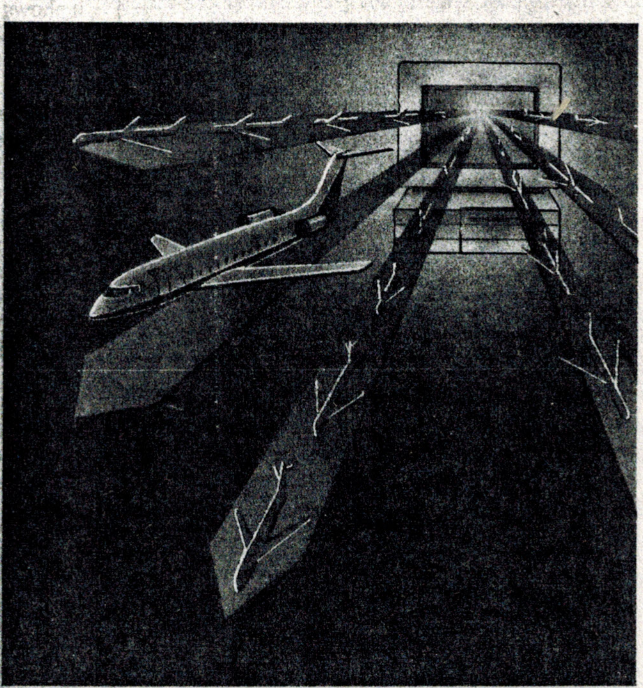
T800 and Counting

The T800 transputer and the Occam language are a hardware/software team designed to work together

Richard M. Stein



Mankind constantly seeks ways to solve the technological challenges found by observing the natural universe. Today, our understanding of nature is increasingly dependent on computer-based simulation of theories, suppositions, and curiosity.

Such complex operations as verifying the fluid flow of air around the wing section of an airplane to determine drag and stability, studying chemical reactions in the preparation of a new drug, and studying weather patterns require the increases in performance provided by parallel-processing computers. In fact, many problems are beginning to require that increase in speed to quench our growing thirst for immediate responses. As parallel-processing computers become more and more available, these workhorses of science and industry are emerging as a key to continued technological growth.

Multiprocessors vs. Multicomputers

Parallel-processing computers can be divided into two basic architectures: the shared-memory multiprocessor (see figure 1a) and the multicomputer (see figure 1b). The shared-memory multiprocessor comprises a collection of CPUs connected by a bus to a common pool of memory.

A multiprocessor performs parallel computations in several ways. One is to dedicate a complete processor to each active process; this is called *control parallelism*. Each process is free to operate on memory without appreciable interference from the others. Why not add more processors and keep partitioning the problem, one process to each processor, to gain more speed? A problem arises as you add CPUs to the bus: When one CPU tries to access an address in memory, it must first get permission from the others. Arbitration among the CPUs leads to contention. Each CPU requires a finite amount of time to fetch something from memory, and while this is going on, the other CPUs must wait if they need data as well. Adding more CPUs simply makes the problem worse, and a bottleneck results.

This is the so-called von Neumann bottleneck. It's the reason multiprocessors seldom have more than four CPUs simultaneously operating on a common pool of memory. The bus bandwidth is saturated by simultaneous requests from the CPUs. The shared resource leads to a form of inflation, where the cost of performing an operation becomes increasingly expensive and therefore less efficient. This is the key limitation to multiprocessor architectures: Finite bus bandwidth means that only a fixed num-

continued

ILLUSTRATION: ROBERT PASTERNAK © 1988

ber of instructions can be carried out each second. While this may be appreciable—more than 500 million floating-point operations per second (MFLOPS) is possible—the growth rate in the bandwidth is limited by technology.

The multicomputer differs from the multiprocessor in several ways. Processors are not connected to a common bus. Instead, each processor has a small (64K-byte to 4-megabyte) RAM connected to a local bus—the processor and RAM are called a *node*—and communication between processors occurs through high-speed serial links. There is no bottleneck. Since a node doesn't share a common bus with any other node and communication occurs through serial links, the multicomputer's bandwidth rises linearly with the number of nodes.

The multicomputer has other advantages as well. The most notable is cost. Multicomputers require less glue logic and fewer support chips on a per-node basis than do multiprocessors. A multicomputer typically runs between one-tenth and one-hundredth the cost of a comparable multiprocessor.

