A Self-Adapting Distributed Memory Package for Fast Signal Transforms

Kang Chen and J. R. Johnson
Department of Computer Science
Drexel University, Philadelphia, PA 19104
kchen, jjohnson@cs.drexel.edu

Abstract

This paper presents a self-adapting distributed memory package for computing the Walsh-Hadamard transform (WHT), a prototypical fast signal transform, similar to the fast Fourier transform. A family of distributed memory algorithms are derived from different factorizations of the WHT matrix. Different factorizations correspond to different data distributions and communication patterns. Thus, searching over the space of factorizations leads to the best data distribution and communication pattern for a given platform. The distributed memory WHT package provides a framework for converting factorizations of the WHT matrix into MPI programs and exploring their performance by searching the space of factorizations.

1 Introduction

In [6] a package for automatically implementing and optimizing the Walsh-Hadamard transform (WHT) was presented, and in [2] this package was extended to provide self-optimizing parallel implementations on symmetric multiprocessors. Further sequential improvements based on the techniques of [10] and [5] have also been incorporated. The parallel package was implemented using OpenMP, a standard for shared-memory parallel programming. In this paper the package is extended further to provide parallel implementations on distributed memory parallel computers.

The package provides a flexible software architecture that can be configured to implement many different algorithms, with potentially different performance, for computing the WHT. Algorithmic choices are represented by a simple grammar which provides mathematical formulas corresponding to different algorithms. Automatic optimization is performed by searching through the space of WHT formulas for the formula that leads to the best performance. This package and method of self-adaptation is similar to the approach used by FFTW [4], a well-known and efficient package for computing the FFT.

Distributed memory parallel algorithms are also obtained from mathematical formulas corresponding to different factorizations of the WHT matrix. Using a natural interpretation of the formulas based on a block cyclic distribution of the input signal, alternate distributed memory algorithms can be generated and explored. The resulting programs are implemented using MPI, and the permutations arising in the factorizations that require global data exchange are implemented using different message passing primitives. Alternative strategies for implementing different per-

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

mutations are explored, and a factor of three improvement is obtained over an approach based on the MPI_Alltoall primitive. There are many possible sequences of permutations that can be used to implement the WHT, and since different permutations can lead to different communication patterns and costs, several possibilities are explored to select the best communication pattern on a given computing platform.

The WHT is a prototypical digital signal processing (DSP) transform with applications to signal and image processing [1] and coding theory [8]. Fast algorithms for computing the WHT are similar to the fast Fourier transform (FFT) and its variants [3, 7, 12]. The only difference is that there are no twiddle factors and bit-reversal. The lack of these extra complications allows us to focus on the role of different divide and conquer strategies and data access patterns as they relate to performance.

In Section 2 the WHT is defined and the sequential WHT package is reviewed. A family of distributed memory algorithms for computing the WHT are derived in Section 3, and the implementation of the distributed memory permutations arising in these algorithms is discussed in Section 4. Sections 5 and 6 provides empirical performance data for distributed permutation and WHT computation on a cluster of 32 processors. Concluding remarks and future work are provided in Section 7.

2 Walsh Hadamard Transform

The WHT applied to a signal x is the matrix-vector product $\mathrm{WHT}_N \cdot x$, where the signal x is represented by a vector of size $N = 2^n$ and the transform WHT_N is represented by an $N \times N$ matrix. The WHT is conveniently defined using the tensor (Kronecker) product. The tensor product of two matrices is the block matrix whose (i, j) block is equal to the (i, j) element of the first matrix multiplied by the second matrix.

$$\mathrm{WHT}_N = \bigotimes_{i=1}^{n} \mathrm{WHT}_2 = \overbrace{\mathrm{WHT}_2 \otimes \cdots \otimes \mathrm{WHT}_2}^{n},$$

where

$$\mathrm{WHT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

For example,

$$\mathrm{WHT}_4 = \mathrm{WHT}_2 \otimes \mathrm{WHT}_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Algorithms for computing the WHT can be obtained by factorizations of the transform matrix WHT_N. The following factorization encompasses a wide range, $O(7^n)$, of possible algorithms. Let $n = n_1 + \cdots + n_t$ be a partition of the exponent n and let I_m denote the $m \times m$ identity matrix.

