North-Holland
Microprocessing and Microprogramming 24 (1988) 519–524

The communication processor of TUMULT-64.

Gerard J.M. Smit,
Pierre G. Jansen

Twente University of Technology
Department of Computer Science
P.O. Box 217
7500AE Enschede, The Netherlands

Abstract

Tumult (Twente University MULTi-processor system) is a modular extendible multi-processor system designed and implemented at the Twente University of Technology in co-operation with Oce Nederland B.V. and the Dr. Neher Laboratories (Dutch PTT). Characteristics of the hardware are: MIMD type, distributed memory, message passing, high performance, real-time and fault tolerant. A distributed real-time operating system has been realized, consisting of a multi-tasking kernel per node, inter process communication via typed messages and a distributed file system. In this paper first a brief description of the system is given, after that the architecture of the communication processor will be discussed. Reduction of the communication overhead due to message passing will be emphasized.

Key-words: Distributed operating system, real-time, fault tolerance, performance, message passing.

1. Introduction

Due to the availability of powerful, low cost micro-processors, the cost performance ratio of multi-processor systems decreases and new applications become feasible. This paper will a.o. address the problem associated with inter-processor communication overhead of multi-processor systems with distributed memory such as the Tumult system. This is one of the major problems the designers of multi-processor systems with message passing communication are faced with to day.
For the purpose of this introduction the Tumult system is defined as a collection of computing nodes interconnected with a fast network. The nodes do not share memory. Message passing across the network is the only mechanism for communication between processors.

Tumult belongs to a class of systems in which parallelism is programmed explicitly by the user. He is responsible for the applied parallelism in his application program. Therefore the Tumult operating system supports functions to create processes and dynamic communication structures between the processes. The user has complete control over the amount of parallelism for his application. However if the granularity of the parallelism is too fine the communication overhead will degrade the performance. On the other hand a large grain size limits the amount of parallelism [Kruatrachue 88].
For optimal load balancing a trade off has to be made between communication and parallelism.
In a certain way the communication overhead determines the practical "grain size" of a parallel machine. The lower the overhead the more freedom an application programmer has to choose the degree of parallelism.

From the discussion above it will be obvious that for a multi-processor architecture with message passing a low overhead of the basic messages is essential.
In this light, the factors that contribute to the communication overhead are of interest. Besides algorithmic limitations there are two important factors. First there is an overhead associated with **data transfer**: moving data from one place to another. This is a function of network parameters, such as bandwidth, network diameter, single/multi path, etc. The second factor is associated with **protocol overhead**. No message can be received before it is sent, so synchronization and context switches are inherent. In addition message passing needs queuing, flow control, some form of protection (type checking), exception handling etc.

Our experiments showed that the limiting factor, especially for small messages, is typically not data transfer overhead but protocol overhead. This has been confirmed in two recent papers [Scott 87], [Ramachandran 87].
In [Scott 87] performance figures for message passing overhead of a number of distributed operating systems are given. The overhead ranges from tens of milliseconds to 1.26 ms for the V-kernel [Cheriton 85], which puts great emphasis on performance.
Note: these figures have to be interpreted with caution, because they cannot be compared as such. It is important to define what is measured. Comparisons between the systems are not only limited to differences in organization

and speed of the underlying hardware, but also by differences in functionality of the supported primitives. In some systems the functionality is very restricted: no type checking, no abort possibility in case of errors, only fixed size message etc.

The initial implementation of the Tumult system (Tumult-15) resulted in a considerable overhead, notwithstanding the high bandwidth of the network (at least 20 Mbytes per second). (note: Tumult-# stands for a Tumult system with # number of nodes). Hence the overhead involved in data transfer can be neglected for small messages (<1000 bytes). It will be no surprise that decreasing the protocol overhead is stressed in this paper. The target of the overhead in Tumult-64 is in the order of magnitude of 200usec.
For that reason a special purpose co-processor to support message passing is designed and is presented in this paper.

Section 2 gives an overview of the Tumult system. Section 3 describes the hardware and the basic communication protocol of Tumult-15 and provides some performance figures. In section 4 the communication co-processor of Tumult-64 is presented.

2. Overview of Tumult

The purpose of this section is to give an overview of Tumult-64. It provides as much information as needed to understand the remainder of the paper. This section is based on [Jansen 88] where an extensive survey is given. Fig. 1 shows the architecture.

