Some new developments for the R engine

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

June 24, 2012
Introduction

- R is a language for data analysis and graphics.
- Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.
- Now developed and maintained by a distributed group of 20 people.
- 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.
- Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.
- Now developed and maintained by a distributed group of 20 people.
- 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.
- Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.
- Now developed and maintained by a distributed group of 20 people.
- 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.
- Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.
- Now developed and maintained by a distributed group of 20 people.
- 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.
R is a language for data analysis and graphics.

- Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.
- Now developed and maintained by a distributed group of 20 people.
- 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.
R is a language for data analysis and graphics.

Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.

Now developed and maintained by a distributed group of 20 people.

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.
R is a language for data analysis and graphics.

Originally developed by Ross Ihaka and Robert Gentleman at University of Auckland, New Zealand.

Now developed and maintained by a distributed group of 20 people.

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.
This talk outlines three areas of development in the core R engine:

- New large vector support.
- Fine-grained parallelization of vector and matrix operations.
- Byte code compilation of R code.
This talk outlines three areas of development in the core R engine:

- New large vector support.
- Fine-grained parallelization of vector and matrix operations.
- Byte code compilation of R code.
This talk outlines three areas of development in the core R engine:
- New large vector support.
- Fine-grained parallelization of vector and matrix operations.
- Byte code compilation of R code.
This talk outlines three areas of development in the core R engine:

- New large vector support.
- Fine-grained parallelization of vector and matrix operations.
- Byte code compilation of R code.
Large Vector Support

- **Big Data** is a hot topic in this session.
- Some categories:
  - fit into memory
  - fit on one machine's 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 handle smaller large data problems on machines with enough memory.
**Big Data** is a hot topic in this session. Some categories:

- fit into memory
- fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
• *Big Data* is a hot topic in this session.

• Some categories:
  • fit into memory
  • fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
Big Data is a hot topic in this session.

Some categories:
- fit into memory
- fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
Big Data is a hot topic in this session.

Some categories:
- fit into memory
- fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
Big Data is a hot topic in this session.

Some categories:
- fit into memory
- fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
• *Big Data* is a hot topic in this session.

• Some categories:
  • fit into memory
  • fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
Big Data is a hot topic in this session.

Some categories:
- fit into memory
- fit on one machine’s 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 handle smaller large data problems on machines with enough memory.
Large Vector Support

Initial Objectives

- Through R 2.15.1 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.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Through R 2.15.1 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.

We need a way to raise this limit that meets several goals:
- avoid having to rewrite too much of R itself
- avoid requiring package authors to rewrite too much C code
- avoid having existing compiled C code fail if possible
- allow incrementally adding support for procedures where it makes sense

For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Large Vector Support

Initial Objectives

- Through R 2.15.1 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.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Through R 2.15.1 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.

We need a way to raise this limit that meets several goals:

- avoid having to rewrite too much of R itself
- avoid requiring package authors to rewrite too much C code
- avoid having existing compiled C code fail if possible
- allow incrementally adding support for procedures where it makes sense

For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Large Vector Support

Initial Objectives

- Through R 2.15.1 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.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep \(2^{31} - 1\) limit on matrix rows and columns.
Large Vector Support

Initial Objectives

- Through R 2.15.1 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.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Large Vector Support

Initial Objectives

- Through R 2.15.1 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.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Through R 2.15.1 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.

We need a way to raise this limit that meets several goals:

- avoid having to rewrite too much of R itself
- avoid requiring package authors to rewrite too much C code
- avoid having existing compiled C code fail if possible
- allow incrementally adding support for procedures where it makes sense

For now, keep \(2^{31} - 1\) limit on matrix rows and columns.
Large Vector Support

Initial Objectives

- Through R 2.15.1 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.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep $2^{31} - 1$ limit on matrix rows and columns.
Through R 2.15.1 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.

We need a way to raise this limit that meets several goals:

- avoid having to rewrite too much of R itself
- avoid requiring package authors to rewrite too much C code
- avoid having existing compiled C code fail if possible
- allow incrementally adding support for procedures where it makes sense

For now, keep $2^{31} - 1$ limit on matrix rows and columns.
C level changes:
- Preserve existing memory layout
- Use special marker in length field to identify long vectors
- LENGTH accessor (returning int) signals an error for long vectors
- Long vector aware code uses XLENGTH to return R_xlen_t.

R code should not need to be changed:
- double precision indices can be used for subsetting
- length will return double for long vectors
- .C and .Fortran will signal errors for long vectors.

