IMPLEMENTATION OF THE MULTIPLUS/MULPLIX PARALLEL PROCESSING ENVIRONMENT

Júlio S. Aude
Alexandre M. Meslin
Aluísio A. Cruz
Cláudio M. P. Santos
Gerson Bronstein
Iuri N. Cota
Luiz F. M. Cordeiro
Márcio O. Barros
Mário João Jr.
Serafim B. Pinto
Sidney C. Oliveira

NCE/UFRJ
Caixa Postal 2324
Rio de Janeiro - RJ - 20001-970
e-mail: salek@nce.ufrj.br

ABSTRACT

The MULTIPLUS project aims at the development of a modular distributed shared-memory parallel architecture able to support up to 1024 processing elements based on SPARC microprocessors and at the implementation of MULPLIX, a Unix-like operating system which provides a suitable parallel programming environment for the MULTIPLUS architecture. After reviewing the main features of the definition of the MULTIPLUS architecture and the MULPLIX operating system, this paper describes in detail the current implementation of the main modules of the MULTIPLUS architecture and presents, with an illustration example, the parallel programming primitives already implemented within MULPLIX.

1. INTRODUCTION

The MULTIPLUS project [AUDE91, AUDE94] has been under development at NCE/UFRJ for some years now and has provided a nice and challenging framework for research work in several areas related to the world of High-Performance Computing: Parallel Architectures, Operating Systems, IC Design, CAD Tools for IC Design and Parallel Algorithms.

The main objectives of the MULTIPLUS project includes the development of a distributed shared-memory parallel architecture and the MULPLIX operating system. General aspects of the MULTIPLUS architecture have been discussed in previous papers [BRON90, MESL90, OLIV90, MESL92, OLIV92, BRON93] as well as the main features of the MULPLIX operating system [AZEV90, AZEV93]. The focus of this paper is to give some detailed insight into aspects of the implementation of the first prototype of the MULTIPLUS/MULPLIX environment for paralell processing applications.

Section 2 of this paper reviews the main features of the MULTIPLUS architecture and of the MULPLIX operating systems. Section 3 presents the current implementation of each processing element within the MULTIPLUS architecture. In Section 4, the implementation of the multistage interconnection network and of its interface to each MULTIPLUS cluster of processors is presented. Section 5 comments on the implementation of the I/O Processor and its control system. Section 6 describes the parallel programming primitives which have been implemented within MULPLIX and illustrates the use of these primitives in a very simple parallel application. Finally, Section 7 comments on the current status of the project and its perspectives for the near future.

2. THE MULTIPLUS/MULPLIX PARALLEL PROCESSING ENVIRONMENT

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 and 32 Gbytes of global memory address space. The MULPLIX operating system has been designed to adequately support parallel applications within the MULTIPLUS architecture. Section 2.1 describes the main aspects of the MULTIPLUS architecture while Section 2.2 presents the main features of the MULPLIX operating system. Current implementation details of the MULTIPLUS architecture modules and of the MULPLIX parallel programming environment are given in the following sections of the paper.

2. 1 The MULTIPLUS Architecture

Figure 1 shows the MULTIPLUS basic architecture. Within MULTIPLUS, up to eight 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 [CATA94], but is implemented as an asynchronous bus.

The MULTIPLUS architecture supports up to 128 clusters interconnected through an inverted n-cube multistage 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 without the use of an extremely expensive or slow interconnection network.

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. The fastest memory access is a direct read operation on the local caches, which is performed within a processor cycle. The second fastest memory access is any read/write operation within the local bank of memory since, in principle, it does not require the use of the cluster bus system for its completion. The third fastest memory access is a write or a read access with cache failure to a memory position belonging to an external memory bank within the same cluster. In this case, the bus system must be used and the bus arbitration time is added to the access time. Lastly, there are the accesses generated by a processing element requesting information which is not in its local caches but is stored within a memory bank sitting on another cluster. In this case, the bus system of the source cluster , the multistage interconnection network and the bus system of the destination cluster need to be used for the access operation to be performed. Therefore, the arbitration times of both bus systems and the multistage interconnection newtork delay are added to the access time.

