The Impact of Communication Style for the iWarp Parallel System

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

March, 1994

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. Supported also in part by the Advanced Research Projects Agency/CSTO, monitored by SPAWAR 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 document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

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

Abstract

Programs executing on private-memory parallel system exchange data by explicitly sending and receiving messages. Two communication styles have been identified for such systems: memory communication and systolic communication. In memory communication each message exchanged between two processors is buffered in memory. One example of this communication style is message passing as implemented on many parallel and distributed systems: the sender forms a message in a memory buffer, and when the message is complete, it is handed to the communication subsystem for transmission. In systolic communication, each word of a message is transmitted as it is generated. A result computed on one node passes immediately to the destination node, without any buffering in memory. Each communication style has its advantages and disadvantages. Memory communication is simpler to program and more flexible. Systolic communication offers the promise of increased instruction level parallelism and reduced access to memory.

This paper reports on measurements of the machine resources usage (use of instruction formats, sources of operands, instruction level parallelism etc.) for a set of application kernels, which have been mapped to the iWarp parallel system. The iWarp system supports both communication styles well and provides an excellent platform for parallel program generators regardless of their communication style. The results indicate that (1) programs based on systolic communication execute fewer memory access and fewer control flow operations, (2) programs based on systolic communication exhibit a higher degree of instruction level parallelism, and (3) the communication subsystem can be a useful source of additional operands for the processor.

Summary

Two communication styles have been identified for private-memory parallel systems: *memory communication* and *systolic communication*. With memory communication, messages are transferred between the communication subsystem and memory. With systolic communication, messages are transferred between the communication subsystem and the processor, without any buffering in memory. Each communication style has its advantages and disadvantages. Memory communication is simpler to program and more flexible. Systolic communication offers the promise of increased instruction level parallelism and reduced access to memory.

This paper empirically evaluates, for the first time, how the choice of communication style affects the usage of resources in a parallel system. The experiments were performed on the iWarp system, which supports both communication styles efficiently. The results indicate that (1) programs based on systolic communication execute fewer memory access and fewer control flow operations, (2) programs based on systolic communication exhibit a higher degree of instruction level parallelism, and (3) the communication subsystem can be a useful source of additional operands for the processor.

Author biographies

Thomas Gross is a faculty member in the School of Computer Science at Carnegie Mellon University. In 1983, he received his Ph. D. 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 still interested in the practical aspects compilers, esp. code generation, scheduling, and debugging.

Atsushi Hasegawa is an engineer with Hitachi Microcomputer Systems Ltd.'s Microcomputer System Engineering Department, where he designs microprocessors. He was at Carnegie Mellon University during 1991 - 1992 as a visiting scientist. His research interests include microprocessor architecture, massively parallel processing, and performance measurement. He received a BS degree in computer science from the University of Electro-Communications in Japan in 1979.

Susan Hinrichs is a doctoral candidate in the School of Computer Science at Carnegie Mellon University. Her primary research interests are in the area of parallel architecture, particularly compiling and optimizing communication for distributed memory machines. She received a BS degree in computer science from the University of Illinois in 1988.

David R. O'Hallaron is a faculty member in the School of Computer Science at Carnegie Mellon University. He received his Ph.D. in Computer Science from University of Virginia in 1986. From 1986 to 1989, we was 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 of Carnegie Mellon University. His research interests include communication architectures for parallel computers and high speed networking. Before entering the doctoral program at CMU he worked on staff at IBM Research in Yorktown and received his Dipl. Ing. ETH degree from the Swiss Federal Institute of Technology (ETH), Zuerich, Switzerland in 1988.

1 Introduction

Distributed memory parallel systems rely on the explicit exchange of messages for communication. The details of the communication operations supported by parallel systems differ in many aspects, but one division can be made based on how 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 directly passed on to the communication subsystem for transmission to another node. In *memory communication*, the complete message is generated, stored in memory and then transmitted to the destination[4]. Since the sender and receiver nodes are controlled individually, 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).

These two communication styles place different demands on the design of a processor. This paper illustrates the impact of the choice of communication style on the usage of processor resources for some key application kernels. We use the iWarp system as target for our evaluation, since this system supports both communication styles. The sample programs are automatically mapped using two parallel program generators, one for each communication style [SEE SIDEBAR]. The measurements of the execution of these programs allows us to determine if there is a correlation between communication style and instruction type 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, or if there is an impact on instruction level parallelism.

2 Experimental setup

2.1 iWarp system

iWarp is a single–chip VLSI processor developed by Intel Corp[3, 4]. It contains separate floating point units for addition and multiplication (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. This communication agent provides long-distance connections between non-adjacent nodes, such connections reserve bandwidth and reduce the startup overhead for communication [4]. An iWarp node contains an iWarp chip and some local memory (on our systems, ranging from 0.5M Bytes to 16.5 MBytes). The peak local memory bandwidth is 160 MBytes/s. An iWarp system is organized as a 2D torus of iWarp nodes, ranging in size from 4 nodes to 1024 nodes; iWarp became operational in 1990.

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. These queues can be mapped to special registers in the register file called *systolic gates*, or simply *gates*, which are shown shaded in Figure 1. Gates can be used as instruction operands like any other register in the register file, and they have the same access time. If a gate supplies 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. Writing to a gate has the effect of inserting the data word at the tail of the associated queue. Should the queue be full, the instruction spins until there there is space in the queue.