A document describing how to add long vector support to a package should be available soon.
C level changes:
- Preserve existing memory layout
- Use special marker in length field to identify long vectors
- LENGTH accessor (returning int) signals an error for long vectors
- Long vector aware code uses XLENGTH to return R_xlen_t.

R code should not need to be changed:
- double precision indices can be used for subsetting
- length will return double for long vectors
- .C and .Fortran will signal errors for long vectors.

A document describing how to add long vector support to a package should be available soon.
Large Vector Support
Current Design

C level changes:
- Preserve existing memory layout
- Use special marker in length field to identify long vectors
- LENGTH accessor (returning int) signals an error for long vectors
- Long vector aware code uses XLENGTH to return R_xlen_t.

R code should not need to be changed:
- double precision indices can be used for subsetting
- length will return double for long vectors
- .C and .Fortran will signal errors for long vectors.

A document describing how to add long vector support to a package should be available soon.
Large Vector Support
Current Design

- **C level changes:**
  - Preserve existing memory layout
  - Use special marker in length field to identify long vectors
  - `LENGTH` accessor (returning `int`) signals an error for long vectors
  - Long vector aware code uses `XLENGTH` to return `R_xlen_t`.

- **R code should not need to be changed:**
  - Double precision indices can be used for subsetting
  - `length` will return double for long vectors
  - `.C` and `.Fortran` will signal errors for long vectors.

- A document describing how to add long vector support to a package should be available soon.
C level changes:
- Preserve existing memory layout
- Use special marker in length field to identify long vectors
- LENGTH accessor (returning int) signals an error for long vectors
- Long vector aware code uses XLENGTH to return R_xlen_t.

R code should not need to be changed:
- double precision indices can be used for subsetting
- length will return double for long vectors
- .C and .Fortran will signal errors for long vectors.

A document describing how to add long vector support to a package should be available soon.
Large Vector Support
Current Design

- **C level changes:**
  - Preserve existing memory layout
  - Use special marker in length field to identify long vectors
  - `LENGTH` accessor (returning `int`) signals an error for long vectors
  - Long vector aware code uses `XLENGTH` to return `R_xlen_t`.

- **R code should not need to be changed:**
  - Double precision indices can be used for subsetting
  - `length` will return double for long vectors
  - `.C` and `.Fortran` will signal errors for long vectors.

- A document describing how to add long vector support to a package should be available soon.
Large Vector Support
Current Design

- **C level changes:**
  - Preserve existing memory layout
  - Use special marker in length field to identify long vectors
  - `LENGTH` accessor (returning `int`) signals an error for long vectors
  - Long vector aware code uses `XLENGTH` to return `R_xlen_t`.

- **R code should not need to be changed:**
  - double precision indices can be used for subsetting
    - `length` will return double for long vectors
    - `.C` and `.Fortran` will signal errors for long vectors.

- A document describing how to add long vector support to a package should be available soon.
Large Vector Support
Current Design

- **C level changes:**
  - Preserve existing memory layout
  - Use special marker in length field to identify long vectors
  - `LENGTH` accessor (returning `int`) signals an error for long vectors
  - Long vector aware code uses `XLENGTH` to return `R_xlen_t`.

- **R code should not need to be changed:**
  - double precision indices can be used for subsetting
  - `length` will return double for long vectors
  - `.C` and `.Fortran` will signal errors for long vectors.

- A document describing how to add long vector support to a package should be available soon.
Large Vector Support
Current Design

- **C level changes:**
  - Preserve existing memory layout
  - Use special marker in length field to identify long vectors
  - `LENGTH` accessor (returning `int`) signals an error for long vectors
  - Long vector aware code uses `XLENGTH` to return `R_xlen_t`.

- **R code should not need to be changed:**
  - double precision indices can be used for subsetting
  - `length` will return double for long vectors
  - `.C` and `.Fortran` will signal errors for long vectors.

- A document describing how to add long vector support to a package should be available soon.
C level changes:
- Preserve existing memory layout
- Use special marker in length field to identify long vectors
- `LENGTH` accessor (returning `int`) signals an error for long vectors
- Long vector aware code uses `XLENGTH` to return `R_xlen_t`.

R code should not need to be changed:
- double precision indices can be used for subsetting
- `length` will return double for long vectors
- `.C` and `.Fortran` will signal errors for long vectors.

A document describing how to add long vector support to a package should be available soon.
A number of internal functions now support long vectors.

