# 3. Multithreading

Chuanhai Liu

Department of Statistics, Purdue University

2016

## Table of Contents

References for the Parallel Computing section:

1. Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp/
2. POSIX Threads Programming, https://computing.llnl.gov/tutorials/pthreads/
3. The JAVA Programming Language, ...

## 3.1 Simple SupR additions

In addition to functions in the `namespace:base` of R, SupR introduces and modifies a set basic functions, which include

- the `implicit(class, fun)` function to make it possible to use objects as function objects,
- functions to handle so-called *iterator* objects, and

## 3.1 Simple SupR additions: the `implicit` function

`implicit(class, fun=NULL)`

'implicit' associates non-function objects of the specified class with an anonymous function so that the associated function can be called with non-function objects of the class.

'implicit(class, fun = NULL)' returns the function associated with the specified class.

'implicit(class, fun = "remove")' removes the function associated with the specified class.

'implicit()' returns the list of the associated classes.

```
> # ?  implicit
> # implicit()
> 100(1000)
100000
> A = matrix(1:10, 5)
> t(A)(A)
     [,1] [,2]
[1,]   55  130
[2,]  130  330
>
```

## 3.1 Simple SupR additions: iterators

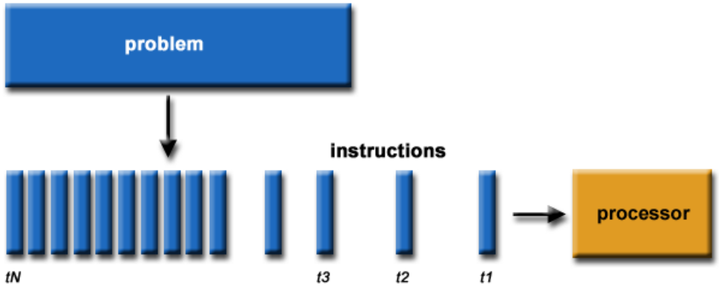
Iterators are used to represent a collection of values that can be obtained sequentially. They consist of an environment and two functions, named 'has.next' and 'get.next'.

The '$has.next()' call, evaluated in the enviroment, returns TRUE if there are more values available and FALSE otherwise. After a '$has.next()' call returns TRUE, the '$get.next()' call, evaluated in the same enviroment, returns the next available value.

```
> # ?  iterator
> iter = as.iterator(1:3)
> while(has.next(iter)) print(get.next(iter))
[1] 1
[1] 2
[1] 3
> implicit("iterator", function(iter, fun) {
        while(has.next(iter))
        fun(get.next(iter))
    })
> # apply a function to each of the components of 1:4
> as.iterator(1:4)(function(x) print(x+10))
[1] 11
[1] 12
[1] 13
[1] 14
>
```

## 3.2 Parallel Computing: serial computation

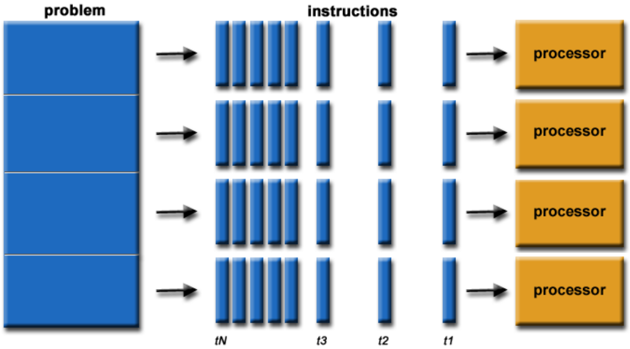
- A problem is broken into a discrete series of (computer) instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time



## 3.2 Parallel Computing: the parallelism

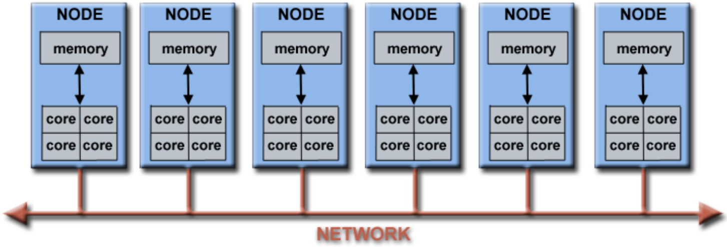
In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

