Hybrid MPI-OpenMP versus MPI Implementations: A Case Study

Mr. Osvaldo Mangual, Honeywell Aerospace

Osvaldo Mangual has a Bachelor degree in Computer Engineering and a Master degree in Electrical Engineering from Polytechnic University of Puerto Rico in the area of Digital Signal Processing. He currently works at Honeywell Aerospace, PR, as a FPGA and ASIC Designer.

Dr. Marvi Teixeira, Polytechnic University of Puerto Rico

Dr. Teixeira is a Professor at the Electrical and Computer Engineering and Computer Science Department at Polytechnic University of Puerto Rico He holds a Ph.D. and MSEE degrees from University of Puerto Rico at Mayaguez and a BSEE degree from Polytechnic University. Professor Teixeira is an IEEE Senior Member, a Registered Professional Engineer and a former ASEE-Navy Summer Faculty Fellow.

Mr. Reynaldo Lopez-Roig, Polytechnic University of Puerto Rico

Mr. Lopez received his B.S. in Computer Engineering from the Polytechnic University of Puerto Rico in 2013. His work as an undergraduate research assistant was related to the implementation and benchmarking of parallel signal processing algorithms in clusters and multicore architectures. Mr. Lopez is currently working at the Jet Propulsion Laboratory as a Software Systems Engineer.

Prof. Felix Javier Nevarez-Ayala, Polytechnic University of Puerto Rico

Felix Nevarez is an Associate Professor in the Electrical and Computer Engineering Department at the Polytechnic University of Puerto Rico, He earned a BS in Physics and a MS in Electrical Engineering from the University of Puerto Rico at Mayaguez and a BS in Electrical Engineering from the Polytechnic University of Puerto Rico. HIs research interests are in numerical simulations and parallel and distributed programming.

©American Society for Engineering Education, 2014

Hybrid MPI-OpenMP versus MPI Implementations: A Case Study

Osvaldo Mangual[+], Marvi Teixeira[*], Reynaldo Lopez[#], Felix Nevarez[*]
+ Honeywell Aerospace, Aguadilla, PR., *Polytechnic University of Puerto Rico, #Jet Propulsion Laboratory, Pasadena, CA.

Abstract

In this paper we explore the performance of a hybrid, or mixed-mode (MPI-OpenMP), parallel C++ implementation versus a direct MPI implementation. This case-study provides sufficient amount of detail so it can be used as a point of departure for further research or as an educational resource for additional code development regarding the study of mixed-mode versus direct MPI implementations. The hardware test-bed was a 64-processor cluster featuring 16 multi-core nodes with four cores per node. The algorithm being benchmarked is a parallel cyclic convolution algorithm with no inter-node communication that tightly matches our particular cluster architecture. In this particular case-study a time-domain-based cyclic convolution algorithm was used in each parallel subsection. Time domain-based implementations are slower than frequency domain-based implementations, but give the exact integer result when performing very large, purely integer, cyclic convolution. This is important in certain fields where the round-off errors introduced by the FFTs are not acceptable. A scalability study was carried out where we varied the signal length for two different sets of parallel cores. By using MPI for distributing the data to the nodes and then using OpenMP to distribute data among the cores inside each node, we can match the architecture of our algorithm to the architecture of the cluster. Each core processes an identical program with different data using a single program multiple data (SPMD) approach. All pre and post-processing tasks were performed at the master node. We found that the MPI implementation had a slightly better performance than the hybrid, MPI-OpenMP implementation. We established that the speedup increases very slowly, as the signal size increases, in favor of the MPI-only approach. Even though the mixed-mode approach matches the target architecture there is an advantage for the MPI approach. This is consistent with what is reported in the literature since there are no unbalancing problems, or other issues, in the MPI portion of the algorithm.

Introduction

In certain fields, where the round-off errors introduced by the FFTs are not acceptable, time domain-based implementations of cyclic convolution guarantee the exact integer result when performing very large, purely integer, cyclic convolution. The trade-off is that these implementations are slower than frequency domain-based implementations. Applications that can benefit from this approach include multiplication of large integers, computational number theory, computer algebra and others[1,2]. The proposed, time domain-based, parallel implementation can be considered as complementary to other techniques, such as Nussbaumer convolution[3] and Number Theoretic Transforms[4], which can also guarantee the exact integer result but could have different length-restrictions on the sequences.

Parallel implementations in cluster architectures of time domain-based, purely integer, cyclic convolution of large sequences resulted much faster than the direct, time domain-based, $O(N^2)$ implementation in a single processor. This is not the case for frequency domain-based implementations where the direct implementation in a single processor is usually faster than the

parallel formulation, and therefore preferably, unless memory limitations or round-off errors become an issue as it happens with the mentioned applications.
The algorithm being benchmarked is a parallel cyclic convolution algorithm with no interprocessor communication. We selected this algorithm because it strongly matches our particular cluster architecture and, at the same time, is amenable to a mixed-mode (MPI-OpenMP) implementation as well as to a direct MPI implementation. In the past, different variants for this algorithm were developed[5,6] and its use within different hardware implementations was proposed[7,8,9]. We have found no studies regarding the implementation of this algorithm in cluster architectures. While further benchmarks and scalability studies will be reported elsewhere, in this paper we are focusing in a MPI-only versus a mixed-mode (MPI-OpenMP) parallel implementation, including performance and scalability studies, carried out in our 16-node, 64 processor cluster.

Based on the prime factor decomposition of the signal length this algorithm, which is based on a block diagonal factorization of the circulant matrices, breaks a one-dimensional cyclic convolution into shorter cyclic sub-convolutions. The subsections can be processed, independently, either in serial or parallel mode. The only requirement is that the signal length, N, admits at least an integer, r_0, as a factor; $N = r_0.s$. The Argawal-Cooley Cyclic Convolution algorithm, has a similar capability but requires that the signal length can be factored into mutually prime factors; $N = r_0.s$ with $(r_0,s) = 1$. Since the block pseudocirculant algorithm is not restricted by the mutually prime constraint, it can be implemented recursively using the same factor[6,7]. The parallel technique, compounded with a serial recursive approach in each parallel subsection, provides a sublinear increase in performance versus the serial-recursive implementation in a single core.

For our scalability studies we first used 16 cores at four cores per node. We accessed the 16 cores directly using MPI and then, for the hybrid approach, we accessed four nodes using MPI followed by using OpenMP to access the four cores in each node. We repeated the computation for several signal lengths. We then used 64 cores. We accessed the 64 cores directly using MPI and then, for the hybrid approach, we accessed 16 nodes using MPI followed by using OpenMP to access the four cores in each node. Again, several signal lengths were used. At each parallel core the algorithm was run in a serial-recursive mode until the recursion became more expensive than directly computing the sub-convolution for the attained sub-length. The details of the serial-recursive implementation will be reported elsewhere.