$$\mathrm{WHT}_N = \prod_{i=1}^{t} (\mathrm{I}_{2^{n_1+\cdots+n_{i-1}}} \otimes \mathrm{WHT}_{2^{n_i}} \otimes \mathrm{I}_{2^{n_{i+1}+\cdots+n_t}}). \quad (1)$$

Equation 1 can be interpreted as a triply nested loop (see [6]). The smaller transforms, $\mathrm{WHT}_{2^{n_i}}$ are computed recursively using further applications of Equation 1. Each algorithm computed this way has exactly the same arithmetic cost, $N \lg(N)$ operations, but has different data access patterns, and consequently can have vastly different performance. The WHT package [6] provides an environment for implementing and exploring the performance of this family of algorithms. The package uses search to find the "best" algorithm on a given computing platform.

3 Distributed Memory Algorithms for Computing the WHT

Equation 1 in Section 2 can be modified to obtain a family of distributed memory parallel algorithms for computing the WHT. Equation 1 is

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

modified using properties of the tensor product. Let L^{mn}_n be the permutation defined by $L^{mn}_n(u \otimes v) = v \otimes u$, where u and v are arbitrary vectors of size m and n respectively. L is called a stride permutation since it permutes a vector by gathering elements at stride. In particular, L^{mn}_n maps the element at index $in + j$ to location $jm + i$. It is easy to show that $(L^{mn}_n)^{-1} = L^{mn}_m$ and (see the Commutation Theorem in [7]) that if A is an $m \times m$ matrix and B is and $n \times n$ matrix, $L^{mn}_m(B \otimes A)L^{mn}_n = A \otimes B$.

Applying the Commutation theorem to each of the factors in Equation 1 and observing that $I_m \otimes I_n = I_{mn}$ leads to the following equation.

$$\text{WHT}_N = \prod_{i=1}^{t} L^N_{N(i)}(\text{I}_{N/N_i} \otimes \text{WHT}_{N_i})L^N_{N/N(i)}, \tag{2}$$

where $N_i = 2^{n_i}$ and $N(i) = N_1 \cdots N_i = 2^{n_1+\cdots+n_i}$. Using the multiplicative property of stride permutations, $L^{rst}_s L^{rst}_t = L^{rst}_{st}$, the permutations in consecutive factors can be combined to obtain

$$\text{WHT}_N = \prod_{i=1}^{t} L^N_{N_i}(\text{I}_{N/N_i} \otimes \text{WHT}_{N_i}). \tag{3}$$

This equation is similar to the Pease variant of the FFT [11].

Equations 2 and 3 can be interpreted as distributed memory parallel algorithms where the permutations L exchange data amongst the processors. Let $N = 2^n$ and assume that the number of processors $P = 2^p$ and that the input vector x is equally distributed amongst the P processors in a block cyclic distribution. Let $(i_{n-1}i_{n-2}\cdots i_0)$ be the binary representation of the index of the i-th element of x, then the leading p bits (pid, the processor id) indicate the processor containing $x(i)$ and the trailing $n - p$ bits indicate the local offset where $x(i)$ is stored. Using this distribution, the computation of $I_{N/N_i} \otimes W_{N_i}$, where $N_i \leq N/P$, is obtained by having each processor compute $I_{N/N_iP} \otimes W_{N_i}$. The stride permutations redistribute the data amongst the processors. Moreover, the indexing required by the redistribution is easily determined from the binary representation of the indices. The index calculation, scheduling, and communication patterns of these permutations are explored in the next section.

Because of the two identity matrices, there are many permutations Θ_i such that $\Theta_i^{-1}(I_{2^{n-n_i}} \otimes W_{2^{n_i}})\Theta_i = (I_{2^{n_1+\cdots+n_{i-1}}} \otimes W_{2^{n_i}} \otimes I_{2^{n_{i+1}+\cdots n_t}})$. This degree of freedom can be used to design alternative distributed memory WHT algorithms of the form

$$\text{WHT}_N = \prod_{i=1}^{t} \Pi_i(\text{I}_{N/N_i} \otimes \text{WHT}_{N_i}), \tag{4}$$

where Π_i, $i = 1, \ldots, t$ is a sequence of permutations. The sequence Π_i should be chosen to minimize communication cost on a given distributed memory computer.