Some statistical functions with long vector support:

- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function dist can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:

- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:

- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:

- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:

- random number generators
  - mean
  - sort
  - fivenum
  - lm.fit
  - glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:

- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:

- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function dist can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:

- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function dist can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function dist can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- colSums, colMeans
- rowSums, rowMeans
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- `colSums`, `colMeans`
- `rowSums`, `rowMeans`
A number of internal functions now support long vectors.

Some statistical functions with long vector support:
- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function `dist` can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:
- `colSums`, `colMeans`
- `rowSums`, `rowMeans`
A number of internal functions now support long vectors.

Some statistical functions with long vector support:

- random number generators
- mean
- sort
- fivenum
- lm.fit
- glm.fit

The function dist can handle more than $2^{16}$ observations by returning a long vector result.

Many matrix and array functions already support large arrays:

- colSums, colMeans
- rowSums, rowMeans
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider:

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:
- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider
- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed \(2^{31} - 1\) as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:

- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider

- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Converting existing methods to support large vectors is fairly straightforward, however:
- more numerically stable algorithms may be needed
- faster/parallel algorithms may be needed
- the ability to interrupt computations may become important
- statistical usefulness may not scale to larger data

The size where these issues become relevant is likely much lower!

Future work will consider
- whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
- possibly adding 8 and 16 bit integer types
- arithmetic and overflow issues that these raise
- whether to allow numbers of rows and columns in matrices to exceed $2^{31} - 1$ as well
Parallelizing Vector and Matrix Operations

Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages *snow*, *multicore*, *parallel*

- Implicit parallelization:
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. colSums)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- **Explicit parallelization:**
  - uses some form of annotation to specify parallelism
  - packages such as multicore, parallel

- **Implicit parallelization:**
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. colSums)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages snow, multicore, parallel.

- Implicit parallelization:
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. colSums)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

**Two ways to take advantage of multiple cores:**

- **Explicit parallelization:**
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.

- **Implicit parallelization:**
  - automatic, no user action needed.

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. `colSums`)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- **Explicit parallelization:**
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.

- **Implicit parallelization:**
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. `colSums`)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- Explicit parallelization:
  - uses some form of annotation to specify parallelism
  - packages snow, multicore, parallel.

- Implicit parallelization:
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. colSums)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- **Explicit parallelization:**
  - uses some form of annotation to specify parallelism
  - packages `snow`, ` multicore`, `parallel`.

- **Implicit parallelization:**
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. `colSums`)
Most modern computers feature two or more processor cores.
It is expected that tens of cores will be available soon.
Two ways to take advantage of multiple cores:
  - Explicit parallelization:
    - uses some form of annotation to specify parallelism
    - packages snow, multicore, parallel.
  - Implicit parallelization:
    - automatic, no user action needed

Implicit parallelization is particularly suited to
  - basic vectorized math functions
  - basic matrix operations (e.g. colSums)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

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

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. `colSums`)
Parallelizing Vector and Matrix Operations

- Most modern computers feature two or more processor cores.
- It is expected that tens of cores will be available soon.
- Two ways to take advantage of multiple cores:
  - Explicit parallelization:
    - uses some form of annotation to specify parallelism
    - packages snow, multicore, parallel.
  - Implicit parallelization:
    - automatic, no user action needed
- Implicit parallelization is particularly suited to
  - basic vectorized math functions
  - basic matrix operations (e.g. colSums)
Most modern computers feature two or more processor cores.

It is expected that tens of cores will be available soon.

Two ways to take advantage of multiple cores:

- **Explicit parallelization:**
  - uses some form of annotation to specify parallelism
  - packages `snow`, `multicore`, `parallel`.

- **Implicit parallelization:**
  - automatic, no user action needed

Implicit parallelization is particularly suited to

- basic vectorized math functions
- basic matrix operations (e.g. `colSums`)
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, \ldots)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ is large enough.
- For some functions, e.g. $\text{qbeta}$, $n \approx 10$ may be large enough.
- For some, e.g. $\text{qnorm}$, $n \approx 1000$ is needed.
- For basic arithmetic operations $n \approx 30000$ may be needed.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Parallelizing Vector and Matrix Operations

Performance Implications

- Basic idea for a $P$-core system:
  - run $P$ worker threads
  - place $1/P$ of the work on each thread

- Idealized view: this produces a $P$-fold speedup.

- Actual speedup is less:
  - there is synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - actual workloads are uneven

- Result: parallel code can be slower!

