Interactive Grid Computing at Lincoln Laboratory

Nadya Travinin Bliss, Robert Bond, Jeremy Kepner, Hahn Kim, and Albert Reuther

■ The Lincoln Laboratory Grid (LLGrid) project was initiated to provide Laboratory staff with an effective way to exploit cluster computing as a solution to the demand for computational power in large-scale algorithm development, data analysis, and simulation tasks. Because sensor capabilities and demands continue to increase, the dataset sizes and algorithm complexities of today's challenging applications have outgrown the processing capabilities of single workstations. Cluster computing technology, where a networked set of workstations is used as a parallel processor, can provide the throughput and storage demands of these applications. Programming a cluster, however, requires algorithm developers to become parallel programmers, which is difficult, time-consuming, and distracting. To allow a large research community (who primarily use MATLAB) to exploit cluster computing, we have developed a parallel programming toolbox called pMatlab, which consists of a library of objects and routines for distributing numerical arrays onto multiple processors, and then carrying out parallel computations on these distributed arrays. A typical MATLAB programmer can use pMatlab to convert a program to a parallel implementation in a few hours or days, and can then run the application on a cluster. Since most Laboratory users do not have access to a cluster, the LLGrid enterprise cluster computing system was developed to provide users with desktop access to these resources. The LLGrid allows a pMatlab program to be run on a remote cluster as simply as it is to run a MATLAB program on the desktop. Several LLGrid satellite clusters dedicated to specific programs have also been established. To quantify the effectiveness of pMatlab on the LLGrid, we present pMatlab High Performance Computing Challenge benchmark results, which evaluate high-performance computing systems over a range of computations. Our results, including user experiences, illustrate the increased user productivity and high computational performance of pMatlab on the LLGrid. We conclude with the future directions of both the LLGrid project and related advanced software developments.

Since the earliest days of Lincoln Laboratory, innovation in interactive high-performance computing has played a central role in the Laboratory mission. Computers like the Whirlwind, the Memory Test Computer, the TX-0, and the TX-2 brought breakthroughs in simulation, sensor processing, computer architecture, system components, and human-computer interaction. These pioneering Laboratory technologies led to several spin-off computer companies, one of the most notable being Digital Equipment Corporation (DEC), the developer of the minicomputer. In the 1970s and early 1980s, minicomputers such as the DEC PDP- and VAX-series machines found extensive use at the Laboratory for applications ranging from simulation and data analysis to algorithm development and real-time processing. When even higher performance was needed, the Laboratory had access to the MIT supercomputing facilities.

Starting in the mid-1980s, the advent of the workstation and the personal computer allowed Lincoln Laboratory staff to carry out—at their desktops—computations that previously would have been relegated to shared minicomputers or even supercomputers. This convenient, interactive computing environment has proved to be revolutionary and has continued to greatly enhance the ability of Laboratory staff to carry out algorithm development and data analysis tasks. Today, desktop computers, be they Intel-based personal computers or any of the many high-end workstations from companies such as Sun or Hewlett-Packard, have become indispensable tools for scientists and engineers. Through Moore's Law, which has accurately predicted the exponential growth in processing in the last few decades, the typical Laboratory staff member has an enormous amount of dedicated computing power and storage capacity at his or her disposal. At the same time, interactive programming environments such as MATLAB have helped to increase individual productivity and prototyping capabilities enormously [1].

The situation is changing, however. The growth rate in computing power is beginning to decrease. The High Performance Embedded Computing (HPEC) Workshop at Lincoln Laboratory in 2004 explored some of the implications of this slowdown. Chip designers are facing fundamental difficulties in increasing microprocessor clock rates past the 4 to 5 GHz regime. The development of smaller-geometry fabrication processes has also slowed. Power management is becoming a critical concern, not just for embedded systems (where it has always been important), but for commercial server systems as well. The impact of this circuit-level crisis at the computer system level is that the improvement rate for single processor systems has slowed from a doubling rate of eighteen months to a doubling rate of about once every three years.

At the same time, the signal processing, simulation, and data processing tasks that Laboratory staff members are addressing have required a steady increase in computational power and digital storage. This increase can be traced to a few trends. First, sensors have been improving at a fast pace. For example, state-of-the-art charge-coupled device (CCD) arrays have grown from a few thousand to several million pixels in the last few decades. Sensor bandwidths have also increased dramatically. Synthetic aperture radars (SAR) operating at 1 to 2 GHz instantaneous bandwidth have been demonstrated. Ground moving-target indication (GMTI) radars for space-based and airborne surveillance systems are being developed that will have bandwidths approaching upwards of 1 GHz. Ultra-wideband analog-to-digital converters (ADC) operating at over 5 GHz are on the drawing table. This ADC technology is the harbinger of even wider-bandwidth radar systems to come.

In addition, sensor signal and image processing has become increasingly sophisticated. Advanced multidimensional signal processing algorithms are finding application in several areas, such as space-time adaptive processing for GMTI radars [2], and high-definition vector imaging (HDVI) for SAR systems [3]. Communication systems are employing advanced techniques such as adaptive processing and turbo decoding with iterative belief propagation [4]. To compound matters, sensor networking and networked decision support systems have emerged as important research areas. For these applications, not only do individual sensor streams need to be processed, but new multisensor data fusion, tracking, intelligence gathering, reasoning, and decision support algorithms are being prototyped in the lab and tested in the field.

The net effect of these different trends is that many of today's applications are dealing with orders of magnitude more data at all time scales, from high-bandwidth raw sensor data to data archived over days or even months. The computational needs for processing these large datasets have increased commensurately, and are now outstripping the capabilities of individual workstations.

To address these emerging requirements, cluster computer systems have begun to find use in many Lincoln Laboratory programs. A computing cluster is built by amassing a number of high-performance commodity computer servers, connecting them with a computer network, and managing them as one large entity rather than as individual computers. Often, high-capacity disks are included in the configuration. Most clusters, following in the footsteps of the original Beowulf clusters that were pioneered by C. Reschke et al. [5] at NASA, use a variant of the open-source Linux operating system. The main attraction of clusters is that they can be custom configured by using inexpensive hardware and open source software to economically deliver high computational performance. A less appealing aspect is that programming a cluster with conventional techniques requires a user to become a parallel programmer. This requirement is particularly cumbersome for scientists and engineers who are mostly interested in using highly productive development environments such as MATLAB.

Recognizing the programming challenge presented by clusters, the Embedded Digital Systems group began to explore ways to effectively use MATLAB on clusters. In 2002, the group developed the MatlabMPI toolbox [6], which provided MATLAB users with a way to parallelize their codes by using message passing based on the Message Passing Interface (MPI) standard [7]. This toolbox was quickly adopted both inside and outside the Laboratory. For MATLAB programmers willing to write explicit parallel codes, MatlabMPI has proven to be a powerful tool.

Not all MATLAB users, though, want to be parallel programmers. Recognizing this, in 2003 the Embedded Digital Systems group began the development of pMatlab [8], a parallel MATLAB toolbox that can be used to quickly convert serial MATLAB programs into parallel programs. MatlabMPI requires explicit, message-based, parallel code. pMatlab, on the other hand, uses a form of implicit parallelism borrowed from the real-time Parallel Vector Library (PVL) [9]. With pMatlab, the user defines distributed arrays and gives these arrays maps that specify how their data are spread across the nodes of a parallel computer. The arrays then coordinate data distribution, distributed computation, and synchronization, so that the MATLAB programmer does not need to explicitly deal with these parallel programming details. A MATLAB programmer can learn how to use the basic pMatlab toolbox library in a few hours. After that, converting a MATLAB code to a parallel code typically takes a few hours or at most a few days.

Out of the Beowulf cluster computing technology, the concept of grid computing has gained momentum [10]. Grid computing is an approach to providing computational resources to a user base, where the computers are located remotely but are as accessible and available as the electric power grid. Analogous to the electric grid, a computational grid should be: (1) highly reliable and accessible; (2) available for use anywhere in the Laboratory; and (3) easy to adopt and use. Connecting to the grid should be supported by a utility service and a user should be able to use the grid without having to change the way he or she works.

Grid computing allows users to share large-scale distributed and parallel computing resources to increase the throughput of computationally intensive programs. A key notion in grid computing is that computational resources should be provided to users 'on demand.' A user should not have to wait to get the resources to run a program. Moreover, the system should interact with the user to provide feedback on program status, and the turnaround time for program execution should be short enough to allow frequent, iterative code development. In short, using a cluster in an on-demand manner should be as much as possible like using a dedicated personal computer, except that programs requiring much greater computational power and storage could be handled in a timely manner.

In 2003, the Lincoln Laboratory Grid (LLGrid) computing team was formed in recognition that high-performance computing clusters were becoming very affordable, to the point that grid computing could be cost effective within the Laboratory. Numerous projects, such as NetSolve [11] and Legion [12], had already developed the software infrastructure to help make grid computing concepts practical. These grid computing technologies, however, tended to require steep learning curves. The grid systems they supported drew a strong distinction between the users' computers and the grid computing resources. Thus, while a user could get the quick response and high-performance benefits of on-demand computing, the human-computer interface was quite different and more difficult to use when interacting with the grid system. This issue of usability was one of the principal concerns of the Lincoln Laboratory grid team. As we explain later in this article, the team devised a novel way to make the user's computer part of the overall grid, thus making it much more natural to interact with the grid software infrastructure.

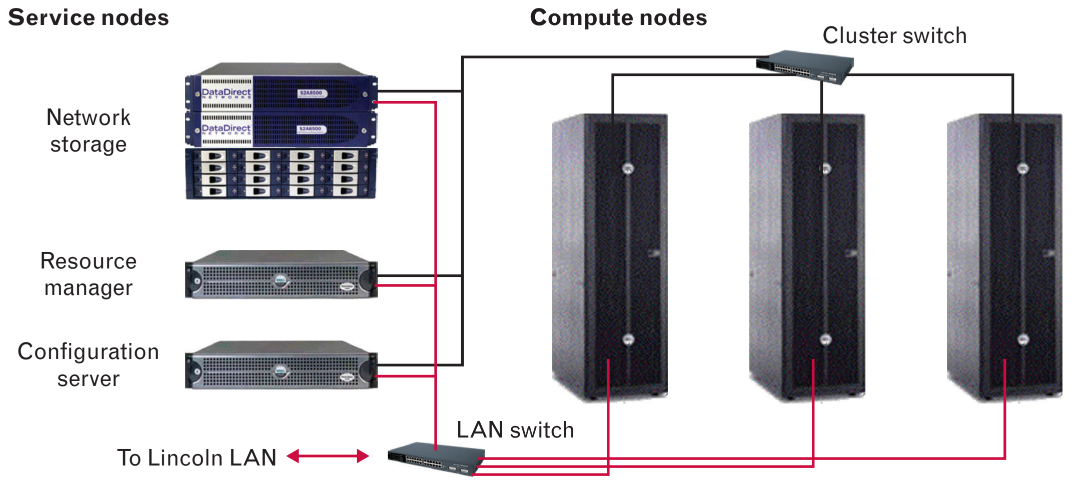


FIGURE 1. The Lincoln Laboratory Grid (LLGrid) system, consisting of Linux compute nodes and Linux service nodes. A user's computer is connected to the LLGrid via the Lincoln local area network (LLAN). The gridsan network storage delivers a common file system to the LLGrid cluster and all of the users. Currently, the LLGrid consists of 460 compute processors, for an aggregate peak throughput of 1.43 trillion floating point operations per second (teraflops), and 36 terabytes of storage capacity.

The team, which currently draws its membership from the Information and Communication Services group and the Embedded Digital Systems group, surveyed Lincoln Laboratory staff members to determine the user perspective on high-performance computing needs. The community reaffirmed that Laboratory researchers are unwilling to exploit high-performance computing resources if they are not interactive and they are difficult to use.

After reviewing the user survey, as well as available grid technologies, the LLGrid team focused on the following design targets for the LLGrid: (1) first-time user setup should be entirely automated and take less than ten minutes; (2) using MATLAB on the grid should be like running a MATLAB job on a personal computer; (3) the grid must be compatible with Windows, Linux, Solaris, and MacOS X operating systems, because of the diversity in the technical activities and computer platforms at the Laboratory; and (4) the grid should require only a few system administrators to maintain it.

The challenge of the LLGrid project was to develop an on-demand grid computing capability for high-performance computing that would satisfy these design targets. To enable easy MATLAB programming on a grid, the LLGrid team took the opportunity to match up the MatlabMPI toolbox with open-source resource management software, so that multiple grid users could simultaneously run MatlabMPI programs. This concept was later extended to include the pMatlab toolbox, and the resulting system, known as gridMatlab, now serves as the principal bridge connecting parallel MATLAB users to the grid.

Figure 1 illustrates the current version of LLGrid, which is housed in F Building at Lincoln Laboratory. The cluster is comprised of 460 compute and service processors. Users have access to the grid through their desktop personal computers across the Lincoln local area network (LLAN). The cluster also has a network storage system that provides up to 36 terabytes of storage for large-scale data analysis and simulations.

A significant percentage of Laboratory staff have used this system and have already demonstrated the value of the grid computing. Figure 2 shows a sampling of the Laboratory research and development programs that have benefited from using the LLGrid. The simulation codes developed in these programs have used thousands of hours of CPU time. It is fair to say that without the LLGrid, neither the size nor the quantity of simulations run by these codes would have been possible in the short time frame that today's research projects demand.

Grid computing research and development has been steadily on going since the original Lincoln

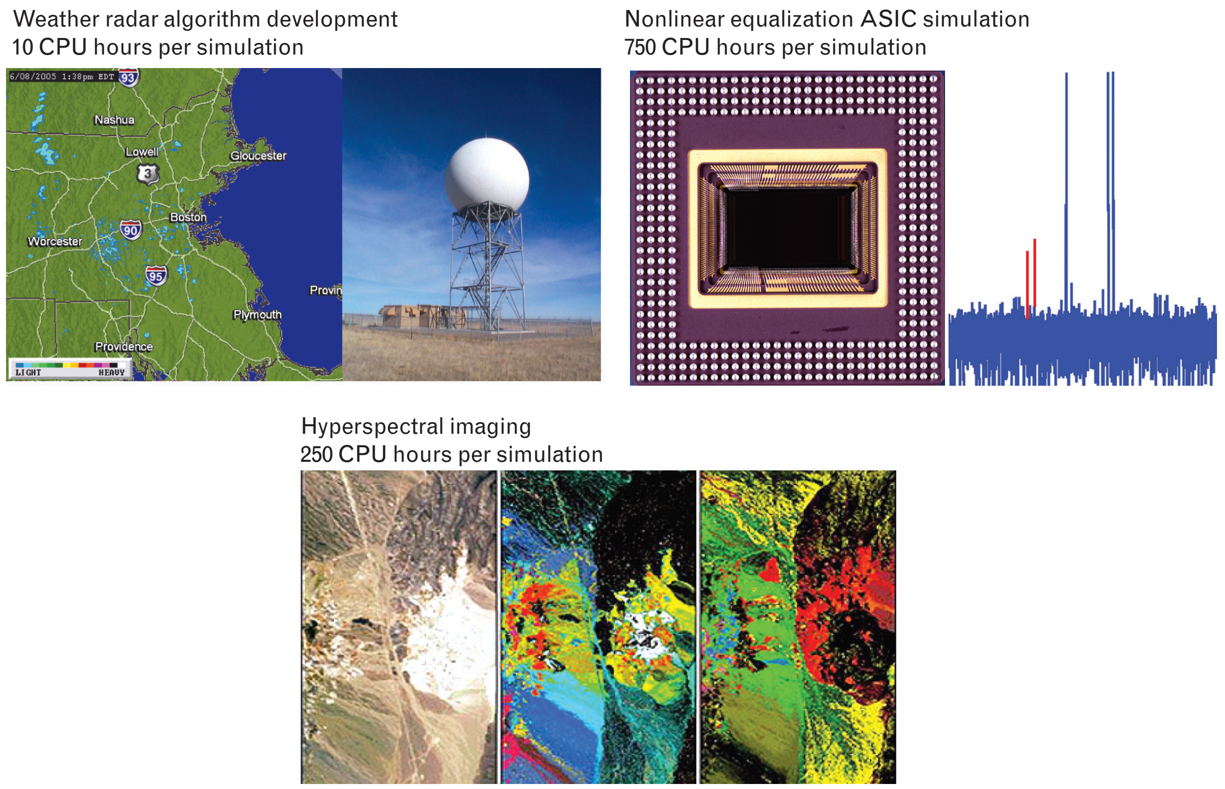

Weather radar algorithm development
10 CPU hours per simulation
Nashua
Lowell
Gloucester
Boston
Worcester
Plymouth
Providence
LIGHT
HEAVY
Nonlinear equalization ASIC simulation
750 CPU hours per simulation
Hyperspectral imaging
250 CPU hours per simulation

FIGURE 2. Examples of Lincoln Laboratory programs benefiting from the LLGrid. The Laboratory specializes in sensor signal processing, and much of the work involves gaining information and insight from a huge variety of sensors in a large number of research projects. This level of effort involves extraordinary amounts of computation capability. Among the wide variety of applications on which Laboratory engineers and scientists work are radar algorithm development for weather sensing, application-specific integrated circuit (ASIC) simulation, and hyperspectral imaging. Each example in the figure gives a rough estimate of the number of CPU hours that a typical simulation executes for each of the applications.

Laboratory α-Grid was launched in 2004. Much effort has focused on developing better scheduling and management tools. As these tools have matured and improved, the grid cluster has increased from 160 processors to its current size at 460 processors. Another major expansion to the LLGrid cluster is planned for 2007; this expansion will make it one of the hundred largest computers in the nation.

Improvements to the pMatlab toolbox have also been the focus of some innovative research and development. In the current version of pMatlab, the programmer must still specify how the arrays are to be mapped onto the parallel or distributed machine. This mapping task can be quite challenging, especially as codes become more complex. To address this, the pMatlab library is currently being augmented with automated mapping capabilities, so that the user merely has to specify which parts of the code to make parallel, and the library determines and builds a set of highly efficient parallel maps [13]. Another promising research area has been in the use of disk space as virtual memory for parallel MATLAB programs [14, 15]. By integrating out-of-core software techniques with parallel processing, extremely large datasets can be handled in a way that provides both high performance at large scales and programming ease of use.

The remainder of this article discusses the LLGrid project in more detail. First we describe the parallel MATLAB toolboxes—MatlabMPI and pMatlab—and we present benchmark results demonstrating the high performance of the parallelized programs. Then we discuss the LLGrid cluster, its development roadmap, and its software infrastructure. User experiences and lessons learned are a key focus of the discussion. Finally, we present some exciting future plans and research projects.

Parallel MATLAB Toolbox

MATLAB has emerged as one of the predominant languages of technical computing. Its popularity for data analysis, simulation, and modeling is largely due to the expressiveness of the language. Additionally, MATLAB provides its users—who tend to be engineers and scientists—with powerful graphics to visualize complex multidimensional datasets. The high-level language of MATLAB allows users to concentrate on their core competency and spend less effort on computer-science–related implementation details. It is common for scientists and engineers to test the validity of data processing algorithms or physical simulations by employing larger datasets, higher-resolution models, or a broader range of input parameters. This need for greater fidelity to physical reality causes execution times to reach hours or even days. Thus a parallel processing capability that provides good speed up in algorithm execution without sacrificing the ease of programming is highly beneficial. pMatlab seeks to provide this capability by implementing standard Parallel Global Array Semantics (PGAS)*, as illustrated in Figure 3. This figure shows the distinction between the pure PGAS approach and the fragmented PGAS approach, which is discussed in more detail later in the article.

The core data structures in MATLAB are arrays—vectors, matrices, and higher-order arrays. To keep the same functionality in a parallel programming paradigm, the core data structures in pMatlab are distributed arrays and maps, which are discussed in greater detail later. These data structures are illustrated in the pMatlab code fragment shown in Figure 4. The code fragment in this figure implements the STREAM benchmark [16], which is part of the High Performance Computing (HPC) Challenge benchmark suite, discussed in detail later in the article. STREAM, which is commonly used to determine the performance of parallel computers, is a simple parallel code that uses basic vector operations, such as scale and add, to measure main memory bandwidth to each processor in the parallel computer. STREAM requires

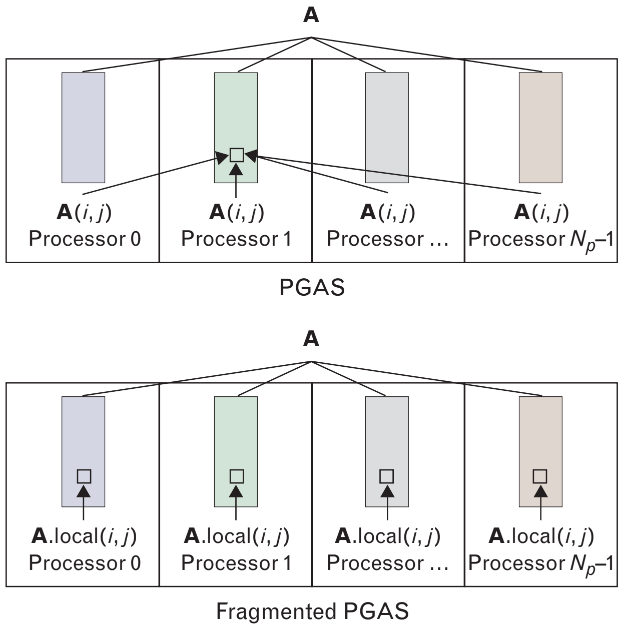


FIGURE 3. Parallel global array semantics (PGAS). The top figure illustrates pure PGAS. Matrix **A** is distributed among N_p processors, where each processor is represented by a different color. The convention of using different colors to illustrate different processors is used throughout this article. Element *i*,*j* is referenced on all the processors. In pure PGAS, the index *i*,*j* is a global index that references the same element in the global array, regardless of the processor that performs the reference (on Processor 1 that element is local; on all other processors it is remote). The lower figure illustrates fragmented PGAS. Here each processor references element *i*,*j* local to the processor; thus each processor references a different element in the global matrix. pMatlab supports both pure and fragmented PGAS.

no communication between processors or data redistribution, and is thus referred to as *embarrassingly* parallel. Distributed arrays allow the serial STREAM program to be transformed quickly into a parallel program by simply adding a map object to selected arrays. The map describes how the distributed array is to be broken up among multiple processors. Additionally, pMatlab also abstracts the communication layer from the application developer. While writing a parallel MATLAB program with pMatlab, the user does not have to worry about parallel programming concepts such as deadlocks, barriers, and synchronization.

Related Work

Parallel MATLAB has been an active area of research for a number of years and many different approaches

* The acronym PGAS also commonly refers to Parallel Global Address Space. Throughout this article, however, PGAS always refers to Parallel Global Array Semantics.

```
Np = pMATLAB.comm_size;          % Set number of processors.
N  = 32;                         % Set size of row vector.
s  = 3.14;                       % A scalar value.

ABCmap = map([1 Np],{},0:Np-1); % Create a map.
A = zeros(1,N,ABCmap);          % Distributed zeros array.
B =  rand(1,N,ABCmap);          % Distributed random array.
C =  rand(1,N,ABCmap);          % Distributed random array.

A(:,:) = B + s*C;                % Local scale and add.
```

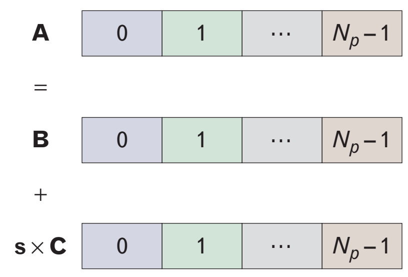


FIGURE 4. Highlights of the STREAM benchmark code. The first three lines set the various constants required by the program, such as the number of processors N_p and the size of the row vector. The next line creates a map, which will cause the second dimension of a distributed array to be broken up equally among all the processors. The following three lines use this map to create three distributed row vectors. The last line performs the basic STREAM triad arithmetic operations in parallel. No communication is required in this example because arrays **A**, **B** and **C** are all mapped identically.

have been developed (see [17] for a comprehensive survey). These different approaches can be roughly divided into three categories: message passing, client/server, and global arrays.

The message passing approach [6, 18] requires the user to explicitly send messages within the code. These approaches often implement a variant of the Message Passing Interface (MPI) standard [7]. Message passing allows any processor to directly communicate with any other processor and provides the minimum required functionality to implement any parallel program. Users who are already familiar with MPI find these approaches powerful. However, the learning curve is steep for the typical user because explicit message passing significantly lowers the level of abstraction and requires users to deal directly with deadlocks, barriers, and other low-level parallel programming concepts. In addition, the impact on code size is significant. Serial programs converted to parallel programs with MPI typically increase in size by 25% to 50%. In contrast, PGAS approaches typically increase the code size by only about 5% [19]. In spite of these difficulties, a message passing capability is a requirement for both the client/server approach and the global arrays approach. Furthermore, message passing is often the most efficient way to implement a program; there are certain programs with complex communication patterns that can be implemented only with direct message passing. Thus any complete parallel solution must provide a mechanism for accessing the underlying messaging layer.

Among the available MATLAB message passing implementations, MatlabMPI is currently the most popular implementation with thousands of users worldwide (see the section on the parallel MATLAB communication interface for a more detailed discussion on MatlabMPI). More recently, the incorporation of MPI into The MathWorks' Distributed Computing Toolbox (DCT) [20, 21] makes message passing available to a much broader range of users.

