PROMOTER: A High-Level, Object-Parallel Programming Language [*]

W. K. Giloi, M. Kessler, A. Schramm
RWCP Massively Parallel Systems GMD Laboratory
Berlin, Germany
w.giloi@computer.org {mk,schramm}@first.gmd.de

Abstract

The superior performance and cost-effectiveness of scalable, distributed memory parallel computers will only then become generally exploitable if the programming difficulties with such machines are overcome. We see the ultimate solution in high-level programming models and appropriate parallelizing compilers that allow the user to formulate a parallel program in terms of application-specific concepts, while low-level issues such as optimal data distribution and coordination of the parallel threads are handled by the compiler. High Performance Fortran (HPF) is a major step in that direction; however, HPF still lacks in the generality of computing domains needed to treat other than regular, data-parallel numerical applications. A more flexible and more abstract programming language for regular and irregular object-parallel applications is PROMOTER. PROMOTER allows the user to program for an application-oriented abstract machine rather than for particular architecture. The wide semantic gap between the abstract machine and the concrete message-passing architecture is closed by the compiler. Hence, the issues of data distribution, communication, and coordination (thread scheduling) are hidden from the user. The paper presents the underlying concepts of PROMOTER and the corresponding language concepts. The PROMOTER compiler translates the parallel program written in terms of distributed types into parallel threads and maps those optimally onto the nodes of the physical machine. The language constructs and their use, the tasks of the compiler, and the challenges encountered in its implementation are discussed.

Keywords: *Distributed memory architecture, high-level programming model, parallelizing compiler, algebraic domain specification, distributed types, coordination schemes, mapping, aggregation.*

[*]This research is supported by the *Real World Computing Partnership*, Japan.

1 Introduction

Distributed memory parallel computers have the potential of providing any desired performance at maximum scalability and cost-effectiveness. These properties will eventually make parallel machines the standard architecture, but only if we succeed in overcoming the parallel programming hurdle.

Of the currently existing programming paradigms for parallel machines, *message-passing* and *shared-memory*, message-passing exhibits the wider gap between the purely algorithmic view of the application and the executing machine, burdening the programmer with the tasks of data distribution, inter-thread communication, and coordination of thread execution. But even in the simpler shared-memory model, data distribution remains as optimization issue, and synchronization is still needed to coordinate the access to shared objects. Moreover, there is a gap between model and architecture that makes the implementation of the shared-memory model on a distributed memory architecture costly, either in terms of hardware expenses in the case of distributed shared memory or software overhead in the case of virtual shared memory.

Therefore, we believe that the ultimate solution of the parallel programming problem will be higher-level programming models and appropriate parallelizing compilers that allows the user to formulate a parallel program in terms of application-specific concepts, while the low-level mechanization of the tasks of finding optimal data distributions and coordinating the parallel threads is handled by the compiler. A first major step in this direction is High Performance Fortran (HPF[1]); however, HPF still lacks the generality of computing domain definition needed to treat other than only regular, data-parallel numerical applications.

A more flexible and more abstract programming language for regular and irregular object-parallel applications, PROMOTER, is discussed in the paper. Object-parallelism is understood as follows: Given arbitrary data structure objects consisting of a potentially large number of data points, then object parallelism is the processing of the data structure objects such that the transformations of a number of data points are performed in one step. The data points of an object and their mutual interdependencies are structured into problem-specific topologies. Object parallelism exhibits the following specific characteristics, which can be exploited to facilitate parallel programming:

1. In contrast to the common understanding of data parallelism, the data structures in object parallelism need not be arrays, and there may be a large number of parallel threads of control. Having a multitude of threads, however, does not necessarily mean their unrestricted asynchronous execution, as is generally implied by the MIMD computation model. In PROMOTER, the individual scheduling of threads is replaced by a global, collective coordination; however, in contrast to the commonly used SPMD computation model, this is carried out with a high degree of local autonomy of thread execution.

2. At the appropriate level of abstraction the multitude of data points and their parallel processing exhibits spatial homogeneity. This allows for the introduction of distributed types (objects and methods).

3. There exists some temporal coordination of the parallel computation steps. E.g., the data points may be collectively subjected to the same number of iterations or, in general, may proceed collectively in simulation time, yet there may as well be meaningful variations of synchronization requirements such as wave fronts or other synchronization patterns.

