Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences, 1984.

A DISTRIBUTED OPERATING SYSTEM FOR PASM

David Lee Tuomenoksa†
Howard Jay Siegel

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907, USA

Abstract

The structure of PASMOS, a distributed operating system for the PASM parallel processing system, is overviewed. PASM is a large-scale dynamically reconfigurable multimicrocomputer system which is being designed at Purdue University for image processing and pattern recognition applications. This special purpose nature of PASM has been exploited in the design of PASMOS. PASMOS has a hierarchical structure and is distributed throughout the hardware components of PASM. PASMOS creates an execution environment for parallel processing tasks. It is responsible for task management and scheduling and provides facilities for user interaction, memory management, file system control, hardware fault detection, protection, and process control. The software components of PASMOS and the way in which they are distributed throughout the PASM system are described.

1. INTRODUCTION

There are several types of parallel processing systems. A *single instruction stream - multiple data stream (SIMD) machine* [10] typically consists of a control unit, a set of N processors, N memories, and an interconnection network (e.g., Illiac IV [5], Staran [3,4]). The control unit broadcasts instructions to all processors and each active processor executes each of the broadcasted instructions on the data in its own memory. Each instruction is executed simultaneously in all active processors. The interconnection network provides a means for interprocessor communication. A *multiple instruction stream - multiple data stream (MIMD) machine* [10] typically consists of N processors, N memories, and an interconnection network, where each processor can follow an independent instruction stream (e.g., C.mmp [33], Cm* [27]). An *SIMD/MIMD machine*, which also typically consists of a control unit, a set of N processors, N memories, and an interconnection network, can operate as either an SIMD or MIMD machine and can dynamically switch between the SIMD and MIMD modes of operation (e.g., CAIP [11]).

A *multiple-SIMD system* is a parallel processing system which can be structured as one or more independent SIMD machines of varying sizes (e.g., MAP [17]). Illiac IV was originally designed to be a multiple-SIMD system [2]. The advantages of a system capable of operating in multiple-SIMD mode include: it allows multiple users to be executing tasks simultaneously, it is more fault tolerant, it permits the size of a virtual machine to be adjusted to the needs of a task, and it can overlap the execution of different SIMD subtasks of an overall task [26]. A *partitionable SIMD/MIMD system* can be dynamically reconfigured to operate as one or more independent SIMD/MIMD machines of varying sizes (e.g., PASM [26], TRAC [24]).

This research was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581, and by a Purdue University Graduate Fellowship. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

† D. L. Tuomenoksa is now with AT&T Information Systems Laboratories, Holmdel, New Jersey 07733.

Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences, 1984.

The PASM parallel processing system is a partitionable SIMD/MIMD system which is being designed at Purdue University for image processing and pattern recognition applications. The motivation for PASM, its architecture, and examples of its use are given in [26]. Simulation studies of PASM are currently being conducted [13]; a prototype has been designed and is planned for construction in the near future. PASMOS is a distributed operating system which is being developed for PASM. The special purpose nature of PASM has been exploited in the design of PASMOS. PASMOS has a hierarchical structure and is distributed throughout the hardware components of PASM. PASMOS utilizes the PASM hardware to create execution environments (virtual machines) for tasks which operate in SIMD and/or MIMD modes.

The PASM System Control Unit is a conventional machine and is responsible for overall coordination of the activities of the other components of PASM. The System Control Unit executes the UNIX† Time-Sharing System [23]. The PASMOS kernel is executed as a process under UNIX and is responsible for task management and scheduling [28,30]. Users interact with PASM through the System Control Unit using the UNIX shell [6] and the PASM Command Language. PASMOS also provides facilities for memory management, file system control, hardware fault detection, protection, and MIMD process control.

Section 2 is an overview of PASM. The organization of PASMOS is presented in Section 3. The definition of a PASMOS task is given in Section 4. Section 5 is a description of the PASMOS kernel. The software associated with the multiple control units and the processing elements is described in Sections 6 and 7, respectively. The PASM Command Language is presented in Section 8. A scenario of an SIMD task is given in Section 9. Protection mechanisms supplied by PASMOS are presented in Section 10.

2. PASM OVERVIEW

PASM, a partitionable SIMD/MIMD machine, is a large-scale dynamically reconfigurable parallel processing system [26]. A block diagram of the basic components of PASM is given in Fig. 1. The *System Control Unit* is a conventional machine, such as a PDP-11/70, and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit (PCU)* contains $N = 2^n$ processors, N memory modules, and an interconnection network. The *PCU processors* are microprocessors that perform the actual SIMD and MIMD computations. The *PCU memory modules* are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode.

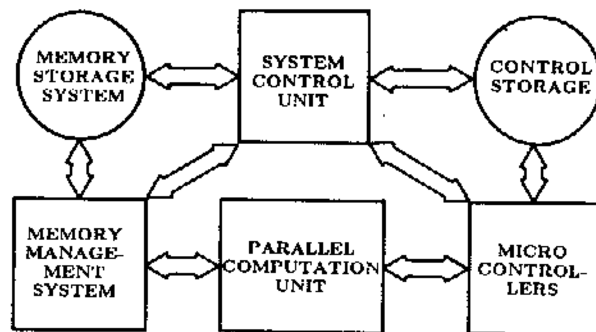

Fig. 1. Block diagram overview of PASM.

