Portable Implementation of Real Time Signal Processing Benchmarks on HPC Platforms

Jinwoo Suh and Viktor K. Prasanna

CENG 97-18

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4483

October 1997

Portable Implementation of Real-Time Signal Processing Benchmarks on HPC Platforms[†][*]

Jinwoo Suh and Viktor K. Prasanna

Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
{jinwoo + prasanna}*@halcyon.usc.edu*
http://ceng.usc.edu/~prasanna

1 Introduction

Many embedded applications require high performance computing to achieve real-time performance. These include Space-Time Adaptive Processing (STAP), Synthetic Aperture Radar(SAR), Sonar systems, Automatic Target Recognition/tracking systems, and Vision applications, among others. In these applications, high throughput is required in addition to satisfying a given latency requirement. Implementing such applications using High Performance Computing (HPC) platforms is becoming wide spread[1, 4]. HPC platforms are being used for these applications because of their high performance, scalability of solutions, and portability of the developed code. A typical HPC architecture for embedded signal processing is shown in Figure 1. Several implementation results of real-time applications on HPC platforms have been reported[2, 5, 15, 17, 19, 21].

To evaluate HPC systems for real-time applications, several benchmarks have been proposed. These include C3I[3] and MITRE benchmarks[7]. Real-time benchmarks are designed to assess real-time signal processing systems such as SAR. The following aspects distinguish real-time benchmarks from "conventional" benchmarks to evaluate HPC platforms. (1) Real-time benchmarks incorporate deadlines for completion of tasks. These deadlines must be met to achieve successful processing. (2) The execution is repeated many times to evaluate the fluctuations in run-time. (3) Throughput is one of the critical requirements in real-time embedded signal processing applications. In a real-time system, the input data rate is determined by system parameters (e.g., data collection rate in SAR processing). The system throughput must be high enough to process the input data stream in real time.

In these benchmarks, one of the essential operations is communication between processors. Because many processors are employed, algorithms for communication between processors must be efficient to obtain high performance. In real-time signal processing that requires high throughput performance, software task pipeline can be used. An example of software task pipeline for real-time STAP is shown in Figure 2 [11]. The software task pipeline consists of several stages of processors. In the above example, the STAP application is divided into 6 stages. Each stage processes incoming data and sends the processed data to the next stage. Since each stage has a different computational requirement, each stage has a different number of processors to meet the throughput requirement. In the pipeline, data remapping communication is needed between consecutive stages. For example, 22-to-176 communication is needed between the Stage 1 and the Stage 2. Since the number of processors in a stage is different from the other stages, M-to-N communication need to be performed, where $M \neq N$. In the M-to-N communication, there are M source processors and N

[*] Effort sponsored by the DoD High Performance Computing Modernization Office, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0016. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

[†] The experimental results were obtained at the Maui High Performance Computing Center (MHPCC) and at the Pittsburgh Supercomputing Center (PSC).

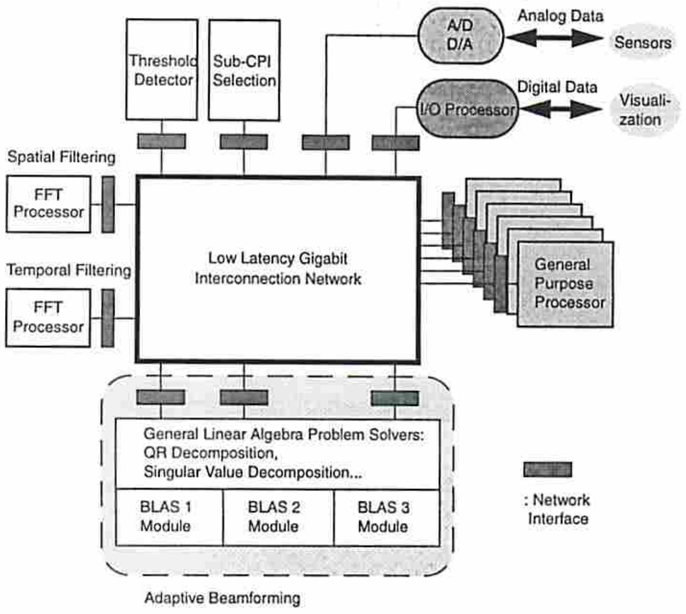


Figure 1: Typical HPC platform for embedded signal processing

destination processors. Each of the M source processors partitions its data into N sets and sends a data set to a destination processor. The M-to-N communication also arises in HPF (High Performance Fortran) when the REDISTRIBUTE directive is used[10].

To analytically evaluate the communication cost of the proposed algorithm, the General-purpose Distributed Memory (GDM) model[13, 20] is used. The GDM model captures the features of the message passing systems and has been recently used to understand communication cost and scalability. This model is simple and accurate when the number of processors is not large and the link and node contentions are not high. To send data from one processor to another, the two processors first set up a communication channel, and then send and receive the data. The overhead incurred for the initial set up is called start-up time and the data transmission overhead is transfer time. Therefore, the communication time to perform a data permutation is modeled as $T_s + D\tau_d$, where T_s is start-up time, τ_d is data transfer time per byte, and D is the amount of data transferred from each processor in bytes. Similar models have been employed by

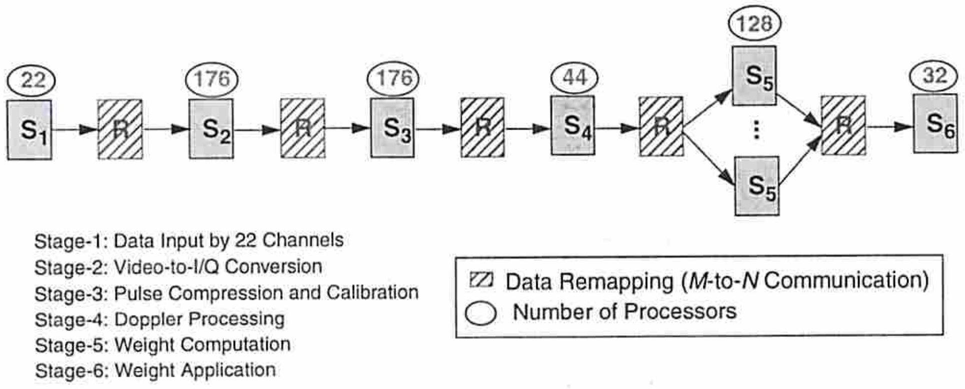

22
S1
R
176
S2
R
176
S3
R
44
S4
R
128
S5
⋮
S5
R
32
S6
Stage-1: Data Input by 22 Channels
Stage-2: Video-to-I/Q Conversion
Stage-3: Pulse Compression and Calibration
Stage-4: Doppler Processing
Stage-5: Weight Computation
Stage-6: Weight Application
Data Remapping (M-to-N Communication)
Number of Processors

Figure 2: An example of real-time STAP (a radar signal processing technique) software task pipeline

others[6, 9, 18].

In this paper, we develop a communication algorithm for M-to-N communication. In our approach, we first divide M-to-N communication problem into two subproblems: (1) M 1-to-N/M personalized communication, and (2) N/M *All-to-all* communication among M processors. The intermediate destinations for the data are deliberately determined so that the two subproblems are combined smoothly. Then, each subproblem is abstracted to a matrix transform problem to derive an efficient algorithm. For the abstraction, *destination processor table* and *circulant matrix form* are used. In the abstracted problem, row circular-shift and column circular-shift operations are used to represent the actual data movement.

The number of communication steps in the proposed communication algorithm is reduced to as small as $\lceil \lg(\lceil N/M \rceil + 1) \rceil + \lceil \lg M \rceil \approx \lg(M+N)$[1]. The communication time of the proposed algorithm is $(\lceil \lg(N/M + 1) \rceil + d + \lceil \frac{M}{2^d} \rceil - 1)T_s + (N + Md/2 + (\lceil M/2^d \rceil - 1)2^d)x\tau_d$, where d is degree of indirection and x is the data block size in bytes.

Previously, some simple approaches have been proposed for the M-to-N communication problem: (i) using *MPI_All_to_all* primitive [8], (ii) a serial algorithm, (iii) a simple parallel algorithm, and (iv) a simple indirect algorithm. In the method using *MPI_All_to_all* primitive, all of the $M + N$ processors are regarded as a set of processors and *MPI_All_to_all* primitive is used. It is not efficient since it introduces extra data communication. Even though only M source processors have to send data, N destination processors also send dummy data. Other approaches are not efficient because (1) they do not minimize the startup cost, and (2) they do not fully exploit the available network bandwidth.

The number of communication steps in a previous serial communication algorithm[5] was MN. The proposed algorithm significantly reduces the number of communication steps. For example, if $M = 16, N = 112$, the number of communication steps is only 7 instead of 22 and 112 in the simple indirect algorithm and in the simple parallel algorithm respectively.

