An Overview of Distributed Application Support on the Homogeneous Multiprocessor

by

Kin Fun Li
Nikitas J. Dimopoulos

Department of Electrical and Computer Engineering
University of Victoria
Victoria, B.C., Canada

ABSTRACT

The Homogeneous Multiprocessor is a tightly-coupled MIMD machine featuring both networking and shared memory. Operating system facilities for the Homogeneous Multiprocessor are based on an operating system nucleus—the HM-Nucleus—which is a hierarchical design that permits modularity, and information and decision hiding. This paper focuses on the features that support distributed applications such as image processing, tree search, and simulation. These features include a nearest neighbour communication scheme for adjoining processors using shared regions and the management of distributed data structures by using reliable and consistent updating mechanisms. The goal of this work is to provide users with abstractions of the prominent hardware features in running distributed applications.

1.0 INTRODUCTION

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], PASM [28], Caltech's Cosmic Cube [27].

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], 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 we shall present one such multiprocessor system, namely the Homogeneous Multiprocessor (HMP) [7,9]. This system provides nearest neighbour communication through shared memory, as well as global communications through a high speed local area network. A layered operating system, currently under implementation for the HMP, will be indtroduced. Support for distributed applications including the shared regions, which is a nearest neighbour communication scheme, and the alogorithms for the management of distributed data structure will be presented and discussed.

2.0 STRUCTURE

2.1 Overview of the Architecture

As shown in Figure 1, the HMP 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 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_i; 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.

2.2 The 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. To determine the behaviour of the interbus switches during the first phase of their operation, a switching algorithm is used [7]. 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. The switching algorithm 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 of this algorithm can be found in [7].

2.3 The H-Network

The second component of the structure is the H-Network [5,6]. This is a high speed (~14 Mbytes/sec.) 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 meters. 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, we expect (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 our implementation [5,6], the Transmission Channel is a 16-line parallel bus, while the Contention Channel utilizes a single line. A new Collision Free (CSMA/CF) Protocol has also been proposed. Details of the CSMA/CF Protocol and its analysis such as the average utilization factor of the network can be found in [4,33].

3.0 OPERATING SYSTEM DESIGN FOR THE HOMOGENEOUS MULTIPORCESSOR

System software implementation for the HMP 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. Such a hierarchical structure provides the template between hardware and software mapping.

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, we are implementing the features discussed in this paper. These include interprocessor communication through the network and the extended buses, and distributed virtual memory management. But first, we shall discuss the implementation and the structure of the HM-Nucleus.

3.1 The HM-Nucleus Overview

Accent [24] and V Kernel [3] are examples of contemporary distributed operating system designs based on the concept of a kernel. Our operating system is also based on the kernel structure; however, we reserve the term kernel for the lowest layer in our model and use 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 and it 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. From bottom up, the layers are: Kernel, Physical Memory Management, Device Management, Capabilities, Universal Datagram Services, Remote Procedure Call, Communications, Virtual Memory Management, File Management, Table and User Interface.

A HMP 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 non-local memory modules belonging to the right and left immediate neighbours. Non-local memory is allocated in the form of shared regions. Allocation of available memory is performed by a Buddy algorithm that finds enough holes in the memory to satisfy a request.

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 disk 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 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.

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 primitve message packets are transferred to neighbouring processors using channels through the extended bus (which will be described in Section 4.0). 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. 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. This layer is based on the UDS layer discussed above, and uses the facilities of the H-Network and the extended bus for process-to-process, multicast, and broadcast-mode communications.

The Virtual Memory Management (VMM) layer is responsible for assigning and managing virtual space for 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, modeled after the Unix file structure [26]. Branches of the tree will be situated at nodes possessing a local disk, while nodes without disks 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 disks. 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.

We are building HM-Nuleus and user applications using VRTX [32] as part of the kernel layer. In the following sections, we shall discuss process communications, virtual memory management, and multiple-copy update mechanisms provided by our system.

4.0 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 in the 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 non-local 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 light-weight 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, we have chosen to implement the second mechanism, and use it for signaling 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 we have 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, we use only the pointer prefixed with a special code indicating the mutation. 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.

4.1 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 processors in the system, which communicate with each other either through the H-Network or the extended bus.

4.1.1 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 [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. Within an unguarded region, the applications programmer himself can implement a synchomization system. Such a synchomization 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.

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], where closely-coupled processors cooperate in a single task.

4.1.2 Distributed Data Structure Management

As it was mentioned before, the file structure will be modeled 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.

We shall be addressing the file system issues 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, we regard any data structure 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.

4.1.2.1 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].

The methods we propose are based on the observation that in certain classes of problems, a global database is read often yet infrequently updated. 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 applications where a global data structure (blackboard) is required (e.g., HEARSAY I and HEARSAY II [17]). Since the HMP 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, we have made the following assumptions and constraints, which permit the use of specialized update control algorithms, that map efficiently onto the architecture of the HMP 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:

(a) The targeted applications are computationally intensive, distributed applications, needing a structured global database (blackboard).

(b) 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.

(c) The readers far outnumber the writers.

(d) 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.

(e) It is assumed that the database is small enough to fit within the available node-memory.

(f) Since there is no on-board cache at individual nodes, the cache coherence problem is not dealt with.

(g) Only a single user process runs at a node.

(h) 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, we utilize a token 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.

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

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.

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.

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.

4.1.2.3 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, we assume a mostly reliable communication environment. 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 message is 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.