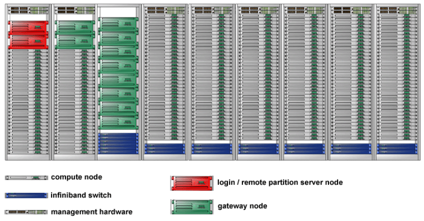
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



## 3.2 Parallel Computing: computer clusters

Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



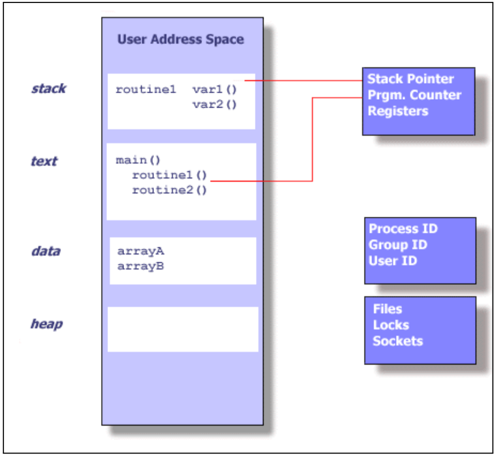


The schematic of a typical parallel computer cluster.

## 3.2 Parallel Computing: processes

A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

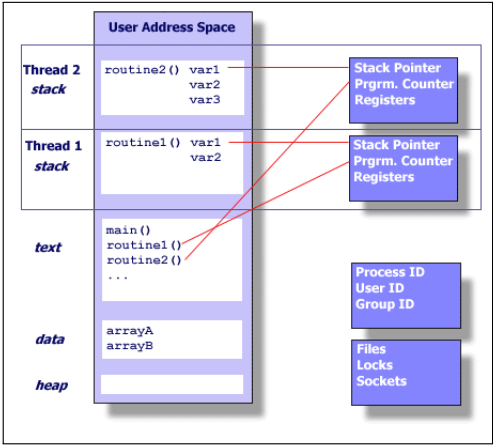


## 3.2 Parallel Computing: lightweight process (LWPs)

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.

This independent flow of control is accomplished because a thread maintains its own:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.


User Address Space
Thread 2 stack
routine2() var1 var2 var3
Stack Pointer Prgrm. Counter Registers
Thread 1 stack
routine1() var1 var2
Stack Pointer Prgrm. Counter Registers
text
main() routine1() routine2() ...
data
arrayA arrayB
heap
Process ID User ID Group ID
Files Locks Sockets

## 3.2 Parallel Computing: lightweight process (LWPs)

In summary, in the UNIX environment a thread:

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies - or something similar
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

Because threads within the same process share resources:

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

## 3.2 Parallel Computing: why threads?

**Light Weight** When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

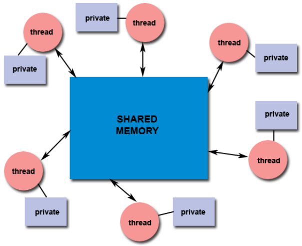
**Efficient Communications/Data Exchange** The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.

**Other Common Reasons**

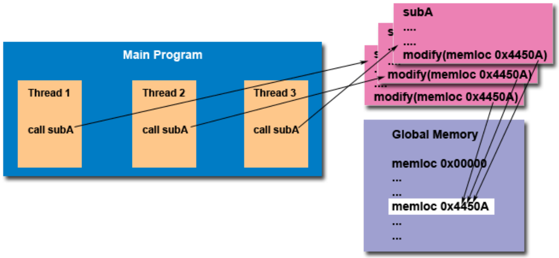
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
  - Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
  - Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
  - Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.
- Another good example is a modern operating system, which makes extensive use of threads.