Another advantage of our algorithm is that it can achieve high throughput performance. The proposed algorithm consists of two stages, and after the first stage, the M source processors are free to processes other tasks. However, if *MPI_All_to_all* communication primitive[8] is used, all the processors are busy during the entire communication which can reduce the resulting throughput.

Implementation results of the MITRE benchmarks on HPC platforms using the proposed algorithm are also shown. Our implementation using C and the Message Passing Interface(MPI) standard[16] is portable. For the sake of comparison, the serial communication algorithm[5] was also implemented. The execution times were measured using the MITRE real-time benchmark evaluation guidelines[7]. IBM SP2 and Cray T3D were compared using the implemented MITRE benchmarks. In general, SP2 showed large fluctuations in execution time which is undesirable for real-time applications. Thus, Cray T3D showed higher performance for real-time benchmarks. Also, our proposed algorithm results in: (1) reduction of the total communication time, (2) reduction of the minimum number of processors needed to process a specific size data, and (3) increase in the maximum input data size that can be processed.

The organization of this paper is as follows: In Section 2, the MITRE benchmarks are described briefly. Section 3 describes the previous communication algorithms for M-to-N communication. In Section 4, we discuss our approach and analyze the execution time of the proposed algorithm. Experimental results are presented in Section 5.

2 MITRE Real-time HPC Benchmarks

The MITRE benchmarks[7] were designed to provide a methodology to assess the performance of HPC platforms for real-time embedded applications. These use the *design-to-specification* methodology. In this method, both the timing and the functional specifications are given. The minimum size of a machine that can process the given problem is determined. In this section, the 2D Real-Time FFT and Corner Turn Benchmarks are briefly described.

The input data for **2D real-time FFT operation** are complex vectors of size $n \times n$, where n is a positive integer. The real-time FFT consists of two stages: Row FFT and Column FFT. In the first stage,

[1] All logarithms in this paper are to base 2.

FFT operation is computed on each row of the input data which results in intermediate data. In the second stage, the FFT operation is computed on each column of intermediate data which results in the final output.

The timing specification consists of two cases as follows: (1) Period = latency = 1 second, and (2) Period = 1 second, latency = no restriction.

The period is the time interval between successive input matrices to the machine. The latency is the elapsed time between the data input to the machine and the corresponding output. For Case 1 and Case 2, the minimum size of the machine in terms of the number of processing nodes is to be determined.

The **corner turn operation** is the transpose of a matrix distributed on more than one processor. This consists of three stages: local data rearrangement, data redistribution, and unpacking of the received messages. In the local data rearrangement, the data stored in row major order are rearranged by columns. In the second stage (data redistribution), the data are communicated among the processors for data remapping. In the third stage, in each destination processor, the received data are rearranged by columns.

3 Simple Communication Algorithms

In this section, several simple algorithms for M-to-N communication are discussed. In an M-to-N communication problem, there are M source processors and N destination processors. The M source processors and N destination processors are distinct. Each of the M source processors partitions its data into N blocks and sends a data block to a destination processor. Let x denote the amount of data in a block in bytes. Note that the total data $D = MNx$. Let $P_{0,i}, 0 \leq i \leq M-1$, be the i^{th} source processor, $P_{1,j}, 0 \leq j \leq N-1$, be the j^{th} destination processor. Also, let $C(i,j), 0 \leq i \leq M-1, 0 \leq j \leq N-1$, denote the data in the i^{th} source processor to be sent to the j^{th} destination processor. In the following discussion, M is assumed to be less than N. If $M = N$, the communication is similar to the All-to-all communication which is a special case of the M-to-N communication. Note that if $M > N$, the communication can be performed by reversing the roles of the senders and receivers of $M < N$ case.

1. All_to_all communication primitive: The easiest way of performing M-to-N communication is by using *MPI_All_to_all* communication primitive supplied by the MPI library[8]. The $M+N$ processors are regarded as a set of processors performing an *All-to-all* communication. Then, the *MPI_All_to_all* primitive is used to distribute the data in the M processors. This method is easy to program. However, it incurs extra communication time due to the transfer of the dummy data from the N processors (even though the N processors have no data to send). Also, the resulting throughput is low because all of the $M+N$ processors take part in the communication during the entire communication phase. In the following simple algorithms as well as in the proposed algorithm, only a subset of the processors is involved in the communication so that others can perform useful computational tasks.

2. Serial Algorithm [5]: The algorithm consists of MN steps. In the s^{th} step ,$0 \leq s \leq MN-1$, source processor $P_{0,\lfloor s/N \rfloor}$ sends data $C(\lfloor s/M \rfloor, s \bmod N)$ to the destination processor $P_{1,s \bmod N}$. The amount of data transferred in each step is x.

In the GDM model, this algorithm takes $MN(T_s + x\tau_d)$ time to complete, where T_s is the startup time and τ_d the data transfer time per byte. The advantage of this algorithm is simplicity. The schedule is very simple and there is very little overhead in implementing the algorithm. However, the communication time for this algorithm is proportional to both M and N resulting in poor scalability. Also, most of the available bandwidth of the system is not utilized.

3. Straightforward Parallel Algorithm: In this algorithm, the M source processors send their data in parallel. In the s^{th} step ,$0 \leq s \leq N-1$, the source processor $P_{0,i}, 0 \leq i \leq M-1$, sends data $C(i, (i+s) \bmod N)$ to destination processor $P_{1,(i+s) \bmod N}$ simultaneously. Thus, after N steps, all the communication is accomplished. In each step, the communication time is $x\tau_d$. The total execution time of the algorithm is $N(T_s + x\tau_d)$. The speed-up of the algorithm compared with the straightforward serial algorithm is M. However, the communication time is still proportional to N which results in poor scalability.

4. Simple Indirect Algorithm: In this algorithm, the N destination processors are partitioned into $\lceil N/M \rceil$ groups. The i^{th} group ,$0 \leq i \leq \lceil N/M \rceil - 1$, is denoted G_i. G_i consists of M processors $P_{1,j}, iM \leq j \leq (i+1)M-1$. For simplicity, we assume that $N/M = w$, where w is a positive integer. During the s^{th} step , $s = 0, 1, ..., N/M-1$, the source processor $P_{0,i}, 0 \leq i \leq M-1$, sends $C(i,k), sM \leq k \leq (s+1)M-1$,

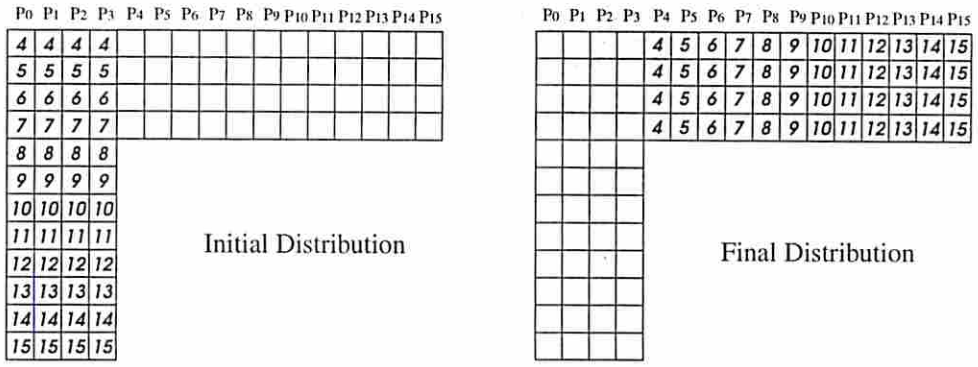


Figure 3: M-to-N communication problem represented using a *dpt* for $M = 4$ and $N = 12$. (Each square in the diagram denotes an x byte data block.)

to $P_{1,i+sM}$. In the s^{th} step, the amount of data transferred from a source processor is xM. Therefore, the communication time for each step is $T_s + Mx\tau_d$. Within each group, the processors need to exchange the data. Therefore, the total communication time is $(\lceil N/M \rceil + M - 1)T_s + x(M(\lceil N/M \rceil - 1) + M - 1)\tau_d$.

This algorithm has better performance than the serial and straightforward parallel algorithms. The speedup is MN and N compared with the serial and straightforward parallel algorithm respectively, if $N \gg M$. However, the proposed algorithm which is discussed in detail in the next section shows better performance. Thus, the simple indirect algorithm is not implemented in our experimental evaluations. However, a comparison of the simple indirect algorithm with the proposed algorithm is shown in Figure 14.

