North-Holland
Microprocessing and Microprogramming 26 (1989/90) 305–317

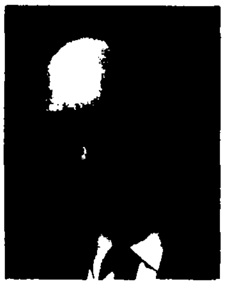
An Experimental Multiprocessor System for Distributed Parallel Computations

L. De Maeyer, A. Di Nicola, R. Maetche, C. von der Malsburg and L. Wiskott
Max-Planck-Institute for Biophysical Chemistry, D-3400 Göttingen, F.R. Germany

The availability of low-cost microprocessor chips with efficient instruction sets for specific numerical tasks (signal processors) has been exploited for building a versatile multiprocessor system, consisting of a host minicomputer augmented by a number of joint processors. The host provides a multi-user-multitasking environment and manages system resources and task scheduling. User applications can call upon one or more joint processors for parallel execution of adequately partitioned, computationally intensive numeric operations. Each joint processor has sufficient local memory for storing procedures and data and has access to regions in host memory for shared data. Kernel processes in the host and in the joint processors provide the necessary mechanism for initialization and synchronization of the distributed parallel execution of procedures.

Keywords: Multiprocessor, Parallel programming, Shared memory, Signal processor, Remote procedure call.

Submitted: October 1988
Revised: 13 March 1989
Accepted: 16 October 1989



Leo De Maeyer (1927) obtained the Dr. Sc. degree in physical chemistry in 1954 at the University of Leuven, Belgium. Since 1965 he is scientific member of the Max-Planck-Gesellschaft and director of the Department of Experimental Methods at the Max-Planck-Institute of Biophysical Chemistry in Goettingen. He is also teaching at the University of Leuven where he founded the Laboratory of Chemical and Biological Dynamics in 1970. From 1978 to 1981 he was head of the EMBL Instrumentation Division in Heidelberg.
One of his main interests is the development and introduction of tools and techniques in the acquisition and evaluation of data in experimental natural sciences.

Christoph von der Malsburg (1942) started out as a physicist, studying at the Universities of Goettingen, Muenchen and Heidelberg. He received his Diploma and Ph.D. from the University of Heidelberg, based on experimental work in nuclear physics and elementary particle physics, respectively. As part of his work he spent 3 years in CERN. He then converted to neurobiology, which he pursued in the Department of Neurobiology at the Max-Planck-Institute for Biophysical Chemistry in Goettingen.
His interest is focussed on processes of organization in the nervous system, in both the developing and in the working brain. He is now in the University of Southern California, where he has a double appointment in the Department for Computerscience and in the Section for Neurobiology.

Roland Maetche (1958) is an electronics technician in the Department of Experimental Methods of the Max-Planck-Institute for Biophysical Chemistry in Goettingen.
In the present study he developed and tested the operating system components and did the system integration, including the FORTRAN interface and a library of graphics routines for the joint processors.

Angelo Di Nicola (1948) received the Diploma in electrical engineering at the Technisch Instituut H. Hart in Hasselt, Belgium. He has been with the Max-Planck-Institute for Biophysical Chemistry since 1972. As head of a development team he is involved with hardware for instruments, data acquisition systems and computers in the Department of Experimental Methods. He designed and tested the joint-processor hardware used in the present project.

Laurenz Wiskott (1964) collaborated in this project as an undergraduate student in physics at the University of Goettingen. He programmed the neural network application and developed the automatic segmentation routines for distributing parallel processes over joint processors from a FORTRAN environment. He is presently a graduate student in theoretical physics at the University of Osnabrueck.

Modern VLSI technology is leading to a rapid evolution of multiprocessor architectures. The classical organization of a computer, based on a single CPU with a homogeneous instruction set for all operations, is changing to a cooperative organization with specialized functional resources. By increasing the number of replicated processors a very large amount of processing power can be associated with a single application at very low cost. All this requires a corresponding evolution in algorithms, programming languages and tools. The programmer must be able to recognize the opportunities for a particular matching between the requirements of his application and the available functionalities of the processing elements. The software instruments for incorporating the available resources in his application must be flexible to optimize the matching.

The present report describes a small parallel processor system designed for laboratory applications in which a ten- to hundredfold faster execution of particular subtasks can be achieved as compared to conventional implementations on the host minicomputer. Speedups of this magnitude are due only partly to parallelization. A major part is due to the execution speed of modern VLSI signal processors. Typical applications for which this development is intended include recognition and discrimination in pattern analysis, data compression and feature extraction, solution of inverse problems by direct transform methods or by iterative optimization, dynamic systems simulation etc. Applications of this kind are often found in experimental situations requiring a close coupling between data acquisition and data analysis.

1. Hardware Organization and Software Environment

It was our main purpose to provide an experimental development system that allows the evaluation of partitioning strategies for distributing and synchronizing concurrent computations without requiring many changes or additions to an available operating system or to the programming languages supported by its compilers. A prototype was implemented using a DEC PDP11/73 with RSX11M PLUS as the supporting host and eight TI TMS 32020 signal processors [1] as joint processors. Each joint processor is mounted on an individual circuit board together with 256 kb of local memory. The board contains a small set of status, control and address registers mapped to a Q-Bus I/O-space. They are used to exchange command and interrupt codes between host and joint processor and to enable memory access from the host processor to joint processor local memory locations and vice versa.

