Simple and Efficient Programming of Parallel Distributed Systems for Computational Scientists

Karsten M. Decker and René M. Rehmann
IAM, University of Berne
Länggassstrasse 51, CH–3012 Berne
SWITZERLAND

October 27, 1992

Abstract

Many problems in computational science can be mapped very efficiently to parallel distributed systems like loosely–coupled workstation clusters or more closely–coupled multicomputer systems. This paper proposes a methodology of parallel distributed programming and discusses its realization by means of a programming environment to overcome the notoriously difficult programming of distributed systems. The environment centers around a knowledge based system offering commonly used algorithmic skeletons together with extensive interactive guidance from the early design to the implementation phase and selection methods for skeletons. This programming assistant in the core is supplemented with three different interfaces for high–level design languages and a graphical user interface allowing the graphical representation of these languages. An interface to extend the knowledge base is also included. The investigation is part of the SPADE project which aims at the development of an integrated program and application development environment for problems in computational science on parallel architectures with distributed memory.

1 Introduction

The interest in simulation and modeling of complex systems on computer systems has significantly grown over the last few years. These activities, often of multidisciplinary nature, define the field of *computational science*. Besides being intensively used in universities and public research institutes, the basic methods of computational science are now also spreading rapidly in industrial research and development laboratories.

In the beginning, scientific calculations relied exclusively on standard department computers and mainframe computers in central computing centers. However, only with the advent of the first commercial single– and multi–processor high performance computer systems in the second half of the seventies, computational science became a fully recognized discipline by itself. It can be expected that the nineties, with its nowadays technically realizable massively parallel (distributed) computer systems, offering scalable computing power from the desk–top workstation up to computer systems with peak computing power in the TFLOPS range, will probably revolutionize the possibilities of computational science. Since many computational problems in natural science and engineering are characterized by the application of local functions to large and homogeneous data structures, the raw performance of massively parallel computers can often be easily exploited. For the first time, these architectures will allow to simulate and model the most complex systems of basic sciences and enginnering in the required size, while simultaneously taken into consideration all their essential properties.

The reason why today massively parallel, or more general, distributed systems, are not used more often by computational scientists, follows immediately from their pure programmability. This is especially true from the computational scientist point of view, who wants to use computers as a *tool* to solve his computational problem(s), but cannot afford to bother about too many purely technical aspects of computing. Adequate problem analysis and representation methods and supporting tools which he can understand and handle are not available. This difficult programmability of distributed systems is in sharp contrast to the convenience offered by today's commercial multi–processor shared memory high performance computer systems. Their programming relies not only on a programming model well–known from single–processor systems, but they are also equipped with auto–vectorizing and auto–parallelizing compilers backed up by a powerful and familiar operating system.

It is the purpose of this paper to propose a way out of the currently existing dilemma of computational scientists when they want to use parallel distributed systems: although they feel heavily attracted, they cannot use them, because they don't know how to program them effectively. Sect. 2 thoroughly investigates what would be required ideally by computational scientists, what is already available and provided by vendors of parallel distributed systems, and what is missing most urgently to let them master the steep learning–curve. Sect. 3 describes the proposed environment for parallel distributed programming. Starting with an analysis of the principal difficulties of programming distributed systems, we then describe a model of parallel distributed programs from which the notion of algorithmic skeletons is derived. These algorithmic skeletons are then used as a building block for the development of a methodology for parallel distributed programming. We then describe how a programming environment realizing this methodology can be put into practise, covering the specification of the functional requirements and major design considerations. Further topics in this section include a description of the operation of the environment and a discussion of the conceptual advantages of the system. The section closes with detailed considerations how the programming environment can be realized, i.e., what are the necessary prerequisites, how we want to proceed, and how the extensibility of the system is ensured. Our current activities are also indicated. The last section, Sect. 4, summarizes the paper and sketches the usefulness of our system also as an educational tool for teaching parallel distributed algorithm design.

2 Programming Parallel Distributed Systems: The Computational Scientist's Point of View

The requirements of computational scientists on the ideal programming environment for parallel distributed systems can be derived immediately from the often implicit assumptions and concepts used in training their programming skills: single processor hardware with single, unique address space, existence of suitable libraries providing reusable and efficient building blocks for their applications, and invariance of program performance behaviour with respect to the chosen computer architecture. In more technical terms this means that the majority of computational scientists would like to have and actually would need completely transparent parallel distributed programming, libraries containing efficient parallel building blocks and a portability platform fully preserving parallel efficiency. At the moment, none of these expectations can be fulfilled.