4 Proposed Algorithm

In this section, we show an efficient algorithm for M-to-N communication. In our algorithm, we assume that $M \leq N$. If $M > N$, then the communication can be performed by reversing the roles in the the senders and receivers in the $M < N$ case. Let w denote M/N. For simplicity, we assume that w is a positive integer. In our approach, we first divide the M-to-N communication problem into two subproblems. The first subproblem is a set of M 1-to-w personalized communication. By determining the intermediate destinations for the data blocks deliberately, the second subproblem becomes a set of w *All*-to-*all* communication among M processors. Then, each subproblem is abstracted to a matrix transform problem to derive an efficient communication algorithm.

Communication between processors in a message passing system incurs cost. The cost involves software cost for invocation of the send and receive system calls, buffer copy overheads, and time for data transmission over the network. Data transfer costs contribute to most of the total communication time in case of large arrays. Startup costs are dominant when the array size per node is small.

In our approach, node contention is eliminated by reorganizing the communication events. Startup cost is reduced by reorganizing the communication pattern. The array elements are combined and transferred to reduce the startup cost.

Our approach uses the notion of *destination processor table* and *circulant matrix form* [12].

Definition: An $I \times J$ *destination processor table* (*dpt*) is a table where the $j^{th}, 0 \leq j \leq J-1$, column consists of distinct indices of the destination processors of data items that processor p_j has to send.
Definition: An $M \times M$ matrix $\mathcal{M}$ is a *circulant matrix* if $m_{i,j} = m_{0,(i+j) \bmod M}, 0 \leq i,j \leq M-1$. An $N \times N$ matrix $\mathcal{N}$ is a generalized circulant matrix if (i) $\mathcal{N}$ can be partitioned into $n \times n$ macroblocks each of which size is $N/n \times N/n, 1 \leq n \leq N$, (ii) macroblock$_{i,j}$ = macroblock$_{0,(i+j) \bmod n}, 0 \leq i,j \leq n-1$, and (iii) each macroblock is in the circulant matrix form.

An example of a *dpt* and an example of an $N \times N$ *circulant matrix* are shown in Figures 3 and 4 respectively. The M-to-N communication problem can be divided into two subproblems as follows.

$$\mathbf{C} = \begin{bmatrix} c_0 & c_1 & \cdots & c_{N-1} \\ c_1 & c_2 & \cdots & c_0 \\ c_2 & c_3 & \cdots & c_1 \\ \vdots & \vdots & & \vdots \\ c_{N-1} & c_0 & \cdots & c_{N-2} \end{bmatrix}$$

Figure 4: An example of circulant matrix.

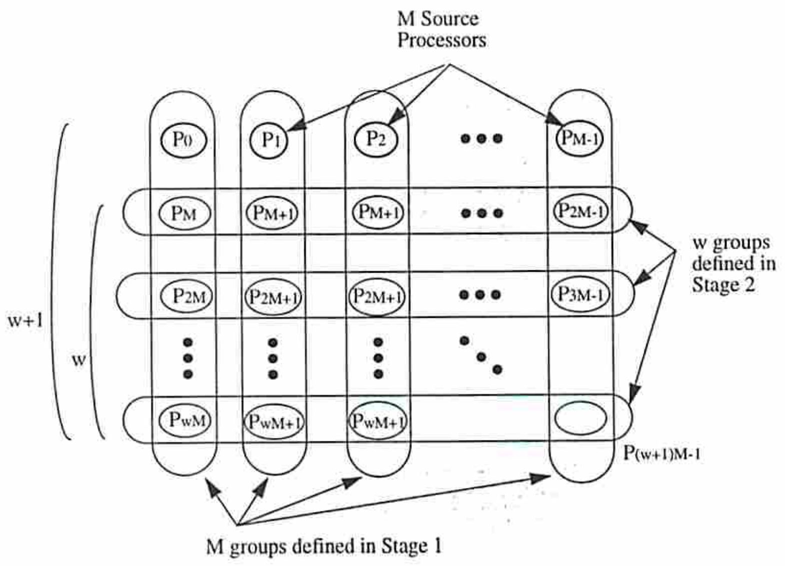


Figure 5: Partitioning of processors in the first and the second stages.

Subproblem 1 (Stage 1): The destination processors are partitioned into M groups as in Figure 5. Let P_j denote the $j^{th}, 0 \leq i \leq M-1$, source processor, and P_{j+M} denote the $j^{th}, 0 \leq i \leq N-1$, destination processor. $i^{th}, 0 \leq i \leq M-1$, group (G_{1i}) consists of the (w+1) processors $P_{kM+i}, 0 \leq k \leq w$. Each source processor $P_j, 0 \leq j \leq M-1$, distributes array elements that need to be sent to $P_l, kM \leq l \leq (k+1)M-1, 1 \leq k \leq w$, to P_{kM+j}. After Stage 1, the data to be sent to $P_i, M \leq i \leq N-1$, are located in the M processors $P_j, \lfloor i/M \rfloor M \leq j \leq \lfloor i/M+1 \rfloor M-1$. Thus, the next stage becomes a set of w *All*-to-*all* communication among the M processors.

Subproblem 2 (Stage 2): In this stage, a new processor partition consisting of w groups is defined. Let G_{2i} denote the $i^{th}, 0 \leq i \leq w-1$, group. G_{2i} consists of $P_j, (i+1)M \leq j \leq (i+2)M-1$. In each group, *All*-to-*all* communication among the M processors is performed.

The two subproblems are abstracted using a matrix notation and matrix operations. Any communication problem can be regarded as transforming the initial *dpt* to another. In our abstraction, two types of circular-shifts are performed on a *dpt*: row circular-shift and column circular-shift. In a column (row) circular-shift, elements in a column (row) are circular shifted. Thus, a column circular-shift corresponds to data rearrangement within a processor, and a row circular-shift corresponds to interprocessor communication.

The Hybrid algorithm[12] is an efficient *All*-to-*all* communication algorithm when initial *dpt* is in circulant matrix form. The Hybrid algorithm for *All*-to-*all* communication[12] is as follows.

Hybrid Algorithm [12]: If an $M \times M$ *dpt* is a circulant matrix, then the specified communication can be performed in $d + \lceil \frac{M}{2^d} \rceil - 1$ steps, where $d, 0 \leq d \leq \lg M$, is the degree of indirection. The degree of indirection is the number of indirect communication steps in which data are sent to their final destinations via intermediate processors. The remaining ($\lg M - d$) steps are performed as direct communication steps in which the data are directly sent to their final destinations. The total communication time is $(d + \lceil M/2^d \rceil - 1)T_s + (Md/2 + (\lceil M/2^d \rceil - 1)2^d)x\tau_d$.

If the above two subproblems can be represented in a circulant matrix form, then the M-to-N communication can be solved efficiently using the Hybrid algorithm. Theorem 1 shows that the two subproblems can be represented in a circulant matrix form. In addition, in Theorem 2, we show how the Subproblem 1 can be further optimized. The complete M-to-N algorithm is shown in Corollary 1.

Theorem 1 *The initial* dpt *in each stage of M-to-N communication can be represented in a circulant matrix form.*

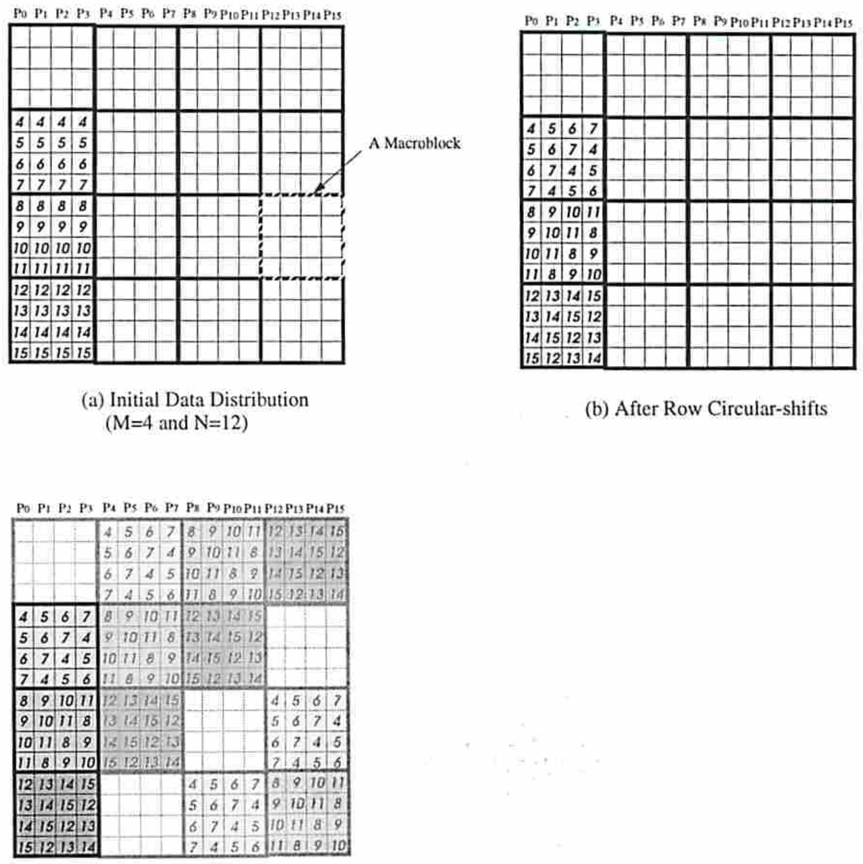
Proof: (1) Stage 1: The initial $N \times N$ *dpt* of M-to-N communication is given by,