The Transputer

About 4 years ago, INMOS introduced the transputer, a multicomputer building block. It has a great cost-performance ratio: A T800 transputer with 4 megabytes of RAM costs about $1000, and a 30-MHz T800 delivers 2.25 IEEE 32-bit MFLOPS and 15 million instructions per second, well under $100 per MIPS. In addition, the transputer requires little support circuitry: You can build a fully functional multicomputer node (a transputer, a 5-MHz crystal, a few pull-up resistors and diodes, four F373 parts [octal latches], some RAM, and a wire-wrap tool) in a few hours. The most costly part of the hardware is the RAM.

Figure 2 shows the internal structure of the T800 transputer. This architecture is unique when compared with conventional CPUs. The T800 incorporates 4K bytes of on-chip static RAM. A program that fits into this on-chip reservoir will execute instructions in the transputer's cycle time—that is, in the 33-nanosecond cycle for the 30-MHz version. The internal RAM is not cache memory per se, as many conventional reduced-instruction-set-computer processors have, but it does serve an important role as stack space.

The internal RAM is used by compilers to hold the base addresses of arrays and local procedure variables. The base address of an array is used far more often than a single array-element address during program execution. The size and type of an array element are fixed at compile time, so a simple calculation can determine the address of any array element. The internal RAM serves as a register stack, an area where variables used repeatedly are held to speed access and program execution. A register variable is accessed in a single cycle, but variables held in external RAM require a handful of cycles to latch and read.

Using the transputer's internal RAM in this way does have one side effect, and it shows up in the language specification for Occam, the transputer's native language. By definition, the Occam language is not recursive. If it were, the repeated stacking of the local variables would generate confusion at each level of recursion during program execution. Stack frame building requires dynamic memory allocation, which Occam does not support. Implementing recursion on the transputer is a bit tricky in Occam, since the application must maintain the stack and you must explicitly manage stack traversal.

The T800 integer-register set is sparse but highly functional (see figure 3). The three accumulators are arranged as a stack and serve as expression evaluators. The workspace pointer tracks the address of the data that the active process is using. The instruction pointer is similar to the program counter found in conventional CPUs and points to the current instruction.

The transputer's operand register serves as the focal point for instruction processing. All transputer instructions are 1 byte long and typically execute in one to two cycles. The transputer forms an operand by loading the instruction data field into the 4 least significant bits of the operand register. The instruction uses the contents of the entire operand register as its operand; it clears the operand register to 0 on completion.

The transputer also performs instruction prefetch. Since each instruction is 1 byte, four will fit into one word on the

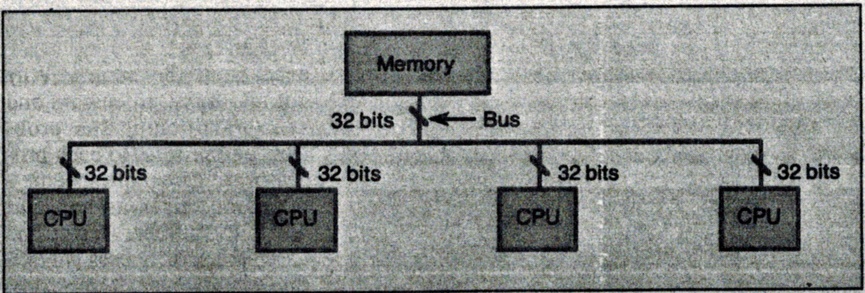


Figure 1a: *A typical shared-memory multiprocessor (without contention memory cache). The bus is a common resource among all CPUs; as more CPUs are added, bus contention can lead to the von Neumann bottleneck.*

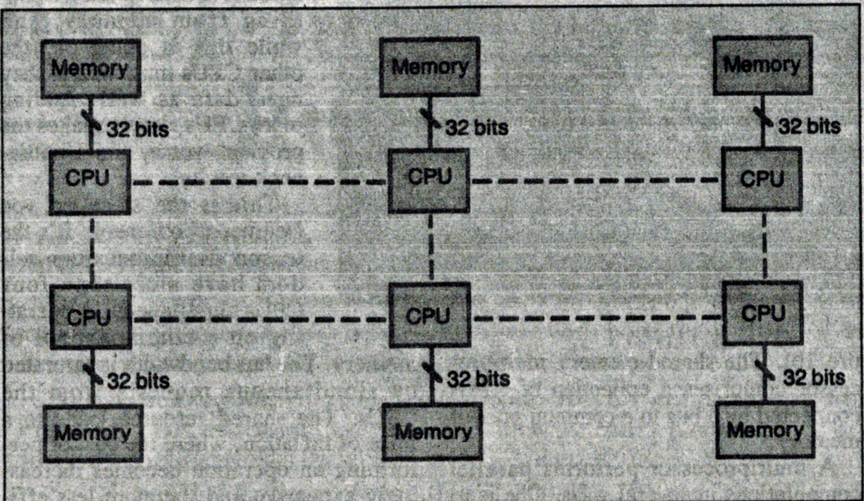


Figure 1b: *A typical multicomputer system. Each node consists of a CPU and memory. Processes communicate with each other through bidirectional serial links (dotted lines).*

