3. Advanced Multithreading

Chuanhai Liu

Department of Statistics, Purdue University

2016

Table of Contents

4.1 More SupR additions

To make everything simple but not simpler, we have the following examples of the current more experiments.

Simplicity Handy tools.

Caching The original R caching seemed to have to be changed entirely for SupR. A few functions are currently provided at the user level for a new scheme.

Threading More built-in tools/functions for understanding and application of multi-threading.

4.1 More SupR additions: two new simple functions

- `address(x)`, a primitive function that returns the internal memory location of the object 'x'; See also `.Internal(inspect(x))` and `pryr::address(x)` R functions.

  > This function is useful, for example, for checking if multithreads are indeed synchronized on intended common objects.

- `c(...) <- expr` is introduced as a convenient tool to do multi-assignment. The name-value matching is done sequentially. Whenever there are at least two unbound names in the argument list, the unassigned value is divided into a pair of what we can call 'head' and 'tail'. For example, `c(x, y, z) <- 1:10` will be equivalent to the three expressions: `x <- 1; x <- 2; x <- 3:10`

4.2 Object caching

Caching is important for both object-sharing and efficiency in the multi-threading context. Caching in R is implemented for the `'.GlobalEnv'` environment with an internal hash table environment.

The current SupR experimentation is based on a new mechanism, which we call "local caching".

To some extent, the current SupR implementation is user-transparent.

Here we discuss four levels of caching introduced in SupR.

4.2 Object caching: system-level

In the current experiment, only named objects with locked bindings are cached.
Internally, caching is implemented with "hashtables", which uses vectors of `pairlist`.
You can see what have been cached for an environment with `cacheenv(env)` function.

Recall that the global environment `.GlobalEnv` is shared by all threads and that the search mechanism for installed packages remains the same as the that in R.

```
> cacheenv(.GlobalEnv)
<environment:  0x??????>
> ls(cacheenv(.GlobalEnv))
...
```

4.2 Object caching: thread-level

By default caching is enabled for threads. The following code shows how to take a look at what have been cached for threads.

```
> new.thread(ls(cache(threadenv())), join=T)
...
```

4.2 Object caching: evaluation-level

In addition, the user can control caching at the R programming level with the three functions, `import()`, `cache()`, and `nocache()`.

- `import(..., from)` imports objects from the 'from' environment to the current evaluation environment. See also the `importIntoEnv()` function.

```
> new.thread({
        import(var, sd, from=as.environment("package:stats"))
        ls.str()
    }, join=T)
```

Note: In the future, imported objects may be changed to be placed in the cache hashtable if it exists...

4.2 Object caching: evaluation-level

- `cache(TRUE)` turns local/environment caching on.
- `cache(FALSE)` turns local/environment caching off.

Examples: TO DO

4.2 Object caching: evaluation-level

- `nocache(...)` prevents the named objects in the parent environments from being cached in the local/evaluation environment, where the `nocache()` call is made.

  For each argument, the current implementation creates special and fancy internal active binding objects in the current evaluation environment.

Examples: TO DO

4.2 Object caching: thread attach and detach tools

In R, you can insert/remove environment-like objects into/from the search path from `.GlobalEnv` with the the `attach()` and `detach()` functions.

In SupR, with the `thread.attach()` and `thread.detach()` functions, you can do the same on the search path from `.ThreadEnv` to `.GlobalEnv`.

Examples: TO DO

4.2 Object caching: evaluation-level

Here we consider a few more examples.

Examples: TO DO

More examples are given in the Multithreaded-EM application.

4.2 Object caching: cluster-level

This topic will be discussed along with that on cluster computing.

4.3 Computing with local variables

While experimental, SupR introduces a concept of computing with local variables. It eliminates the need for caching and can run much faster, especially for computing with a large number of iterations.

To use it, you simply place a `define()` function call to declare the variables that follow but within the same block, i.e., up to the nearest close brace `}`. Also, such definitions can be nested. However, this new idea has not fully implemented yet.

Type `?cache` and see the online example of using `define()` function.

Internally, evaluation of the `define()` function call amounts to "compile" the expressions that follow. This does something different from the R compiler, which hasn't been enabled yet in SupR.

4.3 Computing with local variables

Like R, everything of SupR should be made transparent to users.

"Show me the code!"

is perhaps a good way to tell everything.

For this, a simple function, `show.code()`, is provided and subject to improvement.

Again, type `?cache` and see the online example of using `define()` function.

This could be an approach to make SupR run fast!

4.4 Signal handling

The C-level implementation of most SupR thread functions, such as `pthread_join()`, uses the corresponding functions in the pthread library.

But it is implemented in a user-friendly way so that user can cancel `thread_join`, by type Ctr-C to interrupt the thread joining process. More discussion follows shortly.

