Int J Parallel Prog (2009) 37:3–36
DOI 10.1007/s10766-008-0082-5

MATLAB®: A Language for Parallel Computing

Gaurav Sharma · Jos Martin

Received: 5 March 2008 / Accepted: 7 August 2008 / Published online: 15 October 2008
© The Author(s) 2008. This article is published with open access at Springerlink.com

Abstract Parallel computing with the MATLAB® language and environment has received interest from various quarters. The Parallel Computing Toolbox™ and MATLAB® Distributed Computing Server™ from The MathWorks are among several available tools that offer this capability. We explore some of the key features of the parallel MATLAB language that these tools offer. We describe the underlying mechanics as well as the salient design decisions and rationale for certain features in the toolset. The paper concludes by identifying some issues that we must address as the language features evolve.

Keywords Parallel MATLAB · Parallel language design · Parallel Computing Toolbox

1 Introduction

The MATLAB® technical computing language and development environment is used in a variety of fields, such as image and signal processing, control systems, financial modeling, and computational biology. MATLAB offers many specialized routines through domain specific add-ons, called "toolboxes", and a simplified interface to high-performance libraries such as BLAS, FFTW, and LAPACK. These features appeal to domain experts who can quickly iterate through various designs to arrive at a functional design more quickly than with a low-level language such as C.

G. Sharma (✉)
The MathWorks, 3 Apple Hill Drive, Natick, MA 01760, USA
e-mail: Gaurav.Sharma@mathworks.com

J. Martin
The MathWorks Ltd., Matrix House, Cambridge Business Park, Cambridge CB4 0HH, UK
e-mail: Jos.Martin@mathworks.com

Advances in computer processing power have enabled easy access to multiprocessor computers, whether through multicore processors, clusters built from commercial, off-the-shelf components, or a combination of the two. This created demand for desktop applications such as MATLAB to find mechanisms to exploit such architectures.

There have been several attempts at producing MATLAB based utilities for parallel programming. Among the most notable are pMATLAB [1] and MatlabMPI [2] from MIT Lincoln Laboratory, MultiMATLAB [3] from Cornell University, and bcMPI from Ohio Supercomputing Center [4].

This paper focuses primarily on The MathWorks extensions to the MATLAB language: Parallel Computing Toolbox™ and MATLAB® Distributed Computing Server™ software. We do not intend this paper to document all the features available in the language, nor do we fully detail all creative aspects or design choices of the features that we do describe. Instead, we highlight some of the salient features and provide insight into the motivations, rationale, and eventual design decisions that went into the feature implementation. We will present solutions to some of the problems we encountered while implementing these language features. Where appropriate we try to draw parallels with other HPC languages such as HPF, Co-array FORTRAN, and UPC.

The paper is organized as follows. We touch briefly on the history of parallel MATLAB and the various attempts to produce it. We then dive into the tools themselves and describe the conceptual framework for various capabilities. We make a distinction between language and infrastructure features that the tools offer, and describe the underlying mechanics and rationale behind components that make up the two pieces. We conclude with questions that we pose to ourselves as language features evolve to meet users' needs.

2 A History of Parallel MATLAB®

Even in its relative infancy there were attempts to develop MATLAB for parallel computers. Cleve Moler, the original MATLAB author and cofounder of The MathWorks, himself worked on a version of MATLAB for both an Intel® HyperCube and Ardent Titan in the late 1980s [5]. Moler's 1995 article "Why there isn't a parallel MATLAB" [5] described the three major obstacles in developing a parallel MATLAB language: memory model, granularity of computations, and business situation.

The conflict between MATLAB's global memory model and the distributed model of most parallel systems meant that the large data matrices had to be sent back and forth between the host and the parallel computer. Also at the time MATLAB spent only a fraction of its time in parallelizable routines compared to parser and graphics routines which made a parallelization effort not very attractive. The last obstacle was simply a dose of reality for an organization with finite resources—there were simply not enough MATLAB users who wanted to use MATLAB on parallel computers, and we focused instead on improving the uniprocessor MATLAB. However, these did not stop the user community from developing utilities for parallel computing with MATLAB.

Several factors have made the parallel MATLAB project a very important one inside The MathWorks: MATLAB has matured into a preeminent technical computing

environment supporting large scale projects, easy access to multiprocessor machines, and demand for a full fledged solution from the user community.

There have been three approaches to creating a system for parallel computing with MATLAB. The first approach aims at translating MATLAB or similar programs into a lower-level language such as C or FORTRAN, and uses annotations and other mechanisms to generate parallel code from a compiler. Examples of such projects include CONLAB [6], and FALCON [7]. Translating regular MATLAB code to C or FORTRAN is a difficult problem. In fact, the MATLAB Compiler software from The MathWorks switched from producing C code to producing wrappers around MATLAB code and libraries to be able to support all the language features [8].

The second approach is to use MATLAB as a "browser" for parallel computations on a parallel computer while MATLAB itself remains unmodified and the MATLAB environment does not run natively on the parallel computer. This approach does not really classify as a "parallel MATLAB" solution any more than a Web browser used to access a portal for launching parallel applications is itself a parallel application. The earliest solutions on Intel Hypercube and Ardent Titan used this approach. More recently, the MATLAB*p project at MIT [9], now a commercial project called Star-P®, has revived this approach.

Both of these approaches have significant limitations due to the limited language and library support. Users must discard their existing MATLAB code or choose to extensively re-implement it through the reduced set of constructs these systems provide. During our initial survey, reusability of existing MATLAB code was cited as the most important feature of any parallel computing toolset.

The third approach is to extend MATLAB through libraries or by modifying the language itself. Recently, the MatlabMPI and pMATLAB projects at MIT Lincoln Laboratory and the MultiMATLAB project at Cornell University (with which The MathWorks was also involved) are among the more successful and widely used libraries for parallel computing with MATLAB. Other projects include ParaM and GAMMA projects [10], Parallel Toolbox for MATLAB [11] (which uses PVM for message passing), and various MPI toolbox implementations for MATLAB, including the most recent bcMPI (Blue Collar MPI) from Ohio Supercomputer Center.

To answer the need for a set of parallel MATLAB tools, The MathWorks introduced Parallel Computing Toolbox software and MATLAB Distributed Computing Server in November 2004, (originally named Distributed Computing Toolbox™ and MATLAB® Distributed Computing Engine™, respectively). These fall into the last category of solutions. When we started to expand the capabilities of MATLAB into parallel computing, we decided to target embarrassingly parallel problems, given that our initial survey showed a large number of our users wanted to simplify the process of running Monte Carlo or parameter sweep simulations on their groups' computers. As more sophisticated users have taken to the toolset, we continue to incorporate more language features, including message passing and higher-level abstractions such as parallel *for* loop and global array semantics. Implicit multithreading of computations is another method to parallelize MATLAB computations on a single multicore or multiprocessor machine; the language design team at The MathWorks continues to invest heavily in this project. However, for the purposes of this paper we will focus

our attention on the explicit parallel programming paradigm presented by Parallel Computing Toolbox and MATLAB Distributed Computing Server.

3 Design Goals

Our overarching goal was to extend the traditional strengths of MATLAB to the cluster environment. This encompasses features such as interactivity, multiplatform support, and the ability to express ideas in a language close to mathematical expression while abstracting out non-pertinent detail. We laid out the following design goals for the toolset:

- Users should have the ability to execute arbitrary MATLAB code and Simulink® models on the cluster. This continues to be our most important design goal.
- Users should be able to use the familiar MATLAB language for all the tasks associated with writing *and* executing parallel MATLAB programs.
- Users should have access to first-class language constructs to express parallelism. They should not have to make significant changes to their mode of operation to create parallel programs. Nor should they have to worry about architecture or system-specific constructs, or have to deal with threading, data management and synchronization.
- The language should be completely independent from resource allocation. The same program should be able to function correctly on a single processor or hundreds of processors. Programs should scale appropriately with resources and should function in the absence of cluster resources.
- Programmability will always trump other issues. Users should be able to create programs that are correct, easy to read, easy to debug, and easy to maintain.

4 Framework and Terminology for Discussion

To examine the toolset, we distinguish between the infrastructure and the language components presented by the MATLAB parallel computing tools. The language exposes constructs such as parallel loops, distributed arrays, and message passing functions. The infrastructure component is the machinery underlying the language constructs including mechanics for data and code transfer, setting up of execution environment, etc. In the next sections we examine some of the salient features of these two components.

MATLAB Distributed Computing Server comprises several workers which receive computation tasks from MATLAB client through functions in Parallel Computing Toolbox. In our discussion, we will use the term *workers* as a generic term for MATLAB computational engine processes that run on a cluster as part of the MATLAB Distributed Computing Server.

The Server supports a mode of operation in which the workers function completely independently of each other without requiring any communication infrastructure setup between the workers. However, this communication infrastructure is required for using constructs such as message passing functions and distributed arrays. In this situation,

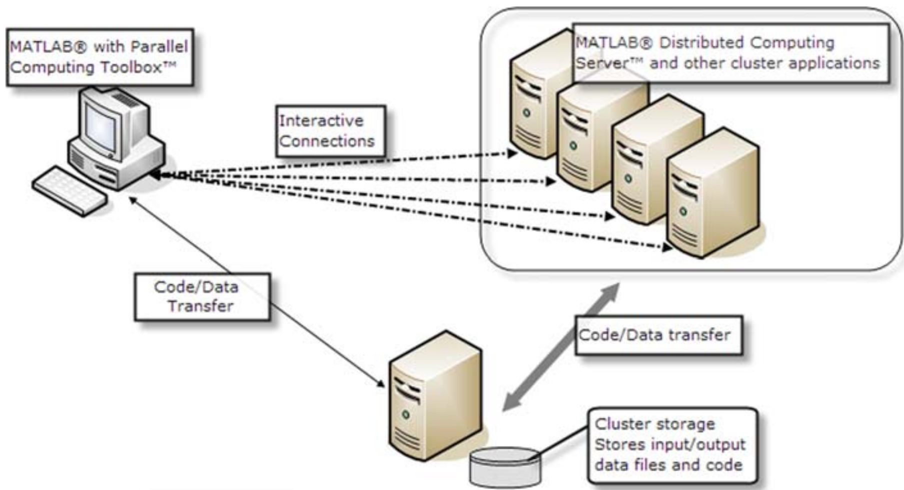

Fig. 1 MATLAB® parallel computing tools in a typical cluster setup. The tools can be used in an interactive as well as batch fashion

we call these connected workers "*labs*". The various message passing functions carry *lab* in their names to make this distinction (Fig. 1).

