# ACCELERATED SIMULATION OF COMMUNICATION WAVEFORMS USING THE MATLAB DISTRIBUTED COMPUTING TOOLBOX

Brendan Garvey
(General Dynamics C4 Systems, Scottsdale, AZ, USA)
brendan.garvey@gdc4s.com

## ABSTRACT

Developing new communication algorithms and waveforms typically requires significant engineering effort and extensive, time-consuming simulations. MATLAB and Simulink are often used for this type of work. Although these tools have proven valuable in developing and simulating waveforms, designs of even modest complexity can require long simulation times. The Mathworks now offers a distributed computing solution, the MATLAB Distributed Computing Toolbox, which has the potential to address this problem.

This paper presents a methodology and algorithm for significantly accelerating the simulation of communication waveforms using the MATLAB Distributed Computing Toolbox running on a computer cluster. In the proposed approach, a single Monte Carlo simulation is broken into numerous smaller (shorter) Monte Carlo simulations, running on multiple computers, and the results averaged together to create the final answer. The parallel Monte Carlo approach is already being used by researchers and engineers in other technology areas [1, 2]. A MATLAB m-file, **dc_sim.m**, has been developed to implement this approach and is described in this paper. The proposed algorithm is general purpose in nature which makes it easier to add additional computers to the cluster if desired.

## 1. INTRODUCTION

The purpose of this paper is to show how the MATLAB Distributed Computing Toolbox (DCT) can be used to create a parallel Monte Carlo (PMC) waveform simulation in order to significantly reduce simulation times. A MATLAB function, dc_sim(), has been written to implement the proposed approach. The core algorithm within dc_sim() is described in detail in order to show how a PMC waveform simulation can be implemented. One key goal was to create a general purpose algorithm that would allow additional computers to easily be added to the cluster in order to attack complex waveform problems which require significant simulation time.

The approach proposed here is currently being evaluated at General Dynamics C4 Systems on a cluster of 10 computers connected over the standard company network.

The outline of this paper is as follows. First, the basics of the MATLAB Distributed Computing Toolbox are covered, in order to provide the necessary background and terminology. Next, the key concept and general requirements for a parallel Monte Carlo (PMC) waveform simulation are presented. This is followed by implementation details of the dc_sim() MATLAB function. Finally, performance results are presented, based on an IS-95 MATLAB model which has been modified to run with dc_sim().

## 2. OVERVIEW OF THE MATLAB DISTRIBUTED COMPUTING TOOLBOX

There are two separate products that make up The Mathworks distributed computing environment: 1) the Distributed Computing Toolbox (DCT) and 2) the MATLAB Distributed Computing Engine (MDCE or DCE). DCT is a set of commands and functions that are executed from the MATLAB command window, or from an m-file, just like other toolboxes. The computer on which DCT is running is defined here as the DCT client. MDCE is a service that runs in the background on computers running either worker or job manager processes.

Figure 1 shows the basic MATLAB distributed computing environment [3]. The DCT client defines and submits jobs to the job manager. The job manager process, which is controlled by MDCE, "coordinates the execution of jobs and the evaluation of their tasks" [3]. The job manager can run on any computer that has MDCE installed. A worker is a separate process, also controlled by MDCE, which evaluates a task assigned to it by the job manager, and returns the results to the job manager. A worker process is usually installed on its own computer, so that multiple workers can operate simultaneously to complete a job faster. However, for initial development and debugging of a waveform, DCT, the job manager, and worker processes can all be installed on the same computer. In the rest of this paper, the term “worker” will be used to refer to a computer running a worker process.

Beginning with version 2, DCT now supports both “distributed jobs” and “parallel jobs” (Mathworks’ terminology). In a distributed job, the tasks do not directly communicate with each other; in a parallel job, the tasks can communicate with each other. The algorithm described here uses the simpler distributed job functionality. Going forward, this paper uses the terms “distributed” and “parallel” somewhat interchangeably, but remember that the algorithm is using the distributed job approach.