The PCU is organized as shown in Fig. 2. A pair of memory units is used for each PCU memory module so that data can be moved between one memory unit and secondary storage while the PCU processor operates on data in the other memory unit. A processor and its associated memory module form a *PCU processing element (PE)*. The PEs are physically addressed (numbered) from 0 to N−1. The *interconnection network* provides a means of communication among the PEs. PASM will use either an Extra Stage Cube [1] or Augmented Data Manipulator [16] type of multistage network.

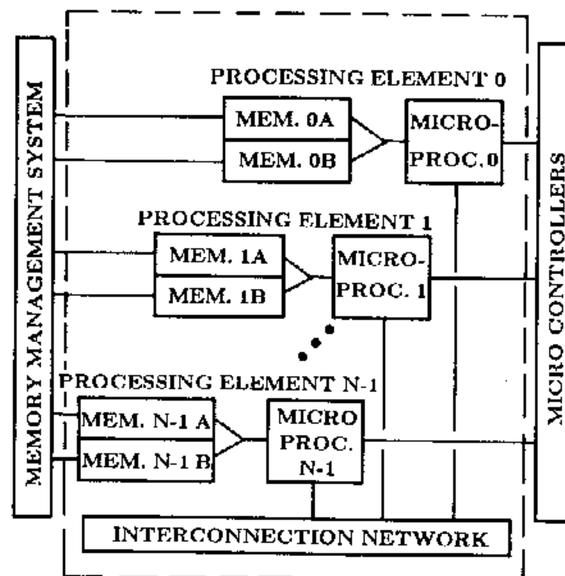

Fig. 2. PASM Parallel Computation Unit.

The *Micro Controllers (MCs)* are a set of microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. The organization of the MCs is given in Fig. 3. There are $Q = 2^q$ MCs, addressed (numbered) from 0 to Q−1. Like the PEs, the MC memory modules are double-buffered. Each MC controls N/Q PEs, where possible values of N and Q are 1024 and 16, respectively. A PASM *MC-group* is composed of an MC processor, its memory module, and the N/Q PEs which are controlled by the MC. The N/Q PEs connected to MC i are those whose addresses have the value i in their low-order q bit

† UNIX is a Trademark of AT&T Bell Laboratories.

positions. *Control Storage* contains the programs for the MCs and is controlled by the *Control Storage Controller*.

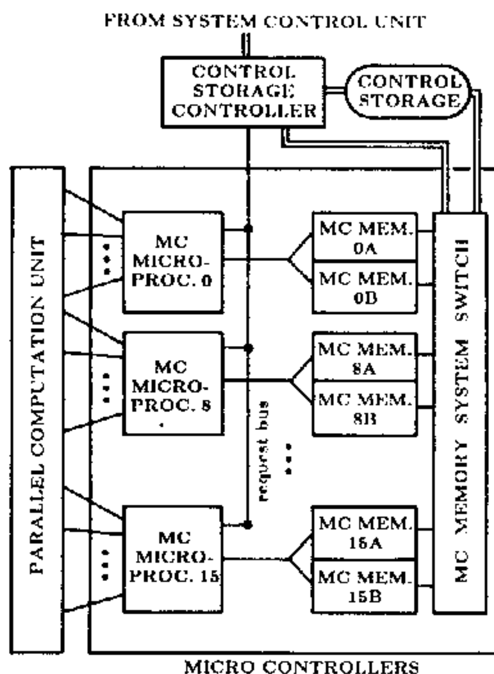

Fig. 3. Organization of the PASM Micro Controllers (MCs) for Q=16.

A virtual SIMD/MIMD machine (partition) of size RN/Q, where $R = 2^r$ and $0 \leq r \leq q$, is obtained by combining the efforts of R MCs and their associated PEs. According to the partitioning rule for PASM [26], the physical addresses of these MCs must have the same low-order q−r bits so that all of the PEs in the partition have the same low-order q−r physical address bits. For example, for Q = 16, an allowable MC partition groups together the following MCs: (6), (14), (2,10), (0,4,8,12), and (1,3,5,7,9,11,13,15). Q is therefore the maximum number of partitions allowable, and N/Q is the size of the smallest partition. The reason for using this particular partitioning rule is because it allows multistage networks like the Extra Stage Cube and the Augmented Data Manipulator, which are being considered for PASM, to be partitioned into independent subnetworks [25]. This rule is also valid for multistage Omega [15], shuffle-exchange [20], and indirect binary n-cube [22] networks, as well as other data manipulator [9] type networks such as the Gamma [21] network [25].

The *designator* of a virtual machine composed of an allowable MC partition is the smallest physical address of the MCs in the virtual machine. This designator corresponds to the low-order q−r bits of the physical address of each MC in the virtual machine. For the MC partitions in the above example, the designators are: 6, 14, 2, 0, and 1, respectively.