Section 2 of the paper elaborates on the basic concepts of PROMOTER, Section 3 deals with the mechanisms by which these concepts are implemented, and Section 4 discusses the task of the automatically parallelizing and optimizing PROMOTER compiler, as well as our approach to implementing the compiler. The Conclusion considers briefly the relationship to other developments and the future refinements of the PROMOTER system.

2 PROMOTER Basic Concepts

2.1 Aims and scope of the PROMOTER programming model

PROMOTER (programming model to enable real-world computing) is a programming model and, based on it, a programming system, PROMOTER, has been designed

1. to enable its user to program massively parallel applications at a high level of abstraction

2. for a large variety of applications and solution algorithms,

3. making the underlying application-specific graph structures explicit to achieve an optimal locality-preserving mapping.

To make the system sufficiently general, the classes of massively parallel applications that can be handled by PROMOTER are not restricted to regular problems but encompass also highly irregular and/or dynamic structures.

PROMOTER allows for a problem-oriented description of object-parallel algorithms, which are then implemented by the compiler on a given distributed-memory machine. Hence, PROMOTER closes the semantic gap between message-passing architectures and parallel applications. PROMOTER aims at being a high-level programming paradigm that is applicable to the entire spectrum of object-parallel numerical applications. This goes far beyond PDE solvers or applications of linear algebra, extending its scope to highly irregular problems, e.g., finite element computation or neural networks.

One key to PROMOTER's linguistic simplicity lies in the exploitation of regularity in the applications wherever possible. To this end, an important notion of PROMOTER is the dichotomy of static global homogeneity versus dynamic local autonomy. The program text describes the level at which the element types and the parallel operations look homogeneous, while local differentiation may evolve at run time. This balance between two complementary aspects is expressed by local control flow autonomy and object-oriented polymorphism.

2.2 Raising the level of abstraction of message-passing programming

PROMOTER avoids the problems of message-passing programming by orthogonalizing the steps of domain specification, communication, and coordination (synchronization). Thus, programming is simplified as well as automatic parallelization by the compiler. The code to be executed by the parallel threads reflects only the numerical algorithm and is totally free of communication constructs.

2.3 Object-parallel execution

In PROMOTER, data are arranged in the form of problem-specific (virtual) topologies. Object parallelism exhibits by nature a certain degree of temporal coordination of the parallel computation steps. Thus we have a global uniformity "in the large", from which deviations may be given on a finer time scale. While the replicated method is called in all points, there may be local differentiations at run time in the method body.

The parallel steps are globally coordinated, i.e., they are embedded in a common global control flow in which they appear as object-parallel statements and expressions. The common global control flow may consist of any combination of parallel atomic steps on distributed variables as defined above such as loops, operations on non-distributed variables, reduction expressions, etc. Hence, a coordinated behaviour of all data points is achieved.

3 PROMOTER Mechanisms and Language Components

3.1 Distributed data types

To meet the goals defined above, the constructs of the PROMOTER programming language must enable the user to express spatial regularity and temporal coordination of the application algorithms. This is provided by introducing distributed data types and parallel operations on them. A distributed data type consists of objects (data structures) of an appropriately defined discrete topology and of methods (functions) with built-in replication semantics. Replication is automatically performed by the system over the given topologies. Defining the appropriate topology for the application becomes the main intellectual effort of the programmer, while the compiler takes care of the details of an optimized execution.

Every element of a distributed object resides in a logical address space of its own; there are no pointers between elements of different objects or different elements of the same objects.

3.2 Communication

Communication is viewed in each domain point as the observation of the states of some other points. As a consequence of the notion of distributed state variables, "observation of state" means a "call by value" of certain elements of distributed variables. Hence, the programmer need not invoke any message-passing constructs explicitly. This approach offers the simplicity of the shared memory model, yet there is a significant difference. In the shared memory model data access is by reference, i.e., through pointers, whereas in PROMOTER the notion of pointers as vehicle of inter-node communication does not exist. Instead, copies or clones, respectively, of non-local data are obtained by special language constructs that appear as input operands of data parallel operations. To this end, the programmer declares a communication topology that determines the copying procedure. There are no side effects, which makes this approach simpler and safer than shared variables.

