A Facility for Simulating Multiprocessors

James M. Butler
Raytheon Company

A. Yavuz Oruc
Rensselaer Polytechnic Institute

Simulation models are better than analytical ones for understanding systems that behave unpredictably. But no such models have been available for asynchronous parallel systems—a situation the Euclid simulator corrects.

Parallel computer systems make possible the fast execution of a vast set of algorithms that can be handled by conventional uniprocessors only with great difficulty. However, parallel computer systems are much more complex than uniprocessors, and this complexity causes many design and implementation problems. One such problem is the development of working models for evaluating the behavior and performance of parallel computer systems under varying conditions. Because these systems are so expensive to build, the need for such models is great.

Several computation models of parallel computer systems have been reported. Among these, dataflow graphs,[1,2] PMS diagrams,[3] and Petri nets[4] are notable. These abstract models are especially useful for analyzing the control of dataflow in multiprocessor computers. They are supported by software tools such as Dijkstra's P and V primitives and semaphores[5] and Conway's FORK and JOIN mechanisms.[6] These, and other tools,[7-9] provide means for scheduling and synchronizing tasks and resolving conflicts among processors in asynchronous parallel computer systems. On the other hand, abstractions such as pipeline and array processor models provide formalisms useful for the study of synchronous parallel computer systems.

Though analytical models of parallel computer systems have been available for some time, simulation models of them have not. Simulation models are particularly important for evaluating the performance of asynchronous parallel computers, the behavior of which is very difficult to predict with analytical models. Here, we describe the design and implementation of a simulation model we call Euclid.

An overview of Euclid

Euclid is based on a parallel computer model called a processing network.[14] It accepts as input from the user four files containing information about the architecture that is to be modeled and the algorithm that is to be run on that architecture. It has facilities for creating, editing, displaying, and printing these files as well as runtime facilities for altering the conditions of the simulation as the simulation proceeds. It can also provide information and statistics about the simulation and give, both during and at the conclusion of the simulation session, the actual computational results of running the algorithm.

The processing network model. The processing network, or PN, is defined as a triplet PN = (T, f, P), where T is a finite set of objects called terminals, P is a finite set of objects called processors, and f is a mapping from P to the power set of T denoted P(T). Each processor p is specified as having a set of maps and/or operations H_p, each member of which is defined over the set $T_p = f(p)$, which is a member of the power set P(T). We call the mapping f the *domain map* of PN to emphasize that it describes the domain of terminals for each $p \in P$. Note that the model does not restrict the user to any one type of terminal—it can be a memory, an input or output device, or any other component that can be tied to a processor.

Let us look at an example of a processing network specification. Let

$T = \{t_i ; 1 \leq i \leq 12\}$
$P = \{p_i ; 1 \leq i \leq 4\}$

and let the map of P $\rightarrow$ P(T) be defined as

$$\begin{aligned} f(p_i) &= \{t_1, t_2, t_3, t_4\} \\ f(p_2) &= \{t_3, t_5, t_6, t_7\} \\ f(p_3) &= \{t_4, t_8, t_9, t_{10}\} \\ f(p_4) &= \{t_7, t_9, t_{11}, t_{12}\}. \end{aligned}$$

We complete the description of PN by adding a computational dimension to these specifications. Thus, let

$$H_{p_i} = \{h_{p_i}\}$$

such that

$$\begin{aligned} h_{p_1} &: \quad t_3 := t_1 + t_2 ; \quad t_4 := t_1 + t_2, \\ h_{p_2} &: \quad t_5 := t_3 ; \quad t_6 := t_7, \\ h_{p_3} &: \quad t_8 := t_4 ; \quad t_{10} := t_9, \\ h_{p_4} &: \quad t_7 := t_{11} + t_{12} ; \quad t_9 := t_{11} + t_{12}. \end{aligned}$$

The above representation of PN completely describes the activities that take place between PN's processors and terminals. In this simple example, p_1 and p_4 are set to add the contents of their terminals and send the sums to their other two terminals, whereas p_2 and p_3 are set to route the contents of their two terminals to their other two terminals. It is clear that the user can specify each processor to have other operations, including conditional statements; hence, each processor can be as powerful as the user desires.

We should note that the above model does not directly incorporate time—an essential ingredient of all processes—into the activities of the processors. The user can accomplish this easily, however, by adding execution-ordering relations and processor-activity rules to the model. He has many ways in which he can coordinate the activities of the processors.

PNs can be divided into two categories. A PN is *fully asynchronous* if the execution of the maps or operations of any one of its processors has no time dependency on the execution of those of any other of its processors. If otherwise, the PN is *synchronous*. Each processor of an asynchronous PN can perform its operations at its own speed and access its terminals at any time regardless of what the other processors are doing, as long as there are no conflicts among processors.

The PN model is an ideal base on which to build a simulation system because it is general, flexible, and simple. To make it fully functional in a simulation environment, however, we must refine its definition.

The terminal entry. In the formal PN definition, a terminal is specified as any device that can be connected to one or more processors. To develop a functional simulator, we must narrow this specification somewhat. Therefore, we will consider a terminal to be a data path rather than a device. For example, if there is a terminal t connected to processors i and j, a legal data transfer may take place between processors i and j via terminal t. Communication may take place in only one direction at a time, however, since a terminal may never simultaneously possess more than one value.

Dead-end terminal. A dead-end terminal is specified as one that is connected to one and only one processor. Therefore, it cannot be used as a data path. However, we will define it to be a local—though external—register for its host processor. A processor may host a file of several of these registers for use in complex arithmetic operations.

Memory module. A memory module is defined as a collection of data registers or locations in which each data register or location has a unique numerical identifier called an address. The organization of addresses in a memory module is completely arbitrary. A memory module has one and only one input/output terminal, which is called its port terminal. Therefore, one can access only one location per module at a time. However, one can specify a collection of memory modules that share the same port terminal. This collection is called a port-connected set of modules. Since data transfers to and from these modules are made over the same port terminal, only one location per port-connected set can be accessed at a time.

User-supplied components. The four files supplied by the user are the architecture file, the priority memory access file, the user-defined instruction set file, and the Multisoft object code file.

The architcture file contains the hardware description of the architecture to be simulated; this description is based on the refined PN model. The architecture file contains the terminal map of each processor specified in the architecture, the port terminal of each memory module specified in the architecture, and the address map of each memory module, which is simply a list of addresses belonging to that module.

The priority access file—also called simply the priority file—contains information that is not provided by the refined PN model and that can be thought of as firmware support. This information is of three types. The first is information about the memory access priority level. This priority level is used by the simulated system's arbitration mechanism in the event of memory access conflicts. The second type of information in the priority file is the instruction execution rate of each processor. A different rate can be specified for each processor in the simulated system; each rate is similar to the clock period of a real processor. Hence, one can simulate systems having processors with different clock speeds. The third type of information in the priority file is the memory access bandwidth of each memory module. This bandwidth determines the speed with which a memory module can be accessed; it can be different for each module in the simulated system. Thus, one can specify some modules as fast cache memories and others as slower bulk memories.