T800. Each instruction fetch retrieves four instructions simultaneously; this requires less frequent accesses to memory. The transputer maintains a double-buffered instruction queue. The prefetch-and-buffering scheme delivers most of the performance benefits of an instruction cache, but without the silicon's cost. The prefetch-and-buffering sequence almost completely decouples instruction execution time from memory speed.

The transputer is designed to execute concurrent processes under direct hardware control. A by-product of hardware process scheduling is realized in the extraordinarily short context switches. Due to a hardware stack maintained by transputer microcode, most context switches require only between 1 and 2.5 microseconds (μs). A transputer process consists of a local workspace and a small reserved area for linkage information, which holds the pointers used to maintain multitasking and I/O protocols.

The hardware scheduler supports two process priorities: high and low. It gives the high-priority process unconditional control over the CPU and prevents the low-priority processes from executing until the high-priority one relinquishes the CPU. You use a high-priority process for very short—less than one time slice—sequences of instructions that are not to be interrupted by external events or bumped by other processes. The time-slice period is approximately 1 millisecond for a low-priority process. All low-priority processes execute asynchronously. This asynchronous execution scheme is an important concept for multicomputer software systems.

The transputer has an on-board hardware timer, which can be used to obtain synchronous interrupts for time-critical processes. The timer derives its signals from the externally connected 5-MHz crystal. The crystal supplies the transputer's phase-lock loop with the synchronization signal coordinating all the internal mechanisms. The timer is accessed through a channel assignment, and it has either 1-μs or 64-μs resolution. With these clock intervals, you can schedule processes tightly to execute at precise intervals.

The T800's floating-point unit conforms to IEEE standard 754-1985. At 30 MHz, it is 50 percent faster than a 16-MHz 80386/80387 combination. The T800 performs 32-bit floating-point multiplication, addition, and subtraction in less than 1 μs, and it requires just over 1 μs for division. The transputer's FPU can run in parallel with the integer CPU,

continued

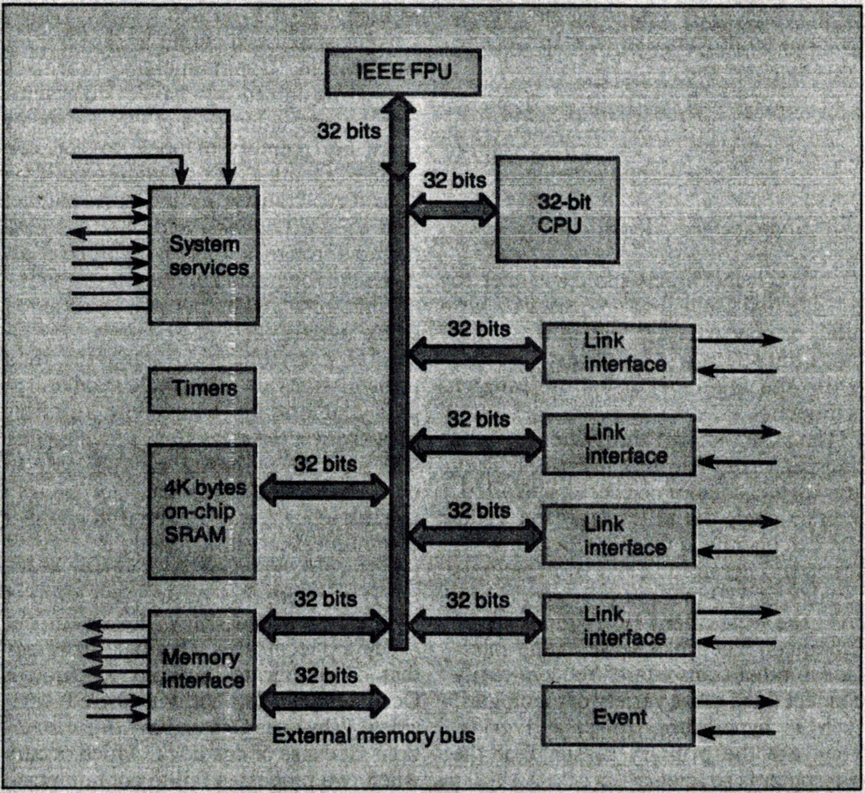


Figure 2: *The INMOS T800 transputer. The CPU and FPU can execute in parallel by organizing Occam to exploit both processors simultaneously. The chip is designed for concurrency.*

