Parallel Programming in MATLAB

Piotr Luszczek*

July 20, 2009

Abstract

A visit to the neighborhood PC retail store provides ample proof that we are in the multi-core era. The key differentiator among manufacturers today is the number of cores that they pack onto a single chip. The clock frequency of commodity processors has reached its limit, however, and is likely to stay below 4 GHz for years to come. As a result, adding cores is not synonymous with increasing computational power. To take full advantage of the performance enhancements offered by the new multicore hardware, a corresponding shift must take place in the software infrastructure – a shift to parallel computing.

1 Introduction

MATLAB® and Parallel Computing Toolbox™ address the challenge of getting code to work well in a multicore system by enabling you to select the programming paradigm most appropriate to your application. Using a typical numerical computing problem as an example, this article describes how to use the three most basic of these paradigms: threads, parallel for-loops and SPMD (Single Program Multiple Data) blocks.

2 Use Case: a Numerical Problem

To explore MATLAB's parallel capabilities we will use a computationally intensive numerical problem as a case study. Numerical experiments that help in forming a conjecture or developing a matrix algorithm are one of many common uses of MATLAB.

To illustrate different parallel programming paradigms, we will use MATLAB to test a hypothesis regarding Girko's circular law. Girko's law states that the eigenvalues of a random N-by-N matrix whose elements are drawn from a normal distribution lie inside a circle of radius $\sqrt{N}$ [6]. Our hypothesis is that Girko's circular law can be modified to apply not only to eigenvalues but also to singular values. The hypothesis is justified because singular values are eigenvalues of a modified matrix. Applying our modified Girko's law could save computation time by enabling us to estimate the singular values for any normally random matrix without performing singular value decomposition (SVD).

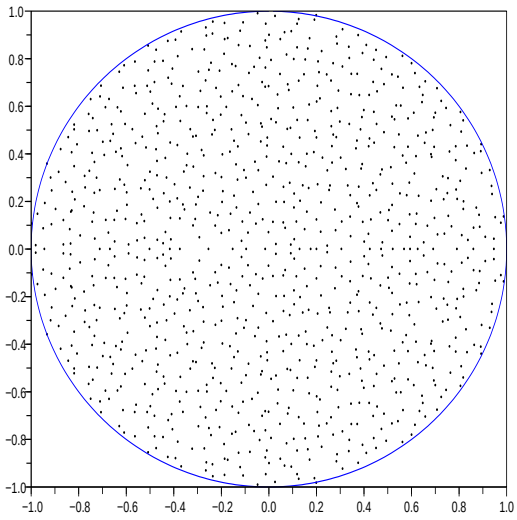

Figure 1: Eigenvalues of a random matrix of size 1000 scaled by $1/\sqrt{1000}$. The elements of the matrix are drawn from the normal distribution.

This theorem can be represented with the following MATLAB code:

```
N = 1000;
plot(eig(randn(N)) / sqrt(N), '.');
```

The code produces the visualization shown in Figure 1.

Each dot represents an eigenvalue on a complex plane. Notice that all the eigenvalues reside inside a circle of radius 1 and are centered at the origin of the axis – a compelling indication that, in accordance with Girko's circular law, the magnitude of an eigenvalue does not exceed the square root of the matrix size.

To apply Girko's law to singular values, we generate random matrices in MATLAB and then examine their singular values to see if we can formulate a hypothesis based on numerical experiments. In particular, we want to

*University of Tennessee Knoxville, 1122 Volunteer Blvd, Suite 203, Knoxville, TN 37996-3450

```
y = zeros(1000,1); % pre-allocate storage for 'y'
for n = 1:1000
   y(n) = max(svd(randn(n)));
end
plot(y);
```

Figure 2: A for-loop that generates normally distributed random matrices and finds their singular values

Threads	Time	Speed-up	Efficiency
1	902.6	1.00	100%
2	867.2	1.04	52%
3	842.3	1.07	35%
4	862.3	1.05	26%

Table 1: Number of threads, time to run the for-loop from Figure 2 and the corresponding speed-up and efficiency.

compute the value of max(svd(randn(N))) for arbitrary values of variable N and then look for a pattern in the results. We can use our theoretical knowledge of SVD to explain the pattern. The code for our numerical experiment is shown in Figure 2.

Running the for-loop on a typical desktop computer with only one core enabled would take more than 15 minutes. To reduce computing time, we will run the loop on a four-core machine using threads and parallel for-loops, and then compare the performance results.