The user-defined instruction set file contains the instruction set of each processor defined in the architecture file. In terms of the refined PN model, the instruction set is the set of H maps that defines the set of operations each processor may perform over its set of terminals. The user-defined instruction set file is written in Pascal and linked directly to the Euclid Simulation System task image. Thus, one can

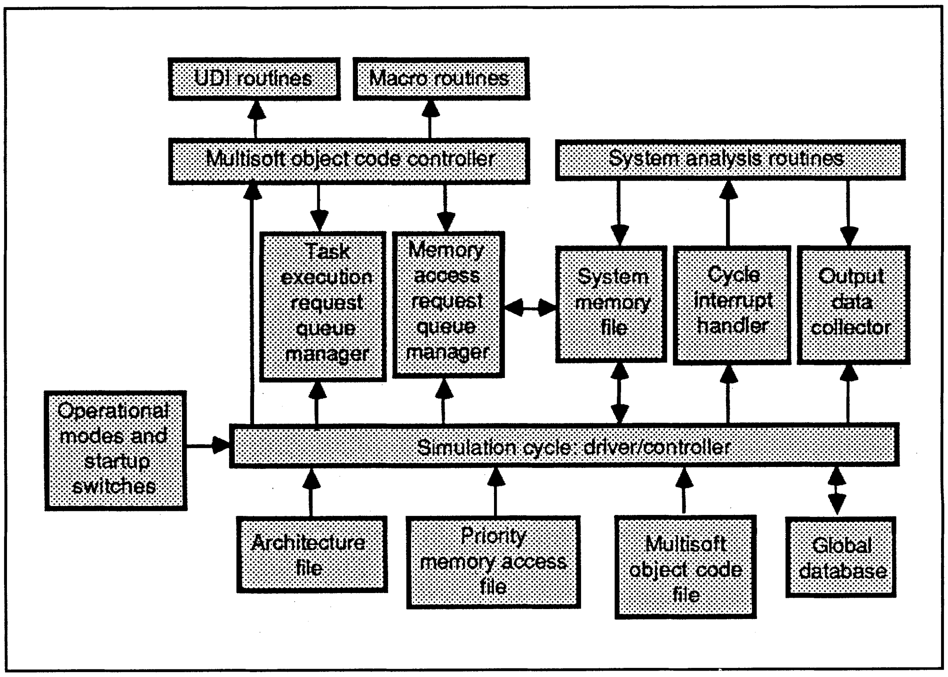

UDI routines
Macro routines
Multisoft object code controller
System analysis routines
Task execution request queue manager
Memory access request queue manager
System memory file
Cycle interrupt handler
Output data collector
Operational modes and startup switches
Simulation cycle: driver/controller
Architecture file
Priority memory access file
Multisoft object code file
Global database

Figure 1. The Euclid simulation cycle.

make an instruction as complex as one desires. One can specify any legal set of Pascal operations, including conditionals and loops.[10] However, we should note that although Euclid allows looping, it still considers it to be a single instruction performed by the calling processor. A user-defined instruction, or UDI, has access to all system terminals and can issue memory access requests.

The Multisoft object code file contains the user's program, which is written in a pseudo-machine-language called Multisoft. Euclid supports a full object code editor for creating and editing programs.

Once the user creates the four input files and links the UDI file to the task image, he must set several presimulation operation modes. Although these modes have default values, the user should set them according to his needs or to certain other conditions. These modes set the memory organization; the type of simulation—free run, single step, or checkpoint; the scheme for resolving memory access conflicts; and the format for displaying the simulation output. A detailed discussion of these modes is provided by Butler.[11]

Once the user specifies the input files, sets the operational modes, and prepares the memory system as he desires, he can start the simulation by doing little more than issuing a command to that effect. A simulation run can terminate in three ways—in the normal completion of the test algorithm, in an abort caused by a user-issued cycle interrupt, and upon the occurrence of a simulation time error (that is, a runtime error). Most runtime errors stem from syntax errors in the Multisoft instructions. Others arise from PN-model-related problems such as a terminal being written to by more than one source, a nonexistent address being specified, or a processor issuing a memory access request to a memory module that has no connection to that processor. Butler provides a complete list of runtime errors.[11]

A Euclid simulation run yields two sets of results: the algorithm results and performance statistics. One can examine the algorithm results by accessing a memory preparation facility and using the previously described tools for printing and displaying the contents of a memory module. The performance statistics take the form of a report on how the simulated system behaved during the simulated execution.

Implementation of Euclid

We should keep in mind that the Euclid simulator runs on a sequentially organized, uniprocessing computer. Therefore, the actions of the processors defined in a simulated system are not taking place concurrently, but sequentially. Since concurrent processes must not be dependent on one another in a real distributed system, the sequential ordering of intracycle processes in a simulated environment can be arbitrary. The Euclid object code controller is responsible for sequentially assuming the role of each processor and executing the necessary instructions. Because the order in which the processors execute can be arbitrary, the controller, for convenience, simulates the processors in ascending processor number order (that is, processor 0 executes first, processor 1 executes next, and so on). Traffic procedures watch processes to make sure they are nondependent and monitor data transfers and flags generated by violations.

The Euclid simulation cycle is shown in Figure 1. The outer-level procedure that controls the tasks to be performed in each cycle is called the cycle driver. Data to be input to the cycle driver are obtained from the settings of the operational mode switches and the execution of the setup block. Throughout simulation, the cycle driver refers to the architecture file, the priority file, and the Multisoft object code file and maintains a dialog with the global database.

In the first phase of its activity, the cycle driver calls the Multisoft object code controller. This controller executes instructions for all active processors that are not suspended by calling the appropriate UDI and Multisoft macro routines. It issues task execution requests and memory access requests to the appropriate queues.

The second phase of the cycle driver's activity involves the granting of task execution requests. The cycle driver calls on the *task execution request queue manager* to collect any requests that have been scheduled on an inactive processor. If the manager finds any such requests, it grants the oldest of them.

The third cycle driver phase involves the granting of memory access requests. The cycle driver calls on the *memory access request queue manager* to collect the requests for each memory module or each port-connected set of modules. The manager grants the oldest of the highest-priority requests. In the event of an access conflict, it invokes arbitration procedures. Any processor that has outstanding memory access requests at the end of this phase is suspended and will not be allowed to execute an instruction in the first phase of the next cycle.

The final phase of the cycle driver's activity involves updating statistical data, checking the error flags, executing the cycle interrupt handler, if necessary, and checking for program halt conditions.

File structure and organization. Because the amount of data needed to describe the simulated system can be quite large, it is maintained in the form of disk files. An upper limit on system parameters, imposed in part by these files, still remains, however. We should note that the structure of the architecture file, in particular, is not organized for optimum compactness. Compactness was sacrificed for simpler and more efficient file maintenance.

The architecture file. This file is an integer file comprising four sections. The first section is a global set of parameters, the second contains the processor terminal maps, the third is a list of the port terminals for the memory modules, and the fourth contains the memory module address maps. The first section contains the maximum parameter values that were in effect on the version of Euclid under which the file was created. They are included in the file because the file size depends on these values. Their inclusion allows expanded versions of Euclid to read files that were created on smaller versions. Modules are separated with negative number delimiters. For example, the address list for module i is preceded by a $-i$.

