Some possible directions for the R engine

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

July 22, 2010

Introduction

This talk outlines a few possible directions for development in the core R engine over the next 12 to 18 months:

- Taking advantage of multiple cores for vectorized operations and simple matrix operations.
- Byte code compilation of R code.
- Further developments in error handling.
- Increasing the limit on the size of vector data objects.

Why Parallel Computing?

- Most modern computers feature two or more processor cores.
- It is expected that tens of cores will be available soon.
- A common question:

  How can I make R use more than one core for my computation?

  - There are many easy answers.
  - But this is the wrong question.
- The right question:

  How can we take advantage of having more than one core to get our computations to run faster?

  This is harder to answer.

Some Approaches to Parallel Computing

Two possible approaches:

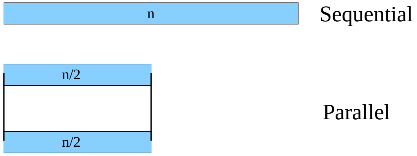
- Implicit parallelization:
    - automatic, no user action needed
- Explicit parallelization:
    - uses some form of annotation to specify parallelism

I will focus on implicit parallelization of

- basic vectorized math functions
- basic matrix operations (e.g. colSums)
- BLAS

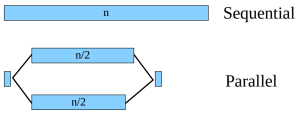
Parallelizing Vectorized Operations
An Idealized View

- Basic idea for computing f(x[1:n]) on a two-processor system:
  - Run two worker threads.
  - Place half the computation on each thread.
- Ideally this would produce a two-fold speed up.



Parallelizing Vectorized Operations
A More Realistic View

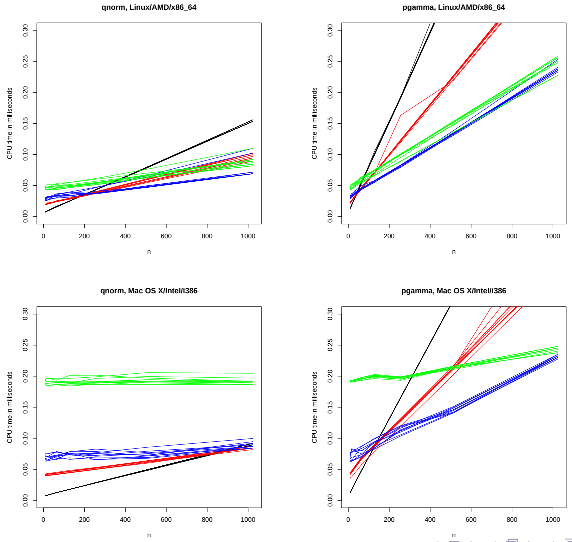
- Reality is a bit different:



- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, . . .)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is needed.
  - For basic arithmetic operations $n \approx 30000$ may be needed.
- Careful tuning to ensure improvement will be needed.
- Some aspects will depend on architecture and OS.

Parallelizing Vectorized Operations

Some Experimental Results



Parallelizing Vectorized Operations
Some Experimental Results

Some observations:

- Times are roughly linear in vector length.
- Intercepts on a given platform are roughly the same for all functions.
- If the slope for P processors is s_P, then at least for $P = 2$ and $P = 4$,

$$s_P \approx s_1/P$$

- Relative slopes of functions seem roughly independent of OS/architecture.

A simple calibration strategy:

- Compute relative slopes once, or average across several setups.
- For each OS/architecture combination compute the intercepts.

The appropriate time to run calibration code is still open.

Parallelizing Vectorized Operations
Implementation

- Need to use threads
- One possibility: using raw `pthreads`
- Better choice: use Open MP
- Open MP consists of
    - compiler directives (#pragma statements in C)
    - a runtime support library
- Most commercial compilers support Open MP.
- Current gcc versions support Open MP; newer ones do a better job.
- MinGW for Win32 also supports Open MP; an additional `pthreads` download is needed.
- Support for Win64 is not yet clear.

Parallelizing Vectorized Operations
Implementation

- Basic loop for a one-argument function:

```
#pragma omp parallel for if (P > 0) num_threads(P) \
        default(shared) private(i) reduction(&&:naflag)
    for (i = 0; i < n; i++) {
        double ai = a[i];
        MATH1_LOOP_BODY(y[i], f(ai), ai, naflag);
    }
```

- Steps in converting to Open MP:
  - check f is thread-safe; modify if not
  - rewrite loop to work with the Open MP directive
  - test without Open MP, then enable Open MP

Parallelizing Vectorized Operations
Implementation

- Some things that are not thread-safe:
    - use of global variables
    - R memory allocation
    - signaling warnings and errors
    - user interrupt checking
    - creating internationalized messages (calls to gettext)
- Random number generation is also problematic.
- Functions in `nmath` that have not been parallelized yet:
    - Bessel functions (partially done)
    - Wilcoxon, signed rank functions (may not make sense)
    - random number generators

