The Impact of Communication Style on Machine Resource Usage for the iWarp Parallel Processor

T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker

November, 1992
CMU-CS-92-215

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Supported in part by the Defense Advanced Research Projects Agency, 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 DARPA/CMO to Carnegie Mellon University.

Keywords: Processor architecture design and evaluation, parallel systems, inter-processor communication architecture, instruction level parallelism

Abstract

Programs executing on a private-memory parallel system exchange data by explicitly sending and receiving messages. Two communication styles have been identified for such systems: memory communication (each message exchanged between two processors is buffered in memory, e.g. as in message passing) and systolic communication (each word of a message is transmitted directly from the sender processor to receiver processor, without any buffering in memory). The iWarp system supports both communication styles and therefore provides a platform that allows us to evaluate how the choice of communication style impacts the usage of processor resources.

Parallel program generators map a machine independent description of a computation onto a private-memory parallel system. We use two different parallel program generators that employ the two communication styles to map a set of application kernels onto iWarp. By using tools to generate the parallel programs, we are able to obtain realistic data on the execution of programs using the different communication styles. This paper reports on measurements of instruction format usage, the utilization of the communication ports (gates), and instruction frequencies on the iWarp system. It is a first step towards understanding how features and capabilities of parallel processors are actually used by parallel programs that have been mapped automatically.

1 Introduction

Parallel systems consisting of processors with private memories rely on the explicit exchange of messages for communication. Programs executing on such systems use some form of "send" and "receive" to transfer data from one processor to another. The details of the communication operations supported by different parallel systems differ in many aspects, but we can identify two communication styles: *systolic communication* and *memory communication*[9]. These styles differ in how messages are generated and consumed.

In systolic communication, a message is generated word–by–word, and each word is transmitted immediately (i.e., on the fly, without explicit buffering) to the destination processor. In memory communication, the complete message is generated, stored in memory and then transmitted to the destination. Since the complete message is generated before it is transmitted, the words of a message can be generated in any order for memory communication, whereas for systolic communication, the words of a message must be produced in the order they are sent. Since the sender and receiver processors operate independently, they can use different communication styles; message passing is an example of memory communication where both the sender and the receiver use memory communication (i.e., they buffer the message in memory before sending and after receiving).

Different communication styles require different architectural support. For example, efficient systolic communication requires direct program access to the communication system (e.g., the ports that connect a processor to its neighbors). On the other hand, the choice of communication style influences how operands are accessed. Since systolic communication allows the processor to directly retrieve operands from the communication system, fewer load operations must be executed. In this case the communication system is another source of operands (in addition to memory and registers), and such operands may have an effect on the amount of instruction-level parallelism that can be exploited.

To evaluate the impact of the communication style, we investigate the execution of a set of programs on the iWarp system. The program for each node contains explicit communication operations, but it is generally recognized that writing programs with explicit communication is difficult and error prone. For this reason, a number of parallel program generators (or parallelizing compilers) have been implemented[22, 1, 14, 2, 3, 12], and the development of such tools is still an highly active area. These tools ease the task of programming significantly since the tool maps the data and the computation, described in a machine independent format, onto the specific parallel machine. That is, the tool is responsible for all the details of management of the parallelism, although a human programmer may assist the tool with directives or hints. Any serious evaluation of a uni-processor has to be based on compiler generated code, and similarly, an evaluation of the features of a parallel system must consider programs that have been mapped automatically.

For this evaluation, the programs are mapped by two parallel program generators onto the system, one for each communication style[1, 2]. The iWarp system was developed by Intel Corp. to support both memory communication and systolic communication[13, 8, 9]. Since both communication styles are supported, the iWarp system provides a unique opportunity to empirically evaluate how parallel programs based on these communication styles use the processor resources. Such information can be used in many ways: to evaluate if there is any correlation between communication style and instruction frequencies, to evaluate how operands are stored (in registers or in memory), to assess which machine resources are used (or not used) by specific parallel programs, and to conclude which machine features are essential if a future processor aims to be a host for parallel programs that employ memory and (or) systolic communication.

The structure of the paper is then as follows. Section 2 briefly describes the hardware platform as well as the two program generators. Section 3 describes our results for our set of four common numerical application kernels (matrix multiplication, LU decomposition, QR decomposition, and successive over-relaxation). A comparison of the two communication styles invites a discussion of the relative merits of these two styles, and we address this issue in Section 4. Section 5 summarizes the key points of the paper and contains our conclusions.

We provide one detailed example of how the communication style influences the mapping of computations onto a parallel system. The appendix describes this example (matrix multiplication) in more detail. Although both versions of matrix multiplication compute the same result, they differ in the way the computation is mapped onto the array; selection of the communication style has far–reaching consequences for the program structure. A reader unfamiliar with parallel program generators is invited to read the appendix before proceeding to Section 2.

2 Background

2.1 iWarp system

iWarp is a single–chip VLSI processor developed by Intel Corp[13, 8, 9]. It contains a computation agent (20 MFLOPS single precision, 20 MIPS) and a high throughput (320 MBytes/s), low latency (200 ns) communication agent for transferring data between other iWarp processors. An iWarp system is a 2D torus of iWarp nodes, ranging in size from 4 nodes to 1024 nodes.

Figure 1 depicts a block-level sketch of the iWarp processor. The communication agent of each processor contains a number of FIFO queues, each 8 words deep. Any data received from neighboring nodes are stored in one of these queues until the program is ready to process these data. The heads of the queues can be mapped to special registers in the register file called *systolic gates*, or simply *gates*.

Gates can be used as instruction operands like any other register in the register file, *and with the same access time*. If a gate is used as an input operand, a data word is removed from the head of the associated queue and passed to the functional unit; if the queue is empty, the instruction spins until a data word arrives. If a gate is used as an output operand, the data word is placed at the tail of the associated queue; if the queue is full, the instruction spins until a data word leaves the queue. The key points are (1) access time to the communication system via the queues is the same as a register access, and (2) word–level flow control is handled automatically by the hardware.