The approach of permanently assigning a fixed number of PCU processors to each MC has the advantages that the operating system need only schedule Q MCs, rather than N PCU processors, and that it simplifies the MC/PE interaction, from both a hardware and software point of view, when a virtual machine is being formed. In addition, this fixed assignment scheme supports partitioning of the interconnection network and is exploited in the design of the Memory Storage System in order to allow the effective use of parallel secondary storage devices.

The *Memory Storage System*, provides secondary storage space for the PCU memory modules. It consists of N/Q independent *Memory Storage Units (MSUs)* and is controlled by the *Memory Management System* [14]. The MSUs are numbered from 0 to (N/Q)−1. Each is connected to Q PCU memory modules, as shown in Fig. 4. For $0 \leq i < N/Q$, MSU i is connected to the ith logical PE of each MC-group. The two main advantages of this approach for a partition of size N/Q (i.e., one MC-group) are that (1) all of the PCU memory modules can be loaded in one parallel block transfer and (2) the input data and programs are directly available no matter which partition (MC-group) is chosen. This is done by storing the input data and program for a task which is to be loaded into the ith logical PE of the virtual machine in MSU i, $0 \leq i < N/Q$. Thus, no matter which MC-group is chosen, the data and program from the ith MSU can be loaded into the ith PCU memory module of the virtual machine, for all i, $0 \leq i < N/Q$, simultaneously.

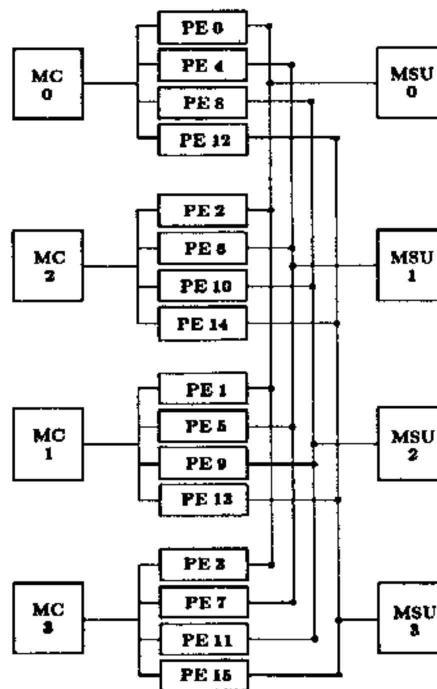

Fig. 4. Organization of the Memory Storage System for N = 16 and Q = 4.

For virtual machines of size N/Q, this secondary storage scheme allows all N/Q PCU memory modules to be loaded in one parallel block transfer. Consider the situation where a virtual machine of size RN/Q is desired, $1 < R \leq Q$. Only R parallel block loads are required if the data for the PCU memory module whose high-order n−q logical address bits equal i is loaded into MSU i. This is true no matter which partition of R MCs (which agree in the low-order q−r address bits) is chosen [26].

3. PASMOS OVERVIEW

The PASMOS operating system provides the control for the PASM parallel processing system. The code for PASMOS is distributed among the System Control Unit, the Q MCs, the N PEs, the Memory Management System, and the Control Storage Controller. In this paper, the PASMOS facilities associated with the System Control Unit, the MCs, and the PEs are considered. The System Control Unit runs the UNIX Time-Sharing System [23]. Users access the system through the System Control Unit. The facilities for user interaction are provided by the UNIX shell [6]. The PASM Command Language is a set of commands which supplement the commands supplied by the UNIX shell and allow the user to execute tasks on PASM. PASMOS is a multiuser operating system since more than one user can be logged into the System Control Unit. It is also multiprogrammed since more than one task can be executed on the system simultaneously. Individual MCs and PEs in PASM are not multiprogrammed.

The PASMOS kernel is executed as a UNIX process on the System Control Unit. The PASMOS kernel runs with highest priority so that the System Control Unit will not become a bottleneck for PASM. Each MC executes an MC kernel which orchestrates the activities of the PEs in MIMD mode. Each of the PEs has a monitor routine which it uses for MIMD process execution.

In [18] three operating system strategies for a hypothetical multiple-SIMD computer system [17] are compared. With the first strategy, one control unit executes the operating system while the remaining control units operate as conventional SIMD machines sharing the processing elements. The second strategy uses a dedicated control unit for the operating system, but the remaining control units are multiprogrammed. The third strategy is for all control units to be multiprogrammed, with the operating system processes being executed on any control unit that requires system services, i.e., a distributed operating system. PASMOS fits into the first strategy, except that the System Control Unit executes the kernel of the operating system rather than a dedicated MC. The results of Nutt's study indicate that for an environment that does not make excessive demands on the machine, the first strategy is an attractive solution. As the job load increases to a point of saturation, the third strategy appears to have a performance edge over the second that makes its added complexity worthwhile. He also concludes that the performance of the first strategy would be better than the model if input/output buffering were incorporated to allow input/output-compute overlap. In [32] it is shown that input/output-compute overlap significantly improves the performance of the PASM system.

