Published in the Proceedings of the IEEE 1992
International Conference on Computer Languages ICCL, Oakland, California, USA, pages 54–64

Modula-P
A Language for Parallel Programming
Definition and Implementation on a Transputer Network*

Jürgen Vollmer Ralf Hoffart
GMD Research Group at the University of Karlsruhe†

Abstract

The programming language Modula-P *extends* Modula-2 *with CSP* (Communicating Sequential Processes) *based language constructs, i.e. parallel processes, synchronous message passing, and simultaneous waiting for events. The language and its implementation on a Transputer network is presented.*

1 Introduction

Now that parallel computers with various architectures have become increasingly available, the problem has arisen of how best to program such computers. There are at least two answers: either the compiler should automatically generate a parallel program from the sequential program text, or the programmer should make the parallelism explicit.

We have chosen the second approach, and use a MIMD[1] machine model with a distributed memory architecture. A corresponding computational model is Hoare's *Communicating Sequential Processes* (CSP) [4], which provides a powerful framework for the description and analysis of parallel programs. We have integrated CSP into the programming language *Modula-2* [14]. The new language, called *Modula-P* [12], its key design ideas, and some problems of its implementation on a network of *Transputer* processors are presented.

2 Design Goals and Decisions

Why should one add more features to a good language like *Modula-2* ? As a matter of fact, each new dialect of a language decreases the portability of programs to other machines which don't support this special dialect. On the other hand there are new (parallel) computer architectures that offer special features which should be supported by a language.

Based on some experience at our institute with the *occam* language [6], we have discovered the following requirements for a parallel language[2]:

- Concurrency should be based on a well founded concept.
- Software engeneering aspects should be considered that is programming "in the large" should be supported as well as programming "in the small".
- The language should include a rich set of data and program structuring mechanisms and modules.
- Existing sequential software should be reusable.
- A clear syntactic and semantic definition of what a process is must be given.
- The program should be executable on different kinds of network topologies without having to change the program text.
- Explicit "process placement" should be possible to increase execution speed, But correct execution should not depend on it.

CSP has been considered as a good foundation of concurrency for the MIMD architecture. Instead of creating an entirely new language, *Modula-2* is used as sequential basis. To facilitate easier learning, the syntax of the new constructs should be close to that of *Modula-2*.

For this project there are three challenges:

1. The resulting language must be designed orthogonally to the base language, such that the design goals are met.
2. An implementation of an appropriate *Modula-P* runtime model on a single processor / Transputer must be designed.
3. A communication system for a point-to-point connected processor network must be designed and implemented.

The next section mentions some other attempts to integrate CSP into a language.

3 Languages based on CSP

There are several languages which are based on or influenced by CSP. Experimental languages are for example [15, 8]. Their goal was to solve shortcomings of the first CSP proposal, they do not support any software engeneering aspects.

occam [6] is an attempt to implement CSP on a distributed computer system, a Transputer network. It is an elegant and simple language. It has only limited

*This research was supported in part by ESPRIT under ESPRIT project 5399 (COMPARE).

†Vincenz-Prießnitz-Straße 1, D-7500 Karlsruhe 1

[1] Multiple Instruction Multiple Data

[2] Language with explicit constructs expressing parallelism.

data structuring mechanisms; procedures can't be recursive, and the important concept of data abstraction using modules is missing. Process distribution over the network must be done on the top of the process hierarchy.

A *Pascal*-based language for distributed programming is *Joyce* [2]. A Joyce program defines concurrent agents which communicate through synchronous channels. Several agents may communicate over one channel.

Tasks are used in *Ada* [1] to specify concurrency. Task communication is based on the *rendezvous* concept. One or more tasks perform an *entry call* which one receiving task (which defined that entry call) *accepts*.

Wirth [13] and Collado [3] implemented CSP-like features in *Modula-2*, but without extending *Modula-2*. They map processes onto *Modula-2* coroutines. Channels must be declared as variables. The type of information transmitted over a channel is either fixed (Wirth) (using a separate module for each information type) or arbitrary (Collado) (using the "ARRAY OF WORD" concept of *Modula-2*). Both languages implement simultaneous waiting for input from several channels by polling over the channels, a method which is very expensive and hard to code. As a consequence, these implementations receive little benefit from running on a multi computer network.

4 Language Design Problems

A main problem of integrating concurrency into languages like *Modula-2* and *Ada* concerns their modularization concepts and notion of global variables.

The problem is illustrated by the following example: *Modula-2*'s `DEFINITION` and `IMPLEMENTATION MODULE`s are used to specify and implement an abstract data type (ADT), e.g. *stack* with operations *push(elem)*, etc. Let us assume that its implementation is based on a global *array* variable, declared in the *implementation module*. In the "sequential" case, each call to one of the procedures implementing an operation uses the same stack by accessing the global *array* variable. Now consider the "parallel" case, where a program consists of several processes. Two usages may be distinguished: first, each process needs its own stack independently of all other processes, and second, the processes want to communicate in a stack like fashion.

The questions is whether it is possible to use the original stack module. Unfortunately, the answer must be "no", since it is not clear from the program text which usage is meant by the programmer.

More generally stated: what is the semantics of global variables of modules in the presence of several executed processes? Which process on which processor owns (i.e. provides storage for) the global variables of a module? How are the global variables accessed if the processes are executed on distributed memory processors?

One solution to the problem is to forbid global variables inside modules and using parameters instead. But this contradicts the information hiding principle, because the data structure implementing the ADT is now externally visible. It also contradicts the design goal that the new language should be a superset of the basic language. Another solution is to introduce new concepts of dealing with modules as done in the next section for *Modula-P*.