$$dpt_{i,j} = \begin{cases} i & : \quad i \geq M \quad \text{and} \quad j \leq M-1 \\ \phi & : \quad \text{otherwise} \end{cases}$$

where ϕ denotes an empty entry in the cell. An example of *dpt* is shown in Figure 6(a). Let a *macroblock* denote each $M \times M$ portion of the *dpt*. Each macroblock is first transformed to a circulant matrix using row circular-shifts as in Figure 6(b). The $i^{th}, 0 \leq i \leq M/n-1$, column in each macroblock is up circular-shifted by i rows. Then, the entire *dpt* is transformed into circulant matrix form as follows. The elements $dpt_{i,j}, i \geq M$ or $j \leq M-1$, are empty. Thus, we may insert any dummy data as in Figure 6 (c). The dummy data is for representation purpose only; it does not actually need to be sent. Thus, if we consider Figure 6 (c), then *dpt* is in circulant matrix form.

(2) Stage 2: After the construction of the initial *dpt*, the Hybrid algorithm is used. Figure 8 shows the steps. After $\lg(w+1)$ steps, the resulting *dpt* is as shown in Figure 9(d). Note that all the data to be sent to processors in $G_{2i}, 0 \leq i \leq w-1$, are located in G_{2i}. Thus, the resulting *dpt* is the initial *dpt* for Stage 2. Therefore, the initial *dpt* of Stage 2 is also in circulant matrix form. □

Note that the circulant matrix in Figure 7(c) is simpler than the one in Figure 6(c) because $i^{th}, 0 \leq i \leq N-1$, column is up circular-shifted by i rows. However, the data transfer time is longer than that using the circulant matrix in Figure 6(c). The data transfer time in each step using the Hybrid algorithm when $d = \lg(w+1)$ is $Nx\tau_d/2$. Since there are $\lg(w+1)$ steps, the total data transfer time is $Nx\tau_d \lg(w+1)/2$. In Theorem 2, we show that the data transfer time is reduced to $Nx\tau_d$ using *dpt* in circulant matrix form

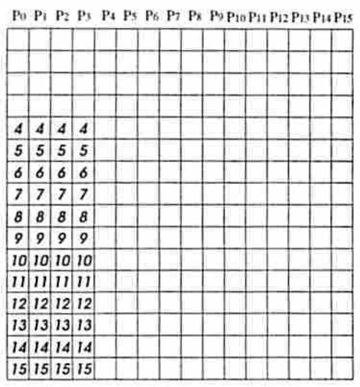


(a) Initial Data Distribution (M=4 and N=12)

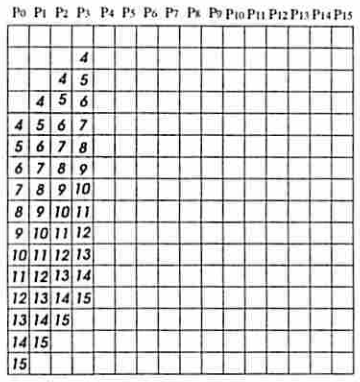
(b) After Row Circular-shifts

(c) After Inserting Dummy Data

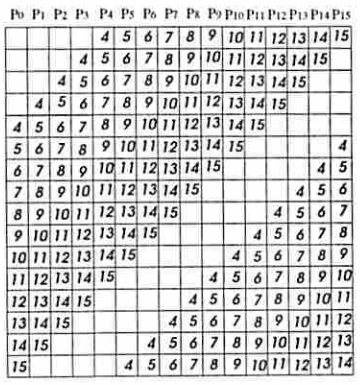
Figure 6: Transforming initial *dpt* to circulant matrix form



(a) Initial Data Distribution
(M=4 and N=12)

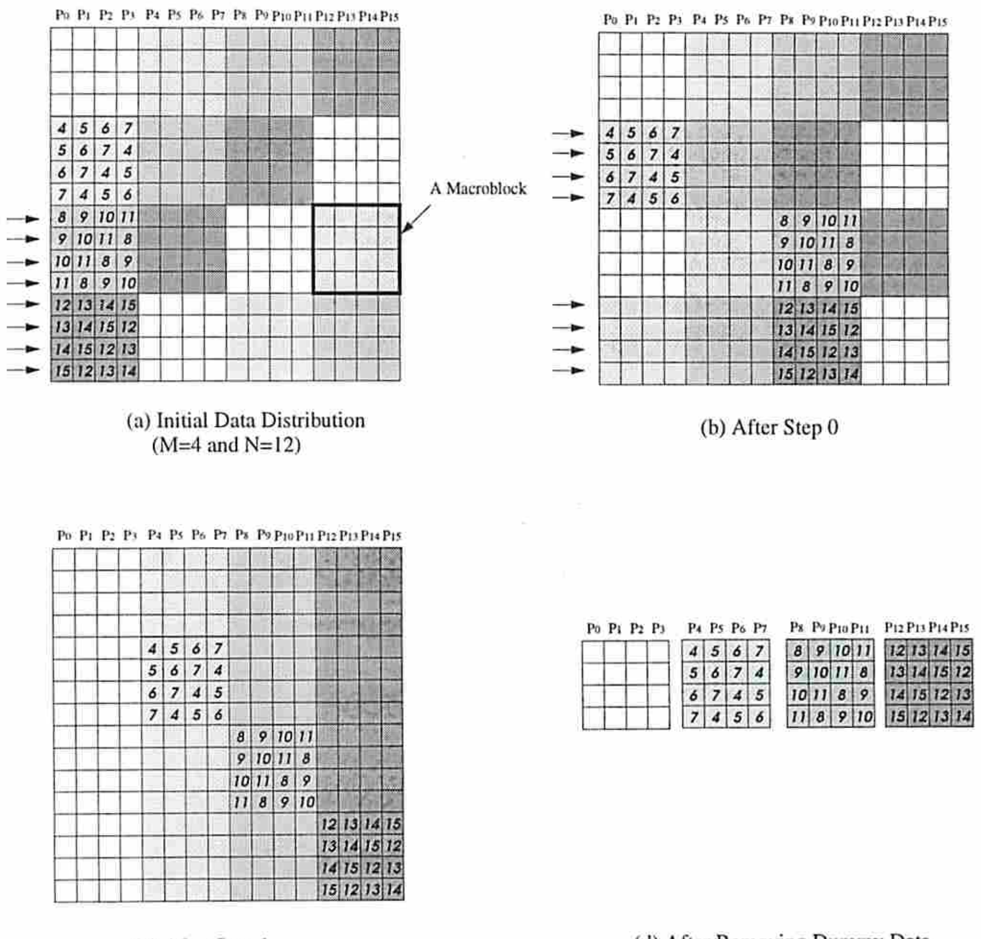


(b) After Row Circular-shifts



(c) After Inserting Dummy Data

Figure 7: Transforming initial *dpt* to a circulant matrix in the straightforward method



(a) Initial Data Distribution (M=4 and N=12)

(b) After Step 0

(c) After Step 1

(d) After Removing Dummy Data

Figure 8: Communication steps in the Stage 1 using the Hybrid algorithm

as in Figure 6(c). Also, in Theorem 2, we show that index computation does not incur data movement in intermediate processors. Index computation refers to the operation to compute the location of the data in an array to be used for communication. If the data to be sent are not in consecutive locations, the data must be compacted.

Theorem 2 *The transform of initial dpt in Stage 1 can be further optimized with respect to data transfer time and index computation time. The data transfer time is reduced to $Nx\tau_d$ from $Nx\tau_d \lg(w+1)/2$. The index computation cost is one bitwise shift operation and does not incur local data movement in the processors.*

Proof: (1) Data Transfer Time: In the Hybrid algorithm, the data transfer time in each step is $Nx\tau_d/2$. Since there are $\lg(w+1)$ steps, the total data transfer time is $\frac{Nx\tau_d \lg(w+1)}{2}$. To optimize the data transfer time, a "partition and shrink" operation is used after each communication step. Let $z = 2^{\lceil \lg(w+1) \rceil}$. Using the Hybrid algorithm, the macroblocks are circular-shifted as follows. During s^{th}, s=0,1,..., $\lceil \lg(w+1) \rceil$ -1, communication step, macroblocks in the i^{th} row, such that $i \bmod z/2^s \geq z/s^{s+1}$, are right circular-shifted by $z/2^{s+1}$. The resulting *dpt* after the Step 0 is shown in Figure 9(b). After each row circular-shift, the *dpt* is partitioned into 4 *subdpts* as in Figure 9(c). The upper-right *subdpt* and lower-left *subdpt* contain dummy data only. They are discarded. Then, the number of rows are reduced which results in less data transfer time (see Figure 9(d)). After partitioning and removing the two *subdpts*, the remaining *subdpts* maintain circulant matrix forms. Thus, in the next step, same procedure can be repeated.

