Part I: Introductory Materials
Introduction to Parallel Computing with R

Dr. Nagiza F. Samatova
Department of Computer Science
North Carolina State University
and
Computer Science and Mathematics Division
Oak Ridge National Laboratory
What Analysis Algorithms to Use?

The Computer Science & HPC Challenges

Analysis algorithms fail for a few gigabytes.

Algorithmic Complexity:
- Calculate means $O(n)$
- Calculate FFT $O(n \log(n))$
- Calculate SVD $O(r \cdot c)$
- Clustering algorithms $O(n^2)$

If $n=10\text{GB}$, then what is $O(n)$ or $O(n^2)$ on a teraflop computer?

$1\text{GB} = 10^9$ bytes $1\text{Tflop} = 10^{12}$ op/sec

<table>
<thead>
<tr>
<th>Data size, $n$</th>
<th>Algorithm Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100\text{B}$</td>
<td>$10^{-10}$ sec.</td>
</tr>
<tr>
<td>$10\text{KB}$</td>
<td>$10^{-8}$ sec.</td>
</tr>
<tr>
<td>$1\text{MB}$</td>
<td>$10^{-6}$ sec.</td>
</tr>
<tr>
<td>$100\text{MB}$</td>
<td>$10^{-4}$ sec.</td>
</tr>
<tr>
<td>$1\text{GB}$</td>
<td>$10^{-2}$ sec.</td>
</tr>
</tbody>
</table>

For illustration chart assumes $10^{-12}$ sec. ($1\text{Tflop/sec}$) calculation time per data point.
Strategies to @ Computational Challenge

• Reduce the amount of data for the algorithm to work on, $n$
• Develop “better” algorithms in terms of big-O
• Take advantage of parallel computers with multi-core, multi-GPU, multi-node architectures
  • Parallel algorithm development
  • Environments for parallel computing
• Optimize end-to-end data analytics pipeline (I/O, data movements, etc.)
End-to-End Data Analytics

- Domain Application Layer
  - Biology
  - Climate
  - Fusion

- Interface Layer
  - Dashboard
  - Web Service
  - Workflow

- Middleware Layer
  - Automatic Parallelization
  - Scheduling
  - Plug-in

- Analytics Core Library Layer
  - Parallel
  - Distributed
  - Streamline

- Data Movement, Storage, Access Layer
  - Data Mover Light
  - Parallel I/O
  - Indexing

Our focus
Introduction to parallel computing with R

• What is parallel computing?
• Why should the user use parallel computing?
• What are the applications of parallel computing?
• What techniques can be used to achieve parallelism?
• What practical issues can arise while using parallel computing?

http://www.hcs.ufl.edu/~george/sci_torus.gif
The world is parallel

- The universe is inherently parallel
- Solar system, road traffic, ocean patterns, etc. exhibit parallelism


http://upload.wikimedia.org/wikipedia/commons/7/7e/Bangkok-sukhumvit-road-traffic-200503.jpg
What is parallel computing?

Parallel computing refers to the use of many computational resources to solve a problem.

http://www.admin.technion.ac.il/pard/archives/Researchers/ParallelComputing.jpg
Why should parallel computing be used?

- Solve bigger problems faster
- If serial computing is not viable (a large dataset or a single CPU cannot handle the entire dataset)
- Improve computational efficiency
- Save time and money
Applications of parallel computing

- Weather prediction
- Computer graphics, networking, etc.
- Image processing
- Statistical analysis of financial markets
- Semantic-based search of web pages
- Protein folding prediction
- Cryptography
- Oil exploration
- Circuit design and microelectronics
- Nuclear physics

http://www.nasm.si.edu/webimages/640/2006-937_640.jpg
http://jeffmohn.files.wordpress.com/2009/04/stock_market_down2.jpg
http://bfi-internal.org/dsnews/v8_no11/processing.jpg
Division of problem set: **Data parallel**

- **Data is broken** into a number of subsets.

- The *same instructions* are executed simultaneously on different processors for *different data subset*.
Division of problem set: **Task parallel**

- **Instructions are broken** into a number of independent instructions.

- **Different instructions** are executed on the same data simultaneously on different processors.
Embarrassingly Parallel Computing

- Solving many similar problems
- Tasks are independent
- Little to no need for coordination between tasks
Niceties of embarrassing parallelism

- Communication cost is lowered.