Client/server approaches [22, 23, 20] use MATLAB as the user's front end to a distributed library. For example, Star-P [22] keeps the distributed arrays on a parallel server, which calls the necessary routines from parallel libraries such as ScaLAPACK and FFTW. These approaches often provide the best performance once the data are transferred to the server. However, these approaches are limited to those functions which have been specifically linked to a parallel library and require the users to install additional libraries and set up a dedicated server to execute the libraries. We include DCT in this category, although in this instance the back-end server is MATLAB running on each processor, and the user is responsible for breaking up the calculation into embarrassingly parallel tasks that can be independently scheduled.

pMatlab, Star-P [22], and Falcon [24] fall into the third category, the global arrays approach, or PGAS, which provides a mechanism for creating arrays that are distributed across multiple processors. Global arrays appear in other languages such as High Performance Fortran [25, 26] and Unified Parallel C [27], as well as in many C++ libraries such as POOMA [28], GA Toolkit [29], PVL [9], and the C++ binding of the Vector Signal and Image Processing Library (VSIPL++) [9]. The global array approach allows the user to view a distributed object as a single entity, as opposed to multiple pieces as is the case with message passing. This approach allows operation on the arrays as a whole or on local parts of the arrays. Additionally, these libraries are compatible with MPI. Parallel VSIPL++ is implemented in C++. The GA Toolkit is implemented in a number of languages, including Fortran, C, and C++.

pMatlab supports both pure PGAS and fragmented PGAS programming models, as illustrated in Figure 3. The pure PGAS model presents an entirely global view of a distributed array. Specifically, once they are created, distributed arrays are treated the same as non-distributed arrays. When using this pure programming model, the user never accesses the local part of the array, and all operations (such as matrix multiplies, fast Fourier transforms, or convolutions) are performed on the global structure. The benefits of pure PGAS are ease of programming and the highest level of abstraction. The drawbacks include the need to implement parallel versions of serial operations and library performance overhead.

Fragmented PGAS maintains a high level of abstraction but allows access to local parts of the arrays. Specifically, a distributed array is created in the same manner as in pure PGAS; however, the operations can be performed on just the local part of the array. Later, the global structure can be updated with locally computed results. This approach allows greater programming flexibility; at the same time it does not require function coverage or implementation of parallel versions of all existing serial functions. Furthermore, fragmented PGAS programs often achieve better performance by eliminating the library overhead on local computations.

pMatlab is a unique parallel MATLAB implementation for a number of reasons. pMatlab supports both pure and fragmented PGAS programming models, and allows combining PGAS with direct message passing for optimized performance. While pMatlab does use message passing in the library routines, a typical user does not have to explicitly incorporate messages into the code. pMatlab supports embarrassingly parallel computation but is not limited to it. In addition, pMatlab does not link in any external libraries, nor does it compile the language into an executable. Our library is implemented entirely in MATLAB. This design significantly reduces the size of the library and has allowed pMatlab to become the most complete implementation of PGAS available in any language.

pMatlab Interface and Architecture Design

The primary challenge in implementing a parallel computation library is how to balance the conflicting goals of high performance, ease of use, and ease of implementation. With respect to pMatlab, we have specifically defined each of these design goals in a measurable way, which are listed in Table 1. The performance metrics, typical of those used throughout the high-performance computing community, primarily look at the computation and memory overhead of programs written with pMatlab relative to serial programs written using MATLAB and parallel programs written using C with MPI. The metrics for ease of use and ease of implementation are derived from the software engineering community [30–32], and they look at code size, programmer effort, and required programmer expertise. These metrics are not perfect, but they are useful tools for measuring progress toward these goals. In the rest of this section we discuss the particular choices made in pMatlab to satisfy these goals.

Ease of use. The first step in writing a parallel program is to start with a functionally correct serial program that achieves the best performance possible. The conversion from serial to parallel requires the user to add new constructs to their code. pMatlab adopts a separation-of-concerns approach to this process which seeks to make functional programming and mapping a program to a parallel architecture orthogonal. A serial program is made parallel by adding maps to arrays. Maps contain information only about how an array is broken up onto multiple processors, and the addition of a map *should not* change the functional correctness of a program. A map, illustrated in Figure 5, is composed of three fundamental parts: a grid, specifying how each dimension is partitioned; a distribution, specifying a block, cyclic, or block-cyclic partitioning (discussed in the later section on maps and distributions); and a list of processors, specifying which processors actually hold the data. (Note that the use of

Table 1. pMatlab Design Goals *

Goal	**Ease of use**
Metrics	Time for a user to produce a well-performing parallel code from a serial code.
	Fraction of serial code that has to be modified.
	Expertise required to achieve good performance.
Approach	Separate functional coding from mapping onto a parallel architecture.
	Abstract message passing away from the user.
	Ensure that simple (embarrassingly) parallel programs should be simple to express.
	Provide a simple mechanism for globally turning pMatlab constructs on and off.
	Ensure backward compatibility with serial MATLAB.
	Provide a well-defined and repeatable process for migrating from serial to parallel code.
Goal	**High performance**
Metrics	Execution time and memory overhead as compared to serial MATLAB, the underlying MatlabMPI communication library, and C+MPI benchmarks.
Approach	Use underlying serial MATLAB routines wherever possible (even if it means slightly larger user code).
	Minimize use of overloaded functions whose performance depends on how distributed arrays are mapped.
	Provide a simple mechanism for using lower-level communication when necessary.
Goal	**Ease of implementation**
Metrics	Time to implement a well-performing parallel library.
	Size of library code.
	Number of objects.
	Number of overloaded functions.
	Functional and performance test coverage.
Approach	Implement a layered design that separates math and communication.
	Leverage well-understood PGAS and data redistribution constructs.
	Minimize the use of overloaded functions.
	Develop a 'pure' MATLAB implementation to minimize code size and maximize portability.

* Metrics were defined for each of the high-level pMatlab design goals: ease of use, performance, and ease of implementation. These metrics led to specific approaches for addressing the goals in a measurable way.

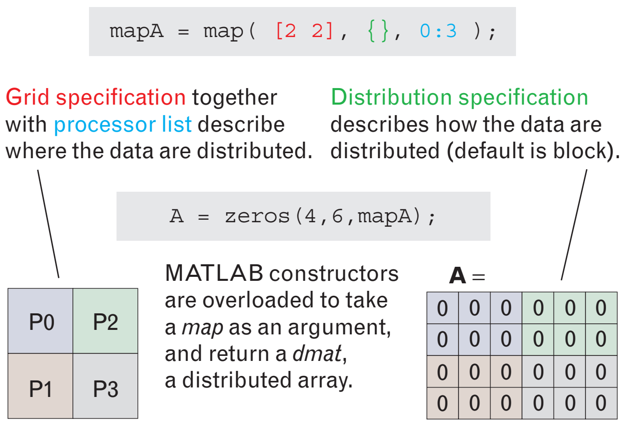
the word "grid" in this context is not related to grid computing.)

The next step in writing a parallel program is implementing communication. Perhaps the largest benefit of PGAS is the ability to abstract complex message passing away from the user. More specifically, redistribution between any two distributed arrays in pMatlab is accomplished with the '=' operator. In the STREAM benchmark example shown in Figure 4, the '=' operator is used in the statement

```
A(:,:) = B + s*C,
```

but since the arrays **A**, **B**, and **C** all have the same map, no communication is required. The overloaded '=' operator in pMatlab figures this out and correctly performs a simple assignment of the local data on the right hand side to the local data on the left hand side.

A more complex example is the HPC Challenge



```
mapA = map( [2 2], {}, 0:3 );
```

```
A = zeros(4,6,mapA);
```

Grid specification together with processor list describe where the data are distributed.
Distribution specification describes how the data are distributed (default is block).
MATLAB constructors are overloaded to take a *map* as an argument, and return a *dmat*, a distributed array.
P0
P2
P1
P3
A =
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

FIGURE 5. Anatomy of a map. A map for a numerical array is an assignment of blocks of data to processing elements. It consists of a grid (specified in this case as a 2 × 2 arrangement), a distribution (the braces {} specify that the default block distribution should be used in this case), and a processor list (the array is mapped to processors 0, 1, 2, and 3).

fast Fourier transform (FFT) benchmark shown in Figure 6. This benchmark computes the FFT of a large one-dimensional (1D) vector. The standard parallel algorithm for this benchmark is to transform the 1D vector into a row-distributed matrix, perform an FFT on the rows of the matrix, multiply by a set of weights, redistribute into a column-distributed matrix, and perform an FFT on the columns. A key step in the process is the redistribution performed by the statement

```
Z(:,:) = X,
```

which determines and executes the N_p^2 messages—where N_p is the number of processors—that need to be sent to complete this operation.

PGAS enables complex data movements to be expressed compactly without making parallelism a burden to code. For example, removing the maps from either the STREAM or FFT example returns the program to a valid serial program that simply uses standard built-in operations. This feature is a direct result of the orthogonality of mapping and functionality, and allows the pMatlab library to be turned off by simply setting all the maps equal to the scalar value of 1. This ability to turn the library on and off is a key debugging feature and allows users to determine whether the bugs are from problems in their serial code or due to their use of the pMatlab constructs.

All of these steps—making the code parallel, managing the communication, and debugging—need to be directly supported in the library. Our experience with many pMatlab users has resulted in a standardized and repeatable four-step process, summarized in Figure 7, for going quickly from a serial code to a well-performing parallel code. This four-step process is important for a user to learn and follow, because the natural tendency of new pMatlab users is to add parallel functions and immediately attempt to run large problems on a large number of processors.

The four-step process begins by adding distributed matrices to the serial program, but then assigning all the maps to a value of 1 and verifying the program

```
Np = pMATLAB.comm_size;   % Set number of processors.
P = 2^10;    Q = 2^10;    % Set dimensions of array.

Xmap = map([Np 1],{},0:Np-1);    % Row map.
Zmap = map([1 Np],{},0:Np-1);    % Column map.
% Create complex global arrays X and Z for FFT.
X = complex(rand(P,Q,Xmap),rand(P,Q,Xmap));
Z = complex(zeros(P,Q,Zmap));

X = fft(X,[],2);          % FFT rows.
X = X.* W;.               % Multiply by weights.
Z(:,:) = X;               % Redistribute data.
Z = fft(Z,[]1);           % FFT columns.
```

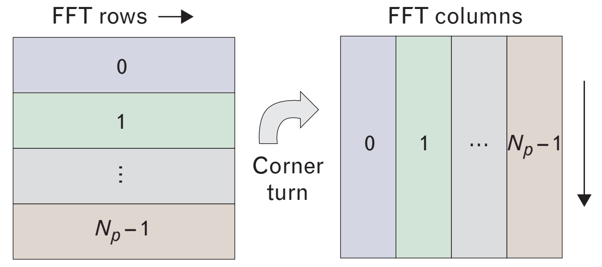


FIGURE 6. Highlights of the fast Fourier transform (FFT) benchmark code. The first two lines set the various constants required by the program, such as the number of processors N_p and the size of the matrix. The next two lines create two map objects for breaking the matrix up into rows and into columns, and the following two lines use the maps to create two matrices. The subsequent four lines execute an FFT on the rows, multiply the data by a set of local pre-computed weights, redistribute the data (using the '=' operator) into the matrix broken up by columns, and then perform an FFT on the columns.

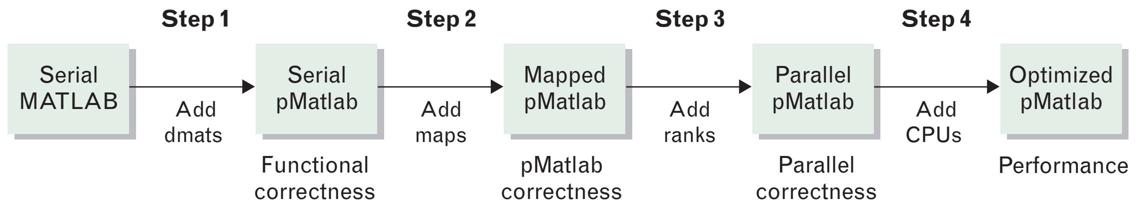


FIGURE 7. The four-step process for going from serial code to parallel code. Step 1 adds distributed matrices to the serial program, then assigns all the maps a value of 1 and runs with the number of processors N_p set equal to 1 on the local machine. Step 2 turns on the maps and runs the program again with N_p set equal to 1. Step 3 runs with N_p greater than 1 on the local machine. Step 4 runs with N_p greater than 1 on multiple machines. Debugging should always be performed at the lowest numbered step where a problem occurs.

with N_p set equal to 1 on the local machine. The second step is to turn on the maps and run the program again with N_p still set equal to 1, which will verify that the pMatlab overloading constructs are working properly. It is also important to look at the relative performance of the first and second steps, as this will indicate if any unforeseen overheads are incurred by using the pMatlab constructs. The third step is to run with N_p greater than 1 on the local machine, which will verify that the pMatlab communication constructs are working properly. The fourth and final step is to run with N_p greater than 1 on multiple machines, which validates that the remote communication is working properly. Only after these four steps have been performed is it worthwhile to attempt to run large problems on many processors. In addition, it is important to always debug problems at the lowest numbered step in which they occur.

High Performance. The primary goal of using a parallel computer is to improve run-time performance. The first step in achieving high performance is to minimize the overhead of using pMatlab constructs, compared to their serial equivalents. The previous examples (Figures 4 and 6) show the ideal pure PGAS case when all the required functions are overloaded to work well with the pMatlab distributed arrays. It is impractical (and unnecessary) to provide optimized implementations of the approximately 8000 built-in functions for every combination of array distributions. Instead, we adopt a coding style that uses some fragmented PGAS constructs, illustrated earlier in Figure 3. This style is less elegant but provides strict guarantees on performance. More specifically, distributed arrays are used as little as possible and only when interprocessor communication is required.

Figure 8 shows examples of the STREAM and FFT benchmarks written with fragmented PGAS constructs that minimize the use of overloaded functions by employing the *local* and *put_local* functions (described later in the section on parallel support functions). The *local* function extracts the local part of the distributed array and returns a regular MATLAB array that will work with any serial MATLAB function. The *put_local* function replaces the local part of a distributed array with a regular serial MATLAB array. Thus in the STREAM and FFT examples the key expressions

```
Alocal = Blocal + s*Clocal
```

and

```
fft(local(X),[],2)
```

are guaranteed to have the same performance as the equivalent serial function calls, and eliminate the need for pMatlab to overload the calls to +, `*`, and *fft*. In addition to providing a local performance guarantee, this style of coding minimizes the potential for accidental communication, which is easy to do with the '=' operator. This style of coding has proven to be very effective, and most users are able to adapt their code to this style with minimum effort. In support of this style, the pMatlab library also provides serial equivalents of the *local* and *put_local* functions so that the code will still work if parallel arrays are turned off.

The power of PGAS is its ability to hide underlying communication from the user and eliminate the need for writing lengthy and complex message passing code. Unfortunately, PGAS constructs are not appropriate for all circumstances. There are communication patterns that simply would be more efficient if direct message passing can be employed. Thus it is impor-

Optimized STREAM code

```
Np = pMATLAB.comm_size;
N = 32;
s = 3.14;

ABCmap = map([1 Np],{},0:Np-1);
Alocal = local(zeros(1,N,ABCmap));
Blocal = local(rand(1,N,ABCmap));
Clocal = local(rand(1,N,ABCmap));

Alocal = Blocal + s*Clocal;
```

Optimized FFT code

```
Np = pMATLAB.comm_size;
P = 2^10;   Q = 2^10;


Xmap = map([Np 1],{},0:Np-1);
X = complex(rand(P,Q,Xmap),rand(P,Q,Xmap));

X = put_local(X, fft(local(X),[],2) .* Wlocal );
Z = transpose_grid(X);
Z = put_local(Z, fft(local(Z),[],1) );
```

FIGURE 8. Highlights of optimized STREAM and FFT code. Programs are rewritten with the *local* and *put_local* functions to minimize the required number of overloaded functions. These programs are guaranteed to provide the same local performance as their serial equivalents.

tant to have mechanisms that allow PGAS and the underlying communication constructs to interact easily. pMatlab provides this ability by allowing the user to directly access the underlying MatlabMPI library and its data structures. At any time in the program the user, if she or he so desires, can choose to send messages directly with MatlabMPI. In fact, we have found that PGAS and message passing work very well together since the PGAS constructs can still be used to quickly figure out which data to send and where to send it. But this style of programming is recommended only for advanced pMatlab programmers, since it requires experience with message passing.

Several of the HPC Challenge benchmarks fall into the class of codes that do best by allowing some use of direct message passing. In the case of the FFT code, we have used a special function called *transpose_grid* (see Figure 8) that directly uses MatlabMPI messaging to optimally perform the all-to-all communication for going from a row-distributed matrix to a column-distributed matrix. This function is able to use memory more efficiently and to optimize the order in which messages are sent and received. The *transpose_grid* function replaces the '=' operator, which can incur significant overhead. The RandomAccess benchmark, described later in the RandomAccess section, requires that all processors are able to randomly communicate with all other processors. It is a more explicit example of using messaging and PGAS together. The HPL Top500 benchmark, described in the section on High Performance Linpack, requires that one processor be able to broadcast to a subset of all the other processors, which is also most easily dealt with by using direct message passing.

Ease of implementation. The ease of use and high-performance goals are well understood by the HPC community. Unfortunately, implementing these goals in a middleware library often proves to be quite costly. A typical PGAS C++ library, such as the Parallel Vector Library (PVL), can be 50,000 lines of code and requires several programmers years to implement. pMatlab has adopted several strategies to reduce implementation costs. The common theme among these strategies is finding the minimum set of features that still allow users to write well-performing programs.

One of the key choices in implementing a PGAS library is which data distributions to support (see the section on maps and distributions). At one extreme we can argue that most users are satisfied by 1D block distributions. At the other extreme, we can find applications that require truly arbitrary distributions of array indices to processors. We have chosen to support all four-dimensional (4D) block-cyclic distributions with overlap in pMatlab because the problem of redistribution between any two such distributions has been solved a number of times by different parallel computing technologies (see Appendix A).

The pMatlab '=' operator supports data redistribution between arrays. The next question is what other functions to support and for which distributions. Table 2 shows an enumeration of different levels of PGAS support. The ability to work with the local part of a distributed array and its indices has also been demonstrated repeatedly. The big challenge is overloading all mathematical functions in a library to work well with every combination of input distributions. This capability is extremely difficult to implement and is not entirely necessary if users are willing to tolerate the slightly less elegant coding style associated with fragmented PGAS. Thus pMatlab provides a rich set of data distributions, but a relatively modest set of overloaded functions, which are mainly focused on array construction functions, array index support functions, and the various element-wise operations (+, –, .*, ./, …).

The final implementation choice was to implement pMatlab purely in MATLAB without relying on binding to other languages. This choice has minimized code size and maximized portability. For example, pMatlab is the most complete implementation of PGAS, but it is only about 3000 lines of code and has introduced only two new objects (maps and distributed arrays). pMatlab also runs on any combination of heterogeneous systems that support MATLAB, which includes Windows, Linux, Mac OS X, and SunOS.

The Parallel MATLAB Toolbox Implementation

The pMatlab library builds upon concepts from PVL and Star-P, and uses MatlabMPI as the communication layer. Figure 9 illustrates the layered architecture of the parallel library. In the layered architecture, the pMatlab library implements distributed constructs, such as distributed matrices and higher dimensional arrays. In addition, pMatlab provides parallel implementations of a select number of functions such as redistribution, FFT, and matrix multiplication. However, providing high-performance parallel implementations for all possible serial functions and combinations of distributions is impractical. This problem is solved by introducing the user to fragmented PGAS programming style, as discussed previously, which significantly simplifies the task of writing specialized parallel routines focused on the user's particular data sizes and data distributions.

The pMatlab library uses the parallelism through polymorphism approach [22]. Monomorphic languages require that each variable is of only one type; on the other hand, in polymorphic languages variables

Table 2. Parallel Implementation Levels*

Data Level	*Description of Support*
Data0	Distribution of data is not supported [not a parallel implementation]
Data1	One dimension of data may be block distributed
Data2	Two dimensions of data may be block distributed
Data3	Any and all dimensions of data may be block distributed
Data4	Any and all dimensions of data may be block or cyclicly distributed
Operations Level	*Description of Support*
Op0	No distributed operations supported [not a parallel implementation]
Op1	Distributed assignment, get, and put operations, and support for obtaining data and indices of local data from a distributed object
Op2	Distributed operation support (the implementation must state which operations those are)

* Levels of parallel support for data and functions. Note: Support for data distribution is assumed to include support for overlap in any distributed dimension.

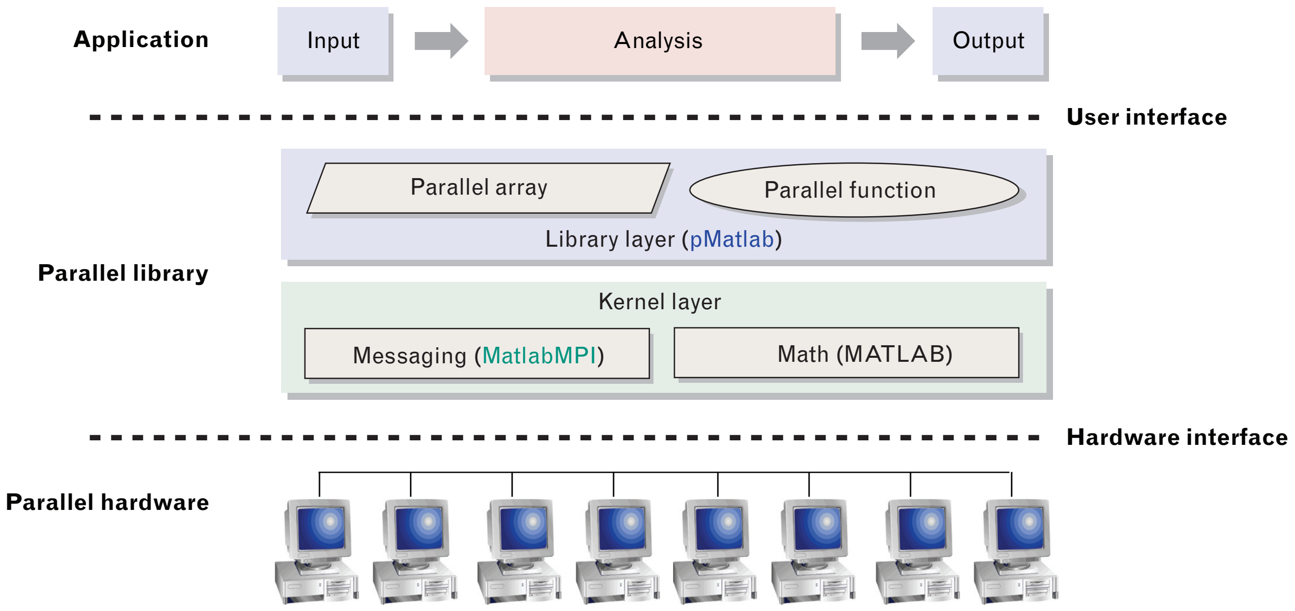


FIGURE 9. Layered architecture. The pMatlab library implements distributed constructs, such as vectors, matrices, and multidimensional arrays and parallel algorithms that operate on those constructs, such as redistribution, FFT, and matrix multiplication.

can be of different types and polymorphic functions can operate on different types of variables [33]. The concept of polymorphism is inherent in the MATLAB language; variable types do not have to be defined, variable types can change during the execution of the program, and many functions operate on a variety of data types such as double, single, and complex.

In pMatlab, this concept is taken one step further. The polymorphism is exploited by introducing the map object. Map objects belong to a pMatlab class *map* and are created by specifying the grid description, the distribution description, and the processor list, as discussed in the section on ease of use and as illustrated in Figure 5. The map object can then be passed to a MATLAB constructor, such as *rand*, *zeros*, or *ones*. The constructors are overloaded and when a map object is passed into a constructor, the library creates a variable of type *dmat*, or a distributed array. A PITFALLS structure, described in Appendix A, is created when each dmat object is constructed. PITFALLS is a mathematical representation of data distribution information. pMatlab supports numerical arrays of up to four dimensions of different numerical data types and allows creation of distributed sparse matrices.

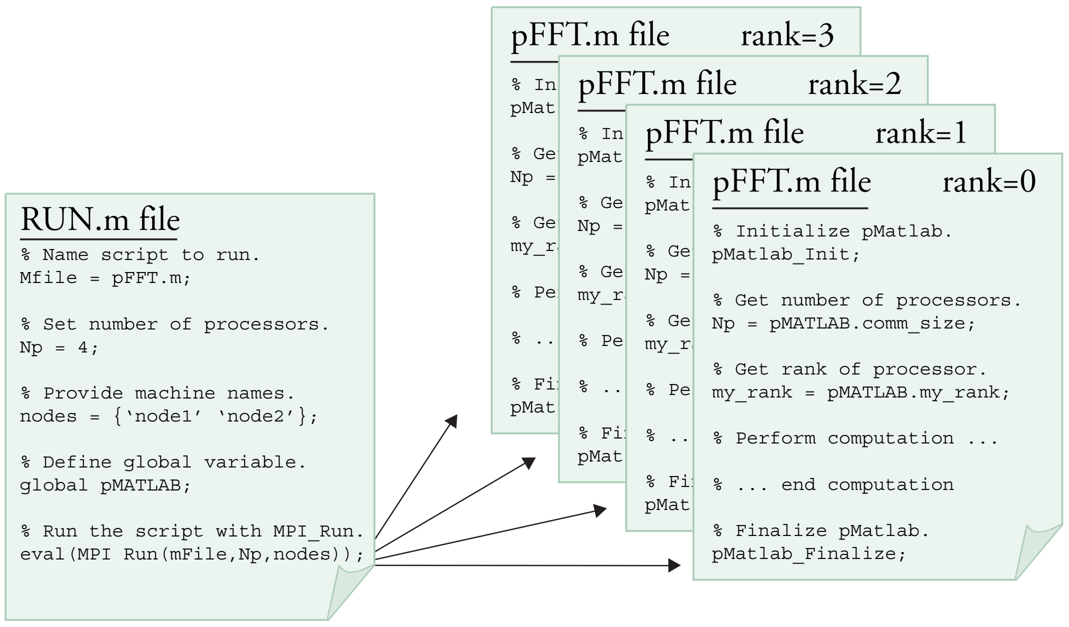
As discussed previously, a subset of functions, such as *plus*, *minus*, *fft*, *mtimes*, and all element-wise operations are overloaded to operate on dmat objects. When using a pure PGAS programming model and an overloaded function, the dmat object can be treated as a regular array. Functions that operate only on the local part of the dmat object (element-wise operations) simply perform the operations requested on the local array, which is a standard MATLAB numerical type specified at array creation. Functions that require communication, such as redistribution (or *subsasgn* in MATLAB syntax) use MatlabMPI as the communication layer.