The data transfer time in the first step is $(w+1-z/2)Mx\tau_d$. Then, in the s^{th} step, $s = 1, 2, ..., \lceil \lg(w+1) \rceil - 1$, the data transfer time is $z/2^{s+1}Mx\tau_d$. Thus, the total data transfer time is

$$T_d = Mx\tau_d(w+1-z/2+\sum_{s=1}^{\lceil \lg(w+1) \rceil -1} z/2^{s+1}) = Mwx\tau_d = Nx\tau_d.$$

(2) Index Computation Cost: In each step, the first data element to be sent is $\lceil D/2 \rceil$ because an intermediate processor always sends the second half of the received data, where D is the amount of data in the processor. Thus, index computation can be performed using one bitwise right shift operation on the index variable which points to the last data element in the data array.

Also, because the the second half of the received data are sent out, the data in an intermediate processor need not be reorganized. Thus, data movement in a processor is not needed.

Note that if Figure 7 is used, index computation needs to incur actual data movement in each step in the sender processors. This contributes high index computation time. Also, more data transfer time is required because the number of rows is not reduced. □

The communication steps in Stage 2 are shown in Figure 10 and Figure 11 for $d = 0$ and $d = \lg M$ respectively.

Corollary 1 *M-to-N communication can be performed in* $(\lceil \lg(w+1) \rceil + d + \lceil M/2^d \rceil - 1)T_s + (N + Md/2 + (\lceil M/2^d \rceil - 1)2^d)x\tau_d$ *communication time.*

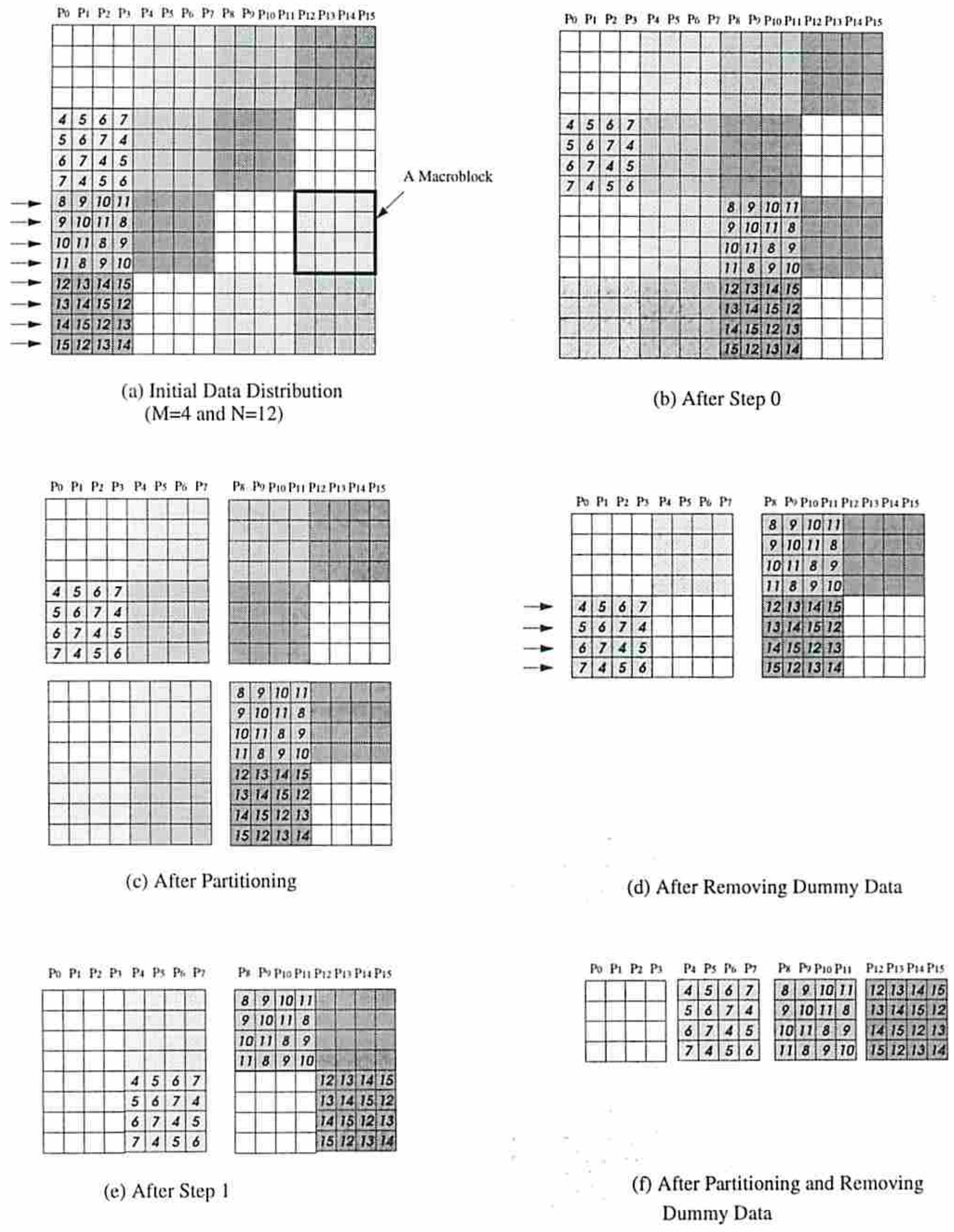
Comparisons of the previous algorithms and the proposed algorithm are shown in Figure 12.

5 Experimental Results

The Real-time FFT and corner turn benchmarks were implemented on the IBM SP2 at the Maui High Performance Computing Center (MHPCC) and on the Cray T3D at the Pittsburgh Supercomputing Center(PSC) using our communication algorithm.

A comparison of the proposed algorithm and the simple algorithms are shown in Figure 13 (proposed algorithm vs. the *MPI_All_to_all* primitive) and Figure 14 (proposed algorithm vs. the simple indirect algorithm). These results show that the proposed algorithm has better performance than the simple algorithms.

For the sake of comparison, the FFT and corner turn benchmarks were implemented using the previous serial algorithm for M-to-N communication[5]. $d = 0$ was used in our algorithm implementation.



(a) Initial Data Distribution (M=4 and N=12)

(b) After Step 0

(c) After Partitioning

(d) After Removing Dummy Data

(e) After Step 1

(f) After Partitioning and Removing Dummy Data

Figure 9: Communication steps in the Stage 1 using Theorem 2

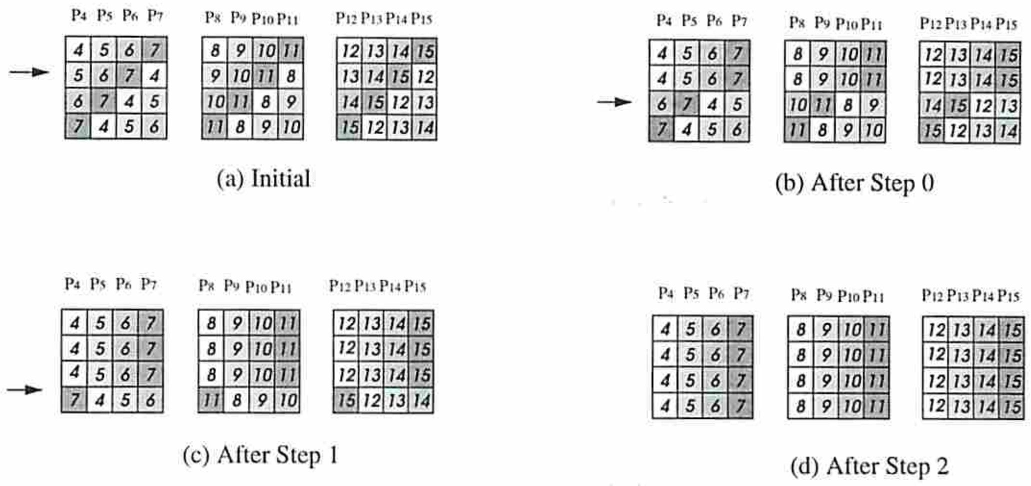


Figure 10: Communication steps in the Hybrid Algorithm ($d = \lg M$)