Table 1 shows the structure of a typical architecture file. The maximum values for system parameters are shown in parentheses for the first five cells. Unused cells contain zeroes.

The priority memory access file. The priority file is also a file of integers and, like the architecture file, begins with some maximum parameter values (Table 2). The size of this file depends on the maximum number of processors and memory modules that were in the simulation when the file

Table 1.
Structure of the architecture file.

Address	Explanation
0	Maximum total number of terminals. (1024)
1	Maximum total number of processors. (64)
2	Maximum total number of memory modules. (32)
3	Maximum number of addresses/module. (512)
4	Maximum number of terminals/processor. (64)
5	Largest terminal number used.
6	Number of processors defined.
7	Number of memory modules defined.
8	Number of terminals connected to P_1.
9-72	P_1 terminal map.
73	Number of terminals connected to P_2.
74-137	P_2 terminal map.
.	
.	
.	
4103	Number of terminals connected to P_{64}.
4104-4167	P_{64} terminal map.
4168-4199	Port terminals for modules 1-32.
4200	Delimiter for M_1 address map. (-1)
.	M_1 address map.
.	Delimiter for M_2 address map. (-2)
.	M_2 address map.
.	
.	
.	Delimiter for M_n address map. $(-n)$
.	M_n address map.

Table 2.
Structure of the priority memory access file.

Address	Explanation
0	Maximum total number of processors. (64)
1	Maximum total number of memory modules. (32)
2	Number of defined processors.
3	Number of defined memory modules.
4-67	Priority levels of processors 1-64.
68-131	Instruction execution rates of processors 1-64.
132-163	Access bandwidths of modules 1-32.

was created. Note that in this file cells 2 and 3, which show the number of defined processors and modules, are not used just for displaying and printing the file. If these numbers do not correspond with those in the architecture file, the architecture file data are used in the simulation instead. The only rule that applies here is that the number of processors specified in cell 2 of the priority file must be equal to or greater than the number specified in cell 6 of the architecture file.

Note that Table 2 shows the priority file structure for the parameters in cells 0 and 1.

Multisoft object code file. The object code file is a file of integers. Its structure is defined by the Multisoft object code instruction syntax and by the Multisoft programmer. The size of this file is limited to 32,768 addresses numbered from 0 to 32767.

The memory file. This file serves as the memory system for the simulated hardware. It is a file of n real numbers, where n is the total number of addresses in the system. It is aligned with the address maps of the architecture file, although the module delimiters are removed. Let us examine how this alignment works. Suppose that module 1 has 100 locations and address x appears as the fifth address in the address map of module 2. Data written to location x will be stored in the 105th cell of the memory file, which will be memory file location 104 since the numbering begins at zero. When a memory read or write is to be performed, the address maps must be consulted so the correct memory file offset can be found.

The global database. Euclid maintains a database that is global to all its functions and procedures. It maintains it in real memory, not in disk files. The database comprises various control structures, request queues, and statistical data structures that are needed during simulation. We need to examine these structures to understand how the Euclid simulation machine operates.

Simulation control structures. One of the most important control structures is the processor activity vector Pro_Active(i), where $1 \le i \le p$, and where p is the maximum number of processors. The array elements are defined as follows: If processor i is inactive and, therefore, ready to accept a new task, Pro_Active(i) = -1. If processor i is active and not suspended, Pro_Active(i) = 0. If processor i is active but suspended due to n outstanding memory access requests, Pro_Active(i) = n.

When a task is scheduled onto processor i, Pro_Active(i) is set to 0. When processor i is released, it is set back to -1. Each time a memory access request is issued by processor i, Pro_Active(i) is incremented. Each time such a request is granted, it is decremented. The object code controller is only allowed to execute an instruction for processor i if Pro_Active(i) = 0.

Another important control structure is the multiple program counter MPC(i). It is an integer vector and serves as the program counter for each of the processors. When a task is scheduled onto processor i, MPC(i) is set equal to the address of the task block header. When the processor is released, MPC(i) is cleared to zero. The object code controller uses this vector to read the proper instruction for each processor and update it as needed.

The processor frequency array is used to control the number of cycles required to execute an instruction. It is denoted Pfreq(i,j), where $1 \le i \le p$ and $1 \le j \le 2$. The Pfreq(i,1) vector is loaded with the instruction execution rates obtained from the priority file before simulation. The corresponding elements of the Pfreq(i,2) vector are used as counters that start at one. When the object code controller attempts to execute an instruction for processor i, it checks this array. If Pfreq(i,2) = Pfreq(i,1), the instruction executes and Pfreq(i,2) is reset to one. If, however, Pfreq(i,2) < Pfreq(i,1), Pfreq(i,2) is incremented and the instruction does not execute. Thus, if the instruction execution rate of processor i is n, then $n-1$ cycles will pass before the instruction finally executes in the nth cycle.

The memory frequency array performs an analogous function for the memory access granting process. It is specified as Mfreq(i,j), where $1 \le i \le m$, m being the maximum number of memory modules, and where $1 \le j \le 2$, as before.

File synopsis data. Maintaining so much of the simulated data in disk files has one marked disadvantage—disk references are very slow and add up to large amounts of time over long simulation runs. To improve its performance in this respect, Euclid extracts much of the smaller or often used data from the disk files at the time the files are opened and stores that data in the database. In fact, it completely reads the priority file when it is opened and immediately closed. It stores the priority level of each processor i in Pma_Vec(i), and the processor and memory speed figures in Pfreq and Mfreq.

Euclid cannot do this with the architecture file because of that file's size, but it can do other things to improve its performance in respect to the architecture file. It assigns the maximum number of terminals used, the number of specified processors, and the number of specified memory modules to integers nt, np, and nm, respectively. It loads the list of memory module port terminals into the vector Mport(i), where $1 \le i \le m$ and where m is the maximum number of memory modules.

Euclid also uses an array called Tdef. This array is a Boolean vector. Euclid searches the terminal maps of all processors and memory modules for the purpose of setting Tdef(i) to TRUE if it finds that a terminal i exists. The result of this process is a list of all the terminals used in the architecture.

The last data structure derived from the architecture file is the memory parameters array Mem_Par(i,j), where $1 \le i \le m$, as before, and $1 \le j \le 5$. Table 3 explains the meaning of the five vectors. With the statistical data about the memory system that is given in this array, Euclid can search for memory file offset values much more efficiently. If the memory organization is random, it will have to search the architecture file, but since the address maps are arranged into ascending order upon their entry, it can use the Mem_Par(i,1) vector to skip quickly to the next module as soon as it has discounted a module from the search.

Statistical data structures. The processor statistics are compiled into a simple array Pro_Stat(i,j), where $1 \leq i \leq p$, as before, and $1 \leq j \leq 2$. Pro_Stat(i,1) is a count of the number of cycles during which processor i is active; therefore, it is incremented at the end of each cycle in which Pro_Active(i) ≥ 0. Pro_Stat(i,2) is a count of the number of cycles during which processor i is used, i.e., is active and not suspended. This count is incremented at the end of each cycle in which Pro_Active(i) $= 0$. From these two vectors and the total cycle counter CYCLE, Euclid can easily calculate the performance statistics we call Activity, Utilization, Suspension, Task Run, and Task Delay. It calculates the total number of operations of processor i by dividing Pfreq(i,1) into Pro_Stat(i,2), since processor i requires Pfreq(i,1) cycles to complete one instruction.

