Naval Command,
Control and Ocean
Surveillance Center RDT&E Division San Diego, CA
92152–5000

(4)

Technical Document 2381
September 1992

Applications Development on the Intel iWarp System

J. Z. Lou

Technical Document 2381
September 1992

Applications Development on the Intel iWarp System

J. Z. Lou

NAVAL COMMAND, CONTROL AND OCEAN SURVEILLANCE CENTER
RDT&E DIVISION
San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commanding Officer

R. T. SHEARER
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this report was sponsored by the Office of Naval Technology. The Navy Standard Matrix Processor Program under NAVSEA PMS 412 via the Naval Air Warfare Center supported preparation of this report. The parallel programming cases examined form a basis for subsequent comparison to matrix processor programming approaches.

Released by
G. W. Byram
Technology Research and
Development Branch

Under authority of
J. R. Wangler
Space Systems and Technology
Division

ACKNOWLEDGMENTS

The author would like to thank Dr. George Byram and Dr. Keith Bromley for their suggestions and support on this work. Gratitude is also expressed to Ms. Elizabeth Wald of the Office of Naval Technology (Code 227).

LH

CONTENTS

1 INTRODUCTION 1

1.1 WHAT IS PARALLEL PROCESSING? 1

1.2 PARALLEL SIGNAL AND IMAGE PROCESSING MODELS 2

2 THE iWARP SYSTEM 4

2.1 iWARP SYSTEM CHARACTERISTICS 4

2.2 MESSAGE-PASSING ON THE iWARP SYSTEM 5

2.3 UNIX-C PROGRAMMING INTERFACE TO THE iWARP SYSTEM 5

3 iWARP PROGRAMMING TOOL DEVELOPMENT 6

4 MAPPING APPLICATIONS TO THE iWARP SYSTEM 9

4.1 TWO-DIMENSIONAL FFT 9

4.2 TWO-DIMENSIONAL CONVOLUTION 16

4.3 NUMERICAL LINEAR ALGEBRA ALGORITHMS 18

4.4 THE IMPLEMENTATION OF AN IMAGE WEIGHTED FRAME-DIFFERENCING SCHEME 27

4.5 ON PARALLELIZATION OF A TWO-DIMENSIONAL ADAPTIVE LMS ALGORITHM 28

4.6 ON A PARALLEL NEURAL NETWORK TRAINING 30

5 CONCLUSIONS 32

6 REFERENCES 33

APPENDIX A: CODES A-1

FIGURES

1. Function partition and pipelined processing 3

2. Data partition and SIMD/MIMD processing 3

3. A ring network embedded in a two-dimensional network 7

4. Step 1: Data exchange between columns of processors 12

5. Step 2: Data exchange between rows of processors 12

6. Step 1: Data exchange between columns of processors 13

7. Step 2: Data exchange between rows of processors 13

8. Performance of two-dimensional FFT (128 x 128) 15

9. Performance of two-dimensional FFT (256 x 256) 15

10. Performance of Cholesky factorization 20

11. Performance of solving a linear triangular system (256 x 256) 21

12. Performance of solving a linear triangular system (512 x 512) 21

13. Performance of QR matrix factorization (256 x 256) 24

14. Performance of QR matrix factorization (512 x 512) 24

15. Performance of a MVDR beamformer 25

16. Performance of the weighted differencing program 28

TABLES

1. Performance of matrix transpose (128 x 128) 14

2. Performance of matrix transpose (256 x 256) 14

3. Performance of matrix transpose (512 x 512) 14

4. Performance of two-dimensional FFT (512 x 512) 16

1 INTRODUCTION

1.1 WHAT IS PARALLEL PROCESSING?

The task of parallel processing is to develop a good strategy to partition a certain computational task into a set of subtasks and to assign each subtask to a processor on a parallel system. It seems to be difficult to design a single, high-level parallel processing model and a set of tools that are efficient for all applications and/or for all parallel systems. For a given (type of) application and a target parallel system, however, good parallelism can be achieved by minimizing message-passing overhead and maximizing load balance. Accurate, theoretical performance evaluation of a nontrivial parallel algorithm running on a message-passing system is generally not easy except for algorithms that involve few interprocessor communications. An upper limit of the performance of a parallel application can be found using the well-known Amdahl's law (reference 1): Suppose a computational task needs a sequential execution time T_0 and

$$T_0 = T_p + T_{np},$$

where T_p is the execution time taken by the part of the computation that is fully parallelizable, and T_{np} is the part that is basically nonparallelizable. Now, if we perform the computation on a parallel system with P processors, the execution time is (assuming a perfect load balance)

$$T_1 = \frac{T_p}{P} + T_{np}.$$

The speed-up using P processors is

$$S = \frac{T_0}{T_1} = \frac{T_0}{\frac{T_p}{P} + T_{np}}, \tag{1}$$

and the efficiency is defined as

$$E = \frac{S}{P} = \frac{T_0}{T_p + PT_{np}}.$$

Although this model is oversimplified for analyzing many practical parallel applications, it does indicate how efficiently one can use a multiprocessor system. Let us assume there is an application for which $T_0 = 1$ and $T_p = 0.95$. According to equation 1, using 10 processors would produce a speed-up of 7 and an efficiency of 70%. But using 100 processors would only produce a speed-up of 17 and an efficiency of 17%. This

example tells us that a massively parallel machine can be used efficiently only when the nonparallelizable part of an application is very small compared with the parallelizable part.

Programming a distributed memory, multiple-instruction, multiple-data (MIMD) parallel computer is indeed more difficult than programming a shared memory or single-instruction, multiple-data (SIMD) parallel computer. This difficulty is related to the parallel processing flexibility of such a system. This flexibility offers actual, efficient implementations of various efficient parallel algorithms. Though it is possible to write some basic, low-level parallel programming tools, it is generally not easy to design an efficient high-level programming tool that fits all applications. For a parallel application developer on such a system, a possible way to efficient implementation of a parallel application algorithm is to use a conventional high-level programming language like C or Fortran plus some low-level parallel system library function calls. One reason is that programming at this level gives one the flexibility of reconfiguring in software the processor network so as to make it fit the best algorithm one can come up with. Another reason is that at this level, the programmer has full control over actual message-passing operations. For fine-grain applications, one may need to be as 'greedy' as possible in terms of data movement to minimize communication overhead, since otherwise the parallel performance may be consumed by this overhead, as can be seen from Amdahl's law.

1.2 PARALLEL SIGNAL AND IMAGE PROCESSING MODELS

Many composite signal and image processing algorithms can be implemented on a two-dimensional mesh of processors with different components of the algorithm processed in a pipelined fashion. In pipelined processing, for example, we could partition the whole iWarp array into several connected subarrays, with each subarray performing a different part of the algorithm. The throughput of pipelined processing is clearly determined by the slowest part in the chain. Pipelined processing falls into the category of "parallel processing by function partitioning." Another popular model for parallel processing is "data partitioning," in which each processor deals with a subset of the data set through the entire algorithm. Using this model, it is important to find a reasonable way to partition and distribute the data so that the interprocessor communication overhead is minimized for an application algorithm. The throughput of the data partition model is limited by the processor(s) with the largest workload. Therefore, load balancing is one of the crucial factors affecting the performance using this model. In mapping some complex signal and image processing algorithms to a parallel system, it should be possible to design an efficient approach using a combination of these two models. The two computational models are illustrated in figures 1 and 2, respectively.

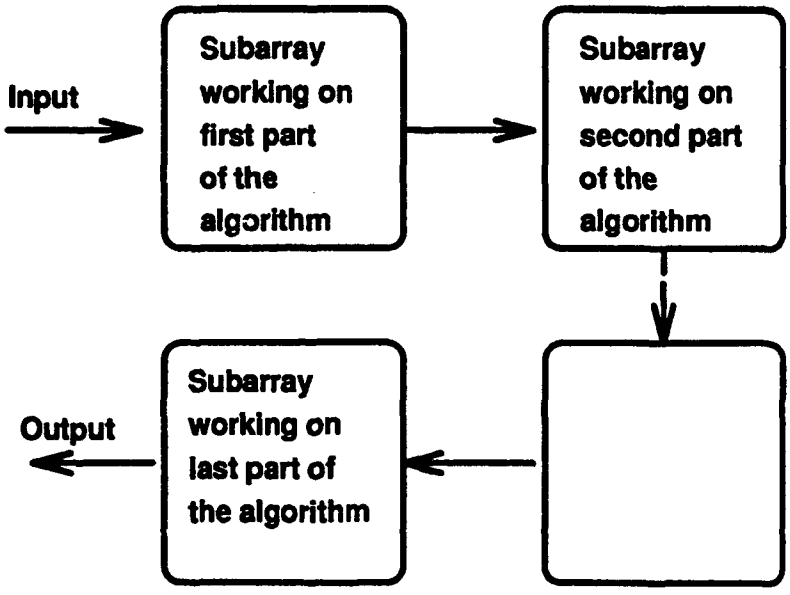

Figure 1. Function partition and pipelined processing.

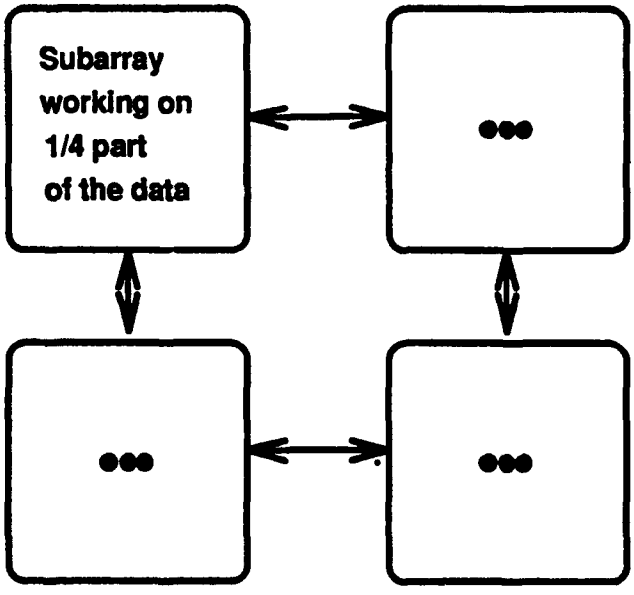

Figure 2. Data partition and SIMD/MIMD processing.

Load balancing seems to be a less challenging problem for many signal and image processing applications since, in many cases, computational tasks can be evenly partitioned (or nearly so) and assigned to processors before the computation starts and the computational loads for different processors are static through the whole computation. Minimizing communication cost is still an important issue to consider when mapping signal and image processing algorithms to a message-passing parallel system. For many pipelined types of processing, it is crucial to have a set of processors capable of performing external I/O to keep up with the processing rate within the parallel system.

Section 2 discusses iWarp architecture and programming on the iWarp system. Section 3 discusses programming tools development on the iWarp system. Section 4 presents mapping strategies for implementing signal and image processing applications onto the iWarp system and their performance results. In appendix A, we include some codes that illustrate how to write an application code on the iWarp system: a head file showing necessary include files and global declarations; a processor-network set-up function illustrating setting up a logical network and performing message-passing; a main program using our parallel programming tools to implement a weighted differencing of two images.

2 THE iWARP SYSTEM

2.1 iWARP SYSTEM CHARACTERISTICS

The iWarp system is a distributed memory, MIMD multiprocessor system. Each iWarp processor consists of three distinct components: a computation agent, a communication agent, and a local memory unit. The computation agent performs the normal programmed computation. The communication agent handles message-passing between different processors. The local memory unit provides access to local memory for both the computation agent and communication agent. Each iWarp processor has a 0.5-megabyte (Mbyte) base memory that can be upgraded in 0.5-Mbyte increments. Each iWarp processor has an upper-limit speed (or "peak performance") of 20 MFLOPS on a C-Step chip for single precision floating-point operations and 10 MFLOPS on double precision operations. The peak performance for a B-step chip is half that of a C-step chip. The bandwidth of interprocessor message-passing is 40 Mbytes/second in each of the four directions. So roughly speaking, a minimum of four single precision operations are needed on each word (4 bytes) of data being sent on to gain through parallelism. iWarp has a interprocessor communication library called PathLib. The PathLib implementation offers a low-latency, high-bandwidth interprocessor communication. The iWarp system was designed for fine-grain types of applications, for example, signal and image processing.

2.2 MESSAGE-PASSING ON THE iWARP SYSTEM

The processor network on the iWarp system is a two-dimensional toroidal mesh, which we call the iWarp array. One advantage of such a topology is that any processor on the array is no different from any other processors on the array. This processor connection structure makes it convenient to write scalarable parallel code and some important parallel program development tools. For example, it is easy to write an application code that runs on any rectangular iWarp array and performs I/O at specified processors.