The structure of the operating systems for Illiac IV [2,5], PEPE [7,8], and STARAN [3,4] are all similar to PASMOS in that the principle part of the operating system is executed by a host machine or system control unit. The original Illiac IV system [2] is most like PASM in that there are three levels in the system: the Processing Elements, Control Units, and the Host which correspond to the PEs, MCs, and the System Control Unit in PASM, respectively.

Since PASM is being designed for use in a research environment, the first layer of system security will be provided by limited access to the machine. Since the users interact with PASM through the System Control Unit which is running the UNIX Time-Sharing System, system security will also depend on those mechanisms provided by UNIX. Additional protection mechanisms provided by PASMOS are presented in Section 10.

If an MC or PE processor has a hardware failure which prevents it from executing its part of a task, the system is unable to use the entire MC-group which contains the faulty processor. With the loss of the MC-group, the system is still able to execute tasks of all sizes except for those tasks which require the entire system (i.e., Q MC-groups). The task which was executing on the MC-group when the hardware failure occurred will need to be rescheduled (i.e., assigned to a virtual machine which does not contain the faulty MC-group) and restarted my the task manager.

4. THE PASMOS TASK

A *job* which is to be executed by PASM can use either SIMD and/or MIMD mode. A job which strictly uses SIMD mode (e.g., image smoothing [26]) consists of an *MC-program*, and a job which uses both SIMD and MIMD modes (e.g., image contour extraction [31]) consists of an MC-program and one or more *PE-programs.* The MC- and PE-programs are compiled on the System Control Unit. The object code from the same MC-program is executed by each MC which is assigned to the job [26]. An MC-program is composed of SIMD instruction/procedures, synchronization instructions, and mode switching instructions. For MIMD operation, the same object code can be used by all of the PEs in the virtual machine or each PE can use its own PE-program. A PE-program is composed of one or more procedures.

A *task* is defined to be the execution of a job on a virtual machine. A task is created whenever a user executes an MC-program. Tasks which are executed by PASM can operate in SIMD and/or MIMD modes. A task initially operates in SIMD mode. When a task is in SIMD mode, the MCs, which are assigned to the task, fetch instructions from their memory units, execute control flow instructions (e.g., branches), and broadcast the data processing instructions to the PEs. The PEs fetch the instructions from their instruction buffer. To enter MIMD mode, the MCs which are executing the task broadcast a jump instruction to their respective PEs causing the PEs to jump to one of the procedures in their PE-programs. When a virtual machine is in MIMD mode, the PEs fetch their own instructions from their memory units and execute them, while the MCs coordinate the activities of the PEs. When a task is operating in MIMD mode, the PEs can all be executing the same procedures (asynchronously) or can all be executing different procedures. Recall that different virtual machines can be operating in both SIMD and MIMD modes simultaneously in the PASM system. A *process* is defined to be the execution of an MIMD procedure on a PE. A task operating in MIMD mode which is executed by RN/Q PEs, where $R = 2^r$ and $0 \leq r \leq q$, will have from one to RN/Q processes associated with it at any given time (i.e., at most one process per PE).

5. THE PASMOS KERNEL

The kernel of the PASM operating system is executed by the System Control Unit. It is responsible for task management and scheduling. Task management is concerned with the sharing of the MC processors, the PCU processors, and their associated memory modules amongst the various tasks. The *task table* resides in the primary memory of the System Control Unit and contains an entry for each active task. A task is active from the time the programmer requests execution of a task until the time when the task completes execution and all result data files have been unloaded from the PCU memory units. Hence, when a task is created an entry is put into the task table. The task table entry contains the current status of the task.

When a user requests the execution of a program, the task manager initializes the task table entry. It then assigns the task to the appropriate task queue for scheduling. The first-fit multiple-queue (FFMQ) task scheduling algorithm which is being considered for PASM makes use of $q + 1$ FIFO task queues, one for each possible task size [30]. After the task manager has assigned the task to the task queue it executes the FFMQ scheduler to see if the task can be scheduled. The task scheduler is executed by the task manager whenever a new task is added to one of the task queues or whenever a task completes execution.

After the task scheduler has been executed and all possible tasks have been assigned to MCs, the task manager calls the task preloader. The task preloader uses the scheduling algorithm to predict which task or tasks may be executed next by the MC-groups which are currently forming a given virtual machine [32]. The task manager then requests that the Memory Storage System preload the PCU input data file (and the PCU programs for tasks which use MIMD mode) for these tasks. An MC-group's PEs can have preloaded input data/program files for at most one task at any time.

When a task is scheduled for execution, the task manager updates the task table. If the input data files (and program files for tasks which use MIMD mode) for the task have not been preloaded, the task manager requests that the Memory Storage System load them. The task manager also requests that the MC-program for the task be loaded from the Control Storage to the MC memory units. At this point the responsibility for the task is passed down to the MC processors.