Memory access statistics are accumulated in an array Mem_Stat(i). Each time memory module i grants an access request, Mem_Stat(i) is incremented. Terminal traffic statistics are handled in a similar manner: Ter_Stat(i) is incremented once for each cycle in which terminal i is read from or written to. Note that if terminal i is read by several elements in a given cycle, Ter_Stat(i) is still incremented only once.

The task execution request queue. The queueing system for task execution requests consists of two arrays and a queue length pointer. The pointer is an integer called PQ_Length that simply keeps track of the queue length. The arrays are aligned and are called PQ_Start and PQ_Location. PQ_Start(i) denotes the task header address of the ith task execution request. PQ_Location is an array of Boolean vectors that serves as the compatibility mask C-MASK for each request. By scanning across the vector j in PQ_Location(i,j) for values of TRUE, Euclid determines the processors that request i can be scheduled onto.

In each cycle and for each inactive processor, Euclid searches the queue, beginning with the oldest entries, for a task execution request that can be scheduled onto such a processor. If it finds one and that entry is the kth request in the queue, it loads the program counter for the processor with PQ_Start(k), removes PQ_Start and PQ_Location from the queue, and updates the queue while decrementing PQ_Length.

The memory access queue. A processor makes a memory access request using a request record. This record contains the requesting processor's number, the cycle in which the request was issued, the target terminal of the data transfer, and a Boolean variable which is TRUE if the access type is READ and FALSE if the access type is WRITE. The memory request queue can be considered a slotted queue because each memory module is assigned its own slot.

Table 3.
The memory parameters array.

Mem_Par(i,1)	The offset of the first address of module i in the architecture file.
Mem_Par(i,2)	The number of locations (addresses) in module i.
Mem_Par(i,3)	The address of the first location in module i.
Mem_Par(i,4)	The index value of module i. (Valid only for sequential or indexed organization.)
Mem_Par(i,5)	The memory file offset for the first location in module i.

Therefore, the queue can be considered an array of request records MQ(i,j), where i is the request number and j is the memory module slot number. A record called MQ_Length (j) tracks the queue slot length for memory module j.

When a processor makes a request, Euclid determines from the address the memory module the location belongs to. It then places the request at the end of the appropriate queue slot and increments the appropriate MQ_Length element. One request per cycle may be granted for each module or port-connected set of modules. Whether a request is granted or not depends on the priority level of the requesting processor and the conflict resolution scheme chosen by the user.

The terminal structure. Terminal entities are actually vectors of real numbers T(i), where $1 \leq i \leq t$ and where t is the maximum number of terminals. There are also two supporting Boolean vectors, T_Read(i) and T_Write(i). At the beginning of each cycle, they are both cleared to FALSE. Whenever terminal 1 is read from, T_Read(i) is set to TRUE. Likewise, whenever terminal 1 is written to, T_Write(i) is set to TRUE. This allows Euclid to flag illegal data transfer attempts such as multiple writes to a terminal.

Data transfers such as memory access grants performed by Euclid automatically set these vectors and increment the statistical data vector. Data transfers and mathematical operations performed with UDIs do not do this automatically. A pair of procedures is provided to ensure the integrity of the simulation run and the performance statistics. For example, if a particular UDI adds the data on terminals 4 and 5 and then writes the sum to terminal 8, the UDI will appear as

T(8) := T(4) + T(5); TR(4); TR(5); TW(8);

However, one may wish to allow an operation such as T(4) := T(4) + T(5). This violates a traffic rule that states that a terminal may not be read from and written to in the same cycle. However, for a working register that is local to a processor, for example, it may be convenient and nondestructive to allow this operation, provided the user is aware that

Table 4.
System parameters.

Constant	Value	Explanation
Pro_Max	64	Number of processors.
Mem_Max	32	Number of memory modules.
Term_Max	1024	Number of terminals.
Addr_Max	512	Number of addresses/module.
Conn_Max	64	Number of terminals/processor.
TEQ_Max	128	Task execution queue length.

an exception is being made. To prevent Euclid from flagging this situation as an error, the UDI should be written as

T(4) := T(4) + T(5); TR(5); TW(4).

The rules of dataflow. Euclid follows a number of operating rules to ensure the integrity of a simulation run. Some are imposed by the refined PN model, and others arise from various concerns about dependency and concurrency. These rules are listed below:

(1) A terminal may not be written to more than once in a cycle. Such writing to would imply that the terminal can be multiply defined, which it cannot.

(2) A terminal may not be both read from and written to in the same cycle. Since the two possible sequential orderings of these events can produce different results, concurrency is not possible.

(3) A processor is allowed to execute instructions involving terminals only if all of those terminals appear in the terminal map of that processor. A processor is not allowed access to terminals that are not in its terminal map.

(4) A processor is allowed to issue memory access requests only for those memory modules whose port terminals appear in the terminal map of that processor. This is a consequence of Rule 3.

(5) A memory access request issued by a processor must specify a target terminal that appears in the terminal map of that processor. This is also a consequence of Rule 3.

(6) A memory module is allowed to grant only one memory access request in a given cycle. Since a memory module has only one port terminal, this rule is a consequence of Rule 1.

(7) A port-connected set of memory modules is allowed to grant only one memory access request for the entire set in a given cycle. This is also a consequence of Rule 1.

Maximum system parameters and limitations. The Euclid simulator, like any system, has limitations. Most of them are imposed by the computer on which it runs. Euclid was designed for use on small office systems such as the IBM PC and the DEC Professional 350. But since Euclid simulates high-performance multiprocessors and distributed computers, certain upper bounds on the simulations can be expected to present themselves. The limitations we will discuss concern the IBM PC implementation of Euclid.

Program constants and memory usage. The maximum values of the simulation system parameters are referenced throughout the Euclid package as constants declared in the global database. Table 4 shows these constants and the values that are in effect for them in the current version of IBM PC Euclid. The IBM PC under MS-DOS allows the task image to occupy a 64K-byte region of memory and the globally declared database to occupy a separate 64K-byte region. The constants shown in Table 4 cause approximately 54K bytes of data space to be occupied.[11]

Another restriction is created by the file handling procedures. Since the addresses of the input files are integer data types, these files are limited to 32,768 addresses numbered 0 to 32767. This limitation applies chiefly to the memory system. The total number of allowed addresses must not overflow the architecture file. This is one reason why so much attention is paid to port-connected modules. If large, single memory modules are needed, several can be port-connected to fit the requirement. The trade-off is that the number of definable separate modules decreases.