In contrast, the current situation is as follows. Vendors still need to concentrate on the provision of basic functionality, and cannot invest time and efforts in the provision of sufficient machine abstraction. In the past few years, a lot of work has been spent into the development of vendor–independent unified message–passing interfaces like, for example, PVM [1], PARMACS [2, 3], Express [4], and Trollius [5]. Some of these interfaces have been efficiently implemented on several different hardware platforms, thus providing some quasi–portability. Nevertheless, they still offer only the first step towards the specification of a virtual machine. A major obstacle of these interfaces is that they are in general too complicated to handle for the average computational scientist. Another line of research consists of the development of so–called *harnesses* which abstract to some extend from specific applications [6]. Tools to assist parallelization and tuning of already parallelized sequential code are now emerging, e.g., [4], [7]. Further, in the last few years, major efforts have been put into the definition and development of parallel languages like, for example, Fortran D [8], Vienna Fortran [9], HPF [10] and Kali [11]. These languages are designed to capture fine–grain data parallelism and enhance the existing Fortran standards with a rich set of data decomposition specifications. Whether they can efficiently handle the natural grain–size of parallelism inherent to most problems in science and engineering is questionable.

From the practical point of view, real *high–level* programming tools ensuring good parallel efficiency are missing most urgently. We believe that the latter is best guaranteed, if these tools are conceptually based on a programming model relying on explicit message–passing. There exist several research efforts into this direction. For example, the program template project by J. Dongarra [12], is targeted at understanding basic algorithmic features, and should allow some customization, retaining delicate numerical details. It is currently only in its infancies. However, the average (so–called) high–level programming tool proposed today does *not* really operate on a high level of abstraction; in general it has insufficient degree of abstraction to appropriately handle the natural level of granularity, and aspects of usage and reusability of programs or program components are insufficiently addressed. Currently, a real effort on algorithm design languages and tools supporting them does not exist.

3 An Environment for Programming Parallel Distributed System

3.1 Difficulties in Programming Distributed Systems

Efficient exploitation of the inherent parallelism of an algorithm necessarily requires a careful and detailed analysis of the data flow, i.e., the data dependencies. The crucial point is that the data flow analysis needs to be done to an extend far beyond that required for sequential programming and thus can in general not sufficiently addressed by computational scientists, in particular when the analysis is not method based and tool supported. Another basic difficulty of parallel programming based on the message–passing paradigm is the programmer's responsibility for the memory management. This follows immediately from the distributed nature of the non–uniform address space of a distributed system and implies that the decision whether data is available locally or needs to be communicated, is also offloaded to the programmer.

Further difficulties in programming distributed systems depend on the selected computational model:

The Master–Slave Model (MSM) introduces two distinct types of processes, a single master–process coordinating many slave–processes which perform the actual computational work in a synchronized fashion. The distribution of the data is such that the data structures are partitioned at the beginning of an application, and distributed to the slave–processes. Each slave–process executes the same sequential program on his part of the physical domain to be processed, interleaved with slave–slave communication steps. The master–process coordinates the computational activities on all the slave–processes. Difficulties in parallel distributed programming based on the MSM include the programmer's responsibility for the selection of the (most efficient) decomposition topology, the partitioning and the distribution of his data structures, and data coherence for communicated data and shared variables. All these difficulties often cannot appropriately be addressed by programmers which have no or only rudimentary experience in programming distributed systems.

The Farmer–Worker Model (FWM) also has two distinct types of processes, a single farmer–process which distributes the not necessarily identical computational tasks, and many worker–processes which do the work and return the results to the farmer. Besides the data to be processed, the workers can also receive the code required to process the data. A characteristic feature of a computational task is that it requires only the data provided by the farmer. It never relies on data which are simultaneously modified by other workers. Difficulties in parallel distributed programming based on the FWM include the coordination of computational tasks, i.e., proper synchronization and reduction of data returned from the workers to the farmer–process, and reasonable handling of the trade–off between balance of computational load and communication requirements.

All these considerations clearly imply that programming of distributed systems is much more demanding than sequential programming.

3.2 A Model of Parallel Distributed Programs

Although algorithms may in general differ considerably with respect to their computational action, it turns out that whole classes of algorithms have similar communication and coordination requirements, or have other meta–structures in common. This suggests to model parallel distributed programs by means of conceptual separation of the new, difficult features of parallel distributed programs from known features of single, unique address space programs, i.e., programs for shared memory architectures.