Message-passing protocols are included in the PathLib system. There are three types of modes a programmer can use for performing message-passing. In iWarp terminology, these modes are called streaming, express, and spooling (references 2 and 3). Streaming provides the mechanism for fast, 'door-to-door' small message delivery, particularly useful for pipeline or systolic processing. It was implemented using fast, special registers as output and input 'gates' so that a message does not need to go through local memories of sending and receiving processors during the message-passing process. Express is a facility that allows each processor to connect a pair of message-passing I/O gates so that messages can go through the processor's communication hardware from/to its neighboring processors without affecting the activities taking place at this processor. I/O gates can be connected or disconnected at runtime, which provides a means for flexible and efficient interprocessor communication. Spooling allows transmitting a large message from a message buffer at a sending processor to the message buffer at a receiving processor. Message-passing in the spooling mode is asynchronous in the sense that the sending processor can start the next step of processing after telling the processor's communication hardware to send a certain message buffer to a destination processor; and the receiving processor's communication hardware will put the received messages in its local memory without interfering with other activities going on at the processor. Hence, the proper use of message spooling makes it possible to overlap communication and computation. On the other hand, streaming mode message-passing is synchronous in the sense that sending and receiving processors must coordinate properly during the message-passing. Inconsistency in the pair of processors' actions may cause the program execution to stop there.

2.3 UNIX-C PROGRAMMING INTERFACE TO THE iWARP SYSTEM

If we use a parallel computer to perform a computationally intensive and well-parallelized part of an application algorithm, other parts of the algorithm (e.g., visualization and some sequential part of the algorithm) might better be executed on another (may be sequential) machine. Indeed, there has been a growing interest in the supercomputing world for a heterogeneous computing environment in which parallel systems can interface with reasonable efficiency with various workstations and other systems. For example, it would be nice to be able to perform message-passing between a program running on a Sun workstation and a program running on any of the processors on a parallel system.

The Intel iWarp system provides a software mechanism called the *imsg* facility that allows, at the application programming level, the message-passing between a C program running on a host workstation and a C program running on the iWarp system. The way it works can be briefly described as follows. We call the program running on the host workstation the host program and call the program running on the iWarp array the cell program. We first need to load the cell program to the iWarp array because the iWarp load program will also start a controlling process, called the iWarp daemon process, which is needed for the host program's initialization action (e.g., the host program needs to get the memory address of the iWarp controlling process). Then we can load the host program. The communication between the host program and cell program is realized by reading/writing from/to a certain memory address of the iWarp controlling process. At the application programming level, this facility makes one think it is possible to pass data from a host machine to any iWarp cell directly. But the actual implementation of this mechanism on the iWarp is a token-ring message-passing. Therefore, even assuming the *imsg* implementation itself is efficient, the *imsg* facility cannot do parallel message-passing on the iWarp array, which would give a bad performance for some parallel applications. We think the *imsg* facility is useful when we need to have the host program control the executions of cell programs or when we need to pass some data from the host program, which may be the result from a computation performed on the host workstation, to the iWarp array.

3 iWARP PROGRAMMING TOOL DEVELOPMENT

Although using a conventional programming language with some PathLib function calls offers great flexibility and efficiency in mapping various application algorithms onto the iWarp system, we do not want to rewrite much of the high-level communication and simple global operation code in every application program development. To increase the parallel code reuse, we have been writing some efficient parallel programming tools for the iWarp system. These tools are in the form of C functions and can be called to facilitate parallel code development on the iWarp system.

The parallel programming tools we have developed include functions for setting up processor networks of rings (unidirectional and bidirectional) and two-dimensional toroidal meshes of various sizes; data movement functions for broadcasting, scattering, and gathering a data array to or from an array of iWarp processors; a function for summing up a set of vectors or matrices distributed on an iWarp array and putting the result at one of the processors in the iWarp array; and a function for transposing a two-dimensional data array distributed on an iWarp array. We point out that with the code that sets up a two-dimensional toroidal mesh, an embedded bidirectional ring can be obtained by properly defining the message-passing I/O handles (for an explanation of

these handles, see references 2 and 4). Figure 3 shows a ring network embedded in a 4 x 4 toroidal mesh.

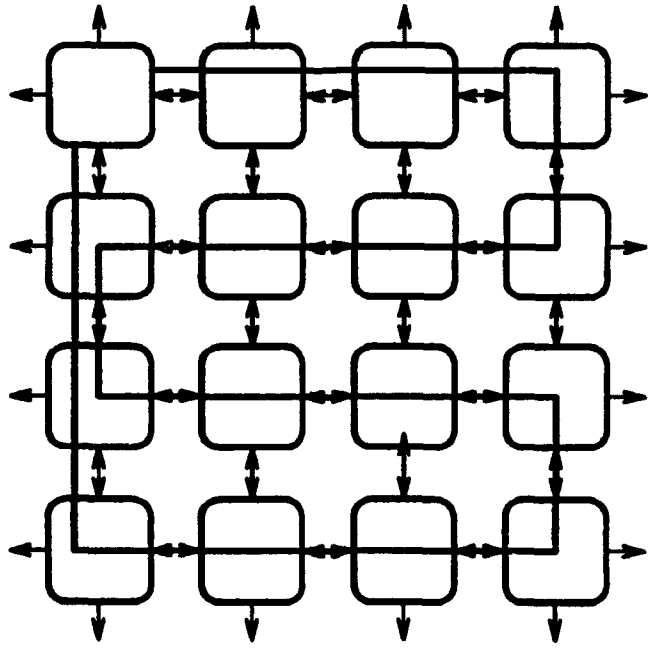

Figure 3. A ring network embedded in a two-dimensional network.

For image processing applications, the typical data structure is two-dimensional. A basic assumption we have made in terms of data partition is to have each processor hold a few rows or columns of the whole data array. The scatter function can be called to read from a disk file a two-dimensional data array and assign a submatrix (by rows or by columns) to each processor of the iWarp array. The gather function can be called to output submatrices distributed on an iWarp array to a disk file. The scatter and gather functions never hold the whole data array since we also assume no processor would have enough local memory to store the entire data array for an application. At this moment, the performance of such I/O operations is restricted by the relatively low bandwidth between the iWarp system and external I/O devices. For efficient, pipelined real-time computing, e.g., for systolic processing, we may need at least one row or column of processors connected to external I/O devices to get enough I/O bandwidth. For some applications, data overlapping at different processors is necessary (e.g., to form a window of certain size). Therefore, we have generalized the scatter function so that an argument to the function can be specified for the number of overlapping rows (columns) needed.

The global sum operation is often needed in computing on a distributed memory system. An efficient implementation of such a function is important to the performance of

an application requiring many such operations. On a toroidal mesh, we implemented a global sum function using the following algorithm, assuming the linear dimension of the iWarp subarray used is a power of 2:

A Reduction Sum Algorithm Using Express (Split-Join)

```
Redefine row and column ID for this processor relative to the output processor ID;
/* right to left sum */
Bind output gate to left and input gate to right;
if column ID = 0
    Receive and add width − 1 buffers to the local buffer;
    Send out a software-mark
    Goto label A;
flag ← 1;
while(flag ≠ 0)
     if column ID is odd
        Send out a buffer;
        Join right-left gates;
        if a software-mark comes
            Receive and send out a software-mark;
        flag ← 0;
    if column ID is even
        Receive and add a buffer to the local buffer;
        column ID ← column ID / 2;
if column > 1, exit;
label A: ;
/* down to up sum */
Bind output gate to up and input gate to down;
if column ID = 0
    Receive and add height − 1 buffers to the local buffer;
    Send out a software-mark
    Goto label B;
flag ← 1;
while(flag ≠ 0)
     if column ID is odd
        Send out a buffer;
        Join down-up gates;
        if a software-mark comes
            Receive and send out a software-mark;
        flag ← 0;
    if column ID is even
        Receive and add a buffer to the local buffer;
        column ID ← column ID / 2;
label B: exit;
```

This function takes six parallel additions to get the sum of a linear array of data distributed on an 8 x 8 iWarp array, with the result put at one of the processors specified as an argument to the function. In general, the reduction sum algorithm needs $\lg_2 N$ parallel additions to get the sum of N numbers. It is often the case that the result of a global operation is needed at all processors for subsequent computing. This can be done by a broadcast (or replicate) function, which sends a copy of a buffer of data to each processor on the iWarp array. An efficient way to implement a broadcast function on iWarp is to use the express facility.

Another global data movement operation is the transpose of a two-dimensional data array distributed on the iWarp array. The reason that this operation is useful can be explained as follows. Suppose we have distributed the data matrix by columns to the iWarp array and we would like to do a left transform and then a right transform on the data matrix. In matrix notation, we want to compute the product of three matrices:

$$\begin{bmatrix} \textit{left} \\ \textit{transform} \\ \textit{matrix} \end{bmatrix} \begin{bmatrix} \textit{data} \\ \textit{matrix} \end{bmatrix} \begin{bmatrix} \textit{right} \\ \textit{transform} \\ \textit{matrix} \end{bmatrix}$$

Here we assume the first and the third transform matrix can be formed locally at each processor. The left transformation (i.e., finding the product of first two matrices) can be readily performed in parallel. To perform the second transformation (i.e., find the product of the resulting matrix from the first transform and the third matrix) in parallel, we need to transpose the data matrix since it is not partitioned correctly for that operation. This example is applicable to the two-dimensional fast Fourier transform (FFT) algorithm (see section 4.1) and to the singular value decomposition of a matrix. We will discuss strategies for implementing a matrix transpose on the iWarp array in section 4.

4 MAPPING APPLICATIONS TO THE iWARP SYSTEM

In this section, we describe strategies for the mappings of two-dimensional FFT, two-dimensional convolution, some image processing and neural network algorithms, and some numerical linear algebra algorithms onto a message-passing parallel system. We also present performance results obtained from the implementations of some of these algorithms on the iWarp system. We basically used the data partition model in all our discussions and implementations. However, the function partition model may be used as a global strategy when we want to map a composite signal or image processing algorithm to such a parallel system using some of the applications discussed here as components.

4.1 TWO-DIMENSIONAL FFT

Two-dimensional discrete Fourier transform is a tool frequently used in signal and image processing algorithms. Its mathematical definition is

$$H(n1, n_2) = \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} t_1(n_2, k_2) t_2(n_1, k_1) h(k_1, k_2), \quad n_1 = 0 \cdots N_1 - 1, \; n_2 = 0 \cdots N_2 - 1;$$

where $t_i(n, k) = exp(2\pi ink/N_i)$. We can also write the above equations into a matrix equation that is more instructive for our purpose

$$\begin{bmatrix} H(1,1) & \cdots & H(1,N_2) \\ \cdot & \cdots & \cdot \\ H(N_1,1) & \cdots & H(N_1,N_2) \end{bmatrix} =$$

$$\begin{bmatrix} t_1(1,1) & \cdots & t_1(1,N_1) \\ \cdot & \cdots & \cdot \\ t_1(N_1,1) & \cdots & t_1(N_1,N_1) \end{bmatrix} \begin{bmatrix} h(1,1) & \cdots & h(1,N_2) \\ \cdot & \cdots & \cdot \\ h(N_1,1) & \cdots & h(N_1,N_2) \end{bmatrix} \begin{bmatrix} t_2(1,1) & \cdots & t_2(1,N_2) \\ \cdot & \cdots & \cdot \\ t_2(N_2,1) & \cdots & t_2(N_2,N_2) \end{bmatrix}$$

This matrix equation shows more clearly what needs to be done if we want to parallelize the row-column transform algorithm for the two-dimensional Fourier transform. Symbolically (see reference 5), the row-column transform algorithm, e.g., can be arranged as

$$H(n_1, n_2) = \text{FT-on-index-2}(\text{FT-on-index-1}[h(k_1, k_2)]).$$

In matrix notation, this is equivalent to computing the product of the first two matrices and then computing the product of the first product matrix with the last matrix. Suppose we let each processor hold a few columns of the data matrix. The first matrix multiplication can then be readily parallelized. To do the second parallel matrix multiplication, however, we first need to transpose the first product matrix. Thus, a simple parallel two-dimensional FFT algorithm using row-column transform is

1. scatter the two-dimensional data array by columns to an iWarp array;
2. perform a parallel one-dimensional FFT on each processor in the iWarp array;
3. transpose the data array distributed on the iWarp array;
4. perform another parallel one-dimensional FFT on each processor in the iWarp array; and,
5. collect the transformed data array for the next processing step.

