THE MULTIPLUS/MULPLIX PARALLEL PROCESSING ENVIRONMENT

J. S. Aude, A. M. Meslin, C. M. P. Santos, G. Bronstein, L. F. M. Cordeiro,
M. O. Barros, M. João Jr., S. C. Oliveira

NCE/UFRJ, Federal University of Rio de Janeiro, PO Box 2324,
Rio de Janeiro - RJ - 20001-970, BRAZIL, e-mail: salek@nce.ufrj.br

Abstract

The MULTIPLUS project aims at the development of a modular parallel architecture suitable for the study of several aspects of parallelism in both true shared memory and virtual shared memory environments. The MULTIPLUS architecture is able to support up to 1024 Processing Elements based on SPARC microprocessors. The MULPLIX Unix-like operating system offers a suitable parallel programming environment for the MULTIPLUS architecture by providing facilities for the creation of threads, the allocation of private and shared memory space and the efficient use of synchronization primitives. After presenting the main features of the MULTIPLUS architecture and of the MULPLIX operating system, the paper describes in detail the design and the implementation of the three MULTIPLUS architecture basic hardware modules: the Processing Element, the Multistage Interconnection Network and the I/O Processor. In addition, the definition of the MULPLIX parallel programming primitives is discussed and their use is illustrated through an example. Finally, future directions in the development of the MULTIPLUS research project are commented.

1: Introduction

The MULTIPLUS research project includes the development of a modular distributed shared-memory parallel architecture and of its Unix-like operating system called MULPLIX. Its main motivation is to provide a flexible platform to enable the development of research work on different aspects of parallelism in both true shared memory and virtual shared memory environments.

The MULTIPLUS architecture is designed around clusters of up to 8 Processing Elements based on SPARC microprocessors. The communication among the several clusters is performed through a Multistage Interconnection Network. The MULPLIX Unix-like operating system offers a suitable parallel programming environment for the MULTIPLUS architecture by providing facilities for the creation of threads, the allocation of private and shared memory space and the efficient use of synchronization primitives.

Previous papers on the MULTIPLUS project [1, 2] dealt with the main features of the MULTIPLUS architecture and of the MULPLIX operating system. This paper also gives an overview on the MULTIPLUS/ MULPLIX parallel programming environment, but, in addition, provides a detailed insight into the main features of its hardware and software implementation and proposes mechanisms for the implementation of a virtual shared memory environment.

Sections 2 and 3 discuss the main aspects of the MULTIPLUS architecture and of the MULPLIX operating system, respectively. Section 4 presents the architecture and implementation of each Processing Element within MULTIPLUS. In Section 5, the main features of the design of the Multistage Interconnection Network and of its interface to each MULTIPLUS cluster of processors are presented. Section 6 comments on the architecture and implementation of the I/O Processor and its software control system. Section 7 describes and illustrates the use of the parallel programming primitives which have been implemented within MULPLIX Finally, Section 8 comments on the current status of the project and its perspectives for the near future.

2: The MULTIPLUS architecture

MULTIPLUS is a distributed shared-memory high-performance computer designed to have a modular architecture which is able to support up to 1024 Processing Elements (***PE***) and 32 Gbytes of global memory address space.

Figure 1 shows the MULTIPLUS basic architecture. Within MULTIPLUS, up to 8 Processing Elements can be interconnected through a 64-bit double-bus system making up a cluster. Each bus follows a similar protocol to the one defined for the SPARC MBus [3], but is implemented as an asynchronous bus.

1087-4089/96 $5.00 © 1996 IEEE

The MULTIPLUS architecture supports up to 128 clusters interconnected through an inverted n-cube ***Multistage Interconnection Network***. Through the addition of processing elements and clusters, the architecture can cover a broad spectrum of computing power, ranging from workstations to powerful parallel computers. With the adopted structure, the cost and delay introduced by the interconnection network is small or even non-existent in the implementation of parallel computers with up to 64 processing elements. On the other hand, very large parallel computers can be built at a reasonable cost.

