Comput Econ (2016) 48:535–546
DOI 10.1007/s10614-015-9516-5

A Note on Julia and MPI, with Code Examples

Michael Creel[1]

Accepted: 10 August 2015 / Published online: 22 August 2015

Abstract This note explains how MPI may be used with the Julia programming language. An example of a simple Monte Carlo study is presented, with code. The code is intended to serve as a general purpose template for more relevant applications. A second example shows how the template code may be adapted to perform a Monte Carlo study of the properties of an approximate Bayesian computing estimator of actual research interest. All of the code is available at https://github.com/mcreel/JuliaMPIMonteCarlo.

Keywords Julia programming language · Message passing interface · Monte Carlo · Approximate Bayesian computing

Julia (http://julialang.org) is a fairly new programming language that is oriented toward scientific computing. Development began in 2009, and the current release version as of May 2015 is 0.3.8. While this low release number indicates that the language is still evolving quickly, nevertheless, it is already stable enough to warrant development of code for research. Julia also has a reasonably complete set of features of interest to economists, such as methods for optimization, probability distributions, data manipulation, plotting, and so forth[1]. Julia is a high level language with syntax that is as readable as that of Matlab or Python, but through use of just in time compilation, the developers aim to provide performance similar to that of C, a compiled language. Julia is a free and open source language, and it runs on all popular operating systems. The

[1] See http://pkg.julialang.org/ for a list—there is even a preliminary Dynare package.

✉ Michael Creel
michael.creel@uab.es

[1] Universitat Autònoma de Barcelona, Barcelona Graduate School of Economics, and MOVE, Barcelona, Spain

combination of all of these features makes the language a very appropriate choice for research and teaching in economics.

This note shows how Julia may use the message passing interface (MPI) for parallel computing on multicore computers or clusters of computers. Because MPI is such a widely used method of achieving high performance computing for challenging problems, it is very useful to be able to use this method with Julia, as existing code for other languages can easily be adapted. A major advantage of using MPI is code portability: the same code will run on a notebook, a powerful desktop, a cluster, or a supercomputer. While Julia does provide other mechanisms for parallel computing such as parallel for loops and distributed arrays, these methods are not well suited to all problems or all hardware configurations, so having access to the MPI framework is an important extension. Fortunately, the MPI package for Julia makes access to MPI a simple matter. This note covers installation and use of the MPI package, and gives some basic examples, including a very basic Monte Carlo study. The note then goes on to show how the same framework may be easily adapted to study the properties of an econometric estimator of actual research interest. This is an approximate Bayesian computing (ABC) estimator that uses adaptive importance sampling and local linear nonparametric regression, applied to the estimation of a structural auction model. This estimator is computationally intensive, so the ability to speed up computations is important when studying its properties. It is shown that it is possible to accelerate computations by a factor of $9\times$, using a single multicore computer. Because of the portability offered by MPI, greater speedups could easily be obtained by running exactly the same code on a cluster or supercomputer.

The paper is organized as follows: The first section gives a description of some of the resources available to economists who are interested in the Julia language. The second introduces the MPI package for Julia, and the third second discusses using MPI for conducting Monte Carlo exercises. A final section offers brief conclusions.

1 Julia Resources for Economists

While use of Julia in economics is still quite limited, some early adopters have made materials available that illustrate how Julia may be used in economics, and their enthusiasm for the language may provide a stimulus to those who are doubting whether or not to experiment with the language:

- Sargent and Stachurski's Quantitative Economics (Sargent and Stachurski 2014) is a web page that provides a number of lectures on quantitative economic modeling, with code examples in Python and Julia. Regarding these languages, they state: "Our point prediction is that these will be the two most important languages for scientific computing in the medium term, for economics and more broadly. Both are modern, well designed, open source, high productivity languages. Julia is a recent arrival specifically orientated towards scientific computing that mixes high productivity and high performance." The code library for the lectures can be very easily installed in Julia simply by executing the command `Pkg.add("QuantEcon")` in a running Julia instance.