- Parallelizing will only pay off if data size $n$ is large enough.
  - For some functions, e.g. $q\text{beta}$, $n \approx 10$ may be large enough.
  - For some, e.g. $q\text{norm}$, $n \approx 1000$ is needed.
  - For basic arithmetic operations $n \approx 30000$ may be needed.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, . . .)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ is large enough.
- For some functions, e.g. $q\text{beta}$, $n \approx 10$ may be large enough.
- For some, e.g. $q\text{norm}$, $n \approx 1000$ is needed.
- For basic arithmetic operations $n \approx 30000$ may be needed.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, . . .)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ is large enough.
- For some functions, e.g. $q\text{beta}$, $n \approx 10$ may be large enough.
- For some, e.g. $q\text{norm}$, $n \approx 1000$ is needed.
- For basic arithmetic operations $n \approx 30000$ may be needed.
Basic idea for a \( P \)-core system:
- run \( P \) worker threads
- place \( 1/P \) of the work on each thread

Idealized view: this produces a \( P \)-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, \ldots)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size \( n \) is large enough.
- For some functions, e.g. \( \text{qbeta} \), \( n \approx 10 \) may be large enough.
- For some, e.g. \( \text{qnorm} \), \( n \approx 1000 \) is needed.
- For basic arithmetic operations \( n \approx 30000 \) may be needed.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, . . .)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ 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.
Basic idea for a $P$-core system:
- run $P$ worker threads
- place $1/P$ of the work on each thread

Idealized view: this produces a $P$-fold speedup.

Actual speedup is less:
- there is synchronization overhead
- sequential code and use of shared resources (memory, bus, ...)
- actual workloads are uneven

Result: parallel code can be slower!

Parallelizing will only pay off if data size $n$ is large enough.
- For some functions, e.g. \texttt{qbeta}, $n \approx 10$ may be large enough.
- For some, e.g. \texttt{qnorm}, $n \approx 1000$ is needed.
- For basic arithmetic operations $n \approx 30000$ may be needed.
OpenMP provides a convenient way to implement parallelism at the C/FORTRAN level.

Good performance of the synchronization barrier is critical for fine-grained parallelization.

On Linux/gcc OpenMP performance is very good.

On Mac OS X and Windows gcc’s OpenMP barrier performance is not adequate.

High performance on Linux is achieved by careful use of spin waiting.

We can use the same approach on Mac OS X and Windows and achieve good performance.

Unfortunately, this means abandoning OpenMP.
OpenMP provides a convenient way to implement parallelism at the C/FORTRAN level.

Good performance of the synchronization barrier is critical for fine-grained parallelization.

On Linux/gcc OpenMP performance is very good.

On Mac OS X and Windows gcc’s OpenMP barrier performance is not adequate.

High performance on Linux is achieved by careful use of spin waiting.

We can use the same approach on Mac OS X and Windows and achieve good performance.

Unfortunately, this means abandoning OpenMP.
OpenMP provides a convenient way to implement parallelism at the C/FORTRAN level.

Good performance of the synchronization barrier is critical for fine-grained parallelization.

On Linux/gcc OpenMP performance is very good.

On Mac OS X and Windows gcc’s OpenMP barrier performance is not adequate.

High performance on Linux is achieved by careful use of spin waiting.

We can use the same approach on Mac OS X and Windows and achieve good performance.

Unfortunately, this means abandoning OpenMP.
OpenMP provides a convenient way to implement parallelism at the C/FORTRAN level.

Good performance of the synchronization barrier is critical for fine-grained parallelization.

On Linux/gcc OpenMP performance is very good.

On Mac OS X and Windows gcc’s OpenMP barrier performance is not adequate.

High performance on Linux is achieved by careful use of spin waiting.

We can use the same approach on Mac OS X and Windows and achieve good performance.

Unfortunately, this means abandoning OpenMP.
OpenMP provides a convenient way to implement parallelism at the C/FORTRAN level.

Good performance of the synchronization barrier is critical for fine-grained parallelization.

On Linux/gcc OpenMP performance is very good.

On Mac OS X and Windows gcc’s OpenMP barrier performance is not adequate.

High performance on Linux is achieved by careful use of spin waiting.

We can use the same approach on Mac OS X and Windows and achieve good performance.