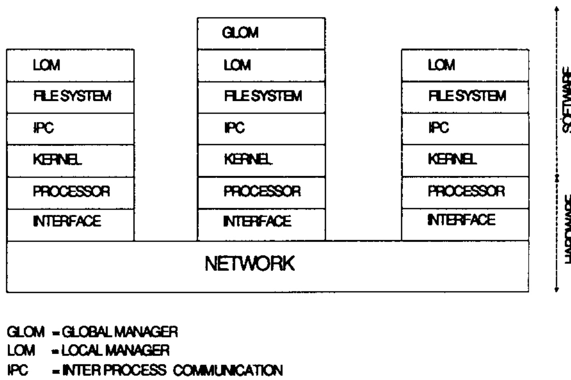

Fig. 1. The layers of the tumult system.

According to [Hockney 85] Tumult can be characterized as a MIMD computer with distributed memory of which the processing elements (nodes) are interconnected via a bi-directional ring network. The network transfers single (16 bits) data- or control words. The data words are used for the actual data transfer. The control words are used for controlling a data transfer: claiming resources, flowcontrol, error recovery, etc. Section 3 and 4 describes the hardware in more detail.

Note: in the current implementation the system is not distributed geographically; the operating system however allows for geographical distribution without any changes.

The operating system is written in Modula-2 [Wirth 85]. It is structured according to the layers shown in fig.1.
An efficient *real-time multi-tasking kernel* [Luttmer 87] runs at each processor node. It offers primitives such as memory allocation, process creation and -termination, deadline scheduling, interrupt handling, and exception handling.

The *interprocess-communication* (IPC) layer allows for dynamic creation and deletion of logical communication links , shortly referred to as links. A link is a flexible communication structure, that can be created by processes and can be adapted to the current communication needs of the system.

The number of processes connected to link can be changed dynamically, i.e. processes can be added to-, or removed from the link by *connect* or *disconnect* commands. These commands change implicitly the link topology. Links offer transparency of locality for connected processes: processes do not need to know each others physical location in order to communicate.
Links are identified by a system wide unique name. It serves as a key (or password) for the use of the link. A process can refer to a link via *a port* (like a file-pointer for files).

There are two types of links:
- Message Transfer (MT) links
- Remote Procedure (RP) links.

In this overview only Message Transfer links are described, for Remote Procedure links we refer to [Jansen 88].
A Message Transfer link is a uni-directional "many to any" communication topology, which enables "many" sender processes to transfer typed records to "any" receiver process, that is connected to the link.
In order to increase the amount of parallelism a MT link may support buffered communication. The number of buffers is determined at application level. This offers the possibility of avoiding deadlock in circular communication structures. If the number of buffers is declared zero, the communication proceeds synchronously.

Links can be manipulated with the following system functions:

```
CreateMTLink ('LinkKey', <record type>,
        NrOfBuffers);
RemoveMTLink ('LinkKey')
Connect (xp,'LinkKey');
Disconnect (xp);
Send (sp, f);
Receive (rp, a);
```

Where "LinkKey" is the unique key of the link. <record type> gives a type indication of the records to transfer. "NrOfBuffers" determines the number of buffers (of <record type>). These buffers are allocated at the node of the process that calls CreateMTLink. If "NrOfBuffers" is zero, all communication via this link will proceed synchronously.
"xp" is either a sender-port or a receiver-port. These ports have to be declared by a sender- or receiver process.
"f" denotes the result of an expression to be sent and "a" denotes the variable in which a result has to be received. The types of "f" and "a" must correspond. Before a link

can be used, exactly one process calls the CreateMTLink function. Thereafter any number of processes can connect to the link, by using the Connect primitives. If a node is connected successfully it can send or receive messages via the link. If the process does not need the link anymore, it disconnects and if no processes are connected the link anymore it can be removed with a RemoveMTLink function.

A *distributed file system* allows files and devices to be distributed over the nodes transparently [Langen 87]. The file system inherits its dynamic behavior from the IPC-primitives and allows for the dynamic installation of devices and files.

The *Local Manager* receives, interprets and executes commands from the Global Manager, such as load - , start - , or terminate a (sub) tasks, and it collects local status information .

