Homogeneous multiprocessor system: a status report

Nikitas J Dimopoulos, Kin Fun Li, Eric Chi-Wah Wong*, R V Dantu and J W Atwood*

The homogeneous multiprocessor system described in this paper provides nearest-neighbour communication through shared memory, as well as global communications through a high-speed local area network. These aspects of the homogeneous multiprocessor architecture are investigated in detail. For the high-speed local area network, its data-like protocol and performance analysis are provided. A layered operating system currently under implementation for the homogeneous multiprocessor is introduced, and finally the paper presents the performance of the system for specific applications, based on experiments conducted with the simulator, specifically written for this multiprocessor.

Keywords: multiple processing, homogeneous multiprocessor system, communications, local area network, architecture, protocols, performance, operating system

In recent years, multiprocessors have become important in solving problems where a large amount of computation is needed. Several multiprocessors have been proposed or built; some of the best known machines being the C.mmp[34], Cm*[31], NYU Ultracomputer[11], PASM[28], Caltech's Cosmic Cube[27], etc.

A major architectural issue involved in the design of such machines is the availability of information pathways that enable the exchange of information between processors. Most of the existing MIMD designs have opted for a complete graph solution incorporating crossbars, multistage interconnection networks or point-to-point connections.

There are significant examples of computations though (e.g. relaxation processing[10,35], neural network simulation[8], digital signal processing), where the computation may be formulated in such a way so that each computational subtask needs to cooperate with a limited number of neighbouring subtasks in order to complete its computation. Such computations map into and benefit from architectures that limit the scope of interprocessor communication but make these limited communication pathways fast.

In this work, one such multiprocessor system, namely the homogeneous multiprocessor[7,9], shall be presented. This system provides nearest-neighbour communication through shared memory, as well as global communications through a high-speed local area network.

In particular, the discussion shall be focused on the following topics:

- Introduce the structure of the homogeneous multiprocessor, including the nearest-neighbour communication scheme as well as the high-speed local area network. For the latter, its data-link protocol and performance analysis shall be provided.
- Introduce a layered operating system, currently under implementation for the homogeneous multiprocessor.
- Present the performance of the system for specific applications, based on experiments conducted with the simulator specifically written for the homogeneous multiprocessor.

STRUCTURE

Overview of the architecture

As shown in Figure 1, the homogeneous multiprocessor is a tightly-coupled MIMD architecture, composed of k ($k \geq 3$) processing elements, k memory modules, $k+1$ interbus switches S_i isolating the processing elements

Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC Canada V8W 2Y2
*Concordia University, Montreal, Quebec, Canada

0267-6192/89/040227-13 $03.00

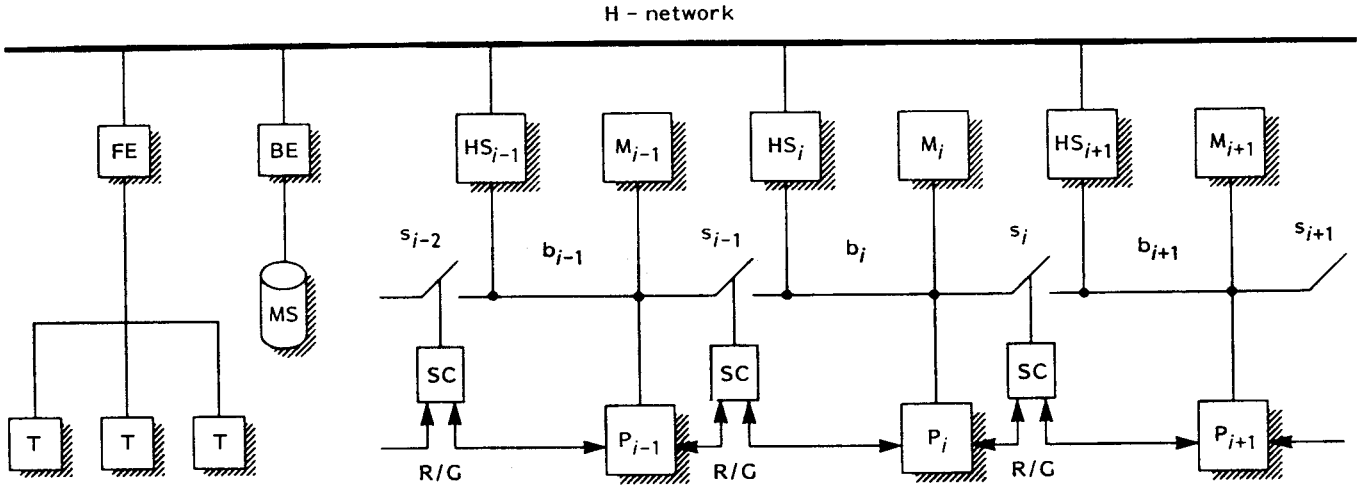

Figure 1. Homogeneous multiprocessor architecture with following notation: P: processor, FE: front end, b: local bus, HS: H-network station, M: memory module, BE: back end, SC: switch controller, T: terminal, MS: mass storage, R/G: bus request/grant

from each other, and the H-network which is a fast local area network used for point-to-point and broadcast mode communications. The architecture is considered to be composed of two parts: namely the homogeneous multiprocessor proper incorporating the processors, memories and interbus switches, and the H-network.

Each processing element P_i owns its local memory module M_i which it accesses via its local bus b_j; it also has the exclusive use of the respective network station HS_i. The local buses are separated by the intervening switches S_i. These switches provide each processor P_i with the ability to access the memory modules of either one of its two immediate neighbours by requesting the appropriate switch to close. Also, for I/O or data transfers to/from distant processors, each processor may utilize the H-network.

In the following sections, the functions of the processing elements communicating with each other shall be discussed in detail.

Network of the interbus switches

As it was outlined in the previous section, each processing element P_i, operates independently of its neighbours, but it is also provided with the capability of communicating with either one of its two immediate neighbours P_{i-1} and P_{i+1} via their respective memory modules. This is facilitated through the creation of 'extended buses'. An 'extended bus' is the dynamic fusion of two neighbouring local buses b_i and b_{i+1} effected through the closing of the intervening switch S_i after a request from either one of the two processors which are adjacent to the switch. Once an 'extended bus' is created, it exists for the duration of the request, normally one memory cycle, and it deteriorates to its component local buses once the request ceases to exist.

The process of creation of an 'extended bus' passes through two phases. The first phase, which is processor independent, ensures the safe and live operation of the S switches[7]. The second phase is processor dependent[18], and it is during this phase that a switch physically closes. In the next section, the operational algorithm shall be presented; this determines the behaviour of the interbus switches during the first phase of their operation.

Switching Operational Algorithm 1.2

A switch can exist in one of three logical states. The next state transition of a switch is based on the presence of a request for it to close, and the present state of itself and its two neighbouring switches. The three states a switch can exist at are as follows:

Open. This state signifies that no requests exist or if a request exists it will not be honoured immediately, because a neighbouring switch is currently servicing a request.

