Some Developments for the R Engine

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

November 10, 2011

Introduction

- R is a language for data analysis and graphics.
- R is based on the S language developed by John Chambers and others at Bell Labs.
- R is widely used in the field of statistics and beyond, especially in university environments.
- R has become the primary framework for developing and making available new statistical methodology.
- Many (over 3,000) extension packages are available through CRAN or similar repositories.

Introduction

- R is a language for data analysis and graphics.
- R is based on the S language developed by John Chambers and others at Bell Labs.
- R is widely used in the field of statistics and beyond, especially in university environments.
- R has become the primary framework for developing and making available new statistical methodology.
- Many (over 3,000) extension packages are available through CRAN or similar repositories.

Introduction

- R is a language for data analysis and graphics.
- R is based on the S language developed by John Chambers and others at Bell Labs.
- R is widely used in the field of statistics and beyond, especially in university environments.
- R has become the primary framework for developing and making available new statistical methodology.
- Many (over 3,000) extension packages are available through CRAN or similar repositories.

Introduction

- R is a language for data analysis and graphics.
- R is based on the S language developed by John Chambers and others at Bell Labs.
- R is widely used in the field of statistics and beyond, especially in university environments.
- R has become the primary framework for developing and making available new statistical methodology.
- Many (over 3,000) extension packages are available through CRAN or similar repositories.

Introduction

- R is a language for data analysis and graphics.
- R is based on the S language developed by John Chambers and others at Bell Labs.
- R is widely used in the field of statistics and beyond, especially in university environments.
- R has become the primary framework for developing and making available new statistical methodology.
- Many (over 3,000) extension packages are available through CRAN or similar repositories.

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka in the early 1990's for a Macintosh computer lab at U. of Auckland, NZ.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Seth Falcon	Kurt Hornik
Stefano Iacus	Ross Ihaka	Friedrich Leisch
Uwe Ligges	Thomas Lumley	Martin Maechler
Duncan Murdoch	Paul Murrell	Martyn Plummer
Brian Ripley	Deepayan Sarkar	Duncan Temple Lang
Luke Tierney	Simon Urbanek	

- Strong community support through
  - contributed extension packages
  - mailing lists and blogs
  - contributed documentation and task views

Introduction

This talk outlines some recent developments in the core R engine I have worked on and developments I expect to work on over the next 12 to 18 months:

- Byte code compilation of R code.
- Taking advantage of multiple cores for
  - basic vectorized operations
  - simple matrix operations.
- Increasing the limit on the size of vector data objects.

Introduction

This talk outlines some recent developments in the core R engine I have worked on and developments I expect to work on over the next 12 to 18 months:

- Byte code compilation of R code.
- Taking advantage of multiple cores for
  - basic vectorized operations
  - simple matrix operations.
- Increasing the limit on the size of vector data objects.

Introduction

This talk outlines some recent developments in the core R engine I have worked on and developments I expect to work on over the next 12 to 18 months:

- Byte code compilation of R code.
- Taking advantage of multiple cores for
  - basic vectorized operations
  - simple matrix operations.
- Increasing the limit on the size of vector data objects.

Introduction

This talk outlines some recent developments in the core R engine I have worked on and developments I expect to work on over the next 12 to 18 months:

- Byte code compilation of R code.
- Taking advantage of multiple cores for
  - basic vectorized operations
  - simple matrix operations.
- Increasing the limit on the size of vector data objects.

Introduction

This talk outlines some recent developments in the core R engine I have worked on and developments I expect to work on over the next 12 to 18 months:

- Byte code compilation of R code.
- Taking advantage of multiple cores for
  - basic vectorized operations
  - simple matrix operations.
- Increasing the limit on the size of vector data objects.

Introduction

This talk outlines some recent developments in the core R engine I have worked on and developments I expect to work on over the next 12 to 18 months:

- Byte code compilation of R code.
- Taking advantage of multiple cores for
  - basic vectorized operations
  - simple matrix operations.
- Increasing the limit on the size of vector data objects.