Unfortunately, this means abandoning OpenMP.
Parallelizing Vector and Matrix Operations

Implementation Issues

- **OpenMP** provides a convenient way to implement parallelism at the C/FORTRAN level.
- Good performance of the synchronization barrier is critical for fine-grained parallelization.
- On Linux/gcc **OpenMP** performance is very good.
- On Mac OS X and Windows gcc’s **OpenMP** barrier performance is not adequate.
- High performance on Linux is achieved by careful use of *spin waiting*.
- We can use the same approach on Mac OS X and Windows and achieve good performance.
- Unfortunately, this means abandoning **OpenMP**.
OpenMP provides a convenient way to implement parallelism at the C/FORTRAN level.

Good performance of the synchronization barrier is critical for fine-grained parallelization.

On Linux/gcc OpenMP performance is very good.

On Mac OS X and Windows gcc’s OpenMP barrier performance is not adequate.

High performance on Linux is achieved by careful use of *spin waiting*.

We can use the same approach on Mac OS X and Windows and achieve good performance.

Unfortunately, this means abandoning OpenMP.
Parallelizing Vector and Matrix Operations

Implementation Issues

- We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.
- We expect to make an interface to this framework available to package authors as well.
- Care is needed to make sure that all functions called from worker threads are thread-safe.
- 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.
We are using a *pthread*ased implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
We are using a `pthreads`-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
Parallelizing Vector and Matrix Operations

Implementation Issues

- We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.
- We expect to make an interface to this framework available to package authors as well.
- Care is needed to make sure that all functions called from worker threads are thread-safe.
- 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.
We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
We are using a pthreads-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
We are using a *pthreads*-based implementation using atomic integer operations for synchronization during the spin wait.

We expect to make an interface to this framework available to package authors as well.

Care is needed to make sure that all functions called from worker threads are thread-safe.

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.
Parallelizing Vectorized Operations

Some Experimental Results

Luke Tierney (U. of Iowa)
Some observations:

- Times are roughly linear in vector length.
- Intercepts on a given platform are roughly the same for all functions.
- 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.
- 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.
- 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.
- 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.
- Relative slopes of functions seem roughly independent of OS/architecture.

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

The appropriate time to run calibration code is still open.
Parallelizing Vectorized Operations
Some Experimental Results

Some observations:
- Times are roughly linear in vector length.
- Intercepts on a given platform are roughly the same for all functions.
- 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.
- 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.
- 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.
An experimental package `pnmath0` that parallelizes many basic vectorized math functions is available at

http://www.stat.uiowa.edu/~luke/R/experimental/

The functions `colSums` and `dist` in the current R distribution can run in parallel but do not by default.

Hopefully more will be included in the R distribution before too long.

Still need to find clean way for a user to control the maximal number of threads allowed.

Also need to resolve whether slight changes of results are acceptable, especially in reductions.
An experimental package *pnmath0* that parallelizes many basic vectorized math functions is available at

http://www.stat.uiowa.edu/~luke/R/experimental/

The functions *colSums* and *dist* in the current R distribution can run in parallel but do not by default.

- Hopefully more will be included in the R distribution before too long.
- Still need to find clean way for a user to control the maximal number of threads allowed.
- Also need to resolve whether slight changes of results are acceptable, especially in reductions.
An experimental package `pnmath0` that parallelizes many basic vectorized math functions is available at

http://www.stat.uiowa.edu/~luke/R/experimental/

The functions `colSums` and `dist` in the current R distribution can run in parallel but do not by default.

Hopefully more will be included in the R distribution before too long.

Still need to find clean way for a user to control the maximal number of threads allowed.

Also need to resolve whether slight changes of results are acceptable, especially in reductions.
An experimental package **pnmath0** that parallelizes many basic vectorized math functions is available at

http://www.stat.uiowa.edu/~luke/R/experimental/

- The functions **colSums** and **dist** in the current R distribution can run in parallel but do not by default.

- Hopefully more will be included in the R distribution before too long.

- Still need to find clean way for a user to control the maximal number of threads allowed.

- Also need to resolve whether slight changes of results are acceptable, especially in reductions.
An experimental package `pnmath0` that parallelizes many basic vectorized math functions is available at

http://www.stat.uiowa.edu/~luke/R/experimental/

- The functions `colSums` and `dist` in the current R distribution can run in parallel but do not by default.
- Hopefully more will be included in the R distribution before too long.
- Still need to find clean way for a user to control the maximal number of threads allowed.
- Also need to resolve whether slight changes of results are acceptable, especially in reductions.
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.
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.
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.
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.
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 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 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 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 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.
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.

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.
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.

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.
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.

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.
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.

