INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

The United Nations University

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/772 - 18

INTERNATIONAL WORKSHOP ON PARALLEL PROCESSING AND ITS APPLICATIONS IN PHYSICS, CHEMISTRY AND MATERIAL SCIENCE
5 - 23 September 1994

PARALLEL PROGRAMMING LANGUAGES

Domenico TALIA
CRAI
Localita' di Santo Stefano
Rende
87036 Cosenza
ITALY

These are preliminary lecture notes, intended only for distribution to participants.

Parallel Programming Languages

Domenico Talia

CRAI - Consorzio per la Ricerca e le Applicazioni di Informatica

International Workshop on
Parallel Processing and its Applications
Trieste, September , 1994

Email: dot@crai.it

CRAI

OUTLINE

- Introduction
- Background
- Shared-Memory Languages
- Distributed-Memory Languages
- Parallel Object-Oriented Languages
- Parallel Functional Languages
- Parallel Logic Languages
- Conclusions

INTRODUCTION

- To assure the success of parallel computation it is necessary to
  - invest in the development of parallel software languages and tools.
  - train many people to write parallel software.

- **Aim of the lecture**:

  To give an overview of the most important parallel programming languages designed in the last decade and to introduce issues and concepts related to the development of parallel software.

- The lecture covers 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 next ten years.

INTRODUCTION

- The main reasons to use parallelism are because:
  - it can deliver high performance when necessary,
  - many real world problem are naturally parallel and can be modeled more naturally and directly in a parallel way,
  - a multiprocessor can be cheaper per computation than the leading edge uniprocessor, especially as development costs increase for each new generation of processor,
  - the speed of light is a bound on the speed that can be achieved by a single processor and so there is ultimately no choice.

- These will lead to a mass market in parallel computation if the software development will not be much complex.

INTRODUCTION

- Parallel computers represent more and more a great opportunity to develop high performance systems and to solve large problems in many application areas.

- This trend is driven by parallel programming languages and tools which contribute to make parallel computers useful in supporting a broad range of applications.

- Parallel programming languages allow the design of parallel algorithms as a set of concurrent actions mapped onto different computing nodes. The cooperation between two actions can be performed in many ways according to the selected language.

- 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.

INTRODUCTION

- Typical issues in parallel programming are
  - *process creation,*
  - *synchronization,*
  - *communication handling,*
  - *deadlock, and*
  - *process termination.*

- These issues mainly arise from the fact that when a concurrent program is running there are many flows of control through the program, one for each process.

- A designer of parallel applications is concerned with the problem of ensuring the correct behavior of all of the processes that comprise the program.

- Parallel programming languages were designed to help programmers cope with these problems, aiming to allow the programming of parallel computers to be not much harder than programming sequential computers.

BACKGROUND

- Parallelism in programming languages was originally studied as a branch of operating system programming.

- The proposed primitive mechanisms are ***semaphores***, ***conditional critical regions***, and ***monitors***.

- ***Semaphores*** are a simple mechanism for the implementation of synchronization between processes which access the same resource.
  - Each access to a shared resource must be preceded by a request operation and followed by a signal operation on the same semaphore.

- In ***conditional critical*** regions each shared resource may be accessed only using conditional critical region statements which guarantee mutual exclusion on the resource.

- ***Monitors*** encapsulate both a resource and operations that manipulate it. Resources defined in a monitor must be accessed *using only* the operations defined by the monitor itself.

BACKGROUND

- These mechanisms were used to extend sequential programming languages. For example, the monitor concept was introduced by Brinch Hansen in the language *Concurrent Pascal.*

- As a consequence of these early research activities several high-level parallel programming languages were designed and implemented on the parallel computers that were commercially available.

- These languages were mainly based on the conventional imperative approach, and began to be used to express and develop parallel programs, extending their application area from operating systems to many others.

- In the last decade a large number of parallel programming languages providing high-level constructs and mechanisms have been designed. They reflect many paradigms and architecture models and cover a wide range in terms of goals, performance, and applications.

SHARED-MEMORY LANGUAGES

- The concept of shared memory is a useful way to decouple program control flow issues from issues of data mapping, communication, and synchronization.