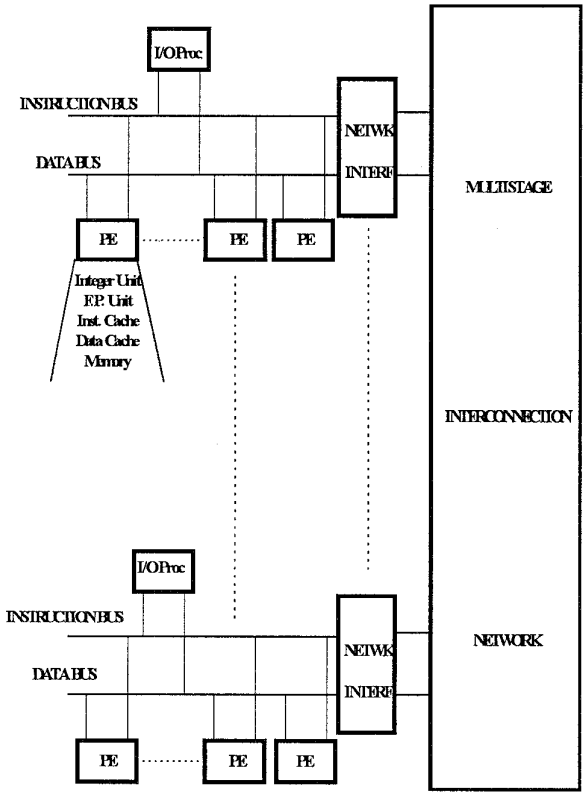

Figure 1: The MULTIPUS Architecture

The MULTIPLUS architecture can be classified as a Non-Uniform Memory Access (NUMA) architecture since a Processing Element access to memory can be performed in four different ways: a direct read operation on the local caches; a read/write operation within the local bank of memory; an access to a memory address belonging to a non-local memory bank within the same cluster; and, finally, an access to a memory bank sitting on another cluster.

As shown in Figure 1, MULTIPLUS uses a distibuted I/O system architecture. It is possible to assign all Processing Elements within a cluster to a single I/O Processor (***I/O Proc***) which is responsible for dealing with all I/O requests started by these Processing Elements.

Two design decisions have been taken to simplify the problem of maintaining consistency among the private caches of the Processing Elements. The first one is to have one cluster bus dedicated to data read/write operations and the other one dedicated to instruction read operations. Under this scheme, only the data bus needs to be "snooped" by the cache controller and, as a result, the cache consistency problem can be solved within a cluster with the methods usually adopted in bus-based systems. The second design decision was to define that read/write data memory pages are only cacheable within the cluster that holds the page. With this approach, cache consistency does not need to be maintained through the multistage interconnection network and the consequent loss in performance can be minimized through careful consideration of data location.

Simulation experiments [4] have shown that the use of the data bus is much more intense than the use of the instruction bus, since the hit rate of instruction caches is significantly higher than that of the data caches. Because of that, the instruction bus is also used for data block transferrences which occur in I/O or in memory page migration or copy operations. The use of the instruction bus for these operations cannot cause any cache consistency problem since the operating system flushes all cache positions occupied by data which might be overwritten by block transferrences.

3: The MULPLIX operating system

MULPLIX is a UNIX-like operating system designed to support medium-grain parallelism and to provide an efficient environment for running parallel applications within MULTIPLUS. In its initial version, MULPLIX will result from extensions to Plurix, an earlier Unix-like operating system developed to support multiprocessing [5]. However, Plurix supports only large-grain parallelism or concurrency and assumes that the underlying machine is implemented by a Uniform Memory Access architecture with only a few processors.

For the MULTIPLUS environment it is essential for the operating system to be very efficient in supporting applications which consist of a large number of processes that may run in parallel, demanding synchronization and, consequently, a lot of context switching operations. One of the basic conditions to reach this goal is to heavily reduce the overhead in such operations.

