

# COMBINING PARALLELIZATION TECHNIQUES TO INCREASE ADAPTABILITY AND EFFICIENCY OF MULTIPROCESSING DSP SYSTEMS

*Daniel M. Lorts*

Mercury Computer Systems, Inc.  
199 Riverneck Rd.  
Chelmsford, MA 01824  
978-256-1300, 978-256-0852  
dlorts@mc.com

## ABSTRACT

Current digital signal processing (DSP) applications are requiring processing-performance metrics on the order of Tereflops ( $10^{12}$  floating point operations per second). The announcements of RISC processors capable of performing billions of floating point operations per second and nearly an order of magnitude better on integer arithmetic address this need. Additionally, interconnect fabrics that scale nearly linearly to billions of bytes per second of traffic throughput have been developed to connect these processors together seamlessly. The remaining challenge for designers of parallel DSP systems is to architect solutions that efficiently and effectively utilize the available resources. These solutions must address key features such as adaptability, fault tolerance, and load balancing. The parallelization technique employed in the mapping of the application to the hardware assets greatly affects the success of achieving the key features. This paper presents and discusses the characteristics and benefits of four parallelization techniques — Round Robin, Data Parallel, Functional Parallel, and Hybrid Parallel — and provides some evaluation data from each. Quantitative results are collected by applying these techniques on a “generic” DSP application where the primary functions implemented (Hilbert Transform, Fast Fourier Transform (FFT) and frequency domain filtering) are representative of those required in other applications such as image processing and digital communications. The target system architecture is an embedded, shared-memory multiprocessing DSP system. The initial implementation and resulting performance data was performed on a Sequent parallel system configured with 20 CPUs on a shared-bus architecture. The intent of this initial effort was not the development of a system that was capable of real-time processing; rather, it was to experiment with the partitioning and allocation (parallelization) of DSP functions onto a multiprocessing system platform to find the optimal

task granularity and resource mapping. The parallelization metrics gathered will be utilized in the development of a real-time, embedded multiprocessing system. Concluding remarks provide a roadmap extending this research to provide additional adaptability through the incorporation of dynamic scheduling principles from an improved performance standpoint as well as increased system fault tolerance.

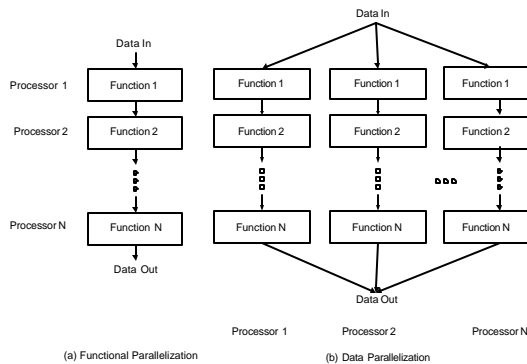
## 1.0 INTRODUCTION

As a result of the increased computational capabilities of digital signal processors (DSP), the development of new systems has migrated from systems based on general-purpose CPUs to those based on DSP optimized devices. This transfer of technology is occurring in both the commercial as well as the military industry. DSP devices are finding insertion opportunities in other industries once dominated by the general-purpose CPU including digital control applications and graphic applications.

The optimal approach in developing high-performing DSP systems requiring multiple processors is to develop a building block architecture of a limited number of specialized modules. In this scenario, systems could efficiently adapt to a changing environment. For example, if additional computing power was required of a system to perform the required functionality, additional processing modules (building block components) could be reallocated into the architecture to raise the performance level to the appropriate level. Similarly, if the functional requirements of the system are enhanced (or modified), the firmware of the existing computing resources could be modified. Ideally, the existing computing resources would be capable of performing the additional function and would simply require modifying the controlling routine to call the new routine. From the software developer's frame of

reference, the hardware platform appears as a pool of processing assets available for tasking.