5 The Language Modula-P – Reference Manual–

Modula-P is a superset of *Modula-2* [14], enriched with language constructs based on the model of the *Communicating Sequential Processes (CSP)* [4] for writing parallel programs[3].

5.1 Language Overview

Modula-P knows the notion of processes which are parallelly executed as components of the `PAR` statement. Two kinds of processes are syntactically distinguished: *local* and *global* processes. Synchronous communication between processes is done via typed channels. Input and output statements are defined for channels. Channels must be initialized by the parents of the communicating processes using the standard procedure `OPEN`. Processes may wait for several events simultaneously by means of an `ALT` statement. Processes and alternatives may be replicated using a *replicator* inside a `PAR` and `ALT` statement, respectively. The language knows the notion of an abstract clock (`TIME`), which may be read. Processes may delay themselves. Global processes may be executed on processors having no shared memory, while local ones require access to shared variables. Messages between processes running on different processors are passed automatically through the underlying communication network.

5.2 Processes

A *process* is a piece of executed program text; a *process declaration* is a piece of program text which may be executed as process. Parallel execution of processes is expressed by the `PAR` statement. The process executing the `PAR` statement is called the *parent process* of its components, the *child processes*. Processes may be either declared as *local* or *global* processes.

5.2.1 Local Processes

Local processes are specified by a statement sequence as component of a `PAR` statement. Hence, several local processes with the same parent share variables with their parent process[4]. A local process declaration is

[3] The syntax is given in Backus-Naur-Form according to [14].

[4] Notice: The language doesn't specify synchronized access of variables.

allowed to contain neither a `RETURN` nor an `EXIT` statement which is related to a `LOOP` outside of that local process declaration.

A local process terminates if the last statement of the statement sequence is executed.

5.2.2 Process Modules and Global Processes

A global process is declared in a `PROCESS MODULE`. This is something like a *Modula-2* program module. The syntax of the process declarations is given in figure 2. A simple example of a process module is given in figure 1. The following rules must be obeyed using process modules and global processes:

A process module declaration may have *value parameters*. The types used in the formal parameter list are either predefined types or are implicitly imported; hence, this type identifier must be qualified. These qualified identifiers are known only inside this parameter list. A process module may import other (process) modules. A process module is another compilation unit.

The body of a process module is executed as a component of the `PAR` statement. Syntactically this looks like a procedure call with the name of the process module and passing actual parameters which are evaluated before the process starts its execution. The bodies of any imported ordinary modules are executed each time the process is started. This is done prior to the execution of the body of the process module. A global process terminates if the last statement of the process module body is executed.

Each global process has its own instance of all imported global variables. A global process has access neither to variables of other global processes, nor to those of the parent process. Interaction between global processes is only possible using communication over channels.

```
PROCESS MODULE p
  (ch:m.channel; x:m.type; i:CARDINAL);
  (* imports *)
  (* local declarations *)
BEGIN
  (* actions executed by that process *)
END p.

MODULE main;
  IMPORT p, m;  ...
BEGIN
  PAR
    | p(ch,x,i) (* execute global process *)
    | ...
  END
END main.
```

Figure 1: A `process module` stub

5.2.3 Distributed Processes Execution

Local processes must be executed together with their parent process on processors which share a common memory. Global processes may be executed on processors having no shared memory.

Modula-P supports two kinds of process placements. Process placement does not affect the program semantics, except for execution speed and consequences thereof (e.g. race conditions).

1. The runtime system may distribute the execution of processes using a default strategy.
2. The user gives a placement directive using `AT expression`. The *expression* must be assignment compatible with `INTEGER`. The meaning of this integer value is determined by the runtime system. The standard module *net* contains some functions to compute this value.

5.3 The PAR Statement

The `PAR` statement specifies the concurrent execution of its components.

`PAR` $p_1 | \cdots | p_n$ `END`

The processes $p_1, \cdots, p_n$ are executed in parallel. The process executing this PAR statement is suspended until all of its child processes $p_1, \cdots, p_n$ have terminated. The component p_i may be either a statement sequence or a call of a process module.

5.4 Communication

Channels are used for unidirectional, synchronous communication between two concurrently executing processes. The message goes in one direction from the sender to the receiver. The term synchronous specifies that the process which reaches its communication statement first will wait until the second process reaches the corresponding statement. Then the communication takes place, i.e. the message is passed, and both processes continue independently. A process may either input or output to a specific channel, but mixing these operations is not allowed (i.e. channels are unidirectional).

Channels are treated in the same manner as *Modula-2* variables i.e., they have a type and must be declared. The base type of the channel may be any other *Modula-P* type. A type and a variable declaration looks like:

```
TYPE tChannel : CHANNEL OF BaseType
VAR  channel1 : tChannel
VAR  channel2 : CHANNEL OF BaseType
```

Before the first communication over a channel can take place, the channel must be opened once with the standard procedure `OPEN`. This is usually done by the parent process of the communication partners. A channel remains opened as long as the opening process continues to execute. Its signature is:

`PROCEDURE OPEN (VAR channel:` *ChannelType*`)`

For variables of type channel, input and output operations are defined. The statement

`channel ! expression`

outputs the value computed by *expression* to *channel*. The statement

`channel ? variable`

```
CompilationUnit   = ... | ProcessModule.
LocalProcessDecl  = StatementSequence.
ProcessModule     = PROCESS MODULE ident [FormalParameters] ";" {import} block ident ".".
ParStatement      = PAR Processes {"|" Processes} END.
Processes         = [[replicator] LocalProcessDecl [Placement] [";"]]
                  | [[replicator] GlobalProcess [Placement] [";"]]
GlobalProcess     = ProcedureCall [";"].
Placement         = AT expression.
```

Figure 2: Syntax of process declarations and the **PAR** statement

reads a value from *channel* and assigns it to *variable*. The *variable* and the *expression* must be assignment compatible with the base type of the channel. A channel variable may be passed to a procedure or may be assigned to other variables.

