# Tutorial: Parallel Computing of Simulation Models for Risk Analysis

Allison C. Reilly[1]

*University of Michigan, Industrial and Operations Research, 1205 Beal Ave., Ann Arbor, MI 48109, acreilly@umich.edu*

Andrea Staid

*Johns Hopkins University, Department of Geography and Environmental Engineering, 313 Ames Hall, 3400 N. Charles St., Baltimore, MD 21218, astaid@jhu.edu*

Michael Gao

[1] Corresponding Author

*SolarCity, 3055 Clearview Way, San Mateo, CA 94402, mgao@solarcity.com*

Seth D. Guikema

*University of Michigan, Industrial and Operations Research, 1205 Beal Ave., Ann Arbor, MI 48109, sguikema@umich.edu*

## ABSTRACT

Simulation models are widely used in risk analysis to study the effects of uncertainties on outcomes of interest in complex problems. Often, these models are computationally complex and time consuming to run. This latter point may be at odds with time-sensitive evaluations or may limit the number of parameters that are considered. In this article, we give an introductory tutorial focused on parallelizing simulation code to better leverage modern computing hardware, enabling risk analysts to better utilize simulation-based methods for quantifying uncertainty in practice. This article is aimed primarily at risk analysts who use simulation methods but do not yet utilize parallelization to decrease the computational burden of these models. The discussion is focused on conceptual aspects of embarrassingly parallel computer code and software considerations. Two complementary examples are shown using the languages MATLAB and R. A brief discussion of hardware considerations is located in the Appendix.

Key Words: Risk Analysis, Parallel Computing

## 1.0 INTRODUCTION

Risk analysts often rely on simulation models to estimate probabilities, consequences, or both for uncertain future situations.[1–7] These are often computationally complex models for which the run time of the model can be a limiting factor in how accurately the quantities of interest can be estimated. This is particularly true for situations in which the probabilities of interest are very small, requiring a large number of iterations for accurate estimation, even with appropriate variance reduction techniques. For example, in Booker et al.,[8] 2 billion replications were used in a Monte Carlo simulation to accurately estimate the reliability of the European fiber optic communications backbone network. This required approximately 1 h of computer run time despite the use of a hash table, a specific type of data structure aimed at reducing run times, to reduce the computational burden. For some time-sensitive applications, such as an approaching hurricane, this run time for a hazard assessment would be prohibitive especially since the run time can grow exponentially with respect to the size of the network or region of interest. Reducing the computational burden of simulation models can be of significant benefit to risk analysts, both practitioners and academic researchers.

There are two general approaches to reducing the computational run time of simulation models when starting from a basic Monte Carlo simulation that is run single-stream on a single CPU (Central Processing Unit): (1) change the simulation approach or (2) change the computational structure. In the first approach, methods such as variance reduction and importance sampling can be used to estimate the quantities of interest in a more computationally efficient manner (e.g., Law and Kelton[9]). The goal in this approach is to sample the underlying and generally unknown probability distribution in a way that estimates the quantities of interest accurately while requiring fewer samples. The approach generally requires changes in how the sampling is done. In some cases, these are simple changes, while in others they can be quite complex. The second approach changes the computational structure to parallelize the simulation process. In the simplest sense, this can involve running iterations of the simulation simultaneously on different computational cores within the computer or within a cluster. This approach does not require changing the logic of the sampling process, unlike the first, but does require creating a parallel version of the code. There are advantages and disadvantages to both approaches—more intelligent sampling and parallelization—and in some cases they can be combined. Our goal in this article is to focus on the parallelization approach and provide a tutorial on how to easily parallelize existing simulation code for risk analysis studies.

There are two main reasons why risk analysts who use simulation models should be interested in parallel computing. First, there have been substantial advances in computing hardware, with multi-core processors now the norm and small clusters now within the financial and technical reach of practicing and academic risk analysts, even those without substantial training in parallel processing. Second, parallelizing simulation models can, in many cases, yield a substantial reduction in processing time without changing the underlying structure of an already developed simulation. Our goal in this article is to provide an overview of how a risk analyst can start parallelizing their simulation models. We assume only that the analyst has enough programing knowledge and skill to write a simulation model in a programming language, not that they have any particular knowledge of parallel processing.

This article is structured as followed. In Section 2, we provide both a conceptual overview of parallel computing and different types of software approaches for code parallelization. We do not provide a comprehensive set of instructions for one particular language here. Rather, we provide an overview of several approaches together with more detailed examples in two particular languages. In Section 3, we provide two realistic case studies with code in MATLAB[10] and R Core Team.[11] We provide conclusions in Section 4. Appendix A contains definitions of words often used in association with parallel computing. Appendix B provides an overview of hardware considerations for parallel computing and more specifically the hardware we employ. The information in this appendix could be tedious for the casual reader or for someone who does not want to build a multi-node cluster. Our hope is that this article will provide a starting point for risk analysts as they begin to further take advantage of modern computational capabilities to more efficiently estimate risk with simulation-based models.