The UDI file size. The UDI file size, UDISET.PAS, must be linked into the Euclid task image. Thus, size considerations are of interest. Of the 64K bytes allocated to the task image, the current version of Euclid uses only about 34K bytes after overlaying. This leaves a contiguous block of about 30K bytes available for the UDI file, which should probably be enough for any application. If it is not, however, the Turbo Pascal compiler which must be resident on the system running Euclid allows procedures that are local to a procedure to be overlaid. In other words, the UDI set is a single procedure that can be up to about 30K bytes long. But, by declaring a collection of nondependent procedures within this UDI procedure as overlaid, Euclid can accommodate even a file originally much larger than 30K bytes.

One must consider another factor when dealing with the UDI file. There is a file called UDIVAR.PAS that may contain global declarations for the UDIs that the user may require. Such declarations can be counters or any other type of data that must be preserved across UDI calls. These declarations, if they are to be made, will have space allocated in the 64K-byte data region. One must exercise caution to ensure that no conflicting declarations are made and that allocation of these variables does not try to exceed the 64K-byte limit.

Multisoft programming language

Multisoft was developed specifically for the Euclid simulator. It comprises elements borrowed from several programming environments. It includes generic instructions for controlling program flow, such as LOAD, INCREMENT, JUMP, and various conditional branching instructions. Also of interest are its READ and WRITE instructions, which issue memory access requests. It provides a class of instructions for initiating processes on various processors. A program written in Multisoft is actually a collection of these processes, which are separately defined; henceforth, we will refer to these processes as *tasks*. The instructions for initiating processes include the SPAWN, SUBTASK, and PARAFOR constructs, each of which will be defined below.

User-defined instruction sets are also supported by Multisoft. An interface exists that calls the UDI procedure when a UDI reference is made during simulator execution. Since in most cases the bulk of the computing power of a simulated system will exist in the UDIs, the macro instruction set supported by Multisoft has been kept to a minimum. Placing the burden on the UDI set also serves the purpose of maintaining generality within the Euclid environment.

The setup block. The setup block is used to specify which task or tasks should be initially scheduled and onto which processors they should be scheduled. During program execution, tasks may schedule other tasks, but at least one processor must be initially activated. The setup block provides a list of operand pairs, each of which specifies a processor and the starting address of its initial task. Valid elements of this list are put into effect immediately, with the following exception: If the list contains one or more tasks that specify the same processor, the first of these is scheduled and the remainder are queued. The assembly code syntax for the setup block is

```
SBH       ; Setup block header
  P1 T1   ; 1st processor/task schedule pair
  P2 T2   ; 2nd processor/task schedule pair
     .
     .
     .
  Pn Tn   ; nth processor/task schedule pair
SBT       ; Setup block terminator
```

Task structure. To be properly structured, a task must have two components: the header block and the task body. The header block performs certain housekeeping functions for Euclid as well as three optional subinstructions. The body of the task contains Multisoft instructions and/or UDIs. There must be at least one HALT instruction in the body of every task, but it does not necessarily have to be the last instruction.

The task header block. A macro called HEADER is always the first action a task performs. It does so automatically, and HEADER is considered to be the first operation of the task. Thus, the first instruction specified by the programmer after the task header block will be the second operation performed by the task.

For each task, five locations in memory must be allocated to Euclid so it can store status information about the task. These addresses are known as the A-block addresses and are referred to as A1, A2, A3, A4, and A5. In the next section, we will discuss in more detail what these locations are used for.

Task scheduling instructions. There are three instructions that can be used to schedule tasks for execution. Each of these is described below.

The SPAWN instruction. A task may spawn other tasks by placing requests on the task execution queue. It does this by using the SPAWN instruction. Each request within the SPAWN instruction has two parts. The first is the address of the header block of the task to be spawned. The second is a compatibility mask. This mask designates the set of processors upon which the spawned task may execute. The request will wait in the queue until one of these processors is available.

The compatibility mask comprises 16-bit integers; each bit corresponds to a processor. In a Euclid version in which the maximum number of allowed processors is set to 16, the mask requires only one integer. But if, for example, the version allows 64 processors, the mask must be four integers long even if a particular architecture uses only 16 processors. The integers that comprise the mask are designated M1, M2, . . . , M*n*. The least significant bit (LSB) of M1 corresponds to processor 1, the most significant bit (MSB) of M1 to processor 16, the LSB of M2 to processor 17, the MSB of M2 to processor 32, and so on. In the descriptions of syntax, the entire mask is referred to as the C-MASK.

The assembly code syntax for the SPAWN instruction is

```
SPAWN          ; SPAWN instruction
  OP1t OP1m    ; Request 1: task address, C-MASK
  OP2t OP2m    ; Request 2: task address, C-MASK
     .
     .
     .
  OPnt OPnm    ; Request n: task address, C-MASK
VIT            ; Variable instruction terminator
```

The SUBTASK instruction. This instruction allows a task to jump to another task (i.e., to a subtask) on the same processor and return when the subtask is completed. It is similar to a traditional subroutine call except that no explicit parameter passing takes place and the block of subroutine code must always be a completely defined task. There is no Multisoft-imposed bound on the level of subtask nesting. The assembly code syntax for the SUBTASK instruction is

```
SUBTASK OP1 ; Task header address
```

The PARALLEL FOR instruction. The PARALLEL FOR construct is by far the most complex of the task scheduling instructions. It is similar to a traditional FOR loop, with one major difference. With a FOR loop that has *k* iterations, the same block of code is executed sequentially *k* times. The PARALLEL FOR construct assumes that the iterations of the block are nondependent and therefore can be executed concurrently on as many as *k* processors.

In Multisoft, the block of code is a complete task, and for *k* iterations, *k* task execution requests will be issued.

Operands 1, 2, and 3 make up the traditional loop parameters of lower limit, upper limit, and index or incrementer, and all of these can be memory referenced and/or memory indexed. Operands 4 and 5 are memory addresses that must be allocated to the control variable and the index, respectively. Operand 6 is the task header address of the task to be generated followed by its compatibility mask. The processor calling the PARALLEL FOR may or may not take part in the iterations, depending on its C-MASK bit. In either case, the calling processor will not continue with the next instruction after the PARALLEL FOR until the last iteration has completed. PARALLEL FOR calls may not be nested. The assembly code syntax for the PARALLEL FOR instruction is

```
PARAFOR OP1  OP2  OP3  ; Lower, upper limit, index
        OP4  OP5       ; CV, index address
        OP6  C-MASK    ; Task header address,
                           C-MASK
```

Branching instructions. These instructions are the Multisoft versions of conditional and unconditional branching. In assembly code syntax, the jump target operand *n* may appear as

- OP*n*, a symbol to be resolved by the assembler,
- A(OP*n*), an absolute jump to the address OP*n*, or
- R(OP*n*), a relative jump by offset OP*n*.