When an MC-group completes execution of a task it signals the System Control Unit. After the System Control Unit has received completion signals from all of the MC-groups which are executing a given task, the task manager requests that the Memory Storage System unload the output data from the PCU memory units, it frees the MCs, and it executes the FFMQ scheduler. When the System Control Unit receives the signal from the Memory Management System indicating that the output data has been unloaded, it deletes the task table entry and informs the user that the task has been completed.

6. MICRO CONTROLLER KERNEL

The MC kernel is the code for PASMOS which resides on each of the MCs. The MC kernel code is the same on all MCs, i.e., all MCs have the same capabilities. This is unlike the Medusa operating system [19] for Cm* which is divided into utilities that are distributed around the system. A Cm* processor executes a particular operating system utility only if it can do so locally [19]. The MCs can operate in either SIMD, MIMD, or diagnostic modes.

Two storage allocation policies have been considered for the MCs: 1) static allocation where the entire program must be loaded into the MC memory units before execution begins and 2) virtual memory (dynamic allocation) where the program is paged into the MC memory units [29]. Due to the decreasing cost of primary memory and the intended application of PASM, it has been determined that static allocation is the better policy for solving the allocation problem for the MCs on the PASM prototype. Specifically, since the MCs are not multiprogrammed, they have only one program in their memory at any given time. Since the actual computations on data are done by the PCU, the MC memory units will not have to store data files. Hence, each MC memory unit will contain at most one program file and no data files.

With the decreasing cost of hardware, it will be possible to put 256 kilowords of random access memory in each MC memory unit. It is unlikely that any MC control program will be greater than 256 kilowords, and if a program did exceed this length, it could be overlaid. The use of static allocation also eliminates the extra overhead resulting from the use of paging. Another advantage of virtual memory is that the entire program does not need to be loaded into primary memory. However, for PASM loading the entire program is not a disadvantage since the loading of the program can be overlapped with the loading of the data which must occur anyway [32].

Each task which is to be executed by PASM initially uses SIMD mode. The System Control Unit signals the MC processors when it assigns a task to them. After the MC processor determines that the correct input data files and PE-program files have been loaded into the PCU memory units and that the MC-program file has been loaded into the MC memory unit, execution begins. If the correct files were not loaded into one or more of the PCU memory units, an error condition occurs. When the error occurs, the MC informs the System Control Unit of the error, and the System Control Unit aborts the task and informs the user.

In order to synchronize the MC processors which control a task which requires more than one MC-group, the PASM system supplies software break points. Before an MC can go past one of these break points, all MCs which are executing the task must reach the break point. The MCs execute one of these break points before they start execution of the actual task. This causes all MCs to begin execution together. After the MC executes the last instruction in the MC-program, it signals the System Control Unit that the task has been completed and waits for another task to be assigned to it. A description of the operation of an MC-group in SIMD mode is given in [13].

When the MC-group is operating in MIMD mode, the MC manages PE-to-PE communication. PE-to-PE com-

munication is discussed in the next section. A *process table* resides in the primary memory of each MC. It contains an entry for each PE in the MC-group. Each entry contains the current status of the corresponding process.

Each MC can enter a diagnostic mode in which each of the PEs which it controls can be tested along with the communication lines between the PEs and MCs. The MC reports any hardware faults to the System Control Unit, which in turn reports them to the console. If the MC-group controlled by the MC has a fault so that it cannot be used for task execution, the task manager records that the MC-group is not available to be scheduled.

7. PROCESSING ELEMENT MONITOR

The PE monitor is the code for PASMOS which resides on each of the PEs. Each PE processor can operate in either SIMD or MIMD mode. While in SIMD mode, the PE fetches instructions from the its *instruction buffer* [13]. The PE is only able to fetch an instruction after all PEs in the MC-group have completed the previous instruction. When operating in MIMD mode, the operation of the PE is more complex. MIMD operation is initiated by the MC broadcasting a jump instruction to its PEs instruction buffers. The execution of this instruction causes the PEs to jump to one of their procedures. The PEs execute the instructions from their respective procedures. The last instruction in each procedure is a jump instruction which causes the PE to return to the instruction buffer. When all PEs have completed their procedures, the next instruction from the buffer is executed. Since the next instruction in the buffer is not executed until all of the PEs have completed the current instruction (i.e., the procedure), it is possible to synchronize the PEs when they are operating in MIMD mode.

The PE monitor provides facilities for dynamic memory allocation. This enables a process to request additional memory space as it is running. The PE monitor returns a pointer to the free memory location. In order to allow dynamic memory allocation to be done locally within each PE, the PE monitor must maintain a record of allocated memory space.

The user is able to specify MIMD process communication (i.e., PE-to-PE communication) at a high level by giving the logical name for the destination process. One problem is determining the physical address of the PE which is executing the destination process. Different approaches to this problem are currently being considered. One possible approach is a follows. When a communication request is made by an executing process, the PE asks its MC for the physical address of the PE which is executing the destination process. If it is not contained in the MC's process table, either the process is not executing or it is being executed by a different MC-group. With this approach, the MC queries all other MCs which are orchestrating the task to determine which MC-group is executing the destination process. The physical address is then transferred back to the MC, and in turn to the PE.