More specifically, the model describes a parallel distributed program as consisting of:

- An algorithmic skeleton, serving as an integrating framework for sequential computational components, and
- Sequential computational components, suitably determined from the shared memory model–based sequential algorithm.

To illustrate this model, we consider the algorithmic skeleton for a problem of the Single Program, Multiple Data (SPMD) type which is useful for all those problems that can be handled by (static) domain decomposition. Problems of this type can be conveniently modeled with the Master–Slave computational Model introduced above. In this special case, the algorithmic skeleton reduces to a communication and coordination skeleton.

Problems of another type might require to farm (not necessarily identical) computational tasks to independently operating worker–processes. Problems of this type can be modeled with the Farmer–Worker Model also introduced in the previous section. Here the key part of the parallel distributed algorithm is to dynamically distribute the data at run–time, to start new worker–processes and to properly synchronize and combine the data returned from the workers to the farmer–process. The algorithmic skeleton for this class of algorithms clearly differs from the simple communication and coordination skeleton that is sufficient for the MSM.

3.3 The Notion of Algorithmic Skeletons

Founding parallel distributed computing on the notion of algorithmic skeletons bears a number of advantages:

Good parallel efficiency. This is certainly one of the desirable goals in parallel distributed programming, at least from the computational scientist's point of view. Experience tells that the achievable parallel efficiency is deeply rooted in the communication and coordination structure. It will thus already be decided in the early stages of the algorithm design phase – unfortunately in that part of the software development cycle in which in general we cannot count on the computational scientist's expertise as explained above. Here carefully designed algorithmic skeletons can help.

Reusability. This is an important issue for economic software development. Prefabricated algorithms to the disposal of the programmer can help again, since they nicely introduce reusability of the critical parts of parallel distributed programs.

Conceptual reduction of parallel programming complexity. Since algorithmic skeletons help to reduce the efforts required to program a distributed system to those required to program a shared memory system with a single, unique address space, skeletons also reduce the complexity of programming parallel distributed system to a task which is surely manageable by computational scientists.

3.4 A Methodology for Parallel Distributed Programming

All these considerations suggest to prefabricate algorithmic skeletons, and to make them available to the programmer. To make the notion applicable, a methodology is required. The starting point for our discussion is a basic methodology which has been proposed independently by Cole [13] and Burkhart [14]:

3.4.1 A Basic Methodology for Parallel Distributed Programming

The methodology consists of the following three steps:

Step 1: Take a suitable prefabricated algorithmic skeleton.

Step 2: Refine the algorithmic structure.

Step 3: Integrate suitably fragmented computational part into refined algorithmic skeleton.

Although being conceptually very clean and appealing on the first view, problems occur in all three steps of this programming methodology:

Selection of most suitable algorithmic skeleton (Step 1): Even the selection of very crude, qualitatively different algorithmic skeletons is only manageable for the experienced parallel programmer who has knowledge of basic parallel computational models, like the Master–Slave Model or the Farmer–Worker Model discussed above. Problems involved are how skeletons should be characterized and presented to the programmer, and how it can be ensured that he selects the correct or at least the most appropriate one.

Refine algorithmic structure (Step 2): Crude skeletons require a big gap to be closed between the prefabricated, reusable skeleton and the desired parallel distributed algorithm. To close this gap is conceptually a very difficult task. This necessarily requires a suitable environment which supports the refinement process.

Fragmentation and integration of computational part (Step 3): This necessarily requires a suitable environment which supports the optimal disintegration of the shared memory model–based sequential algorithm.

3.4.2 Improved Methodology for Parallel Distributed Programming

The improved methodology is a refinement of the basic 3–step methodology proposed by Cole and Burkhart, addressing and solving the problems indicated above. The key features of our methodology are as follows:

Foundation on hierarchically organized algorithmic skeletons. Each hierarchy level delivers algorithmic knowledge on a different level of abstraction/detail. Such an organization requires an ordering principle which can be deduced from practical experience in parallel distributed programming. This addresses the problem in step 2 of the basic methodology.

Careful and extensive guidance through the program development cycle, i.e., from the early design phase throughout the detailed design and the implementation phase. The existence of guiding methods is of central importance and compensates for the non–familiarity of computational scientists with most technical aspects of distributed systems and parallel distributed programming. It supports the identification of the most suitable grain–size. Continuous guidance also ensures improved exploitation of the performance relevant features of distributed systems. Guiding methods thus guarantee in general improved efficiency as compared to parallelizing compiler approach. This addresses the problem in step 1 of the basic methodology.

