KENNETH W. KOONTZ

EMBEDDED PARALLEL ARCHITECTURES IN REAL-TIME APPLICATIONS

Within the past five years, parallel processing has rapidly emerged to the forefront of computer design. The number of processors that can be simultaneously used to work on a single task has grown dramatically. The reduction in physical size and power requirements due to increased levels of logic integration has been just as startling. Embedding a parallel processor into a real-time system to perform a single, dedicated task is now both technically and economically feasible. The potential of embedded, real-time, parallel processors will be explored through an example of an early prototype designed to filter targets from high-clutter noise in a surveillance system.

INTRODUCTION

The idea of inserting a computer into a system to perform a dedicated task repeatedly is about as old as the computer itself. Replacing a piece of special-purpose hardware with a computer provides one great advantage: the computer software can be reprogrammed quickly to fix a problem or to change its algorithm completely. It was not until the microprocessor was invented, however, that embedded computers grew in such numbers. Today, embedded computers are in everything, including military vehicles, commercial appliances, spacecraft, and automobiles.

Embedded computer systems can be classified into two categories: those that operate in real time and those that do not. A real-time embedded computer is normally associated with a process that is time-critical. Tracking and identifying returns from a radar are good examples of real-time applications. An embedded microprocessor in a telephone-answering machine is an example of a non-real-time application; although the phone must be answered, the caller must wait until the answering machine is ready to record the message. A radar return will wait for no one.

As the microprocessor has evolved from a simple 4-b (4-bit) calculator in the early 1970s to a 32-b desktop mainframe in the late 1980s,[1] the size and complexity of tasks that can be performed in real time have increased phenomenally. Problems that used to require special-purpose hardware devices can now be solved with a programmable, embedded microprocessor. Many problems are so computationally demanding, however, that they cannot be computed by a single microprocessor within the time constraints involved. For such problems, the choices have been to build special-purpose hardware to perform the required processing, or to stop the project and let the problem await future technology.

Within the past five years, some of that future technology has come of age in parallel processing. A parallel processor provides many similar computing modules (or processors) within a single enclosure. Each processor is given a small portion of the overall problem to solve. If one processor can execute a job in time T, then P processors can, ideally, execute the same job in time T/P. The basic idea behind parallel processing has been around since the early 1960s, but the maximum value for P of those days was limited because of the level of logic-device integration; 64 processors could be tied together if one had a large enough room and bank account.[2-4]

Today, advances in very large scale integration (VLSI) have reduced the physical size of a computing module and, in so doing, have increased P. A value of $P = 100$ is quite common today, and some machines with $P = 65{,}536$ have been built![5] Various architectures have been created to interconnect and control such large numbers of processors as a single entity (see the appendix). Some architectures are quite flexible, allowing variable numbers of processors to be incorporated. With the increased use of high-speed complementary metal oxide semiconductors (CMOS), computer systems that once required special environments and cooling techniques can now be placed in harsher environments and cooled with forced ambient air.

As a result, real-time problems that previously had to be implemented in hardware or abandoned can now be implemented in an embedded parallel processor. Although the cost is much more than the single-microprocessor solution, it is still much less than the cost of designing and constructing special hardware. The parallel processor is not a panacea, however; many areas still require special-purpose hardware. But the amount of this hardware and the time and cost required to implement it can be significantly reduced if used in conjunction with a parallel computer architecture.

By presenting an example of an early prototype, this article explores some of the issues involved in embedding a parallel processor into a real-time system. The prototype implements a unique, backward-searching, nonrecursive algorithm called the retrospective filter, or "retro," which is useful for extracting targets from high-clutter noise in a radar installation. The parallel architecture used is as unique as the algorithm; a special microprocessor called the "transputer" was used to provide a collection of expandable computing modules. Before describing the transputer and our example application, some problems associated with programming real-time systems are presented. Many of the ideas introduced in the next section are later applied to the retro example.

REAL-TIME CONSIDERATIONS

The developer of an embedded real-time computer system faces several issues,[6] and when the computer system involved is a parallel processor, some additional considerations must be addressed. In a real-time system, the processing requirements of a computer are directly related to the input rate of raw data to be processed. If, for example, a new piece of input data is available on average every 4 ms, then the average time to process the data (T_p) must not extend past 4 ms or the processing will fall behind. This input/processing restriction is referred to as the real-time period (T_{rt}).

The transfer rate of input/output channels is also tied to T_{rt}. All input data to be processed must be transferred into the computing module within T_{rt}; additionally, all resulting output data must be transferred out of the computing module within T_{rt}. These two restrictions on input time (T_{in}) and output time (T_{out}) assume that input, output, and processing of data can occur simultaneously. In addition, little computing overhead should be required to perform this concurrency.

To perform the concurrent input/output and processing, the architecture must be capable of multitasking, which allows more than one thread of execution in a program to occur; a good analogy is a multiuser operating system in which every user is a different task in a complex program. One of the most important requirements in a multitasking system is its task-switching time, which is the time required to stop execution at one thread and start execution at another. In a real-time system, if the time required to switch tasks is a significant percentage of T_{rt}, then considerable time will be wasted. Ideally, the task-switching time should be 1% or less of T_{rt}.

Latency is another important consideration. The latency through a system is the time from when the sensor detects the signal to when the result of processing is available. For a single processing stage, the latency ($T_{latency}$) is just the sum of T_{in}, T_{out}, and T_p. If several stages of processing are performed by different computing modules, then the total latency will be a function of the transfer and processing rates through each stage. Latency requirements are very system-dependent. For instance, a robotic-arm control system will require a small latency on the order of milliseconds, whereas a surface-ship surveillance system can withstand latency on the order of seconds.

Use of a parallel processor in an embedded real-time system adds two more considerations to the list. The primary one is load balancing, which provides a means to distribute the computing task evenly among all of the processors present. A good load-balancing algorithm is imperative if all of the processing power available is to be fully used. Otherwise, one processor may be provided with too much work while all others remain idle. The other consideration is scalableness. Ideally, the number of processors provided should be scalable with the requirements of the system. If the requirement on maximum input rate should increase, then the number of processors required in the system should be allowed to increase with few restrictions. The change in the number of processors should be made easily, either at compile time or run time. At compile time, a constant representing the number of processors can be increased; at run time, the number of processors can be dynamically detected before the program is loaded.

Theoretically, if the input rate increases linearly, the number of processors should scale linearly, too. Linear scaling does not always occur, however, because the application's algorithm may not scale linearly; for example, the retro scales to a squared term.

THE TRANSPUTER

The transputer, developed by Inmos Limited of Bristol, England, in 1984, is the generic name of a family of devices designed for constructing parallel-processing systems based on the multiple-instruction multiple-data (MIMD) loosely coupled architecture[7,8] (see the appendix). The transputer resulted from a British government initiative to compete with the United States and Japan in supercomputer and fifth-generation architectures. The name "transputer" is derived from the combination of the words transistor and computer. Developers at Inmos believe the transputer (or transputer-like devices) will become the basic building block of parallel processors in much the same way that transistors became the basic building block of today's semiconductor technology.

A transputer is a complete, single-chip, computing module. A single transputer contains a central processor, on-chip local memory, programmable external-memory interface, optional special-purpose peripheral interfaces, and high-speed serial communications channels called links. Transputers are interconnected with these links; each link provides a full-duplex, point-to-point communications path between two transputers. Transputers can be interconnected in a variety of networks to satisfy the requirements of a given application. One article on transputers aptly defined them as "computer Lego."[9]

To help program the transputer, Inmos developed a language called "occam."[10] Occam allows a programmer to define a problem as a set of concurrent processes interconnected by channels. Processes communicate through message exchanges over these channels. A concurrent program simply becomes a collection of processes, channel interconnections, and message protocols. An abstract occam program can then be assigned to execute on a transputer array through a mapping operation that assigns processes to transputers and channels to links.