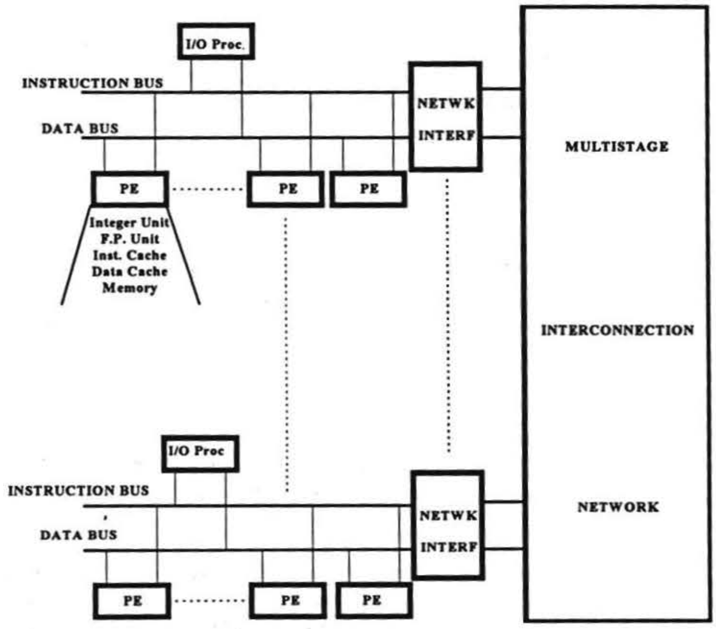

Figure 1: The MULTIPLUS Architecture

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 which is responsible for dealing with all I/O requests to or from mass storage devices 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 within the MULTIPLUS architecture. The first one is to have in every cluster one 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 impose some restrictions on the type of information which is cacheable within MULTIPLUS. Read-only data and instructions are always cacheable. However, data which can be modified is only cacheable within a cluster. 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 [MESL92] have shown that the use of the data bus by the processing elements 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 transfers 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 are to be overwritten by block transfers.

2.2 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 within the Pegasus architecture [FALL89].

Plurix main goal was to provide an efficient environment for running general-purpose processes on an architecture consisting of a few processors and a global memory which can be accessed with the same time penalty by all processors. Therefore, Plurix supports only large-grain parallelism or concurrency and assumes that the underlying machine is implemented by a Uniform Memory Access architecture.

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 in a different way, since it is essential to avoid hot spots through the interconnection network. The algorithm which has been adopted for the solution to this problem is an adaptation of the one proposed by Anderson [ANDE90] and is based on the following ideas [AZEV90]: the use of a circular buffer to implement the queue of processors waiting for the binary semaphore and the detection of the availability of a 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, to support the concept of a process consisting of several threads 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 node; 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 to the one adopted in Plurix. 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 will not run in time-sharing mode. Such processors will be scheduled to run threads of parallel scientific applications. The non-time sharing policy ensures that 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 with 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. Every queue can be accessed by any processor. However, a free processor will only look for a thread to run in another cluster queue if it finds its own cluster queue empty.

3. THE MULTIPLUS PROCESSING ELEMENT

Each MULTIPLUS processing element consists of:

. a RISC microprocessor based on the SPARC architecture definition;
. a floating-point co-processor;
. up to 32 Mbytes of memory belonging to the global address space;
. separate instruction and data caches;
. a serial interface;
. a ROM;
. an identification register;
. interruption registers
. three timers (one for DRAM refresh and two for general use)

The current implementation of the Processing Element is based on the use of a SPARC chipset supplied by Cypress and Ross Technology running at 25 MHz. The chipset consists of the following modules: a CY7C601 (integer unit); a CY7C602 (floating point unit), a CY7C604 (memory management unit and cache controller used for instruction accesses), a CY7C605 (memory management unit and cache controller for multiprocessing systems used for data accceses), four CY7C157 (128 Kbytes of cache RAM: 64Kbytes for the instruction cache and 64 Kbytes for the data cache).

Figure 2 shows a block diagram of the Processing Element architecture which is built around the Cypress chipset. The number of address lines followed by the number of data lines is annotated next to every bus in Figure 2. Each MULTIPLUS processing element has separate data and instruction caches, but only the data cache controller 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 [MESL92].