Selection methods incorporated and tailored to the needs of computational scientists. This also addresses the problem in step 1 of the basic methodology.

High–level problem description and design methods, abstracting away from purely implementation language dependent details. The improved methodology logically separates between the application– and system–oriented programmer. It matches more closely the natural requirements of computational scientists who avoid (and cannot afford) to get involved into too many technical aspects of computing. This addresses the problems in steps 2 and 3 of the basic methodology.

3.4.3 Foundation of Methodology

Most importantly, the user aspects and requirements have been derived from a profound knowledge of research characteristics of computational scientists and a thorough analysis of their needs. The algorithmic skeletons have been abstracted from practical experiences collected during the design and implementation of various parallel distributed algorithms addressing problems in the computational sciences. It represents practical experiences collected over the last two years. Since we believe that from a practical point of view, a methodology becomes especially powerful and handy if it is supported by a tool environment, tool support is anticipated.

3.5 The Programming Environment: Turning the Methodology into Practise

The key feature of the programming environment is interactivity, as suggested by the methodology. In more detail, we propose to successively solve the difficulties of parallel programming for distributed systems by means of combining the complementary strengths of human intelligence and machine–based knowledge. We want to achieve this by reinforcing the strengths of a human programmer who provides the global algorithmic picture by means of a powerful programming assistant, carefully watching and analyzing the activities of the programmer, making clever design suggestions on the grounds of his knowledge base, and asking intelligent questions all along the way from the demanding early design phase down to the completion of the implementation in a high–level programming language.

Structurally, we model the proposed program development environment as an *interactive programming assistant* that is supplemented with suitable *programming assistant interfaces*, completed by a *graphical user interface*. This environment provides tool support for the improved programming methodology described in the previous section. The programming assistant assures careful interactive guidance, from the early program design phase up to the implementation phase. The interfaces to the programming assistant support the algorithm design in a hierarchical fashion, based on formal languages. They thus naturally support the optimal fragmentation of the computational part of the parallel distributed algorithm, i.e., the shared memory model–based sequential algorithm. The graphical user interface allows to diagrammatically and textually represent the formal design languages.

To fully cope with the user requirements on the programming assistant, we envisage different types of interfaces that strongly vary with the application area: Assuming the message–passing paradigm, in the most general case a program for a distributed system can be modeled by a collection of communicating (software) processes. This process graph with processes and interprocess communications corresponding to the nodes and edges of the graph, respectively, represents the flow of control of the underlying application. Such a process graph may be highly irregular. A good example where such a representation is most appropriate is given by a real–time system. On the one hand the processes handle measuring devices to control temperature, pressure, etc. On the other hand they handle machine parts which operate as demanded by the measuring instruments. This kind of parallelism is called *algorithmic* parallelism, and is characterized by asynchronously operating processes with often small–grained parallelism and largely varying connectivities.

In contrast, many problems in numerically intensive computing belong to the class of data parallel applications which are characterized by highly regular process structures, operating synchronously or loosely–synchronously with medium to large grain–sizes and low– to medium–connectivity. In these kind of applications, rather than focusing on the flow of control, it is more appropriate to represent the data and their dependencies. Moreover, it is in general sufficient to model and design the behaviour of one instance of the entity of identical processes. Finally, in the extreme case of all those numerical methods based on iterative relaxation type of techniques, the computational scientist even would like to concentrate on some abstract specification of his computational stencil (pictorially representing the relaxation operation), expecting that both the design and the implementation of his problem is covered largely transparently by the programming environment.

To summarize, we shall provide three different interfaces to the programming assistant,

- An interface for the generic case of parallel distributed programming,
- An interface for the special case of data parallel programming, and
- An interface for the particular case of programming stencil based problems.

3.5.1 Specification of Functional Requirements

To practically support the improved methodology, the functionality of the programming assistant should be as follows:

Programming Assistant (PA). The programming assistant should have the capability to

- Interpret programming activities,
- Analyze programming activities,
- Provide design suggestions,
- Offer prefabricated algorithmic skeletons on different levels of abstraction,
- Call for additional information to be provided by the programmer,
- Provide support for automatic program synthesis.

Here automatic program synthesis techniques will continue where the high–level problem description and design methods that are part of our improved methodology end. They might finally lead to parallel distributed programming in a completely implementation language independent way.