Selection of the transputer for these investigations was based on price, availability, and ease of interconnection. A single transputer module consisting of the transputer plus external memory costs about $1000. Boards containing from one to eight transputers, each with 32 KB to 2 MB of local memory, are available from many vendors (Fig. 1). A variety of networks can be created simply by interconnecting links into the desired topology.

The transputer is also ideal as a computing module for embedded real-time parallel processing, particularly when applied to signal processing and conditioning.[11-14] Because it is so highly integrated, it requires little physical space; hundreds of transputers can be easily inserted into a system. The CMOS logic of the transputer allows it to run fast while using low power; a single transputer running at 20 MHz requires only 100 mA at 5 V. Thus, many transputers can be powered by a conventional switching power supply and cooled with inexpensive forced air.

The internal architecture of a transputer can be described by exploring the features of the standard, floating-point transputer, the T800 (Fig. 2). In the following sections, keep in mind that all of these features are provided in a single 84-pin CMOS integrated circuit.

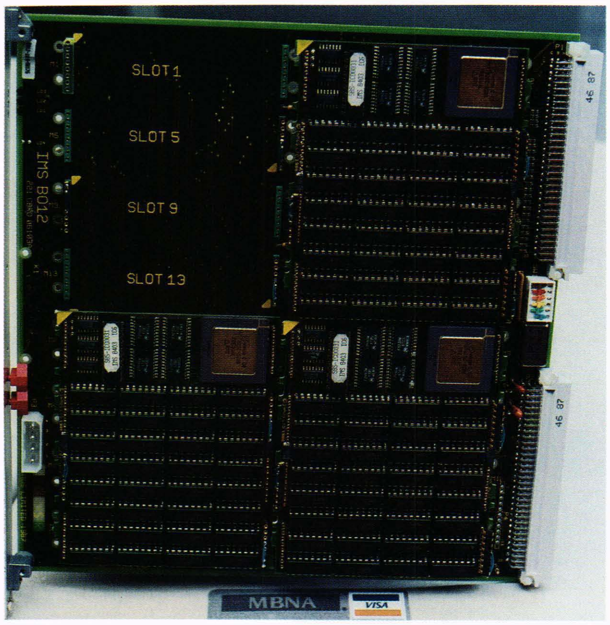


Figure 1. Board configured with four transputers, each with 1 MB of local memory. Boards containing multiple transputers are available off the shelf from a variety of vendors.

Processor

The T800 contains two processors: the scalar processor (called the 32-b processor) and the floating-point unit. Each processor is based on a stack architecture that operates in a manner similar to a Hewlett-Packard calculator using postfix notation for ordering operands and the operator (e.g., adding A and B is AB+). Both processors can operate concurrently, allowing the 32-b processor to compute an address for an operand while the floating-point unit performs a floating-point operation. For a 20-MHz device, the 32-b processor is rated at 10 million instructions per second; the floating-point unit is rated at 1.5 million floating-point instructions per second for single-precision (32-b) operations. A single T800 is roughly equivalent to a DEC VAX 8600 in computational performance or 2.5 times faster than an AN/UYK-43, a common embedded military computer.

The instruction set is encoded in a reduced-instruction-set computer (RISC) format. The basic instruction is 1 B long and contains 4 b of operand and 4 b of data. Thus, 16 basic instructions are available. Three of the basic instructions allow multiple operand fields to be constructed to extend the instruction set; the other thirteen support the most common instructions found in a program (e.g., load, store, jump, add constant, and test). Each basic instruction executes in a single cycle. For a 20-MHz processor, this cycle is 50 ns.

Additional instructions and registers are provided to handle multitasking at the hardware level. A set of registers is provided to maintain two queues of processes ready to execute on the 32-b processor. These registers support a two-level, preemptive, time-sharing operating environment. Two timers and instructions for interprocess communications are provided to allow processes to wait until a future time is reached or until an input/output operation is complete. This hardware support for multitasking reduces or eliminates the requirement for a run-time executive, as is traditionally required by other microprocessors. With hardware support providing task-switching times on the order of microseconds and not milliseconds, the software engineer can freely specify concurrency in a program without worrying about performance penalties and the relation of task-switching times to a real-time processing requirement.

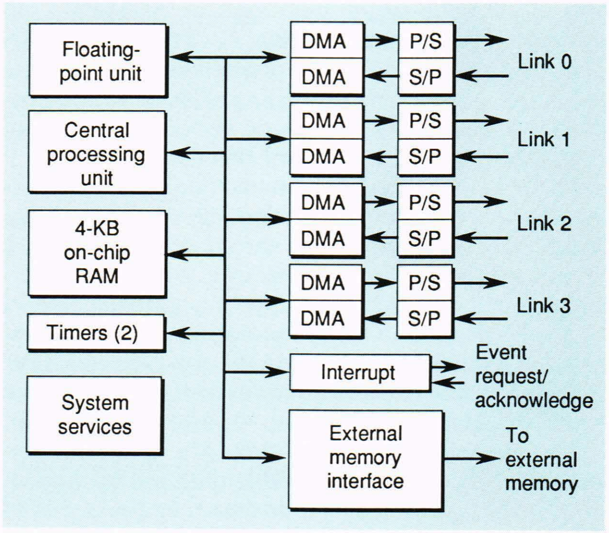


Figure 2. Block diagram of a T800 transputer. The T800 contains a scalar-processing floating-point unit, on-chip memory, external-memory interface, and serial-link interfaces, all on a single chip. In this figure, RAM is random-access memory, DMA is direct memory access, P/S is parallel-to-serial converter, and S/P is serial-to-parallel converter.

Memory

A small (4-KB) memory is provided on-chip and is configured as conventional random-access memory. The on-chip memory supplements the three stack registers in the processor by providing storage for intermediate results; its access time approaches that of the registers. Normally, high-level language compilers use this area as stack space for dynamic variables. The user may also place frequently accessed global variables or small buffers in this memory to reduce load/store times. If the application code is small enough, the user's code may also be placed in this memory.

To supplement the on-chip memory, a programmable, 32-b, external-memory interface is available to provide a full 32-b address and data bus. This interface can expand the memory size of the T800 to 4 GB. Both external memory and memory-mapped peripheral devices can be mapped into this external-memory space in a fashion similar to other microprocessors. The memory interface is programmable, however, and thus allows the user to connect either static or dynamic memory devices of various speeds. Dynamic memory refresh is provided automatically by the interface. To connect the T800 to an external memory, a few latches are all that is normally required.

Links

Four link interfaces are provided to interconnect with other transputers. Each link interface provides a full-duplex, asynchronous, 20-Mb/s serial channel. For large buffers (500 B and up), the link throughput is 1.75 MB/s in one direction (unidirectional) and 2.35 MB/s if transferring in both directions (bidirectional). Each link is supported by two direct memory access controllers, a parallel-to-serial converter, and a serial-to-parallel converter. All eight direct memory access units can operate concurrently with processor instruction and data fetches.