Gray. This state signifies that a request is acknowledged and that service (i.e. switch closure) will be granted in the immediate future.

Closed. This state signifies that it is safe for a switch to close.

The actual closure of the switch will take place during the second phase which commences immediately after a switch enters the closed state.

The Operational Algorithm 1.2 that decides the next station transition is as follows:

Algorithm 1.2
For a switch S_i
  If no request exists, it becomes Open;
  Otherwise, if a request exists then:
    If Open, it becomes Gray provided that the switch is to its left, S_{i-1}, is Open.
      Otherwise, it remains Open.

If Gray, it becomes Closed provided that the switch to its right, S_{i+1}, is Open.
Otherwise it remains Gray.
If Closed, it remains Closed.
The leftmost S_0, and rightmost S_k, switches are always Open.

A request (from a processor to a switch) remains asserted during the request period which ends with an acknowledgement (from the requested memory module to the requesting processor). In other words, a processor requests the access of a neighbouring memory module. This request is intercepted by the appropriate switch and forwarded to the memory module after the switch closes; then the memory module acknowledges the transfer of data to the requesting processor, and this terminates the cycle. Algorithm 1.2 guarantees the safe (i.e. no two adjacent switches will close at the same time) and live (i.e. a switch is requested to close, will eventually do so) operation of the network of the interbus switches. Details can be found in Reference 7.

H-network

The second component of the structure is the H-network[5,6]. This is a high-speed ($\sim$14 Mbyte/s) local area network with a structure resembling that of the Ethernet[21], yet it utilizes separate pathways for data transmission and network acquisition. The H-network has been designed for network spans of the order of 10 m. This makes the signal propagation delay extremely short, which coupled with the parallelism employed has as an effect the increased performance of the network. The H-network operates under a carrier-sense multiple access protocol which eliminates the existence of collisions by incorporating prescheduling and parallelism. In other words, the ready stations contend for the network during the transmission of a packet. Thus, at the end of the current transmission period, it is expected (with high probability), that the next master of the network has been chosen. Thus, collisions are eliminated, and the utilization of the network increases correspondingly.

The two activities, that is, channel contention and packet transmission must occur without interference from each other, and they are therefore provided with two separate channels. The contention channel is used by the stations to decide on the next master of the network, while the transmission channel is used for the actual packet transmissions.

Normally, the contention channel is of a much lower capacity as compared to the transmission channel. In this implementation[5,6], the transmission channel is a 16-line parallel bus, while the contention channel utilizes a single line.

Similar approaches have been proposed by Hamacher *et al.*[12], with a contention channel resembling a token ring rather than a bus, by Mark[20], with a slotted contention channel, and by Jafari *et al.*[14], where the contention channel is a loop, with a loop master that controls traffic on the 'contention loop' and access to the 'data loop'. In this scheme, the contention channel has the topology of a bus, which does not impose any transmission priorities on the competing stations, and also avoids the time delays incurred during the passing of the token.

In the subsequent sections, the proposed collision-free protocol shall be presented and also its throughput as a function of the participating network nodes. More details of the new CSMA/CF protocol can be found in References 4 and 33.

New collision free (CSMA/CF) protocol

The transmission channel is considered as a resource to be shared among several stations. The contention channel plays a similar role as a semaphore and its function is to ensure that the transmission channel will be allocated to at most one station at any particular time.

Each station is capable of transmitting a carrier signal over the contention channel, and it is also capable of distinguishing whether zero, one or more carrier signals are present. An example of such a channel and the hardware required, can be found in References 4 and 33. In addition, a network station, is capable of determining whether the transmission channel is free or not. All these actions take finite time, in accordance with the signal propagation and processing delays in the network.

At time t_0, a ready station inspects the contention channel. If it finds at least one carrier present (indicating that a contention is already in progress), the attempting station is blocked, and it must try again at a later time. Otherwise, if no carrier was found, then the station initiates the transmission of its carrier at time $t_0 + w_1$.

The delay w_1, was introduced here because two different operations (inspecting the channel and initiating the transmission of its carrier) are performed by the attempting station. These two operations, because of the finite processing speed, can neither happen instantaneously nor simultaneously.

Once a station has initiated the transmission of its carrier, thus obtaining the right to compete for the use of the transmission channel, it must ensure that it is the only one with this right. Thus it waits until all the other ready (that sensed the contention channel as being idle) stations initiated the transmission of their own carriers, and then it re-examines the contention channel at time $t_0 + w_1 + w_2$. The time delay w_2, is chosen so that all potential ready stations will be given the opportunity to initiate the transmission of their own carrier, and it also reflects the various signal propagation delays over the network. The range of values for the delay w_2 has been calculated to be:

$$w_2 \geq w_1 + 2a \qquad (1)$$

where $2a$ is the round-trip propagation time to the farthermost station in the network.

If at time $t_0 + w_1 + w_2$, the station finds only one carrier present, then it is assured that it is the only station remaining and it can, therefore, utilize the transmission

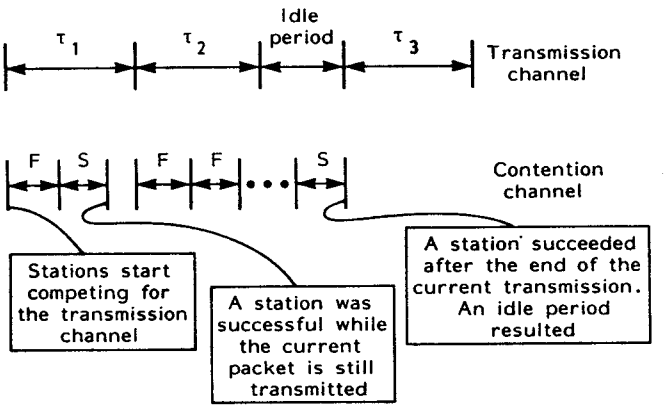

Figure 2. Activities at the transmission and contention channels

channel as soon as the current transmission period terminates. Such a station will be called a successful station.

A successful station shall withdraw its carrier simultaneously with the commencement of the transmission of its own packet. This is done so that the commencement of the next contention period will coincide with the commencement of a packet transmission period.

Otherwise, if at time $t_0 + w_1 + w_2$, the station found more than one carrier present, then it withdraws its own carrier at time $t_0 + w_1 + w_2 + w_3$ and it re-executes the protocol. Again, the introduction of the delay w_3 is due to the two distinct operations of determining the number of carriers present, and of withdrawing a single carrier.

Calculation of the average utilization factor

It is assumed that there are always N competing stations, and that the elapsing time between successive attempts by a station to gain the network is constant and finite. This period is denoted as T.

Thus, the probability distribution of the arrival time of a station attempting to acquire the network, is continuous and uniformly distributed in $[0, T]$.