5 Language

As noted in our design goals, our intention was to extend the MATLAB language with a set of parallel data structures and parallel constructs, which would abstract details as necessary and present to users a language independent of resource allocation and underlying implementation. Our goal was for the language not to require major changes to users' existing working models. Most importantly, it would be as easily programmable as the MATLAB language.

In this section, we describe the parallel MATLAB constructs starting from the lowest-level constructs, message passing functions. We then present distributed arrays and parallel *for* loops, which we think of as the first steps towards Lurie's [12] annotation-based language model in which domain experts make only minimal annotations to their high-level language code to express the intent of using multiple compute resources.

5.1 Message Passing Functions

We introduced message passing functions in the second version of the toolset. As tool builders our intent was not to implement an MPI specification directly in MATLAB language. Instead we wanted to rely on an existing MPI implementation, possibly chosen by the user, and make it available in a highly simplified and easily usable form, but without sacrificing the richness of the message passing programming model. We did not want to provide a "C/FORTRAN MPI"-like programming experience, which other parallel extensions of MATLAB, such as MatlabMPI and bcMPI, aim to provide. We

felt that the loading, initialization, and finally unloading and cleaning up were generic tasks we could handle. End users could immediately start using the available message passing functions independent of the environment they wanted to work in—for example, the parallel interactive environment or batch job environment. Moreover, we intended to provide enough generalization for users to be able to exchange any MATLAB data type without any special set up such as declaring an *MPI_Datatype*.

Message passing functions in MATLAB are high-level abstractions of functions described in the MPI-2 standard. For point-to-point communication, the *labSend*, *labReceive*, and *labSendReceive* functions are available. The *labBarrier*, *labBroadcast*, and *labProbe* functions are direct equivalents of corresponding MPI functions. The environment query functions *labindex* and *numlabs* are equivalent to *MPI_Comm_rank* and *MPI_Comm_size*. Currently, the language does not support user-specified MPI communicators.

5.1.1 A Protocol for Exchanging Arbitrary MATLAB Data Types

One of the most important requirements for this library of message passing functions was the ability to trivially set up the exchange of arbitrary MATLAB data types—including numerical arrays of any precision, structure arrays, and cell arrays (MATLAB arrays that can contain any arbitrary data type). In normal usage, the MPI library expects the knowledge of the size of data that is being exchanged. For an arbitrary MATLAB array we do not expect the user to have this knowledge.

To solve this problem we established a protocol in which we send two messages—the first a very short header message of a known size indicating the expected MATLAB data type, the size of the data and other information as necessary, and the second message containing the actual payload. For small enough data sizes we simply squeeze the entire data into the header message.

For non-numeric data types, we insert the information about the MATLAB type inside the header information and convert the incoming byte-stream on the receiving lab. The actual payload is constructed by serializing the MATLAB data array which can then be deserialized and reconstructed on the receiving lab. For MATLAB data types that can be mapped directly on to MPI data types, e.g., MATLAB double to *MPI_DOUBLE*, etc., we skip this serialization-deserialization step and directly send the data.

This protocol allows for error detection, deadlock detection, and clearing messages and message queues when errors are detected. Messages that are ready are sent as soon as possible without any noticeable impact on the user. However if, for example, it is not possible to send a message and we need to block MATLAB, we use the spare cycles to handle errors such as deadlocks (Sect. 5.1.5). The following sections address some of the salient details for various message passing operations and error detection.

5.1.2 Point-to-Point and Broadcast Operations: labSend, labReceive, and labBroadcast

The *labSend* and *labReceive* functions are the two fundamental point-to-point communication functions available in MATLAB. Users can send any arbitrary MATLAB

data type without any special set up. Thus, the following is a complete program that uses the *labSend* and *labReceive* functions to exchange a MATLAB structure and a MATLAB double array. Note also that multiple destinations are targeted in a single function call.

```
function aMsgPassingProgram()
    source = 1;
    destination = [2, 4];
    if labindex() == source
    % Send some data from source lab
        testData.rpm = 1000; % Set up a structure
        testData.speed = 35;
        labSend(testData, destination); % send a structure
        labSend(rand(1000), destination); % send an array
    elseif any(destination == labindex())
    % Receive on destination lab
        recvdata{1} = labReceive(source);
        recvdata{2} = labReceive(source);
    end
end
```

Users can choose among several variants of *labSend* and *labReceive* function calls.

```
labSend(data, destinations) % internally generated tag
labSend(data, destinations, tag) % user specified tag
data = labReceive() % any source, any tag
data = labReceive(source) % specific source, any tag
data = labReceive('any', tag) % any source, specific tag
data = labReceive(source, tag) % specific source and tag
[data, source, tag] = labReceive(...) % multi-output version
```

We use non-blocking MPI calls (such as *MPI_Isend*, etc.) to implement the *labSend* functionality, but we only return from a call to *labSend* once the MPI layer has finished processing the data on the send side. A call to *labSend* may complete irrespective of whether or not the corresponding call to *labReceive* has started. This is a side-effect of the way MPI libraries implement size-dependent data buffering for send calls; the MPI standard permits send operations to return even if corresponding receive operation has not started. Our build of MPICH2 buffers messages smaller than 256 KB. As a result, for messages smaller than 256 KB *labSend* immediately returns, potentially even before the corresponding *labReceive* (and underlying MPI receive calls) has started. On the other hand, for larger messages, the underlying MPI library forces us to wait until the corresponding *labReceive* (and underlying MPI receive calls) has started. This has important implications in certain cases, for example, where neighboring labs are trying to send and receive data from each other in a cyclic shift pattern. The *labSendReceive* section below proposes a solution for this cyclic shift pattern problem.

Although the *labBroadcast* operation is directly equivalent to the *MPI_Bcast* operation, we do not use the *MPI_Bcast* function, because implementing it in the MATLAB language using the *labSend* and *labReceive* operations allows us to detect cyclic deadlocks and miscommunications. The calls to *labBroadcast* take the following form:

```
%% Case 1: The variable "data" exists on all labs
data = []; % Declare on all labs
if labindex() == senderlab
    data = rand(1e6, 1);
end
shared_data = labBroadcast(senderlab, data);
%% Case 2: The variable "data" exists only on sender
if labindex() == senderlab
    data = rand(1e6, 1);
    % Function call on sender lab
    shared_data = labBroadcast(senderlab, data);
else
    % Function call on receiving labs
    shared_data = labBroadcast(senderlab);
end
```

5.1.3 labSendReceive Operation

As noted above, the *labSend* operation switches between non-blocking and blocking behavior at a certain data size based on the underlying MPI library's buffering behavior. This can cause latent bugs that can manifest for larger problem sizes. Safe code must therefore assume that calls to *labSend* block. However, this can be particularly tricky in the case where labs are attempting to send data in a cyclic shift pattern. Consider the following pattern where each lab sends data to its *labindex()*+1 neighbor and receives from its *labindex()*−1 neighbor. This can be achieved by simply ensuring that labs with odd valued *labindex()* send first and others receive first.

```
offset = 1;
to = mod(labindex() + offset - 1, numlabs()) + 1;
from = mod(labindex() - offset - 1, numlabs())+1;
if mod(labindex(), 2) == 1
   labSend(data, to);
   indata = labReceive(from);
else
   indata = labReceive(from);
   labSend(data, to);
end
```

However, this pattern can be difficult to generalize safely for cyclic shifts of more than 1 lab. With large enough data (where *labSend* doesn't return immediately) and *numlabs()* equal to 4, the above code will deadlock if "to" and "from" were offset by

2 from *labindex()*. In fact there will be two deadlocks: *labindex* 1 and 3 will simultaneously try to send to each other while 2 and 4 are stuck trying to receive from each other. In C using one of the various flavors of sends and receives one could potentially use a buffer where completion can be repeatedly checked for. However, this requires exposing these sends and receives to the user and putting the responsibility of choosing the right set of operations on the user. Another option is to let the user reorder the operations, but this is hard to generalize for an arbitrary number of labs and data sizes. Both options induce complexity beyond the reach of the naïve user.

The *labSendReceive* function was designed to enable this cyclic type communication (or any paired exchange) to be written more simply. Thus, a cyclic shift with any offset can be written safely using *labSendReceive* as follows:

```
to = mod(labindex() + offset - 1, numlabs()) + 1;
from = mod(labindex() - offset - 1, numlabs())+1;
indata = labSendReceive( to, from, data );
```

We prevent deadlocks by using asynchronous send and receive at the C-MPI layer and blocking until all sends and all receives have completed. That is, we still expose a blocking send-receive, but underneath we use asynchronous exchange to prevent deadlocking behavior.

5.1.4 Reduction Operations

The *gop* (Global OPeration) function, along with its specifically targeted variants *gplus* and *gcat*, is the MATLAB equivalent of various reduction and gather operations in MPI implementations including *MPI_Allreduce*, *MPI_Reduce*, and *MPI_Gather*. The expression *result = gop(red_func, x [, targetlab])* reduces the variant array *x* (an array which resides in the workspace of all labs but whose content differs on these labs) using the reduction function *red_func*, and distributes the results to all the labs if *targetlab* is not specified. The *gop* function performs a tree-reduce operation for any built-in or user-specified reduction operation. The *gop* operation, like the *labBroadcast* function, is based on a system of *labSend* and *labReceive* function calls that enables us to do error detections described in the following section.

5.1.5 Error Detection While Using Message Passing Functions

Cyclic deadlocks and mismatched communications are commonly encountered runtime errors in parallel programs using message passing. We implement detection mechanisms for both error types. We set up a dedicated MPI communicator for each error type at the beginning of program execution so that we do not affect the regular execution of the algorithm. Users can optionally turn the error detection mechanisms on for debugging purposes in batch jobs. The detection mechanisms are automatically turned on during interactive sessions.