- Highly efficient for large data sets.

- Little bit of tweaking in code and you are ready to go!!

- Suitable for MapReduce programming paradigm.
Parallel R aims:

(1) to automatically detect and execute task-parallel analyses;
(2) to easily plug-in data-parallel MPI-based C/C++/Fortran codes;
(3) to retain high-level of interactivity, productivity and abstraction.

Embarrassingly-parallel:
- Likelihood Maximization
- Sampling: Bootstrap, Jackknife
- Markov Chain Monte Carlo
- Animations

Data-parallel:
- k-means clustering
- Principal Component Analysis
- Hierarchical clustering
- Distance matrix, histogram
Towards Enabling Parallel Computing in \( R \)
http://cran.cnr.berkeley.edu/web/views/HighPerformanceComputing.html

- **snow** (Luke Tierney): general API on top of message passing routines to provide high-level (parallel apply) commands; mostly demonstrated for embarrassingly parallel applications.

```
<table>
<thead>
<tr>
<th>snow API</th>
<th>High Level Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>parLapply</td>
<td>parallel lapply</td>
</tr>
<tr>
<td>parSapply</td>
<td>parallel sapply</td>
</tr>
<tr>
<td>parApply</td>
<td>parallel apply</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Routines</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>clusterExport</td>
<td>export variables to nodes</td>
</tr>
<tr>
<td>clusterCall</td>
<td>call function on each node</td>
</tr>
<tr>
<td>clusterApply</td>
<td>apply function to arguments on nodes</td>
</tr>
<tr>
<td>clusterApplyLB</td>
<td>load balanced clusterApply</td>
</tr>
<tr>
<td>clusterEvalQ</td>
<td>evaluate explicit expression on nodes</td>
</tr>
<tr>
<td>clusterSplit</td>
<td>split vector into pieces for nodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Administrative Routines</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>makeCluster</td>
<td>create a new cluster of nodes</td>
</tr>
</tbody>
</table>
```

- **Rmpi** (Hao Yu): \( R \) interface to MPI.
- **rpvm** (Na Li and Tony Rossini): \( R \) interface to PVM; requires knowledge of parallel programming.

```R
> library (pvm)
> .PVM.start.pvmd ()
> .PVM.addhosts (...)  
> .PVM.config ()
```
Parallel Paradigm Hierarchy

Parallel Paradigms

Explicit Parallelism
- Rmpi
- rpvm

Task-Parallel
- taskPR

Data-Parallel

Hybrid: Task + Data Parallel
- pR
- taskPR

Implicit Parallelism

Intensive Inter-Process Communication
- pR
- RScaLAPACK

No or Limited Inter-Process Communication
- pRapply
- multicore
- snow
Parallel Paradigm Hierarchy

Parallel Paradigms
- Explicit Parallelism
  - Rmpi
  - rpvm
- Implicit Parallelism
  - Task-Parallel
    - taskPR
  - Data-Parallel
  - Hybrid: Task + Data Parallel
    - pR
    - taskPR
- Intensive Inter-Process Communication
  - pR
  - RScaLAPACK
- No or Limited Inter-Process Communication
  - pRapply
  - multicore
  - snow
APPLY family of functions in R

- **apply()**: Applies a function to sections of array and returns result in array.
  
  Structure:
  
  apply(array, margin, function, ...)

- **lapply()**: Applies function to each element in a list.
  
  Structure:
  
  lapply(list, function, ...)

- **sapply()**: Applies function to each element in a list and returns numeric vector.
  Structure:
  
  sapply(list, function, ...)

- **tapply()**: Applies function to each element in a list and returns numeric vector.
  
  Structure:
  
  tapply(list, INDEX, FUN, ...)
R’s `lapply` Method is a Natural Candidate for Automatic Parallelization

- Examples: Bootstrapping, Monte Carlo, etc.