4 Bit Permutations

In this section the distributed memory implementation of permutations arising in Equation 4 is discussed. In order to simplify the implementation, the permutations Π_i are restricted to a class of permutations called bit (or tensor) permutations, which include stride permutations and other well known permutations such as bit-reversal. A bit permutation is a permutation of $N = 2^n$ elements obtained by permuting the bits in the binary representation of the indices $i = 0, \ldots, N-1$. Table 1 shows three examples of size $2^4 = 16$. Let σ be a permutation of the integers $\{0, 1, \ldots, n-1\}$. The bit permutation B_σ is the permutation of $\{0, 1, \ldots, 2^n - 1\}$ obtained by permuting the bits in the binary representation of the numbers $i = 0, 1, \ldots, 2^n - 1$. Using cycle notation for σ, the permutations in Table 1 are $B_{(2,3)}$, $B_{(0,2)}$ and $B_{(1,3)(0,2)}$. The stride permutation $L^N_{2^k}$ is the bit permutation $B_{(0,k-1,k-2,\ldots,1)}$ obtained by rotating the bits k positions to the right.

Not all bit permutations can be used for the permutations occurring in Equation 4. In the derivation of Equation 4, the permutation Θ_i must

Table 1. Example Bit Permutations

$b_2b_3b_1b_0$	$b_3b_0b_1b_2$	$b_1b_0b_3b_2$
$0000 = 0$	$0000 = 0$	$0000 = 0$
$0001 = 1$	$0100 = 4$	$0100 = 4$
$0010 = 2$	$0010 = 2$	$1000 = 8$
$0011 = 3$	$0110 = 6$	$1100 = 12$
$1000 = 8$	$0001 = 1$	$0001 = 1$
$1001 = 9$	$0101 = 5$	$0101 = 5$
$1010 = 10$	$0011 = 3$	$1001 = 9$
$1011 = 11$	$0111 = 7$	$1101 = 13$
$0100 = 4$	$1000 = 8$	$0010 = 2$
$0101 = 5$	$1100 = 12$	$0110 = 6$
$0110 = 6$	$1010 = 10$	$1010 = 10$
$0111 = 7$	$1110 = 14$	$1110 = 14$
$1100 = 12$	$1001 = 9$	$0011 = 3$
$1101 = 13$	$1101 = 13$	$0111 = 7$
$1110 = 14$	$1011 = 11$	$1011 = 11$
$1111 = 15$	$1111 = 15$	$1111 = 15$

have the property that $\Theta_i^{-1}(I_{2^{n-n_i}} \otimes W_{2^{n_i}})\Theta_i = (I_{2^{n_1+\cdots+n_{i-1}}} \otimes W_{2^{n_i}} \otimes I_{2^{n_{i+1}+\cdots n_t}})$. Let $n(i) = n_1 + \cdots + n_i$. Then, if Θ_i is a bit permutation, the bits in positions $n-n(i)$ through $n-n(i+1)$ must be permuted to the leading n_i bits. The remaining bits can be permuted in an arbitrary fashion. This leaves $(n - n_i)!$ possible choices for Θ_i.

The redistribution of data of size $N = 2^n$ stored in a block cyclic distribution on $P = 2^p$ processors caused by a bit permutation can be determined by the permutation of the bits in the pid (leading p bits) and offset (trailing $n - p$ bits) fields. For example, assume that 16 elements are stored on 4 processors. Initially processor 0 stores x_0, x_1, x_2, x_3, processor 1 stores x_4, x_5, x_6, x_7, processor 2 stores x_8, x_9, x_{10}, x_{11}, and processor 3 stores $x_{12}, x_{13}, x_{14}, x_{15}$. After the bit permutation that swaps bits 2 and 3 (the first permutation in Table 1) processor 0 contains x_0, x_1, x_2, x_3, processor 1 contains x_8, x_9, x_{10}, x_{11}, processor 2 contains x_4, x_5, x_6, x_7, and processor 3 contains $x_{12}, x_{13}, x_{14}, x_{15}$. This permutation permutes the entire blocks of elements at each processor. In this case only processors 1 and 2 exchange blocks. Figure 1 shows this communication pattern.