As can be seen in Figure 2 the processing element 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.

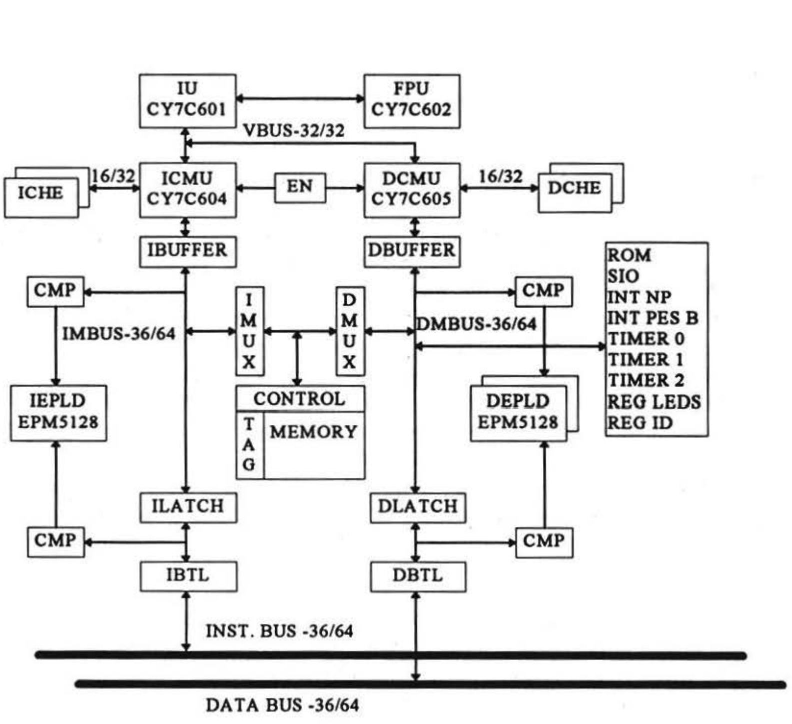

Figure 2: The Processing Element Architecture

The control logic of the Processing Element is implemented with the use of six EPLDs. The first EPLD is used to control the Instrcution Section. It arbitrates the accesses to the IMBUS between requests issued by the MULTIPLUS Instruction Bus and by the Instruction Cache Controller, performs the master and slave fuctions within the MULTIPLUS Instruction Bus and arbitrates the use of the common bus for memory acces within the processing element bewteen requests issued by the IMBUS and the DMBUS. In the control of the data section, two EPLDs are used. The first one performs address decoding and control of the access to the processing element registers and I/O devices. The second 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. The other three EPLDs perform the control of the Dynamic RAM. The first one decodes the access type and allows page mode access. The second one generates the memory control signals within the timing constraints and the third one implements an atomic fetch-and-increment instruction as a modification of the SPARC atomic instructions.

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. The importance of this bit is to reduce the need for broadcasting any data access to the MULTIPLUS Data Bus in order to maintain cache consistency. If the TAG bit is not set, the data access can be performed within the Processing Element and without the use of the Data Bus.

4. 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. The modularity provided by this network enables the MULTIPLUS architecture to grow in numbers of clusters through a simple addition of extra switching elements to the network. No re-wiring of the interconnections between the elements already present in the network is required in such operations. The partitioning feature of the network provides the MULTIPLUS architecture with the possibility of supporting several independent or loosely-coupled groups of clusters. In fact, the network ensures that it is possible to choose 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 newtork 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 types of message are supported by the Multistage Inteconnection Network: Write, Read, Write Reply, Read Reply, DMA and DMA Reply. Every message can have only a single source and single destination, therefore broadcast or multicast type messages are not handled by the network.

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 address, 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 64-bit address lines of the source cluster. It is only needed in Read, Write, DMA and DMA Reply messages.