Using R:

```r
x = c(1:16);
lapply(x, sqrt)
```
Existing R Packages with Parallel `lapply`

• **multicore**
  – Limited to single-node, multi-core execution
  – `mclapply()`

• **pRapply**
  – Multi-node, multi-core execution
  – Automatically manages all R dependencies
  – `pRlapply()`

• **snow**
  – Built on Rmpi – uses MPI for communication
  – Requires users to explicitly manage R dependencies (libraries, variables, functions)
  – `clusterApply()`
Function Input/Output, R Environment

Using R:

```r
a=5;
y = matrix(1:12,3,4);
fn <- function(x){
  z = y+x;
  b = cbind(y,z);
}
d=fn(a);
d;
```

- How many inputs in fn()?
- What are the inputs to the function?
- What are the outputs?
- How many outputs?
- Will fn() know the value of y?
- What cbind() does?
- What d is equal to?
- How to return more than one output?

```r
a=5;
y = matrix(1:12,3,4);
fn <- function(x){
  z = y+x;
  b = cbind(y,z);
  return(list(z,b));
}
d=fn(a);
d;
```
pRapply Example

Using pRapply:

```r
library(pRapply);
library(abind);
x = as.list(1:16);
y = matrix(1:12,3,4);
fn <- function(x){
z = y+x;
#w = abind(x,x);
b = cbind(y,z);
}
pRlapply(x, fn)
```

Using R:

```r
library(abind);
x = as.list(1:16);
y = matrix(1:12,3,4);
fn <- function(x){
z = y+x;
w = abind(x,x);
b = cbind(y,z);
}
lapply(x, fn)
```

If I run on multiple machines, how non-local host would know about the R environment (e.g., `y` and `abind`) created before function call?
library(snow); library(abind); x = as.list(1:16); y = matrix(1:12,3,4); fn <- function(x){
  z = y+x;
  w = abind(x,x);
  b = cbind(y,z);
}
cl = makeCluster(c(numProcs=4), type = "MPI")
clusterExport(cl, "y");
clusterEvalQ(cl, library(abind));

Explicitly send libraries, functions, and variables before clusterApply()

clusterApply(cl, x, fn);
stopCluster(cl);
pR Automatic Parallelization
Uses a 2-Tier Execution Strategy

R End-User System

lapply(list, function)

list

pR

R Worker

C_1

C_2

C_4

C_3

R Worker

C_1

C_2

C_4

C_3

C_i = i^{th} core
MULTICORE package and mclapply()

- Multicore provides a way for parallel computing in R.
- Jobs share the entire initial work space.
- Provides method for result collection.
Multicore’s mclapply():

- Function mclapply() is the parallelized notion of lapply().
- Takes several arguments in addition to lapply().
- Arguments are used to set up parallel environment.
- By default input list is split into as many parts as there are cores.
- Returns the result in a list.
The conversion of `lapply()` to `mclapply()` is relatively very simple.

**Serial version:**
```r
myList = as.list(1:100)
lapply(myList, sqrt)
```

**Parallel version:**
```r
library(multicore)
myList = as.list(1:64)
mclapply(myList, sqrt)
```
Problems using mclapply() with smaller data sets

- mclapply() is not always faster than lapply() and sometimes is slower.

- lapply() works well with smaller data sets than mclapply().

- Overhead in setting up parallel environment.

- Distributing work.

- Collecting results.
mclapply() on large and computationally intensive problems

- Matrix multiplication is more intensive in terms of computations and problem size.
- Multiplication of two 1024*1024 (A*A) matrices using mclapply() is substantially quicker than lapply().
- It is done by splitting rows of the left matrix equally among all the processors.
- Each matrix then computes local product by multiplying with original matrix A.
- Results are unlisted into a matrix.
Parallel Paradigm Hierarchy

Parallel Paradigms

Explicit Parallelism
- Rmpi
- rpvm

Implicit Parallelism

Task-Parallel
- taskPR

Data-Parallel

Hybrid: Task + Data Parallel
- pR
- taskPR

Intensive Inter-Process Communication
- pR
- RScaLAPACK

No or Limited Inter-Process Communication
- pRapply
- multicore
- snow
What is RScaLAPACK?

• Motivation:
  – Many data analysis routines call linear algebra functions
  – In R, they are built on top of serial LAPACK library: [http://www.netlib.org/lapack](http://www.netlib.org/lapack)

• ScaLAPACK:
  – parallel LAPACK: [http://www.netlib.org/scalapack](http://www.netlib.org/scalapack)

• RScaLAPACK is a wrapper library to ScaLAPACK:
  – Also allows to link with ATLAS: [http://www.netlib.org/atlas](http://www.netlib.org/atlas)
Ex: RScaLAPACK Examples

A = matrix(rnorm(256),16,16)
b = as.vector(rnorm(16))

Using RScaLAPACK:
- library (RScaLAPACK)
- sla.solve (A,b)
- sla.svd (A)
- sla.prcomp (A)

Using R:
- solve (A,b)
- La.svd (A)
- prcomp (A)
Matrix Multiplication w/ RScaLAPACK

• \texttt{sla.multiply} function is used to parallelize matrix multiplication.