The *Global Manager* interprets commands from console for the execution of (parallel) tasks. A task includes one or more different sub-tasks. The GM distributes the sub-tasks over the nodes and orders the LMs to execute them. The GM also collects global status information.

3. Implementation of Tumult-15.

In this section an overview of the hardware of Tumult-15 will be given. Tumult-15 (a predecessor of Tumult-64) is discussed first, in order to provide the essential background for understanding the design decisions of the communication-processor for Tumult-64. This processor will be presented subsequently in section 4. A complete description of Tumult-15 is given in [Jansen 87].
The organization of a node is shown in fig. 2.

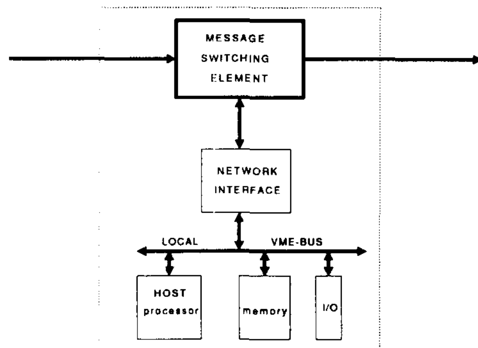

Fig. 2 Node organization of Tumult-15.

The hardware of Tumult-15 consists of up to 15 nodes that communicate via a modular extendible uni-directional ring network. Each node contains a Message Switching Element (MSE), a Network Interface (NI) and at least one Host processor with local memory. The Network Interface is connected to the Host processor(s) via a VME bus.

The MSE has two inputs and two outputs and is capable of transferring a message from any input to any output. A message consists of: destination field (4 bits), control field (7 bits) and a data field (16 bits). If two messages are offered to the same output, the message from the ring-input has priority. By connecting the MSE's in a ring structure a time slotted (store forward) ring network is formed. According to [Feng 81] the network has a demand assignment access mechanism. The switching elements are synchronized with a global clock running at 10 Mhz.

The Network Interface connects the MSE to the local VME bus. It contains receive buffers for data and control messages, DMA controllers and a hardware flowcontrol mechanism. The hardware flowcontrol mechanism and DMA controllers contribute to a high throughput: at least 20 Mbytes per second.
The local VME bus gives each node a flexible structure: nodes can be tuned to the requirements of the application. They can be extended with standard VME compatible products such as: I/O boards, memory modules, (dedicated) processors.

The communication protocol for Tumult-15 is implemented in software and executes on the Host. Fig. 3 gives the low level protocol of a send primitive (the receive primitive is analogue).

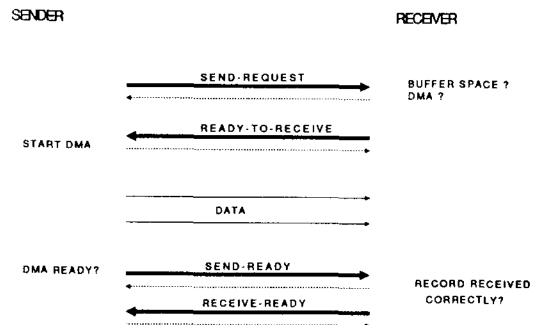

Fig. 3 Low level protocol for a send.

The sender and receiver communicate with each other via control messages, indicated as arrows in fig. 3. Each control messages generates a local interrupt at the host processor. To avoid overflow of the receiver control buffer every control message is acknowledged with an "ACK" control message (dotted arrow).

The sender transmits a "*send-request*" control message to the receiver where the needed resources (buffer space and a receiver DMA with data-fifo) are claimed. If buffer space is available and the DMA is ready the control message "*ready-to-receive*" is returned. Now the sender DMA is started and the actual transfer of the record from sender to receiver takes place. If the sender DMA is finished the sender sends the control-message "*send-ready*" to the receiver. The receiver checks if the record is received correctly, by inspecting the DMA controller and the receiver buffer. If everything is correct the control message "*receive-ready*" is returned to the sender. The claimed resources are released and the buffer administration is updated.

In the initial implementation, which has been written in Modular Pascal [Bron 84], every interrupt at the Host-

processor forced a process-switch to a "control-message-handler" process that interprets and handles the interrupt. It will be clear that this implementation leads to at least 10 interrupts (with 10 process-switches) per send/receive action. This introduces a relative long control overhead.