## 2.0 SOFTWARE

In this section, we discuss the vocabulary typically associated with parallel computing and a conceptual overview of what it means to parallelize code. Later in this section, we discuss more technical aspects of parallel computing including software requirements and the changes necessary to execute parallel code. The discussion in this section generally focuses on broader procedures and themes rather than on technical nuances of any particular language. Technical details can be found in users help manuals for the language of choice. However, we discuss a few commands in compiled languages—specifically MATLAB and R—to initiate the process for the reader. Section 3 contains two case studies, and technical details are included in each.

We focus on embarrassingly parallel code in this article, as it is more common among risk analysts and simpler to develop and execute. Embarrassingly parallel code means that each logical processor (i.e., core or thread) is given a task and there exists no dependency among those tasks. This concept is illustrated in Figure 1. Imagine a for-loop where, in each loop, a function, called my_function, should be run. In a traditional for-loop with a single-core processor, loops are run sequentially. This is illustrated in Figure 1a. Conceptually, embarrassingly parallel code divides the iterations of a for-loop among the cores allotted. In Figure 1b, the code is divided among four cores, with one thread (i.e., compute job) running on each. Neither the input nor the output of the second iteration in my_function depends on either input or the output of the first iteration, and so on.

Embarrassingly parallel code is iteration independent; the current iteration is independent of the previous iterations. Similarly, future iterations don't depend on the results of the current iteration. For example, a parallel program analyzing 10,000 independent failure scenarios for an electrical-grid is embarrassingly parallel. A program that assesses the evolution of damage given a particular scenario is not because future time-steps rely on the results from previous time-steps. Typically, embarrassingly parallel code is viewed, either implicitly or explicitly, as an iteration-independent for-loop.

Before continuing, we provide definitional notes on parallel computing vocabulary, including cores, nodes, head nodes, processors, threads, and clusters. We provide a more complete list of common words with definitions in Appendix A. All underlined words are defined in this appendix. A node is simply a computer with some amount of RAM contained within it. A personal computing machine has one node and a unique network address. A cluster is often an assembly of nodes. One of these nodes is the head node and it is through this node that the other nodes communicate. Furthermore, a head node communicates with both the private network that forms the communication fabric for the cluster and the public network (e.g., the Internet). A processor is the workhorse of the machine or node. Some processors are internally divided to form more than one independent central processing units, or cores. Machines can have multiple nodes, each with multiple cores. Many modern multi-processor nodes can run multiple threads – compute jobs – at the same time, a feature often termed hyperthreading. The total number of threads that a personal-computing machine or a cluster can run dictates the number of computations that may be performed simultaneously. For example, assuming no software restrictions, two nodes with one core each allows for two parallel computations, and *nearly* halves your computing time. Each core is tasked with one thread. In contrast, a modern laptop with four cores each capable of two threads, allows for eight parallel computations.

Assuming the entire program can be parallelized, the maximum theoretical speed-up is *X*, where *X* is the number of threads being used. This is called Amdahl's Law.(12) Due to the fact that there is additional communication time for the head node to relay information to the worker nodes and then for the worker nodes to relay results back, the speed up in actuality will never reach *X*, but it can be close.

*2.1 Software requirements*

Most languages have available libraries, modules, code, or toolboxes that communicate with the hardware of the computer and administer the backend parallelization tasks, such as optimal cluster setup detection, and dynamic load balancing. These libraries, etc., must be downloaded prior to the code's execution and some language's libraries, etc., are freely available while others are not. Many languages have multiple options, each with their own benefits and offerings. Most are easily implemented.

As an example, consider MATLAB and R, two run-time languages popular among risk analysts. MATLAB requires either the Parallel Computing Toolbox or the Distributed Computing Server Toolbox. The later is more expensive, but allows more cores to be used. Popular options for R include the "parallel," "doSNOW," and "foreach" packages.(11,13,14) R documentation best describes their functionality. For Python, C, and C variants, popular compiled languages, many options exist, including Parallel Python (for Python) and OpenMP (for C). Some reduce the onus on the user but lack customization while others are reserved for the more experienced users and are more customizable.(15)

*2.2 Cluster Creation and Preparation*

Almost universally, the user is required to “set up” a virtual cluster via code. This is distinct from physically building a cluster of nodes as described in Appendix B. In essence, the user tells the computer to prepare to run parallel code and which cores to use to build the cluster. Sometimes this means that licenses, libraries, files, and/or data are forwarded or “pushed” to all nodes of the cluster. The complexity of this execution ranges dramatically; MATLAB, for example, simply requires one additional line of code while R requires more of the user when the code is not run locally (i.e., on a cluster rather than on a personal computer).

The examples in Section 3 demonstrate cluster creations for both MATLAB and R.

*2.3 Code changes to support parallel code*

Few structural changes are necessary to parallelize code as long as it is of the embarrassingly parallel type. The main challenge is to decide *which* part of the code to parallelize. The examples that follow demonstrate these changes to the code.