To solve this problem, one major extension to Plurix included in the MULPLIX definition is the concept of ***thread***. Within MULPLIX, a thread is basically defined by an entry point within the process code. A parallel application consists of a process and its set of threads. Therefore, when switching between threads of a same process, only the current processor context needs to be

saved. Information on memory management and resource allocation is unique for the process as a whole and, therefore, remains unchanged in such context-switching operations.

In relation to synchronization, MULPLIX makes available to the user synchronization primitives for the manipulation of mutual exclusion and partial order semaphores. In addition, MULPLIX implements the busy-waiting primitives avoiding "hot spots" through the interconnection network. The adopted algorithm is an adaptation of the one proposed by Anderson [6]. It is based on the use of a circular buffer to implement the queue of processors waiting for the binary semaphore and on the detection of the availability of the binary semaphore by testing a cacheable local variable.

Within Plurix the memory space allocated to a process consists of a data segment, a code segment and a stack segment for the user and supervisor modes. Memory sharing between processes is not allowed. Within MULPLIX, it is essential for the memory management system to worry about data locality and to allow memory sharing between threads of the same process. The following facilities are supported by the MULPLIX memory management system: replication of the MULPLIX kernel code in every processing node; replication of the process code in every cluster where a given process is running; definition of an additional non-shared local data segment for each thread; definition of an additional local data segment in supervisor mode which is shared by all threads running on the same Processing Element; and definition of stack segments in the user and supervisor modes for each thread.

Process scheduling is another area in which MULPLIX must use a different approach. Within Plurix, there is a single queue of processes which are ready for execution and the scheduling policy does not take into consideration data locality. In addition, time-sharing between processes is always used. Within MULPLIX, a specified number of processors can be set not to run in time-sharing mode in order to run threads of parallel scientific applications more efficiently. Therefore, these threads may run as fast as possible and without interruptions as long as they can or wish. On the other hand, the execution of interactive processes is ensured by the fact that there will always be a fraction of processors running in time-sharing. Data locality is taken into consideration by the MULPLIX scheduling system through the use of separate queues of threads which are ready to be run in each cluster.

4: The MULTIPLUS processing element

Each MULTIPLUS Processing Element consists of:

- an Integer Unit (***IU***) based on a SPARC RISC microprocessor;
- a Floating-Point Unit (***FPU***);
- up to 32 Mbytes of memory belonging to the global address space;
- separate 64 Kbyte instruction and data caches;
- independent instruction and data MMUs;
- I/O devices

As shown in Figure 2, the current implementation of the Processing Element is based on the use of a SPARC chipset supplied by Cypress and Ross Technology. The chipset consists of the following modules: a CY7C601 (***IU***); a CY7C602 (***FPU***); a CY7C604 (memory management unit and cache controller used for instruction accesses, ***ICMU***); and a CY7C605 (memory management unit and cache controller for multiprocessing systems used for data accceses, ***DCMU***).