The choice of 16-bit host and joint processor architectures is not a serious limitation for an experimental system. The TMS 32020 was one of the most advanced signal processors available when this project was started. Its architecture is well adapted for numeric computations and also for processing data streams from sensors in real-time applications. The PDP11/73 as a host was compatible with an existing network of laboratory computers. It offers a well designed operating system with support for many development services but still providing simplicity for its adaptation. The limitation of host programs to an instruction virtual address space of 64 kb is less convenient. A considerable amount of this space is taken up for addressing the runtime routine library of the programming language. In DEC FORTRAN 77 under RSX11MPLUS the address limitation is somewhat less restricted as far as data space is concerned. This FORTRAN implementation [2] allows for the declaration of one or more large-sized VIRTUAL ARRAY's, accessed via the address remapping mechanisms of the operating system in a manner which is transparent to the programmer. Almost the whole available physical address space can thereby be made accessible to the host program for data storage shared with joint processors. We have therefore used FORTRAN as the main language for the host processor in our applications. FORTRAN remains to be a language of choice for scientific computations.

A more serious limitation of our experimental system is the use of the LSI 11 Q-Bus [3] as a single common bus for access from all processors, including the host, to shared memory. The host processor even must use this bus for fetching instructions and for communicating with disk, terminals and network devices. As a result, the Q-Bus becomes a bottleneck during exchange of large amounts of data between shared and local memory. We can ex-

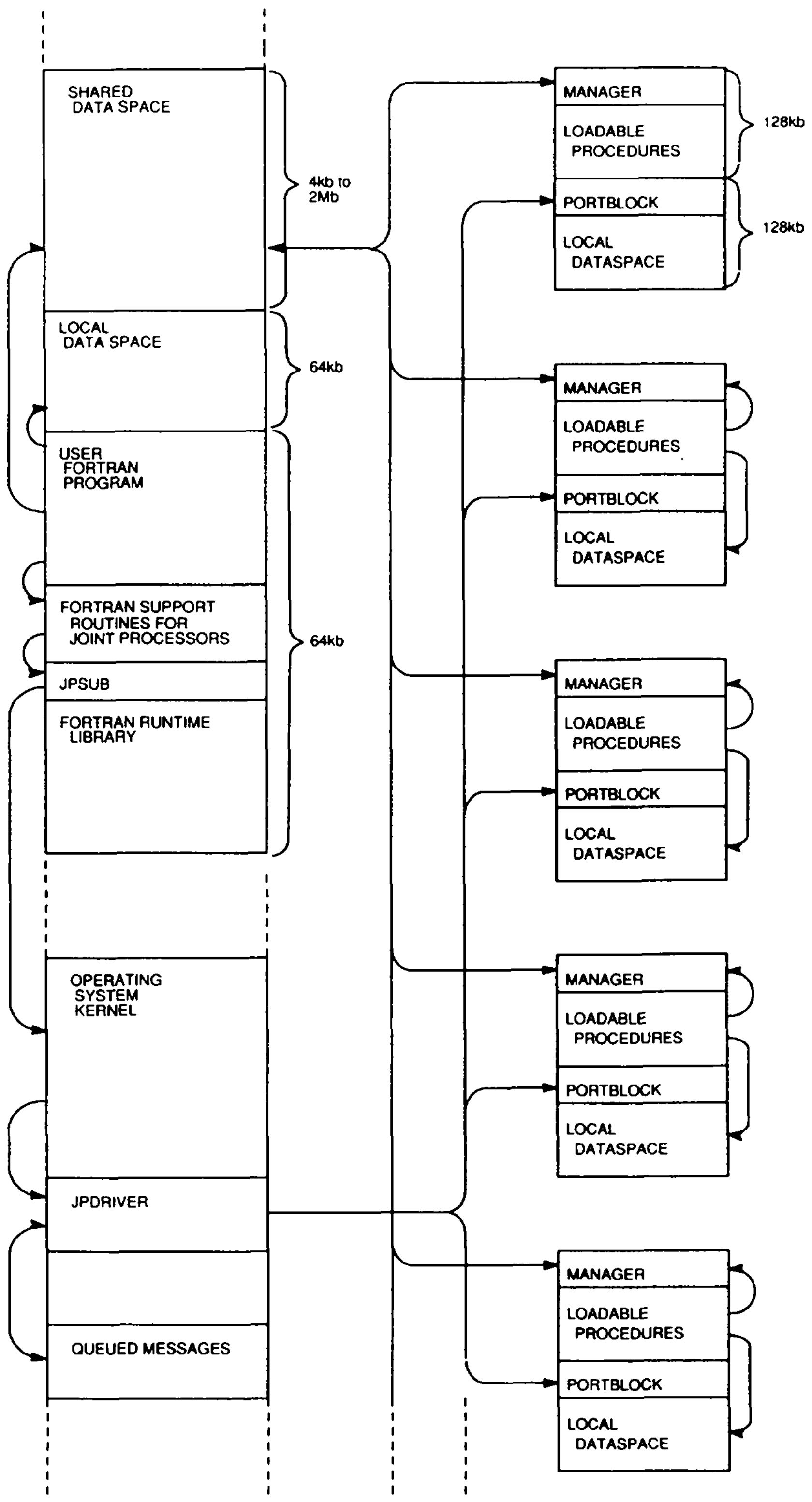


Fig. 1. Communication between host and joint processors takes place via JPDRIVER, which can access all joint processor memory locations via hardware registers. Procedures in joint processors can start DMA transfers between shared and local memory. The PORTBLOCK is a reserved communication region.

pect real speed improvements from our system only for applications that involve numerically intensive transformations or numerous iterations on the local data sets once they have been loaded from shared memory. The system is less suited for applications requiring fine-grained data exchange between several processors. An additional communication channel is provided by the serial link implemented in the TMS 32020 architecture. This feature has not been used in present applications.

