resources (e.g. memory) to run the program. The systems programmer must be sufficiently familiar with each of these areas in order to isolate the problem. If the problem is with the program itself, a debugger would be used to obtain more information. In the early days of computing, debugging programs was an especially difficult task. Often, a dump (*q.v.*) of the program's memory contents was all that was available. More recently, interactive debuggers allow a user to set *breakpoints* at arbitrary statements in their program. When the program reaches a breakpoint, the debugger suspends the program and allows the user to display the contents of active variables by their symbolic names. Likewise, when a program error terminates the program, the debugger allows interactive inspection of its data structures, from which the high-level cause of the problem can be ascertained.

Compilers translate high-level programs into assembler language or directly into machine language. *Assemblers*, in turn, translate assembly language programs into object files, and a *linker* combines object files into a single executable load module. Because of their dependence on the underlying hardware and operating system, the creation and maintenance of assemblers, linkers, and loaders belongs to the realm of systems programming.

The availability of low-cost workstations and personal computers has led to the development of *distributed computing*. One of the most common applications of distributed computing—electronic mail (*q.v.*)—allows a user on one machine to send mail ("email") to someone having an account on another. One challenging aspect of systems programming is the configuration and maintenance of networked systems. Adding a new machine to the network requires configuring it into the local system, including the assignment of a low-level network address and a user-friendly name by which email users can refer to it. In addition, support software, such as name servers and file servers, may need to have the new machine registered with them so that existing machines can determine how to communicate with the new one.

A basic understanding of computer networks (*q.v.*) helps in the maintenance of systems in a distributed environment. The machines of various vendors may be unable to interoperate properly because one or the other (or both) fails to adhere to the appropriate protocol (*q.v.*) specification precisely. If each vendor blames the other, the systems programmer may be forced to locate the offender, perhaps with the aid of a network monitor and a description of the protocol specification.

In addition to networked systems, the availability and proliferation of low-cost computers has produced several interesting challenges for systems programmers. First, because users prefer consistent environments, it is often necessary to install the same version of a software program on multiple machines. With the plethora of machine architectures and operating systems, porting an application from one system to another may pose difficulties (*see* SOFTWARE PORTABILITY). Applications may use a subtle operating system feature, or a compiler for one architecture may accept a slightly different language dialect than the compiler for another. Moreover, existing compilers are often modified to generate code for a new architecture, and some of its more advanced features, such as a code optimizer, may not generate correct code in all cases. The systems programmer must be prepared to recognize these potential pitfalls and take corresponding action.

Distributed and parallel processing (*q.v.*) systems pose special problems for systems programmers. Maintaining a system frequently requires that the programmer be able to monitor its state. The state of a distributed or parallel system, which has many activities going on at once, is difficult to observe, and sometimes difficult even to define precisely. Utilities and management tools for operating such systems are only beginning to become available, and to date they do not make the task easy.

Although most users make use of compilers, editors, and other tools, a systems programmer generally needs to have a more detailed understanding of such tools. The more information systems programmers have about a system, the better they are able to improve its performance.

Bibliography

1990. Beck, L. L. *Systems Software: An Introduction to Systems Programming*. Reading, MA: Addison-Wesley.

1996. Oney, W. *Systems Programming for Windows 95*. Redmond, WA: Microsoft Press.

1997. Clarke, D. L., and Merusi, D. E. *System Software Programming: The Way Things Work*. Upper Saddle River: Prentice Hall.

Thomas Narten

SYSTOLIC ARRAY

For articles on related subjects *see* PARALLEL PROCESSING; PIPELINE; and SUPERCOMPUTERS.

Systolic arrays are a family of parallel computer architectures capable of using a very large number of processors simultaneously for important computations in applications such as scientific computing and signal processing. This article gives a general description of systolic arrays, illustrates the idea by two simple examples, lists some applicable computations, and describes fine-grain interprocessor communication in systolic arrays.

General Description

Systolic arrays are suited for processing repetitive computations. Although this kind of computation usually requires a great deal of computing power, such computations are highly regular and parallelizable. The systolic array architecture exploits this regularity and parallelism to deliver the required computational speed.

