

## MIT Open Access Articles

*Rapid prototyping of radar algorithms [Applications Corner]*

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

**Citation:** Reuther, A.I., and J. Kepner. "Rapid prototyping of radar algorithms [Applications Corner]." Signal Processing Magazine, IEEE 26.6 (2009): 158-162. © Copyright 2009 IEEE

**As Published:** <http://dx.doi.org/10.1109/MSP.2009.934178>

**Publisher:** Institute of Electrical and Electronics Engineers

**Persistent URL:** <http://hdl.handle.net/1721.1/60052>

**Version:** Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

**Terms of Use:** Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.



## Rapid Prototyping of Radar Algorithms

**R**apid prototyping of advanced signal processing algorithms is critical to developing new radars. Signal processing engineers usually use high-level languages like MATLAB, IDL, or Python to develop advanced algorithms and to determine the optimal parameters for these algorithms. Many of these algorithms have very long execution times due to computational complexity and/or very large data sets, which hinders an efficient engineering development workflow. That is, signal processing engineers must wait hours, or even days, to get the results of the current algorithm, parameters, and data set before making changes and refinements for the next iteration. In the meantime, the engineer may have thought of several more permutations that he or she wants to test.

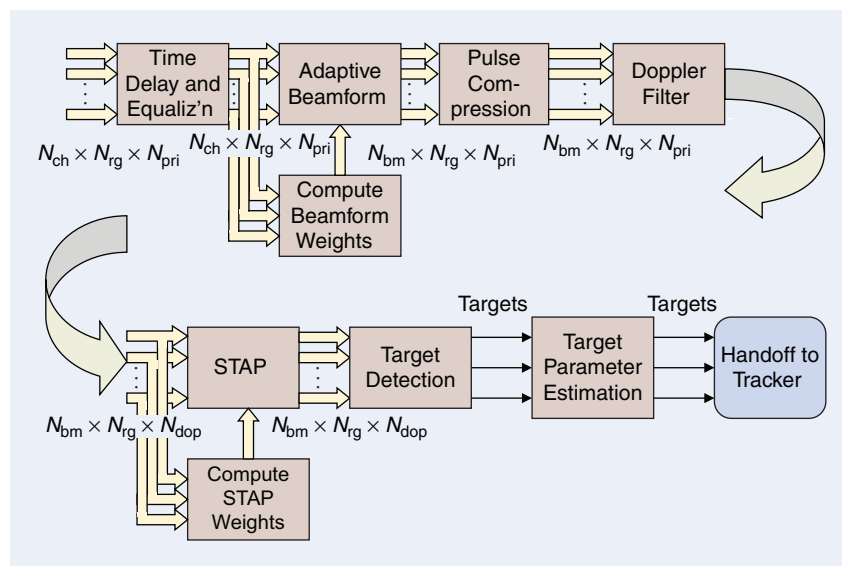
Ideally, such core algorithm development requires interactive, on-demand, high-performance computing using high-level programming languages and tools. The environment would enable parallel programming in the same language in which the original serial code was written. Also, the environment should provide a low barrier to entry and functionality similar to the environment with which the signal processing engineer is already familiar: the environment should be interactive and provide immediate execution of the algorithm, rather than having jobs wait for execution in a batch queue. Finally, the high-performance rapid prototyping environment should provide a logical transition path for the algorithm code from the development workstation to deployment in an embedded processing sys-

tem. In this article, an experimental radar processing chain is used to describe how parallel MATLAB can enable a software design workflow that satisfies these requirements.