• \texttt{sla.multiply(A, B, NPROW, NPCOL, MB, RFLAG, SPAWN)}

• \texttt{NPROW} and \texttt{NPCOL} allows to split the rows and columns of a matrix, so that it becomes separate blocks.

• Each processor will execute each section.
Matrix multiplication w/ RscaLAPACK

- Given matrices are divided based on NPROWS and NPCOLS.
- Resulting blocks are distributed among the participating processors.
- Each processor calculates the product for its allocated block.
- Finally, the results are collected.

Division of a matrix for matrix multiplication using RscaLAPACK
Matrix multiplication (contd.)

Example

Multiplying two 64 X 64 matrices
- **Generate two matrices**
  
  ```r
  library(RScaLAPACK)
  M1 = matrix(data=rnorm(4096), nrow=64, ncol=64)
  M2= matrix(data=rnorm(4096), nrow=64, ncol=64)
  ```

- **Multiplication using sla.multiply**
  
  ```r
  result = sla.multiply(M1, M2, 2, 2, 8, TRUE, TRUE)
  class(result)
  dim(data.frame(result))
  ```

- **If there is at least 4 processors, the execution time would be faster than the serial computation**
  
  `dim(M1 %*% M2).`
## Currently Supported Functions

<table>
<thead>
<tr>
<th>Serial R Functions</th>
<th>Parallel RScaLAPACK</th>
<th>RScaLAPACK Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svd</td>
<td>sla.svd</td>
<td>Compute a singular value decomposition of a rectangular matrix</td>
</tr>
<tr>
<td>eigen</td>
<td>sla.eigen</td>
<td>Computes the Eigen values and Eigen vectors of symmetric square matrix</td>
</tr>
<tr>
<td>chol</td>
<td>sla.chol</td>
<td>Computes the Choleski factorization of a real symmetric positive definite square matrix</td>
</tr>
<tr>
<td>chol2inv</td>
<td>sla.chol2inv</td>
<td>Invert a symmetric, positive definite, square matrix from its Choleski decomposition</td>
</tr>
<tr>
<td>solve</td>
<td>sla.solve</td>
<td>This generic function solves the equation a*x=b for x</td>
</tr>
<tr>
<td>qr</td>
<td>sla.qr</td>
<td>Computes the QR decomposition of a matrix</td>
</tr>
<tr>
<td>factanal</td>
<td>sla.factanal</td>
<td>Perform maximum-likelihood factor analysis on a covariance matrix or data matrix using RScaLAPACK functions</td>
</tr>
<tr>
<td>factanal.fit.mle</td>
<td>sla.factanal.fit.mle</td>
<td>Perform maximum-likelihood factor analysis on a covariance matrix or data matrix using RScaLAPACK functions</td>
</tr>
<tr>
<td>prcomp</td>
<td>sla.prcomp</td>
<td>Performs a principal components analysis on the given data matrix using RScaLAPACK functions</td>
</tr>
<tr>
<td>princomp</td>
<td>sla.princomp</td>
<td>Performs a principal components analysis on the given data matrix using RScaLAPACK functions</td>
</tr>
<tr>
<td>varimax</td>
<td>sla.varimax</td>
<td>These functions rotate loading matrices in factor analysis using RScaLAPACK functions</td>
</tr>
<tr>
<td>promax</td>
<td>sla.promax</td>
<td>These functions rotate loading matrices in factor analysis using RScaLAPACK</td>
</tr>
</tbody>
</table>
Dimension Reduction w/ RscaLAPACK

- Multidimensional Dimensional Scaling (MDS) – a technique to place data into Euclidean space in a meaningful way.

- Function `cmdscale` corresponds to MDS in R.

- MDS requires high computation due to which parallelizing will reduce the running time significantly.

- Cmdscale has a pair of calls to eigen function to calculate eigenvectors and eigenvalues.
How to convert cmdscale to pcmdscale?