2. Host-joint Processor Communication

In our implementation joint processors are resources managed by the host operating system in more or less the same way as I/O devices or communication lines. User programs must explicitly request the allocation of the required number of coprocessors and assign logical numbers for their identification. All communication between host programs and their allocated joint processors are handled by a common device driver, incorporated in the operating system kernel. The standard set of operating system calls is used for communication with the driver. The necessary calls are encapsulated in a set of FORTRAN callable subroutines. The driver process loads a small supervisor kernel (manager) in each joint processor. This kernel provides facilities for communication with the driver and maintains a list of numerical and non-numerical procedures which a joint processor can execute as a service for host programs. Among these procedures are basic ones, e.g. for downloading additional procedures and entering them in the list, for deleting procedures that may be overwritten, or for message or data transfer etc.

A joint processor can be regarded as a server for more or less complex procedures upon sets of data. Procedure names and parameters identifying the data set on which they must execute are passed in messages for each joint processor. These messages have a short standard format, containing essentially the parameters transmitted by the FORTRAN host program via a standardized reprocedure call CALL JPSUB (processor-id, procedure-name, shared-workspace, workspace-size, parameter-buffer, time-out, sync-flag) in which the last argument is a logical variable. If this variable is TRUE, further messages to the joint processor are rejected by the driver until the manager acknowledges completion of the procedure. A host program using this call for executing a named procedure in a joint processor does not need to wait for completion of the procedure or even the transmission of the message. As soon as the driver has received the message, control returns to the host program, which can continue with other operations or send another message to the same or to different joint processors. Program and data storage areas in host and joint processor memories involved in communication between a host user task and the procedures that it invokes in joint processors are indicated in *Fig. 1.*
The driver maintains a queue of messages for each joint processor. Upon termination of a procedure in a joint processor an interrupt to the host from the processor's manager signals the driver that the processor is ready to accept the next message. The driver then copies a message from the queue into a designated area (portblock) of the processor's data memory and interrupts the manager, indicating that a new procedure is to be started. End status information about the terminated process is available in the processor status register. Before copying a new message into the processor's portblock, the driver makes this information available to the intermediate routine JPSUB. This routine maintains a count of messages and provides an entry that returns the number of the last terminated procedure and its termination status. This information is useful for testing and debugging new procedures during program development.

On top of this communication mechanism a number of other auxiliary host routines is available for supporting segmentation of data structures, distributed procedure execution, requesting processor status information or waiting for completion of critical procedures.

Procedures running in joint processors must be programmed and compiled separately from the main program running in the host. Different programming languages most appropriate for the joint processor type may be used for this purpose. These programs normally operate only on data in local joint processor memory. The joint processor kernel provides basic procedures for transferring data between local and host memory. User application programs in host and joint processors therefore do not need to have compile-time information on where the data structures operated upon will reside in each others memory. Parameters are mostly passed by value. Offset pointers in data structures copied from shared to local memory can be passed in this manner. Some parameters (e.g. routine identifications) are passed by name. The joint processor kernel then retrieves the routine address from its procedure namelist.

Two classes of joint processor routines may be distinguished: (i) MANAGER-procedures, which are part of the manager kernel, (ii) LOADABLE-procedures, supplying a repertoire of useful numerical or other operations. Both classes of procedures are called by the same communication protocol (JPSUB and JPDRIVER). Manager procedures cannot be deleted from the name list; they provide basic mechanisms for loading and unloading data or programs and for iterating the execution of sequences of procedures. Loadable procedures are arbitrary procedures needed for a particular application. The host program loads these procedures according to its needs. The joint processor manager kernel maintains the list of names and entry addresses of loaded procedures.

A user task can allocate joint processors for its exclusive use or for shared use with other user tasks. In the latter case successive remote procedure calls to a processor are not guaranteed to come from the same user. User tasks should therefore not rely on data left in the processor's memory after termination of a procedure. When used on a shared basis each individual procedure must copy the result of its operation back to the allocated workspace in host memory before terminating. When a processor is allocated to a user task on an exlusive basis, the sequence of messages received by the processor is that of their issuing by the user task. In this case it is possible to use a chaining procedure that will accept a sequence of portblock messages that are entered in a list. Execution of the procedures named in these messages is delayed until an execution command message is received. The chained sequence may be repeated a specified number of times or until a condition is true before termination is announced by an interrupt to the host. Chaining is an efficient mechanism for iterations. It enables to formulate a complex operation in terms of simpler procedures and to execute these in sequence without overhead of interrupts and repeated message passing.

In the family of TMS 320 signal processors data addresses and program addresses form two separate sets. Paged and indexed data addressing is well supported by the TMS 320 architecture. It is not difficult to write all loadable procedures in such a way that they operate on run-time-relocatable data.

3. Data Segmentation Descriptors

When a number of joint processors is used to operate in an SIMD-similar fashion, a segmentation of data structures is required. It is a simple matter to have every joint processor execute an identical sequence of identical procedures. The data segments, however, assigned to each processor must reside in different, identifiable workspaces in host memory. From there the data are copied to identical regions in each joint processor data memory, so that identical programs accessing a given storage location in different local processor memories will operate on different segments of the data structure. Data segmentation by automatic vectorizing compilers is usually based on the index range of array variables explicitly named in parallelisable DO loops [4–6]. Our approach is not based on an extended interpretation of DO loops or similar language constructs, although this could be implemented in the form of a precompiler. Instead, we have designed a procedural segmentation scheme and a naming convention for data segments residing or created in different processors.