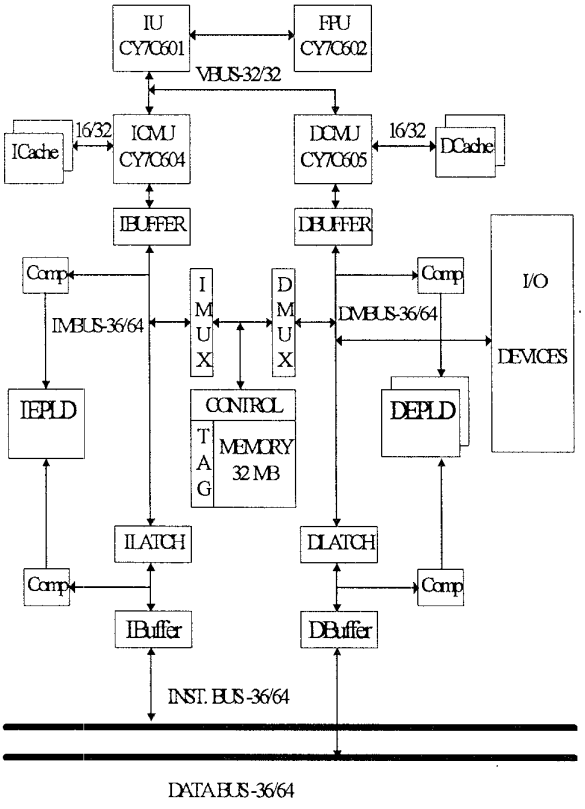

Figure 2: The Processing Element Architecture

Each MULTIPLUS Processing Element has separate 64Kbyte data and instruction caches (***Dcache and Icache***), but only the data cache controller (***DCMU***) needs to snoop the data bus. The data cache controller works in write-through mode with invalidation of shared cache copies, which is a very simple approach and has proved to be as efficient as the write-back mode in simulation experiments carried out considering typical

values for the data cache hit rate and the rate of write operations [4].

As can be seen in Figure 2, the Processing Element internal architecture can be split into two sections: one which deals with instructions and communicates with the ***MULTIPLUS Instruction Bus*** and the other one which deals with data and communicates with the ***MULTIPLUS Data Bus***. Both the instruction and data sections access the same piece of the global memory which sits within the Processing Element. The number of address lines followed by the number of data lines is annotated next to every bus in Figure 2.

The control logic of the Processing Element is implemented with the use of four EPLDs. The first one arbitrates the accesses to the ***IMBUS*** between requests issued by the ***MULTIPLUS Instruction Bus*** and by the Instruction Cache Controller (***ICMU***), performs the master and slave fuctions within the ***MULTIPLUS Instruction Bus*** and arbitrates the use of the common bus for memory access within the Processing Element bewteen requests issued by the ***IMBUS*** and the ***DMBUS***. The second one performs address decoding and control of the access to the Processing Element I/O devices. The third one performs the master and slave functions within the ***MULTIPLUS Data Bus*** and the arbitration of the ***DMBUS*** between requests issued by the ***MULTIPLUS Data Bus*** and by the Data Cache Controller (***DCMU***). The fourth EPLD performs the control of the Dynamic RAM. It decodes the access type, allows page mode access and generates the memory control signals within the specified timing constraints.

Within the memory, a ***TAG*** bit is associated with each memory data block in order to indicate if a copy of this block may exist in another cache. The bit is set whenever the block is read by a different processing element sitting within the same cluster. It is reset whenever that block is rewritten by the local processing element. With the use of the ***TAG*** bit, the need for broadcasting any data access to the ***MULTIPLUS Data Bus*** in order to maintain cache consistency is considerably reduced. Whenever the ***TAG*** bit is not set, the data access can be performed within the Processing Element and without the use of the ***Data Bus***.

5: The multistage interconnection network

The MULTIPLUS Multistage Interconnection Network is an inverted n-cube network consisting of ***2x2 cross-bar switching elements***. Separate networks are used to interconnect the instruction and the data busses in different clusters. The adopted network topology provides the MULTIPLUS architecture with two very desirable features: ***modularity and partitionability***. Modularity enables the MULTIPLUS architecture to grow in numbers of clusters through a simple addition of extra switching elements to the network. Partitioning provides the MULTIPLUS architecture with the possibility of supporting several independent or loosely-coupled groups of clusters such that the communication within a group does not interfere with the communication within any other group of clusters.

The MULTIPLUS Multistage Interconnection Network can support up to 128 clusters. Each communication path between switching elements in the network is unidirectional and nine bits wide. The transmitted messages can have variable length up to a maximum of 128 bytes. ***Wormhole routing*** is used in the network and a single bit of the destination address field of the messages is examined by each stage of switching elements to direct the message to the next stage.

