CSCC'99 PLENARY LECTURE
Recent Advances in Parallel Programming Languages and Tools

DOMENICO TALIA
ISI-CNR
c/o Deis, UNICAL, 87036 Rende (CS)
ITALY

Abstract: - Parallel programming models, languages, and tools are the basic instruments for the design and implementation of high performance applications on scalable computer architectures composed of a collection of processors (multiprocessors or multicomputers). In the latest years several high-level languages and software tools have been designed for the programming of parallel software. This paper introduces and discusses the recent advances in the area of parallel and distributed programming tools and languages. The paper describes different parallel programming languages and tools that reflect different parallel computation models. It introduces the design goals and issues of parallel programming models and languages belonging to the following classes: message-based languages, shared-space based languages, data-parallel languages, parallel toolkits, parallel declarative languages, parallel object-oriented languages, and parallel skeleton languages.

Example tools and languages in each class, such as HPF, Linda, Java, OpenMP, PVM, MPI, Parallel C++, Sisal, Orca, Mentat, SkieCL, BSP and other languages are described in some detail, and their features for high performance applications implementation are discussed. Finally, we discuss future directions of research and development in the parallel programming area with a special attention to novel approaches based on high-level programming structures that make transparent to the users the architectural details of parallel computing machines.

Key-Words: - parallel computing, parallel programming languages, software tools, concurrent programming.
3rd World Multiconference on CIRCUITS, SYSTEMS, COMMUNICATIONS AND COMPUTERS, pp. 32-42

1 Introduction

During the latest years parallel computers ranging from tens to thousands of processing elements became commercially available. They continue to gain recognition as powerful engines in scientific research, information management, and engineering applications. This trend is driven by parallel programming languages and tools that contribute to make parallel computers useful in supporting a broad range of applications.

Several models and languages have been designed and implemented to allow the design and development of applications on parallel computers. Parallel programming languages (called also *concurrent languages*) allow the design of parallel algorithms as a set of concurrent actions mapped onto different computing elements [1]. The cooperation between two or more actions can be performed in many ways according to the selected language. The design of programming languages and software tools for parallel computers is essential for wide diffusion and efficient utilization of these novel architectures [2]. High-level languages decrease both the design time and the execution time of parallel applications, and make it easier for new users to approach parallel computers.

The aim of this paper is to give an overview of the major parallel programming paradigms designed in the latest decade to program parallel computers. The paper discusses a set of representative languages and tools designed to support different models of parallelism. It discusses both parallel languages currently used to develop parallel applications in many areas from numerical to symbolic computing and novel parallel programming languages that will be used to program parallel computers in the near future.

Languages are classified into five main classes according to the paradigm they use to express parallelism. Next sections describe languages that follow the imperative paradigm and that express parallelism at the process or statement level. In particular, section 2 discusses languages based on the shared-memory model. These programming

languages present a view of memory as if it is shared, although the implementation may or may not be. Processes communicate and synchronize through the use of shared data. Section 3 discusses concurrent languages based on the distributed-memory model. A distributed concurrent program consists of a set of processes, located on one or many computers, and cooperating by message passing. This paradigm reflects the model of distributed-memory architectures composed of a set of processors connected by a communication network.

In Section 4, parallel object-oriented languages are discussed. Objects and parallelism can be integrated since the modularity of objects makes them a natural unit for parallel execution. Section 5 examines both parallel functional and logic programming languages. Parallel functional languages express a fine-grain parallelism at the level of expressions whereas concurrent logic languages express parallelism at the clauses level. The aim of concurrent logic languages is the exploitation of the parallelism inside logic programs by means of a parallel proof strategy. They offer another declarative approach to programming parallel computers.

Finally, section 7 presents a collection of more innovative approaches to parallel programming that use a modular approach in designing parallel programs. These languages are designed with stronger semantics directed towards software construction and correctness. Some of these may be the programming languages to be used for the development of parallel applications in the next century.

2 Shared Memory Languages

This class of parallel languages use the shared-memory model that is implemented by parallel computers composed by several processors that share a single memory space. The concept of shared memory is a useful way to decouple program control flow issues from issues of data mapping, communication, and synchronization. Physical shared memory is probably difficult to provide on massively parallel architectures, but it is a useful abstraction, even if the implementation it hides is distributed. Significant parallel languages based on the shared-memory models are Orca, Linda, OpenMP, Java, Pthreads, Opus, SDL and Ease.