Let us return to the pMatlab FFT code in Figure 6. Lines 3 and 4 define two pMatlab map objects: *Xmap* and *Zmap*. The user defines maps to specify how and where the numerical arrays in the program are mapped. In this example, all available processors are used (numbered sequentially from 0 to $N_p - 1$). Distributed arrays are created by using the standard MATLAB array constructors. The outputs of the overloaded constructors are dmat objects. Lines 5 and 6 in Figure 6 create two distributed complex matrices split up among N_p processors. *Xmap* indicates that the matrix should be distributed row-wise with P/N_p rows per processor, where as *Zmap* defines a column-wise distribution with Q/N_p columns per processor. If a dimension is not evenly divisible by N_p, pMatlab figures this out and shorts the last processor. Line 7 calls the overloaded FFT function on the distributed array **X** and returns the result into an array with the same map as the input. Line 9 uses the overloaded '=' operator, which performs an all-to-all communication that results in **Z** having the same data as **X**, while distributing this data according to the distribution defined in *Zmap*.

Since all functions supported in pMatlab are implemented in pure MATLAB, the pMatlab library maintains the portability of MatlabMPI. pMatlab can run anywhere MATLAB runs, given that there exists a common file system, a constraint imposed on pMatlab by MatlabMPI. A further benefit of the layered architecture of pMatlab is that any other communication library could be substituted for MatlabMPI, given that it implements the six basic MPI functions required by pMatlab (described in the section on the parallel MATLAB communication interface).

Parallel MATLAB Execution. All pMatlab code resides within a generic execution code framework (illustrated in Figure 10) for initializing pMatlab (*pMatlab_Init*), determining the number of processors the program is being run on (*pMATLAB.comm_size*), determining the rank, or ID, of the local processor (*pMATLAB.my_rank*), and finalizing the pMatlab library when the computation is complete (*pMatlab_Finalize*). pMatlab uses the single-program multiple-data (SPMD) execution model, where the same code runs on all of the processors being used but each processor might operate on a different part of the data. The user runs a pMatlab program by calling the MatlabMPI command *MPI_Run* to launch and initialize the multiple instances of MATLAB required to run in parallel. Figure 10 shows an example of a RUN.m script that uses *MPI_Run* to launch four copies of the pFFT.m script.

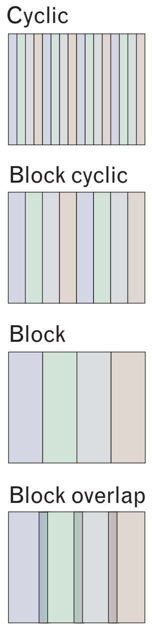
Maps and Distributions. The concept of using maps to describe array distributions has a long history. The ideas for pMatlab maps are principally drawn from the High Performance Fortran community [34, 35], the Lincoln Laboratory Space-Time Adaptive Processing Library (STAPL) [36], and the Parallel Vector Library (PVL) [9]. The map concept has been adopted



```
% Name script to run.
Mfile = pFFT.m;

% Set number of processors.
Np = 4;

% Provide machine names.
nodes = {'node1' 'node2'};

% Define global variable.
global pMATLAB;

% Run the script with MPI_Run.
eval(MPI Run(mFile,Np,nodes));
```

```
% Initialize pMatlab.
pMatlab_Init;

% Get number of processors.
Np = pMATLAB.comm_size;

% Get rank of processor.
my_rank = pMATLAB.my_rank;

% Perform computation ...

% ... end computation

% Finalize pMatlab.
pMatlab_Finalize;
```

FIGURE 10. pMatlab execution code framework. A pMatlab program (pFFT.m) is launched by using the MPI_Run command shown in the RUN.m file, which sets the number of processors and specifies which machines to run on. MPI_Run starts N_p instances of MATLAB, each with a different rank. Within the pMatlab program the pMatlab environment is initialized and the number of processors and local rank are obtained. The pMatlab_Finalize command completes the program.



```
Np = pMATLAB.comm_size;  % Set number of processors.
N = 16;                  % Set size of row vector.

dist_spec.dist = ‘c’;    % Define cyclic distribution.

dist_spec.dist = ‘bc’;   % Define block-cyclic distribution.
dist_spec.size = 2;      % Set block size = 2.

dist_spec.dist = ‘b’;    % Define block distribution.
Amap = map([1 Np],dist_spec,0:Np-1);  % Create a map.

% Map with overlap of 1.
Amap = map([1 Np],dist_spec,0:Np-1,[0 1]);

A = zeros(1,N,Amap);     % Create a distributed array.
```

FIGURE 11. Supported distributions. Block distribution divides the object evenly among available processors. Cyclic distribution places a single element on each available processor and then repeats. Block-cyclic distributions places the specified number of elements on each available processor and then repeats. Block overlap allows for replication of rows and/or columns of data on neighboring processors.

by VSIPL++, a C++ standard for writing embedded signal- and image-processing applications. A map for a numerical array defines how and where the array is distributed (as shown earlier in Figure 3). PVL also supports task parallelism with explicit maps for modules of computation. pMatlab explicitly supports only data parallelism; however, implicit task parallelism can be implemented through careful mapping of data arrays.

The pMatlab map construct is defined by three components: (1) the grid description, (2) the distribution description, and (3) the processor list. The grid, together with the processor list, describes where the data object is distributed, while the distribution describes how the object is distributed (as shown earlier in Figure 5). pMatlab supports any combination of block-cyclic distributions up to four dimensions. Figure 11 shows the Application Programmer Interface (API) for defining these distributions.

Block distribution is the default distribution, which can be specified explicitly or by simply passing an empty distribution specification to the map function. Cyclic and block-cyclic distributions require the user to provide more information. Having access to a variety of data distributions allows the users to implement efficient algorithms. For example, a block distribution is efficient for computations where operations on the array are largely uniform; on the other hand, a cyclic distribution is often necessary for load balancing in some matrix decomposition routines. Distributions can be defined for each dimension, and each dimension could potentially have a different distribution scheme. Additionally, if only a single distribution is specified and the grid indicates that more than one dimension is distributed, that distribution is applied to each dimension.

Some applications, particularly image processing, require data overlap, or replicating rows or columns of the data on neighboring processors. This capability is also supported through the map interface. If overlap is necessary, it is specified as an additional fourth argument. In Figure 11, the fourth argument indicates that there is 0 overlap between rows and 1 column overlap between columns. Overlap can be defined for any dimension and does not have to be the same across dimensions.

Maps introduce a new construct and potentially reduce the ease of programming. However, maps have significant advantages over message passing approaches and predefined limited-distribution approaches. Specifically, pMatlab maps are scalable, allow optimal distributions for different algorithms, and support pipelining.

Maps are scalable in both the size of the data and the number of processors. Maps allow the user to separate the task of mapping the application from the task of writing the application. Different sets of maps do not require changes to be made to the application code. Specifically, the distribution of the data and the number of processors can be changed without making any changes to the algorithm. Separating mapping of the program from the functional programming is an important design approach in pMatlab.

Maps make it easy to specify different distributions to support different algorithms. Optimal or suggested distributions exist for many specific computations. For example, matrix multiply operations are most efficient on processor grids that are transposes of each other. Column- and row-wise FFT operations produce linear speedup if the dimension along which the array is broken up matches the dimension on which the FFT is performed (see Figure 6 as an example).

Maps also allow the user to set up *pipelines* in the computation, thus supporting implicit task parallelism. Task parallelism allows the user to perform different computations (tasks) on different sets of processors, while pure data parallelism implies simply splitting up the data between processors and performing all computations on all processors on the local part of the data. For example, pipelining is a common approach to hiding the latency of the all-to-all communication required in parallel FFT. The following slight change in the maps can be used to set up a pipeline where the first half of the processors performs the first part of the FFT and the second half performs the second part:

```
% Row map on first set of processors.
Xmap = map([Np/2 1],{},[0 :Np/2-1]);
% Column map on second set of processors.
Zmap = map([1 Np/2],{},[Np/2:Np-1]);
```

When a processor encounters such a map, it first checks if it has any data to operate on. If the processor doesn't have any data, it proceeds to the next line. In the case of the FFT with the above mappings, the first half of the processors (rank 0 to $N_p/2 - 1$) will simply perform the row FFT, send data to the second set of processors, skip the column FFT, and proceed to process the next set of data. Likewise, the second set of processors (ranks $N_p/2$ to $N_p - 1$) will skip the row FFT, receive data from the first set of processors, and perform the column FFT.

Parallel MATLAB Communication Interface. MatlabMPI is a pure MATLAB implementation of the most basic MPI functions [7]. Table 3 lists the functions required by pMatlab. The file input/output (I/O)-based communication is done through a common file system, illustrated in Figure 12. The advantage of this approach is that the library is small (about 300 lines), is highly portable, has a huge message buffer (the size of disk storage), and allows for non-blocking message sends. The price for this portability is that while MatlabMPI performance is comparable to C+MPI for large messages, its latency for small messages is much higher, as shown in Figure 13.

Table 3. Selected MPI Functions Provided by MatlabMPI

Function Name	*Function Description*
MPI_Init	Initializes MPI
MPI_Comm_size	Gets the number of processors in a communication
MPI_Comm_rank	Gets the rank of current processor within a communicator
MPI_Send	Sends a message to a processor
MPI_Recv	Receives a message from a processor
MPI_Finalize	Finalizes MPI

* pMatlab can be built on top of any communication library that implements these six functions.

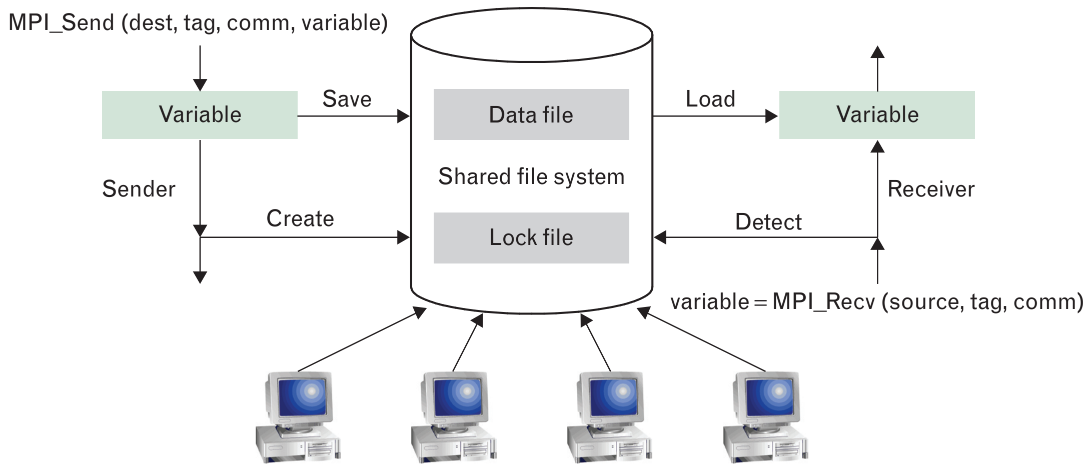


FIGURE 12. MatlabMPI file input/output (I/O)-based communication. MatlabMPI uses file I/O to implement point-to-point communication. The sender writes variables to a buffer file and then writes a lock file. The receiver waits until it sees the lock file, and it then reads in the buffer file.

When we were designing pMatlab, it was important to ensure that the overhead incurred by the library did not significantly impact performance. From a library perspective, this means that the performance of the communication operations using the overloaded '=' operator should be as close as possible to the equivalent MatlabMPI code. Figure 14 shows the performance of an all-to-all operation using MatlabMPI, pMatlab '=', and the pMatlab *transpose_grid* function.

From an application perspective, minimizing overhead means using algorithms that use fewer larger messages instead of many smaller messages. In the section on HPC Challenge benchmarks we see that the relative performance of these benchmarks can essentially be derived from the performance of the underlying MatlabMPI library. STREAM (no communication) delivers essentially equivalent performance to the C+MPI implementation. FFT (all-to-all) and Top500 (broadcast) fall into the large message regime and deliver reasonable performance. RandomAccess is designed to stress small messages, and the relative performance of pMatlab is much worse. Fortunately, most real pMatlab programs tend to involve large messages.

Parallel Support Functions. Every PGAS implementation must provide a set of functions for managing and working with global arrays, which have no serial

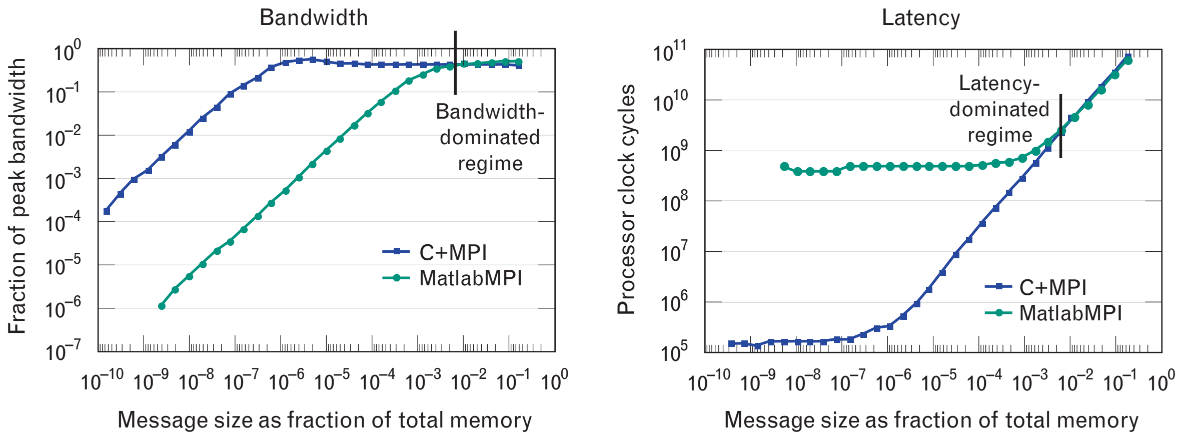


FIGURE 13. MatlabMPI versus C+MPI. Bandwidth and latency versus message size. Bandwidth is given as fraction of the peak underlying link bandwidth. Latency is given in terms of processor cycles. For large messages the performance is comparable. For small messages the latency of MatlabMPI is much higher.

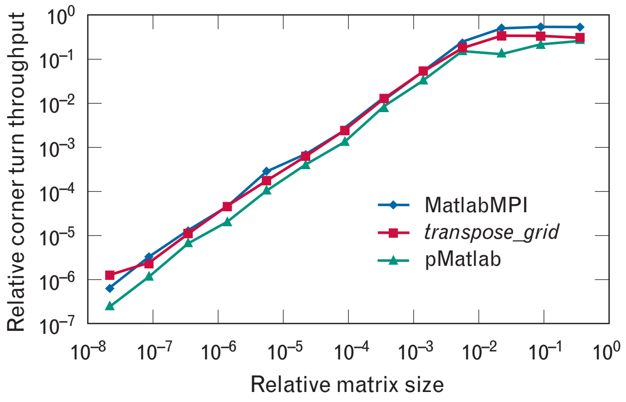


FIGURE 14. MatlabMPI versus pMatlab. Relative all-to-all performance for a pure MatlabMPI implementation, an A(:,:) = B pMatlab implementation, and a *transpose_grid* implementation. The *x*-axis represents size of each matrix relative to node memory. The *y*-axis represents throughput relative to peak bandwidth. The comparison illustrates that pMatlab incurs only a small overhead compared to MatlabMPI. The overhead can be further reduced by using a specialized function such as *transpose_grid*.

equivalents. Table 4 shows the set of pMatlab parallel support functions. These functions allow the user to aggregate data onto one or many processors, determine which global indices are local to which processors, and get/put data from/to the local part of a distributed array. This set of functions is relatively small. To support the development process discussed in the section on ease of use, all these functions have been overloaded to also work on serial MATLAB arrays so that the code will still work if the pMatlab maps have been turned off.

High Performance Computing Challenge Benchmarks

In this section we focus on benchmark results to determine the limits of pMatlab performance. We are interested in looking at performance from a number of viewpoints. First, we are interested in the performance of pMatlab relative to serial MATLAB since this is what most users care about. Second, we are interested in the performance of pMatlab relative to C+MPI as a way of gauging the quality of the implementation and as a guide to future performance enhancements. We have chosen to use the HPC Challenge benchmark suite [37] developed under the Defense Advanced Research Projects Agency (DARPA) High Productivity

Table 4. pMatlab Parallel Support Functions

Function name	*Function description*
synch	synchronizes the data in the distributed matrix
agg	aggregates the parts of a distributed matrix on the leader processor
agg_all	aggregates the parts of a distributed matrix on all processors in the communication scope
global_block_range	returns the ranges of global indices local to the current processor
global_block_ranges	returns the ranges of global indices for all processors in the map of distributed array **D** on all processors in communication scope
global_ind	returns the global indices local to the current processor
global_inds	returns the global indices for all processors in the map of distributed array **D**
global_range	returns the ranges of global indices local to the current processor
global_ranges	returns the ranges of global indices for all processors in the map of distributed array **D**
local	returns the local part of the distributed array
put_local	assigns new data to the local part of the distributed array
grid	returns the processor grid onto which the distributed array is mapped
inmap	checks if a processor is in the map

Computing Systems program, summarized in the sidebar of the same name, for this comparison. HPC Challenge, illustrated in Figure 15, is designed to represent a range of computations that focus on different parts of the memory hierarchy. In addition, HPC Challenge computations are sufficiently well defined that they can be implemented in a variety of programming models. We will first present the performance results and then discuss each of the benchmarks in more detail.

The four primary HPC Challenge benchmarks (STREAM, FFT, Top500, and RandomAccess) were implemented with pMatlab and run on the LLGrid β system. Details of the LLGrid β system are discussed in the section on grid computing architecture. Both the pMatlab and C+MPI reference implementations of the benchmarks were run on 1, 2, 4, 8, 16, 32, 64, and 128 processors. At each processor count the largest problem size that would fit in the main memory was run. The collected data measure the relative compute performance and the memory overhead of pMatlab with respect to C+MPI, as illustrated in Figure 16. In addition, we also look at the relative code sizes of the benchmarks as an approximate measure of the complexity of the implementations. The three parts of Table 5 summarize the relative memory required, the benchmark performance, and the code size comparisons, respectively.

In general, we see from Figure 16 that the pMatlab implementations can run problems that are typically half the size of problems in C+MPI implementations. This is mostly due to the need to create temporary arrays when using high-level expressions. The pMatlab performance ranges from being comparable to the C+MPI code (FFT and STREAM), to somewhat slower (Top500), to a lot slower (RandomAccess). In contrast, the pMatlab code is typically three to forty times smaller than the equivalent C+MPI code.

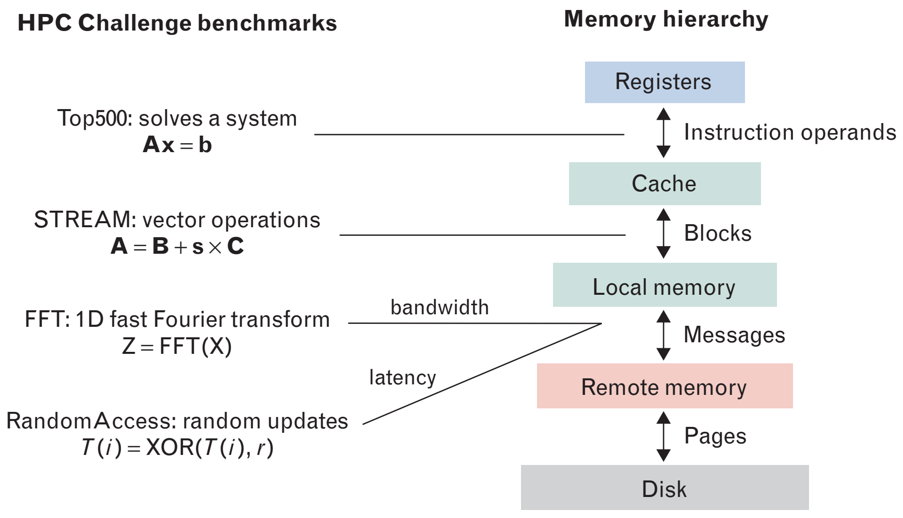


FIGURE 15. High Performance Computing (HPC) Challenge and the memory hierarchy. HPC Challenge benchmarks have been chosen to cover a range of memory access patterns and stress different parts of the memory hierarchy. Top500 performance is mostly dominated by local matrix multiply operations. RandomAccess is dominated by all-to-all communications of very small messages. FFT is also dominated by all-to-all communications, but for very large messages. STREAM requires no communication, is dominated by local vector operations, and stresses local processor-to-memory bandwidth.

STREAM

The STREAM benchmark consists of local operations on distributed vectors. The operations are copy, scale, add, and triad, with triad defined as

$$\mathbf{a} \leftarrow \mathbf{b} + \alpha \mathbf{c}$$

where **a**, **b**, and **c** are double precision vectors of length *m*, with the constraint

$$\begin{aligned} \text{size}(\mathbf{a}) + \text{size}(\mathbf{b}) + \text{size}(\mathbf{c}) &= 24m \text{ bytes} \\ &> \tfrac{1}{4} \text{ system memory.} \end{aligned}$$

The goal of the benchmark is to measure local main memory bandwidth, so performance is reported in terms of bytes/sec

$$\text{Gigabytes/sec} = 10^{-9}\, 24m/\text{time}$$

The operations are embarrassingly parallel and are implemented entirely with the pMatlab fragmented PGAS approach (see Figure 8).

The maximum problem size of the pMatlab code is 1.5 times smaller than the C+MPI code, which is due to the need to create intermediate temporary arrays. The need for temporaries is a side effect of most

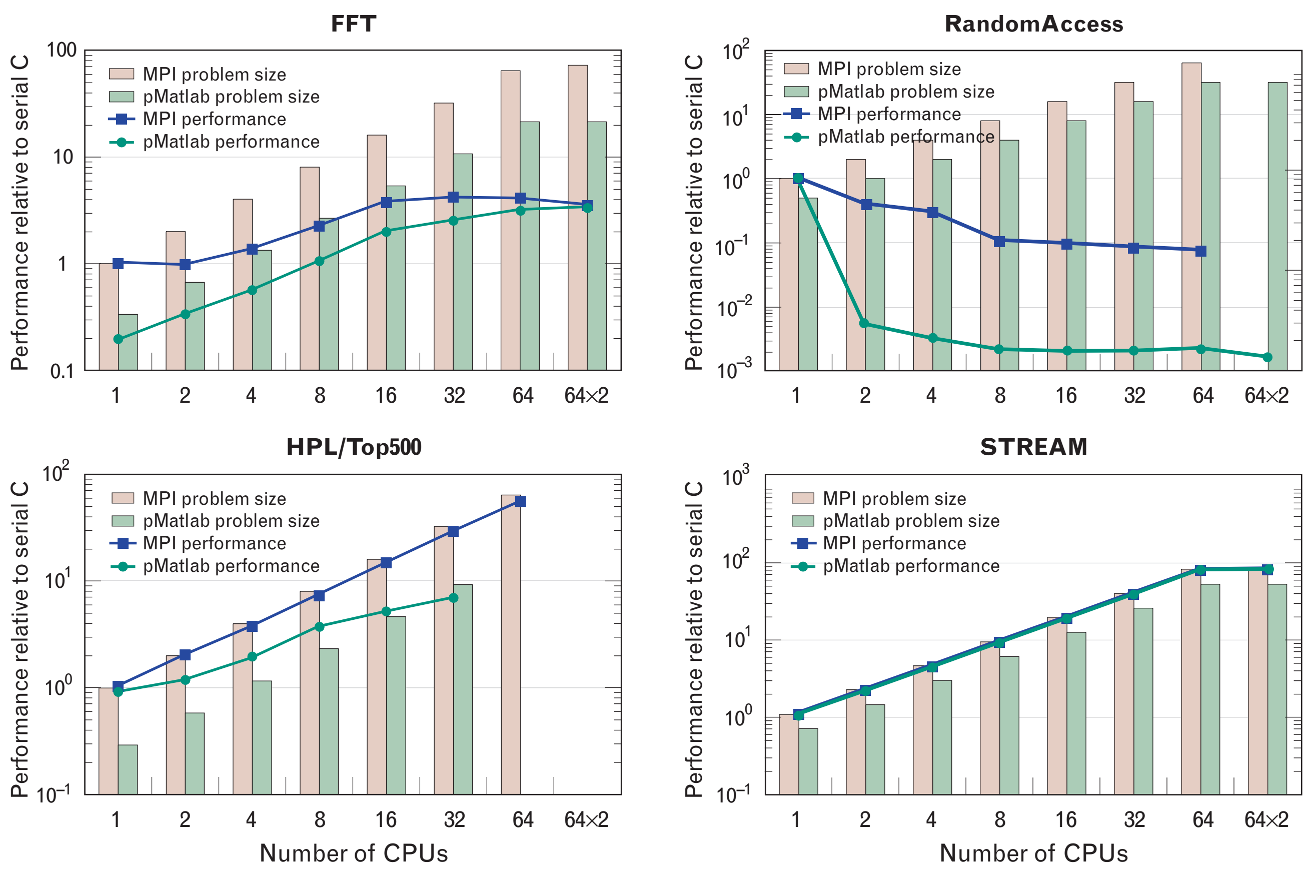


FIGURE 16. pMatlab and C+MPI HPC Challenge performance. pMatlab can run problems that are typically half the size of C+MPI problem size. pMatlab performance varies from being comparable to the C+MPI code (FFT and STREAM), to somewhat slower (Top500), to a lot slower (RandomAccess). The figure presents performance relative to the one-processor C+MPI case.

high-level programming environments. The performance of the pMatlab code is the same as the C+MPI code. This is because the MATLAB interpreter recognizes the scale-and-add statement and replaces it with a call to the appropriate optimized Basic Linear Algebra Subroutine (BLAS). The pMatlab code is about three times smaller than the C+MPI code due to the elimination of various *for* loops and the use of built-in MATLAB functions.