Although the link speeds are high for a serially encoded data stream, they are low when compared with other interconnection media (e.g., the shared bus). This transfer rate is perceived by most technology spectators to be the main disadvantage of the architecture, although it is the chief advantage. As processors are added to a system using a shared-bus network, the requirement for connections (and throughput) between the processors grows, but the available bandwidth of the bus remains fixed. Eventually, a limit is reached as a result of the bus's throughput and its arbitration mechanism. The addition of more processors will not decrease the execution time for a task. Because a link is a dedicated connection between two transputers and not a shared resource, the addition of more transputers to a system not only increases processing power but also adds to the total aggregate throughput of the interconnection network. For example, a network of 50 transputers using all four links has an aggregate transfer potential of 940 MB/s. If 100 transputers are used, the aggregate increases to 1820 MB/s.

Besides providing the main facility for intertransputer communications, the links are used to load the parallel program. On power-up or reset, a transputer remains idle, listening for a message to arrive over one of its links. When a message arrives, it is loaded into a specified memory location and is then executed. The program contained in this first message sends copies of itself to the other links (minus the one from which the message was received) to initialize all other transputers in the network. Because the size of the first message is limited, the loading process proceeds in stages (e.g., loading and executing a simple bootstrap program, followed by loading and executing the application processes for each transputer).

This link function eliminates the need for read-only memory in each computing module. During software development, a host computer outputs a file from a secondary storage device through an adapter to one of the root node's links. For embedded operation, this file can be stored in read-only memory along with a small loader program and placed in a specially configured root node. Using the read-only memory only as a storage device allows the program to execute faster (the processor routinely fetches instructions or data not from read-only memory but from random-access memory). Additionally, when a program change is required, only one set of read-only memory has to be replaced. Initialization across links is indispensable for software development on large parallel systems.

THE PARALLEL RETRO

The retro algorithm was developed at APL in the late 1970s by Richard J. Prengaman.[15] It is a limiting filter that operates by examining all possible trajectories of a target in a backward-searching, or retrospective, manner. Only those targets that have reasonable trajectories (e.g., straight line at constant velocity) are allowed to pass through the filter to an automated tracking system. The retro is also an estimation filter; it can provide initial estimates of velocity that can then be refined by a tracking algorithm. Use of the retro can significantly improve the performance of a surveillance system, particularly in high-clutter environments.

The retro has been used successfully in various situations with many different sensors. It has had only limited application, however, because of its computational demands; the amount of processing required grows as the square of the amount of input data. Early implementations concentrated on the surveillance of slow-moving targets (e.g., surface ships, boats, and helicopters). We have found that the velocity limit is a function not of the algorithm but of the sequential architecture in which it was implemented. By applying parallel-processing techniques, we have shown that the upper limit on velocity can be almost eliminated, making the algorithm suitable for a broader class of applications.

Algorithm Basics and Computational Requirements

The algorithm works by extracting a signal (S) from a combination of signal and noise [$E \in \{S(x,y,z,t), N(t)\}$] through permutations on all events recorded in the retrospective memory during some past time period. A simple predictor of signal behavior (e.g., straight path at a constant velocity) is used as the kernel algorithm for the permutations. Every new event is compared with all stored previous events to develop a velocity profile representing all potential trajectories of a target. If the predictor of signal behavior is correct and old signals from the target are present, then the profile will contain a peak that can be detected and compared with a threshold. If the peak is larger than the threshold, the event is classified as a target contact and sent out of the filter for further processing. The event is then stored in the retrospective memory for comparison with future events, providing the basis for the retrospective search (Fig. 3).

The efficiency of this algorithm in comparison with more traditional recursive (e.g., Kalman filtering) approaches was discussed by Bath.[16] In summary, for large densities of extraneous events, the nonrecursive solution produces fewer ambiguities than the recursive solution. By implementing a two-stage model using nonrecursive and recursive filters, the advantages of both approaches can be combined,[17] although the approach requires an enormous number of computations.

The ideal retro algorithm increases at a rate of $O(n^2)$, where O means on the order of and n is the number of past returns stored. To illustrate, consider the parameters for a hypothetical radar with a 4-s scan rate. With an input rate of 250 events per second, a radar with a 4-s scan rate will record 8000 events over 32 s or eight scans (the filter's depth). Events in the same scan are not permuted with one another but only with the other seven scans. Therefore, each new event must potentially be permuted with 7000 previous events. With an input rate of 250 events per second, each permutation must be completed within 4 ms for the filter to keep up with real time. A total of 1,750,000 comparisons must be performed every second! If the input rate is increased to 500 events per second, 14,000 comparisons must be performed in 2 ms for a total of 7,000,000 comparisons per second (doubling of the input rate quadruples the number of computations).

Limiting the Number of Correlations

Past implementations relied on traditional sequential computer architectures to perform these calculations (e.g., a single Motorola 68020), but the number of kernel operations required by the retro algorithm far exceeds the abilities of current sequential machines. Various techniques have been used to reduce the number of comparisons so that the algorithm may be implemented on these architectures.

For example, one technique divides the retrospective memory into a series of range and bearing sectors of roughly equal areas. A linked list of events is maintained for every sector. When a new event is available, an assumption is immediately imposed on its maximum velocity before any calculations are performed. By using this maximum velocity, the scan rate of the radar, and the retrospective memory's depth, a search envelope is defined. Past events lying within sectors partially or completely contained by this envelope are then used for comparison. This technique has the advantage of substantially reducing the number of comparisons required.

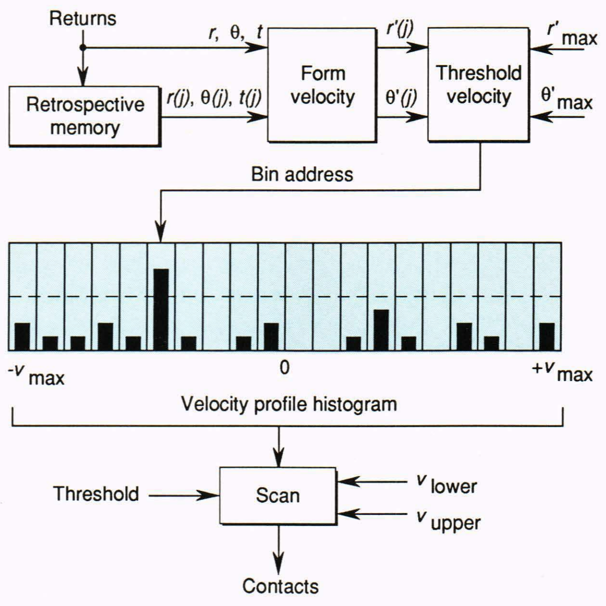


Figure 3. Block diagram of a retro, where r is range, θ is bearing, t is time, and v is velocity.

Only those events within a specified distance from the new event are used. Events outside the envelope are, for the most part, ignored.

This technique has two distinct disadvantages. If one wants to track faster targets, the area of the envelope grows at a rate of $O(n^2)$, where n is either the radius of the search envelope or the number of past events used for comparison. A larger envelope encompasses more events and increases processing requirements. The maximum radius of the envelope must therefore be carefully selected so that the sequential processor can perform the task. The second disadvantage occurs when the events are not uniformly dispersed over the sectors. If a large number of events occur in a small number of sectors, the number of comparisons has not been reduced; the processor will not finish the correlation before the next event is input by the radar. This algorithm is thus prone to overflow.

Locating the Parallelism

Several ways of implementing the retro in many transputers were tried. Some approaches mirrored the use of the linked lists, assigning a single processor to each range/bearing sector. These approaches suffered from a severe load-balancing problem; too many events in one sector would cause one processor to be overloaded while others were idle. Assigning one processor to a scan was also attempted, but this approach limited the maximum number of processors to the filter's depth. In addition, the use of spatial neighborhoods, not fixed as in the sectors but dynamically changing, was envisioned, but the method of assigning neighborhoods and maintaining nonoverlapping regions was complicated and potentially very time-consuming.