The deadlock detection algorithm causes calls to *labReceive* to error out if a circular dependency of *labReceive* function calls is detected. Thus the following triggers an error:

```
function realDeadlock
   if labindex == 1
      labReceive( 2 );
   elseif labindex == 2
      labReceive( 1 );
   end
end
```

However, the following code is not really an instance of deadlock because there is no circular dependency—it just happens that no one is sending the required data, but it still causes a hang. This is a case of mismatched communication error where lab 1 blocks waiting to receive data but all the other labs go to completion.

```
function notDeadlock
% assume numlabs > 1
    if labindex == 1
       labReceive; % from any other lab
    end
end
```

This observation that the rest of the labs have gone to completion while one or more labs wait to receive data forms the basis for the test of mismatched communication error condition in our implementation. When one of the labs is waiting for communication, it sends probe messages to test if the lab on which it is dependent has completed. If it has, the lab reports a mismatched communication error.

For the *cyclic deadlock detection* we use a technique described by Chandy et al. [13]. The algorithm essentially tries to detect a cycle in the send-receive call graph. During program execution, if we suspect that a particular lab is in a deadlock we send a small probe message to each process this lab is dependent on. The probe is constructed to identify the path it has taken so far—it contains information about initiator, sender, and target. If a lab is not in deadlock (or is executing) it simply discards the probe message; otherwise it forwards it to the lab it is dependent on. If eventually the initiator receives the probe message, we can say that we are in a deadlock. For example, suppose we are in a deadlock situation where lab 1 is dependent on lab 2 which itself is dependent on 3, which in turn is dependent on lab 1. Suppose lab 1 initially suspects that it is in a deadlock and sends out a probe message to lab 2. Because lab 2 is also waiting, it forwards this message to lab 3 which in turn performs the same operation. At this point the probe message initiated from lab 1 is returned to lab 1 and we immediately determine that we are deadlocked and error out. As we noted above probe messages are sent over a separate MPI communicator.

5.1.6 Interruptibility of Message Passing Functions

The message passing functions in MATLAB can be used in both batch and interactive modes of operation. To support user-initiated interrupts in an interactive session (such as CTRL+C at the parallel command window prompt), as well as breaking out of deadlocks as described above, the execution of message passing functions in MATLAB can be interrupted. More specifically, the execution of message passing functions in MATLAB, i.e., *labSend*, *labReceive*, and functions based on these, can be interrupted provided the corresponding receive or send has not started on the other end. The details of implementation are beyond the scope of this paper.

5.2 Distributed Arrays

While message passing is the most commonly used method to develop parallel programs on computer clusters, it has been criticized as being equivalent to the Assembly language of parallel programming. Global array semantics reduce the programming complexity by abstracting out the details of message passing and letting users write programs that look serial.

In the PGAS (Partitioned Global Address Space) model for SPMD (Single Program Multiple Data) programs, multiple SPMD threads or processes share a part of their address space [14]. This shared space is partitioned with portions localized to each thread or process. Programs exploit locality by having each thread or process principally compute on data that is local to it. MATLAB distributed arrays are an implementation of the PGAS model. These arrays can be constructed by concatenating pieces of similarly sized arrays on the workers, distributing a large matrix, or using specialized constructors that are overloaded forms of their serial counterparts. Some examples of these are shown below:

```
% Distributing an existing array "A" which
% is the same on all labs
dA = distributed(A, 'convert');
% Joining pieces on workers
localB = labindex() * rand(10000, 10000);
dB = distributed(localB);
% Using constructor functions
dC = rand(10000, 10000, distributor());
```

In the first example above, A is a replicated array containing the same values on all labs. The distributed array dA is the same size as A, but the pieces local to each lab hold only a subset of data contained in A. In the second example, *localB* is a variant array containing different values but of the same size on different labs. The calling syntax in the example 'concatenates' arrays along columns. Thus, dB has same number of rows as *localB*, but has *numlabs* times the number of columns of *localB*. Additional arguments can be supplied to specify a data distribution. In this example, the labs need only exchange meta-data to construct the

distributed array d*B*. The third example uses an overloaded *rand* function with a distribution object (*distributor()* is the default distribution object constructor) as the additional argument to make a random distributed array d*C*. Additional arguments to the distribution object constructor can be used to specify a custom data distribution.

Distributed arrays are implemented as a library layer on top of the MPI infrastructure. The user gets a shared memory view of the execution environment. However, we do not make any assumptions about this environment except that we require MPI infrastructure to be set up and initialized for these arrays to be able to function as a parallel data structure.

One of the biggest strengths of distributed arrays is that they work with any MATLAB data type including singles, doubles, cells, structures, as well as sparse matrices. As with the message passing functions, the fundamental designing principle was that a user should be able to trivially use any data type, not just floating-point types.

The user interface provided by our implementation of distributed arrays is quite similar to that of pMATLAB from MIT Lincoln Laboratory. Instead of distribution objects, the MIT Lincoln Laboratory implementation uses a construct called *maps* to indicate how data is distributed across processors [1].

5.2.1 Data Distribution

Distributed arrays are implemented as MATLAB objects where each lab stores a piece of the array. Each piece of this array, in addition to storing the data, also stores information about the type of distribution, the local indices, the global array size, blocking, the number of worker processes, etc.

Distributed arrays support two data distributions—one-dimensional distribution and two-dimensional block cyclic distributions. Users have access to various parameters to specify data distributions for their distributed arrays.

An important characteristic of data distributions in MATLAB is that they are dynamic. This is in contrast to High-performance FORTRAN (HPF) which actually assumes data distribution upfront [15] and Co-array FORTRAN in which distributed arrays are really private arrays hooked up via MPI rank as an index for an additional dimension [16]. The HPF compiler parallelizes code guided by the way data is distributed—shipping or reorganizing code and performing other optimizations based on data distribution. Co-Array FORTRAN essentially provides users a language-level message passing syntax.

With MATLAB distributed arrays, users can change distributions as they see fit by redistributing data. Users can even create distributions on the fly by, for example, reading arbitrary portions of data from a file and concatenating individual pieces to construct distributed arrays. Certain operations and math functions can also change data distribution. For example, in the extreme case, the gather operation brings all the data onto a single lab (provided it fits) leaving the rest of the pieces on other labs empty. Some examples are shown in Table 1.

Table 1 Data distributions with MATLAB distributed arrays

MATLAB commands	Distribution dimension	Size of local part of array on labs
`% Distributed along columns (default)` `dA = rand(1000, 1000, distributor());`	Columns	1000×250 On all labs
`% Redistribute data - distribute along rows` `dB = redistribute(dA, distributor ('1d', 1));`	Rows	250×1000 On all labs
`% Functions can change data distribution` `tdB = transpose(dB);`	Columns	1000×250 On all labs

`% A custom data partition (1 specifies row-based)`
`dist = distributor('1d', 1, [500 250 150 100]);`
`pdA = redistribute(dA, dist);`
	Rows	500×1000 (lab 1)
		250×1000 (lab 2)
		150×1000 (lab 3)
		100×1000 (lab 4)

`% A custom data partition (2 specifies column-based)`
`dist = distributor('1d', 2, [600, 200, 200, 0]);`
`dA = redistribute(dA, dist);`
	Columns	1000×600 (lab 1)
		1000×200 (lab 2)
		1000×200 (lab 3)
		1000×-0- (lab 4)

Examples assume *numlabs()* is four

5.2.2 Indexing and Accessing Remote Data

Various methods are employed by different libraries and PGAS languages to access remote memory. For example, in Co-array FORTRAN each piece of a co-array is a private array, i.e., an $A(m, n)$exists for each worker. Access to a remote element requires a special syntactic construct, the processor rank as the last dimension ($A(m, n)[MPI_RANK]$). This requires explicit knowledge of the data distribution beforehand. Intel® OpenMP for Clusters implements remote data access via system exceptions (an attempt to access memory beyond local array bounds generates the exception) which the library handles by making the necessary arrangements to bring in the required data [17].

With MATLAB distributed arrays, we chose programmability over execution model transparency. There is no syntactic difference in the way users can access elements in MATLAB distributed arrays and regular MATLAB arrays. We take the responsibility of appropriately shipping data as necessary. Thus, if d*A* is a distributed array, d*A*(4, 3) provides user access to the (4, 3) element in the entire d*A* distributed array and not just the piece of d*A* local to a specific worker. Depending on the context, there may be a penalty in terms of the communication costs.

5.2.3 Operations on Distributed Arrays

Distributed arrays may be used with almost all of the nearly 150 core built-in MATLAB functions including reduction operations, indexing, and linear algebra operations such as LU factorization. For dense linear algebra operations we use ScaLAPACK whenever we can. Other algorithms, such as those for sparse matrices, are implemented in the MATLAB language. Figure 2 shows the MATLAB implementation of the NAS Conjugate Gradient benchmark using distributed arrays and parallel functions. Note that the only annotations we must make are in lines 10 and 13, while the actual conjugate gradient iteration implementation remains unchanged.

```
1: function CG
2: %CG     NAS Conjugate Gradient benchmark
3: %   This function is similar to the NAS CG benchmark described in:
4: %   http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf
5: %   See code on page 19-20 for the pseudo code translated below.

6: n = 1400; nonzer = 7; lambda = 20; niter = 15;
7: nz = n * (nonzer + 1) * (nonzer + 1) + n * (nonzer + 2);

8: % Create A, a distributed sparse random matrix
9: % using default distribution scheme returned by distributor()

10: A = sprand(n, n, 0.5 * nz/n^2, distributor());
11: A = 0.5 * (A + A');

12: % I is a distributed sparse identity matrix
13: I = speye(n, distributor());
14: A = A - lambda * I;            % Shift at lambda, an approx eigenvalue

15: % x is a vector of ones
16: x = ones(n, 1);

17: % Outer iteration: exactly same code will work for distributed and
18: % non-distributed arrays.
19: for iter = 1 : niter
20:     [z, rnorm] = cgit(A, x);     % Conjugate Gradient iteration.
21:     zeta = lambda + 1/(x' * z);
22:     x = z / norm(z);
23: end

24: %% ------------------------------------------------------------ %
25: function [z, rnorm] = cgit(A, x)
26: %CGIT  Conjugate Gradient iteration
27: %   [z, rnorm] = cgit(A, x) generates approximate solution to A*z = x.
28: z = zeros(size(x));
29: r = x;
30: rho = r' * r;
31: p = r;

32: for i = 1 : 15
33:     q = A * p;
34:     alpha = rho / (p' * q);
35:     z = z + alpha * p;
36:     rho0 = rho;
37:     r = r - alpha * q;
38:     rho = r' * r;
39:     beta = rho/rho0;
40:     p = r + beta * p;
41: end

42: rnorm = norm(x - A * z);
43: end  % of function CGIT
44: end  % of function CG
```

Fig. 2 Implementation of the NAS conjugate gradient benchmark in MATLAB using distributed arrays. Line numbers are provided for convenience

5.2.4 ScaLAPACK Interface

MATLAB uses several ScaLAPACK routines for parallel linear algebra functions that operate on distributed arrays. We ship a ScaLAPACK build with the parallel computing

toolset. The distributed array functions in MATLAB access ScaLAPACK routines via several native gateway routines (or MEX-functions as they are called in MATLAB).

