Cheap and Easy Parallelism for Matlab on Linux Clusters

Simon D. Levy, Peter L. Djalaliev, Jitendra M. Shrestha, and Alexandr Khasymski
Computer Science Department

Christopher D. Connors
Geology Department

Washington & Lee University
Lexington, VA 24450

Correspondence: levys@wlu.edu

KEYWORDS: *Matlab, Parallel, Distributed, Linux, Java, XML, Genetic Algorithms, Mandelbrot Set*

Abstract

Matlab is the most popular platform for rapid prototyping and development of scientific and engineering applications. A typical university computing lab will have Matlab installed on a set of networked Linux workstations. With the growing availability of distributed computing networks, many third-party software libraries have been developed to support parallel execution of Matlab programs in such a setting. These libraries typically run on top of a message-passing library, which can lead to a variety of complications and difficulties. One alternative, a distributed-computing toolkit from the makers of Matlab, is prohibitively expensive for many users. As a third alternative, we present PECON, a very small, easy-to-use Matlab class library that simplifies the task of parallelizing existing Matlab programs. PECON exploits Matlab's built-in Java Virtual Machine to pass data structures between a central client and several "compute servers" using sockets, thereby avoiding reliance on lower-level message-passing software or disk i/o. PECON is free, open-source software than runs "out of the box" without any additional installation or modification of system parameters. This arrangement makes it trivial to parallelize and run existing applications in which time is mainly spent on computing results from small amounts of data. We show how using PECON for one such application – a genetic algorithm for evolving cellular automata – leads to linear reduction in execution time. Finally, we show an application – computing the Mandelbrot set – in which element-wise matrix computations can be performed in parallel, resulting in dramatic speedup.

1 Introduction

Programming language support for science and engineering is evolving away from general-purpose, hard-to-master languages like C/C++ and Java, and toward mathematically-oriented languages and environments like Maple, Mathematica, and Matlab. Such programming environments provide a level of abstraction that allows scientists and engineers to focus on the details of their problem, instead of low-level programming language issues. The use of interpreters (rather than compilers) in such environments makes them ideal for rapid, interactive prototyping of algorithms and data structures, and makes them especially well-suited for research and teaching. Although there will always be a need for fast-running compiled languages like C/C++ in high-*performance* computing applications, the high *productivity* offered by Maple, Mathematica, Matlab, and the like, makes their increasing presence a near certainty.

Of these three platforms, Matlab appears to dominate the market for rapid prototyping and development of scientific and engineering applications. An informal test of Google hits for *Maple*, *Mathematica*, and *Matlab* combined with terms like *programming scientific*, and *engineering* fails to reveal consistently significant differences among the popularity of the three platforms. Nevertheless, the authors' subjective experience at research and teaching institutions shows Matlab to be the language of choice for computing in biology, geology, psychology, engineering, and other "number crunching" / data-driven disciplines, with Maple and Mathematica being more popular for symbolic math applications.

Another recent computing trend – especially in academia – is the growing presence of networks of inexpensive workstation PC's running the free, open-source Linux

operating system. Together, these two trends make it common to see a university computing lab with Matlab installed on a set of networked Linux workstations.

2 Related Work

In addition to relatively lower cost and higher security as compared with Windows, Linux supports remote launching of processes, through the `ssh` (Secure Shell) protocol. Because of this situation, and the growing availability of distributed computing networks in general, a number of software libraries have been developed to support parallel execution of Matlab programs. These libraries fall into two basic categories: (1) commercial packages, the only one of which (to the authors' knowledge) is the Matlab Distributed Computing Toolbox/ Engine, available from Mathworks, and (2) free third-party packages, built on top of a free message-passing library like MPI [9]. With costs running into the tens of thousands of dollars for even a modest-sized (16-node) network, the former type of solution is likely to be prohibitively expensive for universities and other non-commercial settings.