We start by providing, as stated in the literature[6], the mathematical framework for the algorithm using a tensor product formulation. We then develop a block matrix factorization that includes pre-processing, post-processing and parallel stages. The parallel stage is defined through a block diagonal matrix factorization[6]. The algorithm block diagram is clearly developed through a one-to-one mapping of each block to the algorithm block matrix formulation. We follow by stating the benchmark setup, benchmark results and conclusions.

Algorithm Description

Here we briefly describe the sectioned algorithm used for this benchmark as reported in the literature[5,6]. In this particular implementation, the sub-convolutions are performed using a recursive time domain-based cyclic convolution algorithm in order to avoid round-off errors. The

proposed algorithmic implementation does not require communication among cores but involves initial and final data distribution at the pre-processing and post-processing stages.
Cyclic convolution is a established technique broadly used in signal processing applications. The Discrete Cosine Transform and the Discrete Fourier Transform, for example, can be formulated and implemented as cyclic convolutions[6,7]. In particular, parallel processing of cyclic convolution has potential advantages in terms of speed and/or access to extended memory, but requires breaking the original cyclic convolution into independent sub-convolution sections. The Agarwal-Cooley cyclic convolution algorithm is suitable to this task but requires that the convolution length be factored into mutually prime factors, thus imposing a tight constraint on its application[3]. There are also multidimensional methods but they may require the lengths along some dimensions to be doubled[3]. There other recursive methods, which are also constrained in terms of signal length[3]. When the mentioned constraints are not practical, this algorithm provides a complementary alternative since it only requires that the convolution length be factorable[5,6]. This technique, depending on the prime factor decomposition of the signal length, can be combined with the Agarwal-Cooley algorithm or with other techniques.

By conjugating the circulant matrix with stride permutations a block pseudocirculant formulation is obtained. Each circular sub-block can be independently processed as a cyclic sub-convolution. Recursion can be applied, in either a parallel, serial or combined fashion, by using the same technique in each cyclic sub-convolution. The final sub-convolutions at the bottom of the recursion can be performed using any method. The basic formulation of the algorithm as stated in the literature is as follows[6],

The modulo-N cyclic convolution of the length-N sequences x[n] and h[n] is defined by,

$$y[n] = \sum_{k=0}^{N-1} x[k]h[((n - k))_N] \quad (1)$$

which can be formulated in matrix form as,

$$y = H_N x \quad (2)$$

where H_N is the circulant matrix,

$$H_N = \begin{bmatrix} h_0 & h_{n-1} & h_{n-2} & . & h_2 & h_1 \\ h_1 & h_0 & h_{n-1} & . & h_3 & h_2 \\ h_2 & h_1 & h_0 & . & h_4 & h_3 \\ . & . & . & . & . & . \\ h_{n-2} & h_{n-3} & h_{n-4} & . & h_0 & h_{n-1} \\ h_{n-1} & h_{n-2} & h_{n-3} & . & h_1 & h_0 \end{bmatrix} \quad (3)$$

Given $N = r_0 s$, and conjugating the circulant matrix, H_N, with stride-by-r_0 permutations we obtain,

$$y = H_N x \tag{4}$$

$$P_{N,r_0} y = P_{N,r_0} H_N P_{N,r_0}^{-1} P_{N,r_0} x \tag{5}$$

$$H_{pr_0} = P_{N,r_0} H_N P_{N,r_0}^{-1} \tag{6}$$

The decimated-by-r_0 input and output sequences are written as,

$$y_{r_0} = P_{N,r_0} y, \qquad x_{r_0} = P_{N,r_0} x \tag{7}$$

The conjugated circulant matrix has the form of a *Block Pseudocirculant Matrix*[6], represented as H_{pr0}. *Block Pseudocirculant Matrices* have the circulant sub-blocks above the diagonal multiplied by a cyclic shift operator. Equation (5) can be re-written using (6) and (7) as,

$$y_{r_0} = H_{pr_0} x_{r_0} \tag{8}$$

The Block Pseudocirculant in (8) is written as[5,6],

$$y_{r_0} = \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ . \\ Y_{r_0-2} \\ Y_{r_0-1} \end{bmatrix} = \begin{bmatrix} H_0 & S_{N/r_0} H_{r-1} & S_{N/r_0} H_{r-2} & . & S_{N/r_0} H_2 & S_{N/r_0} H_1 \\ H_1 & H_0 & S_{N/r_0} H_{r-1} & . & S_{N/r_0} H_3 & S_{N/r_0} H_2 \\ H_2 & H_1 & H_0 & . & S_{N/r_0} H_4 & S_{N/r_0} H_3 \\ . & . & . & . & . & . \\ H_{r_0-2} & H_{r_0-3} & H_{r_0-4} & . & H_0 & S_{N/r_0} H_{r_0-1} \\ H_{r_0-1} & H_{r_0-2} & H_{r_0-3} & . & H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ . \\ X_{r_0-2} \\ X_{r_0-1} \end{bmatrix} \tag{9}$$

$S_{N/r0}$ is the *Cyclic Shift Operator Matrix* defined by,

$$S_{N/r_0} = \begin{bmatrix} 0 & 0 & 0 & . & 0 & 1 \\ 1 & 0 & 0 & . & 0 & 0 \\ 0 & 1 & 0 & . & 0 & 0 \\ . & . & . & . & . & . \\ 0 & 0 & 0 & . & 0 & 0 \\ 0 & 0 & 0 & . & 1 & 0 \end{bmatrix} \tag{10}$$

All sub-convolutions, represented by the sub-blocks in the block pseudcirculant matrix, can be processed in parallel. The cyclic shifts above the diagonal represent a circular redistribution of each sub-convolution result.

Example: For $N = 4$, $r_0 = 2$, $H_N = H_4$, (5) becomes,

$$P_{4,2}y = P_{4,2}\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = P_{4,2}\begin{bmatrix} h_0 & h_3 & h_2 & h_1 \\ h_1 & h_0 & h_3 & h_2 \\ h_2 & h_1 & h_0 & h_3 \\ h_3 & h_2 & h_1 & h_0 \end{bmatrix} P_{4,2}^{-1} = P_{4,2}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{11}$$

