



# Solving Large-Scale Linear Algebra Problems Using SPMD and Distributed Arrays

By Edric Ellis

Today's linear algebra problems are often too large to fit into the memory of a 32-bit machine. Many are even too large for a single 64-bit processor. This is especially true if the application processing the data does not take advantage of additional processing cores on the computer. A better solution is to use the combined memory and computing power of several machines operating together while maintaining the structure of the original program.

This article describes how to solve large linear algebra problems by spreading them across multiple machines using distributed arrays and the single program multiple data (SPMD) language construct, `spmd`, in Parallel Computing Toolbox™. One advantage of SPMD is that when a computation is run across multiple machines, you have access to the machines' combined physical RAM as well as to their combined processing power. And by working with `spmd`, you can express explicit parallelism in the MATLAB® language.

The example described here was joint winner in the High Performance Computing Challenge.

## Products Used

- MATLAB®
- MATLAB Distributed Computing Server™
- Parallel Computing Toolbox™

## The High Performance Computing Challenge

The High Performance Computing Challenge (HPC) is one standard measure of the performance of the world's most powerful supercomputers. The challenge comprises benchmarks measure six characteristics of system performance:

- The number-crunching performance of the processor cores when applied to solving large systems of linear equations
- The rate at which the cores can be supplied with data from local memory
- The rate at which small messages can be passed between processors
- The efficiency of computing the FFT of a long distributed vector
- How quickly pairs of processors can communicate
- Latency and bandwidth of inter-process communications

These benchmarks test the ability of the parallel system to solve huge linear systems of equations and determine

how well suited a given system is to real-world applications, such as detailed climate simulation or other high-resolution physics simulations.

Each year, two categories of competition are held: Class 1, for the overall best performing system, and Class 2, for the most elegant implementation of four of the benchmarks. The MathWorks entry was a joint winner of Class 2.

## Introducing SPMD Programming

In SPMD, multiple instances of the same program run on different MATLAB workers. Each instance has a unique index, which it uses to determine which part of the problem to operate on. In high-performance computing, instances of an SPMD program usually communicate with each other using Message Passing Interface (MPI) functionality. MPI provides not only point-to-point communication but also collective operations that enable, for example, a combined total to be aggregated efficiently from contributions calculated by each process.

## Applying SPMD: A Simple Example

In this example, SPMD programs approximate the value of  $\pi$  by integrating “ $4/(1+x^2)$ ” over the range  $[0, 1]$ .

To perform this calculation in parallel, each process calculates the integral over a portion of the range corresponding to its unique index. Finally, communication is used to compute the total.

Using the `spmd` parallel language construct in Parallel Computing Toolbox, we can express the parallel program as shown in Figure 1.

When MATLAB enters the `spmd` block, the body of the block is executed simultaneously on the MATLAB worker processes, previously launched using the `matlabpool` command (see sidebar).

Computation on the workers proceeds as follows:

1. Each lab uses its unique `labindex` and the shared value of `numlabs` to calculate a subset of the full range over which to integrate the function “ $f$ ”.

2. `my_integral` is calculated on each lab using `quadl` over the distinct ranges.

3. `gplus` is called on each lab with the different values of `my_integral`. `gplus` uses communication to combine the results and return the same global total result to each lab as `total_integral`.

Outside the SPMD block, the value of `total_integral` can be accessed on the MATLAB client as a Composite object—effectively, as a reference to the values stored on the labs.

Figure 2 shows the ranges of integration when there are four workers.