Read and Write messages occur when a module within a cluster wants to access a memory position belonging to another cluster. The Write Reply message can be used to tell the module that has generated the write operation that the requested operation has been completed. The Read Reply message returns the requested data to the processing element which had issued the corresponding Read message. 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 architecture of the switching element of the Interconnection Network implements a 2x2 cross-bar switch with FIFO buffers assigned to each switch output. Its detailed design has been presented by Bronstein [BRON90]. Each switching element has been implemented with a single EPLD, which performs the function of the 2x2 switch, and two 2Kx8 FIFOs.

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 Newtork 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 transfers through the Instruction Bus. Within each section, the Network Interface consists of 8 modules: the bus interface module with a master and a slave section, the FIFO memory for messages to be transmitted, the message transmission module, a dual-port memory for received messages, the message reception module, registers, the bus arbiter and the logic for bus reset generation.

The implementation of the Network Interface has been carried out with 11 EPLDs, five for each section and one for the DMA Controller. The five EPLDS in each section perform the following functions: master of the bus; slave of the bus; message transmission control; message reception control; store the status of the messages sent by the interface and generate the address of the memory for received messages.

The Master section of the Network Interface is activated when some 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 occurs because at some point a cluster module requested a remote read operation to the Network Interface. As an immediate answer to this read request, the Slave section sent an instruction for the cluster module to relinquish the use of the cluster bus and retry the read operation later on. Hopefully, in the meantime, the Newtork Interface has enough time to process the read request and get a Read Reply message as a result. Therefore, when the cluster module retries the read operation, the Slave section is able to send back the requested data to the cluster module. This appraoach avoids blocking the cluster bus while the Network Interface gets the answer for a remote read operation.

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 instructs the bus interface module to generate the appropriate cluster bus access.

In addition to the EPLDs, a FIFO memory has been used to implement the memories for the messages to be transmitted. This FIFO memory consists of two sections: a 64-bit wide data section and an 18-bit wide control section. The dual-port memories for message reception consist of 64-bit words and are divided into three different regions. The first one works as a FIFO for the received messages. The second one works as a RAM which stores the replies to messages sent by modules within the local cluster and the third one stores 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 8-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

5. 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. Attached to each bus there is a 68020 CPU. The one associated with the CPU BUS is responsible for managing the I/O requests sent by the processing elements to the 16 Kbyte 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 4 Kbyte Communication Memory and for controlling a serial interface. It uses a 4 Mbyte RAM for its work area and a 64 Kbyte ROM to store the initialization procedure.

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 for the connection of disks, tapes and floppies; a parallel interface for the connection of printers; a 32 Mbyte write-through Disk Cache; a DMA Controller which is responsible for the data transfer from the SCSI and Parallel Interfaces to the Disk Cache; and an 8 Kbyte 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 transfers 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 68020 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 68020 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. Again, both tasks are issued to the DMA BUS through the Communication Memory. Once all steps of a processing element command have been executed by the DMA BUS, the CPU BUS does a write operation to the interruption register of the processing element through the MULTIPLUS Data Bus.