One way to make programming easier is to use techniques adapted from operating systems to enclose accesses to shared data in critical sections. These can then be further modified to make them light-weight enough to use for single memory references. Some machines provide *test&op* instructions in the hardware.

Another approach is to provide a high-level abstraction of shared memory. One way to do this is called *virtual shared memory*. The programming language presents a view of memory as if it is shared, but the implementation may or may not be. The goal of such approaches is to emulate shared memory well enough that the same number of messages travel around the system when a program executes as would have traveled if the program had been written to pass messages explicitly. In other words, the emulation of shared memory imposes no extra message traffic.

One way to emulate shared memory is to extend techniques for cache coherence in multiprocessors to software memory coherence. This involves weakening the implementation semantics of coherence as much as possible to make the problem tractable, and then managing memory units at the operating system level. The other way is to build a system based on a useful set of sharing primitives. This is the approach used by the Orca language.

Orca. The Orca language is based on a hierarchically structured set of abstractions [3]. At the lowest level, reliable broadcast is the basic primitive so that writes to a replicated structure can rapidly take effect throughout a system. At the next level of abstraction, shared data are encapsulated in passive objects that are replicated throughout the system. Orca itself provides an object-based language to create and manage objects. Rather than a strict coherence, Orca provides serializability: if several operations execute concurrently on an object, they affect the object as if they were executed serially in same order. Orca has only been implemented on small parallel machines, and its performance depends on the relative infrequency of writes compared to reads, so its scalable performance is a question mark.

Linda. A second orthogonal approach to providing high level abstractions of shared memory is embodied in Linda, which provides an associative memory abstraction called tuple space [4]. Threads communicate with each other only by placing tuples in and removing tuples from this shared associative memory. As a result, programs written in any imperative language can be augmented with tuple space operations to create a new parallel programming language. These languages are called *coordination languages* because the tuple space abstraction coordinates, but is orthogonal to, the computation activities.

In Linda, tuple space is accessed by four actions: one to place a tuple in tuple space, *out(T)*, two to remove a tuple from tuple space, *in(T)* and *rd(T)*, one by copying and the other destructively, and one which evaluates its components before storing the results in tuple space (allowing the creation of new processes), *eval(T)*. The efficient implementation of tuple space depends on distinguishing tuples by size and component types at compile time, and compiling them to message passing whenever the source and destination can be uniquely identified, and to hash tables when they cannot. In distributed-memory implementations, the use of two messages per tuple space access is claimed, which is an acceptable overhead.

There are two difficulties with Linda, which have been addressed by two extensions to it: SDL and Ease. The first is that a single, shared, associative memory does not provide any way to structure the processes that use it, so that Linda programs have no natural higher-level structure. The second is that, as programs get larger, the lack of scoping in tuple space makes the optimizations of tuple space access described above less and less efficient. For example, two sets of communications in different parts of a program may, by coincidence, use tuples with the same type signature. They will tend to be implemented in the same hash table and their accesses will interfere.

OpenMP. OpenMP is a library (*application program interface - API*) that supports parallel programming on shared memory parallel computers [5].

OpenMP has been developed by a consortium of vendors of parallel computers (DEC, HP, SGI, Sun, Intel, etc.) with the aim to have a standard programming interface for parallel *shared-memory machines*. (like PVM and MPI for distributed memory machines).

The OpenMP functions can be used inside Fortran, C and C++ programs. They allow the parallel execution of code (*parallel DO loop*), the definition of shared data (*SHARED*) and synchronization of processes. OpenMP allows a user to define regions of parallel code (*PARALLEL*) where it is possible to use local (*PRIVATE*) and shared variables (*SHARED*); to synchronize processes by the definition of critical sections (*CRITICAL*) for shared variables (*SHARED*); to define synchronization points (*BARRIER*). However, support for general task parallelism is not included in the OpenMP specification.

Java. A shared-memory programming language is Java that is popular because of its connection with platform-independent software delivery on the Web [6]. Java is an object-oriented language that supports the implementation of concurrent programs by process (called *threads*) creation (*new*) and execution (*start*). For example, the following instructions create two processes:

new *proc (arg1a, arg1b, ..) ;*
new *proc (arg2a, arg2b, ..) ;*

where *proc* is an object of a *thread* class.