A data structure which is to be distributed over several processors will, in general, have an internal organization that cannot be arbitrarily segmented. In the assignment of a distributed operation on a two-dimensional array, for example, it is often required to allocate each individual processor to a complete subset of rows (or columns). It will not always be possible to divide the total number of rows (or columns) into subsets of equal size. This problem has been solved by maintaining a set of segmentation descriptors for each distributed variable. They contain information on the distribution of subdomains of a data structure in shared and local memories and on the allocation of processors.

A data structure that is to be subdivided and distributed over several joint processors (or that is to be composed from results obtained from several processors) will be associated temporarily with more than one name. One is the genuine FORTRAN variable name, under which the structure is identified for its use in normal FORTRAN assignment or I/O list statements. This name refers to the complete, unsegmented structure, defined by the usual FORTRAN type and dimension declarations. The second name refers to data segments operated upon by a joint processor. The regions that these segments occupy in local processor memory are identified by an operand name. Together with the logical unit number identifying a joint processor, the operand name forms a pointer tuple used to access three segment descriptors, associating the operand with the individual segments of a data structure:

SIZ	(proc-id, op-name) contains the length of the segment,
ADR	(proc-id, op-name) contains the virtual address of an operand region for the segment in the local data memory,
OFS	(proc-id, op-name) contains the offset (index) of the first element of the segment from the origin of an associated data structure in host memory.

The association between an operand name and a FORTRAN variable is not necessarily unique or permanent: the operand descriptors do not contain references to the absolute location of a FORTRAN variable in host memory. At different times the same FORTRAN variable may be associated with different segment descriptors, corresponding to a different operand name. The same operand name and segment descriptors can also be used with different FORTRAN variables if they have the same data structure. The actual association between a FORTRAN variable and a set of segment descriptors is established when the FORTRAN name of the variable is used for the argument "shared-workspace" in the subroutine JPSUB, and the segment descriptors SIZ, ADR and OFS are used as argument resp. parameters when JPSUB refers to manager or loadable routines that access host memory.

Segment descriptors are generated by a FORTRAN subroutine CRESEG (proc-range, op-name, elem-len, el-num, offset). The argument proc-range specifies a number of joint processors to which operand segments will be assigned. op-name designates a variable of type integer whose name the programmer will use as a generic name for the operands defined by the segment descriptor. CRESEG supplies an integer value to this name, later used to access the created descriptor set. elem-len is the length (in word units) of an element of this operand. This must be specified by the calling program. It may represent the array length of a matrix column, a fixed record length, the length of an n-tuple or other smallest element of the data structure that should not be subdivided by segmentation. el-num is the maximum number of elements into which a structure can be divided. It will usually indicate the number of rows in a matrix, the number of records or tuples, etc., to be distributed for a parallel operation. The last argument enables the programmer to specify the segmentation of a subdomain of a data structure, starting at an offset from its origin. If el-num cannot be distributed evenly over the given range of processes, CRESEG will put a larger number of elements in the segments assigned to the processors with lower unit numbers in the range. CRESEG initializes the SIZ, ADR and OFS descriptors. SIZ is calculated from elem-len and the number of elements in the segment assigned to a processor. If k is the number of processors in the specified range, the segment size assigned to processor $i (i = 1 \text{ to } k)$ is

$$\begin{aligned}\text{SIZ}(i, \text{op-name}) &= \text{elem-len}\,(\text{el-num}/k + 1) \text{ for } i \leqslant \text{el-num}\,(\text{mod } k)\\ &= \text{elem-len}\,(\text{el-num}/k) \text{ for } i > \text{el-num}\,(\text{mod } k).\end{aligned}$$

The virtual address in local data memory of processor i for the operand region designated by op-name is

$$\text{ADR}(i,\text{op-name}) = \sum_{j=1}^{j=\text{op-name}-1} \text{SIZ}(i,j).$$

ADR is calculated incrementally, depending on the number of different operand names and their segment sizes already defined and assigned to a processor. The operand names are essentially needed to create and designate separate regions in local data memory of a processor. OFS is also calculated incrementally, starting with the offset argument specified in the subroutine call, and augmented according to the size of the segment of elements on which consecutive processors in the range will operate.

$$\mathrm{OFS}(i,\text{op-name}) = \text{offset} + \sum_{j=0}^{j=i-1} \mathrm{SIZ}(j,\text{op-name}).$$

The argument el-num has a special meaning when it is given the value zero or one. el-num = 0 signifies that no segmentation takes place but a region of SIZ = elem-len is reserved in the local data space of each processor. All processors have the same OFS value for the associated operand name. This enables one to load an identical data segment into all processors. When el-num = 1 the operand region is reserved only in the lowest processor in the range.

