LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR STATISTIK

Markus Schmidberger, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, Ulrich Mansmann

State-of-the-art in Parallel Computing with R

Technical Report Number 47, 2009
Department of Statistics
University of Munich

http://www.stat.uni-muenchen.de

State-of-the-art in Parallel Computing with R

Markus Schmidberger
Ludwig-Maximilians-Universität
München, Germany

Martin Morgan
Fred Hutchinson Cancer
Research Center, WA, USA

Dirk Eddelbuettel
Debian Project,
Chicago, IL, USA

Hao Yu
University of Western Ontario,
ON, Canada

Luke Tierney
University of Iowa,
IA, USA

Ulrich Mansmann
Ludwig-Maximilians-Universität
München, Germany

Abstract

R is a mature open-source programming language for statistical computing and graphics. Many areas of statistical research are experiencing rapid growth in the size of data sets. Methodological advances drive increased use of simulations. A common approach is to use parallel computing.

This paper presents an overview of techniques for parallel computing with R on computer clusters, on multi-core systems, and in grid computing. It reviews sixteen different packages, comparing them on their state of development, the parallel technology used, as well as on usability, acceptance, and performance.

Two packages (**snow**, **Rmpi**) stand out as particularly useful for general use on computer clusters. Packages for grid computing are still in development, with only one package currently available to the end user. For multi-core systems four different packages exist, but a number of issues pose challenges to early adopters. The paper concludes with ideas for further developments in high performance computing with R. Example code is available in the appendix.

Keywords: R, high performance computing, parallel computing, computer cluster, multi-core systems, grid computing, benchmark.

1. Introduction

R (R Development Core Team 2008a) is an open-source programming language and software environment for statistical computing and graphics. The core R installation provides the language interpreter and many statistical and modeling functions. R was originally created by R. Ihaka and R. Gentleman in 1993 and is now being developed by the R Development Core Team. R is highly extensible through the use of packages. These are libraries for specific functions or specific areas of study, frequently created by R users and distributed under suitable open-source licenses. A large number of packages are available at the Comprehensive R Archive Network (CRAN) at `http://cran.r-project.org` or the Bioconductor repository (Gentleman *et al.* 2004) at `http://www.bioconductor.org`. The R language was developed to provide a powerful and extensible environment for statistical and graphical techniques. However, it was not a development goal to provide software for parallel or high performance

computing (HPC) with R.

Nonetheless, during the last decade more and more research has focussed on using parallel computing techniques with R. The first available package was **rpvm** by Li and Rossini. This package provides a wrapper to the Parallel Virtual Machine (PVM) software. Early papers describing parallel computing with R are Li and Rossini (2001), Yu (2002), Sevcikova (2003) and Rossini *et al.* (2003). Interest in high performance computing with R has been increasing particularly in the last two years. The R mailing lists (`http://www.r-project.org/mail.html`) now frequently host discussions about using R for parallel computing. The UseR 2008 Conference (Ligges 2008) in Dortmund, Germany, contained a tutorial (Eddelbuettel 2008), as well as three sessions on HPC with R where several new packages were presented.

Two primary drivers for the increased focus on high performance computing with R can be identified: larger data sets (Hey and Trefethen 2003), and increased computational requirements stemming from more sophisticated methodologies.

Bioinformatics provides a good illustration for the 'growing data' problem. Consider that just a few years ago, experimentally-generated data sets often fit on a single CD-ROM. Today, data sets from high-throughput data sources—as for examples 'next generation' DNA sequencing (Shaffer 2007)— no longer fit on a single DVD-ROM. In genome research, data sets appear to be growing at a rate that is faster than the corresponding performance increases in hardware.

At the same time, methodological advances have led to more computationally demanding solutions. A common thread among these recent methods is the use of simulation and resampling techniques. Markov Chain Monte Carlo (MCMC) and Gibbs sampling, bootstrapping, and Monte Carlo simulation are examples of increasingly popular methods that are important in diverse areas of statistical analysis investigating geographic, econometric, and biological data.

Both increased data size and increased simulation demands have been approached by researchers via parallel computing. Data sets can frequently be subdivided into 'chunks' that can undergo analysis in parallel. This is particularly the case when the analysis of a given data 'chunk' does not depend on other chunks. Similarly, simulation runs that are independent of other simulations can be executed in parallel. Hence, both drivers for increased interest in high performance and parallel computing with R can be seen as so-called 'embarrassingly parallel' problems that are suitable for execution in parallel.

This paper reviews the state of parallel computing with R, and provides a starting point for researches interested in adopting parallel computing methods. The paper is organized as follows. Section 2 discusses code analysis with R, and introduces parallel computing concepts at the hardware and software level. Section 3 presents R packages for computer cluster and grid computing, multi-core systems and resource management. Section 4 provides a discussion of these different packages. The next section focusses on application packages and general tips for parallel computing. Section 6 concludes with a summary and outlook.

2. Overview

2.1. Code Analysis for R

Adapting code for parallel evaluation requires extra effort on the part of the researcher. For this reason it is important to ensure that parallelization is necessary. Important considera-

tions include use of appropriate serial algorithms, adoption of 'best practices' for efficient R programming, and knowledge of packages tailored to processing of large data sets.

A number of tools are available to profile R code for memory use and evaluation time. How to use these tools and how to detect probable bottlenecks is described in R Development Core Team (2008b), Venables (2001), Gentleman (2008), and in the R help page `?Rprof`. The CRAN packages **proftools** (Tierney 2007) and **profr** (Wickham 2008) provide more extensive profiling tools, an overview is available in Table 6 in the Appendix.

There exist a lot of different ways to solve performance issues. The solution mostly depends on the specific problem. Therefore this paper only presents a list of well known techniques to improve the R code (Gentleman 2008; Chambers 2008):

Vectorized Computation: Vectorized computation means any function call that, when applied to a vector, automatically operates directly on all elements of the vector. Many functions in R are vectorized. These are often implemented in C (see below), and hence very fast. This greatly reduces the need for loops and similar constructs that are prevalent in other languages. Vectorized computations should be used when possible.

apply-functions: There are a number of functions that can be used to apply a function, iteratively, to a set of inputs. One of the main reasons to prefer the use of an apply-like function over explicitly using a for-loop is that it more clearly and succinctly indicates what is happening and hence makes the code somewhat easier to read and understand. Appropriately implemented apply-like functions also often very readily adopted to parallel computation.

Foreign Language Interfaces: R provides a powerful interface to functions and libraries written in other languages. The main reason for using foreign languages is efficiency, since R is not compiled. In some situations the performance of R can be substantially improved by writing code in a compiled language. The most widely used interfaces are the `.C()` and `.Call()` functions which provides linkage to C code. R itself is largely written in C and it is easy to make use of the internal data structures, macros and code from routines written in C (see R Development Core Team 2008b).

There are a lot of other solutions for special problems. Most of them are presented at the UseR conferences and support is available by the R mailing lists of the R user community. This paper will focus onto techniques improving R code with parallel computing.

2.2. Parallel Computing

To understand and to evaluate the different packages that are available, it is important to have an overview of existing technologies for parallel computing. Detailed introductions can be found in various books for parallel computing or computer clusters (Sloan 2004; Dongarra *et al.* 2003).

In general, parallel computing deals with hardware and software for computation in which many calculations are carried out simultaneously. The main goal of parallel computing is the improvement in calculating capacity.

Hardware

There are several hardware types which support parallelism. For the work presented here,

however, the software part is more important. Therefore only the most popular hardware types are mentioned.

Multi-processor and multi-core computers have multiple processing elements within a single machine. In computers available to the general statistical computing community, the main memory is usually shared between all processing elements (shared memory). The idea of multicomputers — often called distributed computers — is the use of multiple computers to work on the same task. A distributed computer is a computer system in which the computers (processing elements with their own memory) are connected by a network. If the computers are located in a local area network (LAN) they may be called a computing cluster. In grid computing, machines are connected in a wide area network (WAN) such as the Internet.

Software

Several programming languages and libraries have been created for programming parallel computers. They can be classified based on the underlying memory architecture.

Shared memory: Shared memory programming languages communicate by manipulating shared memory variables. Widely used shared memory application programming interfaces (APIs) include OpenMP (Dagum and Menon 1998) and POSIX Threads (Pthreads) (Butenhof 1997). These are implementations of multithreading, a method of parallelization whereby the manager (master) thread forks a specified number of worker (slave) threads and a task is divided among them. Each thread executes the parallelized section of code independently.

OpenMP: Open Multi-Processing is an API that supports multiplatform shared memory multiprocessing programming in C/C++ and Fortran. Commercial and open source compilers for many architectures including Windows are available. OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications. The section of code that is meant to run in parallel is marked accordingly by using a preprocessor directive that will cause the threads to form before the section is executed.

Pthreads: POSIX Threads is a POSIX standard for threads. Libraries implementing the standard are often named Pthreads. For many architectures including Windows open source implementations exist. Pthreads defines a set of C programming language types, functions and constants. Programmers can use Pthreads to create, manipulate and manage threads, as well as to synchronize between threads using mutexes and signals.

OpenMP is under active development. Recent work indicates that is easier to program with OpenMP than Pthreads and that OpenMP delivers faster execution times (Breshears and Luong 2000; Binstock 2008).

Distributed memory: Message-passing APIs are widely used in distributed memory systems. Message passing is a form of communication which is made by sending messages to recipients. A manager-worker architecture—often called master-slave—is most common. In this model of communication one device or process (manager) controls one or more other devices or processes (workers). Once a manager-worker relationship between devices or processes is established (spawned), the direction of control is always from the manager to the

workers. Well known implementations of the standard are MPI (Message-Passing Interface) (Forum 1998) and PVM (Parallel Virtual Machine) (Geist *et al.* 1994).

MPI: Message-Passing Interface is a standardized and portable message-passing system designed by a group of researchers to function on a wide variety of parallel computers. The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes in a language-independent way. Open source implementations for many architectures, including Windows, exists. MPICH2 (Buntinas *et al.* 2007) and DeinoMPI (Deino Software 2006) currently provide a Windows implementation and OpenMPI (Gabriel *et al.* 2004) and MPICH2 are popular on Unix. OpenMPI is a project combining technologies and resources from several other projects (e.g., LAM/MPI) with the stated aim of building the best MPI library available.

PVM: The Parallel Virtual Machine is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel computer. Open source implementations for many architectures including Windows exist.

A comparison of both approaches is available in Gropp and Lusk (2002). For computer clusters PVM is still widely used, but MPI appears to be emerging as a de-facto standard for parallel computing.

3. Packages for Parallel Computing with R

A number of different approaches for high performance computing with R have been developed. This section describes different packages and technologies for parallel computing with R. The Appendix A contains short code examples for all packages discussed here. An overview containing hyperlinks to all packages is available in Table 1 and in Table 7 in the Appendix.

This section is divided into four parts: Packages for computer clusters, for grid computing, for multi-core systems, and for technologies for resource management. The packages are discussed and compared based on different software engineering criteria in Section 4.

3.1. R Packages for Computer Clusters

Nine different R packages for parallel computing on computer clusters have been published and are available at CRAN (see Table 1). Most of these are based on technologies described in Section 2.2. It is also possible to compile R with external libraries which support computer clusters.

Rmpi

The package **Rmpi** (Yu 2002) is an interface, or wrapper, to MPI. It provides an interface to low-level MPI functions from R so that users do not have to know details of the MPI implementations (C or Fortran). It requires that MPI is installed, and runs under popular MPI implementations: LAM/MPI, OpenMPI, MPICH, MPICH2 and DeinoMPI. This allows the **Rmpi** package to be run on clusters with Linux, Windows or Mac OS X.