Java *threads* communicate and synchronize through *condition* variables. Shared variables are accessed from within *synchronized* methods. Java programs execute *synchronized* methods in a mutually exclusive way generating a critical section. However, *notify* and *wait* operations must be explicitly invoked within such sections, rather than being automatically associated with entry and exit.

This concurrent programming model is useful for using Java on a sequential computer (*pseudo-parallelism*) or on shared-memory parallel computers. To use Java on distributed-memory parallel computers there are different solutions outlined in the next sections.

3 Distributed Memory Languages

A parallel program in a distributed-memory parallel computer (multicomputer) is composed of several processes which cooperate by exchanging data. The processes might be executed on different processing elements of the multicomputer. In this environment, a high-level distributed concurrent programming language offers an abstraction level in which resources are defined like abstract data types encapsulated into cooperating processes. This reflects the model of distributed memory architectures composed of a set of processors connected by a communication network.

This section discusses imperative languages for distributed programming. Other approaches such as logic, functional, and object-oriented languages are discussed in the following sections. Parallelism in imperative languages is generally expressed at the level of processes composed of a list of statements.

Distributed memory languages and tools are: Ada, CSP, Occam, Concurrent C, CILK, HPF, Fortran M, PVM, MPI, C*, ZPL. Some of these are complete languages, others are libraries or *toolkits* that are used inside sequential languages.

PVM. A distributed-memory programming tool that is used currently to implement parallel applications on heterogeneous computers is the Parallel Virtual Machine (PVM) toolkit [7]. PVM provides a set of primitives that can be incorporated into existing procedural languages to implement

parallel programs. PVM is gaining widespread acceptance as a methodology and toolkit for heterogeneous distributed computing. Problems of network delays and dynamically changing machine loads are important considerations in trying to debug or improve the performance of a PVM application.

The PVM environment provides primitives for *process creation* and *message passing* that can be incorporated into existing procedural languages. To create *n* copies of a process, PVM uses

nump = ***pvm_spawn****("proc", …, …, n, pids);*

PVM primitives for point-to-point and global message passing are

- ***pvm_send****(pid, mess);*
- ***pvm_recv****(pid, mess);*
- ***pvm_mcast****(pids, mess);*
- ***pvm_bcast****(pids, mess);*

PVM is widely used as a programming environment for workstation clusters; in fact, PVM runs on many platforms from several vendors. In a PVM program, a process can run on a workstation and another process can run on a supercomputer. For these reasons PVM is widely used and programs are portable, although it offers a low-level programming model. In fact, using PVM, programmers must program all of the process decomposition, placement, and communication explicitly.

MPI. The *Message Passing Interface* or MPI [8] is a de-facto standard message-passing interface for parallel applications defined since 1992 by a Forum with a participation of over 40 organizations. MPI provides *a rich set* of messaging primitives (129), including point-to-point communication, broadcasting, barrier, reduce, and the ability to collect processes in groups and communicate only within each group. MPI has been implemented on massively parallel computers, workstation networks, PCs, etc., so MPI programs are portable on a very large set of parallel and sequential architectures.

An MPI parallel program is composed of a set of processes running on different processors that use MPI functions for message passing. Examples of point-to-point communication primitives are

MPI_Send*(msg, leng, type,…, tag, MPI_COM);*
MPI_Recv*(msg, leng, type,0, tag, MPI_COM, &st);*

Group communication is implemented by the primitives:

MPI_Bcast *(inbuf, incnt, intype, root, comm);*
MPI_Gather *(outbuf, outcnt, outype, inbuf, incnt,..);*
MPI_Reduce *(inbuf, outbuf, count, typ, op, root,...);*

For program initialize and termination the ***MPI_init*** ***MPI_Finalize*** functions are used. Like PVM, MPI offers a low-level programming model, but it is widely used for its portability. Should be mentioned that MPI 1 does not make provision for process creation. Although , in the MPI 2 version additional features should be provided for

- active messages,
- process startup, and
- dynamic process creation.

HPF. Another interesting language is *High Performance Fortran* (HPF) [9]. HPF is the result of an industry/academia/user effort to define a de facto consensus on language extensions for Fortran-90 to improve data locality, especially for distributed-memory parallel computers. HPF is a language for programming computationally intensive scientific applications. A programmer writes the program in HPF using the SPMD style and provides information about desired data locality or distribution by annotating the code with *data-mapping directives*. Examples of data-mapping directives are *Align* and *Distribute:*

!HPF$ Distribute D2 (Block, Block)
!HPF$ Align A(I,J) with B(I+2, J+2)

An HPF program is compiled by an architecture-specific compiler. The compiler generates the appropriate code optimized for the selected architecture. According to this approach, HPF could be used also on shared-memory parallel computers.