As we mentioned previously, we focus on for-loops and embarrassingly parallel code. If a for-loop is nested (i.e., a for-loop within a for-loop), the user must decide which loop to parallelize; a parallel-loop cannot lie within another parallel-loop in embarrassingly parallel code. Generally, the most external parallel loop is parallelized, but we encourage the reader to experiment.

## 3.0 CASE STUDIES

In this section, we provide two real risk-analysis examples from the work in our research group where parallelizing the code provides distinct advantages in terms of run time. Both examples— network reliability estimation and synthetic hurricane generation—are types of problems relevant to risk analysts. The first example demonstrates parallel computing in MATLAB while the second example demonstrates parallel computing in R.

### *3.1 Example 1 – MATLAB example*

In the following example, we parallelize a Monte Carlo simulation in MATLAB. The code necessary for parallelization is stated while the remainder of the code is simplified to function names for the sake of brevity.

#### 3.1.1 Problem Definition

In this example, we use the simple network connectivity problem discussed in Guikema and Gardoni.[16] An urban network with 9 nodes, 16 links, and 19 embedded bridges experiences a magnitude 8 earthquake with a known epicenter as shown in Figure 2. A hospital is located at Node 1 and the remaining nodes are neighborhoods within the urban area. The squares represent bridges and the numbers within the squares represent the type of bridge: (1) single-bent overpass and (2) double-bent overpass. The earthquake may cause bridges to fail. We assume that bridge failures are conditionally independent events given the occurrence of an earthquake and that travel through nodes and links other than bridges is unaffected by the earthquake. The goal of Guikema and Gardoni(16) is to calculate the likelihood of network connectivity between each of the neighborhood nodes (Nodes 2–9) and the hospital node (Node 1). Our goal here is to compare the runtimes of parallelized code with varying cores on performing the necessary computations.

The bridges each have a known fragility curve that describes the conditional probability that it will perform at or above a specified level of performance for a given ground motion intensity. In other words, given the ground motion intensity at bridge type $i$, where $i = \{1, 2\}$, the probability of failure can easily be estimated. We use a Monte Carlo simulation to determine the probability, with a 95% confidence interval, that all nodes in the network are connected after the earthquake.

When the probability of a bridge failure is quite small (e.g., $P_f = 10^{-5}$), a simulated failure is rarely expected (e.g., once every 100,000 simulations) for each bridge individually. With 19 bridges, the problem requires a large number of iterations to obtain appropriate convergence of the estimated probabilities. This class of problem is known to have a computational burden of size $O(n^2)$ meaning that the computational burden varies on the order of $n^2$ where $n$ is the number of replications. Hence, parallelization is desired here to reduce computation time.

The goal in this article is to compare run times. To do so, we simulate random network states $N$ times and repeat this simulation *numTrials* times in order to produce confidence intervals around the run times. We demonstrate the process using MATLAB 2013b on a MacBook Pro 2.9 GHz Processor with 4 cores and on a server using two 2.66 GHz Processors on up to 12 cores. We run the script on one, two, and four cores both locally and on the server. Additionally, we run the script on 8 and 12 cores exclusively on the server and ultimately compare all run times.

### 3.1.2 Software requirements

Our discussion assumes MATLAB versions 2007b[2] and later. In addition to MATLAB software, the MATLAB Parallel Computing Toolbox license is necessary for parallelization on 12 or fewer cores, termed workers in MATLAB, and the MATLAB Distributed Computing Server Toolbox is required for more than 12 cores. Only one license is necessary and the head node pushes the license information as necessary. The Distributed Computing Server package is compatible with many schedulers.[3]

### 3.1.3 Cluster creation and preparation

Creating a cluster within MATLAB requires only one additional line of code prior to the portion of the code that is parallelized. This step opens a parallel session for 30 minutes or the length of time it takes the code to be executed and initiates communication among the desired number of cores. The user chooses how many cores to include in the cluster, up to the maximum number allowed by the MATLAB license and the computing resources available. Neglecting to create a cluster does not prevent parallel code from being executed, but it is less efficient computationally and a default number of cores are used.

[2] Use versions Matlab2013b or later to avoid java compatibility issues with the Parallel Computing Toolbox. Using an earlier version? Download a recent MATLAB patch.

[3] A scheduler manages and monitors resources across a server. Scheduling software is not required but is helpful when the server is in high demand.

```
(Line 1) >> parpool('local',2)  %creates a cluster consisting of
                cores on a local machine
```

3.1.4 Code changes to support parallel code

MATLB requires few structural changes to support parallelization. However, the user must have only one parallel loop. To change a for-loop to a parallelized for-loop, “parfor” is used instead of “for.” The pseudocode for this problem is shown below. It assumes that the external for-loop is parallelized.

```
(Line 2) >>   calc_Bridge_Prob_of_Failure;  %calculate the
              probability  of failure for each bridge
(Line 3) >>   calc_Link_Prob_of_Failure;    %calculate the
              probability of failure for each link
(Line 4) >>   parfor i = 1:N                %for N realizations
(Line 5) >>      generate_Link_States(i);   %which links are
                 connected in realization i?
(Line 6) >>      determine_Network_Connectivity(i);  %is the network
                 connected in realization i?
(Line 7) >>   end
(Line 8) >>   assess_Network_Connectivity_stats;    %calculate
              statistics
```

The loops are divided among the number of cores specified in “parpool.” We baseline all runs by running the program without parallelization both locally and on the server. See Table I for total runtime comparisons.