In a systolic array, all processing elements, called *systolic cells*, perform computations simultaneously, while data, such as initial inputs, partial results, and final outputs, is being passed from cell to cell. When partial results are moved between cells, they are computed over these cells in a pipeline fashion. In this case, the computation of each single output is partitioned over these cells. This contrasts to other parallel architectures based on data partitioning, for which the computation of each output is computed solely on one single processor.

When a systolic array is in operation, computing at cells, communication between cells and input from and output to the outside world all take place at the same time to achieve high performance. This is analogous to the circulatory system; data is "pulsed" through all cells where it is processed.

Being able to perform many operations simultaneously is just one of the many advantages of systolic arrays. Other advantages include modular expandability of the cell array, simple and regular data and control flows, simple and uniform cells, efficient fault-tolerant schemes, and nearest-neighbor data communications. These properties are highly desirable for VLSI (Very Large-Scale Integration) implementations. Indeed, the advances in VLSI technology have been a major motivation for much interest in systolic arrays.

Two Systolic Array Examples

For illustration, consider first a simple systolic array for implementing a finite impulse response (FIR) filter. Given inputs x_i and weights w_j, the filtering problem is to compute outputs y_i, defined by $y_i = w_1 x_i + w_2 x_{i+1} + \cdots + w_k x_{i+k-1}$. Fig. 1 depicts a one-dimensional systolic array for a FIR filter with $k = 3$ weights, each of which is preloaded into a cell.

During computation, both partial results for y_i and inputs x_i flow from left to right, where the former move twice as fast as the latter. More precisely, each x_i stays inside every cell it passes for one additional cycle, and thus each x_i takes twice as long to march through the array as does a y_i. One can check that each y_i, initialized to zero before entering the leftmost cell, is able to accumulate all its terms while marching to the right. For example, y_1 accumulates $w_3 x_3$, $w_2 x_2$, and $w_1 x_1$ in three consecutive cycles at the leftmost, middle, and rightmost cells, respectively.

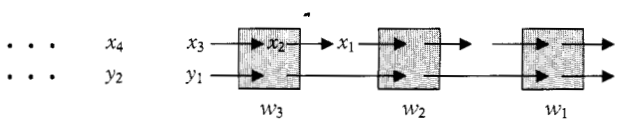

Figure 1. One-dimensional systolic array for implementing FIR filter.

Note that, although each output y is computed using several inputs x and several weights w, and each input and each weight is used in computing several outputs, the systolic array described here uses no "global" communication. More precisely, data communication at each cycle is always between adjacent cells.

Fig. 2 illustrates another systolic array. This is a two-dimensional array capable of performing matrix multiplication, $C = A \times B$ for 3×3 matrices $A = (a_{ij})$, $B = (b_{ij})$ and $C = (c_{ij})$. As indicated, entries in A and B are shifted into the array from left and top, respectively. It is easy to see that the c_{ij} at each cell can accumulate all its terms $a_{i1}b_{1j}$, $a_{i2}b_{2j}$, and $a_{i3}b_{3j}$ while A and B march across the systolic array.

Scope of Applicable Computations

A large number of systolic array designs have been developed and used to perform a broad range of computations. In fact, recent advances in theory and software have allowed some of these systolic arrays to be derived automatically. The following is a representative list of computations for which systolic designs exist:

- *Signal and image processing*—Digital filters, convolution and correlation, discrete Fourier transform, fast Fourier transform (FFT—*q.v.*), encoding/decoding for compression (*see* DATA COMPRESSION)

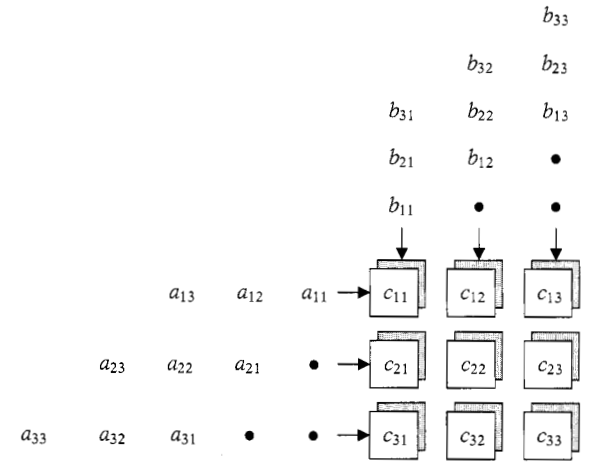

Figure 2. Two-dimensional systolic array for matrix multiplication.

and error-correction (*see* ERROR CORRECTING AND DETECTING CODE), etc.