HPF is based on exploitation of *loop parallelism*. Iterations of the loop body that are conceptually independent can be executed concurrently. For example, in the following loop the operations on the different elements of the matrix *A* are executed in parallel.

ForAll *(I = 1:N, J = 1:M)*
*A(I,J) = I * B(J)*
End ForAll

C*. A language that implements the SIMD model is the C* data-parallel language [10]. This language was designed by Thinking machines Corp. to program the Connection Machine. However, it can be used to program several multicomputers using the dataparallel approach. C* is an extension of C language that incorporates features of the data SIMD programming model.

The language lets programmer express algorithms as if data could be mapped onto an unbounded number of processors. The compiler automatically maps data onto processing elements. When data are mapped to their processing elements, program constructs can be used to express parallel operations. Although C* implement the SIMD model, efficient compilers for C* on MIMD parallel computers are available.

4 Object-Oriented Parallel Languages

The parallel object-oriented paradigm is obtained by combining the parallelism concepts of process activation and communication with the object-oriented concepts of modularity, data abstraction and inheritance [11].

An object is a unit that encapsulates private data and a set of associated operations or methods that manipulate the data and define the object behavior. The list of operations associated with an object is called its class. Object-oriented languages are mainly intended for structuring programs in a simple and modular way reflecting the structure of the problem to be solved.

Sequential object-oriented languages are based on a concept of passive objects. At any time, during the program execution only one object is active. An object becomes active when it receive a request (message) from another object. While the receiver is active, the sender is passive waiting for the result. After returning the result, the receiver becomes passive again and the sender continues. Examples of sequential object-oriented languages are Simula, Smalltalk, C++, and Eiffel.

Objects and parallelism can be nicely integrated since object modularity makes them a natural unit for parallel execution. Parallelism in object-oriented languages can be exploited in two principal ways:

- using the objects as the unit of parallelism assigning one or more processes to each object;
- defining processes as components of the language.

In the first approach languages are based on active objects. Each process is bound to a particular object for which it is created. When one process is assigned to an object, *inter-object* parallelism is exploited. If multiple processes execute concurrently within an object *intra-object* parallelism is exploited also. When the object is destroyed the associated processes terminate.

In the latter approach two different kinds of entities are defined, objects and processes. A process is not bound to a single object, but it is used to perform all the operations required to satisfy an action. Therefore, a process can execute within many objects changing its address space when an invocation to another object is made.

Parallel object-oriented languages use one of these two approaches to support parallel execution of object-oriented programs. Examples of languages which adopted the first approach are ABCL/1, Actor model, MPL, Charm++, and Concurrent Aggregates. In particular, the Actor model is the best-known example of this approach. Although it is not a pure object-oriented model, we include the Actor model because it is tight related to object-oriented languages.

On the other hand, languages like HPC++, Argus, Presto, Nexus, and Java use the second approach. In this case, languages provide mechanisms for creating and control multiple processes external to the object structure. Parallelism is implemented on top of the object organization and explicit constructs are defined to ensure object integrity.

MPL. The Mentat Programming Language is a parallel extension of C++ that combines the *object-oriented* model with the *data-driven* computation model [12]. In the *data-driven model* parallel operations are executed on independent data when they are available. This model supports high degree of parallelism, while the object-oriented paradigm hides much of the parallel environment from a user.

MPL implements both *inter-object* parallelism (one process per object) and *intra-object* parallelism (more processes per object). The compiler generates code to build and execute data dependency graphs. Thus parallelism in MPL is largely transparent to the programmer. Parallelism implemented on objects of the ***mentat*** class. In this approach, the programmer makes granularity and partitioning decisions using ***mentat*** class definition constructs, and the compiler and the run-time system will manage communication and synchronization.

HPC++. High Performance C++ [13] is a standard library for parallel programming based on the C++ language. The HPC++ consortium consists of people from research groups within Universities, Industry and Government Labs that aim to build a common foundation for constructing portable parallel applications as alternative to HPF. HPC++ is composed of two levels:

- Level 1 consists of a specification for a set of class libraries based on the C++ language.
- Level 2 provides the basic language extensions and runtime library needed to implement the full HPC++.