## 3.2 Parallel Computing: thread-safeness

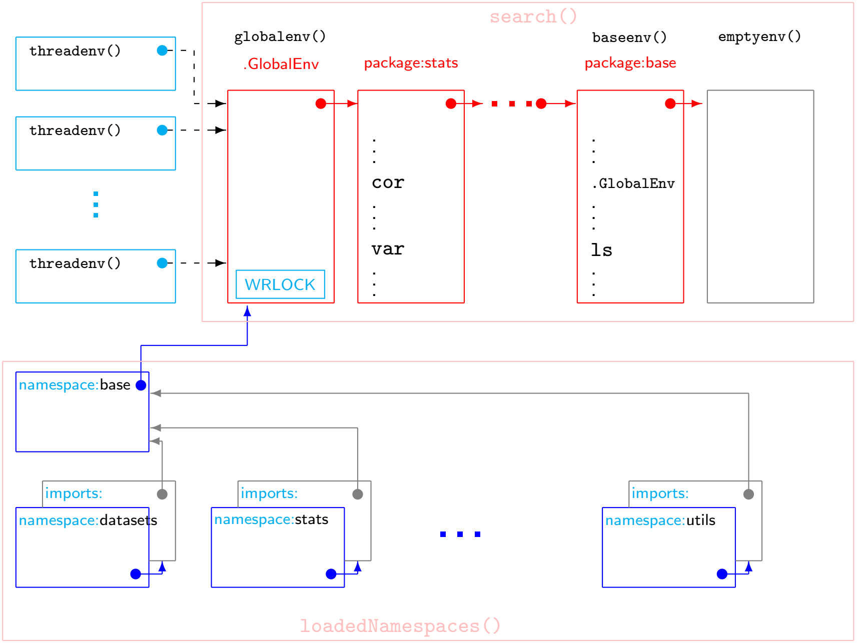
- Shared Memory Model:
  - All threads have access to the same global, shared memory
  - Threads also have their own private data
  - Programmers are responsible for synchronizing access (protecting) globally shared data.



- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.



## 3.3 Multithreading: SupR thread "name spaces"



## 3.3 Multithreading: SupR thread functions

SupR's multithreading and concurrency framework is similar to that of JAVA. It consists of a set of functions, partitioned into the following three categories.

Thread basics
: You can create threads to evaluate expressions with the `new.thread()`, `start.thread()`, `join.thread()`, `cancel.thread()`, `current.thread()`, `threadenv()`, and `thread.info()` functions.

Interruption
: You can put a thread into sleep with a `thread.sleep(time)` call in the thread run script, and during the sleep of the running thread, you can wake up the thread with the `interrupt()` call from a different thread. After thread is waken up due to either timeout expiration or thread interruption, the function `is.interrupted()` call tests if the thread was interrupted.

Synchronization
: You can synchronize thread evaluations with the `sync.eval()`, `set.synchronized()`, `is.synchronized()`, `wait()`, and `notify()` functions

For more information, type the `?thread` command.

## 3.3 Multithreading: create and run threads

The `new.thread()` function creates a new thread of stack size = 'stacksize' to evaluate 'expr' (or '.expr' when 'expr' is missing) in the 'env' environment if 'env' is not globalenv() and in new.env(parent='env') otherwise.

If 'start' or 'join' is TRUE, this thread will be started immediately. In this case, if 'join' is also TRUE, the 'new.thread' call waits for the new thread to terminate.

Otherwise, a separate 'start.thread' call must be used to start it. In this case, the optional 'join' argument and a separate 'join.thread' call can be used for the calling thread to wait for the thread specified by 'name' to terminate.

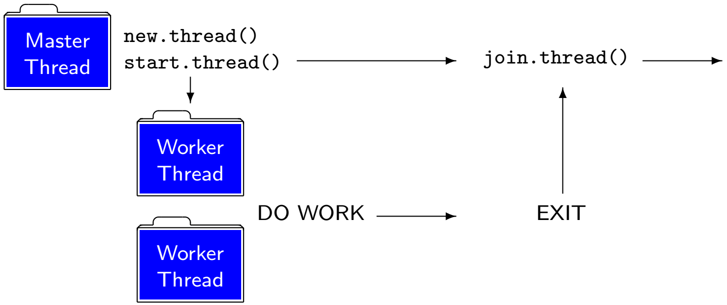
```
> new.thread(1+2, start=TRUE, join=TRUE)
[1] 3
> th1 = new.thread({thread.sleep(20); 1+2})
> th2 = new.thread({thread.sleep(15); 1+3})
> start.thread(c(th1, th2), join = rep(T, 2))
$thread_3
[1] 3

$thread_4
[1] 4

>
```

## 3.3 Multithreading: thread joining

"Joining" is one way to accomplish synchronization between threads. For example:



A schematic of thread "joining".

## 3.3 Multithreading: thread local environments

Objects in evironments on R's name spaces and environments on the search path starting from globalenv() are shared by all threads.

The `current.thread()`, `threadenv()`, and `thread.info()` functions provide limited information on running threads.