Byte Code Compilation

- The standard R evaluation mechanism
  - 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

- The standard R evaluation mechanism
  - 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

- The standard R evaluation mechanism
  - 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

- The standard R evaluation mechanism
  - 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

- The standard R evaluation mechanism
  - 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

- 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

- 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

- 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.
- The first release of the compiler occurred with R 2.13.0 this spring.
- Some improvements in the virtual machine interpreter were released with R 2.14.0 this fall.
- The current compiler and virtual machine produce good improvements in a number of cases.
- Better results should be possible with some improvements to the virtual machine and are currently being explored.

Byte Code Compilation

- Efforts to add byte code compilation to R have been underway for some time.
- The first release of the compiler occurred with R 2.13.0 this spring.
- Some improvements in the virtual machine interpreter were released with R 2.14.0 this fall.
- The current compiler and virtual machine produce good improvements in a number of cases.
- Better results should be possible with some improvements to the virtual machine and are currently being explored.

Byte Code Compilation

- Efforts to add byte code compilation to R have been underway for some time.
- The first release of the compiler occurred with R 2.13.0 this spring.
- Some improvements in the virtual machine interpreter were released with R 2.14.0 this fall.
- The current compiler and virtual machine produce good improvements in a number of cases.
- Better results should be possible with some improvements to the virtual machine and are currently being explored.

Byte Code Compilation

- Efforts to add byte code compilation to R have been underway for some time.
- The first release of the compiler occurred with R 2.13.0 this spring.
- Some improvements in the virtual machine interpreter were released with R 2.14.0 this fall.
- The current compiler and virtual machine produce good improvements in a number of cases.
- Better results should be possible with some improvements to the virtual machine and are currently being explored.

Byte Code Compilation

- Efforts to add byte code compilation to R have been underway for some time.
- The first release of the compiler occurred with R 2.13.0 this spring.
- Some improvements in the virtual machine interpreter were released with R 2.14.0 this fall.
- The current compiler and virtual machine produce good improvements in a number of cases.
- Better results should be possible with some improvements to the virtual machine and are currently being explored.

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - cmpfun compiles a function
  - cmpfile compiles a file to be loaded by loadcmp
- It is also possible to have package code compiled when a package is installed.
  - Use --byte-compile when installing, or specify the ByteCompile option in the DESCRIPTION file.
  - R 2.14.0 by default compiles R code in all base and recommended packages.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiler before they are run

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most `SPECIAL`s, many `BUILTIN`s
  - inlining simple `.Internal` calls: e.g.

    ```
    dnorm(y, 2, 3)
    ```

    is replaced by

    ```
    .Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))
    ```
  - special instructions for many `.Internal`s
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most SPECIALs, many BUILTINs
  - inlining simple .Internal calls: e.g.

    ```
    dnorm(y, 2, 3)
    ```

    is replaced by

    ```
    .Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))
    ```
  - special instructions for many .Internals
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most `SPECIAL`s, many `BUILTIN`s
  - inlining simple `.Internal` calls: e.g.

    ```
    dnorm(y, 2, 3)
    ```

    is replaced by

    ```
    .Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))
    ```
  - special instructions for many `.Internal`s
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most `SPECIAL`s, many `BUILTIN`s
  - inlining simple `.Internal` calls: e.g.

    ```
    dnorm(y, 2, 3)
    ```

    is replaced by

    ```
    .Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))
    ```

  - special instructions for many `.Internal`s
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most `SPECIAL`s, many `BUILTIN`s
  - inlining simple `.Internal` calls: e.g.

    ```
    dnorm(y, 2, 3)
    ```

    is replaced by

    ```
    .Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))
    ```
  - special instructions for many `.Internal`s
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most `SPECIAL`s, many `BUILTIN`s
  - inlining simple `.Internal` calls: e.g.

    ```
    dnorm(y, 2, 3)
    ```

    is replaced by

    ```
    .Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))
    ```
  - special instructions for many `.Internal`s
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.