The **Rmpi** package includes scripts to launch the R instances at the slaves from the master (`mpi.spawn.Rslaves()`). The instances run until they will be closed with the command

`mpi.close.Rslaves()`. The package provides several R-specific functions, in addition to wrapping the MPI API. For example, parallel versions of the R apply-like functions are provide by, e.g., `mpi.parApply()`. Also, R objects can be efficiently sent to slaves using `mpi.bcast.Robj2slave()`. There is also primitive error-handling to report errors form the workers to the manager.

rpvm

The package **rpvm** (Li and Rossini 2001) is an interface to PVM. It supports low-level PVM functions from R. There are no additional R functions, so the user has to provide all communication and parallel execution directives. And there is no command to launch R instances at the slaves from the master. The command `.PVM.spawnR()` can be used to execute a R script file (read by `source()`) at the slaves and after running the script the R instance will be closed automatically.

The package requires a running PVM installation and runs on clusters with Linux, Windows or Mac OS X.

nws

The package **nws** (Bjornson *et al.* 2007) (short for NetWorkSpaces) provides functions to store data in a so-called 'NetWorkSpace', and to use the 'sleigh' mechanism for parallel execution. NetWorkSpaces and Sleigh run on Linux and Windows platforms. It requires a running NetWorkSpace server. NWS is implemented in the widely-available Python language and uses its Twisted framework for network programming. NWS currently supports the Python, Matlab, and R languages, and allows limited cross-language data exchange.

The 'sleigh' class supports writing parallel programs. There are two basic functions for executing tasks in parallel (`eachElem()`, `eachWorker()`). The package uses the manager-worker paradigm, and automatically load-balances R function evaluations. To enable the manager to communicate with workers, sleigh supports several mechanisms to launch R instances at workers to run jobs: local, ssh, rsh, lsf, web launch.

The **nws** package and the NetWorkSpace Server are available as an open source and commercial product from REvolution Computing (and had previously been marketed as LindaSpaces by Scientific Computing Associates).

snow

The package **snow** (Simple Network of Workstations) (Rossini *et al.* 2007) supports simple parallel computing in R. The interface is intended to be simple, and is designed to support several different low-level communication mechanisms. Four low-level interfaces have been implemented: PVM (via the **rpvm** package), MPI (via **Rmpi**), NetWorkSpaces (via **nws**), and raw sockets that may be useful if PVM, MPI or NWS are not available. This means it is possible to run the same code at a cluster with PVM, MPI or NWS, or on single multi-core computer.

The **snow** package includes scripts to launch R instances on the slaves (`cl <- makeCluster()`). The instances run until they are closed explicitly via the command `stopCluster(cl)`. The package provides support of high-level parallel functions like `apply()` and primitive error-handling to report errors from the workers to the manager.

snowFT

The package **snowFT** is an extension of the **snow** package supporting fault tolerance, reproducibility and additional management features. The interface is very similar to the **snow** interface. The heart of the package is the function `performParallel()`, which is a wrapper function for `clusterApplyFT()`. Additional management features are functions for repairing a cluster: add, remove and restart slaves of a cluster. The package is only written for the PVM communication layer, and is no longer actively maintained.

snowfall

The package **snowfall** (Knaus 2008) supports more simple parallel computing in R. It is a toplevel wrapper around the package **snow** for easier development of parallel R programs. The interface strives to be as simple as the **snow** interface, yet differs in some aspects. For example it does not require handling of the R cluster object by the user. The package supports the same low-level communication mechanisms as **snow**. Additionally, all functions also work in sequential mode. This can be useful for debugging or when no cluster is present.

The package is designed as a connector to the cluster management tool **sfCluster** (see Section 3.4), but can be used without **sfCluster**.

papply

The package **papply** was developed to augement the parallel apply functionality of the **Rmpi** package. Similar to the serial `apply()` function, `papply()` applies a function to all items of a list, and returns a list with the results. It uses the **Rmpi** package to distribute the processing evenly across a cluster. If **Rmpi** is not available, it implements this as a non-parallel algorithm. This package is an add-on package to **Rmpi**, contains the function `papply()` and additional debugging support, and is no longer actively maintained.

biopara

The package **biopara** allows the user to distribute execution of parallel problems over multiple machines for concurrent execution. It uses raw socket connections (ssh). The package contains only one function: `biopara()`. For every parallel run the tcp/ip port numbers and computer tcp/ip hostnames are required, and all computers have to be in a trusted environment. An R instance has to be started at every worker, or a custom start script has to be executed, at the beginning in order to spawn the workers manually.

The **biopara** package provides limited fault tolerance and load balancing.

taskPR

The **Task-pR** package is an interface that uses the MPI implementation LAM/MPI for parallel computing. If there is no available LAM universe, tasks will be executed serially on the manager. A function that needs to be run in parallel has to be enclosed within the function `PE()` and will be executed at the first slave. The next `PE()` call will be executed at the next slave, and so on. There is no support of high-level parallel functions like `apply()`.

Using external libraries: BLAS and PBLAS

R relies on third-party BLAS libraries for common linear algebra and matrix operations. Optimized and parallelized shared-memory implementations of these libraries exist, and can be used for parallel computing on multicomputers and multi-core systems with R too (R Development Core Team 2008a).

The Basic Linear Algebra Subprograms (BLAS) is the de-facto application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication. The linear algebra routines in R can make use of enhanced BLAS implementations (e.g. ATLAS, Intel MKL) and of parallel BLAS routines: PBLAS is a collection of software for performing linear algebra operations on any system for which MPI or PVM is available (`http://www.netlib.org/scalapack/html/pblas_qref.html`). The library has to be integrated into R when R itself is compiled, using the configuration option `-with-blas` (R Development Core Team 2008a).

3.2. R Packages for Grid Computing

At present, two R packages and one java-based software collection exist for grid computing in R. These solutions are very new and were presented at the UseR2008 conference in August 2008.

gridR

The package **gridR** (Wegener *et al.* 2007) submits R functions for execution in a grid environment (on a single computer or a cluster) and provides an interface to share functions and variables with other users (`grid.share()`). Job submission is available via a web service, ssh or locally and execution modes are Condor (Tannenbaum *et al.* 2001), Globus (Foster 2005), or single server. The grid infrastructure uses R as interface and client. The server side implementation of **gridR** uses several external software components: Globus Toolkit 4 grid middleware, an installation of the R environment on the grid machines, and a GRMS-Server installation from the Gridge toolkit (Pukacki *et al.* 2006) on a central machine in the grid environment.

The grid configuration can be performed with configuration files (XML style) or at runtime by passing parameters. The main functions are `grid.init()` to initialize the grid and `grid.apply()` to submit a function into the grid environment. This function is different to the R apply-like functions, it does not apply a function to different data. Additionally there are functions to facilitate the management of running grid jobs (e.g., `grid.check()`).

The R package is available at CRAN and the maintainer website (see Table 1).

multiR

The package **multiR** (Grose 2008) is not yet available. At the time of review only the presentation from the UseR2008 conference was available. The implementation should have some similarity to aspects of **snow** and **gridR**, but should be independent of the many different types of hardware and software systems. It requires no additional software components (Globus, CoG, etc.) to be installed before it can be used.

Main differences involve job submission (via Grid middleware) and security (certificates). This is implemented with a 3 tier client/server architecture provided by GTROWL (Hayes *et al.*

2005). In this architecture, the server component communicates with Grid nodes on behalf of the client. The implementation will extend the `apply()` family of functions, and allows multiple function invocations in a distributed environment.

Biocep-R

The Biocep-R project (Chine 2007) is a general unified open source Java solution for integrating and virtualizing the access to servers with R. Tools make it possible to use R as a Java object-oriented toolkit or as a Remote Method Invocation (RMI) server. The Biocep-R virtual workbench provides a framework enabling the connection of all the elements of a computational environment: Computational resources (local machine, a cluster, a grid or a cloud server) via a simple URL, computational components via the import of R packages, GUIs via the import of plugins from repositories or the design of new views with a drag-and-drop GUI editor.

The java software and further information is available at the maintainer website (see Table 1). It requires an installed Java 5 run-time and R. The project is under active development and new features are available in every new version. User manuals and tutorials are not yet available.

3.3. R Packages for Multi-core Systems

There are four different packages using multi-core systems to accelerate computations in R. Three of the implementations are very new, they were all developed and presented in the last year.

pnmath and pnmath0

These two packages (Tierney 2008) offer a parallel implementation of most of the basic R numerical math routines (most of the non-random number, non-BLAS routines) for multi-core architectures. The package **pnmath** uses OpenMP; **pnmath0** uses Pthreads.

On loading, the packages replace built-in math functions with hand-crafted parallel versions. Once loaded, the package provides access to parallel functions with no further changes to user code. The packages use timing estimates (determined at install-time, if possible) to calibrate when the benefits of parallel evaluation outweigh the costs of thread initialization. With numerically intensive code, these packages yield a speed-up of up to a factor given by the number of available processors. The packages work on most platforms, including Window, but require compiler and library support as detailed in the package README files. Both packages are available at the maintainers website (see Table 1).

fork

The package **fork** (Warnes 2007) provides simple wrappers around the Unix process management API calls: `fork`, `signal`, `wait`, `waitpid`, `kill`, and `exit`. This enables construction of programs that utilize and mange multiple concurrent processes.

The main function of the package is `fork(slave_function)`, which will create a new R instance which is an exact copy of the current R process, including open files, sockets, etc.. For communication between the instances socket connections `socketConnection()` have to be used. There is no support of high-level parallel functions like `apply()` or any error handling.

The R package is available at CRAN, it only works in a Unix environment. Commercial support for this package is available from Random Technologies, LLC (http://www.random-technologies-llc.com).

R/parallel

The package **R/parallel** (Vera *et al.* 2008) enables automatic parallelization of loops without data dependencies by adding a single function: `runParallel()`. A data dependency occurs when the calculation of a value depends on the result of a previous iteration (e.g. $a[n] = a[n-1]+1$). For efficient R programming (see Section 2.1) an apply-like implementation of loops without data dependencies is preferred.

The package uses a master-slave architecture. The implementation is based on C++, and combines low level operating system calls to manage processes, threads and inter-process communications. The package is able to locate loops and automate their parallelization by requiring that the user enclose the `runParallel()` function and the target loop within an `if/else` control structure.

The R package is available at the maintainer website (see Table 1).

romp

The package **romp** (Jamitzky 2008) is only available as a presentation from the UseR2008 conference and some example code (version 01a) without documentation is available at the maintainers website. The code transforms R code to Fortran, inserts OpenMP directives, and compiles the Fortran code (`compile.mp()`). The compiled code is then executed in R. The example code includes functions `apply.mp()` and a `sum.mp()`.

3.4. Resource Manager Systems

Deploying parallel computing application becomes increasingly challenging as any one of the number of jobs, users or compute nodes increases. A resource management tool can alleviate such issues. Resource manager systems are software applications to submit, control and monitor jobs in computing environments. Synonyms are job scheduler or batch system. These applications work on the operating system level and mostly directly monitor the communication protocols. Therefore they are independent from the R language. They make the parallel computing environment manageable for administrators and especially if several people work in the same environment to avoid resource or scheduling conflicts.

In the R user community there exists some experience with different resource management software. In the following the mostly used ones are described, an overview containing hyperlinks is available in Table 7 in the Appendix.

SLURM: A Highly Scalable Resource Manager

SLURM (Jette *et al.* 2003), which stands for *Simple Linux Utility for Resource Management*, is an open-source resource manager designed for Linux. Slurm is powerful enough for being deployed on clusters containing over 100k processors, yet simple enough to be used for small personal clusters.

While Slurm works particularly well with OpenMPI, it can be built for use with different message passing libraries. Slurm can be deployed over different high-end fabrics (such as

Quadrics or Federation) as well as Myrinet or Ethernet. Slurm can also use external batch processing systems such as Maui or Moab.