3 Implicit Parallelism: Low-Level Threading

Threads are a common software solution for parallel programming, on multi-core systems, but it is important to bear in mind that multithreading and multi-core processors are not synonymous. The best performance is often obtained when the number of threads and the number of cores correspond, but there are circumstances when there should be fewer threads than cores. We will experiment to determine the optimal number of threads for our computation.

We run the code, adjusting the number of threads using either the Preferences window on the MATLAB Desktop or the MATLAB function maxNumCompThreads().

Table 1 shows results for different thread counts: In addition to time, it also shows parallel speed-up and parallel efficiency. The former is a ratio of execution time on N cores to execution time on one core – ideally we'd like to achieve a speedup of N on N cores. The latter is a ratio of speed-up to the number of cores – ideally, it should be 100%.

Our computational experiment produced relatively poor results. To explain this, we must again examine the most computationally intensive part of our for loop: the call to the svd() function. The matrices passed to svd() range from 1-by-1 to 1000-by-1000. On average the size of the matrices is 500-by-500, Such small matrix sizes do not yield sufficient performance gains on multi-core machines [17]. Clearly, a different parallelization approach is required. The implicit parallelization scheme could be used for larger matrices, say 5000-by-5000, which have sufficiently large computational load to keep all the cores busy.

```
y = zeros(1000,1);
for n = 1:1000
   z = randn(n);
   y(n) = max(svd(z));
end
plot(y);
```

```
y = zeros(1000,1);
parfor n = 1:1000
   z = randn(n);
   y(n) = max(svd(z));
end
plot(y);
```

Figure 3: The for-loop on the left can be made parallel by changing for into parfor as was done on the right.

Using threads did make the computation faster, but bear in mind that in our example, only the call to the svd() function was actually made to run in parallel. This is because threading support in MATLAB is implicit: the user doesn't determine which parts of the code should run in parallel. Some statements are more amenable to implicit parallelization than others – in our case, only the call to the svd() function. None of the other statements benefit from multi-threading because they do not have enough computational load.

On one hand, we can speed up the calculations by using more cores and without changing the original code. On the other, we quickly reach a point of diminishing returns where adding cores does not appreciably reduce execution time.

4 Explicit Parallelism: Parallel for-loops with parfor Keyword

A parfor (parallel for) loop is useful in situations that require many loop iterations of a simple calculation, such as a Monte Carlo simulation. To run parfor we use the Parallel Computing Toolbox. We begin by adapting our original code, as shown in Figure 3.

Just as maxNumCompThreads() controls the parallelism of the multithreading approach, the matlabpool command controls the parallel behavior of the parfor syntax. matlabpool sets up a task-parallel execution environment in which parfor loops can be executed interactively from the MATLAB command prompt.

The iterations of parfor loops are executed on *labs*. A lab is an independent instance of MATLAB that runs in a seperate operating system process. Commonly, labs exe-

Threads	Time	Speed-up	Efficiency
1	870.1	1.00	100%
2	487.0	1.79	89%
3	346.2	2.51	83%
4	273.9	3.17	79%

Table 2: Number of threads, time to run the for-loop from Figure 3 (right) and the corresponding speed-up and efficiency.

cute in headless mode, i.e., they do not have a GUI front-end attached to them – any of their interaction with the rest of the system happens through messages exchanged through the operating system kernel and/or a network interconnect. Like threads, labs are executed on processor cores, and the number of labs does not have to match the number of cores. Unlike threads, labs do not share memory with each other. As a result, they can run on separate computers connected via a network. For our example, however, we only need to know that Parallel Computing Toolbox makes parfor work efficiently on a single multi-core system. Each core, or a local worker, can host one lab.

A question naturally arises: is changing the code worthwhile? The most accurate answer is, "It depends." In our case, changing the code is worthwhile because the results clearly indicate the benefits of using the parfor syntax as indicated by the results in Table 2.

Adding more cores would further reduce computation time, since we have not reached the point of diminishing returns with four cores. The technical term for this behavior is scalability: for our SVD computation, parfor scales better than multithreading. It also provides the kind of performance that might be expected from four cores. We see substantial speedup and acceptable efficiency, which was not the case when we used multithreading.