The iWarp instruction set contains two types of instructions:

1. *Short instructions* contain a single operation and control a single functional unit. Examples

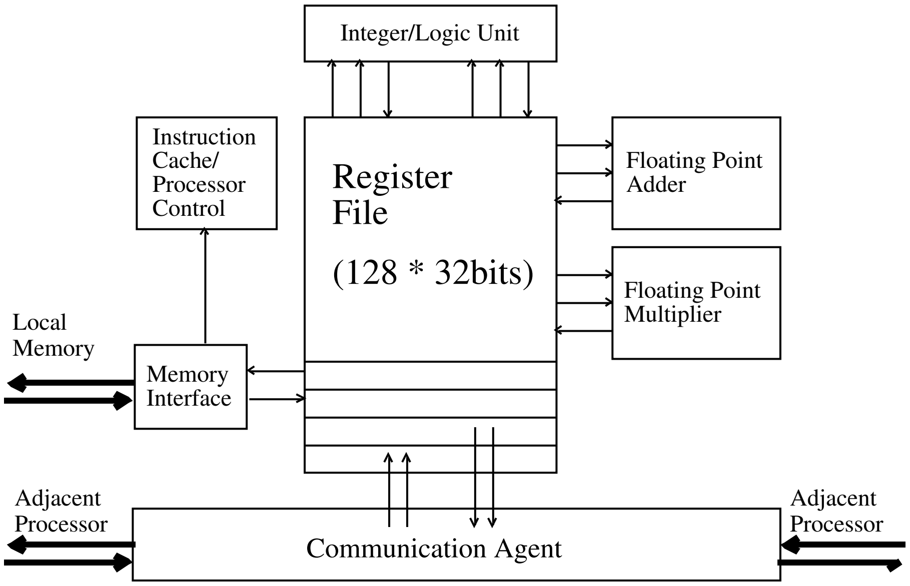


Figure 1: iWarp processor

include load, store, floating–point add, floating–point multiply, and integer add. Loads, stores, and the floating point operations execute in 2 cycles. Most other short instructions take 1 cycle. Short instructions are encoded in 32 bits.

2. *Long instructions* control multiple functional units and therefore contain multiple operations. A long instruction word (LIW) is encoded in 96 bits, and the execution time is the maximum execution time of each of the constituent operations. A single 2–cycle LIW instruction can perform a load and a store (or two loads) with auto-increment of the address register, a floating point add, and a floating point multiply, as well as decrement and test a counter (to implement loops). An LIW contains a maximum of 5 operations.

The iWarp node contains several features to support memory communication. Each node provides a high bandwidth to local memory. When using the LIW format, there can be one load for every floating point operation, for a peak memory bandwidth of 80MBytes/s. This bandwidth (and the flexibility of the LIW format) ensure that the processor is not starved for data, even if no data are supplied by the communication system. E.g., a scalar dot product can proceed at the peak floating point rate. To move medium-sized and large messages directly from the communication system to memory, the iWarp processor contains 8 DMA-like controllers called spoolers. Each spooler can move data from the communication system to the memory at a rate of 40 MBytes/s. This rate matches the bandwidth of the communication buses to the neighbor processors. Up to 4 spoolers can proceed at full bandwidth, i.e., the total memory bandwidth is 160 MBytes/s, but in this case, the spoolers steal all memory cycles from the computation agent. Spoolers are attractive when multiple messages arrive at the same time, or if message arrival occurs asynchronously with respect to program execution. However, the setup of a spooler requires the execution of instructions. If only one or two messages must be moved to memory, and if the message arrival is synchronized with program execution, then moving the message to memory via explicit store operations is faster.

2.2 Sample programs

We analyze the single–node resource usage for four standard linear algebra application kernels: matrix multiply (MM), LU decomposition, QR decomposition, and successive over relaxation (SOR). There are two different parallel programs for each kernel: one program based on memory communication, with a structure as discussed in Section A.1, and one based on systolic communication as discussed in Section A.2. We refer to the programs based on the systolic communication style as *systolic programs*; we refer to the programs based on the memory communication style as *memory–based programs*. All of the programs, except for systolic SOR, are produced automatically from a high–level description of the computation using two program generators developed locally [2, 5, 1]. (Due to the structure of SOR, the parallel program generator can not automatically create a systolic program.)

2.3 Program generation issues

All programs are generated for a linear array of 16 nodes. The actual iWarp system is organized as a 2-dimensional torus, but it is easy to map a linear array onto the torus. We also investigated in a separate study the tradeoff between producing code for a linear array and then mapping the linear array onto the torus versus directly producing code for a torus. For applications and problem sizes like the ones discussed here, the linear array is actually superior for the memory communication style. We do not know of any tool that maps computations automatically onto a 2–dimensional torus using systolic communication. Furthermore, the innermost loops for the programs based on systolic communication are tight, so the current systolic program generator provides a realistic picture.

There are many models of memory communication, including many forms of message passing, used by private–memory computers[6, 7, 19]. These models differ in their functionality and overhead. Our parallel program generator uses only neighbor–to–neighbor communication and a high-speed broadcast primitive. A broadcast message is stored and forwarded on a word–by–word basis. This allows the sender to operate at full speed, solely determined by the communication bandwidth. Since all the connections used by a parallel program can be setup at load time (and the parallel program generator takes advantage of this feature), there is no protocol overhead associated with any message. There may be a place for more sophisticated memory communication schemes, but for the regular programs used in this study, the zero-overhead "protocol" is adequate and produces the best overall results.

The memory communication style is more general and more intuitive to a human programmer. At this time, there are several program generators based on memory communication that can handle a large class of computations [22, 1, 14, 2, 3, 12]. Research into systolic algorithms has produced a number of efficient systolic algorithms for specific computations. Methods to automatically transform algorithms into systolic programs have also been developed[2, 10, 18], but at this time, these methods work only for a subset of the computations that can be handled by program generators based on memory communication.

2.4 Profiling and measurement strategy

Each parallel program generated by the respective parallel programming tool consists of one C program (called a *node program*) for each node in the system, plus a program that sets up the connections between the node programs. Each node program is compiled using a conventional single–node compiler, loaded onto the array, and executed. Connection setup is part of loading (and hardly contributes to execution time) and therefore not included in our measurements. All programs are compiled to use single precision arithmetic for computations involving only `float` values; this format is sufficient for our application domain. The quality of the single–node compiler influences our measurements, and therefore, we consider two different optimization levels.

First, each program is compiled by the production C compiler (Release 3.0) developed by Intel Corporation for iWarp. We call this the *standard* version of a program. The production compiler performs a range of conventional local and global optimizations on the intermediate code and instruction scheduling in the backend to exploit instruction level parallelism.

The production compiler performs only limited instruction scheduling; inter-block (i.e., global) code scheduling is effective only for the simplest loops. To gauge the effect of optimizations that are not yet included in the production compiler [4] and to better approach the usage model for the domain of signal processing, we created an additional version of each program. For this version the inner loops have been checked by hand to verify that these loops are tightly packed and operate at peak floating point performance. We call these the *optimized* programs.

The code generated by the compiler for each node is then annotated to gather profiling information to determine how often each basic block is executed. During execution, each node program, executing on a separate node, keeps track of how often each basic block is entered. After termination of the user code, the runtime system writes the history information into a log file on the front–end computer. An auxiliary program combines the basic block frequency counts with the object code to obtain the information presented in the next section.

Practical program generators often produce a single program for all nodes to reduce the compile time, and this program is replicated for all nodes. It contains node–specific run time tests that determine the position of the node in a virtual grid and that control how boundary situations are handled. If such a test ends up in a loop, the overhead imposed by these tests can be significant. We have taken care to ensure that these simplifications do not disturb our measurements by creating several specialized versions of the node program that remove the run time tests.