3.1.5 Closing out a parallel session

One command is required to close a parallel-session in MATLAB. Hereafter, communication among cores ceases for this MATLAB instance.

```
(Line 9) >>  delete(gcp('nocreate'));
```

<u>3.1.6 Results and Discussion</u>

We run the model for 1,000,000 simulated network states. Only the external loop is parallelized. The results are faster on the server relative to the personal computer simply due to a faster processor and memory. We can see in Figure 3 and Table I that the speed-up roughly follows Amdahl's Law. We also see that there is a substantial speed-up due to parallelization on the server with 12 cores. The run time drops from about 1 hour to less than 10 minutes. This is a substantial practical benefit for risk analysts.

### 3.2 Example 2 – R example

In the following example, we demonstrate parallel computing in R. The code necessary for parallelization is stated while the remainder of the code is simplified to function names for the sake of brevity.

<u>3.2.1 Problem Definition</u>

We demonstrate the capabilities of parallel computing within R by generating a large number of virtual hurricanes. Generated hurricanes allow a risk analyst to study a variety of storm characteristics and their impacts, from power system disruptions to evacuations, without needing to first experience a storm in reality. Generating each storm requires sampling from historical distributions, applying track models to prescribe the storm movement, and modeling the storm decay until it weakens substantially. An example can be found in Staid et al.[7] These tasks are computationally intensive, especially when a large number of storms are needed to achieve convergence in the results. These characteristics make this problem an ideal candidate for parallel computing. Each storm is generated independently, so there is no dependence on previous iterations.

Below we focus on the changes to the code that are necessary to create a cluster and to run parallel code using R. It is simplest if the code that is to be parallelized is structured as a standalone function. This makes every iteration explicitly parallel. In this example, the function "hurr_simulation" is called 10,000 times and contains the code that creates a storm by sampling from historical hurricane distributions, identifies the storm path and movement, and then estimates the decay of the storm's strength as it moves over land. The function "hurr_simulation" requires the data file of historical storms, "hurricane_data," as well as the R statistical library called "randomForest." A random forest is an ensemble learning method for regression. Details can be found in Staid et al.[7]

### 4.2.2 Software requirements

We assume the latest version of R is being used (e.g., version 3.1.2 at the time of press). Three additional libraries (packages) are needed to parallelize the code in this case study – parallel, doSNOW, and foreach. They are freely available.

### 4.2.3 Cluster creation and preparation

Like all R code, libraries and data are loaded first. The libraries listed below are in addition to the libraries needed in the function “simulation.”

```
(Line 1) library(parallel); library(foreach); library(doSNOW)
         #load libraries to support parallel computing (R Core
         Team 2014, Revolution Analytics 2012a, Revolution
         Analytics 2012b)

(Line 2) load('hurricane_data.RData') #load input data of historical
         hurricanes
```

After the libraries are loaded, we create and register a cluster named “cl.” The cluster is first built using 2 nodes. These nodes each have 8 cores. This size cluster is reflected in the code. Later we perturb the size of the cluster to compare run times. We then export the data files and libraries associated with ‘hurr_simulations’ to each node of the cluster, cl.

```
(Line 3)  cl <- makeCluster(c(rep('node1@node1',8),
           rep('node2@node2', 8)) type='SOCK')         # Make
           cluster on nodes 1 and 2, each with 8 cores

(Line 4)  registerDoSNOW(cl)     # Register cluster

(Line 5)  clusterExport(cl, hurricane_data)  # Export required data
           files to cluster nodes, cl

(Line 6)  clusterEvalQ(cl, library(randomForest))  # Push required
           library to each cluster node
```

### 3.2.4 Code changes to support parallel code

Like MATLAB, R requires few structural changes to support parallelization. We recommend structuring the code so that each iteration calls a function. This way, complex computations are separated from the parallel for-loop and the code is cleaner. The function can contain any number of steps, processes, and sub-functions. After each iteration, only the function's final output is stored, allowing for easier data management.

```
(Line 7)  parallel_output_file <- foreach(y = 1:10000, .combine =
            'rbind', .packages = 'randomForest') %dopar%
            {hurr_simulation(hurricane_data)}      # This command
            executes a parallel for-loop via 'dopar.' The user can
            specify the means for combining the output of each
            iteration, here, output is appended in rows using
            'rbind.' The user also specifies packages, here
            'randomForest,' that the tasks need. The '.packages'
            command loads the package on each worker.
```

This simulation was tested on the research group's server, which is a physical cluster of 10 nodes with 8 cores each, for a total of 80 cores. The code was tested in series (i.e., on one core) and on increasing cluster sizes up to the maximum available number of cores (80).