A Simple Example

R Code

```
f <- function(x) {
    s <- 0.0
    for (y in x)
        s <- s + y
    s
}
```

VM Assembly Code

```
    LDCONST 0.0
    SETVAR s
    POP
    GETVAR x
    STARTFOR y L2
L1: GETVAR s
    GETVAR y
    ADD
    SETVAR s
    POP
    STEPFOR L1
L2: ENDFOR
    POP
    GETVAR s
    RETURN
```

A Simple Example

R Code

```
f <- function(x) {
    s <- 0.0
    for (y in x)
        s <- s + y
    s
}
```

VM Assembly Code

```
    LDCONST 0.0
    SETVAR s
    POP
    GETVAR x
    STARTFOR y L2
L1: GETVAR s
    GETVAR y
    ADD
    SETVAR s
    POP
    STEPFOR L1
L2: ENDFOR
    POP
    GETVAR s
    RETURN
```

Some Performance Results

Timings for some simple benchmarks on an `x86_64` Ubuntu laptop:

Benchmark	*Interp.*	*Comp.*	*Speedup*	*Exper.*	*Speedup*
`p1`	32.19	7.98	4.0	1.47	21.9
`sum`	6.72	1.86	3.6	0.59	11.4
`conv`	14.48	4.30	3.4	0.81	17.9
`rem`	56.82	23.68	2.4	4.77	11.9

Interp., *Comp.* are for the development version of R

- includes some variable lookup improvements for compiled code

Exper.: experimental version using

- separate instructions for vector, matrix indexing
- typed stack to avoid allocating intermediate scalar values

Timings for some simple benchmarks on an x86_64 Ubuntu laptop:

Benchmark	*Interp.*	*Comp.*	*Speedup*	*Exper.*	*Speedup*
p1	32.19	7.98	4.0	1.47	21.9
sum	6.72	1.86	3.6	0.59	11.4
conv	14.48	4.30	3.4	0.81	17.9
rem	56.82	23.68	2.4	4.77	11.9

Interp., *Comp.* are for the development version of R

- includes some variable lookup improvements for compiled code

Exper.: experimental version using

- separate instructions for vector, matrix indexing
- typed stack to avoid allocating intermediate scalar values

Some Performance Results

Timings for some simple benchmarks on an `x86_64` Ubuntu laptop:

Benchmark	*Interp.*	*Comp.*	*Speedup*	*Exper.*	*Speedup*
`p1`	32.19	7.98	4.0	1.47	21.9
`sum`	6.72	1.86	3.6	0.59	11.4
`conv`	14.48	4.30	3.4	0.81	17.9
`rem`	56.82	23.68	2.4	4.77	11.9

Interp., *Comp.* are for the development version of R

- includes some variable lookup improvements for compiled code

Exper.: experimental version using

- separate instructions for vector, matrix indexing
- typed stack to avoid allocating intermediate scalar values

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
    - avoiding the allocation of intermediate values when possible
    - more efficient variable lookup mechanisms
    - more efficient function calls
    - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
    - Partial evaluation when some arguments are constants
    - Intra-procedural optimizations and inlining
    - Declarations (sealing, scalars, types, strictness)
    - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

Some promising preliminary results are available.

- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
    - avoiding the allocation of intermediate values when possible
    - more efficient variable lookup mechanisms
    - more efficient function calls
    - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
    - Partial evaluation when some arguments are constants
    - Intra-procedural optimizations and inlining
    - Declarations (sealing, scalars, types, strictness)
    - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

  Some promising preliminary results are available.
- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches
- Collaborations with computer scientists are being explored.
- Maintainability is a key concern.

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

Parallelizing Vector and Matrix Operations

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

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

Some Approaches to Parallel Computing

Two possible approaches:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.
- Implicit parallelization:
  - automatic, no user action needed

I will focus on implicit parallelization of

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

- 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

An Idealized View

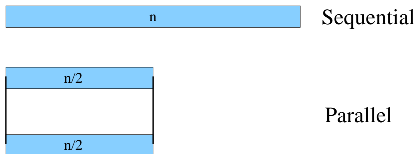
- Basic idea for computing f(x[1:n]) on a two-processor system:
    - Run two worker threads.
    - Place half the computation on each thread.
- Ideally this would produce a two-fold speed up.

