

# COMPUTER

December 1994 • Vol. 27, No. 12

**Membership  
Magazine of  
the IEEE  
Computer  
Society**

**Circulation:** *Computer* (ISSN 0018-9162) is published monthly by the IEEE Computer Society, IEEE Headquarters, 345 East 47th St., New York, NY 10017-2394; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264, phone. (714) 821-8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Annual subscription included in society member dues. Non-member subscription rates: available upon request. Single copy prices: members \$10.00; nonmembers \$20.00. This magazine is also available in microfiche form.

**Postmaster:** Send undelivered copies and address changes to *Computer*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855. Second class postage is paid at New York, New York, and at additional mailing offices. Canadian GST #125634188. Printed in USA.

**Copyright and reprint permission:** Copyright © 1994 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved. Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US copyright law for private use of patrons: (1) those post-1977 articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Dr., Danvers, MA 01923; (2) pre-1978 articles without fee. For other copying, reprint, or republication permission, write to Permissions Editor, *Computer*, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264.

**Editorial:** Unless otherwise stated, bylined articles, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *Computer* does not necessarily constitute endorsement by the IEEE or the Computer Society. All submissions are subject to editing for style, clarity, and space.



## FEATURE ARTICLES

### 15 **Demonstration of an Interactive Multimedia Environment**

*Charles Rich, Richard C. Waters, Carol Strohecker, Yves Schabes, William T. Freeman, Mark C. Torrance, Andrew R. Golding, and Michal Roth*

The images in this article tell the story of one user's experience with a virtual reality environment that promotes collaboration and learning.

### 24 **LAN and I/O Convergence: A Survey of the Issues**

*Martin W. Sachs, Avraham Leff, and Denise Sevigny*

Once two distinctly separate technologies, LANs and I/O are becoming more alike through similar distances, media, and purposes. What few differences exist may disappear in the next decade.

### 34 **Communication Styles for Parallel Systems**

*Thomas Gross, Susan Hinrichs, David R. O'Hallaron, Thomas Stricker, and Atsushi Hasegawa*

Communication style significantly impacts the way parallel systems use their resources. This article compares systolic and memory communication on the iWarp system.

## CYBER SQUARE

### 45 **Innovation Delayed Is Innovation Denied**

*Al Gore, Vice President of the United States*

Competition in the local telephone exchanges is a fundamental component of competition in the information marketplace.

### 48 **Information Technologies in South Africa: Problems and Prospects**

*Gay A. Wood, Seymour E. Goodman, and Jan Roos*

Development of South Africa's information technologies infrastructure has been skewed by apartheid and the constraints of past policies, but IT can be part of the solution.

## COMPUTING PRACTICES

### 58 **The Challenges of Designing Groupware: A Case Study**

*Mark Davies, Kris Pettersen, Shankar Srinivasen, and Katherine Kinsman*

Designing a commercial software program is a study in market need and technological feasibility. This case study traces the development of a groupware product for managing scientific projects and data.

## SPECIAL FEATURE

### 108 **Annual Index**

# Communication Styles for Parallel Systems

Thomas Gross, Susan Hinrichs, David R. O'Hallaron,  
and Thomas Stricker, Carnegie Mellon University

Atsushi Hasegawa, Hitachi Microcomputer Systems

**Communication style significantly impacts the way parallel systems use their resources. This article compares systolic and memory communication on the iWarp system.**

All distributed-memory parallel systems rely on explicit message exchange for communication, but the communication operations they support can differ in many aspects. One key difference is the way messages are generated or consumed. With systolic communication, a message is transmitted as it is generated. For example, the result computed by the multiplier is sent directly to the communication subsystem for transmission to another node. With memory communication, the complete message is generated and stored in memory, and then transmitted to its destination.<sup>1</sup> Since sender and receiver nodes are individually controlled, they can use different communication styles. One example of memory communication is message passing: Both the sender and receiver buffer the message in memory.

These two communication styles place different demands on processor design. This article illustrates each style's effect on processor resources for some key application kernels. We are targeting the iWarp system because it supports both communication styles. Two parallel-program generators — one for each communication style — automatically map the sample programs (see sidebar, "Parallel-program generators"). Measuring these program executions lets us identify any correlation between communication style and instruction-type frequencies, evaluate how operands are stored (in registers or in memory), assess which machine resources are used (or not used) by specific parallel programs, and determine the impact on instruction-level parallelism.

## Experimental Setup

We are the first to empirically evaluate how communication style affects parallel-system resource use. The experiments are performed on the iWarp system, which efficiently supports both systolic and memory communication styles and provides an excellent platform for parallel-program generators.

**The iWarp system.** The iWarp is a single-chip, VLSI processor developed by Intel<sup>1,2</sup> that became operational in 1990. It contains separate floating-point units for addition and multiplication (20 Mflops single precision, 20 MIPS) and a high throughput (320

MBytes/second), low latency (200 ns) communication agent, which transfers data between other iWarp processors. This communication agent provides nonadjacent nodes with long-distance connections that reserve bandwidth and reduce communication start-up overhead.<sup>1</sup> An iWarp node contains an iWarp chip and some local memory (ranging from 0.5 MBytes to 16.5 MBytes on our system). The peak local-memory bandwidth is 160 MBytes/second. An iWarp system is organized as a 2D torus of four to 1,024 iWarp nodes.

Figure 1 depicts a block-level sketch of the iWarp processor. Each processor's communication agent contains first-in, first-out (FIFO) queues, each eight words deep. Data received from neighboring nodes is stored in one of these queues until the program is ready to process it. Queues can be mapped into systolic gates, which are special registers in the register file (shaded in Figure 1). Systolic gates can be used as instruction operands with the same access time as any other register in the register file. When a gate supplies an input operand, a data word is transported from the associated queue's head to the functional unit. If the queue is empty, the instruction spins until a data word arrives. Writing to a gate sends the data word to the associated queue's tail. If the queue is full, the instruction spins until there is space.

The iWarp instruction set contains two types of instructions:

- Short instructions contain a single operation and control a single functional unit. Examples include load, store, floating-point add, floating-point multiply, and integer add. Loads, stores, and floating-point operations execute in two cycles (100 ns). Most other short instructions take one cycle. Short instructions are encoded in one 32-bit word.
- Long instructions contain multiple operations and control multiple functional units. A multioperation long-instruction word (LIW) is encoded in 96 bits and contains a maximum of five constituent operations. Its execution time is limited by the maximum execution time of each operation. A single, two-cycle LIW can perform one load and one store or two loads (with the address-register autoincrement), a floating-point add, and a floating-point multiply, while decrementing and testing a counter for loop implementation.