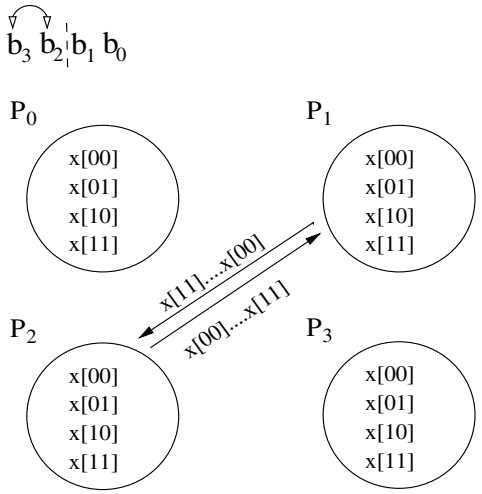

Figure 1. Dataflow on global bit permutation

After permuting the data with the second permutation in Table 1, processor 0 contains x_0, x_4, x_2, x_6, processor 1 contains x_1, x_5, x_3, x_7, processor 2 contains $x_8, x_{12}, x_{10}, x_{14}$, and processor 3 contains $x_9, x_{13}, x_{11}, x_{15}$. Observe that half of processor 0's data is exchanged with half of processor 1's data, and half of processor 2's data is exchanged with processor 3's data. Figure 2 shows the communication pattern required by this permutation. Similarly after permuting the data with the third permutation in Table 1, processor 0 contains x_0, x_4, x_8, x_{12}, processor 1 contains x_1, x_5, x_9, x_{13}, processor 2 contains x_2, x_6, x_{10}, x_{14}, and processor 3 contains x_3, x_7, x_{11}, x_{15}. In this case the communication pattern is "all-to-all" where each processor sends one fourth of its data to every processor including itself. Figure 3 shows the communication pattern required by this permutation.

These examples can be generalized. Let k be the number of bits that are exchanged between the pid and offset fields. Then each process sends $N/2^k$ data elements each to 2^k different processes. If no bits cross the pid/offset boundary then the corresponding bit permutation induces a

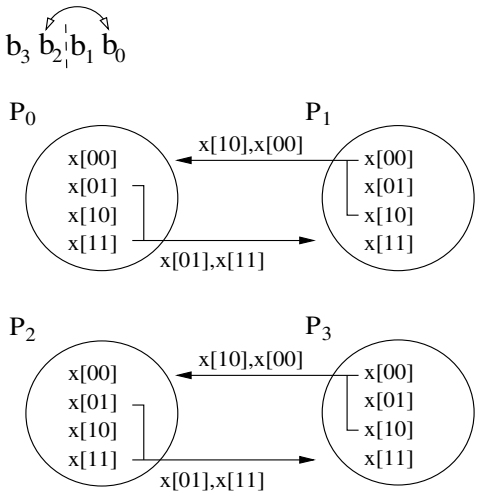

Figure 2. Dataflow on mixed global/local bit permutation

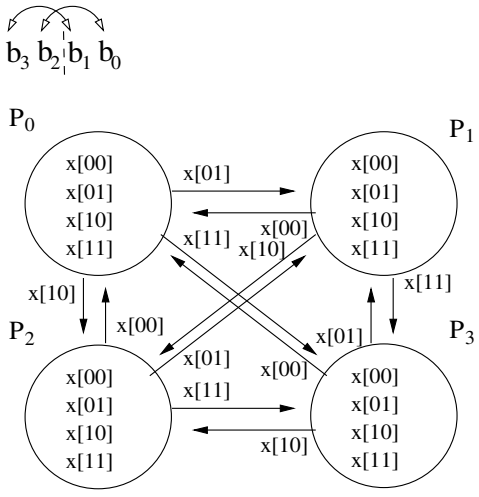

Figure 3. Dataflow on all-to-all bit permutation

permutation on the P processes since all of the data from each process is permuted as a block. Such a permutation is called a global permutation. If in fact the pid bits remain fixed then there is no communication between processes, and the permutation is purely local.