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.
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first use
- loops are compiler before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first use
- loops are compiler before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where:

- functions are compiled on first use
- loops are compiler before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first use
- loops are compiler before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first use
- loops are compiler before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where:
- functions are compiled on first use
- loops are compiled before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first use
- loops are compiler before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where

- functions are compiled on first use
- loops are compiled before they are run
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.
- Since R 2.14.0 R code in all base and recommended packages is compiled by default.

Alternatively, the compiler can be used in a JIT mode where
- functions are compiled on first use
- loops are compiler before they are run
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.
  \[ \text{dnorm}(y, 2, 3) \]
  is replaced by
  \[ \text{.Internal(dnorm}(y, \text{mean} = 2, \text{sd} = 3, \text{log} = \text{FALSE})) \]
- 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.
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.
    
    ```r
    dnorm(y, 2, 3)
    ```

    is replaced by
    
    ```r
    .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.
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.
  \[ \text{dnorm}(y, 2, 3) \]
  is replaced by
  \[ \text{.Internal(dnorm}(y, \text{mean} = 2, \text{sd} = 3, \text{log} = \text{FALSE})) \]
- 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.
The current compiler includes a number of optimizations, such as:

- constant folding
- special instructions for most SPECIALs, many BUILTINs
- inlining simple \texttt{.Internal} calls: e.g.
  \[
  \text{dnorm}(y, 2, 3)
  \]
  is replaced by
  \[
  \text{.Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))}
  \]
- special instructions for many \texttt{.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.
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.
  
  \[
  \text{dnorm}(y, 2, 3)
  \]
  
  is replaced by
  
  \[
  .\text{Internal}(\text{dnorm}(y, \text{mean} = 2, \text{sd} = 3, \log = \text{FALSE}))
  \]

- 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.
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.
  \[
  \text{dnorm}(y, 2, 3)
  \]
  is replaced by
  \[
  \text{.Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))}
  \]
- 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.
**R Code**

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

**VM Assembly Code**

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Interp.</th>
<th>Comp.</th>
<th>Speedup</th>
<th>Exper.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>32.19</td>
<td>7.98</td>
<td>4.0</td>
<td>1.47</td>
<td>21.9</td>
</tr>
<tr>
<td>sum</td>
<td>6.72</td>
<td>1.86</td>
<td>3.6</td>
<td>0.59</td>
<td>11.4</td>
</tr>
<tr>
<td>conv</td>
<td>14.48</td>
<td>4.30</td>
<td>3.4</td>
<td>0.81</td>
<td>17.9</td>
</tr>
<tr>
<td>rem</td>
<td>56.82</td>
<td>23.68</td>
<td>2.4</td>
<td>4.77</td>
<td>11.9</td>
</tr>
</tbody>
</table>

Interp., Comp. are for the current released version of R
Exper.: experimental version using
- separate instructions for vector, matrix indexing
- typed stack to avoid allocating intermediate scalar values
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
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
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
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
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
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
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
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
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
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
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
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
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
There is synergy among these three areas of development; for example:

- Many functions applied to large data are excellent candidates for parallelization.
- The compiler may be able to fuse operations and allow more efficient parallelization at the fused operation level.
- The compiler may also be able to compile certain uses of sweep and apply functions.

Exploring these opportunities will be a goal of work over the coming year.
There is synergy among these three areas of development; for example:

- Many functions applied to large data are excellent candidates for parallelization.
- The compiler may be able to fuse operations and allow more efficient parallelization at the fused operation level.
- The compiler may also be able to compile certain uses of sweep and apply functions.

Exploring these opportunities will be a goal of work over the coming year.
There is synergy among these three areas of development; for example:

- Many functions applied to large data are excellent candidates for parallelization.
- The compiler may be able to fuse operations and allow more efficient parallelization at the fused operation level.
- The compiler may also be able to compile certain uses of `sweep` and `apply` functions.

Exploring these opportunities will be a goal of work over the coming year.
There is synergy among these three areas of development; for example:

- Many functions applied to large data are excellent candidates for parallelization.
- The compiler may be able to fuse operations and allow more efficient parallelization at the fused operation level.
- The compiler may also be able to compile certain uses of `sweep` and `apply` functions.

Exploring these opportunities will be a goal of work over the coming year.
There is synergy among these three areas of development; for example:

- Many functions applied to large data are excellent candidates for parallelization.
- The compiler may be able to fuse operations and allow more efficient parallelization at the fused operation level.
- The compiler may also be able to compile certain uses of `sweep` and `apply` functions.

Exploring these opportunities will be a goal of work over the coming year.