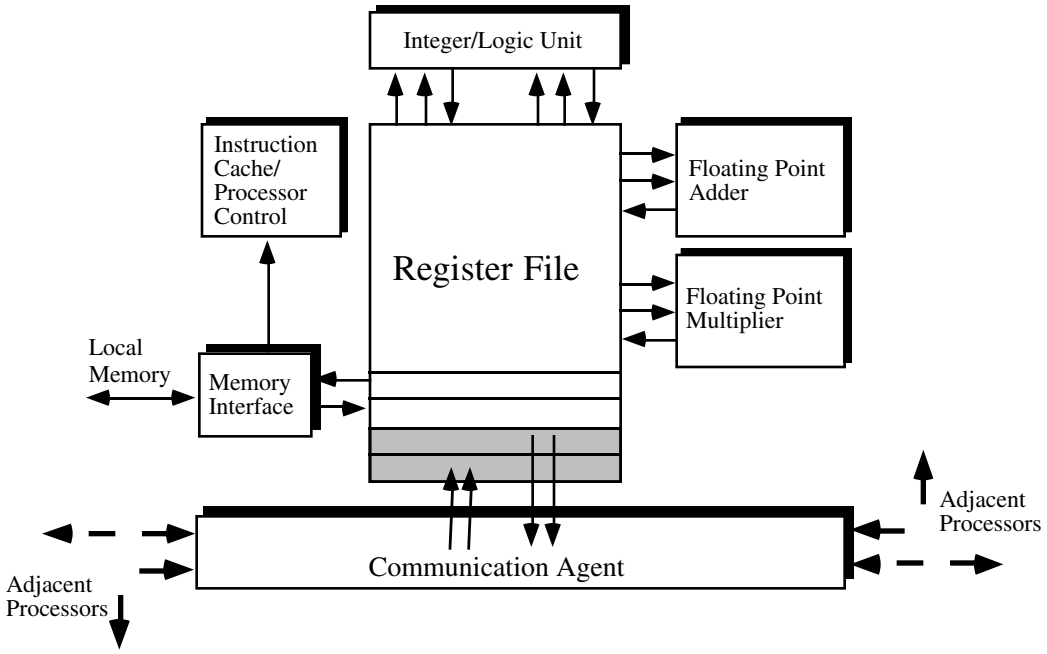

Figure 1: iWarp processor

The iWarp instruction set contains two types of instructions:

1. *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 the floating point operations execute in 2 cycles (i.e., 100ns). Most other short instructions take 1 cycle. Short instructions are encoded in 1 word (32 bits).

2. *Long instructions* control multiple functional units and therefore contain multiple operations. A multi-operation 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 80 MBytes/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, at a cost of stealing one 50ns cycle to transfer two 32-bit words. 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 multiplication (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, and one based on systolic communication. 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 [8, 11]. (Due to the structure of SOR, the parallel program generator can not automatically create a systolic program.)

There are many models of memory communication, including many forms of message passing 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 be mapped automaticly is regular enough so that this choice of communication primitives results in 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, solely determined by the communication bandwidth. Since all the connections used by these parallel programs 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. Other, more sophisticated memory communication schemes, may produce programs with different characteristics. However, for the regular programs used in this study, the zero-overhead "protocol" adopted by our program generator is adequate and produces the best overall results on the iWarp parallel system.

2.3 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 information to set 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`

	Profiled Time (ms)	Actual Time (ms)	Difference
Systolic communication	274	270	1.45 %
Memory communication	760	710	6.57 %

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

values; this format is sufficient for our application domain of image processing, computer vision, and signal processing.

The code generated 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.

All programs are given input matrices of size 160×160 and are run on 16 iWarp nodes. 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 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), programs based on memory communication 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 matrix 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 bypasses between the functional units. Table 2 shows the measured MFLOPS for all programs.

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

3 Results and evaluation

Systolic communication promises two measurable benefits[4]:

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

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 operations that were performed by LIW instructions during the execution of these programs on the iWarp system.

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

Figure 3 shows the average number of operations per instruction for each program. The programs that execute a higher percentage of operations in LIW instructions also execute a larger

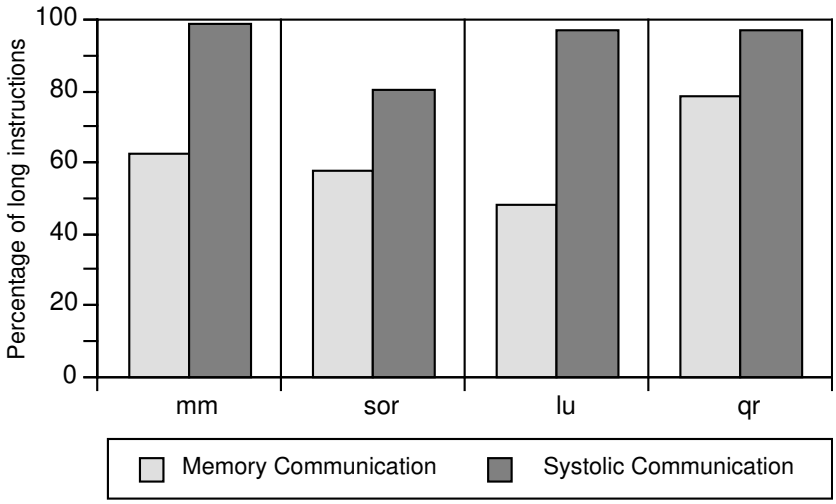

Figure 2: Percentage of operations executed in LIW instructions

number of operations per instruction. This proves that a reasonable number of operations are packed in each LIW instruction.