Slurm provides three key functions. First, the `salloc` command can be used to allocate exclusive and/or non-exclusive access to resources (such as computing nodes) to users for some duration of time so they can perform work. Second, the `sbatch` or `srun` commands provide a framework for starting and executing work (typically a parallel job) on a set of allocated nodes. The state can be monitored using the `sinfo` and `squeue` commands as well as the `sview` graphical user interface. Third, Slurm can arbitrate contention for resources by managing a queue of pending work. Lastly, job accounting functionality, including record-keeping in SQL backends, is also available.

Slurm is under active development by a consortium comprised of a major research organisation (Lawrence Livermore National Laboratory) as well as several industrial partners (Hewlett-Packard, Bull, Cluster Resources and SiCortex). It is being released under the standard GNU GPL, but is also available as a component of commercial clusters such as IBM's BlueGene.

SGE: Sun Grid Engine

SGE is an open source batch-queuing system, supported by Sun Microsystems (Sun Microsystems 2002). It is available as an open source and a commercial product. SGE is responsible for accepting, scheduling, dispatching, and managing the remote execution of large numbers of standalone, parallel or interactive user jobs. It also manages and schedules the allocation of distributed resources such as processors, memory, disk space, and software licenses.

SGE is a very powerful and complex system to manage a cluster system. It is optimized to manage large environments. There is a big command line user interface to communicate with the SGE. The main functions are `qsub` to submit jobs and `qstat` to get a status listing of all jobs and queues. There is also a X-window command interface and monitoring facility (`qmon`).

Rsge - Interface to the SGE Queuing System: The **Rsge** package provides functions for using R with the SGE environment via the `qsub` command. The package requires an installed and configured SGE, a shared network device, and R with an installed **snow** package.

The package offers functions to get information from the SGE (`sge.job.status()`) and to submit work (R code) to an R cluster started from the SGE (`sge.apply(), sge.submit()`). At the manager the data object will be split as specified by the number of jobs and each data segment along with the function, argument list are saved to a shared network location. Each R instance loads the saved file, executes it, and saves the results to the shared network location. Finally the master script merges the results together and returns the result.

At the time of review the first version of the package was available. The code still contains a lot of open topics and a proper documentation is missing. The package does not support any of the parallel computing technologies described in Section 2.2. For the commercial LSF Queuing System a very similar package **Rlsf** exists on CRAN (`http://cran.r-project.org/web/packages/Rlsf`).

sfCluster

sfCluster (Knaus 2008) is a small resource manager optimized for cluster solutions with R.

It is a Unix tool written in perl for managing and observing R instance running on clusters. The parallel R code has to be written with the **snowfall** package.

sfCluster automatically sets up the cluster for the user and shutdowns all running R sessions after finish. It allows multiple clusters per user and offers four execution modes: batch mode (`-b`), interactive R-shell (`-i`), monitoring mode (`-m`) and sequential mode (`-s`). For developing parallel code the sequential mode and the `parallel=FALSE` option in the **snowfall** package can be used for developing on a single machine.

The software is available in a beta version at the maintainers website (see Table 7 in the Appendix). At the time of review sfCluster is only available for Unix systems and is based on LAM/MPI. The scheduler does not support queue mode and is optimized for the requirements of the maintainers institute.

Further Resource Manager

There are a lot of other possibilities for resource manager (TORQUE Resource Manager, Condor High-Throughput Computing System, LSF Queuing System, etc.) which can not be reviewed in detail here due to space constraints. The choice of an appropriate tool could be very difficult. The decision should depend on the used architecture and the specific requirements.

Package	Version	Websites	Technology
Computer Cluster			
Rmpi	0.5-6	http://cran.r-project.org/web/packages/Rmpi http://www.stats.uwo.ca/faculty/yu/Rmpi	MPI
rpvm	1.0.2	http://cran.r-project.org/web/packages/rpvm http://www.biostat.umn.edu/~nali/SoftwareListing.html	PVM
nws	1.7.0.0	http://cran.r-project.org/web/packages/nws http://nws-r.sourceforge.net	NWS and socket
snow	0.3-3	http://cran.r-project.org/web/packages/snow http://www.cs.uiowa.edu/~luke/R/cluster	Rmpi, rpvm, nws, socket
snowFT	0.0-2	http://cran.r-project.org/web/packages/snowFT	rpvm, snow
snowfall	1.60	http://cran.r-project.org/web/packages/snowfall http://www.imbi.uni-freiburg.de/parallel	snow
papply	0.1	http://cran.r-project.org/web/packages/papply http://math.acadiau.ca/ACMMaC/software/papply.html	Rmpi
biopara	1.5	http://cran.r-project.org/web/packages/biopara http://hedwig.mgh.harvard.edu/biostatistics/node/20	socket
taskPR	0.31	http://cran.r-project.org/web/packages/taskPR http://users.ece.gatech.edu/~gte810u/Parallel-R	MPI (only LAM/MPI)
Grid Computing			
GridR	0.8.4	http://cran.r-project.org/web/packages/GridR http://www.stefan-rueping.de	web service, ssh, condor, globus
multiR	-	http://e-science.lancs.ac.uk/multiR	3 tier client/server architecture
Biocep-R	NA	http://biocep-distrib.r-forge.r-project.org	java 5
Multi-core System			
pnmath(0)	0.2	http://www.cs.uiowa.edu/~luke/R/experimental	openMP, Pthreads
fork	1.2.1	http://cran.r-project.org/web/packages/fork	Unix: fork
R/Parallel	0.6-20	http://www.rparallel.org	C++, file
romp	0.1a	http://code.google.com/p/romp	openMP

Table 1: Overview about parallel R packages for computer clusters, grid computing and multi-core machines, including the latest version numbers, the corresponding hyperlinks and technologies.

Package	Version	Releases	First Version	Last Version	Development
Computer Cluster					
Rmpi	0.5-5	12	2002-05-10	2007-10-24	++
rpvm	1.0.2	12	2001-09-04	2007-05-09	++
nws	1.7.0.0	8	2006-01-28	2008-08-08	+
snow	0.3-3	9	2003-03-10	2008-07-04	++
snowFT	0.0-2	2	2004-09-11	2005-11-29	0
snowfall	1.60	5	2008-01-19	2008-10-22	+
papply	0.1	1	2005-06-23	2005-06-23	0
biopara	1.5	3	2005-04-22	2007-10-22	−−
taskPR	0.31	3	2005-05-12	2008-03-12	−
Grid Computing					
GridR	0.8.4	1	2008-12-04	2008-12-04	−
Multi-core System					
fork	1.2.1	6	2003-12-19	2007-03-24	+

Table 2: Versions of R parallel computing packages available at CRAN, characterized by state of development (++: highly developed, stable; −−: in development, unstable).

4. Comparison and Discussion

This section provides a discussion and critical comparison of packages for parallel computing with R.

4.1. State of development

It is important to have stable software whose basic functionality changes in a structured fashion. New features need to be introduced to allow leading-edge research. Legacy code might be deprecated and then removed in an orderly fashion, and over several release cycles, to minimize unnecessary disruption of user work flows. With these considerations in mind, we characterize the state of development of R packages in Table 2. The characterization is based on the number of released versions, the release frequency and the results from the performance analysis tests shown in Section 4.6 and Appendix B. The table shows only packages for parallel computing which are available via CRAN.

Packages for basic communication protocols MPI and PVM are very well developed, having been available and in wide use for close to a decade. The code and functionality are stable over recent releases. The higher-level wrapper package **snow** is also mature and stable. On the other hand, the package **snowfall** is still being developed, with recent addition of functionality. The **nws** (NetWorkSpaces) package is a relatively recent addition based on previous work; it is mature, while also still being extended. The packages **snowFT** and **papply** are no longer actively maintained. The **biopara** package contains many comments and preliminary 'debugging' output, and has seen only yearly releases; it is classified as 'under development'. Packages assessed as '0', '−' or '−−', experience or have the opportunity for major code and functionality changes.

4.2. Technology

Table 1 provides an overview of the different technologies employed by the various parallel computing packages. Differences among technologies were described in Section 2.2. For future developments and wide adoption, it is important for all packages to use standardized communication protocols, and to establish platform-independent solutions.

For computer clusters, PVM is still widely used. However, MPI appears to be emerging as a de-facto standard for parallel computing. For parallel computing with R, MPI is the best-supported communication protocol. Different MPI implementations exist and cover many different computer architectures. Most of them are supported by the package **Rmpi**. The higher-level package **snow** can then deploy **Rmpi** (as well as **rpvm**, **nws** or sockets) whereas the package **taskPR** only supports the LAM/MPI implementation. NWS and sleigh represent a slightly different approach to parallel computing with R. The shared global workspace model of these approaches could be very useful for special tasks.

For grid computing the only available package **GridR** uses the Globus Toolkit. Globus, developed by the Globus Alliance, is probably the most standardized and most widely used middleware for grid computing.

Early interfaces and integration of multi-core technology into R exist as research projects that are under active development. The OpenMP standard for multi-core systems and multiprocessors provides an established technology for parallelization at the compiler level. The package **pnmath** is a promising integration of this technology.

Several packages use coarse-grained processes to evaluate R expressions in parallel. The package **R/parallel** uses a master-slave architecture to launch several R instances on the same machine using system-level commands. Every R instance requires its own main memory, and the amount of main memory will frequently limit scalability. The same problem appears for the package **fork**, which is limited to Unix environments.

Installation

In general, installation of packages in the R environment is simple, with the user invoking the `install.packages()` function. This function takes a vector of package names, downloads these packages from on-line repositories, and installs them in user- or system-wide R libraries. However, packages for parallel computing are often based on underlying communications technology (e.g., MPI, PVM) and the packages have to be linked to libraries providing the corresponding technology. The underlying communication libraries have to be installed and configured on the computer cluster. In many cases the communication libraries will be installed by a system administrator, and the R packages must be customized (e.g. using the `configure.args` argument to `install.packages`, or in extreme cases by altering the R package configuration files) to find the appropriate communication library.

In order to use some external libraries (such as PBLAS) the R program itself has to be recompiled and reinstalled.

4.3. Fault-Tolerance and Load Balancing

Fault-tolerance for parallel algorithms provides continued operations in the event of failure of some slaves. On the other hand, load balancing is the technique to spread processes between resources in order to get optimal resource utilization. Both aspects are very important in the

	Learnability	Efficiency	Memorability	Errors	Satisfaction
Rmpi	+	−−	++	+	+
rpvm	−−	−−	+	−	−−
nws	+	+	++	+	+
snow	+	++	++	+	++
snowFT	+	++	++	+	++
snowfall	++	++	+	+	++
papply	+	+	++	+	0
biopara	−−	−−	0	0	−−
taskPR	+	−	++	0	−

Table 3: Assessment of the usability of the R packages for computer clusters (ranging from ++ to −−).

field of parallel computing.

Only the packages **snowFT** and **biopara** have integrated solutions for fault tolerance. Thus for many common use cases the users must accommodate the possibility of failure during lengthy calculations.

The packages **Rmpi**, **snow**, **snowFT**, **snowfall** and **biopara** support load balancing for the apply functions. Resource managers (discussed below) can help with load balancing.

4.4. Usability

Usability describes how easily software can be deployed to achieve a particular goal. Given the focus of R on statistical computing, this section stresses use cases that can arise in a variety of statistical analyses.

For computer clusters, the basic communication packages **Rmpi** and **rpvm** cover the full functionality of both communication standards and provide an essentially complete API to these standards. The complexity of the full API makes these packages somewhat more difficult for researchers with primarily statistical interests to become familiar with. The other R packages for computer clusters were developed to improve usability by providing a somewhat higher level of abstraction. Table 3 shows an estimation of the components of usability (Shneiderman and Plaisant 2004) for R packages for computer clusters. 'Learnability' describes how easy it is for first-time users to learn functionality. 'Efficiency' describes how quickly users can perform tasks. 'Memorability' is the assessment of how easily users can reestablish details required for use. How to avoid errors and to fix them is reviewed in 'Errors'. 'Satisfaction' describes how pleasant it is to use a package.