### EXAMPLE APPLICATION: GROUND MOVING TARGET INDICATOR RADAR PROCESSING

We shall use the development of a ground moving target indicator (GMTI) [1] testbed radar processing system that was developed at MIT Lincoln Laboratory [2] as a representative application that helped drive the development of the parallel MATLAB suite. GMTI uses two interference-nulling stages to extract targets that are moving relative to the background noise. The Doppler return of these targets is exploited to differentiate them from surrounding interference sources and ground clutter interference. In a radar system, multiple transmitters emit an outgoing signal pulse that is reflected

off of various objects including buildings and other structures, ground, and entities of interest. Upon returning to the radar, the receiver array captures the reflected signal pulses, combines multiple received signals into a handful of signals, and digitizes them using analog-to-digital converters. It is at this point that the array of signals enters the digital signal processing chain, which is depicted in Figure 1. Each transmit-receive cycle produces the data of a pulse repetition interval (PRI), and a series of PRIs produces a coherent processing interval (CPI). Each CPI is organized into a radar data cube, a three-dimensional tensor whose dimensions are receive channel, range gate, and PRI; the notation beneath each stage transition in the figure denotes the size of the data cube that is passed between the stages. Each of the subsequent stages in the signal processing chain refine the data in the data cube to locate and extract moving



**[FIG1]** A typical narrowband GMTI processing chain.

targets. The functionality and computations of each of the stages follows:

- The time delay and equalization stage time aligns and equalizes the incoming signal to compensate for differences in the transfer function between channel sensors. Computationally, time delay and equalization is implemented using finite impulse response (FIR) filters: fast Fourier transforms (FFTs) along the range dimension, multiplication, and inverse FFTs (IFFTs) of the resulting range vectors.

- The adaptive beamforming stage transforms the filtered data from the channel space (first dimension of the radar data cube) into the beam-space domain to allow for detection of target signals coming from a particular set of directions of interest while filtering out spatially localized interference. For each narrowband data cube, a covariance matrix is computed, and its computation is dominated by a matrix inverse computation. This covariance matrix is then matrix-multiplied with each of the PRI-by-range-gate matrices in the data cube.

- The pulse compression stage filters the data to concentrate the signal energy of a relatively long transmitted radar pulse into a relatively short pulse response. Computationally, pulse compression is also implemented using FIR filters: FFTs along the range dimension, multiplication with the pulse filter, and IFFTs of the resulting range vectors.

- The Doppler filter stage processes the data so that the radial velocity of targets relative to the platform can be determined. This is accomplished by computing an FFT along for each vector of the PRI dimension.

- The space-time adaptive processing (STAP) stage is a second beam-forming stage that removes further interference and ground clutter interference. Determining the interference-nulling covariance matrix again is dominated by a matrix inverse calculation, while using the covariance matrix on the data cube

involves matrix-matrix multiplies of the covariance matrix and each of the matrices in the channel/beam-range dimensions of the radar data cube.

- The detection stage uses constant false-alarm rate (CFAR) detection to compare a radar signal response to its surrounding signal responses to determine whether a target is present and uses target grouping to eliminate multiple target reports that are actually just one target.

## THE DEVELOPMENT OF THE RADAR SYSTEM ALGORITHM IS JUST THE FIRST STAGE IN DESIGNING AN EMBEDDED RADAR PROCESSING SYSTEM.

- The estimation stage estimates target positions within the sample bins to pass to the tracking algorithm. It usually involves several spline interpolations for each target that was identified.