Six message types are supported by the Multistage Inteconnection Network: ***Write, Read, Write Reply, Read Reply, DMA and DMA Reply***. A message can be seen as a sequence of packets consisting of eight data bits and one parity bit. In general, a message has three basic sections: the header, the preamble and the data. The header is four byte long and contains information on the destination cluster, message size, message type and identification of the module that has generated the message within the source cluster. The preamble contains an image of the address lines of the source cluster during the address phase of a bus operation. It is only needed in Read, Write, DMA and DMA Reply messages.

Read and Write messages are generated when a module wants to access a memory position belonging to another cluster. ***The Write Reply message*** informs that the requested write operation has been completed. ***The Read Reply message*** returns the requested data to the corresponding Processing Element. ***A DMA message*** sets the Multistage Inteconnection Network to perform a block transferrence of length up to 64 Kbytes from a region of memory within a given cluster to the local memory of the Processing Element which issued the DMA request. ***The DMA Reply message*** uses the Instruction Bus to transfer the requested data in blocks of 128 bytes between clusters. On completion of the DMA Reply operation, the Network Interface interrupts the Processing Element which issued the DMA request.

The DMA facility within the Network Interface allows an efficient implementation of operations based on memory page copy or migration. Therefore, it provides the basic hardware resource for an efficient implementation of a Virtual Shared Memory model within the MULTIPLUS architecture based on the migration or copy of pages whenever a page fault is detected within a cluster.

The architecture of the switching element of the Interconnection Network implements a 2x2 cross-bar switch with FIFO buffers assigned to each switch input. Each switching element has been implemented within a single EPLD.

The ***Network Interface*** interconnects the cluster bus systems to the Multistage Interconnection Network and

also performs the functions of bus arbiter and bus reset generation. The Network Interface consists of two identical sections: one that deals with the Instruction Bus and another which deals with the Data Bus. In addition, it has a DMA Controller which is programmed through the Data Bus and performs data block transferrences through the Instruction Bus. Within each section, the Network Interface consists of: the bus interface module with a ***Master*** and a ***Slave*** section; the FIFO ***Memory for Messages to be Transmitted***, ***the Message Transmission Control Module***; a dual-port ***Memory for Received Messages*** and the ***Message Reception Control Module.***

The Master section of the Network Interface is activated when remote Read, Write or DMA message arrives at the Interface or when a Write Reply message is received. ***The Slave*** section is activated either when a remote access is generated within the cluster or when a Read Reply message is received. In the first case, the infomation on the requested access is stored in the ***Memory for Messages to be Transmitted*** for later processing. The Read Reply message is a result of a cluster module request for a remote read operation to the Network Interface. As an immediate answer to this remote read request, the ***Slave*** section asks the corresponding cluster module to relinquish the use of the cluster bus and retry the read operation later on. In the meantime, the Network Interface tries to process the read request and to get a Read Reply message as a result. Therefore, when the cluster module retries the read operation, the ***Slave*** section may be already able to send back immediately the requested data to the cluster module. This approach avoids blocking the cluster bus while the Network Interface gets the answer for a remote Read operation. For the implementation of a Virtual Shared Memory mode of operation within the MULTIPLUS architecture, the relinquish and retry signal generated by the Network Interface can be used to indicate the occurrence of a page fault within a cluster.

The ***Message Transmission Control Module*** is responsible for taking messages byte by byte out of the ***Memory for Messages to be Transmitted***, packing them and transmitting them through the Interconnection Network. The ***Message Reception Control Module*** receives the messages coming from the Interconnection Network, stores them in the ***Memory for Received Messages*** and asks the bus interface module to generate the appropriate cluster bus access.

