Introduction to parallel computing with R

Hana Ševčíková

University of Washington
hanas@uw.edu

useR!2017 Conference, Brussels, July 4–7, 2017

Motivation

Goal:

Motivation

Goal: Solve a given problem faster by utilizing more resources.

Motivation

Goal: Solve a given problem faster by utilizing more resources.

Motivation

Goal: Solve a given problem faster by utilizing more resources.

Motivation

Goal: Solve a given problem faster by utilizing more resources.

Serial processing

Motivation

Goal: Solve a given problem faster by utilizing more resources.

Serial processing

Parallel processing

Hardware Considerations

Hardware Considerations

- Central processing unit (CPU):

Hardware Considerations

- Central processing unit (CPU):
  - Multi-processor (CPU, core) computer

Hardware Considerations

- Central processing unit (CPU):
  - Multi-processor (CPU, core) computer
  - Cluster of single- or multi-processors computers

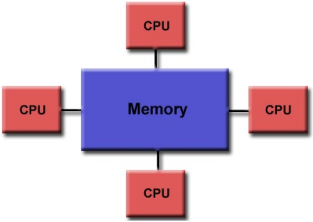
Hardware Considerations

- Central processing unit (CPU):
    - Multi-processor (CPU, core) computer
    - Cluster of single- or multi-processors computers
- Memory:

Hardware Considerations

- Central processing unit (CPU):
  - Multi-processor (CPU, core) computer
  - Cluster of single- or multi-processors computers
- Memory:

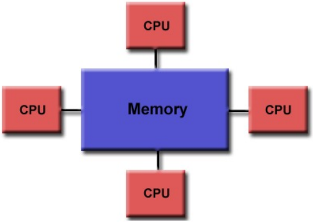
Shared Memory



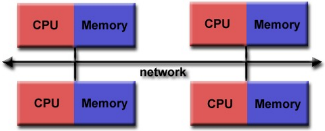
Hardware Considerations

- Central processing unit (CPU):
  - Multi-processor (CPU, core) computer
  - Cluster of single- or multi-processors computers
- Memory:

Shared Memory



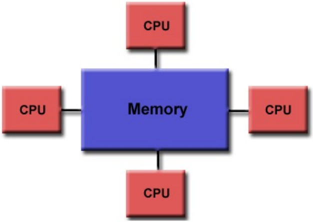
Distributed Memory



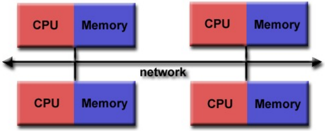
Hardware Considerations

- Central processing unit (CPU):
  - Multi-processor (CPU, core) computer
  - Cluster of single- or multi-processors computers
- Memory:

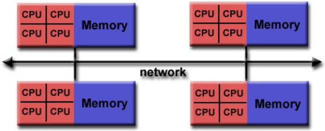
Shared Memory



Distributed Memory



Hybrid Memory



Parallel Programming

- Programming Paradigms:

Parallel Programming

- Programming Paradigms:
    - Implicit Parallelism:

Parallel Programming

- Programming Paradigms:
  - Implicit Parallelism:
    - Does not need directives for parallelization from user

Parallel Programming

- Programming Paradigms:
  - Implicit Parallelism:
    - Does not need directives for parallelization from user
    - Examples: parallel compilers, HPF, OpenMP, ScaLAPACK, PBLAS

Parallel Programming

- Programming Paradigms:
  - Implicit Parallelism:
    - Does not need directives for parallelization from user
    - Examples: parallel compilers, HPF, OpenMP, ScaLAPACK, PBLAS
  - Explicit Parallelism:

Parallel Programming

- Programming Paradigms:
    - Implicit Parallelism:
        - Does not need directives for parallelization from user
        - Examples: parallel compilers, HPF, OpenMP, ScaLAPACK, PBLAS
    - Explicit Parallelism:
        - Message-passing libraries (MPI)

Parallel Programming

- Programming Paradigms:
  - Implicit Parallelism:
    - Does not need directives for parallelization from user
    - Examples: parallel compilers, HPF, OpenMP, ScaLAPACK, PBLAS
  - Explicit Parallelism:
    - Message-passing libraries (MPI)
    - Supported by various programming languages including R

Parallel Programming Models

- Abstraction above hardware and memory architectures.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
    - Processes share a common address space which they can read & write to.
    - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.
  - Suited for *embarrassingly parallel* applications.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.
  - Suited for *embarrassingly parallel* applications.
  - R: snow (now partly in parallel), snowFT, snowfall, foreach