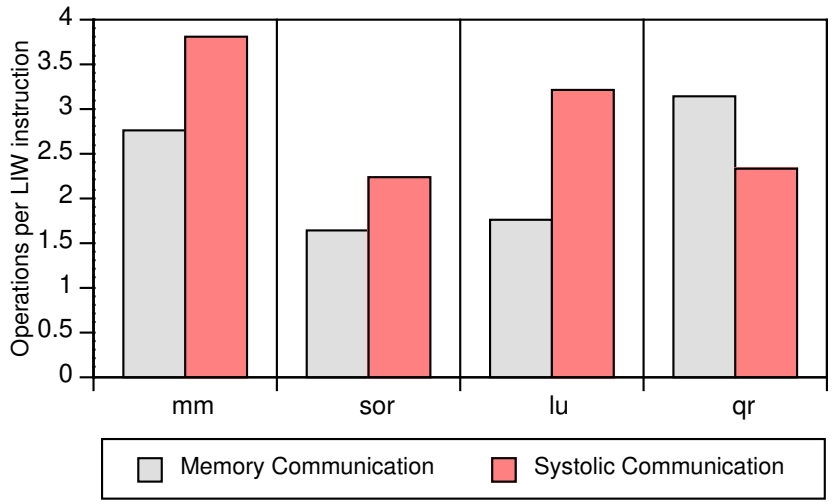

Figure 3: Operations per instruction.

Figure 2 shows the relative contribution of LIW instructions as percentage of all instructions, and Figure 3 shows the contribution as a percentage of all operations. Both metrics (derived from the execution of the programs) capture the relative frequency of usage of the LIW format. However, 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 the short instruction typically ranges from 50 ns (integer operations) to 200 ns (double precision

floating point operations). A compiler will therefore try to put as many "expensive" operations together in a LIW instruction; if any operation takes 100 ns, then any other 100ns-operation is done for free, and it is more advantageous to put such an operation into this LIW instruction than to add e.g. a 50ns-operation. Figure 4 presents the breakdown of execution cycles based on the instruction format. By this metric, the programs with systolic communication exhibit higher instruction level parallelism.

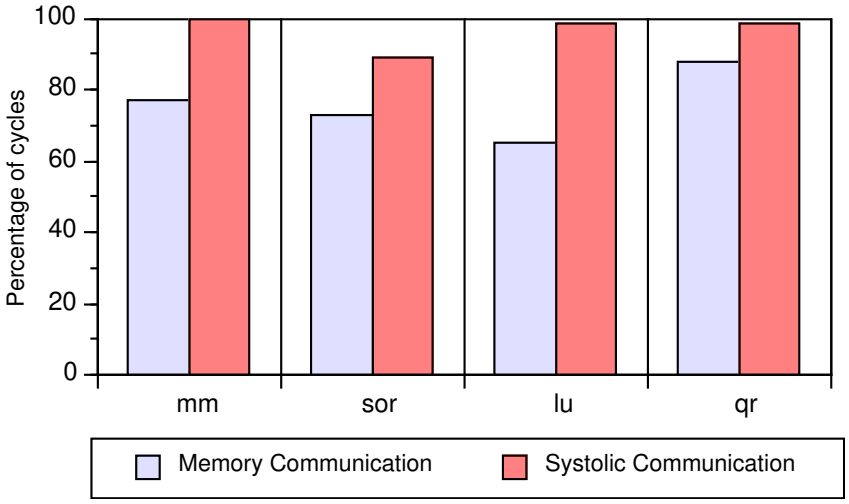

Figure 4: Classification of execution cycles: percentage of cycles executed by LIW instructions.

3.2 Operation distribution

Figure 4 depicts the instruction usage for the programs, broken into two broad categories (LIW instructions and short instructions). Although there is only a single LIW format, LIW instructions can differ dramatically on which operations are executed as part an instruction[5]. A more meaningful metric is what operations are executed for each program, and this information is presented in Figure 5. This figure shows the operation frequencies for the different programs and provides another picture of how the processor resources are used by the 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 Ops: 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. (The 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.)