Direct use of ScaLAPACK presents two problems. First is the calling syntax, which for ScaLAPACK is FORTRAN mixed with C-MPI. The second, which is the most challenging part of using ScaLAPACK, is devising an optimal data distribution and processor gridding for specific problems. ScaLAPACK expects users to lay out the data in a proper fashion. It also expects users to indicate how it must organize the computations. That is ScaLAPACK must know how it must grid the available processors. For example, for 12 processors a user must indicate which grid is desired: a 6×2, 3×4 or 4×3 or 12×1 grid. There are certain ratios that result in optimal performance. Typically this is a fairly opaque and complicated process, a daunting task for an average user.

We make what we think are the right choices for the user. We make sure that the data distribution actually confirms to our choice of the processor gridding. We encode the data and decode the results that are returned from the ScaLAPACK library.

We also try to restructure the data distribution for optimal use. For a large number of nodes, the cost of doing this restructuring ($O(n^2)$) is much less than the benefits one can derive ($O(n^3)$) from proper data distribution and gridding.

5.2.5 *FOR-DRANGE: Parallel Loops Over Distributed Arrays*

The *for-drange* construct lets users iterate over a distributed range. Each worker executes on the piece of range that it owns. The order of iterations is always the same, and given a certain number of workers and the iteration range, the division of the range remains the same across multiple runs. The *for-drange* construct requires loops iterations to be independent of each other and that no communication should occur between labs when executing the loop. These two requirements mean that in a *for-drange* loop one can access only the portion of a distributed array that is local to each lab, i.e., a subscript can access only the local portion of the distributed array. In general, this means the data distribution along the distribution dimension should match the range distribution returned by *drange*.

```
distributedArray = rand(30, 1, distributor());
for itr = drange(1 : 30)
  d(itr) = distributedArray(itr);
end
```

The *for* loop does not own the responsibility of moving code or data around. It simply operates on an already divided iteration range. This is in contrast to the *parfor* loops which we discuss in the next section. Readers will notice the similarity with constructs in languages such as Chapel (*forall* ... *in* ... *do*) [18], HPF (*INDEPENDENT*... *do*...), etc.

5.3 Parallel for Loop (*PARFOR*)

The *parfor* loop is a parallel control flow construct. It is a work-sharing construct that executes the loop body in an order-independent fashion over a set of available workers.

This parallel *for* loop differs significantly from the *for-drange* construct in its intent and design. We discuss this in a later section.

By annotating a *for* as a *parfor*, the user explicitly expresses that the contents of the *for* loop may be executed in any order. If additional computational resources are available (through *matlabpool* Sect. 6.3.1), the underlying execution engine can evaluate the code in parallel for faster results. In the absence of these resources, on a single processor system, *parfor* behaves like a traditional *for* loop. The *parfor* loop requires iterations to be completely independent of each other.

The *parfor* syntax is as follows:

```
parfor (itr = m : n, [NumWorkers])
   % loop body
end
```

NumWorkers is an optional argument that indicates an upper-bound on the number of MATLAB workers the user wants to use for executing the loop body. If fewer resources are available, MATLAB uses the maximum number available.

The *parfor* construct is quite similar to OpenMP parallel loops [19] (parallel for or parallel do directives) or the *for* (parallel by default) construct in the Fortress language specification [20]. The key difference is that OpenMP parallel loops and Fortress loops run on threads within a single process on a single physical computer, while the *parfor* iterations are distributed onto multiple processes, potentially running on multiple physically separate computers. At the same time, we ensure there is a single workspace before and after the *parfor* loop is executed by transporting data and code for execution and gathering results back from multiple workers on the fly. This has interesting implications on how the work distribution is actually implemented. We discuss these next.

5.3.1 PARFOR Mechanics

There are five essential steps in the execution of a *parfor* loop (Fig. 3):

1. Initialization of the resources by the user by executing the *matlabpool* command
2. A static analysis of the loop
3. Transfer of appropriate code and data to the compute resources by *parfor* infrastructure
4. Code execution on workers initiated by *parfor* infrastructure
5. Gathering and collating results from the compute resources, again by *parfor* infrastructure

The initialization of compute resources through *matlabpool* is discussed in the scheduler interface section later. Steps 2, 3, and 5 are the more interesting pieces that we will describe here. The requirements for correct *parfor* behavior fall into three rule categories: syntactic, deterministic, and runtime.

5.3.1.1 PARFOR Syntactic Rules These rules prescribe the permissible constructions that may be used within the *parfor* loop. We prohibit the use of break and return statements from within the *parfor* loop and also place restriction on the iteration ranges

```
>> % Set up data, paths etc.
>> MassSpecDataSetup
>> % Start MATLAB pool for parallel processing
>> matlabpool 4
--Connected to a matlabpool session with 4 labs.--
>>
>> %% Processing a large set of mass spectrometry signals

parfor k = 1 : K
    file     = [repository files{k}];
    D        = textread(file);             % mass-charge & in
    [mz,YR]  = resample(D, 15000);         % resample to 1500

    P        = [3883.766 7766.166];        % Reference peaks
    YA       = alignToRef(mz, YR, P);      % align to P
    YB       = adjBckgrd(mz, YA);          % estimate & adjus
    Y(:,k)   = mslowess(mz,YB,'SPAN',5);   % reduce noise

    if k == 1
        MZ(:,k) = mz;                      % needed once
    end
end
>> %Temp variables (YA,YB,YR,P,mz) not transferred back
>>
```

Fig. 3 An interactive *parfor* session (highlighted). The *parfor* construct detects workers and makes required code and data transfer to and from workers using mechanics described in Sect. 5.3.1

(numeric, positive, non-decreasing only). These rules exist to force the loop to be deterministic and to ensure that the code writer understands some of the restrictions that apply to *parfor* loops.

5.3.1.2 PARFOR Deterministic Rules These rules exist to ensure that the loop is deterministic, (that is, the results will not depend on the order in which iterations are executed), and the necessary components for the loop are initialized. Failure of either of these conditions is a programming error.

In step (2) above, a static analysis of the *parfor* loop body and the surrounding dependencies is performed for these first two sets of requirements. The internal parser performs the standard steps of building a parse tree, resolving names, building a symbol table, and performing a use-define analysis. This analysis results in a classification of variables that appear in the *parfor* body into five types:

1. *Loop index variable*: A variable that is controlled by *parfor*
2. *Broadcast variable*: A variable that is never the target of an assignment and must be sent over from the host
3. *Sliced variable*: An indexed variable used in an expression within the loop body, either on the right-hand side of an assignment, in which case it must be transferred to workers, or on the left-hand size of an assignment in which case it is an output variable and for which data must be transferred back
4. *Reduction variable*: A variable that appears in unindexed expressions of the form

$$r = f(expr, r) \text{ or } r = f(r, expr)$$

If a variable is classified as a reduction variable, MATLAB reduces the values of r from different iterations in any order it chooses. When *expr* is a commutative and an associative operation, then the results of the function f are deterministic, e.g., multiplication. In our implementation concatenation also behaves correctly although it is not an order-independent operation. Thus, the following loop:

```
x = []; % empty array
parfor itr = 1 : 10
   x = [x, itr]; % concatenation into a larger array
end
```

produces $x = [1, 2, 3, 4, 5, 6, \ldots 10]$ in "correct" order and similarly for character strings. We assume that a user deliberately using a reduction function that is not commutative accepts that the resulting *parfor* behavior is nondeterministic.

5. *A temporary variable*: A variable that is subject to a simple assignment within the loop. These are cleared at the beginning and at the end of each iteration. In Fig. 4, any values assigned to variables d and a are lost after the execution of the loop. Moreover, d does not exist outside the loop body (it was not declared before the loop). Any attempt to use d after the loop results in an error. The variable a continues to exist after the loop, but its value remains unchanged from what it was assigned before the loop.

We attempt to classify all the variables according to this scheme. If we fail to classify any variable, we declare the loop illegal, i.e., MATLAB reports an error. Once a determination of the variable types in the loop is complete, *parfor* makes the necessary decisions to chop up the iteration range, package up executable code and data for execution on workers, and determine the result holding matrices after the loop execution. We provide a code analysis tool called *M-Lint* that can be used to determine if a *for* loop is a good candidate for converting to a *parfor* loop.

5.3.1.3 PARFOR Runtime Rules These rules specify the runtime transparency requirement. It places a restriction on the use of MATLAB functions within the *parfor* loop to those that are statically analyzable. Certain constructs in MATLAB, such

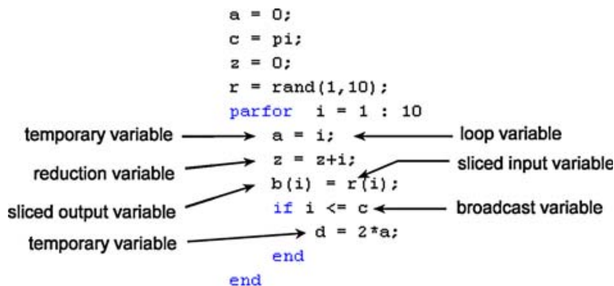

Fig. 4 Classification of variables in a parfor

as *eval*, *evalin*, *load*, and *save* , modify their workspaces in a non-statically analyzable way. Thus, $eval(string)$ or $evalin(string, workspace)$ will modify respective workspaces as defined by the variable *string*, which can be an arbitrary piece of MATLAB code (and in a particularly malicious case even *exit*). Similarly, *load*, which loads a MATLAB data file into memory with an arbitrary set of variables, modifies the workspace in a way that cannot be determined by static analysis alone. As we note above there is a single workspace before and after the *parfor* loop. When multiple workers are trying to operate on this workspace, using such functions will destroy its integrity—that is, there will be no deterministic way to find out what data must be transferred to and from the workers. Attempts to execute a *parfor* loop containing such constructs causes MATLAB to throw an error.