A new paradigm for resource allocation and load balancing is required in this building-block approach to system construction. No longer is it efficient to strictly partition the application based on *functional parallelization* as typically done in pipelined systems (Figure 1a). On the other hand, strictly partitioning a problem based on the even distribution of the dataset, referred to as *data parallelization*, is unfeasible since the complex nature of present day applications typically require sharing of some or all of the data among functions (Figure 1b). These two techniques represent the extremes of a two-dimensional parallelization spectrum. Balancing the degree of functional and data parallelization implemented within a multiprocessing system provides the best solution. This latter case will be referred to as the *hybrid parallelization* technique.



This paper documents the evaluation of the three types of parallelization techniques described in the

Figure 1 Visualization of functional and data parallelization.

previous paragraph. The test algorithm used in the research is representative of the functions that could exist in many different applications from a hi-fidelity digital audio system to a radar system. The evaluation model was developed on a shared-memory multiprocessor system (Sequent Symmetry S81). In addition, the test algorithm was executed on a single processor as a baseline case for analyzing the resulting "speedup" of the different parallelization techniques. This uniprocessor implementation provides the analytical data for a fourth parallelization technique called *round robin*. The round robin technique takes advantage of multiprocessors in a temporal sense tasking available

resources to process the next time frame of information.

Section 2.0 presents an overview of the test algorithm functions and key parallelization content. Details of the implementation of each of the parallelization techniques are provided in Section 3.0. The analytical results from the different implementations of the parallelization techniques are reported in Section 4.0. The final section, Section 5.0, concludes with a review of the results and offers suggestions for mapping the system onto a real-time, embedded multi-DSP architecture.

## 2.0 TEST ALGORITHM

The dataflow diagram of Figure 2 illustrates the sequence of operations performed in the evaluation system. It also happens to represent the dataflow of a typical DSP platform with the Digital Filter being replaced by some large scale processing task(s). The system consists of three primary stages with each stage composed of one or more functions. The three stages consist of a Preprocessing Stage, the System Function, and the Post Processing Stage. Details of each test function are provided in the following sections.

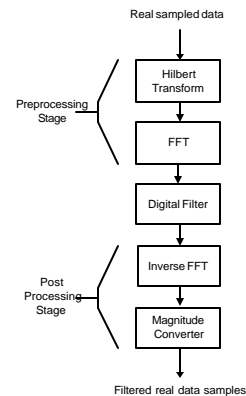


Figure 2 Process flow of application to be parallelized.

### 2.1 PREPROCESSING STAGE

The input data is assumed to be a vector of *real* numbers,  $x_r[n]$ , which are transformed into the frequency domain within the Preprocessing Stage. Prior to the transformation to the frequency domain the real input sequence is processed via Hilbert transform relationships to generate the corresponding *imaginary* sequence,  $x_i[n]$ . Together, the real and imaginary components completely define the

complex input signal structure as required to meet the constraints for causality [1]. The impulse response of the Hilbert transformer is defined in Equation 1.

$$h[n] = \begin{cases} \frac{2}{\pi n} \sin^2\left(\frac{\pi n}{2}\right), & n \neq 0, \\ 0, & n = 0. \end{cases} \quad (1)$$

Convolving the impulse response coefficients of Eq. (3) and the real input sequence samples we obtain the expression for the imaginary component as defined in Equation 2.

$$x_i[n] = \sum_{k=-\infty}^{\infty} x_r[k]h[n-k]. \quad (2)$$

In actual implementations the upper and lower bounds of the convolution sum are determined by the desired characteristics of the resulting impulse response. Experience has shown that a sum of 191 elements provides sufficient representation for calculating the remaining component of a complex pair. To avoid distorting the first and last 95 components generated by the Hilbert transformer, an extra 190 elements are utilized from the Input Stage to properly fill the processing pipeline. The output of the Hilbert Transform function is the original real sequence along with its corresponding complex component.