Without delving too deeply into the implementation details, it is necessary to explain the success that resulted from using parfor. The most notable feature of our sample code is that each iteration of the loop is independent. This feature alone makes the application of parfor so easy yet so effective. The only tasks left for the runtime system inside the parfor are distributing the loop iterations to the cores and gathering results for use outside the parfor loop.

A word of caution about the effect of parfor on random number generation. Matrices generated inside a parfor loop with functions such as randn() will not be identical to their for-loop counterparts because of the way the parfor loop iterations are scheduled. In most cases, this discrepancy is perfectly acceptable.

```
sum = 0;
A = zeros(101, 1);
parfor i = 1:100
   % OK: this is a known reduction operation
   sum = sum + i;

   % error: loop iterations are dependent
   A(i+1) = A(i) + 1;
end
```

Figure 4: A parfor loop with two statements inside that depend on the iteration variable i.

While using parfor has its advantages, it also has limitations. For example, if there is a dependence between the loop iterations and the dependence can be detected through code analysis, then executing the parfor loop will cause an error. If the dependence cannot be detected, then the only indication of the problem will be incorrect results. The problem with dependent loop iterations is illustreted by the code in Figure 4.

The expression that accumulates sum depends on the iteration variable i, but this is not a problem. The parfor runtime can easily work with such expressions by evaluating partial sum on each available lab and then combining the results.

The second expression that operates on array A does, however, pose a problem. The iteration with i=2 cannot compute the value of A(3) until the value A(2) is computed in iteration 1. By the same token, the iteration with i=3 depends on the iteration with i=2, and so on.

Let's try to fix this problem by taking a closer look at what happens at each iteration:

Iteration 1 i = 1: A(2) = A(1) + 1 = 0 + 1 = 1

Iteration 2 i = 2: A(3) = A(2) + 1 = 1 + 1 = 2

Iteration 3 i = 3: A(4) = A(3) + 1 = 2 + 1 = 3

It soon becomes clear that we can achieve the same effect as the loop in Figure 4 by rewriting it as follows:

```
parfor i = 1:100
sum = sum + i;
A(i+1) = i;
end
```

Now the parfor loop executes and yields the result that was initially intended.

5 Explicit Parallelism: spmd Keyword

In the previous section we started to make changes to the original loop from Figure 2 in order to introduce

```
spmd
  y = zeros(1000,1);
        for        n        =
labindex:numlabs:1000
    z = randn(n);
    y(n) = max(svd(z));
  end
end
```

Figure 5: Parallel loop for computing singular varlues in parallel using the spmd keyword.

labindex	numlabs	labindex:numlabs:1000
1	4	[1, 5, 9, 13, ...]
2	4	[2, 6, 10, 14, ...]
3	4	[3, 7, 11, 15, ...]
4	4	[4, 8, 12, 16, ...]

Table 3: Distribution of loop iteration that guarantees load balance on four cores.

parallelism to the computation. We need to make further changes to show yet another parallelization method available in the Parallel Computing Toolbox. The method in question uses an spmd keyword. The modified loop is shown in Figure 5. The name of the keyword originates from the SPMD acronym often used in parallel programming literature. It stands for Single Program Multiple Data. It indicates that there is a single code for execution by each parallel core but each core has its own data to operate on. Unlike parfor loops, spmd blocks (the code between spmd and its corresponding end) require much larger mental leap from sequential loops. The reason is that any code executed inside spmd can behave differently on each core. To be more specific, the loop in Figure 5 there is a call to two functions not discussed before: labindindex and numlabs. The former returns the core number: on the first core it will return 1, on the second – 2 and so on. The latter function returns the number of cores that execute the code inside an spmd block. Using these functions it is possible to customize the for-loop to spread the

labindex	numlabs	(labindex-1)*1000/numlabs :labindex*1000/numlabs
1	4	[1, 2, 3, 4, ...]
2	4	[251, 252, 253, 254, ...]
3	4	[501, 502, 503, 504, ...]
4	4	[751, 752, 753, 754, ...]

Table 4: Distribution of loop iteration that results in load imbalance on four cores.

```
spmd
  y = zeros(1000,1, codistributor);
  for n = 1:1000
   z = randn(n, codistributor);
   y(n) = max(svd(z));
  end
end
```

Figure 6: Using distributed arrays for computing singular varlues in parallel inside the spmd keyword.

iteration space across all the available cores. Table 3 shows which loop iterations are executed on each core. It is important to note that the choice of loop iterations for each core was made with load balance in mind. An alterntive, is to use the following iteration vector: (labindex-1)*1000/numlabs:labindex*1000/numlabs. Table 4 shows the mapping of iterations to the computational cores. Core number 4 gets the largest matrices and which represents the most of the work and therefore it will finish last. This will completely defeat the effort of parallelization. No such considerations were necessary when using the parfor-loops: load-balancing is done automatically behind the scenes by the runtime scheduler inside the Parallel Computing Toolbox.