- Aruoba and Fernández Villaverde's paper A Comparison of Programming Languages in Economics (2014) compares a number of programming languages by solving a stochastic growth model using value function iteration. They find that "Julia, with its just-in-time compiler, delivers an outstanding performance. Execution speed is only 2.64–2.70 times slower than the speed of C++. Julia is slightly faster than Java and close to four times faster than Matlab. Given how close Julia's syntax is to Matlab's, the fact that it is open-source, and that the language has been designed from scratch for easy parallelization, many researchers may want to learn more about it."
- Bradley Setzler's blog Julia/Economics (Setzler 2014) is "a tutorial series for economists learning to program in the Julia language." It provides code examples and discussion of econometric methods, including use of `pmap()` for parallel computing. This provides an alternative method to the MPI-based methods discussed in this note.
- Sébastien Villemot presented an introduction to Julia at the 2014 CEF conference (Villemot 2014). This covers the features of the language and offers performance tips using examples such as the Hodrick–Prescott filter and the solution of a rational expectations model using value function iteration.
- Creel and Kristensen (2015a) use Julia and MPI to select statistics out of a large candidate set, for subsequent use in ABC. Their code is available at http://www.runmycode.org/companion/view/1116 .

This list is likely to become out of date fairly quickly, but it will provide an economist who is a new user of Julia with a set of materials that will help to get started.

2 MPI and Julia

The Julia language has a number of mechanisms for parallel computing, including parallel for loops, distributed and shared arrays, and the `pmap()` and `@spawn` methods. Some of these possibilities, for example, shared arrays, are not yet mature as of version 0.3.8. Other methods, such as `@spawn`, may be somewhat cumbersome and unnatural for the sorts of computations that economists are likely to require. The `pmap()` method is well illustrated in an example of bootstrapping on Bradley Setzler's blog, mentioned above. Even though these methods might be used, and while they may become more efficient and easier to use in the future, the well-known and well-tested method of parallel computing using MPI is already available to Julia users, in the same form that it has been available for a number of years to users of languages such as C, Fortran, Octave, R, Ox and Python. The accumulated experience with these and other languages, and the available code base, are resources that may profitably be brought to bear when programming using Julia.

The "message passing interface" (MPI) is a standard that specifies the syntax and semantics for a set of methods of data interchange within C and Fortran computer programs. Many implementations have been developed by different providers. A well known implementation is Open MPI (www.open-mpi.org), which is what was used to run the examples given below. MPI is one of the primary data interchange methods that is used to run parallel programs on clusters and supercomputers. MPI may be made

available to higher level languages through the use of "wrapper functions" which allow high level languages such as Ox or Python to make use of C or Fortran code. Julia has excellent support for using C and Fortran code, so it is not surprising that wrappers for MPI functions have become available for Julia.

Within economics, MPI is already fairly well known. Early descriptions and simple examples of use of MPI in economics were given by Swann (2002), Racine (2002), Creel (2005), and Doornik et al. (2006). Examples of use of MPI in the course of serious research in economics are given by Creel and Kristensen (2012, 2015a, b) and the EMM code by Gallant and Tauchen (2013), which has been used in a number of research papers.

The MPI interface for the Julia language is described at https://github.com/JuliaParallel/MPI.jl. To build the package, CMake, and a working MPI installation for C and Fortran are required. The results reported here were obtained on systems running Debian GNU/Linux. On such a system, one can install Julia, CMake and Open MPI using the command `apt-get install julia cmake openmpi-bin`. Once the dependencies are installed, the MPI package is installed by executing `Pkg.update()` then `Pkg.add("MPI")` from the Julia prompt.

3 Basic Example

Most new MPI users will probably want to know how to send and receive messages. The example `example1.jl`, which accompanies this paper, illustrates this. The code is:

```
1  import MPI
2  function main()
3      MPI.Init()
4      comm = MPI.COMM_WORLD
5      size = MPI.Comm_size(comm)
6      rank = MPI.Comm_rank(comm)
7      recv_mesg = zeros(100,1)
8      if rank == 0
9          send_mesg = rand(100,1)
10         for i = 1:size-1
11             MPI.Send(send_mesg, i, 0, comm)
12         end
13     else
14         MPI.Recv!(recv_mesg, 0, 0, comm)
15     end
16     m = mean(recv_mesg,1)
17     sleep(rank*2)
18     println("Results on rank ",rank,": ",m)
19     MPI.Barrier(comm)
20     MPI.Finalize()
21 end
22 main()
```