The PROMOTER communication scheme of simply selecting values of distributed variables without any need for explicit synchronization reduces communication to the execution of communication expressions. PROMOTER's "communication product" is a generalization of vector-matrix multiplication to arbitrary topologies. For example, a vector of values observed from some other points can be obtained by an extended vector-matrix multiplication where the vector operand is a distributed variable, the matrix is a Boolean "connectivity matrix" that selects the values to be read, and the result is the vector of selected values. Communications may as well be many-to-one, in which case the result is obtained by reduction. Arithmetic communication matrices are permitted to extend the concept to linear operators. For example, in the simulation of a neural network the weighted inputs of a neuron may be obtained by one single communication product expression.

3.3 Coordination

In contrast to the message-passing model, thread coordination is separated from communication, carried out simply by specifying a coordination scheme which is automatically executed by the system. Existing coordination schemes are: (i) lock-step, (ii) wave fronts, (iii) asynchronous iteration, and (iv) chaotic iteration.

Regardless of the coordination scheme(s) employed, communication is either explicit or non-existent, but never implicit via dereferencing. Therefore there is no need of additional global barriers of any kind.

3.4 Main components of the PROMOTER language

It would exceed the scope of this paper to present a detailed specification of the PROMOTER language. We will mention only the salient points of the language. PROMOTER hides the communication mechanisms (sends, receives) from the programmer. Hence, the program text concerns only computation. For the mechanization of the concepts outlined above, the PROMOTER language provides the following components for an algorithmic description.

- *Domains and their topology.* The domains of computation are discrete point sets with a user-declared topology. Topologies are subsets of $\mathbf{Z}^n$, with $\mathbf{Z}$ being the set of integer numbers; in PROMOTER they are declared in terms of index expressions. Domains may be regular or irregular, dense or sparse, static or dynamic.
- *Distributed types and variables.* Distributed variables are data structures with a user-declared topology defining the domain of computation. Their role is to hold the states of object-parallel computations. Distributed variables are built of objects or sub-objects of distributed types. The elements of a distributed type are accessed by indexing. At a first glance, this might suggest that distributed types are nothing but distributed arrays. However, there are significant differences given by the following features that make distributed types significantly more general.
    1. Distributed types may have arbitrary and even dynamic topologies, rather than only rectangular ones.
    2. Distributed types have a built-in replication semantics. Operations on distributed objects are intrinsically parallel. There is no need for any analysis in order to derive the parallelization. Assignments to distributed variables have replication semantics as well; i.e., every element of the target is assigned the value of the corresponding source element, possibly after a conversion of the source value. The replication space is the dynamic topology of the left-hand side.
    3. Distributed types have point-wise local address spaces; there are no pointers between distinct data points.

(A) Declaration

```
topology My_top: 1:8, 1:8
  {$i, $j |: j <= i+1;
   };
```

(B) Topology

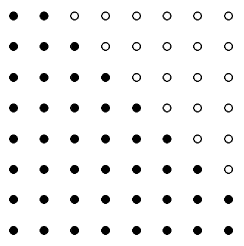

Figure 1: A simple topology declaration with constraints and the resulting topology

The PROMOTER language is an extension of the object-oriented language C++. It provides all the constructs needed to specify domain and communication topologies, declare distributed variables and their types, and select the coordination scheme. The governing goal of the language design was to make the formulation of an object-parallel program as easy and straightforward as possible, eliminating the semantic gap between the procedural patterns of the applied algorithms and the manner in which the message-passing architecture works. Specifically, the language design pursued the goal of providing descriptive means for domain specification that are as general and flexible as possible, to obtain a programming system that caters to the needs of as large a variety of object-parallel applications as possible.

3.5 Topology declarations

Note that topologies are application-oriented and abstract, that is, machine independent. Irregular structures may be generated by adding constraints to the index expressions or by specifying a new topology as the union of already existing topologies. Figure 1 shows the flavor of domain declaration by a simple example of an irregularly structured topology.