The complex sample pairs are then transmitted to the FFT which does the transformation of a time domain sequence to the frequency domain. The FFT is actually a collection of algorithms optimized to reduce the  $O(n^2)$  computational requirements of the discrete Fourier transform (DFT) illustrated in Equation 3.

$$Y[k] = \sum_{n=0}^{N-1} x[n] * W_N^{kn} \quad (3)$$

$$\text{where: } W_N^{kn} = e^{-j\left(\frac{2\pi}{N}\right)kn}.$$

Cooley and Tukey (1965) observed that the weighting term,  $W_N^{kn}$  (commonly referred to as the twiddle factor), exhibited special symmetry and periodicity which could be exploited to decompose the DFT into successively smaller DFT problems. The resulting decomposition consists of  $\log_2 N$  stages with  $N/2$  kernel operations referred to as the *butterfly* per stage. Therefore, an FFT requires  $O(n \log_2 n)$  operations.

The decomposition of a trivial 8-point FFT is illustrated in Figure 3. The complex output sequence,

$Y[n]$ , is the resulting weights of the corresponding frequency bin. The kernel *butterfly* operation is illustrated in Figure 4. It is with the careful partitioning and allocation of these butterfly operations that the FFT can be efficiently mapped to a parallel architecture.

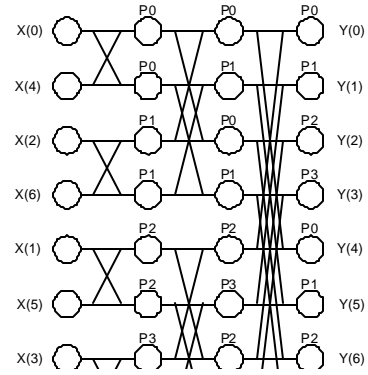


Figure 3 Simple illustration of the FFT algorithm.

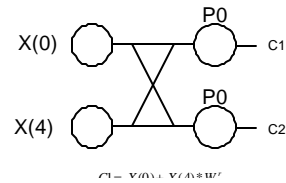


Figure 4 The FFT Butterfly kernel.

## 2.2 MAIN PROCESSING STAGE

The main processing stage may incorporate any operation or combination of operations that manipulates the data stream to realize the end results of the system. Everything up to this point has been performed to condition the data appropriately. For this research a simple digital domain filter was selected. Essentially, this process will consist of modulating the frequency bins calculated by the FFT according to a predefined weighting file defined prior to runtime. The values can be greater than unity thereby integrating digital gain capabilities into the filter function.

## 2.3 POST PROCESSING STAGE

The modulated frequency domain signal is transformed into a complex time domain sequence by

the inverse-FFT function. The complex time-domain sequence is converted back into a real-data sequence via the magnitude converter function. The final magnitude conversion operation is required only if the desired output of the system was a real, time-domain sequence. Otherwise, any other appropriate post processing function would be substituted.

### 3.0 PARALLELIZATION TECHNIQUES

Parallel processing provides hope for system developers that the ever-increasing performance demands of applications will be achievable. Most parallel-processing strategies are based upon the efficient partitioning and distribution of work onto the available processing resources to minimize the amount of time that any one resource is not productive (idle). The various approaches include hardware and software methods each aimed at specific optimization goals. This project concentrates on three methods associated with parallelization at the algorithm level. The three techniques differ in the way that the work is partitioned.

#### 3.1 SEQUENTIAL EXECUTION

A fourth mode of operation was implemented to execute the application on a single processor of the target system. This was to gather baseline performance information that would be used in determining the relative speedup of the parallel modes.

#### 3.2 FUNCTIONAL PARTITIONING

In *functional partitioning*, the application's computational functions are assigned to specific processing resources and execution proceeds in a pipeline fashion. An advantage of this approach is a reduced development time since the functions written as typical sequential code require no modifications (assuming shared-memory architecture). The primary disadvantage is that the pipeline's overall performance is limited by the most computationally intensive function in the processing thread.

The program image consisting of five functions is loaded onto five processors. The processors execute a function within the pipeline based upon their processor ID value. Since the target architecture is based on a linear bus architecture the static assignment principles implemented did not have to account for relative processor positioning. This will not be the case for most embedded processing system.