Any type of message may be passed over a **CHANNEL OF ANY**[5], using only the size of the message and the address of the source / destination.

```
TYPE tCh = CHANNEL OF ANY
VAR ch1  : CHANNEL OF ANY
VAR ch2  : tCh
ch ? dest_adr, size
ch ! src_adr, size
```

dest_adr and *src_adr* are expressions which have to be of type **ADDRESS**. *src_adr* gives the address of the message to be send. *dest_adr* gives the memory address where the received message should be stored. *size* is an expression which must be assignment compatible to **CARDINAL**. *size* specifies the length of the message in bytes. The message size of the sender and receiver must match. The receiver process has to provide enough memory for storing the received message before the communication takes place. This message size may change from one communication to the next. This form of the **!** and **?** statement is allowed only for channels of type **CHANNEL OF ANY**. A **CHANNEL OF ANY** may be used at any place, where a **CHANNEL OF** *BaseType* may be used and the same rules apply.

TIMER is a special predefined channel from which the actual system time may be read. **TIMER** need not be opened. The base type of this **TIMER** channel is the new scalar type **TIME**. Values of type **TIME** may be compared to equality (**=**) and unequality (**#**). The new standard module *SysTime* specifies further operations for objects of type **TIME**.
A process may suspend itself from execution for some time executing the delay statement:

```
TIMER ? AFTER expression
```

The process continues, if the system time is later the the time specified by the *expression* (of type **TIME**).

5.5 The ALT Statement

The **ALT** statement may be used for simultaneous waiting for several events. Events may be communication with other processes or time events. The syntactic structure is:

[5] Notice: **ANY** is not a language type like **WORD**; it is a keyword.

```
type                 = ... | ChannelType | TIME.
ChannelType          = CHANNEL OF BaseType
                     | CHANNEL OF ANY.
BaseType             = type.
ChannelStatement     = InputStatement
                     | TimerInputStatement
                     | DelayStatement
                     | OutputStatement.
InputStatement       = channel "?" designator
                     | channel "?" adr, size.
TimerInputStatement  = TIMER "?" designator.
DelayStatement       = TIMER "?" AFTER time.
time                 = expression.
OutputStatement      = channel "!" expression
                     | channel "!" adr, size.
channel              = designator.
adr                  = expression.
size                 = expression.
```

Figure 3: Syntax of the channel operations

```
ALT
    guard_1 : stmts_1
  | guard_2 : stmts_2
    ...
  | guard_n : stmts_n
  ELSE stmts_0
END
```

The process executing the ALT statement is suspended until one of the guards $guard_1, \cdots, guard_n$ is ready. From the set of ready guards, one arbitrary $guard_i$ is selected, and the corresponding statements $stmts_i$ of the alternative are executed. There are three types of guards: *simple guards*, *channel guards*, and *time guards*. They look like:

```
bool_expression
bool_expression , channel ?  variable
bool_expression , TIMER ? AFTER expression
```

The *bool_expression* may be omitted for channel and time guards. The *bool_expression* will be evaluated first if it is present; if it is omitted, it is assumed to be TRUE. A guard is ready if the evaluation of the *bool_expression* yields TRUE, and

- for the simple guard, no other condition is necessary,
- for the channel guard, another process waits for communication over *channel*,
- for the time guard, the actual system time is later than the time specified by *expression*.

Before executing the statements of a selected channel alternative, the communication over this channel takes place. The `ELSE` part of the `ALT` statement may be omitted. If it is present and none of the *bool_expressions* is evaluated to TRUE the statements $stmts_0$ are executed. If the `ELSE` part is omitted and none of the *bool_expressions* is evaluated to `TRUE` a runtime error is raised.

5.6 Replicators

The components of a PAR or ALT statement may be replicated. A replicated process component has the form:

`[ident : lower TO upper] p`

(ORD(*upper*) - ORD(*lower*) + 1) processes are started all executing *p* in parallel. Each process gets a unique value of *ident* in the range [*lower* ··· *upper*], which is accessed using *ident*.

Similarly a replicated alternative has the form:

`[ident : lower TO upper] guard : stmts`

(ORD(*upper*) - ORD(*lower*) + 1) alternatives are set up, waiting for the guards. Each alternative gets a unique value of *ident* in the range [*lower* ··· *upper*]. Constraints for the use of the replicator variable *ident* are: The type of *ident* must either an enumeration, or an integer or cardinal type. The type of *lower* and *upper* must be assignment compatible to that of *ident*. The replicator variable *ident* is not allowed to be a component of a structured variable, nor may it be imported or a parameter of a procedure. The replicator variable may be used inside of the child process or the alternative only like a constant. If (ORD(*upper_bound)* - ORD(*lower_bound*)+1)≤0 then no component (process or alternative) is created.

```
replicator = "[" ident ":" lower TO upper "]".
lower      = expression.
upper      = expression.
```

Figure 5: Syntax of the replicators

5.7 Other Extensions

Modula-P uses the other extensions of the *Modula-2* system *MOCKA* it is based on. There are additional long and short integer, cardinal and real types.