The approach eventually implemented incorporates the neighborhood idea but uses time rather than location as its dimension. Instead of assigning events to a processor with a criterion based on physical orientation (e.g., sectors), new events are assigned to processors containing previous events that have just occurred (i.e., time neighborhoods). A storage threshold is also used for load balancing to guarantee that a processor can always finish all computations before the next event is available. Thus, event storage is controlled by processing ability and current processor use.

The network used to interconnect transputers is a simple pipeline with an end-around connection (Fig. 4). All transputers within the pipeline execute a copy of the parallel retro kernel. The end transputers require a slightly modified version to handle the end-around channel. A preprocessor interfaces to the front of the pipeline to provide input data; a postprocessor interfaces to the rear to scan for peaks in the velocity profiles. Both preprocessors and postprocessors are also implemented in transputers.

Cooperative Permutations

Each transputer within the pipeline stores a number of past events N. It must be emphasized that each transputer contains only a fraction of the total number of events stored in the retrospective memory. The sum of all events in all transputers corresponds to the complete retrospective memory in the sequential version. A brief description of how these events are stored is given in the next section. For this discussion, however, assume that each processor stores the same number of past returns.

When the sensor detects a new event, a return message consisting of range and bearing fields is sent to the preprocessor. The preprocessor appends the return with an initialized velocity profile histogram. The histogram is then transmitted to the first transputer in the pipeline. The transputer compares this event with its N locally stored events and updates the bins and velocity profiles in the histogram corresponding to the computed radial and angular velocities. After all N events have been correlated, the histogram is transmitted to the second transputer, which then proceeds to compare this new event with its N stored events. This process continues down the pipeline, forming a computational wavefront.[18,19] After the histogram is processed by the last transputer, it represents the complete velocity profile of this event compared with all past events stored in all processors. As a final step, the postprocessor scans the histogram, assigning a quality to each velocity profile. The highest quality found is compared with a threshold. If this quality is greater than or equal to the threshold, a contact message is issued by the filter.

To allow simultaneous input/output and processing, three processes are multitasked in each transputer. Two input/output processes operate at high priority, interrupting the retro process when data must be transferred. These processes require very little overhead because their main function is to initialize the direct memory access units to send or receive a message. Use of a pipeline allows many events and their histograms to be in various stages of construction at any time. For example, while transputer T_1 is comparing event e_i, transputer T_0 is comparing event e_{i+2} and transputer T_2 is comparing event e_{i-2}. Event e_{i+1} is in transit from transputer T_0 to T_1; e_{i-1} is in transit from transputer T_1 to T_2.

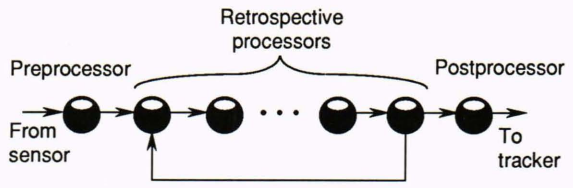


Figure 4. Network topology for transputers used in the parallel retro algorithm.

Memory Management Through Time Neighborhoods

The events in each transputer are stored in a set of time neighborhoods. When used with a scanning radar, one neighborhood corresponds to one full scan. Each neighborhood is implemented as a single threaded linked list. A set of pointers and a counter are used to implement the neighborhood. Two additional pointers are used to mark the current neighborhood and the oldest neighborhood, and yet another counter is used to keep track of the number of events stored in all neighborhoods. Figure 5 illustrates the basic data structure.

The distributed memory is maintained by using a storage token and a memory threshold. The presence of the storage token in a processor causes the event fields in a histogram message to be stored in the local retrospective memory at the end of the current neighborhood. The memory threshold limits the number of events that any one processor can store and is the basis for load balancing in the system. When a processor reaches the limit set by the memory threshold, it assigns the storage token to the next processor in the pipeline. The storage token thus travels around the ring, determining which processor has available processing resources to perform permutations against this event. Additional housekeeping devices periodically purge the oldest scan of events stored by all processors.

In certain environments, the number of events input by the radar may exceed the total storage and processing capacity of all processors. Previous sequential-architecture implementations dealt with this problem by purging the oldest scan of returns early. Then, the retro operated in a degraded mode using one less scan of data. If the environment became more severe, more than one scan might be deleted early. This condition is handled in the parallel algorithm by forcing a processor receiving a storage token to purge the oldest neighborhood if the storage threshold has already been reached. Thus, degradation is limited to a neighborhood being deleted (which is normally a small portion of a scan) and not the entire oldest scan.

Load Balancing

One of the most important aspects of parallel algorithms is load balancing, which helps maintain similar computational loads on all computing modules of

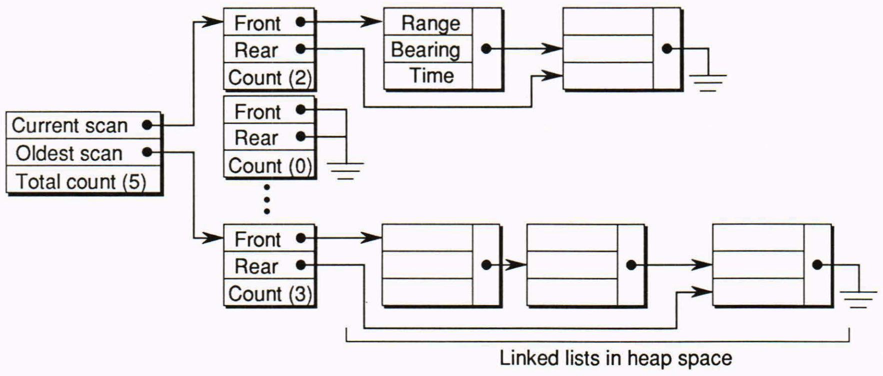


Figure 5. Data structure for time neighborhoods.

a parallel architecture. Without load balancing, one computing module may become overburdened with computations while all others remain idle, destroying the advantages of using a parallel architecture.

As previously mentioned, load balancing is implemented within the memory-management algorithm. The total number of events N that each processor maintains is used as a threshold on the processor's peak processing ability; N is selected so that the processor can maintain real time (T_{rt}) without falling behind. Note that N is a function of the processing power available and not the size of the processor's local memory.

As an example of how N is selected, consider that the parallel algorithm is first implemented on a single processor. A calibration test is then performed to determine the relationship between processor computation time T_p and the number of events N stored in the local retrospective memory. The resulting data for a 20-MHz T414 transputer used in early implementations are given as follows:

Number of returns	Correlation time (ms)
1	0.068
2	0.082
5	0.123
10	0.192
20	0.335
50	0.755
100	1.457
200	2.858
500	7.062

The next step is to select the input rate of return from the radar, f_{in}. A typical value for f_{in} is 250 returns per second. Thus, T_{rt} ($= 1/f_{in}$) is found to be 4×10^{-3} s. We also need the maximum number of events N_{total} that will be stored in the retrospective memory across all processors:

$$N_{total} = T_{scan} \times f_{in} \times D \ , \qquad (1)$$

where T_{scan} is the scanning period of the radar and D is the number of scans of data stored in the retrospective memory (the filter's depth).

To calculate a rough estimate of N, we can use the inequality

$$T_p \leq T_{rt} \ . \qquad (2)$$

Interpolating the data given in the preceding paragraph for the 20-MHz T414 transputer, we find that for $T_{rt} = 4 \times 10^{-3}$ s, $N = 281$ returns. Thus, if a processor is forced never to store more than 281 returns, it will always maintain the real-time input requirement. Given N, the total number of processors required in the pipeline P can be determined as

$$P = \lceil N_{total}/N \rceil \ . \qquad (3)$$

The ceiling function is required because a nonzero remainder increases the number of processors by one. In this case, one of the processors will be slightly underutilized. For the hypothetical installation discussed earlier, $N_{total} = 8000$, so $P = 29$.