6 Explicit Parallelism: Distributed Arrays

It is possible to further modify the original code from Figure 2 to show yet another method of parallelization that is available from the Parallel Computing Toolbox. The method in question are distributed arrays and the modified code is shown in Figure 6. Distributed arrays are a software abstraction that attempts to hide the complexity of parallel programming behind MATLAB's object-oriented interface [13] and polymorphic functions [15]. A class called codistributor overloads MATLAB's matrix creation functions such as zeros() and returns a distributed array object that, for the most part, has the same functionality like the standard matrix in sequential MATLAB. In particular methods such as svd() and max() are overloaded and return results that are themselves distributed arrays. It is instructive to note that this is similar to the parallelism based on threads. The parallelism is hidden inside computational methods and functions that are called from otherwise seqential code. As was the case with threading, it releaves the programmer from thinking about parallelism but it can easily suffer with poor use of resources as it was shown in section 3. The advantage of using distributed arrays over threading is the fact that they can be scaled beyond a single multi-core computer. It means both more computa-

```
spmd
  % Part 1: computing SVDs
  y = zeros(1000,1);
  for n = labindex:numlabs:1000
   z = randn(n);
   y(n) = max(svd(z));
  end

  % Part 2: synchronizing 'y'
  if labindex == 1
   labSend(y, 2)
   y = labReceive(numlabs);
  else
   yReceived = labReceive(labindex-1);
   for n = 1:1000
    y(n) = max(y(n), yReceived(n));
   end
   labSend(y, labindex+1)
  end
end
```

Figure 7: Parallel loop for computing singular varlues in parallel using the spmd keyword.

tional power and larger matrices to work with.

7 Explicit Parallelism: Message Passing

Message passing is the most extreme parallel programming paradigm out of the ones presented so far in a sense that gives the most flexibility but at the same some requires the most effort to get it right. Message passing paradigm uses the spmd keyword and additional functions available in the Parallel Computing Toolbox to allow writing scalable algorithms that take advantage of both multi-core chips as well as clusters of multi-core computers. The additional functions synchronize and exchange data between the labs using an MPI (Message Passing Interface) library [7, 8, 9]. MPI interacts directly with either the memory controllers or the interconnect hardware to provide low-latency and high bandwidth interface that is portable accross a variety of network hardware.

To give a sample of a message passing paradigm we first note that the code from Figure 5 has a feature that probably is not apparent at the first look: the array y is not the same on all labs. Each lab has its own copy of the array and does not see the values that are produced on other labs. The code in Figure 7 synchronizes array y across labs by sending a local copy of y from lab 1 to lab 2, then to lab 3, and so on. The last lab sends y back to lab 1. Prior to forwarding its local y, each lab computes the maximum of each element of the received array yReceived and its local copy. In this way, the initial zeros in y are replaced with singular values which, by linear algebra theory, are greater than zero. The MATLAB implementation in Figure 7 uses functions labSend() and labReceive() to exchange messages between the labs. The messages contain partial results from each lab.

There are many details left out of this exposition because the message passing paradigm requires a great attention to detail is mostly intentended for seasoned parallel programmers. On a positive note it is worth noting that some parallel idioms have been captured in functions available in the Parallel Computing Toolbox. In particular, synchronization of array y that is done in the second part of Figure 7 may be replaced with single line:

```
y = gop(@max, y);
```

that uses global high-order function [1] gop() that takes another MATLAB function as an argument and applies it globally across all the labs. There is an added bonus of using gop(): not just one but all labs receive updated array y and the synchronization is scalable in a sense that the number of message passing phases is not proportionate to the number of labs but to the base two logirthm of the number of labs.