A global bit permutation, B_σ can be implemented using MPI_SendRecv. Each process computes the addresses it will send data to and receive data from. The destination address is equal to $\sigma(pid)$ and the source address is equal to $\sigma^{-1}(pid)$. A general bit permutation with k bits crossing the pid boundary can be implemented as a sequence of k global permutations each exchanging $N/(P2^k)$ data elements. In this case, the communication schedule, i.e. sequence of permutations, must be calculated using σ and the number processes involved in the communication. Once the schedule is determined each process loops through a sequence of global permutations using the schedule. For each permutation appropriate data, determined by σ, must be gathered from the input vector into a buffer before sending and scattered from the receiving buffer to the output vector. The gather and scatter operations may require local permutations. In some cases MPI types can be constructed to describe the patterns.

Two examples will be given to illustrate how a schedule can be calculated from the action of σ on the binary representations of the indices. Let $N = 2^7$ and assume $P = 2^3$. The first example is the stride permutation L_4^{128} and the second example is the permutation $B_{(0,6)(1,5)}$. In both examples, the local data size is 16 and each process send 4 elements to four different processes.

The stride permutation rotates the bits two places to the right.

$$b_6b_5b_4|b_3b_2b_1b_0 \rightarrow b_1b_0b_6|b_5b_4b_3b_2 \qquad (5)$$

For a fixed process, the low order 4 bits vary over the indices of the 16 local data elements. Thus the process with pid $b_6b_5b_4$ sends data to the four processes with pid equal to $**b_6$ where $* \in \{0, 1\}$.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Table 2. Communication schedules

Schedule for L_4^{128}							
0	1	2	3	4	5	6	7
0	2	4	6	1	3	5	7
2	0	6	4	3	1	7	5
4	2	0	6	5	7	1	3
6	4	2	0	7	5	3	1
Schedule for $B_{(0,6)(1,5)}$							
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
2	3	0	1	6	7	4	5
4	5	6	7	0	1	2	3
6	7	4	5	2	3	0	1

This lead to the following process map.

$$\begin{array}{rcl} pid & \rightarrow & \text{processor set} \\ 000 = 0 & \rightarrow & \{0,2,4,6\} \\ 001 = 1 & \rightarrow & \{0,2,4,6\} \\ 010 = 2 & \rightarrow & \{0,2,4,6\} \\ 011 = 3 & \rightarrow & \{0,2,4,6\} \\ 100 = 4 & \rightarrow & \{1,3,5,7\} \\ 101 = 5 & \rightarrow & \{1,3,5,7\} \\ 110 = 6 & \rightarrow & \{1,3,5,7\} \\ 111 = 7 & \rightarrow & \{1,3,5,7\} \end{array}$$

A schedule can be obtained by cyclically rotating the receiving processors. This is shown in Table 2.

The data to be sent from a given process to a specified process can be determined from the permuted bit sequence. In this case, process $b_6b_5b_4$ sends data with indices $b_3b_2b_1b_0$ to process $b_1b_0b_4$. The data is stored in location $b_3b_2b_6b_5$. For example process 0 sends x_3, x_7, x_{11}, x_{15} to process 6, which stores the four elements in locations $0, 4, 8, 12$.

A similar analysis can be done for the second example.

$$b_6b_5b_4|b_3b_2b_1b_0 \rightarrow b_1b_0b_4|b_3b_2b_6b_5, \qquad (6)$$

which leads to the following process map.

$$\begin{array}{rcl} pid & \rightarrow & \text{processor set} \\ 000 = 0 & \rightarrow & \{0,2,4,6\} \\ 001 = 1 & \rightarrow & \{1,3,5,7\} \\ 010 = 2 & \rightarrow & \{0,2,4,6\} \\ 011 = 3 & \rightarrow & \{1,3,5,7\} \\ 100 = 4 & \rightarrow & \{0,2,4,6\} \\ 101 = 5 & \rightarrow & \{1,3,5,7\} \\ 110 = 6 & \rightarrow & \{0,2,4,6\} \\ 111 = 7 & \rightarrow & \{1,3,5,7\} \end{array}$$

In this example, the communication pattern decomposes into two subsets each requiring an "all-to-all" communication. This allows a schedule to be created which requires only point-to-point communication (i.e. each permutation in the sequence is of order 2). The schedule is shown in Table 2.

5 Performance of Distributed Memory Permutations

Several families of bit permutations were implemented for use in the distributed memory WHT package. Preliminary experiments were performed on a cluster of 32 Pentium III processors (450 MHz), each equipped 512 MB of 8ns PCI-100 memory and 2 SMC 100mbps fast-Ethernet cards. The slow interconnect speed leads to certain choices and conclusions that we do not expect to hold with faster communication.