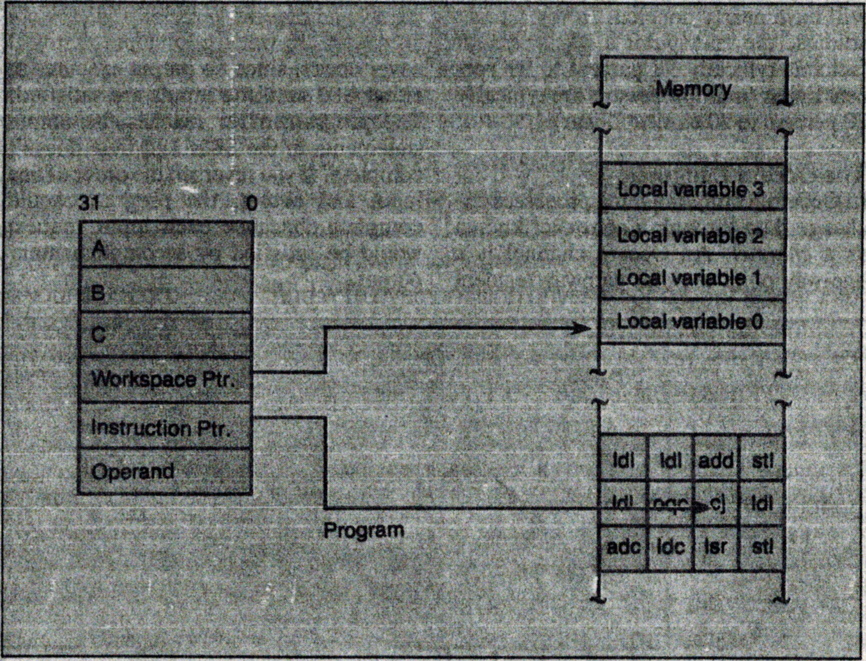


Figure 3: *The T800 integer CPU has three accumulators, registers A, B, and C, arranged as a stack for expression evaluation. Zero-address instructions operate on values in the stack; single-address instructions load values from memory into the stack, and so on.*

programmatically separating integer and floating-point computations. In Occam, you can write this as

```
PAR
  SEQ
    ...do floating-point calculations
  SEQ
    ...do integer calculations
```

This fragment directs the transputer to execute two sequential processes in parallel, or concurrently: One process executes only floating-point instructions, while the other performs the integer arithmetic.

The most distinguishing feature of the transputer is its four link interfaces. They are direct-memory-access-controlled, bidirectional, serial-transmission links and can operate at up to 30 megabits per second. Therefore, each transputer is capable of 120-megabit-per-second link I/O, the equivalent throughput of 12 Ethernets. The links serve as the interface to other transputers. You can easily connect them into a variety of topologies, such as hypercubes, rings, and grids. They are the primary reason that the transputer is so linear.

The transputer is linear in the sense that if you execute a program on a single transputer, gather some performance data, and then partition the software to run on two transputers, the performance will have nearly doubled. In ray-tracing studies, the transputer's improvement factor is typically 98 percent to 99 percent linear (multiprocessors are typically 70 percent to 80 percent linear).

The Occam Language

In Occam, communicating processes exchange data through a construct known as a *channel*. An Occam channel is a one-way point-to-point pathway resident in memory; it is termed a *soft channel*. Channel input and output are special for two reasons: Communicating processes are synchronized by channel communication (see figure 4), and the transputer links are memory-mapped so you can "place" channels at link addresses. This placement transforms the soft channel into a *hard channel*, and data is transferred through a link to a process attached to the link on another transputer.

Link input and output share a common clock signal (not a prerequisite for successful communication), and the data transmissions are self-synchronizing. Inputs and outputs can occur simultaneously over the same link, provided that two separate processes are available to both send and receive data.

Link I/O, or, more generally, channel communication, is a pivotal feature of the Occam language. Occam provides the framework for constructing parallel processes (processes with concurrent execution contexts). Parallel processes that communicate must do so through Occam channels, not via shared variables. Why does Occam have this restriction? Because of deadlock, which occurs when two processes fail to communicate correctly as a result of improper coding or design.