Supplementary capabilities, ensuring real, efficiency preserving portability within the class of multicomputer systems include

- Support for optimization of individual communication steps (low level optimization of communication),
- Support for improved communication hiding, i.e., improved overlapping of communication and computation steps (higher level optimization of communication).

Generic Programming Assistant Interface (GPAI). This component interfaces the programming assistant for the generic case of parallel distributed programming. It supports the design and implementation of data parallel and also more general applications in a hierarchical fashion. It expects that the computational scientist can formulate his problem in a process graph representation.

The generic programming assistant interface should have the capability to

- Parse the generic process graph design language.

Data Modeling Programming Assistant Interface (DMPAI). This component interfaces the programming assistant and provides additional support for the particular case of programming data parallel problems. It supports the partitioning of large regular data structures. It expects that the computational scientist can model data structures and their decomposition.

The data modeling programming assistant interface should have the capability to

- Parse the data structure representation language.

Stencil Modeling Programming Assistant Interface (SMPAI). This component interfaces the programming assistant and provides special support for the particular case of programming stencil–based problems. It supports the specification of the computational grid and the data structure at each grid element (points, edges, etc.), the grid decomposition scheme, the boundary conditions of the grid, the data structure decomposition scheme (coloring), and the properties of the relaxation stencil (geometry, weights, etc.).

The stencil modeling programming assistant interface should have the capability to

- Parse the stencil specification language.

Graphical User Interface (GUI). The graphical user interface should have the capability to

- Graphically and textually represent the specification and design languages, and
- Graphically and textually edit the specification and design languages.

3.5.2 Design Considerations

Basic design considerations concern the design methods. After careful investigation, we decided to use a bottom–up approach for all components of the programming assistant PA. The three design languages that become available to the programmer via the three interfaces to the programming assistant will be based on the object–oriented programming paradigm. The design of the graphical user interface will also rely on the object–oriented approach to programming.

The bottom–up approach for the PA provides a natural framework to gradually extend the range of algorithmic generality covered by the programming environment. In a first phase, we shall focus on support for data parallel applications; later, this will be generalized successively according to the needs and the obtained feedback when the system is in productive use.

Founding the design languages on the object–oriented paradigm has the advantage that by providing carefully designed and implemented object class libraries we can hide most parts of the communication structure from the user by encapsulating the communication structures within the class methods. Additionally, performance enhancements and tuning for specific hardware platforms can be incorporated in the class libraries without any influence of the user program.

Object–oriented design for complex graphical user interfaces has already proven to be most adequate in the context of other projects with similar goals and thus needs no further justification.

Further design considerations include the following:

Algorithmic skeletons. For each place of an algorithmic skeleton where a sequential computational component can be integrated, an interface specification must be provided which defines the interaction between the algorithmic skeleton and the computational component. Depending on the abstraction level used for the skeleton, this interface specification can consist either of a formal specification, a function definition with a predefined name and parameter list, or predefined variable names which can be used within a code block.

Hierarchically organization of algorithmic skeletons. The important point is to choose an intelligent storage concept that ensures optimal reusability of human knowledge of parallel distributed programming, once a formal encapsulation of (part of) this knowledge in the skeletons has been achieved.

There exist three qualitatively distinct options:

- The library approach,
- The data base approach, and
- The knowledge–based approach.

The simple library approach has been suggested previously as a means to assist parallel programming. Although it includes some ordering principle, it is too crude and error prone for unexperienced parallel distributed programmers. A selection mechanism is missing completely. If the library is large, even an experienced user does might find the best skeleton for his algorithm. Most significantly, representation of 'deep' knowledge is not possible.

The more advanced data base approach contains more advanced ordering principles but has only a primitive selection mechanism which requires exact matching. There is no real representation of 'deep' knowledge possible.

The knowledge–based approach not only provides elaborate ordering principles and selection methods, but allows in particular genuine representation of 'deep' knowledge. The knowledge base can be naturally decomposed into two parts, a *static knowledge base*, storing the knowledge encapsulated in the algorithmic skeletons, and the *dynamic knowledge base*, storing the deep, conceptual knowledge together with practical experience of parallel distributed programming. In particular, the knowledge–based approach also allows meta–representation of algorithmic skeletons by means of algorithmic skeleton fragments combined with rules from the dynamic knowledge base.