- *Matrix arithmetic*—Matrix multiplication, solution of linear systems of equations, solution of Toeplitz linear systems, QR-decomposition, least-squares computation, singular value decomposition, eigenvalue computation, etc. (*see* NUMERICAL ANALYSIS).
- *Polynomial and multiple precision integer arithmetic*—Multiplication, division, greatest common divisor, etc.
- *Nonnumeric applications*—Searching (*q.v.*), sorting (*q.v.*), pattern matching, regular language recognition, dynamic programming, relational database (*q.v.*) operations such as join and intersection, data structures (*q.v.*) such as priority queues, and graph and geometric algorithms such as minimum spanning trees, convex hull calculations, etc.

Fine-Grain Communication in Systolic Arrays

In a systolic array, each cell processes a data word immediately after it arrives, and sends out a data word immediately after it is processed. Therefore, the unit of communication is a single word. This contrasts to the classic message-passing communication, where the unit of communication is an entire message, which typically consists of a large number of words. Thus, supporting this fine-grain communication, also called *systolic communication*, is a unique architectural feature of a systolic array computer.

To support systolic communication, each cell in a systolic array allows its CPU to access the input and output ports directly, in addition to the local memory. This differs from message-passing machines for which each processor can access only its local memory. Fig. 3 illustrates the difference.

Summary

Systolic arrays are an effective parallel architecture for a wide range of computations that are repetitive in nature. By using fine-grain communication, the architecture can use a large number of processors simultaneously. Moreover, systolic arrays have a simple and regular design, so their implementations are relatively easy. As software and hardware technologies continue to advance to allow efficient systolic designs or programs to be derived automatically and routinely, and to allow very large systolic arrays to be implemented inexpensively, widespread use of systolic arrays can be expected in solving many computationally demanding problems.

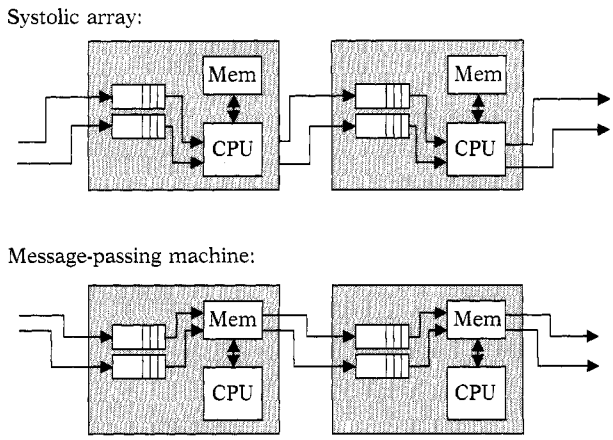

Figure 3. Comparing a systolic array with a message-passing machine.

Bibliography

1990. Lang, T., and Moreno, J. H. "Matrix Computations on Systolic-type Meshes," *Computer*, **23**, *4* (April), 32–51. Begins with an excellent tutorial on systolic parallel processing.

1990. Dostie, A. J., Seidman, S. B., and Clessas, A. C. "Systolic Computing on Transputer Networks," *Proceedings of North American Transputer Users Group*, Durham, SC, October 1989, 123–137. Amsterdams: IOS.

1991. Quinton, P., Robert, Y., and Craig, I. *Systolic Algorithms & Architectures*. Upper Saddle River, NJ; Prentice Hall.

1991. Evans, D. J. (ed.) *Systolic Algorithms*. London: Gordon & Breach.

1992. Gruska, J. *Systolic Computation*. New York: Springer-Verlag.

1992. Megson, G. M. *An Introduction to Systolic Algorithm Design*. Oxford: Oxford Science Publications.

1992. Moreno, J. H., and Lang, T. *Matrix Computations on Systolic-Type Arrays*. New York: Kluwer-Academic Press.

1993. Petkov, N. *Systolic Parallel Processing*. Amsterdam: North-Holland.

H. T. Kung