The assembly code syntax for the JUMP instruction is

```
JUMP OP1 ; OP1 is the jump target operand
```

The conditional branching instructions. There is actually one conditional branching instruction with six variations. The instruction reads as follows: If OP1 * OP2, then branch to OP3, where * represents one of the six relational operators shown in Table 5 and OP3 is the jump target operand. OP1 and OP2 are allowed to be memory referenced and/or memory indexed. The assembly code syntax for the conditional branching instruction is

```
BRANCH OP1 OP2 OP3 ; BEQ, BNE, BLT, BGT, BLE,
  BGE
```

Miscellaneous Multisoft instructions. Multisoft also supports instructions that, when combined with the branching instructions described above, make possible the convenient specification of loops. Other instructions allow read and write accesses to the memory modules. A short description of these is given below. A more detailed treatment of these instructions and the ones discussed earlier can be found in Butler.[11]

The LOAD instruction. This instruction allows the direct loading of a memory location specified by OP2 with a value specified by OP1. The assembly code syntax for the LOAD instruction is

```
LOAD OP1 OP2 ; Load OP1 onto OP2
```

The INCREMENT instruction. This instruction allows the incrementing of a memory location specified by OP2 by a value specified by OP1. The assembly code syntax for the INCREMENT instruction is

```
INC OP1 OP2 ; Increment OP2 by OP1
```

Table 5.
The six variations of the conditional branching instruction.

Operator	Mnemonic	Operator	Mnemonic
$=$	BEQ	$>$	BGT
$\neq$	BNE	$\leq$	BLE
$<$	BLT	$\geq$	BGE

The READ and WRITE instructions. These instructions allow access to the memory module contents by issuing memory access requests. The syntax for the READ instruction is shown below; the WRITE syntax is identical except that READ is replaced by WRITE:

```
READ OP1 OP2 ; Read address OP2 to terminal OP1
```

The multiple access construct. In many cases, it may be necessary or convenient to issue several READ and WRITE requests in a single instruction. The multiple access construct makes this possible. In the following syntax, the word "access" refers to a valid READ or WRITE instruction that follows the above syntax for those instructions:

```
MAH          ; Multiple access header
  access-1   ; READ/WRITE instruction 1
  access-2   ; READ/WRITE instruction 2
     :
     :
  access-n   ; READ/WRITE instruction n
VIT          ; Variable instruction terminator
```

UDI and no-op syntax. The set of user-defined instructions must be numbered in the range 1 to 512. Any instruction within this range is assumed to be a UDI and not a Multisoft macro instruction. The assembly code syntax for UDI number *n* is UDI *n*. If Euclid reads a zero while executing a Multisoft program, the program counter is incremented and the zero is treated as a NOP (no operation).

Simulation example: jigsaw puzzles

Here we will examine, as a sample simulation problem for Euclid, a method for solving jigsaw puzzles. Other simulation examples can be found in Butler and Oruc.[12] This example was inspired by the work of Green and Juels,[13] in which a clever analogy was made between an uncooperative group of children and asynchronous multiprocessing. The jigsaw puzzle makes a good distributed performance test because of the level of list searching, communications, and decision making involved. It also lends itself quite well to asynchronous solution techniques.

The jigsaw puzzle is modeled as a collection of square tiles whose orientations are known. Each of the four edges of a tile has an integer value assigned to it. If an edge of a tile is also part of a boundary edge of the puzzle, it has a value of zero. Any internal edge of a tile has a positive value that is equal to that of the adjacent edge of the matching tile. Only one solution to the puzzle exists.

Solution philosophy. The solution used was a combination of techniques that mimic humans and worker ants. Two stores of puzzle tiles exist: the input store and the output store. For the input store, an ant colony provides the best analogy. Assume that there is a pile of food crumbs large enough so each ant can pick from it without conflict. This does not guarantee that no conflicts will occur, since there is no "foreman." Likewise, processors will choose tiles from the input store at random, in order to inspect them. Euclid will resolve conflicts in a random fashion, deciding who will obtain a tile and who will have to look elsewhere.

On the output side, processors will have the ability to inspect the solution space simultaneously. To make the problem more interesting, however, a limit will be set on the number of processors that are allowed to simultaneously place tiles in the output (or solution) space. We impose this rule to emulate the situation that would prevail if people were working on the puzzle—many people could view and inspect the puzzle but fewer could gain physical access to it to place their tiles.

The requirement that a tile will have to meet before it can be placed in the output store is that it form part of the boundary edge of the puzzle or that it mate with at least one tile already in the output store.

Architectural model. The simulated distributed architecture for jigsaw puzzle solving is shown in Figure 2. There are *n* identical processing elements, where *n* will take on values of 1, 2, 4, 8, and 16. The input store is a collection of *n* memory modules that are accessible by each of the processors. The tiles are initially distributed evenly over these modules. The first address of each module is a count of the tiles resident in that module.

The output store is a collection of memory modules which are *n* in number but are limited to four. Again, all processors have access to each module and the first location of each module is the module tile count. This bound imposes the rule of limited simultaneous access to the output store.

Each processor has address space allocated to it in the general-purpose memory module. This space is used for loop counters, temporary storage, and so on, and it is an additional source of memory contention. Memory module locks serve as semaphores that allow processors to control access rights to the input and output memory modules. The rules are as follows: If an input module is locked by a processor, no other processor may have read or write access to the module. If an output module is locked by a processor, other processors may have read access only. This is as if to say that while one person is placing a tile in the puzzle, others may still inspect the area near this placement.

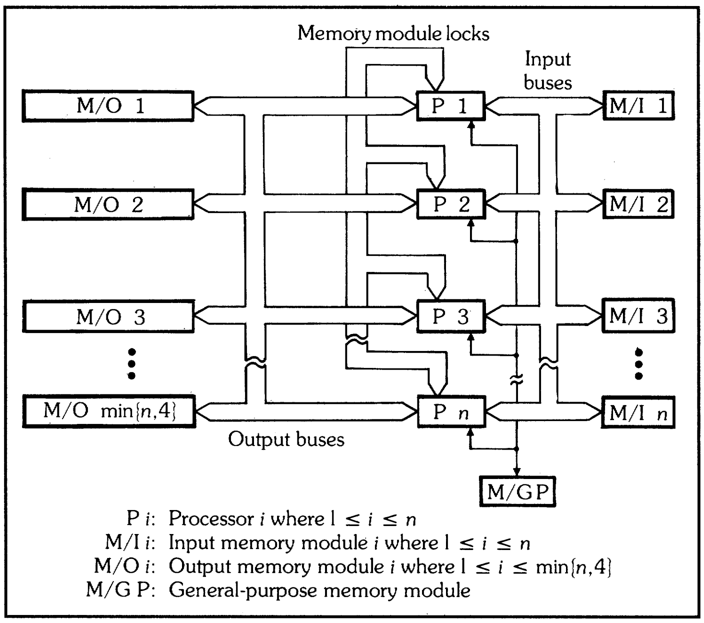

Memory module locks
Input buses
M/O 1
M/O 2
M/O 3
M/O min{n,4}
P 1
P 2
P 3
P n
M/I 1
M/I 2
M/I 3
M/I n
Output buses
M/GP
P i: Processor i where 1 ≤ i ≤ n
M/I i: Input memory module i where 1 ≤ i ≤ n
M/O i: Output memory module i where 1 ≤ i ≤ min{n,4}
M/G P: General-purpose memory module

Figure 2. An architecture for solving jigsaw puzzles.