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

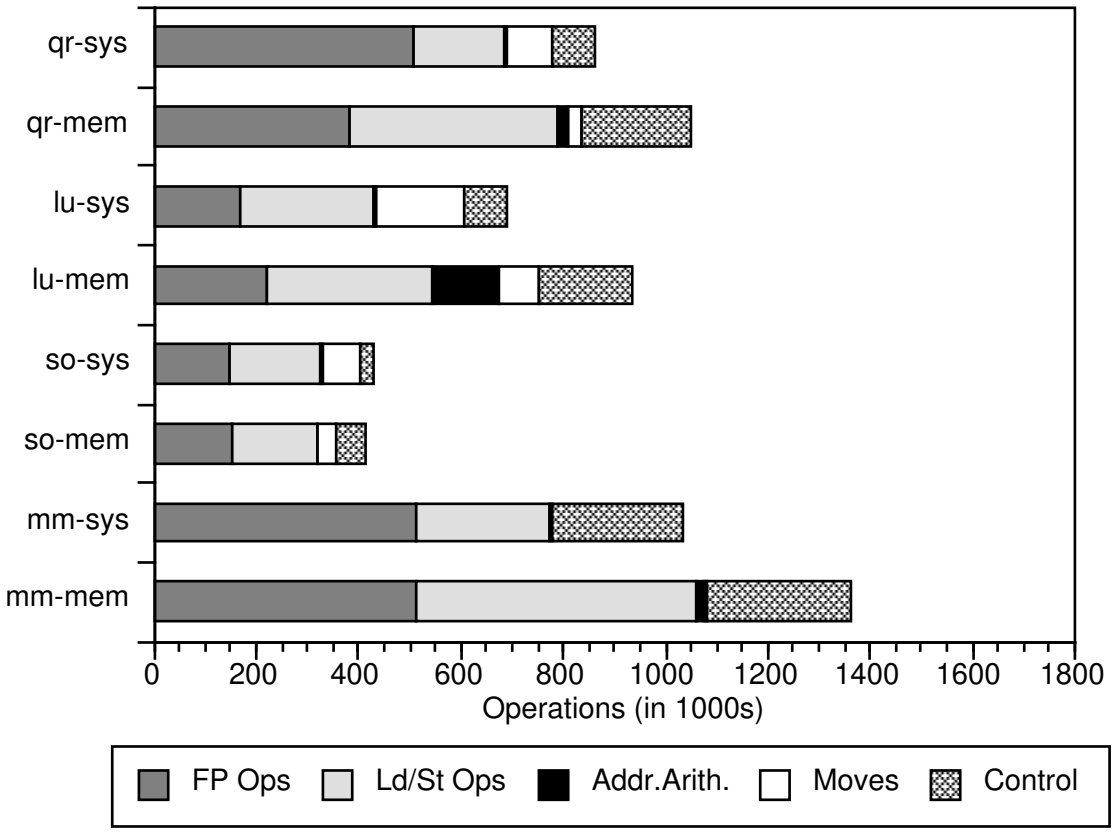

Figure 5: Classification of operations executed.

In Figure 5, 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 5 shows that memory-based programs perform more memory accesses, because memory-based programs must retrieve all operands from memory and store intermediate results to memory. Related to memory operations, Figure 5 shows a larger number of address computations for memory-based programs. Since memory-based programs access memory more frequently, they must calculate more memory addresses.

The programs with systolic communication use far fewer control flow operations because communication and computation loops are combined. The programs based on memory communication also contain more tests to determine the ownership of data. Since data flows through the program with systolic communication, 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 6 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.

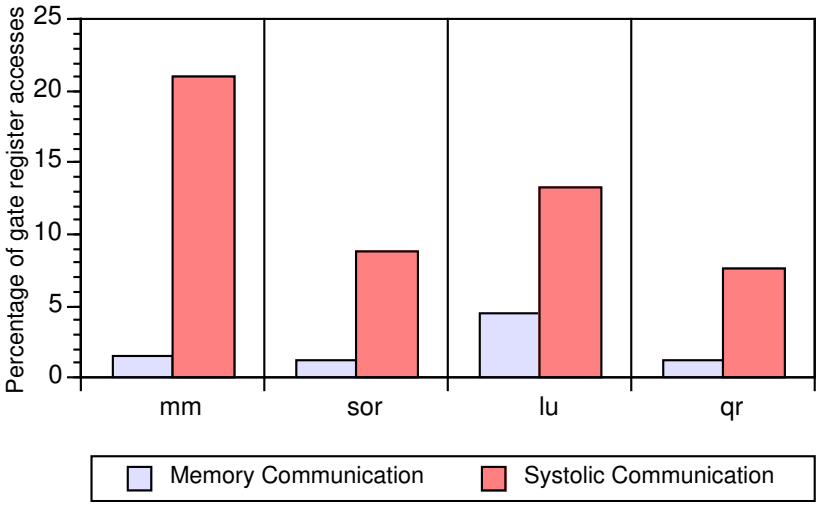

Figure 6: Percentage of variable accesses through the gates

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 shown in Figure 6, only programs with systolic communication use the 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 the programs based on memory communication solely in the data distribution loops. Since the program always stores a complete block in memory, the operands must still be fetched from memory for the computation. Therefore, the percentage of gate operands is much lower than for systolic communication.

4 Discussion

This paper analyzed the execution of parallel programs generated by two parallel program generators for the iWarp system. The two parallel program generators employ two different communication styles, systolic and memory communication, and differ in their code generation strategies.

The data presented in this paper for the iWarp system thus provide a view of the communication styles that is shaped by these parallel program generators. Since the systolic programs are close to optimal for iWarp, different systolic program generators are unlikely to produce significantly different results.

A comparison of the relative merits of the communication styles cannot be done in isolation of other key features of the measured programs. 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 system. For arbitrarily large data sets, the performance difference caused by the communication style diminishes, if the processor supports sufficient memory bandwidth. However for common input sizes, a systolic program enjoys a noticeable performance advantage. On the other hand, there are computations for which systolic programs currently cannot be automatically generated, even though programs with memory communication can be generated.