### 4.2.5 Closing out a parallel session

The user must close the cluster session to cease communication among the nodes and to free-up computer resources. This requires one line of code.

```
(Line 8)  stopCluster(cl)
```

### 4.2.6 Results and Discussion

We iterate over 10,000 simulated hurricane tracts and present the results below. As expected, the run time is nearly cut in half when the number of cores is doubled. The reduction in run time can be considerable for large simulations. In this example, the run-time reduction, from nearly 2 hours to less than 3 minutes, is practically meaningful for risk analysts. It means that in advance of an approaching hurricane, risk analysts can perform numerous scenario analyses to aid emergency managers in responding more appropriately.

Interestingly, the marginal improvement is fairly small after 16 cores. For example, when 64 cores are added to a machine that already has 16 cores to get 80 cores, the reduction in run time is only 6 minutes. This reinforces the idea that large clusters are not always necessary.

Figure 4 and Table II show the actual and theoretical run time, in seconds, as a function of the number of cores. The theoretical run time is lower than the actual run time due to the overhead communication.

**4.0 CONCLUSION**

By parallelizing computer code, risk analysts are able to reduce the computational time, in some cases dramatically, required to perform simulations. This allows for faster computational convergence and knowledge of the system’s risk and performance sooner—a benefit to both risk analysts and decision makers, especially during time-sensitive events like natural disasters.

The primary drawback of parallelizing code, in our opinion, is the time required to parallelize code for the first time. For novices who seek to run a program with few iterations or with many simple iterations, the time required to parallelize the code may not be worth the computational-time savings. However, there are many instances where the ramp up time to parallelize code initially will be appreciably less than the extra time it would have taken the program to run sequentially (i.e., without parallelization). This is especially true in domains such as real-time control systems, data assimilation, and network optimization. We encourage the reader to time one iteration of the run, and to calculate the potential timesaving that parallelized-code will offer.

In this article, we demonstrate how computer code is changed conceptually to accommodate parallelized code. Appendix B addresses the hardware necessary to support parallelized code, and some emerging trends in parallelization hardware (e.g., GPUs and MICs and cloud services). Through two examples, we provide the changes necessary to MATLAB and R code to support parallelization. We acknowledge that what we provide here is not an exhaustive list of options for these languages, and that MATLAB and R are only two of the many languages that support parallelized code. However, these changes and general procedures are representative of the changes necessary to parallelize code. Once a conceptual understanding is achieved, the many resources targeted at parallelization in specific languages become comprehendible and we encourage the user to explore these resources.

## 5.0 ACKNOWLEDGMENTS

This work is funded by grants from the National Science Foundation (NSF), grants 1243482 and 1149460. The support of the sponsor is gratefully acknowledged. Any opinions, findings, conclusions or recommendations presented in this article are those of the authors and do not necessarily reflect the

view of the National Science Foundation.

## 6.0 REFERENCES

1. Smith RL. Use of Monte Carlo simulation for human exposure assessment at a superfund site. Risk Anal. 1994;14(4):433–39.

2. Rafoss T. Spatial stochastic simulation offers potential as a quantitative method for pest risk analysis. Risk Anal. 2003;23(4):651–61.

3. Reshetin VP, Regens JL. Simulation modeling of anthrax spore dispersion in a bioterrorism incident. Risk Anal. 2003;23(6):1135–45.

4. Pouillot R, Beaudeau P, Denis JB, Derouin F. A quantitative risk assessment of waterborne cryptosporidiosis in france using second‐order Monte Carlo simulation. Risk Anal. 2004;24(1):1–17.

5. Merrick JRW, Van Dorp JR, Dinesh V. Assessing uncertainty in simulation‐based maritime risk assessment. Risk Anal. 2005;25(3):731–43.

6. Quiring SM, Schumacher AB, Guikema SD. Incorporating hurricane forecast uncertainty into a decision support application for power outage modeling. Bull Amer Meteorol Soc. 2014;95:47–58.

7. Staid A, Guikema SD, Nateghi R, Quiring SM, Gao MZ. Simulation of tropical cyclone impacts to the U.S. power system under climate change scenarios. Clim Change 2014;127(3):535–46.

8. Booker G, Sprintson A, Singh C, Guikema SD. Efficient availability evaluation for transport backbone networks. Optical Network Design and Modeling, ONDM 2008, International Conference on, 2008:1–6.

9. Law AM, Kelton WD. Simulation modeling and analysis. vol 2. New York: McGraw-Hill; 1991.

10. MATLAB version 8.2.0.701. Natick (MA): The MathWorks Inc.; 2014.

11. R Core Team. 2014. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria; http://ww.R-project.org

12. Rodgers DP. Improvements in multiprocessor system design. ACM SIGARCH Comput Architect News. 1985;13(3).

13. Revolution Analytics. foreach: Foreach looping construct for R. R package version 1.4.0. 2012a; http://CRAN.R-project.org/package=foreach

14. Revolution Analytics. doSNOW: Foreach parallel adaptor for the snow package. R package version 1.0.6. 2012b; http://CRAN.R-project.org/package=doSNOW

15. Parallel Python – Home, 2014; http://www.parallelpython.com. <AQ>Is there more information for Ref 15?</AQ>

16. Guikema SD, Gardoni P. Reliability estimation for networks of reinforced concrete bridges. J Infrastruct Syst. 2009;15(2):61–9.