Now, if we have an efficient one-dimensional FFT code running on each processor of a parallel system, the efficiency of the matrix transpose operation clearly has a great influence on the performance of the two-dimensional FFT code. In matrix transpose operation, each processor needs to send some data to each of the other processors in the

network. An efficient implementation of the matrix transpose operation should use as many available communication paths as possible. Figures 4 and 5 show one way of performing the matrix transpose on a mesh connection of processors.

This matrix transpose strategy basically consists of two steps. The first step performs data transfer between columns of processors. In the first step, for example, processor(1,1) will send to processor(1,2) all the data needed by processors in the second column. Thus processor(1,2) needs a temporary buffer to store the received data for processors(i,2) for $i = 2, \cdots$, height-of-network. In the case of each processor containing more than one row of data, a local transpose is also performed at each processor in the first step. The second step involves data transfer between rows of processors. In the second step, for example, processor(1,1) will send data stored in its temporary buffer to processors(i,1) for $i = 2, \cdots$, height-of-network. Using this matrix transpose strategy, the number of pathways used simultaneously on a square iWarp array is $\sqrt{M}$, where M is the number of processors in the iWarp array. A major drawback of this strategy is the need to store temporary data on each processor, which is unfavorable if each processor has a limited amount of local memory and the data size of the application is large.

Another way to implement a matrix transpose is illustrated in figures 6 and 7. This matrix transpose strategy also consists of two steps. In the first step, each column of processors sends data needed by processors on the same row but different columns. In the second step, all processors but one on a certain row send data to a different row of processors; the remaining processor does a local transpose; this pattern is repeated for all processors on the row. This transpose strategy uses $\sqrt{M} - 1$ pathways simultaneously. The strategy may be more efficient since all data transfer can be implemented from the source to the final destination without using extra local memory.

We also point out that operation count for the row-column two-dimensional FFT is $O(2n^2 \log_2 n)$. The row-column parallel algorithms discussed above have an operation count $O(2n^2 \log_2 n/P)$, where P is the number of processors.

We measured performances of the matrix transpose operation and the first two-dimensional FFT algorithm from our implementations. Performances on B-Step and C-Step processors are compared. Tables 1 through 3 show the execution time for a complex matrix transpose operation for three different sizes of matrices. The first performance data in table 3 are missing because the execution could not complete in that case. We guess it may be that the data size required for each processor exceeds the memory available for an application program on each processor. Figures 8 and 9 and table 4 give the performance results for two-dimensional FFT computations.

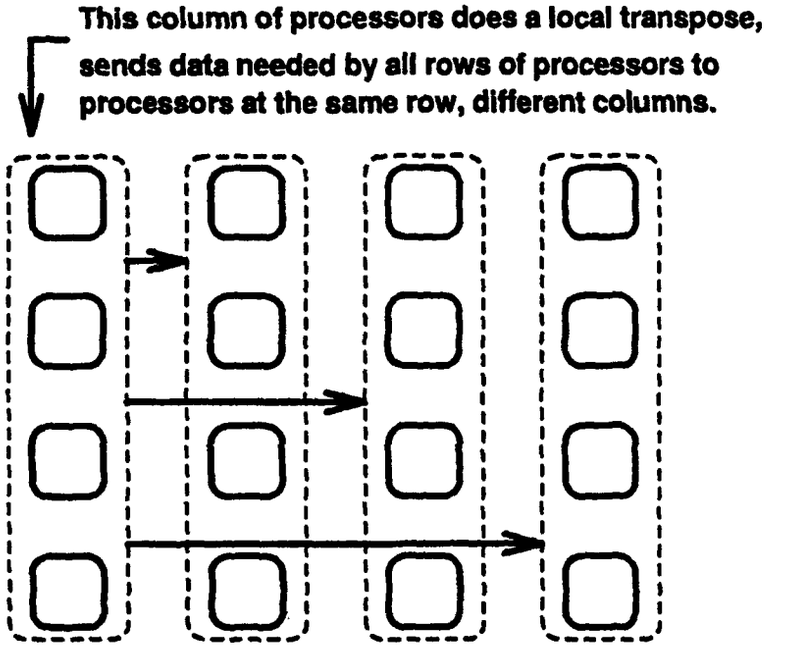

Figure 4. Step 1: Data exchange between columns of processors.

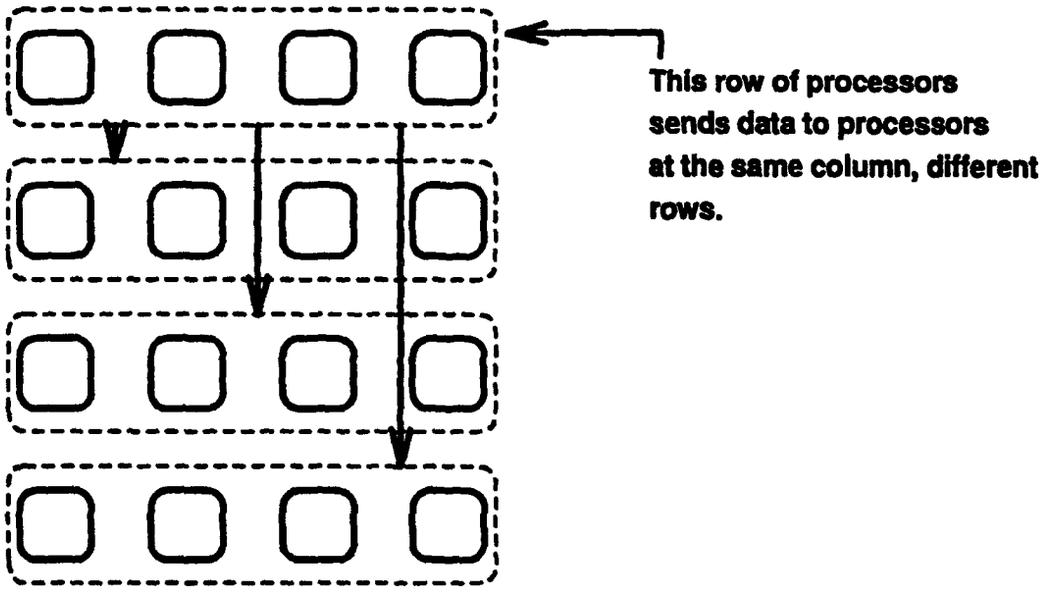

Figure 5. Step 2: Data exchange between rows of processors.

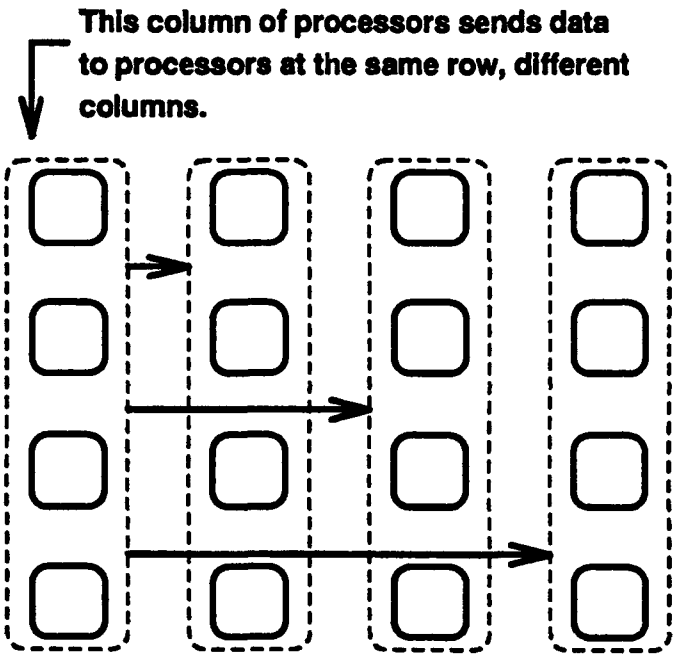

Figure 6. Step 1: Data exchange between columns of processors.

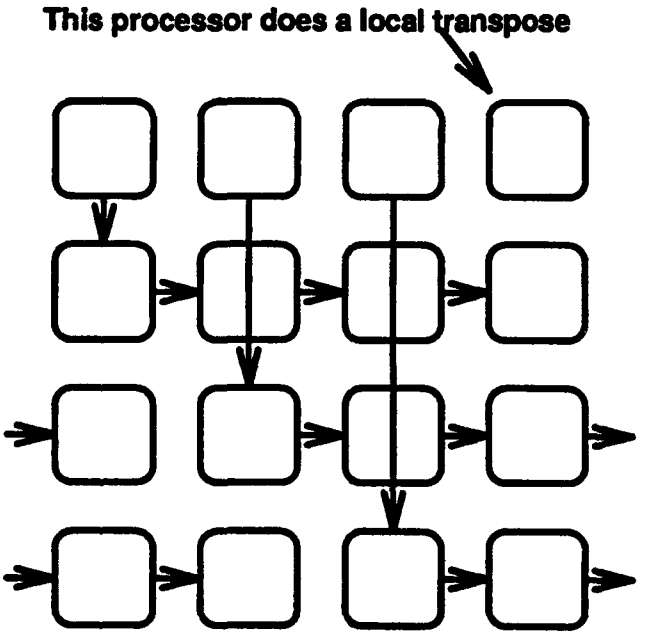

Figure 7. Step 2: Data exchange between rows of processors.

Table 1. Performance of matrix transpose (128 x 128).

COMPLEX MATRIX TRANSPOSE OF 128 x 128				
# PROCESSORS	4	16	32	64
TIME (ms) (B-STEP)	30	13	9	7
TIME (ms) (C-STEP)	15	6	4	3

Table 2. Performance of matrix transpose (256 x 256).

COMPLEX MATRIX TRANSPOSE OF 256 x 256				
# PROCESSORS	4	16	32	64
TIME (ms) (B-STEP)	125	52	40	29
TIME (ms) (C-STEP)	63	25	18	14

Table 3. Performance of matrix transpose (512 x 512).

COMPLEX MATRIX TRANSPOSE OF 512 x 512				
# PROCESSORS	4	16	32	64
TIME (ms) (B-STEP)	***	232	162	121
TIME (ms) (C-STEP)	***	167	79	55

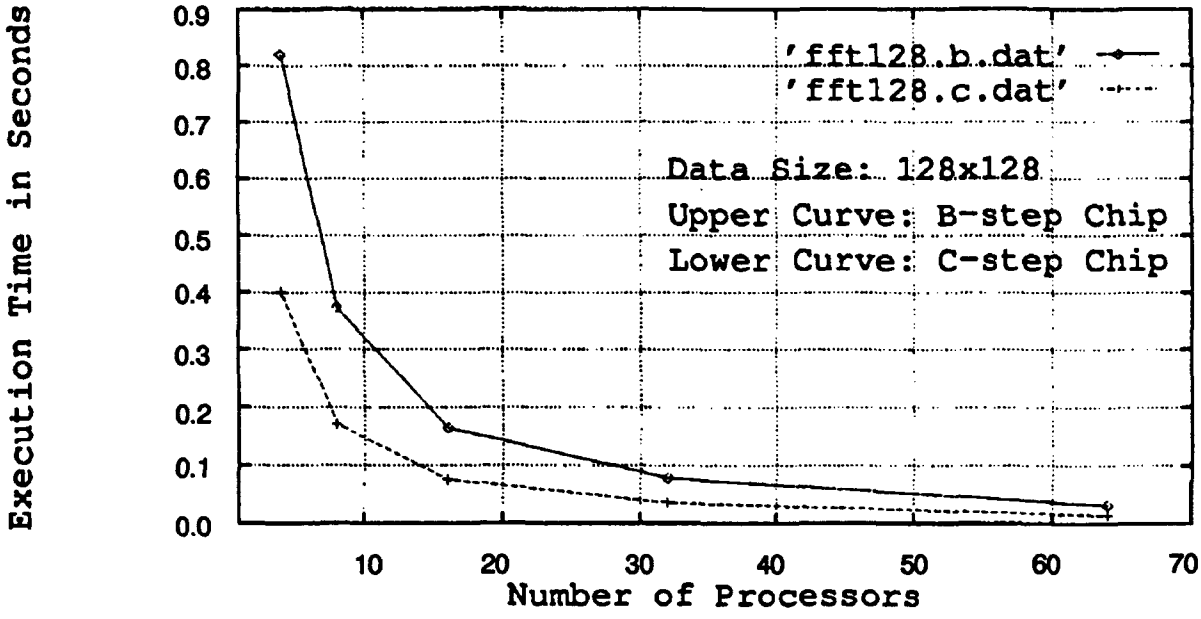

Figure 8. Performance of two-dimensional FFT (128 x 128).