The probability of failure for a single station to acquire the network after one contention period has been calculated[4,33] to be:

$$P_f = 1 - P_s = 1 - \left(\frac{T - w_3 - a}{T}\right)^N \tag{2}$$

The possibility of success is used to calculate the average idle period under the new CSMA/CF protocol. Recall that since there are no collisions, the average transmission and busy periods are equal.

Since separate contention and transmission channels are provided, the network stations may contend while a packet is transmitted over the transmission channel. The probability of failure, as given by Equation (2), refers to a single contention period. It is evident, that as the number of stations N increases, the probability of failure approaches 1. Thus, in very heavy traffic, it is expected that the number of contentions required before the next master is decided increases. This has an effect the corresponding increase of the total contention period. When the total contention period surpasses the packet transmission period, idling of the network occurs, with the corresponding decrease in the network utilization factor.

This section provides the calculation for the average idle period, as it depends on the number of stations contending. The final goal is to compute the average utilization factor in a network operating under the new CSMA/CF protocol. The activities on the transmission and contention channels, are depicted in Figure 2.

Denote by x the length of an unsuccessful contention period, by y the length of a successful contention period, and by C_k the length of a total contention period composed of $k-1$ unsuccessful plus one successful contention. Then, these quantities are related as:

$$C_k = (k-1)x + y \tag{3}$$

and $x = t + w_1 + w_2 + w_3 + a$, $y = t + w_1 + w_2$ where t is the arrival time of the first of the N contending stations. The random variable t is distributed over $[0, T]$ with density

$$\frac{N}{T}\left(\frac{T-t}{T}\right)^{N-1}$$

The length of an idle period, given k contentions, is given therefore as $I_k = H(C_k - \tau)$, where τ is the packet transmission period and k is the number of contention periods, and

$$\mathbf{H}(x) = \begin{cases} x & \text{if } x \geqslant 0 \\ 0 & \text{if } x < 0 \end{cases}$$

The average idle period is obtained as the expected value of the idle periods arising from k contentions. It is calculated as follows:

$$\bar{I} = \mathbf{E}[\bar{I}_k] = \sum_{k=1}^{\infty} \mathbf{Pr}[\text{there are } k \text{ contention periods}]\bar{I}_k$$

$$= \sum_{k=1}^{\infty} P_f^{k-1} P_s \bar{I}_k \tag{4}$$

$$\bar{I}_k = \int_{\tau}^{\infty} (z - \tau)\mathbf{Pr}[C_k = z]\mathrm{d}z \tag{5}$$

Using Equation (5) and the central limit theorem, the average idle period given k contentions is obtained as:

$$\bar{I}_k = \int_{\tau}^{\infty} (z-\tau)\frac{1}{\sigma_k\sqrt{2\pi}} e^{-(z-\eta_k)^2/2\sigma_k^2}\mathrm{d}z$$

$$= \frac{\sigma_k}{\sqrt{2\pi}} e^{-(\tau-\eta_k)^2/2\sigma_k^2} + \frac{\eta_k - \tau}{2}\,\mathrm{erf}\left[\frac{\tau - \eta_k}{\sqrt{2}\sigma_k}\right] \tag{6}$$

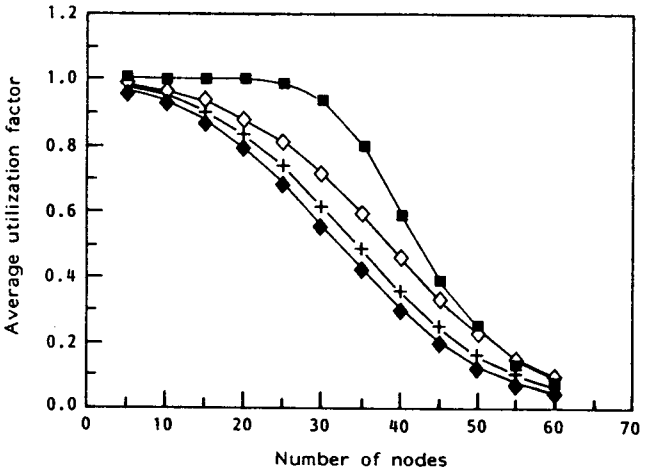

Figure 3. The average utilization factors of comparable CSMA/CD and CSMA/CF networks. —■— *CSMA/CF* ($w_1 = 0.1, w_2 = 0.3, w_3 = 0$); —◆— *CSMA/CD* ($\delta = 1.0$); —+— *CSMA/CD* ($\delta = 0.8$); —◇— *CSMA/CD* ($\delta = 0.5$). *For all cases,* $\tau = 30.0$, $T = 1.0$, $a = 0.1$

where erf is defined as

$$\text{erf}(a) = \frac{2}{\sqrt{\pi}} \int_a^{\infty} e^{-t^2} dt$$

It has been proven[4,33] that the series in Equation (4) converges. Equation (4) has been used to compute the average utilization factor of a network operating under the new CSMA/CF protocol.

The average utilization factor is given by the well-known[15] formula

$$\bar{S} = \frac{\bar{U}}{\bar{I} + \bar{B}} \tag{7}$$

In our case, since there are no collisions, $\bar{U} = \bar{I} = \bar{t}$ and Equation (7) can be simplified into

$$\bar{S} = \frac{\bar{t}}{\bar{I} + \bar{t}} \tag{8}$$

Equation (8) was used in order to evaluate the average utilization factor as a function of the number of stations N involved for characteristic waiting times. This function is depicted in Figure 3.

OPERATING SYSTEM DESIGN FOR THE HOMOGENEOUS MULTIPROCESSOR

System software implementation for the homogeneous multiprocessor is based on the concept of an operating system nucleus—the HM-nucleus[19], which system and user application software are built upon.

The HM-nucleus itself is a layered design and follows the approach suggested by Brown, Denning and Tichy[2]. In a layered structure, operations are defined in a hierarchy such that an operation at a higher layer is carried out by operations specified in lower layers only. Detailed operations in the sublevels, therefore, are hidden from the current level and transparent to the invoker. Modifications or additions of new modules can be done efficiently in such a layered structure and are pertinent to the affected levels only.

According to Brown, the hierarchical structure of a model operating system consists of fifteen levels*. The first eight levels (1–8) correspond to the general single machine operating system structure; while levels 9 through 14 correspond to the multimachine levels. The last level (15) is the user interface, the shell.

Of particular interest are levels 9 through 12, which are the communications, the file system, the I/O device and the stream I/O levels. Such a hierarchical structure provides the template between hardware and software mapping. Several of the layers, especially communications, map directly to our hardware and this is exploited in our design.

The HM-nucleus follows this hierarchical structure approach and it consists of eleven layers. Functions accessible in each layer are highly modular and their implementation are transparent to invokers. Based on the functions available in the HM-nucleus, the features discussed are being implemented in this paper. These include interprocessor communication through the network and the extended buses, and distributed virtual memory management. But first, the implementation and the structure of the HM-nucleus will be discussed.