In order to eliminate this upward communication resulting from the use of this approach, each PE stores process address information for all processes with which it has communicated. The PE will check its memory before making a request to its MC. When a process completes execution, the PE continues to execute its monitor routine. If a PE receives any messages through the interconnection network which are for the process which it completed or any other process, an error condition occurs. Possible causes for this type of error are a fault in the interconnection network, a bad entry in the process address table of the PE which is executing the source process, or a programming error. This error can be reported to the System Control Unit in one of two ways: 1) the PE which detected the error can notify its MC which in turn can signal the System Control Unit or 2) the PE which has detected the error can send a message back to the source PE, which in turn can report the error. The first method is most likely a better choice since it has the most direct route to the System Control Unit and can avoid the section of the system where the error occurred. When the System Control Unit receives the report of an error, it terminates the task and notifies the user.

8. PASM COMMAND LANGUAGE

The *PASM Command Language (PCL)* is a set of commands which supplement the UNIX shell [6]. The commands can be grouped into three types: 1) Memory Storage System access commands, 2) Control Storage access commands, and 3) PASM task creation and monitoring commands. Just as for any other UNIX command, a UNIX process is created for each Parallel Computation Unit command which is executed.

The mmsdown and mmsup commands are used to transfer files between the System Control Unit's file system and the Memory Storage System's file system. The *mssdown* command allows the user to download data and program files from the System Control Unit to the Memory Storage System and the *mssup* command allows the user to upload data and program files from the Memory Storage System to the System Control Unit. Commands will also be provided to directly transfer data files between the Memory Storage System and tape, from the Memory Storage System to display devices, and from real-time sampling devices to the Memory Storage System.

File manipulation commands for Memory Storage System files are also provided by PCL. The *mmsls* command allows the user to list the names of the files which he/she owns. The *mmscp*, *mmsmv*, and the *mmsrm* commands allow the user to copy, move (rename), and remove (delete) files, respectively.

The csdown and csup commands are used to transfer files between the System Control Unit's file system and the Control Storage file system. The *csdown* command allows the user to download a MC-program from the System Control Unit to the Control Storage and the *csup* command allows the user to upload a MC-program from the Control Storage to the System Control Unit. The *csls*, *cscp*, *csmv*, and the *csrm* commands allow the user to list the file names in a Control Storage directory and to copy, move, and remove Control Storage files, respectively.

The last set of commands allow the user to create tasks and to monitor task status. The *rm* command issues a

request to the PASM task manager for a virtual machine. The command string specifies the task which is to be executed on the virtual machine. As with any other UNIX command, to execute a PCL command the UNIX shell normally creates a new UNIX process and waits for it to finish [6]. The UNIX process associated with the execution of the vm command is the parent process of any PASM task created by the command. As with any other UNIX command, a PCL command can be executed without waiting for it to finish. A list of currently active tasks can be obtained using the *ts* command. A task can be terminated by either killing the parent UNIX process using the UNIX kill command or by using the PCL *killt* command to directly terminate the task. It is noted that the syntax of the PCL commands is beyond the scope of this paper.

9. SCENARIO OF AN SIMD TASK

As an example scenario of an SIMD task, an image processing application is considered. An M-by-M image is represented by an array of M^2 pixels (picture elements), where the value of each pixel is assumed to be an eight bit unsigned integer representing one of 256 possible gray levels. In image smoothing each pixel is assigned a gray level equal to the average of the gray levels of the pixel and its eight nearest neighbors in the unsmoothed image. The operation is performed for each pixel in the image, with the possible exception of the edge pixels. In [26] an algorithm is given for performing the smoothing algorithm in parallel. SIMD algorithms for PASM will either be written in a "parallel" assembly language [13] or a "parallel" version of the C Programming Language [12].

To implement image smoothing on PASM using a virtual SIMD machine of 256 PEs, assume that the PEs are logically configured as a 16-by-16 grid, on which an M-by-M image is superimposed, i.e., each processor has a M/16-by-M/16 subimage. For M = 1024, each PE stores a 64-by-64 subimage. In the 256-PE SIMD algorithm, smoothing will be performed on the 256 subimages in parallel. At the boundaries of each 64-by-64 array (or subimage), data must be transmitted between PEs in order to calculate the smoothed values.

The sequence of commands used to execute the 256-PE smoothing algorithm on PASM is given in Fig. 5. The user must log into the System Control Unit and enter the PASM shell, *psh* (1). The smoothing algorithm is written in parallel C and includes the information that it is to be executed using 256 PEs. It is stored in the file *smooth.pc*. The program is compiled using the parallel C compiler, *pcc* (2), and downloaded to the Control Storage (3). The *a.out* file is the object file created by the compiler. The image data file, *image*, is downloaded to the Memory Storage System (4). The algorithm is then executed using the *vm* command (5) leaving the smoothed image in the Memory Storage System file name *image.s*. The smoothed image is uploaded from the Memory Storage System (6). After typing control-D (EOF) (7), the user is returned to the regular UNIX shell (8). The smoothing program can be removed from the Control Storage file system, or can be left there for future use. This also holds for the Memory Storage System data files.

```
(1)  * psh
(2)  @ pcc smooth.pc
(3)  @ csdown a.out smooth
(4)  @ mmsdown -d image image 256
(5)  @ vm smooth < image > image.s
(6)  @ mmsup -c image.s image.s 256
(7)  @ ^D
(8)  *
```

Fig. 5. Sequence of commands used to execute the 256-PE smoothing algorithm on PASM. * is the prompt for the UNIX shell and @ is the prompt for the PASM shell.