### 3.3 DATA PARTITIONING

The partitioning of the data set among the available processing resources is referred to as *data partitioning*. In this mode of operation each function is executed to completion before the next function is allowed to begin. The data to be processed by each function is partitioned across an optimum number of processing resources and operated upon in parallel. Optimal performance gains are realized if the data sets are mutually exclusive thereby allowing each partition to execute to completion without the need for interprocess synchronization or communication. Ideally, the reduction in time would be linearly proportional to the number of resources used. Unfortunately not all functions readily lend themselves to this tactic and require some manipulation to get to a form that can be data partitioned. The FFT was an instance of such an algorithm because of the recursive nature between the multiple stages of processing. The advantage to this method of partitioning is the capability to spread the processing of a function across a number of resources determined to optimum for that function (through experimentation) or limited by resource availability.

The primary concern in data partitioning is the size of the partitions. If the partitions are made too granular (small number of elements) the processors will incur a significant amount of time performing overhead chores for synchronization and communication. Conversely, if the partitions are too coarse there exists a strong possibility that a number of processors will sit idle while the remaining processors are overworked due to the large data set allocation.

The optimal partitioning requires experimentation with the granularity size of the data partitioning and the number of processors allocated to the problem. In this research we tried several different processor set sizes with results tabulated in the next section. The granularity size varies with the function implemented. For example, the FFT function is partitioned such that a processor calculates all of the butterflies using the same twiddle factor within the current stage of execution. Initially this translates to only two processors being utilized. As the algorithm progresses to the next stage of processing, the number of processors is doubled. Eventually, the number of butterfly groups exceeds the number of processors requiring some (or all) of the processors to be assigned additional work.

### 3.4 HYBRID PARTITIONING

Hybrid partitioning combines the advantages of both data partitioning and functional partitioning. The system is implemented in a pipeline architecture such that a continuous stream of information can be processed. But instead of each function being allocated to a single-processing resource, as in functional partitioning, it is data partitioned across a number of resources. Typically, the more resources available the better the performance improvements achievable up to the point where communication and synchronization requirements become significant.

The result is that functions that are more computationally intensive can be partitioned into more sub-functions and allocated more processing resources. In this research the functions are statically scheduled onto the processing resources. Both static and dynamic scheduling techniques are utilized in processing the sub-functions. Static methods were utilized in programming the FFT and Inverse FFT algorithms due to their rather difficult structure and strong recursive needs.

The dynamic scheduling principles were incorporated in the algorithms for the Hilbert Transform and the Digital Filter. These functions could also as easily been coded using the static methods. But in an attempt to illustrate the various techniques available to the developer these two functions were coded as such. The structure of the loop control for dynamic scheduling is significantly different in that the initial index values are not dependent on a processor ID value. There exists a monitor responsible for allocating chunks of work to requesting processors, specifically, it provides an initial index value to the requester. The amount of work to be processed for each call to the loop is determined by the 'chunksize' variable. The best chunksize value is ultimately determined by trial and error. A chunksize of 128 samples was chosen for the Hilbert Transform and for the Digital Filter a chunksize of 1 was selected.

### 4.0 ANALYTICAL RESULTS

The results of the processing were very impressive and reassuring. Table 1 summarizes the initial timing results for the four parallelizing techniques.

FUNCTION	SEQUENTIAL	FUNCTIONAL	DATA (4)	HYBRID
HILBERT	6.27	6.27	1.60	3.14
FFT	1.05	1.08	0.47	0.47
FILTER	0.08	0.08	0.05	0.06
IFFT	1.07	1.07	0.50	0.50
MAGNITUDE	0.11	0.12	0.03	0.12
TOTAL	8.58	8.62	2.65	4.29

As predicted, the individual function results of the pipeline implementation were equal to that of the sequential execution. To reiterate, the primary advantage of the pipeline mode of operation is that new data sequences can be processed by the system in a fraction of the amount of time on a uniprocessor. This fraction is determined by the slowest link in the pipeline chain. In this example the Hilbert transformer had the longest processing stage. What this translates to is that new data packets of 2048 samples can be processed every 6.27 seconds instead of the 11.60 seconds of the sequential execution. It would be advantageous to spend time optimizing the Hilbert transformer algorithm if increased performance is required.