Where $P_{4,2}$ is a stride-by-2 permutation matrix,

$$P_{4,2} = P_{4,2}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying (6) and (7), (11) becomes,

$$y_{r_0} = y_2 = \begin{bmatrix} \begin{bmatrix} y_0 \\ y_2 \end{bmatrix} \\ \begin{bmatrix} y_1 \\ y_3 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} h_0 & h_2 \\ h_2 & h_0 \end{bmatrix} & \begin{bmatrix} h_3 & h_1 \\ h_1 & h_3 \end{bmatrix} \\ \begin{bmatrix} h_1 & h_3 \\ h_3 & h_1 \end{bmatrix} & \begin{bmatrix} h_0 & h_2 \\ h_2 & h_0 \end{bmatrix} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} \end{bmatrix} \tag{12}$$

where the circulant matrix H_4 in (11) has become blocked in a pseudocirculant fashion and can be written as,

$$y_{r_0} = y_2 = \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} H_0 & S_2H_1 \\ H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \tag{13}$$

where S_2 is the cyclic shift operator matrix,

$$S_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and the blocked cyclic sub-convolutions are defined as,

$$H_0 = \begin{bmatrix} h_0 & h_2 \\ h_2 & h_0 \end{bmatrix}, \; H_1 = \begin{bmatrix} h_1 & h_3 \\ h_3 & h_1 \end{bmatrix} \tag{14}$$

Further examples can be found in the literature[6,7]. It is clear that each sub-convolution can be separately processed, followed by a reconstruction stage to provide the final result. Parallel algorithms can be developed by factorization of the Block Pseudocirculant matrix shown in (9), (12) and (13) into diagonal blocks. The general tensor product formulation for a block diagonal factorization of the Block Pseudocirculant Matrix is[6],

$$y_{r_0} = R_{r_0}(A_{r_0} \otimes I_{N/r_0}) D_{H_{r_0}} (B_{r_0} \otimes I_{N/r_0}) x_{r_0} \tag{15}$$

where x_{r0} and y_{r0} are the decimated-by-r_0 input and output sequences and A_{r0} and B_{r0} are the post/pre-processing matrices, which are determined by each particular factorization. Matrix R_{r0} take account of all the cyclic shift operators and D_{Hr0} is the diagonal matrix with blocked subsections that are suitable to be implemented in parallel. At each parallel sub-section the same algorithm can be applied again. The general form of these matrices for different variants of this algorithm can also be found in the literature[6]. We are using the simplest approach, which is the *Fundamental Diagonal Factorization*[6] that implies r_0^2 independent parallel sub-convolutions that uses straight forward pre-processing and post-processing stages. This formulation is the best match to our target architecture and it was used to benchmark the mixed-mode, MPI-OpenMP, implementation versus the MPI implementation.

For an N-point cyclic convolution, where $N = r_0.s$, a radix-2 ($r_0 = 2$) fundamental diagonalization gives $r_0^2 = 4$ diagonal subsections. Equations (5) and (15) with $r_0 = 2$ can be written as,

$$y_2 = \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} H_0 & S_{N/2} H_1 \\ H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \tag{16}$$

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \underbrace{\begin{bmatrix} I_{N/2} & 0 & S_{N/2} \\ 0 & I_{N/2} & 0 \end{bmatrix}}_{R} \cdot \left[\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{A} \otimes I_{N/2} \right] \cdot \underbrace{\begin{bmatrix} H_0 & 0 & 0 & 0 \\ 0 & H_1 & 0 & 0 \\ 0 & 0 & H_1 & 0 \\ 0 & 0 & 0 & H_0 \end{bmatrix}}_{D} \cdot \left[\underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}}_{B} \otimes I_{N/2} \right] \cdot \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \tag{17}$$

The block diagram realization of the algorithm embodied in (17) is shown in Fig. 1. The parallel realization is given by the diagonal matrix; D, while the pre-processing distribution is given by matrix B and the post-processing cyclic shifts and sums are given by matrices R and A. Note that, following (5), (6) and (7), X_0 and X_1 are the decimated subsections yielded by the stride permutation applied to the input sequence. We also need to apply the inverse stride permutation to the output vector, formed by Y_0 and Y_1 in order to reorder the output and give the final result. In the present implementation the parallel cyclic sub-convolutions are performed at the cores/nodes and the pre-processing and post-processing stages are computed at the master node.

The implementation in each core is time domain-based and follows a serial recursive approach, to be discussed elsewhere using a variant of this algorithm termed *Basic Diagonal Factorization (BDF)* [6].

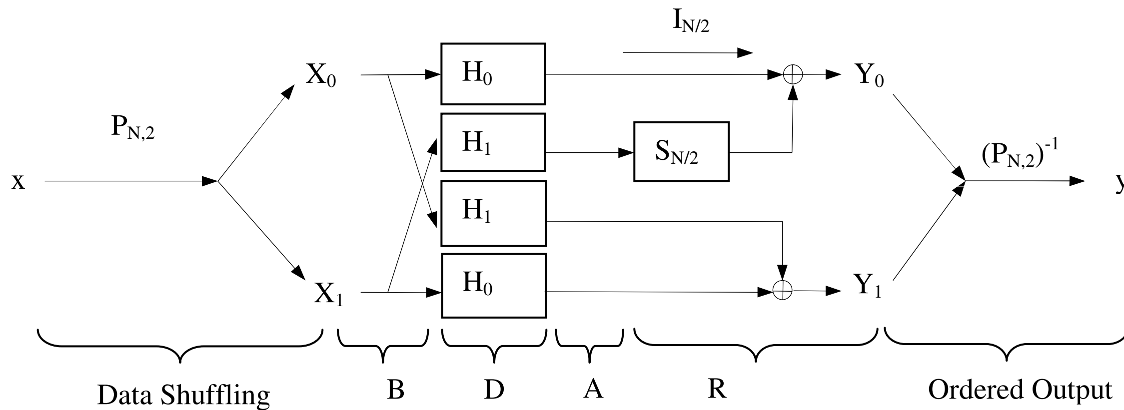

Figure 1. Parallel cyclic convolution realization based on a *Fundamental Diagonal Factorization* ($r_0 = 2$) of the Block Pseudocirculant Matrix. Flow graph stages B, D, A, R directly map to same name matrices in equations (15). (17).

Hybrid, MPI – OpenMP, versus a Direct MPI Implementation

MPI is capable of transmitting the data to the nodes but it is not necessarily efficient distributing the data to the cores inside the multi-core nodes. OpenMP, on the other hand, is optimized for shared memory architectures. In theory, by using MPI for distributing the data to the nodes, and then using OpenMP for distributing the data inside the multi-core nodes, we can match the architecture of our algorithm to the architecture of the cluster.

Usually, there is a hierarchical model in mixed-mode implementations: MPI parallelization occurs at the top level and OpenMP parallelization occurs at the low level, as it is shown in Figure 2. This model[10] matches the architecture of our cluster. For example, codes that do not scale well as the MPI processes increase, but scale well as OpenMP processes increase, may take advantage of using mixed-mode programming. Nonetheless, previous studies show that the mixed-mode approach is not necessarily the most successful technique on SMP systems and should not be considered as the standard model for all codes[10].