FFT

The FFT benchmark performs a complex-to-complex 1D FFT

$$\mathbf{Z} \leftarrow \mathrm{FFT}(\mathbf{z}),$$

where $\mathbf{Z}$ and $\mathbf{z}$ are m-element double-precision complex vectors, with the constraint

$$\mathrm{size}(\mathbf{z} + \mathbf{Z}) = 32m \text{ bytes} > \tfrac{1}{4} \text{ system memory.}$$

The input $\mathbf{z}$ should be in linear 'time' order. The output $\mathbf{Z}$ should be in standard frequency order. Any necessary reordering time should be included. Different implementations of the FFT algorithm may use optimizations to reduce the number of operations required. Regardless of the number of operations performed, however, the performance in gigaflops is reported by using the standard radix 2 FFT algorithm operations count:

$$\text{gigaflops} = 10^{-9} \times 5m \log_2(m)/\text{time}.$$

The standard parallel implementation of a 1D FFT performs two two-dimensional (2D) FFTs with a corner turn, or an all-to-all redistribution, between the two FFTs (see Figures 6 and 8). In our pMatlab implementation we deviated from the FFT specification in two ways. First, the input data is initialized by using a random selection of cosine and sine waves, which does not affect performance, but is a significant aid to

DARPA HIGH PRODUCTIVITY COMPUTING SYSTEMS PROGRAM

HIGH-PERFORMANCE computing has seen extraordinary growth in peak performance from megaflops to teraflops in the past decades, as illustrated in Figure A. This increase in performance has been accompanied by a large shift away from the original national security user base of the 1970s and 1980s to more commercially oriented applications such as bioinformatics, finance, and entertainment. In addition, there has been a significant increase in the difficulty of using these systems, which are now the domain of highly specialized experts.

In response to these trends, the DARPA High Productivity Computing Systems (HPCS) program was established to produce a new generation of economically viable, high-productivity computing systems for the national security and industrial user communities. The primary technical goals of the program are to produce petascale throughput computers that can better run national security applications and are usable by a broader range of scientists and engineers. The HPCS program is fostering many technological innovations, and one of the most important is the concept of a flatter memory hierarchy. The memory hierarchy refers to the levels of memory accessed by processors, which include (from nearest to farthest from the local processor) the local processor registers, local caches, local memory, the memory of other processors in the parallel system, and the storage subsystems.

Memory levels nearer to the processor have lower data-retrieval latency but have smaller storage capacity (see Figure 15 in the main text). Historically, the latency differences between accessing local processor registers, local cache, and local memory versus accessing remote memory and storage subsystems have been two or more orders of magnitude. Having a flatter memory hierarchy means a significant reduction in the differences in latencies between memory levels. Lowering the latency difference between the local and remote memory hierarchy levels is particularly important. This implies that the latency for retrieving data from remote memories is more commensurate with the access latency of local cache and memory. A flatter memory hierarchy will result in a significant performance increase (up to 2000 times) in certain important classes of applications that exploit random memory ac-

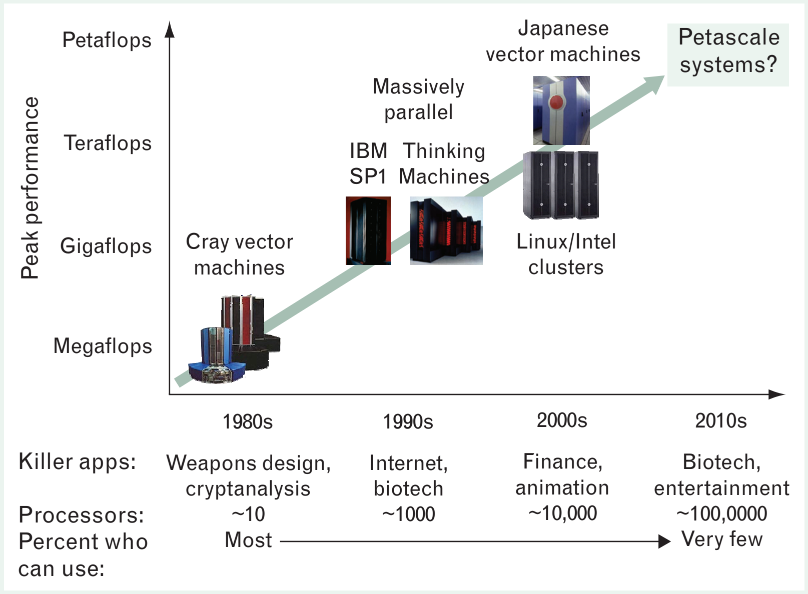


FIGURE A. Evolution of supercomputing. The performance of supercomputers has grown from megaflops to teraflops in three decades. Meanwhile, the user base of supercomputers has shifted from national security in the 1970s and 1980s to more commercially oriented applications such as animation and entertainment. The difficulty of using these systems has also increased. Currently, supercomputers are the domain of highly specialized experts.

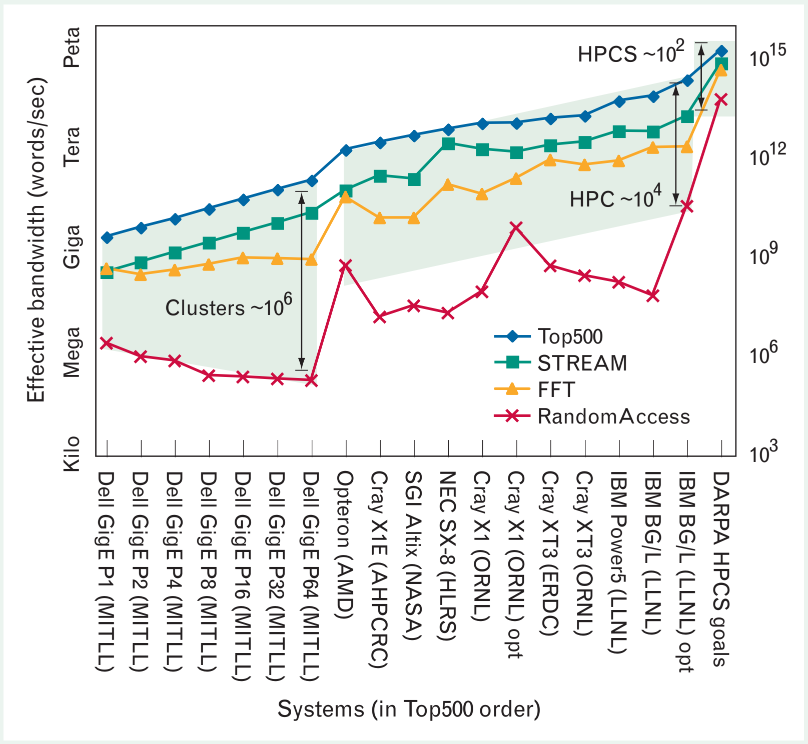


FIGURE B. HPC Challenge competition. The results illustrate three major points: (1) the memory hierarchy of clusters (the left most seven systems) gets more pronounced as the systems get larger; (2) current HPC systems (center to right) do a good job of keeping the memory latency difference constant as the machine gets larger; (3) HPCS performance targets (at the far right) will result in a system that is not only larger but will also have a memory hierarchy that is 100 times flatter than current HPC systems.

cesses. In addition, a flatter memory hierarchy is much easier to program because the users don't have to worry as much about precisely tailoring their applications to avoid the high latency cost of retrieving remote data.

To measure the performance of the memory hierarchy, the HPCS program has developed the HPC Challenge benchmark suite and has sponsored the HPC Challenge contest that awards a prize for the best performance on each benchmark. In addition, there is also a coding contest, which rewards the best and most clearly written implementations of the benchmarks. Figure B shows selected performance results from the first year of the contest. The flatness of the memory hierarchy is determined by the spread between the top points (Top500 result) and bottom points (RandomAccess result) for a given system.

These results illustrate three major points. First, the memory hierarchy of clusters (the leftmost seven systems) gets more pronounced as the systems get larger. This can be seen in the widening gap between the performance of the Top500 and the RandomAccess benchmarks as cluster systems get larger. Second, current HPC systems (eighth through eighteenth system) do a good job of keeping the memory latency difference constant as the machine gets larger. Third, the HPCS performance targets will result in a system that is not only larger than current systems, but will also have a memory hierarchy that is 100 times flatter than current HPC systems, as illustrated by the rightmost system.

debugging the code. Second, our implementation uses an ordering scheme that eliminates initial and final all-to-all communication steps, which is more consistent with the use of this function for most real applications and provides a better predictor of 2D and three-dimensional (3D) FFT performance. We have properly removed the time due to initial and final all-to-all steps in the C+MPI code so that a legitimate comparison can be made. The optimized pMatlab code (illustrated earlier in Figure 8) uses local arrays and the *transpose_grid* function with optimized message ordering.

The maximum problem size of the pMatlab code is 3.5 times smaller than the C+MPI code, which is due to the need to create intermediate temporary arrays. In addition, MATLAB internally uses a 'split' representation for complex data types, while the serial FFTW library being called uses an 'interleaved' representation. As a result, the data needs to be transformed between

Table 5(a). Maximum Problem Size Relative to the C+MPI Single Processor Case on 128 Processors

Implementation	*STREAM*	*FFT*	*RandomAccess*	*HPL(32)*
C+MPI/C serial	63.9	72.7	48	32.6
pMatlab/C serial	42.8	21.3	32	9.3
C+MPI/pMatlab	1.5	3.4	1.5	3.5

Table 5(b). Benchmark Performance Relative to the C+MPI Single Processor Case on 128 Processors

Implementation	*STREAM*	*FFT core*	*RandomAccess*	*HPL(32)*
C+MPI/C serial	62.4	4.6	7.4×10^{-2}	28.2
pMatlab/C serial	63.4	4.3	1.6×10^{-3}	6.8
C+MPI/pMatlab	1	1	46	4

Table 5(c). Code Size Comparisons *

Implementation	*STREAM*	*FFT*	*RandomAccess*	*HPL*
C+MPI	347	787	938	8800
pMatlab	119	78 **	157	190
C+MPI/pMatlab	3	10	6	40

* Code size is measured in terms of source lines of code. The parallel code sizes of the HPC Challenge C+MPI reference code are taken from the HPC Challenge FAQ.

** Includes code used to create random waves; does not include code for initial and final all-to-all operations. Combined, these should roughly offset each other.

these representations, which takes additional memory. On one processor the MATLAB FFT performance is about five times slower than the C code because of the time overhead required to perform the conversion between complex data storage formats. As the problem grows, the FFT time becomes dominated by the time to perform the all-to-all communication necessary between computation stages. Since these are primarily large messages, the performance of pMatlab becomes the same as the C+MPI code at large numbers of processors. The pMatlab code is about ten times smaller than the C+MPI code because of the use of built-in local FFT calls and the elimination of MPI messaging code.

RandomAccess

The RandomAccess benchmark generates a sequence of random array indices and uses these indices to update a large table. Let T be a table of size 2^m and let $\{a_i\}$ be a pseudorandom stream of 64-bit integers of length $2^m + 2$. Then for each a_i we update the table as follows:

$$T(\text{AND}(a_i, m-1)) \\ = \text{XOR}(T(\text{AND}(a_i, m-1), a_i) ,$$

with the additional constraints that each processor can buffer no more that 1024 updates and

$$\text{size}(T) = 8m \text{ bytes} > \tfrac{1}{4} \text{ system memory.}$$

The goal of the benchmark is to measure the rate at which atomic updates can be performed to global memory:

$$\text{number of giga-updates per second (GUPS)} = 10^{-9}\ N_{updates}/\text{time}\ ,$$

where $N_{updates}$ is the number of updates. RandomAccess requires communication patterns that are significantly more complicated than STREAM or FFT. In addition, communication is sufficiently fine-grained that there is significant overhead associated with computing global-to-local array index mappings every time a global array is accessed. Thus RandomAccess uses the pMatlab constructs to determine the global-to-local array index mappings once, but subsequently uses fragmented PGAS with direct message passing to perform the appropriate redistributions (see Figure 17 and Appendix B). This methodology allows us to implicitly exploit the fact that the array redistributions are static. For example, to minimize contention, each processor is able to compute in advance the optimal send order and optimal receive order of its messages. RandomAccess is a good illustration of how PGAS and messaging can work together to reduce the bookkeeping necessary for a parallel program, while still allowing a complex messaging scheme that is outside of the traditional PGAS formalism.

The maximum problem size of the pMatlab code is 1.5 times smaller than the C+MPI code, because of the need to create intermediate temporary arrays. On one processor the pMatlab RandomAccess performance is comparable to the C+MPI code. On larger numbers of processors, however, the pMatlab code is forty-five times slower than the RandomAccess code. This performance difference is due to the large latency of using file I/O for communicating small messages, which should be eliminated if pMatlab was built on a more traditional MPI implementation such as that used in DCT. The pMatlab code is six times smaller than the C+MPI code.

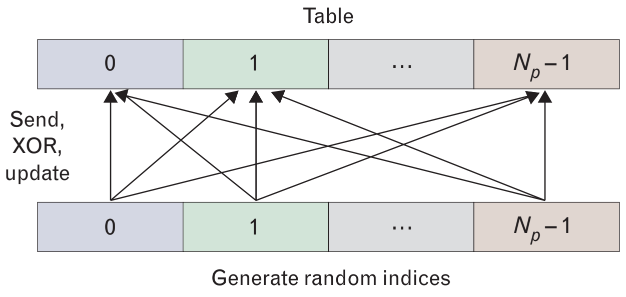


FIGURE 17. The RandomAccess benchmark generates a sequence of random array indices and uses these to update a large table. For code highlights, see Appendix B.

High Performance Linpack (Top500)

The High Performance Linpack (HPL) benchmark solves a dense linear system $\mathbf{Ax} = \mathbf{b}$ by using LU factorization with partial pivoting [38], where $\mathbf{b}$ is an n-element vector, and $\mathbf{A}$ is an $n \times n$ double-precision matrix with the constraint

$$\text{size}(\mathbf{A}) = 8n^2 \text{ bytes} > \tfrac{1}{2} \text{ system memory.}$$

The LU factorization is the dominant computation step in this algorithm and is principally made up of repeated matrix multiplies. The traditional parallel algorithm uses a sophisticated 2D block-cyclic distribution for the matrix $\mathbf{A}$ [39]. This algorithm has demonstrated very good performance even on computers with relatively slow networks. More recently it has become apparent that there is a complexity performance trade-off associated with using 2D block-cyclic distributions. Thus the pMatlab version uses a simpler but poorer-performing algorithm, with a 1D block distribution for $\mathbf{A}$ (see Figure 18 and Appendix B). The pMatlab code uses distributed arrays to break up the array and keep track of the various global indices. A key step in the algorithm requires broadcasting the results to a subset of the other processors, which is best done with a simple MPI multicast command.

The maximum problem size of the pMatlab code is 3.5 times smaller than the C+MPI code, which is due to the need to create intermediate temporary arrays. In particular, the lower and upper triangular matrices are returned as full matrices, while in the C+MPI code they can be merged into a single array. The pMatlab code provides a ten-times speedup on 32 processors, which is about four times slower than the C+MPI code. pMatlab achieves the performance limits of the 1D block algorithm on the system. Improving the network of this hardware should significantly improve the pMatlab code performance, relative to the C+MPI

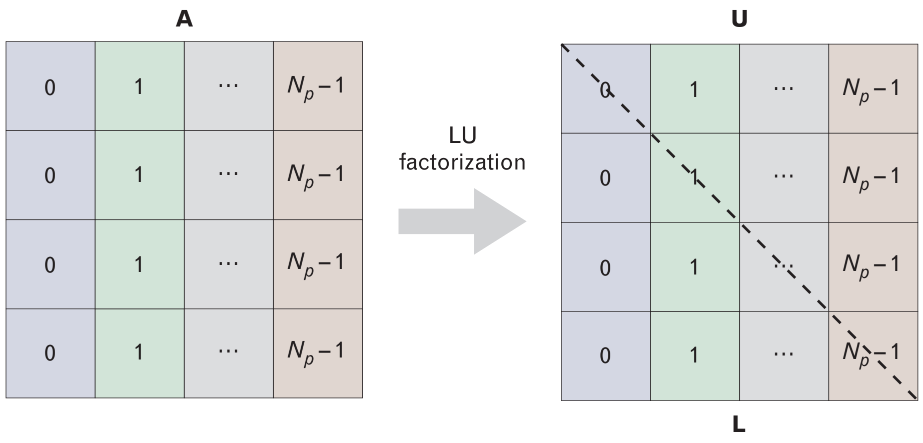


FIGURE 18. The High Performance Linpack (HPL) Top500 benchmark solves a dense linear system $\mathbf{Ax} = \mathbf{b}$ using LU factorization with partial pivoting, where $\mathbf{b}$ is an n-element vector, and $\mathbf{A}$ is an $n \times n$ double-precision matrix. For code highlights, see Appendix B.

code. The pMatlab code is forty times smaller than the C+MPI code. About 25% of this improvement is due to the higher-level abstractions from pMatlab, and about 10% is due to using the simpler algorithm.

Benchmark Performance Summary

Returning to our initial metrics we see that, relative to serial MATLAB, all the pMatlab codes allow problems sizes to scale linearly with the number of processors. Likewise, the pMatlab codes all experience significant performance improvements (with the exception of RandomAccess). Relative to C+MPI, the pMatlab problem sizes are smaller by a factor of two and the performance of pMatlab on both the STREAM and FFT is comparable.

Figure 19 shows one approach to summarizing the performance of the HPC Challenge benchmarks. The speedup and relative source lines of code (SLOC) for each implementation were calculated with respect to a serial C/Fortran implementation. In this plot we see that with the exception of RandomAccess, the C+MPI implementations all fall into the upper right quadrant of the graph, indicating that they deliver some level of parallel speedup, while requiring more SLOC than the serial code. Of course, the serial MATLAB implementations do not deliver any speedup, but they all do require fewer SLOC than the serial C/Fortran code. The pMatlab implementations (except RandomAccess) fall into the upper left quadrant of the graph, delivering parallel speedup while requiring fewer lines of code.

The LLGrid On-Demand Interactive Grid Computing System

While the pMatlab library is the part of LLGrid with which users interface the most, there are several other parts to the system that work behind the scenes to provide users with an easy-to-use high-performance computing capability. This section describes the other components of the LLGrid system in greater detail, which includes the LLGrid architecture, the gridMatlab toolbox, and the system management tools.

Grid Computing Architecture

Figure 20 shows the roadmap of the LLGrid cluster system, and Table 6 logs the pertinent data for each of these cluster improvements. The approach that the LLGrid team used was to learn as we went along. In early 2003, the team built several small four-to-eight processor clusters to gain experience and understanding in building cluster systems. After demonstrating the capability of launching MatlabMPI jobs onto the cluster using gridMatlab (see the section on the grid computing toolbox) to interface with a resource manager, and demonstrating the first prototypes of pMatlab in the fall of 2003, the team purchased their first set of cluster nodes, including 500 GB of shared storage, which was called the ‘gridsan’ shared file server. The 32 nodes (64 processors) were deployed to the first LLGrid users in January 2004, and they are depicted on the roadmap as the initial operating capability (IOC). As the user base of this system increased, another set of 48 nodes (96 processors) was added in July 2004. These 160 processors, which form the α-Grid system in the roadmap, were deployed in laboratory space in Lincoln Laboratory's S Building (illustrated in Figure 21). The improved cluster was used to learn about configuring and maintaining modestly heterogeneous compute nodes, scaling the shared file system, and adding users to the system.

The recent purchase of 150 compute servers (300 processors) and a high-performance parallel file system in April 2005 tripled the size of the available compute nodes. The parallel file system is comprised of a 36-terabyte Data Direct Networks (DDN) S2A 8500 storage-attached network (SAN) disk array connected to eight Dell PowerEdge 1750 servers running the IBRIX FusionFS parallel file system. The network switches are connected directly to the LLAN backbone via optical fiber cable so that users have the fastest possible network connection to the LLGrid hardware. This upgrade of compute servers and parallel file system helped the LLGrid team learn more about how to manage a relatively large cluster (we use the Rocks [40] cluster maintenance system) and parallel file systems, and how to enable parallel MATLAB jobs using over 32 processors. These systems are depicted in the roadmap as the β-Grid, and it was first installed in ten networking closets throughout S Building. In the first months of 2006, these systems were all moved into the F1 shared computing facility, a new facility built specifically to provide space to house computer systems for Laboratory projects. The F1 shared computing facility includes the necessary power and cooling to house a larger number of computer systems, removing the need to refit laboratory spaces across Lincoln Laboratory to accommodate computing systems. Figure 22 shows photographs of the exterior and interior of the F1 shared computing facility.

Figure 1, at the beginning of the article, shows the architecture of the LLGrid system, including the Linux compute nodes, the Linux service nodes, the network configuration, and the gridsan shared network storage. For the operating system on each of the computational and management nodes of the LLGrid cluster,

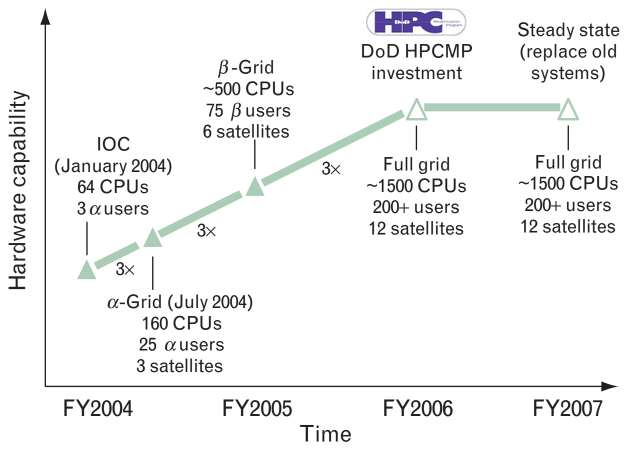


FIGURE 20. LLGrid hardware roadmap. The first LLGrid, the initial operating capability (IOC) grid system, started as 32 dual-processor nodes (64 processors) in early 2004. Later, 48 more dual-processor nodes (96 processors) were added to complete the 160-processor α-Grid system. In spring 2005 the team acquired 150 more compute servers (300 processors), which make up the β-Grid. Finally, we are in the process of acquiring an additional thousand processors, as part of the High Performance Computing Modernization Program (HPCMP), which will result in a full-grid system of nearly 1500 processors capable of serving over 200 users.

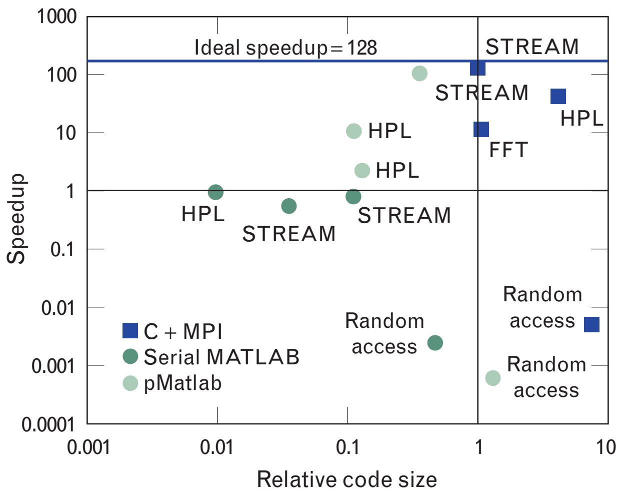


FIGURE 19. Speedup (relative to serial C) versus code size (relative to serial C). The upper right quadrant is the traditional HPC regime; more coding is required to give more performance and most of the C+MPI codes fall here. The lower left quadrant is the traditional regime of serial high-level languages that produce much smaller codes, but are slower. RandomAccess lies in the lower right and represents algorithms that are simply a poor match to the underlying hardware. The upper left quadrant is where most of the pMatlab implementations are found and represent smaller codes that are delivering some speedup.