5.3.1.4 PARFOR Code and Data Transfer To support dynamic work sharing across multiple worker processes that potentially run on multiple physically separate computers, the *parfor* construct must provide mechanisms not only to transport the data but also to ship the required code for execution.

In reality, the *parfor* loop is an implementation of the master-worker pattern. In an interactive session the client MATLAB (with which the user interacts directly) acts as the master, while the MATLAB worker processes on the cluster receive work from the master. In a batch session, one of the MATLAB workers is reserved as a master. This setup requires a persistent communication channel between the master and the workers. As we describe in the *matlabpool* section, the persistent communication is set up via socket connections between the client MATLAB and the workers.

The novelty of our implementation lies in the way we transport code and data over this communication channel. While the details of the technique are beyond the scope of this paper, we present some of the salient concepts. MATLAB supports the concept of function handles and anonymous functions. Roughly speaking, function handles are equivalent to pointers to functions in C. Thus, the expression $myFun = @sin$ provides a function handle to a built-in function “sin” and stores it in the variable *myFun*. One can then perform various operations with *myFun* just as one would with the function *sin*. Thus $myFun(pi)$ will produce exactly the same result as $sin(pi)$. Anonymous functions are MATLAB expressions that exist as function handles in the workspace but do not necessarily have a corresponding user-defined or built-in MATLAB function. Thus the expression $myfun = @(x, y)(x\hat{}2 + y\hat{}2)$ creates an anonymous function that returns the sum of squares of the two inputs. The user accesses this expression via its function handle, which is stored in *myfun*. Thus, $myfun(2, 3)$ will produce the same output as $2\hat{}2 + 3\hat{}2$.

To parcel the work and data across workers, *parfor* encapsulates work and data into a function that can be referenced using a function handle, with sliced input variables, broadcast variables, and index variable (see above) as its inputs, and sliced output variables as its output. Depending on the context in which it is executing *parfor* sends this executable payload in two different ways. When it is executing from the MATLAB command line, the executable package is an anonymous function that is transported along with required data over to the workers for execution. When executing within a MATLAB function body (which exists in the form of a MATLAB file with appropriate function declaration), the executable payload is a handle to an automatically generated

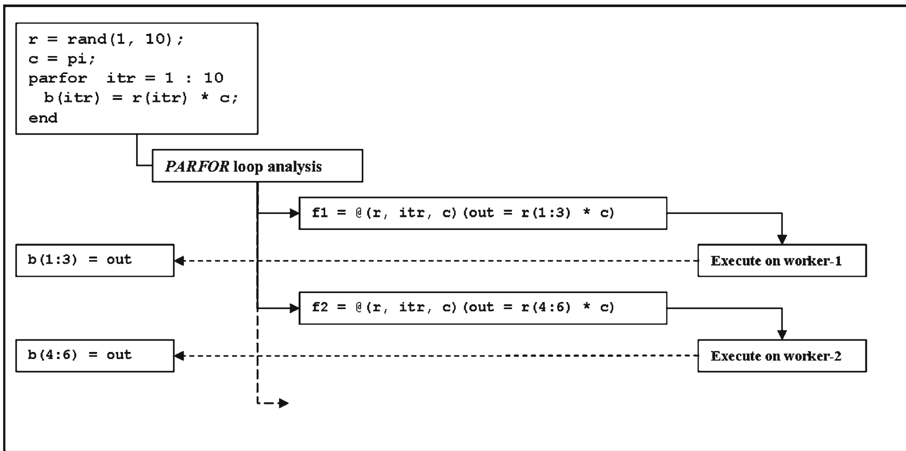

Fig. 5 Task parallelism using *parfor*

nested function from the *parfor* loop body, and assumes the existence and visibility of the MATLAB function on the cluster workers. In this case, the loop body is *not* transported to the workers, unlike when the parallel loop is entered at the MATLAB command line.

This automatically generated payload is serialized and transmitted over the persistent communication channel that is established between multiple workers and the client MATLAB. On workers the package is deserialized, executed, and results returned in the sliced output variables. Pieces of iteration range are dynamically assigned to the workers. A rough schematic of the process for a simple *parfor* loop is shown in Fig. 5.

The parceling and transport of the loop body to various workers happens continuously. Thus in Fig. 5, *f2* is created and dispatched for execution before results from executing *f1* are actually returned. Depending on the availability of workers the iteration range may be divided differently. We use a standard scheme to dynamically ration the loop iterates among the several available workers. For example, one of the runs of the following *parfor* loop with four workers produces a division of iteration range as shown in Fig. 6:

```
parfor k = 1 : 60
  a(k) = max(abs(eig(rand(300))));
end
```

In this example, each of the four workers (on the vertical axis) picks up different pieces of the iteration range. For example, worker 3 was allocated $k = 21$–30, then 45–48, and finally 59–60. The horizontal time axis shows when each worker starts and ends executing its share of iterates. Thus, by the time worker 1 had finished processing 1–10, worker 4 and worker 3 had finished their first round of work and had been allocated other iterates. Worker 2 finished last in the first round and picked up 52–54 range in the next round.

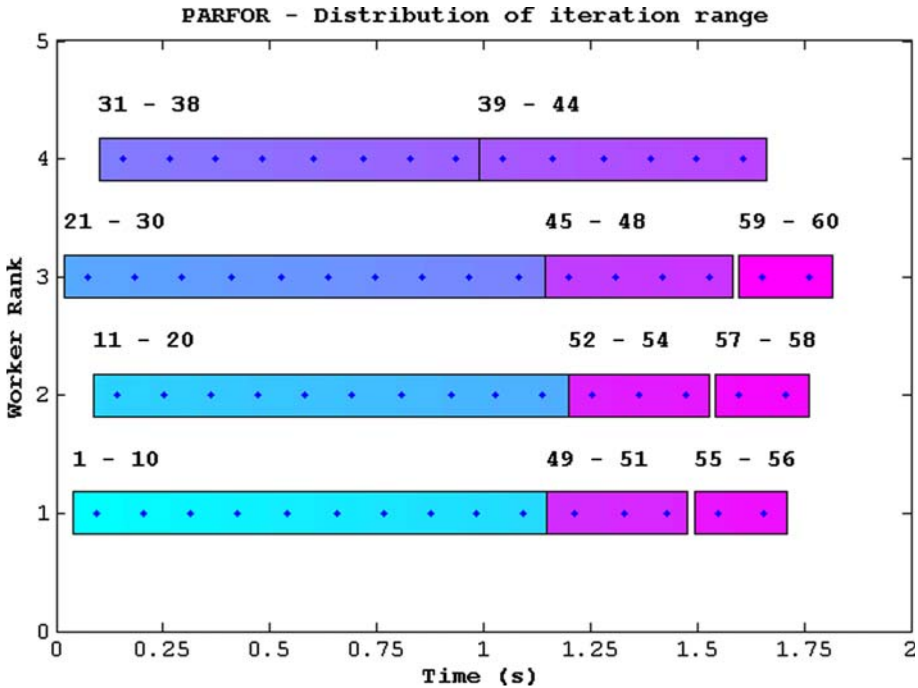

Fig. 6 *PARFOR* distribution of iteration range

5.3.2 A Tale of Two Parallel Loops: PARFOR vs. FOR-DRANGE

Both *parfor* and *for-drange* seem to provide equivalent parallel loop functionality. However, the two constructs are fundamentally different. The biggest difference between the two is that *parfor* does not require any prior parallel computing knowledge for use within a seemingly normal MATLAB program. There is a single workspace before and after the *parfor* loop is executed, because MATLAB handles variable transfer implicitly. There is no commitment to a parallel context.

The *for-drange* construct operates within a context that requires the user to have made the mental commitment to the parallel world. It requires an understanding that there are multiple workspaces (for each lab) and that there is no implicit transfer of variables between these workspaces. Also, there is a static 1:1 relationship between the labs and the iterations executed there. The *parfor* construct does not enforce such a relationship.

Consider the examples in Fig. 7. In the case of *for-drange*, the user has made a commitment that columns of A are distributed across multiple workspaces. In fact, MATLAB will error out if the loop is changed to sweep over the rows (iteration range modified to 1000 and $A(:, k)$ changed to $A(k, :)$) and A's distribution scheme is not changed to an equivalent row-wise distribution. Also, note that in the *for-drange* example variable A resides on the workers. In case of *parfor*, A resides on the user's desktop and the user has no knowledge of how A will be 'communicated' during the course of

```
A = rand(1000, 1e6);
parfor k = 1 : 1e6
  out(k) = f(A(:, k));
end
```

```
A = rand(1000, 1e6, distributor());
for k = drange(1 : 1e6)
  out(k) = f(A(:, k));
end
```

Fig. 7 A comparison of *parfor* and *for-drange* forms

the computations. *f(A(:, 42424))* in a *parfor* loop can be executed on any worker, but in case of *for-drange*, assuming that 100 workers are available, we know that it will be executed on the lab with rank 5.

While *for-drange* is useful in certain execution contexts, we recommend a *parfor* if the only intent is to speed up a *for* loop using additional compute resources.

6 Infrastructure

6.1 Introduction

When we began to expand the capabilities of MATLAB into the parallel computing market we needed to decide on an initial direction for our development, while keeping our longer-term goals in mind and ensuring that our infrastructure would be suitable for our mid-term and long-term needs. In addition, we needed to ensure we could continually deliver a competitive product that solved user needs. It is worth noting that the above statements are basically contradictory. Throughout the project we balanced the competing needs of our current users with what we felt the product needed in terms of infrastructure to try to navigate a development path through feature-space.

Early in the project, we canvassed our user community to try and find out what tools we might provide to help them in a cluster environment. With a strong industry focus (rather than academic) in our polling, we discovered that few of our users had any experience with a cluster-type environment. In addition, virtually none of them had even come across anything like OpenMP or MPI-type programming. This was not particularly surprising for typical MATLAB users in 2003 and 2004. What was surprising was the number who had set up batch scripts to run small-scale Monte Carlo simulations or parameter sweeps on their group's computers overnight. This resonated with much of our experience at The MathWorks, and we decided that our initial parallel computing offering would be a simple scheduler that could be used to run independent MATLAB jobs and gather the results. This would solve a significant class of problem.