As for the second option, using a Matlab library that implements a message-passing interface like MPI may require the user to install a platform-dependent implementation of the interface "underneath" the Matlab library, which the Matlab library will then access through a low-level mechanism like Mex files (C programs that can be called directly from Matlab). Even if the implementation is already installed on the user's system (as is common nowadays with MPI), the user may still be responsible for compiling Mex files or otherwise performing low-level tasks for which s/he may be unprepared – exactly the kind of work that Matlab is supposed to help users avoid.

MatlabMPI, a popular message-passing implementation [3] avoids this problem by using file i/o to simulate message-passing. Given the usual necessity of running Matlab programs on a shared file system (in order to access a common set of user-defined scripts or functions), this solution offers a straightforward way of avoiding the aforementioned problems associated with a low-level message-passing implementation. On Linux, however, this solution requires changing low-level O/S parameters like the duration for which cached disk attributes are held after disk update. If the user of the system is sharing a Linux cluster with persons who are using the cluster for other purposes (as is typically the case), system administrators may be understandably reluctant to make such changes.

A second issue with message-passing libraries is the level of abstraction that they provide a user. Successful use of MPI, for example, requires programmers to attend to many low-level details that may have nothing to do with the problem that they are trying to solve – precisely the sort of issues that high-level languages like Matlab seek to hide from programmers. The pMatlab library [4] library successfully hides these details, implementing parallelism through a few simple commands. This library is however built on the disk-based message-passing solution MatlabMPI [3], making users deal with the disk issues just described.

3 PECON

To fulfill this need in our own work, we developed PECON, a Parallel Evaluation CONtroller for Matlab. PECON is a Matlab class library containing only a constructor, a function-evaluation method, and a destructor. The main insight that led to the development of PECON is that Matlab's built-in Java Virtual Machine (available in versions 6.0 and later) can be used to avoid "outsourcing" the interprocess (client/server) communication layer, as many other libraries do. Instead, PECON constructs Java objects representing clients and servers, which in turn use the `Socket` and `ServerSocket` classes from the `java.net` package. Matlab's Java interface does not support passing user-defined structures, but does support passing Matlab strings. Therefore, PECON uses a simple, publicly available Matlab toolbox that converts Matlab data structures to and from strings of XML.[1] These strings get passed to the servers as function arguments, and back to the client as results, via Java socket calls internal to the PECON code. The following sections describe each of the three PECON methods.

3.1 Constructor

The constructor (`pecon`) method accepts a list of names of hosts on a Linux cluster and launches a Matlab process (server) remotely on a each specified host using a backgrounded call to `ssh`. The constructor then creates a set of Java objects, each of which will be used to communicate with a corresponding server via calls to methods in the `java.net.Socket` class. These Java objects are stored in a Matlab object, which is returned to the user.[2]

Meanwhile, each remote server process launches Matlab and runs a special pre-written Matlab function. This function creates a Java object for handling socket calls from the client, and then waits, via infinite loop, for function-evaluation requests from the client. The servers do not

[1] `http://www.geodise.org/toolboxes/generic/xml_toolbox.htm` I am grateful to Ken Lambert for suggesting XML.

[2] An anonymous reviewer of an earlier draft of this paper suggested using `ssh` for the data exchange as well as the initial server launch, making the Java calls unnecessary. Although this alternative would increase the security of the data exchange, we see no way to implement it without having to launch a new Matlab session for every exchange of data. The additional overhead from this would probably undo any speedup gained from parallelizing.

communicate with each other and are not directly accessible to the user. Their only job is to execute Matlab function calls remotely, and return the results to the client.

3.2 Function evaluator

The function-evaluation (`feval`) method works similarly to the standard Matlab `feval` function, except that its first argument is the PECON client object returned by the constructor. Its second argument is the name or handle of a Matlab function. Its remaining arguments are one or more lists of arguments to the named function. If the lists are not of equal length n, an exception is thrown. Otherwise, the Java socket objects contained in the PECON object are used to send the server the function's name, expected number of inputs, and expected number of outputs. In this way, the server knows how many function arguments to expect from the client, and how many output results to send back. (Matlab supports multiple return values for a single function.) Each of these is sent as a string via methods in the `java.io.ObjectOutputStream` class.