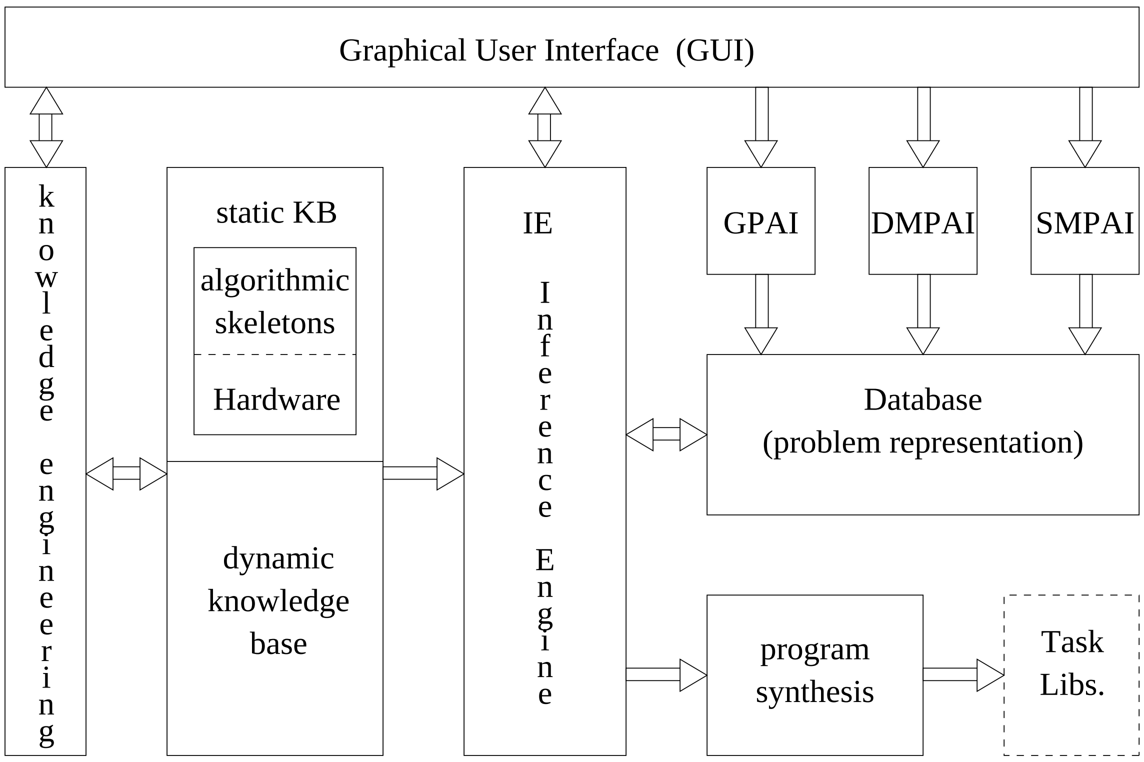

Figure 1: Design of the Program Development Environment, consisting of the programming assistant PA, the three interfaces to the PA, and the graphical user interface. The PA itself is structured into the data base, the knowledge base, the inference engine, the knowledge engineering interface, and the code generator.

Comparing the three different options, the knowledge–based storage concept is certainly the most advanced and powerful one. It is therefore the system of our choice. An important point of any such system is how the required formal knowledge is acquired. We shall pay particular attention to it (cp. sect. 3.8). To cope with the problem of knowledge engineering, a knowledge engineering interface will be incorporated into the over–all design of the system. To acquire formal knowledge, for the time being, we anticipate expert interviews as the only approach of practical importance. This method is known for its subjectivity and often incomplete exploitation of the available human knowledge. Less subjective and more complete methods like automated learning might be investigated later. Given the knowledge engineering interface, a very significant design feature of our system is that experienced users can enlarge the static and dynamic knowledge base by themselves. They can thus adapt and tune the entire system according to their specific needs.

Guiding mechanism. One of the advantageous features of the knowledge–based approach chosen for the intelligent storage concept is that the guiding mechanism required by our programming methodology is naturally provided.

Selection methods. They can also be incorporated into the entire system by natural extension of the intelligent storage concept. Powerful selection methods are absolutely essential and necessarily required by the existence of algorithmic skeletons on several levels of abstraction/detail and their hierarchical organization.

Problem description and design methods. The question is whether problem description and design methods should be based exclusively on textual or graphical representations, or a combination of both. Careful investigation of the issues involved showed that both have their advantages and appear to be more naturally and powerful in different phases of problem description and design. The system will hence rely on a combination of both representation techniques.