All programs are given input matrices of size 160×160 and are run on 16 iWarp nodes, so each node is responsible for 10 rows. We present numbers from a single representative node in each case, because there is almost no program variation between the different nodes due to the regular structure of the programs. This input size seemed roughly representative of the common case. While larger matrices would hide some of the start up overheads, this would present a

		Profiled Time	Actual Time	Percentage Difference
systolic	standard	0.145	0.142	2.06
	optimized	0.0274	0.0270	1.45
memory based	standard	0.103	0.099	3.74
	optimized	0.076	0.071	6.57

Table 1: Differences between profiled time and actual execution time for the matrix multiplication implementations. Times are in seconds.

	Memory based		Systolic	
program	standard	optimized	standard	optimized
MM	81	113	57	302
SOR	70	89	62	104
LU	24	41	51	145
QR	52	150	88	174

Table 2: MFLOPS measured on a system of 16 nodes for all implementations of the test programs. The maximum rate possible is 320 MFLOPS.

picture that is too optimistic in many cases.

This overall approach of static profiling has some advantages and limitations, which we list here to help the reader understand what exactly is being measured. First, by basing our analysis on the frequency information for each basic block of the user program, we cannot include any time spent in system libraries (e.g., the C math library) or the runtime system. For the programs that we investigate here, the fraction of time spent in the system libraries is minimal, and no time is spend in the runtime system.

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

To get a handle on any potential difference between the time reported on a node and the overall execution time, we compare in Table 1 these two times for maxtrix multiplication. Sources of discrepancies (in addition to instruction cache misses) are delays due to the competition for network resources and stalls between adjacent instructions due to incomplete interlocks between the functional units. Table 2 shows the measured MFLOPS for all programs.

3 Results and evaluation

Systolic communication promises two measurable benefits[9]:

- Increased instruction–level parallelism.
- Reduced access to local memory.

Dynamic measurement of instruction and operation frequencies provides the opportunity to empirically evaluate these benefits. In this section we present the instruction profiling results for the programs discussed above (Sections 3.1 and 3.2) and discuss which microarchitectural features are responsible for our observations (Section 3.3).

Recall that we refer to programs based on systolic communication as *systolic programs* and to programs based on memory communication as *memory-based programs.*

3.1 Instruction set usage

The two iWarp instruction formats (short and LIW) allow us to directly measure the impact of the two communication styles on instruction–level parallelism. Figure 2 shows the percentage of the executed operations that were performed by LIW instructions.

Figure 2 shows that a properly optimized program can use the LIW instructions effectively. Although on average only 5% of the instructions in the optimized systolic object code are LIW instructions, 88% of all operations are executed in an LIW instruction. This also holds true on average for the optimized memory-based programs, where LIW instructions account for 10% of the instructions in the code but execute about 52% of the operations. Therefore, LIW instructions that are inserted in the code are frequently used at runtime. The small percentage of LIW instructions in the code also means that the average instruction length (and code size) does not grow excessively. For the optimized systolic programs, the average instruction is 1.1 words long, and for the optimized memory-based programs, the average instruction is 1.2 words long.

An LIW instruction can be used to compute multiple results in parallel, on different functional units, but it can also be used to copy data from one register to another. Up to 4 registers (6 if registers can be organized in odd/even pairs) can be copied by a single LIW instruction, and this is by far the fastest way to copy registers. Since the functional units are busy in this case executing operations scheduled by the compiler, we include these moves in our count of operations. This explains why for MM and SOR, the standard memory-based programs execute a higher percentage of operations in LIW instructions than is observed for the standard systolic programs. Otherwise, the number of operations per instruction is higher for programs based on systolic communication than on memory communication.

Figure 3 shows the average number of operations per instruction for each program. As you would expect, the programs that execute a higher percentage of operations in LIW instructions also execute a larger number of operations per instruction. This proves that a reasonable number of operations are packed in each LIW instruction.

The optimized systolic programs show more pronounced improvements over their standard counterparts than do the memory-based programs. There are two principle but related reasons for

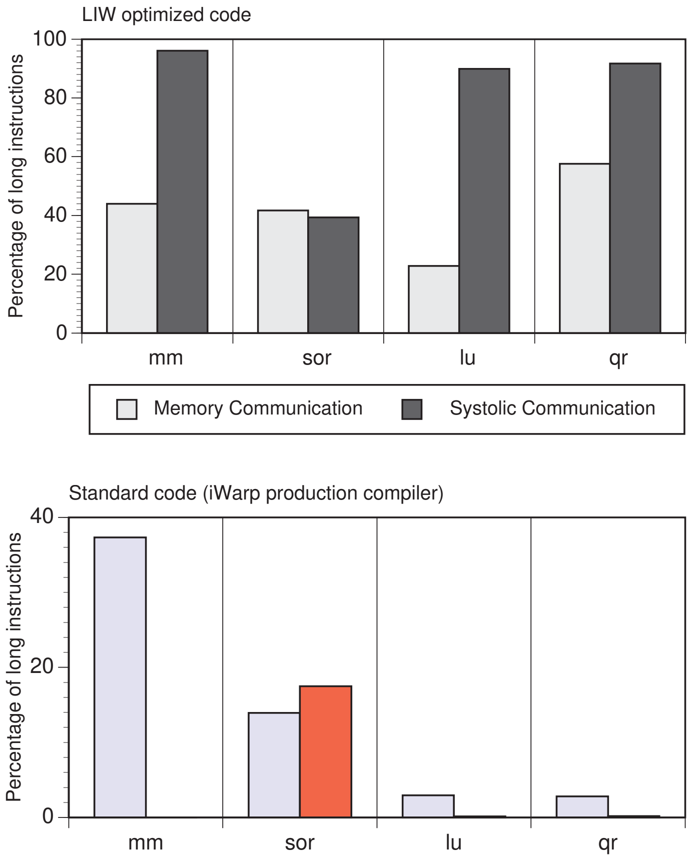


Figure 2: Percentage of operations executed in LIW instructions, optimized (top) and standard (bottom)