Another decision we made early on was to target small- to medium-sized clusters. Looking at our user feedback, we saw that very few users had access to clusters larger than 128 nodes. In addition, many users were happy to purchase their own small-scale cluster for their group or division. This trend of moving compute resources closer to the end-user rather than having centralized resources fits with the MATLAB view of giving users control over what they do and how they do it. While we did not want to preclude running on a large system, if a decision rested on knowing the size of a cluster, then we generally optimized for a smaller, rather than larger one.

6.2 Schedulers

6.2.1 MathWorks Job Manager

Having decided to provide our users with a simple scheduling system that worked well for MATLAB, we evaluated several options. We looked at existing scheduling systems such as Platform LSF® (LSF) and Sun™ Grid Engine (SGE), some service-orientated SOAP-based services, and JINI™/RMI-based services. Whilst realizing that we would eventually need to integrate with existing scheduler systems, we rapidly discarded the notion of requiring something like SGE because the install, configuration, and management overhead would be far too large for our users. We needed a simple, agile system that would work cross-platform (Windows®, Linux®, Solaris®, and Macintosh®) that was easy to set up and maintain, and could be used to build ad-hoc clusters. These requirements appeared to fit well with the JINI/RMI framework, so we built our queuing and dispatch services using that infrastructure.

We should emphasize one design decision that we made early in the project: we decided not to require a shared file-system. With hindsight this decision has made subsequent expansion of our capabilities more difficult to implement, but has meant that we have been able to reach a wider user base. At the outset we did not realize the importance of this decision, and it was only much later in the project when the ramifications became clear. In the original JINI/RMI implementation of our scheduler we used the JavaSpace distributed tuple-space technology as our storage mechanism. However, its performance proved inadequate and we ended up using a simple relational database to store all data needed to define the jobs and tasks. This database can grow rapidly if the amount of data passed to or returned from a task is large.

Another decision we needed to make revolved around how much of MATLAB we should expose on the execution host; or in our product terminology "What was the feature set of the MATLAB Distributed Computing Server?" Was this simply a small process that could undertake linear algebra, or could it understand the full MATLAB language? Should it be able to draw graphics somewhere, just print them out, or have no graphics ability? Should it have a full JVM as MATLAB does, or could it give up this functionality? Looking at existing parallel MATLAB implementations did not give any guidance as they spanned the gamut, from server processes without MATLAB to complete MATLAB processes running on the execution hosts. We debated a variety of design options for how the engine would work, but rapidly settled on using a complete MATLAB process that knew it had no display. The major factor that swayed us in this direction was stability; we already had very extensive internal testing on the MATLAB product, and by using exactly the same code-base we would not have to invest time ensuring that our server product was stable.

However this decision also meant that we needed to consider the launch characteristics of the engine. In most existing schedulers the expected workflow is for the scheduler to start the executable as requested, and shepherd that process until it is finished. However, with MATLAB often installed in a shared file-system environment, the launch time of the executable can be significant. Therefore we opted for a scheduler that operated in a Service Orientated Architecture (SOA), which also fits well

with the JINI/RMI framework. Using an SOA framework ensures that before computations are submitted to the scheduler, the MATLAB engine is already running, ready to undertake work. It also ensures that any failures in cluster startup must be dealt with by an administrator rather than an end user. This fail-fast feature is particularly useful because it gives a clear delineation between the responsibilities of the system administrator and the end user. Administrators are responsible for ensuring that MATLAB engine processes are up and running, leaving end users to assume control of a series of MATLAB processes with which they should feel comfortable. A consequence of this decision is that different users may sequentially share an engine process and there are some associated risks with certain data persistence. We provide an API to restart the engine processes before use if this is a risk a user is not willing to take. In addition, restarting a worker can help reduce the overall memory footprint of that worker by releasing un-freed memory.

6.2.2 Batch Interface

In our original design, we decided to implement the concepts of *jobs* and *tasks*. This was unusual at the time, because most existing schedulers worked as off-load engines where each job was independent of all others and constituted starting some process on an execution host. We decided our users needed more richness to express the relationship between a number of individual tasks. Referring back to one of the use cases our users mentioned, a job would be a parameter sweep and each simulation would be a task within that job. This allowed users to logically group a series of related tasks together, and allowed us to specify things that were common to all the tasks in a single place.

The workflow we expected our users to follow was a typical prototype-test-run cycle. We assumed that generally code would be developed in MATLAB using the interactive development environment and visualization tools available. This prototyping stage would continue, probably on smaller data sets, until it was in a suitable state to try testing on a cluster. Users would try out prototype code on a small subset of the cluster, to ensure that it works correctly. There would be iterations on these prototype-test stages until the code behaved as expected. Finally the code would be run on larger problems. We focused the API design on making this type of usage as simple as possible.

6.2.3 Data and Code Transfer

Unlike most schedulers, our API must provide significant data transfer facilities. This is because the base unit of work in the task API is the evaluation of a single MATLAB function of the form:

$$[O_1, O_2, O_3, \ldots, O_{nOutput}] = foo(I_1, I_2, I_3, \ldots, I_{nInput});$$

To execute this function on the cluster a user would create a task:

$$task = createTask(job, @foo, nOutput, \{I_1, I_2, I_3, \ldots, I_{nInput}\});$$

The `createTask` API function creates a task that is part of a particular job. Once that job is submitted its tasks are run on available engine process. Note that `nOutput` is a required input to the `createTask` function because MATLAB functions can change their behavior depending on the number of requested output variables. Thus to call the function `foo` correctly the engine needs to know how many output variables to specify.

Implicit in the above statement is that the cell array of input variables $\{I_1, I_2, I_3, \ldots, I_{nInput}\}$ needs to be transferred to the worker MATLAB process and the output arguments from the function call to `foo`, $\{O_1, O_2, O_3, \ldots, O_{nOutput}\}$ must be transferred back to the MATLAB client. Our design must incorporate facilities to deal with these transfers being large. In addition to being transferred, this data must also be persistent with respect to the MATLAB client and MATLAB worker processes, because a user who submits a job does not expect that job to be unable to get its input data just because their MATLAB client is no longer running.

This need for large, persistent data storage within the scheduling environment was the motivation for integrating a relational database into our SOA architecture. Both the input and output data related to a given task is stored in this database, and associated with that task and its parent job. We provide API functions to find jobs and tasks, and their associated data, from within the database such that any MATLAB process with the appropriate privileges can interact with these objects. The lifetime of the job and task are defined by the user since it requires an explicit API call to remove them from the database. The internal implementation is such that we only assume the database has a JDBC driver that supports Binary Large Objects (BLOBs). During development we tried a number of different embedded databases to assess their suitability.

Alongside input and output data we must also address the problem of how a user's code gets to the cluster. In general cluster environments this is usually achieved using a shared file-system, or some feature of the scheduler that stages files onto the execution host. Our design reflects the need to incorporate both mechanisms, because whilst the file copy solution is general and will always work, it can be far less efficient that the shared file system approach. The solution we implemented defined two properties of the job, called `FileDependencies` and `PathDependencies`, both of which are cell arrays of strings. All files or directories in `FileDependencies` are compressed and persist within the job, to be uncompressed and added to the MATLAB path on the worker. All directories contained in `PathDependencies` are added to the MATLAB path on the worker (treated as relative to the file system on the execution host).

6.2.4 Interface to Other Scheduling Systems

Having built our own scheduling environment, our next problem was how to replace our SOA scheduling system with an existing scheduling system such as LSF, SGE, etc. This would allow code written for the toolbox to be agnostic to the scheduling system it was run on. Our users already had a mental model of how our scheduling system functioned, and we had to ensure that all other schedulers had similar syntax and essentially the same semantics. To give other scheduling systems the same user model, we had to solve a data transfer problem which we had neatly side-stepped

within our own environment by implementing a database. Having already committed to not requiring a shared file system, we needed to assume one of the following

- The cluster would provide a shared file-system
- The scheduler would provide a data staging system
- The administrator would be willing to run a database-like daemon on the cluster

In practice, most clusters running a scheduling system have a shared file system, so we decided that all the data we had previously been writing into our database could now be written to disk. When there was a shared file system between the client and the execution hosts, this data could then be read by the MATLAB engine started by the scheduler.

However, a relatively common situation that does not have this level of support from its environment arises when communication with the cluster is exclusively over SSH, or some other remote connection protocol. Under these circumstances, users usually run remote terminal sessions on a login node on the cluster and undertake cluster activities from that node. However, with MATLAB, the expectation is that the client machine's MATLAB instance should be able to submit and run jobs on the cluster without requiring a MATLAB instance on the login node. This requires us to somehow use the remote connection protocol to stage files onto the cluster file system before submitting a job to the cluster. We support this mode of operation, provided that we have been given a MATLAB function that will carry out a specific file copy for the cluster.

It is worth noting that we strongly considered the third option above, which requires the cluster system to provide a database-like daemon to support shared data semantics. However, having talked with several large cluster administrators, it became apparent that their tolerance for yet another daemon with large disk and bandwidth requirements was low. While not currently supporting this third option, our solution is architected to allow for the insertion of a framework of this nature.

Once we had made the decision on data transfer mechanisms, we could implement our complete interface to other schedulers. We followed the same job and task model that we had already used with our scheduler. The only assumptions we needed to make about a scheduling system were that it had the ability to start a number of MATLAB processes on the cluster, possibly using *mpiexec* to bind them into an MPI ring, and that we had the ability to pass some environment variables into those processes. Thus far, all scheduling systems we have encountered fulfill these assumptions. It should be noted that the persistence of MATLAB processes is controlled by the scheduling system, which mostly treat a task as finished when the process executing that task exits. Thus, unlike our SOA architecture, each task is executed in a newly started MATLAB process.

6.3 Interactive Interface

MATLAB has always been a cross-platform, interactive environment in which algorithm and code development is intended to occur in a simple, iterative fashion from the command line, leading to the generation of MATLAB files that carry out some task.

The typical MATLAB user has a simulation or data analysis task to undertake, for which they need to develop or use custom tools in MATLAB code. Interactivity with the environment makes this an easy system to learn and use. Thus far our discussion of parallel extensions to MATLAB has addressed uses that rely wholly on the behavior of an external scheduling system. To quote Cleve Moler *'I wrote MATLAB so I didn't have to run a batch job overnight, go down to the computing center the next morning, and pick up my results'*. Moler's point is that the user is at the mercy of the scheduling system which will often have policies that do not allow for true interactive use, where the word *interactive* implies that the requested computation start as soon as the user presses the return key. Not having such an interactive mode would significantly reduce the effectiveness of our tools.