There are two conventional modes of executing an HPC++ program. The first is *multi-threaded shared memory* where the program runs within one context. Parallelism comes from the parallel loops and the dynamic creation of threads. This model of programming is very well suited to modest levels of parallelism. The second mode of program execution is an *explicit SPMD model* where *n* copies of the same program are run on *n* different contexts. Parallelism comes from parallel execution of different tasks. This model is well suited for massively parallel computers.

Distributed Java. Java is an object-oriented language that was born for distributed computing programming, although it embodies a shared-memory parallel programming model. To develop parallel distributed programs using Java, a programmed can use

- *sockets*: at the lowest programming level Java provides a set of socket-based classes with methods (socket APIs) for inter-process communications using datagram and stream sockets.
- *RMI*: the Remote Method Invocation toolkit [14] provides a set of mechanisms for communication among Java methods that resides on different computers having separate address spaces.
- *Java + CORBA*: At higher level, Java programs can be combined with CORBA [15] object servers for implementing parallel applications running on several remote hosts connected by a communication network (e.g., the Internet).

In the latest two years several efforts have been done to extend Java for high performance scientific applications (e.g., the *Java Grande* consortium). The outcome of these efforts are a set of languages and tools such as: *HPJava, MPIJava, JMPI, JavaSpace, jPVM, JavaPP,* and *JCSP*.

5 Parallel Declarative Languages

Parallel functional languages and *parallel logic* languages are two declarative approaches to parallel programming, concentrating on *what is to be done* rather than *how it is done*. Programs do not specify in any direct way how they are to be executed in parallel, so that decomposition does not need to be explicit. It is still an open question how efficiently these approaches can be implemented.

Parallel Logic Languages. Parallel logic programming is born from the integration of logic programming and parallel programming to evaluate logic clauses in parallel. There are two major forms of parallelism in logic programs are:

- *AND parallelism* and
- *OR parallelism*.

OR parallelism means the parallel evaluation of several clauses whose head unifies with the goal. If we have the subgoal *?p(X)* and the clauses:

p(X) :- q(X).
p(X) :- r(X).
p(X) :- s(X).

OR parallelism is exploited by unifying in parallel the subgoal with the head of the three clauses. AND parallelism consists of the parallel evaluation of each subgoal that composes the current goal. If the goal to be solved is

? p(X), q(Y).

using AND-parallelism, subgoals *p(X)* and *q(Y)* are solved in parallel.

Parallel logic programming models can be divided according to how they exploit parallelism. According to the *explicit parallelism* approach, the programmer specifies parallelism in a logic program by annotations [16]. Languages based on this approach are PARLOG, Concurrent Prolog, GHC, Delta-Prolog, and Strand. On the other hand in implicit parallel logic languages parallelism is extracted both during static analysis and at run-time. System that use this model are PPP, AND/OR Process model, ANDORRA, and Reduce/OR [17].

Parallel Functional Languages. There are some issues in extracting parallelism in a functional setting. The first is that normal order evaluation of expressions does not generate very much parallelism. Because it is so conservative, nothing is computed until it is certain to be needed, which introduces long dependent chains of decisions about which parts of a computation are needed. Implemented naively, the only opportunity for parallelism in a normal order reductive implementation of a functional program is in computing the strict arguments built-in functions.

The second issue is related to the order of functions that are allowed in the programming language. At one extreme is higher-order functional programming, exemplified by Haskell, in which functions of all orders are permitted. At the other extreme is dataflow in which all functions are first-order. Higher-order functional programming can be a very powerful and compact style, but it is hard to implement implementations that approach the performance of imperative languages are only just becoming available after fifteen years of research. The third issue is the trade-off between the implicit approach (the compiler discover parallelism) and the explicit approach where programmers must specify parallelism via annotations. Example of parallel functional languages are Multilisp, ParAlfl, SISAL, Concurrent Lisp, *Lisp, and Qlisp.

The *Multilisp* language is an extension of Lisp in which opportunities for parallelism are created using *futures* [18]. A future applied to an expression (***future (x)***) creates a task to evaluate that expression (which begins immediately, that is eagerly).