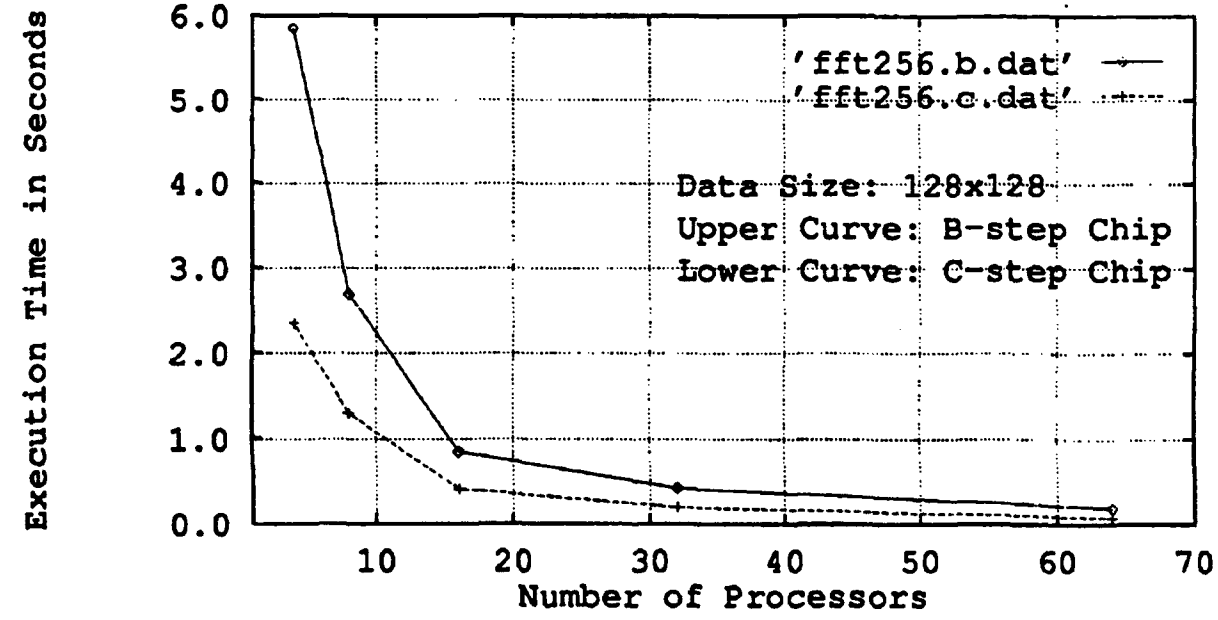

Figure 9. Performance of two-dimensional FFT (256 x 256).

Table 4. Performance of two-dimensional FFT (512 x 512).

TWO-DIMENSIONAL FFT 512 x 512		
# PROCESSORS	16	64
TIME (sec) (B-STEP)	4.814	0.923
TIME (sec) (C-STEP)	2.322	0.403

It is interesting to see from these two-dimensional FFT performances that the execution time is more than doubled when the number of processors used is just halved. This is because the transpose time increases as the number of processors used decreases. It can be seen from tables 2 and 3 that on an 8 x 8 iWarp array, the communication time (i.e., the matrix transpose) takes about 13% of the total execution time for the two-dimensional FFT of data size 512 x 512, but the communication takes about 20% of total execution time for a data size of 256 x 256. The ratio of communication to computation increases as the the data size decreases. We also note that the actual performances on C-step processors, using the compiler designed for the B-step processor, for both matrix transpose and two-dimensional FFT, are twice as fast as on B-step processors. The actual performance on a C-step processor is expected to improve significantly when the new compiler for that chip (not available to us now) is used. Using an 8 x 8 array of C-step processors, the achieved performances of two-dimensional FFT is about 57 MFLOPS for a 128 x 128 complex data array, 45 MFLOPS for a 256 x 256 complex data array and 43 MFLOPS for a 512 x 512 complex data array.

4.2 TWO-DIMENSIONAL CONVOLUTION

Convolution is a fundamental operation in signal and image processing since it is a filtering operation. Hence, a fast two–dimensional convolution operation is important to speed up many image processing applications. The mathematical definition of the discrete two-dimensional convolution is

$$y(n1, n_2) = \sum_{k_2=-\infty}^{\infty} \sum_{k_1=-\infty}^{\infty} x(k_1, k_2) h(n_1 - k_1, n_2 - k_2), \qquad (2)$$

where we assume $h(,)$ is the convolution kernel. In typical cases, we can also assume

$$x(n_1, n_2) \neq 0 \text{ only for } 0 \leq n_1, n_2 \leq N-1,$$
$$h(n_1, n_2) \neq 0 \text{ only for } 0 \leq n_1, n_2 \leq M-1,$$

and $N >> M$. With the above given supports for input and kernel functions, it is seen that

$$y(n_1, n_2) \neq 0 \text{ only for } 0 \leq n_1, n_2 \leq N+M-1 .$$

It can be shown that (see reference 6) the two-dimensional convolution using equation 2 has an operation count $O((N+M-2)^2M^2)$. If we assume the convolution operation is an intermediate step of a larger (parallel) processing algorithm and that the data are partitioned by rows consecutively (here we assume the first variable of $x(n_1, n_2)$ is row index), then an obvious way to do a parallel two-dimensional convolution is the following:

1. Broadcast or compute locally the kernel function $h(n_1, n2)$;
2. Each processor computes $y(n_1, n_2)$ for n_1 that falls in the range of the input data $x(n_1, n_2)$ and all n_2.

Since there are $N+M-2$ nonzero output rows, the processor that holds the last few rows of input data will have $M-1$ more output rows to compute. Since $M << N$ and there should be fewer computations for the last few output rows due to the fact that $x(n_1, n2) = 0$ for $n_1 \geq N$, workload is largely balanced in this case. Since computing output data points at the input data partition boundary will need input data rows outside the boundary, proper input data overlapping is necessary, which means the data transfer operation between neighboring processors (in terms of data partition, not physical location of processors) must be performed before the parallel convolution starts. The operation count for the parallel convolution is approximately $O((N+M-2)^2M^2)/P$, where P is the number of processors.

If the kernel function $h(n_1, n_2)$ is separable, we can write

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{\infty} h_1(n_1 - k_1) \sum_{k_2=-\infty}^{\infty} x(k_1, k_2) h(n_2 - k_2).$$

Now if we first compute

$$f(k_1, n_2) = \sum_{k_2=-\infty}^{\infty} x(k_1, k_2) h(n_2 - k_2)$$

for all k_1, and n_2 for which $f(k_1, n_2)$ is nonzero, then compute

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{\infty} h_1(n_1 - k_1) f(k_1, n_2)$$

for all nonzero output data points; the total operation count from these two steps is approximately (see reference 6) $NM(N+M-1)+M(N+M-1)^2$, which is about a factor of $M/2$ less than that for a nonseparable kernel. It is not difficult to see that the reduction in operation count for a separable kernel can be achieved in the parallel implementation discussed above in essentially the same way as in the sequential implementation. The operation count for the parallel implementation is $O\left(NM(N+M-1)/P+M(N+M-1)^2/P\right)$, where P is the number of processors.

4.3 NUMERICAL LINEAR ALGEBRA ALGORITHMS

In this section, we present several basic, parallel numerical linear algebra algorithms and their implementations on a message-passing parallel system. These linear algebra tools are frequently used in many signal and image processing algorithms. Most linear algebra algorithms belong to the category of fine-grain applications, which means the ratio of computations and communications is not very high. For such applications, minimizing communication overhead is crucial to get a good parallel performance.

Broadly speaking, linear algebra algorithms can be classified into two types, one for solving a linear system of equations, the other for finding eigenvalues of a matrix under various assumptions. The computation could become very expensive when the problem size (or the matrix) is large or when it is necessary to solve a certain problem many times. For real-time applications, the problem size may not be very large, but the solution to a problem must be obtained as quickly as possible. All these situations require us to have an efficient way of problem solving. Parallel processing is a popular choice in that regard.

Before constructing a parallel algorithm for mapping to a parallel system, we may want to think about possible ways of data distribution on a network of processors. Under the assumption that we want to distribute a two-dimensional data array by rows or by columns, there are two common data distribution strategies. The first strategy is to have each processor hold a few consecutive rows/columns of data. The second strategy is to distribute rows/columns of data in a round-robin way. Here is an example of the second data distribution strategy. Suppose we want to partition a matrix with eight columns among four processors. The round-robin way of data distribution will put first and fifth columns in the first processor, the second and sixth columns in the second processor, the third and seventh columns in the third processor, and the fourth and eighth columns in the fourth processor.

We now present these parallel algorithms in a pseudo-code form. We also discuss implementation issues and give performance results on the iWarp system. More detailed explanations of some of these algorithms can be found in references 7, 8, 9, and 10.

Performance results for these linear algebra algorithms were obtained from B-Step processors. We expect the performance on C-step processors would be doubled using the current compiler, as has been verified on the two-dimensional FFT code.

Suppose we need to solve a symmetric linear system and the coefficient matrix is positive definite; Cholesky algorithm can then be used. Here is a parallel Cholesky algorithm that can be implemented on a ring network of processors (reference 7). We implemented this algorithm using express communication with a ring connection of processors on the iWarp system. The performance on a 512 x 512 data matrix was measured and is shown in figure 10.

A Ring Parallel Cholesky Algorithm

```
s ← 0, j ← 0, j1 ← 1;
last ← i + (k − 1)p;
while j ≠ last
   if s + 1 ∈ {i, i + p, ···, last}
      j ← s + 1;
      Generate G(j : n, j) in gloc(j : n) and copy it into Rloc(j : n, j1);
      if s < k
         send(gloc(j : n), right);
         update Rloc(:, j1 + 1 : k);
         s ← s + 1;
      end
      j1 = j1 + 1;
   else
      receive(gloc(s + 1 : k, left);
      if s + 1 ∈ {right, right + p, ···, right + (k − 1)p}
         send(gloc(s + 1 : n), right);
      end
      s ← s + 1;
      update Rloc(:, j1 : k);
   end
end
```

After matrix factorization (an LU algorithm can be used for nonsymmetric systems), we need to solve some triangular linear systems to get the solution to the linear system. Here is a parallel program for implementing a triangular system solver. This algorithm is called a scalar-sum, fan-in algorithm in reference 10. The result of the $fan - in(buffer, pid)$ operation is that all processors send out a number stored in buffer, and the processor with ID equal to pid gets the sum of those numbers, including its own part. Map(i) is the processor ID that contains the ith column of the matrix. How the fan-in operation (which is a global sum operation) is carried out depends on the processor network.

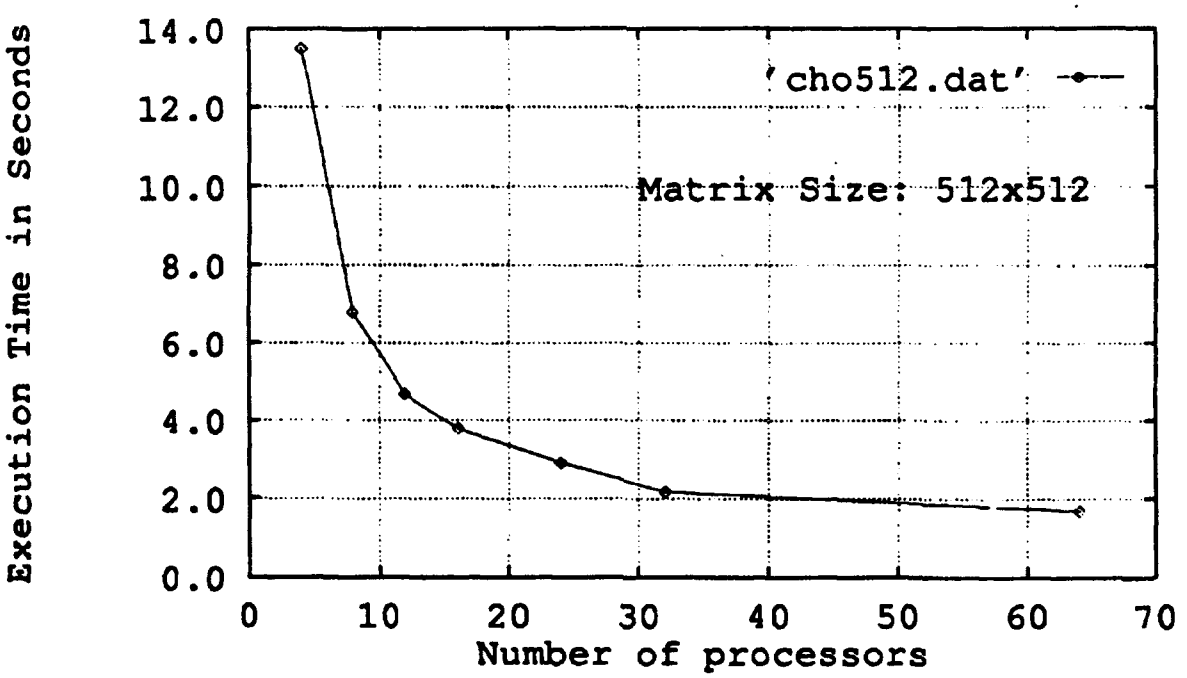

Figure 10. Performance of Cholesky factorization.