A 'current.thread' call returns the thread name of the calling thread. 'threadenv' returns the thread local environment, which typically has globalenv() on its search path.

```
> a <- new.env()
> new.thread({ThreadEnv <- threadenv(); Thread <- current.thread()},
          env=a, start=T, join=T)
[1] "thread_6"
> ls.str(a, all=T)
T_MainExpression :  length 3 { ThreadEnv <- threadenv(); Thread <-
current.thread() }
Thread :  chr "thread_6"
ThreadEnv :  <environment:  0x27899e8>
> identical(a, a$ThreadEnv)
[1] TRUE
>
```

## 3.3 Multithreading: thread information

The 'thread.info()' call returns a data.frame object showing the existing threads with limited gc information [FIXME]. If 'name' is not NULL, 'thread.info' requests the thread specified by 'name' to produce its evaluation backtrace.

```
> th1 = new.thread({thread.sleep(20); 1+2})
> th2 = new.thread({thread.sleep(15); 1+3})
> thread.info()
       name int_sig LWP_id gc_int mutex thread_id
1     __MAIN__      NA   1692      0     0     0x69c
2   thread_10      NA   5300      0     0    0x14b4
3   thread_11      NA   5301      0     0    0x14b5
> start.thread(c(th1, th2), rep(T, 2))
...
> thread.info()
       name int_sig LWP_id gc_int mutex thread_id
1     __MAIN__      NA   1692      0     0     0x69c
> new.thread(thread.sleep(30), start=T)
[1] "thread_13"
> unlist(thread.info("thread_13")) # what is it doing?
[1] "thread.sleep(30)"
[2] "new.thread(expr, env, stacksize, as.logical(start),
as.logical(join))"
[3] "new.thread(thread.sleep(30), start = T)"
>
```

## 3.3 Multithreading: thread interruption

'thread.sleep' puts the calling thread into sleep for a specified time period. But the sleep can be interrupted by other threads with an 'interrupt' call. When woken up, with an 'is.interrupted' call the thread can test if it was woken up due to a thread interruption.

```
> a = new.env()
> th = new.thread(env = a, start = T, {
        thread.sleep(30)
        interrupted <- is.interrupted()
    })
> interrupt(th)
> ls.str(a)
...
interrupted :  logi TRUE
> a = new.thread(start = T, {
        thread.sleep(30)
        interrupted <- is.interrupted()
        environment()
    }, join = T)
> ls.str(a)
...
interrupted :  logi FALSE
> # replace thread.sleep(30) with st <- system.time(thread.sleep(30))
>
```

## 3.4 Concurrency: the wait-and-notify mechanism

The pair of functions 'wait' and 'notify' can be used to provide a way for threads to communicate with each other when necessary. The pair also provides a mechanism for object sharing.

wait 'wait(x, timeout)' causes the current thread to wait until the timeout expires in the 'timeout'>0 case or another thread invokes the notify() function with the same object specified by 'x'.

notify 'notify(x, all)' wakes up a single thread or all threads (specified with all=TRUE) waiting on the waiting list of the object specified by 'x'.

```
> x = 1:10
> expr1 = quote({
     sync.eval(x, {
           cat(current.thread(), "OKAY 1-1", date(), "\n")
           wait(x)
           cat(current.thread(), "OKAY 1-2", date(), "\n")
         })
       paste(current.thread(), "returned_value is 1", sep=" :  ")
   })
>
```

## 3.4 Concurrency: the wait-and-notify mechanism [Cont'd]

```
> expr2 = quote({
      sync.eval(x, {
            cat(current.thread(), "OKAY 2-1", date(), "\n")
            wait(x)
            cat(current.thread(), "OKAY 2-2", date(), "\n")
          })
        paste(current.thread(), "returned_value is 2", sep=" :  ")
    })

> th1 = new.thread(.expr = expr1, start=T)
> th2 = new.thread(.expr = expr2, start=T)
thread_11 OKAY 1-1 Fri May 13 17:20:28 2016
thread_12 OKAY 2-1 Fri May 13 17:20:30 2016
thread_12 OKAY 2-2 Fri May 13 17:20:35 2016
thread_11 OKAY 1-2 Fri May 13 17:20:35 2016
>
```

## 3.4 Concurrency: synchronized evaluation

'sync.eval', 'set.synchronized', and 'is.synchronized' provide a way to synchronize evaluations on specified objects. With the object 'x' synchronized, 'sync.eval(x, expr, .expr, env=parent.frame())' evaluates the expression 'expr' in the environment 'env'.