If the parallel subsections shown in Figure 1 are cores, the cyclic sub-convolutions are directly computed at each core. If the parallel subsections are multi-core nodes, the algorithm can be applied again in a parallel-recursive fashion at each node to redistribute data among the cores within the node. The cyclic sub-convolutions are again computed directly at each core. The former scheme lends itself to a direct MPI implementation while the latter scheme is amenable to a hybrid, MPI-OpenMP implementation as shown in Figure 2. Note that, in all cases, the final sub-convolutions at the cores can be performed through a serial-recursive approach using the *BDF* variant[6] of this algorithm. The serial-recursive process stops at some recursive step when continuing the recursion becomes more expensive than directly computing the sub-convolution. The techniques used to tackle the serial-recursive approach will be reported elsewhere.

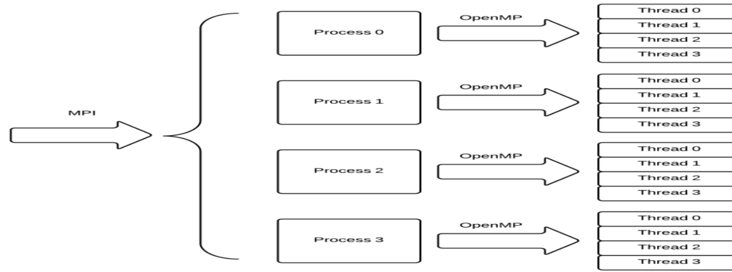

Figure 2. Example of a Data Distribution Model for a Mixed-Mode Implementation[10]

16-Processor Direct MPI Implementation

For the direct MPI implementation we partitioned the cyclic convolution into 16 parallel subsections of one-fourth the original convolution length using a radix-4 approach ($r_0 = 4$). This implementation was tackled using MPI where the data was distributed from the master node to 16 parallel cores. In each core a serial approach was used by recursive application of the same algorithm until the final cyclic sub-convolutions were performed using a straightforward, time domain-based, convolution algorithm. We repeated the overall process for eight different signal lengths (2^{15} to 2^{22}). The block diagram realization is shown in Figure 3.

16-Processor Hybrid, MPI-OpenMP Implementation

We then implemented a hybrid, MPI-OpenMP, approach. Using this technique, data of one-half the original length was distributed from the master node to four parallel nodes using MPI following the architecture in Figure 1 (Radix-2, $r_0 = 2$). Then, under OpenMP, the structure in Figure 1 was applied again (Radix-2, $r_0 = 2$) in a parallel-recursive fashion at each of the four nodes. The final processing length is one-fourth of the original signal length and both methods use 16 processors. In each core, as in the previous case, a serial-recursive approach is used to compute the final sub-convolutions. We repeated the overall process for eight different signal lengths (2^{15} to 2^{22}). The block diagram realization is shown Figure 4.

64-Processors MPI versus Hybrid, MPI-OpenMP, Implementations

The procedure described in the past two paragraphs was repeated for 64 processors using MPI for the direct implementation (radix-8, $r_0 = 8$). The hybrid implementation used MPI to distribute data among 16 multi-core nodes (radix-4, $r_0 = 4$) followed by OpenMP to distribute data among the four processors within each node (radix-2, $r_0 = 2$). In both approaches the data length, after parallel distribution, was one-eighth of the original signal length. As with the 16-processor implementation the parallel sub-convolutions in each core were computed using a serial-recursive approach. The block diagram for the direct implementation is similar to the one shown in Figure 3, but now distributing data to 64 cores using MPI. The block diagram for the hybrid implementation is similar to the one shown in Figure 4, but now we are distributing data to 16 multi-core nodes using MPI followed by further data distribution to the four cores within each node using OpenMP. We repeated both processes for 10 different signal-lengths (2^{15} to 2^{24}).

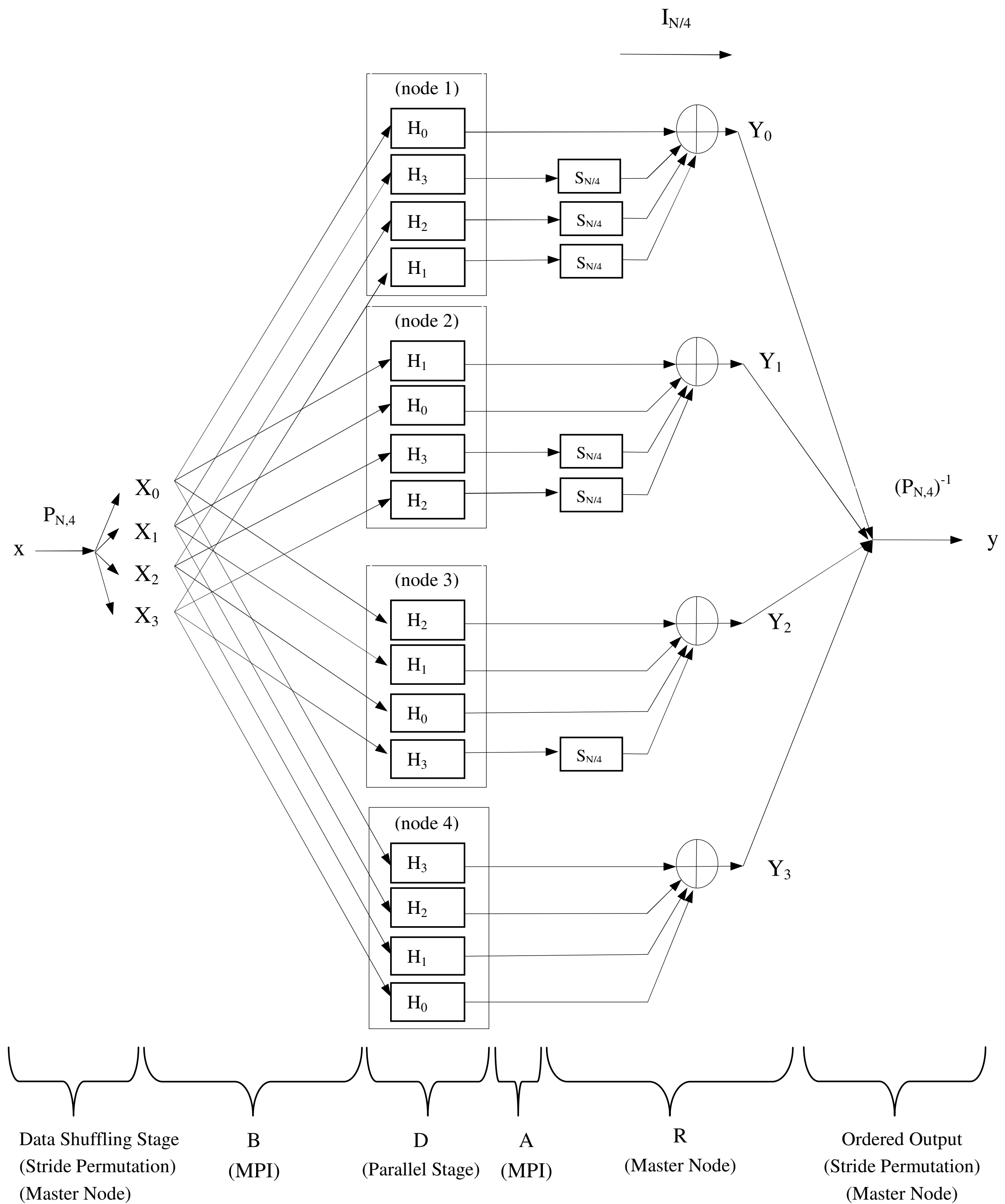

Figure 3. MPI Implementation using Radix-4 (16-Core). Stages B, D, A, R directly map to the same name matrices in equations (15) and (17). Note that no node intercommunication occurs.