8 Parallel Language Evolution

Parallel Computing Toolbox has been in the making for many years now. Consequently, its API (Application Programming Interface) and syntax have evolved. Parallel for-loops' syntax changed only slightly – the old syntax was parfor(i = 1:100). Changes to the API distributed arrays were more pronounced. Initially, in 2005, distributed arrays were constructed using distributed constructor but with slightly different syntax and limited functionality. As more functionality was added the name of the contructor changed to darray in 2006. After a year the name of the constructor returned to distributed but it was changed again in 2008 to codistributed. Despite the evolution of the API, it is quite easy to accomodate the changes in MATLAB scripts because the functionality remained stable and much of it came from overloaded functions and methods which bear the same name as their sequential counterparts.

9 Alternative Solutions

Parallel functionality offered by the Parallel Computing Toolbox is supported by the makers of MATLAB: the MathWorks. This is in contrast with their earlier position [16] dating back over a decade ago which might explain a large number of parallel MATLAB extensions [4] that were created over the years.

One of the still actively developed parallel exten-

sions is pMatlab [12] which offers the abstraction of distributed arrays very similar to the one described in section 6. A very important feature of pMatlab is its portability because it only relies on MatlabMPI [11] for message passing capabilities – a low-level implementation layer. Portability in MatlabMPI is achieved via exclusive use of MATLAB code rather than MEX extensions in C/C++ or FORTRAN. The messages between labs are exchanged via a distributed file system. Portability comes at a price of performance but the latter may be increased by use of bcMPI[1] (Blue Collar MPI) which provides a native binding to MPI.

Aspiring for commercial viability is the StarP project that was intially called Matlab*P [5, 3, 2, 10]. The project now has commercial support offered by Interactive Supercomputing and can be used not just from MATLAB but also from Python. The available functionality includes the abstraction of distributed arrays both dense and sparse and extensive plotting capabilities for these arrays.

10 Extending Parallel Computing

Multi-core processors are here to stay, and so is parallel programming. MATLAB already supports several parallelization methods. Support for additional methods will be provided in future versions of the product.

Consumers and engineers alike believe that we'll see more cores inside future computers. The trend so far has been to double the number of cores every few years. This translates into a doubling of computational power. Harnessing that power will require the right software, and writing that software will require the right software tools. MATLAB is well positioned to fulfill that requirement [14].

References

[1] John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., 1978.

[2] Long Yin Choy. MATLAB*P 2.0: Interactive supercomputing made practical. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 2002.

[3] Long Yin Choy and Alan Edelman. MATLAB*P 2.0: A unified parallel MATLAB. Technical report, Massachusetts Institute of Technology, January 2003. URI: http://libraries.mit.edu/dspace-mit/.

[4] Ron Choy. Parallel matlab survey, 2004. Available at http://supertech.lcs.mit.edu/~cly/survey.html.

[5] Ron Choy and Alan Edelman. Parallel matlab: Doing it right. In *Proceedings of the IEEE*, volume 93:2, pages 331–341, February 2005.

[6] Alan Edelman. *Eigenvalues and Condition Numbers of Random Matrices*. PhD thesis, Massachusetts Institute of Technology, May 1989.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: http://www.mpi-forum.org/.

[9] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 18 July 1997. Available at http://www.mpi-forum.org/docs/mpi-20.ps.

[10] Parry Jones Reginald Husbands. *Interactive Supercomputing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1999.

[11] Jeremy Kepner and Stan Ahalt. MatlabMPI. *Journal of Parallel and Distributed Computing*, 64(8):997–1005, Aug 2004.

[12] Jeremy Kepner and Nadya Travinin. Parallel Matlab: The Next Generation. In *Seventh Annual High Performance Embedded Computing Workshop (HPEC 2003)*, MIT Lincoln Laboratory, Lexingtom, MA, September 2003. http://www.ll.mit.edu/HPEC/agenda03.htm.

[13] Barbara Liskov and Stephen Zilles. Programming with data types. In *ACM SIGPLAN Conference on Very High Level Languages*, Santa Monica, California, United States, 1974. ACM.

[14] Roy Lurie. Language design for an uncertain hardware future. *HPCwire*, September 28, 2007. See www.hpcwire.com/features/17902899.html.

[15] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[16] Cleve Moler. Why there isn't parallel Matlab. *Mathworks Newsletter*, 2, 1995. Cleve's corner.

[17] Cleve Moler. Parallel MATLAB: Multiple processors and multiple cores. *The MathWorks News & Notes*, June 2007. See www.mathworks.com/clc_multiproc.

[1] For more details see: http://www.osc.edu/bluecollarcomputing/applications/bcMPI/index.shtml