HM-nucleus overview

The V Kernel[3], Roscoe[30], and Accent[24] are examples of contemporary distributed operating system designs based on the concept of a kernel. The operating system described in this paper is also based on the kernel structure; however, the term kernel is reserved for the lowest layer in our model and uses the term nucleus (i.e. the HM-nucleus) to denote the abstraction referred to as kernel in the aforementioned systems.

The HM-nucleus provides primitives for interprocessor communication, capability checking, memory management, process management and I/O handling. An identical copy of the nucleus resides in each individual processor. The HM-nucleus provides 'secure' abstractions of the underlying hardware. These abstractions, in the form of communication primitives and process management, are needed by the operating system and user applications.

The HM-nucleus consists of eleven layers, as shown in Table 1. From bottom up, the layers are: kernel,

* The model operating system described by Brown *et al.*[2] consists of the following levels, from 1 to 15: electronic circuits, instruction set, procedures, interrupts, primitive processes, local secondary store, virtual memory, capabilities, communications, file system, devices, stream I/O, user processes, directories, and shell.

Table 1. Data abstraction used in the HM-nucleus

Layer	Data abstraction
User interface	Shell commands
Table	Capabilities and catalogues
File management	Unix compatible files
Virtual memory	Local users' program, data, and stack space; shared region; global memory
Communications	Byte streams
RPC	Byte streams
UDS	Messages of arbitrary length
Capabilities	Capabilities
Device	Packets, mutant packets
Physical	Memory segments
Kernel	Descriptors

physical memory management, device management, capabilities, universal datagram services, remote procedure call, communications, virtual memory management, file management, table and user interface. Data abstraction used in different layers are also presented in Table 1, while designed and specified function and procedure calls available in some of these layers are shown in Table 2.

A homogeneous multiprocessor node includes the main processing unit (MPU) and its associated memory module, the memory management unit (MMU), the interbus switches, and the H-network interface. The kernel is the hardware/software interface layer that provides mechanisms to drive the hardware within a single node, and it incorporates no policy-making modules. These kernel primitives serve as extensions of the bare hardware and are used by higher layers. These extensions include process switching, primitive I/O, interrupt handling, and MMU manipulation.

The kernel also has provisions to enable and disable external interrupts coming into the local node. Since the kernel is the lowest structure residing in individual processors and does not interact directly with the rest of the system (interprocessor communication is handled by higher layer software), it thus acts as a single machine abstraction and hence single-node mutual exclusion can be enforced (e.g. monitor abstraction).

The physical memory management layer is responsible for the allocation and deallocation of memory space for processes and communication packets. With the cooperation of the MMU, it provides virtual-to-physical memory mapping and low-level access rights checking. Our present implementation provides a 1 Mbyte local memory module per processor, plus two extra 1 Mbyte of nonlocal memory modules belonging to the right and left immediate neighbours. Nonlocal memory is allocated in the form of shared regions. Allocation of available memory is performed by a Buddy algorithm that finds enough holds in the memory to satisfy a request. Minimum segment size is 4 kbyte and segments are allocated through the *AssignSegments* and *BindSegment* calls (see Table 2). Garbage collection, returning segments that are no longer needed (e.g. the packets used for communications), is invoked through the *MakeSpace* call.

The device management layer provides device drivers for the peripherals attached to a node. Usually, the only devices will be the H-network controller and the interbus switches implemented as virtual channels for neighbouring-processor communications, but specialized nodes may include disc controllers.

The bottom fourth layer, or capabilities, provides mechanisms for the creation and management of capabilities. A capability, in the context of the HM-nucleus, is a unique name consisting of a type, a processor number, and an identification that points to the information concerning that object.

The universal datagram service (UDS) layer implements various interprocessor communication protocols, thus providing a uniform access interface to the H-network, serial channels, and the neighbouring memory modules. Services provided in this layer format messages of arbitrary length into packets or reconstruct received packets into meaningful messages according to the specific protocol used. The UDS layer can also be expanded to encompass any additional communication pathways deemed necessary[22]. The placement of the UDS layer that low in the hierarchy is necessitated by the capability of our architecture to function as a multiprocessor in addition to being a distributed system. This is an important difference between our model and the one proposed by Brown *et al.*[2].

On top of UDS layer is the remote procedure call (RPC) layer. This layer serves as an interface between the abstractions of single machine and multimachine. Any internucleus communication initiated by a local machine is routed to the appropriate recipient, through the UDS by the RPC Manager.

The next layer in the HM-nucleus is the communications layer. This provides communication links between *user* processes (processes outside the HM-nucleus). Pipelines are supported by the H-network, while primitive message packets are transferred to neighbouring processors using channels through the extended bus (which will be described later). Two types of pipe communication mechanisms are designed into the system. The first is the individual pipe communication on a one-to-one basis. The second is a broadcast facility with a process sending messages to a selected group of receiving processes[2]. Both types of pipe communication mechanisms are supported directly by the H-network, which being a local area network, supports both point-to-point and broadcast communications. Due to the similarities between the H-network and the Ethernet, the network communication protocol used will adhere to the IEEE 802 standard[1]. This layer is based on the UDS layer discussed above, and uses the facilities of the H-network and the extended but for process-to-process, multicast, and broadcast-mode communications.

The virtual memory management (VMM) layer is responsible for assigning and managing virtual space for

Table 2. HM-nucleus and its functions

Layer	Function	Description
User interface	Shell commands	User and nucleus interface
Table	*Map	Associate the capability of an object with a name
	GetName	Searches for the given named object, and returns its associated capability
	*UnMap	Disassociates the name of a named object from its capability
File management	*Read	Read a file
	*Write	Write a file
	*Open	Open a file
	*Close	Close a file
Virtual memory	*CreateRegion	Allocates space for a shared region
	*BindRegion	Binds a created region to a specific virtual port
	*EnterRegion	Obtains exclusive access of a shared region
	*ExitRegion	Exits a shared region
	*UnBindRegion	Disassociates a region from its virtual port
	*DestroyRegion	Destroys a shared region and frees the allocation space
Communications	*link.open	Initiates handshaking protocol to create a link
	*link.send	Sends message through established link
	*link.rece	Receives message through established link
	*link.close	Disconnects communication link
RPC	*Rpc	Performs a remote procedure call to a specified node
UDS	HNet.send	Formats messages of arbitrary length into packets according to the H-network communications protocol
	HNet.rece	Strips away communication overhead of incoming messages and reconnects fragmented message packets into a unique coherent message of arbitrary length interpreted by upper layers
Capabilities	*CreateName	Generates a unique machine name
	*Map	Associates an object with an already created capability
Device management	HNet.send	Invokes network driver for send operation
	HNet.rece	Invokes network driver for receive operation
	channel-send	Invokes extended bus driver for send operation
	channel-rece	Invokes extended bus driver for receive operation
Physical memory	AssignSegment	Assigns physical memory blocks upon request
	BindSegment	Binds a virtual address to a physical address
	DeleteSegment	Deallocates the specified segment
	MakeSpace	Collects deallocated memory blocks to the free list
Kernel	LoadDescriptor	Loads a descriptor into the MMU
	ReadDescriptor	Obtains the contents of a descriptor from the MMU
	EnableDescriptor	Enables a specific descriptor in the MMU
	DisableDescriptor	Disables a specific descriptor in the MMU

* Denotes exported function to applications

processes in collaboration with the physical memory management layer. There are three kinds of such space: program, data, and stack space for user programs; shared regions; and global memory. At a later stage of our development, swapping of user spaces will be implemented in this layer.