There is another module kind called `FOREIGN MODULE`, which is like a definition module, but the procedures specified there may be implemented in other languages, for example assembler or *C*. A `FOREIGN MODULE` has no Modula `IMPLEMENTATION MODULE`; only constants, types and procedures may be declared. Procedures declared there may be called like procedures declared in a `DEFINITION MODULE`.

Modula-2 allows the assignment of priorities to program modules, this is not supported by *Modula-P*.

6 Program Examples

This section presents some program stubs to give an impression of the language. They show also, how *Modula-P* programs are "scalable" just by increasing a constant (or a variable) which specifies the number of channels / server processes to be used.

6.1 Simple Producer / Consumer

The program example in figure 6 shows a how a simple producer, consumer problem may be expressed. Three processes are started, two of them producing data, and sending them over channels to the process which gathers the information and gives them to one consumer. The `ALT` statement is needed, because the data must be given sequentially to the consumer (for example a printer device). If the processes are too slow, a message is printed, caused by the time guard. Replication is used in the `PAR` and `ALT` statement.

```
MODULE prod_cons;
  FROM netIO IMPORT WriteString;
  PROCEDURE produce (i:INTEGER):INTEGER; ...
  PROCEDURE consume (value:INTEGER); ...
  VAR chn : ARRAY[1..2] OF CHANNEL OF INTEGER;
  VAR i, j, value:INTEGER; time:TIME;
BEGIN
  OPEN (chn[1]); OPEN (chn[2]);
  PAR
     [i : 1 TO 2] LOOP chn[i] ! produce(i) END
   | LOOP
       TIMER ? time;
       ALT
          [j : 1 TO 2]
             chn[j] ? value : consume (value)
        | TIMER ? AFTER time + 100 :
             WriteString ("time out")
       END
     END
  END
END prod_cons;
```

Figure 6: Program example producer / consumer

6.2 Pipeline Processing

The example in figure 7 shows a parallel program that computes prime numbers. It seems silly to compute prime numbers in this way, but this example shows how to construct an arbitrary large pipeline of processes using recursion. One can think of two kinds of processes: a `generator` and several `worker` processes chained by channels. The generator produces odd numbers and sends them to the first worker (which has number 2). Each worker process represents a prime number. It reads numbers from its input channel, where the first number is its prime number. If a number read is divisible by the worker's own prime number, this number is not prime; hence, it is discarded. Otherwise it must be sent to the next worker process for further examination. A negative number is interpreted as the termination signal. When the `worker` process is called two further processes are started. One of them checks the numbers read. The other recursively starts the next worker, which waits for input of its prime number or the termination signal, sent by the checking process.

```
AltStatement = ALT alternative { "|" alternative } [ELSE StatementSequence] END.
alternative  = [[replicator] guard ":" StatementSequence].
guard        = expression | [expression ","] InputStatement | [expression ","] DelayStatement.
```

Figure 4: Syntax of the `ALT` statement

```
MODULE p_prime;
FROM defs IMPORT IntChannel;
(* TYPE IntChannel = CHANNEL OF INTEGER *)
FROM netIO IMPORT ReadInt;
IMPORT worker;
VAR max, i: INTEGER; out : IntChannel;
BEGIN
  netIO.ReadInt (max);
  OPEN (out);
  PAR
     worker (out) (* start first worker *)
   | (* test number generator *)
     out ! 2; (* first prime number *)
     FOR i:=3 TO max BY 2 DO out ! i END;
     out ! -1 (* termination *)
  END
END p_prime.
```

```
PROCESS MODULE worker (in : defs.IntChannel);
FROM defs IMPORT IntChannel;
FROM netIO IMPORT WriteInt; FROM net IMPORT left;
VAR out : IntChannel; prime, x : INTEGER;
BEGIN
  in ? prime;
  IF prime < 0 THEN RETURN ELSE WriteInt(prime) END;
  OPEN (out);
  PAR
     LOOP (* read numbers, -1 terminates *)
       in ? x;
       (* pass termination signal and terminate *)
       IF x < 0 THEN out ! -1; EXIT END;
       IF x MOD prime # 0 THEN out ! x; END;
     END;
   | worker (out) AT left() (* start next worker *)
  END;
END worker;
```

Figure 7: The *pipeline* program example

7 The Standard Library

There is a set of standard modules supporting the *Modula-P* language features. The important ones are: the `net` module which provides information about the computer network. Procedures like `left`, `top`, etc.[6] allow the specification of a process topology independently of the actual network topology[7]. The module `netIO` offers procedural input / output facilities in the well known fashion of the *Modula-2* `InOut` module . `netIO` contains procedures for reading and writing to the standard input/output devices, as well as to arbitrary files and windows. This may be done from each process running on an arbitrary processor. `Storage` is used to allocate and deallocate dynamic storage. The `SysTime` module offers procedures to compare `TIME` values and to compute out of `TIME` differences a real time. The `channels` module specifies the kind of additional information to be passed to a call to the `OPEN` standard procedure. To implement critical regions, simple boolean semaphores are offered by the `Mutex` module. The `TRANSPUTER` module offers direct access to some of the Transputer's machine instructions. The compiler recognizes this module, and inserts the instructions directly.

8 Assumptions for the Implementation

We have made the following assumptions about the network for our runtime system. They are all fulfilled by a Transputer based network: the network supports only point-to-point communication. The number of processors is known when a program starts. Each processor has a unique identifier. The number and identifier don't change during the program execution. The communication among processors over their physical channels is error-free, i.e. no messages are lost, duplicated or changed in any other way. All processors are available during program execution, and don't fail. Hence, no fault detection is required. The topology of the network is known when starting a program, and doesn't change during the program execution. The hardware or software supports concurrent execution of several processes on a single processor (this may also be in a time-shared way).