The following brief discussion goes over this example, which may be sufficient in order to understand how to use MPI-enabled Julia code, such as the Monte Carlo example code in the next section. For learning to write MPI-enabled Julia code, a separate study of basic MPI would be needed. Line 1 shows how to make the functions

in the MPI package available. Lines 3–7 are very standard for MPI programs. Line 3 initializes MPI communications. Line 4 establishes the MPI communicator, which is a handle to a set of processes on the physical computer(s), the elements of which can communicate with one another. Line 5 determines the size of the communicator, and Line 6 lets each parallel instance of the program determine its own rank in the communicator. Lines 7 creates an array to hold data that will be exchanged. Lines 8–15 allow for processes to take one of two paths. The process with rank = 0 (as determined in Line 6) sends a message to all other nodes, while the other processes in the communicator receive the message from rank 0. In Lines 16–18, each node reports some results. Line 19 ensures that all ranks reach the same point, before the communicator is shut down, in Line 20. Note that Line 17 causes each rank to pause for an amount of time related to the rank number, which ensures that the output from different processes printing results to the screen will not be mixed together.

To run this, one executes a command like `mpirun -np 5 julia example1.jl` from the system prompt (not the Julia prompt). The "5" in this command means that five ranks are to be used. The output is

```
1 michael@yosemite:~/Desktop/Papers/JuliaMPI$ mpirun -np 5 julia
      example1.jl
2 Results on rank 0: [0.0]
3 Results on rank 1: [0.45776271888466374]
4 Results on rank 2: [0.45776271888466374]
5 Results on rank 3: [0.45776271888466374]
6 Results on rank 4: [0.45776271888466374]
7 michael@yosemite:~/Desktop/Papers/JuliaMPI$
```

Rank 0 reports the mean of the array of zeros, while the other ranks report the mean of the 100 uniform number that were generated in Line 9. Line 17 has caused the ranks to report their results in order. This simple example has introduced some basic ideas, such as different code paths within the communicator, sending and receiving, barriers, and so forth. With these ideas, one can understand simple MPI programs.

4 Monte Carlo

Monte Carlo studies are a basic part of investigation in econometrics, to study and verify properties of estimators. Monte Carlo studies have an embarrassingly parallel structure, which means that the replications are independent of one another and may run simultaneously, without any interaction. Such problems are ideal candidates for parallel computing (see Creel 2005, for discussion and references). The MPI methods seen in the `example1.jl` example are almost enough to write a general purpose Monte Carlo function. In this section we present a general purpose function to do Monte Carlo and illustrate it with two examples, a simple one involving approximating π, and a more realistic one that investigates the properties of an econometric estimator.

4.1 Approximating π

If one were to throw random darts at a square target, the expected value of the proportion of darts that land in a circle inscribed in the square is $\pi/4$: see http://en.wikipedia.org/wiki/File:Pi_30K.gif[2]. This leads us to the following code which approximates π:

```
1 reps = Int(1e6) # desired number of MC reps
2 results = zeros(reps,1)
3 for sofar = 1:reps
4     results[sofar,:] = 4.*(norm(rand(2,1)) .< 1.)
5 end
6 println("pihat: ", mean(results))
```

Running this gives:

```
1 michael@yosemite:~/Desktop/Papers/JuliaMPI/Pi$ time julia pi.jl
2 pihat: 3.139056
3
4 real    0m1.243s
5 user    0m1.272s
6 sys     0m0.444s
7 michael@yosemite:~/Desktop/Papers/JuliaMPI/Pi$
```

We see that the approximation seems to work, and that it is pretty quick. There's clearly no need to parallelize this code, but nevertheless, this code has the basic structure of a Monte Carlo study: the computations in lines 3–5 of the program are a number of independent repetitions of the same calculation. Here's a version that can make use of a general purpose MPI-enabled Monte Carlo program:

```
1  include("montecarlo.jl")
2
3  function pi_wrapper()
4      pihat = 4.*float((norm(rand(2,1)) .< 1.))
5  end
6
7  # this function reports intermediate results during MC runs
8  function pi_monitor(sofar, results)
9      # examine results every 2.5*10^5 draws
10     if mod(sofar,2.5e5)==0.
11         m = mean(results[1:sofar,:],1)
12         println("reps so far: ", sofar)
13         println("pihat: ", m)
14         println()
15         #if sofar == size(results,1)
16         #    writedlm("mcresults.out", results)
17         #end
18     end
19 end
20
21 # do the monte carlo: 10^6 reps of single draws
22 function main()
23     reps = Int(1e6) # desired number of MC reps
24     nreturns = 1
```

[2] We make the unrealistic assumption that one never misses the target completely!

```
    pooled = Int(50000)
    montecarlo(pi_wrapper, pi_monitor, reps, nreturns, pooled)
end

main()
```

