home page, http://software.sci.utah.edu/scirun.html.

[99] D.W. Walker, M. Li, O.F. Rana, M.S. Shields, Y. Huang, The Software Architecture of a Distributed Problem-Solving Environment, Concurrency: Practice & Experience 12 (2000) 1455–1480.

[100] M. Li, O.F. Rana, D.W. Walker, Wrapping MPI-Based Lagacy Codes as Java/CORBA Components, Future Generation Computer Systems 18 (2) (2001) 213–223.

[101] C. René, T. Priol, MPI Code Encapsulation Using Parallel CORBA Object, in: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (IEEE, 1999) 3–10.

[102] B. Norris et al., Parallel Components for PDEs and Optimization: Some Issues and Experiences, to appear on "Parallel Computing".