A "Fan-In" Parallel Triangular System Algorithm

```
for i = 1 : n
    s = 0
    for j = 1 : i − 1
        if j ∈ {rid, rid + p, ··· , rid + (k − 1)p}
            s = s + L_ij x_j;
    end
    s = fan-in(s, map(i))
    if j ∈ {rid, rid + p, ··· , rid + (k − 1)p}
        x_i = (b_i − s)/L_ii
end
```

The above triangular system solver algorithm can be implemented on various types of processor networks. We tried ring and two-dimensional mesh implementations. In the ring implementation, we just accumulate the sum one processor at a time along the ring. In the mesh implementation, we use the reduction sum tool to do the fan-in operation. We found that the mesh implementation is 30% faster. Figures 11 and 12 show the performance results for the mesh implementation.

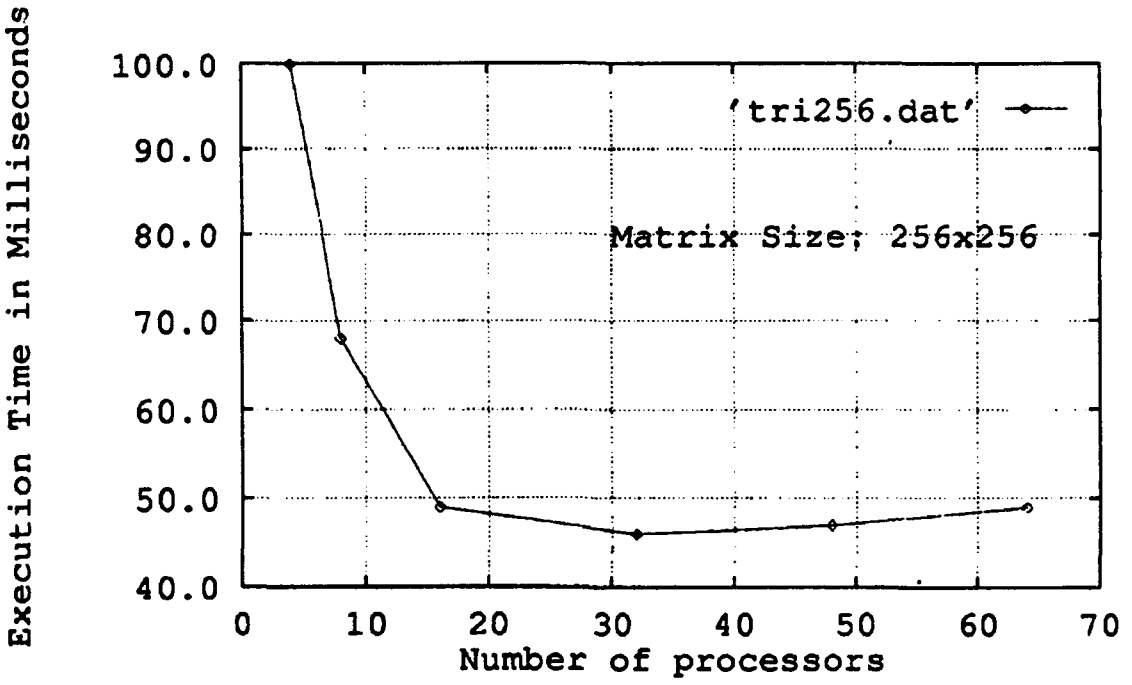

Figure 11. Performance of solving a linear triangular system (256 x 256).

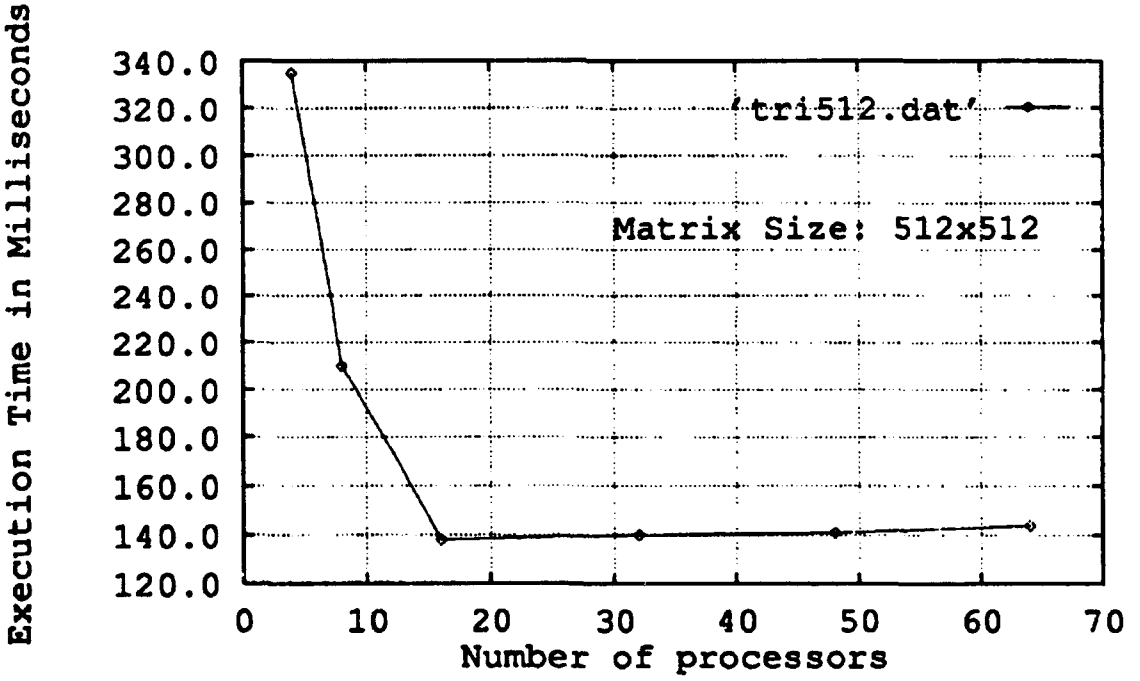

Figure 12. Performance of solving a linear triangular system (512 x 512).

It can be seen that the performance does not improve much as the number of processors used exceeds 16. This is somehow expected since the algorithm we used still contains a substantial sequential part. Another parallel algorithm for solving a triangular linear system, called the wave-front algorithm in reference 10, is also discussed in reference 10. Assuming a column-oriented data partition, here is the pseudo-code for an upper-triangular system:

A Wavefront Parallel Triangular System Algorithm

```
for j ∈ mycols
    for k = 1 to #segments
    if j! = 1 receive segment
    if k = 1 then
        x_j = (b_j − z_j)/L_jj
        segment = segment − {z_j}
    for z_j ∈ segment z_i = z_i + x_j L_ij
    if |segment| > 0 then

send segment to processor map(j + 1).
```

In the above wavefront algorithm, each column of the upper triangular matrix is divided into a number of segments. Each segment has a length σ of components. Professor map(1) first computes x_1, then proceeds to compute the components $z_i = x_1 L_{i1}$ of the first segment vector z for the first column. Once computing z is done, processor map(1) sends z to processor map(2) so that the latter can compute x_2 and then begin further updates of z. Meanwhile, processor map(1) has resumed work on the next segment of the first column. A more detailed explanation of this algorithm can be found in reference 10. Intuitively, it may perform better than the fan-in algorithm, since by controlling the parameter σ, it is possible to make all processors busy most of the time. We think a good choice of the parameter σ is to let it equal the number of processors. We have not implemented the wavefront algorithm at this moment. But the idea of this algorithm is very interesting, and it could be useful in parallelizing other applications.

Orthogonal factorization of a matrix is a useful tool. For example, a regular QR factorization on a matrix with full column rank is a numerically stable and computationally efficient approach to computing a solution to least square problems. QR factorization can also be used to find eigenvalues of a matrix. We now list a parallel QR algorithm using Householder transformations on a ring network of processors.

```
s ← 1, j ← 1;
last ← i + (k − 1)p;
next_rid ← (i + 1) mod(p);
next_last ← next_rid + (k − 1)p;
while s < n
    if s ∈ {i, i + p, ···, last}
        v(j : m) ← house(A_loc(j : m, j));
        Store v(j : m) into Q(j : m, j);
        send(v(j : m), right);
        A_loc(j : m, j : k))
         ← row.house(A_loc(j : m, j : k)), v(j : m));
        s ← s + 1, j ← j + 1;
    else
        receive(v(j : m), left);
        if s ∉ {next_rid, next_rid + p, ···, next_last}
            send(v(j : m), right);
        end
        Store v(j : m) into Q(j : m, j);
        if s ≤ last
            A_loc(j : m, j : k)
             ← row.house(A_loc(j : m, j : k)), v(j : m));
        end
        s ← s + 1;
    end
end
```

The performance results for QR factorization are given in figures 13 and 14. We can see that the parallel performance for this matrix factorization is much better than that of solving a triangular system. This is unfortunate for applications where one needs to solve a linear system with one coefficient matrix and many right-hand-side vectors.

These linear algebra algorithm tools have been used to implement an acoustic signal processing algorithm. Specifically, we picked the Minimum Variance Distortionless Response (MVDR) beamformer computation as a test case. In MVDR beamforming (reference 11), one needs to compute the following quantity (called beam power)

$$P_{MVDR} = \frac{1}{(E' R_e^{-1} E)}.$$

where $R_e = XX'$, X is a sensor output data matrix here, and we assume X has fall column rank. One way to compute P_{MVDR} is to perform a QR factorization on X first, then we can write P_{MVDR} as

$$P_{MVDR} = ((R^{-1}Q'E)'(R^{-1}Q'E))^{-1},$$

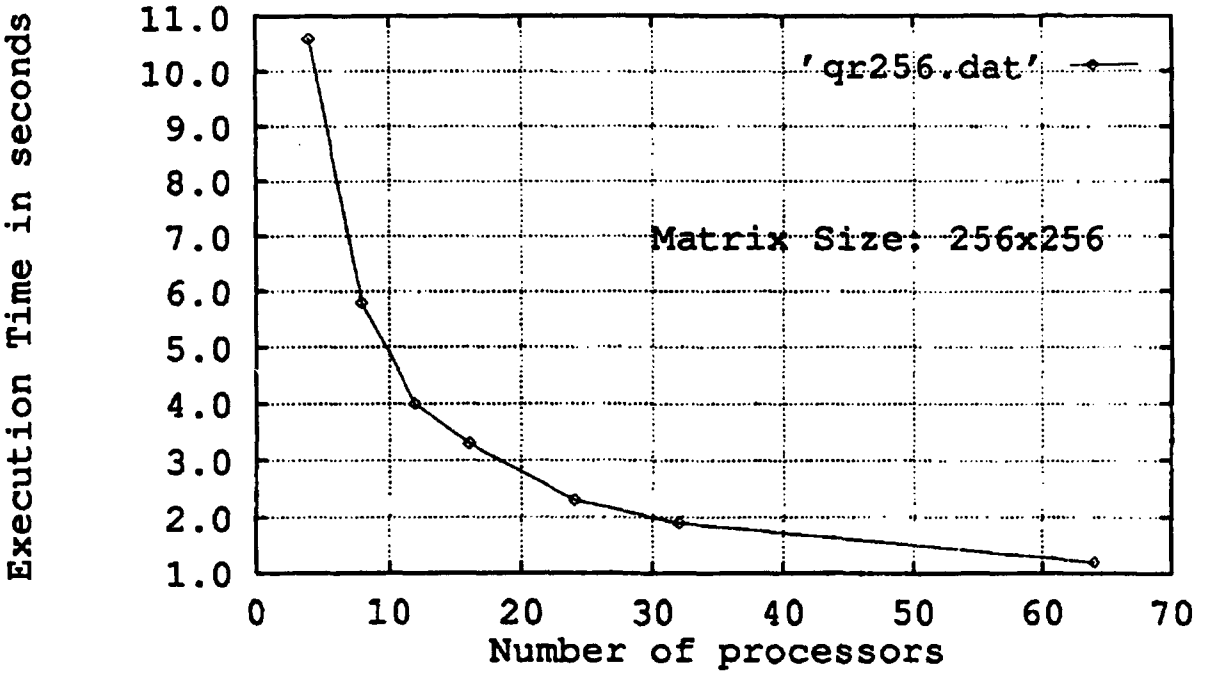

Figure 13. Performance of QR matrix factorization (256 x 256).