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.
  - Suited for *embarrassingly parallel* applications.
  - R: snow (now partly in parallel), snowFT, snowfall, foreach
- Single Program Multiple Data (SPMD) :

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.
  - Suited for *embarrassingly parallel* applications.
  - R: snow (now partly in parallel), snowFT, snowfall, foreach
- Single Program Multiple Data (SPMD) :
  - Emphasis on data parallelism.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.
  - Suited for *embarrassingly parallel* applications.
  - R: snow (now partly in parallel), snowFT, snowfall, foreach
- Single Program Multiple Data (SPMD) :
  - Emphasis on data parallelism.
  - One program, processors are autonomous, no manager.

Parallel Programming Models

- Abstraction above hardware and memory architectures.
- Shared Memory Model:
  - Processes share a common address space which they can read & write to.
  - Examples: POSIX Threads (Unix/Linux), OpenMP (multi-platform)
  - R: Rdsm, multicore (now in parallel), foreach, Rcpp, rJava and other packages supporting OpenMP.
- Message Passing Model:
  - Processes use their own memory. Communication via sending and receiving messages.
  - Examples: MPI (de-facto standard)
  - R: Rmpi, pbdMPI, pbdR
- Master-Worker Model:
  - Master assigns jobs to a pool of workers. Workers perform their task independently. Master collects results.
  - Suited for *embarrassingly parallel* applications.
  - R: snow (now partly in parallel), snowFT, snowfall, foreach
- Single Program Multiple Data (SPMD) :
  - Emphasis on data parallelism.
  - One program, processors are autonomous, no manager.
  - R: pbdR et al, RHadoop, ddR

Structure of Statistical Simulations

Many statistical simulations have the following structure:

Structure of Statistical Simulations

Many statistical simulations have the following structure:

```
initialize.rng(...)
for (iteration in 1:N) {
    result[iteration] <- myfunc(...)
}
process(result,...)
```

Structure of Statistical Simulations

Many statistical simulations have the following structure:

```
initialize.rng(...)
for (iteration in 1:N) {
    result[iteration] <- myfunc(...)
}
process(result,...)
```

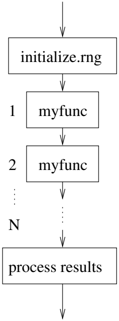
If `myfunc` can be processed independently, then:

Structure of Statistical Simulations

Many statistical simulations have the following structure:

```
initialize.rng(...)
for (iteration in 1:N) {
    result[iteration] <- myfunc(...)
}
process(result,...)
```

If `myfunc` can be processed independently, then:

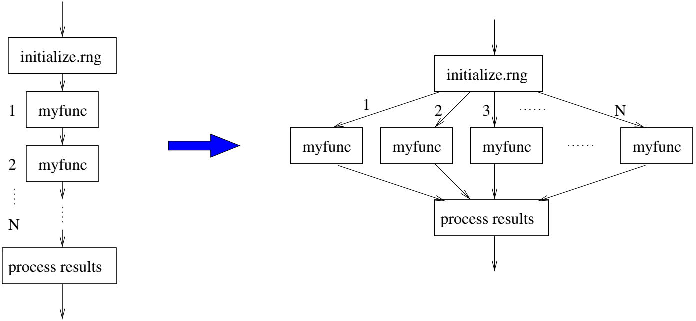


Structure of Statistical Simulations

Many statistical simulations have the following structure:

```
initialize.rng(...)
for (iteration in 1:N) {
    result[iteration] <- myfunc(...)
}
process(result,...)
```

If `myfunc` can be processed independently, then:



initialize.rng
1 myfunc
2 myfunc
N
process results

initialize.rng
1 2 3 ...... N
myfunc myfunc myfunc ...... myfunc
process results

Master-Worker Paradigm

Master-Worker Paradigm

- A cluster consists of a master and a set of workers

Master-Worker Paradigm

- A cluster consists of a master and a set of workers (often one worker per physical node).

Master-Worker Paradigm

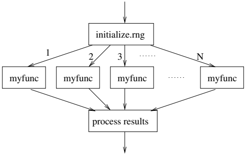
- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.

Master-Worker Paradigm

- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.
- Master is responsible for sending workers everything they need and for collecting results.