Listing 1 illustrates deadlock. The two `SEQ` processes execute simultaneously; the first requests input on `chan2`, while the second requests input on `chan1`. Both processes are waiting for input that will never occur, since no output executes in either `SEQ` until the inputs are satisfied. The program never reaches the output statements in the `SEQ`s and thus doesn't complete. If you reversed the order of one input and output, the program would complete, because each input request would be satisfied by a complementary output.

Attempting to share a variable among parallel processes gives rise to a similar conflict: One process may try to write the variable at the same time another one is trying to read it. Since parallel processes run asynchronously (at their own rate), reading a variable that has been modified by another process means the value would be uncertain. Since you can't know when the variable will be modified, you could be trying to read it when another process is writing it. To prevent this collision, Occam precludes parallel processes from sharing variables. But Occam variables can and do store data, and channels are available for interprocess communication.

The point-to-point nature of the Occam channel discourages the design of a program dependent on routing data through intermediate nodes. While "through-routing" is typically implemented on the transputer as a separate software process that enqueues and dequeues packets, the hardware provides an easier approach. (Second-generation transputers are likely to have this feature.) You can't always achieve a logically concurrent description that isolates communication dependency to a nearest neighbor. Through-routing circumvents this design limitation, and it's much faster in hardware than in software.

Several manufacturers have developed transputer plug-in boards for both the IBM PC and the Macintosh. CSA, MicroWay, and Definicon Systems all build plug-in PC boards with varying amounts of RAM, transputers, crossbar switches, and price. Nth Graphics manufactures a transputer-based PC plug-in graphics engine running the Hoops graphics package from Ithaca Software. A typical single transputer board with a 20-MHz T800 and 2 megabytes of exter-

continued

Figure 4: *When process 1 must wait for one time unit before process 2 is ready to accept its data, it is said to be "blocked." Processes synchronize only when they communicate via Occam channels.*

nal dynamic RAM costs about $3000, including an Occam compiler, documentation, and some utilities. Levco makes a plug-in transputer board for the Mac.

Occam is a secure language with a robust and efficient compiler, but it does have some shortcomings. For one, Occam fails to provide hierarchical data-structure typing, an essential for object-oriented programming. You need to have a means of abstracting the problem domain into more than just assignments, inputs, and outputs. The success of a computer model is often characterized by the correctness of the problem-domain abstraction; this is far more easily achieved in languages like Ada, C, and C++. Occam doesn't support the `struct` syntax of C, enumeration, or dynamic memory allocation. It also doesn't support a recursive syntax, so the application program must stack and unstack recursive data structures like binary trees and linked lists.

C, FORTRAN, and Pascal compilers are available for the transputer. An Ada language compiler from Alsys is planned for August 1989, and rumor has it that Glockenspiel, Ltd., in Dublin, Ireland, is working on a C++ compiler based on the 3L Parallel C compiler. Software tools for the transputer are becoming more widespread.

Logical Concurrency

Logical concurrency is a natural part of any problem domain composed of multiple degrees of freedom. Any system you can view as a collection of processes is said to possess logical concurrency. A formal definition states that the amount of logical concurrency is equal to the number of simultaneous processes or composite coincidental actions occurring in a closed system modeled by a computer program or simulation.

This definition applies to multicomputers. In multicomputer systems, you gain speed by partitioning the processes among different processors that perform work concurrently. Multicomputers excel in applications where the problem domain possesses data parallelism. You can process a large quantity of data when many nodes simultaneously operate on small, independent parts of the database.

It's customary to classify logical concurrency in terms of granularity. For instance, say a balloon filled with a gas contains 10^{23} molecules. If you attempt to model the equations of motion for each molecule—no small undertaking—you would need a fine-grained logically concurrent description of the problem (see figure 5). However, if you treat the balloon as a composite of 1024 volume elements (and compute an average value for some observable quantity, such as the temperature or pressure in each element), you would consider a medium-grained logically concurrent model. Even fewer volume elements would lead to a coarse-grained logically concurrent description.

Identifying the composite processes of a system is the first, but not the only, step when deriving multicomputer software architectures (see figure 6). Not only must we know *what* the composite processes are, but more important, we must know *how* they interact. Determining the interfaces between the processes is the next most important step.

The interfaces between the processes define the precise format for information exchange. Process A needs input from process B, which might consist of a stream of real numbers, an interrupt, or a binary-encoded number. The interfaces between processes resemble somewhat the argument specifications for a subroutine, function, or procedure.