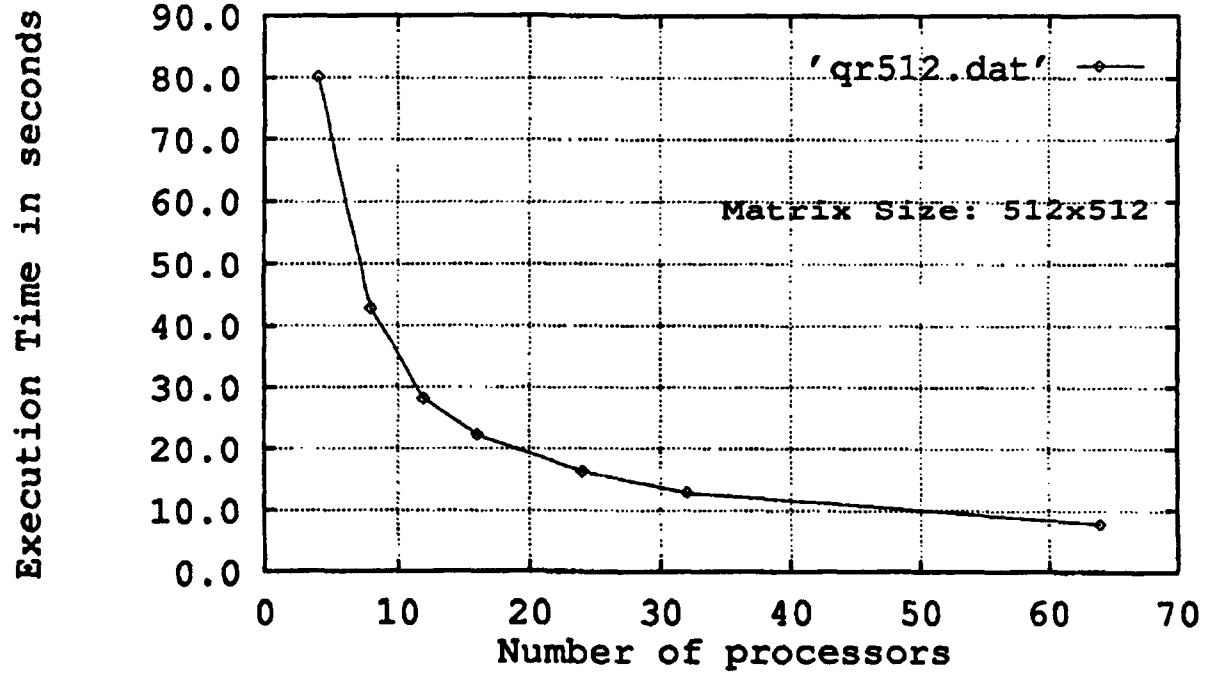

Figure 14. Performance of QR matrix factorization (512 x 512).

where Q and R are the matrix factors from QR factorization of X. Thus, the MVDR computation using QR consists of a QR factorization, a matrix-vector multiplication, and a solution of a linear upper triangular system. We do not give further details about the MVDR implementation in this report (these can be found in reference 7), but give the performance results for a 512 x 512 data matrix using different numbers of processors in figure 15.

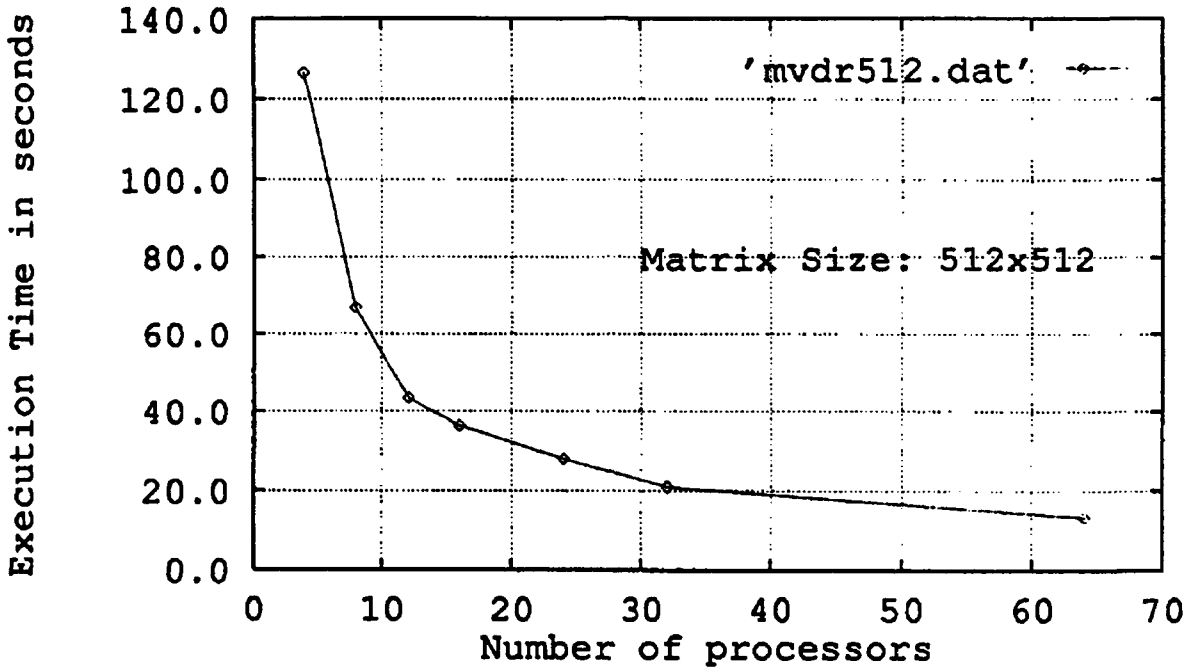

Figure 15. Performance of a MVDR beamformer.

For a rank-deficient matrix, singular value decomposition (SVD) of the matrix is a better approach for finding the minimum norm solution to the least square problem, (see reference 8). SVD also provides the option of a reduced rank approximation in the case of an ill-conditioned matrix, which is accomplished by zeroing the singular values that are smaller than a prescribed threshold. Again the programming tools we have developed, including the matrix transpose function, can be used to implement a parallel SVD algorithm on a distributed memory, message-passing system. The SVD theorem states: if A is a real, $m \times n$ matrix, then there exist orthogonal matrices

$$U = [u_1, \cdots, u_m] \in R^{m \times m} \quad and \quad V = [v_1, \cdots, v_n] \in R^{n \times n}$$

such that

$$U^T A V = diag(\sigma_1, \cdots, \sigma_p) \in R^{m \times n}$$

where $\sigma_1 \geq \sigma_2 \cdots \geq 0$. There are several ways to find the SVD factorization of a matrix; some of them need to form the matrix $A^T A$, which could give rise to an ill-conditioned matrix to work on. We now describe a parallelization strategy for a SVD algorithm proposed in reference 8. This SVD algorithm first transforms the matrix into a bidiagonal form

$$B = \begin{bmatrix} a_1 & b_1 & & \cdots & 0 \\ 0 & a_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & & 0 & a_n \end{bmatrix}.$$

Then some techniques similar to the symmetric QR algorithm also discussed in reference 8 are used to diagonalize the matrix B. This sequential SVD algorithm is listed below.

A Singular Value Decomposition Algorithm

Use Householder matrices to bidiagonalize the input $m \times n$ ($m \leq n$) matrix A:
$B \leftarrow (U_1 \cdots U_n) A (V_1 \cdots V_{n-2})$,
where U_i is $n \times m$, V_i is $n \times n$ and B is $n \times n$;
until $q = n$
For any $i = 1 : n-1$
Set $a_{i,i+1}$ to zero if $|a_{i,i+1}| < \epsilon(|a_{i,i}| + |a_{i+1,i+1}|)$;
Find the largest q and the smallest p such that if

$$B = \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix},$$

(where B_{11} is $p \times p$, B_{22} is $(n-p-q) \times (n-p-q)$ and B_{33} is $q \times q$)
then B_{33} is diagonal and B_{22} has no nonzero superdiagonal;
if $q < n$
if any diagonal entry in B_{22} is zero,
Zero the superdiagonal entry in the same row,
else
Apply a symmetric QR step to B_{22},
$B = diag(I_p, U, I_{q+m-n})^T B diag(I_p, V, I_q)$;
end
end
end

The bidiagonalization step of this algorithm basically has an operational count of twice that of QR factorization when $m \approx n$ and is therefore $O(n^3)$. The second step of this

algorithm works on a bidiagonal matrix to annihilate superdiagonal elements through an iteration process. If we need to do a SVD on a matrix in a signal or image processing algorithm, under the assumption that the data matrix is distributed in, say, columns, the first step can be parallelized by multiplying a transformation matrix to A from left, transposing the resulting matrix, multiplying a transformation matrix to A from right, transposing the resulting matrix again, and so on. Since at each step we are working on a matrix of smaller size, the size of the submatrix to be transposed may also decrease. This will require some additional decision code in a matrix transpose program. The second step of the algorithm is perhaps even more communication intensive and it is probably efficient to do it in a single processor. In summary, SVD is a fine-grain computation and it contains some sequential parts that seem to make it harder to get a good parallel performance.

4.4 THE IMPLEMENTATION OF AN IMAGE WEIGHTED FRAME-DIFFERENCING SCHEME

Using our parallel programming tools described in section 3, we implemented a multi-spectral, weighted frame-differencing scheme on the iWarp array. Given a set of images $A(i), i = 1, \cdots, n$, we would like to compute cross-covariances of pairs of images and output images of the form

$$A_d(i,j) = A(i) - \rho(i,j)A(j)$$

where $\rho(i,j)$ is the cross-covariance between A(i) and A(j). This operation is often used as one of the steps of a detection algorithm for identifying small targets embedded in background clutter, under the assumption that background clutter is statistically correlated between different image frames and targets are not. A pseudo-program for performing this is as follows:

A Parallel Weighted Differencing Program

Input and scatter a set of images by rows to the iWarp array;
Normalize each image by rows at each processor;
Compute local sum of pixel values of images;
Compute global sum of pixel values of images;
Compute means of pixel values of images at one processor;
Broadcast the means to all processors;
Compute partial cross-covariances of each pair of images at each processor;
Using the global sum to get complete cross-covariances;
Broadcast the cross-covariances to all processors;
Compute weighted frame-differences at each processor;
Gather and output differenced images from the iWarp array.

Except for distributing and gathering the data array, the weighted frame-differencing operation is highly parallelizable, an example of a coarse-grain application. In figure 16, we display a performance curve of execution time versus the number of processors used, where the execution time is measured after data scattering and the weighted difference is performed on two images of size 128 x 128. The performance was measured on the C-step array. Almost linear speed-up is achieved, as it should be expected for a typical coarse-grain application.