9 Problems with Channels

Considering these assumptions, what difficulties arise when designing a channel communication model? To answer this question, we examine the behavior of *Modula-P* channel in detail. It is usually not known, on which processor global processes are executed. Why does this influence a channel communication model? Since processes may have parameters which are transmitted to the destination processor, channels passed as parameters are affected also by the distribution, when processes are started remotely (see figure 8). But unlike the ability to notice when a process has to migrate it is not possible to notice the migration of channels. Moreover, channels are bound neither to processors nor to processes. One channel may be used by different processes successively. For example(see figure 8), after a process has communi-

[6] If there is such a neighbor processor these procedures return the number of that neighbor, otherwise another neighbor number is returned.

[7] A network of this topology may increase only the execution speed.

cated over a channel, this channel may be used by a child process. After termination of this child process, the parent can use this channel again. Since two partners are involved in a channel communication, this migration problem arises for both.

Hence, the usual solution leaving forward references when a channel migrates can't be used. So, another solution has to be found.

```
PAR Process1 (ch) | Process2 (ch) END;

Process1          Process2          Process3
   :                 :                 :
LOOP              ch ? y;           ch ? z;
  ch ! x;         PAR                  :
   :                Process3(ch);
   :               | ...
END;               END;
   :               ch ? y;
                      :
```

Figure 8: Migrating channels

10 The Tasks of MOPS[8]

Some special problems arise implementing *Modula-P* on a distributed computer system. Since a programmer should not be aware of the implementation of the channel communication in the network, the runtime system, called *MOPS*, has to support this communication transparently. Moreover, the distribution of global processes and the remote start and termination of such processes must be done automatically. The main task now is to design a model for communication over channels and starting processes.

10.1 Channel Agent Model

As a channel may migrate between any two communications, one has to know the new sender and receiver location, respectively, before exchanging messages. The exchange of the current identifier of communication partners is done using an *agent*. Our way is to subdivide a channel communication into four parts (see figure 9).

Assume a sender arrives at its output operation first, but it doesn't matter which partner reachs a channel operation first.

1. The sender sends a message *RequestToSend* to the channel agent, containing its identification. Then it waits to receive the partner identification send by the agent.
2. When the receiver has reached the corresponding input operation it sends *RequestToReceive* to the agent containing its identification. Now it waits for the proper data from the sender.
3. After having both identifications the agent can pass the receiver identification to the sender and its work is done for this time.
4. Having received the partner identification the sender is able to transmit the data directly to the receiver.

After the data exchange, both processes may again run independently. This procedure has to be done every time a channel communication takes place and guarantees that the semantics (i.e. synchronicity) of *Modula-P* channel communication is fulfilled.

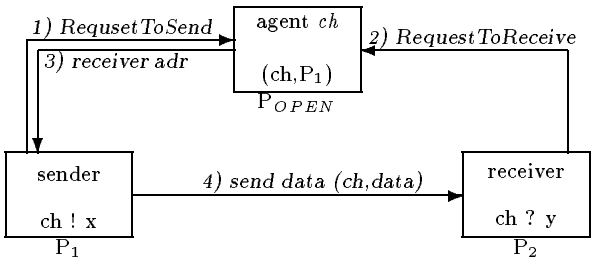

Figure 9: Communication model using an agent

For this model, both communication partners must know the identification of the agent of the channel before each *Modula-P* communication takes place. The only requirement of channels is that they must be opened before they are used. Hence, an agent for a channel may reside on the processor where the channel was opened. When opening a channel the processor identifier is stored in the internal structure of the channel. If a channel migrates, this information migrates too. As a side effect, agents of different channels may be distributed over the entire network, depending on the dynamics of a program.

Next we consider the case of a channel being used as an input guard contained in an **ALT** statement. During the time the receiver process waits for a channel guard, the information that a sender is ready has to be passed to this receiver process, so it can continue, i.e. select a sender and execute the corresponding alternative.

The agent protocol for this case works as follows (see figure 10):

1. The receiver process send the message *EnableChnl* to all agents of the channels involved in this **ALT** statement. Then the receiver process suspends.
2. If a sender process now becomes ready to send a message, it sends *RequestToSend* to its agent (see above). Perhaps, the agent already received this message.
3. If the agent knows that there is a ready sender it informs the receiver by sending *SReadyForAlt*. The receiver is awakened and sends to all agents a *DisableChnl*, which informs the agents to send no more *SReadyForAlt*. If there are more than one *SReadyForAlt* messages present, they are ignored.
4. The next step is to choose one ready sender out of the set of ready senders. Then, the message is passed (as described above) and the statements of the chosen alternative are executed.

[8] Modula-P Operating System

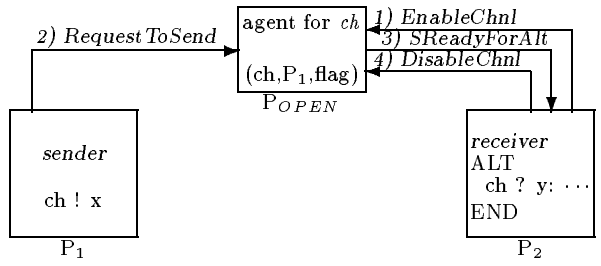

Figure 10: Selective waiting using an agent

10.2 Starting of Processes

Starting and executing of local processes is mapped onto the underlying system. Starting the execution of a global process on an other processor is mapped onto communication between the MOPS running on the different processors. Several things have to be done therefore:

1. If the user has not specified a placement, it has to be decided where the global process should be executed.
2. A message is send to the MOPS of that processor.
3. The parameters of the global process are sent to that processor. A special treatment is needed for open array parameters, since their size is not known on compile time. Hence, the global process first requests to receive them (via MOPS) before the ordinary process statements are executed.
4. On the destination processor, the global process is executed similarly to a local one, except that storage for the global variables must also be allocated.
5. After termination, a message is sent back to the parent process.