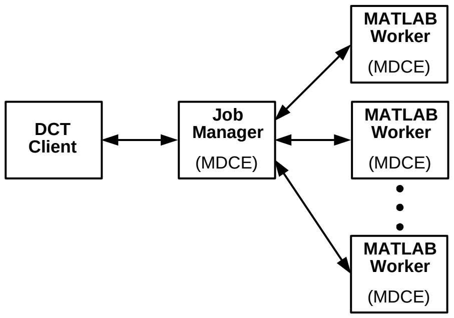

**Figure 1: Basic MATLAB Distributed Computing Environment**

## 3. KEY CONCEPT

Most simulations of communication waveforms utilize the Monte Carlo method to create an estimate of the desired observable. Usually the desired observable is the bit-error-rate (BER), the symbol-error-rate (SER), or, for acquisition simulations, the probability of detection or probability of false alarm. Typically, when simulating new waveforms or algorithms, the system engineer repeatedly modifies different aspects of the design and monitors the impact on the observables in order to create a design that meets system requirements.

A Monte Carlo simulation can be broken into smaller (shorter) Monte Carlo simulations, running in parallel, and the results averaged together to create the final estimate. This is referred to as “parallel Monte Carlo”. For example, let us say that a particular waveform model requires the simulation of 10 million symbols in order to create an adequate estimate of the SER. The waveform model could be simulated on 10 computers, with each individual computer running 1 million symbols through the waveform model. The final estimate of the SER would be the sum of all symbol errors divided by 10 million. The actual functionality of dc_sim() is more involved, but this is the main idea. Although this idea is conceptually straightforward, *care must be taken in how the simulation is partitioned, and how the random noise sequences are generated, to ensure that the individual simulations are statistically independent.*

Figure 2 is a simple example which shows the key concept in picture form. A communications signal is generated, and a noise signal is added to it. The signal is partitioned into 4 blocks and each block is simulated on a different worker. There are 4 copies of the same waveform simulation, 1 per worker. The only thing different about each simulation is the noise sequence and the input data. The unique noise sequences are represented by $N_j$ in the figure. The segments are simulated in parallel, and the results are averaged to form the final estimate (parallel Monte Carlo).

Figure 2 presents a convenient way to visualize the PMC process at a high level. However, several points need to be made concerning the actual implementation.

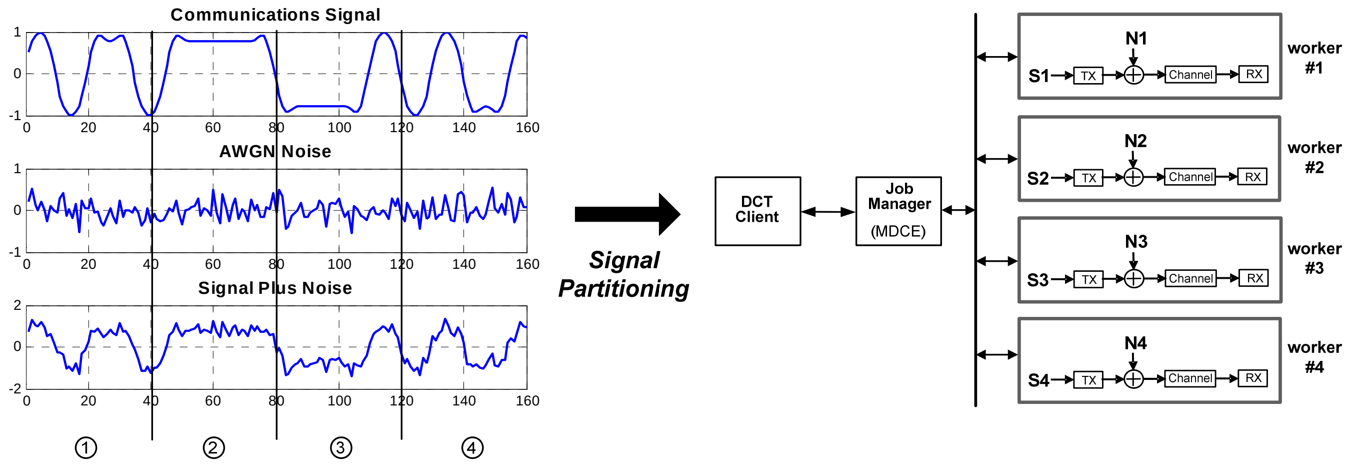
Communications Signal
AWGN Noise
Signal Plus Noise
① ② ③ ④
Signal Partitioning
DCT Client
Job Manager (MDCE)
N1 S1 TX Channel RX worker #1
N2 S2 TX Channel RX worker #2
N3 S3 TX Channel RX worker #3
N4 S4 TX Channel RX worker #4