To assess the learnability of R packages the quality of the vignettes and help files were reviewed. Only the package **snowfall** integrates a vignette into the R package. The existence of online tutorials for the packages **Rmpi** and **snow** improves their learnability score.

To get good efficiency for an R package it is important to provide functions that facilitate writing simple code. Code based directly on the **Rmpi** and **rpvm** packages can become complex, because the user must interact with the communication API at a relatively low level. **biopara** and **taskPR** only have functions for sending commands to workers, so a lot of additional code for basic functionality has to be implemented by the user.

	Learnability	Efficiency	Memorability	Errors	Satisfaction
Grid Computing					
gridR	+	−	+	−	−
R/parallel	0	+	+	−	0
Multi-core System					
pnmath(0)	++	++	++	++	+
fork	+	−	+	−	−

Table 4: Assessment of the usability of the available R packages for grid computing and multi-core systems. (ranging from ++ to −−).

R's package structure provides for help files, and allows optional vignettes, so the memorability score for every R package is elevated. However, some packages provide incomplete, complex, or difficult to navigate help files. This leads to lower scores.

In order to correct and fix errors, it is important that functions provided by a package help the user to identify sources of error. For example parameters should be checked for correctness before use. The scoring criteria for 'Errors' was determined based on experiences during creation of test code in the appendix.

For satisfaction, the functionality provided by the different packages was compared. The packages **rpvm**, **biopara** and **taskPR** provide only very basic functions for parallel computing. The packages **snow**, **nws** and **Rmpi** have scripts to launch R instances at the workers. The packages **snowFT** and **snowfall** use scripts from the package **snow**.

The usability assessment of packages for grid computing is very difficult to determine, due to lack of mature and stable packages. Table 4 shows an estimation of the components of usability for available R packages for grid computing and multi-core systems.

In the **GridR** package the user has to deal with intricate configuration and connection to the grid system. Once the configuration is finished, the parameters can be reused in subsequent runs. Complex jobs require the user to implement a lot of additional code.

The usability of the packages **pnmath** is excellent. Once the package is loaded, parallelization runs without further code modification; the user employs familiar R functions. The package **R/parallel** is still in development and adequate documentation is missing, but the usability of the code is promising. Implementing R's apply-like functions for loops without data dependencies would be a valuable addition to this package (see Section 2.1).

4.5. Acceptance

To evaluate the acceptance of parallel computing packages, the number of links from the DESCRIPTION files of other packages was counted. We used the Biocondcutor (Gentleman *et al.* 2004) repository as a reference set of packages. This repository comprises about 294 R packages oriented towards research in bioinformatics and computational biology. Four packages use **snow**, three **Rmpi** and one package carries an additional references to **rpvm**. Therefore about 2% of the bioinformatic packages use parallel computing technologies, the packages **snow** and **Rmpi** are used exclusively.

The packages for grid computing and multi-core systems are in active development and not yet used in any other package. Only example code and applications exist for proof of principle

and functionality.

4.6. Performance

Performance of the different parallel computing packages can differ due to a number of reasons. Among these reasons, design and efficiency of the implementation as well the technology and hardware are likely the dominant factors.

A benchmark was designed to evaluate the performance of the cluster packages. A performance evaluation of the packages for grid computing and multi-core systems was not possible, mainly due to unavailability or complex system requirements of the packages. The package **R/parallel** was excluded from the benchmark, because the benchmark does not include a component for loops without data dependencies. The package **pnmath** was excluded because in the benchmark no NMATH routines were used.

The benchmark analysis was further restricted by focusing on packages providing functionality that allows all benchmark code to be run in a meaningful fashion. The packages **snowFT**, **papply**, **biopara**, **taskPR** and **rpvm** were excluded from the benchmark. The package **biopara** only supports socket connections, the package **taskPR** only supports the LAM/MPI implementation and both packages are unstable. The package **rpvm** does not have scripts to launch R instances at the slaves. There is only a function (`.PVM.spawnR()`) to run a R script file at the workers, therefore data can not be stored in R worker instances. The packages **snowFT** and **papply** are add-on packages which use the same communication mechanism as **snow** and **Rmpi**. They include additional functionality for better usability or error handling. Therefore they execute additional checks that reduce performance. The same behavior can be found for the **snowfall** package. For these packages all time measurements for the three components tests are about 10% slower than the computations with the **snow** package.

The benchmark comprises three different components: Sending data from the manager to all workers, distributing a list of data from the manager to the workers and a classic parallel computation example from the numerical analysis literature, the integration of a three-dimensional function. Details and code are provided in Appendix B. The computation times for each package and scenarios are summarized in Table 5.

Component test 1 sends a lot of data to all workers. Data transmission is relatively expensive in terms of time. In the **Rmpi** package the `mpi.bcast()` command is used for sending data to the workers. This uses a very efficient tree-like algorithm to broadcast the data. All other packages send the data in a serial loop over all slaves from the manager to the workers. One data sending operation must finish before the next starts (the `snow.time()` function in **snow** helps to visualize this process; see Figure 1).

The package **nws** sends every communication through the NetWorkSpace server. Component 1 and 2 show a bottleneck due to this communication model. Even in component 3 the **nws** implementations are the slowest solutions.

For component 3 there are three different ways for a serial implementation: using nested for-loops (75.6 sec), using vectorization (0.01 sec) and using the `apply()` function (21.1 sec). As mentioned in Section 2.1, improvements in the serial version of the code are of great importance. For the parallel implementation an apply-like implementation was used (see Appendix B.4). This implementation is comparable to the serial `apply()` function. The parallelization achieves an improvement of a factor 10 compared to the serial `apply()` function.

	Component 1	Component 2	Component 3
Rmpi	0.91	1.83	3.00
nws	14.87	3.49	3.03
snow			
MPI	10.33	2.00	2.96
PVM	4.13	1.01	2.93
NWS	8.68	1.58	2.98
Socket	3.47	0.93	2.90
snowfall			
MPI	13.99	2.07	2.93
PVM	4.29	0.94	2.93
NWS	8.82	1.62	2.99
Socket	3.77	1.00	2.91

Table 5: Computing time in seconds of the performance evaluation of R packages for computer clusters.

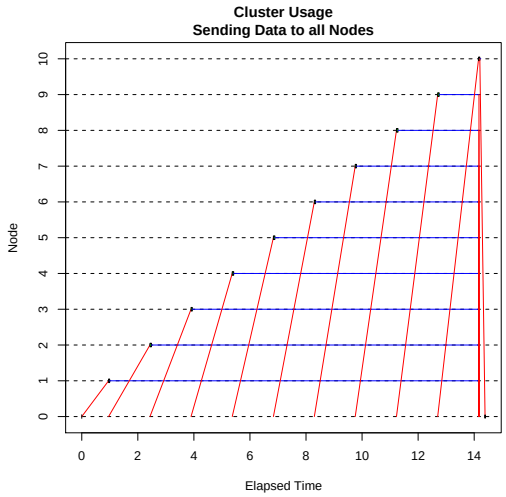

Figure 1: Visualization of timing information for cluster usage for component test 1 with the **snow** package using MPI. Green rectangles representing active computation (in this case very small), blue horizontal lines representing a worker waiting to return a result, and red lines representing master-slave communications.

The package **snow** (when using sockets) is the fastest solutions for parallel computing with R. If the ratio of communication and calculation time is good (see component test 3 and Appendix Figure 3), all packages have similar performance.

5. Application Examples and Tips for Parallel Computing

In this section we briefly describe two application packages which use the technologies that were discussed above. The section ends with some general tips for parallel computing in R.

5.1. Application Examples

These applications examples serve as concrete illustration for the increased efficiency of parallel computing with R. Both packages are available via the Bioconductor repository (`http://bioconductor.org`).

affyPara

The package **affyPara** (Schmidberger and Mansmann 2008) contains parallelized functions for preprocessing of exploratory oligonucleotide microarray analysis. The package is designed for large numbers (>200) of microarrays. It distributes arrays to different workers and accelerates the standard preprocessing methods by up to a factor of 15. The package is based on the cluster package **snow** and extends the package **affy**. Users switching from **affy** to **affyPara** must make only minor code changes. One limitation in the design of **affyPara** is that it re-implements rather than reuses **affy**; changes in **affy** require implementation of corresponding changes in **affyPara**.

ShortRead

The package **ShortRead** (Morgan *et al.* 2008) implements base classes, functions, and methods for representation of high-throughput, short-read sequencing data. Raw data from single experiments is very large (e.g., 10's of GB). Considerable computational processing can be required to summarize this data for down-stream analysis. If a running **Rmpi** cluster is available, **ShortRead** distributes the data processing across available workers. The only user-level requirement for accessing this performance benefit is to load the **Rmpi** package. **ShortRead** illustrates how parallel processing can be into a R packages without requiring extensive user participation.

A number of other application packages have similar integration of parallel computing in R. All of these offer good performance improvements using parallel computing techniques. In general, the integration of serial and parallel code into one package is preferred as it provides the possibility of using the packages on both a single-processor machine and in parallel environments, without requiring duplicate code development.

5.2. Tips for Parallel Computing with R

In closing this section, it is appropriate to reiterate some generally-accepted best practices from parallel computing:

- Communication is much slower than computation; care should be taken to minimize

unnecessary data transfer to and from workers, and to maximize the amount of computation performed in each remote evaluation.

  - Some functions produce large results, so it pays to reduce these on the workers before returning.
  - Very large objects, e.g., provided as additional `apply`-like function parameters, might in some cases be more efficiently constructed on each worker, rather than being forwarded from the manager to each worker.

- Random number generators require extra care. It is important to avoid producing the same random number stream on each worker, and to facilitate reproducible research. Special-purpose packages (**rsprng**, **rlecuyer**) are available; the **snow** package provides an integrated uniform random number generator. An overview of random number generator packages, with hyperlinks, is available in Appendix Table 8. Reproducible research requires the ability to partition and schedule jobs across the same number of workers, and in a reproducible order. The `apply.seq` argument of `apply`-like functions in the **Rmpi** package provides facilities for achieving this scheduling.

- R's rules for lexical scoping, its serializing functions and the environments they are defined in all require some care to avoid transmitting unnecessary data to workers. Functions used in `apply`-like calls should be defined in the global environment, or in a package name space.

The enthusiasm for high performance computing in R has lead the user community to create a new mailing list, starting in October 2008 (https://stat.ethz.ch/mailman/listinfo/r-sig-hpc). The list is the recommended place for questions, and problems using packages for parallel computing with R should be submitted to this list.

6. Summary and Outlook

In this section, we provide a brief summary of the 'state of the art' of parallel programming with R. We also provide an outlook about possible future developments.

6.1. State of the Art

As we discussed in Section 4, two well developed R packages stand out for use in high-performance multicomputer environments: **Rmpi** and **snow**. Both have acceptable usability, support a spectrum of functionality for parallel computing with R, and deliver good performance. Other packages try to improve usability, but so far usability gains have usually been achieved at the expense of lower functionality.

Packages to harness the power of multi-core architectures for the R language exist. These packages are still in development and can be difficult to use. At the moment the best integration of R to multi-core systems is the use of external and architecture optimized libraries (PBLAS). But linear algebra operations are not commonly the bottleneck in statistical computation. Another opportunity is the use of multicomputer technologies on multi-core systems. The packages **Rmpi** and **snow** can use several R instances started at one machine, using sockets or MPI for communication. The operating system allows each R instance to run on its own

processing unit. However, every R instance requires its own main memory, and the amount of main memory will frequently limit scalability.

Early-stage packages for grid computing are available. Further developments hinge on a clearer understanding of how grid infrastructure can be leveraged for statistical applications.