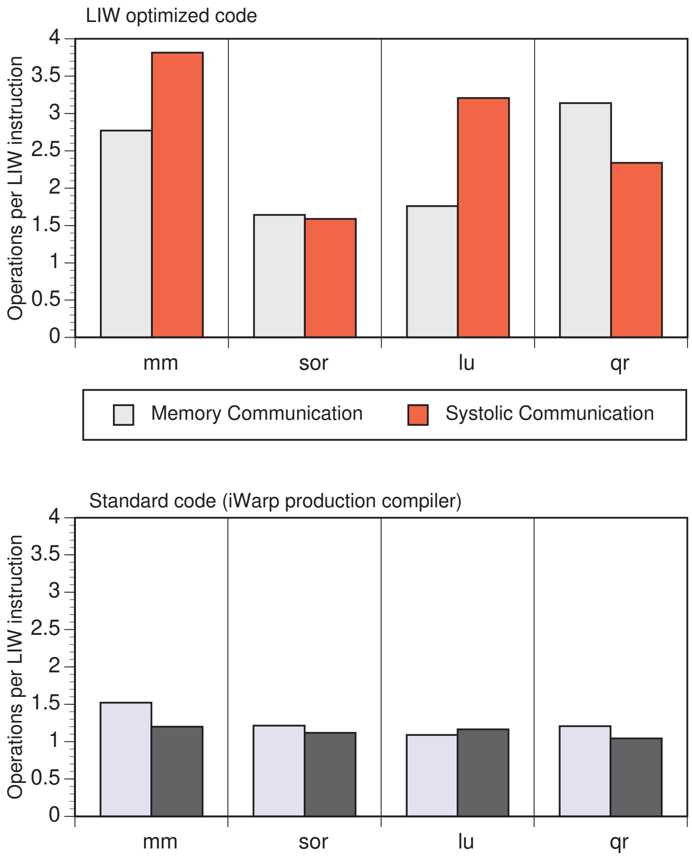


Figure 3: Operations per instruction, optimized (top) and standard (bottom).

this. First, the systolic programs are compact, including both computation and communication in the same loop. Thus, in the optimized programs, a large number of copy operations as well as unnecessary loads can be removed from the loop. Second, communication and computation are coupled; this reduces the number of operations even more, because operands for the floating point multiplier and adder can be supplied from the communication system. Operands can be read from or written to the communication system directly, without first copying them to a register. However, this effect may not be so pronounced, because the value read from the communication system is popped off the network queue, so the program must still copy the value to a register if it wants to use the value multiple times.

3.2 Operation distribution

Figure 4 provides another picture of how the processor resources are used by the programs. This figure shows the operation frequencies for the programs. The top graph shows the frequencies for the standard programs, and the bottom graph shows the frequencies for the optimized programs. The operations are divided into five major classes:

FP Ops: floating point operations Any operation that uses the floating point multiplier or adder.

Ld/St: memory operations Loads a value from or stores a value to local memory.

Addr. Arith.: address arithmetic ALU and load literal operations that calculate memory addresses, including shifts, bit field operations, and count leading zeros.

Moves: ALU operations that copy a value from one register to another register[1].

Control: Control flow instructions like branches, jumps, procedure call, and return are included in this group.

In Figure 4, all integer operations are counted as address computations. This is not absolutely correct, because some integer operations are needed solely to compute loop trip counts. However, the number of such operations is negligible. Figure 4 shows that memory-based programs perform far more memory accesses, because memory-based programs must retrieve all operands from memory and store intermediate results to memory. Related to memory operations, Figure 4 shows a larger number of address computations for memory-based programs (for both standard and optimized programs). Since memory-based programs access memory more frequently, they must calculate more memory addresses, although optimizing memory-based programs reduces the number of memory accesses and associated address computations. For systolic programs, the number of memory accesses and related address computations is low to begin with and is even lower after optimization.

The systolic programs use far fewer control flow operations because communication and computation loops are combined. The memory-based programs also contain more tests to

[1]The floating point units can also copy values between registers. Our profiling tool counts these moves as floating point operations since this floating point unit is busy.

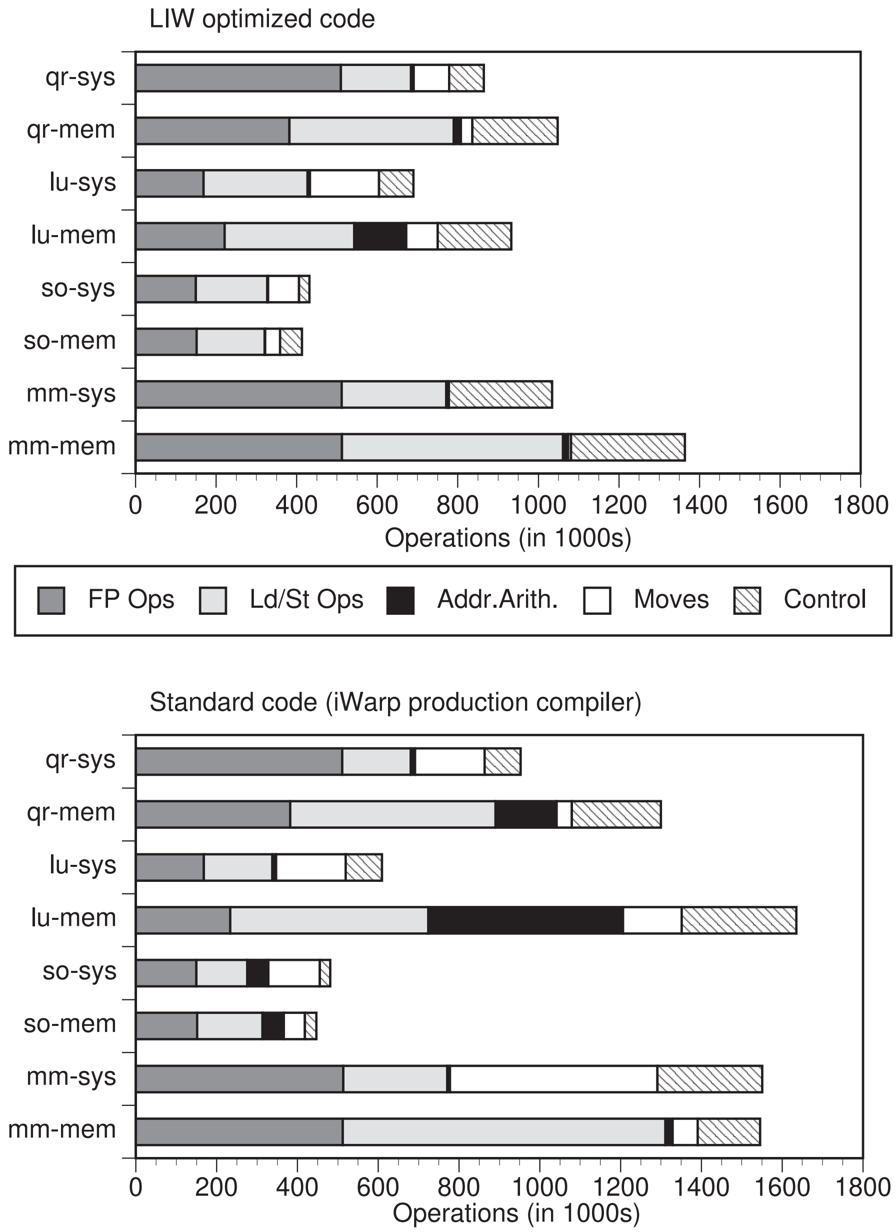


Figure 4: Classification of operations executed, optimized (top) and standard (bottom) programs.

determine the ownership of data. Since data flows through the systolic program, it does not need to calculate the addresses of data that are to be distributed, it only needs to read the next value from the neighbor node.