Tuning with Input/Output Considerations

To determine a more refined value for N, the input/output transfer rate of the channels interconnecting processors must be considered. Real time affects not only the number of computations each processor can perform but also the size and number of messages that can be transferred. To determine a refined value of N, the input/output capacity of the parallel processor must be analyzed.

To maximize the concurrency of input/output and processing, three processes are multitasked: two processes perform input and output on the pipeline, and the other process executes the kernel algorithm. This arrangement allows three histograms to be active in a processor: two in transit between stages of the pipeline and one in the process of being modified by the retro kernel.

Two additional memory-to-memory transfers must be performed: one to move results from the kernel to the output buffer for transfer to the next pipeline stage, and one to move new data from the input buffer into the kernel's buffer. Thus, the input/output transfer rates of the external channels connecting the processors (T_{ext}) and of the internal channels connecting the buffers (T_{int}) must be measured. These rates are functions of the hardware implementation and can be measured by using a simple benchmark program. The results for a 20-MHz T414 using a 10-Mb/s external channel and buffer memory mostly in internal 50-ns random-access memory are presented in Tables 1 and 2.

To satisfy the constraints imposed by T_{rt}, the following input/output inequalities must be met (Eq. 4 is for the buffering processes, and Eq. 5 is for the kernel process):

$$T_{ext} + T_{int} \leq T_{rt} \ , \tag{4}$$

$$T_p + 2 \times T_{int} \leq T_{rt} \ . \tag{5}$$

Use of the latter inequality instead of Equation 3 reduces the value of N by a small amount. Using our previous example, we find that $N = 278$ instead of 281; P remains unchanged.

Finally, the maximum latency through the pipeline, $T_{latency}$, can be determined by

$$T_{latency} = P \times (T_{ext} + 2 \times T_{int} + T_p) + T_{ext} \ . \tag{6}$$

Table 1. Link-to-memory transfer rates of the T414 10-Mb/s link.

Message size (B)	Transfer time (ms)	Effective throughput (MB/s)
20	0.042	0.476
40	0.079	0.506
60	0.117	0.513
80	0.153	0.523
100	0.190	0.526
200	0.375	0.533
400	0.746	0.536
600	1.115	0.538
800	1.485	0.539
1000	1.855	0.539

Table 2. Memory-to-memory transfer rates of the T414 internal (50-ns) memory.

Message size (B)	Transfer time (ms)	Effective throughput (MB/s)
20	0.015	1.333
40	0.016	2.500
60	0.016	3.750
80	0.017	4.706
100	0.017	5.882
200	0.020	10.000
400	0.025	16.000
600	0.030	20.000
800	0.035	22.857
1000	0.050	20.000

Note that this delay is the maximum possible. If processors store fewer than N events, then the value of T_p will be less than T_{rt}. For our example using 29 processors, $T_{latency} = 1.3 \times 10^{-1}$ s.

If the value of $T_{latency}$ introduced is deemed excessive for the application, the delay can be reduced by connecting the processors with parallel pipelines. Figure 6 illustrates a topology using four pipelines. The preprocessor has been modified to transmit all histograms to the four pipelines concurrently. The postprocessor then combines all four histograms from the same return before performing target detection. Each pipeline has a front and back node with an extra channel. The back of each pipeline is connected to the front of the next to allow the storage token and unstored returns to circulate through a ring, as in the single pipeline.

If x parallel pipelines with y processors in each are used, then $T_{latency}$ can be reduced to approximately $T_{latency}/x$ when $y \gg 1$. As y approaches 1, the increased processing at the postprocessor may degrade performance over a configuration with fewer pipelines. Additionally, fan-in and fan-out may be constrained by the hardware implementation; for example, four pipelines are more easily handled in a transputer implementation than five pipelines. (Fan-in refers to the number of links coming into a unit, and fan-out refers to the number of links emanating from a unit.)

Implementation

The algorithm described in this article has been successfully implemented and demonstrated to be practical with an array of 34 T414 transputers (Fig. 7). Thirty transputers were assigned to perform the parallel retro algorithm, one was used as a postprocessor to scan for peaks in the finalized histogram, and another simulated a radar, producing signals interlaced with uniformly distributed noise. The other two transputers provided interfaces to the user: one interfaced to a terminal, and the other produced a color graphics display. The total processing power provided by the array was 340 million instructions per second.

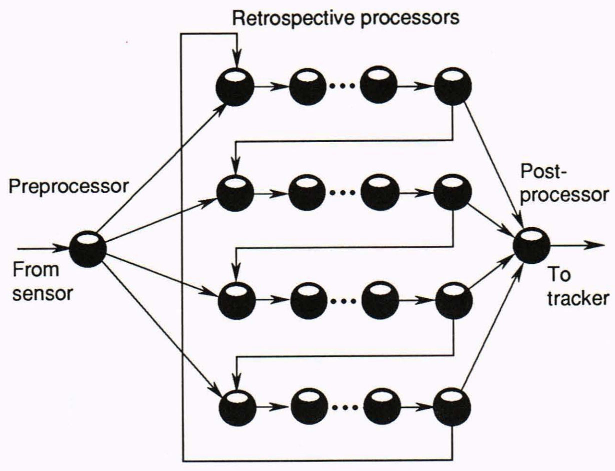


Figure 6. Parallel pipelines that can be used to reduce latency.

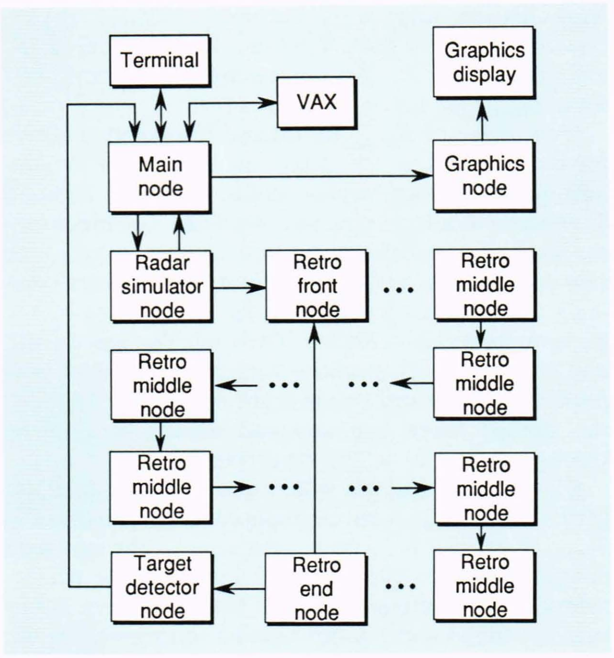


Figure 7. Configuration of all transputer processes executing both the retro and supporting functions.