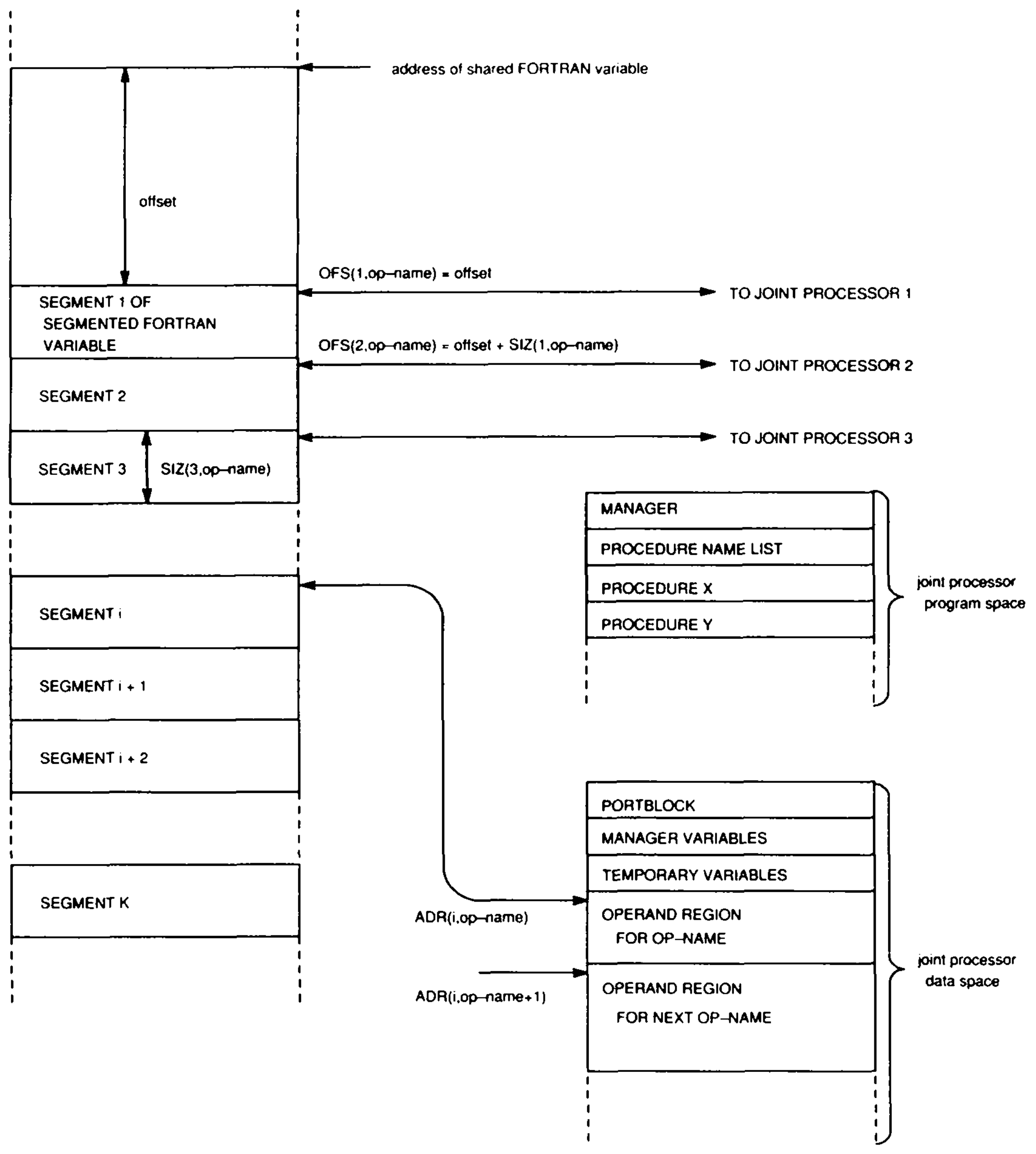


Fig. 2. Mapping segments of a data structure in shared memory to operand regions in joint processor local memories.

Although this allocation scheme for segmented operands in local processor memories may seem rather complicated, its implementation is efficient and its use is rather simple. The five arguments to CRESEG is all the programmer has to supply. An operand name for data segments residing in local processor memory must be supplied at compile-time and initialized to zero. The values for all other arguments can be defined at run-time, allowing a program to operate with array structures whose dimensions are unknown at compile-time. The mapping of shared memory segments to operand regions in joint processors is shown in *Fig. 2*.

The process of subdivision and the use of the segmentation descriptors for managing areas in joint processor memories in which segments of host variables can reside are made as much as possible transparent to the user. Specialized languages [7,8] or specialized compilers for existing languages [9,10] may be designed that hide these aspects of parallelization from the application programmer. We have not followed this approach because of its inflexibility, requiring very rigid specifications of the new language constructs. At the present stage we prefer to provide a library of auxiliary routines for the supported languages and compilers. Segmentation descriptors created by procedures called at run-time may be changed dynamically whereas compile-time segmentation is static.

The programmer can avoid explicitly referencing the segment descriptors by making use of already available library subroutines. *Fig. 3* shows a FORTRAN program for segmented vector × matrix multiplication.

The value 1017 for RANGE indicates that operand spaces for MATSEG, VECIN, VECOUT have to be reserved in processors 10 to 17. The four last subroutine calls invoke library routines that make use of JPSUB and segment descriptors for starting execution of processor routines with corresponding names. LOADD and STORED are managing routines that transfer data between host and local memory and thereby associate the segmented operand with a host FORTRAN variable. The FORTRAN subroutine VMULM starts the distributed execution of a loadable matrix multiplication procedure. It was loaded by calling the library routine INIPRG with arguments indicating a filename VMULM.MPO, containing the program image in binary format, and a name 'VMULM' under which the loaded routine is entered in the namelist. The local operand names MATSEG, VECIN, VECOUT enable the host-library subroutine VMULM to access the segmentation descriptors and to communicate to the local routine VMULM in each joint processor the location and size of its local operands. All but the last four program lines are initializations to be executed only once in a larger program.

In this example the number of elements is not an integer multiple of the number of processors. Processors 10 to 13 will generate 13 elements of the result vector while processors 14 to 17 will generate 12. The descriptors created by CRESEG are not visible for the programmer. They are passed between CRESEG and other library routines (e.g. VMULM) in a COMMON array.

For MIMD applications the segmentation procedure is not always required, but even in this case the possibility to control the allocation of local data memory for host-identifiable operands in a rather simple manner is very useful.