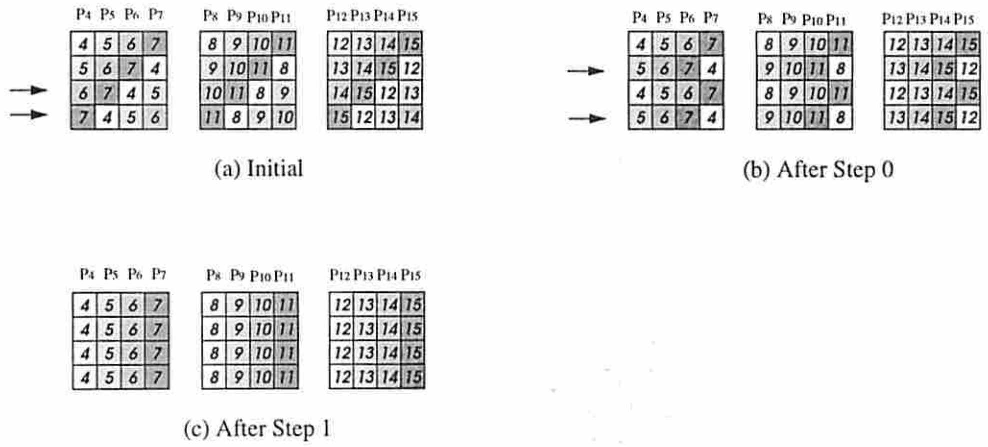


Figure 11: Communication steps in the Hybrid Algorithm ($d = 0$)

Algorithm	Startup Time	Data Transfer Time	Total Number of Communication Steps	
			M=4, N=256	M=4, N=1024
Serial	$MNTs$	$MNx\tau_d$	1024	4096
Parallel	NTs	$Nx\tau_d$	256	1024
Simple Indirect	$(w+M-1)Ts$	$(M+N-1)x\tau_d$	67	259
Proposed Algorithm	$\left(\lceil\log(w+1)\rceil + d + \left\lceil\frac{M}{2^d}\right\rceil - 1\right)Ts$	$\left(N + \frac{M}{2}d + \left\lceil\frac{M}{2^d} - 1\right\rceil 2^d\right)x\tau_d$	10(d=1)	12(d=1)

Figure 12: Comparison of the communication algorithms

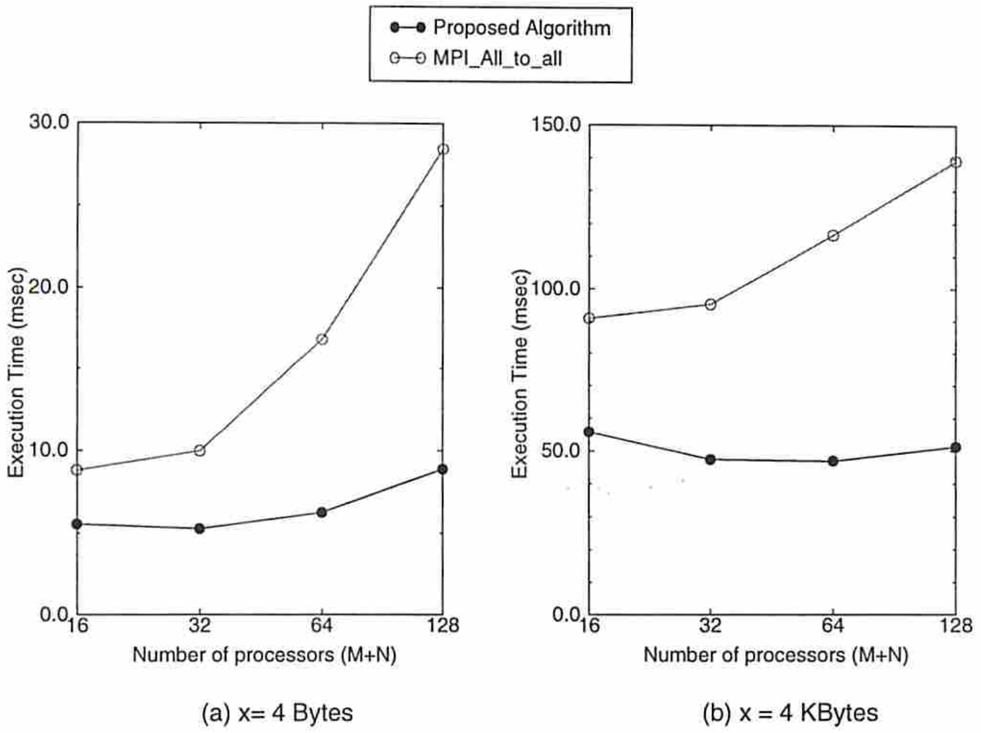


Figure 13: Comparison of the *MPI_All_to_all* primitive and the proposed algorithm (for $d = 0$) for $M = 4$ on the SP2.

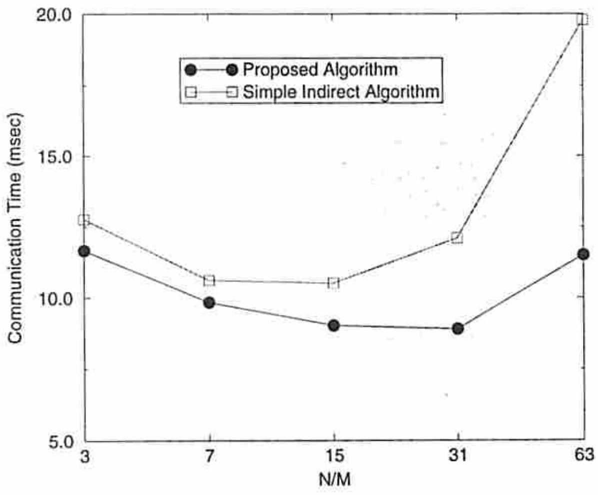
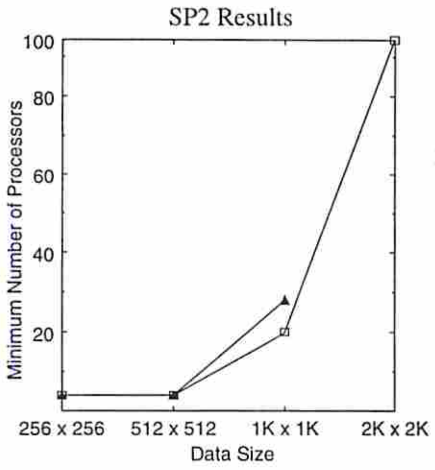
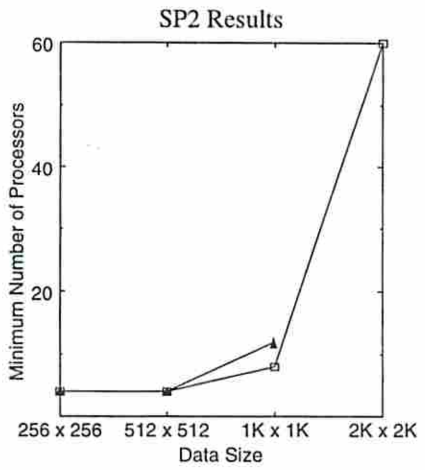


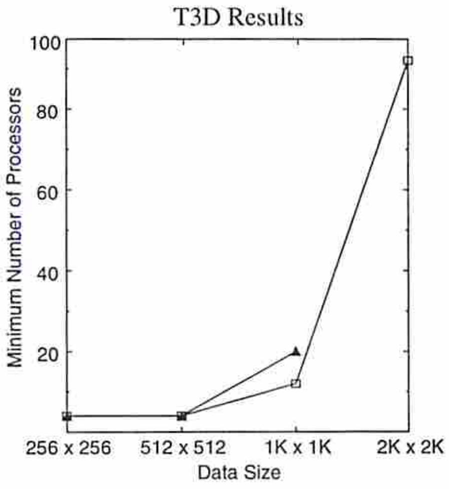
Figure 14: Comparison of the simple indirect algorithm and the proposed algorithm for $M = 2$, $x = 256$ Kbytes, on the SP2.



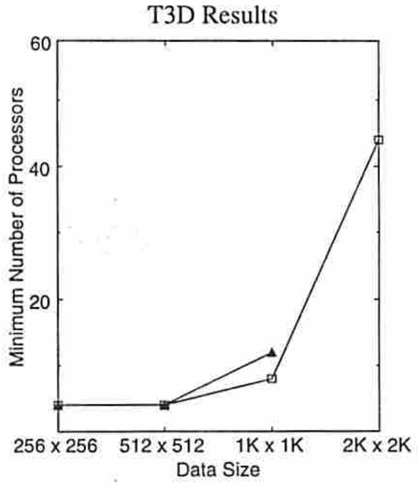
(a) latency = throughput = 1 sec.



(b) latency = infinity,
throughput = 1 sec.