The client then loops n times; on each iteration, it sends the ith member of each function-argument list to the ith server. When i exceeds the number of servers, the servers are reused, starting with the first. (This scheme implements a primitive form of load-balancing.) For each function argument, the client determines whether the argument is a two-dimensional floating-point array (the fundamental data type in Matlab). If it is, the client creates a Java object containing a pointer to the array, and sends this object to the server via methods in the `java.io.ObjectOutputStream` class. For any other kind of function argument, the client uses the XML toolbox to convert the argument into a string containing an XML representation of the object, and sends this string to the server via `ObjectOutputStream`. The client waits for the server to return the results of calling the named function on the arguments passed to it. Each server receives the function name, input/output argument counts, and single set of function arguments. The server calls the function on the arguments, and sends the results back to the client.

The special case for floating-point arrays was added after profiling revealed that most of the clients' cycles were being spent in converting to/from XML, when large arrays were being passed. (See the Mandelbrot Set example below.) Although a full description or DTD for the format of the XML strings is beyond the scope of this paper, the following simple example shows how the XML toolbox works:

```
>> a = struct('pi', 3.14159,  'primes', [2 3 5 7 11 13]);
>> s = xml_format(a)
s =
<root xml_tb_version="2.0" idx="1" type="struct" size="1 1">
  <pi idx="1" type="double" size="1 1">3.14159</pi>
  <primes idx="1" type="double" size="1 6">2 3 5 7 11 13</primes>
</root>
>> xml_parse(s)
ans =
        pi: 3.1416
    primes: [2 3 5 7 11 13]
```

Using `ZipInput/OutputStream` instead of `ObjectInput/OutputStream` for socket read/write caused `feval` to take more, rather than less, total time, suggesting that the time taken to de/compress the XML strings may be longer than the time taken to send/receive the uncompressed strings. Therefore, PECON currently does not use compression.

3.3 Destructor

Calling the destructor (`halt`) method on the PECON client object causes it to send an empty message to each server. Upon detecting such a message, the server function breaks out of its infinite loop and exits. Because Matlab was invoked with the "run this function only" option on the server, the Matlab process on the server exits as well, causing the original backgrounded `ssh` process on the client to exit too.

3.4 Example

All this complexity is hidden from the PECON user, who writes programs on the client machine that look very much like ordinary, serial Matlab programs. Here is a simple example, showing a function (square root) of one argument:

```
>> p=pecon({'node10','node11','node12','node13'})
Server node10: ready
Server node11: ready
Server node12: ready
Server node13: ready
>> s = feval(p, 'sqrt', {1 4 9 16 25 36})
s =

    [ 1]    % returned from node10
    [ 2]    % returned from node11
    [ 3]    % returned from node12
    [ 4]    % returned from node13
    [ 5]    % returned from node10
    [ 6]    % returned from node11

>> halt(p)
```

To use PECON, a Matlab programmer simply needs to download the Matlab source code and JAR file from `http://www.cs.wlu.edu/~levy/pecon`. (The Java source is also available for the curious.) Because of the minimalist approach taken in the project, the Matlab source is only 245 lines of code, and the `JAR` file is just over 20 Kbytes. A three-line `startup.m` file is used to provide access to the PECON source code, the JAR file, and the XML toolbox from the user's working directory.

4 Applications

To demonstrate how PECON can be used to parallelize existing Matlab programs with a minimum of effort,

we present our results on two different "embarrassingly parallel" problems. Although these problems differ greatly from each other in the amount of data being passed back and forth between the client and servers, they both show the dramatic speedup afforded by using PECON.

4.1 Genetic Algorithms

One motivation to write PECON came from our work in genetic algorithms. Genetic algorithms (GAs) [6] are a search/optimization technique inspired by biological evolution. Given a problem to solve or optimize, an initial population of candidate solutions is generated, more or less at random. A *fitness* value is then assigned to each member of the population, based on its relative success at solving the problem. High-fitness solutions are allowed to survive and reproduce, either by direct copying or by a genetic crossover process simulating sexual recombination. The resultant offspring can then be mutated, usually by flipping a bit or adding Gaussian noise to the solution. This evaluate-reproduce-mutate cycle can be repeated for some pre-determined number of generations, or until an adequate solution emerges in the population.