6.2. Further Developments

R packages for multicomputer environments are well developed and very useful. There is room for minor improvements in documentation (e.g., adding vignettes to aid new users in establishing simple clusters) and performance optimization. The most exciting prospects for new development are in areas other than the multicomputer environment covered so well by **snow** and **Rmpi**.

Today multi-core systems are a standard for every workstation, and the number of processors per chip is growing. There is a compelling need for integration of R code into multi-core environments. The **pnmath** packages are a first step in this direction. This approach has the potential to speed up the code, yielding a factor of improvement given by the number of available processors. Enabling multi-core functionality as implemented in **pnmath** requires manual adjustment of all C-level functions. To design a package to support a more abstract use of multi-core systems for package development seems to be a very difficult task; the interactive nature of R and existing multi-core technology implies runtime compilation.

From an end-user perspective, computer cluster are often unavailable, the physical costs (acquisition, maintenance, etc.) are high, and effective use requires a diverse skill set (e.g., advanced user-level system configuration, mastery of batch job submission, careful formulation of problems to effectively manage communication costs). Therefore it is difficult to use this technology. Cloud computing (Vouk 2008) allows users to access services from the Internet without knowledge of the technology infrastructure that supports them. These web services provide resizable computer capacity in the cloud. There are several companies which offer computing services for relatively modest costs. Cloud computing usually offers pre-defined 'machine images' that contain OpenMPI support, and that can be extended with the latest R version and packages. An exciting prospect would be the development of a new R package for communication with the cloud, a web portal to control access, and an appropriate business model for managing costs. Such a cloud computing package could provide cost-effective cluster-based computing to end users.

In terms of hardware, the (parallel) computing power of graphic processing units (GPUs) might provide significant performance benefits to R users. Following on the earlier releases of vendor-specific APIs (e.g. nVidia (2007) proposed CUDA), standards for programming GPUs in a vendor-independent fashion are beginning to emerge (Khronos Group 2008). They offer a programming model that is designed to allow direct access to the specific graphics hardware, with the graphics hardware running a very high number of threads in parallel. A bioinformatics application for sequence alignment with GPUs (C code, no R integration) has been published by Manavski and Valle (2008) and illustrates the usability of GPUs for acceleration and management of large amounts of biological data.

The flexibility of the R package system allows integration of many different technologies to address practical statistical questions. This provides an excellent opportunity to explore and prototype how different approaches can be effectively integrated into end-user software. This leads us to expect that whichever technologies emerge, R should be well-positioned to take

advantage of them.

References

Binstock A (2008). "Threading Models for High-Performance Computing: Pthreads or OpenMP?" *Intel software network*, Pacific Data Works LLC. URL http://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmp.

Bjornson R, Carriero N, Weston S (2007). "Python NetWorkSpaces and Parallel Programs." *Dr. Dobb's Journal*, pp. 1–7. URL http://www.ddj.com/web-development/200001971.

Breshears CP, Luong P (2000). "Comparison of openmp and pthreads within a coastal ocean circulation model code." In "Workshop on OpenMP Applications and Tools," .

Buntinas D, Mercier G, Gropp W (2007). "Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem." *Parallel Comput.*, **33**(9), 634–644. ISSN 0167-8191.

Butenhof DR (1997). *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-63392-2.

Chambers JM (2008). *Software for Data Analysis: Programming with R*. Statistics and Computing. Springer.

Chine K (2007). *Biocep: a federative collaborative user-centric and cloud-enabled computational open platform for e-Research*. Cloud Era Ltd, Cambridge, UK. URL http://www.biocep.net.

Dagum L, Menon R (1998). "OpenMP: an industry standard API for shared-memory programming." *IEEE Computational Science and Engineering*, **5**(1), 46–55.

Deino Software (2006). *DeinoMPI – User Manual*, version 1.1.0 edition. URL http://mpi.deino.net/DeinoMPI.pdf.

Dongarra J, Foster I, Fox G, Gropp W, Kennedy K, Torczon L, White A (eds.) (2003). *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-871-0.

Eddelbuettel D (2008). "Introduction to High-Performance R." Tutorial presented at the UseR 2008 conference. URL http://dirk.eddelbuettel.com/papers/useR2008introhighperfR.pdf.

Forum MPI (1998). "MPI2: A Message Passing Interface standard." In "International Journal of High Performance Computing Applications," volume 12, pp. 1–299.

Foster IT (2005). "Globus Toolkit Version 4: Software for Service-Oriented Systems." In H Jin, DA Reed, W Jiang (eds.), "NPC," volume 3779 of *Lecture Notes in Computer Science*, pp. 2–13. Springer.

Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS (2004). "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation." In "Proceedings, 11th European PVM/MPI Users' Group Meeting," pp. 97–104. Budapest, Hungary. URL http://www.open-mpi.org.

Geist A, Beguelin A, Dongarra J, Jiang W, Manchek B, Sunderam V (1994). *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge.

Gentleman R (2008). *R Programming for Bioinformatics*, volume 12 of *Chapman & Hall/CRC Computer Science & Data Analysis*. Chapman & Hall/CRC.

Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J, Hornik K, Hothorn T, Huber W, Iacus S, Irizarry R, Leisch F, Li C, Maechler M, Rossini AJ, Sawitzki G, Smith C, Smyth G, Tierney L, Yang JY, Zhang J (2004). "Bioconductor: Open software development for computational biology and bioinformatics." *Genome Biology*, **5**. URL http://genomebiology.com/2004/5/10/R80.

Gropp W, Lusk E (2002). "Goals Guiding Design: PVM and MPI." In "CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing," p. 257. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-1745-5.

Grose DJ (2008). "Distributed Computing using the multiR Package." In "UseR 2008 - Book of Abstracts," .

Hayes M, Morris L, Crouchley R, Grose D, van Ark T, Allan R, Kewley J (2005). "GROWL: A Lightweight Grid Services Toolkit and Applications." In "4th UK e-Science All Hands Meeting," Nottingham, UK.

Hey T, Trefethen A (2003). "Grid Computing: Making the Global Infrastructure a Reality." *Wiley Series in Communications Networking & Distributed Systems*, pp. 1–17.

Jamitzky F (2008). "ROMP - an OpenMP binding for R." In "UseR 2008 - Book of Abstracts," .

Jette MA, Yoo AB, Grondona M (2003). "SLURM: Simple Linux Utility for Resource Management." In "Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003," pp. 44–60. Springer-Verlag.

Khronos Group (2008). "OpenCL: The Open Standard for Heterogeneous Parallel Programming." Overview presentation. URL http://www.khronos.org/developers/library/overview/opencl_overview.pdf.

Knaus J (2008). "sfCluster/snowfall: Managing parallel execution of R programs on a compute cluster." In "UseR 2008 - Book of Abstracts," .

Li MN, Rossini A (2001). "RPVM: Cluster Statistical Computing in R." *R News*, **1**(3), 4–7. URL http://CRAN.R-project.org/doc/Rnews/.

Ligges U (ed.) (2008). *UseR! 2008 - Book of Abstracts*. Technische Universität Dortmund. URL http://www.statistik.uni-dortmund.de/useR-2008/abstracts/_Abstracts.pdf.

Manavski SA, Valle G (2008). "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment." *BMC Bioinformatics*, **9 Suppl 2**, S10.

Morgan M, Lawrence M, Anders S (2008). "ShortRead: Base classes and methods for high-throughput short-read sequencing data." BioConductor package. URL http://bioconductor.org/packages/2.3/bioc/html/ShortRead.html.

nVidia (2007). *nVidia Compute Unified Device Architecture (CUDA) Programming Guide, version 1.0*. nVidia.

Pukacki J, Kosiedowski M, Mikolajczak R, Adamski M, Grabowski P, Jankowski M, Kupczyk M, Mazurek C, Meyer N, Nabrzyski J, Piontek T, Russell M, Stroinski M, Wolski M (2006). "Programming Grid Applications with Gridge." *Computational Methods in Science and Technology*, **12**.

R Development Core Team (2008a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

R Development Core Team (2008b). "Writing R Extensions." *Technical report*, R Foundation for Statistical Computing, Vienna, Vienna, Austria. ISBN 3-900051-11-9.

Rossini A, Tierney L, Li NM (2003). "Simple Parallel Statistical Computing in R." *UW Biostatistics Working Paper Series*, **193**.

Rossini AJ, Tierney L, Li NM (2007). "Simple Parallel Statistical Computing in R." *Journal of Computational and Graphical Statistics*, **16**(2), 399–420.

Schmidberger M, Mansmann U (2008). "Parallelized preprocessing algorithms for high-density oligonucleotide arrays." In "Proc. IEEE International Symposium on Parallel and Distributed Processing IPDPS 2008," pp. 1–7.

Sevcikova H (2003). "Statistical Simulations on Parallel Computers." *Journal of Computational and Graphical Statistics*, **13**(4), 886–906. URL http://www.stat.washington.edu/hana.

Shaffer C (2007). "Next-generation sequencing outpaces expectations." *Nat Biotechnol*, **25**(2), 149.

Shneiderman B, Plaisant C (2004). *Designing the User Interface : Strategies for Effective Human-Computer Interaction (4th Edition)*. Addison Wesley. ISBN 0321197860.

Sloan JD (2004). *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI*. O'Reilly.

Sun Microsystems I (2002). *Sun^{TM} ONE Grid Engine Administration and User's Guide*. 4150 Network Circle Santa Clara, CA 95054 U.S.A., sun grid engine 5.3 edition. URL http://gridengine.sunsource.net/download/Manuals53/SGE53AdminUserDoc.pdf?content-type=application/pdf.

Tannenbaum T, Wright D, Miller K, Livny M (2001). "Condor – A Distributed Job Scheduler." In T Sterling (ed.), "Beowulf Cluster Computing with Linux," MIT Press.

Tierney L (2007). *proftools: Profile Output Processing Tools for R*. R package.

Tierney L (2008). "Code analysis and parallelizing vector operations in R." *Computational Statistics*, **accepted for publication**.

Venables W (2001). *Programmer's Niche. R News, 1(1):27-30.* URL http://cran.r-project.org/doc/Rnews.

Vera G, Jansen RC, Suppi RL (2008). "R/parallel - speeding up bioinformatics analysis with R." *BMC Bioinformatics*, **9**, 390+. ISSN 1471-2105.

Vouk M (2008). "Cloud computing - Issues, research and implementations." In "Information Technology Interfaces," .

Warnes GR (2007). "fork: R functions for handling multiple processes." CRAN package. URL http://cran.r-project.org/web/packages/fork.

Wegener D, Sengstag T, Sfakianakis S, Rüping S, Assi A (2007). "GridR: An R-based grid-enabled tool for data analysis in ACGT clinico-genomic trials." In "Proceedings of the 3rd International Conference on e-Science and Grid Computing (eScience 2007)," Bangalore, India.

Wickham H (2008). *profr: An alternative display for profiling information*. R package. URL http://had.co.nz/profr.

Yu H (2002). "Rmpi: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL http://CRAN.R-project.org/doc/Rnews/.

Affiliation:

Markus Schmidberger
Division of Biometrics and Bioinformatics, IBE
Ludwig-Maximilians-Universität Munich
81377 Munich, Germany
E-mail: Markus.Schmidberger@ibe.med.uni-muenchen.de
URL: http://www.ibe.med.uni-muenchen.de

A. R Code Examples for Parallel Computing Packages

This appendix gives some short code examples for all presented R packages, to introduce the reader to the basic functionality. The code chunks do not cover the full functionality of the packages. For further details and more functions see the R help files or the vignettes of the packages. For details starting the corresponding technology, please contact your cluster administrator or see the manuals.