- In shared-memory parallel architectures this programming style can be mapped directly. However, programming is difficult because programmers are responsible for ensuring that simultaneous memory references to the same location do not occur.

- One way to make shared-memory programming easier is to use techniques adapted from operating systems to enclose accesses to shared data in *critical sections*.

- Physical shared memory cannot be provided on massively parallel systems, but it is a useful abstraction, even if the implementation it hides is distributed.

SHARED-MEMORY LANGUAGES

- In highly parallel computers a useful 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 approach is to emulate shared memory well enough that the same number of messages travel around the system when a program executes as would have travelled if the program had been written to pass messages explicitly.

  - The emulation of shared memory imposes no extra message traffic.

- The other way is to build a system based on a useful set of sharing primitives.

- Shared-Memory languages:

  Concurrent Pascal, *Linda*, *Orca*, *SDL* (Shared Dataspace Language), *Ease*.

LINDA

- Linda provides an associative memory abstraction called *tuple space.*

- Processes 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, one by copying: `rd(T)` and the other destructively: `in(T)`
  - and one which evaluates its components before storing the results in tuple space (allowing the creation of new processes): `eval(T)`.

LINDA

- An example program which finds prime numbers:

```
main()
{
  int i, ok;

  for(i=2; i < LIMIT; ++i)
  {
   eval("primes", i, Is_Prime(i));
  }
  for(i=2; i <= LIMIT; ++i)
  {
   rd("primes", i, ? ok);
   if (ok)
     printf ("%d\n", i);
  }
}
```

- 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.

ORCA

- The approach used in Orca is to build a system based on a useful set of sharing primitives.

- The Orca system is a hierarchically structured set of abstractions.

  - 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

- Parallelism in Orca is based on processes: an explicit **fork** primitive is provided for spawning a new child process and passing shared data object:

```
fork child on (processor);
```

- Processes communicate indirectly by shared data objects.

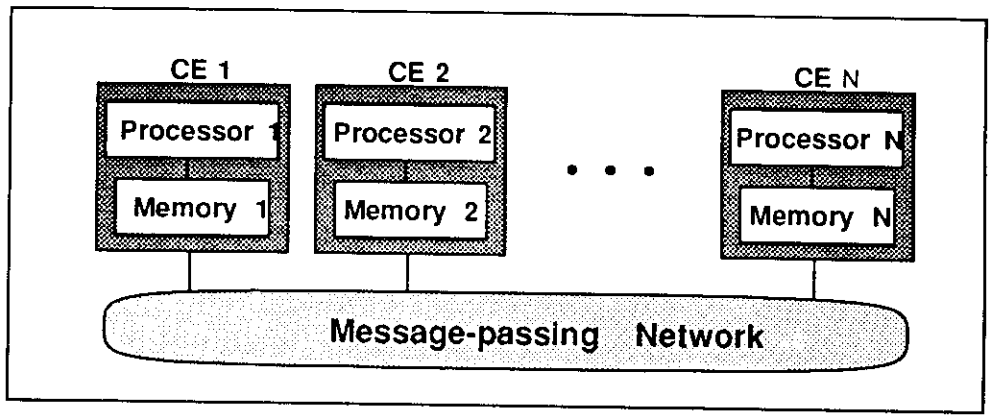
- A child process may pass an object as **shared** to its children, and so on there will be a hierarchy of processes sharing objects:

```
process child (Id: integer; ObjX: shared ObjectType);
begin
  .....
end;
```

- The implementation of Orca is based on
  - **replication**
  - **reliable broadcast**.

DISTRIBUTED-MEMORY LANGUAGES

- These languages reflect the model of distributed-memory architectures composed of a set of computing elements (CE) connected by a communication network.



- In this approach, a distributed concurrent program consists of a set of processes cooperating by message passing and located on one or many computers.

- The two major issues in designing distributed languages for parallel programming are related to **process spawning** and **process cooperation**.

DISTRIBUTED-MEMORY LANGUAGES

- Some languages provide primitives for explicit process creation during the parallel program execution (*dynamic creation*), **fork/join**, **new**, and **create**.