- 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

An Idealized View

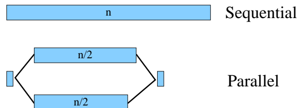
- Basic idea for computing f(x[1:n]) on a two-processor system:
  - Run two worker threads.
  - Place half the computation on each thread.
- Ideally this would produce a two-fold speed up.



Parallelizing Vectorized Operations

A More Realistic View

- Reality is a bit different:

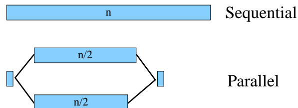


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

A More Realistic View

- Reality is a bit different:

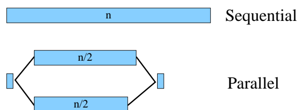


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, . . .)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

A More Realistic View

- Reality is a bit different:

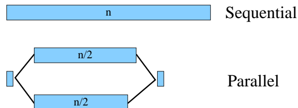


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, . . .)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

A More Realistic View

- Reality is a bit different:

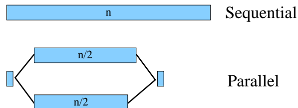


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

A More Realistic View

- Reality is a bit different:

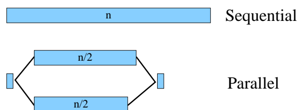


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

A More Realistic View

- Reality is a bit different:

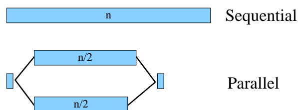


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

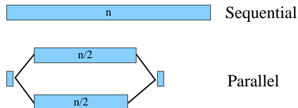
A More Realistic View

- 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.

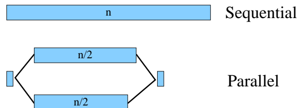
- Reality is a bit different:



- 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
A More Realistic View

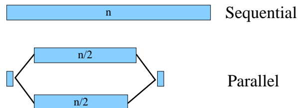
- Reality is a bit different:



- 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
A More Realistic View

- Reality is a bit different:

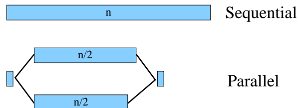


- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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

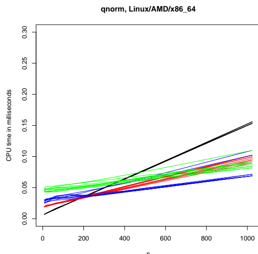
A More Realistic View

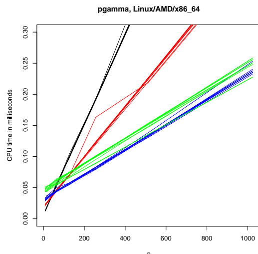
- Reality is a bit different:

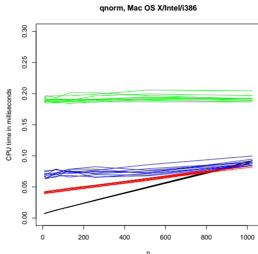


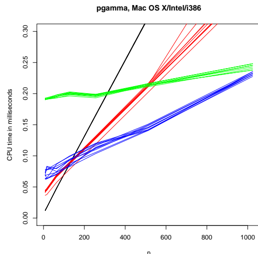
- There is
  - synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - uneven workload
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. qbeta, $n \approx 10$ may be large enough.
  - For some, e.g. qnorm, $n \approx 1000$ is 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









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.

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.

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.

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.

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.

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.

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.

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.

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.

- 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 now available also and should be in the toolchain soon.

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 now available also and should be in the toolchain soon.

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 now available also and should be in the toolchain soon.

- 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 now available also and should be in the toolchain soon.

- 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 now available also and should be in the toolchain soon.

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 now available also and should be in the toolchain soon.

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 now available also and should be in the toolchain soon.

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 now available also and should be in the toolchain soon.

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 now available also and should be in the toolchain soon.

- 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 now available also and should be in the toolchain soon.