The R help files contain descriptions for all R functions. To get more information on any specific named function, for example `mpi.spawn.Rslaves()`, the command is

```
> library("Rmpi")
> help(mpi.spawn.Rslaves)
```

Vignettes contain executable examples and are intended to be used interactively. You can access the PDF version of a vignette for any installed package from inside R as follows:

```
> library("Rmpi")
> vignette()
```

This will present a menu where to select the desired vignette. Selecting a vignette should cause the corresponding PDF file to open on the system.

A.1. R Packages for Computer Cluster

Rmpi: R package for MPI

Depending of the used MPI implementation and resource manager there are different ways for starting R with connection to the running MPI universe. Using Unix the commands for LAM/MPI and openMPI in the latest version look like the following examples:

LAM/MPI Start LAM universe: `lamboot HOSTFILE`
Start R: `R`
Stop LAM universe: `lamhalt`

openMPI The MPI universe (openMPI daemons) should be running for default.
Start R: `orterun -n 1 -hostfile HOSTFILE R -no-save`

Start cluster R code to start or initialize the cluster:

```
> library("Rmpi")
> ## Starting 4 workers and initialize the master
> mpi.spawn.Rslaves(nslaves=4)
```

Sending data to all slaves R code to send data `a` and `b` to all slaves:

```
> a <- 3
> b <- 1:10
> mpi.bcast.Robj2slave(a)
> mpi.bcast.Robj2slave(b)
```

Calculations at all slaves R code to calculate `a+5` and `sin(5)` at all slaves:

```
> mpi.remote.exec(a+5)
> mpi.remote.exec(sin, 5)
```

Calculations at list element distributed to slaves R code to calculate `sqrt(3)` at first slave, `sqrt(6)` at second slave and `sqrt(9)` and third slave.

```
> mpi.apply(c(3,6,9), sqrt)
```

Stop cluster R code to stop cluster and to close R instances at workers.

```
> mpi.close.Rslaves()
```

rpvm: R package for PVM

Start pvm: `pvm HOSTFILE`
Exit pvm console: `pvm> quit`
Start R: `R`
Stop pvm: `pvm> halt`

Start cluster There is no command to lunch R instances at the workers. There is only a command to execute R script files at the slaves.

```
> library("rpvm")
> slaves <- .PVM.spawnR(R_Script_File,ntask=4)
```

For example the 'R instance lunch files' from the package **snow** can be used. But it is easier to use the package **snow** with PVM as communication layer.

The following examples are code chunks which could be used for R script files.

Sending data to all slaves

```
> TAG <- 22
> a <- 3
> b <- 1:10
> for (id in 1:length(slaves)) {
+         .PVM.initsend()
+         .PVM.serialize( assign('a',a, envir=.GlobalEnv) )
+         .PVM.send(slaves[[id]], TAG)
+          cat("Work sent to", slaves[id], "\n")
+          .PVM.recv(slaves[[id]], TAG)
+ }
> for (id in 1:length(slaves)) {
+         .PVM.initsend()
+         .PVM.serialize( assign('b',b, envir=.GlobalEnv) )
```

```
+         .PVM.send(slaves[[id]], TAG)
+         cat("Work sent to", slaves[id], "\n")
+         .PVM.recv(slaves[[id]], TAG)
+ }
```

Calculations at all slaves

```
> for (id in 1:length(slaves)) {
+         .PVM.initsend()
+         .PVM.serialize( get('a', envir=.GlobalEnv) + 5 )
+         .PVM.send(slaves[[id]], TAG)
+         cat("Work sent to", slaves[id], "\n")
+ }
> results <- list()
> for (id in 1:length(slaves)) {
+         .PVM.recv(slaves[[id]], TAG)
+          results[id]=.PVM.unserialize()
+ }
> print(results)
> for (id in 1:length(slaves)) {
+         .PVM.initsend()
+         .PVM.serialize( sin(5) )
+         .PVM.send(slaves[[id]], TAG)
+         cat("Work sent to", slaves[id], "\n")
+ }
> results <- list()
> for (id in 1:length(slaves)) {
+         .PVM.recv(slaves[[id]], TAG)
+         results[id]=.PVM.unserialize()
+ }
> print(results)
```

Calculations at list element distributed to slaves

```
> array <- c(3,6,9)
> for (id in 1:length(array)) {
+         .PVM.initsend()
+         .PVM.serialize( sqrt( array[id] ) )
+         .PVM.send(slaves[[id]], TAG)
+         cat("Work sent to", slaves[id], "\n")
+ }
> results <- list()
> for (id in 1:length(array)) {
+         .PVM.recv(slaves[[id]], TAG)
+         results[id]=.PVM.unserialize()
+ }
> print(results)
```

Stop cluster

```
> .PVM.exit()
```

nws: R package for Net Work Spaces

Start nws: `twistd -y nws.tac`
Start R: `R`
Stop nws: `kill 'cat twistd.pid'`

Start cluster

```
> library("nws")
> ws <- netWorkSpace('test')
> system("cat HOSTFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> ## Starting 4 workers and initialize master and
> ## removing first workername from list. In most cases this is the master,
> ## we do not want to have a slave running at the same machine than the master.
> s <- sleigh(nodeList=mycluster[-1],  launch=sshcmd, workerCount=4)
```

Sending data to all slaves Using the package **nws** you have to store the data in the network space.

```
> a <- 3
> b <- 1:10
> nwsStore(ws, 'a', a)
> nwsStore(ws, 'b', b)
```

To get data from the network space call:

```
> a <- nwsFindTry(ws, 'a')
> b <- nwsFindTry(ws, 'b')
```

Calculations at all slaves

```
> eachWorker(s, function(host, name){
+                         ws <- netWorkSpace(serverHost=host, name)
+                         a <- nwsFindTry(ws, 'a')
+                         return(a+5)
+                 }, ws@server@serverHost, ws@wsName)
> eachWorker(s,sin,5)
```

Calculations at list element distributed to slaves

```
> eachElem(s, sqrt, c(3,6,9))
```

Stop cluster

```
> stopSleigh(s)
```

snow: Simple Network of Workstations

Start the underlying technology you want to use. For details see the other sections.

Start cluster

```
> library("snow")
> ## Starting 4 workers and initialize master
> #c1 <- makePVMcluster(4)
>
> #system("cat HOSTFILE > cluster.txt")
> #mycluster <- scan(file="cluster.txt" ,what="character")
> ## Starting 4 workers and initialize master and
> ## removing first workername from list. In most cases this is the master,
> ## we do not want to have a slaves running at the same machine than the master.
> #c1 <- makeNWScluster(mycluster[2:5])
>
> #system("cat HOSTFILE > cluster.txt")
> #mycluster <- scan(file="cluster.txt" ,what="character")
> ## Starting 4 workers and initialize master and
> ## removing first workername from list. In most cases this is the master,
> ## we do not want to have a slaves running at the same machine than the master.
> #c1 <- makeSOCKcluster(mycluster[1:4])
>
> ## Starting 4 workers and initialize master
> c1 <- makeMPIcluster(4)
```

Sending data to all slaves

```
> a <- 3
> b <- 1:10
> clusterExport(cl, list("a", "b"))
```

Calculations at all slaves

```
> clusterEvalQ(cl, a+5)
> clusterCall(cl, sin, 5)
```

Calculations at list element distributed to slaves

```
> clusterApply(cl, c(3,6,9), sqrt)
```

Stop cluster

```
> stopCluster(cl)
```

snowFT: Simple Network of Workstations with Fault Tolerance

For details starting the pvm see Section A.1.2.

Start cluster

```
> library("snowFT")
> ## Starting 4 workers and initialize master
> cl <- makeClusterFT(4)
```

Sending data to all slaves

```
> a <- 3
> b <- 1:10
> clusterExport(cl, list("a", "b"))
```

Calculations at all slaves

```
> clusterEvalQ(cl, a+5)
> clusterCall(cl, sin, 5)
```

Calculations at list element distributed to slaves

```
> clusterApplyFT(cl, c(3,6,9), sqrt)
```

Stop cluster

```
> stopCluster(cl)
```

snowfall: Simple Network of Workstations with more userfriendly interface

Start the underlying technology you want to use. For details see the other sections.

Start cluster

```
> library("snowfall")
> ## Starting 4 workers and initialize master
> sfInit(parallel=TRUE, cpus=4, type='MPI')
```

Sending data to all slaves

```
> a <- 3
> b <- 1:10
> sfExport("a", "b")
```

Calculations at all slaves

```
> sfClusterEval(a+5)
> sfClusterCall(sin, 5)
```

Calculations at list element distributed to slaves

```
> sfClusterApply(c(3,6,9), sqrt)
```

Stop cluster

```
> sfStop()
```

papply

The package **papply** extends the package **Rmpi** with an improved `apply()` function. All functions of the **Rmpi** package can or have to be used.

Start cluster

```
> library("Rmpi")
> library("papply")
> ## Starting 4 workers and initialize the master
> mpi.spawn.Rslaves(nslaves=4)
```

Calculations at list element distributed to slaves

```
> papply(as.list(c(3,6,9)), sqrt)
```

Stop cluster

```
> mpi.close.Rslaves()
```

biopara

In the package **biopara** all R instances have to be started manually and biopara has to be initialized.

Start cluster

```
> #Start workers: 4 times
> library("biopara")
> biopara(37000,"/tmp","localhost",38000)
> #Start master
> library("biopara")
> slaves <- list( list("slave1",37000,"/tmp",38000,""),
+                 list("slave1",37000,"/tmp",38000,""),
+                 list("slave1",37000,"/tmp",38000,""),
+                 list("slave1",37000,"/tmp",38000,"") )
> biopara("master",36000,39000,slaves)
> #Start one more R instance for client
> library("biopara")
```

Sending data to all slaves

```
> out<-biopara(list("master",39000), list("localhost",40000), 4,
+                 assign('a', 3, envir=.GlobalEn) )
> out<-biopara(list("master",39000), list("localhost",40000), 4,
+                 assign('b', 1:10, envir=.GlobalEn) )
```

Calculations at all slaves

```
> out<-biopara(list("master",39000), list("localhost",40000), 4,
+                 get('a', envir=.GlobalEn)+5 )
> out<-biopara(list("master",39000), list("localhost",40000), 4,
+                 sin(5) )
```

Calculations at list element distributed to slaves not possible

Stop cluster Kill all R instances!

taskPR

Depending of the used MPI implementation and resource manager there are different ways for starting R with connection to the running MPI universe.

Start cluster

```
> library("taskPR")
> ## Starting 4 workers and initialize the master
> StartPE(4)
```

Sending data to all slaves

```
> PE(a <- 3, global=TRUE)
> PE(b <- 1:10, global=TRUE)
```

Calculations at all slaves

```
> PE(x <- a+5, global=TRUE)
> PE(y <- sin(5), global=TRUE)
> POBJ(x)
> print(x)
> POBJ(y)
> print(y)
```

Calculations at list element distributed to slaves

```
> sapply(c(3,6,9), function(x){
+                         PE(temp<-sqrt(x))
+                         POBJ(temp)
+                         return(temp)
+                 })
```

Stop cluster

```
> StopPE()
```

A.2. R Packages for Grid Computing

GridR

Start cluster / Grid

```
> library("GridR")
> grid.init()
```

Calculation somewhere in the Grid This is no parallel calculation!

```
> grid.apply("erg", sin, 5)
```

Parllel calculation in the Grid

```
> func <- function(x){
+         library("GridR")
+         grid.init(new-parameters)
+         grid.apply("result2", sin, 5)
+         return(result2)
+ }
> grid.apply("result", func, 5)
```

multiR

Up to the time of review, there was no code available.

Biocep-R

Biocep-R is a GUI project. Therefore there is no special code.

