THE MATHWORKS DISTRIBUTED AND PARALLEL COMPUTING TOOLS FOR SIGNAL PROCESSING APPLICATIONS

Ali Behboodian, Silvina Grad-Freilich, Grant Martin

The MathWorks, Inc.
3 Apple Hill Drive, Natick, MA, 01760 USA

ABSTRACT

As requirements for technical computing applications become more complex, engineers and scientists must solve problems of increasing computational intensity that frequently outstrip the capability of their own computers. Some are distributed applications (also called coarse-grained or embarrassingly parallel applications), where the same algorithm is independently executed over and over on different input parameters. Others consist of parallel (or fine-grained) applications, which contain interdependent tasks that exchange data during the execution of the application. This article introduces the distributed and parallel computing capabilities in The MathWorks distributed computing tools and provides examples of how these capabilities are applied to signal processing applications.

Index Terms — Distributed algorithms, parallel processing, computation time, distributed computing

1. INTRODUCTION TO THE MATHWORKS DISTRIBUTED COMPUTING TOOLS

Technical computing applications demand increasing computing resources as the complexity of algorithms and the size of data sets continue to grow. Engineers and scientists who have traditionally run problems on local computers are experiencing unacceptably long run-times or are unable to fit an entire data set into their computer's memory. Commercial off-the-shelf (COTS) computer clusters and multiprocessor, multicore computers now provide affordable high-performance computing environments that help technical computing users to address these limitations. However, taking advantage of distributed environments requires the user to divide up a problem so that different processors can simultaneously work on separate parts of the problem.

Some problems are called distributed or coarse-grained because they can be segmented easily to run on several nodes without communication, shared data, or synchronization points between the nodes. Parameter sweeps and Monte Carlo simulations are two examples of applications that often fall into this category. Other applications, called parallel or fine-grained applications, are far more difficult to segment. These applications are difficult to run on local computers because they involve more data than can fit into the computer's memory, and this data cannot be broken up easily into independent pieces. Even when a large set of data can be squeezed into available memory, there is rarely enough memory left over to do any significant computation. The solution is to have multiple processors working on distinct portions of the complete data set with a considerable amount of interim communication among the processors.

Figure 1: Distributed computing configuration.

The MathWorks has responded to this need by providing distributed computing tools that make it possible to develop distributed and parallel MATLAB® applications interactively and to distribute complete Simulink® models for execution in a cluster or in a multicore or multiprocessor computer.

Users can now prototype applications in MATLAB and use Distributed Computing Toolbox functions to define jobs that are made up of a series of independent or

interdependent tasks. A *task* is simply a unit operation, such as a MATLAB function or complete Simulink model.

MATLAB Distributed Computing Engine schedules and executes the job on available *workers*, which are MATLAB processes running on a cluster. In turn, each worker executes a task within a job by calling the specific MATLAB function specified by the task, passing the appropriate input data to the function and producing a result. The results are then made available for retrieval.

The distributed computing tools allow users to develop distributed or parallel applications using the full MATLAB language and most toolboxes. Each worker requires only the MATLAB engine license; separate toolbox and blockset licenses are not required. Workers can execute jobs that use any eligible MATLAB toolboxes and Simulink blocksets for which users are licensed in their computers.

2. DISTRIBUTED COMPUTING APPLICATIONS

Distributed applications that can be segmented into different independent tasks can be very easily distributed to cluster nodes. In the simplest case, where the problem can be divided into tasks consisting of the same function with the same number of input and output variables, a single function call parallelizes the problem for a distributed computing environment.

```
>> result = dfeval(@myFunction, {1 2 3 4 5})
```

In more complex cases, only several lines of code are required. A typical Distributed Computing Toolbox client session includes the following steps:

1. Find a job manager or scheduler.
2. Create a job.
3. Create tasks.
4. Submit a job to the job queue.
5. Retrieve the job's results.

2.1. Application of distributed computing to bit error rate calculation in MATLAB

A typical example of a distributed application is the calculation of the bit error rate (BER) in a communications system under different Signal to Noise Ratios (SNR). The BER is the percentage of bits that are received in error at a receiver relative to the total number of bits received in a communications system. Calculation of BER usually takes a relatively long time for high SNR condition. As a result, repetitive simulations of different receiver structures can be quite hectic.

An example that shows the advantage of distributed computing over single machines for calculation of BER is included in the demos that ship with Distributed Computing Toolbox. This demo simulates four different equalizers: a linear equalizer, a decision feedback equalizer (DFE), and a maximum likelihood sequence estimation (MLSE) equalizer. The MLSE equalizer is first invoked with perfect channel knowledge, then with a straightforward but imperfect channel estimation technique. We choose to let one task consist of calculating the BER for a single equalizer type and a range of SNRs. Because our cluster has four workers, we have four different tasks, one for each equalizer.