Parallelizing Vectorized Operations

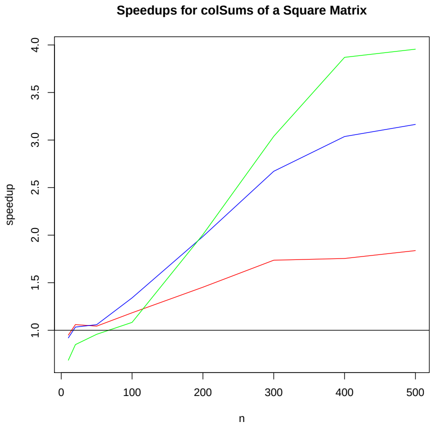
Availability

- Package `pnmath` is available at

  `http://www.stat.uiowa.edu/~luke/R/experimental/`
- This requires a version of `gcc` that
  - supports Open MP
  - allows dlopen to be used on libgomp.so
- A version using just `pthreads` is available in `pnmath0`.
- Loading these packages replaces builtin operations by parallelized ones.
- For Linux, Mac OS X predetermined intercept calibrations are used.
- For other platforms a calibration test is run at package load time.
- The calibration can be run manually by calling calibratePnmath
- Hopefully we will be able to include this in R soon.

Parallelizing Simple Matrix Operations

Very preliminary results for colSums on an 8-core Linux machine:



Parallelizing Simple Matrix Operations

Some issues to consider:

- Again using too many processor cores for small problems can slow the computation down.
- colSums can be parallelized by rows or columns:
  - Handling groups of columns in parallel produces identical results to a sequential version.
  - Handling groups of rows in parallel changes numerical results slightly (floating point addition is not associative).
- rowSums is slightly more complex since locality of reference (column major storage) need to be taken into account.
- A number of other basic operations can be handled similarly.
- Simple uses of apply and sweep might also be handled along these lines.

Using a Parallel BLAS

- Most core linear algebra calculations use the Basic Linear Algebra Subroutines library (BLAS).
- R has supported using a custom BLAS implementation for some time.
- Both Intel and AMD provide sequential and threaded accelerated BLAS implementations.
- Atlas and Goto's BLAS also come in sequential and threaded versions.
- Very preliminary testing suggests that the Intel threaded BLAS works well for small and large matrices.
- Anecdotal evidence, that may no longer apply, suggests that this may not be true of some other threaded BLAS implementations.
- More testing is needed.

Byte Code Compilation

- The current R implementation
  - parses code into a *parse tree* when the code is read
  - evaluates code by interpreting the parse trees.
- Most low level languages (e.g. C, Fortran) compile their source code to native machine code.
- Some intermediate level languages (e.g. Java, C#) and many scripting languages (e.g. Perl, Python) compile to a simpler language called byte code.

Byte Code Compilation

- Byte code is the machine code for a virtual machine.
- Virtual machine code can then be interpreted by a simpler, more efficient interpreter.
- Virtual machines, and their machine code, are usually specific to the languages they are designed to support.
- Various strategies for further compiling byte code to native machine code are also sometimes used.

Byte Code Compilation

- Efforts to add byte code compilation to R have been underway for some time.
- Current R implementations include a byte code interpreter, and a preliminary compiler is available from my web page.
- The current compiler and virtual machine produce good improvements in a number of cases.
- However, better results should be possible with a new virtual machine design.
- This redesign is currently in progress.

A Simple Example

Here is an example that example has appeared in discussions of language performance in the R mailing lists:

```
p1 <- function(x) {
    for (i in seq_along(x))
        x[i] <- x[i] + 1
    x
}
```

A few comments:

- There is no good reason to write code like this in R; the expression

  ```
  x + 1
  ```

  is more general, simpler, and much, much faster.
- This sort of code does appear often in benchmark discussions.
- Quantitative improvements obtained for such benchmarks do not usually carry over to real code.
- Qualitative results can be useful.

Some Performance Results

Some timings from

```
    x <- rep(1, 1e7)
    system.time(p1(x))
```

on an `x86_64` Ubuntu laptop:

Method	Time	Speedup
Interpreted	32.730	1.0
Byte compiled	9.530	3.4
Ra	1.647	19.9
Experimental	1.128	29.0
`x+1`	0.119	275.0

Ra is Stephen Milborrow's Ra/jit system.

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special opcodes for most `SPECIAL`s, many `BUILTIN`s
  - inlines simple `.Internal` calls: `dnorm(y, 2, 3)` is replaced by `.Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))`
  - special opcodes for many `.Internal`s
- Currently the compiler has to be called explicitly to compile single functions or files of code.
- An alternative design would have code compiled automatically at parse time or at time of first use.