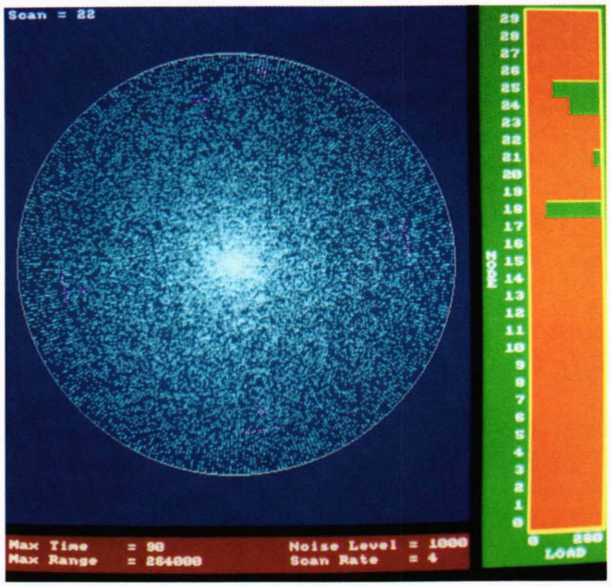


Figure 8. Photograph of "retrospection in parallel" graphics plot. This test scenario injected 250 noise events per second over a signal containing 20 tracks. Noise is displayed in light blue, and tracks from the filter are shown in pink. The processor load graph reveals that all 30 processors were maintaining near-maximum event loads.

Many simulated scenarios were tested. Figure 8 shows the graphics display from one of the tests. The region in the upper-left corner of the screen is a plan position indicator of all events, both signal and noise, from the entire test. The blue pixels represent noise, and the red pixels are signals identified by the filter. The region on the right is a load graph of the number of events stored per processor and thus the computational load for each transputer executing the retro kernel. The region in the lower left shows some of the scenario's parameters.

PREDICTIONS FOR THE FUTURE

During the two years of this investigation, the transputer matured from the T414 to the T800. The T800 provided an on-chip, floating-point unit that improved floating-point computational rates by 20-fold. More on-chip memory was added (from 2 KB to 4 KB) to increase access to critical data and code. The link protocols were also refined, resulting in a doubling of the throughput rate.

Since then, transputer technology has advanced primarily in software and network areas. Originally, all programming had to be performed in occam; today, compilers exist for almost any common programming language (e.g., Fortran, Pascal, C, Basic, Modula-2, Prolog, Lisp, Forth, and Ada). Software tools are also emerging to provide better debugging and load-balancing support. Networks, originally configured through hardware jumpers on a backplane, can now be configured by software through the use of link crossbar switches. Static networks, which specify the same topology for the life of a program, are being augmented by quasi-static networks that provide the optimal topology for different phases of a program (e.g., raw data are loaded by using a tree, intermediate results are exchanged by using a toroid, and results are unloaded by using a tree). Dynamic networks, which allow hundreds to thousands of direct connections between transputers to be created and destroyed per second, are also being investigated.

Within the near future, embedded parallel processing will find continued acceptance. The hardware technology will continue to increase computational rates and input/output bandwidths by at least an order of magnitude; VLSI and ultra large scale integrated circuits containing 10 to 100 processing elements on a single chip will become more commonplace. Whereas architectures containing a total of 10 to 100 processing elements are common today, 1000 to 10,000 elements will be common tomorrow.

To assist people in using this massive computing asset, computer-assisted software engineering tools will be developed to perform functions that are currently done by hand. Many of these tools will evolve from computer-aided engineering tools used for circuit layout and simulation because many operations in parallel-program development are similar to circuit design (e.g., mapping processes and channels to processors is very similar to routing circuit connections between chips to produce a printed circuit board). These new tools will provide insight into locations of parallelism within a sequential program, suggest usable network topologies, help configure the program for execution on the hardware, and fine-tune the application during run time.

Application areas for parallel architectures will continue to appear. Scientists and engineers will use parallel processors to solve today's problems in record time and to provide the solutions to problems numerically unsolvable today. Embedded systems using parallel processing will appear in all types of military platforms (e.g., surface, subsurface, airborne, and spaceborne). We are just beginning to understand and select application areas for parallel processing that can use the processing rates available and provide system solutions where traditional techniques are limited.

REFERENCES

[1] Garetz, M., "Evolution of the Microprocessor: An Informal History," *BYTE* **10**, 209-215 (1985).

[2] Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L., et al., "The ILLIAC IV Computer," *IEEE Trans. Comput.* **C-17**, 746-757 (1968).

[3] Texas Instruments, Inc., *Description of the ASC System: Parts 1 to 5*, Manual Nos. 934,552 to 934,666, Dallas, Tex. (1971).

[4] Wulf, W. A., and Bell, C. G., "C.mmp—A Multi-minicomputer," in *Proc. AFIPS Fall Joint Comput. Conf.*, Anaheim, Calif., Vol. 41, pp. 765-777 (1972).

[5] Tucker, L. W., and Robertson, G. G., "Architecture and Application of the Connection Machine," *IEEE Comput.* **21**, 26-39 (1988).

[6] Stankovic, J. A., "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Comput.* **21**, 10-19 (1988).

[7] Whitby-Strevens, C., "The Transputer," *IEEE SIGARCH Newsletter* **13**, 292-300 (1985).

[8] Walker, P., "The Transputer," *BYTE* **5**, 219-235 (1985).

[9] Stevens, M., "Transputers: LEGO for Fifth Generation Machines," *Comput. Bull.* **2**, 21-23 (1984).

[10] Inmos Limited, *Occam 2 Reference Manual*, Prentice Hall, London, p. 133 (1988).

[11] Taylor, R., "Signal Processing with Occam and the Transputer," *IEE Proc.* **131**, 610-614 (1984).

[12] Harp, J. G., Roberts, J. B. G., and Ward, J. S., "Signal Processing with Transputer Arrays (TRAPS)," *Comput. Phys. Commun.* **37**, 77-86 (1984).

[13] Wilson, P., "Digital Signal Processing with the IMST424 Transputer," in *Proc. WESCON '84*, London, pp. 4/7/1-12 (1984).

[14] Cok, R., "A Medium-Grained Parallel Computer for Image Processing," in *Proc. 8th Occam User Group Tech. Meeting*, Sheffield, U.K., pp. 113-123 (1988).

[15] Prengaman, R. J., Thurber, R. E., and Bath, W. G., "A Retrospective Detection Algorithm for Enhanced Target Acquisition in Clutter and Interference Environments," in *Proc. 1982 IEE Int. Radar Conf.*, London, pp. 341-345 (1982).

[16] Bath, W. G. "Evolution and Challenges in Signal Processing," *Johns Hopkins APL Tech. Dig.* **9**, 261-268 (1988).

[17] Bath, W. G., Baldwin, M. E., and Stickey, W. D., "Cascaded Spatial Temporal Correlation Processes for Dense Contact Environments," *Proc. 1987 IEE Int. Radar Conf.*, London, pp. 125-129 (1987).

[18] Kung, S. Y., Arun, K. S., Galezer, R. J., and Rao, D. V. B., "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Trans. Comput.* **C-31**, 1054-1066 (1982).

[19] Broomhead, D. S., Harp, J. G., McWhirter, J. G., Palmer, K. J., and Roberts, J. G. B., "A Practical Comparison of the Systolic and Wavefront Array Processing Architectures," in *2nd Proc. IEEE Conf. on Acoustics, Speech, and Signal Processing*, Tampa, Fla., pp. 296-299 (1985).

ACKNOWLEDGMENTS—The parallel retrospective filter described in this article was developed through Independent Research and Development under the auspices of the Computer Architecture Thrust Area. I would like to thank Guy W. Riffle, Linda A. Biddison, and Richard J. Prengaman for providing background material and suggestions for a parallel implementation, Mars J. Gralia for being a helpful mentor, and Thomas P. Sleight for chairing the thrust area.