Table 6. Technical Specifications of the LLGrid System

New Hardware	*IOC Grid*	*α-Grid*	*β-Grid*
Number of servers	32	48	150
Server	Dell PowerEdge 2650	Dell PowerEdge 1750	Dell PowerEdge 1855
Total processors in update set	64	96	300
Total processors in cluster	64	160	460
Processors	Dual Intel Xeon 2.8 GHz	Dual Intel Xeon 3.06 GHz	Dual Intel Xeon 3.2 GHz
RAM	4.0 GB	4.0 GB	6.0 GB
Hard drives	Two 36 GB	Two 36 GB	Two 144 GB
Network interfaces	Two 1-gigabit Ethernet	Two 1-gigabit Ethernet	Two 1-gigabit Ethernet
Network switches	Nortel BayStack 5510 gigabit switches	Nortel BayStack 5510 gigabit switches	Nortel BayStack 5510 gigabit switches
Shared file system	Dell PowerVault 220s SCSI-attached disk array	Same as IOC Grid	IBRIX Parallel File System with Data Direct Network S2A 8500
Shared file system capacity	500 GB	1.0 TB	32 TB
File system switch	None	None	Brocade Silkworm 6010 2-GB Fibre Channel switches

* The columns in the table correspond to the first three points on the hardware roadmap in Figure 20, while the rows provide details for each of the hardware components.

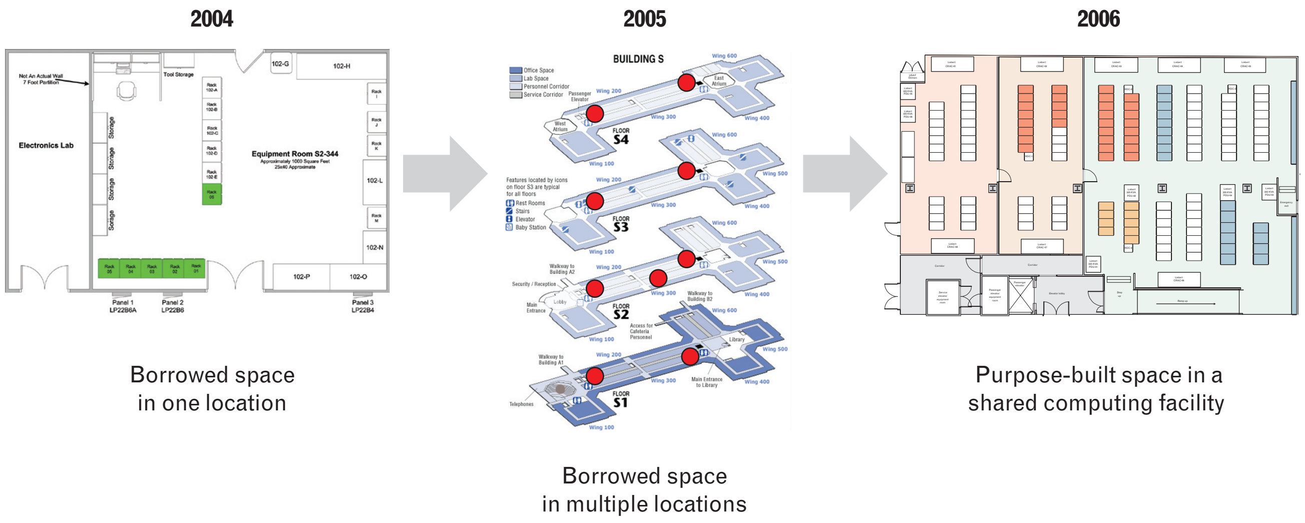

2004
Electronics Lab
Equipment Room S2-344
Borrowed space in one location
2005
BUILDING S
FLOOR S4
FLOOR S3
FLOOR S2
FLOOR S1
Borrowed space in multiple locations
2006
Purpose-built space in a shared computing facility

FIGURE 21. The accommodations of the LLGrid. In 2004, the first 160 processors of the LLGrid were installed in laboratory space in Lincoln Laboratory's S Building. In 2005, 300 processors were added to the first 160, and they were installed in ten network closets throughout S Building. In the first several months of 2006, the 300 processors in the network closets were moved to the shared computing facility in Building F. The remaining 160 processors will also be moved into the computing facility, along with (eventually) the HPCMP cluster (described in the section on future hardware).

FIGURE 22. Building F and the shared computing facility, built to provide space for Lincoln Laboratory's computing resources. The building supplies the necessary power and cooling to house a large number of computers and provides infrastructure for collateral compute resources. The LLGrid system was moved into Building F in the spring of 2006.

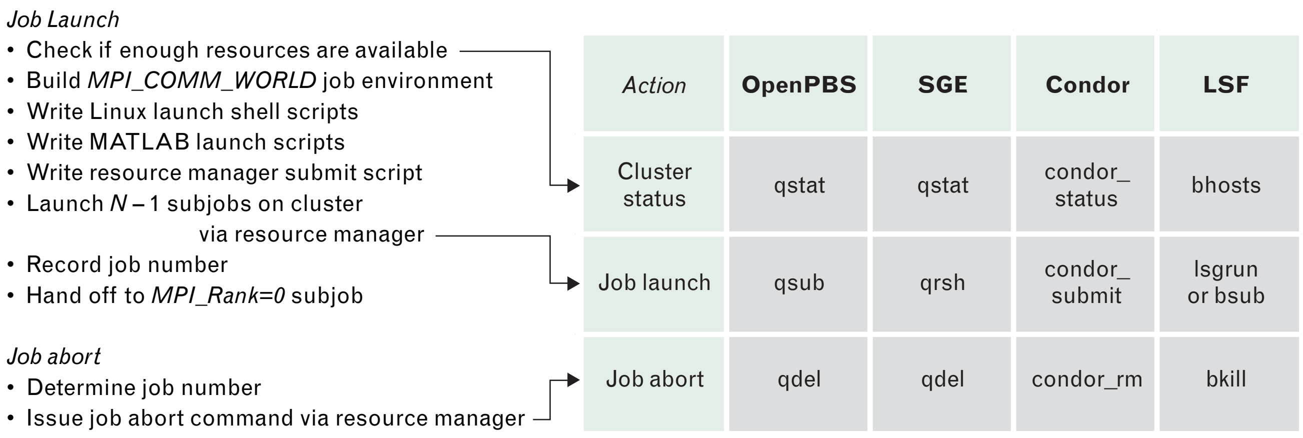
we chose Linux [41] because it is a capable operating system that enables distributed and parallel computing. The LLGrid system requires a single common file system that is mounted on the user's computer and on each of the computational nodes. This single common file system is required by MatlabMPI, as discussed earlier in the article. Windows and MacOS users mount the gridsan file server as a Samba share, while Linux and Unix users use Network File System (NFS).

Grid Computing Toolbox

In order for a user to interact with the parallel computing system, the LLGrid project has required several key innovations: (1) developing dynamic grid technology that automatically pulls the user's desktop system into the grid when a job is launched; (2) setting the user's desktop system to be the job leader node (MPI Rank 0), thereby making the user a full participating member of the grid computation; (3) providing a shared network file system as the primary user interface to the grid, and (4) integrating all services into a single one line *MPI_Run* command.

The gridMatlab toolbox enables these innovations [42]. It transparently integrates the MATLAB system on each user's desktop with the shared grid cluster through a cluster resource manager. When a user runs a MatlabMPI or pMatlab job in a desktop MATLAB session, the gridMatlab toolbox automatically amasses the requested LLGrid computational resources from the shared grid resources to process in parallel with the user's MATLAB session. Each of the parallel MATLAB processes communicate via MatlabMPI with one another and the user's desktop computer through the gridsan shared network file system. By integrating the user's MATLAB session into the set of grid cluster MATLAB sessions working on the user's code, the user interactively receives immediate feedback on the status of the job via text and graphics, thereby making the parallel MATLAB session virtually identical to running MATLAB code on a single computer.

The gridMatlab toolbox interfaces with an underlying resource manager for three activities: cluster status monitoring (how many processors are available), job launching, and job aborting. A resource manager oversees the compute servers in a cluster, and it matches up jobs with unoccupied processors in the computing system. Once a match up is made, the resource manager launches the processes of the job onto the processors. The resource manager can also report on the status of jobs and processors, and can terminate the execution of processes. Four of the most popular resource managers are Condor (University of Wisconsin), OpenPBS (open source) [43], Sun GridEngine (Sun MicroSystems, open source) [44], and LSF (Platform Computing). Figure 23 illustrates the commands used by these four different resource managers for cluster monitoring, job launching, and job aborting. Currently, the LLGrid system is bound to Con-


Job Launch
- Check if enough resources are available
- Build *MPI_COMM_WORLD* job environment
- Write Linux launch shell scripts
- Write MATLAB launch scripts
- Write resource manager submit script
- Launch *N* – 1 subjobs on cluster via resource manager
- Record job number
- Hand off to *MPI_Rank=0* subjob

Job abort
- Determine job number
- Issue job abort command via resource manager

Action	**OpenPBS**	**SGE**	**Condor**	**LSF**
Cluster status	qstat	qstat	condor_status	bhosts
Job launch	qsub	qrsh	condor_submit	lsgrun or bsub
Job abort	qdel	qdel	condor_rm	bkill

FIGURE 23. The gridMatlab launching mechanism. The figure depicts the steps that the gridMatlab toolbox takes to launch and abort pMatlab jobs along with the commands that are called in the bindings to four popular cluster resource managers.

dor as the resource manager. Condor was chosen for several reasons, including its free license model and its ease of installation, configuration, and maintenance. However, gridMatlab can also bind to LSF, OpenPBS, or Sun GridEngine.

Through calls to the resource manager, gridMatlab determines whether enough resources are available to satisfy the user's request, thereby enabling on-demand resource allocation. On the occasion when there are insufficient resources, the gridMatlab toolbox negotiates with the user whether the job can be run on fewer processors or whether the job should be resubmitted later. The system has been sized, however, so that such a situation rarely occurs. This scalability is one of the key differences between the LLGrid and typical supercomputing centers. Other systems measure success by showing maximally high utilization numbers to their sponsors. In other words, every processor needs to be occupied as much as possible. In order to achieve this maximum utilization, a queue is required from which to launch a new job when an executing job finishes. As a result, users must submit their job to the queue and wait an unspecified time until it executes. The philosophy of the LLGrid team is that hardware is inexpensive in comparison to a user's time and salary. LLGrid is sized such that on average half of all of the processors are being used at any one time. This ensures that at peak usage times, there are enough processors available to minimize the number of rejections. Each user is advised of the maximum number of processors that are to be used for a conventional job on the LLGrid system. If a user needs more than this maximum number of processors, special arrangements can be made.

No matter which resource manager is used in the LLGrid system, the key innovation of gridMatlab is that it abstracts away all of the user interaction with the on-demand resource manager while it draws the user's computer into the computational pool so that the user's execution environment is consistent from serial execution to interactive LLGrid parallel execution.

System Management Tools

The LLGrid system management tools implement a number of capabilities that make system maintenance and usage easier for the LLGrid team and the LLGrid users. The system management tools include the Rocks cluster provisioning system, the LLGrid website, and some internally developed tools.

The Rocks cluster provisioning system provides an automated facility for installing a Linux operating system, compilers, applications, monitoring tools, resource manager, file systems, and configuration files onto compute servers [40]. Each system 'appliance' type (e.g., compute node, web server node, resource manger master node) is described by an XML file that specifies what software bundles are installed on it. Thus a compute node appliance can be customized with certain software and configurations, while a service node like the resource manager master node can be customized with a different set of software and

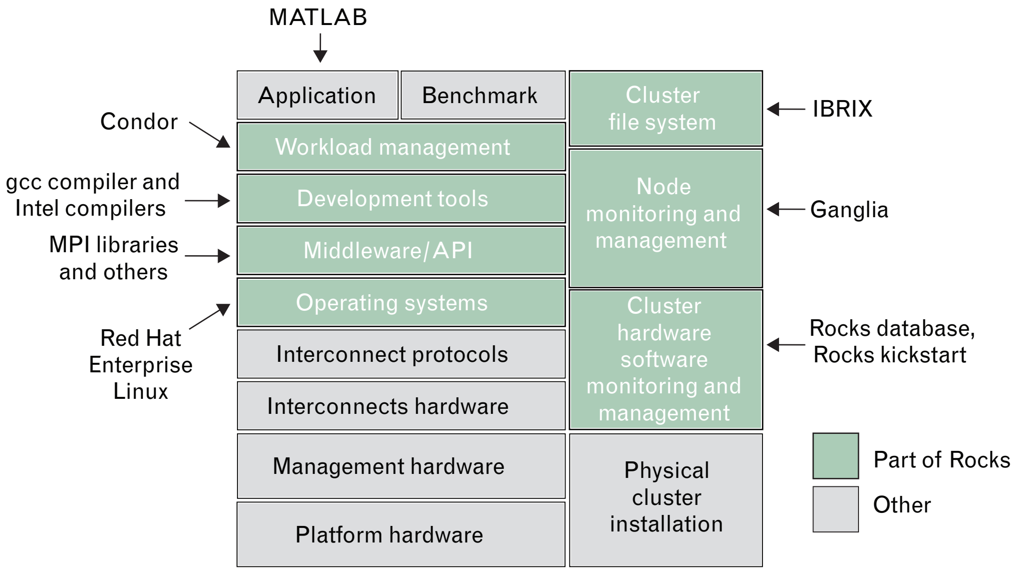


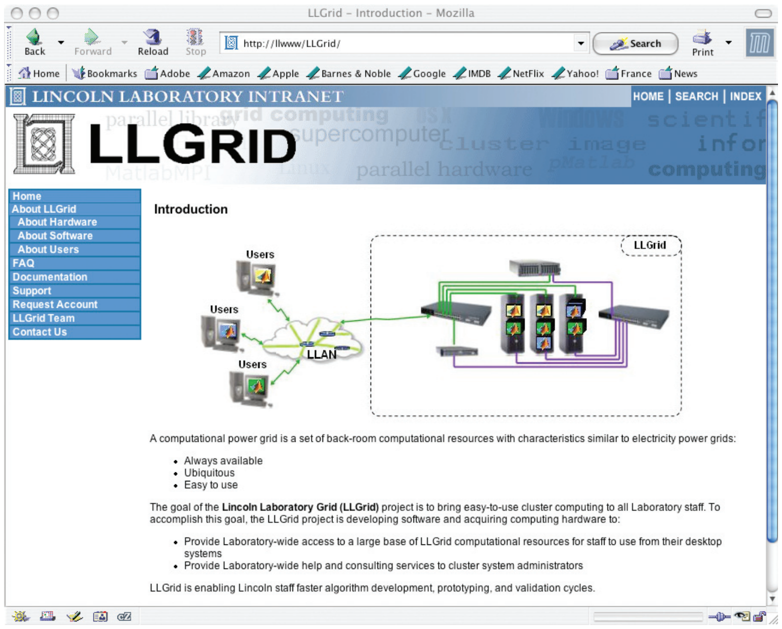
FIGURE 24. The parts of the LLGrid cluster system that the Rocks cluster provisioning system automates for installation. Rocks is the framework with which the LLGrid team builds the LLGrid system, as well as cloned satellite clusters of the LLGrid system for other projects within the Laboratory.

configurations. Figure 24 shows the LLGrid components of the cluster installation that Rocks automates, including the Red Hat Linux operating system, MATLAB, the IBRIX parallel file system, the Condor resource manager, various compilers and libraries, and the Ganglia cluster monitor. Rocks is the framework with which the LLGrid team has built the LLGrid system and has cloned satellite clusters of the LLGrid for restricted projects within the Laboratory. Without Rocks, reconfiguration and reinstallation of all the compute nodes in the LLGrid takes four hours; with Rocks this task can be completed in less than fifteen minutes.

The LLGrid website serves three fundamental purposes: (1) presenting the front end to the account creation script on the grid cluster; (2) providing users with the status of the entire grid cluster and each individual computation node; and (3) providing online documentation on pMatlab, the LLGrid system, and descriptions of other users' projects. Figure 25 shows the LLGrid home page. Accounts are created by using a simple web interface, which eliminates at this stage the need for a system-administrator level of expertise. After the appropriate information is entered via the interface, a Perl CGI script creates the user's account and establishes the user's secure shell (SSH) certificates, toolbox links, and sample pMatlab scripts and tutorials.

Once the account is created, another web page provides a hyperlink to a grid cluster monitoring website, and a hyperlink to a user system configuration script that mounts the LLGrid file system in the user's environment, configures the user's SSH encryption keys, writes the proper LLGrid configuration file, and adds the gridMatlab, pMatlab, and MatlabMPI toolbox paths to the user's MATLAB environment. The user downloads this system configuration script onto his or her desktop and runs it once. Within minutes the LLGrid file system is mounted and the user is ready to run his or her first LLGrid parallel job. To provide users the status of the grid cluster, we use Ganglia [45] because it is efficient and easy to install and maintain, and it presents information about the computational nodes in an easily understood format. Figure 26 shows a screenshot of the Ganglia web page.

Finally, the LLGrid team has built a number of tools. Among these is the *gridgui*, which interfaces with the resource manager and displays the processors that are currently occupied with jobs in the cluster, the user job associated with each processor, and the most recent jobs that have run on the LLGrid. Figure 27 shows a screenshot of the *gridgui* tool.



Introduction

A computational power grid is a set of back-room computational resources with characteristics similar to electricity power grids:

- Always available
- Ubiquitous
- Easy to use

The goal of the **Lincoln Laboratory Grid (LLGrid)** project is to bring easy-to-use cluster computing to all Laboratory staff. To accomplish this goal, the LLGrid project is developing software and acquiring computing hardware to:

- Provide Laboratory-wide access to a large base of LLGrid computational resources for staff to use from their desktop systems
- Provide Laboratory-wide help and consulting services to cluster system administrators

LLGrid is enabling Lincoln staff faster algorithm development, prototyping, and validation cycles.

FIGURE 25. The LLGrid home page on the internal Lincoln Laboratory intranet. This page provides LLGrid users with grid cluster status and software documentation. The page also has an interface that allows quick account creation without requiring a system-administrator level of expertise.

User Experience

The true measure of success for any technology is its effectiveness for real users. The initial LLGrid system has been operational for over two years with 160 processors and approximately a hundred users. The engineers and scientists at Lincoln Laboratory use the LLGrid to rapidly prototype their ideas. The system is very flexible, and its users are working on a wide variety of technologies, including analysis of hyperspectral images from satellites, simulation of laser beam dispersion mitigation in the atmosphere, and simulation of Navy target-tracking communication systems. The LLGrid has run over 37,000 processor days of computation, as of February 2006. The log files of this activity are filled with interesting usage results, and interviews with users have further enhanced the utility of the system.

Figure 28 shows a scatter plot of the execution time of these parallel jobs versus the number of processors per job. Each dot is a job that was run on the α-Grid cluster. The plot is partitioned into three sections: the bottom left, the top half, and the bottom right. On the bottom left are jobs that were run on fewer than ten processors and executed in less than eight CPU hours. These parallel jobs probably could have been executed on a desktop computer in a serial manner. Most of these jobs, however, were probably used for debugging and validating algorithms subsequently targeted for longer execution runs. The top half of the plot identifies jobs that ran in parallel for more than eight CPU hours on anywhere from two to a hundred processors. These jobs clearly required HPC resources to execute, and probably would have been acceptable to run in a batch queue system. The bottom right of the plot shows jobs that executed on more than eight processors in less than an hour. Because of the relatively quick turnaround time of these jobs, they required interactive, on-demand HPC access to effectively employ the users' time. The shaded overlay on the graph shows the jobs enabled by the LLGrid system that would not

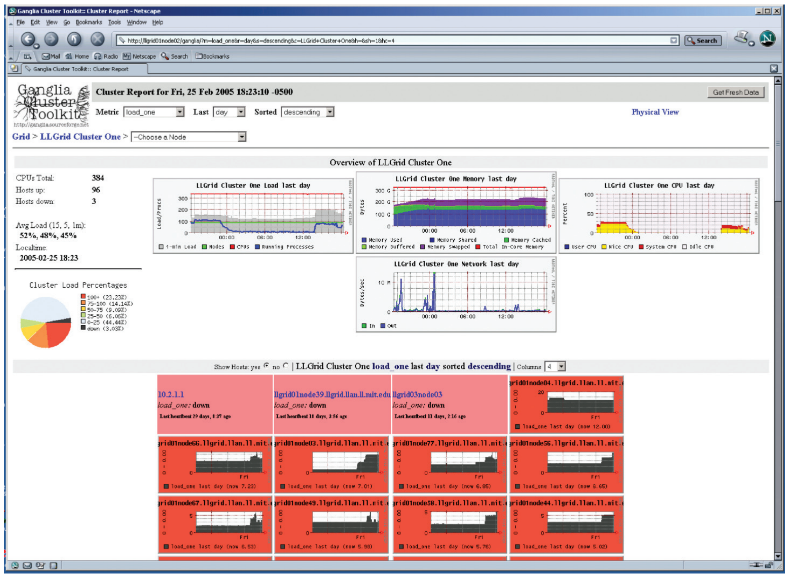
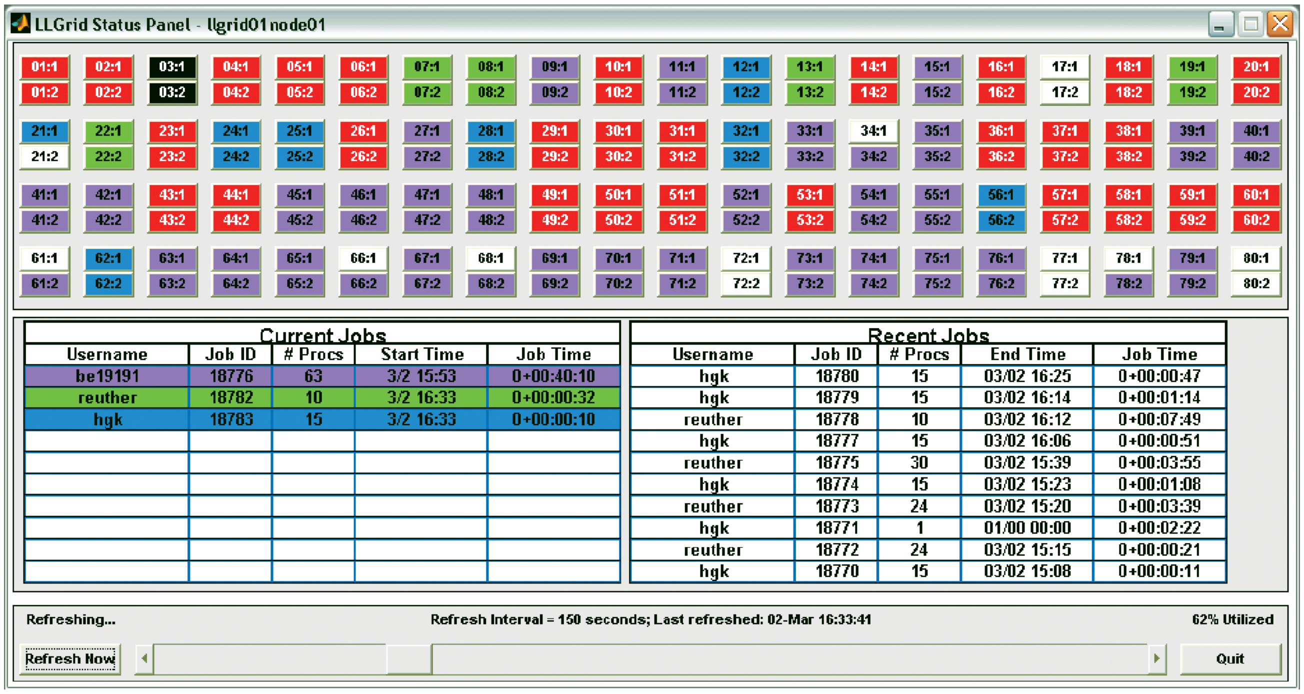

Ganglia Cluster Toolkit: Cluster Report - Netscape
Search
Ganglia Cluster Toolkit
Cluster Report for Fri, 25 Feb 2005 18:23:10 -0500
Get Fresh Data
Metric load_one
Last day
Sorted descending
Physical View
Grid > LLGrid Cluster One > --Choose a Node
Overview of LLGrid Cluster One
CPUs Total: 384
Hosts up: 96
Hosts down: 3
Avg Load (15, 5, 1m): 52%, 48%, 45%
Localtime: 2005-02-25 18:23
LLGrid Cluster One Load last day
LLGrid Cluster One Memory last day
LLGrid Cluster One CPU last day
LLGrid Cluster One Network last day
Cluster Load Percentages
LLGrid Cluster One load_one last day sorted descending
load_one: down

FIGURE 26. The Ganglia cluster monitoring tool allows LLGrid users to view the status of the LLGrid cluster. Ganglia provides information such as the CPU load, network load, memory usage, and disk usage.


LLGrid Status Panel - llgrid01node01

Current Jobs

Username	Job ID	# Procs	Start Time	Job Time
be19191	18776	63	3/2 15:53	0+00:40:10
reuther	18782	10	3/2 16:33	0+00:00:32
hgk	18783	15	3/2 16:33	0+00:00:10

Recent Jobs

Username	Job ID	# Procs	End Time	Job Time
hgk	18780	15	03/02 16:25	0+00:00:47
hgk	18779	15	03/02 16:14	0+00:01:14
reuther	18778	10	03/02 16:12	0+00:07:49
hgk	18777	15	03/02 16:06	0+00:00:51
reuther	18775	30	03/02 15:39	0+00:03:55
hgk	18774	15	03/02 15:23	0+00:01:08
reuther	18773	24	03/02 15:20	0+00:03:39
hgk	18771	1	01/00 00:00	0+00:02:22
reuther	18772	24	03/02 15:15	0+00:00:21
hgk	18770	15	03/02 15:08	0+00:00:11

Refreshing... Refresh Interval = 150 seconds; Last refreshed: 02-Mar 16:33:41 62% Utilized

FIGURE 27. A screenshot of the *gridgui* tool, which shows the occupied processors on the LLGrid system and the user jobs associated with each processor.

have been easily executable or even possible on a serial desktop computer.