This code has put the computation that is repeated into a function, in Lines 3–5. Lines 7–19 define a function that can be used to monitor results during the course of the Monte Carlo replications. Note that final results can be written to disk if Lines 15–17 are uncommented. Finally, Lines 22–24 define arguments, and Line 26 calls the general purpose Monte Carlo function. This function is general purpose, because the computations that are repeated are defined by the first argument, and the results that are displayed are defined by the second argument. Running this code is done from the system prompt, as we need to use `mpirun`, just as in the example of the previous section:

```
michael@yosemite:~/Desktop/Papers/JuliaMPI/Code/Pi$ time mpirun -
    np 5 julia pi_mpi.jl
reps so far: 250000
pihat: [3.13328]

reps so far: 500000
pihat: [3.1348719999999997]

reps so far: 750000
pihat: [3.1357706666666667]

reps so far: 1000000
pihat: [3.13508]


real    0m14.094s
user    0m36.284s
sys     0m10.388s
michael@yosemite:~/Desktop/Papers/JuliaMPI/Code/Pi$
```

The code takes more than ten times longer than the simple version! Actually, this is not surprising. The computations are very simple, so any gains from parallel execution are wiped out by the communication overhead introduced by using MPI, plus the use of the monitoring function, which does not exist in the simple serial code. To see where the communication overhead is incurred, let's examine the `montecarlo.jl` code, which is called in line 26 of the previous listing:

```
import MPI
function montecarlo(mc_eval::Function, mc_monitor::Function, reps,
     n_returns, pooled=1, sleeptime=0.)
    blas_set_num_threads(1)
    # set up the MPI communicator
    MPI.Init()
    comm = MPI.COMM_WORLD
    MPI.Barrier(comm)
    rank = MPI.Comm_rank(comm)
    commsize = MPI.Comm_size(comm)
```

```
10
11      # containers and book keeping
12      contrib = zeros(pooled, n_returns)
13      results = zeros(reps, n_returns)
14      pernode = reps / (commsize - 1)
15
16      # check arg
17      if mod(pernode,pooled) != 0
18          if rank == 0
19              println("error \(montecarlo\): pooled must be even
                    divisor of reps/(ranks-1)")
20          end
21          MPI.Barrier(comm)
22          MPI.Finalize()
23          return 0
24      end
25
26      # workers' code
27      if rank > 0
28          @inbounds for i = 1:pernode/pooled
29              # do work
30              for j = 1:pooled
31                  contrib[j,:] = mc_eval()
32              end
33              MPI.Isend(contrib, 0, rank, comm)
34          end
35
36      else # frontend
37          sofar = 0 # results collected so far
38          done = false
39          while ~done
40              sleep(sleeptime) # flood control if job is costly
41              @inbounds for node = 1:commsize-1
42                  # check for results
43                  ready = false
44                  ready, junk = MPI.Iprobe(node, node, comm)
45                  if ready # get them if they're ready
46                      sofar +=1
47                      if sofar*pooled <= reps
48                          MPI.Recv!(contrib, node, node, comm)
49                          results[sofar*pooled-pooled+1:sofar*pooled
                                ,:] = contrib
50                          mc_monitor(sofar*pooled, results)
51                      end
52                      if sofar == reps/pooled
53                          done = true
54                          break
55                      end
56                  end
57              end
58          end
59      end
60      MPI.Barrier(comm)
61      MPI.Finalize()
62  end
```

One can see that there is a fair amount more going on than in the case of the simple serial code. Line 3 restricts computations that use the linear algebra library to use a single thread, because the model of parallelization being used is to parallelize the separate Monte Carlo replications, rather than to try to parallelize individual replications, too. The MPI communicator is set up in lines 4–9, just as before. Lines 16–24 check an argument, and possibly report an error message. Then the code splits into two paths, one for worker ranks (those with rank > 0), which perform the Monte Carlo replications, and another for rank 0, which collects and reports the results.