Although many variants of this Simple Genetic Algorithm exist, they share some common properties: an individual population member is typically a small data structure (like a bit string or matrix), and most of the compute time is spent in evaluating the fitnesses of these individuals. Because one individual's fitness does not usually depend on the fitness of another, fitnesses can usually be evaluated in an arbitrary order, or in principle, simultaneously. And because most GA operations can be easily coded in a line or two of Matlab, Matlab is a natural choice for implementing and teaching genetic algorithms. Our goal was therefore to develop a way of parallelizing existing and future Matlab GA code with a minimum of additional work for the GA programmer. PECON emerged as a straightforward solution to this problem.

As a test case for parallelizing a GA, we chose a classic example from the GA literature, the problem of evolving a cellular automaton. A cellular automaton (CA; [11]) is a simple model of computation consisting of an array of N cells, each of which can take on a small number of discrete values, and an update rule, which changes the value of a cell based on its current value and the values of its neighbors. The number of relevant neighbors determines the size of the rule table; for a binary-state CA with r neighbors on either side of each cell, the table contains 2^{2r+1} bits. The update process is iterated for some predetermined number of steps, or until the values of the cells no longer change.

An interesting problem for a CA is to see whether the update rule can correctly determine, after some finite number of iterations, whether the initial cell configuration contained a majority of zeros or ones. This *density-classification* task is judged by seeing whether the CA ends up in the right configuration – all zeros for an initial majority of zeros, or all ones for an initial majority of ones. Because the exponential size of the rule table makes exhaustive rule-table search prohibitive, a number of researchers have investigated this question using a genetic algorithm [7, 8].

4.1.1 Experiment

Following [7], we wrote a GA program to evolve a CA rule table for the density-classification task, with $r = 3$ (128 bits per rule table), $N = 149$, approximately 320 CA iterations, and 300 initial configurations for evaluating each rule table. As in [7], each rule table was treated as an individual whose fitness on an initial configuration was computed as the fraction of correct bits in the final configuration – the fraction of ones (zeros) for an initial configuration with mostly ones (zeros). The overall fitness of an individual was the average of these fractions over the 300 initial configurations. Our goal was not to replicate the results in [7], but rather to determine the extent to which the execution time of the algorithm could be reduced by evaluating fitnesses in parallel. Abbreviated Matlab code for our main GA algorithm is shown below. As this code shows, the algorithm was easily parallelized by adding a single input argument (`pobj`, a PECON object) and a call to the PECON `feval` method.

```
function pop = caga(ngen, pobj)

pop = num2cell(unif(100, 128), 2);

for i = 1:ngen

  if nargin > 1
    fits = feval(pobj, 'fitness', pop);
  else
    for j=1:length(pop)
       fits{j} = fitness(pop{j});
    end
  end

  % crossover, mutate to get next pop

end
```

4.1.2 Results

Figure 1 shows execution times for one generation of our cellular automaton GA. As the figure illustrates, we achieved nearly perfect scaling using PECON, the actual execution times being very close to their theoretical limit (100% of time spent computing fitness, with no communication cost). This result can be attributed to the "embarrassingly parallel" nature of GA fitness evaluation, combined with the trivial size of the data being sent between the client and servers (128 bits for each individual, one floating-point number for its fitness).