**Figure 2: Key Concept – Partitioning a Monte Carlo Simulation into Parallel Pieces**

First, figure 2 implies a one-time division of the input signal. Actually, the input waveform is divided into numerous simulation blocks that are continuously fed to the workers. Second, no attempt is made to maintain the state of the model at the signal block boundaries; each worker runs a semi-independent Monte Carlo simulation and does not receive input from any other worker. Third, in an actual simulation the number of symbols per block would be much larger than what is implied by figure 2.

## 4. REQUIREMENTS / FEATURES FOR A PMC WAVEFORM SIMULATION

Now that the key concept has been presented, this section presents general requirements and desirable features for a distributed waveform simulation using a PMC approach.

**Parallel Random Number Streams** - "Parallel random number generators should … have no correlations between the sequences on different processors, produce the same sequence for different numbers of processors, and not require any data communication between processors" [2, 4]. Although not a requirement, it is also desirable that the selected method for generating parallel random numbers build upon existing MATLAB functionality. Function dc_sim() meets these requirements by using "sequence splitting" [4], centrally controlling the noise seed table at the DCT client, and using MATLAB's randn() function.

**Repeatability** - The simulation should produce identical results regardless of the number of computers used in the cluster. If a simulation is run on Friday with 5 computers, and the same simulation run again on Monday with 10 computers, the results should be identical. This requirement is also met by centrally controlling the noise seeds at the DCT client.

**Reliability** - As the number of computers in the cluster goes up, the probability of having a computer problem also goes up. The simulation should provide a means to recover from a worker failure and remove the worker from the cluster. This functionality has not yet been added to dc_sim().

**Usability/Expandability** - When engineers have multiple computers available for a simulation, an ad hoc approach is typically used. For example, each computer might run an Eb/No curve using a different value of a key parameter. However, for increased usability, a PMC waveform simulation should provide a method of parallelization that is independent of the waveform model, so that computing resources can be easily added or subtracted from the simulation. Function dc_sim() has been designed to meet this requirement.

## 5. IMPLEMENTATION DETAILS

In this section we will explain how a PMC waveform simulation can be implemented by using dc_sim() and the modified IS-95 MATLAB model as an example. We will assume that there are $\boldsymbol{k}$ workers available, and that the goal is to generate an Eb/No vs. BER curve. Let $\boldsymbol{L}$ be the block size in packets (definition of block size is below), and let $\boldsymbol{M}$ be the quantity of random numbers needed per block. Before presenting a detailed description of dc_sim(), several supporting topics are discussed below.

**Computer Cluster –** Strictly speaking, this usually means a dedicated group of computers networked together, somewhat isolated from the company network. In this paper we will use the term in a looser sense to also include a group of computers connected over the company LAN via DCT and MDCE.

**Dynamic Cluster Size** – Although a job manager may have access to $\boldsymbol{k}$ workers, a simulation may limit itself to $\boldsymbol{j}$ workers, where $\boldsymbol{j} < \boldsymbol{k}$. The amount of computing resources actually used in a simulation will be called the dynamic cluster size.

**Jobs and Tasks** – In DCT, a job is essentially a group of tasks that need to be performed. Function dc_sim() only uses 1 task per job. A task has a function handle and input arguments associated with it; the function handle points to the function that the task is supposed to evaluate when the parent job is submitted.

**Packet Size -** for the IS-95 forward channel, each packet is 20 ms in duration and represents 184 information bits. Eight tail bits are added, the packet is R=1/2 encoded and then spread by a factor of 64. There are therefore 24576 chips per 20 ms packet, which is a chipping rate of 1.2288e6.