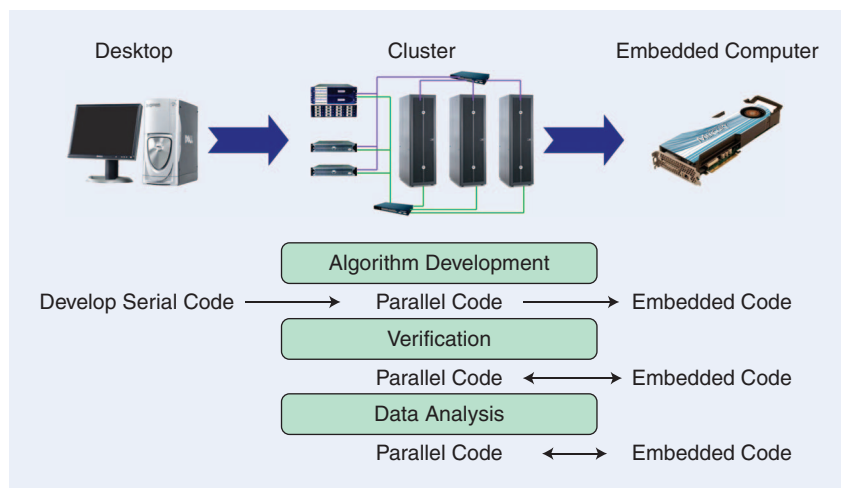
Finally, the target report is passed to the tracking algorithms. Executing this signal processing chain on a typical radar data cube (4–8 channels, 10,000 range gates, and 16–128 PRIs) will take a significant amount of time. Running it in serial MATLAB on a modern Intel-based workstation can take between 5–15 min for each radar data cube. Processing any reasonable stream of data cubes takes processing time into the hours.

## SOFTWARE DESIGN WORKFLOW

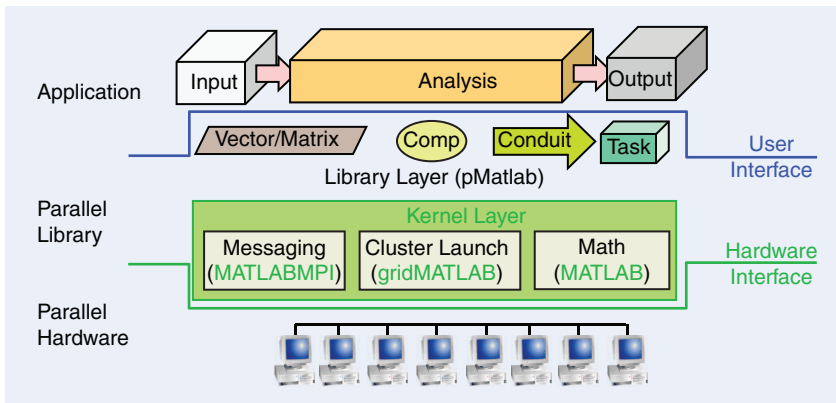
The development of the radar system algorithm is just the first stage in designing an embedded radar processing system. Figure 2 depicts a typical software development workflow that requires parallel embedded code due to the amount of computation that is required [3]. Early in the development, the serial code (usually in MATLAB) is translated to parallel C, C++, or Fortran. Once the parallel code has been validated with the serial code, it is transferred or translated into the embedded code. The two parallel code sets are then used for validation and data processing for the remainder of the development project. Parallel signal processing libraries like PVL [4] and parallel VSIP++ [5] can simplify the transition from parallel to embedded code, but translating the code from MATLAB to parallel C, C++, or Fortran often slows or even halts further algorithm development since it takes more time to change the C, C++, or Fortran codes. Ideally, translating the serial MATLAB code into parallel MATLAB would be a more effective way to explore the parallel design space while speeding up the execution of complex algorithms and enabling the processing of more representative, large data sets.

## PARALLEL MATLAB AND LLGRID

To enable a more effective software design workflow, we developed three



**[FIG2]** The parallel software development workflow.



**[FIG3]** The software stack of the pMatlab rapid prototyping environment.

technologies that enable algorithm developers to develop parallel MATLAB codes with minimal changes to their serial codes and enable them to run parallel MATLAB jobs transparently on the LLGrid On-Demand Grid Computing system [6]. The three toolboxes that we developed are:

- MatlabMPI for point-to-point messaging (available at <http://www.LL.mit.edu/MatlabMPI>)
- pMatlab [7] for global array semantics (similar to High Performance Fortran) (available at <http://www.LL.mit.edu/pMatlab> and includes MatlabMPI)
- gridMatlab for integrating user's computers into the LLGrid and automatically allocating grid computing resources.