The features of the processor 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 can therefore often determine the memory location where incoming data should be stored. That means that either the sender or receiver node must compute these addresses. For programs with cyclic or block-cyclic distributions (as e.g. included in High Performance Fortran), these address computations can be quite elaborate. If there multiple messages that arrive, it is sometimes advantageous to compute these addresses on the sender node [9]. So instead of first receiving a message and then copying the data, the data are directly deposited into their 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 are stored locally. This step is easily implemented using the gates: the address is moved from a gate into a register, then the data value is moved from the gate to the memory location determined by this register.

5 Concluding remarks

There are three main conclusions that we can draw from these measurements of iWarp programs: First, programs generated to implement systolic algorithms contain a smaller number of memory access operations than programs based on memory communication. This fact has been claimed by a number of earlier papers, but this paper is the first to demonstrate this characteristic by measuring and monitoring the execution of real programs on a real system. Surprisingly, however, not only is the number of load or store operations reduced when using systolic communication, but the number of control flow operations is effected as well. This is due to the more regular structure of the systolic programs. The programs produced by our parallel program generator using memory communication contain more conditional tests: mainly to determine when and how data are to be transmitted or received.

Second, systolic programs exhibit a higher degree of instruction level parallelism. This is exhibited in the larger contribution of multi-operation instructions; this instruction format is employed more frequently for systolic programs than for the programs based on memory communication. Furthermore, the multi-operation instructions in systolic programs contain on

average a larger number of operations. Both observations explain why the systolic versions of the programs take fewer cycles to execute; this performance advantage is reflected in the better use of the processor resources.

Third, to capitalize on the reduced number of memory accesses, there must be a way to provide operands independently of the memory system. In the case of iWarp, the gates, which are a register mapped window into the communication system, provide this additional source of operands. We see that systolic programs 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.

In summary, a parallel system that aims to support fine-grain programs must pay attention to how operands can be supplied to the 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 source of operands enables a higher degree of instruction level parallelism.

Acknowledgements

We appreciate the efforts by other member of the iWarp team at Carnegie Mellon and Intel. Sonia Singh assisted in the retarget of the systolic program generator.

An expanded version of this paper with further references is available as ftp://warp.cs.cmu.edu-/iwarp/archive/iWarp-papers/cmu-cs-92-215-impact-of-comm-styles.ps

References

[1] E. Anderson and J. Dongarra. Implementation Guide for LAPACK. LAPACK Working Note 18, Univ. of Tennessee, Apr 1990. CS-90-101.

[2] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, and J. A. Webb. The Warp Machine: Architecture, Implementation and Performance. *IEEE Trans. on Computers*, C-36(12):1523–1538, Dec. 1987.

[3] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[4] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proc. 17th Intl. Symposium on Computer Architecture*, pages 70–81. ACM, May 1990. A revised version has appeared as technical report CMU-CS-90-197.

[5] R. Cohn, T. Gross, M. Lam, and P. S. Tseng. Architecture and Compiler Tradeoffs for a Wide Instruction Word Microprocessor. In *Proc. Third. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 2–14, Boston, Apr 1989. ACM/IEEE.

[6] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, 1991.

[7] H.T. Kung. Why Systolic Architectures? *Computer Magazine*, 15(1):37–46, Jan 1982.

[8] H. B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops*. PhD thesis, Carnegie Mellon University, June 1990.

[9] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation *Journal of Parallel and Distributed Computing*,?,?, (accepted) 1994.

[10] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Programming Task and Data Parallelism on a Multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–22, May 1993.

[11] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989.

[12] H. Zima and B. Chapman. Compiling for Distributed-Memory Systems. *Proc. IEEE*, 81(2):264–287, 1993.

A Parallel Program Generators

A parallel program takes a high-level description of the program and maps computation and data onto the nodes of a parallel system. A program *without* explicit communication is turned into programs *with* explicit communication for each node of the parallel system. The parallel program generator is responsible for all the details of management of the parallelism, although a human programmer may assist the tool with directives or hints.