3.3 Program access to communication system

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 register file of the processor, as shown in Figure 1. There is no operating or runtime system overhead involved in reading or writing a gate. Use of the gates as a source or destination of operands provides the mechanism that reduces the number of load or store operations. Figure 5 shows what percentage of all operands are supplied by a gate or written to a gate, for the different instances of the applications. Operands are read from a gate about as often as they are written to a gate.

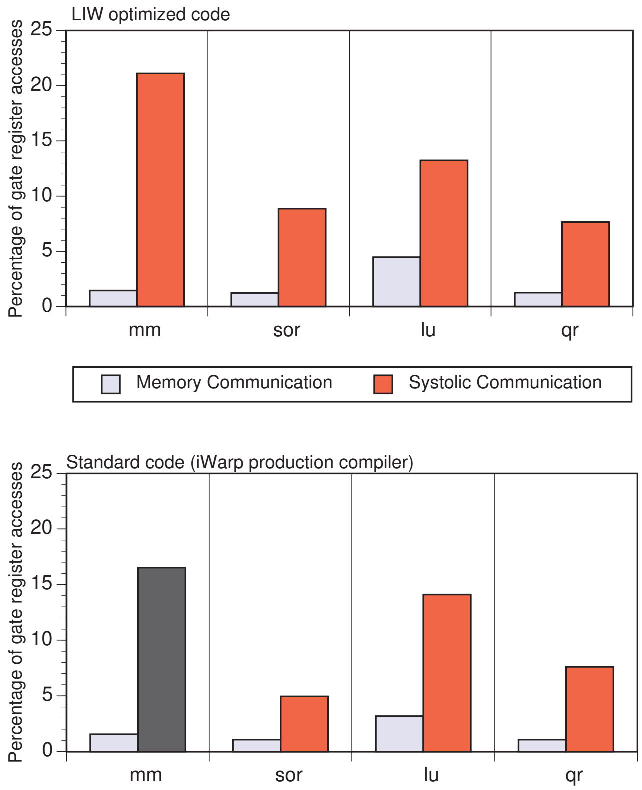


Figure 5: Percentage of register accesses through the gates

iWarp is a load/store architecture; the operands for all computations must be retrieved from a register or gate, and the destination for all computations is either a register or a gate. As can be seen in Figure 5, only optimized systolic programs use the gates for a significant fraction of computation operands. The standard systolic programs perform the same number of gate accesses, but there are many more operations, so the percentage of gate accesses is much lower. Only the optimized systolic programs contain instructions that write the result of a floating point computation directly to a gate. In the standard program, the results are first written to a general register and then copied (by a move instruction) to a gate. The elimination of such unnecessary moves by the optimization is another reason for the pronounced improvement of the systolic programs.

The absolute number of gate operands in the memory-based programs is not affected much by optimizations. Since the communication and computation are in separate loops, the operands must still be fetched from memory for the computation. Therefore, the percentage of gate operands does not change significantly.

4 Discussion

The data presented in Section 3 compare the effect of memory and systolic communication for programs mapped by two program generators. The limitations must be understood when using the above data to compare memory and systolic communication. The program generators described in Section 2.3 do not cover the entire space of parallel programs. There exist a large number of efficient systolic algorithms that have not been developed by a program generator, and strategies to block or tile programs are being developed by various researchers[21, 20]. We are not aware of any tool that automatically maps programs onto a distributed memory machine using blocking or tiling, but we can use some programs from a library that is based on blocking[16].

The key idea of blocking or tiling is to divide the 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 the program generator based on memory communication but should have better communication characteristics due to the increased locality.

The systolic communication model attempts to utilize network bandwidth instead of memory bandwidth, while the memory communication model performs computation only on values stored in memory. This difference reveals itself in two ways. First, the memory-based program must retrieve all of the computation arguments from memory, but the systolic program can also retrieve arguments from the network. Second, the memory-based program must initially store all of its messages in memory, but the systolic program avoids those memory accesses entirely because it never stores messages.

As an example of the computation phase differences, consider the statement that forms the inner kernel of memory-based matrix multiplication and LU decomposition. This kernel requires all arguments to be fetched from memory:

```
x[i] = x[i] + p*y[i]
```

it requires three memory operations and two floating point operations. The systolic inner loop passes the `x` vector along:

```
tmp = receive()
tmp = tmp + p*y[i]
send(tmp)
```

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

Whether the systolic program can surpass the performance of the memory-based inner loop depends on the balance of network and memory access provided by the processor, and this depends on how network accesses are implemented. Two forms of network access are memory–mapped network queues and register-mapped network queues. One example of the memory–mapped approach is found in the CM-5[17]. In this approach, network access requires memory bandwidth, so trading off memory access for network access makes little sense. In this example, trading one memory access for one memory–mapped network access saves no memory bandwidth unless the memory–mapped network access uses memory resources that are separate from the resources used by ordinary memory opertions.

The register-mapped approach avoids the memory bottleneck by feeding directly into the register file. This is the approach taken by iWarp. Since the network queues are seen as registers, each floating point operation can read two network values and write one. Since floating point operations execute in parallel with memory operations, the network can be accessed in parallel with memory.

The memory–mapped approach has the advantage that the network interface can be inserted without completely redesigning the processor. In [11], Henry and Joerg review the design space for fast implementations of message passing. They reach the conclusion that to achieve good performance the network interface must be on the chip, and they argue that mapping the network queues into on-chip cache in easier than mapping the network interface into the register file. With a reasonably realistic memory hierarchy, it is not clear that memory–mapping the interface into the cache avoids any problems. Also, if the system designer is going to the trouble of changing the processor chip, it is probably worth considering going to the extra effort of avoiding memory to gain the possibility of additional instruction level parallelism and the ability to support systolic algorithms.

The number of instructions a processor can issue in parallel is limited by the number of functional units, the amount of memory bandwidth, and the amount of network bandwidth that the machine supports. Assuming a procesor can multiply and add in the same cycle, retrieving the arguments from memory or the network is the limiting factor. If a processor could perform three memory accesses in parallel, the memory-based inner loop could be performed in one instruction. On iWarp at most two memory accesses can be performed in parallel, so the memory-based inner loop must use two instructions, while the systolic inner loop can be performed in one instruction.