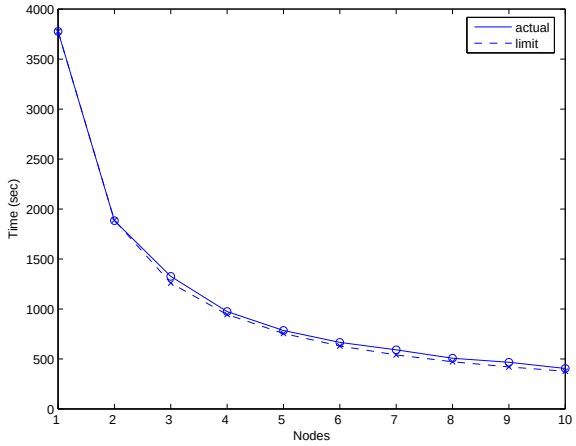

Figure 1: Total execution time in seconds versus number of cluster nodes, for one generation of a cellular automaton GA. Times are averaged over five trials. Dashed line is theoretical lower limit based on serial execution time.

4.2 Mandelbrot Set

The Mandelbrot Set [5] is obtained by iterating the equation $z_{n+1} = z_n^2 + C$ on each point C in a region of the complex plane, with $z_0 = 0$. The set consists of those points for which z_n does not tend toward infinity, and can be approximated by sampling the region and determining the points for which $abs(z_n)$ does not cross some fixed threshold (*e.g.*, 2) after some fixed number of iterations (*e.g.*, 100). By coloring or shading the array of sampled points (pixels) according to the number of iterations at which the threshold is crossed, beautiful fractal (self-similar) images can be obtained.

4.2.1 Experiment

The motivation for this experiment came from a student project in a Matlab-based scientific computing textbook [2]. Because Matlab supports complex numbers as primitive data types, and can color and draw an entire array using a single command, it is an ideal environment in which to program and explore the Mandelbrot Set. Because generating the iteration count for each point is an inherently iterative process, the textbook uses it as an example project for learning about loops. On the other hand, the the time required to compute the iteration counts for all the pixels of of the array can make the exercise painfully slow, for a decent pixel resolution.

For this reason, we thought would be worthwhile to see whether PECON could be used to speed up the Mandelbrot Set computation. As with the GA example above, the code fragment below shows that this was accomplished by adding a few lines of code to the existing solution:

```
function drawMandelbrot(realrange, imagrange, pobj)

NPIX = 400;
NCHUNKS = 4;

x = repmat(rangevals(realrange, NPIX), NPIX, 1);
y = repmat(rangevals(imagrange, NPIX)', 1, NPIX);

if nargin < 3 % serial

  s = mandelbrotChunk(x, y);

else

  xx = partition(x, NCHUNKS);
  yy = partition(y, NCHUNKS);
  ss = feval(pobj, @mandelbrotChunk, xx, yy);
  s = assemble(ss, NCHUNKS);

end
```

The `partition` and `assemble` functions are utilities external to the PECON class library but included with the forthcoming PECON release. The `partition` function divides an array into an NxN grid of sub-arrays, returning a length-N^2 list suitable for passing as arguments to PECON's `feval` method. The `assemble` function performs the inverse operation, taking such a list (as returned from `feval`) and assembling it into a single array. This partitioning is illustrated for the Mandelbrot set in figure 2.

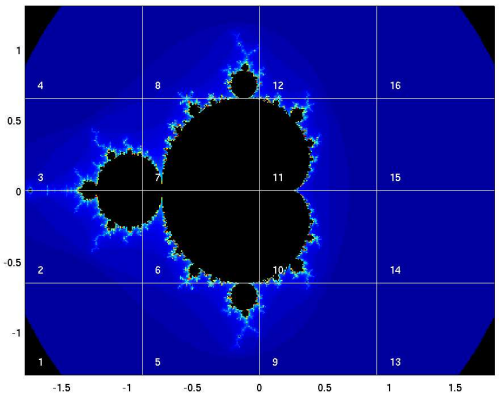

Figure 2: Partitioning of the complex plane region used in computing the Mandelbrot Set in parallel. Indices correspond to servers used to compute each sub-region.

As an informal test of parallel Mandelbrot Set program, we collected execution times for five successive zooming-in operations on on the set, using a modified version of the `mandelgui` program in [2]. Each such zoom operation required computing the iteration counts for the same size array of points (though not precisely the same number of iterations per point).