Sisal (*Streams and Iterations in Single Assigment Language*) is an interesting parallel functional language. Most of the parallelism in Sisal programs comes from parallel loops. Sisal syntax is very like conventional imperative languages, but the meaning

of most statements is different in important ways. Sisal is a single-assignment language, so that only a single value can be assigned to each named variable in each scope. Thus Sisal instructions have the same semantics of *functional expressions*. In fact, to assign a new value to a variable on the basis of its previous value the keyword *old* must be used:

i:= ***old*** i + 1;

Exists a powerful Sisal compiler for shared-memory parallel machines. Many Sisal scientific programs have equal or better speedups than equivalent Fortran programs [19].

6 Composition Based Languages

This section describes novel models and languages for parallel programming that have not yet become widely accepted, but have properties that make them of interest. Some are not yet complete programming languages but programming models or abstract machines.

The general trend that is discernible in these languages, to those discussed in the previous sections, is that they are designed with stronger semantics directed towards software construction and correctness. There is also a general realization that the level of abstraction provided by a parallel programming language should be higher than was typical of languages designed in the past decade. This is no doubt partly due to the growing role of parallel computation as a methodology for solving problems in many application areas.

The languages described in this section can be divided into two main classes:

- Languages based on predefined structures that have been chosen to have good implementation properties or that are common in applications (***restricted computation forms*** or ***skeletons***) [20];
- Languages that use a ***compositional*** approach - a complex parallel program can be written by composing simple parallel programs while preserving the original proprieties [21].

Skeletons are predefined parallel computation forms that that embody both data- and control-parallelism. They abstract away from issues such as the number of processors in the target architecture, the decomposition of the computation into processes, and communication supporting high-level *structured parallel programming.* However, using skeletons cannot be wrote arbitrary programs, but only those that are compositions of the predefined structures. The most used skeletons are *geometric*, *farm*, *divide&conquer*, and *pipeline*. Examples of skeleton languages are SCL, SkieCL, BMF (*Bird-Meertens formalism)*, Gamma, NESL, and BSPlib.

SkieCL. The *Skie Coordination Language* [22] uses a skeleton-based model and implements a set of parallel programming skeletons such as:

- ***pipe*** for pipeline computations,
- ***farm*** for defining a set of server processes,
- ***tree*** to run a process tree,
- ***geometric*** for implementing *data parallelism*,
- ***loop*** for implementing loop parallelism.

Sequential code of processes inside *skeletons* is sequential *(C, Fortran, C++, Java)*. For example worker farms, in P3L are modeled by means of the ***farm*** constructor. When the skeleton is executed, a number of workers *W* are executed in parallel with the two *P* processes (the emitter and the collector). Each worker executes the function *f()* on its data partition.

BSPlib. The BSP model is not a pure skeleton model, but it represents a related approach in which the structured operations are single threads. Computations are arranged in ***supersteps***, each of which is a collection of these threads. A superstep does not begin until all of the global communications initiated in the previous superstep have completed (so that there is an implicit barrier synchronization at the end of each superstep) computational results are available to all processes after each superstep. The *Oxford BSPlib* is a toolkit that implements the BSP operations in C and Fortran programs [23]. Some examples of BSPlib functions are

- ***bsp_init(n)*** to execute *n* processes in parallel,
- ***bsp_sync*** to synchronize processes,
- ***bsp_push_reg*** to make a local variable readable by other processes,
- ***bsp_put*** and ***bsp_get*** to get and put data to/from another process,
- ***bsp_bcast*** to send a data towards *n* processes.

The BSPlib provides automatic process mapping and communication patterns.

Significant examples of compositional languages are PCN, Compositional C++, and Seuss. PCN and Compositional C++, are second-generation parallel programming languages in the imperative style. Both are concerned with composing parallel programs to produce larger programs, while preserving the original properties.

PCN. PCN programs consist of collections of procedures, with internal structure in the Occam style (parallel, sequential, or choice collections of statements) [24]. In PCN communication is handled by single-use variables, which may be written once within their scope but may be read multiple times (blocking if the write has not yet occurred). Streams

can be implemented by recursive procedure calls.
PCN also includes higher-level structuring tools: arrays of communication variables, sets of processes and an associated virtual topology for reuse, and structured sets of cells (which start to resemble skeletons). There are also annotation features: built-in functions to determine topology, number of processors, and location of a particular piece of code, which allow procedures to modify their actions according to implementation context.

CC++. Compositional C++ [25], contains many of the same ideas but in an object-oriented framework that forms a superset of C++. Three constructors, *par*, *parfor*, and *spawn*, can be used to structure code. Communication again uses single-use variables, called *sync* variables, and using the same syntax as constants. Unlike PCN, threads can share other variables, but the final value of a variable that is written by multiple threads is only guaranteed to be one of the possible values. Logical processor objects allow code to be grouped to preserve locality, and provide a name space.