5.0 CONCLUSION AND DISCUSSION

In this work, we have presented an overview of the HMP 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, we are using 8-MHz MC68000 processors together with the MC68451 MMU. Each processing node includes 1Mbyte of DRAM, while specialized VLSI components have been designed, and are currently under implementation using Northern Telecom's 5 micron CMOS process, for the interprocessor switch and H-Network controllers. We are also designing and implementing a modular distributed operating system for the described architecture. Distributed application support features that exploit the underlying hardware (i.e., the network and the shared memory) are provided to users in application-specific abstractions.

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 a 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., P.J. Denning, and W.F. Tichy, "Advanced Operating Systems," *IEEE Computer*, vol. 17, no. 10, pp. 173-190, Oct. 1984.
[3] Cheriton, D.R., "The V Kernel : A Software Base for Distributed System," *IEEE Software*, pp. 19-42, Apr. 1984.
[4] Dimopoulos, N.J., and C.W. Wong, "Collision-free Protocol for Local Area Networks," *Computer Communications*, vol.11, no. 4, pp. 208-214, Aug. 1988.
[5] Dimopoulos, N.J., and C.W. Wong, "Performance Evaluation of the H-Network through Simulation," *Digital Techniques in Simulation, Communication and Control*, pp. 315-320, S.G. Tzafestas (Ed.), Elsevier Science Publishers B.V. (North-Holland), 1985.
[6] Dimopoulos, N.J., and D. Kehayas, "The H-Network: A High Speed Distributed Packet Switching Local Computer Network," *Proceedings of MELECON '83 Mediterranean Electrotechnical Conference*, pp. A1.02, Athens, Greece, May 1983.
[7] Dimopoulos, N.J., "On the Structure of the Homogeneous Multiprocessor," *IEEE Transactions on Computers*, vol. C-34, no. 2, pp. 141-150, Feb. 1985.
[8] Dimopoulos, N.J., "Organization and Stability of Neural Network Class and the Structure of a Multiprocessor System," *Ph.D. Dissertation*, University of Maryland, College Park, MD, 1980.
[9] Dimopoulos, N.J., "The Homogeneous Multiprocessor - Architecture, Structure and Performance Analysis," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 520-523, Aug. 1983.
[10] Fekete, G., J.O. Eklundh, and A. Rosenfeld, "Relaxation: Evaluation and Applications," *IEEE Transactions on Patter Analysis and Machine Intelligence*, vol. PAMI-3, pp. 459-469, 1981.
[16] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453-455, Aug. 1974.
[17] Lesser, V.R., et. al., "Organization of the Hearsay II Speech Understanding System," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, pp. 11-24, Feb. 1975.
[18] Li, K.F., and N.J. Dimopoulos, "The Performance Analysis of the Homogeneous Multiprocessor Proper," *Canadian Electrical Engineering Journal*, pp. 3-10, Jan. 1987.
[19] Li, K.F., N.J. Dimopoulos, and J.W. Atwood, "The HM-Nucleus: A Distributed Operating System Nucleus for the Homogeneous Multiprocessor," *IEEE Micro*, pp. 14-24, Feb. 1987.
[21] Metcalfe, R.M., and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol.19, pp. 395-404, Jul. 1976.
[23] Ramanamurthy, R.V., N.J. Dimopoulos, K.F. Li, R.V. Patel, and A.J. Al-Khalili, "Parallel Algorithms for Low Level Vision On the Homogeneous Multiprocessor," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Miami Beach, pp. 421-423, Jun. 22-26, 1986.
[24] Rashid, R.F., and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Operating Systems Review*, vol. 15, no. 5, pp. 64-754, 1981.
[25] Ricart, G., and A.K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9-16, Jan. 1981.
[27] Seitz, C.L., "The Cosmic Cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22-23, Jan. 1985.
[28] Siegel, H.J., et. al., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers*, vol. C-30, pp. 934-937, Dec. 1981.
[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., R. Levin, and S.P. Harbison, *C.mmp - An Experimental Computer System*, McGraw-Hill, New York, 1981.

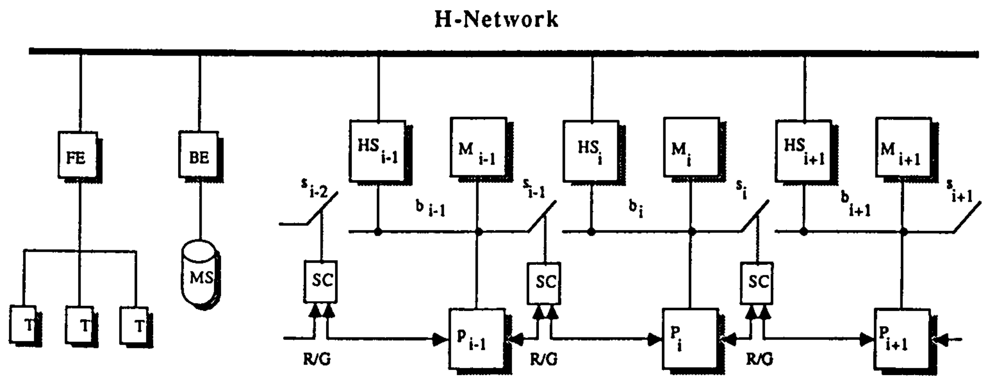

Figure 1. The Homogeneous Multiprocessor Architecture.

P: Processor	M: Memory Module	s: Bus Switch
FE: Front End	BE: Back End	SC: Switch Controller
b: Local Bus	T: Terminal	MS: Mass Storage
HS: H-Network Station	R/G: Bus Request/Grant	