4. A Neural Network Application

The system has been used in several applications. One is the simulation of a particular neural network model, which is capable of translation-invariant pattern recognition. This application can be formalized in an algorithm in which matrix and vector operations are dominant. The joint processors are mainly used for the iterated updating of a vector representing the activities of a large set of neurons. In the model the connections between neurons form a sparse matrix of zeroes and ones. In the implemented efficient updating algorithm the coefficients of this $m \times m$ matrix are numbered from 1 to m^2, and the ordinal number of the coefficients which are different from zero are listed in a linear array. The signal activity emitted by a neuron along a connection is proportional to its own activity. A linear function of the total (summed) signal strength arriving at a neuron from all its connections, modified by a (negative) normalizing term that depends on the activity of all neurons in the network, is added to the actual activity of the receiving neuron for deter-

```
VIRTUAL HOSTVE(150), HOSTMA(100,150)
INTEGER HOSTVE, HOSTMA
INTEGER MATSEG, VECIN, VECOUT        ! define operand names
INTEGER RANGE, ELLEN, ELNUM, OFFSET
DATA MATSEG, VECIN, VECOUT/0,0,0     ! initialize operand name value
 .
 .
 .
RANGE = 1017                         ! use processors 10 to 17
OFFSET = 0                           ! no offset in vector or matrix data
                                     ! structure
ELLEN = 100                          ! input vector is long data structure
ELNUM = 0                            ! all processors must receive full
                                     ! vector
CALL CRESEG (RANGE, VECIN, ELLEN, ELNUM, OFFSET) ! create segment
                                     ! descriptors for full vector in each
                                     ! processor
ELLEN = 1                            ! elements of output vector are
ELNUM = 100                          ! divided over processors
CALL CRESEG (RANGE, VECOUT, ELLEN, ELNUM, OFFSET) ! create segment
                                     ! descriptors for dividing vector over
                                     ! processors
ELLEN = 100                          ! matrix contains columns of 100
                                     ! integers
ELNUM = 100                          ! 100 columns can be divided over
                                     ! coprocessors
CALL CRESEG (RANGE, MATSEG, ELLEN, ELNUM, OFFSET) ! create segment
                                     ! descriptors for dividing sets of
                                     ! columns over coprocessors
CALL INIPRG (RANGE, 'VMULM.MPO', 'VMULM', ERR) ! load all processors with
                                     ! loadable procedure from disk file
                                     ! and enter procedure name VMULM in
                                     ! namelist
 .
 .
 .
CALL LOADD (RANGE, HOSTVE, VECIN, ERR)  ! load HOSTVE in operand space VECIN
CALL LOADD (RANGE, HOSTMA, MATSEG, ERR) ! load data segments from HOSTMA in
                                     ! operand space for MATSEG (sets of
                                     ! matrix columns)
CALL VMULM (RANGE, MATSEG, VECIN, VECOUT, FIXPNT, ERR) ! Execute matrix
                                     ! multiplication procedure in
                                     ! coprocessors
CALL STORED (RANGE, VECOUT, HOSTVE, ERR) ! replace original HOSTVE data by
                                     ! combined segments from operand space
                                     ! VECOUT
```

Fig. 3. FORTRAN program for segmented vector × matrix multiplication.

mining its activity in the next iteration. The determination involves an amplifying, piecewise linear, updating function setting the next activity equal to a maximum value when the argument exceeds an upper treshold, or equal to zero when it drops below a lower treshold.

The neural system simulated here is based on the representation of images and objects by labeled graphs. Recognition is done by graph matching [11]. The system consists of subnetworks which form an image domain (affected by image data) and an object domain (containing stored objects). The image domain is formed by a two-dimensional sheet of neurons, a rectangular lattice of points. Each point is occupied by a complement of feature specific cells (50 feature types in our simulations). Neurons in neighbouring points of the lattice are connected to each other (there being 8 neighbours to a point), irrespective of feature type. A concrete image is formed by the selection of one neuron (feature) for each lattice point in the image plane.

The object domain is also formed by two-dimensional lattice sheets of neurons, one sheet per object. Each object network can best be viewed as a copy of a concrete image seen earlier (the storage process is not modeled here). Thus, each point in the lattice of an object network is occupied by a neuron with a particular feature type. The distribution of features over the lattice represents the structure of the object. Neurons in neighbouring lattice points are connected with each other. In our simulation, the objects stored in the object domain (and later presented in the image) were random distributions of feature types. Neural connections between the image domain and the object domain are feature preserving. Thus, two cells in the two substructures are connected precisely when they are of the same feature type, irrespective of position.

The process of recognition makes use of the fact that a familiar (i.e. stored) object appearing in the image is isomorphic to one of the stored objects, in the sense that it contains the same feature types in the same topological arrangement. Basic to the process of recognition is the tendency of the networks in the image and in the object domain to activate local clusters of neurons. In our simulations, dynamic parameters were set such as to favour clusters of 3 × 3 active neurons. The image domain