A topology declaration may be given in terms of formal parameters whose values are determined at run time, thus allowing for the creation of dynamic topologies. There exist several possibilities to declare objects with dynamic topologies. These are with increasingly dynamic behavior:

1. *Non-parametrized (static) topologies*—the index space of non-parametrized topologies is fully determined at compile time.

2. *Parametrized topologies*—the index space of parametrized topologies can be determined at the time of an object declaration or creation and remains fixed throughout the object's lifetime.

3. *Dynamic unions of parametrized topologies*—the instances of a union topology may vary at the granularity of the parametrized topologies they consist of.

4. *Point-granular dynamic unions*—the instances of a point-granular topology may vary at the granularity of the single points they consist of.

4 The PROMOTER Compiler

4.1 The role of the compiler

Figure 2 illustrates the PROMOTER approach by the example of an object-parallel computation in the lock-step mode of execution. At the program level, parallel computation consists in a sequence of applications of replicated operations on distributed variables. Hence, the program entities are parallel operations on distributed objects. This language model is converted by the compiler into the parallel execution of threads, which thus become the entities of execution.

The resulting large amount of fine-grained threads are distributed by the compiler over the nodes of the physical machine such that the overall cost of communication is minimized. Subsequently, as a further optimization step the threads in each node may be aggregated into one coarse-grained "operating system process" which then is ultimately executed.

In the case of dynamic topologies, the compiler will also have to insert code for dynamic load balancing. Both, static mapping and dynamic load balancing, are supported by the visibility of the application graph structures and possibly their dynamicity. In many irregular and/or dynamic applications, the graph structures are just subgraphs of a certain (possibly infinite) *static regular supergraph*. PROMOTER allows to express and exploit this fact, e.g., by generating code for incremental load balancing that is a compile-time generated partial evaluation of a more general method.

4.2 Mapping

A mapping pass maps the large number of threads stemming from the PROMOTER model of applying replicated methods on distributed variables onto the nodes of the target machine. The aim of the mapping

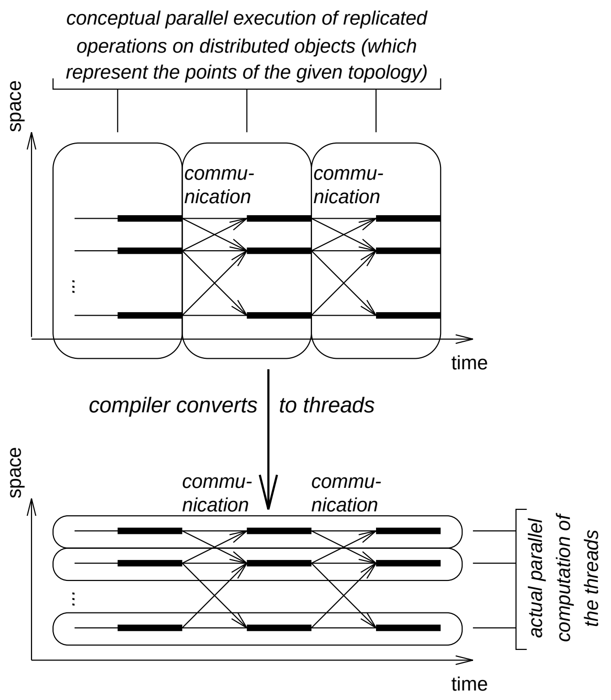
conceptual parallel execution of replicated operations on distributed objects (which represent the points of the given topology)
space
commu-nication
commu-nication
time
compiler converts
to threads
space
commu-nication
commu-nication
actual parallel computation of the threads
time

Figure 2: PROMOTER programming in terms of spatial entities which the compiler transforms into threads

is an even workload distribution and the minimization of the communication overhead. To this end, the graph representing all threads must be mapped onto another graph representing the physical processing nodes. In both graphs the edges are marked by weights representing the frequency of communication and latency of the communication channels. This graph-theoretical problem is NP-hard, thus, it is solved by appropriate heuristics, e.g., Recursive Spectral Bisection (see for example [5]).

4.3 Cloning

A main feature of the PROMOTER computation model is the non-existence of global pointers. Consequently, pointers to local objects which naturally are permitted in the C++ code lose their meaning when passed to other instances of distributed objects. To eliminate that problem, PROMOTER incorporates a facility to generate structural identical objects by cloning objects that contain pointers (this is the difference between cloning and copying). Cloning is automatically inserted by the compiler whenever necessary. However, the user may also explicitly clone an object by calling the cloning intrinsic.