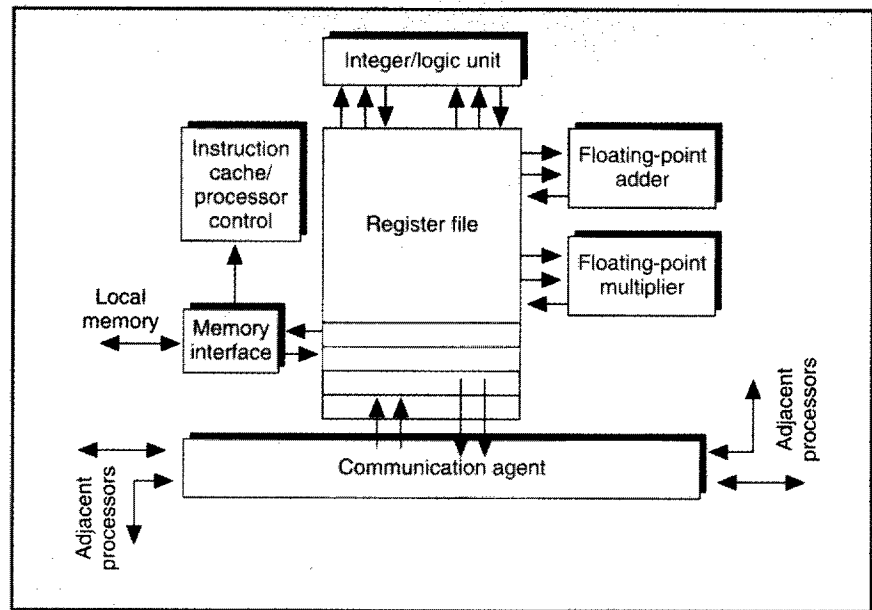


Figure 1. The iWarp processor.

The iWarp node contains several features that support memory communication. Each node provides a high bandwidth to local memory. The LIW format allows one load for every floating-point operation with an 80-MByte/second peak-memory bandwidth. This bandwidth and the flexibility of the LIW format make sure the processor is not starved for data, even when none is supplied by the communication system. For example, a scalar dot product can proceed at the peak floating-point rate.

The iWarp processor contains eight spoolers, which are DMA-like controllers that move medium-sized and large messages directly from the communication system to memory. Each spooler moves 40 MBytes/second of data and steals one 50-ns cycle to transfer two 32-bit words. The total memory bandwidth is 160 MBytes/second. Thus, up to four spoolers can proceed at full bandwidth but, in this case, steal all memory cycles from the computation agent.

Spoolers are valuable when multiple messages arrive at the same time or when message arrival and program execution occur asynchronously. However, spooler setup requires the execution of instructions. When only one or two messages must be moved to memory and the message arrival is synchronized with program execution, then moving the message to memory via explicit store operations is faster.

**Sample programs.** We analyze single-node resource use for four standard linear-algebra application kernels: matrix

multiplication (MM), LU decomposition, QR decomposition, and successive over relaxation (SOR). There are two parallel programs for each kernel: one program based on memory communication and one based on systolic communication. Two locally developed program generators use high-level computation descriptions to automatically produce all programs except systolic SOR<sup>3,4</sup> (because of SOR's structure, the parallel-program generator cannot automatically create a systolic program).

There are many memory communication models, including message-passing variations used by private-memory computers. These models differ in their functionality and overhead. In this study, we employ only neighbor-to-neighbor communication and a high-speed broadcast primitive, since the class of programs that can automatically be mapped is regular enough for this choice to yield the fastest programs. A broadcast message is stored and forwarded on a word-by-word basis; this allows the sender to operate at full speed, determined solely by the communication bandwidth. Since all parallel-program connections can be set up during load time (and the parallel-program generator takes advantage of this feature), there is no protocol overhead associated with a message. More sophisticated memory communication schemes may produce programs with different characteristics. However, the zero-overhead protocol adopted by our program generator is adequate for the regular programs used in this study and produces the best overall results on the iWarp parallel system.

**Table 1. Differences between profiled time and actual execution time for matrix-multiplication implementations.**

	Profiled time (ms)	Actual time (ms)	Difference (percent)
Systolic communication	274	270	1.45
Memory communication	760	710	6.57

**Table 2. Mflops measured on a 16-node system for all test programs. (Maximum rate possible is 320 Mflops.)**

Program	Memory communication (Mflops)	Systolic communication (Mflops)
MM	113	302
SOR	98	128
LU	41	145
QR	150	174

**Profiling and measurement strategy.** Each parallel program generated by its respective parallel-programming tool consists of one C program (called a node program) for each node in the system, plus information to set up connections between node programs. A conventional single-node compiler loaded onto an array compiles each node program. We do not include connection setup in our measurements because it is part of loading and hardly contributes to execution time.

All programs are compiled to use single-precision arithmetic for computations involving only *float* values. This format is sufficient for our application domain: image processing, computer vision, and signal processing.

The code generated for each node is then annotated to gather profiling information that determines how often each basic block is executed. Each node program executes on a separate node and tracks the frequency of entries to each basic block. After termination of the user code, the runtime system writes history information into a log file on the front-end computer. An auxiliary program combines the basic-block entry frequency with the object code to obtain profiling information.

All programs are given  $160 \times 160$  input matrices and are run on 16 iWarp nodes. Because of the regular structure of the programs, there is almost no program variation between different nodes. Therefore, in each case we present numbers from a single representative node.

This static-profiling approach does have some limitations, but the effects of these limitations are minimal. First, because we base our analysis on the frequency information for each basic block of the user program, we cannot include any time spent in system libraries (such as the C math library) or the runtime system. However, only a small fraction of time is spent in the system libraries and

no time is spent in the runtime system for the programs we are investigating.

Second, the information collected is sufficient to determine which user instructions are executed, but does not account for time spent waiting or spinning (for example, time owing to instruction-cache misses or waiting for messages to arrive). Nevertheless, for the single-node characterization, the static-profiling information is sufficient. By design, systolic programs do not spend any cycles waiting for data to arrive, and for the given size of the parallel system (16 nodes), programs based on memory communication do not have to wait either.

Table 1 compares the time reported on a node to overall execution time for a matrix-multiplication program. Discrepancies are due to network-resource delays, stalls between adjacent instructions caused by incomplete bypasses between functional units, and instruction-cache misses.

Table 2 shows the measured Mflops for all programs.

## Results and evaluation

Systolic communication promises two measurable benefits<sup>1</sup>:

- increased instruction-level parallelism (since the communication system is another source of operands, more operations can be performed in parallel) and
- reduced access to local memory (operands are received from other nodes, and results can be sent to other nodes; there is no need to store these values in memory).

Dynamic measurement of instruction and operation frequencies permits em-

pirical evaluation of these benefits. Since systolic programs are close to optimal for iWarp, different systolic program generators are unlikely to produce significantly different results.

