Parallelized Monte Carlo Integration With MPI

Bobby Roy

June 2011

1 Introduction

For my Monte Carlo assingment, I chose to implement parallelized Monte Carlo using the Message Passing Interface (MPI). In exploring this topic, I hoped to learn more about crude Monte Carlo integration, its implementation in code and also to learn more about parallel computing which, prior to the assignment, I knew nothing about. My goal was to produce a functional, simple, integral calculator using Crude Monte Carlo and from there to expand the project to include more complex integrals as well as to produce code samples and documentation for others to leverage to explore these concepts further and to implement their own, working, parallelized, integral calculators using MPI and Crude Monte Carlo integration.

Worth mentioning at the outset is that my experience with high-level math is extremely lacking, and so I attempted to reproduce programatically the multidimensional integrals to the best of my understanding. While it's entirely possible (and perhaps, even quite likely) that I misunderstood the multidimensional math, hopefully the general design of my integral calculators would still be useful, and the programs themselves will only require a slight tweaking to accurately approximate the integrals that they claim to work on.

I developed my implementation of parallelized Monte Carlo on an Ubuntu 10.10, dual-core machine and programmed it in C.

2 Tools

2.1 Parallel Programming

Among the tools I used to implement parallelized, Crude Monte Carlo, the most integral (and most abstract) of which was the concept of parallelization. Parallel programming is a form of programming that leverages the power of multicore and multiprocessor machines. Specifically, it allows for certain calculations to be carried out simultaneously, assuming that those calculations don't

depend on one another. It splits a program off into independent threads of execution such that large numbers of calculations can be processed concurrently and later condensed into a single result or a single set of results. An individual thread of execution is mapped to its own core or processor which then performs the calculations specific to that thread of execution. A master process of some sort is typically responsible for handling the results of these independent calculations and also for delineating tasks to some number of slave processes that generally perform most of the heavy lifting in the execution[7]. This process of breaking up a single process into a number of concurrent processes has the potential to greatly increase the speed at which calculations are performed.

2.2 MPI

The Message Passing Interface (MPI) is a standard used in parallel computing that allows individual processes to communicate through the passing of messages. It's a parallel computing standard that has been implemented for a wide range of architectures and programming languages. An MPI implementation usually consists of a set of libraries (the API), a frontend for the relevant compiler and a method for executing the parallelized code in such a way as to simplify the execution process for the user and hide the messiness of executing different jobs on different devices.[1] The version of MPI I installed on my machine was OpenMPI.

For the purposes of my programs, I made use of two functions in particular that allow a process to send data to a second process in the form of a message. Those functions are called MPI_Send() for sending a message and MPI_Recv() which the recipient process uses to receive the message[2]. The following is a diagram of how MPI handles send and receive messages.

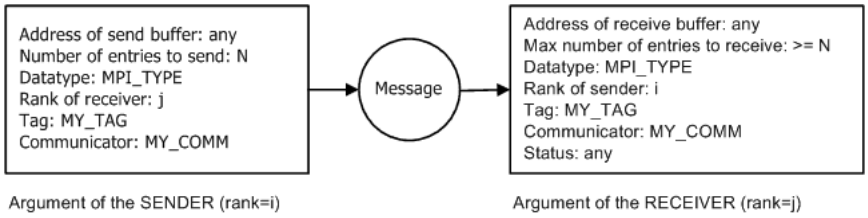

Figure 1: A diagram of how send and receive calls are made in MPI, including each function's parameters. Source: www.wikipedia.org

MPI uses a suite of functions to allow different processes to communicate with one another during runtime. These processes pass messages containing data to and from one another allowing them to perform calculations in parallel. Because each process is typically assigned its own core or processor, these parallelized programs can perform calculations simulateneously and then combine their results later, greatly shortening the overall runtime of a given program.

Within any given MPI execution, a program is divided up into a series of independent processes that fall under a communicator object. Communicator objects are responsible for facilitating point-to-point communication between two process and essentially describes the realm within which processes can communicate with one another. For the purposes of my programs, the only communicator that was necessary to employ was the global communicator, giving processes access to all other processes associated with a given program's execution.

To install OpenMPI on my Ubuntu 10.10 machine, I ran

```
1 sudo apt-get install openmpi-bin libopenmpi-dev
```

Once installed, I had access to the MPI libraries as well as a frontend for my gcc compiler (mpicc) and a shell script for executing my parallelized programs (mpirun).