The ***Memory for Messages to be Transmitted*** is organized as a FIFO consisting of a 64-bit wide data section and an 18-bit wide control section. The dual-port ***Memory for Received Messages*** consists of 64-bit words and is divided into three different regions: a FIFO for the received messages; a RAM which stores the replies to messages issued by modules within the local cluster; and an address and access code table for the interruption registers of all the modules within the local cluster. From one port, this memory is accessed for the reception of messages coming from the Network in 9-bit packets. From the other port, this memory is connected to the corresponding 64-bit cluster bus and can be read by the ***Master*** or ***Slave*** section of the Interface and written by the ***Slave*** section or by the DMA

6: The MULTIPLUS I/O processor

The architecture of the MULTIPLUS I/O processor is shown in Figure 3. It consists of two bus systems: the ***CPU BUS*** and the ***DMA BUS***. A ***CPU*** is attached to each bus and they operate concurrently and cooperatively. The one associated with the ***CPU BUS*** is responsible for managing the I/O requests sent by the Processing Elements to the dual-port ***Command Memory***, for performing the ***Disk Cache*** control, for sending commands to be executed by the devices on the ***DMA BUS*** through the ***Communication Memory*** and for controlling a ***Serial Interface*** mainly used for test purposes.

The CPU on the ***DMA BUS*** controls the execution of the internal tasks issued by the ***CPU BUS*** through the ***Communication Memory***. Attached to the ***DMA BUS*** there are: a ***SCSI*** interface; a ***Parallel Interface***; a 32 Mbyte write-through ***Disk Cache***; a ***DMA Controller*** which is responsible for data transferrences from the ***SCSI*** and ***Parallel Interfaces*** to the ***Disk Cache***; and a ***BIFIFO*** which is used as a temporary storage to transmit data between the ***Disk Cache*** and the Processing Elements through the ***MULTIPLUS Instruction Bus.***

Two EPLDs are used to perform some control functions within the I/O Processor. The first one performs the master/slave functions on the ***MULTIPLUS Data Bus***. The second one performs the master/slave functions on the ***MULTIPLUS Instruction Bus*** and controls the burst data transferrences between the ***Disk Cache*** and the ***BIFIFO*** on the ***DMA BUS***.

The operation of the I/O Processor is started when a Processing Element writes an I/O command into its assigned region within the ***Command Memory***. This generates an interruption to the ***CPU BUS processor*** which, then, interprets the command and, if necessary, splits it into sub-tasks that will be performed by the I/O Processor hardware attached to the ***DMA BUS***. For instance, if the command is a disk block read operation, the ***CPU BUS processor*** firstly checks if the block is stored within the ***Disk Cache***. If it is, a command to transfer the block from the cache to the Processing Element memory is issued to the ***DMA BUS*** through the ***Communication Memory***. Otherwise, the command is split into two tasks: the reading of data from the disk to the cache under the supervision of the ***DMA Controller*** and the transferrence of the data from the cache to the Processing Element memory through the ***BIFIFO*** under

the control of the EPLD. Once all steps of the command have been executed by the ***DMA BUS***, the ***CPU BUS*** processor performs a write operation to the interruption register of the corresponding Processing Element through the ***MULTIPLUS Data Bus***.