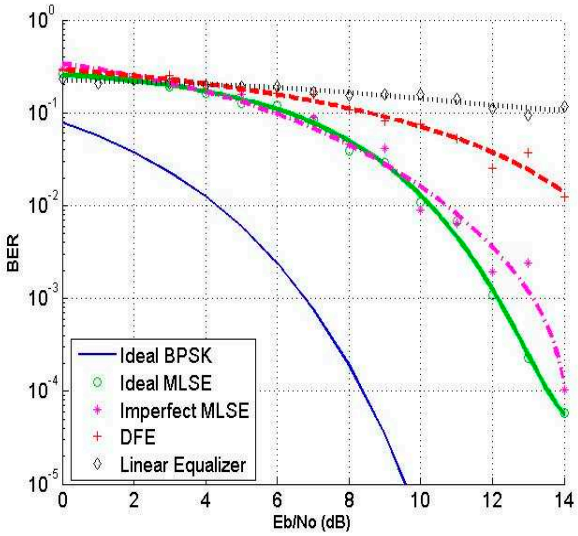

Figure 2: BER comparison.

The results are depicted in Figure 2. This simulation runs in 81 seconds on a single node of our cluster and in only 25 seconds on all the four nodes using distributed computing. In this example our cluster consists of four machines that are connected together to create a cluster.

3. PARALLEL COMPUTING APPLICATIONS

Now let's look at the more challenging task of programming parallel applications in which the same instruction set may operate over a large data set spread across *labs* (workers participating in a parallel computation). The MathWorks distributed computing tools support two alternative ways of prototyping parallel applications: message passing constructs and global array semantics.

3.1. Message passing constructs

The distributed computing tools provide functions for inter-task communication. Based on the Message Passing Interface (MPI) standard, these functions are available for

point-to-point and broadcast communications. They give users explicit control over the parallelization scheme for their applications.

However, programming with MPI also requires users to pay attention to low-level details such as distributing data in a way to minimize communication, determining which processor data needs to be delivered to or received from, etc. This level of detail is not present in serial programming and makes the transition from serial to parallel programming very difficult.

3.2. Parallel `for` loops and global array semantics

Distributed Computing Toolbox and MATLAB Distributed Computing Engine provide support for parallel `for` loops and global array semantics via distributed arrays. Distributed arrays store segments of an array on participating labs and are visible as regular arrays on all the labs. Each lab has its own array segment to work on yet has access to all segments of the array. As a result, users can perform array operations such as indexing, matrix multiplication, decomposition, and transforms directly on the arrays. With version 3.0, the distributed computing tools support more than 150 overloaded MATLAB functions for distributed arrays, including linear algebra routines based on ScaLAPACK.

Distributed arrays enable users to develop parallel applications without having to manage the low-level details of message passing. Let's suppose that a user wants to transpose a large matrix that is distributed in many processors. Using MPI, this would require a tedious, error-prone programming task. With the new parallel computing capabilities, programmers simply use the transpose function on a distributed array:

```
>> E = D'
```

A MATLAB user views a distributed array as a single global array rather than multiple, independent arrays located on separate processors. The ability to view related data distributed across processors as a single array closely matches the serial programming model and makes parallel programming much easier. In general, few changes are required to convert the MATLAB serial code to parallel. The programmer doesn't have to worry about the low-level programming details because communication takes place automatically and arrays are automatically redistributed when necessary. Reducing the size of the array that each lab has to store and process enables more efficient use of memory and faster processing, especially for large data sets. Access to the local segment is faster than to a remote segment, because the latter requires sending and receiving data between labs and thus takes more time.

Parallel `for` loops or `parfor` commands let users distribute similar but independent tasks over a set of labs. Each lab operates on a subset of the loop indices. This command eliminates the need to manually create and submit jobs and retrieve results. It uses the familiar `for` loop syntax in MATLAB and is ideal for parameter sweeps and similar tasks. No communication can occur between workers during the execution of the loop.

It is readily clear that parallel computing requires a certain amount of overhead. As such, parallel computing is not efficient on small array sizes. (Small is a relative term and it depends on the processing power and memory size of a machine.) Figure 3 depicts the result of executing two-dimensional FFT on matrices of different sizes. We apply an FFT of size 2^L to the rows and then columns of a matrix of size $2^L x 2^L$ where L varies between 5 to 13. As the size of the matrix grows, the required processing time increases. However it is not until the size of the matrix grows to $2^{-10} x 2^{10}$ that parallel computing pays off. The machine used in this experiment was a quad processor 32-bit Windows machine with 8 GB of RAM.

3.3. Interactive execution

The interactive parallel mode (or `pmode`) of MATLAB makes it possible to work interactively with a parallel job running simultaneously on several labs. Commands typed at the `pmode` prompt are executed on all labs at the same time. Each lab executes the commands in its own workspace on its own variables. Results are returned immediately to the command window. The `pmode` makes it possible for users to follow the normal MATLAB workflow with parallel algorithms and can be used for iterative exploration, design, and problem solving.