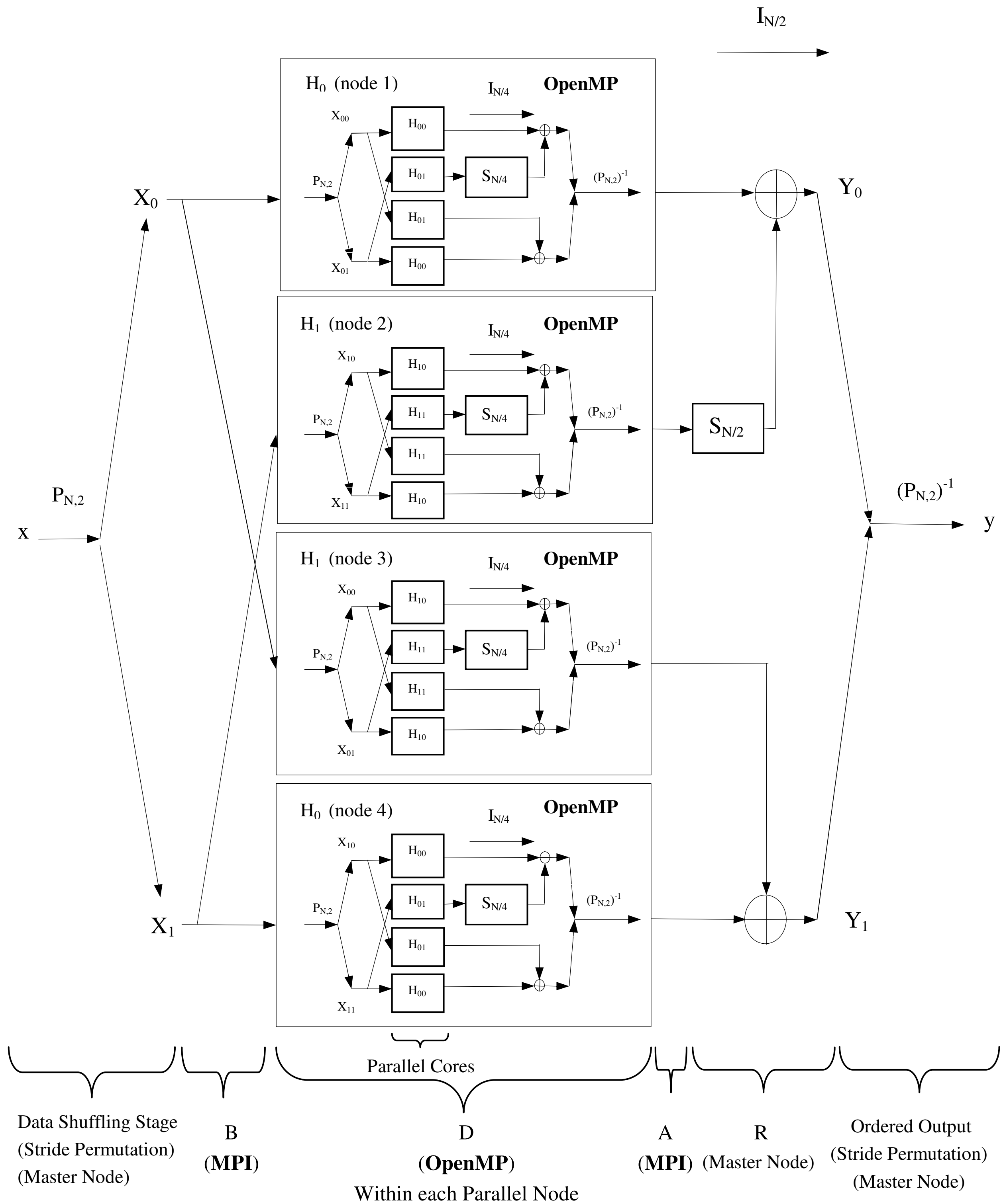

Figure 4. Mixed-Mode, MPI-OpenMP Implementation Recursively using Radix-2 (16-Cores). Stages B, D, A, R in the signal flow graph map to matrices B, D, A, R in equations (15) and (17).

Hardware Setup

Hardware Setup: 16-Nodes, 64 Processors, Dell Cluster

The current benchmark was carried out using a 16-node cluster with multi-core nodes. Each node has two, dual-core processors for a total of 64 cores. The cluster is somewhat dated but still useful for comparative and scalability studies. The cluster hardware specifications are:

Master Node:

- Brand: Dell PowerEdge 1950
- Processor: 2 X Intel Xenon Dempsey 5050, 80555K @ 3.0 GHz (Dual-Core Processor). 667MHz FSB. L1 Cache Data: 2 x 16KB, L2 Cache: 2 x 2MB.
- Main Memory: 2GB DDR2-533, 533MHz.
- Secondary Memory: 2 X 36GB @ 15K RPM.
- Operating System: CentOS Linux 5.6 64-bit.

Compute Nodes:

- Brand: Dell PowerEdge SC1435
- Processor: 2 X AMD 2210 @ 1.8GHz (Dual-Core Processor). L1 Cache Data: 2 x 128KB, L2 Cache: 2 x 1MB.
- Main Memory: 4GB DDR2-667, 667MHz.
- Secondary Memory: 80GB @ 7.2K RPM.
- Operating System: Linux 64-bit.

Software: C++

Benchmark Results

a) Direct MPI versus Hybrid, MPI-OpenMP, Implementations: 4-Nodes, 16-Cores

Table 1. Speedup of direct MPI over the Hybrid Mode for 16 Cores. Signal Length 2^M.