## Appendix A: GLOSSARY

**Cluster** – assembly of machines or nodes

**Core** – an independent processing unit within a CPU. One or more cores form a processor.

**CPU** – the chip in a computer that preforms calculations. It includes the cores and the supporting communication pathways.

**Dynamic load balancing** – an optimal balancing of jobs across a machine for maximal efficiency

**Embarrassingly parallel** – parallelized code that consists of only independent loops

**Head node** – the node through which other nodes communicate in a cluster. The head node also communicates with the external network (e.g. the Internet)

**Node** – a machine (computer) with some amount of RAM contained within. It may be comprised of many cores and can be connected with other nodes to make a larger machine. A personal computing machine has one node and a unique network address.

**Processor** – the workhorse of a computing machine that executes out a computational task

**Scheduler** – software that schedules, prioritizes, and manages a workload across a cluster or other computing system

**Thread** – a task that is being executed (http://en.wikipedia.org/wiki/Task_parallelism)

## Appendix B: HARDWARE

This section focuses on the hardware necessary to run parallelized code. The hardware options range from personal computing machines to clusters to pay-per-use cloud services, and each option comes with considerable advantages and disadvantages.

Table BI contains a summary of topics discussed in this section. Column 1 lists the computing methods discussed in this section: single- and multicore personal-computing machines, multicore clusters, coprocessing cards such as GPUs (graphical processing units) and MICs (many integrated core), and cloud services. Column 2 gives a qualitative assessment of the additional coding burden to parallelize the code in order for it to be compatible with the associated computing method. Column 3 displays qualitatively the cost relative to a personal computer and Column 4 gives a qualitative assessment of the potential speed up relative to non-parallelized code. Column 5 displays, again qualitatively, how complex it is to physically set up the computing machine relative to a personal computer.

### *B.1 Personal-computing machine*

We begin the discussion with a local, personal-computing machine (e.g., a desktop, a laptop, a tablet). Most personal-computing machines sold today have multiple cores, which allows for parallel computing. A machine with one core is not compatible with parallel computing.

As we show in Table BI, personal computers are relatively inexpensive relative to other hardware one may consider for parallel computing. However, as the number of cores increases, so does the price, when controlling for add-ons like graphics cards. At the time of press, the current upper limit of number of cores on personal computing machines is approximately 40 for high-end workstations as of early 2015, though laptops and many desktops generally have no more than 8 cores. Personal computing machines are very simple to physically setup.

### *B.2 Multinode Cluster Computing*

As mentioned previously, a cluster is a larger computing machine with multiple nodes, each typically with multiple cores. Components, including hard drives, cables and routers, are often purchased separately and then assembled by the user or an integrated cluster provider. As such, the complexity of setting up a cluster is greater than that of other computing methods. Typically, the upper limit on the number of cores allowed in a cluster is dictated by resource constraints (i.e., budget, cooling capacity, and available power) and not technical constraints. As such the cost of a cluster varies widely. However, clusters with modest capacity appropriate for many risk analysis simulation problems can be assembled at a reasonable cost.

#### B.2.1 Multicore Cluster Hardware Configuration

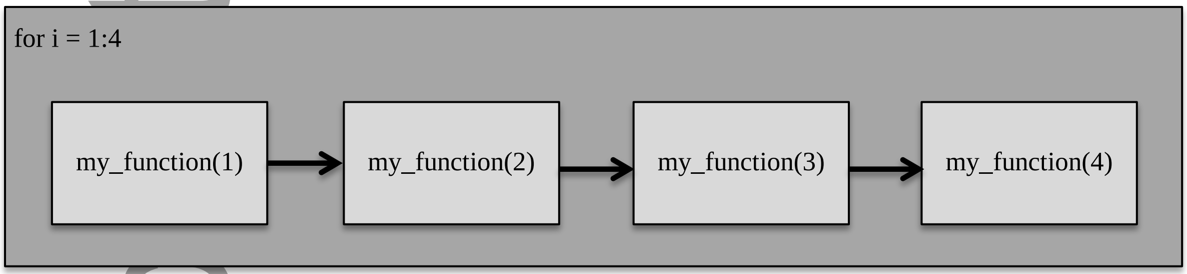
We discuss hardware configuration through the example of the cluster in the authors' research group. The server is a multi-node cluster that is modestly-sized, consisting of ten Apple Xserve rack unit computers, each with two quad-core Xeon processors for 80 physical cores. It runs on the Mac OS X Server software and the process to set it up is fairly representative of modest-sized clusters. Table BII provides a breakdown of the cluster's components and their respective cost. Total equipment cost was approximately $7,000. Some of the equipment was purchased used through a popular on-line auction site.

There are operational and storage considerations which are important to consider for clusters and which are often overlooked for personal computers. These considerations include power, ventilation, cooling, and rack support. For example, our server generates enough heat to require placement in a room climate-controlled from cluster use. Additionally, a standard room circuit proved insufficient in terms of power.