7 Conclusion

Parallel programming languages support the implementation of high-performance applications in many areas: from the Internet to computational science.

New models, methods and languages proposed and implemented in the latest years allow users to develop more complex programs with minor efforts. In the recent years we registered a trend from low-level languages towards more abstract languages to simplify the task of designers of parallel programs and several middle-level models has been designed as a tradeoff between abstraction and high performance.

These efforts may bring parallel programming to the right direction for supporting a wider use of parallel computers. In fact, parallel computation can be a key technology of the next century if

- languages and programs will be architecture independent;
- the complexity of parallel programming will not greater that the complexity of sequential programming; and
- there will be portable and standard parallel software.

References:

[1] D. Skillicorn, D. Talia, *Programming Languages for Parallel Processing*, *IEEE Computer Society Press*, 1994.

[2] D. Skillicorn, D. Talia, Parallel Programming Models and Languages, *ACM Computing Surveys*, Vol. 30, No. 2, 1998, pp. 123-169.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, Orca: A Language for Parallel Programming of Distributed Systems, *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, 1992, pp. 190-205.

[4] N. Carriero, D. Gelernter, Application Experience with Linda. *Proc. ACM/SIGPLAN Symposium on Parallel Programming*, Vol. 23, 1988, 173–187.

[5] OpenMP Consortium, *OpenMP C and C++ Application Program Interface*, Version 1.0, 1997.

[6] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1997.

[7] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam, PVM and HeNCE: Tools for heterogeneous network computing, *Software for Parallel Computation*, Vol. 106 of NATO ASI Series F, Springer-Verlag, 1993.

[8] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra, *MPI: The Complete Reference*, The MIT Press, 1996.

[9] D.B. Loveman, High Performance Fortran, *IEEE Parallel & Distributed Technology*, pp. 25-43, 1993.

[10] M.J. Quinn, P.J. Hatcher, Data-parallel Programming on Multicomputers. *IEEE Software*, pp. 69-76, 1990.

[11] A. Yonezawa et al., *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[12] A.S. Grimshaw, Easy-to-use Object-oriented Parallel Processing with Mentat, *IEEE Computer*, pp. 39-51, 1993.

[13] S. Diwan, D. Gannon, A Capabilities Based Communication Model for High-Performance Distributed Applications: The Open HPC++ Approach, *Proc. IPPS/SPDP'99*, 1999.

[14] Javasoft, RMI, *The JDK 1.1 Specification*, 1997.

[15] S. Baker, V. Cahill, and P. Nixon, Bridging Boundaries: CORBA in Perspective, *IEEE Internet Computing*, Vol. 1, No. 5, pp. 52-57, 1997.

[16] D. Talia, A Survey of PARLOG and Concurrent Prolog: The Integration of Logic and Parallelism, *Computer Languages*, Vol. 18, No. 3, pp. 185-196, 1993.

[17] D. Talia, Parallel Logic Programming Systems

on Multicomputers, *Journal of Programming Languages*, Vol. 2, No. 1, pp. 77-87, 1994.
[18] R.H. Halstead, Parallel Symbolic Computing, *IEEE Computer*, Vol. 19, No. 8, pp. 35-43, 1986.
[19] D. Bollman, J. Seguel, and J. Feo, A Functional Approach to Radix-r FFTS, *Parallel and Distributed Computing Practices*, Vol. 1, No. 1, pp. 51-74, 1998.
[20] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.
[21] K.M. Chandy, C. Kesselman, The Derivation of Compositional Programs, *Proceedings of the Joint Int. Conf. and Symp. on Logic Programming*, pp. 3-17, 1992.
[22] B. Bacci, et al., *SkIECL: User's Guide*, Quadrics Supercomuputers World Ltd., 1998.
[23] J.M.D. Hill et al., BSPlib: The BSP Programming Library, *Parallel Computing*, Vol. 24, No. 14, pp. 1947-1980, 1998
[24] I. Foster, B. Olson, Productive parallel programming: the PCN approach, *Scientific Programming*, Wiley, pp. 51-66,1992.
[25] K.M. Chandy, C. Kesselman, CC++: A Declarative Concurrent Object-Oriented Programming Notation, *Research Directions in Concurrent Object Oriented Programming*, MIT Press, 1993.