M	Hybrid Mode	MPI Mode	Speed Up
15	0.1325	0.1225	1.081632653
16	0.555	0.3375	1.644444444
17	1.0425	1.0575	0.985815603
18	3.59	2.775	1.293693694
19	9.1125	8.03	1.134806974
20	25.81	22.1925	1.16300552
21	75.305	64.1225	1.174392764
22	223.218	188.06	1.186950973

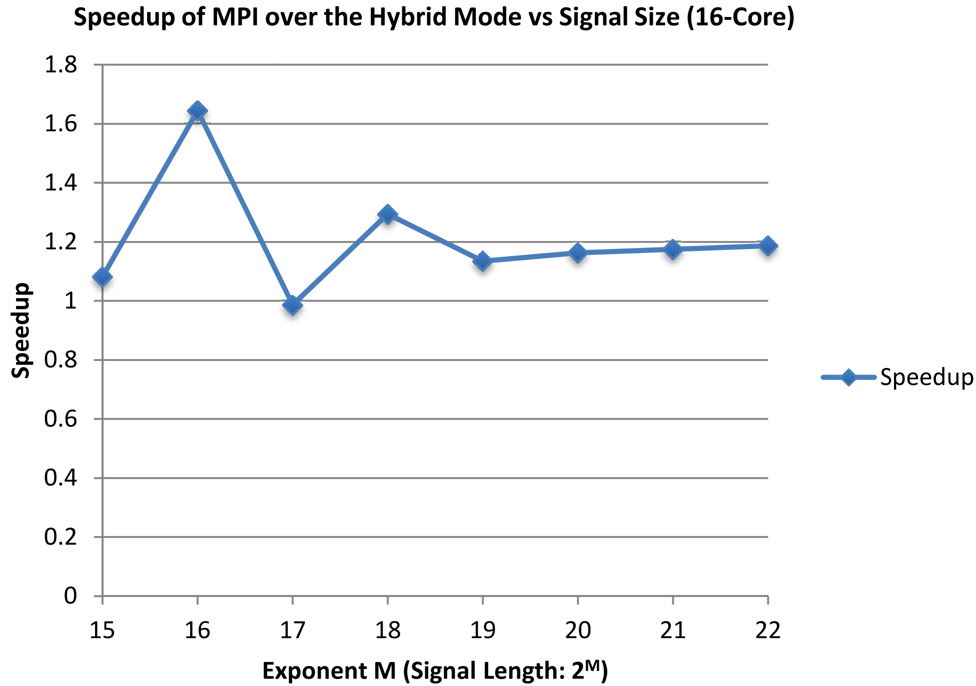

Figure 5. Speedup of Direct MPI over the Hybrid Mode for 16 Cores. Signal Length 2^M.

b) Direct MPI versus Hybrid, MPI-OpenMP, Implementations: 16-Nodes, 64-Cores

Table 2. Speedup of Direct MPI over the Hybrid Mode for 64 Cores, Signal Length 2^M.

M	Hybrid Mode (Seconds)	MPI Mode (Seconds)	Speedup
15	0.11	0.25	0.44
16	0.305	0.2575	1.184466019
17	0.7775	0.665	1.169172932
18	1.97	1.545	1.275080906
19	4.2825	3.835	1.116688396
20	10.58	9.7975	1.079867313
21	28.69	26.4	1.086742424
22	80.1725	72.3175	1.108618246
23	233.21	203.01	1.148761145
24	736.34	597.607	1.232147548

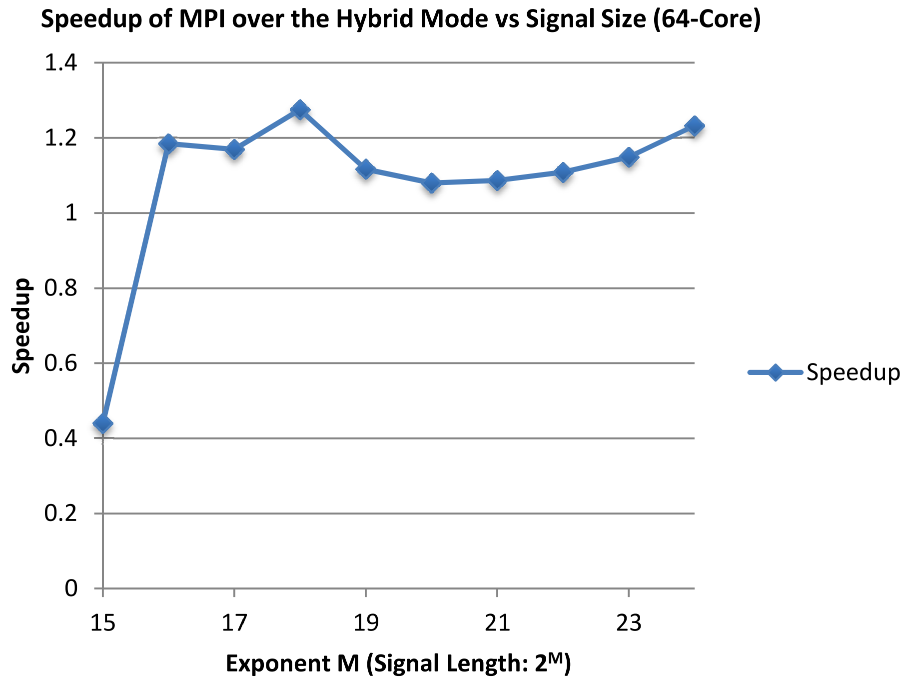

Figure 6. Speedup of Direct MPI over the Hybrid Mode for 64 Cores. Signal Length 2^M.

We can see that as the signal length increases the direct MPI implementation becomes faster than the mixed-mode implementation, Figure 5, 6, Table 1, 2. The fact that the hybrid implementation is slower than the MPI implementation appears to be mainly because of the compute time at the sub-convolution stage within the nodes. Considering that the absolute time is slightly larger for the mixed-mode approach, and looking to Figures 7, 8, it can be concluded that the transfer times are similar for both methods. There could be some particularity in the OpenMP portion of the implementation that is causing a relative slowdown in the compute time for the mixed-mode approach. If such a problem could be detected, further code optimization may be needed if we were to improve the performance of the hybrid implementation versus the MPI-only approach.

c) Transfer Time Studies

Figure 7 shows the transfer times for the direct MPI implementation. The transfer time percentage is slightly less for the hybrid MPI-OpenMP parallel implementation shown in Figure 8. The percentage of transfer time, with respect to the total time, increases as we increase the number of processors. This is because the total time for a fixed signal size decreases as more parallel processors are added. For a fixed number of processors as the signal length increases the percentage of transfer time, compared to the total execution time, decreases. Note that the mixed-mode approach needs a minimum of 16 parallel processors using the proposed formulation. The four-processor MPI implementation is shown just for reference since we are comparing the 16 and 64-processor implementations using the MPI approach versus the mixed-mode approach.