Table 7 provides a number of interesting statistics from the usage of the LLGrid system from December 2003 to February 2006. First, the median number of CPUs used for a single job was 17, while the mean number of CPUs used for a single job was 22. More interesting, though, is the median and mean job execution time: the median job duration is just 84 seconds, while the mean job duration is 31 minutes. Clearly, users running jobs that would ordinarily need more than eight hours to execute on a desktop computer are typically executing on 32 or fewer processors in less than thirty minutes. Some users are executing

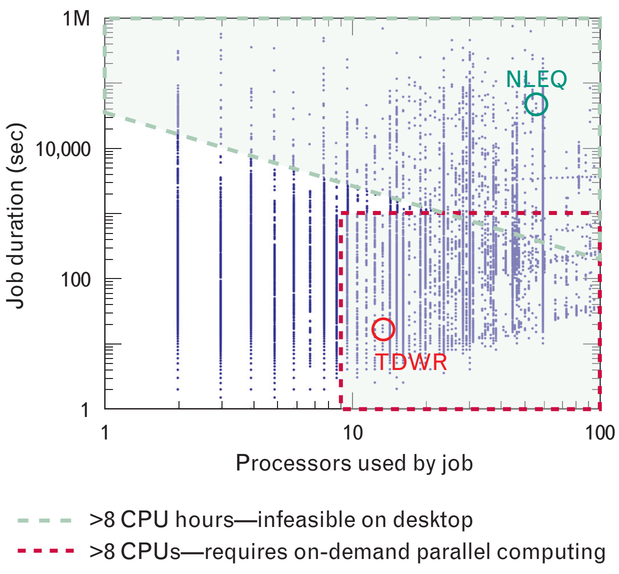


FIGURE 28. LLGrid usage. Scatter plot of the number of processors per job (*x*-axis) versus the duration of the job per second (*y*-axis). Each dot in the plot represents a job that was run on the α LLGrid cluster. The three regions in the plot illustrate different modes of usage. On the bottom left are the jobs that used fewer than ten processors and executed in less than eight CPU hours. Most of these jobs can be executed on a desktop computer and are most likely debugging runs in preparation for larger runs. In the top half are jobs that required more than eight CPU hours on from two to a hundred processors. These jobs could have been run in a batch queue regime and are characteristic of traditional jobs at a supercomputer center. On the bottom right are the jobs that are unique to the LLGrid system. These jobs, which used more than ten processors and executed in less than an hour, required the on-demand, interactive capability that the LLGrid system provides. Two particular projects discussed in detail are highlighted—nonlinear equalization (NLEQ) and Terminal Doppler Weather Radar (TDWR).

Table 7. LLGrid System Usage Statistics as of February 2006

Total jobs run	37,101
Median CPUs for single job	17
Mean CPUs for single job	22
Maximum CPUs for single job	144
Total CPU time	21,421 days 6 hours
Median job duration	84 sec
Mean job duration	31 min 17 sec
Max job duration	7 days 7 hours 2 min

debugging runs in preparation for longer parallel execution runs, while other users are running simulations in parallel that complete in less than half an hour. These same simulations would have taken many hours on a desktop computer.

There are many examples of how various research programs at the Laboratory are using the LLGrid system. Two of these projects are highlighted here.

The Terminal Doppler Weather Radar (TDWR) Data Quality Improvement program is developing signal processing algorithms to mitigate range-velocity ambiguity [46]. For instance, gust fronts that are within close range of the radar can be obscured by range-overlaid signal from distant out-of-trip weather using the current operational algorithm. The new algorithms are able to discriminate these gust fronts, which will be a significant improvement for helping pilots and control towers understand the weather in which they are flying and directing airplanes. The program team needed to rapidly write, evaluate, and revise its algorithms, which were written in MATLAB. Running the algorithms on simulation datasets on a desktop workstation typically executed for eight to ten hours. The results of each simulation direct the parameter and algorithm development choices for subsequent simulations, and they usually could execute only two of these simulations in a 24-hour period—one over the course of the business day and one overnight. That is, they were only executing two engineering turns per day. After parallelizing the simulations, the team now runs the simulations on the desktop machines with eight to sixteen LLGrid processors coming alongside the desktop computer to complete the computations. These simulations now complete in thirty to sixty minutes, affording between eight and ten engineering turns per day. Figure 29 illustrates the usage patterns and Figure 30 demonstrates data improvement.

Another example is the DARPA-sponsored Nonlinear Equalization (NLEQ) program, which is developing an application-specific integrated circuit (ASIC) that will improve signal-to-noise and distortion ratios of radar signals up to 10 dB, as illustrated in Figure 31. Phased-array radar systems have many analog-to-digital converters to improve dynamic range. Increasing the dynamic range lowers the noise floor and causes nonlinearities to rise above it. Nonlinearities become an issue for Department of Defense (DoD) applications that require detections of faint signals. The NLEQ program is developing algorithms to reduce nonlinearities to below the noise level.

In order to capture these algorithms in an ASIC, a huge number of simulations are required to converge

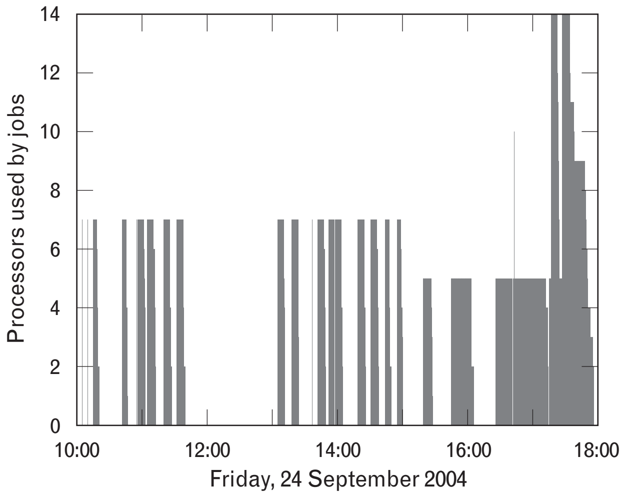


FIGURE 29. LLGrid processor usage data for the Terminal Doppler Weather Radar (TDWR) project. Development of TDWR algorithms required changing parameters on the basis of the results of the previous algorithm analysis runs. Thus one run had to be completed before the next one could start. A single run on a single processor took approximately eight to ten hours, while using eight to sixteen LLGrid processors reduced the run time to thirty to sixty minutes. The LLGrid system allowed the algorithm developer to raise the number of engineering turns per 24-hour day from two to ten.

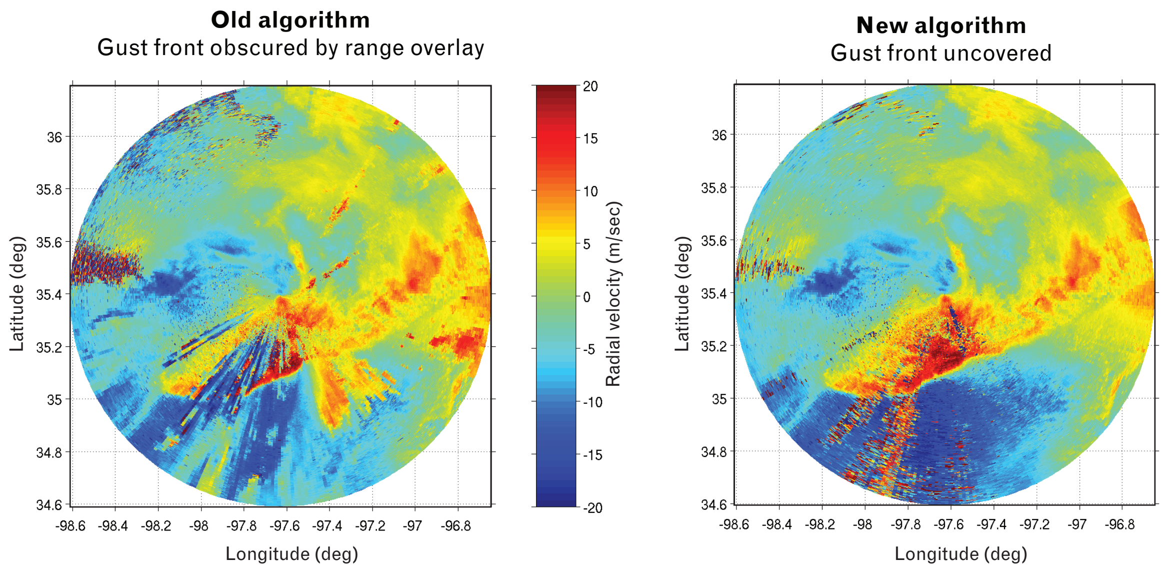


FIGURE 30. Improvement in TDWR algorithms. This figure depicts weather data improvements enabled by computations executed on the LLGrid. After repeated runs, followed by parameter selection, the algorithms are able to discriminate gust fronts from weather that is not moving.

on the proper set of coefficients for the best filtering performance. That is where LLGrid comes in; the simulations can each consume up to 750 hours of computational time, which are run on nights and weekends on 64 or more processors. The long simulation runs are often set by executing shorter ten-to-thirty-minute practice runs on 32 or fewer processors during the day. This provides greater confidence that the long runs will produce the expected results.

Figure 32 is a usage plot of the project's typical daily activity on the LLGrid system. It plots the time of the day on the *x*-axis versus the number of processors that this individual was using in an interactive, on-demand fashion. The work on the LLGrid system started after ten a.m., when several short 64-processor jobs were run to post-process and validate results from a previous run. During the early afternoon a four-processor job executed for more than two hours and prepared parameters for later in the day. In the late afternoon, several more short-duration 64-processor debugging runs were executed in preparation for long runs that started just after six p.m. Being able to debug algorithms and validate results in an interactive, on-demand fashion facilitates more rapid time to results both with few

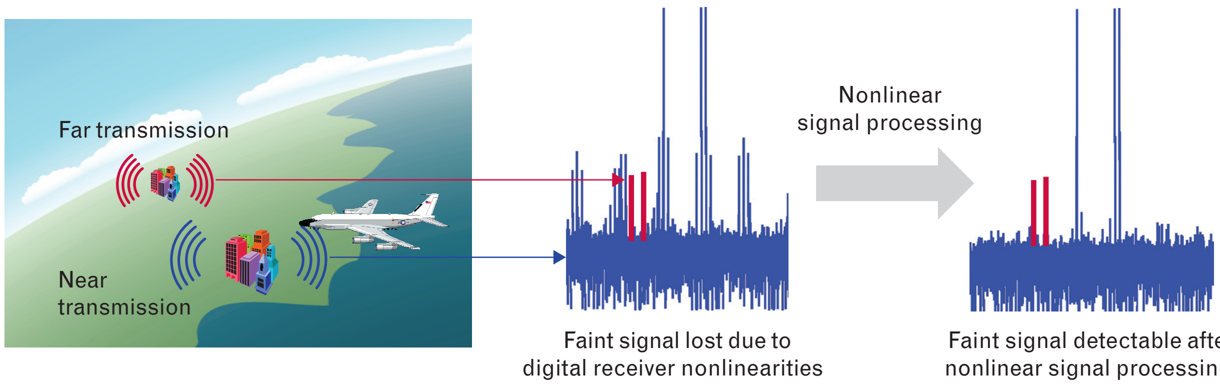

Far transmission
Near transmission
Nonlinear signal processing
Faint signal lost due to digital receiver nonlinearities
Faint signal detectable after nonlinear signal processing

FIGURE 31. Nonlinear equalization (NLEQ). The DARPA NLEQ program is developing algorithms to counter nonlinearities in signal processing. The NLEQ team is also developing an application-specific integrated circuit that will improve signal-to-noise and distortion ratios of radar signals up to 10 dB.

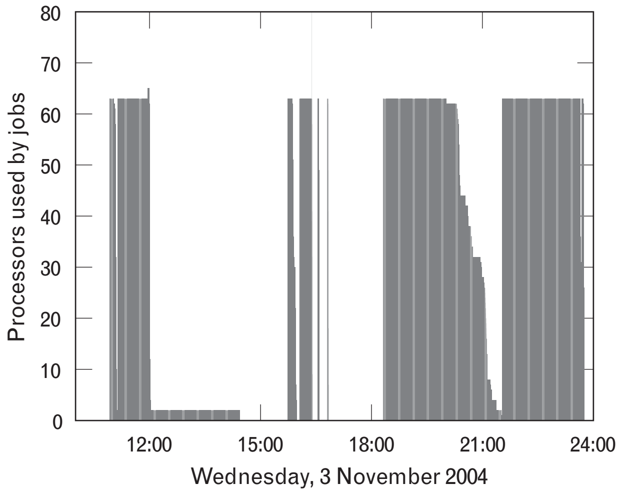


FIGURE 32. LLGrid usage data for the NLEQ project, which requires a large number of simulations to converge on a proper set of coefficients for the NLEQ application-specific integrated circuit. These large simulations can take up to 750 hours of computation time, and run on 64 or more processors. In addition, algorithm developers usually require many practice runs on 32 or fewer processors.

processors and with a full set of processors, and it enables this project to run a set of long simulations every night rather than every two or three nights.

These examples typify the value of the interactive, on-demand capabilities of the LLGrid system in enabling Lincoln Laboratory engineers and scientists to work more effectively and efficiently. However, there are a larger number of Laboratory staff members that have computationally demanding, non-MATLAB applications or require specialized cluster computing systems. The sidebar entitled "The LLGrid Project: Enterprise Cluster Computing Services" provides a description of the LLGrid project's non-MATLAB users.

Another aspect of the user experience is how easily they are able to take their serial code and make it into parallel code to run on the LLGrid system. Table 8 highlights several projects representative of the user base. Of particular interest are the columns showing the time to parallelize and what parallelization enables. The time to parallelize shows how quickly MATLAB code can be converted from serial code to parallel code as well as how quickly the user is able to get the parallel code running on the LLGrid.

The Future of Lincoln Laboratory Grid Computing

The current LLGrid system is providing on-demand, interactive, computational resources to Lincoln Laboratory technical staff. The system currently consists of 460 processors, has performed over 37,000 processor days of computation, and supports approximately 100

Table 8. Selected pMatlab Applications*

Code description	*Serial / parallel dev time (hours)*	*Parallelization enables more or faster*
Missile and sensor simulations	2000 / 8	Higher fidelity radar
First-principles ladar	1300 / 1	Speckle image simulations
Analytic TOM leakage	40 / 0.4	Parameter space studies
Hercules metric TOM	900 / 0.75	Monte Carlo simulation
Coherent laser propagation	40 / 1	Run time
Polynomial coefficient approximation	700 / 8	Faster training algorithm
Ground motion tracker	600 / 3	Faster and larger datasets
Automatic target recognition	650 / 40	Target classes and scenarios
Hyperspectral image analysis	960 / 6	Larger datasets of images

* The first and last columns provide a brief description of the code and what the parallel version of the code has enabled. The middle column shows estimated time to write the original serial code and the additional time to parallelize the code with pMatlab and get it running well on the LLGrid system.

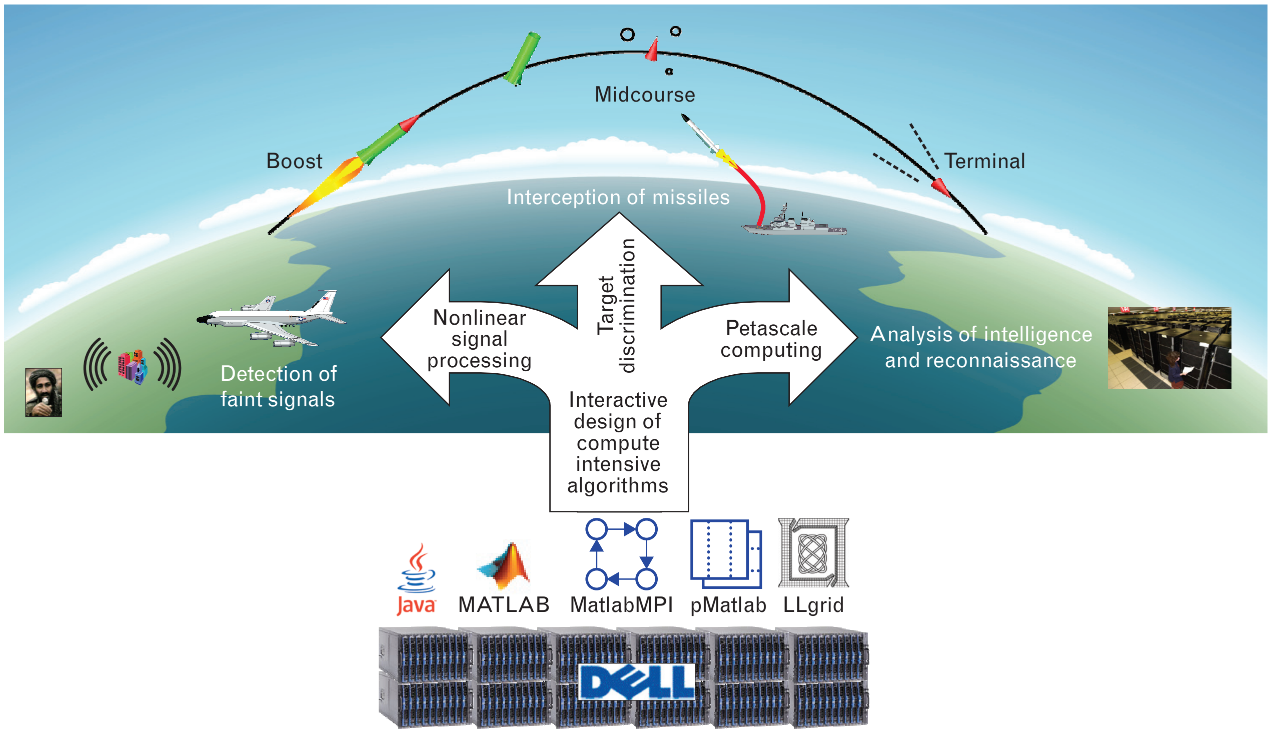

Midcourse
Boost
Terminal
Interception of missiles
Target discrimination
Nonlinear signal processing
Petascale computing
Analysis of intelligence and reconnaissance
Detection of faint signals
Interactive design of compute intensive algorithms
Java
MATLAB
MatlabMPI
pMatlab
LLgrid
DELL

FIGURE 33. HPCMP system to support three large programs. The 1000-processor system will be principally used by the DARPA NLEQ Program, MDA Project Hercules, and the DARPA High Productivity Computing Systems (HPCS) program. Providing these programs access to such a powerful computational resource will accelerate development and testing of algorithms critical to defense against weapons of mass destruction.

users. Over the next few years, we see the following areas of need. First, we expect the user base to at least double to 200 users. Second, the system has to be able to scale to support special users that have extremely high computational demand. Finally, as the system grows, the software infrastructure needs to be easier to use at full scale. The following sections address how we plan to address these needs.

Future Hardware

Recently, Lincoln Laboratory acquired a 1000-processor computer system as part of the DoD High Performance Computing Modernization Program (HPCMP). This system will be principally used by three large programs that have extremely high computational needs: DARPA NLEQ for detection of faint signals, MDA Project Hercules for target discrimination, and DARPA High Productivity Computing Systems (HPCS) program for petascale computing [47]. Figure 33 illustrates these three research programs. Allowing these programs access to such a powerful computing resource will significantly accelerate development and testing of algorithms critical to defense against weapons of mass destruction.

The full HPCMP system will consist of 1000 processors with one petabyte of storage. The system will be housed in Building F at Lincoln Laboratory, as illustrated in Figure 34, and will allow for collateral secret computations on a subset of the processors. While the principal purpose of the additional 1000-processor system is supporting the three large programs mentioned above, the rest of the LLGrid user base could also use the system during downtime. Additionally, the LLGrid team is constantly working on upgrading the entire system to support the growing user base.

Future Software

The LLGrid and the associated toolboxes discussed in this article provide the users with the ability to run parallel applications in an on-demand and interactive manner. However, the current software has some limitations. First, very large datasets are still not accom-

THE LLGRID PROJECT: ENTERPRISE CLUSTER COMPUTING SERVICES

When the LLGrid project was initially formed, the team was aware of the diverse Laboratory computing requirements. Many projects develop software applications using technologies other than MATLAB. Consequently, the LLGrid team has designed the LLGrid as an enterprise cluster—available across the Laboratory—that can accommodate applications written with a number of languages and technologies. This cluster is a core service, funded and maintained by the Laboratory.

Additionally, a number of projects have special computing needs that cannot be satisfied by the LLGrid cluster. Such projects may require clusters built for their specific needs. Typically, members of these projects have little experience with cluster computing. The LLGrid team provides cluster consulting services to assist such projects.

Cluster Computing Using the LLGrid Enterprise Cluster

The LLGrid enterprise cluster has the capability to run applications written with many different technologies. In addition to pMatlab and MatlabMPI, these technologies include the MATLAB Distributed Computing Toolbox, Message Passing Interface (MPI) and Java. A number of projects have used the LLGrid system in such a capacity.

For example, one of the efforts of the Weather Sensing group was to develop an optimized HPC solution for generating high-resolution numerical weather prediction forecasts for experimental FAA air traffic management decision guidance tools. The LLGrid cluster was used to examine various configurations of the numerical weather prediction model (e.g., Fortran compilers, MPI implementations, and I/O subsystems) to quantify the model and software configuration settings that can enhance simulation speed [1]. Figure A shows an example of how a weather forecast region can be divided into sub-domains, with each sub-domain assigned to a different processor.

In another example, the Directed Energy group has used the LLGrid cluster to support a number of projects on laser beam propagation and incoherent imaging through turbulent atmosphere. A Laboratory program called Parallel Optical Propagation Software (POPS) models beam propagation and adaptive optics servos. Recently, POPS has been used to support the Directed Energy group's Target-in-Loop adaptive optics program and the Communications and Information Technology division's High-Altitude Pseudo-Satellite program for developing free space communication between planes and satellites.

The LLGrid team encourages Laboratory staff members to utilize LLGrid resources. In general, projects can use the LLGrid cluster rather than purchasing, building, and main-

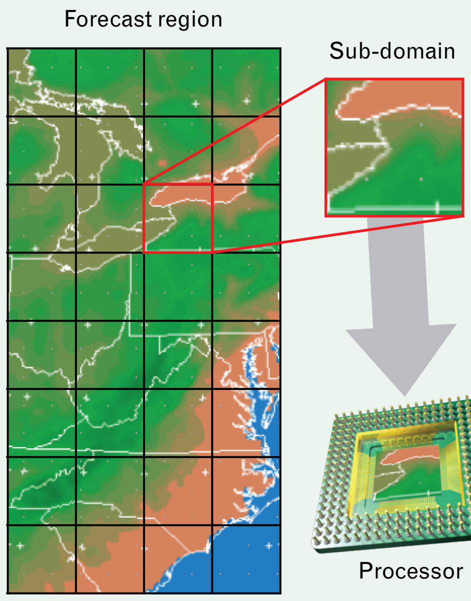


FIGURE A. A sample distribution of a weather forecast region. This example has both row and column blocked distributions. Alternate distributions include row blocked or column blocked.

taining their own cluster. Consequently, projects that use the LLGrid cluster can dedicate more of their funds and staff time to their respective research needs. However, there are cases in which building a cluster dedicated for a specific project may be necessary. In these instances, projects can benefit from the LLGrid project's cluster consulting services.

Cluster Consulting Services

Many projects have unique processing requirements and are unable to take advantage of the LLGrid enterprise cluster, such as those in closed areas and those that require real-time, mobile, or remote processing capabilities. These projects can benefit from the LLGrid project's cluster consulting services, which assists groups with purchasing and building clusters tailored to their specific needs. These services include (1) assisting with selecting and purchasing hardware and software; (2) providing a standard cluster image, including a preconfigured operating system and automated cluster management software; (3) training on building and maintaining clusters; and (4) assisting with specifying sufficient power and cooling for the cluster hardware.

A number of projects at the Laboratory have already benefited from the LLGrid's cluster consulting services. The Lincoln Multimission ISR Testbed (LiMIT), developed by the RF Array Systems group, is an experimental radar testbed installed on a Boeing 707. The LiMIT system includes a cluster computer installed in the cargo bay, running pMatlab for inflight synthetic aperture radar and GMTI processing. This 'quick-look' cluster allows an onboard analyst to determine whether the radar is collecting useful data [2].

The Aerospace Sensor Technology group has developed a prototype image processing system that generates, displays, and analyzes 3D laser radar (ladar) data in real-time. To achieve real-time throughput, the imagery-generation algorithms are parallelized to run on a Linux cluster. Multiprocessor software plus blade hardware results in a compact, real-time imagery generation system adjunct to an operating ladar [3]. Figure B shows a single frame generated by the real-time 3D ladar system, and a photograph of the system's cluster computer.

By centralizing cluster administration expertise and standardizing cluster hardware and software, the LLGrid team helps staff avoid 'reinventing the cluster computing wheel' at Lincoln Laboratory.