The need for interactivity is a somewhat unique challenge, not faced by many other systems. What we wanted to achieve was a workflow identical to the serial MATLAB development experience, in which our users could interactively use a parallel machine to explore distributed data, write parallel code, and develop tools that could subsequently be used in any of the batch environments discussed above.

Our most important decision about this interactive environment was that it should be independent of the choice of scheduler used to interface to the cluster. In essence, this implies that the interactive layer would sit on top of the job scheduling layers previously described because this had a consistent API to any scheduler. This in turn implies that an interactive environment consists of a number of MATLAB processes running on the cluster, necessarily with a communication channel back to the client machine. This leads to a slightly controversial resource allocation decision, namely that we expect a scheduler to dispatch these specific jobs in an interactive fashion, and that while a user has an interactive session running, our processes will use resources on the cluster nodes until the user closes the session.

In particular, our processes have a significant memory footprint that is not released until the session is finished, and when computing or communicating with other processes we will not endeavor to reduce our CPU usage to allow other processes CPU time on the system. We believe that while this policy is currently at odds with some cluster administrators' views, it represents the most viable way of exposing interactive parallel computing to users.

Looking back at the history of computing, we see that early computers had batch-type operation with clearly defined time slots allocated for jobs (remarkably similar to some of today's batch clusters). However, as machines became more available and the ownership of a machine moved to an individual, so the assumptions about resource usage of programs changed. We believe that the same changes will occur in the cluster environment, with single people or small groups having exclusive access to the resources available.

Cross-platform use was another requirement that our interactive environment needed to fulfill, particularly where the client machine is a Windows desktop and the execution hosts are part of a UNIX cluster. This, coupled with the need for effectively one-sided communication within the processes in the interactive session, required us to implement our own communication protocol, since a general MPI2 implementation would not provide enough functionality.

This second communication layer between the client and lab processes is used as a control layer rather than as a data transfer layer. This should be contrasted with the MPI communication layer between the lab processes on the cluster, which is exclusively used to transfer data between different parts of an SPMD program. Obviously some data needs to be transferred over the control layer, but it is not optimized for this purpose. It is intended to enable the operation of our parallel language constructs (such as *parfor*) by transferring the requisite control messages that execute the remote code, and returning relevant information back to the client, such as any display output, completion statuses, etc.

It is worth pointing out that this second communication layer joins the client machine with the back-end processes, and thus it is likely that it will need to traverse a firewall. This means it is important to consider in which direction the connection of the layer is implemented. We could ask the remote processes to open sockets and let the client connect, or we could open a socket on the client and expect connection back from the remote processes. We opted for the latter approach because it is much more likely that a cluster will allow out-going connection attempts than incoming connections. With hindsight this was a good strategy because the alternative would have required administrators to create specific firewall rules for our connections, which reduces the security of their clusters. We rejected a general tunneling approach (using for example SSH) as that infrastructure is not generally available cross-platform, and is not always enabled even if it is installed.

6.3.1 The Parallel Command Window and MATLABPOOL

The control communication infrastructure allowed us to develop a parallel command window (`pmode`) that is a counterpart to the normal MATLAB command window. This window provides a command line interface to an SPMD programming model with interrupt capability and also displays output from all the computational processes. This interface allows both prototyping and development of SPMD algorithms and interactive use of the distributed array language features. In fact, all the code in the distributed array layer was developed using the parallel command window. The parallel command window is a relatively simple use of the control layer, where simple evaluation request messages are sent to all labs when a user types a command in the parallel command window. All display output from the labs is streamed back to the client machine using observed return messages (messages are returned from remote processes, received by the I/O infrastructure and are then passed on to designated ‘observers’ for further processing). The parallel command window has display features that allow this output to be viewed in several different ways. Interrupt and other control requests can be sent as messages that are out-of-band and affect all the labs. The only relatively difficult part is ensuring that the state of all the labs remains consistent when a user types a command that might affect one lab differently from others (Fig. 8).

The problem with the current parallel command window as described above is that it is not the MATLAB command window. The problem with having two windows is that users have to choose whether they are currently trying to work in serial or parallel and then use the appropriate window. Getting variables from the serial to the parallel workspace, or vice-versa, requires users to say that they want a data transfer to occur.

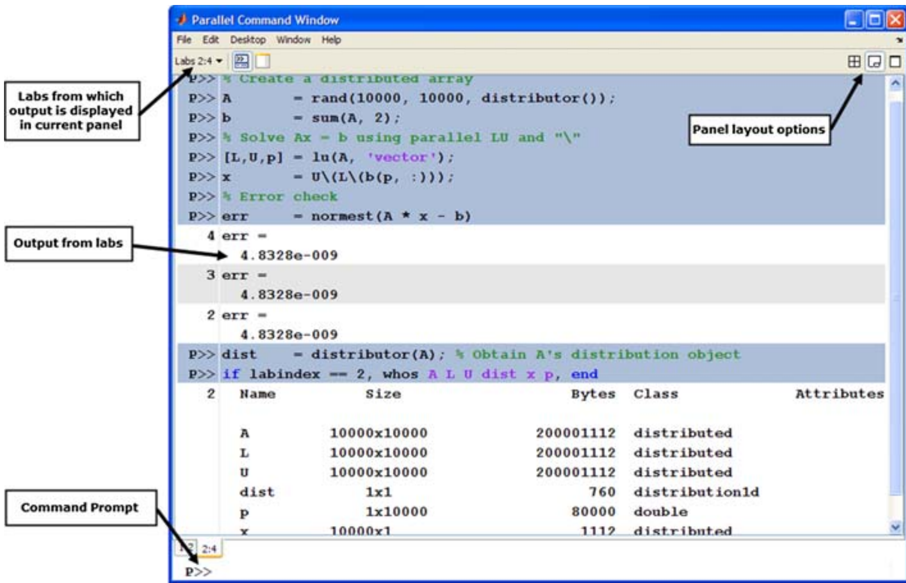

Fig. 8 An interactive session in parallel command window

This is currently not just a brief annotation on the serial MATLAB language that we described at the beginning of the paper.

Our solution to this problem uses the concept of a pool of MATLAB engines combined with language annotations and keywords. This will allow us to express parallelism within the language rather than through the use of a particular command window. We noted that the parallel language features, such as *parfor*, would be intermingled with serial code, and that they would be independent of actual resource request and allocation. However, the language does need to use extra processes when it hits a parallel construct. These processes are held in what we term a *matlabpool*, which is just a series of MATLAB engine processes in an MPI ring, with a control communication connection back to the client MATLAB.

When the client is running serial code and encounters a *parfor* language construct, it parses the construct as described previously and then runs code to see if there are any available resources in the current *matlabpool*. If there are no resources, then a temporary interface to a local lab is set up and the parsed code is run locally. If there are available resources, an interface to the remote labs is returned. This has the same API as the local interface and the client code run is the same; however, the result is that multiple requests are sent out simultaneously across the control layer and many labs start working on small intervals of the *parfor* loop at the same time. In this way, the parallel language constructs independent of the resources used to carry out their processing.

Figure 9 shows the salient parts of the software stack that enable both the parallel command window and the *matlabpool* features. Within the Parallel Computing Toolbox the API to a parallel job is consistent across all schedulers. This job layer is exposed to users and is the main interface to batch control of schedulers within the

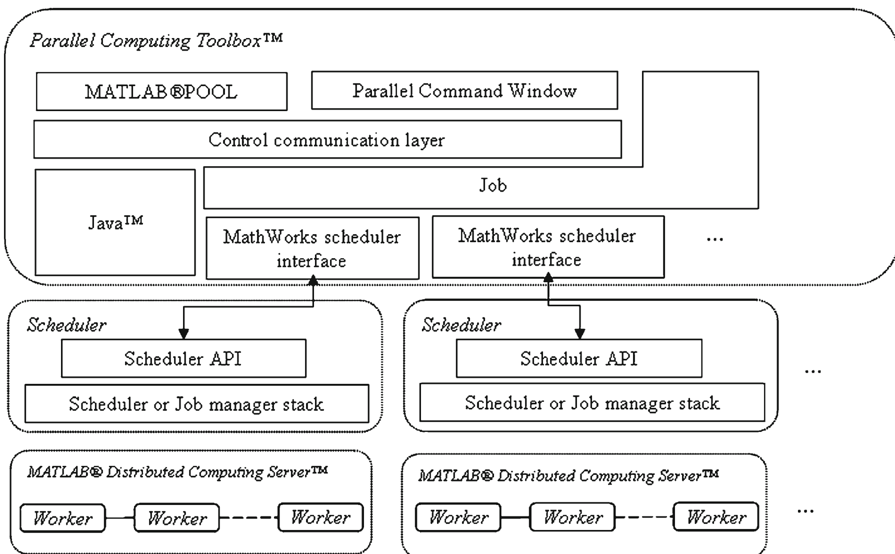

Fig. 9 Infrastructure for schedulers, jobs and parallel interactive sessions. The uniform interface provided by the tools lets users connect to clusters managed by different schedulers, with different number of workers from the same MATLAB session without making code changes

toolbox. Built on top of that job layer is the control communication layer, which is used by both *matlabpool* and the parallel command window. Each scheduler interface is slightly different, reflecting the differences in schedulers. However, the schedulers all have the common feature that they will be starting, or already have started, a number of MATLAB engine processes, which can connect back to the client machine to form the control communication layer (Fig. 9).

6.4 Message Passing Infrastructure

The message passing infrastructure tries to satisfy two key requirements: uniform messaging behavior across platforms, and to provide a uniform interface to dependent layers built on top of this. Several parallel MATLAB constructs use this infrastructure, including the message passing functions, distributed arrays, functions that operate on distributed arrays, and the ScaLAPACK interface (Fig. 10).

There are two important layers in the MATLAB MPI infrastructure: (a) the MPI library layer, and (b) the binder layer which provides a uniform interface to the underlying MPI library layer. We discuss some of the salient details next.