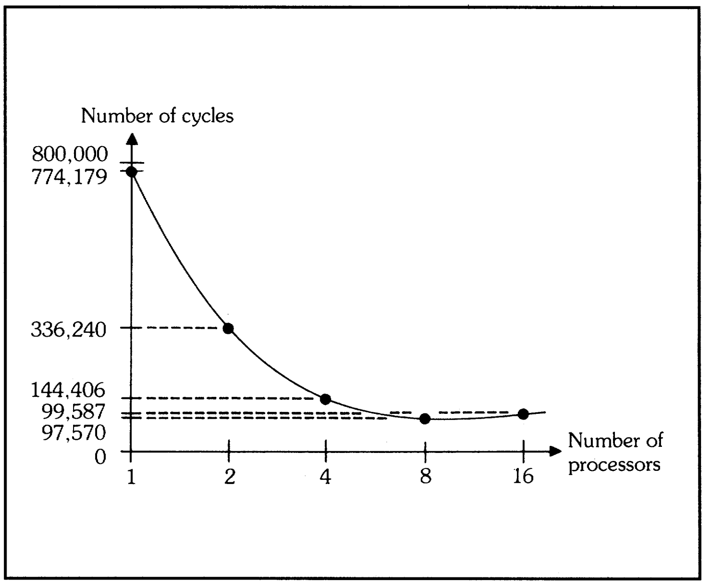


Figure 3. Number of cycles vs. number of processors.

The algorithm. The distributed algorithm that implements the puzzle solution was written in Multisoft, the distributed simulation language supported by Euclid. All the processors execute the same algorithm and all are defined as having the same instruction set. The only difference among them is in their mapping of general-purpose memory and local registers. An understanding of the algorithm can best be derived from the English version in the box on page 43.

Results. Figures 3 to 5 illustrate the performance characteristics of the puzzle-solving distributed architecture as it

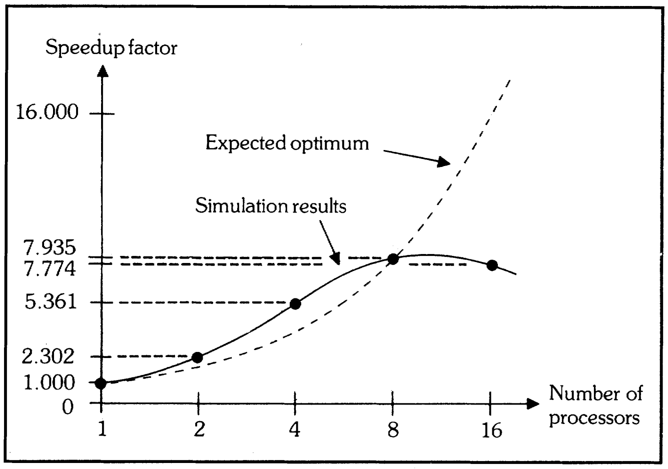


Figure 4. Speedup (referenced to the single-processor case).

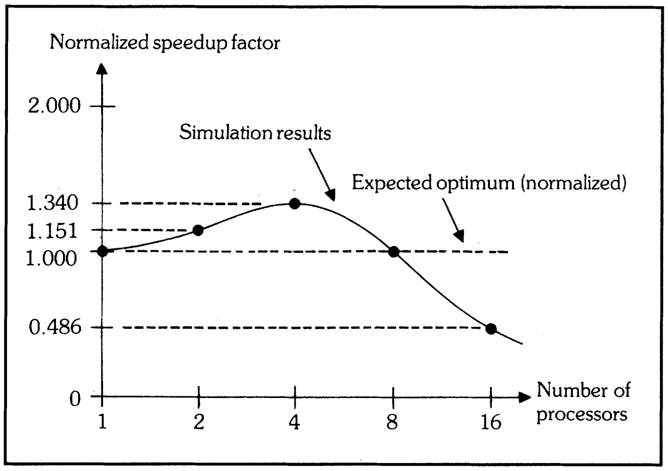


Figure 5. Normalized speedup (referenced to the single-processor case).

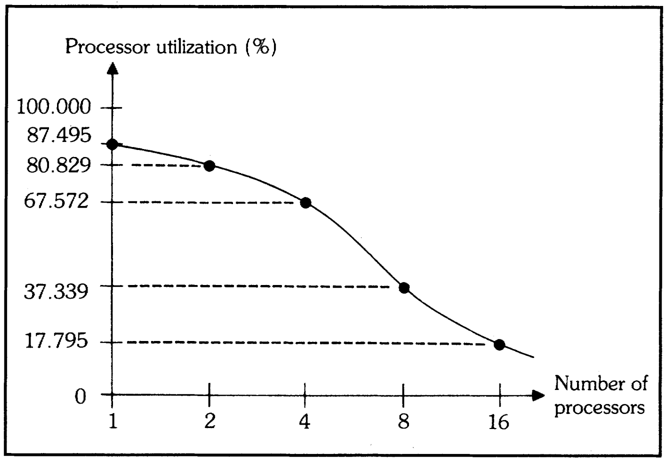


Figure 6. Processor utilization vs. number of processors.

executes the distributed algorithm for a 16 × 16-tile puzzle. Figure 3 shows that the single-processor case required 774,179 clock cycles from the simulated system. It is clear from this figure that additional processing hardware provides notable improvements in performance until the 8- and 16-processor cases are reached. Performance actually declined somewhat with 16 processors.

The speedup resulting from additional processors is shown in Figure 4 and reveals an unexpected anomaly. The points on this curve are referenced to the single-processor case. The dashed curve shows what one might expect to be the upper bound on speedup—a speedup of *n* for the *n*-processor case. However, the two- and four-processor cases actually provide an increase in performance that exceeds this expected optimum. The eight-processor case produces approximately the expected optimum, whereas the 16-processor case produces less than half of the expected optimum.

We can explain the better-than-expected performance by intuitively examining the algorithm. In the single-processor case and during the early part of the simulation, the chances that a tile will be placed in the output module are very small, since only 24.2 percent of the tiles have puzzle-boundary edges and all the other tiles must mate with a tile already in the output store. Thus, the accumulation of tiles starts very slowly and eventually speeds up. When a second processor is added, the early buildup of tiles proceeds at about twice the rate. The increase in performance occurs because the rate of growth of the probability that a tile will be successfully placed increases proportionally with the tile buildup. This process dynamic introduces the nonlinear shift in performance by causing the probability of success to rise faster in the earlier stages of execution than in the later stages.

Figure 5 shows the speedup profile after we normalized each point by dividing the speedup factors by the corresponding number of processors. The normalized expected optimum is, therefore, flat at 1.000. This curve shows the point at which one can expect to get the best performance per processor dollar.

Figure 4 indicates that of the cases which were simulated, the eight-processor one realized the fastest execution time. Figure 5 shows that for a given level of hardware, the four-processor case performed the best. These results provide the information one needs to make an intelligent decision. If the cost of hardware is of less concern than the speed of execution, eight processors may be the best choice. On the other hand, if the best marriage of system economy and performance is desired, a four-processor architecture may be the better way to go.