APPENDIX

A TUTORIAL ON PARALLEL ARCHITECTURES

To describe the differences and similarities between various parallel architectures, a loose set of definitions has been informally adopted by the computing community. The basis for these terms comes from a taxonomy defined by Flynn.[A1] Additional classifications have been developed by Feng,[A2] Händler,[A3] and most recently Skillicorn,[A4] but Flynn's taxonomy continues to be used as the standard.

Flynn described parallel architectures as a function of the location of parallelism in instruction and data streams. A stream could contain one source (single) or many sources (multiple) of instructions or data. Using all possible combinations, Flynn specified four different architectures: single-instruction single-data (SISD), single-instruction multiple-data (SIMD), multiple-instruction single-data (MISD), and multiple-instruction multiple-data (MIMD).

An SISD architecture is typical of today's sequential computer systems. An SISD machine is composed of a single processing unit attached to a local memory (Fig. A1). The local memory contains instructions for both the processing unit and data to be operated on by those instructions. SISD refers to the presence of a single instruction stream and single data stream between the processing unit and local memory.

Parallelism within an SISD architecture is very limited. The internal organization of the processing unit may contain a pipeline to parallelize instruction fetch, decode, and execution stages and may also contain multiple functional units (e.g., adders, multipliers, and shifters) that can operate concurrently. Separate instruction and data memories may be available to improve memory bandwidth. An optional input/output processor may be attached to the local memory, allowing the processing of instructions and transfer of data to proceed concurrently, but all algorithm executions and data manipulations are still performed solely by the single processing unit.

An SIMD architecture (Fig. A2) is composed of several processing units, each connected to a separate local memory containing all or part of a program's data. All processing units are under the control of a central control processor that feeds instructions from a central instruction memory to all processing units. Parallelism in computations is achieved by allowing each processing unit to execute the same instruction but on different local data. Extra features can be added to the architecture to allow processing units to communicate with one another and the control processor through a communications network. Additionally, the control processor may inhibit the execution of an instruction by any processing unit through the use of an instruction-mask mechanism.

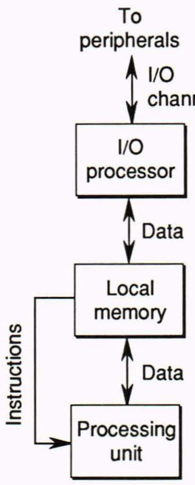


Figure A1. A single-instruction single-data (SISD) architecture. The term I/O denotes input/output.

The processing units of an SIMD architecture are usually much less complex than the control processor or processing units found in SISD

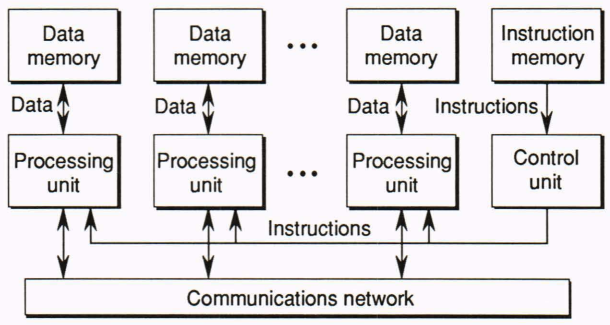


Figure A2. A single-instruction multiple-data (SIMD) architecture.

or MIMD architectures, but because of their simplicity, many more can be provided. For example, the Goodyear Aerospace massively parallel processor (MPP) has 16,384 processing units, each capable of operating on a single bit of datum.[A5] Likewise, the local data storage is usually quite small; each MPP processing unit has 1024 b of local data. The control processor is much more complex and will typically be a 16- or 32-b sequential computer similar to a Motorola 68000 or 68020. A large instruction memory is usually provided as well.

In an MISD architecture, multiple processing units execute different instructions on the same data stream (Fig. A3). Each processing unit contains a separate local memory from which instructions are executed. Data to process are either available in a global memory or copies are provided in local data memories to reduce contention. One example of this architecture's use is in signal processing. A single stream of raw sampled data may be processed differently to detect features (e.g., correlate in one processor, autocorrelate in another, and fast Fourier transform in a third).

Many algorithms have multiple data streams and/or multiple instruction streams, but few have the single data and multiple instruction streams required to best use this architecture. An MIMD architecture can mimic MISD and also provide data parallelism. Thus, of the four classes, MISD has not found favor within commercial parallel systems; it is used mainly in very specialized applications.

An MIMD architecture consists of processing units operating on multiple data streams with multiple instruction streams. Unlike the processing units in an SIMD architecture, each MIMD processor is quite complex. Typically, the processor has a 32-b word size and can execute a wide variety of instructions (including floating point). A vector (as well as a scalar) floating-point unit may also be provided to perform repeated operations on arrays efficiently. Although not specified by Flynn, two different configurations for MIMD exist: loosely and tightly coupled. The coupling term specifies the connections between the processing units and memories and the method used for interprocessor communications.

In an MIMD tightly coupled architecture, the processing units and memory modules are connected through a communications network (Fig. A4). Any processing unit may access data in any memory module. Optional input/output processors may also be present, which shuffle data between the memories and the real world. A guard mechanism must be provided in hardware and/or software to prevent multiple processing units and input/output processors from accessing the same location in a shared memory module at the same time. To alleviate congestion in this guard function, each processing unit may have private access to a local memory that contains code and/or data. This memory may be organized as random-access or cache memory.

Processors communicate with each other by leaving messages in mailboxes. A mailbox is simply a set of locations in shared memory designated to contain messages from or for a processor. Messages are usually short and contain references to address pointers that specify the location of data to be processed within the memory modules.

An MIMD loosely coupled architecture, on the other hand, does not rely on shared global memory for communications. Each processing unit is coupled to a private local memory. An input/output processor or direct memory access device is used to transfer data between the local memory and a communications network (Fig. A5). Dual-ported memories may also be used to interconnect two processing units in a loose fashion. As in the tightly coupled machine, a guard must also

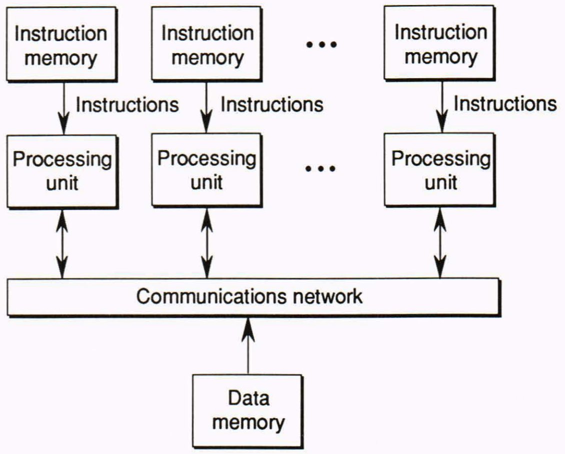


Figure A3. A multiple-instruction single-data (MISD) architecture.

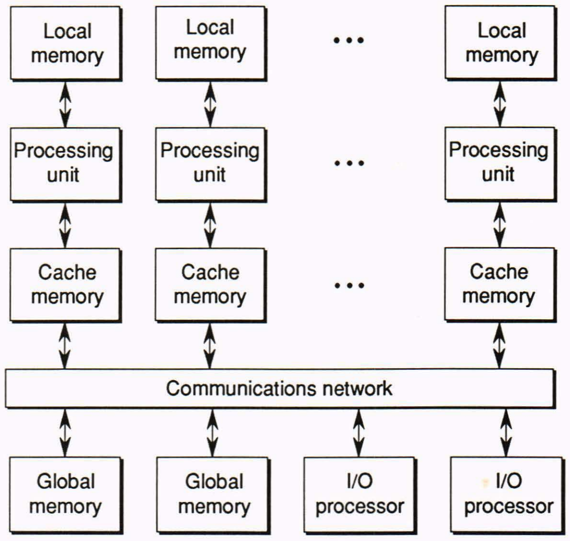


Figure A4. A tightly coupled multiple-instruction multiple-data (MIMD) architecture. The term I/O denotes input/output.