Interfaces to the programming assistant. The general programming assistant interface GPAI will be developed in a step–wise fashion. At the beginning, only methods assisting problems in the domain of numerically intensive computing will be supported. This phase will be followed by successive generalization to the generic case. This evolution of the GPAI will go hand in hand with the successive enlargement of the knowledge base of the programming assistant.

The overall design reflecting the system specification is illustrated in Fig. 1.

3.6 Operation

In this section we present a short description how the programming assistant and the other components of the programming environment will be used in practise. For the time being, we assume that the PA already has been 'filled' with static and dynamic knowledge; how we are going to do this is explained later in sect. 3.8. According to the programming problem at hand, the programmer selects the most suitable programming assistant interface, i.e., the SMPAI for all stencil–based problems, the DMPAI for all data parallel problems, and the GPAI for all other problems. The programmer then specifies his problem in a diagrammatic way via the graphical user interface using the method that is supported by the selected programming assistant interface and its corresponding problem design language. Then the programming assistant interface transforms the design of the problem solution into a common (internal) problem representation. Once this has been achieved, the inference engine tries to map the internal problem representation to an algorithmic skeleton contained in the static knowledge base of the programming assistant. If the problem representation is incomplete, or if the PA's dynamic knowledge is insufficient to establish the map, the inference engine queries the programmer for further information. If the inference engine finds a suitable algorithmic skeleton with the characteristics defined in the problem representation, it returns it to the programmer. If no algorithmic skeleton can be found, the inference engine tries to create one with the help of the rules defined in the dynamic knowledge base. If no skeleton can be found or created, the programmer can specify a new one and can insert the necessary entries and rules in the static and dynamic knowledge base via the knowledge engineering interface. Once the design phase is finished and the algorithmic skeleton is supplemented with computational components, the program synthesis component of the PA supports the actual implementation and code generation phase.

3.7 Conceptual Advantages

There are several significant advantages that can be related to our specific approach towards a powerful programming environment for parallel distributed programming. The knowledge–based approach guarantees successive and rapid extensibility of the entire system. It provides an intelligent medium to store and conserve algorithmic knowledge in a reusable and, most importantly, implementation language independent way. Reusability of (even abstract) algorithmic knowledge itself is ensured by sufficient guiding and selection methods also for the unexperienced parallel programmer such as the typical computational scientist at present. A nice feature of the programming assistant is that it conserves algorithmic knowledge on a high abstraction level and actively supports its efficient exploitation.

3.8 Realization

There are two necessary requirements to realize such an ambitious project: close familiarity with the needs and programming knowledge of computational scientists and extensive experience in parallel distributed programming. The former becomes particularly important for the design of the interfaces to the programming assistant and the graphical user interface, because these components define the look and feel of the entire programming environment. The latter is absolutely mandatory to fill the static and dynamic knowledge base. Both forms of knowledge and expertise have been necessary on the whole to even design the entire system. We believe that our interdisciplinary research team fully accomplishes both requirements.

Our strategy to realize the system is as follows. To begin with, we started to provide the required functionality for stencil–based problems. These are toy problems of practical importance since they cover an important class of applications. From the technical point of view, they have low demands on the capabilities of the knowledge–based system. They thus serve simultaneously as a learning environment for the more difficult steps to be covered later for more difficult types of applications. This approach also ensures fast usability by computational scientists, i.e., short 'time–to–market' of our system. In turn, valuable information on the usefulness of the system under conditions of practical importance is quickly fed back to the developers of the system.

Specifically, in a first phase, we shall implement a set of algorithmic skeletons for solving two–dimensional grid–type problems with stencil–based operations. This amounts to provide programming support for finite difference methods in two dimensions. At the time of writing this document, activities include the development of formal stencil specification methods [15] and the implementation of a prototype of the programming assistant. We plan to finish this first phase with the preparation of a demonstrator application within the next four months. The demonstrator application should serve to demonstrate the functionality of the entire system. Once this has been achieved, we shall extend the capabilities to cover also three–dimensional problems.

As already mentioned in sect. 3.5.2, the filling procedure of the knowledge base deserves special attention. We shall extend the static and dynamic knowledge base in a step–wise fashion. In a first step, we intend to evaluate the experience gained in realizing the class of toy problems by means of the programming assistant prototype mentioned above. In further steps, we shall successively exploit and transform our algorithmic knowledge collected over the last years into a suitable formal, machine–accessible form. It is important to note that easy extensibility is ensured by the very nature of the knowledge–based concept underlying our system.