Several improvements are implemented since then:

1. The "ACK" control message is used to avoid overflow of the control buffer. The interrupts and process-switches for the "ACK" are eliminated by using an option of the network interface. In every interface there is a one place buffer ("ACK-register") used for the reception of "ACK" control messages. The protocol is adapted to prevent overwriting of this register: no control messages are sent, before the previous control message has been acknowledged with the reception of an "ACK" in the ACK-register. The status of the register is read by the Host, by busy waiting, so without any interrupt. With this change 4 of the 10 interrupts are avoided.

During communication three modules are called: InterProcessorCommunication (IPC), which may call the Kernel, which may call the Runtime package. The table below gives the average values of the the time spent in these modules.

a. Runtime package	30 %
b. Kernel	50 %
c. IPC	20 %

ad a. The runtime support package is a module providing some basic functions for Modular Pascal such as integer multiplication, addressing of records, etc.
ad b. The kernel takes care of process-switching, queuing, operations on semaphores, claiming and releasing resources, scheduling and address conversion.
ad c. The Inter Process Communication (IPC) includes primitives described in section 2.

2. The table above shows that the time involved in process-switching is a considerable portion of the overhead. To improve this two methods have been tried. First the computation involved in the "Control-message-handler" process was moved to the interrupt routine of the control interrupts. This lead to an important reduction of the control overhead (from 30 to 7 ms). However this also influenced the real-time behaviour of the Host, because interrupts have to be disabled during the execution of the protocol software in the interrupt routine. Therefore a new software technique was introduced called "software interrupts". For a more detailed presentation of this concept see [Luttmer 87]. In this technique the protocol routines can be executed in the software-interrupt handler, with the hardware interrupts enabled. This leads to a better real-time response as well as to a low control overhead.

3. In the original implementation there was no provision for recovering from lost control messages. (note: all words on the network are protected with parity bits, mutilated messages are removed from the network.). In the improved protocol an alternating bit protocol is used to recover from lost control messages.

All the just mentioned suggestions and improvements contribute to a protocol overhead reduction. Currently there is an implementation (written in Modula-2), on a Motorola 68000 (8 Mhz) nodeprocessor with an overhead of 4.2ms and 1.2 ms overhead on a Motorola 68020 (16 Mhz).

4. Implementation of the communication processor for Tumult-64.

Our primary goal is to improve the message throughput by applying dedicated hardware to support message passing. Besides that a number of other improvements are implemented in Tumult-64. The new communication network has a bi-directional ring topology mainly for improved reliability. However this topology also gives a performance improvement, because the network diameter is smaller. (N/2 instead of N-1 for a uni-directional network).

To improve the real-time behaviour extra hardware has been added. Each network interface is equipped with a local timer that keeps the global time. All timers are reset at startup and synchronously advanced by the global clock, so that every timer shows the same time. A second hardware feature is a mechanism to guarantee access to the network. Slots can be claimed (and released) for exclusive use during a real-time transaction.

In order to improve the flexibility of the system each node has a small dual-ported memory, in which each node has its own partition. This memory called "parameter memory" is used for control messages longer than 16 bits. The parameter memory can be used for future extensions of the communication protocol such as: (real-time) messages with priority, variable record length or extensions of control messages. In this section only the message throughput will be discussed in detail.

An important question is how should the Inter Process Communication (IPC) be partitioned between the Host and the message co-processor. As mentioned before the protocol overhead dominates the transfer overhead for small messages.
The send and receive functions are the most time critical primitives. (The connect, installMb and disconnect functions are only used during startup and termination of a communication (as in circuit switching). Once a connection between a link has been established, only send and receive primitives are used.) Therefore we decided to allocate the send and receive primitives on the communication co-processor.
In fig. 4 the architecture of a Tumult-64 node is given.

The Host processor is loaded with its part of the distributed operating system and executes its part of an application program. The network interface, the DMA controller, the communication controller and the shared memory together function as a single unit in assisting the host with the send and receive primitives. The host and communication co-processor interact and synchronize via the shared memory.
A functional similar structure can be found in [Ramachandran 87]. However these systems use a LAN as communication network and the communication processor can be seen as a front-end processor for the host.