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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
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

- 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
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

- 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

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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 Vectorized Operations
Availability

- Package `pnmath` is available at

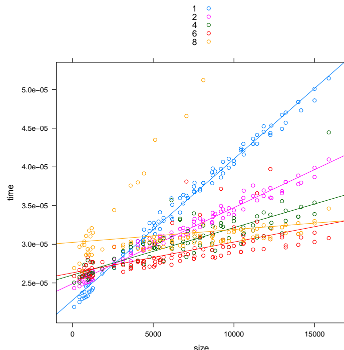
  `http://www.stat.uiowa.edu/~luke/R/experimental/`
- This requires a version of `gcc` that
  - supports Open MP
  - allows dlopen to be used on libgomp.so
- A version using just `pthreads` is available in `pnmath0`.
- Loading these packages replaces builtin operations by parallelized ones.
- For Linux, 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.

- 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

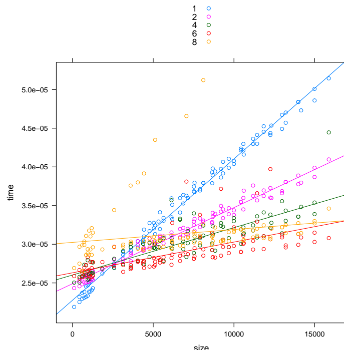
- Very preliminary results for colSums on an 8-core Linux machine:



- Preliminary results on OS X indicate cutoff levels may be much higher.
- Part of the implementation work was done by Xiao Yang.

Parallelizing Simple Matrix Operations

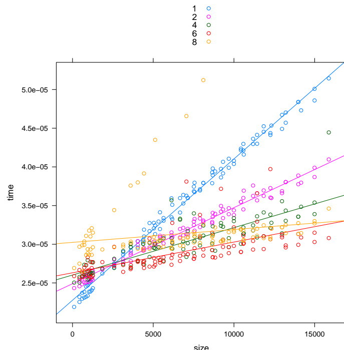
- Very preliminary results for colSums on an 8-core Linux machine:



- Preliminary results on OS X indicate cutoff levels may be much higher.
- Part of the implementation work was done by Xiao Yang.

Parallelizing Simple Matrix Operations

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



- Preliminary results on OS X indicate cutoff levels may be much higher.
- Part of the implementation work was done by Xiao Yang.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

Parallelization and Compilation

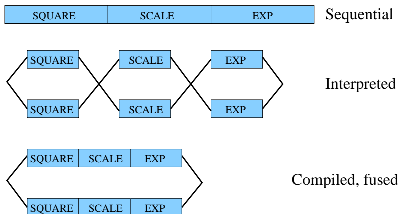
- Compilation may be useful for parallelizing vector operations:
  - Many vector operations occur in compound expressions, like `exp(-0.5*x^2)`
  - A compiler may be able to fuse these operations:
- This will allow gains from parallelizing compound operations on shorter vectors.

Parallelization and Compilation

- Compilation may be useful for parallelizing vector operations:
  - Many vector operations occur in compound expressions, like `exp(-0.5*x^2)`
  - A compiler may be able to fuse these operations:
- This will allow gains from parallelizing compound operations on shorter vectors.

Parallelization and Compilation

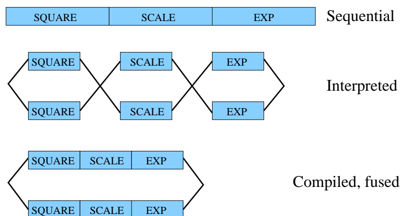
- Compilation may be useful for parallelizing vector operations:
  - Many vector operations occur in compound expressions, like `exp(-0.5*x^2)`
  - A compiler may be able to fuse these operations:



- This will allow gains from parallelizing compound operations on shorter vectors.

Parallelization and Compilation

- Compilation may be useful for parallelizing vector operations:
  - Many vector operations occur in compound expressions, like `exp(-0.5*x^2)`
  - A compiler may be able to fuse these operations:



- This will allow gains from parallelizing compound operations on shorter vectors.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Big Data

- Automated data acquisition in science and commerce is producing huge amounts of data.
- *Big Data* is a hot topic in the popular and trade press.
- Some categories:
  - fit into memory
  - fit on one machines disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to smaller large data problems on machines with enough memory.