The file management layer will implement a hierarchical file system, modelled after the Unix file structure[26]. Branches of the tree will be situated at nodes possessing a local disc, while nodes without discs will implement the semantics of the Unix file system interface and refer all requests to a server node. Server nodes will have a complete file system manager and will store portions of the file system data on their local discs. The initial implementation will support only access to the device (special) files.

The table layer consists of catalogues where the correspondences between symbolic names and their capabilities are entered. These catalogues are resident in the machines to which the capabilities belong. This level can be developed into a distributed directory layer when necessary.

The outermost layer is the user interface layer. Initially, this will contain the actual code of the application. Subsequent versions will be capable of managing user processes and responding to a general system call interface, thus providing the minimum user interface requirements to run application programs.

This paper is building HM-nucleus and user applications using VRTX[32] as part of the kernel layer. In the following sections, process communication, virtual memory management, and multiple-copy update mechanisms provided by our system, shall be discussed.

Communication using extended bus

There are two ways for a process to communicate with processes residing on neighbouring processors through the extended bus, by using channels and shared regions. Channels, a low-level communication abstraction available in the device layer, are primarily used for system functions and to transfer small messages. More extensive and frequent interprocess communication between processes running on neighbouring processors can be achieved more efficiently through the shared region schemes, as a virtual shared memory, which will be described in the next section.

Channels are predefined well-known addresses established during system start-up time that are used for handshaking dialogue between adjacent processors. They have fixed locations and are available throughout the system up-time. Channels are assigned to running processes upon request through the peer-to-peer protocol available in the communications layer (analogous to the OSI transport layer).

Communication using channels can be achieved by requesting a switch to close. The MMU, together with the switching controller, perform the address translation and mapping to nonlocal memory, and create an extended bus for communication with neighbouring processors. Packets, the basic unit of information used for interprocessor communication, are stored in the local memory space. When a packet is created, its address is transformed into a pointer which is subsequently stored in the predefined channel for the receiving processor.

The sending processor then asserts an interrupt to the receiving processor. The receiving processor, upon the interrupt, will read the packet address pointer from the sender's memory space and pass this information to a waiting lightweight process/handler, which will use the pointer to obtain the packet when it is scheduled to run.

The process of packet delivery thus involves two phases. During the first phase, the packet pointer and its handshaking is the responsibility of the interrupt handling modules. The handshaking and interpretation of the packet itself is handled by both the consuming process and the producing process in the second phase.

Occasionally, it is desirable for a processor P_i to signal a waiting processor that is two processors further away (i.e. P_{i-2} or P_{i+2}). There are two ways of reaching such a processor: through the H-network or through the intervening processor P_{i-1} or P_{i+1}. Given the close proximity of the processors, the route through the network is believed to be more expensive due to network communication overhead. Therefore, it has been chosen to implement the second mechanism, and use it for signalling purposes such as the one found in the mutual exclusion algorithm for gaining entrance to a shared region[19].

This mechanism uses a 'mutant packet' that forces processor P_{i-1} or P_{i+1} to signal its neighbour P_{i-2} or P_{i+2}. As described earlier, any processor with a packet ready to be consumed by a neighbour indicates this state by interrupting the neighbour and providing the neighbour with a pointer to the packet to be consumed. For the mutant packet mechanism, only the pointer prefixed with a special code indicating the mutation shall be used. The pointer points to an address in a neighbour's memory that needs to be altered, instead of pointing to a packet. The interrupt handler, upon receiving such a mutant packet, immediately carries out the operation indicated and does not pass any information to waiting processes, as it would have normally done.

Distributed virtual memory management

There are three types of virtual space that exist in our system: user programs, shared regions, and global memory. The first type, user programs, normally maps into local physical memory, reserving space for Unix-like processes partitioned into program (text), stack, and data areas. The other two kinds of virtual space are managed by cooperating processes in a distributed fashion. The shared regions implement bounded global memory among units of three neighbouring processes that use the extended bus to communicate. The global virtual memory, on the other hand, is provided by replicating data structures distributed on a number of processors in the system, which communicate with each other either through the H-network or the extended bus.

Shared regions

A shared region, an abstraction of global memory shared by three adjacent processors, encapsulates a collection of data and, if desired, an implicit mechanism for mutual exclusion. The synchronization is achieved by a spin-lock, where a central semaphore is maintained but a waiting processor spins within its local memory waiting for a signal from the processor which is currently using the shared region. Hence, interprocessor interference through the switching network is minimized. Details of such a mutual exclusion algorithm can be found in Reference 19.

A shared region is created through the *CreateRegion* call available in the virtual memory management layer. During creation time, the caller specifies if the region is guarded or unguarded. A guarded region indicates that the data will be shared on a mutually exclusive basis and therefore synchronization primitives are also created as a result of the *CreateRegion* call. The shared region is created by the processor local to the node where the region resides. Neighbouring processor(s) will have to issue the *GetName* call in the table layer to obtain the region's capability. This capability is then bound to the virtual address space of the sharing processor by the *BindRegion* call.

The *EnterRegion* call provides mutually exclusive access to a shared guarded region by executing the spin-lock algorithm discussed earlier and enabling the pertinent MMU descriptors. If the descriptor corresponding to the shared region is disabled, any reference made to the shared region will result in a fault, notifying the MPU in the form of a trap.

ExitRegion is used to exit from the shared region by manipulating the appropriate synchronization primitives. An unguarded region does not need any synchronization,* and therefore the *EnterRegion* and *ExitRegion* calls are transparent.

The concept of shared regions facilitates data and message passing between neighbouring processing elements, for applications such as image processing[23] and relaxation processing[10,35], where closely-coupled processors cooperate in a single task.

Distributed data structure management

As it was mentioned before, the file structure will be modelled after Unix's. This arrangement will enable us to implement an efficient file structure distributed over the multiprocessor, and also, by adhering to the Unix file system semantics, to interface with any Unix-based system or network.

The file system issues shall be addressed in a later work. The current section discusses the design of some mechanisms which are necessary in order to manage any distributed data structure. In particular, for the purposes of this work, any data structure can be regarded as a database entity. Such an entity is then allowed to be distributed and/or replicated on several nodes. Reliable and consistent access and update mechanisms are provided, so that the file system and applications, located at the top levels of our hierarchy, can benefit.