(c) latency = throughput = 1 sec.



(d) latency = infinity,
throughput = 1 sec.

□–□ The Proposed Algorithm
▲–▲ The Previous Serial Algorithm

Note: Input data of size 1K x 1K can not be processed using the previous serial algorithm because (i) if the number of processors is small, the data transfer time is longer than 1 second and (ii) if the number of processors is increased, then the startup time becomes longer than 1 second.

Figure 15: FFT benchmark results

1. Real-Time 2D FFT Benchmark: The results of the 2D real-time FFT experiments are shown in Figure 15 (a) through (d). $N/M = 3$ was used in our experiments. The reason for using $N/M = 3$ (instead of $N/M = 1$) is that this implementation is to assess the performance of an M-to-N software task pipeline, where the number of processors in each stage is different from the other stages.

The minimum numbers of processors required to process the FFT for various data sizes are shown in Figure 15. The experiments were performed for two cases: (1) Case 1: latency = throughput = 1 sec, (2) Case 2: throughput = 1 sec, latency = no constraint. The experimental results show that the proposed algorithm required fewer processors than the previous serial algorithm when the data size was large. The reduction in the number of processors was possible by reducing the number of communication startups.

Another advantage of our algorithm was that the maximum size of the input data that could be processed was larger than that of the previous serial algorithm. The maximum input data size that could be processed by our algorithm and the serial algorithm were 2K × 2K and 1K × 1K respectively, given that the total number of processors $(M + N)$ that could be allocated to a task was 128.

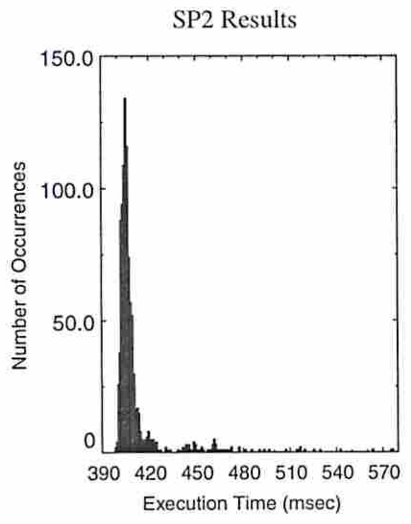
The results show that the required number of processors to perform real-time FFT on the SP2 was larger than that on the T3D. The main reason was large fluctuation of the execution time on the SP2. Some of the execution times on the SP2 are much longer than the average execution times for the same parameter settings. Because of the large variation, there was a large difference between the performance determined by using the longest execution time and that determined by using the average execution time. However, the execution times on the T3D showed little fluctuation.

2. Corner Turn Benchmark: The experimental results for the corner turn operation are shown in Figure 17. The data size was 1K × 1K in floating point format (4 bytes/data). $N/M = 3$ was used in our experiments. The corner turn throughput/node and bandwidth utilization are shown in Figure 18 and Figure 19 respectively. The maximum bandwidths of the SP2 and the T3D used in Figure 19 are from [14].

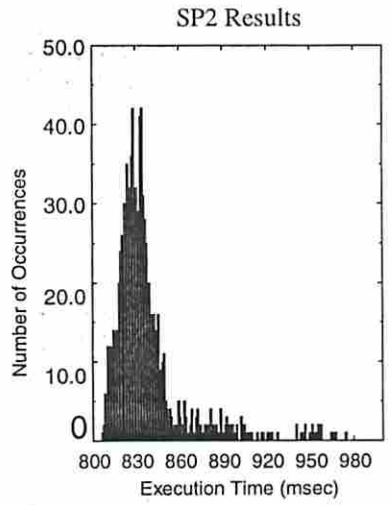
In Figure 17, the communication time of the previous algorithm increases monotonically except when the number of processors was 128 on the T3D. However, the communication time of the proposed algorithm decreases as the number of the processors increases. The reason for the sudden reduction of communication time when the number of processors is 128 on the T3D is being investigated.

In the proposed algorithm, there are two factors which affect the communication time. The first one is increased communication startup time due to the increase in the number of communication steps. The other one is reduced data transfer time from increased parallelism. As the number of source processors increases, there is more available bandwidth which is exploited by the proposed algorithm. For the range of data sizes considered in this study, the overall communication time continuously decreases as M increases. It is due to the fact that the reduction in the communication time from increased parallelism was more significant than the increase in the communication time due to the larger number of communication steps.

In the Corner Turn Behchmark, the SP2 and T3D showed mixed result. The T3D showed better performance when the previous serial algorithm is used. However, the SP2 showed better performance when the



(a) Proposed algorithm, Latency = Throughput = 1 sec.
Number of processors = 20, data size = 1K x 1K



(b) Previous algorithm, Latency = Throughput = 1 sec.
Number of processors = 28, data size = 1K x 1K

Figure 16: FFT histograms

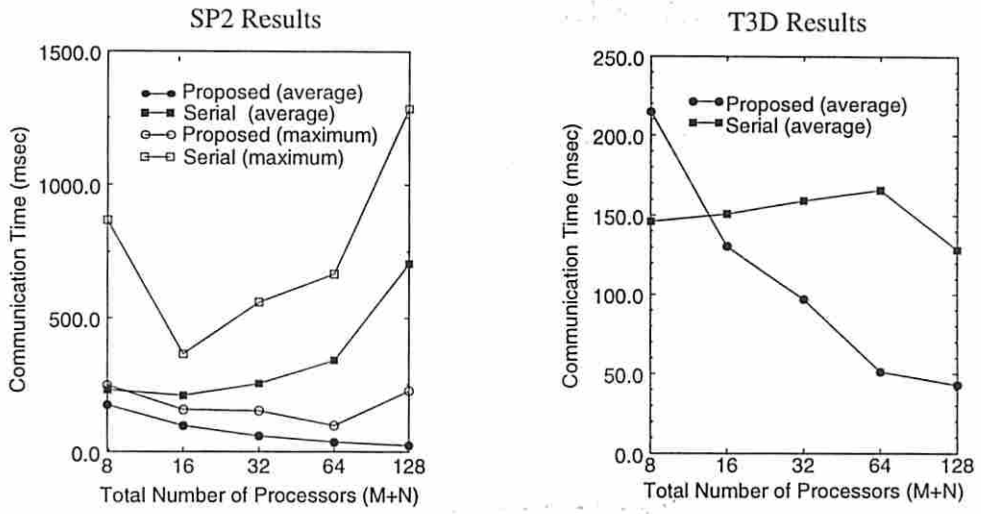


Figure 17: Corner turn benchmark implementation results

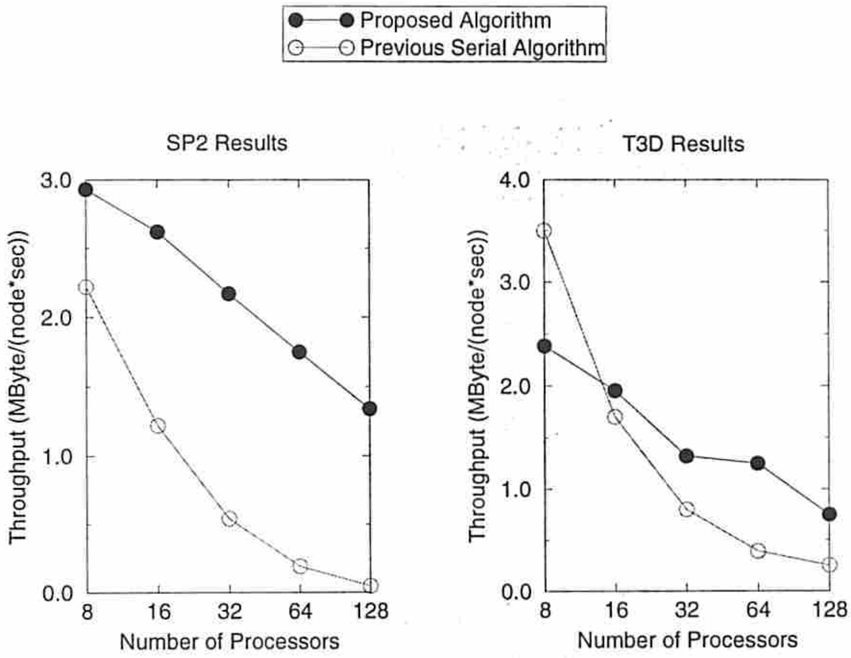


Figure 18: Corner turn throughput/node (data size = 1K × 1K, N/M=3)

proposed algorithm is used. This is due to the smaller startup time on the T3D. Because the proposed algorithm reduces the number of startups, the effect of the the reduced communication time on the performance was more significant on the SP2.