These technologies have combined to create a unique on-demand grid-computing experience, whereby running a parallel MATLAB job on the grid is identical to running MATLAB on the desktop. Users can run their parallel MATLAB jobs from their

desktops on Windows, Linux, Solaris, and Mac OS X computers. The users launch their jobs onto the flagship LLGrid cluster, named TX-2500 (Note: TX-0 was developed at Lincoln in the 1950s as the world's first interactive high performance computing system), which consists of 432 compute nodes connected with a high speed network. TX-2500 and user desktop computers all mount a large shared central parallel file system.

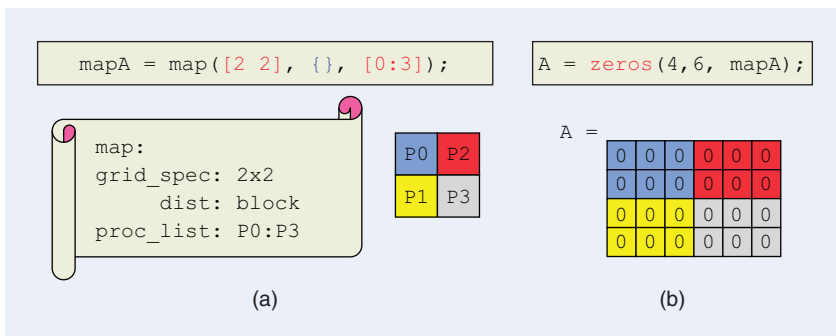
Figure 3 depicts the software stack for pMatlab applications. The kernel layer is comprised of MATLAB along with its many toolboxes, MatlabMPI for messaging, and gridMatlab for launching onto clusters. MatlabMPI consists of a set of MATLAB scripts that implement a subset of MPI, allowing any MATLAB program to be run on a parallel computer. The key innovation of MatlabMPI is that it implements the widely used MPI "look and feel" on top of standard MATLAB file I/O, resulting in a "pure" MATLAB implementation that is exceedingly small (~300 lines of code). Thus,

MatlabMPI will run on any combination of computers that MATLAB supports. MatlabMPI uses a common file system as the communication fabric, which enables easier debugging of parallel program messaging and compatibility across any OS platform that runs MATLAB or Octave. The gridMatlab toolbox transparently integrates the MATLAB on each user's desktop with the shared grid cluster through a cluster resource manager like LSF, Torque, or Sun Grid Engine; when a MatlabMPI or pMatlab job is run by the user in his or her MATLAB session, gridMatlab automatically amasses the requested LLGrid computational resources from the shared grid resources to process in parallel with the user's MATLAB session. The gridMatlab toolbox interfaces with the underlying scheduling resource manager to run interactive, on-demand jobs.

While the cluster system and the MatlabMPI and gridMatlab toolboxes are integral to the functionality of the system, the pMatlab toolbox is the application programming interface with which algorithm designers write their parallel MATLAB codes as depicted in Figure 3. pMatlab is a partitioned global address space (PGAS) [8] toolbox, which allows MATLAB users to parallelize their program by changing a few lines of code. The two key concepts in pMatlab are parallel maps and distributed arrays [7].

## PARALLEL MAPS

Parallel maps specify how to allocate distributed arrays. Figure 4(a) depicts the contents of a parallel map. The grid field specifies the dimensions in which the parallel distribution is partitioned, while the distribution field determines whether the parallel distribution should be block, cyclic, or block cyclic. The default distribution is block. Finally the processor list specifies the job processor number over which the distribution is made. The example code line shows the map command for the given distribution, a  $2 \times 2$  matrix across processors P0, P1, P2, and P3, as depicted by the colored box in the figure.



**[FIG4]** Sample parallel map and distributed array.

## DISTRIBUTED ARRAYS

The default object in MATLAB is the array. Using parallel maps in an array constructor like `zeros`, `ones`, and `rand` as illustrated in Figure 4(b) instantiates a distributed array on each of the processors specified in the parallel map. In Figure 4(b), a  $4 \times 6$  matrix of zeros is instantiated on processors P0, P1, P2, and P3, using `mapA` created in Figure 4(a). Each of the four processors generates and stores the local portion of the distributed array along with information about the distribution of the array. This distributed array can then be used in mathematical operations that have been written to use (overloaded) with distributed array. For example, the `fft` function can be executed on the matrix *A* because the `fft` function has been overloaded in the pMatlab library to work with distributed arrays.