**Simulation Block Size** - The input signal is chopped up into blocks. Each worker simulates a block at a time, and returns the results to dc_sim() via the job manager. The simulation block size is the granularity of the PMC waveform simulation. The block size is somewhat arbitrary – it should be large enough so that the workers are not constantly interacting with the DCT client, but should be small enough so that progress can be seen by the user when the simulation is run interactively. For the IS-95 simulation, the block size was typically set between 2 and 10 packets. As will be seen in the section on performance results, the efficiency of the cluster is a direct function of the simulation block size.

**Random Numbers per Simulation Block -** Each simulation block requires $\boldsymbol{M}$ random numbers. The chipping rate is 1.2288e6 complex chips/sec, and the IS-95 model uses an analog (oversample) rate of 5. The random number requirement is therefore:

***M*** = (1.2288e6)(2)(5)(20 ms)****L*** = 245760****L*** random numbers per simulation block.

**Noise State Table** - One of the fundamental challenges for the PMC waveform simulation is the generation and coordination of random numbers - each block of random numbers must be uncorrelated from every other block. Because of the network bandwidth it would require, we certainly wouldn't want to pass the random numbers from the client to the workers; the ***k*** workers should generate their own random numbers. However, the resulting ***k*** streams of random numbers must be uncorrelated.

One straightforward technique to create uncorrelated random number streams is to take a single random number sequence and repeatedly divide it up into non-overlapping sub-sequences. In the technical literature, this is sometimes referred to as "sequence splitting" [2, 4]. The assignment of these sub-sequences to workers is centrally controlled by function dc_sim() via the noise state table. This table holds an array of noise seeds, each noise seed represents the start of a sub-sequence. When dc_sim() creates a job, one of the input parameters to the job is a noise seed taken from the noise state table.

Sequence splitting can be accomplished in a straightforward manner using the MATLAB randn() function. The MATLAB command *state1=randn('state')* returns the current internal state of the randn() generator. Later, the randn() generator can be re-initialized to *state1* using the command *randn('state', state1).*

To generate the noise state table, we create a loop which generates ***M*** random numbers at a time. After each block of ***M*** random numbers has been generated, we throw away the generated numbers, but save the state (a.k.a. the noise seed) into the noise state table. Later, each time dc_sim() submits a new job to a worker, one of the parameters passed to the worker is a noise seed out of the noise state table. The noise state table is generated once, saved to a MAT file, and loaded when dc_sim() is initialized.

Now that the supporting concepts have been presented, let's discuss in detail the functionality of dc_sim() and how it is used within a PMC waveform simulation. Figure 3 is a simplified block diagram that will be used to explain the overall functionality.

**Top-Level Driver** - The top-level driver first calls dc_sim() to load the noise state table. Then, from within a "for" loop, it repeatedly calls dc_sim() for each Eb/No point to be evaluated. The top-level driver passes in to dc_sim() the model parameters needed to evaluate the model, simulation parameters needed by dc_sim(), and a pointer to the waveform evaluation function.

**dc_sim()** – Function dc_sim() submits jobs to the workers (via the job manager) and collects results. It continues to do this until the target number of errors has been reached. It sums all of the bit/symbol errors, and returns this data to the top-level driver. Function dc_sim() acts as an interface between the top-level driver and the job manager. The top-level driver doesn't contain any distributed computing toolbox commands and doesn't know about the job manager; dc_sim() takes care of all that. The main components within dc_sim() are the noise state table, the run queue, and the results queue. The noise state table has already been described. The run queue is an array that contains an object for each active simulation job. Function dc_sim() continuously monitors the run queue. When a job finishes, dc_sim() records the results in the results queue, removes the old job from the run queue, and submits a new job in order to keep all ***k*** workers busy.