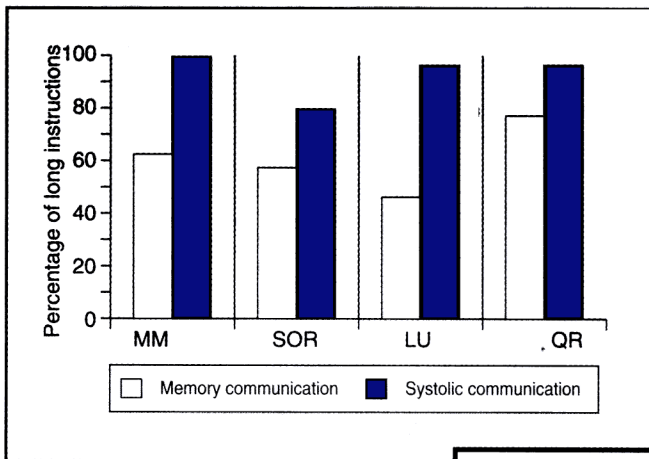
Here, we present instruction-profiling results and describe the microarchitectural features responsible for our observations.

**Instruction sets.** The two iWarp instruction formats (short and LIW) let us directly measure the impact of the two communication styles on instruction-level parallelism. Figure 2 illustrates the percentage of iWarp system operations performed by LIW instructions during each communication style's execution.

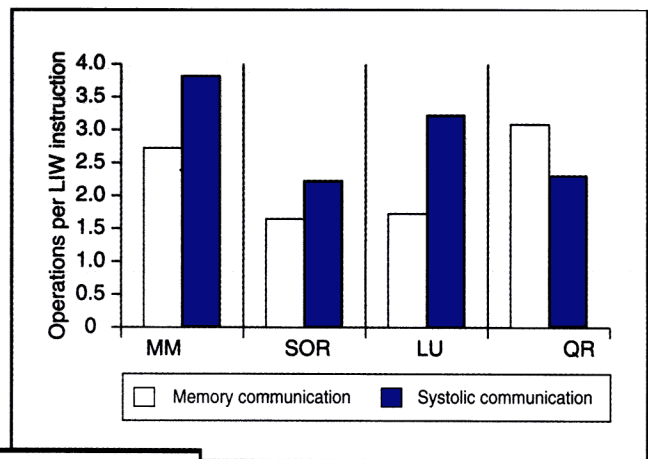
Figure 2 reveals that a properly optimized program can effectively use LIW instructions. Although on average only 5 percent of the (static) instructions in the systolic object code are LIW instructions, 88 percent of all instructions are part of LIW instructions. This usually holds true for memory-based programs as well, where LIW instructions account for 10 percent of the static instructions in the code but execute about 52 percent of operations. LIW instructions occurring in the code are frequently used during runtime. The small percentage of static LIW instructions means that the average instruction length — and code size — does not grow excessively. Programs with systolic communication average 1.1 words per instruction, and programs with memory communication average 1.2 words per instruction.

Figure 3 shows the average number of operations per instruction for each program. Programs that execute a higher percentage of operations with LIW instructions also execute more operations per instruction. This proves that each LIW instruction includes a reasonable number of operations.

Figure 2 represents the relative contribution of LIW instructions as a percent-



**Figure 2. Percentage of operations executed in LIW instructions.**

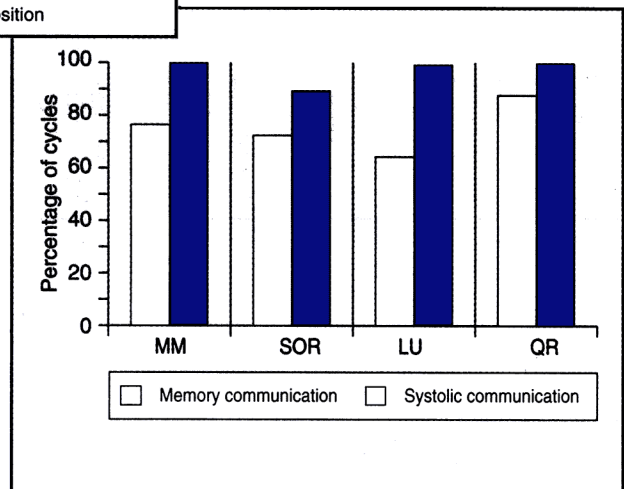


**Figure 3. Operations per instruction.**

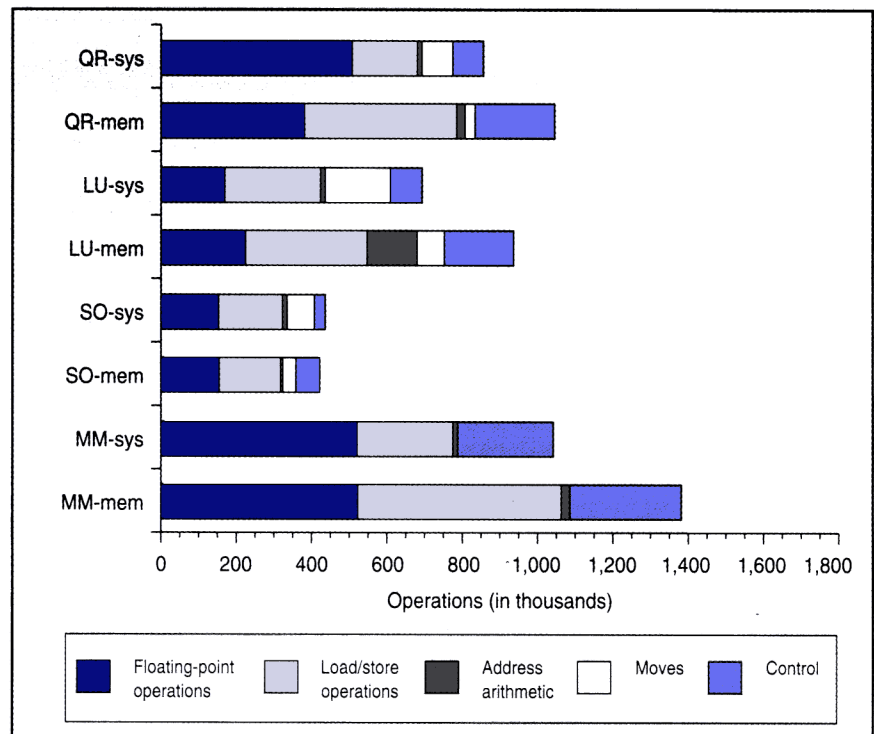
MM Matrix multiplication  
 SOR Successive over relaxation  
 LU Matrix decomposition  
 QR Matrix decomposition

age of all instructions, while Figure 3 represents the contribution as a percentage of all operations. Both metrics reflect how frequently the LIW format is used. But the execution time of LIW instructions and short instructions can differ: The execution time of an LIW instruction is the execution time of its longest operation (usually 100 ns for single-precision, floating-point operations; 200 ns for double-precision), whereas the execution time of a short instruction typically ranges from 50 ns (integer operations) to 200 ns (double-precision, floating-point operations). A compiler therefore tries to combine as many time-consuming operations in an LIW instruction as it can. If one operation takes 100 ns, then any other 100-ns operation is done at the same time, and it is more advantageous to put such an operation into this LIW instruction than, for example, to add a 50-ns operation. Figure 4 presents the breakdown of execution cycles based on the instruction format. According to this metric, the programs with systolic communication exhibit higher instruction-level parallelism.