## PARALLELIZING GMTI

The greatest challenge of developing parallel software is determining where to parallelize the serial code so that it runs most effectively and efficiently. This determination is strongly influenced both by the algorithms that are being parallelized as well as the architecture of the parallel computing system on which the algorithms are to run. For instance, if the network connecting all of the computational nodes is relatively slow (both in latency and communication bandwidth), it would be detrimental to parallelize the algorithms in a way that requires a lot of communication.

■ Often the simplest solution for parallelizing algorithm code is to look for the largest loops and parallelize the code around those loops. This is sometimes called the embarrassingly parallel solution. For instance, in parameter sweep or Monte Carlo simulations, that loop is the one that selects each of the many scenarios. In the case of the GMTI example, this would be parallelizing on a data cube-by-data cube basis and having each processor work on its own data cube independently as depicted in Figure 5(a). However, par-

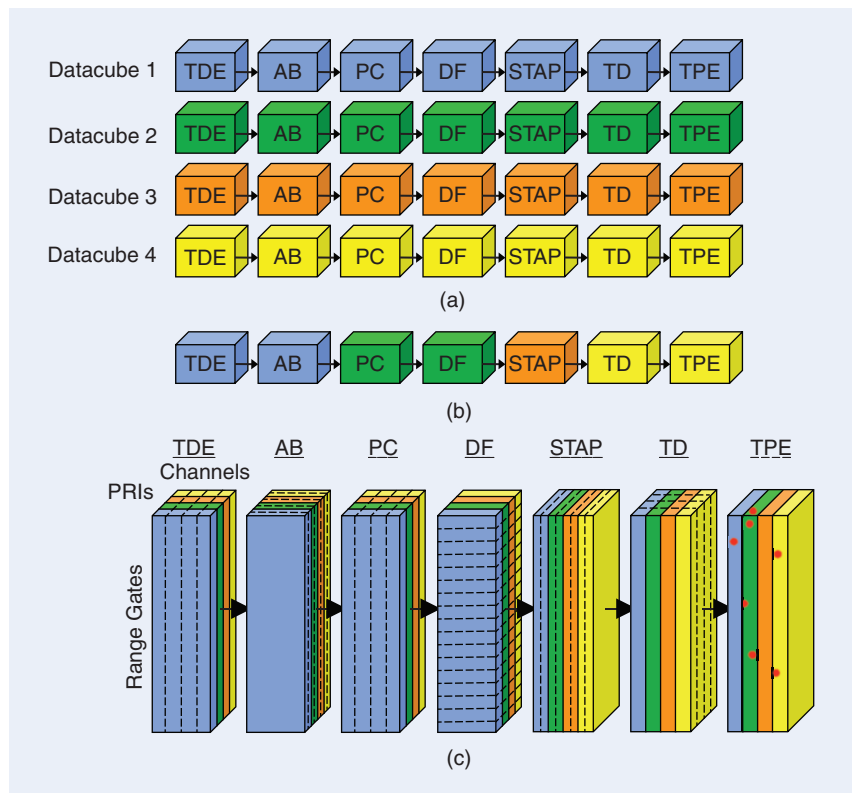
allelizing the largest loops usually incurs the most latency in receiving results; there is a price for the simplicity. To realize this solution, we define a parallel map that parallelizes a vector of loop indices across the CPUs executing the code. The map would look like this: `map1 = map([N 1 1], {}, [0:Np-1])`, where *Np* is the number of MATLAB instances running.

## THE GREATEST CHALLENGE OF DEVELOPING PARALLEL SOFTWARE IS DETERMINING WHERE TO PARALLELIZE THE SERIAL CODE SO THAT IT RUNS MOST EFFECTIVELY AND EFFICIENTLY.