For good reasons, SupR doesn't use the `pthread_cancel()` function in the pthread library. It implements its own `cancel.thread()` function via signal handling.

It can be particularly problematic in the case the "main" thread joins threads that can be potentially involved in deadlocks in synchronization operations.

Here we discuss how to use the Interrupt signal (SIGINT) at the user level. An SIGINT signal is generated when you type `Ctr-C`, i.e. hit the C key while holding the Control key down.

4.4 Signal handling: user interruption and default actions

Signal handling is a tricky problem in interactive computing environments, especially in computing with multithreads. When typing `Ctr-C`, for example, the system delivers a SIGINT to one of the running threads.

For example, when I used `Scala`, a nice "interactive-JAVA" language, to run Spark, I often felt frustrated with user-interruptions. Handling SIGINT is not that trivial as we may think. Killing threads is easy, but you have to make sure that the interactive system remains to work in the way that you would expect. expected.

In SupR, SIGINT is redirected to the thread that you directly interact with. For convenience, let's call it the *user thread*. Typically this is the "main" thread you start with and normally use for interactive data analysis.

You may like the experimental "double-interruption", i.e., type Ctr-C twice quickly. For example, it can be used as a simple way to kill all running threads.

4.4 Signal handling: user interruption and default actions

What action the user thread will take to respond to SIGINT depends on what it is doing when receiving the signal. Here are the current default reactions to SIGINT.

- Reading commands from the key board: *restart* reading.
- Joining (waiting for) some thread to exit: *cancel* the joining/waiting process itself but not the thread it is waiting for.
- Waiting for jobs to finish (in cluster computing): *stop* waiting and let the job continue to run in somewhat a detached mode.
- Evaluating expressions: *stop* the evaluation and return.
- Waiting for synchronization: this is treated as a special case of evaluating expressions.

4.5 More on thread.join

The C-level implementation uses the `pthread_join()` function in the pthread library. Its man page says:

> The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.
> ...

Here are some relevant points:

- All threads are created initially as joinable. This means that a thread_join call must be invoked to release system resources allocated for created threads.

  > However, SupR handles thread_join operation automatically if you don't call it before they return from `join.thread()` function calls.

- The values of expressions evaluated by threads are returned to the `thread.join()` calls or kept in the `.ThreadSet` environment, which is located in `.GlobalEnv`. Such results will be automatically removed after they are accessed through the `thread.join()` function.

4.5 More on thread.join: Examples

There are functions that you may find useful:

- `alive.threads()` displays all alive threads.
- `all.threads()` displays all alive threads and previous threads that are waiting for the user to handle their returned results. Currently, `all.threads()` simply returns the `.ThreadSet` environment object. Some user interface functions can be easily developed later.

See on-line examples.

A remark: the concept of thread group may be considered later.

4.6 More on cancel.thread

SupR doesn't use the pthread library function `pthread_cancel()`. It implements its own `cancel.thread()` function via signal handling.

This low-level treatment allows the R-side cleanups to be handled normally. For example, certain semaphore-based synchronizations can be taken care of easily and safely.

TO DO: Introduce the concept of thread groups so that when some threads in a group raise an error, appropriate actions can be taken on all threads in the same group

4.6 The interrupt and is.interrupted functions

These are two more functions for which the C-level implementation is signal-based.

The action taken by the target thread in responding to `interrupt(target_thread)` can be ignored but can be handled programmingly in R with the test `is.interrupted()` function.

Two remarks:

- Threads can call `interrupt(target.thread)` any time.
- The target threads may not react immediately.
- You use the pair `interrupt(target.thread)` and `is.interrupt()` to synchronize two threads running concurrently. For this, you make use of the `thread.sleep(time)` function to put one thread into sleep and use another to wake it up with the `interrupt(target.thread)` function.

More examples :

TO DO if necessary: A similar user interface for the `cancel.thread()` but with restrictions in that it must terminate as soon as possible.

4.8.1 Working with the browser function

The `browser()` and `debug()` functions are useful tools for debugging R code. In SupR with multithreads, they can be used in two ways.

- One way is to use them with the `interact()` function, as is shown in the last example.
- The other way is to insert calls to these functions in your R code.

```
> new.thread(x = 1:4; browser(); y = sin(x), start = T)
>
```

MORE TO DO ?

4.8.2 Understanding synchronization with the `sync.inspect` function

A simple version of this function is available. This function summarizes the basic synchronization info by two data.frame objects.

- One data.frame gives a detail list of objects on which operations of relevant threads are synchronized.
- The other is a Object x Thread table showing how some threads may depend on other some threads.

You may also use two related functions: `sync.cleanup` and `sync.remove`. See the online documentation.

4.8 Additional functions

More functions/command that can be useful:

- thread.exit
- thread.time
- where command, experimental.

See more such function by typing, for example, `?thread` and `?cache`.

4.9 Exercises