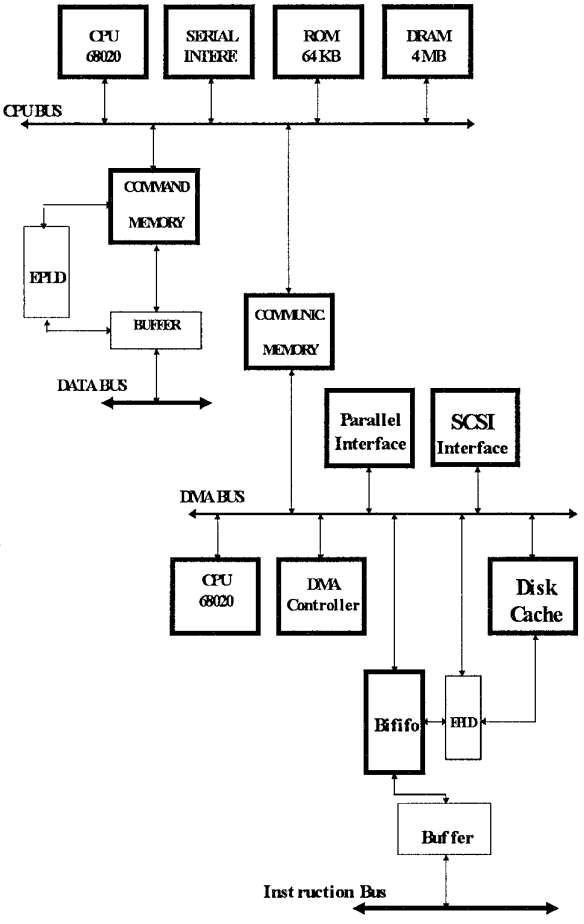

Figure 3: The I/O Processor

7: The MULPLIX parallel programming environment

The MULPLIX parallel programming environment provides a set of system calls for the development of parallel programming applications within the MULTIPLUS architecture. These primitives deal with the following aspects: the creation of threads; memory allocation; and synchronization. They have been created with the MULTIPLUS architecture in mind and are not fully equivalent to the ones defined in Pthreads, the POSIX 1003.4a standard [7]. However, there are a lot of similarities between the two sets and MULPLIX will also provide a Pthreads library interface to the user in the near future.

The system call ***"th_spawn"*** is provided for the parallel creation of a group of threads. The number of threads to be created, the name of the procedure to be executed by these threads and a common argument are the parameters of this system call. It is possible to have synchronous as well as asynchronous creation of threads. If the thread creation is synchronous, the parent thread will suspend its execution until execution completion by all the children threads it has started

The memory allocation primitives can perform shared and private data memory allocation. For shared data, the primitive ***"me_salloc"*** offers two options: a concentrated and a distributed allocation. The first alternative is used when most of the accesses to the data will be made by the thread which has performed the system call and, therefore, all memory space is allocated within the local memory of the thread current Processing Element. The distributed allocation is used when a uniformly distributed access pattern among the threads is expected. The primitive which performs private memory allocation is ***"me_palloc"***.

The MULPLIX operating system offers two explicit synchronization mechanisms. The first one is used for mutual exclusion relations and the second one is employed when a partial ordering relation is to be achieved. For the manipulation of mutual exclusion semaphores, primitives are provided for creating (***"mx_create"***), allocating (***"mx_lock"***), extinguishing (***"mx_delete"***) and releasing (***"mx_free"***) a semaphore.

For partial ordering semaphores, which can implement barrier-type synchronization, primitives for creating (***"ev_create"***), asynchronous signalling (***"ev_signal"***), waiting on the event occurrence (***"ev_wait"***), synchronous signalling (***"ev_swait"***) and extinguishing (***"ev_delete"***) an event are provided.

The following example illustrates the use of some of the MULPLIX primitives in the implementation of a parallel dot product , ***vetc = veta . vetb***. It is assumed that the vectors are of size *n* and that ***P*** threads are used to run the algorithm.

```
#include <threads.h>
#include <stdio.h>

float veta[n], vetb[n], vetc[n];
EVENT product;

main ( )
{
   int i;
   float sum;
   product = ev_create (P, 1);
   th_spawn (P, dot_product, 0);
   ev_wait (product);
```

```
    sum = 0.0;
    for (i = 0; i < P; i++)
        sum = sum + vetc[i];
    printf ("Dot Product: %f\n", sum);
    ev_delete (product);
}

void dot_product (int arg, int p)
{
    int i;
    vetc[p] = 0.0;
    for (i = p*(n/P); i< (p+1)*n/P; i++)
        vetc [p] = vetc[p] + veta[i] * vetb[i];
    ev_signal (product);
}
```

In this example, the system call ***th_spawn*** starts ***P*** threads to run the procedure ***dot_product*** with no common argument. The main thread waits on the event ***product***, which is to be signalled by ***P*** threads and recognized by a single thread (first and second parameters of the ***ev_create*** system call). Each of the ***P*** threads receives from the system information on its order in the group of threads that has been created through the variable ***p***, calculates the dot product associated with the section number ***p*** of length ***n/P*** of ***veta*** and ***vetb***, stores the result in the corresponding ***p*** position of vector ***vetc*** and signals the event ***product***. The main thread restarts on the occurrence of the event ***product*** and sums up all the elements of ***vetc*** to find the final result of the dot product.