## Distributed Data Types

To harness the processing memory and power of SPMD, it is convenient to use the high-level distributed array rather than the SPMD block. In a distributed array, a single large array is stored in the RAM on several machines. For example, consider a large square matrix “ $D$ ” spread across the memory of 4 workers. In the default distribution

```
% The expression that we wish to integrate:
f = @(x) 4./(1 + x.^2);

% Enter an SPMD block to run the enclosed code in parallel on a
number
% of MATLAB workers:
spmd

% Choose a range over which to integrate based on my unique index:
range_start = (labindex - 1) / numlabs;
range_end   = labindex / numlabs;
% Calculate my portion of the overall integral
my_integral = quadl( f, range_start, range_end );
% Aggregate the result by adding together each value of "my_integral"
total_integral = gplus( my_integral );
end
```

Figure 1. Parallel program for approximating the value of  $\pi$  by integrating “ $4/(1+x^2)$ ” over the range  $[0, 1]$ .

### matlabpool

The `matlabpool` command launches several MATLAB *worker* processes and connects them to the desktop MATLAB. These workers are essentially full instances of MATLAB. They can be launched either on the same machine as the desktop MATLAB or on a cluster of machines. The workers are available to execute the body of SPMD blocks while the desktop MATLAB waits for them to complete. The desktop MATLAB can then access the processing power and memory of a cluster of machines.

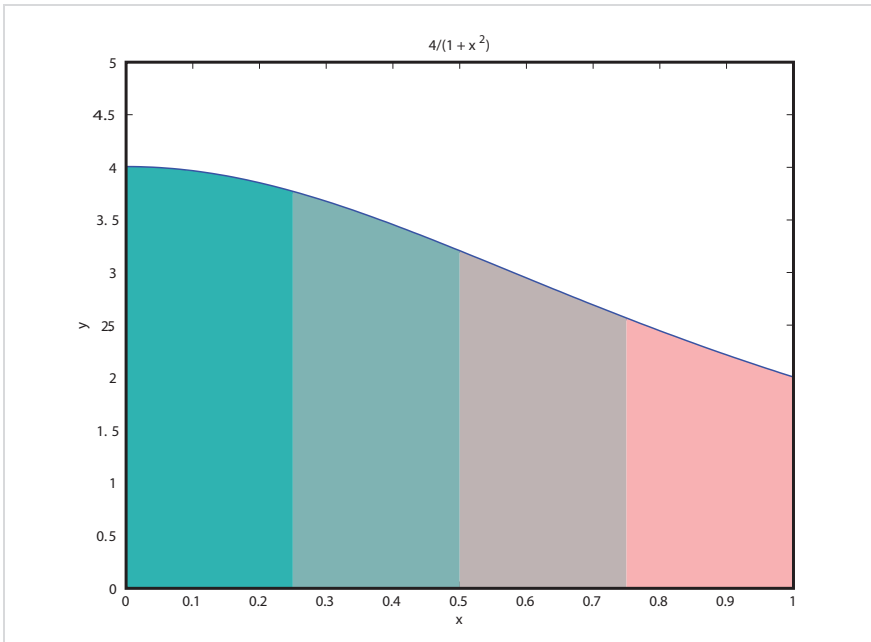


Figure 2. Ranges of integration with four workers. Each worker computes a different shaded region.

scheme, the elements are arranged as shown in Figure 3.

Distributed arrays can be accessed both inside and outside SPMD blocks. For example, the following code creates a distributed array of random numbers by using the client-side API and then uses the value within an SPMD block.

```
D = distributed.rand(40000);
% a very large distributed
random matrix

spmd

m1 = max( svd( D ) )
% Calculate the largest singular
value

end

m2 = max( svd( D ) );
% Calculate the largest singular
value
```

The large array is allocated only once; the Parallel Computing Toolbox infrastructure ensures that, in all cases, “D” refers to the

memory allocated on the workers.

When “D” is passed into an SPMD block, the variable still refers to the same data, and the worker-side interface is used. In the previous code sample, the “svd” method of

distributed arrays is used to calculate the singular values of “D”, and then “max” is used to find the largest. These methods use MPI communication between the labs to perform the computations, but the details of the communication patterns required are hidden. In this way, many complex calculations can be computed without the need to program any communication.