*B.3 Modern Cluster Computing Without a Cluster*

There are a number of options available for parallel simulation modeling other than owning and operating a cluster. These include cloud-based cluster services, GPUs, and MICs. A cloud-based cluster is a cluster, typically very large, owned and operated by an independent company that provides cluster services for hire. Payment is typically made per processor-hour of cluster usage. Costs vary widely as do the capabilities and flexibility of the cloud-based clusters. This can be a highly attractive option for those not wanting to own, maintain, and operate their own cluster.

Another alternative is card-based cluster computing. There are two main options here, both relatively recent. One approach that has received considerable attention is the use of GPUs— graphical processing units—for computing. The latest (early 2015) GPUs have nearly 5000 processing cores and 24GB of RAM per card. These compute cards mount in a PCI slot offering dense computing capabilities. The downside is that they require custom coding in computer languages not in widespread general use. A different card-based parallel processing option is a MIC (multiple integrated cores) coprocessor. A MIC uses standard Intel Xeon cores, which gives the advantage of programming directly in widely used languages. At present (again, early 2015) MICs can run approximately 250 threads per card.



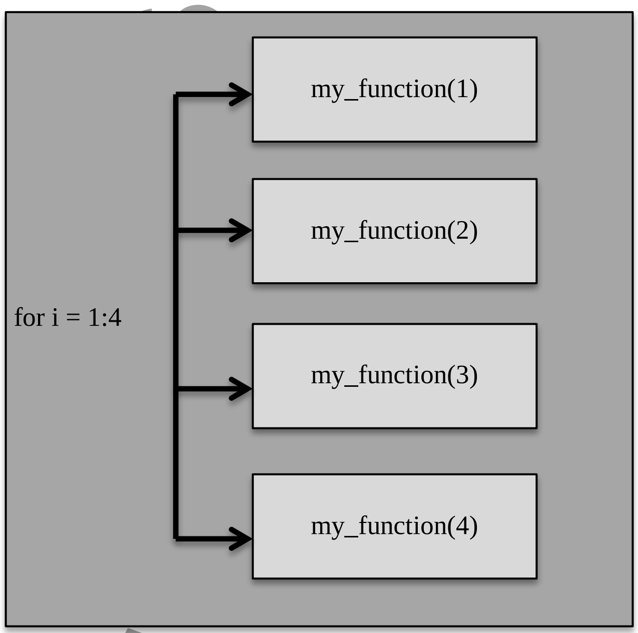


**Figure 1.** A demonstration of embarrassingly parallel code to highlight the difference between serial and parallel computing. This is shown using a for-loop and four iterations of a function called my_function. my_function(i) does not depend on my_function(j) when i is not j. Figure 1a demonstrates serial computing; all iterations are performed sequentially on the same core. Figure 1b demonstrates parallel computing of four threads on four cores; all iterations are performed simultaneously.

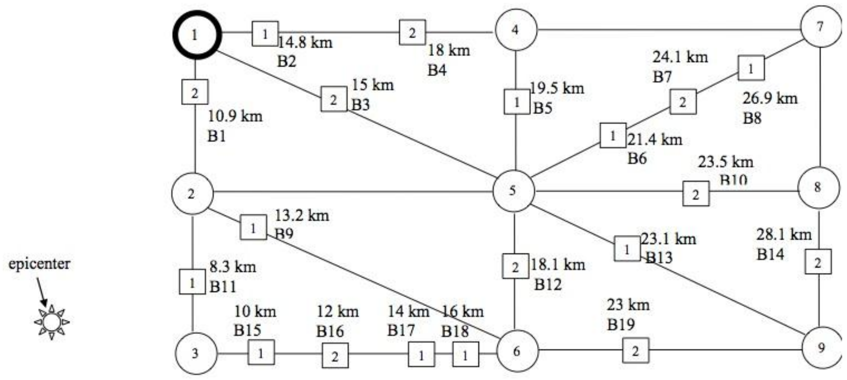


**Figure 2.** Test network for bridge network reliability estimation problem. (image source: Guikema and Gardoni 2009. Used with permission from ASCE.)