Compiler Operation

The new virtual machine will support additional optimizations, including

- avoiding the allocation of intermediate values when possible
- more efficient variable lookup mechanisms
- more efficient function calls
- possibly improved handling of lazy evaluation

Future Directions

Some possible directions that may also be explored:

- Partial evaluation when some arguments are constants
- Intra-procedural optimizations and inlining
- Run-time specialization and threaded code generation
- Vectorized opcodes
- Declarations (sealing, scalars, types, strictness)
- Advice to programmer on possible inefficiencies
- Machine code generation using LLVM or other toolkits
- Replacing the interpreter entirely

Further Developments in Error Handling

Errors can occur in many situations, for example

- extreme random number values may result in square roots of negative numbers in simulations
- code depending on network connections can fail due to network issues

Other situations may be suspect but not necessarily always errors; these can be signaled as warnings.

Further Developments in Error Handling

Being able to catch and continue after errors is very useful, and R has a rich set of mechamisms for this:

- Code executed in a tryCatch expression will jump back to the level of the tryCatch and continue with the handler defined there.
- The older try function is a special case.
- Code executed in a withCallingHandlers expression will call the specified handler from within the error signaler; this allows errors or warnings to be ignored or a debugger to be entered.
- A useful idiom is

  ```
  withCallingHandlers(<<some suspect code>>,
                      error = function(e) recover())
  ```

  will enter the debugger provided by recover if an error occurs in the suspect code.

Further Developments in Error Handling

Some additional features:

- Handlers can chose to deal with an error, have an error ignored, or defer to another handler.
- Continuation points, called *restarts* can be established that allow a computation to continue after an error.
- These restarts can be invoked with arguments to provide new data to use.
- For example, an optimizer can provide a restart that accepts an alternative function value to use if the computation of the optimized function generates an error.

A few areas are currently lacking:

- Documentation
- A hierarchy of error and warning classes that can be used for computing appropriate responses.‘

Error Handling Documentation

- The error handling mechanism is currently documented in the help pages.
- More extensive documentation is needed, with extended examples showing the use of the different mechanisms in different contexts
- The first step of course is writing such a document.
- The best way to make such a document accessible is not clear; perhaps as a vignette in one of the core packages.

Error and Warning Classes

- Currently the core C code raises errors in nearly 2,400 places.
- All these are currently signaled as errors of class as simpleError.
- Error handling code should be able to respond to different errors in different ways.
- But there is currently no way to distinguish among different errors other than by reading the error messages, and these vary based on the language locale used.
- In addition, to be able to handle errors appropriately, handling code needs to have relevant data.
- For example, a handler for a failed `http` request would need the URL and the error code.

Error and Warning Classes

- What is needed is
  - A careful study of the error situations currently signaled
  - To classify these into an appropriate hierarchy
  - To design the classes in the hierarchy to contain appropriate information relevant to the error.
- A number of other languages use an object oriented approach to error handling and can serve as examples.
- The approach will have to be incremental, perhaps starting with input/output and networking related errors.

Increasing the Limit on Vector Object Size

- Currently The total number of elements in a vector cannot exceed $2^{31} - 1 = 2,147,483,647$
- This is fairly large, but is becoming an issue with larger data sets with many variables on 64-bit platforms.
- Can this limit be raised without breaking too much existing R code and requiring the rewriting of too much C code?

Some Considerations

- The current limit represents the largest possible 32-bit signed integer.
- For all practical purposes on all current architectures the C `int` type and the Fortran `integer` type are 32 bit signed integers.
- The R memory manager is easy enough to change.
- Finding all the other places in the C code implementing R where `int` would need to be changed to a wider type, and making sure it is not changed where it should not be, is hard.
- External code used by R is also a problem, in particular the BLAS.
- Reliance on BLAS may limit individual dimensions to $2^{31} - 1$ but not necessarily restrict total abject size to that value.

Changing the R Integer Data Type

- Changes would be needed to the R integer data type and/or to return values of functions that currently return integer values, such a the length function.
- Changes to the R integer type would create some issues with saved workspaces.
- Using a wider integer type on 64 bit platforms than on 32 bit ones is possible but creates issues with porting workspaces.
- Using a 64 bit integer on all platforms may be a better choice.
- One possibility is using double precision floating point numbers to internally represent R integers as well as reals. This would have the advantage of allowing better handling of integer NA values.
- If the integer representation is changed, a possible direction to explore is whether *smaller* integer types could be added (one byte and two byte, for example).

Status of Explorations

- Exploration of this issue is still at a very preliminary stage.
- The constraints and limitations are not yet fully understood.
- The magnitude of the effort is also not yet clear.
- The next year or so will likely see some significant effort to understand the constrains and the options.
- Once that point has been reached, directions for moving forward, and time frames for doing so, should become clearer.

Summary

- This talk has outlined several areas I believe are important and to which I hope I can make some contributions during the next 12 to 18 months.
- The R development model is quite distributed: other R developers are working on a wide range of other areas.
- Fortunately conflicts are rare and the different efforts, so far at least, have merged together quite very successfully.