The second calculation outside the SPMD block performs the same computation using the client-side interface to distributed arrays. This allows distributed arrays to be used without explicitly entering an SPMD block—the computations are still performed on the workers. The worker-side interface to distributed arrays gives more control over details such as the precise layout of the data across the memories of the workers, at the expense of a little more complexity.

We will use distributed arrays in MATLAB to tackle one of the HPC benchmarks.

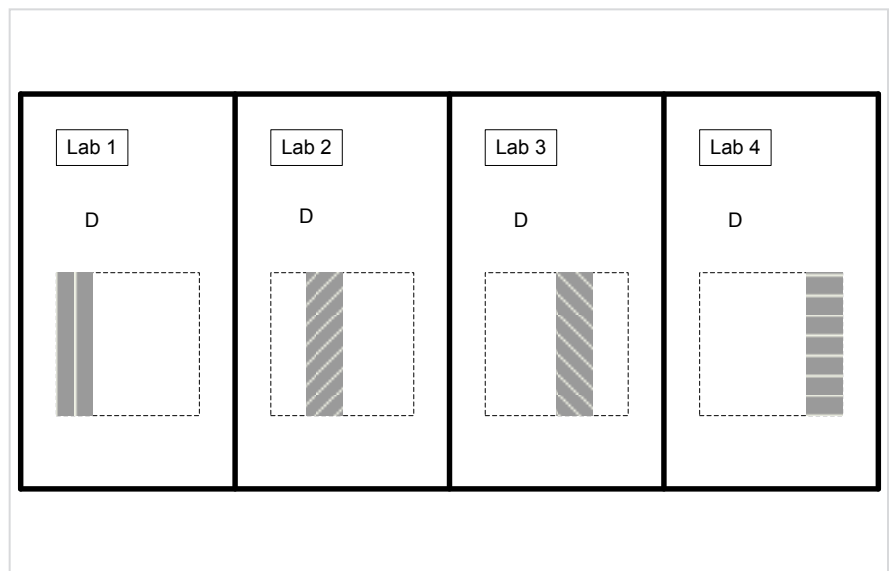


Figure 3. Distribution scheme for a large square matrix, “D.”

## Tackling the Large Dense Linear Algebra Benchmark

This benchmark tests the performance of a parallel implementation of the MATLAB backslash operator. The MATLAB code can be implemented entirely from the MATLAB desktop using the client-side distributed arrays. The code for the benchmark itself simply sets up a large random matrix “A” and a corresponding random vector, “b”, and then solves the linear system:

```
% Build the arrays using the
memory on the worker machines

A = distributed.rand( N );
b = distributed.rand( N, 1 );

tic % Start of timed region
    x = A \ b; % Solve the linear
    system
t = toc; % End of timed region
```

Note that to implement this important calculation, very few lines of MATLAB code are required, and that these lines are almost identical to those needed to solve smaller problems using standard MATLAB arrays. The distributed array syntax provides a natural way to implement the computation for execution on the workers.

The accuracy and performance of the solution must then be calculated (Figure 4).

We measured the performance of the MATLAB implementation on a cluster of 16 Linux machines. Each machine has two dual-core 285 AMD Opteron processors (2.6GHz, 1MB cache per core), and 4GB of RAM (1GB per core). The network interconnect is standard GigE.

Figure 5 compares the results to the HPCC-supplied implementation. The performance of our example is about 25% slower than that of the benchmark.

```
% Check numerical correctness
r1 = norm( A * x - b, Inf ) / ( eps * norm( A, 1 ) * N );
r2 = norm( A * x - b, Inf ) / ( eps * norm( A, 1 ) * norm( x, 1 ) );
r3 = norm( A * x - b, Inf ) / ( eps * norm( A, Inf ) * norm( x, Inf )
    * N );
if max( [r1, r2, r3] ) > 16
    error( 'Failed the HPC HPL benchmark' );
end

% Performance in GFlops
perf = ( 2/3 * N^3 + 3/2 * N^2 ) / t / 1.e9
```

Figure 4. Calculating solution accuracy and performance.