1. Open the code for `cmdscale` using `fix (cmdscale)`.

2. Create a new function `pcmdscale` by writing the code given.

3. Replace all instances of the serial eigen function calls in the code with `sla.eigen`.

4. `require(RScaLAPACK)` is to load the RscaLAPCK library.

```r
pcmdscale <- function (d, k = 2, eig = FALSE, 
add = FALSE, x.ret = FALSE, NPROWS=0, 
NPCOLS=0, MB=48, RFLAG=1, SPAWN=1) 
# include options for parallelization 
{
  if (require("RScaLAPACK", quietly = TRUE))
    # parallel eigen function 
    e <- sla.eigen(Z, NPROWS, NPCOLS, MB, 
    RFLAG, SPAWN)$values 
  else
    # serial eigen function 
    e <- eigen(Z, symmetric = FALSE, 
    only.values = TRUE)$values 
}
```
Scalability of \( pR: \text{RScaLAPACK} \)

\[ R > \text{solve} \left( A, B \right) \quad \text{pR} > \text{sla.solve} \left( A, B, NPROWS, NPCOLS, MB \right) \]

\( A, B \) are input matrices; \( NPROWS \) and \( NPCOLS \) are process grid specs; \( MB \) is block size.

Architecture: SGI Altix at CCS of ORNL with 256 Intel Itanium2 processors at 1.5 GHz; 8 GB of memory per processor (2 TB system memory); 64-bit Linux OS; 1.5 TeraFLOPs/s theoretical total peak performance.
RedHat and CRAN Distribution

CRAN R-Project

Available for download from R’s CRAN web site (www.R-Project.org) with 37 mirror sites in 20 countries

http://cran.r-project.org/web/packages/RScaLAPACK/index.html

http://rpmfind.net/linux/RPM/RByName.html
RScaLAPACK Installation

• **Download** RscaLAPACK from R’s CRAN web-site

• **Install dependency packages:**
  – Install R
  – MPI (Open MPI, MPICH, LAM MPI)
  – ScaLAPACK (with the proper MPI distribution)
  – Setup environment variables
    ```
    export LD_LIBRARY_PATH=<path2deps>/lib:$LD_LIBRARY_PATH
    ```

• **Install RScaLAPACK:**
  – R CMD INSTALL --configure-args="--with-f77
  --with-mpi=<MPI install home directory>
  --with-blacs=<blacs build>/lib
  --with-blas=<blas build>/lib
  --with-lapack=<lapack build>/lib
  --with-scalapack=<scalapack build>/lib"
  RScaLAPACK_0.6.1.tar.gz
Parallel Paradigm Hierarchy

Explicit Parallelism
- Explicit Parallelism
  - Task-Parallel
    - Intensive Inter-Process Communication
      - pR
        - RScaLAPACK
    - No or Limited Inter-Process Communication
      - pR
        - pRapply
  - Data-Parallel
    - pR
      - rpvm
  - Hybrid: Task + Data Parallel
    - pR
      - multicore
      - snow

Implicit Parallelism
- Implicit Parallelism
  - Task-Parallel
    - Intensive Inter-Process Communication
      - pR
        - RScaLAPACK
  - Data-Parallel
    - pR
  - Hybrid: Task + Data Parallel
    - pR
      - multicore
      - snow
Introduction to Rmpi

- What is MPI?
- Why should we use Rmpi?
- Different modes of communication
  - Point-to-point
  - Collective
- Performance issues

http://upload.wikimedia.org/wikipedia/commons/thumb/9/96/NetworkTopologies.png/300px-NetworkTopologies.png
What is MPI?

- Message Passing Interface
- Allows processors to communicate
- Different software implementations: MPICH, OpenMPI, etc.

http://financialaliyah.files.wordpress.com/2008/12/whisper.jpg
Advantages/Disadvantages of Rmpi

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Flexible – can use any communication pattern</td>
<td>• Complex</td>
</tr>
<tr>
<td>• No C/Fortran required</td>
<td>• Hard to debug</td>
</tr>
<tr>
<td></td>
<td>• Less efficient than C/Fortran</td>
</tr>
</tbody>
</table>
Using Rmpi

1. Spawn slaves
   • mpi.spawn.Rslaves
2. Distribute data
   • mpi.bcast.Robj2slave
3. Do work
   • mpi.bcast.cmd, mpi.remote.exec
   • Communication
4. Collect results
5. Stop slaves
   • mpi.close.Rslaves, mpi.quit
Point-to-Point Communication