To decide on which processor a global process is executed, two very simple strategies are currently implemented. They do not use any load information of the network. One strategy is to execute the processes on the entire network, the other is to execute a process only on its direct neighbor. The latter strategy reaches after some time the same result as the first, but child processes are executed at the direct neighbor.

10.3 Routing Messages through the Network

Due to our assumptions, the network is connected in a point-to-point fashion. Hence, communication must be routed over other processors if the communication partners are not direct neighbors. Again it is the task of MOPS to do this in a transparent way, by exchanging routing-messages and the data between the incarnations of MOPS running on different processors. This is done in a "store and forward" manner.

Since the network topology is known and fixed, a static, deterministic router based on a shortest path algorithm is used. The path information is computed off-line before the execution of a program and is down loaded to each processor while initializing the network. So the routing information doesn't change during execution and making a decision is just a table lookup. Since Transputers use synchronous links, no flow control has to be done. We omit congestion control for efficiency reasons, and use the entire processor memory to store messages. Nevertheless, no deadlocks (waiting for packets in a cyclic manner) can be produced by MOPS itself. A more detailed discussion of routing algorithms may be found e.g. in [10].

11 Implementation of MOPS

A main aim was to let MOPS work absolutely transparently. It is an entirely distributed system running on every processor of the network as processes independently from the user processes. Agents are represented by MOPS and a channel data structure. Under MOPS, programs can run on different Transputer network configurations without any change in the source file. Furthermore, no recompilation depending on the network is necessary. A complete discussion of the implementation is found in [11, 5].

11.1 The Transputer

For the implementation, the features offered by the Transputer [7] are heavyly used. A Transputer system consists of a number of interconnected Transputers, each executing several processes. The Transputer hardware offers a scheduler, which executes several processes in a time shared manner on two priority levels. Special instructions are used to start and terminate processes. Each Transputer has four bidirectional hardware *link*s which are used to connect the Transputers in a point-to-point way. It has instructions for synchronous communication via physical channels between processes. A physical channel is either a word of memory (memory channel), or the special address of a hardware link (external channel, link). Instructions for implementing the `ALT` statement. The "usual" integer and floating point arithmetic operations.

11.2 Overall Structure of MOPS

MOPS consists of four parts (see figure 11):

1. MOPS has a *procedural interface* and calls will be inserted by the compiler. These procedures construct *commands packets* which are passed to the *command handler*.
2. The communication between Transputers, as well as the routing of command packets, is done by the *router*. The router is the only part which has access to the links of Transputers.
3. A *command handler* which decodes the command packets and implements so the agents. Also it starts global processes. The command handler may obtain command packets either a user processes running on the own Transputer via the user port or from the router.

4. Management of *processor-global data*. Since Modula-P doesn't offer processor global data[9], a separate handling has to be done, for example for routing information.

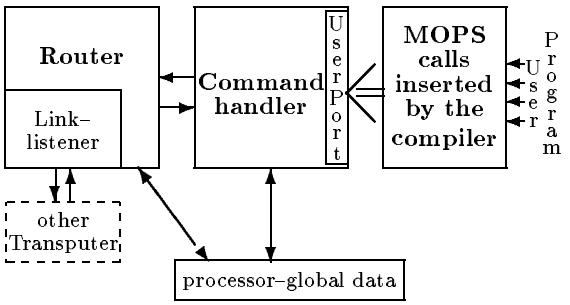

Figure 11: Structure of MOPS

When sending a packet from one Transputer to another currently no packaging is done for large data packets.

To reduce the overhead produced by MOPS no special data structures with the exceptions of link buffers are maintained.

All information about channels is held in the internal structure of a channel itself. A channel is uniquely identified throughout the network by the address of the variable used to open a channel together with the corresponding Transputer identification.

The *router* processes are started initially, after the program is loaded to the processor. They run on high priority for two reasons, first forwarding of messages should be fast and have more precedence than the user program, which runs in the low priority mode. The second reason is that accessing global data shared by the *LinkListener*s (mainly the buffers) may be done without any synchronization, because high priority processes may not be descheduled, until they wait for communication.

To pass information (command packets) from the user program to the router, communication over Transputer memory channels are used. This is the only way a low level and a high level process may communicate without polling a certain data structure.

11.3 The Router

Now a closer look at the structure of the router is given (see figure 12).

To pass messages a store and forward strategy is used to decouple the reading from links and writing to links. This avoids circular deadlocks while routing packets. Packets reaching the router may be produced on the own Transputer or come from another one. For every link there's a *LinkGuardian* consisting of two processes, a *link reader* and an *output processor*. Packets comming from another Transputer are received by the link readers. After reading a packet a table lookup is made to decide what to do. If it's for the own Transputer the packet is directly passed to the command processor otherwise is put to the appropriate output buffer. Then a new packet may be read immediately. Processing the output buffers is done by output processors using a FiFo strategy. Until a buffer is empty a packet is taken from it and sent over the corresponding link. When a buffer is empty the process is stopped and is not restarted before a new packet is put to the buffer.