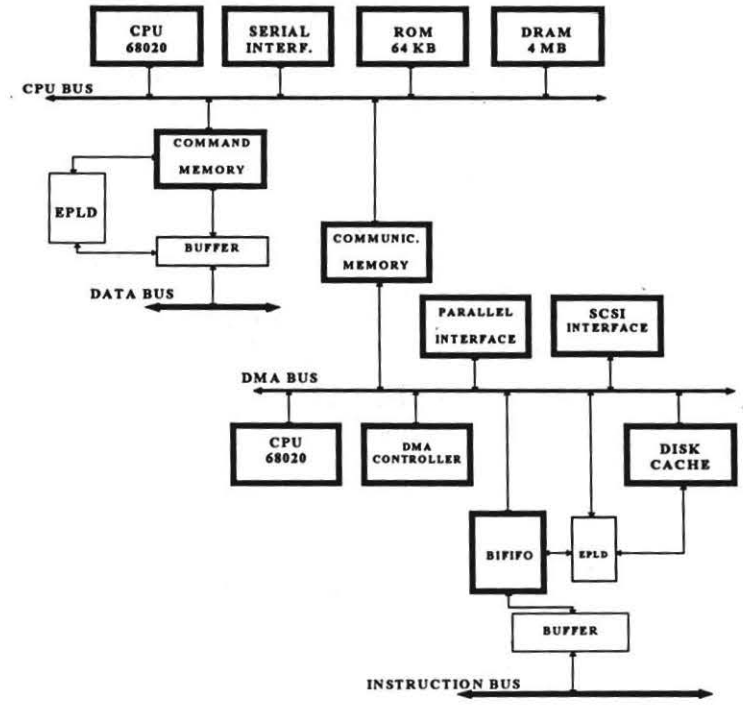

Figure 3: The I/O Processor Architecture

6. THE MULPLIX PARALLEL PROGRAMMING ENVIRONMENT

The MULPLIX parallel programming environment [AZEV93] 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. The current implementation of MULPLIX is running and under development on an EBC 32020, a 68020 based machine to which the Plurix operating system had been previously ported. Due to the limitations imposed by this environment, the implementation of some of the primitives has not been performed yet fully in accordance to the original specification.

The system call, "th_spawn", is provided for the 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 basic parameters of this system call and the ones which are supported by the MULPLIX current implementation. However, two new parameters will be added to this system call. The first one, an optional parameter, is a vector which defines preferential processing elements for the execution of each thread to be created. This facility will allow an experienced user to enforce the assignment of a particular thread to the processing element which is known to host the set of data to be mostly used by that thread. The second extension to this system call will allow 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 allocation. For shared data, the primitive "me_salloc" offers two options: a concentrated and a distributed memory space allocation. In the first case, it is expected that most of the accesses to the memory space to be allocated will be performed by the thread which has performed the system call and, therefore, all memory space is allocated within the local memory of the thread preferential processing element. The distributed allocation is used when a uniformly distributed access pattern among the threads is expected. Within the EBC 32020, there is only a single processing element and the concentrated/distributed option is meaningless. Therefore it has not been implemented yet. 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. Simple and multiple mutual exclusion synchronizations are supported. With multiple mutual exclusion, a maximum of a given number of threads can execute the critical region simultaneously.

For partial ordering semaphores, which 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 these primitives in the implementation of a parallel dot product , vetc = veta . vetb, assuming that the vectors are of size "n" and that P processing elements are available to run the algorithm.

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

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

main ( )
{
   int i;
   float soma;

   produto = ev_create (P, 1);
   th_spawn (P, prod_escalar, 0);
   ev_wait (produto);
   soma = 0.0;
   for (i = 0; i < P; i++)
        soma = soma + vetc[i];
   printf ("Produto escalar: %f\n", soma);
   ev_delete (produto);
}

prod_escalar (arg, p)