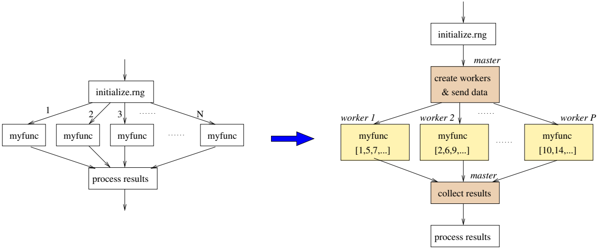
Master-Worker Paradigm

- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.
- Master is responsible for sending workers everything they need and for collecting results.



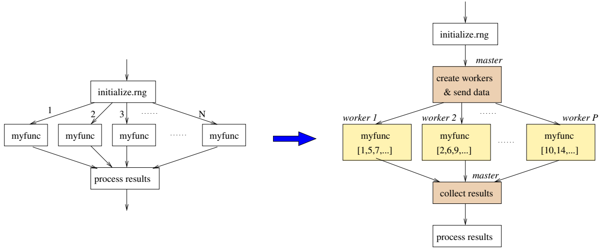
Master-Worker Paradigm

- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.
- Master is responsible for sending workers everything they need and for collecting results.



Master-Worker Paradigm

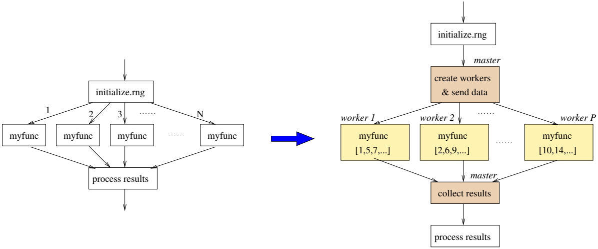
- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.
- Master is responsible for sending workers everything they need and for collecting results.



- Often $P << N$

Master-Worker Paradigm

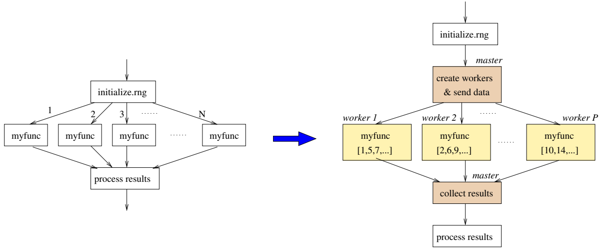
- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.
- Master is responsible for sending workers everything they need and for collecting results.



- Often $P << N$
- Many packages in R work in this fashion.

Master-Worker Paradigm

- A cluster consists of a master and a set of workers (often one worker per physical node).
- Each worker is responsible for one or more calls of `myfunc`.
- Master is responsible for sending workers everything they need and for collecting results.



- Often $P << N$
- Many packages in R work in this fashion.
- One of the pioneers, snow (Simple Network of Workstations) recently re-implemented as parallel (in R core).

Random Number Generators (RNG)

Random Number Generators (RNG)

- Many statistical applications involve random numbers

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment.

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment.
- A good quality RNG with multiple independent streams proposed by L'Ecuyer et al. (2002):

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment.
- A good quality RNG with multiple independent streams proposed by L'Ecuyer et al. (2002):
  - Period 2^{191}; streams have seeds 2^{127} steps apart.

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment.
- A good quality RNG with multiple independent streams proposed by L'Ecuyer et al. (2002):
  - Period 2^{191}; streams have seeds 2^{127} steps apart.
  - Parallel parts of users computation can run independent and reproducible streams.

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment.
- A good quality RNG with multiple independent streams proposed by L'Ecuyer et al. (2002):
  - Period 2^{191}; streams have seeds 2^{127} steps apart.
  - Parallel parts of users computation can run independent and reproducible streams.
  - Direct interface in R: **rlecuyer**, **rstream**

Random Number Generators (RNG)

- Many statistical applications involve random numbers (e.g. MCMCs in Bayesian methods, bootstrap, simulations).
- Important parameters of an RNG:
  - Long period (preferably $> 2^{100}$)
  - Good structural (distributional) properties in high dimensions
- These parameters should hold when used in distributed environment.
- A good quality RNG with multiple independent streams proposed by L'Ecuyer et al. (2002):
  - Period 2^{191}; streams have seeds 2^{127} steps apart.
  - Parallel parts of users computation can run independent and reproducible streams.
  - Direct interface in R: **rlecuyer**, **rstream**
  - In R core: `RNGkind("L'Ecuyer-CMRG")`