Multiple-copy update problem

Numerous algorithms to solve the multiple-copy update problem exist in the literature. For instance, the 'bakery algorithm'[16] generates unbounded sequence numbers to provide first-come first-served priority into critical sections; Ricart and Agrawala's algorithm creates mutual exclusion in a computer network whose nodes communicate by messages[25]. In general, most methods used to solve the multiple-update problem can be categorized as:

- Global locking mechanisms: including the two-phase commit protocol.
- Time stamp approaches: which are based on event ordering.
- Circulating tokens: where the update is performed by one node only at any one time and the serializability of updates is guaranteed.
- Voting or majority consensus: where synchronization is achieved based on a dialogue among the participating nodes.

The methods proposed are based on the observation that in certain classes of problems, a global database is read often but updated infrequently. Many applications do not require up-to-date (in terms of millisecond) information; they will operate correctly (although possibly somewhat more slowly) with 'stale' information. For all applications though, consistency must be maintained. A particular example is those artificial intelligence applcations where a global data structure (blackboard) is required (e.g. Hearsay I and Hearsay II[17]). Since the homogeneous multiprocessor does not have global shared memory, for the classes of problems mentioned above, it is possible to represent a shared data structure as a fully-replicated database, supported by mechanisms ensuring consistent updating.

For these mechanisms, the following assumptions and constraints have been made, which permit the use of specialized update control algorithms, that map efficiently onto the architecture of the homogeneous multiprocessor and take advantage of the broadcast capability of the H-network as well as the fast interprocessor communications channels provided by the extended bus mechanism:

- The targetted applications are computationally intensive, distributed applications, needing a structured global database (blackboard).
- Multiple readers can operate concurrently, but only a single writer is allowed to operate at any given time. A *modify* transaction is treated as an indivisible read-and-write operation on the latest version of the data structure.
- The readers far outnumber the writers.
- A writer, wishing to perform an update operation, is ensured the latest version of the data structure. If an up-to-date read is necessary, a *modify* (X, X) can be used.
- It is assumed that the database is small enough to fit within the available node-memory.
- Since there is no on-board cache at individual nodes, the cache coherence problem is not dealt with.
- Only a single user process runs at a node.
- Since migration of directories is not allowed in our system, the consistency problem due to migration is ignored.

Three methods of achieving the multiple-copy update are proposed. In all three methods, a token is utilized to achieve synchronization and reliability. The methods are distinguished from each other, by the communications pathway the token uses in order to reach a particular node.

Mutual exclusion mechanisms

For all three mechanisms, a token is used to both synchronize and validate the updates. The arrival of the token to a node gives it permission to proceed with an

* Within an unguarded region, the applications programmer himself can implement a synchronization system. Such a synchronization system is quite efficient (in the sense that it does not involve costly operating system function calls), but on the other hand it may not be safe.

update. The token incorporates an update sequence number, and the most recently updating station number.

Token passing through the extended bus. The first method is a token passing mechanism, in which a token is passed among a group of neighbouring processors, each working on its own copy of the same data structure. Only the processor possessing the token is capable of updating, and it broadcasts the update to all the other processors in the group through the H-network. The token circulates among these adjacent processors through the extended buses. This method implements a round-robin update schedule and is very efficient in a heavily-updating environment.

Token broadcast. In this method, a node wishing to perform an update will broadcast its intention and wait until it receives the token. The node currently in possession of the token will place the update request in a node-update-request queue, which is incorporated into the token itself. The node possessing the token proceeds with the consistency check (described below), broadcasts its update message to all the participating nodes, deletes its own entry from the node-update-request queue, and finally sends the token to the next requesting node. The token-passing scheme using the H-network has a fast response time and it is dynamic in nature as compared to the previous method. However, it is expensive due to the acknowledgement overhead involved.

Token controller. A third alternative is to assign a node as a token controller to manage update requests. The token controller maintains a queue of requesting nodes. The node at the head of the queue receives the token and proceeds with the consistency check and update. Once the update is carried out, the token is returned to the token controller.

This method also has a fast response time. If compared with the request broadcast method, it has less overhead since update requests are directed to, and always registered by the token controller. On the other hand, a malfunction of the token controller will cause failure.

Reliable and consistent updates

Synchronization of multiple updates on the same data structure can be resolved by the methods discussed. However, due to the distributed nature of our system, mechanisms to guarantee reliable updates have to be introduced to maintain system consistency. Given the proximity of the nodes, a mostly reliable communication environment is assumed. Yet, occasionally messages are lost, and the mechanisms presented here are designed to handle these cases.

An update message, originating from the node possessing the token, encapsulates the following information: a unique sequence number, a node number, and the update itself. The sequence number is a monotonically increasing number assigned to an update message. This number can be generated based on information obtained from the token. The node number is simply the number of the node that possesses the token at the time of the update.

The token itself must contain the following information: a unique sequence number, and a node number. The sequence number specifies the latest update message number. This sequence number is incremented and copied to the present update message by the node possessing the token. The token is released by the updating station, only after the update message has been sent by the network. The node number is the number of the last updating node.

Each individual node participating in the management of the distributed data structure, remembers the sequence number of the last received update message, and also any previous sequence numbers that are missing. The missing ones are the updating messages that have not been received and will be used for the consistency check.

For all three methods, each time a node receives the token, it compares the sequence number encapsulated in the token, with that of the last received update. If they do not match, then the node empties the network queue and checks again. The network queue consists of incoming packets from the network and since an update message is broadcast before the token is released, then the update messages should have arrived at the destination node before the token. If the two sequence numbers still do not match, then the last updating message is missing, and the node requests re-broadcasting from the last updating node. It is possible that more than one messages are missing, in this case re-broadcasts are requested from each updating nodes.

An updating node retains the updating message until it is ensured that all the participating nodes have performed the consistency check pertaining to this particular message. The condition upon which a node is guaranteed that the consistency check has been performed, depends on the mechanism chosen. For the token-passing mechanism, an updating node retains a particular message until the token reaches the node twice. This condition, since all the nodes are arranged in a linear array, guarantees that the token has circulated through all the nodes in the group, and hence each and every one of the participating nodes has been given the opportunity to perform a consistency check.

For the remaining two methods, a special consistency-control token is broadcast after a predetermined number of updates, which forces all the nodes to perform a consistency check. At the end of this process, when all the participating stations have performed their consistency check, all past update messages can be deleted, and the cycle may be started afresh.

It is understood that in between the issuing of the consistency-control tokens, nodes that receive the update token continue to perform consistency checks of their database. Nodes that receive out-of-sequence updates, will refrain from performing the update, until the missing update messages are received. The missing messages are requested from their source either immediately, or after a consistency check is performed.

Multiple-object updates and control algorithms

The token mechanisms as presented in the previous sections, can be expanded to accommodate a multiple-object data structure. In such a case, a separate token is assigned to each object, and thus updates on mutually exclusive objects can be carried out concurrently. In addition, if the objects (and their associated tokens) are linearly ordered, multiple-object updates can be achieved without deadlocks. The only requirement is to add a 'token-type' field to the token itself.