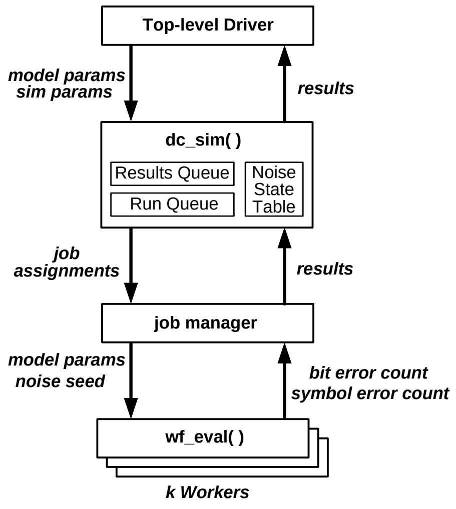

**Figure 3: PMC Simulation Architecture**

The results queue is implemented as a MATLAB cell array. Since the individual workers may have different performance levels, the jobs may not finish in the same order that they were submitted. However, regardless of the number of workers, the results are stored in the cell array in the same order as the jobs were submitted.

Now let's take a more detailed look at dc_sim(); figure 4 on the next page is an outline of the core portion of dc_sim() using MATLAB syntax. There are three main loops in the algorithm, an outer loop and two inner sub-loops.

Let's start with the outer "while" loop. Variable *done* is set to "1" when the simulation end conditions have been met. The end conditions are usually a target number of bit or symbol errors, and a perhaps a minimum number of simulated symbols. Variable *job_cnt* is the number of jobs that have been submitted. Variable *done_jobs* is the number of jobs that have finished. The function of the outer loop is to continue running the session until the target end conditions have been reached (*done* goes high). After *done* goes high, the outer loop continues the session until all of the submitted jobs have been completed (until *done_jobs* no longer less than *job_cnt*).

```
%%------------------------------------------------------
%% Outer Loop
%%------------------------------------------------------
while (done_jobs < job_cnt || done == 0)
   ...
   qlen = length(run_queue);
   %%---------------------------------------------------
   %% Sub-loop #1
   %% Get results from finished jobs, update run queue
   %%---------------------------------------------------
   for i = 1:qlen

      sim_obj = run_queue(i);
      job_stat = get(sim_obj.job, 'State');

      if (strcmp(job_stat, 'finished') == 1)
         ...
      else
         new_run = [new_run sim_obj];
      end

   end
   run_queue = new_run;

   %%---------------------------------------------------
   %% Sub-loop #2
   %% Keep submitting jobs to keep all workers busy
   %%---------------------------------------------------
   while (length(run_queue) < max_wkrs && done == 0)
      job = createJob(jm1);
      ran_state = nst{nst_ptr};
      tsk = createTask(job, fhandle, 3, {,ran_state,});

      %% Initialize object sim_obj with job info
      ...

      submit(job);

      %% Update run queue
      run_queue = [run_queue sim_obj];
   end
end
```

**Figure 4: Function dc_sim() Core Functionality**

Now let's look at sub-loop #1. This loop examines all of the current jobs in the run queue. If a job has finished, the results are extracted. If the job is still running, the job object is added to the new run queue *new_run*. The specific duties of sub-loop #1 are:

1) get results from recently finished jobs, and store them in the results queue,
2) update the bit and symbol error counters,
3) if target end conditions are met, set *done* to 1,
4) create a new run queue which contains a reference to jobs that are still running.

Sub-loop #2 contains the code that actually uses DCT commands to submit jobs to workers. If the end conditions have not been met (*done* still equal to 0), then this loop will execute as long as the number of active jobs is less than the number of workers allotted for this session. Each time this loop is executed, another job is submitted to a worker.

Let's take a look at the insides of sub-loop #2. First, the *createJob* command creates a job object *job* within job manager *jm1*. Second, the next noise seed is retrieved from the noise state table. Third, the *createTask* command creates a task in the job that will evaluate the waveform function using the retrieved noise seed. The first three parameters passed in to *createTask* are the job object, a pointer to the waveform evaluation function, and the number of return arguments expected. The variables in parentheses are the variables that will be passed on to the waveform function which executes on the worker. Finally, the job is submitted to the job manager using the *submit* command, and the run queue is updated to include this job.