Limit on Vector Object Size

- Currently The total number of elements in a vector cannot exceed $2^{31} - 1 = 2,147,483,647$
- This limit represents the largest possible 32-bit signed integer.
- For numeric (double precision) data this means the largest possible vector is about 16 GB.
- 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?

Limit on Vector Object Size

- Currently The total number of elements in a vector cannot exceed $2^{31} - 1 = 2,147,483,647$
- This limit represents the largest possible 32-bit signed integer.
- For numeric (double precision) data this means the largest possible vector is about 16 GB.
- 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?

Limit on Vector Object Size

- Currently The total number of elements in a vector cannot exceed $2^{31} - 1 = 2,147,483,647$
- This limit represents the largest possible 32-bit signed integer.
- For numeric (double precision) data this means the largest possible vector is about 16 GB.
- 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?

Limit on Vector Object Size

- Currently The total number of elements in a vector cannot exceed $2^{31} - 1 = 2,147,483,647$
- This limit represents the largest possible 32-bit signed integer.
- For numeric (double precision) data this means the largest possible vector is about 16 GB.
- 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?

Limit on Vector Object Size

- Currently The total number of elements in a vector cannot exceed $2^{31} - 1 = 2,147,483,647$
- This limit represents the largest possible 32-bit signed integer.
- For numeric (double precision) data this means the largest possible vector is about 16 GB.
- 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

- For all practical purposes on all current architectures the C `int` type and the FORTRAN `integer` type are 32 bit signed integers.
- The R source code uses C `int` or FORTRAN `integer` types in many places that would need to be changed to a wider type.
- 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.

Some Considerations

- For all practical purposes on all current architectures the C `int` type and the FORTRAN `integer` type are 32 bit signed integers.
- The R source code uses C `int` or FORTRAN `integer` types in many places that would need to be changed to a wider type.
- 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.

Some Considerations

- For all practical purposes on all current architectures the C `int` type and the FORTRAN `integer` type are 32 bit signed integers.
- The R source code uses C `int` or FORTRAN `integer` types in many places that would need to be changed to a wider type.
- 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.

Some Considerations

- For all practical purposes on all current architectures the C `int` type and the FORTRAN `integer` type are 32 bit signed integers.
- The R source code uses C `int` or FORTRAN `integer` types in many places that would need to be changed to a wider type.
- 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.

Some Considerations

- For all practical purposes on all current architectures the C `int` type and the FORTRAN `integer` type are 32 bit signed integers.
- The R source code uses C `int` or FORTRAN `integer` types in many places that would need to be changed to a wider type.
- 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.

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
    - Change length fields in internal headers to support longer vectors.
    - Change standard field accessors to signal an error if long vectors are used.
    - Add new accessors that allow long vectors.
    - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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).

Some Possible Directions

- A possible strategy:
  - Change length fields in internal headers to support longer vectors.
  - Change standard field accessors to signal an error if long vectors are used.
  - Add new accessors that allow long vectors.
  - Gradually introduce long vector support into the R internals.
- The initial header change will require recompiling all C code but no further code changes.
- After the header change, support for large vectors can be introduced incrementally in R itself and in packages.
- It may eventually be necessary to introduce a long integer data type or change the integer type from 32 to 64 bits.
- It may also be sufficient to store larger integers as double precision floating point numbers.
- 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

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Status of Explorations

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Status of Explorations

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Status of Explorations

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Status of Explorations

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Status of Explorations

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Status of Explorations

- Initial experiments are promising.
- In some cases just changing data types will be sufficient.
- In other cases more may be needed, such as
  - ability to interrupt computations
  - more stable numerical algorithms
- Some more experimentation is needed.
- If successful, the header changes may occur within the next few weeks.

Summary

- This talk has outlined several areas I believe are important and to which I hope I can make some contributions during the near future.
- 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.

Summary

- This talk has outlined several areas I believe are important and to which I hope I can make some contributions during the near future.
- 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.

Summary

- This talk has outlined several areas I believe are important and to which I hope I can make some contributions during the near future.
- 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.