## Interpreting the Results

The performance of these computations was very close to that achieved by a highly tuned reference implementation, indicating that this simple interface enables large computations to run efficiently on a cluster. The reference implementation is highly optimized for the specific problem of the HPL benchmark, and many contributors have

increased the efficiency of this implementation. In MATLAB we use the widely available ScaLAPACK library to implement the benchmark. This library provides generally applicable implementations of many parallel linear algebra functions (such as various matrix factorizations) at the expense of being slightly less well tuned for the particular case of the HPL benchmark. ■

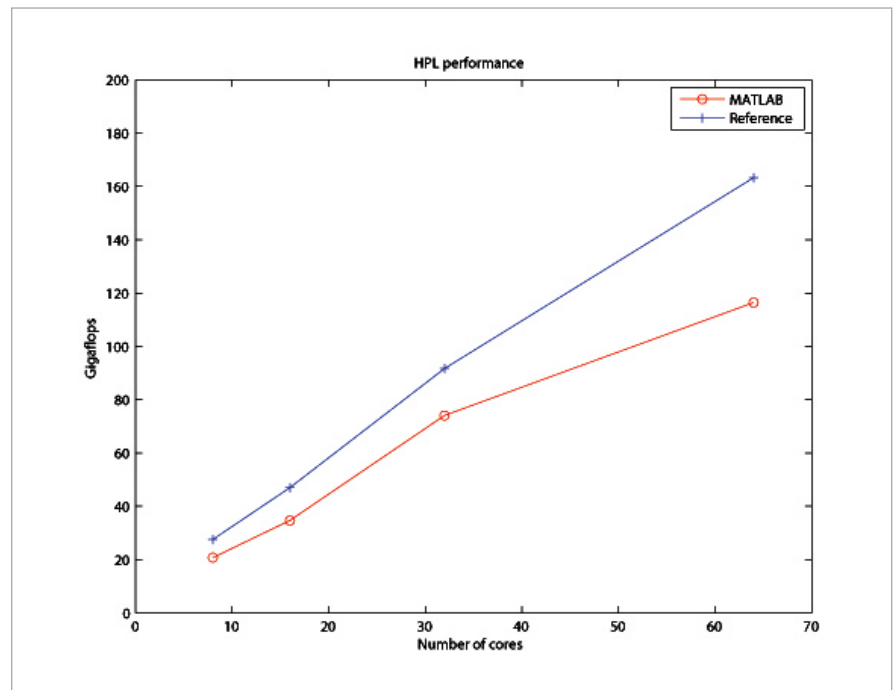


Figure 5. Performance results of MATLAB and HPCC-supplied implementations.

### For More Information

- **Demo: Benchmarking A\b**  
[www.mathworks.com/benchmarking-demo](http://www.mathworks.com/benchmarking-demo)
- **Article: Enhancing Multicore System Performance**  
[www.mathworks.com/multicore-performance](http://www.mathworks.com/multicore-performance)

### Resources

#### VISIT

[www.mathworks.com](http://www.mathworks.com)

#### TECHNICAL SUPPORT

[www.mathworks.com/support](http://www.mathworks.com/support)

#### ONLINE USER COMMUNITY

[www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)

#### DEMOS

[www.mathworks.com/demos](http://www.mathworks.com/demos)

#### TRAINING SERVICES

[www.mathworks.com/training](http://www.mathworks.com/training)

#### THIRD-PARTY PRODUCTS AND SERVICES

[www.mathworks.com/connections](http://www.mathworks.com/connections)

#### Worldwide CONTACTS

[www.mathworks.com/contact](http://www.mathworks.com/contact)

#### E-MAIL

[info@mathworks.com](mailto:info@mathworks.com)

© 2010 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

91819v00 05/10