**Figure 4. Classification of execution cycles: percentage of cycles executed by LIW instructions.**



**Operation distribution.** Figure 4 depicts the percentage of cycles executed by LIW instructions. Although there is only one LIW format, each LIW instruction can execute dramatically different operations.<sup>5</sup> A more meaningful metric would identify the operations executed for each program. This information is in Figure 5, which shows operation frequencies for different programs and provides another picture of how processor resources are used by these programs. Operations are divided into five major classes:



**Figure 5. Classification of operations executed.**

- *floating-point operations* (any operation that uses the floating-point multiplier or adder);
- *load/store operations: memory operations* (loads a value from or stores a value to local memory);
- *address arithmetic* (ALU and load-literal operations that calculate memory addresses, including shifts and bit field operations, and count leading zeros);
- *moves* (ALU operations that copy a value from one register to another; floating-point units can also copy values between registers: our profiling tool counts these moves as floating-

point operations, since a floating-point unit is busy); and

- *control* (control-flow instructions such as branches, jumps, procedure call, and return are included in this group).

In Figure 5, all integer operations are counted as address computations. This is not always correct, because some integer operations are needed solely to compute loop-trip counts, but the number of such operations is negligible. Memory-based programs must retrieve all operands from, and store intermediate results to memory, and therefore perform more memory accesses than systolic programs do

(see Figure 5). The figure also shows that memory-based programs have more address computations: Since memory-based programs access memory more frequently, they must calculate more memory addresses. (The sidebar, “Mapping strategies and system parameters,” describes how communication style impacts memory-access patterns.)

Systolic communication programs use far fewer control-flow operations because communication and computation loops are combined. Programs based on memory communication contain additional tests to determine the ownership of data. With systolic communication,

## Mapping strategies and system parameters

The program generators discussed in this article let us evaluate the impact of communication style on the iWarp system. But a compiler can use other mapping strategies, and there are also many other parallel systems. Code generation strategies can affect our results, and these results depend on the communication system architecture’s key parameters.

Many efficient systolic algorithms are not developed by a program generator. And for some computations, blocking or tiling may improve memory communication program execution, since this strategy reduces the number of communication steps. Therefore, to understand the impact of this strategy, we use some programs from a library that is based on blocking.<sup>1</sup>

Essentially, blocking or tiling divides a problem into subproblems that can be solved with good data locality. Each node solves one of the subproblems and then exchanges data to combine the subresults into the main result. This problem division does the same amount of work as the simple program division employed by our memory-based program generator, but can have better communication characteristics because of the increased locality.

**Impact of mapping strategy.** The systolic communication model attempts to use network bandwidth instead of memory bandwidth, while the memory communication model performs computations only on values stored in memory.

Specifically, the memory communication program must retrieve all computation arguments from memory, but the systolic communication program can also retrieve arguments from the network. And the memory-based program must initially store all messages in memory, whereas the systolic program avoids all memory accesses, because it never stores messages.

As an example of computation phase differences, consider the statement that forms the inner kernel of matrix multiplication and LU decomposition. In a program employing memory communication, this kernel requires the *c* and *b* vectors to be fetched from memory:

$$c[i] = c[i] + a * b[i]$$

It requires three memory operations and two floating-point operations. By contrast, the systolic inner loop passes the *c*

vector along, so that only the *b* vector must be fetched from memory:

```
tmp = receive ( )
tmp = tmp + a * b[i]
send (tmp)
```

The systolic version requires two network operations, two floating-point operations, and one memory operation, and trades two memory operations for two network operations. Therefore, the memory bandwidth requirements for the systolic version are less than for the message-passing version, but the communication requirements are greater.

Whether the program with systolic communication can perform better than the inner loop based on memory communication depends on the balance of network and memory access provided by the processor. And this balance depends on how network accesses are implemented. Two forms of network access are memory-mapped and register-mapped network queues. One example of the memory-mapped approach can be found in the connection machine CM-5.<sup>2</sup> There, network access requires memory bandwidth. Hence, sacrificing memory access for network access only makes sense when accessing the network interface is faster than directly accessing local memory, or when access to the memory-mapped network interface uses processor resources that are separate from those used by ordinary memory operations.

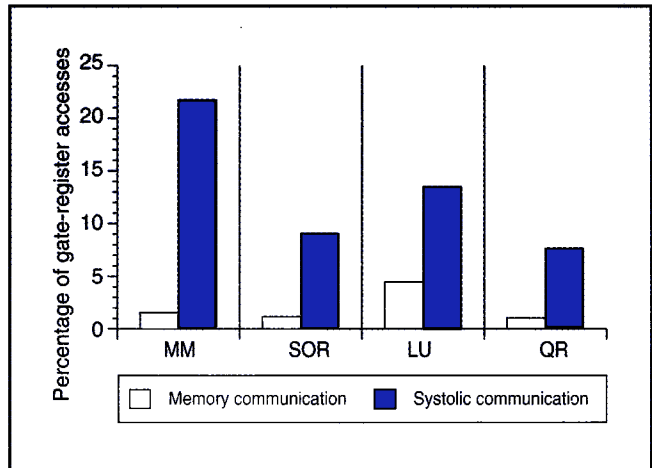
A processor’s number of parallel instructions is limited by the memory bandwidth, network bandwidth, and number of functional units supported by the machine. If a processor can multiply and add in the same cycle, the limiting function is retrieving the arguments from memory or the network. If a processor can perform three memory accesses in parallel, the inner loop of a memory-based program can be performed in one instruction. On iWarp, no more than two memory accesses can be performed in parallel, so the memory-communication inner loop must use two instructions. Of course, a processor with a single path to memory (like many contemporary microprocessors) requires at least three instructions. But the systolic inner loop can be performed in one instruction.

The blocking-based matrix multiplication algorithm rearranges the loop order so that it uses a slightly different inner loop:

data flows through the program, so the program doesn't need to calculate data addresses; it only needs to read the next value from the neighbor node.