Figure 6 shows the processor utilization profile for the five cases. Utilization is defined as the percentage of active time during which a processor is *not* suspended waiting for memory access grants. This curve helps explain the decline in performance exhibited by the two largest architectures. In these two cases, the advantage of greater task distribution was offset by the very low utilization figures. Key sources of memory contention in this study were the general-purpose memory module, locks placed on input and output modules, and, in the 8- and 16-processor cases, the self-imposed restriction of the number of output slots to four. The latter was probably not a dominating factor, however, since output modules are locked only while a tile is being placed in them and are not locked during the searching process.

Final observations. The jigsaw puzzle example helps point out the value of simulation. Prior to executing our simulation runs, we made several predictions about performance trends and execution times. Most of these estimates proved to be off, and some were completely inaccurate. We might have come up with better predictions if we had done a more extensive mathematical analysis. Such an analysis, however, would have had to include a great number of seemingly negligible factors, ones that do not remain negligible when processor interaction and contention increase. Such an analysis was beyond our resources and, moreover, such an analysis usually leaves too many uncertainties and unanswered questions. We can answer most such questions by direct observation—through simulation.

The implementation and development of a system such as Euclid can never be a static proposition—it can never be called complete. Its purpose is to aid computer designers in developing high-performance systems or to help researchers in testing predictions about system behavior and performance. If a simulation system is to remain useful, it must always be able to deal with uncertainty. (Once a designer or researcher learns enough about a design's characteristics and performance, his need for simulation decreases.) The current configuration of Euclid stands very near to what could be called a plateau of usefulness. It works well and offers its users many services. However, it can still be improved—it can use more software to support its interaction with the user, for example. A high-level compiler equipped with parallel constructs such as those in Parallel Pascal[10] could be very helpful for specifying programs to Euclid.

There could be two basic problems in developing such a compiler. First, unless very carefully written, it would tend to encroach upon the generality and flexibility of the current version of Multisoft. Second, it would have difficulties working with Euclid's memory system, since that system is completely definable by the user and thus can be constantly changing.

One possible solution to both of these problems could be to create an interactive compiler. The user could be called upon to guide the compilation process to ensure that the resulting code conformed to the desired specifications. Even if the amount of interaction seemed tedious to the user, it would still be a vast improvement over hand-translating a large set of complex tasks into object code.

The puzzle solution algorithm

Choose an input module to pick a tile from.

(1) If all input slots are empty, HALT.
(2) From the nonempty input modules choose an input module. If the module is locked, go to Step 1. Otherwise, lock the input module.

From this input module, choose a tile to inspect.

(3) Randomly select a tile from those residing in the input module.
(4) Read the tile parameters from the module into local registers, erase the tile from the input module, and decrement the module tile count.
(5) Unlock the input module.

Determine if the tile may be placed in the output store.

(6) If the selected tile has at least one outer puzzle edge, go to Step 14.
(7) From the nonempty output modules select an output module for inspection and note this selection.
(8) Search the tiles in this output module for a mate for the selected tile. If a mate is found, go to Step 14.
(9) If nonempty modules that have not been inspected still remain, go to Step 7.

Put the tile back into its original input module.

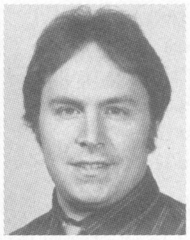
(10) If the input module from which the selected tile was picked is locked, go to Step 10.
(11) Lock the input module.
(12) Write the tile parameters back into the input module and increment the tile count of this module.
(13) Unlock the input module and go to Step 1.

Put the tile in the output store.

(14) From the output modules which are not full, choose an output module.
(15) If this output module is locked, go to Step 15.
(16) Lock the output module.
(17) Write the tile parameters to the output module and increment the tile count of this module.
(18) Unlock the output module and go to Step 1.

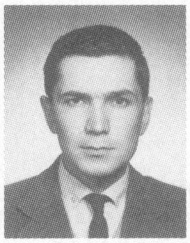
References

1. J.B. Dennis and D.P. Misunas, "A Preliminary Data Flow Architecture for a Basic Data Flow Processor," *Proc. 2nd Symp. on Computer Architecture*, 1978, pp. 144-151.
2. D.A. Adams, "A Model for Parallel Computations, Parallel Processor Systems," in *Technologies, and Applications*, L.C. Hobbs, ed., Spartan Books, New York, 1970, pp. 311-333.
3. D.P. Siewiorek, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982.
4. J.L. Peterson, "Petri Nets," *Computing Surveys*, Sept. 1977, pp. 223-252.
5. E.W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, F. Genuys, ed., Academic Press, London, 1968, pp. 43-112.
6. M. Conway, "A Multiprocessor System Design," *AFIPS Conf. Proc.*, Vol. 24, 1963 FJCC, pp. 139-146.
7. P. Brinch Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J., 1973.
8. A. Silberschatz, "Communication and Synchronization in Distributed Systems," *IEEE Trans. Software Eng.*, Vol. SE-5, No. 6, Nov. 1979, pp. 542-546.
9. G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Mar. 1983, pp. 3-42.
10. A.P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers," *J. Parallel and Distributed Computing*, 1984, pp. 64-80.
11. J.M. Butler, "Euclid: An Architectural Simulator for Distributed Computers and Multiprocessors," MSc thesis, Rensselaer Polytechnic Institute, Troy, N.Y., Dec. 1985.
12. J.M. Butler and A.Y. Oruc, "Euclid: An Architectural Simulator for Multiprocessors," *Proc. 6th Int'l Conf. on Distributed Computer Systems*, Boston, 1986, pp. 280-288.
13. P.E. Green and R.J. Juels, "The Jigsaw Puzzle: A Distributed Performance Test," *Proc. 6th Int'l Conf. on Distributed Computer Systems*, Boston, 1986, pp. 289-295.
14. A.Y. Oruc, "Analysis and Design of Processing Networks," tech. report, ECSE Dept., Rensselaer Polytechnic Institute, Troy, N.Y., 1984.



James M. Butler is a computer systems engineer in the ATE Design Group at Raytheon MSD in Lowell, Massachusetts. His current interests include multiprocessing, methodologies of distributed data acquisition, and highly parallel digital systems.

Butler received his MS degree in computer and systems engineering in 1985 from Rensselaer Polytechnic Institute, his BS degree in electrical engineering in 1984 from the University of Lowell, and his AS degree in electronic engineering technology in 1981 from Boston's Wentworth Institute of Technology. He is a member of Tau Alpha Pi, Eta Kappa Nu, and the Computer Society of the IEEE.



A. Yavuz Oruc has been on the faculty of the Department of Electrical, Computer, and System Engineering at Rensselaer Polytechnic Institute since January 1983. His research interests include parallel processing and analysis and design of connection and computation networks for advanced computer systems.

Oruc received the BSc degree in electrical engineering from Middle East Technical University, Ankara, Turkey, in 1976; the MSc degree in electronics from the University of Wales, Cardiff, United Kingdom, in 1978; and the PhD degree in electrical engineering from Syracuse University, New York, in 1983. He is a member of the Computer Society of the IEEE.

Questions about this article can be directed to Dr. A. Yavuz Oruc, ECSE Dept., Rensselaer Polytechnic Institute, Troy, NY 12180.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 156 **Medium** 157 **Low** 158