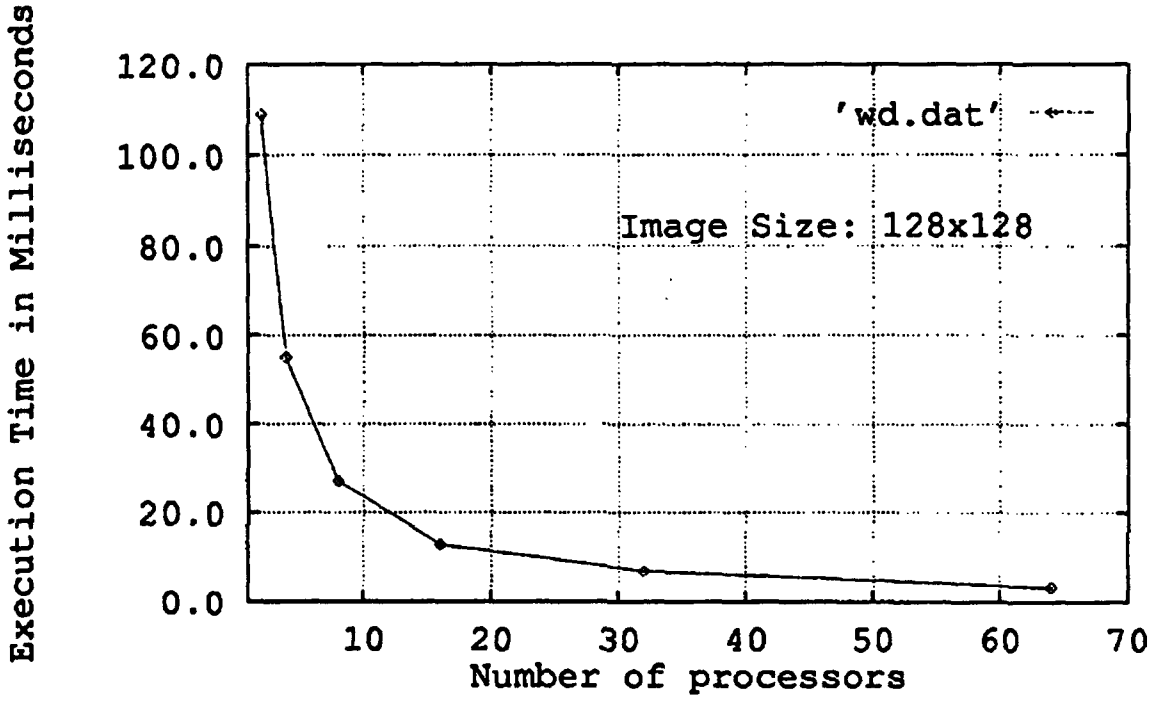

Figure 16. Performance of the weighted differencing program.

4.5 ON PARALLELIZATION OF A TWO-DIMENSIONAL ADAPTIVE LMS ALGORITHM

The two-dimensional adaptive LMS algorithm (TDLMS), as described in reference 12, is an optimal filtering algorithm with respect to the mean squared error measure. The algorithm can be used for image enhancement and is said to to be an improvement to Wiener filtering when applied to nonstationary image signals. The TDLMS algorithm as described in reference 12 is inherently sequential. We now propose a modification to that algorithm to make it possible for an efficient parallel implementation. We now give a brief description of the TDLMS algorithm and show how it may be modified for a possible parallel implementation on the iWarp system.

Suppose we have two input images, each of dimension M by M. We call the first image the desired image, D, and the second image the reference image, X. We want to filter the reference image by the operation

$$y(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} W_j(i,j) X(m-l, n-k),$$

where j is a pixel index with $j = mM + n$. The error signal e_j is defined as

$$e_j = D(m,n) - y(m,n) = D(m,n) - \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} W_j(i,j) X(m-l, n-k). \tag{3}$$

for the jth iteration. The aim is to obtain a set of weights $W(i,j)$ (which can be seen as a N by N matrix) so that the quantity

$$\text{Mean-Squared-Error} = E(e_j^2)$$

is minimized for all j. Wiener filtering does this optimization by evaluating the mean-squared-error $E(e_j^2)$ and finding the optimal weight matrix through solving a linear system of equations with an autocorrelation matrix as the coefficient matrix, under the assumption of wide-sense stationarity. The adaptive LMS algorithm described in reference 12 proceeds as follows. Using the steepest decent idea, one can modify the weight matrix recursively:

$$W_{j+1} = W_j - \nu G_j,$$

where ν is an adjustable parameter used to control the convergence speed and G_j is the gradient matrix of $E(e_j^2)$ with respect to W_j :

$$G_j(m,n) = \frac{\partial E(e_j^2)}{\partial W(m,n)}.$$

For practical application, one may use e_j in place of its statistical average. Therefore, the derivatives $G_j(m,n)$ can be computed using equation 3 and we can get an explicit equation for updating the weight matrix:

$$W_{j+1}(l,k) = W_j + 2\nu e_j X(m-l, n-k).$$

It is hoped that this iterative algorithm will converge to a weight matrix that minimizes the error e_j for all j.

Since the above adaptive algorithm iterates on the pixel index j, it is difficult to parallelize its computation. However, since the goal is to minimize e_j for all j, we argue

that the sequential updating order for the weight matrix is not necessarily the best one. There are other ways to update the weight matrix that may be more efficient. For example, we can first compute the errors e_j for all j and then average them. The updating equation for the weight matrix thus becomes

$$W^{new} = W^{old} - \nu \frac{2}{N^2} \sum_{j=0}^{N^2-1} (e_j X(m-l, n-k)). \tag{4}$$

Using the updating equation 4, we actually replace the ensemble average defined by $E(e_j^2)$ with the spatial average. In addition, this updating operation can be parallelized easily. Here is a pseudo-program for a parallel implementation of the adaptive LMS algorithm:

Distribute input images by rows (or by columns) to all processors;
/* some overlapping of rows of data is needed */
Compute and broadcast the initial weight matrix;
Each processor computes error e_j for indices j belonging to this processor;
Each processor computes a local sum of e_j;
Do a global sum of e_j and put the result in, say, processor 0;
Processor 0 computes the average of e_j and updates the weight matrix W;
If the average error is less than a threshold;
processor 0 output weight matrix, all processors stop execution;
Else
Processor 0 broadcast weight matrix to all processors,
all processors repeat from step 3.

4.6 ON A PARALLEL NEURAL NETWORK TRAINING

The neural net has become a useful tool in many signal processing and pattern recognition applications. We now describe the use of our programming tools to implement a backpropagation (BP) neural network training. A BP neural net is a mapping network that can approximate a multidimensional function. A detailed discussion on this network can be found in many books about neural networks, e.g., reference 13. Given a set of training data consisting of pairs of desired inputs and outputs, the task of training the BP neural net is to adjust the parameters of the network (called weights) so that the difference (error) between the output of the neural net and the desired output is minimized. The error function of a BP neural net can be written as

$$ER(w) = \frac{1}{N} \sum_{k=1}^{N} |f(x_k) - B(x_k, w)|^2,$$

where $f(x_k)$ is the correct value of the function $f(x)$ at x_k, which is to be approximated by the neural net. $B(x_k, w)$ is the output of the BP neural net at x_k. N is the total number of training inputs x_k. The adjustment of the network parameter vector $\mathbf{w}$, called the training of the network, is carried out according to the following rule based on the deepest decent principle:

$$w_{lij}^{new} = w_{lij}^{old} - \alpha \frac{1}{N} \sum_{k=1}^{N} \delta_{li}^{k} z_{(l-1)j}^{k}, \tag{5}$$

where l is the layer level index, i and j are node indices on level l or $l - 1$. The second term on the right-hand side of the above updating rule is the partial derivative of the error function $ER(w)$ with respect to the weight vector component w_{lij}. The parameter α in the updating rule is called the learning rate, which can be adjusted to improve the convergence speed. The training stops when the difference norm of the weight vector is below some threshold. It is often that the training process is very time-consuming and, therefore, parallel processing can be applied to speed up the training process.

It is fortunate that the training rule (equation 5) can be easily parallelized using the data partition model. In particular, we can let each processor work on a subset of the training inputs to get a partial sum in the sum for the partial derivative of the error function with respect to a weight component. Then a *global sum* operation is performed to compute the partial derivative and the sum is put in one of the processors. A broadcast operation can then be performed to replicate the partial derivative to all processors. Each processor can then use the partial derivative to update the weight component. The advantage of this parallel implementation of a BP neural net training is its simplicity. The code executed on each processor is just a sequential BP net training code plus some simple global operations that can be implemented very efficiently. Therefore, a good performance speed-up should be achieved on this application.

A limitation of the above implementation is that it only applies to the training rule (equation 5), called the batch mode training. For some applications, it has been found that a nonbatch mode training rule gives a much faster convergence rate. In the nonbatch mode training, the weight vector $\mathbf{w}$ updating is performed for each input rather than waiting until the contributions from all inputs are accumulated. To parallelize such a training process, we need to decompose the neural net structure itself and let each processor contain a part of the neural net. This is a combination of data and function partition since each input data vector also needs to be distributed among processors. Since the processing is pipelined (from input layer to output layer), it is crucial to have a high bandwidth of data input rate to the parallel system so that the input operation would not become a bottleneck of the whole processing. It seems inadequate to have only one

channel for external I/O, as does the iWarp system, for a parallel system to perform this kind of processing.

5 CONCLUSIONS

We have discussed in this report several mapping strategies for the implementations of a few programming tools, mathematical tools, and signal and image processing algorithms on the iWarp system. Our parallel applications development is based on the data partition MIMD model, which we think is a natural choice for these applications and for the iWarp system. These mapping strategies should apply to other message-passing MIMD systems as well. For example, to make the codes we developed for running on the iWarp system run on Intel's Touchstone system, one only needs to change the network configuration code and the names of message-passing functions.

Good performance speed-ups have been obtained from implementations of two-dimensional FFT, a weighted differencing algorithm for multispectral image processing, and some matrix factorization algorithms. It was also shown, from algorithm analysis and/or from actual performance data, that some applications like solving a triangular linear system and performing a singular value decomposition of a matrix is more difficult to parallelize. If we define the efficiency of a parallel application as the ratio of speed-up over the ratio of numbers of processors used, the efficiency usually improves as the data size of the application increases. For example, as can be computed from figures 11 and 12 for using 16 processors to solve a triangular linear system, the efficiencies are about 51% and 62% for data sizes of 256 and 512, respectively.

When mapping a composite application algorithm to a parallel system, one needs to parallelize different parts of the algorithm. For example, a moving target detection algorithm called the InfraRed-Search-and-Tracking (IRST) algorithm may consist of an image registration component, a weighted differencing component and a three-dimensional matched filtering component. Clearly, the parallel application developer should take the mapping of the whole IRST algorithm as an integral task when devising a mapping strategy. It is possible that a mapping strategy is efficient for one component but is not efficient for another. Thus a balance of efficiency between different components is needed to get good performance of the whole algorithm. For applications discussed in this report, we assume a two-dimensional data array is partitioned by rows or columns, and a matrix transpose is performed when the data distribution is not suitable for a parallel operation on the data.

Finally, we think for the function partition model like pipelined processing, which is popular in many real-time applications, there should be a balance between the external I/O bandwidth to the parallel system and the processing speed within the parallel system. The iWarp system has only one I/O channel connected to the host machine. This will become a performance bottleneck for real-time, pipelined processing.

6 REFERENCES

1. Z. Hussain. 1991. "*Digital Image Processing – Practical Applications of Parallel Processing Techniques*," Ellis Horwood.

2. "*iWarp Programmer's Guide*," Intel Corporation, September 1991.

3. "*iWarp C User's Guide*," Intel Corporation, September 1991.

4. B. Greer. 1991. "A Tutorial on Using iWarp PathLib," Intel Corporation.

5. W. H. Press et. al. 1988. "Numerical Recipes In C," Cambridge University Press.

6. J. S. Lim. 1990. "*Two-Dimensional Signal and Image Processing*," Prentice Hall.

7. J. Z. Lou. 1992. "An Implementation of the MVDR Beamformer on the Intel iWarp System," NCCOSC/NRaD Technical Document 2282.

8. G. Golub and C. Van Loan. 1989. "Matrix Computations," The John Hopkins University Press.

9. C. Van Loan. "A Survey of Matrix Computations," Theory Center Technical Report, Advanced Computing Research Institute, Cornell University, Ithaca, NY, pp. 14853–5201.

10. M. Heath and C. Romine. 1988. "Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors," *SIAM J. Scientific and Statistical Computing*, vol. 9, no. 3.

11. D. Johnson. 1982. "The Application of Spectral Estimation Methods to Bearing Estimation Problems," *Proceedings of the IEEE*, vol. 70, no. 9.

12. M. M. Hadhoud and D. W. Thomas. 1988. "The Two-Dimensional Adaptive LMS (TDLMS) Algorithm," *IEEE Transactions On Circuits and Systems*, vol. 35, no. 5.