**Program access.** The iWarp architecture includes systolic gates to allow fast program access to the communication system. A systolic gate is a port to the communication system that is mapped directly (with hardware) into the processor's register file, as shown in Figure 6. There is no operating or runtime system overhead involved in reading or writing to a gate. Using a gate as a source or destination of operands provides the mech-

**Figure 6. Percentage of variable accesses through gates.**



$$c = c + a[i] * b[i]$$

This loop requires only two memory accesses, because  $c$  can be stored in a register. This kernel still requires one more memory access than the systolic algorithm does, but a machine such as iWarp that can issue two memory accesses in parallel lets the loops be executed in the same number of cycles.

The systolic algorithm still has an advantage, because it avoids initially storing the messages in memory. As problem size grows, this advantage becomes less significant. In matrix multiplication, the program sends  $O(n^2)$  words in messages but performs  $O(n^3)$  inner-loop iterations. However, this effect is still quite noticeable for common array sizes (see Subhlok et al.<sup>3</sup> for a discussion of data set sizes encountered in sensor-based computing).

Table A shows the Mflop measurements for the three matrix multiplication programs (mapped onto 16 nodes for comparison with results presented in the article body). There is a noticeable difference between systolic communication programs and blocking programs even though the inner loop is computed in the same amount of time. As the size of the problem grows, the difference in Mflops between the blocked and systolic implementations decreases.

In summary, Table A underlines three points about parallel computations: First, with blocking, systolic and memory communications perform comparatively only when the matrices are large. If we map a computation onto a larger system (for example, one with 64 or 256 nodes), even larger inputs are required to obtain the same performance on each node. Many applications cannot scale the input set's size to match the parallel system's size, since the input set's size is limited by external constraints.<sup>3</sup> For these applications, systolic communication allows performance on a parallel system.

Second, the iWarp processor supports one memory operation in parallel with each floating-point operation. Memory-communication performance in processors without this balance (for example, those that support only one memory access for every two floating-point operations) cannot approach systolic-communication performance.

Finally, iWarp supports static connection setup, so there is no protocol overhead for memory communication. Therefore, since any protocol overhead reduces performance effectiveness, our analysis of memory communication is actually optimistic.

**Table A. Mflops measured on a 16-node system for three matrix-multiplication implementations of  $n \times n$  matrices. (Maximum rate possible is 320 Mflops.)**

$n$	Simple-distribution memory communication (Mflops)	Blocked (tiled) memory communication (Mflops)	Systolic communication (Mflops)
128	102	163	291
256	114	209	304
512	120	259	310

**Impact of communication system parameters.** Another parameter that influences the trade-offs between memory communication and systolic communication is the way the processor and program handle message arrival. The message-passing program generator we used and the block matrix multiplication explicitly store network data in memory. Other message-passing implementations move message data to the background by stealing memory cycles. However, performing the communication in the background does not avoid memory-bandwidth bottleneck and still requires time to set up data transfers.

The performance of the three matrix multiplications (memory-based, systolic, and block-based) can be compared by defining performance equations in terms of the message start-up overhead  $S$ , the transfer time to send each word  $T$ , the time to execute the body of the inner loop  $L$ , the number of processors  $p$ , and the size of the matrix  $n$ . The execution time  $E_{MP}$  of matrix multiplication with simple memory communication is expressed below (there are  $n^3$  computation steps evenly divided over  $p$  processors, and  $2n^2$  data elements must be moved):

$$E_{MP} = 2n(S + Tn) + n^3/p \times L_{MP}$$

The execution time  $E_{Sys}$  for matrix multiplication based on systolic communication is

$$E_{Sys} = n^3/p \times L_{Sys}$$

anism that reduces the number of load or store operations. Figure 6 shows the percentage of operands supplied by or written to a gate for the different applications. Operands are read from a gate about as often as they are written to a gate.

The iWarp is a load/store architecture: The operands for all computations must be retrieved from a register or gate, and the destination of all computations is either a register or gate. As shown in Figure 6, only programs with systolic communication use gates for a significant fraction of computation operands. Only these programs contain instructions that write

the result of a floating-point computation directly to a gate.

Gate operands occur in memory communication programs solely in the data distribution loops. Since the program always stores a complete block in memory, the operands must still be fetched from there for the computation. Therefore, the percentage of gate operands is much lower than for systolic communication.

## Other factors

A comparison of communication styles

must consider several factors. The input data size, instruction-level parallelism, available memory bandwidth, and parallelization strategy all contribute to the effectiveness of executing a parallel system program. For large data sets, the performance difference caused by the communication style diminishes if the processor supports sufficient memory bandwidth. For common input sizes, a systolic program enjoys a noticeable performance advantage. There are some computations for which systolic programs currently cannot be automatically generated but memory communication programs can.

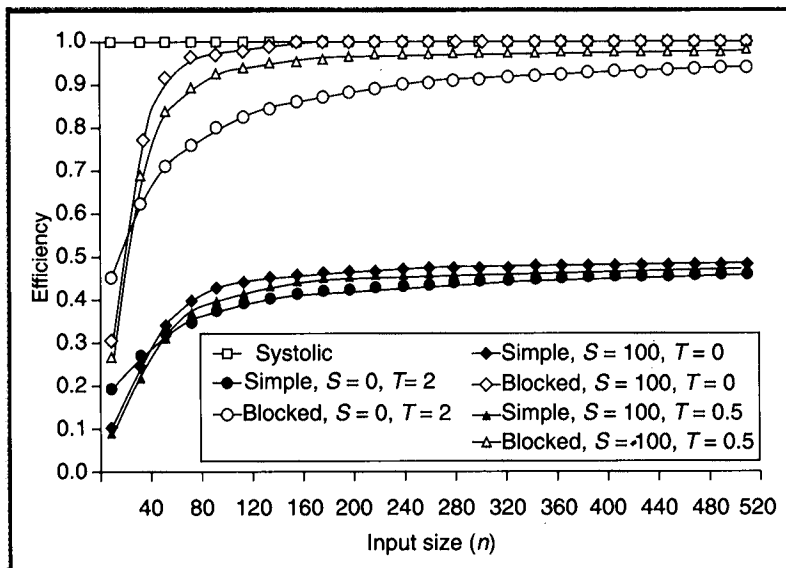


Figure D. Matrix-multiplication performance for  $p = 16$ .

The execution time  $E_{\text{Blk}}$  for the block-based matrix multiplication is

$$E_{\text{Blk}} = 4(\sqrt{p} - 1)(S + Tn^2/p) + n^3/p \times L_{\text{Blk}}$$

Note that the communication time differs for the three versions. The time to compute the innermost loop is of the same order for each, but different constant factors,  $L_{\text{MP}}$ ,  $L_{\text{Sys}}$ , and  $L_{\text{Blk}}$ , serve to offset possible differences in the processors' local computation capabilities. In our program implementation on iWarp,  $S = 0$ . In an implementation that performs the communication completely in the background,  $T = 0$ . If message storing steals no resources from the foreground computation ( $T = 0$ ), then simple memory communication contributes only about  $O(n)$  time steps asymptotically, and the blocked algorithm contributes only  $O(\sqrt{p})$ . However, if message storing requires foreground resources, simple message passing adds

an  $O(n^2)$  effect, and blocking adds  $O(\sqrt{p} \times n^2/p)$  steps.