Here we illustrate the fundamental differences in data movement and mapping between one program with systolic communication and another with memory communication. We use matrix multiplication as an example. The conventional sequential algorithm to compute $C = C + AB$ is shown below in Figure 7. 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]
```

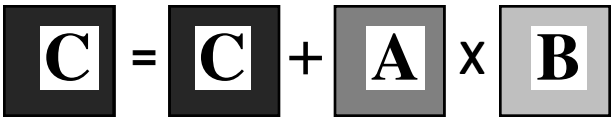

Figure 7: Matrix multiplication.

For ease of explanation, 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.

A.1 Memory communication

The memory communication program follows the standard data parallel paradigm and uses memory communication to move data to the node that is to perform the computation. The matrices are divided, and every element is assigned to a node. The nodes that "own" elements of C are responsible for computing the values of C, so the nodes must fetch non-local data needed to compute their elements of C. 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 elements of C that are stored on this node.

In this case, the matrices are divided by rows. Rows p 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 pseudo code 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 8 illustrates the first two steps for the multiplication of two 3×3 matrices. First, Node 0 broadcasts the row of B stored on Node 0 to all other processors. Then all nodes add $A[i][0] * B[0][j]$ into the elements $C[i][j]$ of C 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 send to all other nodes, where it is stored in memory as well.

A.2 Systolic communication

The basic idea of the program based on the systolic paradigm is to pump data through the parallel system during computation instead of reading all operands from memory[7]. A and B are still assigned to local memory, but the elements of the matrix C flow through the system. Unlike in the memory communication style, no single node is responsible for completely computing the value of a particular element of C. Instead each node 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 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 9 shows the computation of $C[0][0]$ by the nodes of the array. The partial result $c + A[i][p] * B[p][j]$ is passed immediately 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 3 memory accesses but no communication for each iteration. 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.