The two bit-permutations implemented were stride permutations and a multi-bit swap. As indicated in Section 3 stride permutations provide sufficient flexibility to implement an arbitrary distributed WHT split. However, stride permutations always involve some global communication, do

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

not generally allow a point to point communication scheme (see Section 4), and have very few fixed points, and data access patterns with potentially poor strides, which may lead to greater network traffic and higher cost for local permutations. As an alternative multi-bit swap permutations were implemented. A multi-bit swap permutation swaps two contiguous regions of k bits. Compared to a stride permutation with stride 2^k, a multi-swap permutation with swap region of k bits has many more fixed points and potentially less network traffic. Moreover, since it is of order two, it always allows point to point communication to be used for global communication (see the last example in Section 4). In the extreme case when only two bits are swapped, half of the data is fixed, and either the communication is purely local or each process exchanges half of its data with exactly one other process.

When implementing the swap permutation there are three cases two consider depending on whether both swap regions are in the local index bits, one region is in the local index bits and one region is in the pid bits, or one swap region overlaps the pid/offset boundary. Similarly the implementation of a stride permutation requires three cases: 1) when the stride 2^k is less than or equal to the number of processors 2^p, 2) when $N/2^k$ is less than the number of processors, and 3) when $2^p < 2^k < N/2^p$.

For both permutations, when the number of bits crossing from the local index region to the pid region is equal to p, then the communication is all-to-all. When this is the case the all-to-all communication pattern can be implemented with a sequence of order two permutations requiring only point-to-point communication (see Section 4). More generally, point to point communication can be used whenever the communication pattern can be decomposed into disjoint sets of all-to-all communication. This is always the case for multi-swap permutations, but as was shown in Section 4 this can not be done for for cases 1) and 3) for the stride permutation.

An all-to-all communication pattern can be implemented with a sequence of permutations given by the rows of a latin square. A latin square is an $n \times n$ array containing n different numbers with each number occuring exactly once in each row and column. The (i, j) element of the array indicates the process that process j will send its data to in the i-stage. Since the elements in each row are distinct, the i-th communication stage is a permutation, and since the elements in each column are distinct the sequence of stages provide and all-to-all communication. If the latin square has the additional property that the (i, j) element is k if and only if (i, k) element is j, then each permutation is of order two and the communication stages are made up of pairwise exchanges. Such a latin square is called a latin square of order two.

A latin square of size n can be constructed using the multiplication table of a group of size n. The cyclic communication pattern is based on the group table of the cyclic group of order n. A latin square of size $N = 2^n$ of order two can be constructed from the group table of the group equal to the n-fold direct product of cyclic groups of order two. The following example shows an order two latin square of size eight constructed this way. The recursive pattern makes it clear how to construct an arbitrary latin square of size 2^n with this property.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 \\ 2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 \\ 3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 \\ 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\ 5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 \\ 6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 \\ 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}$$

Table 3 compares three methods for implementing all-to-all communication on the cluster using MPI. The first method uses the point-to-point scheme implemented with MPI_Sendrecv, the second, called "three-way talking" uses a cyclic scheduling algorithm implemented with

Table 3. Performance comparison of three all-to-all communication schemes

$log(localN)$	Point-to-point Time(sec)	Three-way talking Time(sec)	MPI_Alltoall Time(sec)
25	45.39	63.17	140.50
24	22.67	29.99	78.70
23	11.33	15.02	44.23
22	5.76	7.72	20.89
21	2.93	3.85	8.44
20	1.62	2.12	5.15
19	0.68	1.12	2.64

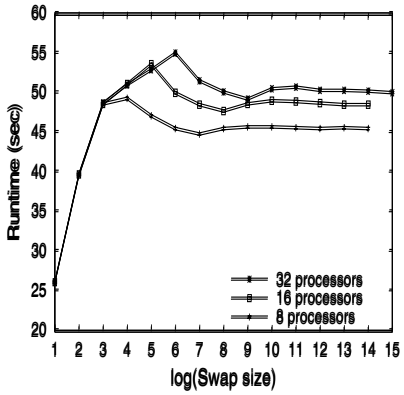

Figure 4. Performance of multi-swap permutations

MPI_Sendrecv, where at each stage a process receives data from one process and sends data to another process, and the third uses the MPI_Alltoall function. In our experimental environment both methods based on MPI_Sendrecv were substantially faster than using MPI_Alltoall, and the point-to-point scheme was 20% to 30% faster than using three way communication.