8: Current status and perspectives

The implementation and test of an initial MULTIPLUS prototype with 8 Processing Elements and a single I/O Processor organized into up to four clusters is currently in progress. Experimental results on performance evaluation analysis of this prototype will be available soon.

The implementation of the MULPLIX initial version as an evolution of Plurix has been developed by introducing the concept of thread, the new system calls which allow the use of the parallel programming environment, and a library of functions suitable for use in a multithreaded environment [8].

The implementation of a virtual shared memory scheme within the MULTIPLUS architecture based on the migration on demand of memory pages between clusters as a mechanism to hide the Interconnection Network latency is currently under investigation.

In adddition, the implementation of PVM [9] primitives within the MULPLIX environment is under development. Two PVM environments will be made available. The first one will be an implementation of the standard PVM in which tasks will be mapped to processes and the communication between tasks will be implemented through the use of shared-memory. The second one, the MPVM (Multiplus PVM) will be an optimized PVM-like implementation which maps concurrent PVM tasks onto MULPLIX threads. The first environment will favour portability to MULTIPLUS of parallel code written for any PVM platform while the second one will provide the users with an efficient and reasonably familiar parallel programming environment based on the message-passing paradigm.

The initial MULTIPLUS prototype running under the MULPLIX operating system is expected to be available soon. The idea is to encourage other research groups which may benefit from the MULTIPLUS/ MULPLIX parallel environment, to heavily use this prototype. It is through such experience of use that new insights into the problem of parallel processing will be obtained for improving the overall performance of the MULTIPLUS/MULPLIX system.

Acknowledgements

The authors would like to thank Finep, CNPq, RHAE and Capes/Cofecub for the support given to the development of this research work.

References

1. Aude, J.S., et. al. , "Multiplus: A Modular High-Performance Multiprocessor", Proc. of the EUROMICRO 91, Vienna, Austria, pp. 45-52, Sep. 1991
2. Aude, J.S., "Multiplus/Mulplix: An Integrated Environment for the Development of Parallel Applications", Proc. of the IEEE/USP Int'l Workshop on High Performance Computing - WHPC'94, pp. 245-255, São Paulo, Mar 1994
3. Catanzaro, B. "Multiprocessor System Architectures", Sun Microsystems - Prentice-Hall, 1994
4. Meslin, A.M., Pacheco, A.C., Aude, J.S., "A Comparative Analysis of Cache Memory Architectures for the MULTIPLUS Multiprocessor", Proc. of the EUROMICRO 92, Paris, France, pp. 555-562, Sep. 1992
5. Faller, N., Salenbauch, P., "Plurix: A multiprocessing Unix-like operating system", Proc. of the 2nd Workshop on Workstation Operating Systems, IEEE Computer Society Press, Washington, DC, USA, pp. 29-36, Sep. 1989
6. Anderson, T.E., "The performance of spin lock alternatives for shared memory multiprocessors", IEEE Trans. on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6-16, Jan. 1990
7. Walter, S., "Put Multiprocessing Systems to Work, Part II", Unix Review, Jan. 1995, pp. 39 - 47
8. Jones, M.B., "Bringing the C Libraries with us into a Multi-Threaded Future", USENIX, Winter 91, Dallas, Texas, USA, pp. 81-91
9. Geist A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., "PVM 3 User's Guide and Reference Manual", ORNL/TM-12187, May 1993