The inputs and outputs mark the entry and exit points for intermediate results generated by a simulation. They are point-to-point communication paths between processes. If the inputs and outputs are defined, the processes are isolated from each other, and the logically concurrent description of the system is complete. A concise interface definition between communicating processes is essential to executing the transformation from logical to physical concurrency.

The process of designing a multicomputer system begins with the idea to be studied, the environment to be simulated, or the problem to be analyzed, not with the selection of a hardware host. This somewhat radical idea—organizing the software through a logically concurrent description without considering a

continued

Listing 1: *This code illustrates deadlock. Two* SEQ *processes execute simultaneously. Both request input and wait for it, while neither performs any output first. The program hangs.*

```
CHAN OF INT chan1, chan2 :-- channel declarations
PAR
INT A :                     -- local variable scope is
                            -- the first SEQ
SEQ
chan2 ?                     -- input into A on chan2
chan1 ! 6                   -- output 6 on chan1
INT B :                     -- local variable scope is
                            -- the second SEQ (not shared)
SEQ
chan1 ? B                   -- input on chan1 into B
chan2 ! 9                   -- output a 9 on chan2
```

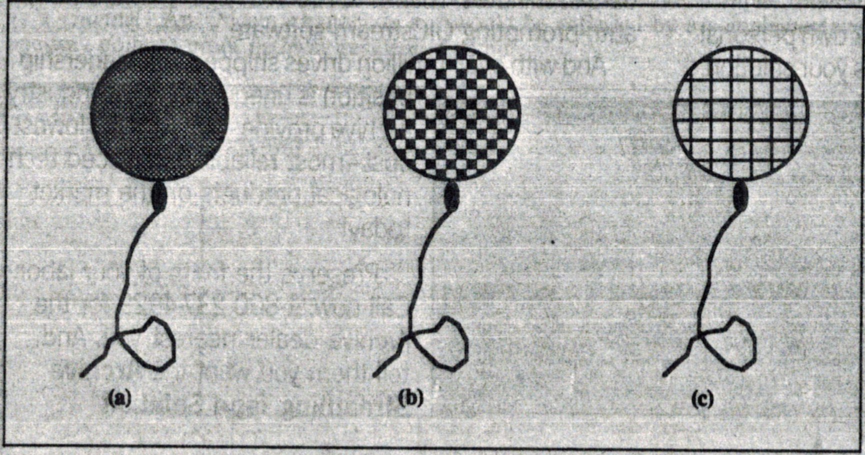


Figure 5: *Treating the molecules in a gas as individual and unique leads to a fine-grained logically concurrent description* **(a)**, *while grouping the gas into small but finite volumes leads to successively coarser descriptions* **(b)** *and* **(c)**.

hardware target for development—is unique to multicomputer software systems.

The end of the design process leads to the construction of a special-purpose computer explicitly organized to execute the software. "Special-purpose" means that the logically concurrent software description, including inputs, outputs, and processes, can now be ported to the physical concurrency of the multicomputer without affecting the software's design or the schedule.

Logical concurrency is used to abstract problem domains into software multicomputer solutions in conjunction with the transputer and Occam: a hardware-and-software team created to facilitate the logical-to-physical transformation.

Transformation Revealed

The transformation from logical concurrency to physical concurrency is the crux of multicomputer development. The speed increase in the algorithms and software is a direct result of this transformation. The entire process of designing transputer-based multicomputer software begins with this assumption: Once you have a logically concurrent description, you can evaluate the software's behavior on a single transputer using soft channels to transfer data between cooperating processes.

This single-transputer implementation is necessary for two reasons: It is unlikely that a "shotgunned" multicomputer software-development cycle (where you "hack" the software out and distribute it among all the nodes) will be successful, and debugging a single-transputer implementation, or any uniprocessor implementation, is easier than debugging software on several processors at once.

The path to physical concurrency starts with observing the logical behavior of the simulation running on a single transputer. The CHAN declarations are the key to performing the mapping. The logical software model, composed of several communicating processes, uses the channels to pass messages. Occam places these channels into the single transputer's address space.

The desire, however, is to achieve physical concurrency, which is accomplished when the logical software model is distributed among the processors according to the software design. With the Occam PLACE construct, you can map the channel addresses to the link addresses. The PLACE construct instructs the compiler to set the address of the predicate at a specific address. For example, the statements

```
INT abcd :
PLACE abcd at #4 :
```

cause the integer variable abcd to be placed at address 4 (hexadecimal).