- Message or data passed between two processors
- Requires a send and a receive call
- Can be *synchronous* or *asynchronous*

**Rmpi functions**
- `mpi.send`
- `mpi.recv`
- `mpi.isend, mpi.irecv, mpi.wait`
- `mpi.send.Robj, mpi.recv.Robj`
Synchronous vs. Asynchronous

**Synchronous** (mpi.send, mpi.recv)
- Waits until message has been received

**Asynchronous** (mpi.isend, mpi.irecv, mpi.wait)
- Starts sending/receiving message
- Returns immediately
- Can do other work in meantime
- Use mpi.wait to synchronize
Collective Communication

- Messages or data passed among several processors
- Several different communication patterns
- All are synchronous

**Rmpi functions**
- mpi.barrier
- mpi.bcast
- mpi.scatter, mpi.scatterv
- mpi.gather, mpi.allgather
- mpi.reduce, mpi.allreduce
**Barrier**

- Waits until all the processors (procs) call `mpi.barrier`

- Used to synchronize between different parts of an algorithm
Scatter and Gather

Scatter
• Divide a matrix/vector between procs

Gather
• Form a vector/matrix from smaller ones
• mpi.allgather sends result to every proc
Broadcast and Reduce

**Broadcast**
- Send a copy of data to many procs

**Reduce**
- Combine data together on one processor
- Can use sum, product, max, etc.
- mpi.allreduce
Rmpi May Not Be Ideal for All End-Users

- R-wraper around MPI
- R is required at each compute node
- Executed as interpreted code, which introduces noticeable overhead
- Supports ~40 of >200 MPI-2 functions
- Users must be familiar with MPI details
- Can be especially useful for prototyping
Rmpi Matrix Multiplication Requires Parallel Programming Knowledge and is Rmpi Specific

**Driver:**

A = matrix (c(1:256),16,16)
B = matrix (c(1:256),16,16);
C = mm_Rmpi(A,B,ncpu=2);

**Master**

```r
mm_Rmpi <- function(A, B, ncpu = 1) {
da <- dim(A) ## dims of matrix A
db <- dim(B) ## dim of matrix B
## Input validation
#mm_validate(A, B, da, db)
if( ncpu == 1)
  return(A %*% B)

## spawn R workers
mpi.spawn.Rslaves( nslaves = ncpu )

## broadcast data and functions
mpi.bcast.Robj2slave( A )
mpi.bcast.Robj2slave( B )
mpi.bcast.Robj2slave( ncpu )

## how many rows on workers ?
nrows_workers <- ceiling( da[1] / ncpu )
nrows_last <- da[1] - ( ncpu - 1 ) * nrows_workers

## broadcast info to apply
mpi.bcast.Robj2slave( nrows_workers )
mpi.bcast.Robj2slave( nrows_last )
mpi.bcast.Robj2slave( mm_Rmpi_worker )

## start partial matrix multiplication
mpi.bcast.cmd( mm_Rmpi_worker() )

## gather partial results from workers
results <- mpi.gather.Robj(local_results)
C <- NULL
for(i in 1:ncpu) C <- rbind(C, results[[i + 1]])
mpi.close.Rslaves()
return C; }
```

**Worker**

```r
mm_Rmpi_worker <- function(){
  commrank <- mpi.comm.rank() - 1
  if(commrank == ( ncpu - 1 ))
  else
  mpi.gather.Robj(local_results, root = 0, comm = 1 )
  return C; }
```
RScaLAPACK Matrix Multiplication

pR example:

library (RScaLAPACK)
A = matrix (c(1:256),16,16)
B = matrix (c(1:256),16,16)
C = sla.multiply (A, B)

Using R:

A = matrix (c(1:256),16,16)
B = matrix (c(1:256),16,16)
C = A %*% B
Recap: The Programmer’s Dilemma

What programming language to use & why?
Lessons Learned from R/Matlab Parallelization

Interactivity and High-Level: Curse & Blessing

- **Back-end approach**
  - data parallelism
  - C/C++/Fortran with MPI
  - RScaLAPACK (Samatova et al, 2005)

- **Automatic parallelization**
  - task parallelism
  - task-pR (Samatova et al, 2004)

- **Embarrassing parallelism**
  - data parallelism
  - snow (Tierney, Rossini, Li, Sevcikova, 2006)