■ To possibly reduce the latency of processing each data cube, we could pipeline the processing chain among the CPUs as illustrated in Figure 5(b). This would involve creating maps that assign one or more stages to sets of one or more CPUs. The maps would

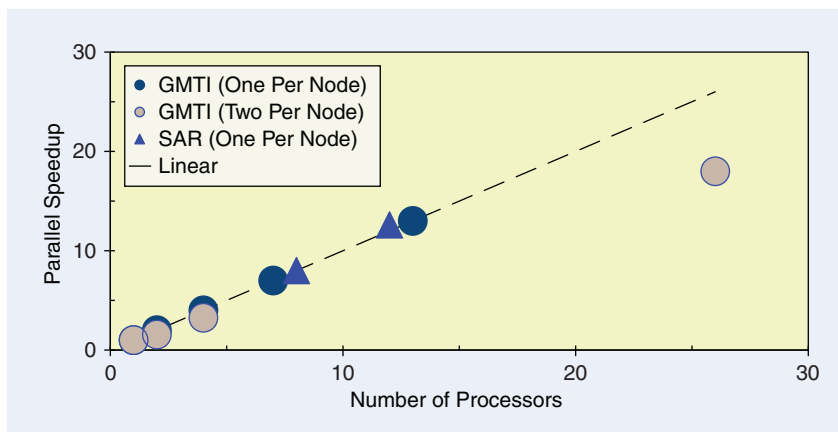
look like: `map1 = map([1 1 1], {}, [0]), map2 = map([1 1 1], {}, [1]), etc.` Then the distributed arrays are processed by the assigned CPUs and passed from CPU set to CPU set as processing continues. This solution would require low-latency, high-bandwidth communication channels between the CPUs. This parallel solution would also be effective if specialized hardware were being applied to one or more of the signal processing stages. For example, certain compute nodes had general purpose graphic processing units (GPGPUs) that were used for the stages that computed FFTs, namely the time delay and equalization, pulse compression, and Doppler filter stages.

■ Alternately, the CPUs could process one radar data cube at a time by parallelizing each of the stages across all of the CPUs as depicted in Figure 5(c). This would involve two maps: one in which the radar data cube is distributed across PRIs, `map1 = map([1 1 N], {}, [0:Np-1])` and the other in



**[FIG5]** Three example parallelization options for a GMTI signal processing chain.





**[FIG6]** Parallel performance of GMTI and SAR processing for an in-flight experimental radar processing system.

which the radar data cube is distributed across channels/beams, `map2 = map([N 1 1], {}, [0:Np-1])`. Between the Doppler filter and STAP stages, a data reorganization occurs. This reorganization is necessary to keep all of the kernel operations local to each processor; that is, no communication is incurred to compute the stage computational kernel as mentioned in the processing chain stage descriptions above.

For most pMatlab applications, we initially target the embarrassingly parallel solution. As the embedded target parallel solution starts to take shape, we will use pMatlab maps and distributed arrays to explore the effectiveness of various solutions before implementing them in the embedded solution.

## PERFORMANCE

The above technologies were used to implement GMTI and synthetic aperture radar (SAR) processing for an in-flight experimental radar processing system [2]. The speedup as a function of number of processors is shown in Figure 6. GMTI processing uses a simple round-robin approach and is able to achieve a speedup of 18x. SAR processing uses a more complex data parallel approach, which involves multiple redistributions and is able to achieve a speedup of 12x. In each case, the required detections and images are produced in under 5 min,

which is sufficient for in-flight action to be taken. Using parallel MATLAB on a

**WE HAVE SEEN A GREAT DEAL OF ENTHUSIASM FOR PARALLEL MATLAB RAPID ALGORITHM PROTOTYPING.**

cluster allows this capability to be deployed at lower cost in terms of hardware and software when compared to traditional approaches.