The following PASMOS actions result from the execution of the vm command:

1. In the command, the user requests the execution of an image smoothing program which requires 256 PEs (four MC-groups for N = 1024 and Q = 16) on the Memory Storage System file called *image*.
2. The task table entry is created by the task manager on the System Control Unit.
3. The task is assigned by the scheduler to the task queue for tasks which require four MC-groups (see [30] for details).
4. The input data for the task may be preloaded into the PCU memory units.
5. The scheduler eventually selects the task and assigns it to a virtual machine of four MC-groups (e.g., MCs 1, 5, 9, and 13).
6. The task manager requests that the Control Storage load the smoothing program into the MC memory units. If the input data for the task has not been preloaded, the task manager requests that the input data for the task be loaded.
7. If input data is being loaded, the MCs wait for signals from the Memory Management System which indicate that the input data has been loaded.
8. The PEs check if correct input data has been loaded.
9. If correct data has been loaded, the MCs begin to broadcasts instructions, otherwise the MC controlling the PE with the bad data signals the System Control Unit to abort the task.
10. After the last instruction is executed by the MCs, the MCs broadcast a jump instruction to the PEs which causes them to jump back to their PE monitors.
11. The MCs signal the System Control Unit that the task has been completed.
12. System Control Unit frees the MCs.
13. The task manager executes the scheduler.
14. The task manager requests that the Memory Storage System unload the output data.
15. Memory Management System signals System Control Unit when data unloading completed.
16. The task table entry is deleted.
17. The user is notified of task completion (user receives prompt).

10. PROTECTION

The protection mechanisms in an operating system control the access of programs (or tasks) to shared objects in the system. Since PASM is a multiuser system, protection mechanisms must be included as a part of PASMOS. Protection requirements within PASMOS can be divided into seven areas: 1) within the System Control Unit, 2) within each MC, 3) within each PE, 4) between the System Control Unit and the MCs, 5) among the MCs, 6) between the MCs and the PEs, and 7) among the PEs.

Each user of the PASM system will have an account under UNIX on the System Control Unit and, as a result, a unique *user identification number (uid)*. Protection within the System Control Unit is handled by UNIX. When a user issues a PCL command, UNIX associates the command with the uid of the user. For example, if the command to kill a task is executed, UNIX passes the uid of the user along with the request to the PASMOS kernel. The PASMOS kernel then checks if the given uid matches the uid of the task owner and, if so, kills the task.

It is not necessary to consider protection mechanisms within each MC since only one task can be assigned to a given MC at a time. Similarly, protection mechanisms are not required within each PE.

All communication between the System Control Unit and the MCs involves the PASMOS kernel on the System Control Unit. As a result, only the PASMOS kernel is able to write to ports which are connected to the MCs. When an MC sends a message to the System Control Unit, only the PASMOS kernel can read the message.

An MC which is executing a given task can only communicate with other MCs which are executing the same task. Protection mechanisms are provided to generate this. The kernel on each MC which is executing a task contains a list of all other MCs which are executing the same task. The MC kernel will only allow the MC to communicate with other MCs on its list.

Since each MC is only connected to the PEs which it controls, no protection mechanism is required. Protection mechanisms for communication among the PEs is handled by partitioning the interconnection network into independent sub-networks. A PE is only able to communicate with other PEs within the same partition of the network.

11. SUMMARY

PASMOS, a distributed operating system for the PASM parallel processing system, has been overviewed. The code for PASMOS, which is distributed between the System Control Unit, the Q MCs, and the N PEs, has been discussed. PASMOS has a hierarchical structure with three levels. At the System Control Unit level, tasks are managed; at the MC level, tasks are executed and processes are managed; and at the PE level, processes are executed.

ACKNOWLEDGEMENT

The authors thank Jim Kuehn for his comments and suggestions.

REFERENCES

[1] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Comput.*, vol. C-31, pp. 443-454, May 1982.

[2] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746-757, Aug. 1968.

[3] K. E. Batcher, "STARAN parallel processor system hardware," *AFIPS 1974 Nat. Comput. Conf.*, May 1974, pp. 405-410.

[4] K. E. Batcher, "STARAN series E," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 140-152.

[5] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proc. IEEE,* vol. 60, pp. 369-388, Apr. 1972.

[6] S. R. Bourne, "The UNIX shell," *Bell System Technical Journal,* vol. 57, pp. 1971-1990, July 1978.

[7] J. R. Dingeldine, H. G. Martin, and W. M. Patterson, "Operating system and support software for PEPE," *1973 Sagamore Computer Conf. on Parallel Processing,* Aug. 1973, pp. 170-178.