2.3 SPRNG

To perform my Monte Carlo calculations, I'd also need access to functions for generating random numbers. I discovered, however, that most random number generators aren't designed to be used in the context of parallel programming. The different random number streams generated in this fashion will be correlated and won't exhibit the proper behaviors of independent, random number streams. To avoid running into that problem, I installed the SPRNG (Scalable Parallel Random Number Generators) libraries onto my machine[3]. I still generated uniform, random numbers in my programs, but I included the GSL library in case I wanted to expand my expriments to include other forms of random number generation.

There exists an SPRNG binary package in the Ubuntu repositories, but the package wasn't compiled with MPI support. Consequently, I had to download and install the SPRNG libraries from source.

To install SPRNG, I had to modify the necessary makefiles to compile the libraries specific to my architecture as well as to enable MPI support. Once compiled an executable appears named checksprng that when executed checks to see if the installation was successful.

2.4 GSL

Once I had SPRNG successfully installed, I went ahead and installed GSL (GNU Scientific Library)[4]. GSL on its own can't be used for parallel Monte Carlo as a consequence of the problems mentioned above. However, used in tandem with the SPRNG library and a useful piece of code that I discovered[5],

it's possible to take advantage of GSL's non-uniform, random number generating capabilities. SPRNG on its own is only equipped to handle uniform, random number generation.

2.5 Crude Monte Carlo

Crude Monte Carlo integration is performed by generating a large number of random values and evaluating them with regard to a given function. The results of this process are then added together and the average of the total is calculated. This average is an estimation of the integral of the function.

$$1/N \sum_{i=1}^{N} f(x_i) \tag{1}$$

Within the context of my project, this meant that I'd need to use SPRNG and GSL to generate streams of random numbers within the program to be used to approximate a particular integral. However, to ensure an accurate approximation, I'd need to generate an extremely large number of random values with which to evaluate the integral. This process of generating random variables and then plugging them into an equation could be extremely time consuming. To reduce the overall time necessary to perform these calculations, I'd parallelize the portions of my code where I performed these calculations using MPI.

3 Problems With Parallel Programming

The advantage of parallelized computing over serialized computing is easy to overestimate however. Despite the theoretical advantages of parallel computing, a number of problems exist that place a ceiling on what sorts of benefits can be derived from parallel programming.

3.1 Race Conditions

For one thing, parallel programs are notoriously difficult to write as compared to serialized programs. Parallel programming introduces a number of new bugs that are unique to parallelization, including race conditions. Race conditions emerge whenever a number of processes depend on the same resource or the same state for their calculations. If this shared state or resource is altered by one of the processes, it can potentially produce unexpected, non-deterministic results that change depending upon which process alters the state or interacts with the shared resource first. These sorts of bugs can be extremely difficult to track down due to their non-deterministic nature and require that the programmer ensure that different processes influence shared resources while observing mutual exclusivity.

Fortunately, my programs are "embarrasingly parallel", meaning that the parallelized components require very little point-to-point communication between the processes and don't share any resources. Aside from sending some initial information from the master process to the slave processes and compiling the final results of my parallel calculations, no communication amongst the different processes is necessary. This ensured that race conditions were largely a non-issue for me and greatly simplified the writing of the code.

3.2 Amdahl's Law

Additionally, there is a limit to how much parallelization can improve the performance of a program. The precept that governs this behavior is referred to as Amdahl's Law[6].

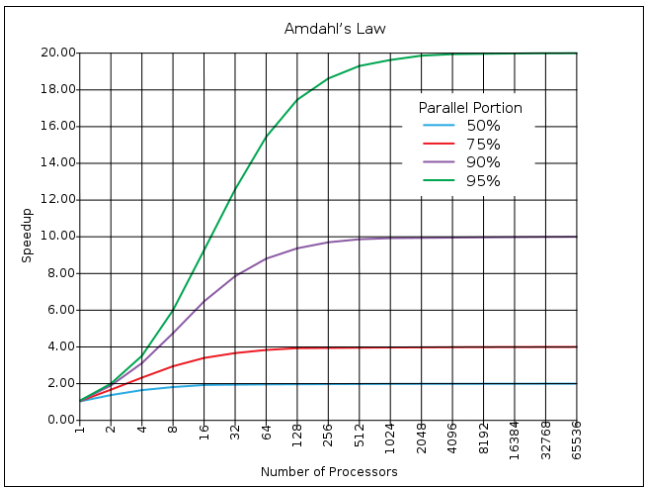

Figure 2: A graph of Amdahl's Law indicating the relationship between the percentage of a program that is parallelizable and the maximum speedup that a program can experience as a result of parallelization. Source: www.wikipedia.org