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 (produto);
}
```

In this example, the system call "th_spawn" issued by the main thread starts *P* threads to run the procedure *prod_escalar* with no common argument. The main thread waits on the event *produto*, which has been defined as an event to be signalled by P threads (first parameter of the *ev_create* system call) and to be recognized by a single thread (second parameter 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 *produto*. The main thread restarts on the ocurrence of the event *produto* and sums up all the elements of *vetc* to find the final result of the dot product.

7. CURRENT STATUS AND PERSPECTIVES

The MULTIPLUS architecture definition and detailed logic design have been completed. Currently, we are working in the implementation and test of an initial prototype with 8 Processing Elements and a single I/O Processor organized into up to four clusters.

In parallel, we are undertaking a new design of the Processing Element based on the use of a the Texas SuperSPARC chip, which implements a superscalar architecture, and a new design of the Interconnection Network and Network Interface, which will use faster and denser EPLDs and a custom CMOS chip to implement the bus arbiters. As a research investigation we are also looking into the problem of implementing a virtual shared memory scheme within the MULTIPLUS architecture.

The implementation of the MULPLIX initial version as an evolution of Plurix is under development on EBC 32020 computers. Up to the moment the implementation of an initial version of the new system calls has been performed. Currently we are working in the development of a library of functions suitable for use in a multithreaded enviornment and in the migration of the system kernel to Sparcstations with re-writing of the assembly code of the kernel and the development of a memory management module which conforms to the SPARC MMU Reference [CATA94]. In adddition, the implementation of PVM primitives within the MULPLIX environment is being performed. The goal here is to simplify the portability to MULTIPLUS of parallel code written for different PVM platforms and also to create an opportunity to have an implementation of a High Performance Fortran compiler under design at CTA running within the MULTIPLUS/MULPLIX platform.

It is expected to have an initial MULTIPLUS prototype running under the MULPLIX initial version by the end of 1995. The idea is to make this proptotype available for use by other research groups, in particular those at the Federal University of Rio de Janeiro, which are currently involved with work in several areas that may benefit from the MULTIPLUS computing power and parallel environment. It is through such experience of use that we hope to have new insights into the problem of parallel processing and, therefore, be able to improve the performance of the MULTIPLUS/MULPLIX system.

ACKNOWLEDGEMENTS

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

REFERENCES

[ANDE90] Anderson, T.E., "The performance of spin lock alternatives for shared memory multiprocessors", IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6-16, January 1990

[AUDE91] Aude, J.S., et. al. , "Multiplus: A Modular High-Performance Multiprocessor", Proc. of the EUROMICRO 91, Vienna, Austria, pp. 45-52, Sep. 1991

[AUDE94] Aude, J.S., "Multiplus/Mulplix: An Integrated Environment for the Development of Parallel Applications", Proc. of the IEEE/USP International Workshop on High Performance Computing - WHPC'94, pp. 245-255, São Paulo, March 1994

[AZEV90] Azevedo, G.P., Azevedo R.P., Figueira, N.R., Aude, J.S., "MULPLIX: Um Sistema Operacional tipo UNIX para o Multiprocessador MULTIPLUS", Proceedings of the III Brazilian Symposium on Computer Architecture - Parallel Processing, Rio de Janeiro, RJ, pp. 122-137, November 1990

[AZEV93] Azevedo, R.P., Azevedo, G.P., Silveira, J.T.C, Aude, J.S., "Primitivas para Programação Paralela no MULTIPLUS", Proceedings of the V Brazilian Symposium on Computer Architecture, Florianópolis, pp. 761-775, Sepetember 1993

[BRON90] Bronstein, G., Cruz, A.J.O, Duarte, O.C.M.B., "Análise de Desempenho de Redes de Interconexão para Máquinas Paralelas", Proc. of the III Brazilian Symposium on Computer Architecture - Parallel Processing, Rio de Janeiro, pp. 345-360, Nov. 1990

[BRON93] Bronstein, G., "O Subsistema de Interconexão do Multiprocessador MULTIPLUS", Proceedings of the V Brazilian Symposium on Computer Architecture, Florianópolis, pp. 166-173, Sepetember 1993

[CATA94] Catanzaro, B. "Multiprocessor System Architectures", Sun Microsystems - Prentice-Hall, 1994

[FALL89] Faller, N., Salenbauch, P., "Plurix: A multiprocessing Unix-like operating system", Proceedings of the 2nd Workshop on Workstation Operating Systems, IEEE Computer Society Press, Washington, DC, USA, pp. 29-36, September 1989

[MESL90] Meslin, A.M, Pacheco, A.C., "Sistemas de Memórias Multicache para uma Máquina Paralela MIMD: Projeto MULTIPLUS", Proc. of the III Brazilian Symposium on Computer Architecture - Parallel Processing, Rio de Janeiro, pp. 179-193, Nov. 1990

[MESL92] Meslin, A.M., Pacheco, A.C., Aude, J.S., "A Comparative Analysis of Cache Memory Architectures for the MULTIPLUS Multiprocessor", Proceedings of the EUROMICRO 92, Paris, France, pp. 555-562, September 1992

[OLIV90] Oliveira, S.C., Aude, J.S., "O Subsistema de Memória de Massa do Multiprocessador MULTIPLUS", Proceedings of the III Brazilian Symposium on Computer Architecture - Parallel Processing, Rio de Janeiro, RJ, pp. 298-313, Nov 1990

[OLIV92] Oliveira, S.C, Aude, J.S., "Uma Avaliação do Impacto das Operações de E/S no Desempenho do Multiprocessador MULTIPLUS", Proccedings of the IV Brazilian Symposium on Computer Architecture - São Paulo, SP, pp. 379-394 , October 1992