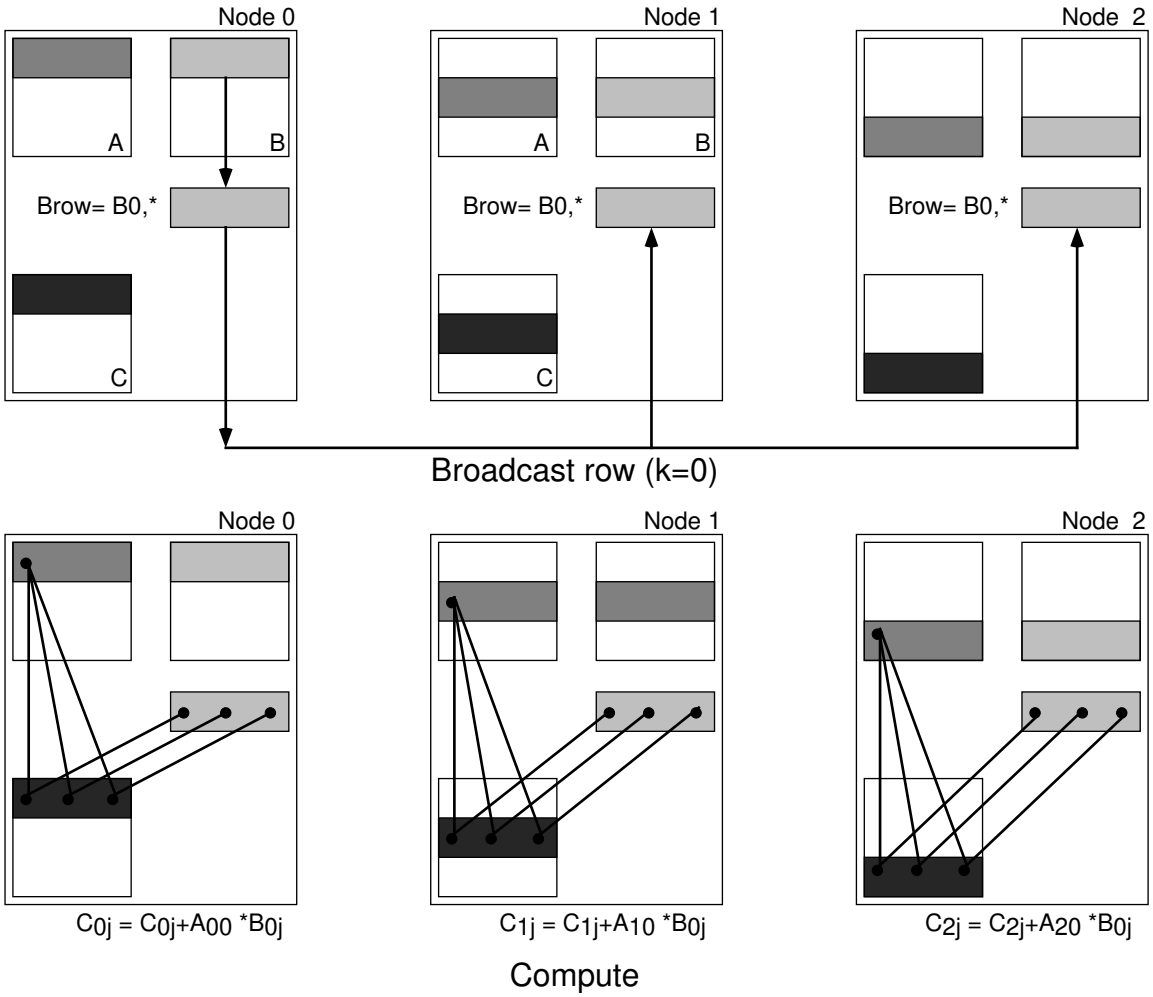

Figure 8: Parallel matrix multiplication based on memory communication

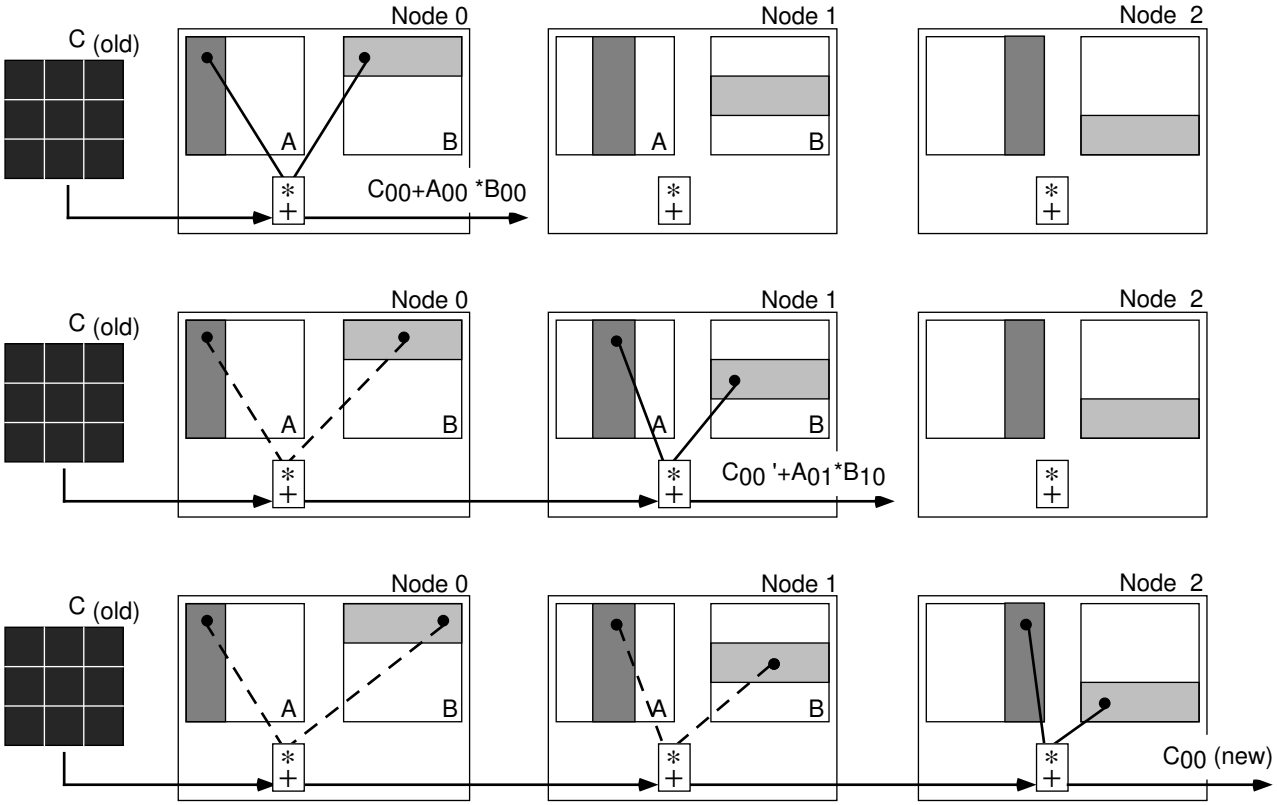

Figure 9: Parallel matrix multiplication based on systolic communication

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 [12]. 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.

B Mapping strategies and system parameters

The program generators used in this paper provide us with an opportunity to evaluate the impact of communication style for the iWarp system. But there are other mapping strategies a compiler can use, and there exist a wide range of other parallel systems. We discuss here how code generation strategies affect our results, and how these results depend on key parameters of the communication system architectures.

The program generators described in Section 2.2 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 for some computations, blocking or tiling may improve the execution of programs based on memory communication, since this strategy reduces the number of communication steps. So to understand the impact of this strategy, we use some programs from a library that is based on blocking[1].

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 our program generator based on memory communication but may have better communication characteristics due to the increased locality.

B.1 Impact of mapping strategy

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 program with memory communication must retrieve all of the computation arguments from memory, but the program based on systolic communication can also retrieve arguments from the network. Second, the program based on memory communication 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 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 only the `b` vector must be fetched from memory.