- While in others the total number of processes is defined at compile time (*static creation*) by **parbegin**, **cobegin/coend**, and **par**.

- Several mechanisms have been defined for the cooperation of concurrent processes. They can be divided into three main classes:
  - *explicit message passing*,
  - *rendezvous*,
  - *remote procedure call*, and
  - *data parallelism*.

- Distributed-Memory languages:

  Ada, ***CSP***, ***Occam***, ***HPF***, *Concurrent C*, *NIL*, *Joyce*, *C**, *DP*, *CLU*, *SR*, ***PVM***, *P4*, *Parmacs and Express*.

CSP

- In the area of distributed-memory paradigms for parallel programming, the CSP (*Communicating Sequential Processes*) model influenced the design of several other languages.

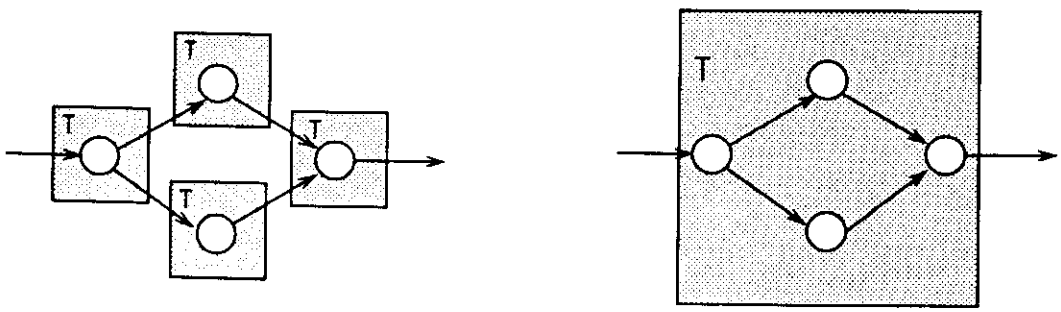
- In CSP a program is regarded as a network of processes, each one with its local environment, and cooperating by means of explicit message passing.

- The main features of the CSP model are:

  - communication management by means of input and output commands [*send* (**!**) and *receive* (**?**)], and the use of channels,
  - the exploitation of parallelism by means of the parallel command [**P** || **Q**] , and
  - non-determinism management by guarded commands.

OCCAM

- Occam is based on the concepts of concurrency and communication derived from the CSP model.
- It has been developed at INMOS as the basic language of Transputer.
- Occam programs are expressed in terms of concurrent processes each one operating on its own variables and communicating through *synchronous* channels.
- A typical Occam program is composed of a network of processes running on a Transputer network. However, the same processes can also be executed concurrently on a single Transputer sharing its time among the processes.



CRAI

OCCAM

- Occam programs are constructed from a small number of primitive processes:

  assignment (**:=**), input (**?**), and output (**!**).

- To design parallel processes, the primitive processes can be combined using the parallel constructor *PAR*:

```
PAR
  Prod (chproducer)
  Cons (chconsumer)
  SingleBuffer (chproducer, chconsumer)
```

- A parallel construct terminates only after all of its components have terminated. That is if only one process of the PAR construct does not terminate because it is waiting to communicate, the entire PAR process becomes waiting.

```
PROC SingleBuffer (CHAN OF BYTE chproducer, chconsumer)
  BYTE c    :
  BOOL end IS FALSE:
  SEQ
    WHILE NOT(end)
      chproducer ? c
      chconsumer ! c
:
```

HPF

- HPF (High Performance Fortran) 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 on SIMD, MIMD and vector processors.

- 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.

- The program is compiled by an architecture-specific compiler. The compiler generates the appropriate code optimized for the selected architecture.

HPF

- ***Processor directive***: declares rectilinear processor arrangement specifying their name, number of dimensions and size of each dimension:

```
!HPF$  Processors P(128,64), Q(8192)
```

- ***Distribute directive***: specifies a mapping of data to abstract processors in a processor arrangement:

```
!HPF$  Distribute D2()
```

- ***Align directive***: specifies that certain data are to be distributed in the same way as other data:

```
!HPF$  Align A(I,J) with B(I+2, J+2)
```

- ***ForAll contruct***: expresses parallel assignement to sections of arrays:

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

- HPF also defines a standard library of computational functions and a set of built-in data-mapping directives.