Figure D illustrates the impact of varying the communication system parameters  $S$  and  $T$  for the three mapping strategies and shows the efficiency (node utilization) for different input sizes. To emphasize the communication system's effect, all other parameters (number of paths to memory, number of functional units, and degree of parallelism) are kept constant and equal the iWarp-system values. Parameters  $S = 0$ ,  $T = 2$  model a system with no start-up overhead and a transfer time of 2 cycles (for example, iWarp).

The simple message-passing inner loop requires two instructions because of insufficient memory bandwidth; the other two strategies pack the operations into a single instruction. Therefore, the simple message-passing program's performance can be at most half that of the other cases. A machine with completely noninterfering background transfers and a start-up overhead of 100

cycles ( $S = 100$ ,  $T = 0$ ) models a system with direct memory access (DMA).

In practice, a zero-cost transfer is difficult to realize (unless there are separate busses or memory ports), so Figure D also includes a more realistic scenario with a 0.5 transfer cost. For all models, as the matrix size increases, the performance is dominated by the cost of performing the inner loop. But for smaller sizes, there are noticeable performance differences.

## References

1. E. Anderson and J. Dongarra, "Implementation Guide for LAPACK," LAPACK Working Note, No. 18, Tech. Report CS-90-101, Univ. of Tennessee, Knoxville, Tenn., Apr. 1990.
2. Thinking Machines Corporation, "The Connection Machine CM-5 Technical Summary," Thinking Machines Corporation, Boston, 1991.
3. J. Subhlok et al., "Programming Task and Data Parallelism on a Multi-computer," *Proc. ACM Symp. Principles and Practice of Parallel Programming (PPoPP)*, ACM Press, New York, May 1993, pp. 13-22.

The processor features that support systolic communication can also be employed to implement novel memory communication schemes. A parallel-program generator manages the memory of all nodes and may determine where incoming data should be stored. Either the sender or receiver node must compute these addresses.

Address computations for programs with cyclic or block-cyclic distributions (such as those included in high-performance Fortran) can be very elaborate. When multiple messages arrive, it is sometimes advantageous to compute these addresses on the sender node.<sup>6</sup> Instead of first receiving a message and then copying the data, the program immediately deposits the data into its final location.

The sender transmits address-data pairs. On the receiver node, a message is read word by word, and the address determines where the data is locally stored. The gates make this step easy: The address is moved from a gate to a register; then the data value is moved from the gate to the memory location determined by this register.

**W**e have shown that programs generated to implement systolic algorithms contain a smaller number of memory-access operations than programs based on memory communication. This fact has been mentioned before, but we are the first to demonstrate this characteristic by measuring and monitoring the execution of real programs on a real system. We also observed something surprising: Not only is the number of load or store operations reduced when using systolic communication, but the number of control-flow operations is affected as well. This is due to the systolic program's more regular structure. The programs produced by our parallel-program generator using memory communication contain more conditional tests, which mainly determine when and how to transmit or receive data.

We also found that systolic programs exhibit a higher degree of instruction-level parallelism. The multioperation instruction format is employed more frequently for systolic programs than for memory-based programs, and multioperation instructions in systolic programs average more operations. Both observations explain why systolic program versions take fewer cycles to execute; this performance

advantage is reflected in more efficient processor resource allocation.

Finally, to capitalize on the reduced number of memory accesses, there must be an operand source independent of the memory system. In the iWarp case, the systolic gates act as a register-mapped window to the communication system and provide this additional operand source. Systolic programs use fine-grained (small message size) communication effectively via these gates; from approximately 10 to 20 percent of all operands/results are either supplied by or written to a gate.

In summary, a parallel system that aims to support fine-grained programs must identify how operands are supplied to functional units. Systolic communication requires a direct path from the communication system to the functional units. If data supplied by the communication system can be read or written independently of the memory system, then this operand source enables a higher degree of instruction-level parallelism. ■

## Acknowledgments

We appreciate the efforts by other members of the iWarp team at Carnegie Mellon and Intel. We thank Sonia Singh for assisting in the retarget of the systolic program generator.

This article was supported in part by the Advanced Research Projects Agency (ARPA) Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by ARPA/CMO to Carnegie Mellon University. It was also supported in part by ARPA/CSTO monitored by the Space and Warfare Naval Systems Command under contract N00039-93-C-0152, and by the Air Force Office of Scientific Research under Contract F49620-92-J-0131.

The views and conclusions contained in this article are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Government.

## References

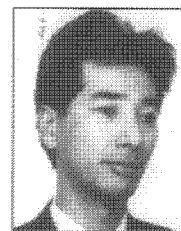
1. S. Borkar et al., "Supporting Systolic and Memory Communication in iWarp," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, May 1990, pp. 70-81.
2. S. Borkar et al., "iWarp: An Integrated Solution to High-Speed Parallel Computing," *Proc. Supercomputing 88, Vol. 1*,

IEEE CS Press, Los Alamitos, Calif., Order No. 882, Nov. 1988, pp. 330-339.

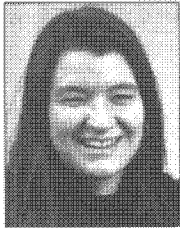
3. H.B. Ribas, "Automatic Generation of Systolic Programs from Nested Loops," doctoral dissertation, Carnegie Mellon Univ., Pittsburgh, June 1990.
4. P.S. Tseng, "A Parallelizing Compiler for Distributed-Memory Parallel Computers," doctoral dissertation, Carnegie Mellon Univ., Pittsburgh, May 1989.
5. R. Cohn et al., "Architecture and Compiler Trade-offs for a Wide-Instruction Word Microprocessor," *Proc. Third. Int'l Conf. Architectural Support for Programming Language and Operating System (ASPLOS III)*, IEEE CS Press, Los Alamitos, Calif., Order No. 1936, Apr. 1989, pp. 2-14.
6. J. Stichnoth, D. O'Hallaron, and T. Gross, "Generating Communication for Array Statements: Design, Implementation, and Evaluation," *IEEE Parallel and Distributed Computing*, Vol. 21, No. 1, Apr. 1994, pp. 150-159.



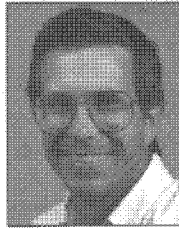
**Thomas Gross** is a faculty member in the School of Computer Science at Carnegie Mellon University. In 1983, he received his PhD in electrical engineering from Stanford University, where he worked on the MIPS project. Since joining Carnegie Mellon, he has been involved in the design and implementation of parallel systems. He is also interested in the practical aspects of compilers, especially code generation, scheduling, and debugging.