## OUTLOOK

We have seen a great deal of enthusiasm for parallel MATLAB rapid algorithm prototyping, both at conferences where we have presented and from the number of downloads of pMatlab and MatlabMPI on the Internet over the past several years. pMatlab and MatlabMPI are not the only solution available; both MathWorks' Parallel Computing Toolbox (PCT) [9] and Interactive SuperComputing's Star-P [10] have also developed strong interest and sales. The pMatlab toolbox, which includes MatlabMPI, is available as an open source software package at <http://www.LL.mit.edu/pMatlab/>. All three of these solutions are making parallel processing more accessible to algorithm development engineers and to the rapid-prototyping community in general. While determining how to parallelize a signal processing application usually requires experience and experimentation, some automation efforts for

domain-specific applications, such as pMapper for digital signal processing chains [11], show promise.

## ACKNOWLEDGMENTS

This work is sponsored by the United States Air Force under Air Force contract FA8721-05-C-0002.

## AUTHORS

**Albert I. Reuther** ([reuther@LL.mit.edu](mailto:reuther@LL.mit.edu)) is a technical staff member in the Embedded Digital Systems group at MIT Lincoln Laboratory. His research interests include distributed and parallel computing, digital signal processing, and numerical methods.

**Jeremy Kepner** ([kepner@LL.mit.edu](mailto:kepner@LL.mit.edu)) is a senior technical staff member in the Embedded Digital Systems group at MIT Lincoln Laboratory. His research focuses on the development of advanced libraries for the application of massively parallel computing to a variety of data intensive signal processing problems on which he has published many articles.

## REFERENCES

- [1] M. A. Richards, *Fundamentals of Radar Signal Processing*. New York: McGraw-Hill, 2005.
- [2] J. Kepner, T. Currie, B. Mathew, A. McCabe, M. Moore, D. Rabinkin, A. Reuther, A. Rhoades, L. Tella, and N. Travinin, "Deployment of SAR and GMTI signal processing on a Boeing 707 aircraft using pMatlab and a bladed Linux cluster," in *Proc. High Performance Embedded Computing (HPEC) Workshop*, Lexington, MA, 2004, pp. 25–26.
- [3] J. Bergmann and J. D. Oldham, "VSIPL++: A signal processing library scaling with Moore's law," in *Proc. High Performance Embedded Computing (HPEC) Workshop*, Lexington, MA, 2005, pp. 77–80.
- [4] J. Kepner and J. Lebak, "Software technologies for high-performance parallel signal processing," *Lincoln Lab. J.*, vol. 14, no. 2, 2003, pp. 181–198.
- [5] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge, "Parallel VSIPL++: An open standard software library for high-performance parallel signal processing," *Proc. IEEE*, vol. 93, Feb. 2005, pp. 313–330.
- [6] N. T. Bliss, R. Bond, J. Kepner, H. Kim, and A. Reuther, "Interactive grid computing at Lincoln Laboratory," *Lincoln Lab. J.*, vol. 16, no. 1, 2006, pp. 165–216.
- [7] J. Kepner, *Parallel MATLAB for Multicore and Multinode Computers*. Philadelphia, PA: SIAM, June 2009.
- [8] B. Carlson, T. El-Ghazawi, R. Numrich, and K. Yelick, "Programming in the partitioned global address space model," in *Proc. Tutorials at Super-Computing*, 2003.
- [9] A. Behboodian, S. Grad-Freilich, and G. Martin, "The Mathworks distributed and parallel computing tools for signal processing applications," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, Apr. 2007, vol. 4, pp. IV-1185–IV-1188.
- [10] R. Choy and A. Edelman, "Parallel matlab: Doing it right," *Proc. IEEE*, vol. 93, Feb. 2005, pp. 331–341.
- [11] N. T. Bliss, "Addressing the multicore trend with automatic parallelization," *Lincoln Lab. J.*, vol. 17, no. 1, pp. 187–198, 2007.