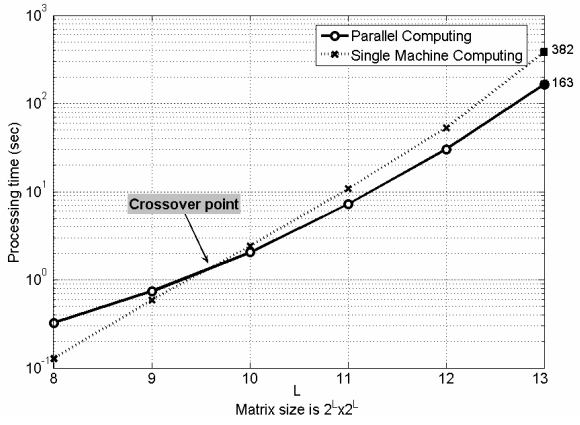

Figure 3: Comparison of processing time for computing 2D FFT between single machine and a cluster.

3.4. Application of parallel computing to synthetic aperture radar (SAR) imaging.

Synthetic aperture radar (SAR) image formation is a typical parallel application. For efficient digital processing and image formation, engineers often use a traditional technique known as polar format processing to reformat SAR phase history data from a polar grid to a Cartesian grid. This polar-to-rectangular "re-gridding" operation is the most processing-intensive portion in SAR image formation. The 2-D operations involved are not easily separable, nor are they independent such that coarse-grained techniques can be used.

For this particular example, the SAR image formation algorithm used is based on the Chirp Z-Transform (CZT). This method offer advantages in image quality because it uses the entire frequency spectrum instead of a zero-padded rectangle or inscribed rectangle. Also, the CZT does not degrade the image with inexact interpolations because there are no interpolations required to resample the grid. The shortcoming of the CZT, however, is that it requires three FFTs of two times the length of the input sequence, and also two complex multiplies of that same length. This can be more inefficient than an interpolation and requires significantly more memory to implement.

To test the MATLAB based CZT image formation, we used phase history data, which is approximately 2000 by 4000 samples of complex, double-precision data (120 MB). This size image can be read into MATLAB, but processing is difficult because of memory concerns. This processing limitation is overcome by using the distributed array function to split the image into four sections which will be operated on in parallel. The Chirp Z-Transform is then performed along the rows of the matrix, followed by an FFT along the columns to complete the image formation process. If we distribute the array column-wise, the CZT will require communication among the labs, and the final FFT can be computed in parallel. Equivalently, we can distribute the array row-wise such that the CZT can occur in parallel, while the final FFT requires communication across the labs. The pseudo code for this process follows:

```
im = LoadImage();

% Create a distributed array
im_distrib = distribute(im,rowdistrib);

% CZT setup
[A, W, m] = createCZTConstants();

% Azimuth Processing (Parallel)
parfor i = 1:numrows

    % Make local variables
    x = local(im_dist(i,:));

    % CZT is a Fast convolution
    y = fastconvCZT(x,A,W,m);

    im_dist(i,:) = y;
end

% Range Processing (Needs to Communicate)
ImageOut = fft(im_dist,[],1);
```

This pseudo code was implemented with both row and column distribution of the input image, and the difference between the time and memory required was found to be negligible. More noticeable was the difference between the Distributed Computing Toolbox implementation and the implementation on a single machine. The single machine required twice the amount of time to run (~9.45 sec) versus (4.95 sec) with the DCT. (The machine used was the quad-core machine deployed in the 2D-FFT example). These times are consistent with our 2-D FFT results and should also scale as the images become larger with finer resolutions. Also worth noting is that the amount of memory required to process the 120 MB image on the DCT was split onto four labs. This is important because as the initial data grows in size, this problem may become unfeasible on a single machine. Thus Distributed Computing Toolbox can provide a viable memory management solution.

4. CONCLUSION

Using the distributed computing tools, engineers and scientists can now develop signal processing applications in MATLAB and divide them into tasks that are evaluated remotely on cluster nodes. With the release of Version 3, the distributed computing tools also provide tools to interactively prototype, develop, and debug parallel signal processing applications using MATLAB. The tools can also distribute complete Simulink models for execution in a cluster. The tools can execute algorithms that include any MATLAB toolbox or Simulink blockset for which the user is licensed on the client machine; there is no need to purchase additional toolbox or blockset licenses for the cluster nodes. Because only a few changes are required to convert a serial program to a parallel one, engineers and scientists without special programming expertise will now be able to fulfill the promise of parallel computing by solving larger problems in less time.

5. REFERENCES

[1] Hahn Kim and Julia Mullen, "Introduction to Parallel Programming and pMatlab v0.7," MIT Lincoln Laboratory.

[2] Grant Martin, Armin Doerry, "SAR Polar Format Implementation with MATLAB," Sandia National Laboratories Report SAND2005-7413, November 2005.