- **Manual parallelization**
  - message passing
  - Rmpi (Hao Yu, 2006)
  - rpvm (Na Li & Tony Rossini, 2006)

- **Compiled approach**
  - Matlab→C→automatic parallelization

Packages: [http://cran.r-project.org/](http://cran.r-project.org/) Parallel Performance
Getting Good Performance

Minimizing Overhead

- Not possible to eliminate all overhead
  - E.g., spawning slaves, distributing data
- Minimize communication where possible
- Use asynchronous calls to overlap with computation
- Balance workloads between processors
  - Take work as needed until all finished
  - “Steal” work from processors with a lot
  - Other strategies
Measuring Scalability

**Strong Scaling**
- Same data, increase processors
- Ideal scaling: reduce time by number of processors

**Weak Scaling**
- Increase amount of data
- Keep amount of work per processor constant
- Ideal: time remains constant
Parallel computing - concerns

- Time gained by parallel computing is less than the time required to set up the machines.

- The output of one processor may be the input of the other.

- Imagine, what if each step of the problem depends on the previous step!

http://softtoyssoftware.com/dbnet/images/puzzle_incomplete.gif
Practical issues in parallelism - Overhead

- Overhead is the “extra” cost incurred by parallel computation.

http://i.ehow.com/images/a04/tl/di/calculate-overhead-cost-per-unit-200X200.jpg
Some major sources of overhead

(a) Initializing parallel environment,
(b) Distributing data,
(c) Communication costs,
(d) Dependencies between processors,
(e) Synchronization points,
(f) Unbalanced workloads,
(g) Duplicated work,
(h) Combining data,
(i) Shutting down parallel environment
Load balancing

- Distribution of workload across all the participating processors so that each processor has the same amount of work to complete.

- Unbalanced loads will result in poor performance.

- Factors to be considered:
  a) Speed of the processors
  b) Time required for individual tasks
  c) Any benefit arising because of processor coordination

Multi core processors without and with load balancing
Static load balancing

Each processor is assigned a workload by an appropriate load balancing algorithm.

Used when time taken by each part of the computation can be estimated accurately and all of the processors run at the same speed.
Dynamic load balancing

- The processors communicate during computation and redistribute their workloads as necessary.
- Used when the workload cannot be divided evenly, when the time needed to compute a task may be unknown, or where the processors may be running at different speeds.
- Strategies:
  - single, centralized “stack” of tasks
  - Push model
  - Pull model
Demonstrating load balancing

- `library(multicore)
  v = runif(16, 1, 10) * .04
  v2 = rep(mean(v), 16)
  system.time(mclapply(as.list(v), Sys.sleep))
  system.time(mclapply(as.list(v2), Sys.sleep))

- The parallel `mclapply` function call with the unbalanced distribution takes nearly twice as long as the `mclapply` call using the even distribution.

Simulation results for solving a problem with an unbalanced vs. a balanced load.
Scalability

Capability of parallel algorithm to take advantage of more Processors.

Overhead is limited to a small fraction of the overall computing time.

Scalability and cost optimality are inter-related.

Factors affecting scalability includes hardware, application algorithm and parallel overhead.
Measuring scalability

- How efficiently a parallel algorithm exploits parallel processing capabilities of parallel hardware?
- How well a parallel code will perform on a large scale system?
- Isoefficiency function
  Rate at which problem size has to increase in relation to number of processors
- How good we can do in terms of isoefficiency?
Strong scaling

- Speed up achieved by increasing the number of processors \(p\)

- Problem size is fixed.

- Speed up = \(\frac{t_s}{t_p}\)

  \(t_s = \text{time taken by serial algorithm}\)
  \(t_p = \text{time taken by parallel algorithm}\)

- Scaling reaches saturation after \(p\) reaches a certain value.
Strong scaling (Contd.)

- Strong scaling can be observed in matrix multiplication.
- The relative speed up is almost proportional to the processors used.
- Amount of time taken is inversely proportional to the number of processors.
- On saturation, further increase in processors does no good.
Weak scaling

- Speed up achieved by increasing both processors (P) and problem size (S).

- Workload / compute element is kept constant as one adds more elements.

- A problem n times larger takes same amount of time to do on N Processors.

- Ideal case of weak scaling is a flat line as both P and S increases.