A.3. R Packages for Multi-core Systems

R/Parallel

Start cluster There is no start command required. You only have to load the package.

```
> library("rparallel")
```

Paralleization of a loop without data dependencies

```
> colSumsPara <- function( x )
+ {
+         result <- NULL
+         if( "rparallel" %in% names( getLoadedDLLs()) ) {
+                 runParallel( resultVar="result", resultOp="cbind", nWorkers=4 )
+         } else {
+                 for( idx in 1:dim(x)[2] ){
+                         tmp <- sum(x[,idx])
+                         result <- cbind(result,tmp)
+                 }
+                 return( result )
+         }
+ }
> x  <- matrix(sample(100), nrow=10)
> colSumsPara(x)
```

pnmath

Loading the packages they replace the builtin math functions by the parallel versions. The user can use the standard math functions without learning any new code syntax.

```
> library("pnmath")
> library("pnmath0")
```

romp

Up to the time of review, there was no package available. Only some example code without documentation was available.

B. R Code for Performance Evaluation

The benchmark contains three different components: Sending data from the master to all slaves, distributing a list of data from the master to the slaves and a small classic parallel computation example. For every component the execution time will be measured. The time for starting and stopping the cluster and as well for calculating results or parameters will not be measured. To avoid foreign network traffic or processor load the measurement will be done five times and the computation time will be averaged. The results are presented in Table 5.

B.1. Used Software and Hardware

For the benchmark the 'hoppy' cluster at the Fred Hutchinson Cancer Research Center in Seattle, WA, USA was used. For the evaluation ten workers were used and at every worker only one R instance was started. The cluster has the following hardware and software configuration:

- each worker: 4 GB main memory, 4 x Intel(R) Xeon(TM) CPU 3.60GHz, Suse Linux version 2.6.12
- R Version 2.8.0 and latest packages.
- PVM 3.4.5
- openMPI 1.2.4
- NWS Server 1.5.2 (located at the master)
- Resource Manager: Torque 2.3.2; Using Torque the HOSTFILE is stored in the variable PBS_NODEFILE.

B.2. Sending Data from the Master to all Slaves

In this test component a 'big' R object will be sent from the master to all slaves and saved in the Global Environment at the slaves. As 'big' R object a list with five 500×500 matrices is used.

```
> dim <- 500
> Robj <- rep( list( matrix(1,nrow=dim, ncol=dim) ), 5)
```

Rmpi

```
> library("Rmpi")
> mpi.spawn.Rslaves(nslaves=10)
> t <- replicate(5,system.time(mpi.bcast.Robj2slave(obj=Robj)))
> tint <- summary(t[3,])
> mpi.close.Rslaves()
```

nws

```
> library("nws")
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> s <- sleigh(nodeList=mycluster[-1], launch=sshcmd, workerCount=10)
> slavefunc <- function(x){ assign('Robj', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( eachWorker(s, slavefunc, Robj) ))
> tint <- summary(t[3,])
> stopSleigh(s)
```

snow

```
> library("snow")
```

snow with MPI

```
> c1 <- makeMPIcluster(10)
> t <- replicate(5, system.time( clusterExport(c1, list("Robj")) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snow with PVM

```
> c1 <- makePVMcluster(10)
> t <- replicate(5, system.time( clusterExport(c1, list("Robj")) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snow with NWS

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeNWScluster(mycluster[2:11])
> t <- replicate(5, system.time( clusterExport(c1, list("Robj")) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snow with SOCKET

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeSOCKcluster(mycluster[2:11])
> t <- replicate(5, system.time( clusterExport(c1, list("Robj")) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snowfall

```
> library("snowfall")
```

snowfall with MPI

```
> sfInit(parallel=TRUE, cpus=10, type="MPI")
> slavefunc <- function(x){ assign('Robj', x, envir=.GlobalEnv); return(NULL)}
> t <- replicate(5, system.time( sfClusterCall(slavefunc, Robj) ))
> tint <- summary(t[3,])
> sfStop()
```

snowfall with PVM

```
> sfInit(parallel=TRUE, cpus=10, type="PVM")
> slavefunc <- function(x){ assign('Robj', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterCall(slavefunc, Robj) ))
> tint <- summary(t[3,])
> sfStop()
```

snowfall with NWS

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> sfInit(parallel=TRUE, cpus=10, type="NWS", socketHosts=mycluster[2:11])
> slavefunc <- function(x){ assign('Robj', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterCall(slavefunc, Robj) ))
> tint <- summary(t[3,])
> sfStop()
```

snowfall with SOCKET

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> sfInit(parallel=TRUE, cpus=10, type="SOCK", socketHosts=mycluster[2:11])
> slavefunc <- function(x){ assign('Robj', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterCall(slavefunc, Robj) ))
> tint <- summary(t[3,])
> sfStop()
```

snowFT, papply, biopara, taskPR, rpvm These packages were excluded from the benchmark. **snowFT** and **papply** are only add on packages. **biopara** only supports socket connections and **taskPR** only supports the LAM/MPI implementation. The package **rpvm** does not have scripts to lunch R instances at the workers.

B.3. Distributing a List of Data from the Master to the Slaves

In this test component a list of R objects will be distributed from the master to the slaves and saved in the global environment at the slaves. Therefore a list with 10 (number of workers) elements will be created. Each element is a 500×500 matrix. Matrix 1 goes to slave 1, matrix 2 goes to slave 2 and so on.

```
> dim <- 500
> Robj <- rep( list( matrix(1,nrow=dim, ncol=dim) ), 10)
```

Rmpi

```
> library("Rmpi")
> mpi.spawn.Rslaves(nslaves=10)
> # the function mpi.scatter.Robj2slav is available in Rmpi_0.5-6
> t <- replicate(5,system.time(mpi.scatter.Robj2slave(Robj)))
> tint <- summary(t[3,])
> mpi.close.Rslaves()
```

nws

```
> library("nws")
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> s <- sleigh(nodeList=mycluster[-1],  launch=sshcmd, workerCount=10)
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( eachElem(s, slavefunc, Robj) ))
> tint <- summary(t[3,])
> stopSleigh(s)
```

snow

```
> library("snow")
```

snow with MPI

```
> c1 <- makeMPIcluster(10)
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( clusterApply(c1, Robj, slavefunc) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snow with PVM

```
> c1 <- makePVMcluster(10)
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( clusterApply(c1, Robj, slavefunc) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snow with NWS

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeNWScluster(mycluster[2:11])
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( clusterApply(c1, Robj, slavefunc) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snow with SOCKET

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeSOCKcluster(mycluster[2:11])
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( clusterApply(c1, Robj, slavefunc) ))
> tint <- summary(t[3,])
> stopCluster(c1)
```

snowfall

```
> library("snowfall")
```

snowfall with MPI

```
> sfInit(parallel=TRUE, cpus=10, type="MPI")
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterApply(Robj, slavefunc) ))
> tint <- summary(t[3,])
> sfStop()
```

snowfall with PVM

```
> sfInit(parallel=TRUE, cpus=10, type="PVM")
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterApply(Robj, slavefunc) ))
> tint <- summary(t[3,])
> sfStop()
```

snowfall with NWS

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> sfInit(parallel=TRUE, cpus=10, type="NWS", socketHosts=mycluster[2:11])
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterApply(Robj, slavefunc) ))
> tint <- summary(t[3,])
> sfStop()
```

snowfall with SOCKET

```
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> sfInit(parallel=TRUE, cpus=10, type="SOCK", socketHosts=mycluster[2:11])
> slavefunc <- function(x){ assign('x', x, envir=.GlobalEnv); return(NULL) }
> t <- replicate(5, system.time( sfClusterApply(Robj, slavefunc) ))
> tint <- summary(t[3,])
> sfStop()
```

B.4. Small classic Parallel Computation Example

In this test component the integral of a three-dimensional function should be calculated. Integration of a function over a defined region is a very common example for parallel computing.

Problem formulation

```
> xint <- c(-1,2)
> yint <- c(-1,2)
> func <- function(x,y) x^3-3*x + y^3-3*y
> library(lattice)
> g <- expand.grid(x = seq(xint[1],xint[2],0.1), y = seq(yint[1],yint[2],0.1))
> g$z <- func(g$x,g$y)
```

Figure 2 shows the surface of the function. The exact solution of the integral can be analytically calculated.

$$\begin{aligned}\int\int_{[-1,2]\times[-1,2]} x^3 - 3x + y^3 - 3y \; dxdy &= \int_{[-1,2]} \left[\frac{1}{4}x^4 - \frac{3}{2}x^2 + xy^3 - 3xy\right]_{x=-1}^{x=2} dy \\ &= \int_{[-1,2]} -\frac{3}{4} + 3y^3 - 9y \; dy \\ &= \left[-\frac{3}{4}y + \frac{3}{4}y^4 - \frac{9}{2}y^2\right]_{x=-1}^{x=2} = -4.5\end{aligned}$$

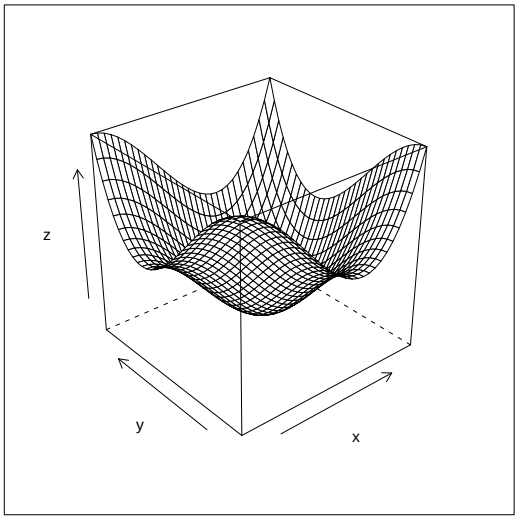

Figure 2: Surface of the example equation for test component 3.

Implementation

The volume under the function has to be split in several parts. Than the volume of the rectangular boxes can be computed. Width and depth are the size of the split intervals and the high is the function value (in the middle or at a corner). Summarizing the values returns the full integral. The more boxes will be calculated (smaller boxes), the more exact the calculation gets.

Serial code

There are principle three ways to implement the integration in serial.

Serial calculation of the Integral with 'for' loops

```
> integLoop <- function(func, xint, yint, n)
+ {
+         erg <- 0
+         # interval sizes
+         xincr <- ( xint[2]-xint[1] ) / n
+         yincr <- ( yint[2]-yint[1] ) / n
+         for(xi in seq(xint[1],xint[2], length.out=n)){
+                 for(yi in seq(yint[1],yint[2], length.out=n)){
+                         # Calculating one rectangular box
+                         box <- func(xi, yi) * xincr * yincr
+                         # Summarizing
+                         erg <- erg + box
+                 }
+         }
```

```
+         return(erg)
+ }
```

Serial calculation of the Integral with vectorization

```
> integVec <- function(func, xint, yint, n)
+ {
+         xincr <- ( xint[2]-xint[1] ) / n
+         yincr <- ( yint[2]-yint[1] ) / n
+         # using vectors instead of loops
+         erg <- sum( func( seq(xint[1],xint[2], length.out=n),
+                 seq(yint[1],yint[2], length.out=n) ) ) * xincr * yincr * n
+         return(erg)
+ }
```

Serial calculation of the Integral with apply functions

```
> integApply <- function (func, xint, yint, n)
+ {
+         applyfunc <- function(xrange, xint, yint, n, func)
+         {
+                 # calculates for every x interval the complete volume
+                 yrange <- seq(yint[1],yint[2], length.out=n)
+                 xincr <- ( xint[2]-xint[1] ) / n
+                 yincr <- ( yint[2]-yint[1] ) / n
+                 erg <- sum( sapply(xrange, function(x) sum( func(x,yrange) )) ) * xincr
+                 return(erg)
+         }
+         xrange <- seq(xint[1],xint[2],length.out=n)
+         erg <- sapply(xrange, applyfunc, xint, yint, n, func)
+         return( sum(erg) )
+ }
```

Parallel code

Slavefunction which will be executed at every slave and calculates the area under the function. This slavefunction will be used for every R package.

```
> slavefunc<- function(id, nslaves, xint, yint, n, func){
+     xrange <- seq(xint[1],xint[2],length.out=n)[seq(id,n,nslaves)]
+     yrange <- seq(yint[1],yint[2], length.out=n)
+     xincr <- ( xint[2]-xint[1] ) / n
+     yincr <- ( yint[2]-yint[1] ) / n
+     erg <- sapply(xrange, function(x)sum( func(x,yrange ) ))* xincr * yincr
+         return( sum(erg) )
+ }
```

R code for different cluster packages. In principle all the same, only the different syntax of the parallel apply function will be used.