```
tmp = receive()
tmp = tmp + a*b[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 for the message passing version, but the communication requirements are greater.

Whether the program with systolic communication can surpass the performance of the inner loop based on memory communication 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[6]. There, network access requires memory bandwidth, so trading off memory access for network access only makes sense if accessing the network interface takes less time than accessing the local memory directly, or the access to the memory–mapped network interface uses processor resources that are separate from the resources used by ordinary memory operations.

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 processor 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 inner loop for a program based on memory communication could be performed in one instruction. On iWarp at most two memory accesses can be performed in parallel, so the inner loop based on memory communication must use two instructions. Of course, a processor with a single path to memory (like many contemporary microprocessors) requires at least three instructions. The systolic inner loop can be performed in one instruction.

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

```
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, 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, 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 common array sizes (see [10] for a discussion of data set sizes encountered in sensor-based computing). Table 3 shows the MFLOP measurements for the three matrix multiplication programs (mapped onto 16 nodes to allow comparison with the results of Section 3). There is a noticeable difference between the programs based on systolic communication and the programs based on blocking 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.

In summary, Table 3 underlines three points about parallel computation: With blocking, matrices must be large before the performance for systolic and memory communication comes close. If we map a computation onto a larger system, e.g. with 64 or 256 nodes, even larger inputs are required to obtain the same performance on each node. Many applications cannot scale the size of the input set to match the size of the parallel system, since the size of the input is limited by external constraints[10]. For these applications, systolic communication provides

n	Simple distribution memory communication (MFLOPS)	Blocked (tiled) memory communication (MFLOPS)	Systolic (MFLOPS)
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.

a way to realize performance on a parallel system. 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. Finally, iWarp supports static connection setup, so there is *no* protocol overhead for memory communication. Therefore the data presented for memory communication here show an optimistic scenario for this communication style since any protocol overhead will reduce the effective performance.

B.2 Impact of communication system parameters

Another parameter that influences the tradeoffs between memory communication and systolic communication is how the processor and the program handle message arrival. The message passing program generator used in this study and the block matrix multiplication explicitly store the data from the network into memory as described in Section 2.2. 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.

The performance of the three matrix multiplications can be compared by defining performance equations in terms of S, the message start up overhead; T, the transfer time to send each word; L, the time to execute the body of the inner loop; p, the number of processors; and n the size of the matrix. The execution time E_{MP} of matrix multiplication with simple memory communication is given by:

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

(There are n^3 computation steps evenly divided over p processors, and $2n^2$ data elements must be moved.) The execution time E_{Sys} for matrix multiplication based on systolic communication is:

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

The execution time E_{Blk} for the block based matrix multiplication is:

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

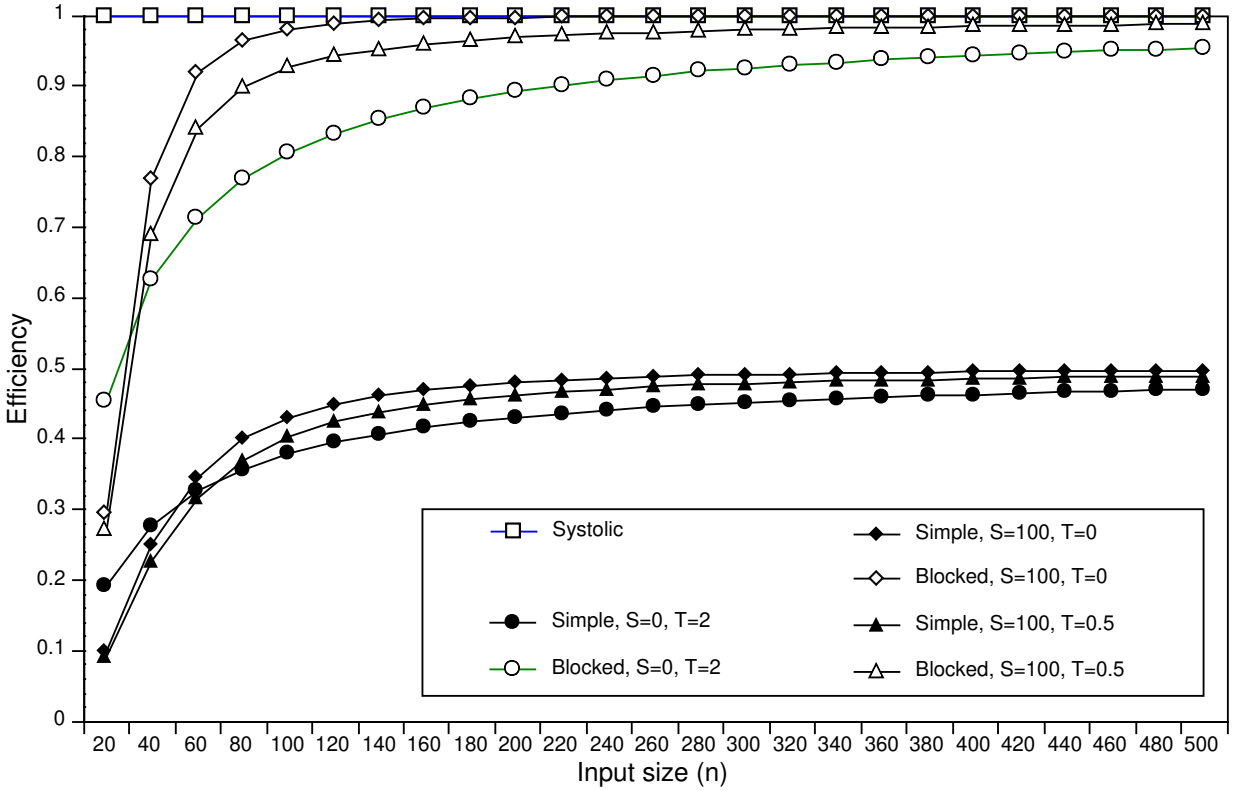

Figure 10: Matrix multiplication performance for $p = 16$

Note that the three versions differ in their communication cost; the costs to compute the innermost loop are of the same order, but the constant factors (L_{MP}, L_{Sys} and L_{Blk}) differ to account for possible differences in the local computation capabilities of the processors. 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 (i.e., $T = 0$), then simple memory communication contributes only on the order of $O(n)$ time steps asymptotically, and the blocked algorithm contributes only $O(\sqrt{p})$ steps. However, if messages 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 10 illustrates the impact of varying the parameters S and T of the communication system for the three mapping strategies and shows the efficiency (node utilization) for different input sizes. To emphasize the effect the communication system, all other parameters (number of paths to memory, number of functional units, degree of parallelism) are kept constant and are the same as for the iWarp system. $S = 0, T = 2$ models a system with no startup overhead and a transfer time of 2 cycles (i.e., iWarp). The simple message passing inner loop requires 2 instructions due to the lack of memory bandwidth; the other two strategies pack the operations into a single instruction. Therefore, the performance of the simple message passing program

can be at most half of the other cases. A machine with completely non-interfering 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 10 also includes a more realistic scenario with a transfer cost of 0.5. For all models, as the matrix size increases, the performance is dominated by the cost to perform the inner loop, but for smaller sizes, there are noticeable performance differences.