Data Parallelization of the algorithms across four processors produced some rather impressive results as the Data column of Table 1 documents. Speedups attained in the data parallelization efforts are also presented in Table 1. Of particular note, the performance of the Hilbert transformer, which we identified as the most computing-intensive function of the simulator, improved nearly linearly. The Fourier-based algorithms improved by a factor of two, somewhat short of expectation but could be the result of selecting too small of chunks in the data partitioning scheme. Also, because the algorithms consist of recursive loops that are relatively "tight" and change indices limits every stage it is difficult to keep all resources active at the beginning of the routine. Additional effort spent in parallelizing the functions may produce incremental speedup but we reach a point of diminishing return.

Additional testing was performed with the data parallelization model by increasing the number of processors allocated to the application. Table 2 records data attained with 8, 10, and 19 processors.

Table 2 Additional resources applied to the data parallelization technique.

Table 1 Summary of initial parallelization implementations.

Speedups are also calculated for each implementation. It is worth noting that the Hilbert Transform remained consistent with its near linear improvements. Overall, the application speedup

FUNCTION	DATA (8)	SPEEDUP	DATA (10)	SPEEDUP	DATA (19)	SPEEDUP
HILBERT	0.84	7.46	0.66	9.50	0.43	14.58
FFT	0.37	2.92	0.34	3.17	0.31	3.48
FILTER	0.03	2.67	0.03	2.67	0.02	4.00
IFFT	0.38	2.82	0.37	2.89	0.33	3.24
MAGNITUDE	0.03	4.00	0.03	4.00	0.04	3.00
TOTAL	1.65	5.22	1.43	6.03	1.13	7.63

diminished as we increased the processor count. This is most likely due to the small dataset used and the somewhat small chunksize.

The timing values for the hybrid technique in which both functional and data partitioning techniques are combined are consistent with previous data. The functions showed similar speedups due to the data partitioning for the number of processors being utilized. The values attained in the hybrid executions indicate that additional resources should have been applied to the Hilbert transformer function and fewer to the Fourier algorithms as well as using only one processor for the Digital Filter functions.

## 5.0 CONCLUSIONS

It has been illustrated that the parallelization techniques of functional partitioning and data partitioning can provide significant speedup when applied to DSP functions. The platform used to analyze this capability was a simple, general-purpose multiprocessor system. Static and dynamic process scheduling strategies were examined briefly during Mode 3 execution. Dynamic scheduling provides improved load balancing characteristics but at a cost of increased algorithm complexity and processing overhead associated with communication and synchronization needs. All of these techniques were applied in the hope of leveling the computational load of an application across the processing resources available.

The implementation of this generic application required the programmer to incorporate explicit parallelism constructs into the functions without knowledge of the resources current loading characteristics. The ultimate approach would be to develop software algorithms (run-time code and/or compiler technology) that would incorporate dynamic scheduling policies for allocating the tasks onto the available hardware resources based on run-time

statistics and characteristics (which in a real implementation is not typically deterministic). This topic is currently being researched for systems based on dataflow principles.

The results of this effort will be ported to a real-time embedded multi-DSP system based upon the Mercury Computer Systems' RACE++™ architecture and components. Ideally, with improvements in architecture and technology our results will achieve throughput performance supportive of the requirements of existing real-time applications whether an audio or radar processor. Increased performance will be attainable through the optimization of the parallelization techniques, optimizing the ratio of implemented techniques as well as improvements in the algorithms. Further research and analysis is intended to continue this concept toward determining if rules-based tool to automatically parallelize an application is feasible.

## REFERENCES

- [1] A.V. Oppenheim and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989