4.2.2 Results

Figure 3 graphs the time in seconds taken by each successive zoom operation for the two versions (serial and

parallel) of the Mandelbrot program, running on 14 cluster nodes. Using PECON has reduced this time from the painfully show average of 9.5 seconds to a much more acceptable 1.7. Linear speedup for this problem would yield approximately 0.8 seconds; the discrepancy is likely caused by the overhead of passing 10,000 double-precision numbers (80,000 bytes) back and forth between the client and each of the servers.

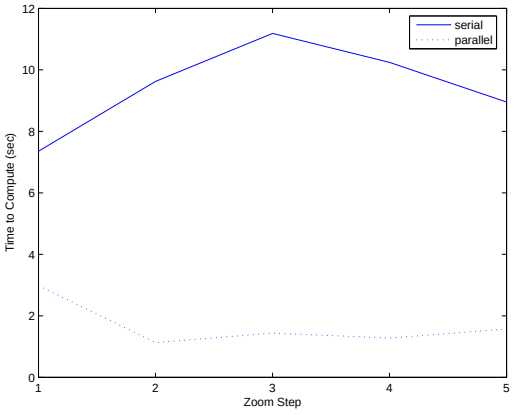

Figure 3: Zooming-in times for serial Mandelbrot Set program compared to corresponding times for parallelized version.

5 Conclusions and Future Work

We have presented PECON, a very small, easy-to-use class library for parallel execution of existing Matlab code on Linux clusters. By not relying on lower-level libraries or file i/o to support passing of messages, PECON avoids many of the problems associated with other attempts to parallelize Matlab. For two kinds of computationally intensive programs – genetic algorithms and the Mandelbrot set – `PECON` achieved significant speedup, approaching the theoretical limit for the former.

In our current work with `PECON`, we are parallelizing an existing genetic algorithm for evolving solutions to problems in geological fault-bend folding [10], the process by which layers of the earth's crust push against and fold over each other through time. A typical problem in this research is to determine the shape of an underground fault by observing the above-ground layers. Existing Matlab code for this problem [1] represents a fault (solution) as a small matrix of (X, Y, Z) coordinates. The final layer configuration is computed by running the folding model over several time steps, after which this configuration can be evaluated by comparison with observed data. Hence, the fitness function for a given fault fits our criteria for parallelization (slow algorithm run on small data structures), and our preliminary results are qualitatively similar to those presented in Figures 1 and 3.

Finally, although PECON itself is written in object-oriented Matlab, it does not support evaluation of methods on objects. Adding this capability to PECON is another possible task for the future.

References

[1] C. Connors and J. Upchurch. A forward modeling program for fault-bend folding. Geological Society of America Abstracts with Programs, v. 34, 2002.

[2] D. T. Kaplan. *Introduction to Scientific Computing and Programming*. Brooks/Cole, Belmont, California, 2004.

[3] J. Kepner and S. Ahalt. MatlabMPI. *Journal of Parallel and Distributed Computing*, accepted.

[4] J. Kepner and N. Travinin. Parallel matlab: The next generation. In *Seventh Annual Workshop on High-Performance Embedded Computing*, Lexington, Mass., September 23-25 2003. MIT Lincoln Labs.

[5] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Company, 1988.

[6] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[7] M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Dynamics, computation, and the "edge of chaos": A re-examination. In G. Cowan, D. Pines, and D. Melzner, editors, *Integrative Themes*, volume 19 of *Santa Fe Institute Studies in the Sciences of Complexity*. Addison-Wesley, Reading, Mass., 1993.

[8] N. Packard. Adaptation toward the edge of chaos. In J. Kelso, A. Mandell, and M. Schlesinger, editors, *Dynamic Patterns in Complex Systems*. World Scientific, 1988.

[9] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, Mass., 1998.

[10] J. Suppe. Geometry and kinematics of fault-bend folding. *American Journal of Science*, 283:684–721, 1983.

[11] S. Wolfram. *Theory and Applications of Cellular Automata*. World Scientific, 1986.