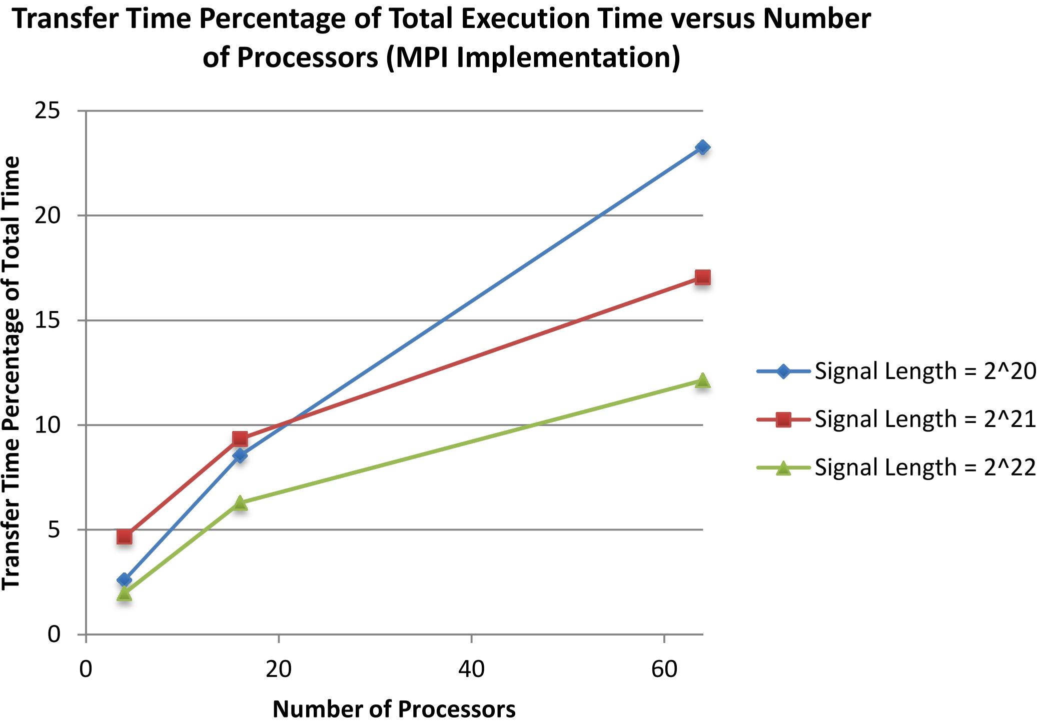

Figure 7. Transfer Time Percentage of Total Execution Time versus Number of Processors (4, 16, 64) for the MPI Implementation.

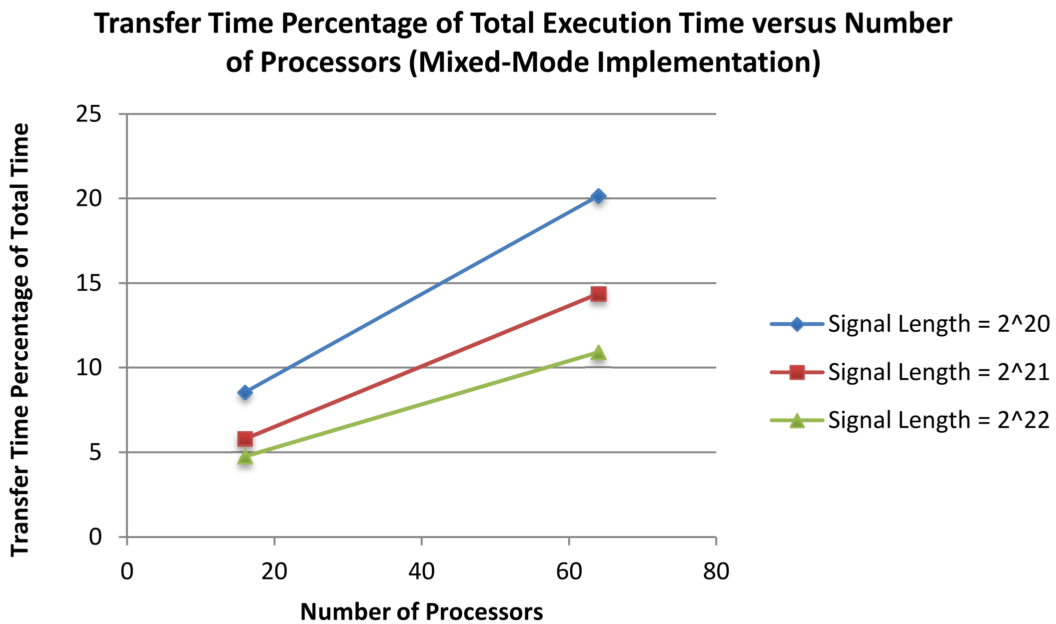

Figure 8. Transfer Time Percentage of Total Execution Time versus Number of Processors (16, 64) for the MPI-OpenMP Hybrid Implementation

d) Absolute Speedup of a Parallel Implementation versus a One-Core Serial Implementation.

For completeness we provide the speedup of the MPI-only parallel-recursive implementation versus the direct serial-recursive implementation, Fig. 9, Fig. 10. Since the parallel technique proposed in this paper uses a serial-recursive approach to perform each parallel sub-convolutions it can be benchmarked against the performance of a serial-recursive implementation in a single core. This benchmark shows that, as the signal length increases, the use of additional parallel processors increases the speedup of the parallel approach versus the serial approach. The increase in performance provided by this parallel scheme is sublinear.

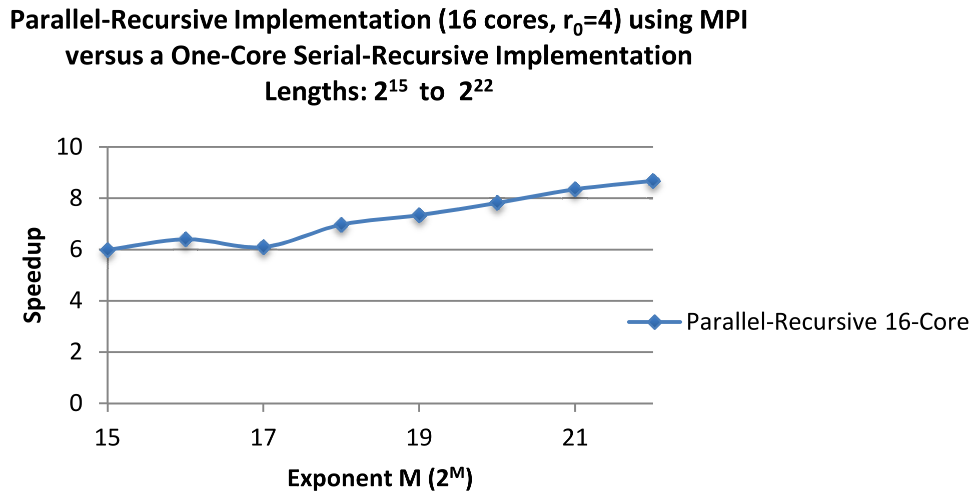

Figure 9. 16-Core Parallel-Recursive versus a One-Core Serial-Recursive Cyclic Convolution Implementation Speedup using C++