- *Workers* In line 31, workers perform the computation that generates a single Monte Carlo replication, by executing whatever function was passed as the first argument. These results are collected, and when the specified number (the fourth argument to the function) are accumulated, they are sent to node 0 in Line 33. The send method, `MPI.Isend()`, is non-blocking, so that the worker can resume generating more replications while the results are still being transmitted to rank 0. The reason why a number of results are accumulated before sending them is to avoid excessive communications overhead. The number of results to be pooled should be larger when the underlying computation of each replication is less time consuming. Each worker continues to generate results until it has performed its share of the total replications, then it exits the main loop.
- *Rank 0* The "frontend" rank starts its part of the main loop in line 36. In line 44, a non-blocking probe is used to check, in turn, if any of the workers has sent results. If, so, they are received in line 48, and stored in line 49. Line 50 calls the monitoring function, which is specified as the second of the arguments to `montecarlo.jl`. Once the requested number of replications has been received, the frontend node exits the main loop.
- *All nodes* After the main loop, lines 60 and 61 first synchronize all nodes, and then shut down the MPI communicator.

Having seen the extra communication that the parallel version introduces, it is not surprising that we observe a slow down for the π approximation. However, when the function that is called in line 31 of the previous listing is computationally intensive, then the communication overhead will be small in relation to the part that is parallelized, and we will observe an improvement from parallelization (see Creel and Goffe 2008). This is the case for the example in the next section.

4.2 Monte Carlo Study of an ABC Estimator of an Auction Model

The π example is simple, but it provides a template for more interesting applications. Creel and Kristensen (2013) report results for an ABC estimator of a structural auction model. That estimator relied on simple sampling from the prior, using Algorithm 1 as described in Creel and Kristensen (2015a). In the same paper, Algorithms 2 and 3 describe an adaptive importance sampling method for ABC estimation, which was applied by Creel and Kristensen (2015b) for estimation of jump-diffusion models, and which is used here for estimation of the auction example of Creel and Kristensen (2013). The use of importance sampling should allow for computation of a precise ABC estimator while using many fewer simulations than if sampling is done directly from

the prior. However, the construction of the importance sampling density is somewhat computationally intensive in itself, because it involves adaptively selecting particles. For this reason, Monte Carlo study of the estimator is computationally intensive, because each Monte Carlo replication involves constructing the importance sampling density from scratch.

Julia code to perform a Monte Carlo study of the ABC estimator of the auction model, using adaptive importance sampling, is provided, but it is not discussed, because the methods have been described in the cited papers, and because the intention here is simply to show how parallelization can achieve a speedup of a Monte Carlo study of the estimator. In order to perform a Monte Carlo study of the estimator, the following code may be used:

```
include("Auction.jl") # load auction model code
include("AIS.jl") # the adaptive importance sampling algorithm
include("montecarlo.jl")

function AuctionWrapper()
    # true theta
    theta = [0.5 0.5]
    # generate 'true' aux. stat.
    Zn = aux_stat(theta)
    nParticles = 500 # particles to keep per iter
    multiples = 5 # particles tried is this multiple of particle
        kept
    StopCriterion = 0.1 # stop when proportion of new particles
        accepted is below this
    AISdraws = 5000
    neighbors = 25
    contrib = AIS_fit(Zn, nParticles, multiples, StopCriterion,
        AISdraws, neighbors)
end

# the monitoring function
function AuctionMonitor(sofar, results)
    if mod(sofar,100) == 0
        theta = [0.5 0.5]
        m = mean(results[1:sofar,1:end],1)
        er = theta - m; # theta defined at top level, so ok to use
        b = mean(er,1)
        s = std(results[1:sofar,:],1)
        mse = s.^2 + b.^2
        rmse = sqrt(mse)
        println()
        println("reps so far: ", sofar)
        println("mean: ", m)
        println("bias: ", b)
        println("st. dev.: ", s)
        println("rmse.: ",rmse)
    end
end

function main()
    reps = 100  # desired number of MC reps
    n_returns = 2
```

```
40      pooled = 1 # do this many reps b
41      montecarlo(AuctionWrapper, AuctionMonitor, reps, n_returns,
            pooled)
42  end
43
44  main()
```

This code is very similar to the code for computing π. The supporting code is loaded in lines 1–3. Lines 5–16 specify the function to be Monte Carlo'd. The heart of this is in line 15, where the ABC estimator is computed. Lines 18-35 define the monitoring function, which is a little more complex that the one used in the π example. Line 41 calls the same general purpose Monte Carlo function as was used in the π example. The main difference is that the function to be replicated, `AuctionWrapper`, which computes the econometric estimator, is much more computationally demanding, because the function called in Line 15 is computationally intensive.