Not all systolic algorithms gain from the reduction in required memory bandwidth in the inner computation loops. Utilizing this reduction depends on the types and numbers of operations that can be issued in parallel. The block based matrix multiplication algorithm rearranges the loop order, so it uses a slightly different inner loop:

n	simple distribution memory communication	blocked (tiled) memory communication	systolic
128	102	163	291
256	114	209	304
512	120	259	310

Table 3: Measured MFLOPS on a system of 16 nodes for three implementations of matrix multiplication of $n \times n$ matrices. The maximum rate possible is 320 MFLOPS.

```
c = c + x[i]*y[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, but for a machine like iWarp that can issue two memory accesses in parallel, the loops can 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 the size of the problem grows towards infinity, this advantage becomes less significant. In the case of matrix multiplication, the program sends $O(n^2)$ words in messages but performs $O(n^3)$ iterations of the inner loop. However, this effect is still quite noticeable for finite (realistic) arrays.

Table 3 shows the MFLOP measurements for the three matrix multiplication programs (mapped onto 16 nodes). There is a noticeable difference between the systolic and block programs even through 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.

There are several important points to note. First, matrices must be large before the performance for systolic and memory communication comes close. Second, the iWarp processor supports one memory operation in parallel with each floating point operation. The performance of memory communication on processors without this balance (e.g., those that support only one memory access for every two floating point operations) cannot approach the performance of systolic communication. Third, systolic programs are efficient for small input sizes. The performance of blocked programs based on memory communication approaches the performance of the systolic programs only for large input sizes. And if we map matrix multiplication onto a larger system, e.g. with 64 or 256 nodes, even larger matrices are required to obtain the same performance on each node. Finally, since iWarp supports static connection setup, so there is *no* protocol overhead for memory communication.

Another parameter that influences the tradeoffs between memory communication and systolic communication is how the processor and the program handle message arrival. The program generator used in this study and the block matrix multiplication explicitly store the data from the network into memory as described in Section 2.3. Other message passing implementations could move the message data in the background by stealing memory cycles. Performing the communication in the background does not avoid the memory bandwidth bottleneck and still requires time to set up the data transfers. For matrix multiplication the execution times of memory communication (T_{MP}) and systolic communication (T_{sys}) are respectively:

$$T_{MP} = 2n \times (T_s + T_t \times n) + 2n^3/p \times T_l$$

$$T_{Sys} = 2n^3/p \times T_l'$$

where T_s is the start up overhead for each message, T_t is the transfer time send each word, and T_l and T_l' are the times to perform the inner loop. In our program implementation, $T_s = 0$. In an implementation that performs the communication completely in the background $T_t = 0$. If message storing steals no resources from the foreground computation, then message passing contributes only on the order of $O(n)$ time steps asymptotically, but if messages storing requires foreground resources, message passing adds an $O(n^2)$ effect. However, for finite sized arrays, we need to know the values of T_s and T_t to calculate the tradeoff between foreground and background communication.

5 Concluding remarks

We analyzed the execution of parallel programs based on two different communication styles, for a specific input size. The systolic versions of the programs take fewer cycles to execute, and this performance advantage is reflected in the better use of the processor resources.

We see that systolic programs, if using properly optimized code, can use the fine grained communication effectively in the form of gates (approximately between 10 % and 20 % of all operands/results are either supplied by a gate or written to a gate). This direct access to the communication system provides the opportunity to optimize the programs extensively, as seen by the overall reduction in instruction and operation counts.

A comparison of the relative merits of the communication styles cannot be done in isolation. We have shown that the size of the input data, the instruction level parallelism, the available memory bandwidth, and the parallelization strategy all contribute to the effectiveness of executing a program on a parallel processor. For arbitrarily large data sets, the performance difference caused by the communication style diminishes, if the processor supports sufficient memory bandwidth. However, for finite, realistic input sizes, a systolic implementation enjoys a noticeable performance advantage. But it is also well-known that message passing is more general, because there are computations for which systolic implementations cannot currently be generated, even though messages passing implementations can be automatically generated.