Rmpi

```
> integRmpi <- function (func, xint, yint, n)
+ {
+         nslaves <- mpi.comm.size()-1
+         erg <- mpi.parSapply(1:nslaves, slavefunc, nslaves, xint, yint, n, func)
+         return( sum(erg) )
+ }
```

nws

```
> integNWS <- function(sleigh, func, xint, yint, n)
+ {
+         nslaves <- status(s)$numWorkers
+         erg <- eachElem(sleigh, slavefunc, list(id=1:nslaves),
+                         list(nslaves=nslaves, xint=xint, yint=yint, n=n, func=func))
+         return( sum(unlist(erg)) )
+ }
```

snow

```
> integSnow <- function(cluster, func, xint, yint, n)
+ {
+         nslaves <- length(cluster)
+         erg <- clusterApplyLB(cluster, 1:nslaves, slavefunc, nslaves, xint, yint, n, fun
+         return( sum(unlist(erg)) )
+ }
```

snowfall

```
> integSnowFall <- function(func, xint, yint, n)
+ {
+         nslaves <- sfCpus()
+         erg <- sfClusterApplyLB(1:nslaves, slavefunc, nslaves, xint, yint, n, func)
+         return( sum(unlist(erg)) )
+ }
```

Results comparison

R code to calculate the results of the integral.

```
> func <- function(x,y) x^3-3*x + y^3-3*y
> xint <- c(-1,2)
```

```
> yint <- c(-1,2)
> n <- 10000
> #serial
> erg <- integLoop(func, xint, yint, n)
> erg <- integVec(func, xint, yint, n)
> erg <- integApply(func, xint, yint, n)
> #parallel MPI
> library(Rmpi)
> mpi.spawn.Rslaves(nslaves=10)
> erg <- integRmpi(func, xint, yint, n)
> mpi.close.Rslaves()
> #parallel NWS
> library("nws")
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> s <- sleigh(nodeList=mycluster[-1],  launch=sshcmd, workerCount=10)
> erg <- integNWS(s, func, xint, yint, n)
> stopSleigh(s)
> #parallel SNOW
> library(snow)
> c1 <- makeMPIcluster(10)
> erg <- integSnow(c1, func, xint, yint, n)
> stopCluster(c1)
> #parallel snowfall
> library(snowfall)
> sfInit(parallel=TRUE, cpus=10, type="MPI")
> erg <- integSnowFall(func, xint, yint, n)
> sfStop()
```

Time measurement

R code to run the time measurements.

Time measurement for serial code

```
> rep <- 5
> tLoop <- replicate(rep, system.time( integLoop(func, xint, yint, n) ))
> summary(tLoop[3,])
> tVec <- replicate(rep, system.time( integVec(func, xint, yint, n) ))
> summary(tVec[3,])
> tApply <- replicate(rep, system.time( integApply(func, xint, yint, n) ))
> summary(tApply[3,])
```

Time measurement for parallel code

```
> #parallel MPI
> library(Rmpi)
```

```
> mpi.spawn.Rslaves(nslaves=10)
> tRmpi <- replicate(rep, system.time( integRmpi(func, xint, yint, n) ))
> summary(tRmpi[3,])
> mpi.close.Rslaves()
> #parallel NWS
> library("nws")
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> s <- sleigh(nodeList=mycluster[-1],  launch=sshcmd, workerCount=10)
> tNWS <- replicate(rep, system.time( integNWS(s, func, xint, yint, n) ))
> summary(tNWS[3,])
> stopSleigh(s)
> #parallel SNOW
> library(snow)
> c1 <- makeMPIcluster(10)
> tSnow1 <- replicate(rep, system.time( integSnow(c1, func, xint, yint, n) ))
> summary(tSnow1[3])
> stopCluster(c1)
> c1 <- makePVMcluster(10)
> tSnow2 <- replicate(rep, system.time( integSnow(c1, func, xint, yint, n) ))
> summary(tSnow2[3,])
> stopCluster(c1)
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeNWScluster(mycluster[2:11])
> tSnow3 <- replicate(rep, system.time( integSnow(c1, func, xint, yint, n) ))
> summary(tSnow3[3,])
> stopCluster(c1)
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeSOCKcluster(mycluster[2:11])
> tSnow4 <- replicate(rep, system.time( integSnow(c1, func, xint, yint, n) ))
> summary(tSnow4[3,])
> stopCluster(c1)
> #parallel snowfall
> library(snowfall)
> sfInit(parallel=TRUE, cpus=10, type="MPI")
> tSnowFall1 <- replicate(rep, system.time( integSnowFall(func, xint, yint, n) ))
> summary(tSnowFall1[3,])
> sfStop()
> sfInit(parallel=TRUE, cpus=10, type="PVM")
> tSnowFall2 <- replicate(rep, system.time( integSnowFall(func, xint, yint, n) ))
> summary(tSnowFall2[3,])
> sfStop()
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> sfInit(parallel=TRUE, cpus=10, type="NWS", socketHosts=mycluster[2:11])
```

```
> tSnowFall3 <- replicate(rep, system.time( integSnowFall(func, xint, yint, n) ))
> summary(tSnowFall3[3,])
> sfStop()
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> sfInit(parallel=TRUE, cpus=10, type="SOCK", socketHosts=mycluster[2:11])
> tSnowFall4 <- replicate(rep, system.time( integSnowFall(func, xint, yint, n) ))
> summary(tSnowFall4[3,])
> sfStop()
```

Time Visualization

In the package **snow** the function `snow.time()` can be used to visualize timing informations for the cluster usage. The function `plot()`, motivated by the display produced by 'xpvm', produces a Gantt chart of the computation, with green rectangles representing active computation, blue horizontal lines representing a worker waiting to return a result, and red lines representing master/worker communications.

```
> library(snow)
> c1 <- makeMPIcluster(10)
> visSnow1 <- snow.time( integSnow(c1, func, xint, yint, n) )
> stopCluster(c1)
> c1 <- makePVMcluster(10)
> visSnow2 <- snow.time( integSnow(c1, func, xint, yint, n) )
> stopCluster(c1)
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeNWScluster(mycluster[2:11])
> visSnow3 <- snow.time( integSnow(c1, func, xint, yint, n) )
> stopCluster(c1)
> system("cat $PBS_NODEFILE > cluster.txt")
> mycluster <- scan(file="cluster.txt" ,what="character")
> c1 <- makeSOCKcluster(mycluster[2:11])
> visSnow4 <- snow.time( integSnow(c1, func, xint, yint, n) )
> stopCluster(c1)
```

C. Additional Tables and Figures

This appendix section includes additional tables and figures.

Affiliation:

Markus Schmidberger
Division of Biometrics and Bioinformatics, IBE

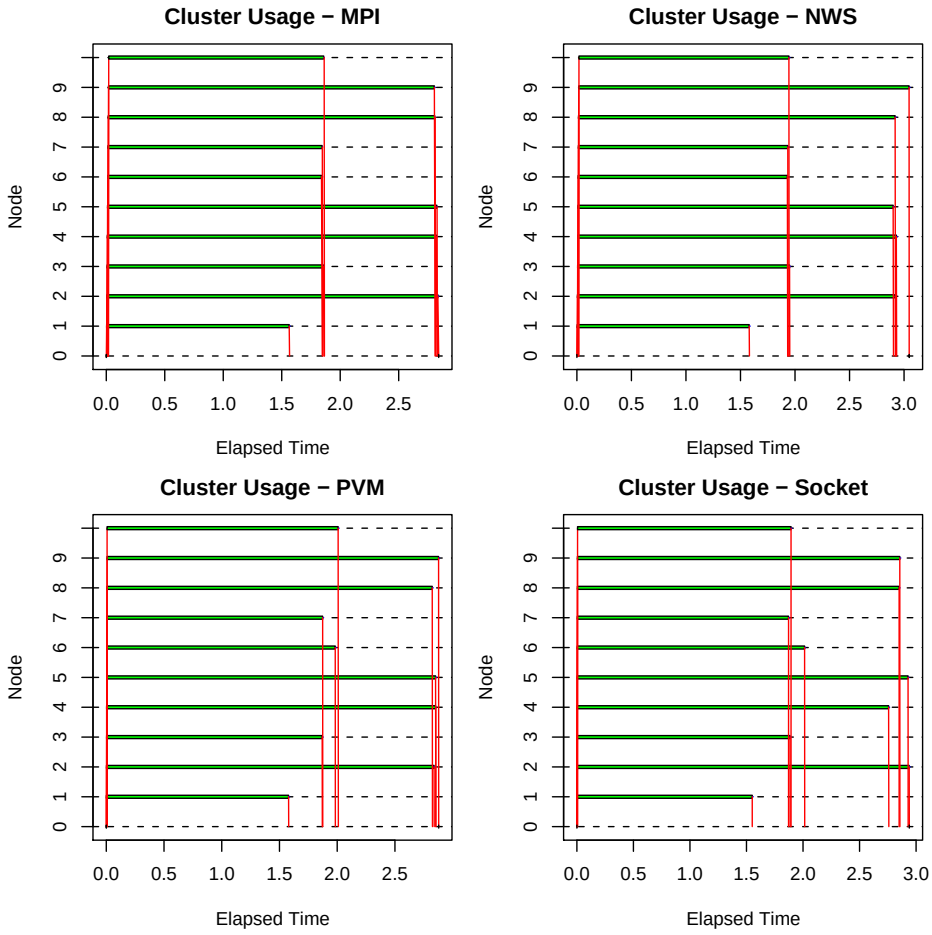

Figure 3: Time visualization of timing informations for the cluster usage for component test 3 with snow package. Green rectangles representing active computation, blue horizontal lines representing a worker waiting to return a result, and red lines representing master-slave communications.

Packages	Version	Websites
proftools	0.0-2	http://cran.r-project.org/web/packages/proftools
		http://www.cs.uiowa.edu/~luke/R/codetools
codetools	0.2-1	http://cran.r-project.org/web/packages/codetools
		http://www.cs.uiowa.edu/~luke/R/codetools
profr	0.1.1	http://cran.r-project.org/web/packages/profr
		http://github.com/hadley/profr

Table 6: Overview about R packages for code analysis available on CRAN, including latest version numbers and corresponding hyperlinks.

Packages / Software	Version	Websites
sfCluster	0.4beta	http://www.imbi.uni-freiburg.de/parallel
SLURM	1.3.11	https://computing.llnl.gov/linux/slurm
		http://sourceforge.net/projects/slurm
SGE	6.2	http://gridengine.sunsource.net
Rsge (Package)	0.4	http://cran.r-project.org/web/packages/Rsge

Table 7: Overview about software and R packages for resource manager systems, including latest version numbers and corresponding hyperlinks.

Ludwig-Maximilians-Universität Munich
81377 Munich, Germany
E-mail: Markus.Schmidberger@ibe.med.uni-muenchen.de
URL: http://www.ibe.med.uni-muenchen.de

Packages	Version	Websites
rsprng	0.4	http://cran.r-project.org/web/packages/rsprng
		http://www.biostat.umn.edu/~nali/SoftwareListing.html
rlecuyer	0.1	http://cran.r-project.org/web/packages/rlecuyer

Table 8: Overview about R packages for parallle random number generators, including latest version numbers and corresponding hyperlinks.