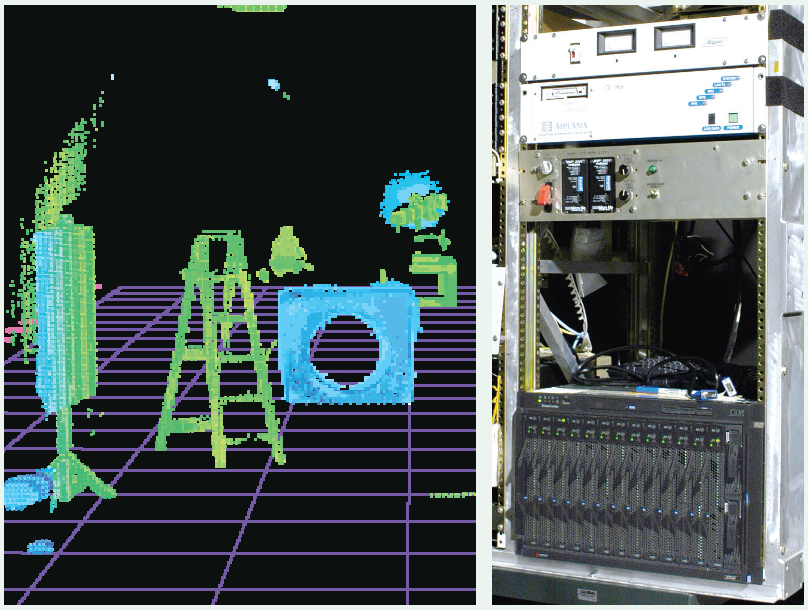


FIGURE B. (left) A single image frame generated by the real-time 3D ladar system, scanning various objects in the laboratory. (right) A photo of the cluster computer in the real-time ladar system.

References
1. J. Hurst, P.E. Bieringer, A. Lewis, D. Jeannotte, and E. Griffin, "Optimizing MM5 on an Intel Based Linux Beowulf Cluster," *6th WRF/15th MM5 Users' Workshop, 27–30 June 2005, National Center for Atmospheric Research, Boulder, Colo.*
2. J. Kepner, T. Currie, H. Kim, A. McCabe, B. Mathew, M. Moore, D. Rabinkin, A. Reuther, A. Rhoades, N. Travinin, and L. Tella, "Deployment of SAR and GMTI Signal Processing on a Boeing 707 Aircraft Using pMatlab and a Bladed Linux Cluster," *High Performance Embedded Computing (HPEC) Workshop, Lexington, Mass., 28–30 Sept. 2004.*
3. Peter Cho et al. "Real-Time 3D Ladar Imaging," in this issue.

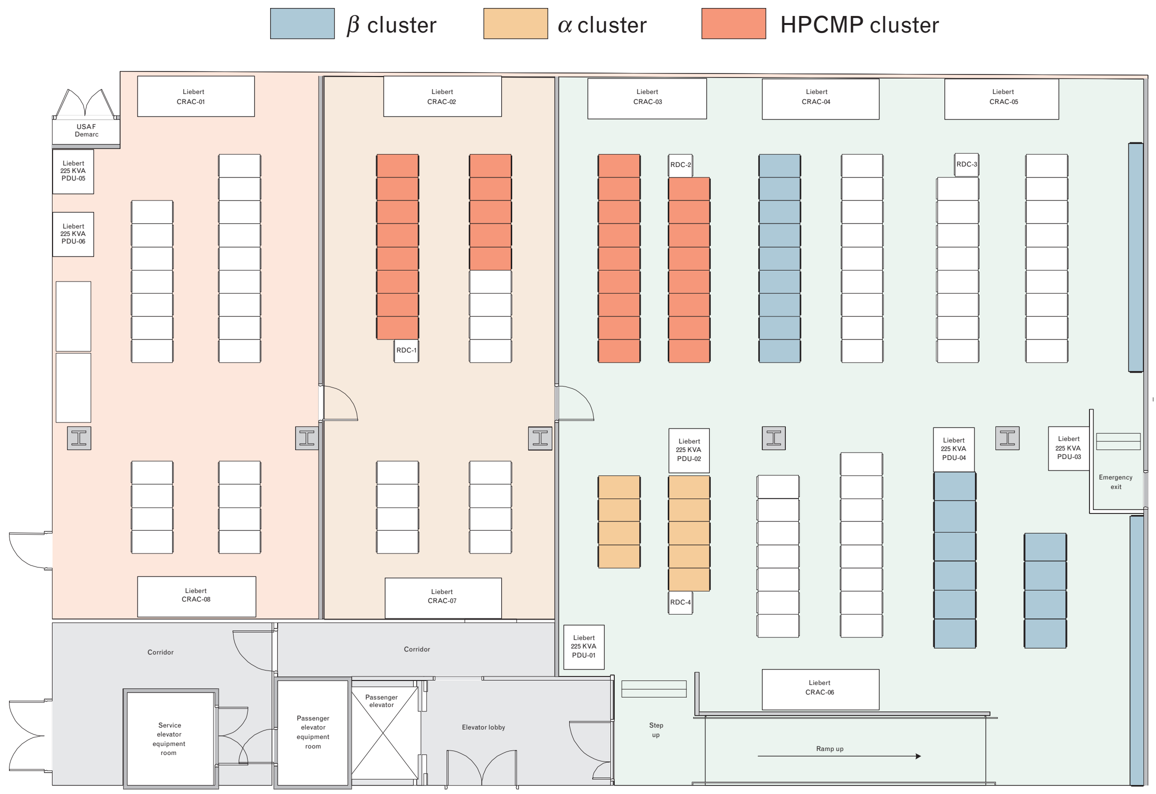


FIGURE 34. Floor plan of the shared computing facility. The figure depicts the arrangement of the LLGrid α, β, and HPCMP clusters. The floor plan is separated into an unclassified area (on the right), collateral secret area (in the middle), and an area reserved for future use (on the left). The unclassified area houses the α and β Grids and part of the HPCMP system. The rest of the HPCMP system is located in the collateral secret area.

modated, thus requiring the user to manage the data size manually. Second, pMatlab still requires the user to specify maps for the program. Determining a set of efficient maps for a program is a non-trivial problem requiring understanding of parallel algorithm and architectures. The following sections discuss the ongoing research to eliminate these software limitations.

Out-of-Core Storage for Parallel MATLAB. Many modern scientific and engineering applications process datasets that cannot fit into memory on a single computer. For example, synthetic aperture radar images generated by the Lincoln Multimission ISR Testbed (LiMIT) system, as described in the sidebar entitled "The LLGrid Project: Enterprise Cluster Computing Services," can require several gigabytes of storage. Using a parallel computer, such as the LLGrid, is a common method of addressing this problem, utilizing its greater memory capacity. Another method is to use out-of-core methods, which store large datasets on disk storage and use physical memory to view a section of the data at a time. Disk storage's greater capacity accommodates much larger datasets than can be addressed by even the largest parallel computer.

A number of technologies have been developed to provide programmers with an out-of-core capability. These technologies range from programming libraries, (e.g., POOCLAPACK, Panda, and SOLAR) to compilers (e.g., PASSION). In fact, a number of these technologies are targeted toward parallel systems. For example, POOCLAPACK is an out-of-core extension to the PLAPACK library, a parallel linear algebra library. However, these technologies still require significant programming expertise to benefit from them and hence are ill suited for rapid prototyping.

The Parallel MATLAB eXtreme Virtual Memory (pMatlab XVM) library combines pMatlab's parallel programming model with out-of-core methods [14]. While other technologies have combined out-of-core methods with parallel programming, the main innovation of pMatlab XVM is the combination of the PGAS

Table 9. A Comparison of PGAS and Out-of-Core Technologies against pMatlab XVM

Technology	*PGAS*	*Out of core*
POOCLAPACK	×	✓
Panda	×	✓
SOLAR	×	✓
PASSION	×	✓
High Performance Fortran	✓	×
Unified parallel C	✓	×
VSIPL++	✓	×
pMatlab	✓	×
pMatlab XVM	✓	✓

programming model with out-of-core techniques. Just as pMatlab enabled a new class of users, namely, engineers and scientists, to benefit from parallel programming to enhance performance of their applications, pMatlab XVM will enable these same users to benefit from out-of-core methods, allowing them to explore problem sizes that were previously unachievable. Table 9 provides a comparison of various PGAS and out-of-core technologies against pMatlab XVM.

pMatlab XVM uses hierarchical arrays to structure and swap data between memory and disk storage in a manner that is optimal for a particular algorithm, hiding the large amount of index bookkeeping typically required by out-of-core algorithms. The result is a capability that allows applications to use all available disk storage to accommodate extremely large datasets while leveraging multiple processors, with little sacrifice in programmability and performance.

pMatlab XVM introduces the *distributed out-of-core array*, or *doocmat*. The doocmat retains the properties of pMatlab dmat while adding the capability to partition data owned by a processor and manage which sections are stored in memory or on the disk. The doocmat is a hierarchical array, organized as a 2-level tree, as shown in Figure 35. The root of the tree is called the global matrix. Each processor contains a copy of the global matrix object, which stores information on how the overall matrix is distributed between processors. On each processor, the global matrix points to a leaf doocmat, called an out-of-core matrix. The out-of-core matrix stores information on how data owned by the local processor are divided into core blocks, manages which core blocks are stored in memory or on the disk, and holds the core block stored in memory.

Each level in the hierarchy requires its own map to describe how data at that level are distributed. The *global map* describes how to distribute the global matrix among processors, similar to dmat maps. The *out-of-core map* describes how to distribute the out-of-core matrix on each processor into core blocks. The global and out-of-core maps are then combined into a single map object, as shown in Figure 36.

Just as different algorithms have different optimal distributions, they may also have different optimal data access patterns. Consequently, pMatlab XVM provides a small, simple-to-use API that allows the

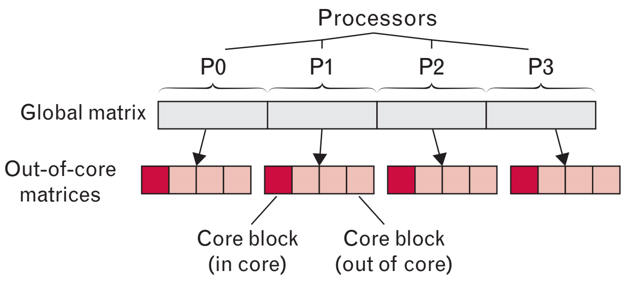


FIGURE 35. Hierarchical matrix. The hierarchical matrix, or *doocmat*, retains the properties of the pMatlab distributed array, or *dmat*, while adding the capability to partition data owned by a processor and managing which sections are stored in memory or on disk.

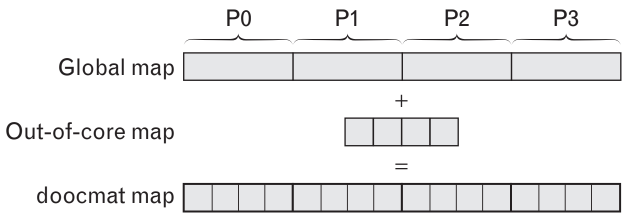


FIGURE 36. Hierarchical map. Each level of the hierarchy of the hierarchical matrix requires its own map to describe how the data are distributed. The hierarchical map consists of the global map, which describes how the array is distributed between processors, and the out-of-core map, which describes how the array is distributed between the disk and memory on each processor.

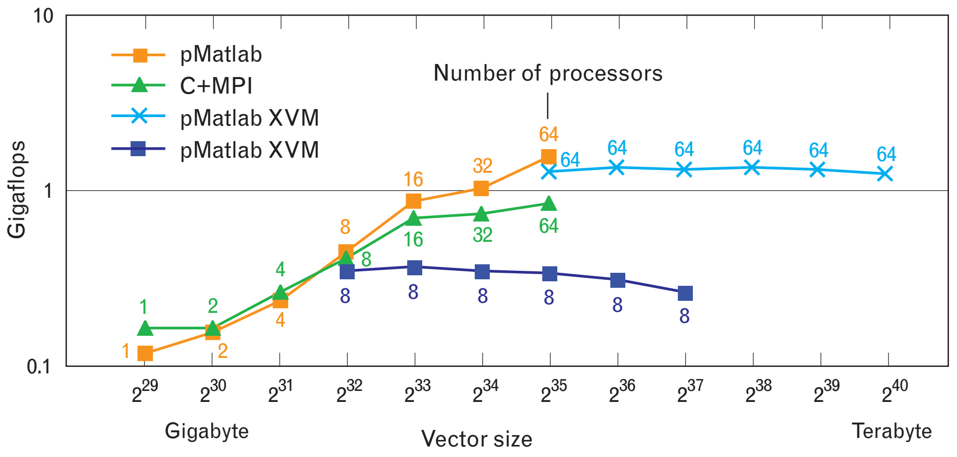


FIGURE 37. pMatlab XVM HPC Challenge FFT Performance. The figure plots size of the vector input into the FFT on the *x*-axis and performance in gigaflops on the *y*-axis. Numbers next to each data point indicate the number of processors used for the given input size. For pMatlab and C+MPI implementations, each data point represents the maximum problem size possible on the indicated number of processors. For example, pMatlab required 64 processors to perform the FFT on a 2^{35}-byte vector, while pMatlab XVM performed the same size computation on 8 processors.

programmer to easily manipulate doocmat objects by selecting core blocks to swap between memory and disk. This gives pMatlab XVM the flexibility to implement a variety of algorithms for processing large datasets while retaining pMatlab's ease of use.

As modern digital receivers move well into the GHz sampling regime, it is now common to have terabytes of data that need to be processed coherently. We have applied pMatlab XVM to the HPC Challenge FFT (as discussed in the section on High Performance Computing Challenge benchmarks). The results, summarized in Figure 37, show that the pMatlab XVM implementation of the FFT benchmark is easy to write and understand, requires few lines of code, and incurs a relatively small performance overhead while performing a one-terabyte FFT. Table 10 shows software-lines-of-code counts for the FFT benchmark for four implementations: serial MATLAB, pMatlab, pMatlab XVM and C+MPI. These results show that even the pMatlab XVM implementation requires over an order-of-magnitude fewer lines of code than the non–out-of-core C+MPI implementation. Figure 37 shows performance of the FFT benchmark for increasing input sizes. Note that for a given number of processors, as input size increases pMatlab XVM performance does not degrade but remains flat.

We plan to apply pMatlab XVM to the full HPC Challenge benchmark suite. If we use next-generation hardware, problem sizes a factor of 100 to 1000 times larger should be feasible. We are also transitioning this technology to several DoD signal processing applications. Some DoD applications can process large datasets offline, after data collection. However, other applications require the ability to perform ultra-long FFTs in real time during data collection. The flexibility of software technologies like pMatlab XVM allows hardware designers to experiment with various FFT parameters before implementing the design in hardware [15].

Automatic Mapping. Throughout this article we have discussed a number of parallel MATLAB capabilities. The first parallel MATLAB library developed at Lincoln Laboratory was MatlabMPI, which enabled parallel processing in MATLAB. However, it required users to use explicit message passing and significantly

Table 10. Software-Lines-of-Code Counts for the Four Versions of the HPC Challenge FFT Benchmark

MATLAB	pMatlab	pMatlab XVM	C+MPI
36	99	152	2509

reduced the ease of programming. Following MatlabMPI, pMatlab was developed, which abstracts away the message passing layer by introducing maps. While pMatlab significantly increases the ease of programming, compared to MatlabMPI, it still requires users to specify maps. Determining an efficient map for a particular computation is a non-trivial problem requiring familiarity with the underlying architecture and the characteristics of the computation. The problem becomes even more difficult if there are multiple computational stages in a program, since the best map for a single operation might not be the best map in the global program flow context.

```
%  Initialize variables
1. M=…; N=…;
%  Create maps
2. map1=map([1 4], {}, [0:3]);
3. map2=map([4 1], {}, [4:7]);
%  Create distributed arrays
4. A=rand(M, N, map1);
5. B=zeros(M, N, map2);
6. C=zeros(M, N, map2);
%  2-D FFT
7. B(:,:)=fft(A,[],1); %FFT along columns
8. C(:,:)=fft(B,[],2); %FFT along rows
%  Print out C
9. C
```

(a)

```
%  Initialize variables
1. M=…; N=…;
%  Create distributed arrays
2. A=rand(M, N, p);
3. B=zeros(M, N, p);
4. C=zeros(M, N, p);
%  2-D FFT
5. B(:,:)=fft(A,[],1); %FFT along columns
6. C(:,:)=fft(B,[],2); %FFT along rows
%  Print out C
7. C
```

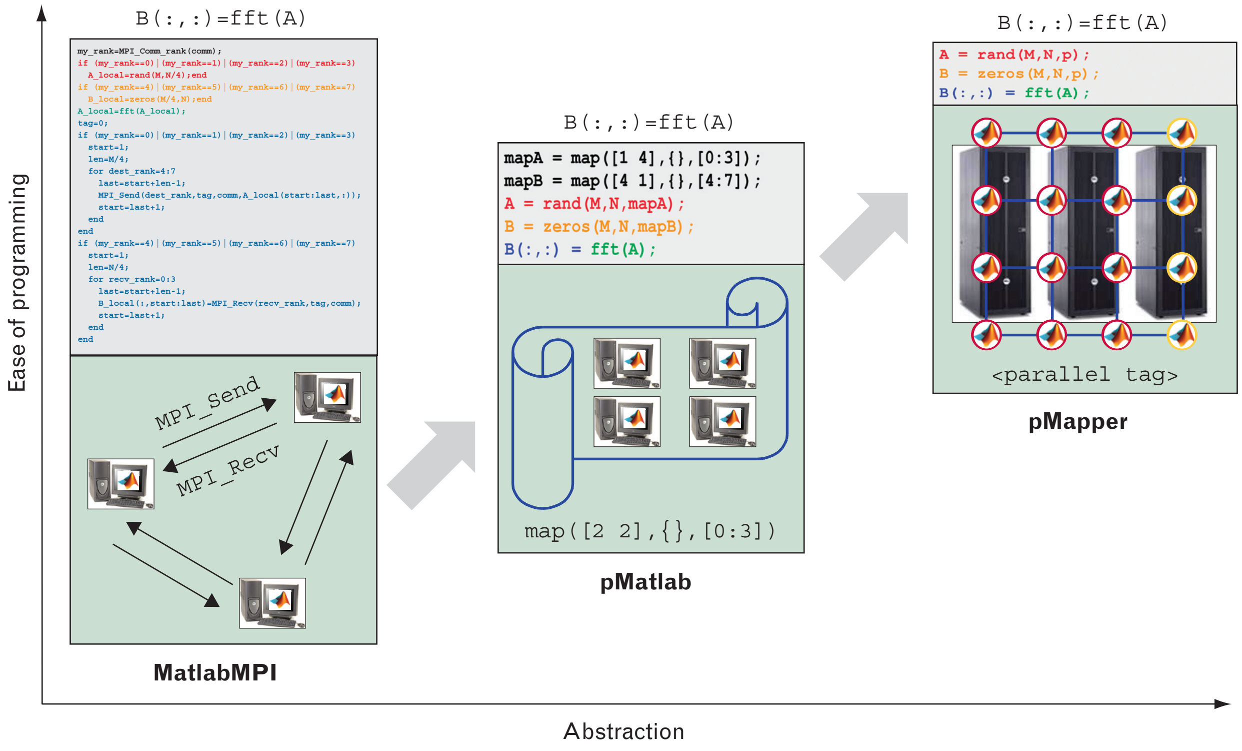
(b)

FIGURE 38. pMatlab and pMapper code segments. (a) The pMatlab code segment performs similar computation to the HPC Challenge FFT. Lines 2 and 3 define maps for arrays declared in lines 4, 5, and 6. Line 7 performs an FFT along columns; line 8 performs an FFT along rows. Both map1 and map2 are locally optimal maps for the computation that follows, but not necessarily globally optimal maps for the entire program. (b) The pMapper code segment is functionally equivalent to the pMatlab code segment in part *a*; however, map definitions are no longer necessary. Instead, the array constructors are tagged with a parallel tag *p*, indicating to pMapper that arrays **A**, **B**, and **C** should be considered for distribution.

Let us consider the simple program similar to the HPC Challenge FFT benchmark illustrated in Figure 38. The code in Figure 38(a) (lines 7–8) and Figure 38(b) (lines 5–6) performs (1) an FFT along columns of matrix **A** and stores the result in **B**; and (2) an FFT along rows of matrix **B** and stores the result in **C**. Consider the maps in Figure 38(a), lines 2–3. The first map splits matrix **A** along columns (line 4); the second map splits matrices **B** and **C** along rows (lines 5–6). This is clearly the locally optimal mapping for each of the two FFT operations—the computation is minimized. However, these maps require a full corner turn, or redistribution from column-wise map to row-wise map, on line 7 during the assignment of the result of the first FFT to matrix **B**. The corner turn requires an all-to-all communication and could be an expensive operation on a low latency system. A better mapping for the program might use fewer processors for one or both of the FFTs.

In order to balance the communication and computation, the writer of the program has to be aware of the properties of the parallel system and the parallel algorithms being developed. The users of pMatlab tend to be physicists, mathematicians, and engineers rather than computer scientists. Abstracting the mapping interface from the users greatly reduces the level of computer science and parallel programming expertise needed to write parallel MATLAB programs.

pMapper is designed with two primary goals in mind: (1) ease of programming and (2) optimized time to solution [13]. A key design concept of the pMapper architecture is lazy evaluation, which allows pMapper to collect as much information as possible about the program structure prior to assigning maps to the numerical arrays. Having access to information about program structure allows pMapper to produce optimized mappings and thus optimized time to solution. Like pMatlab, pMapper takes advantage of object oriented programming. This allows the users to simply annotate the numerical arrays to be considered for distribution. The annotation creates objects that store necessary information about the arrays. The object creation is transparent to the user and satisfies the ease of programming goal.



```
my_rank=MPI_Comm_rank(comm);
if (my_rank==0)|(my_rank==1)|(my_rank==2)|(my_rank==3)
  A_local=rand(M,N/4);end
if (my_rank==4)|(my_rank==5)|(my_rank==6)|(my_rank==7)
  B_local=zeros(M/4,N);end
A_local=fft(A_local);
tag=0;
if (my_rank==0)|(my_rank==1)|(my_rank==2)|(my_rank==3)
  start=1;
  len=M/4;
  for dest_rank=4:7
    last=start+len-1;
    MPI_Send(dest_rank,tag,comm,A_local(start:last,:));
    start=last+1;
  end
end
if (my_rank==4)|(my_rank==5)|(my_rank==6)|(my_rank==7)
  start=1;
  len=N/4;
  for recv_rank=0:3
    last=start+len-1;
    B_local(:,start:last)=MPI_Recv(recv_rank,tag,comm);
    start=last+1;
  end
end
```

```
mapA = map([1 4],{},[0:3]);
mapB = map([4 1],{},[4:7]);
A = rand(M,N,mapA);
B = zeros(M,N,mapB);
B(:,:) = fft(A);
```

```
A = rand(M,N,p);
B = zeros(M,N,p);
B(:,:) = fft(A);
```

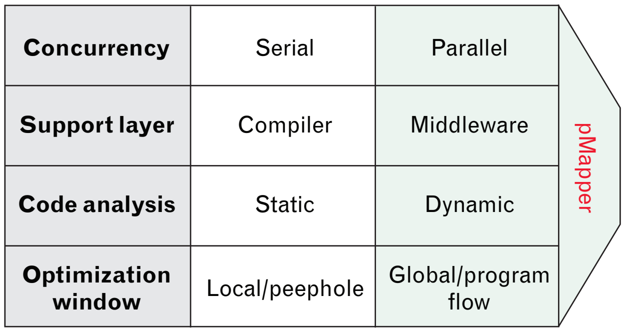
FIGURE 39. Evolution of parallel programming. The graph plots level of abstraction on the *x*-axis against ease of programming on the *y*-axis. MatlabMPI requires explicit message passing and thus has the lowest level of abstraction and ease of programming. pMatlab abstracts the message passing layer, but requires explicit map definitions. pMapper, which has the highest level of abstraction and ease of programming, and which assumes the user is not a parallel programmer, provides the user with implicit parallelism by abstracting the maps.

Eliminating maps and replacing them with annotations raises the level of abstraction. Just as pMatlab abstracted away the Message Passing Interface, pMapper abstracts away the mapping interface, as shown in Figure 39. Figure 38(b) is the pMapper code that is functionally equivalent to the pMatlab code in Figure 38(a).

The pMapper code looks essentially identical to regular MATLAB code, with the exception of the addition of the parallel tags, *p*. The parallel tags are inherent variables in the library and, when passed to an array constructor, indicate to the mapping system that the associated numerical array should be considered for distribution. For example, in Figure 38(b), arrays **A**, **B**, and **C** are tagged and will be considered for distribution.

Automatic program optimization is an active area of research. In order to determine where pMapper fits into the current research, we develop a taxonomy of automatic mapping approaches. We use four characteristics in our classification scheme, as illustrated in Figure 40.

Let us consider each of the characteristics in detail. Concurrency could be either serial or parallel. Serial concurrency [48] implies that the automatic mapper is mapping into the serial memory hierarchy of the system. On the other hand, parallel concurrency [49] implies that the mapper is searching for the best mapping onto a distributed architecture. Support layer defines in which software layer the automatic mapper is implemented. The automatic mapper could be implemented in the compiler layer [50, 51] or in middleware layer [52, 48]. Code analysis specifies whether static or dynamic code analysis is performed. Static code analysis [48] implies looking at code as text, while dynamic code analysis [53] considers the behavior of the code at run time. The last category, optimization window, specifies whether the automated performance tuning is performed on a single function (locally) [50, 48] or the entire program flow (globally) [54].