PVM

- The PVM (Parallel Virtual Machine) environment is not a complete language, but it provides a set of primitives that can be incorporated into existing procedural languages to implement parallel programs.

- The PVM system is gaining widespread acceptance as a methodology and tool kit for heterogeneous distributed computing. Hundreds of sites around the world use PVM for scientific applications.

- PVM supplies functions to start processes and lets them to communicate with each other.

- Users can write parallel programs in FORTRAN or C by calling simple PVM routines such as ***pvm_send()*** and ***pvm_recv()***.

- PVM applications may be run transparently across a wide variety of architectures. Some processes may run on a vector supercomputer and others on a powerful workstation.

PVM

- A process in PVM is spawned by invoking the primitive [redacted] including the process name: [redacted] ("Proc1").

- Several primitives are defined to provide inter-process communication:
  - ***pvm_send****(process, message)*: to send a data to a process;
  - ***pvm_recv****(process, message)*: to receive a data from a process;
  - ***pvm_mcast****(processes, message)*: to send a data to a set of processes;
  - ***pvm_bcast****(processes, message)*: to send a data to all the processes which compose a parallel program;

- The version 3.0 does not:
  - automatically parallelize
  - do automatic load balancing.

PARALLEL OBJECT-ORIENTED LANGUAGES

- 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.

- An object interacts with the outside world exclusively through an interface defined by its operations.

- By *inheritance* a class may be defined as an extension or restriction of another previously defined class.

- 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.*

PARALLEL OBJECT-ORIENTED LANGUAGES

- 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 (*active objects*): If multiple processes execute concurrently within an object intra-object parallelism is exploited.
  - defining processes as components of the language: a process is not bound to a single object, but it is used to perform all the operations required to satisfy an action.

- 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, Actors, Concurrent Smalltalk, and Mentat.

- On the other hand, systems like Argus, Presto, and Nexus use the second approach.

- Other important parallel object-oriented languages are POOL-T, Emerald, COOL, Orient84/K, Cantor.

ACTOR MODEL

- The Actor model was originally proposed by Hewitt. It was then developed by Agha at MIT.

- Actors are autonomous, distributed, parallel objects that can send each other messages asynchronously.

- Actors can be created dynamically during the program execution.

- An actor can send a message to a single actor or broadcast it to an entire group of actors.

- When a message arrives, the actor executes a script which accepts the message if it recognizes it, or delegates the rejected message to a proxy which can respond to the message.

MENTAT

- Mentat is an parallel object-oriented system designed to address the problems of developing architecture-independent parallel applications.

- The Mentat system integrates a data-driven computation model with the object-oriented paradigm.

- The data-driven model supports high degree of parallelism, while the object-oriented paradigm hides much of the parallel environment from the user.

- The Mentat language is an extension of C++ which supports both intra- and inter-object parallelism.

- The basic idea is to let the programmer specify which C++ objects can be executed in parallel. Compiler and run-time support accomplish the task.

PARALLEL FUNCTIONAL LANGUAGES

- Functional programming and logic programming are two declarative approaches to parallel programs, 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.

- Communication and synchronization take place as needed during the execution of a computation, and are not visible, nor even predictable, by the programmer.

- It is still an open question how efficiently these approaches can be implemented.

- Parallel graph reduction is the usual implementation technique for functional programming, and has been only a limited success, particularly implemented on distributed-memory machines.

PARALLEL FUNCTIONAL LANGUAGES

- Parallel functional programming languages:
  Multilisp, ParAlfl, SISAL, Concurrent Lisp, QLisp.

- Multilisp, an extension of Lisp in which opportunities for parallelism are created using *futures*.

- A *future* applied to an expression
  (future (X))
  creates a task to evaluate that expression.

- An attempt to use the result of a future suspends until the value has been computed.

- Futures allow eager evaluation in a controlled way that fits between the fine-grained eager evaluation of dataflow and the laziness of higher-order functional languages.