The mechanisms presented are suitable for systems with different rates of update requests. Multiple protocol synchronization schemes can be used within the same system, depending on the application and the system load. Thus the neighbour-to-neighbour token-passing mechanism is best suited for heavy rates of update requests, while the mechanism that utilizes a token manager is best suited for light rates of update requests. These two mechanisms could be combined together to form a hybrid scheme that adapts itself to a varying demand of update requests. This can be accomplished by incorporating a token manager in the chain of neighbouring nodes managing the database. The token manager can determine through the update sequence number, when possessing the token, if no updates happened during a predetermined maximum time interval. In such a case, the token manager retains the token until an updating node specifically requests the token manager for the token. For this hybrid scheme to work properly, the updating node must know the address of the token manager node, and request for the token, if the token has not reached it within a certain timeout period.

SIMULATED RUNS OF APPLICATIONS ON THE HOMOGENEOUS MULTIPROCESSOR

The homogeneous multiprocessor, being a closely-coupled MIMD architecture, is perfectly suited for context-dependent algorithms. The often used image processing algorithms such as smoothing, edge, detection, histogram generation, relaxation processing etc., can be easily mapped onto the architecture. Maximum concurrency is of course extracted from local algorithms such as smoothing. Nevertheless, global algorithms, such as histogram generation, can also benefit from the architecture.

In the next section, the implementation of a parallel histogram generation algorithm on our architecture shall be discussed, and results obtained through our simulator presented[23].

Histogram generation

A histogram of grey level content provides a global description of the appearance of an image, and it is frequently used in thresholding, while interactive histogram modification is used for enhancing image quality.

It is assumed that there are n ($n = 2^k$) nodes in the homogeneous multiprocessor. The image is divided into n strips, and each strip is loaded in the memory of each of the nodes, which calculate the partial histogram of the region assigned to them in parallel. The next step accomplishes the merging of the partial histograms. This is done through a form of recursive doubling[29]. Suppose that there are B grey levels in the image. Initially, processors P_{2l+1}; $l = 0, 1, 2, \ldots, (n/2 - 1)$ merge the $B/2$ least significant bins of the partial histograms contained in their own as well as the memory of their neighbours to the right. Similarly, processors P_{2l+2} merge the $B/2$ most significant bins located in their own as well as the memory of their neighbours to the left. Next, processors P_{4l+1}; $l = 0, 1, 2, \ldots, (n/4 - 1)$ transfer the $B/2$ least significant bins of their merged histograms to processors P_{4l+2}, while processors P_{4l+4} transfer the $B/2$ most significant bins to processors P_{4l+3}. At this point, processors P_{4l+2} and P_{4l+3}, contain partial B bin histograms, and the process is repeated. The final completed histogram, is found in processor $P_{n/2}$. Under this algorithm, the partial histograms are merged on a tree structure of processors embedded on the homogeneous multiprocessor as depicted in Figure 4.

The algorithm, is further optimized through the implementation of a form of a 'bucket brigade' to efficiently transfer long vectors between distant processors. Thus, in order to transfer a B-bin vector from processor P_i to processor P_j, the intervening processors form a pipeline through which the B-vector is transferred in $O(j - i + B)$ steps.

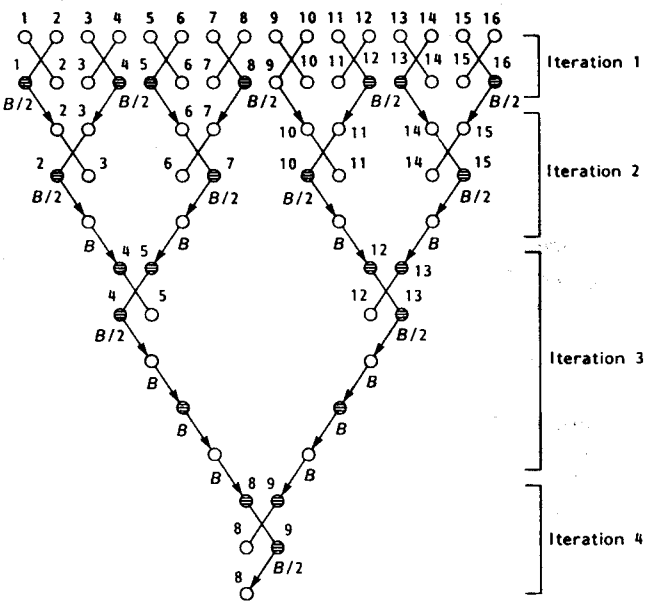

Figure 4. Example of the distributed merge algorithm on 16 processors. $\xrightarrow{x}$ transfer of a vector of x elements; × merge of two neighbouring partial histograms

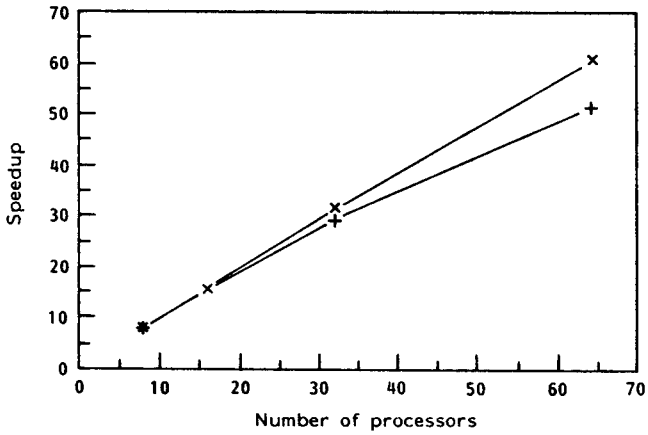

Figure 5. Speedup versus the number of processors for the distributed histogram algorithm as obtained through simulation and analysis. For image size = 1024 × 1024 (× simulation; — theoretical); image size = 512 × 512 (+ simulation; — theoretical)

For the algorithm outlined above, the speedup can be calculated as

$$S = \frac{M^2}{[M^2/n + B\log(n)/2] + r[n/4 + (2B-1)\log n - 2B + 1]} \quad (9)$$

where M^2 is the size of the image, n is the number of nodes, B is the number of grey levels in the image, and $r = t_{tr}/t_a$ is the ratio of the transfer over the add times. Figure 5 shows the speedup factor obtained through both simulations and Equation (9), against the number of processors, for histogram calculation for images of varying grey levels.

CONCLUSIONS AND DISCUSSION

In this work, an overview has been presented of the homogeneous multiprocessor system, a tightly-coupled MIMD architecture incorporating both nearest-neighbour communications as well as a novel and fast local area network. This system is currently under implementation at the Electrical and Computer Engineering Departments, Concordia University and the University of Victoria. As processing elements, 8-MHz MC68000 processors are being used together with MC68451 MMU. Each processing node includes 1 Mbyte of DRAM, while specialized VLSI components have been designed, and are currently under implementation using Northern Telecom's 5 μm CMOS process, for the interprocessor switch and H-network controllers. Also being designed and implemented is a modular distributed operating system for the described architecture. This operating system design is based on the advanced operating system model proposed by Brown *et al.*

ACKNOWLEDGEMENTS

The work reported in this paper was performed at the Departments of Electrical and Computer Engineering, Concordia University and University of Victoria, under grants from the Natural Sciences and Engineering Research Council Canada, the Fonds pour la Formation de Chercheurs et l'aide à la Recherche, and the Centre de Research Informatique de Montreal.