5 Comparisons and Conclusion

First we compare PROMOTER with a few other programming models and highlight the differences.

VSM: Languages featuring a single global address space make little effort to express spatial structures (the problem graphs) explicitly. Instead, they naturally lead to techniques like pointers and indirect indexing. With these, a notion of locality evolves only implicitly and at run time, which impairs the accomplishment of a locality-preserving data distribution. On machines with a physically distributed memory, a Virtually Shared Memory (VSM) is a way to close the paradigmatic gap. PROMOTER, on the other hand, expresses the spatial structures of an application explicitly and thus allows for a direct generation of a message-passing translation.

HPF: The conceptual differences between PROMOTER and HPF become evident especially when it comes to irregular applications, a field which is admittedly problematic in HPF. HPF provides only "regular" data structures (i.e., arrays with rectangular index spaces), whose mapping is controlled by the user. Communication patterns are dispersed over index expressions and a few simple array intrinsics. Irregular patterns can be expressed only dynamically by indirect indexing, exhibiting the same disadvantage as global pointers and virtual shared memory. PROMOTER, on the other hand, permits the declaration and dynamic manipulation of arbitrary spatial structures, where communication structures are expressed as a whole by (possibly sparse) subsets of the Cartesian product of the respective source and target index spaces. Entire linear operators can directly be expressed by this concept. The resulting application graph structures thereby expressed provide the starting point for an automatized locality-preserving mapping of the problem-specific spatial structures onto the physical machine. Finally, PROMOTER has some concepts that do not exist in HPF, e.g., a variety of so-called coordination schemes for parallel operations, and constructs for working with partitionings and coverings.

Data-parallel models with recursive data types (e.g., *CDT*[6], *NESL*[2], *Powerlist*[4]): There is no sharp conceptual separation between PROMOTER and programming models with recursive data types; after all, recursive types are possible in PROMOTER as well. The actual difference is that the models of the latter kind aim at deriving their expressive power from type recursion and regard the leaf types as "basic", while PROMOTER puts its expressive power in its basic type constructor and considers type recursion to be the exception. This depreciation of recursive types is a consequence of the requirements of the envisaged applications: (i) These applications often need *several* graph structures (edge sets) on the same data point sets, and distinct operations may need different and even incompatible hierarchic node set decompositions; (ii) the underlying graph structures are often most conveniently expressed by the means of index arithmetic. Both aspects conflict with recursive types. PROMOTER takes care of these aspects by the separation of the spatial structures of operations from the distributed types and by expressing spatial structures by index sets that are arbitrary subsets of $\mathbf{Z}^n$ of appropriate dimension.

Message-passing models (e.g., *MPI*, *PVM*): Models that feature processes with explicit message passing are usually considered as low-level and circumstantial in the field of parallel programming. PROMOTER abstracts from this level of explicit data partitioning and message passing. One of those models could very well serve as an intermediate or target level in the translation process, though.

Conclusion: PROMOTER does indeed make the spatial (and temporal) structures of parallel application explicit at a level that is sufficiently abstract to be manageable for the user, thus permitting an automatized mapping and a direct generation of a message-passing translation for distributed-memory platforms.

References

[1] High Performance Fortran Forum. High Performance Fortran Language Specification. Scientific Programming 2:1–170, 1993.

[2] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Carnegie-Mellon University, Pittsburgh/PA, 1993.

[3] W. K. Giloi and A. Schramm. PROMOTER, an application-oriented programming model for massive parallelism. In *Proceedings of the Massively Parallel Programming Models Working Conference*, pages 198–205. IEEE, 1993.

[4] J. Misra. Powerlist: A Structure for Parallel Recursion. ACM Transactions on Programming Languages and Systems 16(6):1737–1767, Nov. 1994.

[5] H. D. Simon. Partitioning of unstructured problems for parallel processing. Computing Systems in Engineering 2(2/3):135–148, 1991.

[6] D. B. Skillicorn. Practical Parallel Computation, II. Categorical Data Types. External technical report, Queen's University, Kingston, Ontario, 1991.