SISAL

- Sisal began as an abstract dataflow language. Its syntax is very like conventional imperative languages, but the meaning of most statements is different in important ways.

```
for initial
   i:=1;
   x:=y[1]
while i<n repeat
   i:= old i + 1;
   x:= old x + y[i]
returns array of x
end for
```

- First, it is a single-assignment language, so that only a single value can be assigned to each named variable in each scope. Thus, in effect, all statements are expression.

- Most of the parallelism in Sisal programs comes from parallel loops, whose bounds are defined by range generators, which only incidentally impose an ordering on loop bodies.

- A powerful Sisal compiler for shared-memory machines exists, and many *Sisal scientific programs have better speedups than equivalent Fortran programs.* Much of this gain comes from better compilation, thanks to simpler language semantics.

PARALLEL LOGIC LANGUAGES

- Parallel logic programming is born from the integration of logic programming and concurrent programming.

- These models proposed for the implementation of logic programming on parallel computers can be divided according to the kind of parallelism they exploit and how it is exploited.

  - ☞ If the parallelism is specified in a logic program by the programmer (*explicit parallelism*)
  - ☞ or it is extracted by the language support both during static analysis and at run-time (*implicit parallelism*).

- Models of explicit parallelism are called *concurrent logic programming languages*. Example of these are PARLOG, P-Prolog, Delta-Prolog, GHC, and Concurrent Prolog.

- Using these languages the programmer must specify, by means of annotations, which clauses can be solved in parallel.

- On the other hand, parallel logic models based on implicit parallelism are PPP, the AND/OR Process model, the REDUCE/OR model, ANDORRA, OPERA, and PALM.

PARALLEL LOGIC LANGUAGES

- The two major forms of parallelism among the previous ones, are
  - *AND PARALLELISM*
  - *OR PARALLELISM*

- *AND PARALLELISM* is obtained by parallel resolution of the subgoals in the body of a clause.

```
:- p(X, Y).

p(X, Y) :- q(X), r(Y).
```

- *OR PARALLELISM* is obtained by solving in parallel the clauses whose head unifies with the goal.

```
:- p(X).

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

CONCURRENT ~~PARALLEL~~ LOGIC LANGUAGES

- Concurrent logic languages are designed to execute on parallel machines using explicit parallelism.

- A user must specify, by means of annotations, which clauses can be solved in parallel.

- Concurrent logic languages can be viewed as a new interpretation of Horn clauses, the *process interpretation.*

- According to this interpretation

  - an atomic goal **<- C** can be viewed as a process,
  - a conjunctive goal **<- C1, . . . , Cn** as a process network,
  - a logic variable shared between two subgoals can be viewed as a communication channel between two processes.

CONCURRENT ~~PARALLEL~~ LOGIC LANGUAGES

- A program in a concurrent logic language is a finite set of guarded clauses:

```
H <- G1, G2, .., Gn | B1, B2, .., Bm.      (n, m >= 0)
```

  where H is the clause head, {Gi} is the guard, and {Bi} is the body of the clause.

- Operationally the guard is a test that must be successfully evaluated with the head unification so that the clause could be selected.

- '|' is called a commit operator, and it is used as a conjunction between the guard and the clause body. If the guard is empty (*n=0*), the commit operator is omitted.

- The declarative reading of a guarded clause is: H is true if both {Gi} and {Bi} are true. According to the process interpretation, to solve H it is necessary to solve the guard {Gi}, and if its resolution is successfully, B1, B2,..., Bm are solved in parallel.

NOVEL APPROACHES

- Approaches to parallel programming that have not yet become accepted, but have properties that make them of interest.

- Program Composition Notation (PCN)
- Compositional C++
- Strand
- The Bird-Meertens formalism
- Pisa Parallel Programming Language (P^3L).
- Gamma
- The BSP language

CONCLUSIONS

- Parallel programming languages are tools that any good programmer or researcher should know.

- Parallelism can lead to major gains in performance in many application areas.

- New programming methods and languages should let developers write more complex programs with less effort and make the expression of parallelism simpler.

- We think this survey is a reasonable starting point both for
  - researchers wishing to better understand parallel languages features and uses,
  - programmers which must solve computing intensive or naturally parallel problems.