REFERENCES

1 ANSI/IEEE Standard 802,3-1985, *Local area networks (carrier sense multiple access with collision detection)* IEEE Inc., New York (1985)

2 **Brown, R L, Denning, P J and Tichy, W F** 'Advanced operating systems' *IEEE Comput.* Vol 17 No 10 (October 1984) pp 173–190

3 **Cheriton, D R** 'The V kernel: a software base for distributed system' *IEEE Softw.* (April 1984) pp 19–42

4 **Dimopoulos, N J and Wong, C W** 'Collision-free protocol for local area networks' *Comput. Commun.* Vol 11 No 4 (August 1988) pp 208–214

5 **Dimopoulos, N J and Wong, C W** 'Performance evaluation of the H-network through simulation' in **Tzafestas, S G (Ed.)** *Digital techniques in simulation communication and control* Elsevier Science Publishers BV, The Netherlands (1985)

6 **Dimopoulos, N J and Kehayas, D** 'The H-network: a high speed distributed packet switching local computer network' *Proc MELECON '83 Mediterranean Electrotechnical Conference* Athens, Greece (May 1983) pp A1.02

7 **Dimopoulos, N J** 'On the structure of the homogeneous multiprocessor' *IEEE Trans. Comput.* Vol C-34 No 2 (February 1985) pp 141–150

8 **Dimopoulos, N J** 'Organization and stability of neural network class and the structure of a multiprocessor system' PhD Dissertation, University of Maryland, College Park, MD (1980)

9 **Dimopoulos, N J** 'The homogeneous multiprocessor —architecture, structure and performance analysis' *Proc 1983 Int. Conf. on Parallel Processing* (August 1983) pp 520–523

10 **Fekete, G, Eklundh, J O and Rosenfeld, A** 'Relaxation: evaluation and applications' *IEEE Trans. on Patter Analysis and Machine Intelligence* Vol PAMI-3 (1981) pp 459–469

11 **Gottlieb, AR *et al.*** 'The NYU ultracomputer-designing an MIMD shared memory parallel computer' *IEEE Trans. on Computers* Vol C-32 (1983) pp 175–190

12 **Hamacher, V C and Shedler, G S** 'Access response on a collision-free local bus network' *Comp. Networks* Vol 6 (1982) pp 93–103

13 **Holt, R C** *Concurrent Euclid, the Unix System, and Tunis* Addison-Wesley, UK (1983)

14 **Jafari, H, Lewis, T and Spragins, J** 'A new ring-structured microcomputer network' *Proc. 4th Int. Conf. on Computer Communications* Kyoto, Japan (1978) pp 1434–1440

15 **Kleinrock, L and Tobagi, F A** 'Packet switching in radio channels: part I—carrier sense multiple access modes and their throughput-delay characteristics' *IEEE Trans. Commun.* Vol 23 (December 1975)

16 **Lamport, L** 'A new solution of Dijkstra's concurrent programming problem' *Commun. ACM* Vol 17 No 8 (August 1974) pp 453–455

17 **Lesser, V R *et al.*** 'Organization of the Hearsay II speech understanding system' *IEEE Trans. Acoustics, Speech, and Signal Processing* Vol 23 (February 1975) pp 11–24

18 **Li, K F and Dimopoulos, N J** 'The performance analysis of the homogeneous multiprocessor proper' *Canad. Electr. Eng. J.* (January 1987) pp 3–10

19 **Li, K F, Dimopoulos, N J and Atwood, J W** 'The HM-nucleus: a distributed operating system nucleus for the homogeneous multiprocessor *IEEE Micro* (February 1987) pp 14–24

20 **Mark, J W** 'Distributed scheduling conflict-free multiple access for local area communication networks' *IEEE Trans. Commun.* Vol COM-28 (December 1984) pp 1968–1976

21 **Metcalfe, R M and Boggs, D R** 'Ethernet: distributed packet switching for local computer networks' *Commun. ACM* Vol 19 (July 1976) pp 395–404

22 **Panzieri, F** 'Design and development of communication protocols for local area networks' PhD Dissertation, University of Newcastle upon Tyne (1985)

23 **Ramanamurthy, R V, Dimopoulos, N J, Li, K F, Patel, R V and Al-Khalili, A J** 'Parallel algorithms for low level vision on the homogeneous multiprocessor' *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition* Miami Beach (June 22–23, 1986) pp 421–423

24 **Rashid, R F and Robertson, G** 'Accent: a communication oriented network operating system kernel, *Operating Syst. Rev.* Vol 15 No 5 (1981) pp 64–754

25 **Ricart, G and Agrawala, A K** 'An optimal algorithm for mutual exclusion in computer networks' *Commun. ACM* Vol 24 No 1 (January 1981) pp 9–16

26 **Ritchie, D M and Thompson, K** 'The Unix time-sharing system' *Commun. ACM* Vol 17 (1974) pp 365–375

27 **Seitz, C L** 'The cosmic cube' *Commun. ACM* Vol 28 No 1 (January 1985) pp 22–23

28 **Siegel, H J *et al.*** 'PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition' *IEEE Trans. Computers* Vol C-30 (December 1981) pp 934–937

29 **Siegel, L J, Siegel, H J and Swain, P H** 'Performance measures for evaluating algorithms for SIMD machines' *IEEE Trans. Softw. Eng.* Vol SE-8 (July 1982)

30 **Soloman, M H and Finkel, R A** 'The ROSCOE distributed operating system' *Proc. Seventh ACM Symposium on Operating Systems Principles* (1979) pp 108–114

31 **Swan, R J, Fuller, S J and Siewioriek, D P** 'Cm*—a modular multimicroprocessor' *Proc. AFIPS Conf. 1977* Vol 46 (1977) pp 645–655

32 *VRTX32/68000, versatile real-time executive for the M68000 microprocessor* User's Guide, Ready Systems, Palo Alto, CA (1987).

33 **Wong, C W** 'A collision free protocol for LANs utilizing concurrency for channel contention and transmission' M.Eng. Thesis, Concordia University, Montreal, Canada (1985)

34 **Wulf, W A, Levin, R and Harbison, S P** *C.mmp—an experimental computer system* McGraw-Hill, New York (1981)

35 **Zucker, S W, Hummel, R A and Rosenfeld, A** 'An application of relaxation labelling to line and curve enhancement' *IEEE Trans. Comput.* Vol C-26 (1977) pp 393–403, pp 922–929