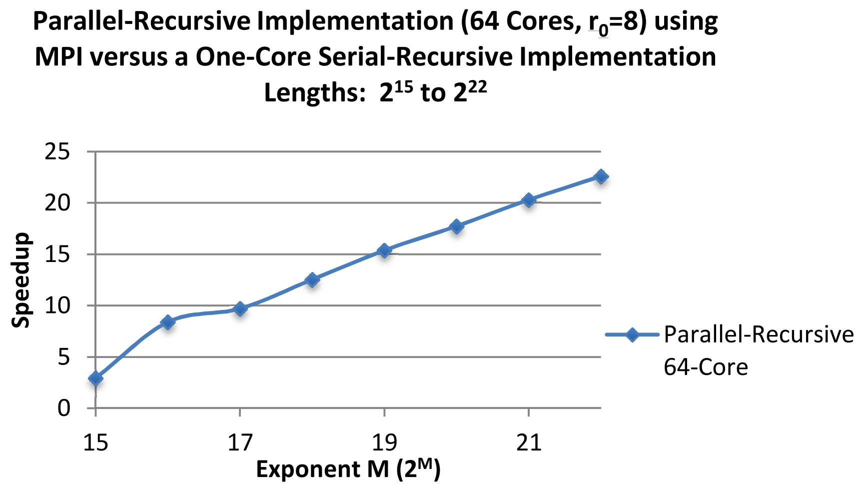

Figure 10. 64-Core Parallel-Recursive versus a One-Core Serial-Recursive Cyclic Convolution Implementation Speedup using C++

Educational Component

The reported case-study represents an example of a complex algorithm that can be reformulated in order to more closely conform to the underlying target architecture. Since case-studies provide a focused framework for the application of engineering knowledge[11], we have provided sufficient amount of detail so it can be used as a point of departure for further research. It can also be used as an educational resource in algorithm manipulation, and/or further code development, regarding the choice of mixed-mode (MPI-OpenMP) versus direct MPI implementations.

Future Work

As a future work we plan to benchmark for memory efficiency, where the mixed-mode approach could have an advantage, and to advance our code optimization efforts seeking an increased performance for both techniques.

Conclusions

By using MPI for distributing data to the nodes and then using OpenMP to distribute data among the cores inside each node, we matched the architecture of our algorithm to our target cluster architecture. Each core processes an identical program with different data using a single program multiple data (SPMD) approach. All pre and post-processing tasks were performed at the master node.

We found that the MPI implementation had a slightly better performance than the mixed-mode, MPI-OpenMP implementation. We established that the speedup increases very slowly, as the signal size increases, in favor of the MPI-only approach. Looking at Figure 3 and 4 it becomes clear that the MPI portion of both implementations should not suffer from load unbalancing problems.

The transfer time, as percentage of total time, is slightly less for the mixed-mode parallel implementation than for the MPI implementation. Since the total time for the hybrid approach is slightly higher, the transfer times for both approaches are basically the same. Given that the transfer times are similar, there could be some particularity in the OpenMP portion of the implementation that is causing a relative slowdown in the compute time versus the MPI approach. If such a problem can be detected further code optimization may be needed if we were to improve the performance of the mixed-mode implementation versus the MPI implementation.

When there are no unbalancing problems or other issues in the MPI portion of the algorithm the direct MPI option may be adequate[10]. This could account for the slightly less performance exhibited by the mixed-mode approach despite its tight match to the target architecture. This underscores the fact, as stated in the literature[10], that the mixed-mode approach is not necessarily the most successful technique on SMP systems and should not be considered as the standard model for all codes.

As expected, when the signal length increases the use of additional parallel processors sublinearly increases the performance of the parallel-recursive approach versus the serial-recursive implementation in a single core.

The presented case-study can be used as point of departure for further research or can also be used as an educational resource in algorithm manipulation, and/or additional code development, regarding the choice of mixed-mode (MPI-OpenMP) versus direct MPI implementations. The authors will gladly provide supplementary information or resources.

Acknowledgement

This work was supported in part by the U.S. Army Research Office under contract W911NF-11-1-0180 and by Polytechnic University of Puerto Rico.

References

1. C. Percival, “Rapid multiplication modulo the sum and difference of highly composite numbers,” Mathematics of Computation, Vol. 72, No. 241, pp. 385-395, March 2002.
2. R. E. Crandall, E. W. Mayer and J. S. Papadopoulos, “The twenty-fourth Fermat number is composite,” Mathematics of Computation, Vol. 72, No. 243, pp. 1555-1572, December 6, 2002.
3. H. J. Nussbaumer, Fast Fourier Transform and Convolution Algorithms. New York: Springer-Verlag, 1982, 2nd Ed., Chapter 3.
4. M. Bhattacharya, R. Cteutzburg, and J. Astola, “Some historical notes on number theoretic transform,” Astola, J. et al. (eds). Proceedings of The 2004 International TICSP Workshop on Spectral Methods and Multirate Signal Processing, SMMSP 2004, Vienna, Austria, 11-12 September 2004 25 pp. 289 - 298.
5. M. Teixeira and D. Rodriguez, “A class of fast cyclic convolution algorithms based on block pseudocirculants,” IEEE Signal Processing Letters, Vol. 2, No. 5, pp. 92-94, May 1995
6. M. Teixeira and Yamil Rodríguez, “Parallel cyclic convolution based on recursive formulations of block pseudocirculant matrices.” IEEE Transaction on Signal Processing, Vol. 56, No. 7, pp. 2755-2770, July 2008.
7. C. Cheng and K. K. Parhi, “Hardware efficient fast DCT based on novel cyclic convolution structures,” IEEE Trans. Signal Process., Vol. 54, No. 11, pp. 4419-4434, Nov. 2006.
8. C. Cheng and K. K. Parhi, “A novel systolic array structure for DCT,” IEEE Trans. Circuits Syst. II: Express Briefs, Vol. 52, pp. 366–369, Jul. 2005.
9. P. K. Meher, “Parallel and Pipelind Architectures for Cyclic Convolution by Block Circulant Formulation using Low-Complexity Short-Length Algoritms”, IEEE Transactions on Circuit and Systems for Video Technology, Vol. 18, pp. 1422-1431, Oct. 2008.
10. L. Smith and M. Bull, “ Development of Mixed Mode MPI/OpenMP Applications”, Scientific Programming 9, pp. 83-98, IOS Press, 2001.
11. L. G. Richards and M. E. Gorman, “Using Case Studies to Teach Engineering Design and Ethics”, Proceedings of the 2004 American Society for Engineering Education Annual Conference & Exposition.