[8] A. J. Evensen and J. L. Troy, "Introduction to the architecture of a 288-element PEPE," *1973 Sagamore Computer Conf. on Parallel Processing,* Aug. 1973, pp. 162-169.

[9] T. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, vol. C-23, pp. 309-318, Mar. 1974.

[10] M. J. Flynn, "Very high-speed computer systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, Dec. 1966.

[11] J. Keng and K. S. Fu, "A special purpose architecture for image processing," *1978 IEEE Comput. Soc. Conf. Pattern Recognition Image Processing,* June 1978, pp. 287-290.

[12] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[13] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int'l. Conf. Parallel Processing,* Aug. 1982, pp. 353-362.

[14] J. T. Kuehn, H. J. Siegel, and M. Grosz, "A distributed memory management system for PASM," *1983 IEEE Comput. Soc. Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* Oct. 1983, to appear.

[15] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145-1155, Dec. 1975.

[16] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Comput.*, vol. C-31, pp. 1202-1214, Dec. 1982.

[17] G. J. Nutt, "Microprocessor implementation of a parallel processor," *4th Symp. Comput. Architecture*, Mar. 1977, pp. 147-152.

[18] G. J. Nutt, "A parallel processor operating system comparison," *IEEE Trans. Software Engr.*, vol. SE-3, pp. 467-475, Nov. 1977.

[19] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: an experiment in distributed operating system structure," *Commun. ACM*, vol. 23, pp. 92-105, Feb. 1980.

[20] D. S. Parker, "Notes on shuffle/exchange-type switching networks," *IEEE Trans. Comput.*, vol. C-29, pp. 213-222, Mar. 1980.

[21] D. S. Parker and C. S. Raghavendra, "The gamma network: a multiprocessor interconnection network with redundant paths," *9th Symp. Comput. Architecture*, Apr. 1982, pp. 73-80.

[22] M. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458-473, May 1977.

[23] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Bell System Technical Journal*, vol. 57, pp. 1905-1929, July 1978.

[24] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas reconfigurable array computer," *AFIPS 1980 Nat. Comput. Conf.*, May 1980, pp. 631-641.

[25] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, D. C. Heath and Company, Lexington, MA, to be published, 1983.

[26] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934-947, Dec. 1981.

[27] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a modular, multi-microprocessor," *AFIPS 1977 Nat. Comput. Conf.*, May 1977, pp. 637-644.

[28] D. L. Tuomenoksa and H. J. Siegel, "Application of two-dimensional bin packing to task scheduling in PASM," *19th Allerton Conf. Communication, Control, and Computing*, Univ. of Ill., Oct. 1981, p. 542.

[29] D. L. Tuomenoksa and H. J. Siegel, "Analysis of the PASM control system memory hierarchy," *1982 Int'l. Conf. Parallel Processing*, Aug. 1982, pp. 363-370.

[30] D. L. Tuomenoksa and H. J. Siegel, "Analysis of multiple-queue task scheduling algorithms for multiple-SIMD machines," *3rd Int'l. Conf. Distributed Computing Systems*, Oct. 1982, pp. 114-121.

[31] D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *1983 IEEE Comput. Soc. Conf. Computer Vision Pattern Recognition*, June 1983, pp. 336-344.

[32] D. L. Tuomenoksa and H. J. Siegel, "Preloading schemes for the PASM parallel memory system," *1983 Int'l. Conf. Parallel Processing*, Aug. 1983, pp. 407-415.

[33] W. A. Wulf and C. G. Bell, "C.mmp--a multi-miniprocessor," *AFIPS 1972 Fall Joint Comput. Conf.*, Dec. 1972, pp. 765-777.

BIOGRAPHIES

David Lee Tuomenoksa received the B.S. degree in 1980 and the Ph.D. degree in 1983, both in electrical engineering from Purdue University, West Lafayette, IN. As a graduate student, Dr. Tuomenoksa was a Research Assistant for the School of Electrical Engineering at Purdue University. In June 1983 he joined AT&T Information Systems Laboratories, Holmdel, NJ, where he is currently a member of the Technical Staff. His research interests include computer architecture, operating systems for distributed/parallel computer organizations, and office information systems. Dr. Tuomenoksa is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi honorary societies.

Howard Jay Siegel received the S.B. degree in electrical engineering and the S.B. degree in management from the Massachusetts Institute of Technology, Cambridge, MA, in 1972; the M.A. and M.S.E. degrees in 1974, and the Ph.D. degree in 1977, all in electrical engineering and computer science from Princeton University, Princeton, NJ. In 1976 Dr. Siegel joined the School of Electrical Engineering, Purdue University, West Lafayette, IN, where he is currently an Associate Professor. His research interests include parallel/distributed processing, multimicroprocessor systems, image processing, and speech processing. Dr. Siegel is the Chairman of the ACM SIGARCH (Special Interest Group on Computer Architecture) and is a member of the Eta Kappa Nu and Sigma Xi honorary societies.