6.4.1 MPI Library Layer

The MPI library layer is a shared library build of an MPI-2 library implementation. By default, we provide an MPI implementation based on MPICH2 distribution from Argonne National Laboratory. In particular, we use the *ch3:sock* device build

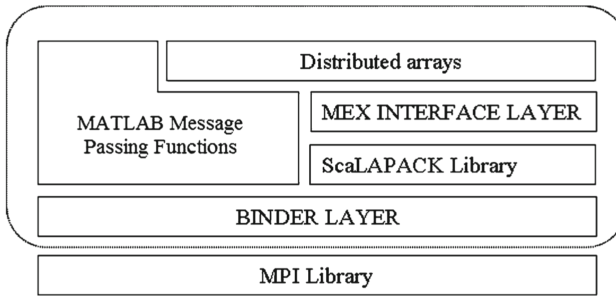

Fig. 10 Message passing infrastructure

of MPICH2 since it happens to work in the same way across all platforms. The MPI library is built as a shared library that is loaded on demand and is not linked directly into MATLAB. We also realized that the MPICH2 library build we supply is a lowest common denominator library. For special hardware requirements (e.g., non-Ethernet connections between nodes) users may want to use another MPI library. Therefore, we support any user-supplied MPI implementation provided that the library satisfies certain requirements. In particular, we expect the user-supplied MPI library to be binary compatible with MPICH2, i.e., the underlying types of various MPI data types such as *MPI_Comm*, MPI_Group, etc. and values of various MPI constants such as *MPI_COMM_WORLD*, MPI_INT, MPI_DOUBLE, etc., are the same.

The binder layer, described next, helps provide a uniform interface to the ScaLAPACK libraries and message passing functions in MATLAB.

6.4.2 Binder Layer

The binder layer is an abstraction layer that provides MATLAB with a standard binary interface to all MPI library functions, given that users can plug-in their own MPI library implementations provided that these are binary compatible with MPICH2. The message passing functions in MATLAB as well as the ScaLAPACK library communicate with this binder layer enabling uniform behavior irrespective of the MPI library being used. When users specify their own MPI library implementation, the binder layer remaps function pointers to the right functions in the MPI library. This is the layer that enables us to use any compatible MPI library without significantly changing the user-visible layers above.

6.4.3 Launch Mechanics with MathWorks Job Manager

MATLAB parallel jobs can be launched in two ways. With all third-party schedulers we rely on the schedulers to provide adequate mechanisms to launch MATLAB workers (as applications that are brought up for each job and shut down after completion). The MPI communication setup parallels typical usage of MPI-based programs. The MPI shared library is loaded and made available to the MATLAB workers, which call

necessary initialization functions. We spawn a few communicators to keep communication namespaces clean and separate.

However, with the MathWorks job manager MATLAB workers remain running between jobs as a service or daemons. As we note in the Schedulers interface section above, we make a distinction between two types of jobs: *distributed* and *parallel* jobs. A parallel job requires an MPI infrastructure, while a distributed job does not. This presents an unusual situation in which MPI must be loaded and communication set up between already running processes. We use functions provided in the MPI-2 standard that let us connect and establish communication between MATLAB worker processes that have been launched independently and not under *mpiexec*.

The method we use is essentially a three-step process.

1. When a parallel job is launched, the MathWorks job manager launches a single parallel task on a lab and defines its rank as 1. This lab opens a port using *MPI_Open_port*, which returns a description of how other labs can connect to that port. The lab 1 opens this port for others to connect to and passes this information back to the job manager.
2. The job manager launches a parallel task on other labs, passing the port information received from lab 1 to all of these labs. At the end of this stage, all the MATLAB workers (labs) are trying to run the task and are aware of the port they need to connect to but are not connected yet.
3. The next stage is a series of connect-accept iterations, in which each lab tries to connect to the others. As an illustration, imagine that we have four labs. In the first iteration lab1 calls accept and all the others call connect. Only one of the others will succeed; suppose that lab2 is successful. Next lab1 and lab2 call accept, while the others call connect, at which point another lab is brought into the MPI ring. At the end of this stage, all the labs are connected to each other and we know that each lab is capable of exchanging data with all others.

7 Bringing it All Together

The MathWorks toolset aims to provide users with a set of constructs that can be applied to exploit various types of parallelism with minimal effort. Thus, *parfor* is a way to exploit task parallelism, while distributed arrays and parallel functions target data parallel problems. The toolset aims to provide adequate levels of control to end users, who can choose to use a specific subset to exploit parallelism in their applications, from low-level message passing functions to high-level distributed arrays and parallel loops. Each of these constructs provides various parameters that users can tweak to apply to their specific problem. Even without these specific modifications, we are able to make decisions for users that we hope are close to optimal. Our goal is to maintain the highest degree of programmability. Programmability for us will always trump performance.

The toolset was designed to be portable across multiple platforms and architectures: Linux, Macintosh (Intel, 32-bit), Solaris (64-bit), and Windows. Barring the assumption that MPI can function in a given environment, and that MATLAB worker processes can be launched, we do not make any other assumptions about the operating

environment. This cleanly separates the algorithm from the underlying architecture details.

As we move forward we ask ourselves several questions. At the beginning we distinguished between the implicit parallel programming model through multithreading and the explicit model that we describe in this paper. Given the proliferation of multi-core, multiprocessor desktop computers and predictions of tens or hundreds of processing cores we are curious as to what the interplay will be between the two models. Multithreading (computational threads) in MATLAB is exposed via a user settable preference which enables certain functions to switch to multithreaded versions for appropriate problem sizes. Since threads execute within a single process, utility of multithreading is restricted by the limitations imposed on the individual processes themselves (e.g. limitations of 32-bit computers). Similarly, using multiple workers to solve large problems has its own share of difficulties. Users could use a hybrid scheme making sure to avoid resource contention [22]. But then the user must make a decision on number of computational threads and number of workers. For example, on a 64 core computer 8 workers each with 8 computational threads could yield better performance than 16 workers with 4 computational threads each. The picture is further complicated in the presence of accelerators and coprocessors such as GPUs and FPGAs.

We also worry about the tradeoffs between ease of use and performance. It is clear that unless parallel programming tools are simplified their adoption is a lost cause. How should MATLAB as a technical computing environment stay true to its goal of enabling users to express their intent and algorithms in the simplest possible way while scaling both in performance and problem sizes with available resources?

Tied to this is how do we enable users to actually use the parallel programming constructs we provide. Even with a highly simplified language it is possible that users are turned away by complexities that are not abstracted out. Thus, the question is what level of complexity users would tolerate.

Finally, for the firms that have large amount of legacy MATLAB code, code changes are also a major concern. Would these firms invest time to re-code some of their MATLAB algorithms, and what amount of code changes would they tolerate? Quite clearly, these users will also require some of the specialized toolbox functions used in their codes to be parallelized. We will have to actively pace and sequence ourselves to deliver this functionality.

We conclude by noting that MATLAB has evolved rapidly and must continue to do so or risk becoming irrelevant in the new world order.

Acknowledgements We would like to acknowledge the contributions of the parallel language design team members at The MathWorks who simplified the esoteric details for the purposes of this paper, particularly Penny Anderson, Edric Ellis, Mike Karr and Brett Baker. We would also like to thank Cleve Moler, Chief Scientist and cofounder of The MathWorks, for being our fiercest critic.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Travinin Bliss, N., Kepner, J.: pMatlab parallel Matlab library. Int. J. High Perform. Comput. Appl. **21**(3), 336–359 (2007). doi:10.1177/1094342007078446
2. Kepner, J.: MatlabMPI. J. Parallel Distrib. Comput. **64**(8), 997–1005 (2004). doi:10.1016/j.jpdc.2004.03.018
3. Trefethen, A.E., Menon, V.S., Chang, C., Czajkowski, G., Myers, C., Trefethen, L.N.: Multimatlab: MATLAB on Multiple Processors. Technical Report, UMI Order Number: TR96-1586, Cornell University (1996)
4. Hudak, D.E., Ludban, N., Gadepally, V., Krishnamurthy, A.: Developing a computational science IDE for HPC systems. In: Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications, International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 20–26 May 2007
5. Moler, C.: Why isn't There a Parallel MATLAB. The MathWorks Newsletter, Spring 1995
6. Jacobson, P., Kågström, B., Rännar, M.: Algorithm development for distributed memory multicomputers using CONLAB. Sci. Prog. **1**, 185–203 (1992)
7. DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., Padua, D.: FALCON: a MATLAB interactive restructuring compiler. In: Languages and Compilers for Parallel Computing, pp. 269–288. Springer-Verlag, New York (1995)
8. Release Notes: MATLAB Compiler (version 4.0), The MathWorks (2004)
9. Husbands, P., Isbell, C.: Matlab*p: a tool for interactive supercomputing. In: The Ninth SIAM Conference on Parallel Processing for Scientific Computing (1999)
10. Panuganti, R., Baskaran, M.M., Hudak, D., Krishnamurthy, A., Nieplocha, J., Rountev, A., et al.: GAMMA: Global Arrays Meets MATLAB. Technical Report OSU-CISRC-1/06-TR15, The Ohio State University, January 2006
11. Hollingsworth, J., Liu, K., Pauca, P.: Parallel Toolbox for MATLAB PT v. 1.00: Manual and Reference Pages. Wake Forest University (1996). URL: www.mthcsc.wfu.edu/pt/pt.html
12. Lurie, R.: Language Design for Uncertain Future. HPCwire, September 2007
13. Mani Chandy, K., Misra, J., Haas, L.M.: Distributed deadlock detection. ACM Trans. Comput. Syst. **1**(3), 145–156 (1983)
14. Carlson, B., El-Ghazawi, T., Numrich, R., Yelick, K.: Programming in the Partitioned Global Address Space Model. Tutorial at Supercomputing 2003, November 2003. URL: http://www.gwu.edu/~upc/tutorials/tutorials_sc2003.pdf
15. HPF Language Specification, Version 2.0, 31 January 1997
16. Numrich, R.W., Reid, J.K.: Co-array FORTRAN for parallel programming. FORTRAN Forum **17**(2), (1998)
17. User Manual, Cluster OpenMP*, Intel Corporation, 2005–2006.
18. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007). doi:10.1177/1094342007078442
19. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel Programming in OpenMP. Morgan Kaufmann, Mosby (2000)
20. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G.L., Jr., Tobin-Hochstadt, S.: The Fortress Language Specification, Version 1.0 β, Sun Microsystems, Inc., 6 March 2007
21. Kepner, J.: Programming with MatlabMPI, Web documentation, URL: www.ll.mit.edu/MatlabMPI
22. Moler, C.: Parallel MATLAB: Multiple processors and multiple cores, The MathWorks News and Notes, June 2007