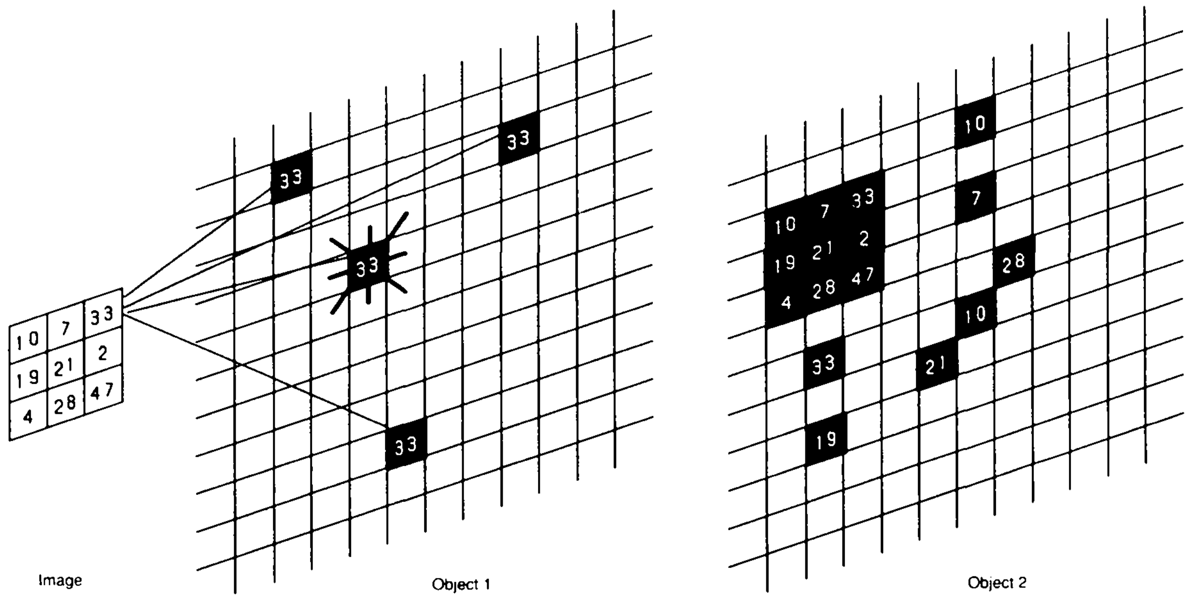


Fig. 4. Appearance of a feature type (e.g. feature 33) in the image excites all neurons of the same feature type in the object planes. An excited neuron in the object plane is connected to (excites) all its 8 neighbours. There is a common suppression proportional to the activity in the objects. Only clusters of neurons with feature types matching those of the image will survive suppression.

spontaneously forms a sequence of such clusters, which play the role of shifting focal attention. When a particular focus is active, the connections from the image to the object domain activate in the latter all those neurons which happen to have feature types which are contained in the focus. This is indicated schematically in *Fig. 4*. Dynamics in the object domain then takes over and selects a subset of those neurons receiving afferent excitation. If a familiar object is presented, there are two kinds of excited neurons: those which form a local focus in the correct object in the correct neighbourhood locations, and neurons scattered randomly over all objects and locations. Dynamical exchange of excitation between neighbouring neurons, and an exchange of global inhibition between all neurons in the object domain, stabilize the focus and switch off the scattered neurons. In this way, the correct position in the correct object can be identified with fair confidence (if the object presented is familiar, to be sure). This identification is completely independent of the position of the object in image space. After a fairly short sequence of such focal identifications the identity of the correct object can be reliably determined, e.g. by integrating the total activity attracted into each object network.

In our implementation of the system described the image plane was handled in the host machine and in each of the joint processors one object was stored. Our program can handle up to 32,768 neurons in the object domain. This is not an absolute limit, but a practical value beyond which not much new is learned about the behaviour of the model in function of its internal parameters. Each of 8 joint processors handles a subnet of max. 4k neurons with 8 neighbourhood connections per neuron. The connection topology is determined by a subroutine in the host program. The connection topology is the same for all subnets and there are no connections between neurons of different subnets in the present version of the algorithm. The latter property makes the configuration of parallel joint processor extremely suitable for this application, which is designed to identify the stored object, if any, in which the combination of features activating a 3×3 square in the image net is represented.

Developing this application required the programming of a number of matrix and vector procedures running in the TMS 32030 joint processors. This has led to a library of assembler-written TMS 320 routines, adapted to the manager kernel and the calling and segmentation conventions indicated above. The procedures provided by this library are application-independent, they can be used for many purposes as a standard routine library for a number of mathematical operations. The C-compiler [12] for the TMS 320C25 was not yet available when our development started. Using C will make it easier to extend this library, perhaps with a slight loss in performance due to the less optimized code generated by the compiler.

5. Shared Joint Processor Applications

Joint processors may relieve the host processor by simultaneously supporting several user tasks. An application of this kind was also developed. Here a single joint TMS 32020 processor is used as a graphics processor creating bit-maps in host memory. Bit-map data are generated in distinct workspaces according to vector lists or image data supplied by the calling programs. These calls can come from different host tasks, each identifying its allocated workspaces for bit-maps and vector lists in the call. The joint processor is shared by the different programs since there remain no relevant data in local memory after processing a list. Once finished, the bit-maps are sent to different printers and plotters by the host tasks. The whole system is used as a print- and plotserver receiving print- and plot-files from an ethernet LAN. Each device is under control of a separate task, calling upon the joint processor for graphics support if needed. Printers and plotters are relatively slow devices. Although the TMS 32020 is not conceived primarily for graphics operations and lacks the usual primitive instructions found in graphic coprocessors, it is fast enough to keep in step with the mechanical devices. Dithering algorithms for rendering monochrome and color images are easily supported by this processor. Most large size ink-jet or thermotransfer color plotters require separate bitmapped data for different colors. The corresponding vector translation and color-map dithering can be distributed over several joint processors. The communication mechanisms for joint processors described above can be applied to special graphics or image processors as well as to signal processors.