13. Y-H Pao. 1989. "Adaptive Pattern Recognition and Neural Networks," Addison-Wesley.

APPENDIX A

CODES

Appendix A: main.h

```
/*
 * A head file for global declarations.
 *
 * Author: John Lou, Code 761
 * Last modified: July 21, 1992
 */

#include <sys/time.h>
#include <stdio.h>
#include <math.h>
#include <iwarp_cc.h>
#include <gates.h>
#include <regnums.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pathlib/pl.h>
#include <iwsys/getcfg.h>


typedef  int *ivector;
typedef  float *fvector;
typedef  fvector *fmatrix;

/* Define communication constants */
#define  left   0
#define  right  1
#define  up     2
#define  down   3
#define  max_out    4
#define  max_out_ring    2
#define  clk    0
#define  cclk   1


/* Function delarations */
extern int getcfg();
char *malloc();
void main();
void my_ids();
void net_connect();
void ring_mesh();
void assign_chan();

/* Other global definitions */
int _cellid, sys_height, sys_width;
int height, width, ncells;
int rowid, colid, linid, ringid;
int o_chan[max_out], i_chan[max_out];
int o_chan_ring[max_out_ring], i_chan_ring[max_out_ring];
int nrows, ncols;
```

Appendix A: mesh_ring.c

```
/*
 * Set up a 2-D toroidal mesh network with width*height
 * iWarp cells. Call a ring setup function to enbed a
 * bidirectional ring in the 2-D mesh. Assume the number
 * of columns of iWarp cells is even. The upper-left
 * subarray is used.
 *
 * Author:  John Lou,  Code 761
 * Last modified:  July 31, 1992
 *
 */


#include "main.h"  /* head file contains global decls */

#define     XR    PL_DIR_XR
#define     XL    PL_DIR_XL
#define     YU    PL_DIR_YU
#define     YD    PL_DIR_YD


/* Assign 2-D mesh ID to each cell */
void
my_ids()
{
 rowid = _cellid/sys_width;
 colid = _cellid - rowid*sys_width;
 linid = rowid*width + colid;
} /* my_ids */


void
mesh_ring()
{
  net_connect();
  /* Set up a clockwise ring */
  if(!(rowid%2)) {
    /* even rows */
    if(colid > 0 && colid < width - 1) {
      o_chan_ring[clk] = o_chan[right];
      i_chan_ring[clk] = i_chan[left];
    }
    if(colid == 0 && width > 1) {
      o_chan_ring[clk] = o_chan[right];
      i_chan_ring[clk] = i_chan[up];
    }
    if(colid == 0 && width == 1) {
      o_chan_ring[clk] = o_chan[down];
      i_chan_ring[clk] = i_chan[up];
    }
    if(colid == width - 1 && width > 1) {
      o_chan_ring[clk] = o_chan[down];
      i_chan_ring[clk] = i_chan[left];
    }
    ringid = linid;
   }
  else {
    /* odd rows */
    if(colid > 0 && colid < width - 1) {
      o_chan_ring[clk] = o_chan[left];
      i_chan_ring[clk] = i_chan[right];
    }
```

```
    if(colid == 0 && width > 1) {
      o_chan_ring[clk] = o_chan[down];
      i_chan_ring[clk] = i_chan[right];
    }
    if(colid == 0 && width == 1) {
      o_chan_ring[clk] = o_chan[down];
      i_chan_ring[clk] = i_chan[up];
    }
    if(colid == width - 1 && width > 1) {
      o_chan_ring[clk] = o_chan[left];
      i_chan_ring[clk] = i_chan[up];
    }
    ringid = linid - colid - 1 + width - colid;
  }
  return;
} /* ring_mesh */



void
net_connect()
{
  int dest, header, src_id[2], i_chan_temp;
  int i;

  /* Initialize PathLib */
  pl_init(0x0ffff);

  /* Assign PCTs for receiving cells */
  pl_rpe_configure(0x00001, 0x00002, 0x00004, 0x00008, 0x000f0);

  if(_cellid ==  sys_width*sys_height ||
     colid >= width || rowid >= height) exit(0);

  /* initializing handles */
  for(i=0; i<max_out; i++) {
    o_chan[i] = -1;
    i_chan[i] = -2;
  }

  /*
   * Create connections to other cells
   * for a regular 2-D wrap-around mesh.
   */

  /* to left */
  if(colid != 0)
   dest = _cellid - 1;
  else
    dest = _cellid + width - 1;
  o_chan[left] = pl_send_oc(PL_GATE0, XL, &dest, 1);
  pl_sendi(PL_GATE0, right);
  pl_sendi(PL_GATE0, _cellid + 1);
  /* to right */
  if(colid != width - 1)
    dest = _cellid + 1;
  else
    dest = _cellid - width + 1;
  o_chan[right] = pl_send_oc(PL_GATE0, XR, &dest, 1);
  pl_sendi(PL_GATE0, left);
  pl_sendi(PL_GATE0, -(_cellid + 1));
  /* to up */
  if(rowid != 0)
```

```
    dest = _cellid - sys_width;
  else
    dest = _cellid + sys_width*(height - 1);
  o_chan[up] = pl_send_oc(PL_GATE0, YU, &dest, 1);
  pl_sendi(PL_GATE0, down);
  pl_sendi(PL_GATE0, _cellid + 1);
  /* to down */
  if(rowid != height - 1)
    dest = _cellid + sys_width;
  else
    dest = _cellid - sys_width*(height - 1);
  o_chan[down] = pl_send_oc(PL_GATE0, YD, &dest, 1);
  pl_sendi(PL_GATE0, up);
  pl_sendi(PL_GATE0, -(_cellid + 1));

  /*
   * accept four connections from other cells
   */
  i_chan_temp = pl_recv_oc(PL_GATE1, &header);
  src_id[0] = pl_recvi(PL_GATE1);
  src_id[1] = pl_recvi(PL_GATE1);
  assign_chan(i_chan, i_chan_temp, src_id);
  i_chan_temp = pl_recv_oc(PL_GATE1, &header);
  src_id[0] = pl_recvi(PL_GATE1);
  src_id[1] = pl_recvi(PL_GATE1);
  assign_chan(i_chan, i_chan_temp, src_id);
  i_chan_temp = pl_recv_oc(PL_GATE1, &header);
  src_id[0] = pl_recvi(PL_GATE1);
  src_id[1] = pl_recvi(PL_GATE1);
  assign_chan(i_chan, i_chan_temp, src_id);
  i_chan_temp = pl_recv_oc(PL_GATE1, &header);
  src_id[0] = pl_recvi(PL_GATE1);
  src_id[1] = pl_recvi(PL_GATE1);
  assign_chan(i_chan, i_chan_temp, src_id);

} /* msh_connt */


/*
 * Assign coming-connection handles to proper
 * array components.
 */
void
assign_chan(i_chan, i_chan_temp, sid)
int  i_chan[], i_chan_temp;
int  sid[];
{
 /*
  * Accept connections from iWarp cells
  */
 /* from left */
 if((sid[1] == -(_cellid) || sid[1] == -(_cellid + width))
    &&sid[0] == left)
   {i_chan[left] = i_chan_temp; return;}
 /* from right */
 if((sid[1] == _cellid + 2 || sid[1] == _cellid - width + 2)
    && sid[0] == right)
   {i_chan[right] = i_chan_temp; return;}
 /* from up */
 if((sid[1] == -(_cellid - sys_width + 1) || sid[1] ==
     -(_cellid + sys_width*(height-1) + 1)) && sid[0] == up)
   {i_chan[up] = i_chan_temp; return;}
 /* from down */
```

```
 if((sid[1] == _cellid + sys_width + 1 || sid[1] ==
     _cellid - sys_width*(height-1) + 1) && sid[0] == down)
   {i_chan[down] = i_chan_temp; return;}
 return;
} /* assign_chan */
```

Appendix A: main.c

```
/*
 * An iWarp cell program.
 *
 * A main program showing how to use parallel programming tools
 * to do a weighted differencing of images using an iWarp array.
 *
 * Author:  John Lou, Code 761
 * Last modified:  July 22, 1992
 */


#include "main.h"

#define   mimag       2    /* maximum number of images */
#define   imagdim     128  /* image linear dimension */
#define   arydim      8    /* iWarp array linear dimension */
#define   nrow_cell   16   /* maximum rows for this cell */

void
main()
{
  struct iwcfg cfg;
  fmatrix  imag[mimag];
  imatrix  nrow_loc;
  int    i, j, ct = 0;
  float  cf, me[mimag], exe_time;


  /* get iWarp array configuration */
  getcfg(&cfg);


  /* ---------- Set up iWarp array network ---------- */

  _cellid = cfg.cellid;
  sys_height = cfg.height, sys_width = cfg.width;
  height = sys_height/2, width = sys_width/2;
  ncells = height*width;

  /* Initialize rowid and colid */
  my_ids();

  /* Initialize PathLib and set up a mesh connection */
  mesh_ring();

  /* ---------- End of network setup ---------- */



  /*
   * The weighted differencing scheme contains the
   * following steps:
   *    1) scatter the 2-D data array onto the iWarp array.
   *    2) normalize each line of image data.
   *    3) compute correlation coefficient of a pair of images.
   *    4) compute weighted difference.
   */

    nrow_loc = (imatrix) malloc(sizeof(ivector)*height);
    for(i=0; i<height; i++)
```

```
      nrow_loc[i] = (ivector) malloc(sizeof(int)*width);

    /* scatter input images */
    for(i=0; i<mimag; i++) {
      imag[i] = (fmatrix) malloc(sizeof(fvector)*nrow_cell);
      for(j=0; j<nrow_cell; j++)
        imag[i][j] = (float *) malloc(sizeof(float)*imagdim);
      scatter(imag[i], nrow_loc, imagdim, imagdim);
    }

    /* normalize each line of image data */
    for(i=0; i<mimag; i++)
      normal_line(imag[i], nrow_loc[rowid][colid], imagdim);

    /* compute correlation coefficient and sum */
    /* compute average */
    for(i=0; i<mimag; i++) {
      me[i] = sumf(imag[i], nrow_loc[rowid][colid], imagdim);
      gsumf(&me[i], 1, 0, 0);
    }
    if(_cellid==0) {
       for(i=0; i<mimag; i++) me[i] = me[i]/(imagdim*imagdim);
    }
    greplicf(me, 2);
    /* compute covariance */
    cf = covar(imag[0], imag[1], nrow_loc[rowid][colid],
               imagdim, me[0], me[1]);
    gsumf(&cf, 1, 0, 0);
    greplicf(&cf, 1);

    /* perform weighted differencing */
    wdif(imag[0], imag[1], nrow_loc[rowid][colid], imagdim, cf);

    /* gather and output weighted, differenced image */
    gather(imag[0], nrow_loc, imagdim);


  exit (0);

} /* main */
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1992	Final

4. TITLE AND SUBTITLE

APPLICATIONS DEVELOPMENT ON THE INTEL iWARP SYSTEM

5. FUNDING NUMBERS

PE: 0604507N
WU: DN308022

6. AUTHOR(S)

J. Z. Lou

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division (NRaD)
San Diego, CA 92152–5000

8. PERFORMING ORGANIZATION REPORT NUMBER

NRaD TD 2381

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Technology
Office of the Chief of Naval Research
Arlington, VA 22217

Naval Sea Systems Command
Washington, DC 20362

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report discusses the development of parallel programming tools, mathematical tools, and signal and image processing applications on the Intel iWarp system. The report begins with a discussion on parallel processing for signal and image processing. Some issues related to parallel programming on the iWarp system are discussed. The report presents algorithms and implementations of parallel programming tools on the iWarp system. The report discusses mapping applications to the iWarp system using these programming tools. The applications discussed in this report include two-dimensional fast Fourier transform (2-D FFT), two-dimensional convolution, a few matrix computation algorithms, two image processing algorithms, and a neural network algorithm. Performance results for the 2-D FFT, matrix factorization algorithms, a linear triangular system algorithm, and a multispectral weighted differencing algorithm are presented and analyzed.

14. SUBJECT TERMS

Intel iWarp System
parallel processing for signal and image processing

15. NUMBER OF PAGES

50

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE (Include Area Code)	21c. OFFICE SYMBOL
J. Z. Lou	(619) 553-3941	Code 761

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 144	V. Ware	(1)
Code 402	R. A. Wasilausky	(1)
Code 412	M. Gherrity	(1)
Code 70	E. Shutters	(1)
Code 7304	B. Marsh	(1)
Code 76	J. Wangler	(1)
Code 7601	K. Bromley	(1)
Code 761	G. Byram	(10)
Code 761	J. Lou	(20)
Code 952B	GIDEP Office	(1)
Code 961	Archive/Stock	(6)
Code 964B	Library	(2)

Defense Technical Information Center
Alexandria, VA 22304-6145 (4)

NCCOSC Washington Liaison Office
Washington, DC 20363-5100

Center for Naval Analyses
Alexandria, VA 22302-0268

Navy Acquisition, Research & Development
Information Center (NARDIC)
Washington, DC 20360-5000

Naval Air Warfare Center
Aircraft Division
Warminster, PA 18974-5000 (6)