The file `SerialRun.jl` which is provided with this paper does 100 serial evaluations of AuctionWrapper(), without using the `montecarlo()` interface. This code executes in 1308 seconds on a test computer. The test machine has 32 codes, 32GB of RAM, runs Debian Linux, and has v0.4 of Julia installed. Running the code using 26 MPI ranks gives the following:

```
1  michael@pelican:~/JuliaMPI/Auction$ time mpirun -np 26 julia
       AuctionMC.jl
2
3  reps so far: 100
4  mean: [0.4957610172450354 0.5127891688119225]
5  bias: [0.004238982754964593 -0.012789168811922491]
6  st. dev.: [0.027288166671378705 0.05802974658106408]
7  rmse.: [0.027615448848096438 0.0594223386207777]
8
9  real    2m22.140s
10 user    58m15.564s
11 sys     0m33.080s
12 michael@pelican:~/JuliaMPI/Auction$
```

We see that wall clock time (real, in the output above) was 2 mins and 22 s, or 142 s. The speedup is roughly $9\times$ (1308/142 = 9.21). The reason that we do not obtain a $20\times$ speedup is because all of computational cores share the same CPU cache resources, as the test machine is a single computer. If one were to use multiple computers, in a cluster, which is a simple matter when parallelization is done using MPI, one would experience less cache contention issues, and a greater speedup would be possible. We see in this example that Julia with MPI is able to achieve a good speedup when the task being parallelized is computationally expensive. Incidentally, the performance of the ABC estimator is becoming apparent, in that we observe that bias and root mean squared error are low over the 100 Monte Carlo replications.

5 Conclusion

This note has shown some examples of how the Julia language may use MPI for parallelization. Parallelization using MPI has the advantage that code is portable:

code can be developed and tested on a lightweight portable computer, and then run on a powerful cluster. The Julia language is fairly new, featuring an intuitive and comfortable syntax, but it also has performance comparable to C and Fortran. It is also free, and it runs on all of the popular operating systems. The ability to combine Julia with MPI leads to a programming environment that is comfortable, powerful, free, OS agnostic, and portable. This is a combination that should be attractive to many economists, and it is a feature combination that may not be offered so completely by any other programming language.

References

Aruoba, S.B., & Fernández Villaverde, J. (2014). A comparison of programming languages in economics. http://economics.sas.upenn.edu/~jesusfv/comparison_languages.

Creel, M. (2005). User-friendly parallel computations with econometric examples. *Computational Economics*, *26*, 107–128.

Creel, M., & Goffe, W. L. (2008). Multi-core CPUs, clusters, and grid computing: A tutorial. *Computational Economics*, *32*, 353–382.

Creel, M., & Kristensen, D. (2012). Estimation of dynamic latent variable models using simulated nonparametric moments. *Econometrics Journal*, *15*, 490–515.

Creel, M., & Kristensen, D. (2013). Indirect likelihood inference (revised), UFAE and IAE Working Papers. http://pareto.uab.es/wp/2013/93113.

Creel, M., & Kristensen, D. (2015a). On selection of statistics for approximate Bayesian computing (or the method of simulated moments), *Computational Statistics & Data Analysis*. doi:10.1016/j.csda.2015.05.005.

Creel, M., & Kristensen, D. (2015b). ABC of SV: Limited information likelihood inference in stochastic volatility jump-diffusion models. *Journal of Empirical Finance*, *31*, 85–108. doi:10.1016/j.jempfin.2015.01.002.

Doornik, J. A., Hendry, D. F., & Shephard, N. (2006). Parallel computation in econometrics: A simplified approach. In E. Kontoghiorgies (Ed.), *Handbook on parallel computing and statistics* (pp. 449–476). London: Chapman & Hall/ CRC.

Gallant, A.R., & Tauchen, G. (2013). EMM: a program for efficient method of moments estimation, v2.6, User's Guide. http://www.aronaldg.org/webfiles/emm/emm.tar.

Racine, J. (2002). Parallel distributed kernel estimation. *Computational Statistics & Data Analysis*, *40*, 293–302.

Sargent, T., & Stachurski, J. (2014). Quantitative economics. http://quant-econ.net/jl/index.html.

Setzler, B. (2014). Julia/Economics. http://juliaeconomics.com.

Swann, C. A. (2002). Maximum likelihood estimation using parallel computing: An introduction to MPI. *Computational Economics*, *19*, 145–178.

Villemot, S. (2014). Julia introduction at CEF 2014. http://econforge.github.io/posts/2014/juil./28/cef2014-julia/ .