[9] The MOCKA-P assembler offers access to processor global data

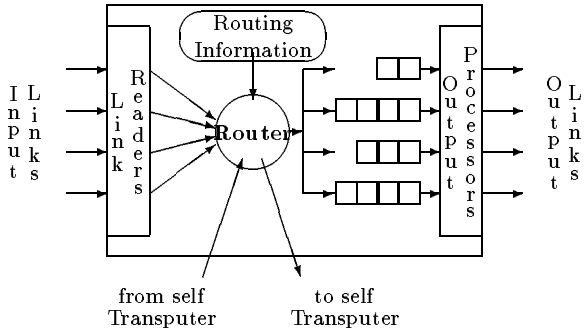

Figure 12: Router

11.4 Executing Processes

For local processes, the execution is started using the machine instructions the Transputer offers. Since the parent process is suspended until all child processes have terminated, it must be informed when this has happened. Again, the Transputer offers instructions to do this.

To identify at runtime the code of a global process, each process module receives a unique number during linking. Using unique numbers has the advantage of being independent of the code memory layout. So, different implementations of a library can be linked for different Transputers.

Due to the dynamics of *Modula-P* processes, the code of entire program is kept on every Transputer. Since the code is reentrant, each module is present only once in the storage.

Since each global process has its own incarnation of all imported global variables, a kind of "virtual memory" must be implemented. When a global process is started, memory for all variables accessed by the process is allocated. After termination of the process, the allocated storage is released again. The imported variables of a global process may be a subset of all declared global variables, since a global process may import only a subset of all modules needed by the entire program. The amount of memory needed for a global process is determined when linking the entire program. The needed memory is allocated as one

large block when starting a global process. Access to the global variables is done now using a table, which maps a module number to the address of of a subblock in that block, which contains all global variables declared in that module. This table must be computed after the storage is allocated.

When a process is started, the process gets a piece of workspace as runtime stack for storing local variables and parameters. If necessary this stack is extended, when a called procedure does not fit into the current workspace piece. Such a piece is deallocated after the procedure allocating it returns.

12 Some Benchmark Results

To get an idea of the efficiency of the implementation of MOPS we show some measurements taken with our system (10Mbit/sec–T800 Transputer network).

We compared communication based solely on the Transputer instructions `in` and `out`, on the agent model and on the Transputer operating system Helios^{TM} [9][10]. Only communication is done by the processes and processors.

The benchmarks use messages with sizes in the range form 1 byte up to 10000 bytes. The messages are send from a process P1 to a process P2 and then back to P1. The times are read from the processor P1 is executed on and divided by 2.

The results are shown in figure 13, 14, and 15. *MOPS A0* (*A1 / A2*) indicates that the agent is executed on the same (a direct / indirect neighbor) processor. The curve *move instr* shows how fast pure memory copy works.

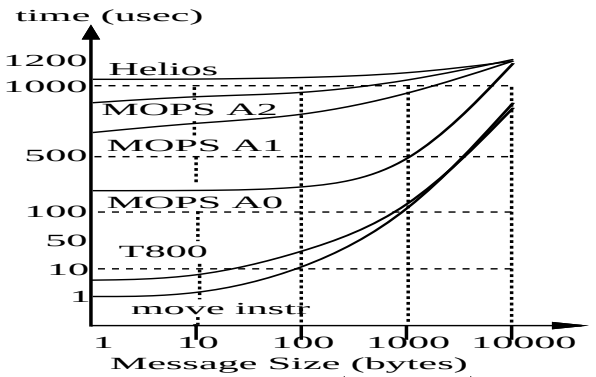

Figure 13: Sender, receiver (and agent) on same Transputer

If the message becomes longer, the pure communication time overrides the communication overhead caused by the systems. Additionally, HELIOS breaks large messages into smaller pieces. The resulting pipeline effect speeds up HELIOS over pure Transputer communication. The more powerful MOPS communication scheme is often faster or even as fast than HELIOS (e.g. 2-5 times for all running on the same Transputer), but slower than the pure Transputer instructions. A strong influence on the MOPS times has the distance of the agent process to the partners.

[10] Similar C programs are used therefor.

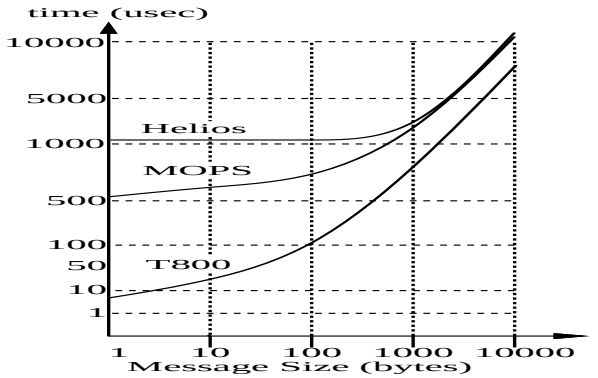

Figure 14: Sender, receiver on direct neighbor Transputer. Agent on the one of the partner processors.

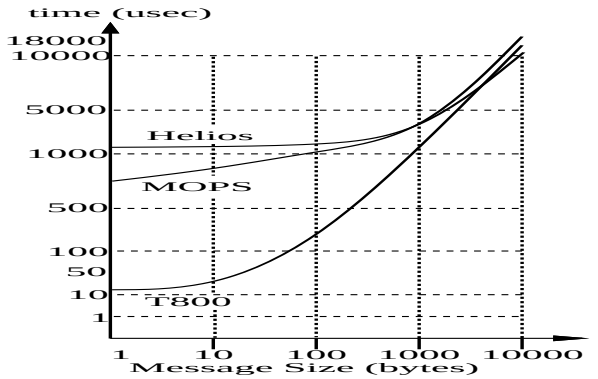

Figure 15: Sender, receiver on indirect neighbor Transputers. Agent on the one of the partner processors.