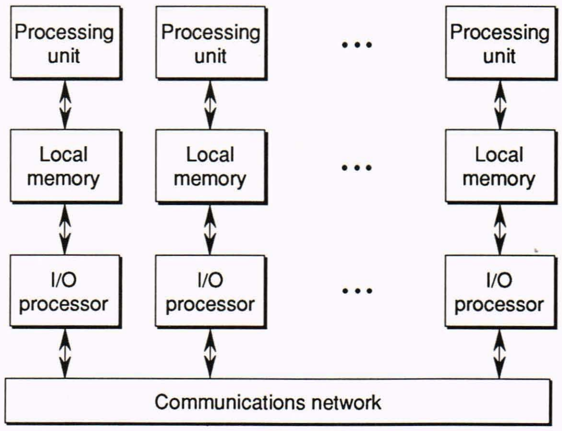


Figure A5. A loosely coupled multiple-instruction multiple-data (MIMD) architecture. The term I/O denotes input/output.

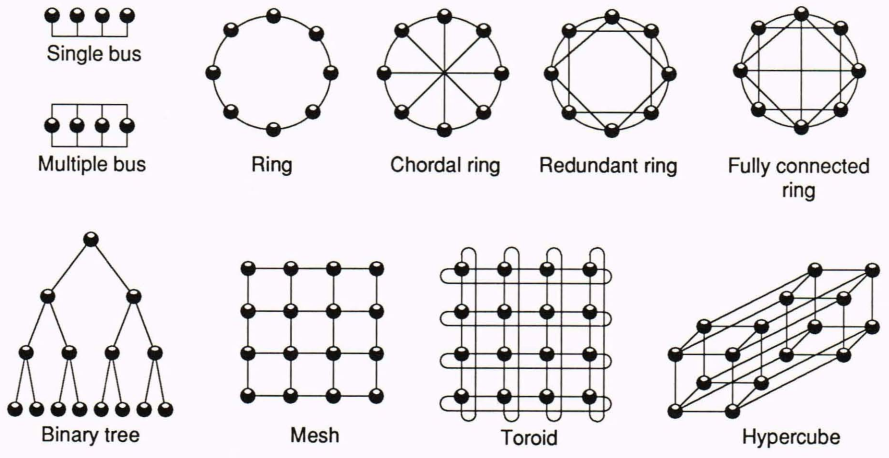


Figure A6. Some examples of topologies of communications networks.

be provided to prevent the simultaneous access of memory by two or more devices. Because a smaller number of devices are contending for the local memory, however, the guard function can be much simpler and operate much faster.

Processors communicate by transmitting messages to one another over the communications network. Multiple direct memory access units may exist in a single processor to allow simultaneous conversations with other processors to take place. Because data may be stored in different locations in each processing unit's local memory, messages must contain all data required for processing and not just references using pointers.

Each MIMD configuration has a variety of advantages and disadvantages. In loosely coupled systems, the main advantage tends to be easy expandability. Because the number of direct memory access units or input/output processors scales linearly with the number of processing units, the addition of more processing units is likely to result in a linear speedup in processing power, given that the network topology can handle the increased input/output throughput. Thus, more powerful parallel processors can be constructed with this configuration. Other advantages include low memory access contention, better fault tolerance, and a higher aggregate input/output bandwidth when all channels in the network are considered. The chief disadvantage lies in the low input/output rates between computing units. Additional disadvantages include input/output bottlenecks and the increased amount of memory required because each computing unit must maintain its own set of data and instructions.

In contrast, tightly coupled systems have relatively high data-transfer rates between processors because memory modules are used for communication. Additionally, data and instructions can be shared between processors, thus reducing the amount of memory required. This configuration has a number of disadvantages, however, including memory access contention between multiple processors accessing a common memory, generally limited expandability in comparison with loosely coupled systems, lower number of main processors possible, and generally smaller parallel-processing systems.

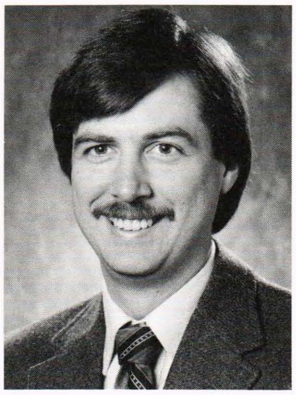
Communications networks may be implemented in a variety of topologies, including single bus, multiple buses, rings, trees, meshes, toroids, hypercubes, and fully connected networks (Fig. A6).[A6] The type of topology used depends on the number of processors provided, the bandwidth of the network connections, and the cost of the interconnection hardware. In contrast, the best topology for an algorithm is one that matches the communications patterns associated with a program's parallelism. Many vendors try to make a network that is general purpose for a large number of users, whereas others try to provide a network that can be reconfigured for each user.

Many implementations of these basic forms exist today. The SIMD architecture has traditionally supported the largest number of processors, although each processor is quite simple (e.g., a bit processor), whereas MIMD loosely and tightly coupled architectures have fewer processors, but each is more sophisticated in nature. Loose coupling is currently more popular than tight coupling because it can support more processors and can have near-linear increases in speedup with the addition of processors. Many hybrids are appearing, however, which provide either SIMD/MIMD modes[A7] or loosely/tightly coupled modes.[A8] As processor and network technology improves, the differences between all parallel architectures will most likely converge.

REFERENCES

[A1] Flynn, M. J., "Very High Speed Computing Systems," *Proc. IEEE* **54**, 1901–1909 (1966).
[A2] Feng, T. Y., "Some Characteristics of Associative/Parallel Processing," in *Proc. 1972 Sagamore Computer Conf.*, Syracuse University, pp. 5–16 (1972).
[A3] Händler, W., "The Impact of Classification Schemes on Computer Architectures," in *Proc. 1977 International Conf. on Parallel Processing*, Chicago, Ill., pp. 7–15 (1977).
[A4] Skillicorn, D. B., "A Taxonomy for Computer Architectures," *IEEE Comput.* **21**, 46–57 (1988).
[A5] Goodyear Aerospace Co., *Massively Parallel Processor (MPP)*, Tech. Report GER-16684 (Jul 1979).
[A6] Reed, D. A., and Grunwald, D. C., "The Performance of Multicomputer Interconnection Networks," *IEEE Comput.* **20**, 63–73 (1987).
[A7] Seigel, H. J., Seigel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., et al., "PASM: A Partionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. Comput.* **C-30**, 934–947 (1981).
[A8] BBN Laboratories, Inc., *Butterfly Parallel Processor Overview*, BBN Tech. Report, Cambridge, Mass. (1986).

THE AUTHOR



KENNETH W. KOONTZ was born in Baltimore in 1960. He received a B.S. in electrical engineering from the University of Maryland in 1982 and an M.S. in computer science from The Johns Hopkins University in 1987. He joined APL in 1983 and is an engineer in the Real-Time Systems Section of the Advanced Systems Design Group in the Fleet Systems Department.

Mr. Koontz has worked on standard embedded computers for Navy shipboard applications and was responsible for parallel I/O testing on the AN/UYK-43 Engineering Development Model. Recently, he has been investigating and applying parallel architectures to nonrecursive signal-processing algorithms for the Aegis Ship Building Program. His interests include computer architectures, real-time systems, fault-tolerant computing, and concurrent programming techniques.