Likewise, the PLACE construct applies to channels. The transputer's address space has eight specific addresses for the links, and once you PLACE a channel there, it's called a hard channel, instead of a soft channel for memory-to-memory channel I/O. This hard channel then writes or reads information from another process resident on another transputer. There are eight link addresses, four for input links and four for output links, designated link.in0, link.out0, link-.in1, link.out1, and so on.

To complete the logical-to-physical transformation, you must also direct the processes to the appropriate transputer. This task is handled by the INMOS configurer, a postcompilation operation that determines a boot path, along which all the processes will flow toward their target destinations. An example of a tem-

continued

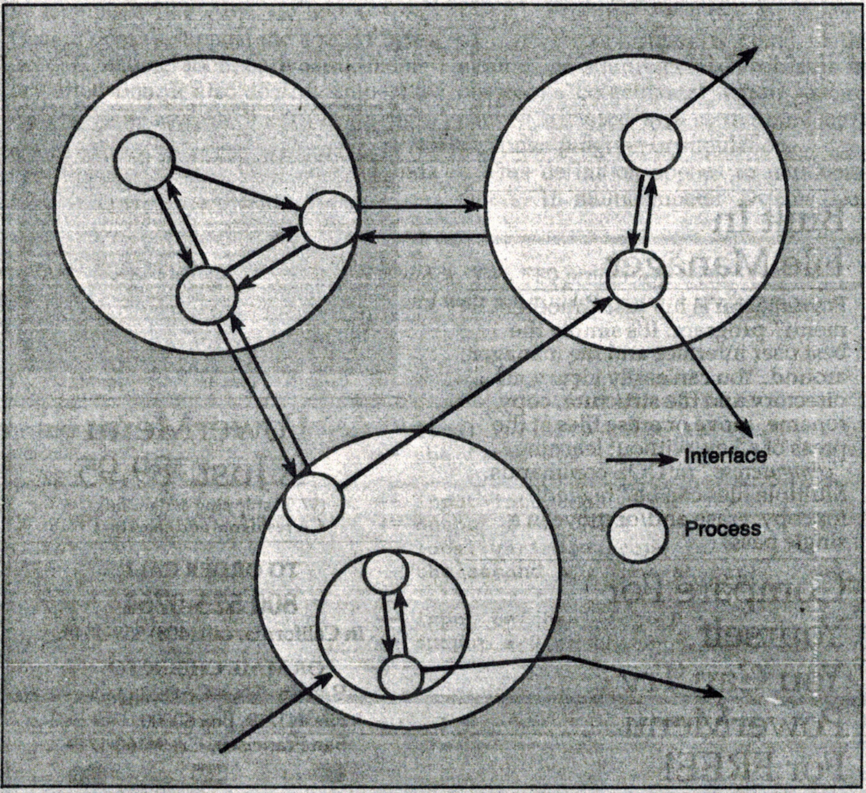


Figure 6: *A process-structure graph. Identifying the composite processes in a multicomputer system is important, but so is isolating the interfaces between processes.*

Listing 2: *A template program. This can serve as input to the configurer.*

```
PLACE chan0.out AT link0.out :
                    -- put chan0.out at hard link0.out
PLACE chan0.in AT link0.in :
                    -- put chan1.in at hard link0.in
PAR                 -- do these processes simultaneously
PROCESSOR 0 T8      -- processor 1 is a T800
navier.stokes()     -- solve the Navier-Stokes equations
PROCESSOR 1 T8      -- processor 2 is also a T800
graphics.output()   -- dump the output (in real time)
:
```

plate program that can serve as input to the configurer is shown in listing 2.

The configurer generates a complete image with the boot path and bootstrap instructions for each node in the multicomputer. The loader PLACEs the processes called navier.stokes() on PROCESSOR 0. The graphics.output() process is PLACEd on PROCESSOR 1. The loader downloads the processes and then begins execution. Communication is synchronized, for the two processes in listing 2 don't know or care whether they read or write from hard channels or soft channels.

Software-Driven

Ideally, a software design should be completely independent of the hardware. The multicomputer system is driven by software, not hardware requirements. However, its success depends on the existence of a suitable hardware host. The INMOS transputer is designed to serve as a multicomputer node.