We compared *Modula-P*'s local to its global processes by measuring the time needed to start and terminate "empty" processes. With our scheduler the overhead of starting a global processes is about a factor of seven to local ones. Notice, the global processes are started on all processors contained in the net. This overhead can be neglected if processes do heavy computing which may be seen in diagram 16. Except of very short programs all programs examined using global processes are executed faster than the corresponding ones running just on one Transputer.

13 Optimizations

These measurements give hints which optimizations should be performed: treat the case where both communication partners reside on the same processor in a special way. Another optimization is to avoid the agent requests. This is possible, if it is known that the migration problem doesn't arise. To decide the applicability of these optimizations, data flow analysis or additional parameters to the `OPEN` procedure

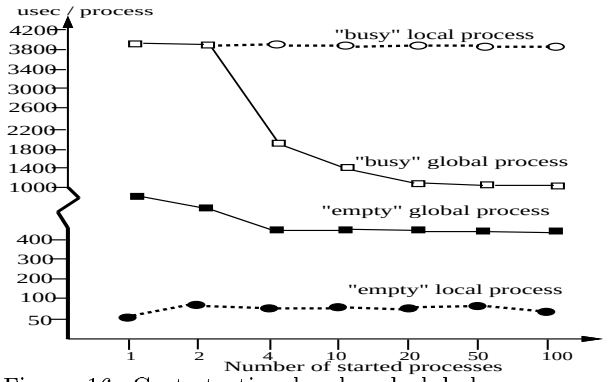

Figure 16: Cost starting local and global processes

may be used.

14 MOCKA-P

MOCKA-P[11] is the name of *Modula-P* Transputer development system [11]. It is based on the *Modula-2* system *MOCKA* developed at GMD. The entire system is written in *Modula-2* and *Modula-P* and some lines of Transputer assembly language. *MOCKA-P* consists of the *Modula-P* compiler *mpcc*, which generates T800 symbolic code. The assembler *astra* which generates T800 binary code. It has a procedural interface used by the code generator and a textual interface for hand written assembler programs. The linker *litra* links the binary object files and produces an executable Transputer program. *Extra* initializes the Transputer network, loads the executable code, and performs all input / output to the host computer and operating system (UNIX). *Extra* runs on the host computer and is active as long as the Transputer program runs. Errors messages from the compiler are presented together with the source text. An important role plays MOPS, the *Modula-P* Operating System. Automatic (re)compilation of all modules is controlled by *Merlin*. The *Modula-P* debugger *mpdb* allows it to observe and debug a program running on a network of Transputers.

The compiler, assembler, linker run as a cross system on different UNIX platforms, while the *Modula-P* program is executed on our Transputer board which is plugged into a UNIX workstation.

15 Future Work

Future work mainly concerned with optimizing the agent model, and porting it to the new Transputer *T9000*. A non-Transputer based system is under development.

16 Summary

The programming language *Modula-P* and the development system *MOCKA-P* offer a very powerful programming environment for Transputer-based parallel computers. The programmer gets a clear and intuitive model of parallelism, which is soundly based on the theory of CSP. As usual for modern programming languages, *Modula-P* supports programming "in the small" as well as "in the large", and frees the programmer from details allocating hardware resources. Measurements show that the system running on a Transputer network is very fast.

[11] Modula-P Compiler KArlsruhe

Acknowledgements

Thanks to the students Markus Armbruster, Claas Hinrichs, Jens Hübel, and Jürgen Richter who did parts of the implementation.

References

[1] Ada. *Reference Manual for the Ada programming language (ANSI / MIL - STD 1815A).* 1983.

[2] P. Brinch Hansen. Joyce language report. *Software–Practice and Experience*, 19(6):553–578, June 1989.

[3] M. Collado, R. Morales, and J.J Moreno. A modula-2 implementation of csp. *ACM SIGPLAN NOTICES*, pages 25–38, June 1987.

[4] C.A.R Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice–Hall, Inc., 1985.

[5] Ralf Hoffart. Übersetzung einer parallelen Programmiersprache für ein verteiltes Rechnersystem – Modula-P auf Transputernetzwerken. Master's thesis, Universität Karlsruhe, March 1991.

[6] INMOS, editor. *occam Programming Manual.* Prentice–Hall, Inc., 1984.

[7] INMOS, editor. *The Transputer instruction set - a compiler writers' guide.* Prentice–Hall, Inc., 1988.

[8] Mehdi Jazayeri, Carlo Ghezzi, Dan Hoffman, David Middleton, and Mark Smotherman. CSP/80 a language for communication sequential processes. IEEE Compcon, 1980.

[9] Perihelion, editor. *The helios operating system.* Prentice–Hall, Inc., 1989. Perihelion Software Ltd.

[10] Andrew S. Tanenbaum. *Computer Networks.* Prentice–Hall, Inc., 2nd edition, 1989.

[11] Jürgen Vollmer. Kommunizierende sequentielle Prozesse in Modula-2; Entwurf und Implementierung eines Transputer – Entwicklungssystems. Master's thesis, Universität Karlsruhe, May 1989.

[12] Jürgen Vollmer. Modula-P, a language for parallel programming. *Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia*, pages 75–79, 1989.

[13] Niklaus Wirth. Schemes for multiprogramming and their implementation in modula-2 – revisions amd amendments to modula-2. ETH Zürich, Institut für Informatik, 1984.

[14] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, Berlin, Heidelberg, New York, Tokyo, third, corrected edition, 1985.

[15] K.L. Wrench. CSP-i : An implementation of communicating sequential processes. *Software–Practice and Experience*, 18(6):545–560, June 1988.