**wf_eval()** – This is the waveform evaluation function which runs on each worker. For our example, this function evaluates the performance of the IS-95 forward channel for ***L*** packets (which is the simulation block size). The main inputs to this function are the necessary model parameters, the noise seed, and the number of packets to simulate. The output is the number of bit and symbol errors. The output is returned to dc_sim() via the job manager. Note that wf_eval() does not contain any distributed computing toolbox commands, it only contains the necessary code to model the IS-95 forward channel.

There is some additional complexity required in order to execute a Simulink model from the waveform evaluation function. The *sim_set*, *load_system*, and *sim* commands are required to execute a Simulink model from the wf_eval() MATLAB script.

**Implementation Summary** – The top-level driver first calls dc_sim() to load the noise state table, and then calls dc_sim() for each Eb/No point to be evaluated. The top-level driver passes in all of the model and simulation parameters to dc_sim(), including a pointer to the waveform evaluation function to be simulated. When writing the top-level driver, the engineer doesn't need to know about DCT; the main constraint is interfacing correctly to the dc_sim() script.

When dc_sim() is executed, it monitors its internal run queue and strives to keep the run queue loaded with ***k*** jobs. Each job contains the information necessary for a worker to evaluate one simulation block. Function dc_sim() will continue to submit jobs and record results in the results queue until the target end conditions have been reached; it then returns the accumulated error counts to the top-level driver. Finally, the top-level driver calculates the BER or SER.

## 6. PERFORMANCE RESULTS

A MATLAB script which models a forward channel in the IS-95 system has been modified to work with dc_sim(). The original model was written by Gennady Zilberman of Ben-Gurion University in Israel and was downloaded from the MATLAB central file exchange. The modified model was used to characterize the performance of the cluster.

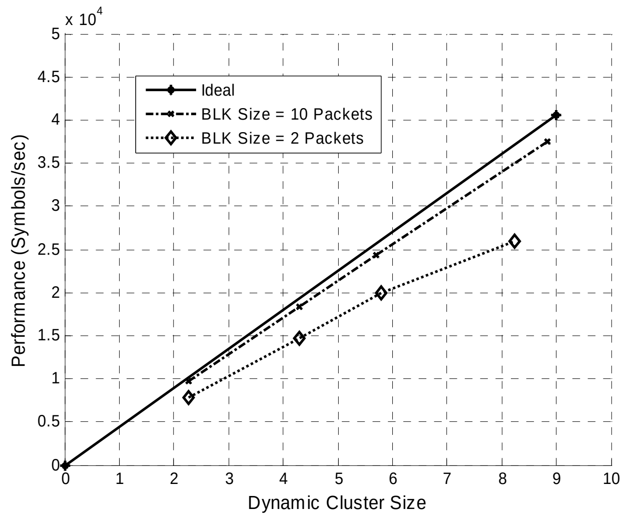

**Figure 5: Cluster Performance Results**

Perhaps the most important characteristic of the cluster to evaluate is its efficiency. By efficiency, we mean how closely the actual performance of the cluster matches the ideal performance. For example, if the computing resources of the cluster are doubled, what is the performance increase of the cluster? Ideally, the performance of the cluster would also double.

The efficiency of the cluster was measured by repeatedly simulating a fixed number of packets while varying the number of workers allotted to the cluster. Figure 5 shows the results of this test. The solid line represents the ideal performance of the cluster; the two other curves show the cluster performance for simulation block sizes of 2 packets and of 10 packets.

Note that the efficiency of the cluster is a direct function of the simulation block size; as the simulation block size increases, the efficiency of the cluster improves. This result is significant – if the block size is large enough, the cluster performance is fairly close to ideal.

Upon reflection, this result makes intuitive sense. For larger block sizes, the workers spend more time processing simulation blocks, and less time communicating with the DCT client. Remember that the workers run semi-independently, only interacting with the job manager when they need new input or when they are returning results. In our IS-95 model, each packet takes about 5 seconds to simulate. Thus a block size of 10 packets means that each worker runs independently for about 50 seconds between interactions with the DCT client. If there are 10 computers in the cluster, the cluster would then be providing feedback to the DCT client every 5 seconds.