Figures 4 and 5 show the performance of multi-swap and stride permutations as a function of the number of bits in the swap region and the log of the stride. In both cases, the size of the local data region was 2^{25} and the number of processors was 32. The multi-swap permutations always swap the leftmost region with the rightmost region.

Figure 4 shows the performance of the multi-swap initially decreasing, peaking and then reaching a local minimum as the size of the swap region increases. As the number of bits increases to 5 the amount of data communicated over the network doubles each time. After reaching swap size 5, local shuffling of the data is required (in this case a matrix transpose). Moreover, the number of fixed points decreases with the size of the swap region. This suggests a continued drop off in performance; however, the performance increase after the swap region reaches 5 bits is due to increased block size on the transpose which is optimized with blocks of size 16. Increased cache misses due to large strides causes the decrease in performance after block size 16 is reached. Despite the local variation performance on this cluster is dominated by the cost of network communication.

The absolute performance of the stride permutation, shown in Figure 5, is similar to that of the multi-swap permutation. This is the case since network communication is the dominant cost. However, for small stride, the cost is significantly worse than for similar multi-swap permutations. This is the case since point-to-point communication can not be used until the stride becomes equal to the number of processors. A minimum in runtime is reached at stride 2^4 and 2^{25-4} (the symmetry in the plot is due to the duality of the stride permutation - loading a vector of size 2^n at stride 2^k is similar to storing at stride 2^k which corresponds to loading a vector at stride 2^{n-k}). The minimum is due to local cache performance. Finally, we remark that the best performance between stride 2^5 and 2^{20}, i.e. the region where all-to-all communication with the 32 processors is required, is slightly better than the corresponding multi-swap permutation.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

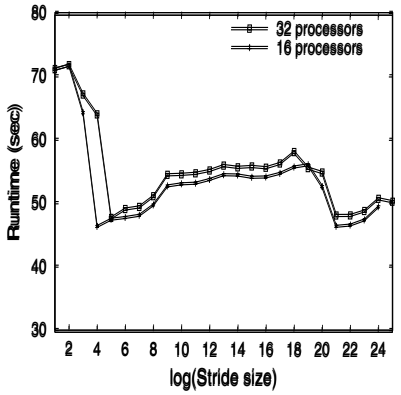

Figure 5. Performance of stride permutations

6 Performance of Distributed WHT Algorithms

The WHT package (available at [9]) of [6] has been extended to support the distributed memory parallel algorithms discussed in this paper. In the sequential package an algorithm based on the split in Equation 1 is denoted by `split[W(n1),...,W(nt)]`, where `W(ni)` is further expanded depending further applications of Equation 1. Ultimately, at the base case, small transforms, WHT_{2^m}, are implemented with straight-line code which is denoted by `small[m]`. The optimal recursive applications of Equation 1 are determined using dynamic programming (note this only finds an approximation of the optimal algorithm since the optimal subalgorithm can depend on the context - e.g. state of the cache - in which the algorithm is called). The distributed memory package introduces the notation `dsplit[W(n1),...,W(nt)]` for the algorithm in Equation 3. An alternative `dsplit` based on multi-swap permutations is also provided. However, this currently is only applicable for binary splits and has the further restriction that the left node size must be greater than or equal to the right node size.

In the distributed memory package, dyanamic programming is used to find the best combination of distributed permutations and sequential WHT algorithms. Due to the high network costs in the experimental platform used (communication costs dominate computation costs), only binary splits were considered. In this case, the algorithms searched were

$$WHT_{2^n} = L^{2^n}_{2^{n_1}}(I_{2^{n-n_1}} \otimes W_{2^{n_1}}) \quad (7)$$
$$L^{2^n}_{2^{n-n_1}}(I_{2^{n_1}} \otimes W_{2^{n-n_1}})$$
$$= M^{2^n}_{2^{n_1}}(I_{2^{n-n_1}} \otimes W_{2^{n_1}}) \quad (8)$$
$$M^{2^n}_{2^{n_1}}(I_{2^{n_1}} \otimes W_{2^{n-n_1}}),$$

where $M^{2^n}_{2^{n_1}}$ is the multi-swap permutation that exchanges the low order and high order n_1 bits.