However, we can state that these communication styles exhibit different usage patterns, with direct implications for any computer architecture that aims to be a base for both communication styles. The results of this study indicate that systolic programs are able to use the gates more often than the message-passing programs. This results in a lower frequency of load instructions (and a higher frequency of gate accesses). Conversely, message passing programs require higher memory bandwidth but can endure reduced communication performance, because independent computation and communication activities can be overlapped. As with any study based on a finite set of programs, we have to caution the reader not to over-generalize, but our results to date indicate that systolic programs are able to benefit from directly accessing the communication system.

References

[1] Tseng, P. S., "A Parallelizing Compiler for Distributed Memory Parallel Computers", Ph. D. thesis, Carnegie Mellon University, May, 1989.

[2] H. B. Ribas. Automatic generation of systolic programs from nested loops. Technical Report CMU-CS-90-143, Department of Computer Science, Carnegie-Mellon University, June 1990.

[3] D. R. O'Hallaron. The ASSIGN parallel program generator. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 178–185, Portland, OR, April 1991.

[4] Adl-Tabatabai, A., Gross, T., Lueh, G., and Reinders, J., Modelling Instruction-Level Parallelism for Software Pipelining, Proceedings of IFIP WG 10.3 (Concurrent Sytems) Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Jan. 1993

[5] Tseng, P., A Systolic Array Parallelizing Compiler, Journal of Parallel and Distributed Computing, Vo. 9, 1990, pp. 117–127.

[6] Ahmad, I. and Wu, M., Express versus iPSC/2 Primitives: A Performance Comparison, Tech. Report, Syracuse Center for Computational Science, SCCS-77, CRPC-TR91-147

[7] Arlauskas, R., iPSC/2 System: A Second Generation Hypercube, *Proc. 3rd Conf. on Hypercube Multiprocessors*, 38-42, 1988.

[8] Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J. and Webb, J., "iWarp: An Integrated Solution to High-Speed Parallel Computing", Proceedings Supercomputing '88, November, 1988, Orlando, Florida, pp. 330-339.

[9] Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J. and Webb, J., "Supporting Systolic and Memory Communication in iWarp" , Proc. 17th Annual Intl. Symposium on Computer Architecture, May, 1990, Seattle, pp. 70-81.

[10] Delosme, J.-M., and Ipsen, I., "Efficient Systolic Arrays for the Solution of Toeplitz Systems: An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI", Adam Hilger, 1986.

[11] Henry, D. S., and Joerg, C. F, "A Tightly-Coupled Processor-Network Interface", Proc. 5th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, October, 1992, Boston, pp. 111–122.

[12] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[13] Intel Corp., "iWarp Microprocessor (Part Number 318153)", Technical Information, Hillsboro, OR., Order Number 281006.

[14] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic domain decomposition in BLAZE. In S. K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524, August 1987.

[15] Kung, H.T., "Why Systolic Architectures?", Computer Magazine, Vol. 15, No.1, pp. 37–46, Jan. 1982.

[16] Kung, H. T. and Subhlok, J., A New Approach for Automatic Parallelization of Blocked Linear Algebra Computations, *Proceedings of Supercomputing '91*, pages 122–129, 1991.

[17] Leiserson, C. E., Abuhamdeh, A. Z., Douglas, D. C., et. al., "The Network Architecture of the Connection Machine CM-5", ACM Symposium on Parallel Algorithms and Architectures, June 1992, San Diego, CA, pp. 272–285.

[18] Moldovan, D. I., "ADVIS: A Software Package for the Design of Systolic Arrays", CAD, Vol. 6, No. 1, pp. 33–40, Jan. 1987.

[19] VonEicken, T., Culler, D., Goldstein, S., and Schauser, K., "Active Messages: a Mechanism for Integrated Communication and Computation", Proc. 19th Intl. Symp. on Computer Architecture, pp. 256–266, 1992.

[20] Wolf, M. E., and Lam, M. S., "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 4, pp. 452–471.

[21] Wolfe, M., "More Iteration Space Tiling", Proc. Supercomputing '89, November, 1989, Reno, pp. 655–663.

[22] Zima, H., Bast, H.-J., and, Gerndt, M., "SUPERB: A Tool For Semi-Automatic MIMD/SIMD Parallelization", Parallel Computing, 6:1–18, 1988.

A Example

Here we describe two versions of a matrix multiplication program to illustrate the fundamental differences in data movement and mapping between systolic programs and memory communication programs. The conventional sequential algorithm to compute $C = C + AB$ is shown below in Figure 6. Both the systolic and memory communication programs compute the same result given A and B, but the programs go about getting this result in a significantly different order.

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

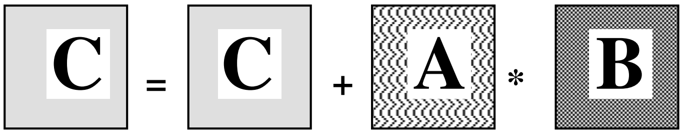


Figure 6: Matrix multiplication.

For ease of explanation, both programs multiply two $N \times N$ matrices on an N processor linear array. Both multiplication schemes can easily be extended to handle two-dimensional processor arrays as well as matrices that do not exactly fit on the processor array.

A.1 Matrix multiplication using memory communication

The memory communication program follows the standard data parallel paradigm and uses memory communication to move data to the processor that is to perform the computation. The matrices are divided and every element is assigned to a processor. The processors that "own" elements of C are responsible for computing the values of C, so the processors must fetch non-local data needed to compute their elements of C. The program proceeds in alternating phases of communication and computation. A processor first sends data needed by other processors or receives the data it needs. Then it updates all elements of C that are stored on this processor.

In this case, the matrices are divided by rows. Rows p from A, B, and C are assigned to processor p. Therefore, processor p is responsible for computing the p^{th} row of C. Here is pseudo code for the computation executed by processor 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 7 illustrates the first two steps, for $k = 0$ and $k = 1$, of the program shown above for the multiplication of two 3×3 matrices. In the first step (Figure 7.a), processor 0 broadcasts the row of B stored on processor 0 to all other processors. Then all processors add $A[i][0] * B[0][j]$ into the elements $C[i][j]$ of C they own (Figure 7.b). In the next step (Figure 7.c), processor 1 broadcasts row 1 of B, so that all processors can compute with row 1 of B. After all rows of B have been broadcasted, C (distributed over all processors) contains the results of $C + AB$.

This program uses the memory communication model for inter-processor communication. All communication is performed from memory to memory. The "owner" of row k sends the data from memory to all other processors, which store the local copy of this row in memory.

A.2 Matrix multiplication based on systolic communication

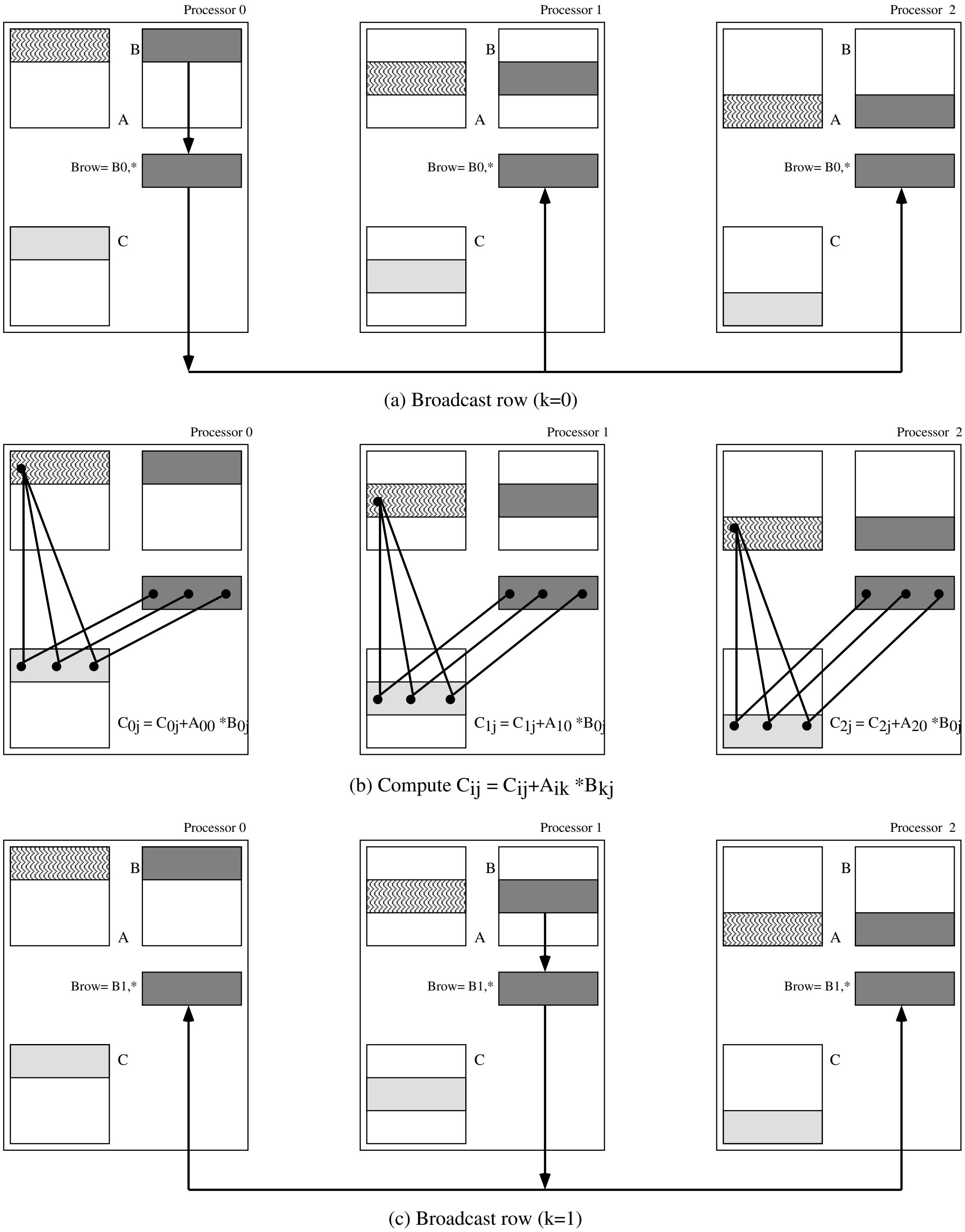
The basic idea of the program based on the systolic paradigm is to pump data through the processor array during computation instead of reading all operands from memory[15]. A and B are still assigned to processor memory, but the elements of the matrix C flow through the processor array. (In the general case of multiplying an $L \times M$ matrix with an $M \times N$ matrix, it is also possible that either A or B are pumped through the processor array.) Unlike in the memory communication style, no single processor is responsible for completely computing the value of a particular element of C. Instead each processor partially computes every element of C that passes through it.

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

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

Note that with an optimizing compiler, the temporary `c` should be kept in a register and not be moved into memory during execution of the loop body.

Figure 8 shows four snap shots of this program for the multiplication of two 3×3 matrices. In Figure 8.a, processor 0 receives C[0][0] (initialized to 0 if C=A*B is to be computed) and adds A[0][0]*B[0][0] to C[0][0]. In the next step, shown in Figure 8.b, the result (called C[0][0]') of processor 0 is passed on to processor 1. On processor 1, A[0][1]*B[1][0] is added to C[0][0]'. At the same time, processor 0 receives C[0][1] and adds A[0][0]*B[0][1] to C[0][1]. Figure 8.c depicts how the product A[0][2]*B[2][0] is added to the result computed by processor 1, C[0][0]''. The last picture (Figure 8.d) shows how the final value of C[0][0] leaves the processor array.



(a) Broadcast row (k=0)

(b) Compute $C_{ij} = C_{ij}+A_{ik} *B_{kj}$

(c) Broadcast row (k=1)

Figure 7: Parallel matrix multiplication based on memory communication

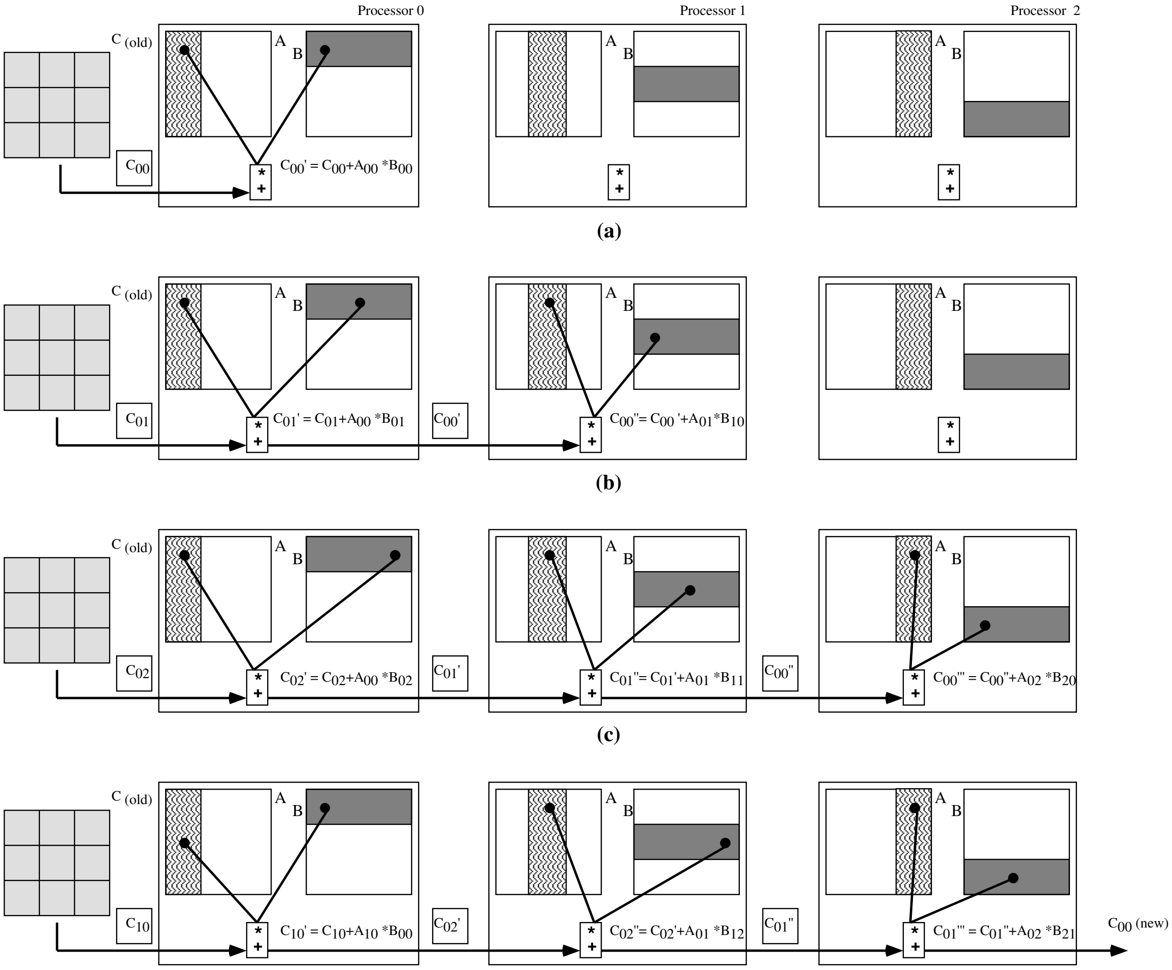


Figure 8: Parallel matrix multiplication based on systolic communication

A.3 Discussion

The systolic and memory communication styles for matrix multiplication use memory and the communication system differently. The inner computation loop of the memory communication program performs 3 memory accesses but no communication for each iteration. (We assume that the loop invariant operands are stored in a register for the inner multiply-accumulate loop.) The systolic program performs 1 memory access and 2 communication steps per iteration of its inner loop, so the systolic program takes advantage of communication system bandwidth in addition to memory bandwidth in tight computation loops.

Since the systolic program uses the communication system in its inner loop, it requires much finer grained communication than the memory communication program does. The memory communication program blocks its communication into more infrequent, larger messages. A high throughput communication system (in balance with the computation capabilities of the processor) and low overhead access to the communication system are needed to make this fine grained communication feasible.

Note that the above presentation omitted a number of details. First, since only parts of the arrays is stored locally, an additional level of mapping takes place so that memory must only be allocated for the locally resident data. Second, multiple rows and/or columns are usually stored on a single node, complicating the loop structure and address arithmetic further. Third, unrolling the bodies is usually difficult unless the bounds of the various matrices are known at compile time. As everyone who has ever looked at code that was produced by a parallel program generator or compiler knows, this code is hard to read and filled with conditional tests.