6. Performance Gain

Performance of a multiprocessor system is measured in terms of speedup ratio, referring to the execution time of a program as a function of the number of processors assigned to it. In our system data must be copied from host memory to local memory before local program execution. Results must be copied back to host memory. This is not essentially different from cached operation, where the shared memory accesses occur more random, however. Also, the host driver program must access the processor-associated registers and load the portblock. An additional overhead is caused by calling the intermediate routines and the driver via the operating system. It is obvious that a considerable excess data traffic is involved over the classical single-processor execution of a program. As long as the execution part of locally executed procedures is long compared to the time needed to load and unload the data, however, a respectable performance gain is nevertheless achievable. It is advantageous to use the chaining facility for different consecutive operations and iterations. The total execution time of a program can be estimated by the following formula:

$$t = \alpha + \beta n + \gamma s + \delta(s/n).$$

Here $n \geqslant 1$ is the number of joint processors over which a program is distributed, s represents a measure of the size of the problem, e.g. the complexity of the underlying algorithm. α represents parts of the FORTRAN host program that are independent of n and s, e.g. initializations, interactive or file I/O etc. βn denotes the time needed for host processor communication, loading programs, starting procedures, initialization of segment descriptors and transfer of parameter data that are independent of the problem size. The term γs includes parts of the host program related to problem size, e.g. initialization of variables for the joint processors and transfers of data segments to and from a joint processor. This transfer overhead must be taken into account for only one joint processor in favorable cases since for the other processors these transfers are taking place during local program execution of the processor that first finished loading its data segments. The last term $\delta(s/n)$ represents the parallel execution of the procedures started in local processors. α, β, γ and δ are factors that will be different for each problem, but may be determined from known hardware and software latencies, instruction execution times etc., or obtained from measurements.

This formula for the performance gain reflects the empirically verified fact that execution time will not monotonically decrease when the number of processors is increased indefinitely, but will reach a minimum for an optimum number of processors. This optimum is reached when the extra communication overhead exceeds the decrease in execution time by further distributing the problem over an additional joint processor. In our neuronal network application the optimum number of processors was between 4 and 8, depending on the selected size of the network. The calculated speedup as a function of n, based on measurements of the execution times of individual parts of the program in a single processor, was in good agreement with the observed values. Compared to execution of the matrix algorithms on the PDP11/73 without the use of joint processors, more than sixty times faster execution could be obtained with the optimum number of 8 TMS 32020 processors. This large improvement is partly due, of course, to the short cycle time (200 nsec) and single cycle execution of most instructions on this processor type.

7. Conclusions

Our experimental multiprocessor system has clearly demonstrated that the advantages of parallel processing can be made available to an applications programmer even with rather simple tools added to the environment he is used to.

Behind these tools it is necessary, however, to devise efficient and flexible communication mechanisms in hardware and operating software. Programming the applications-oriented routines for the joint processors in such a hierarchical scheme requires an additional effort. High-level languages for joint processors with particularly well adapted instruction repertoires for specialized functions (signal processing, graphics, image processing, string manipulation etc.) are not readily available at present. It is to be expected, however, that specialized devices of all kind will very soon be incorporated in small to medium computer systems and greatly enhance their capabilities by distributing a computational task over several cooperating units.

Managing the communication and the transport of data is one of the most important problems that must be solved to realize this goal. Classical bus systems are soon a limiting factor determining the attainable performance gain.

Fast caches or local data and program memories associated with each processor are important parts of the system architecture as cycle times of processing devices keep decreasing. There is a need for devices supporting parallel and unattended communication between these memories. The serial link as implemented in the transputer processor [13,14] from INMOS Ltd. is an extremely well conceived solution to this problem. In multiprocessor systems, data transfer rates must be equivalent to processing rates and the start-up time for a transfer should be as short as possible. The time needed to design and realize the hard- and software for our experimental system has been short. Altogether the total effort did not exceed about six man-months each for hardware, operating system and applications development, showing that this is well within reach of a small research group.

References

[1] TMS 32030 User's Guide, Texas Instruments Inc., (1985).

[2] PDP11FORTRAN 77 Language Reference Manual AA-V193A-TK, Digital Equipment Corporation, (1983)

[3] MICRO/PDP11 Handbook, Digital Equipment Corporation, (1983).

[4] D.J. Kuck, A survey of parallel machine organization and programming, *ACM Comput. Surveys* 9, (1977) 25–59.

[5] D. Padua, D. Kuck and D. Lawrie, High-speed multiprocessors and compilation techniques, *IEEE Trans. Comp.* C29, (1980) 763–776.

[6] M. Wolfe, Multiprocessor synchronization for concurrent loops, *IEEE Software*, (Jan. 1988) 34–42.

[7] G.R. Andrews and F.B. Schneider, Concepts and notations for concurrent programming, *ACM Comput. Surveys* 15, (1) (1983).

[8] R.H Halstead, Parallel symbolic computing, *Computer* 19, (1986) 35–43.

[9] J.R. Allen and K. Kennedy, Automatic loop interchange, *SIGPLAN Notices* 19, (1984) 233–245.

[10] S.P. Midkiff and D.A. Padua, Compiler algorithms for synchronization, *IEEE Trans. Comput.* C36, (1987) 1485–1495.

[11] C. von der Malsburg, Pattern recognition in labeled graph matching, *Neural Networks* 1, (1988) 141–148.

[12] TM 320C25 C Compiler Reference Guide, Texas Instruments Inc., (1987).

[13] D. May, R. Shepherd and C. Keane, Communicating process architecture: Transputers and Occam, in: Future Parallel Computers, P. Treleaven, M. Vanneschi (Eds), Lecture Notes in Computer Science (Springer Verlag) 272, 35–81.

[14] The Transputer Family 1987, Product Information, INMOS Ltd. Bristol, (1987).