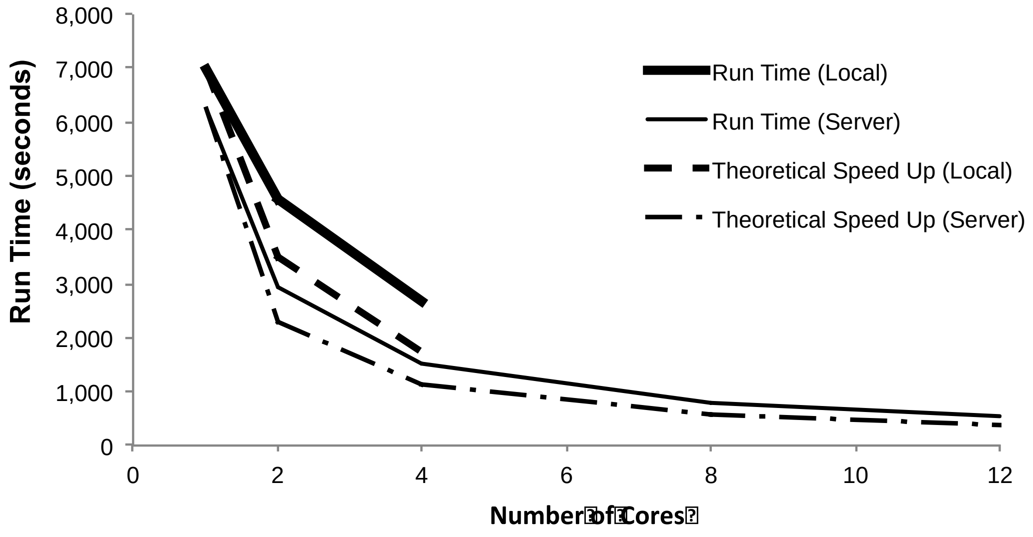


**Figure 3.** The runtime as a function of the number of cores.

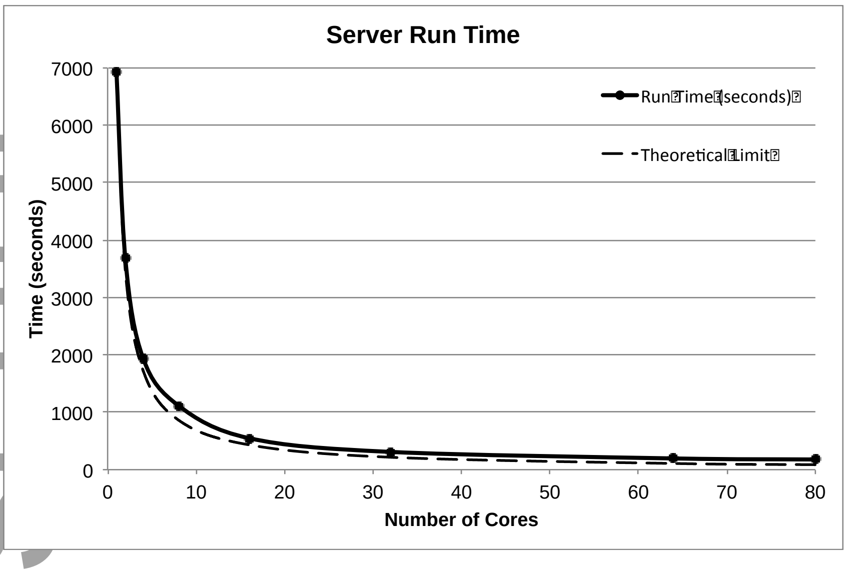


**Figure 4:** Run-time comparison for the hurricane example in R. The speed-up is small beyond 16 cores in this example.

**Table I.** A comparison of runtimes for a variety of cores.

| | 1 Core | 2 Cores | 4 Cores | 8 Cores | 12 Cores |
|---|---|---|---|---|---|
| Local Run time (sec) | 6,959 | 4,587 | 2,668 | N/A | N/A |
| Server Run time (sec) | 6,235 | 2,942 | 1,513 | 800 | 547 |

**Table II:** Run-time comparison for the hurricane example in R

| | 1 Core | 2 Cores | 4 Cores | 8 Cores | 16 Cores | 32 Cores | 64 Cores | 80 Cores |
|---|---|---|---|---|---|---|---|---|
| Run Time (sec) | 6,924 | 3,680 | 1,945 | 1,117 | 549 | 310 | 197 | 178 |
| Run Time (min) | 115 | 61 | 32 | 19 | 9 | 5 | 3 | 3 |

**Table BI:** Comparison of Parallel Computing Machine Options

| Computing Method | Coding Learning Curve | Cost | Potential Speed-up | Setup Complexity |
|---|---|---|---|---|
| Single-core personal-computing machine | none | $ | - | easy |
| Multinode personal-computing machine | low | $ | + | easy |
| Multicore Cluster | high | $$ to $$$ | +++ | difficult |
| GPU | very high | $$ | ++ | medium |
| MIC | high | $$ | ++ | medium |
| Cloud Services | high | Problem-dependent | ++++ | difficult |

**Table BII.** Components of the authors' cluster

| | **Item** | **# of units** | **Approximate Cost per unit** | **Condition Purchased** |
|---|---|---|---|---|
| **1** | Apple Xserve with OS X Server 10.6, Hitachi Hard Drives,[4] and Power Cables | 10 | $650 | Used |
| **2** | Ethernet Cables Category 5/6 | 10 | $3 | New |
| **3** | Ethernet Switch & Router | 1 | $80 | New |
| **4** | 300W Surge Protected Power Strip | 2 | $100 | New |
| **5** | Rack for housing | 1 | $200 | New |

[4] Apple has discontinued the Xserve server since the early 2012. While the Apple Xserve is discontinued, units can be bought pre-owned online. Mac Pro Server is offered in its place.