**Atsushi Hasegawa** is an engineer in the Microcomputer System Engineering Department at Hitachi Microcomputer Systems, where he designs microprocessors. From 1991 to 1992, he worked at Carnegie Mellon University as a visiting scientist. His research interests include microprocessor architectures, massively parallel processing, and performance measurement. In 1979, he received his BS degree in computer science from the University of Electro-Communications in Japan.



**Susan Hinrichs** is a doctoral candidate in the School of Computer Science at Carnegie Mellon University. Her primary research interests include parallel architectures, in particular, compilation and optimization of communication for distributed memory machines. In 1988, she received her BS degree in computer science from the University of Illinois.



**David R. O'Hallaron** is a faculty member in the School of Computer Science at Carnegie Mellon University. He received his PhD in Computer Science from the University of Virginia in 1986. From 1986 to 1989, he worked at the GE Research and Development Center. In 1989, he joined the faculty at Carnegie Mellon, where he works on architectures, tools, and applications for high-performance computer systems.



**Thomas Stricker** is a doctoral candidate at the School of Computer Science at Carnegie Mellon University. His research interests include communication architectures for parallel computers and high-speed networking. Before entering the doctoral program, he worked at IBM Research in Yorktown Heights, New York. In 1988, he received his BS degree in computer science and engineering from the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland.

Readers can contact the authors at the School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213; e-mail [droh@cs.cmu.edu](mailto:droh@cs.cmu.edu).

COMPUTER

## Parallel-program generators

A parallel-program generator uses a high-level program description to map computations and data onto a parallel system's nodes. At each node, a program *without* explicit communication is turned into a one *with* explicit communication. Although a human programmer may assist with directives or hints, the parallel-program generator is responsible for all parallelism management details.

Here we use matrix multiplication to illustrate the fundamental differences in data movement and mapping between systolic and memory communication. Figure A shows the conventional sequential algorithm to compute  $C = C + AB$ . Both the systolic and memory communication programs compute the same result, given  $A$  and  $B$ , but each program gets this result in a significantly different order.

Basically, both programs multiply two  $N \times N$  matrices on an  $N$  processor array. Both multiplication schemes can easily be extended to handle matrices that do not exactly fit on the parallel system.

**Memory communication.** The memory communication program follows the standard data-parallel paradigm and uses memory communication to move data to the node that performs the computation. The matrices are divided, and every element is assigned to a node. The nodes that "own"  $C$  elements are responsible for computing  $C$  values and so must fetch nonlocal data needed to compute their  $C$  elements. The program proceeds in alternating phases

of communication and computation. A node first sends data needed by other nodes or receives the data it needs. Then it updates all  $C$  elements stored on this node.

In this case, matrices are divided by rows. The  $p$  rows from  $A$ ,  $B$ , and  $C$  are assigned to node  $p$ . Therefore, node  $p$  is responsible for computing the  $p$ th row of  $C$ . Here is a pseudocode for the computation executed by node  $p$ :

```
for (k = 0; k < N; k++)
  if (k == p)
    broadcast row p of B
    copy row p into Brow
  else
    receive row k of B into Brow

for (j = 0; j < N; j++)
  C[p][j] = C[p][j] + A[p][k] * Brow[j];
```

Figure B illustrates the first two steps for the multiplication of two  $3 \times 3$  matrices. First, node 0 broadcasts the row of  $B$  stored on node 0 to all other processors. Next, all nodes add

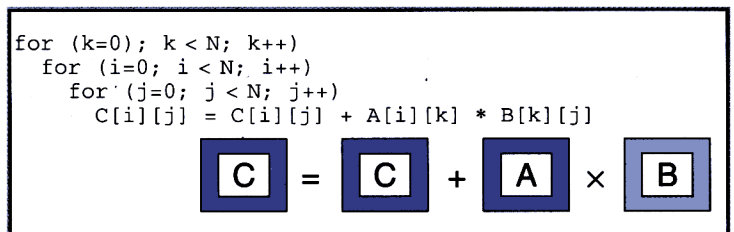


Figure A. Matrix multiplication.