'set.synchronized(x, value)' and 'is.synchronized(x)' set and test for the synchronization state of the object specified by 'x'.

Currently, synchronization is only implemented for binding and accessing objects in synchronized environment objects and for calling synchronized functions.

```
> implicit(class(""), paste0)
> threads = character(10)
> A = TRUE
> for(i in 1:length(threads))
    threads[i] = new.thread(sync.eval(A, {
        print(current.thread()(" printed this line at ")(date()))
        thread.sleep(5);
        print(current.thread()(" printed this line at ")(date()))
    }))
> start.thread(threads, join=rep(TRUE, length(threads)))
[1] "thread_1 printed this line at Fri May 13 17:00:22 2016"
[1] "thread_1 printed this line at Fri May 13 17:00:27 2016"
[1] "thread_7 printed this line at Fri May 13 17:00:27 2016"
...
>
```

## 3.4 Concurrency: synchronized evaluation [Cont'd]

```
> # Synchronization with synchronized objects
> is.synchronized(globalenv())
[1] TRUE
> is.synchronized(baseenv())
[1] FALSE
> foo = function(time) {
    cat(current.thread(),"is going to sleep for",time,"seconds\n")
    thread.sleep(time)
    cat(current.thread(), "continues\n")
  }

> set.synchronized(foo, TRUE)
> th1 = new.thread(foo(10), start=T)
> th2 = new.thread(foo(10), start=T)
thread_13 is going to sleep for 10 seconds
thread_13 continues
thread_14 is going to sleep for 10 seconds
thread_14 continues
> > set.synchronized(foo, FALSE)
> th1 = new.thread(foo(10), start=T)
> th2 = new.thread(foo(10), start=T)
...
```

## 3.5 Lazy evaluation

Lazy evaluation is also known as delayed evaluation. There are two types of evaluation in R that are lazy: promise objects and active bindings.

**Promise objects:** The value of these objects represent (unevaluated) expressions. These expressions are only evaluated once and when they are used the first time.

**Active bindings:** Active bindings refer to a mechanism of dynamically setting and getting values of bindings through symbol/name objects. The mechanism is internally implemented with a function, called *binding function*.

| | |
|---|---|
| Getting | When the binding value is accessed, the binding function is called internally with no arguments to produce a value. |
| Setting | When a binding value is assigned, the binding function is called internally with the value as its argument. The binding function returns a value as the value for this operation. |

## 3.5 Lazy evaluation: promise objects

When such objects are shared, e.g., in the shared name spaces and on the search path, synchronized evaluation must be enforced.

But the `set.promiseSynchronized()` and `is.promiseSynchronized()` functions are available for the user to control synchronized evaluation of other promise objects.

```
> # Defining and accessing promise objects
> delayedAssign("x", sync.inspect("x"))
> x
> new.thread({
        delayedAssign("x", sync.inspect("x"))
        unlist(x)
    }, join=TRUE)
marked active
 FALSE  FALSE
> new.thread({
        delayedAssign("x", sync.inspect("x"))
        setPromiseSynchronized("x", TRUE)
        unlist(x)
    }, join=TRUE)
marked active
  TRUE   TRUE
>
```

## 3.5 Lazy evaluation: active bindings

By default, evaluations of binding functions are not synchronized. Since binding functions are function objects, synchronized evaluation of these functions can be enabled by the user. Internal binding functions can be used with

```
> foo <- local( {
        x <- 1
        function(v) {
            if (!missing(v)) x <<- v
            x
        }
    })
> makeActiveBinding("fred", foo, .GlobalEnv)
> is.synchronized(get.activeBinding("fred"))
[1] FALSE
> .Internal(address(foo))==.Internal(address(get.activeBinding("fred")))
[1] TRUE
> set.synchronized(get.activeBinding("fred"))
[1] TRUE
> fred
[1] 1
> fred <- 2
[1] 2
>
```

## 3.6 New developments

More new developments are necessary and under consideration. To list a few, we have

**Thread caching:** Naming and caching are two fundamental problems in software development. This is particularly true in cluster computing.

**C-level global and static variables:** Currently, no all problems associated with such variables have been taken care of. This is to discussed further when addressing the C-interface topic.

**Loading and attaching packages:** Synchronization will be enforced for such operations as it has something to do with the default shared objects.

## 3.7 Exercises

1. Implement an "iterator" by making use of active binding. Hint: `?makeActiveBinding`.
2. ...