Concurrency	Serial	Parallel
Support layer	Compiler	Middleware
Code analysis	Static	Dynamic
Optimization window	Local/peephole	Global/program flow

FIGURE 40. Taxonomy of automatic mapping approaches, which can be classified by using these four characteristics highlighted in the figure. pMapper is mapped onto a parallel architecture, is implemented in middleware, performs dynamic code analysis, and results in global optimization.

Some approaches that perform local optimization include FFTW [50] and ATLAS [48]. FFTW optimizes the performance of FFTs, while ATLAS considers a variety of linear algebra routines. FFTW addresses the problem at the compiler level and optimizes at both serial and parallel concurrency, while ATLAS is middleware that optimizes at serial concurrency. PLAPACK [49] addresses the parallel concurrency optimization of individual linear algebra routines. The Dynamo project [53] addresses the problem of dynamic code optimization for serial programs. Additionally, the SPIRAL [55] project has made significant impact on automatic tuning of digital signal processing algorithms.

pMapper can be classified as having parallel concurrency, written at middleware layer, performing dynamic code analysis and global optimization. The key advantage of the pMapper framework over other existing approaches is that it performs both dynamic code analysis and global (program flow) optimization, while maintaining the simplicity of the interface.

We tested the pMapper approach on a sample application similar to the one in Figure 38(b) with an additional matrix multiply. Figure 41 shows both the speedup curve and the output maps produced by pMapper. This application has a high communication-to-computation ratio and is made up of important kernels present in many signal processing codes. The results were obtained by using simulated timing

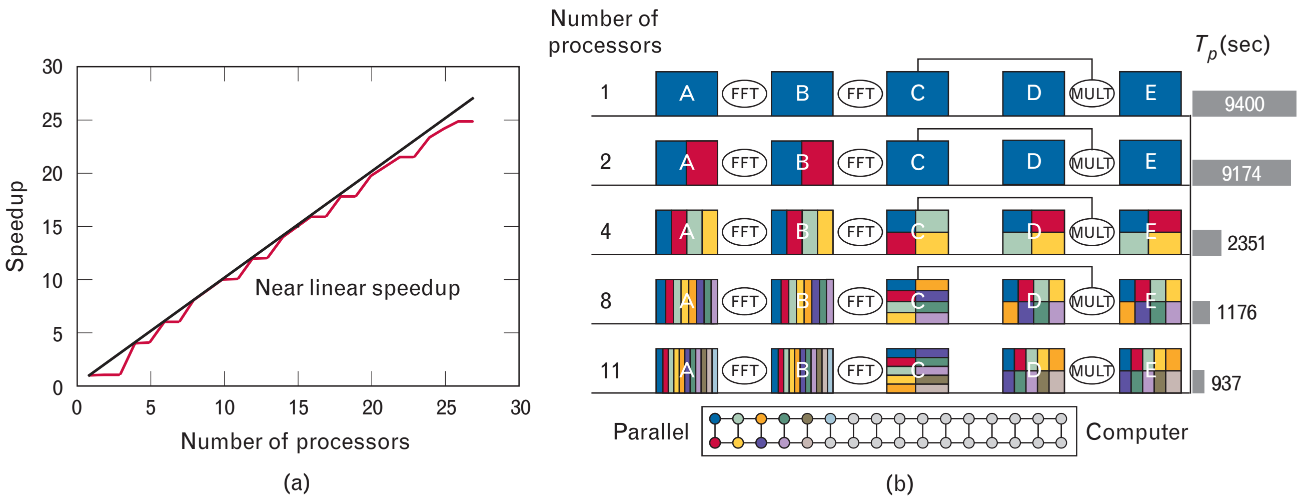


FIGURE 41. (a) Speedup versus number of processors. Speedup is defined as (serial execution time)/(parallel execution time). The black line illustrates optimal speedup, which is commonly achieved by embarrassingly parallel computations, or computations that require no communication. The red line illustrates speedup achieved by the application mapped with pMapper. Even though the application has a high communication-to-computation ratio, it achieves near linear speedup. (b) Simulated mapping results. Each line corresponds to the mappings and time (T_p) produced by pMapper for a given number of processors. The blocks represent arrays and the ovals represent operations performed on those arrays. For the one processor case, the mappings are intuitive—the whole application is mapped onto one processor. On two processors, arrays **A** and **B** are distributed between two processors and **C**, **D**, and **E** are mapped onto one processor. The four-processor case truly illustrates the global mapping nature of the algorithm. The maps for **A** and **B** are column-wise maps that are optimal for the column-wise FFT; however, the map for **C** benefits the matrix multiply operation and not the row-wise FFT of which **C** is the output. For the eight-processor and eleven-processor cases, the maps are similar to the four-processor maps.

Obtaining pMatlab and MatlabMPI

Lincoln Laboratory staff members can learn more about using pMatlab and MatlabMPI on the LLGrid cluster at the following intranet website.

http://llwww/LLGrid

Readers of this article who are not affiliated with Lincoln Laboratory can freely obtain the pMatlab and MatlabMPI libraries from the Laboratory's external website.

http://www.ll.mit.edu/pMatlab

http://www.ll.mit.edu/MatlabMPI

data that described a low-latency architecture, e.g., an embedded computing system. Consider the maps in Figure 41(b). The rectangles in the figure represent numerical arrays, the ovals represent operations on the arrays. Specifically, array **A** is the input into the first FFT, array **B** is the output of the first FFT and the input into the second FFT, and so on. Each color represents a different processor, so the color coding of the arrays represents how the arrays are mapped. The maps produced for this sample code demonstrate that the mapper is performing global optimization. The best map for the output of the row FFT operation, numerical array **C**, would be a map along rows; however, the mapper picks a 2 × 2 block map that benefits the subsequent matrix multiply operation. These results illustrate that while pMapper was originally designed for pMatlab running on a cluster, it also shows great promise as a mapping tool for embedded systems.

Summary

In order to address algorithm development challenges at Lincoln Laboratory, we have developed the LLGrid system to provide users with parallel computing capability without increasing code development costs. The pMatlab and gridMatlab libraries were developed to allow users to write and launch parallel MATLAB programs as easily as writing and launching MATLAB programs on their desktops. The HPC Challenge benchmarks and user experiences show that the pMatlab library provides significant improvement in performance without significant code increases. The move of the LLGrid system to the F1 shared computing facility further establishes the system as a production resource. Additionally, acquisition of the new 1000-processor system will allow for significant improvement in algorithm development for defense against weapons of mass destruction. The advanced research on parallel technologies such as out-of-core and automatic mapping will aid in algorithm development both in MATLAB and in other languages.

Acknowledgments

The authors would like to thank a number of individuals who have contributed to this work. We would like to thank the LLGrid team, Andy McCabe, Tim Currie, Bill Arcand, Peter Michaleas, and Julie Mullen, without whom none of this research would have been possible. We would also like to thank Andy Funk for his benchmarking analysis; Cleve Moler (MathWorks) and Ryan Haney for their assistance with the parallel LU algorithm; and Charlie Rader for his assistance with the parallel FFT algorithm. We would like to thank Ken Senne, Dave Martinez, and Robert Graybill (DARPA) for funding this project, and Leslie Alger and Gerry Banner for supporting it. Finally, we would like to thank all of the LLGrid users, particularly Paul Bieringer, Peter Cho, Steven Michael, John Cho, John Stueve, Brandom Kam, and Cy Chan for providing material for this article.

REFERENCES

1. MATLAB, The MathWorks, Inc., http://www.mathworks.com/products/matlab.
2. J. Ward, "Space-Time Adaptive Processing for Airborne Radar," Lincoln Laboratory Technical Report 1015, DTIC #ADA-293032 (13 Dec. 1994).
3. G.R. Benitz, "High-Definition Vector Imaging," *Linc Lab. J.* **10** (2), 1997, pp. 147–170.
4. J. Goodman and P.A. Jackson, "Iterative and Optimal Frequency Offset Estimation in OFDM Systems," *IEEE 2006 International Waveform Diversity and Design Conf., Lihue, Hawaii, 22–27 Jan. 2006*.
5. C. Reschke, T. Sterling, D. Ridge, D. Savarese, D. Becker, and P. Merkey, "A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation," *Proc. 5th IEEE Int. Symp. on High Performance Distributed Computing, Syracuse, N.Y., 6–9 Aug. 1996*, pp. 626–635.
6. J. Kepner and S. Ahalt, "MatlabMPI," *J. Parallel Distrib. Comput.* **64** (8), 2004, pp. 997–1005.
7. Message Passing Interface (MPI), http://www.mpi-forum.org.
8. J. Kepner and N. Travinin, "Parallel Matlab: The Next Generation," *Proc. High Performance Embedded Computing Workshop (HPEC 2003), Lexington, Mass., 23–25 Sept. 2003.*
9. J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge, "Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing," *Proc. IEEE* **93** (2), 2005, pp. 313–330.
10. I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, 2nd ed. (Elsevier, Amsterdam, 2004).
11. S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar, "Netsolve: Past, Present, and Future; A Look at a Grid Enabled Server," chap. 24 in *Grid Computing–Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, eds. (John Wiley & Sons, Ltd., Chichester, U.K., 2003), pp. 613–622.
12. A.S. Grimshaw, W.A. Wulf, and the Legion team, "The Legion Vision of a Worldwide Virtual Computer," *Commun. ACM* **40** (1), 1997, pp. 39–45.
13. N. Travinin, H. Hoffmann, R. Bond, H. Chan, J. Kepner, and E. Wong, "pMapper: Automatic Mapping of Parallel Matlab Programs," *Proc. Department of Defense High Performance Computing Modernization Program Users Group Conference (DoD HPCMP UGC 2005), Nashville, Tenn., 28–30 June 2005.*
14. H. Kim, C. Kahn, and J. Kepner, "Parallel Out-of-Core MATLAB for Extreme Virtual Memory," *Proc. Department of Defense High Performance Computing Modernization Program Users Group Conference (DoD HPCMP UGC 2005), Nashville, Tenn., 28–30 June 2005.*
15. H. Kim, J. Kepner, M. Vai, and C. Kahn, "Advanced Hardware and Software Technologies for Ultra-Long FFT's," *Proc. High Performance Embedded Computing Workshop (HPEC 2005), Lexington, Mass., 20–22 Sept. 2005.*
16. J.D. McCalpin, 2005, STREAM: Sustainable Memory Bandwidth in High Performance Computers, http://www.cs.virginia.edu/stream.
17. R. Choy and A. Edelman, Parallel Matlab survey, http://www.interactivesupercomputing.com/reference/ParalelMatlabsurvey.htm.
18. Cornell Multitask Toolbox for MATLAB (CMTM), http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html.
19. A. Funk, J. Kepner, V. Basili, and L. Hochstein, "A Relative Development Time Productivity Metric for HPC Systems," *Proc. High Performance Embedded Computing Workshop (HPEC 2005), Lexington, Mass., 20–22 Sept. 2005.*
20. L. Dean, S. Grad-Freilich, J. Kepner, and A. Reuther, "Distributed and Parallel Computing with MATLAB," tutorial presented at *ACM/IEEE Conf. on Supercomputing 2005, Seattle, Wash., 12–18 Nov. 2005.*
21. Distributed Computing Toolbox user's guide, MathWorks Inc. 2005, http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp.
22. R. Choy and A. Edelman, "Parallel MATLAB: Doing It Right," *Proc. IEEE* **93** (2) 2005, pp. 331–341.
23. G. Morrow and R. van de Geijn, "A Parallel Linear Algebra Server for Matlab-Like Environments," *Proc. 1998 ACM/IEEE Conf. on Supercomputing, Orlando, Fla., 7–13 Nov. 1998,* http://www.supercomp.org/sc98/TechPapers/sc98 FullAbstracts/Morrow779/index.htm.
24. Falcon Project: Fast Array Language Computation, http://www.csrd.uiuc.edu/falcon/falcon.html.
25. C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steel, Jr., and M.E. Zosel, *The High-Performance Fortran Handbook* (MIT Press, Cambridge, Mass., 1994).
26. R.W. Numrich and J. Reid, "Co-Array Fortran for Parallel Programming," *ACM SIGPLAN Fortran Forum* **17** (2), 1998, pp. 1–31.
27. T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming* (Wiley, Hoboken, N.J., May, 2005).
28. J.C. Cummings, J.A. Crotinger, S.W. Haney, W.F. Humphrey, S.R. Karmesin, J.V.W. Reynders, S.A. Smith, and T.J. Williams, "Rapid Application Development and Enhanced Code Interoperability Using the POOMA Framework," *Proc. SIAM Workshop on Object-Oriented Methods and Code Interoperability in Scientific and Engineering Computing (OO98), Yorktown Heights, N.Y., 21–23 Oct. 1998.*
29. J. Nieplocha, R.J. Harrison, M.K. Kumar, B. Palmer, V. Tipparaju, and H. Trease, "Combining Shared and Distributed Memory Models: Approach and Evolution of the Global Arrays Toolkit," *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL-02), Int. Conf. on Supercomputing, New York, N.Y., 22–26 June, 2002.*
30. P. Johnson, ed., *Proc. 26th Int. Conf. on Software Engineering (ICSE 2004), Edinburgh, 23–28 May 2004.*
31. P. Johnson, ed., *Proc. 27th Int. Conf. on Software Engineering (ICSE 2005), St. Louis, Mo., 15–21 May 2005.*
32. J. Kepner, ed., *Int. J. High Perform. Comput. Appl.* **18** (4), 2004 (special issue on HPC productivity).
33. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Comput. Surv.* **17**(4), 1985, pp. 471–523.
34. D.B. Loveman, "High Performance Fortran," *IEEE Parallel Distrib. Technol. Syst. Appl.* **1** (1), 1993, pp. 25–42.
35. M.E. Zosel, "High Performance Fortran: An Overview," *Compcon Spring '93, Digest of Papers, San Francisco, Calif., 22–26 Feb. 1993.*
36. C.M. DeLuca, C.W. Heisey, R.A. Bond, and J.M. Daly, "A Portable Object-Based Parallel Library and Layered Frame-Work for Real-Time Radar Signal Processing," *Proc. 1st Conf. International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE '97)* **1343,** *Marina del Rey, Calif., 8–11 1997,* pp. 241–248.
37. P. Luszczek, J.J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "In-

troduction to the HPC Challenge Benchmark Suite," Lawrence Berkeley National Laboratory, Paper LBNL-57493, 25 Apr. 2005.
38. G.H. Golub and C.F. Van Loan, *Matrix Computations*, 3rd ed. (Johns Hopkins University Press, Baltimore, 1996).
39. J.J. Dongarra, R.A. van de Geijn, and D.W. Walker, "Scalability Issues Affecting the Design of a Dense Linear Algebra Library," *J. Parallel Distrib. Comput.* **22** (3), 1994, pp. 523–537.
40. Platform Rocks, http://www.platform.com/Products/Rocks.
41. Red Hat Enterprise Linux, Red Hat, Inc., http://www.redhat.com/software/rhel
42. A.I. Reuther, T. Currie, J. Kepner, H.G. Kim, A. McCabe, and P. Michaleas, "Technology Requirements for Supporting On-Demand Interactive Grid Computing," *Proc. High-Performance Computing Modernization Office Users Group Conference (DoD HPCMP UGC 2005), Nashville, Tenn., 27–30 June 2005.*
43. OpenPBS: Portable Batch System, http://www.openpbs.org.
44. N1 Grid Engine, Sun Microsystems, Inc., http://www.sun.com/software/gridware/sge.html
45. Ganglia Monitoring System, http://ganglia.sourceforge.net.
46. J.Y.N. Cho, G.R. Elkin, and N.G. Parker, "Enhanced Radar Data Acquisition System and Signal Processing Algorithms for the Terminal Doppler Weather Radar," *Proc. AMS 32nd Conf. on Radar Meteorology, Albuquerque, New Mex., 24–29 Oct. 2005*, P4R8.
47. HPCS: High Productivity Computer Systems, http://www.highproductivity.org.
48. R.C. Whaley, A. Petitet, and J.J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Proc. High Performance Embedded Computing Workshop (HPEC 2000), Lexington, Mass., 20–22 Sept. 2000.*
49. R.A. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package* (MIT Press, Cambridge, Mass., 1997).
50. M. Frigo and S.G. Johnson, FFTW, http://www.fftw.org.
51. M.J. Wolfe, *High Performance Compilers for Parallel Computing* (Benjamin/Cummings, Redwood City, Calif., 1996).
52. H. Hoffmann, J. Kepner, and B. Bond, "S3P: Automatic, Optimized Mapping of Signal Processing Applications to Parallel Architectures," *Proc. High Performance Embedded Computing Workshop (HPEC 2001), Lexington, Mass., 27–29 Nov. 2001.*
53. The Dynamo Project, http://www.cs.indiana.edu/proglang/dynamo.
54. K. Kuo, R.M. Rabbah, and S. Amarasinghe, "A Productive Programming Environment for Stream Computing," *Proc. Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05), San Francisco, 13 Feb. 2005.*
55. M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proc. IEEE* **93** (2), 2005, pp. 232–275 (special issue on program generation, optimization, and adaptation).
56. S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," *Proc. Fifth Symp. on the Frontiers of Massively Parallel Computation (Frontiers '95), McClean, Va., 6–9 Feb. 2005,* pp. 342–349.

APPENDIX A. PROCESSOR INDEXED TAGGED FAMILY OF LINE SEGMENTS (PITFALLS)

An efficient and general technique for data redistribution is necessary in order to support PGAS. We chose to use PITFALLS [56], which is a mathematical representation of the data distribution. Additionally, this representation provides an algorithm for determining which pairs of processors need to communicate when redistribution is required and exactly what data needs to be sent.

A PITFALLS P is defined by the following tuple:

$$P = (l, r, s, n, d, p),$$

where l is the starting index, r is the ending index, s is the stride between successive values of l, n is the number of equally spaced and equally sized blocks of elements per processor, d is the spacing between values of l for successive processor FALLS, and p is the number of processors

The PITFALLS intersection algorithm is used to determine the necessary messages for redistribution. The algorithm can be applied to each dimension of the array, thus allowing efficient redistribution of arbitrary dimensional arrays. A detailed discussion of the algorithm and its efficiency is found elsewhere [56]. (Note that the PITFALLS tuple can be derived in a trivial manner from the map definition.)

APPENDIX B. CODE HIGHLIGHTS

RandomAccess

```
Np = pMATLAB.comm_size; % Set number of processors.
N = 2^20;               % Set dimensions of array.

Tmap = map([1 Np],{},0:Np-1);   % Create row map
% Create global unsigned int64 row vector.
Table =  zeros(1,N,Tmap,'uint64');

Imy = global_block_range(X table,2); % Local index ranges
Iall = global_block_ranges(X table,2);  % All index ranges.

% Initialize table ...

% For each block of updates.

  ran = RandomAccess_rand(ran); % Generate random indices.
  I = double(bitand(ran, TABLE_MASK)) + 1; % Compute table index.

  % Find indices for numbers that reside on rank i cpu and send.
  for i_cpu = my_send_order
    j_cpu = find((I >= Iall(i_cpu+1,2)) & (I <= Iall(i_cpu+1,3)));
    MPI_Send(i_cpu, tag_number, pMATLAB.comm, ran(j_cpu));
  end
  % Concatenate receives from all processors.
  for i_cpu=my_recv_order
    ran_recv = [ran_recv MPI_Recv(i_cpu, tag_number, pMATLAB.comm)];
  end

  % Compute local table index and perform vectorized XOR update.
  Ilocal = double(bitand(ran_recv, TABLE_MASK)) - Imy(1) + 2;
  Table(Ilocal) = bitxor(Table(Ilocal),ran_recv);
```

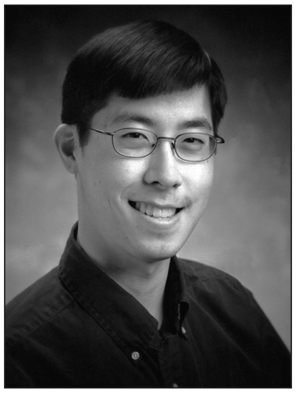
High Performance Linpack (Top500)

```
function [L,U,piv] = lu_parallel(A)
Np = pMATLAB.comm_size; % Set number of processors.
my_index = pMATLAB.comm_rank + 1;  % Get processor index.
col_ranges = global_block_ranges(A,2);  % Cols belong to all processors.

% Get sizes and local part of A.
[m,n] = size(A);  Alocal = local(A);  nlocal = size(Alocal,2);

% ... initialize index counters ...

for p = 1:Np % Loop over all processors.
  p_col = col_ranges(p,3) - col_ranges(p,2) + 1; % Cols on processor p.
  if (p == my_index) % Compute the LU of the p-th block.

    % ... compute index sets i and j ...

    [Alocal(i,j) pivp] = dgetrf(Alocal(i,j));  % Compute local LU and pivots.
    Lp = tril(Alocal(i,j));  % Get lower part.

    % Send Lp and pivp to all the higher processors and just pivp to lower.
    MPI_Mcast(p-1,p:(Np-1),tag_higher,comm,Lp,pivp);
    MPI_Mcast(p-1,0:(p-1),tag_lower,comm,pivp);

    % ... update index counters ...

  elseif (my_index > p)
    [Lp,pivp] = MPI_Recv(p-1,tag_higher,comm); % Receive L and pivots
  elseif (my_index < p)
    pivp = MPI_Recv(p-1,tag_lower,comm); % Receive pivots.
  end
  % ... apply pivots and weights ...
end
% ... Select L and U ...
```

NADYA TRAVININ BLISS
is an associate staff member in the Embedded Digital Systems group. Since joining the Laboratory in 2002, she has been one of the principal innovators and developers of pMapper, the automated parallelization system, and pMatlab, the parallel MATLAB toolbox. The pMatlab library has become a vital part of the Lincoln Laboratory Grid (LLGrid) and has been released to the open source community. The pMapper system, which was prototyped in MATLAB, is currently being adapted for real-time embedded systems. Additionally, she has been involved in the Integrated Sensing and Decision Support program in the area of automatic information extraction. She has also performed research in area cognitive kernels for the DARPA Architectures for Cognitive Information Processing (ACIP) program. Her research interests are in parallel and distributed computing and intelligent/cognitive algorithms. She received a B.S. degree and an M.Eng. degree in computer science from Cornell University.

ROBERT BOND
is leader of the Embedded Digital Systems group. He earned a B.S. degree (honors) in physics from Queen's University, Ontario, Canada in 1978. In his career he has focused on the research and development of high-performance embedded processors, advanced signal processing technology, and novel embedded middleware architectures. He joined Lincoln Laboratory in 1987. In his first assignment, he was responsible for the development of the Mountaintop RSTER radar software architecture and was coordinator for the radar system integration. In the early 1990s, he was involved in seminal studies to evaluate the use of massively parallel processors (MPP) for real-time signal and image processing. Later he managed the development of a 200 billion operations-per-second airborne processor, consisting of a 1000-processor MPP for performing radar space-time adaptive processing and a custom processor for performing high-throughput radar signal processing. In 2001, he led a team in the development of the Parallel Vector Library, middleware for the portable and scalable development of high-performance parallel signal processors. In 2003 he was one of two researchers to receive the Lincoln Laboratory Technical Excellence Award for his "technical vision and leadership in the application of high-performance embedded processing architectures to real-time digital signal processing systems."

JEREMY KEPNER
is a senior staff member in the Embedded Digital Systems group. His research interests are focused on the development of advanced libraries for the application of massively parallel computing to a variety of data-intensive signal processing problems. He has published over a dozen articles on these topics. He is the overall lead of the DARPA High Productivity Computing Systems (HPCS) Productivity Team, the technical lead of the High Performance Embedded Computing Software Initiative (HPEC-SI), the technical chairman of the HPEC Workshop, and the lead software architect of pMatlab and MatlabMPI. He received a B.A. degree with distinction in astrophysics from Pomona College, and a Ph.D. degree in astrophysics from Princeton University, where he was a Department of Education Computational Science Graduate Fellow. He joined Lincoln Laboratory in 1998.



HAHN KIM
is an associate staff member in the Embedded Digital Systems group. He received a B.S.E. degree in computer engineering and an M.S.E. degree in computer science and engineering from the University of Michigan. His research interests are in embedded systems and parallel programming technologies. His work includes developing parallel MATLAB and LLGrid technologies, supporting these technologies at Lincoln Laboratory, and developing distributed systems for networked intelligence, surveillance, and reconnaissance (ISR) assets.

ALBERT REUTHER
is a staff member in the Embedded Digital Systems group. He received a dual B.S. degree with highest distinction in computer engineering and electrical engineering, an M.S. degree in electrical engineering, and a Ph.D. degree in electrical and computer engineering, all from Purdue University. During his graduate work, he received the Andrews Fellowship and an Intel Foundation Ph.D. Fellowship. After completing his Ph.D., he earned an MBA degree from the Collège des Ingénieurs in Paris, France, and Stuttgart, Germany. During his educational pursuits, he worked in a variety of capacities at General Motors, Hewlett-Packard, and DaimlerChrysler. Upon completing his MBA, he joined the Embedded Digital Systems group, where his research has focused on distributed and parallel computing for rapid prototyping of digital signal processing and system simulations. He is the technical leader of the LLGrid project and the lead software architect of the gridMatlab toolbox. Additionally, he has worked on parallel computational analysis projects and is developing a distributed data architecture for military sensor systems.