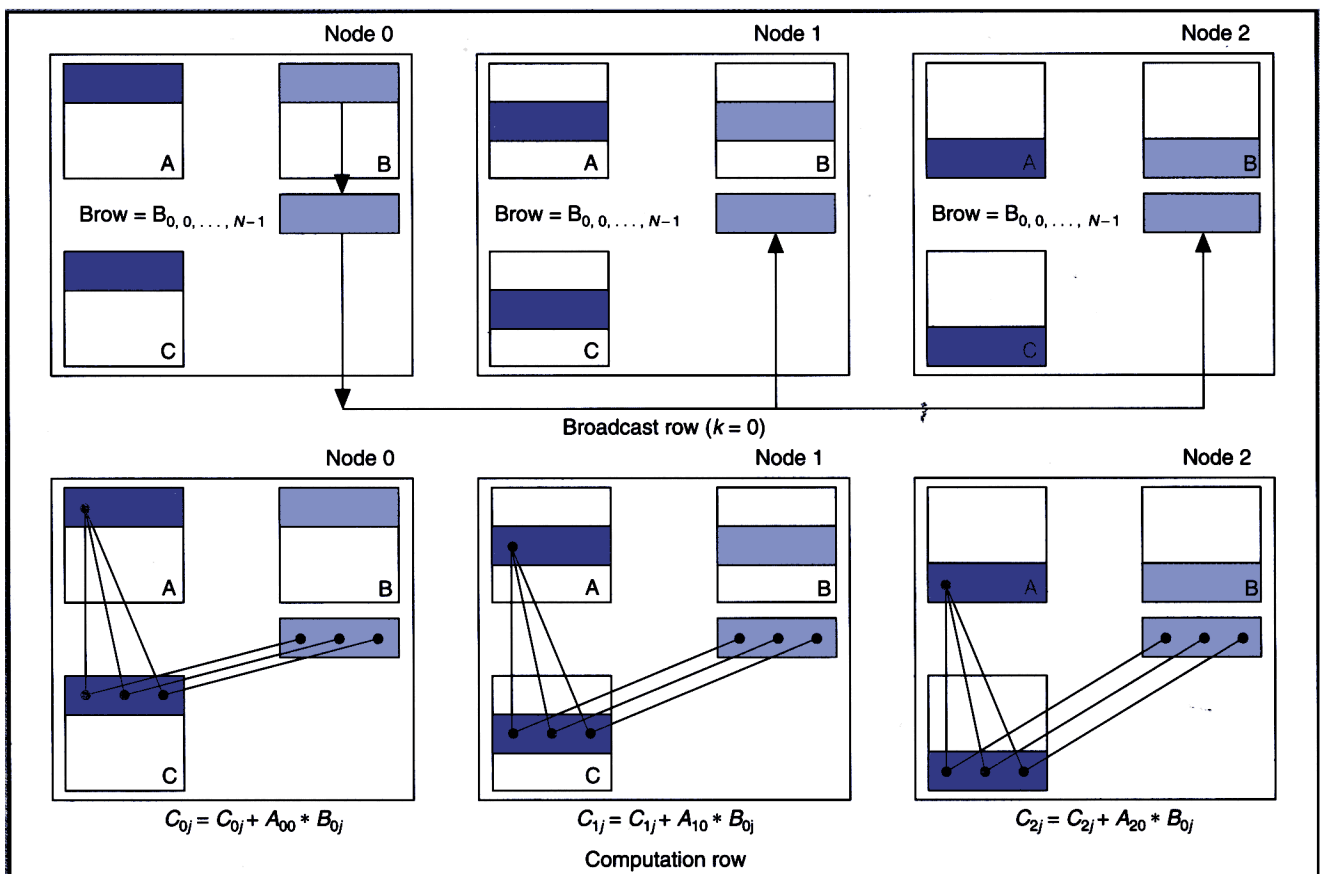


Figure B. Parallel matrix multiplication based on memory communication.

$A[i][0] * B[0][j]$  into the  $C[i][j]$  elements of  $C$  that they own. Then node 1 broadcasts row 1 of  $B$ , and so on. After all rows of  $B$  have been broadcasted,  $C$  (distributed over all nodes) contains the results of  $C + AB$ .

All communication is performed from memory to memory: a row of  $B$  (stored in memory) is sent to all other nodes, where it is also stored in memory.

**Systolic communication.** Basically, the systolic program pumps data through the parallel system during computations instead of reading all operands from memory.<sup>1</sup>  $A$  and  $B$  are still assigned to local memory, but the elements of the  $C$  matrix flow through the system. Unlike memory communication, no single node is responsible for completely computing the value of a particular element of  $C$ . Instead, each node partially computes every  $C$  element that passes through it.

Again, matrix  $B$  is divided by rows, but matrix  $A$  is divided by columns. Each node  $p$  is assigned the  $p$ th row of  $B$  and the  $p$ th column of  $A$ . The initial values of  $C$  are pumped into node 0. Below is the pseudo code for node  $p$ :

```

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    receive partial result for C[i][j] into c
    c = c + A[i][p] * B[p][j];
    send c to node p + 1;

```

Figure C shows the computation of  $C[0][0]$  by the array nodes. The partial result  $c + A[i][p] * B[p][j]$  is immediately passed to the next node, where it is used as an operand. The output of the last node is the new value of  $C$ .

The inner computation loop of the memory communication program performs three memory accesses but no communication, for each iteration. The systolic program performs one memory access and two communication steps per inner-loop iteration. So, in tight computation loops, the systolic program takes advantage of communication system bandwidth in addition to memory bandwidth.

We omitted some details in the above presentation. First, since only parts of the arrays are stored locally, an additional level of mapping takes place so that memory is only allocated for locally resident data.<sup>2</sup> Second, multiple rows or columns are usually stored on a single node, further complicating the loop structure and address arithmetic. Third, unrolling the bodies is usually difficult unless the bounds of various matrices are known during compile time.

## References

1. H.T. Kung, "Why Systolic Architectures?" *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37-46.
2. H. Zima and B. Chapman, "Compiling for Distributed-Memory Systems," *Proc. IEEE*, Vol. 81, No. 2, IEEE Press, Piscataway, N.J., 1993, pp. 264-287.

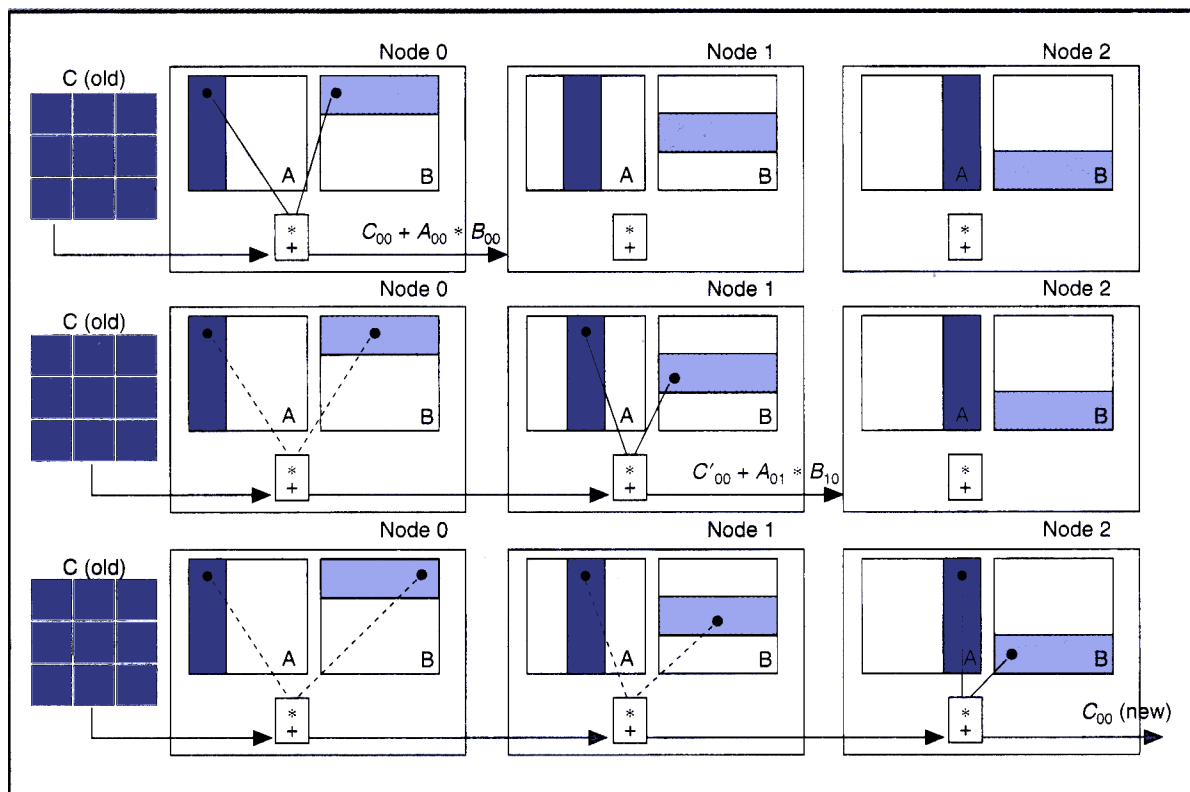


Figure C. Parallel matrix multiplication based on systolic communication.