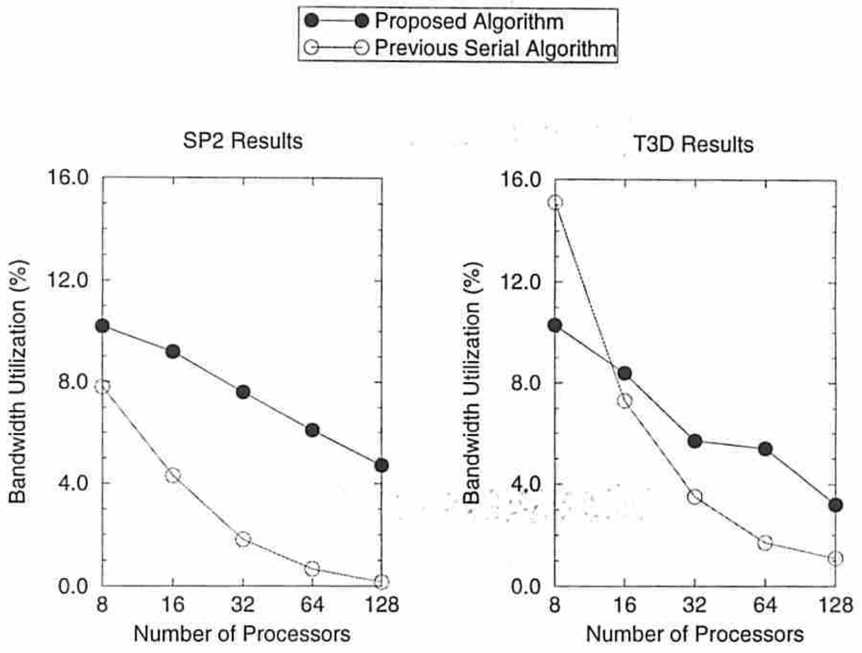


Figure 19: Corner turn bandwidth utilization (data size = 1K $\times$ 1K, N/M=3)

6 Conclusion

In this paper, we implemented real-time signal processing benchmarks on the SP2 and the T3D. For high throughput implementation of the real-time benchmarks, we presented portable M-to-N communication

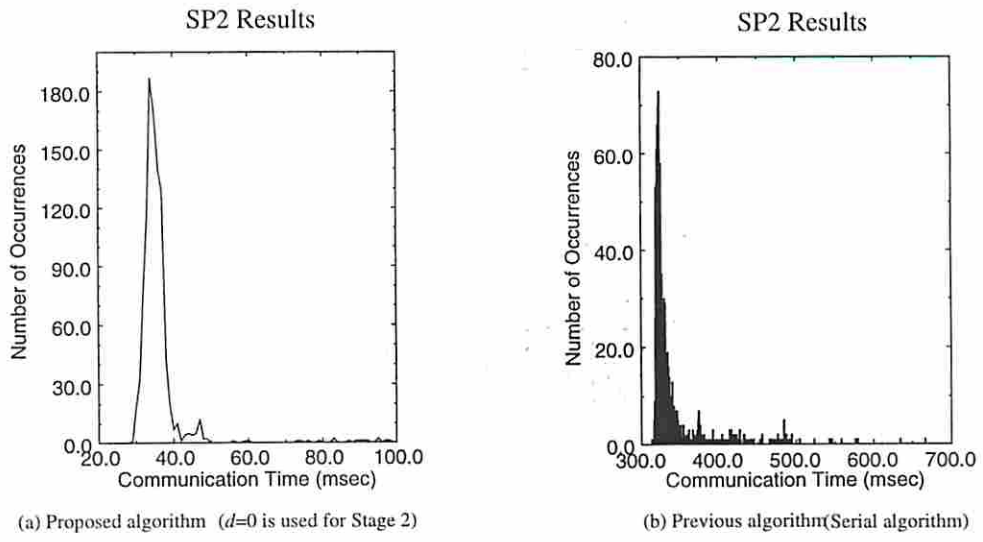


Figure 20: Histograms of the corner turn benchmark implementation (Data size = 1K $\times$ 1K, Number of processors = 16)

primitives.

Using our communication primitives, the number of communication steps was reduced to as small as $\lceil \lg(N/M+1) \rceil + \lceil \lg M \rceil \approx \lg(M+N)$. The previous serial algorithm needed MN communication steps. By using the proposed algorithm, the number of processors required to process 1K $\times$ 1K FFT was reduced by 33% on the SP2 and 40% on the T3D. For 1K $\times$ 1K corner turn, the communication time was reduced by 89% on the SP2 and 69% on the T3D when the number of processors was 64. By using our algorithm, the maximum data size that can be processed is also increased. Using 128 processors, the maximum data size that can processed by our algorithm was 2K $\times$ 2K and that by the previous serial algorithm was 1K $\times$ 1K.

Using the implemented benchmarks, we compared the performance of SP2 and T3D. In general, SP2 shows large fluctuations in execution time which is undesirable for real-time applications. Thus, T3D shows higher performance for real-time benchmarks (using both the previous serial communication algorithm and the proposed algorithm). We suspect that the fluctuation of the execution times was due to the lack of gang-scheduling on SP2. Thus, we believe the execution times will have little fluctuation when SP2 employs gang-scheduling, and the performance of real-time signal processing on SP2 will improve.

The proposed communication primitives can be used in applications that have heterogeneous characteristics so that the computations can be partitioned into series of stages and each stage has different computational requirement. Typical examples of such applications are radar, SAR, automatic target recognition, and vision problems.

References

[1] P. B. Bhat, Y. W. Lim, and V. K. Prasanna, "Issues in Using Heterogeneous HPC Systems for Embedded Real Time Signal Processing Applications," Proc. of 2nd Intl Workshop on Real-Time Computing Systems and Applications, Tokyo, Japan, October 1995.

[2] Y. Chung, and V. K. Prasanna, "Image Feature Extraction on Coarse-grain Parallel Machines," submitted to IEEE Transactions on Pattern Analysis and Machine Intelligence, 1996.

[3] Defense Advanced Research Projects Agency/Rome Lab, "Real-Time Benchmark Summary," http://www.se.rl.af.mil:8001/benchmarks/c3ipbs_rt.html, 1997.

[4] Defense Advanced Research Projects Agency, "Embeddable Systems," http://www.ito.arpa.mil/ ResearchAreas/Embeddable.html, 1996.

[5] T. Einstein, "Realtime Synthetic Aperture Radar Processing on the RACE Multicomputer," Application Note 203.0, 1996.

[6] I. Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Addison-Wesley Co., 1995.

[7] R. A. Games, "Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing," MITRE Technical Report MTR 96B0000010, March 1996.

[8] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, 1994.

[9] J. JáJá, and K. Ryu, "The Block Distributed Memory Model for Shared Memory Multiprocessors," Proceedings of International Parallel Processing Symposium, pp. 752-756, 1994.

[10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, The High Performance Fortran Handbook, The MIT Press, 1994.

[11] M. Lee and V. K. Prasanna, "High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing," 2nd International Workshop on Embedded HPC Systems and Applications (EHPC'97) at IPPS '97, Geneva, Swiss, 1997.

[12] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, "Efficient Algorithms for Block-Cycle Redistribution of Arrays," Eighth IEEE Symposium on Parallel and Distributed Processing, New Orleans, Louisiana, 1996.

[13] W. Liu, W. J. Kostis, and V. K. Prasanna, "Communication Issues in Heterogeneous Embedded Systems," Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, Honolulu, Hawai, April, 1996.

[14] W. Liu, C. Wang, and V. K. Prasanna, "Portable and Scalable Algorithms for Irregular All-to-all Communication," Proceedings of the 16th ICDCS '96, Hong Kong, 1996.

[15] P. G. Meisl and M. R. Ito, "Parallel Synthetic Aperture Radar Processing on Workstation Networks," IPPS '96, Honolulu, Hawaii, 1996.

[16] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Technical Report CS-94-230, University of Tennessee, Knoxville, May 1994.

[17] T. N. Phung, D. G. Payne, and B. Rullman, "Tutorial: Doing Parallel I/O on the Intel Paragon Supercomputer," ISUG '95, Albuquerque, NM, June, 1995.

[18] S. Ranka, R. Shankar, and K. Alsabti, "Many-to-Many Personalized Communication with Bounded Traffic," Symposium on the Frontiers of Massively Parallel Computation, pp. 20-27, February 1995.

[19] C.-L. Wang, "High Performance Computing for Vision on Distributed Memory Machines," Ph.D. Thesis, University of Southern California, August 1995.

[20] C.-L. Wang, P. B. Bhat, and V. K. Prasanna, "High-Performance Computing for Vision," Proceedings of the IEEE, Vol. 84, No.7, July, pp. 931-946, 1996.

[21] J. Ward, Space-time adaptive processing for airborne radar, Technical Report 1015, MIT Lincoln Lab., December 1994.