The optimal values for stride or swap region found in Section 5 may not lead to the optimal algorithm since better sequential code may be found for different sizes. Figure 6 shows the different splits using `dsplit` with multi-swap and stride permutations. The total data size was 2^{30}. The permutation runtime dominates the overall performance in this distributed system. However, the runtime of the two sequential transforms still must be taken into account. For example, the local minimum for multi-swap permutations is at swap size 2^9, but the overall runtime minimum is at size 2^8. The sequential runtime data of the WHT transform indicate that the combination of size $(2^8, 2^{22})$ has half the runtime of the combination of size $(2^9, 2^{21})$. The same effect can be seen when comparing the runtime of `dsplit` using stride permutations in Figure 6 with the runtime of just the stride permutation in Figure 5. The optimal algorithm of data size 2^{30} that was found uses a stride permutation with a split of $(2^7, 2^{23})$. The resulting algorithms show nearly linear scalability. A size 2^{28} transform on 16 processors was 1.87 times faster than with 8 processors and 3.69 times faster with 32 processors. The amount of memory required was to much to run the transform on fewer processors. Scaling the sequential transform time would not show speedup due to the large communication cost required by the slow network speed.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

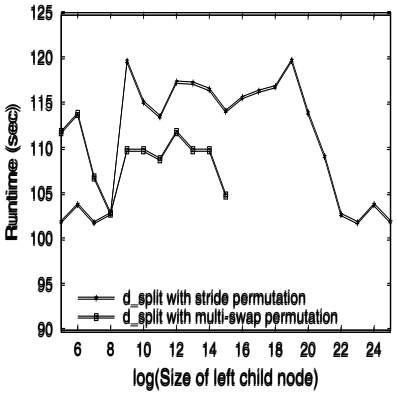

Figure 6. The runtimes of WHT d_split at different binary split

7 Conclusion

This paper describes a distributed memory package for implementing and optimizing algorithms for computing the Walsh Hadamard transform (WHT). The package uses dynamic programming to find the best combination of sequential algorithms and communication patterns. Experimental results were shown for a cluster of 32 Pentium III's connected with 100mbps fast Ethernet. The particular algorithms chosen for this platform are likely very different from those that would be chosen on a platform with a faster interconnection network. Additional experimental results need to be obtained. In particular algorithms with more than two passes (binary split) may lead to faster implementations. However, the ability of the package to automatically consider alternative algorithms would enable these to be found. For these scenerios, a multi-pass algorithm based on swap permutations has been discovered and will be incorporated into the package. Furthermore, notation should be incorporated to make it easy to include alternative permutations so that the search may include more than a fixed set of permutation choices. These issues will be explored as the results of this paper are incorporated into the SPIRAL [9] system for automatically implementing fast signal transforms.

References

[1] K. Beauchamp. *Applications of Walsh and related functions*. Academic Press, 1984.

[2] K. Chen and J. R. Johnson. A prototypical self-optimizing package for parallel implementation of fast signal transforms. *IPDPS*, 2002.

[3] E. Chu and A. George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, 2000.

[4] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP '98*, volume 3, pages 1381–1384, 1998. `http://www.fftw.org`.

[5] K.-S. Gatlin and L. Carter. Faster FFTs via architecture-cognizance. In *Proceedings of PACT 2000*, Oct. 2000.

[6] J. Johnson and M. Püschel. In search of the optimal Walsh-Hadamard transform. *ICASSP*, 2000.

[7] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9(4):449–500, 1990.

[8] F. MacWilliams and N. Sloane. *The theory of error-correcting codes*. North-Holland Publ.Comp., 1992.

[9] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso. SPIRAL: Portable Library of Optimized Signal Processing Algorithms, 1998. http://www.ece.cmu.edu/~spiral.

[10] N. Park and V. K. Prasanna. Cache conscious Walsh-Hadamard transform. *ICASSP*, 2001.

[11] M. C. Pease. An adaptation of the fast fourier transform for parallel processing. *ACM*, 15(2):252–264, April 1968.

[12] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. Siam, 1992.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE