Gretl + MPI

Using MPI with gretl

Allin Cottrell
Department of Economics
Wake Forest University

Riccardo (Jack) Lucchetti
Dipartimento di Scienze Economiche e Sociali
Università Politecnica delle Marche

August, 2023

Contents

1   Readership

You may be interested in this document if you have some large, time-consuming computational tasks that could benefit from parallelization, and you're willing to learn about a paradigm that allows you to take control over the division of labor by cores on a given machine, or by machines on a network, using gretl.

2   What is MPI?

To get the full story, see open-mpi.org or mpich.org; here we just describe the basics as they apply to gretl. MPI (Message Passing Interface) is a de facto standard that supports running a given (C or Fortran) program simultaneously on several cores in a given computer and/or on several networked computers. MPI provides means for dividing labor among the multiple nodes and for communicating data between them. It thereby supports a very flexible sort of parallelism.

It is important to understand that (in the simplest variant, at any rate) each MPI process or node (a) is running exactly the same program, but (b) has its own local copy of all data. This is known as SPMD (Single Program, Multiple Data). Since you're unlikely to want to generate n identical copies of a given set of results, doing something interesting with MPI requires that you implement a division of labor, which will also generally involve sending data between processes. We explain how this is done via gretl in section 5 below.

3   Modes of parallelization

Support for parallelization via OpenMP has long been an option when building gretl, and it is present in the gretl packages for MS Windows. But although MPI and OpenMP are both in the business of parallelization, they operate at different levels and have different characteristics. For one thing, OpenMP is limited to multi-threading on a given (multi-core) machine whereas MPI is capable of operating across machines (e.g. in a "cluster"). Beyond that, there's a fundamental architectural difference. With OpenMP, all data is in common between the threads unless you specifically mark certain variables as "private", while with MPI all data is private to the given process unless you specifically request that it be sent between processes somehow or other.

This means that OpenMP lends itself to what we may call "fine-grained" parallelization, where relatively few variables have to be marked as private or local. A classic example is matrix multiplication, where only the row and column indices have to be "thread-local". Use of OpenMP tends to become unwieldy for larger problems where many variables must be localized to prevent the threads from over-writing each others' calculations. Conversely, MPI lends itself to relatively "coarse-grained" parallelization, where each process can get on with its share of a complex job using its own copy of the relevant data.[1]

In practical terms, while it would be very difficult to allow the gretl user to control the details of OpenMP threading via hansl scripting it is not too hard to enable user-level control over MPI. A further practical point regarding the relationship between MPI and OpenMP is discussed in section 6.1.

4   How do I set this up?

You can get MPI working in gretl using reasonably recent builds (2019a or higher recommended) for MS Windows, macOS, or Fedora or Debian-based Linux distributions. On Linux other then Fedora or Debian you may have to build gretl yourself.[2] In each case you will need to have a suitable MPI package installed. The details differ by platform and are given in section 9 below.

---

[1]For anyone who wants to learn more about the various methods and forms of parallelization, a good reference is https://computing.llnl.gov/tutorials/parallel_comp/.

[2]To see if the gretl build supplied by a distro is MPI-enabled, check for the presence of gretlmpi in /usr/bin.

Running an MPI-enabled program—such as `gretlmpi`, described below—requires a special launcher, `mpiexec`. This is supplied by MPI packages, the best known of which are Open MPI, MPICH and Microsoft's MS-MPI.[3]

At a general level, if you want to run MPI-enabled gretl on multiple hosts you will need to install both gretl and MPI on all of them. You will also need to set up a suitable login infrastructure (e.g. by putting your RSA public key in the right place on each host, for `ssh` connectivity). We won't go into that here; see the MPI documentation (available at the sites mentioned in section 2) for details. Running MPI-enabled gretl on a single machine is more straightforward.

5 How do I actually use this?

5.1 Establishing a hosts file

If you plan to use MPI-enabled software across multiple machines, you will probably want to establish a plain text file that tells MPI what hosts and/or cores are available for use. The basic syntax of this file is simple: it contains one or more hosts (specified by name or by IP number), each on a separate line. The syntax for specifying the number of cores or threads available on each host differs slightly between MPI variants: Open MPI wants `slots=`n (where n is the number of threads) following the hostname, MPICH wants `:`n "stuck onto" the hostname, and MS-MPI wants plain n separated from the hostname with a space. So the line for a machine `myhost.mydomain` with 4 cores would have the following variants:

```
# Open MPI
myhost.mydomain slots=4
# MPICH
myhost.mydomain:4
# MS-MPI
myhost.mydomain 4
```

It doesn't matter what the hosts file is called; you will supply its name when needed, as described below. Note, however, that such a file is not required if you're just running a program on multiple cores on a single local machine; in fact it is recommended that you do *not* use one.

5.2 Running gretlmpi directly

As mentioned above, a special launcher is required to run an MPI-enabled program. Under Open MPI a simple invocation of `gretlmpi` might look like this

```
mpiexec -n 4 gretlmpi myscript.inp
```

Following the `-n` tag you specify the number of processes to be run; then comes the name of the MPI program, `gretlmpi` (give a full path if necessary); then the name of the script to execute. Note that `gretlmpi` runs in batch mode only (not interactively) and the `-b` flag that would be have to be passed to `gretlcli` for batch-mode operation is not required.

One relevant command-line option for `gretlmpi` may be inserted before the name of the script file, namely `--single-rng` (or `-s`).[4] The effect of this is explained in section 7.

If you are using a hosts file the Open MPI command line might look like the following, where you give the path to your file following the `--hostfile` tag:

```
mpiexec --hostfile mpi.hosts -n 16 gretlmpi myscript.inp
```

---

[3]MPI-enabled programs are compiled with a special compiler-wrapper, `mpicc`, which is supplied by both Open MPI and MPICH. You will need `mpicc` only if you are building MPI-enabled gretl from the C sources yourself.

[4]For a complete listing of program options, do `gretlmpi --help`.

Command-line syntax differs slightly across `mpiexec` implementations. In place of Open MPI's `--hostfile` tag you would use `-machinefile` with MPICH, or `/machinefile` with MS-MPI. Also under MS-MPI you should use `/np` rather than `-n` for the number of processes option.

The `mpiexec` program supports several additional options not discussed here; see the documentation for your MPI implementation for details. We do, however, discuss a further question concerning the `mpiexec` command line in section 6.1.

5.3 Running gretlmpi indirectly

Rather than invoking `gretlmpi` directly from the command line you can have gretl invoke it for you. To signal this you need to construct an MPI command block of the form `mpi ... end mpi`, as elaborated in section 5.7. (Note that this is *not* required if you invoke `gretlmpi` directly; in that case the entire script is automatically an "MPI block".)

Certain aspects of the MPI setup on your system can be specified in the tab labeled MPI in the gretl preferences dialog (under Tools, Preferences, General in the GUI program), as follows:

- The path to your MPI hosts file, if applicable (see section 5.1). This can also be set via the environment variable `GRETL_MPI_HOSTS`.

- The full path to the `mpiexec` binary. It should be necessary to set this only if `mpiexec` is not in your PATH.

- The type of your MPI installation, Open MPI or MPICH.[5]

These settings are recorded in the per-user gretl configuration file, and they are picked up if you execute a script containing an `mpi` block in either the GUI program or `gretlcli`.

5.4 Why do we need a separate program?

That is, why can't we just produce MPI-enabled versions of the "traditional" gretl command-line and GUI programs?

Well, given the design of MPI this is not so easy. For one thing, as we've noted, running an MPI-enabled program requires a special launcher which might not be available on all systems. For another, an MPI-enabled program must initialize the MPI system right away, on pain of invoking undefined behavior. Most of the time most users of gretl will have no need for MPI parallelization. The easiest approach is to limit the use of MPI to the loci where we know it's really wanted by running a distinct program—and it's best for you that we don't occupy all your cores or network hosts with running multiple copies of what's really single-threaded code.

5.5 A few MPI concepts

Before getting into the relevant hansl commands and functions it may be worth outlining some of the basic concepts they implement.

Suppose you launch an MPI program with n processes (meaning that n instances of the program are run). Each process has an ID number, or "rank" in MPI parlance. This is zero-based, so the ranks range from 0 to $n - 1$. To get a useful division of labor going you can

- explicitly condition on rank, and/or

- arrange to give the processes different data to work on.

---

[5]On MS Windows, only MS-MPI is currently supported by gretl.

Many tasks that are suitable for parallelization via MPI involve a conceptually single-threaded preparation phase and/or final phase. In that case you would condition on rank such that these phases are carried out by a single process. It is a common convention (though not required) that the process with rank 0 does this sort of work. The overall structure of such a program might then look like this:

1. Process 0 does some preliminary work.

2. Process 0 sends data to all processes.

3. All processes do some work in parallel.

4. Process 0 collects the results.

5. Process 0 does some final work, if required.

Consider step 2 above: depending on the nature of the task, we may want process 0 to send the same data to all processes (**broadcast** the data) and/or send a share of the data to each process (**scatter** the data). There may be several data items to be transmitted, some of which should be broadcast (e.g. each process gets a copy of an initial parameter vector) and others scattered (e.g. each process gets a chunk of the observations on some variables of interest). Broadcast and scatter are basic MPI operations, implemented in hansl via the `mpibcast` and `mpiscatter` functions.

Now consider step 4 above. Again depending on the task in hand, we may want process 0 to **reduce** information from all processes (e.g. form the sum of n values) and/or to concatenate or **gather** results (e.g. form a big matrix by stacking rows from all processes). Reduce and gather are also basic MPI operations; they are implemented jointly in hansl's `mpireduce`.

The procedures just mentioned—**broadcast**, **scatter**, **reduce** and **gather**—are all multilateral. The functions that implement them must be called by all processes (that is, calls to these functions must occur in a common block of the MPI program, outside of any conditioning on rank). In each of them, one process plays a special role and is known as "root": the root process is the *source* of the original data in **broadcast** and **scatter**, and the *recipient* of consolidated data in **reduce** and **gather**. There's no requirement that root be process 0, and moreover there's no requirement that the root process be the same in each call to such procedures; nonetheless, a common case is that root is always process 0, and that is the default assumption in hansl.

Besides multilateral data transfer, MPI also supports the bilateral procedures **send** and **receive**. In **send** a given process supplies some data and specifies a process to which it should be sent; in **receive** a given process requests data from another specified process. These procedures must be suitably paired: e.g. process k issues a **send** to process j while process j calls for a **receive** from process k. Such calls must occur in a block of the program that is conditional on process rank.

5.6   What goes into an MPI-type hansl script?

In a hansl script to be executed via `gretlmpi`—as also in the context of an MPI command block in a "regular" hansl script—you have access to all the standard gretl commands and functions,[6] plus some extra ones.

First, you have these two accessors:

`$mpirank`	gives the MPI rank of "this" process
`$mpisize`	gives the number of processes or size of the MPI "world"

---

[6]The error-throwing command `funcerr` and its sister hansl function `errorif()` constitute exceptions to this rule. You should avoid `funcerr` in hansl functions that are going to be used in an MPI context. The built-in function `errorif()` can be used within an MPI script or block, but in such a context it will not actually terminate execution, printing only the specified error string.

Note that when gretl is not in MPI mode $mpirank returns −1 and $mpisize returns 0.

To shunt data around between the processes you have the following functions:

```
scalar mpisend(object x, int dest)                      send object x to node dest
object mpirecv(int src)                                 receive an object from node src
scalar mpibcast(object *x [,int root])                  broadcast object x
scalar mpireduce(object *x, string op [,int root])      reduce object x via op
scalar mpiallred(object *x, string op)                  reduce object x via op, all nodes
scalar mpiscatter(matrix *m, string op [,int root])     scatter matrix m using op
```

By "object" above we mean a matrix, scalar, bundle, array, string or list. All of these types are supported by mpisend, mpirevc and mpibcast. The application of mpiscatter and mpireduce is more limited. At present only matrices are supported for mpiscatter, while mpireduce can be used for matrices, scalars and arrays.

The scalar return value from these functions (apart from mpirecv, which returns the requested object) is merely nominal: they return 0 if they succeed. The root argument to mpibcast, mpireduce and mpiscatter is optional, and defaults to 0; use of this argument is discussed below.

As you might expect, mpisend and mpirecv have to be paired suitably, as in the following fragment which sends a matrix from the process with rank 2 to the one with rank 3.

```
if $mpirank == 2
  matrix C = cholesky(A)
  mpisend(C, 3)
elif $mpirank == 3
  matrix C = mpirecv(2)
endif
```

The functions mpibcast, mpireduce, mpiallred and mpiscatter *must be executed by all processes.* It follows that the object whose address is passed to these functions must be previously declared in all processes. Calls to these multilateral functions don't have to be paired with anything since they inherently handle both transmission and reception of data.

The mpibcast function sends data from the root process to all processes. Here's a simple example:

```
matrix X
if $mpirank == 0
  X = mnormal(T, k)
endif
mpibcast(&X)
```

After successful completion of the above fragment, each process will have a copy of the matrix X as defined in the process with rank 0.

The mpireduce function gathers objects of a given name from all processes and "reduces" them to a single object at the root node. The op argument specifies the reduction operation or method. The methods currently supported for scalars are sum, prod (product), max and min. For matrices the methods are sum, prod (Hadamard product), hcat (horizontal concatenation) and vcat (vertical concatenation). For arrays, only acat (concatentation) is supported, and reduction is not supported for bundles at present. For example:

```
matrix X
X = mnormal(T, k)
mpireduce(&X, sum)
```

After successful completion of the above, the root process will have a matrix X which is the sum of the matrices X at all processes. Note that the matrices at all processes other than root remain unchanged. If you want the "reduced" variable to replace the original at *all* ranks you can use `mpiallred`: this is equivalent to, but more efficient than, following `mpireduce` with a call to `mpibcast`.

The `mpiscatter` function is used to distribute chunks of a specified matrix in the root process to all processes. The `op` argument must be either `byrows` or `bycols`. Let q denote the quotient of the number of rows in the matrix to be scattered and the number of processes. In the `byrows` case root sends the first q rows to process 0, the next q to process 1, and so on. If there is a remainder from the division of rows it is added to the last allotment. The `bycols` case is exactly analogous but splitting of the matrix is by columns. For example:

```
matrix X
if $mpirank == 0
  X = mnormal(10000, 10)
endif
mpiscatter(&X, byrows)
```

If there are 4 processes, each one—*including root*—will get a 2500×10 share of the original X as it existed in the root process. If you want to preserve the full matrix in the root process you must make a copy of it before calling `mpiscatter`.

The optional trailing `root` argument to the functions `mpibcast`, `mpireduce` and `mpiscatter` allows departure from the default assumption that root is always process 0. For example, if you want process 8 to broadcast a random matrix to all processes:

```
matrix X
if $mpirank == 8
  X = mnormal(T, k)
endif
mpibcast(&X, 8)
```

In addition to the data-transfer functions there's a utility function whose job is simply to enforce synchronization. This is `mpibarrier`, which takes no arguments and has no meaningful return value. It should be placed in a common portion of an MPI-enabled script, where it will be called by all processes: the effect is that no process will continue beyond the "barrier" until the function has been called by all processes.

```
# nobody gets past this until everyone has called it
mpibarrier()
```

Readers who are familiar with MPI will see that what we're offering via hansl is a simplified version of (a subset of) the MPI interface. The main simplification is that the MPI "communicator" is hard-wired as `MPI_COMM_WORLD` and so all processes are members of a single group.

5.7 Use of an MPI block

As mentioned earlier, an MPI block (`mpi ... end mpi`) is a means of embedding commands to be executed by `gretlmpi` in a larger script to be executed in the usual way by `gretlcli` or the GUI program.[7] In this context gretl takes charge of invoking `mpiexec` with appropriate parameters.

The structure here is very similar to that of gretl's `foreign` command-block: gretl takes the statements from within the block in question, sends them for execution by another program, and then

---

[7]Note that an error is flagged if the `mpi` command is encountered in a script being executed directly via `gretlmpi`, since this would amount to an attempt at a duplicate initialization of MPI.

displays the results. As with `foreign`, variables defined within the calling gretl process are not immediately available in the called program, and vice versa. To send data to, or retrieve results from, `gretlmpi` you need to use suitable input/output, for instance via the functions `mwrite` and `mread` for matrices, the functions `bwrite` and `bread` for bundles, or the commands `store` and `open` or `append` for series or lists of series.

The gretl `mpi` command supports the following options:

`--np=`n	specify the number of MPI processes
`--omp-threads=`m	specify the number of OpenMP threads per process
`--send-functions`	share function definitions with `gretlmpi`
`--send-data[=`*list*`]`	share all or part of the current dataset
`--local`	ignore MPI hosts file, if present
`--single-rng`	use a single pseudo-random number generator
`--verbose`	print extra information, including the `mpiexec` command used

The `--np` option plays the same role as the `-n` tag in use of `mpiexec` (section 5.2); it governs the number of processes to be launched. If this option is not given, the default is to use all available processors on the local machine, or all entries in the MPI hosts file if that has been specified.

The `--omp-threads` option is applicable only if gretl is built with support for `OpenMP`: it governs the maximum number of `OpenMP` threads that will be permitted per MPI process. See section 6.1 for an account of why and when you might want to use this option.

The effect of the `--send-functions` option is to send to `gretlmpi` the definitions of any hansl functions present in the workspace of the calling gretl process. (It's OK to define functions within an `mpi` block, but in some cases it may be more convenient to define functions at the "main" script level and pass them on.)

The `--send-data` option can be used to send to `gretlmpi` the content of the current dataset. (It will provoke an error if no dataset is present). The optional *list* parameter restricts the data sent to the series included in the list named by the parameter.

The `--local` option can be used if you have specified a hosts file (see sections 5.1 and 5.3) but in the current context you want the MPI processes to be run on the local machine only.

The `--single-rng` option is explained in section 7.

See section 11 for a complete example of a gretl script that uses an `mpi` block.

6 Performance hints

To get best performance from an MPI-enabled hansl script you need to pay careful attention to certain points. We give a brief account of three relevant issues below. Some of the points mentioned here are taken up in relation to specific examples in sections 11 and 12.

6.1 Contention between MPI and OpenMP

As we mentioned in section 3, MPI and `OpenMP` work at different levels. They are in principle complementary. For example, one might effect a "macro" division of labor across chunks of a dataset via MPI, while at the same time allowing each MPI process to spawn a number of `OpenMP` threads for more "micro" parallelization in tasks such as multiplying matrices. However, given finite computational resources the two modes of parallelization become substitutes at the margin.

Let n_i denote the number of MPI processes running on host i, and m_i the maximum number of `OpenMP` threads permitted per process on the host. If the product $n_i m_i$ exceeds the total number of threads supported by the machine you are liable to get a drastic slowdown, possibly a collapse below the speed of simple single-threaded execution.

It is therefore necessary to budget the use of MPI processes and `OpenMP` threads. Suppose, for example, you're running a gretl/MPI script on a single machine that supports a maximum of 8 threads. To avoid excessive contention you will want to ensure that $n_i m_i \leq 8$. Exactly how you do this depends on whether you are running `gretlmpi` yourself (section 5.2) or having gretl run it for you via an `mpi` block (section 5.7).

When launching `gretlmpi` yourself you can use the environment variable `OMP_NUM_THREADS` to limit the number of `OpenMP` threads that can be used by each process. Here are two examples which limit the total number of threads to 8 in different ways:

```
# give all threads to MPI
OMP_NUM_THREADS=1 mpiexec -n 8 gretlmpi myscript.inp
# divide the resources between MPI and OpenMP
OMP_NUM_THREADS=2 mpiexec -n 4 gretlmpi myscript.inp
```

In the context of an `mpi` block you can use the `--omp-threads` option at the close of the block to set the maximum number of `OpenMP` threads—the effect is that gretl sets `OMP_NUM_THREADS` to your specification when calling `gretlmpi`. Note, however, that when executing an `mpi` block gretl sets `OMP_NUM_THREADS` to 1 by default. It should be necessary to use this option, therefore, only if you want to permit more than one thread per MPI process.

Some experimentation may be necessary to arrive at the optimal budget. See section 11 for an illustration.

6.2 Hyper-threading: help or hindrance?

Current Intel consumer CPUs typically have a certain number of actual physical cores but support twice that number of threads via so-called hyper-threading. This raises the question: when you are figuring the resources available to support MPI processes and/or `OpenMP` should you think in terms of cores or threads, when you have more of the latter than the former?

It depends on the nature of the task in question. Roughly speaking, if a script invokes a lot of "tightly written" C code, capable of driving a machine's cores to their limit, then hyper-threading may actually slow things down. On the other hand, if the invoked code is "looser", hyper-threading can help.

How can you know what sort of C code a given script invokes? Well, if the script does a lot of matrix multiplication it's probably in the "tight" category but other than that it's not so easy to say. It may be necessary to experiment to find the optimum—that is, to determine if you should limit yourself to the number of available cores or run the maximum number of threads.

6.3 Data transfer in MPI

The transfer of data between processes is likely to be a relatively slow phase of an MPI program (particularly if it's taking place between hosts across a network rather than within a single machine). The data-transfer functions discussed in section 5.6 are "blocking" operations; that is, no participating process can move on until the function has finished executing in all the participants. It's therefore important to think carefully about what information is really needed where and when, and to keep transfers to the minimum consistent with the goal of the program.

7 Random number generation

One issue that arises in the MPI context is the distributed generation of pseudo-random sequences. If each of several processes simply uses the same pseudo-random number generator (PRNG) with a different seed, this can end up producing sequences with arbitrary dependency. For this reason,

in gretl/MPI we use by default the DCMT mechanism (Dynamic Creation of Mersenne Twisters) so that each MPI process gets its own, independent PRNG.[8]

However, there are some cases in which you may not wish to use DCMT. The task given to MPI may be such that the several processes are required to produce identical pseudo-random sequences (and then, presumably, do something different with them). Or you may have an MPI-enabled script in which all use of the PRNG occurs in a single process (so you don't have to worry about independence of generators): if you want to get the same results as you would from a single-threaded variant of the script, for a given seed, you need to use gretl's regular PRNG.

You can get gretl to use a single PRNG, of the type used in non-MPI scripts, in various ways depending on the context. If you're running `gretlmpi` yourself, you can use the command-line option `--single-rng`. This option flag can also be attached to the start or end of an `mpi` command block, with the same effect. Alternatively, you can manipulate the state variable `use_dcmt` via the `set` command, as in

```
set use_dcmt off
```

Using this method gives you more flexibility (you can switch back and forth between the types of generator if need be). However, if you know in advance that you have no need for DCMT it is more efficient to use the `--single-rng` option.

Listing 1 illustrates. As written, it will print n identical matrices, but if you comment out the command `set use_dcmt off` the matrices will all be different.

Listing 1: Generating identical sequences

```
set use_dcmt off # or not?
set seed 12337
matrix X = mnormal(3,3)
if $mpirank > 0
  # send matrix to node 0
  mpisend(X, 0)
else
  printf "root, my matrix\n%#13.7g\n", X
  scalar n = $mpisize - 1
  loop i=1..n
    Xi = mpirecv(i)
    printf "matrix from rank %d\n%#13.7g\n", i, Xi
  endloop
endif
```

8 Printing output

Another point to note about MPI is that since each process does its own thing in parallel, the result of a print command that is common to all processes is likely to be messy: lines of output may be interleaved. To ensure a coherent printout it's necessary to send the results to a single process first. This is illustrated in Listing 1: instead of each rank printing its own random matrix, rank 0 collects them all and prints them in sequence.

The usual default behavior when gretl is executing a script is that the input commands are echoed in the output, and various confirmatory messages are printed (e.g. "Generated matrix m"). To get quieter behavior you can issue the commands `set echo off` and/or `set messages off`. When

---

[8]See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html.

gretlmpi is executing a script the default is reversed: `echo` and `messages` are both turned off but you can use the `set` command to turn them on.

9 Platform specifics

9.1 Linux

On Debian-based systems gretl should be MPI-enabled by default. On Fedora an additional package, `gretl-openmpi`, must be installed to provide `gretlmpi`. As mentioned above, on other Linux systems you may have to build gretl yourself (either from a released source package or from the git sources). The tell-tale is whether or not the `gretlmpi` executable is installed, and the command "`which gretlmpi`" in a terminal window should tell you that.

If your distro includes `gretlmpi` you just have to ensure that MPI itself is installed. On Debian and Fedora this should be automatic: Open MPI is a dependency of the gretl package (Debian) or the `gretl-openmpi` package (Fedora). But just in case you need to install MPI yourself here are three variants of the required command:

```
# Debian-based systems
apt-get install libopenmpi
# Fedora
dnf install openmpi
# Arch
pacman -S openmpi
```

If you're building gretl yourself you should ensure that MPI is installed *first*. It's straightforward to install one of the open-source MPI implementations on a Linux system using your distribution's package manager. We recommend Open MPI, since that is what we've mostly used in testing, but MPICH should also work fine. Here are three variants of the required command:

```
# Debian-based systems
apt-get install libopenmpi-dev
# Fedora
dnf install openmpi-devel
# Arch
pacman -S openmpi
```

Once MPI is installed it should probably be found automatically by gretl's `configure` script. If need be you can give the process some help. For instance, if Open MPI is installed under `/opt/openmpi` you might do

```
--with-mpi-lib=/opt/openmpi/lib
--with-mpi-include=/opt/openmpi/include
```

(see `./configure --help`). You can also set the path to the MPI compiler-wrapper, `mpicc`, via the environment variable `MPICC` if need be, as in

```
MPICC=/opt/openmpi/bin/mpicc ./configure ...
```

Note that if MPI is auto-detected but you *don't* want to use it you can give the option

```
--without-mpi
```

when configuring the build.

9.2   MS Windows

Microsoft has its own MPI implementation for Windows, MS-MPI, which is available as a free download; see

https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi

and gretl builds for Windows have supported this since 2017.[9]

When you run the MS-MPI installer this automatically adds the relevant directory to your path, and you should be good to go. Note, however that if you're running MPI on multiple machines (or even if you're just running on a single machine but are using a hosts file) you may have to start the "daemon" program `smpd.exe` first. You can do that by going to the Windows Start menu, selecting Run... and typing

```
smpd -d
```

(This opens a console window, which you can just minimize to get it out of the way.)

9.3   macOS

MPI support is enabled in current builds of gretl for macOS: they include the `gretlmpi` executable linked against Open MPI, which is assumed to be installed under `/opt/openmpi`.

The details differ by processor type. For Macs with Intel processors, `gretlmpi` is linked against Open MPI version 1.6.5, while for Macs with arm64 processors ("Apple Silicon") the linkage is to Open MPI 4.1.5. To activate MPI support it's necessary to install the appropriate package, as follows:

$$\begin{array}{ll} \text{intel} & \texttt{openmpi-1.6.5-mac-intel.zip} \\ \text{arm64} & \texttt{openmpi-4.1.5-mac-arm64.zip} \end{array}$$

These can be found at

http://sourceforge.net/projects/gretl/files/mpi/

To unpack the package you execute the following commands in a Terminal window:

```
# change to root directory
cd /
# for Intel
sudo unzip /where/you/put/openmpi-1.6.5-mac-intel.zip
# OR for arm64
sudo unzip /where/you/put/openmpi-4.1.5-mac-arm64.zip
```

This will install the files under `/opt/openmpi`, which is where gretl will expect to find them; plus, it's a location where they are unlikely to collide with any other MPI implementation you may have installed.

10   MPI dependency questions

From what we've said above it should be clear that `gretlmpi` will not run (and therefore `mpi` script-blocks cannot be executed) unless you have MPI installed. And installing MPI is not something that gretl can do; it's up to you. This raises some questions.

---

[9]In principle you could install Open MPI or MPICH on Windows—though you'd have to build them yourself—but at present gretl on Windows only supports MS-MPI (which is based on MPICH.

10.1   Can I run MPI-enabled gretl without MPI?

Yes. The only thing you *can't* do is use MPI functionality: it's there *in potentia* but the potential is not realized until you install MPI.

The new accessors `$mpirank` and `$mpisize` will "work" but will always return -1 and 0, respectively.

If you try to use MPI-specific functions such as `mpisend` you will get an error, with the message "The MPI library is not loaded".

10.2   How do I test for MPI support in gretl?

To test for MPI support you can use the `$sysinfo` accessor. This is a gretl bundle with several members,[10] one of which is the scalar `mpi`: this has value 0 if gretl has not been built with MPI support enabled (which is the case for the old-style OS X builds that depend on X11). It is also 0 if the gretl build has MPI support but `mpiexec` is not installed. Conversely, if you get a value of 1 from `$sysinfo.mpi`, that tells you that MPI is both supported by gretl and installed (as best gretl can tell).

10.3   For geeks: so is libgretl linked against libmpi?

No. We deliberately avoided doing this, so that an MPI-enabled build of gretl will be usable by people who don't have MPI installed. When the `gretlmpi` binary is running we need access to symbols from the MPI library, but we get them on the fly via `dlopen` rather than linking against the library at build time.

---

[10]For a full listing of the members of the bundle see the current *Gretl Command Reference.*

11 Illustration: bootstrap

This example uses an mpi block to compute bootstrap standard errors following OLS estimation. Not terribly interesting in itself but it's a suitable (and big enough) job to demonstrate benefit from MPI. The overall structure is that process 0 creates an artificial dataset and runs OLS to obtain initial parameter estimates; the data and initial estimates are then broadcast; the bootstrap iterations are divided between all the processes; and finally the results are assembled via `mpireduce`. Listing 2 shows the hansl functions that are used and Listing 3 shows the main program.

Listing 2: hansl functions for OLS bootstrap example

```
# olsboot-funcs.inp: function definitions

function matrix master_task (int T, int k, const matrix b0,
                             matrix *X)
  # construct artificial dataset and run OLS
  X = mnormal(T, k)
  X[,1] = 1
  matrix y = X*b0 + mnormal(T, 1)
  return mols(y, X)
end function

function matrix worker_task (const matrix X, const matrix b,
                             int iters)
  # semi-parametric bootstrap
  matrix Bj = zeros(iters, cols(X))
  matrix U = mnormal(rows(X), iters)
  matrix y0 = X*b
  loop i=1..iters
    yi = y0 + U[,i]
    Bj[i,] = mols(yi, X)'
  endloop
  return Bj
end function

function void B_stats (const matrix B)
  matrix means = meanc(B)
  matrix sds = sdc(B)
  printf "Bootstrap coeff means and std errors:\n\n"
  loop i=1..cols(B)
    printf "%2d % f (%f)\n", i-1, means[i], sds[i]
  endloop
end function
```

You might wonder, why the funny number of total bootstrap iterations (6720)? That's because it's a common multiple of 2, 3, 5 and 7, so we're able to divide the work evenly across various numbers of processes for testing purposes.

To give an indication of the benefit that can be gained by running in MPI mode even without access to a high-performance cluster we timed the execution of the above script on three different multi-core machines (to be clear, in each test just using a single machine).

Machine 1 is a Dell XPS 8300 desktop box of early 2012 vintage running 64-bit Fedora 20 (Intel Core i7-2600, 3.40GHz, with 4 cores and 8 threads). *Machine 2* is a Lenovo ThinkPad X1 Carbon running current 64-bit Arch Linux (Core i7-3667U, 2.00GHz, with 2 cores and 4 threads). *Machine 3* is a Macbook Air of 2010 vintage running OS X 10.6.8 (Core 2 Duo, 1.86GHz, with 2 cores and just 2

Listing 3: Main code for OLS bootstrap example

```
set echo off
set messages off
include olsboot-funcs.inp

# start MPI block
mpi --send-functions

matrix X b B
scalar T = 10000
scalar k = 16
scalar iters = 6720

if $mpirank == 0
  matrix b0 = zeros(k, 1)
  b0[1] = 1
  b0[2] = 5
  set stopwatch
  set seed 123445
  b = master_task(T, k, b0, &X)
else
  scalar my_seed = $mpirank * 1471
  set seed my_seed
endif

# broadcast the data and the initial parameter estimates
mpibcast(&X)
mpibcast(&b)

# divide the iterations among the processes
iters /= $mpisize

B = worker_task(X, b, iters)
mpireduce(&B, vcat)

if $mpirank == 0
  printf "elapsed: %g secs\n", $stopwatch
  mwrite(B, "B_mpi.mat", 1)
endif

end mpi --np=4 --omp-threads=1
# exit MPI block

# retrieve the results from MPI
matrix B = mread("B_mpi.mat", 1)
B_stats(B)
```

threads).

In all tests we compared MPI performance with a single-process baseline. The baseline script was obtained by deleting all the MPI directives from the script shown above and having one process carry out the full number of bootstrap iterations. On machines 1 and 2 we also experimented in search of the fastest combination of number of MPI processes (n) and number of `OpenMP` threads (m).

In Table 1 the time values headed "Calculation" are those printed from the script, using gretl's `stopwatch` apparatus, and those headed "Total" were obtained using the system `time` command (the "real" value). The calculation time is of some interest in its own right but it is net of the MPI overhead and the total time is what really matters to the user.

	Calculation	Total
Machine 1		
baseline, $m = 1$	12.695	12.786
baseline, $m = 8$	8.928	9.036
$n = 4, m = 1$	4.631	6.226
$n = 6, m = 1$	3.300	4.751
$n = 8, m = 1$	2.749	4.835
$n = 4, m = 2$*	3.265	4.214
Machine 2		
baseline, $m = 1$	15.300	15.310
baseline, $m = 4$	13.219	13.231
$n = 2, m = 1$	7.400	9.101
$n = 4, m = 1$*	5.548	7.924
$n = 2, m = 2$	7.381	9.061
Machine 3		
baseline, $m = 1$	25.395	25.451
baseline, $m = 2$	22.569	22.627
$n = 2, m = 1$*	12.475	15.084

Table 1: Bootstrap script, timings in seconds, n = number of MPI processes and m = number of OpenMP threads

What can we conclude from Table 1? For one thing, it appears that (for this problem, on these machines) it is worth running the maximum numbers of threads (that is, hyper-threading is beneficial). We can also clearly see that devoting all threads to parallelization via `OpenMP` (internal to the gretl library) is not nearly as effective as using some (if not all) threads to support MPI. On machine 1 we get best peformance (in the "Total" column) by running 4 MPI processes with 2 threads each; on machines 2 and 3 we do best by devoting all resources to MPI processes. Even on the least capable machine 3, which supports only two MPI processes, we see a substantial gain in speed.

Moreover, the fast Calculation time when all threads are given to MPI on machine 1 ($n = 8, m = 1$) suggests that this might be the winner on a larger problem, where the MPI overhead counts for less. Sure enough, if we multiply the number of bootstrap iterations by 4 (26880), the all-MPI variant is faster than the share-out ($n = 4, m = 2$); we get "Total" times of 14.538 and 18.305 seconds, respectively.

12   Illustration: ML estimation

Our second example illustrates the use of parallelization in computing Maximum Likelihood estimates. We have a large number of drawings (one million) from a gamma distribution and we wish to estimate the parameters of the distribution.

Since the observations are assumed to be independent, the strategy is to divide the data into n chunks and have each MPI process calculate the log-likelihood for its own chunk, given the current vector of parameter estimates; these values can then be summed (a case of **reduce**) to get the overall log-likelihood.

The functions we use are shown in Listing 4 and the main program in Listing 5.

Listing 4: hansl functions for MLE example

```
# gamma-funcs.inp: function definitions

function scalar gamma_llik(const matrix x, matrix param)
  scalar n = rows(x)
  scalar a = param[1]
  scalar p = param[2]
  matrix l =  (p-1) .* ln(x) - a*x
  scalar ret = n*(p * ln(a) - lngamma(p)) + sumc(l)
  return ret
end function

function scalar mpi_gamma_llik(const matrix x, matrix param)
  set warnings off
  scalar llik = gamma_llik(x, param)
  mpiallred(&llik, sum)
  return llik
end function
```

Let's look at the functions first. There's nothing MPI-specific about `gamma_llik`, it just calculates the log-likelihood for a gamma sample in the vector `x` given the parameter values in `param`. The interesting work is done by `mpi_gamma_llik`. In the context of the main program this function is called by all n processes and the input vector `x` is a one-n^{th} share of the full data (the result of a **scatter** operation). Thus the `llik` value produced on the second line of this function is the log-likelihood for a fraction of the data. Recall that `mpiallred` effects a reduction followed by a broadcast. So after the line

```
  mpiallred(&llik, sum)
```

each process is ready to return the total log-likelihood, despite the fact that it only saw its own subset of the data. (Is MPI cool or what?)

Now turn to the main script. Note that there's no `mpi` block here: this script is intended to be executed by `gretlmpi` directly—see section 5.2. One could wrap the entire script in an `mpi` block and run it in the gretl GUI if desired.

After including the functions file we do `set use_dcmt off`. This is not essential, but makes it possible to do a direct comparison of the results from MPI with those from a single-process variant of the script (given a common random seed). Then the rank 0 process prepares the gamma data.

Once the initial set-up is complete, process 0 broadcasts the initial parameter vector (which we deliberately make "off" so that `mle` has some work to do) and scatters the data matrix, `X`.

Listing 5: Main code for MLE example

```
include gamma-funcs.inp
set use_dcmt off

if $mpirank == 0
   scalar N = 1000000
   scalar P = 5
   # generate gamma-distributed data
   matrix X = -sumr(ln(muniform(N, P)))
   scalar mx = meanc(X)
   scalar vx = mcov(X)
   # the known "correct" starting point is:
   # matrix param = {mx/vx, mx*mx/vx}
   # but we'll start from a "wrong" point
   matrix param = {mx/vx, 1}
else
   matrix X param
endif

# broadcast the initial parameter values
mpibcast(&param)

# divide the data up
mpiscatter(&X, byrows)

if $mpirank == 0
   string opt = "--verbose"
else
   string opt = "--quiet"
endif

# all processes do mle
set stopwatch
mle LL = mpi_gamma_llik(X, param)
   params param
end mle @opt

if $mpirank == 0
   printf "elapsed: %g\n", $stopwatch
endif
```

The final step is for all processes to run `mle`. But since we don't want to see n copies of the model results we append the `--quiet` option for all but process 0.

Timings for this example, for machines 1 and 2 as described in section 11, are shown in Table 2. In this case it happens that gretl's internal `OpenMP` threading is not invoked to any appreciable extent, so there's no point in experimenting with different values of `OMP_NUM_THREADS` and the table is simpler than Table 1.

Compared with the first example, hyper-threading is apparently not very helpful. The quickest run on the 4-core, 8-thread machine uses 4 MPI processes, and while the best time on the 2-core, 4-thread machine is obtained with 4 MPI processes, the gain over 2 processes is not large. Nonetheless, we see a substantial gain in speed via MPI compared to the single-process baseline.

	Calculation	Total
Machine 1		
baseline	10.043	10.445
$n = 2$	5.464	6.416
$n = 4^*$	3.201	4.493
$n = 6$	2.720	4.594
$n = 8$	2.405	5.001
Machine 2		
baseline	11.680	12.114
$n = 2$	6.349	8.376
$n = 3$	6.222	8.472
$n = 4^*$	4.934	7.868

Table 2: Gamma MLE script, timings in seconds, n = number of MPI processes

For this example we also have timings from experiments on a cluster comprising 10 "blades", each equipped with two Xeon 2.80GHz CPUs. The Xeons are of the Pentium 4 type, with a single core but two threads. One set of timings was obtained using an MPI hosts file which specified 1 "slot" on the first blade and 2 slots on each of the other 9, giving a maximum of $np = 19$ processes. By stipulating that at most two processes should be run per blade we are restricting the parallelization to distinct physical cores (and avoiding hyper-threading). Figure 1 shows the scatter of calculation time against the number of processes used, along with an inverse fit.
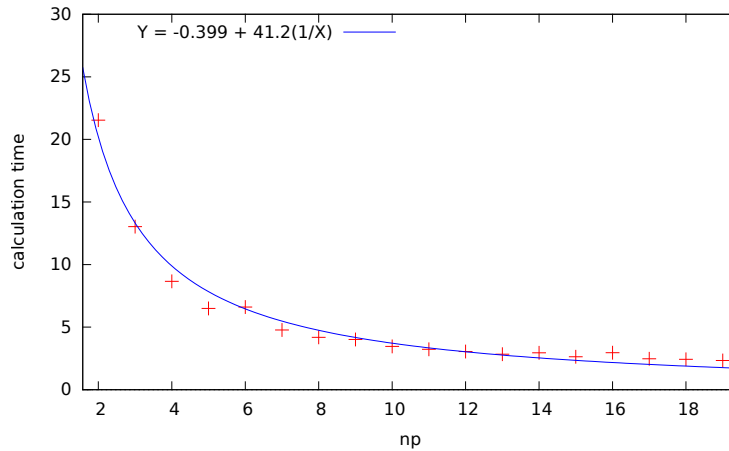

Figure 1: Gamma MLE script: calculation time versus number of processes with inverse fit

A second set of timings was obtained with the hosts file revised to specify 4 slots per blade, thus invoking hyper-threading. Figure 2 compares the results with and without hyper-threading, in the form of a log–log plot. It is apparent that hyper-threading is not advantageous in this context. In fact the calculation time with $np = 19$ and no hyper-threading (2.339 seconds) is much better than the minimum time with hyper-threading (4.018 seconds at $np = 33$).
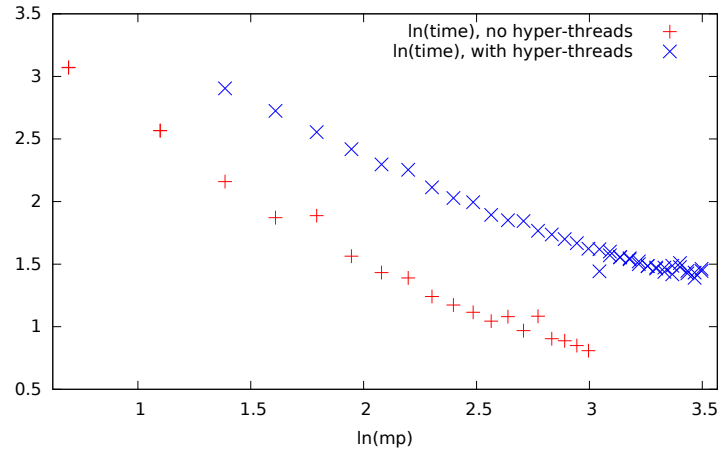

Figure 2: Gamma MLE script: log of calculation time versus log of number of processes, with and without hyper-threading