The innovators and pioneers who elect to invest and pursue multicomputer systems will find an increasing marketplace for this technology. The skills you need to design multicomputer software systems are not radically different from those used in sequential software design. Understanding the Occam language, transputer architecture, and, most of all, logical concurrency are the major requirements. Mostly, however, designing multicomputer software systems depends on creativity, intelligence, and desire. ■

BIBLIOGRAPHY

Athas, William C., and Charles L. Seitz. "Multicomputers: Message-Passing Concurrent Computers." *IEEE Computer*, vol. 21, no. 8, August 1988, pp. 9–24.

Heath, M. T., ed. *Hypercube Multiprocessors*. Philadelphia: SIAM, 1987.

"IMS T800 Architecture." *Technical Note 6*, INMOS Ltd., 1987.

Mackintosh, Allan. "Dr. Atanasoff's Computer." *Scientific American*, August 1988, pp. 90–96.

Packer, Jamie. "Exploiting Concurrency: A Ray Tracing Example." *Technical Note 7*, INMOS Ltd., October 1987.

Pountain, Dick, and David May. "A Tutorial Introduction to Occam Programming." London: Blackwell Scientific Publications, Ltd., 1987.

Richard M. Stein is a software engineer and writer from Irvine, California. He has worked with INMOS transputers for more than 3 years. He can be reached on BIX as "rstein."

People are talking about us.

"This is a very good implementation of FORTRAN, better than the one I use on my minicomputer system." *IEEE Software*

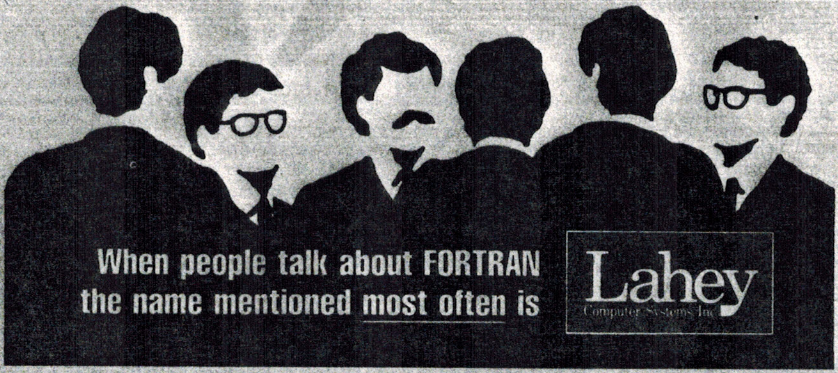
"Lahey F77L is definitely for the programmer's market, with features for the casual and professional user...*EDITOR'S CHOICE" PC Magazine*

"It is a fantastic product." *PC Australia*

"It should be the last FORTRAN compiler you will need to buy..." *Your Computer*

"Lahey F77L was by far the easiest compiler to use..." *Programmer's Journal*

"Compilation speed is Lahey FORTRAN's most remarkable feature; it is unbelievably fast...Realistically, Lahey FORTRAN is the most efficient and productive FORTRAN development tool for the DOS environment..." *Byte*


When people talk about FORTRAN
the name mentioned most often is
Lahey
Computer Systems, Inc.

Contact us to discuss our products and your needs. (800) 548-4778
Lahey Computer Systems, Inc. P.O. Box 6091, Incline Village, NV 89450
Tel: (702) 831-2500 FAX: (702) 831-8123 Tlx: 9102401256

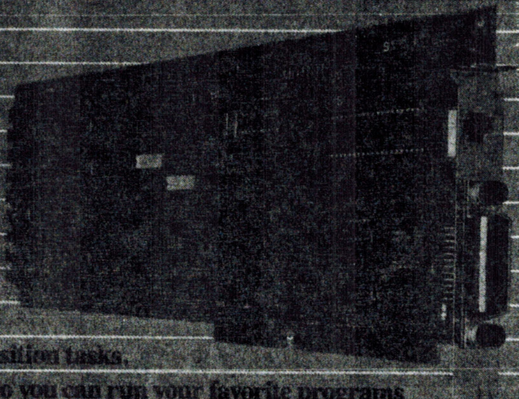
POWER TOOL.

Introducing **4x488**™

You get intelligent IEEE-488 and RS232 ports to make instrument programming fast and easy.

You can have up to 4 Mbytes of memory on the same board for your largest programs, RAM disks, and data acquisition tasks.

Compatibility is built-in so you can run your favorite programs or create new ones with our advanced programming tools.



cec

To get your FREE demo disk—call 617-273-1818.
Capital Equipment Corp.
Burlington, MA 01803

The bottom line—IEEE-488, RS232, par. port, 4MB EEM LIM, runs DOS and OS/2.