## 7. COST BENEFIT ANALYSIS

The performance results look fairly promising; the next step would be to perform a cost benefit analysis. In this section we take a brief look at some of the issues involved in such an analysis, although a complete analysis is not presented – the material presented here is only meant to be a starting point.

The purpose of a cost benefit analysis would be to quantify the benefit of the MATLAB DCT software, and compare this to the software cost. In this section we only discuss the issues on the benefits side; a more complete analysis will be done internal to the company. For this discussion we assume a computer cluster of 16 computers running the MATLAB DCT software, resulting in a speed improvement of 10x to 15x.

A MATLAB computer cluster would benefit a waveform development project in two ways. First, the main benefit would be a direct reduction in the system engineering hours required to create a functional MATLAB/Simulink model of the waveform. Since the system engineer gets his/her simulation results faster, the waveform model can be completed faster. Of course, there is not a 1-to-1 relationship between reduced simulation times and reduction in overall system engineering hours - a speed improvement of 10x does not result in a reduction of 10x in hours allotted for system engineering activities. The system engineer works on a variety of tasks, only some of them are related to waveform simulation. Discussions with several project leaders and system engineers resulted in an estimate of 2.5% for the reduction in overall system engineering hours. Specifically, the team estimated that a speed improvement of 10x to 15x results in a 2.5% reduction in required system engineering hours to develop a system waveform model. Table 1 below shows, for 3 recent projects within our company, the estimated hours that could have been saved using the MATLAB DCT software.

**Table 1: Estimate of Hours Saved**

| Project | SYS ENG HRS for WF Development | SYS ENG HRS Saved |
|---|---|---|
| A | 2352 | 59 |
| B | 3696 | 92 |
| C | 2350 | 59 |

There is also arguably a secondary benefit to employing a MATLAB computer cluster. Since the system engineering tasks occur on the front end of a project, they impact all of the subsequent tasks. The final system model is needed by the FPGA, hardware and software engineers to do their job. So improvements in up front system engineering will benefit the entire development program by allowing the system engineer to perform various "what if" scenarios prior to having them implemented by the engineering team. A final point to consider is that if a MATLAB computer cluster is purchased it can be shared between multiple programs to further defer the project unique cost.

## 8. CONCLUSIONS

This paper has presented an approach for significantly accelerating the simulation of communication waveforms using the MATLAB Distributed Computing Toolbox. It has been shown that, for reasonable simulation block sizes, a parallel Monte Carlo waveform simulation efficiently utilizes computer resources - doubling the size of the computer cluster can come close to doubling the performance of the cluster. Performance of the cluster was verified using a MATLAB (script-based) model. A Simulink model is currently being modified to confirm that the proposed approach also works with Simulink (model-based) simulations.

So far, work has focused on creation and implementation of the PMC algorithm; further work is needed to validate the simulation results of the modified models. At General Dynamics C4 Systems this approach has gone through an initial evaluation; we believe that it provides justifiable benefits to programs to offset its initial investment.

## 9. REFERENCES

[1] J.S. Kim, S.J. Byun, "A Parallel Monte Carlo Simulation on Cluster Systems for Financial Derivatives Pricing", *The 2005 IEEE Congress on Evolutionary Computation*, Vol. 2, pp. 1040-1044, 2-5 Sept. 2005.

[2] Y.K. Dewaraja et al., "A Parallel Monte Carlo Code for Planar and SPECT Imaging …", *The 2005 IEEE Nuclear Science Symposium Conference Record*, Vol. 3, 15-20 Oct. 2005.

[3] *Distributed Computing Toolbox User's Guide*, Version 2, The Mathworks.

[4] P.D. Coddington, "Random Number Generators for Parallel Computers", Syracuse University, pp. 2, April 28 1997.

## 10. ACKNOWLEDGEMENTS

Special thanks to Stuart Williams and Dave Harrison of General Dynamics C4 Systems for supporting this effort. Also, thanks to David Orenstein and Idin Motedayen-Aval of The Mathworks for licensing and technical assistance.