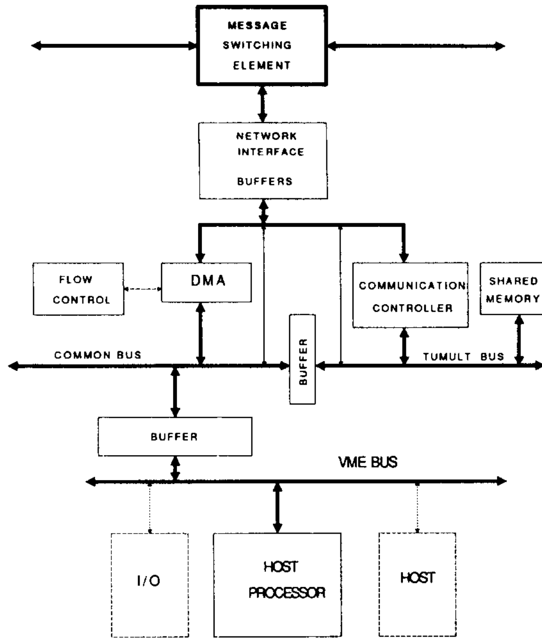

Fig.4 The architecture of a Tumult-64 node.

The control messages are handled by the communication controller. The controller handles the communication protocol: sending/receiving control messages, initializing and starting of DMA's, communication with the host etc. The shared memory serves as a cache for the communication controller. It contains as much information as needed to perform a send or receive operation without intervention of the host processor. The host will only be interrupted at the end of a send or receive action or in exceptional situations like transmission faults. The DMA unit performs the actual data transfer between the network and the host processor memory. The DMA units of sender and receiver are synchronized with a hardware flowcontrol mechanism [Jansen 87].
The communication protocol for a send operation is depicted in fig. 5.

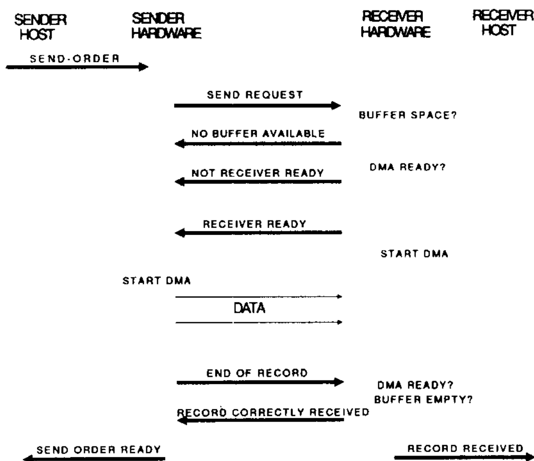

Fig. 5 The communication protocol.

Now we will describe the communication protocol. If the host performs a send action it will send a *Send-Order* message to the communication controller by placing the order in the shared memory. Thereafter the host must wait (busy-waiting or dispatching another job) for an answer (*Send-Order-Ready*) from the communication controller. The send-order contains information about base-address of the message, the link number etc. Upon completion of the send order the host is informed by a change in the interface status (in case of busy waiting) or by an interrupt from the communication controller.

When a communication controller receives a Send-Order a *Send-Request* (SR) will be sent to the communication controller of the receiver node. The Send-Request contains a source-node number, and the linknumber. Upon reception of this control word the controller first checks for buffer space. If no bufferspace is available the request is queued in a buffer queue and an acknowledge called *No-Buffer-Available* (NBA-ACK) is returned. If there is buffer space available, the buffer is claimed and a check for availability of the DMA controller is made. If the DMA is available, it is initiated with the length and the base-address of the buffer and a *Receiver-Ready* (RR) is returned. If the DMA controller is not available the request is queued in the RDMA queue and a *NOT-Receiver-Ready* (NOTRR) is returned. The RDMA queue resides in the shared memory of the communication processor, which handles all queue actions. (The communication controller of the sender is not allowed to send new SR's until it has received a NBA-ACK, RR, or NOTRR to prevent control buffer overflow.)

When the sender controller receives a RR it places the RR in a SDMA queue. The SDMA queue contains processes that have claimed the receiver DMA and are waiting for the sender DMA. If the sender DMA becomes available the first request in the SDMA queue will initiate the DMA and the data is transferred. At the end of the data transport an *End-Of-Record* (EOR) will be sent. Upon receiving the EOR the receiver controller will check both the receiver data buffer and the DMA counter. If the data buffer is empty and the DMA controller is counted out a *Record-Correctly-Received* (RCR) is returned. Otherwise a *Record-Not-Correctly-Received* (RNCR) asking for a retransmission of the record.