In the realization phase of the system, particular attention will be paid to intermediate proofs of the usefulness of the different components under realistic conditions. Testing will benefit from the fact that the research work will be conducted at the Swiss Scientific Computing Center *Centro Svizzero di Calcolo Scientifico* (CSCS). Selected users of the CSCS environment will provide valuable information on the practical usefulness of our improved programming methodology and its realization by means of the programming environment both described in this paper. We believe that CSCS users will also make a significant contribution to further improvements.

4 Summary and Conclusions

Computational scientists present an important user group for the rapidly developing parallel distributed computer architectures. Despite of the still increasing investments made by parallel computer vendors and research institutes into software that efficiently allows to exploit the promising performance potential of these systems, existing programming support does not address the specific needs of computational scientists. In this paper, we have addressed this problem. We have presented in detail a model of parallel distributed programs, and a programming methodology based on it. This methodology provides the conceptual foundation of a corresponding programming environment that is also discussed in the paper. After presenting the specification of the functional requirements of such an environment and fundamental design considerations, we have discussed how the system can be used in practise. A section on the conceptual advantages and a detailed plan how we are currently realizing the project completes the paper.

The programming environment and its underlying programming methodology will become part of the SPADE system, an integrated **S**cientific **P**rogram and **A**pplication **D**evelopment **E**nvironment. It supplements SPADE's application development environment ADE and run–time environment RTE (where the latter is currently only realized rudimentarily and in a non–portable way). With the help of the programming environment introduced in this paper, SPADE will thus provide support for the complete application development cycle.

An interesting feature of our approach to parallel distributed programming is that reusability is addressed on several levels of abstraction. It is realized on the conceptual level by means of abstract algorithmic knowledge, stored in the form of rules in the dynamic knowledge base. And it is also realized on the meta–algorithm level by means of algorithmic skeletons, stored in the static knowledge base. It can be imagined that this knowledge can also be used to *teach* parallel distributed algorithm design and programming in general. Thus our approach to parallel distributed programming does not only support the programming of parallel distributed systems, but also assists education in this field of key importance for the future of computational science.

Acknowledgement. The work presented in this paper is part of the SPADE project which is supported by Grants 20–31057.91 and 20–33949.92 from the Swiss National Science Foundation.

References

[1] V. S. Sunderam. PVM: A Frameworks for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[2] J. Boyle, R. Butler, B. Glickfeld, T. Disz, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. Portable Programs for Parallel Processors. *to be published*, 1987.

[3] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD Macros in Fortran for Portable Parallel Programming and their Implementation on the Intel iPSC/2. *Parallel Computing*, 15, 1990.

[4] ParaSoft Corporation, Pasadena, CA 91107, USA. *Express User's Guide*, 3.0 edition, 1990.

[5] Moshe Braner. *Trollius User's Manual.* Cornell Theory Center, Ithaca, NY, 1988.

[6] Lyndon Clarke and Greg Wilson. Tiny: An Efficient Routing Harness for the INMOS Transputer. Technical report, Edinburgh Parallel Computing Center, Edinburgh, 1990.

[7] T. Bemmerl. The TOPSYS Architecture. In H. Burkhart, editor, *Proceedings CONPAR 90 - VAPP IV*, volume 457 of *Lecture Notes in Computer Science*, pages 732–743. Springer, 1990.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Krem, C-W. Tseng, and M-Y. Wu. FORTRAN D Language Specification. Technical report, Rice University, Department of Computer Science, January 1992.

[9] Hans P. Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley (ACM), 1990. ISBN 0-201-17560-6.

[10] High Performance Fortran Forum. High Performance Fortran Language Specification. Draft, Version 0.4, November 1992.

[11] Charles Koelbel, Piyush Mehrotra, Joel Saltz, and Harry Berryman. Parallel Loops on Distributed Machines. *Proceedings of the Fifth Distributed Memory Computing Conference*, II, Architecture Software Tools, and Other General Issues:1097–1104, April 1990.

[12] J.J. Dongarra. Library Issues in Open Systems: Portability, Scalability. In *Highly Parallel Computing Systems*, IBM Europe Institute 1992, 1992.

[13] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computation. The MIT Press, Cambridge, Massachusetts, 1989. ISBN 0-262-53086-4.

[14] H. Burkhart. Einfaches Programmieren paralleler Systeme mittels Algorithmenklassen. Swiss National Science Foundation Project SNF 21-31171.91.

[15] M. Roth. Generation of Algorithmic Skeletons from Stencil Specifications. Master's thesis, IAM, University of Bern, 1993.