5. Conclusion

Our experiments with Tumult-15 showed that the communication overhead of multi-processor systems, especially for small messages is considerable in general. These results have been confirmed by recent papers of other researchers. The overhead is mainly due to protocol overhead and not data transfer overhead. To decrease the protocol overhead a message passing co-processor is proposed for Tumult-64. The co-processor handles the simple send/receive protocols. We expect an communication overhead in the order of 200us, which is a considerable improvement compared to 4.2 ms for the current software implementation.

In Tumult-64 also attention is paid to reliability (bi-directional network), real-time behaviour and flexibility. For real-time applications special hardware has been added to allow for guaranteed access time to the network.
Tumult-15, a predecessor of Tumult-64, is operational as a

high performance system for recognition of handwritten characters (Dutch PTT). Currently a prototype for Tumult-64 is under construction.

Acknowledgements

We gratefully acknowledge the staff members of the University of Twente, The Dr. Neher Laboratory and Oce Nederland R&D as well as the numerous students without whose support the implementation of Tumult would never have been possible.

References

[Bron 82] : Bron C.W.:"Modular Pascal Definition", Department of Computer Science, Twente University of Technology, The Netherlands, int. rep. nr. INF-82-10,1982.

[Bron 84] : Bron C.W., Dijkstra E.J.: "On the use of exception handling in Modular Pascal", Department of Computer Science, State University Groningen, The Netherlands, Sept. 1984.

[Cheriton 85] : Cheriton D.R., W. Zwaenepoel: "Distributed Process Groups in the V kernel", ACM trans. on Comp. Syst., pp. 77-107, May 1985.

[Dijkstra 68] : Dijkstra E.W.: "Co-operating Sequential Processes", Technical Report EWD-123, Technological University Eindhoven, The Netherlands (1965); reprinted in: Genuys F.: " Programming Languages", (Academic Press, London, 1968), 43-112.

[Feng 84] : Feng T., Wu C.: " Tutorial: Interconnection networks for parallel and distributed processing", IEEE Computer Society Press, 1984.

[Hockney 85] : Hockney R.W.: "MIMD computing in the USA - 1984", Parallel Computing 2, pp. 119-136, 1985.

[Jansen 80] : Jansen P.G., Kessels J.L.W.: "The DIMOND: A component for modular construction of switching networks", IEEE Transaction on Computers, vol. C-29, no. 10, pg 884-889, Oct. 1980.

[Jansen 87] : Jansen P.G., Smit G.J.M.: "TUMULT, a multi-processor: its architecture and communication", Twente University of Technology, Department of Computer Science, int. rep. nr. INF-87-3, 1987.

[Jansen 88] : Jansen P.G., Smit G.J.M.: " TUMULT-64: a real-time multi-processor system", to be published.

[Kessels 81] : Kessels J.L.W.: "The SOMA: A programming construct for Distributed Processing", IEEE Trans. on Softw. Eng. no. 5, Sept. 1981.

[Kruatrachue 88] : Kruatrachue B., Lewis T.: "Grain size determination for Parallel Processing", IEEE Software, pg. 23-32, Jan. 1988.

[Langen 87] : Langen P. v., Sijbers A.: "The distributed file system in Tumult", Conference Computing Science in the Netherlands, Amsterdam, 1987.

[Luttmer 87] : Luttmer M.L.M., Jansen P.G.: "The Tumult kernel", Twente University of Technology, Department of Computer Science, int. rep. nr. INF-87-15, 1987.

[Scott 87] : Scott M.L., Cox A.L.: "An Empirical Study of Message-Passing Overhead", 7th Conference on Distributed Computing Systems, Berlin, pg 536-543, september 1987.

[Ramachandran 87] : Ramachandran U., Solomon M., Vernon M.: "Hardware support for interprocess communication", 1987 Int. Symposium on Computer Architecture, pg 178-188, May 1987.

[Tanenbaum 85] : Tanenbaum A.S., Renesse R. v.: "Distributed Operating Systems", Computing Surveys, Vol. 17, no. 4, Dec. 1985.

[VME 85] : VME-bus specification manual, Revision C, Feb. 1985.

[Wirth 85] : Wirth N.,: "Programming in Modula-2", Third corrected edition, Springer Verlag, 1985.