Amdahl's Law states that the maximum performance increase a program can expect with parallelization is contingent upon the percentage of a program's code that is parallelizable.

$$S = \frac{1}{\alpha} \tag{2}$$

Here, alpha represents the fraction of time that a program spends in non-parallelizable code. So, if only 10% of the code were non-parallelizable, the maximum speedup that could be expected from a program employing parallelization would be 10x irrelevant of how many processors were used to work on the program.

4 Implementation

4.1 Simple Integral Calculator Using MPI

To acquaint myself with the process of writing a parallelized, Monte Carlo calculator, I decided to begin working with a simple function.

$$\int_0^1 x^2 \tag{3}$$

The following is my initial integral calculator:

```
#include <math.h>
#include <mpi.h>
#include <gsl/gsl_rng.h>
#include ``gsl-sprng.h''

#define NUMVARS 10000000

//compile
//mpicc -I/usr/local/src/sprng2.0/include -L/usr/local/src/sprng2
    .0/lib
//       -o monte-carlo monte-carlo.c -lsprng -lgmp -lgsl -lgslcblas

//execute
//mpirun -np 2 monte-carlo

int main(int argc,char *argv[])
{
  int i,rank,size;
  double x,tempsum,totalsum;
  gsl_rng *r;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  r=gsl_rng_alloc(gsl_rng_sprng20);

  //f(x) = x^2
  for (i=0;i<NUMVARS;i++) {
    x = gsl_rng_uniform(r);
    tempsum += pow(x, 2);
  }
  MPI_Reduce(&tempsum,&totalsum,1,MPI_DOUBLE,MPI_SUM,0,
      MPI_COMM_WORLD);
  if (rank == 0) {
    printf("The Monte Carlo estimation for f(x) = x^2 is %f\n", (
        totalsum/NUMVARS)/size);
  }
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

After including the relevant libraries, I began by coding up a preliminary draft of my program which included the initial call to MPI_Init. Afterward, I invoked

the a GSL function to create a random number generator (gsl_rng_alloc) and seeded it with a struct that contained calls to the SPRNG library, thus avoiding the problem of generating a random number generator using exclusively the GSL library inside of a parallelized program. I then created a loop that would continue to generate a uniform [0,1) random variable x and keep a running total of that variable evaluated inside of the function.

```
1   for (i=0;i<NUMVARS;i++) {
2     x = gsl_rng_uniform(r);
3     tempsum += pow(x, 2);
4   }
```

This loop is the portion of the code that benefits the most from parallelization. Each individual process will generate its own series of random variables and use them to evaluate the integral. Once a process is finished calculating a running total of the results of the function, the temporary sum is then passed into a function called MPI_Reduce which reduces all of the tempsum quantities into a single totalsum variable which is then handled by the master process (for my implementation of parallelized Monte Carlo, the process of rank 0 is treated as the master process).

```
1 MPI_Reduce(&tempsum,&totalsum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD
      );
```

Once each process has finished calculating its own temporary total and the summation of the temporary totals has been stored in a single variable named totalsum, the master process prints out totalsum divided by the number of variables generated in total.

```
1   if (rank == 0) {
2     printf("The Monte Carlo estimation for f(x) = x^2 is %f\n",
3     (totalsum/NUMVARS)/size);
4   }
```

To compile the program, I run the following command:

```
1 mpicc -I/usr/local/src/sprng2.0/include -L/usr/local/src/sprng2.0/
      lib
2 -o monte-carlo monte-carlo.c -lsprng -lgmp -lgsl -lgslcblas
```

This invokes the MPI frontend for the gcc compiler. And then to execute the program, I type the following into the console:

```
1 mpirun -np 2 monte-carlo
```

This executes the program as two processors and on two cores.

The topology for this initial integral calculator is exceedingly simple. All processes, including the master process, perform Monte Carlo calculations. The distinguishing feature of the master process in this example is that the master process is responsible for handling the summation of the calculations performed by each process, including its own, printing out the final result.

4.2 Oscillatory Integral

Once I'd finished creating a simple integral calculator using MPI, I moved on to perform integration on a more complex integral. The next integral that I worked with necessitated a different approach, and I designed my second integral calculator with a different topology and overall concept in mind.

The following is the oscillatory integral that I designed my next program around where c is an N-dimensional, scaling vector and r is a parameter.

$$\int_0^1 \cos(2 * \pi * r + \sum_{i=1}^{N} c_i * x_i) \tag{4}$$

Because the scaling vector only needs to be generated once, I decided to reorganize the topology of my program. Instead of the master process being responsible for performing Monte Carlo calculations, the master process would simply generate the scaling vectory and then, using MPI_Send and MPI_Recv, it would send the scaling vector to each of the slave processes which would incorporate it into their Monte Carlo calculations.

What follows is my second program for the oscillatory integral.

```
1  #include <math.h>
2  #include <mpi.h>
3  #include <gsl/gsl_rng.h>
4  #include ''gsl-sprng.h''
5
6  #define MSG_DATA 100
7  #define MSG_RESULT 101
8  #define NDIM 3
9
10 //compile
11 //mpicc -I/usr/local/src/sprng2.0/include -L/usr/local/src/sprng2
       .0/lib -o
12 //oscillatory  oscillatory.c -lsprng -lgmp -lgsl -lgslcblas
13
14 //execute
15 //mpirun -np 2 oscillatory
16
17 double const pi = 3.14159265358979323844;
18
19 void master(void);
20 void slave(void);
21
```

```
22 int main(int argc,char *argv[])
23 {
24   int rank;
25   MPI_Init(&argc,&argv);
26   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
27
28   if(rank == 0)
29     master();
30   else
31     slave();
32
33   MPI_Finalize();
34   exit(EXIT_SUCCESS);
35 }
36
37 void master(void){
38
39   double c[NDIM], total = 0, tmptotal = 0;
40   int size;
41   gsl_rng *r;
42   MPI_Status status;
43   int i;
44
45   MPI_Comm_size(MPI_COMM_WORLD,&size);
46
47   r=gsl_rng_alloc(gsl_rng_sprng20);
48   for(i = 0; i < NDIM; ++i)
49     c[i] = gsl_rng_uniform(r);
50
51   for(i = 1; i < size; ++i)
52     MPI_Send(c, NDIM, MPI_DOUBLE, i, MSG_DATA, MPI_COMM_WORLD);
53   for(i = 1; i < size; total += tmptotal, ++i)
54     MPI_Recv(&tmptotal, 1, MPI_DOUBLE, i, MSG_RESULT,
           MPI_COMM_WORLD,
55     &status);
56
57
58   printf(``The Monte Carlo approximation of the oscillatory
         function
59   is %f with dimension %d with %d processes\n'',
60   (total/10000)/(size - 1), NDIM, size);
61 }
62
63 void slave(void){
64
65   double c[NDIM], x[NDIM], summation = 0,
66   iteration_total = 0, processtotal = 0;
67   gsl_rng *r;
68   MPI_Status status;
69   int i, j, rank;
70
71   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
72
73   MPI_Recv(c, NDIM, MPI_DOUBLE, 0, MSG_DATA, MPI_COMM_WORLD, &
         status);
74
75   r=gsl_rng_alloc(gsl_rng_sprng20);
```

```
76
77    //f(x) = cos ( 2 * pi * r + sum ( c(1:m) * x(1:m) ) )
78    for(i = 0; i < 10000; ++i){
79      for(j = 0; j < NDIM; ++j)
80        x[j] = gsl_rng_uniform(r);
81      for(j = 0; j < NDIM; ++j)
82        summation += x[j] * c[j];
83
84      iteration_total = pi * 2 + summation;
85      iteration_total = cos(iteration_total);
86
87      processtotal += iteration_total;
88
89    }
90
91    MPI_Send(&processtotal, 1, MPI_DOUBLE, 0, MSG_RESULT,
          MPI_COMM_WORLD);
92  }
```

For this program, I created two seperate functions that would be accessed by different classes of processes. The master() function would be accessed by the master process (process 0) and the slave processes would access the slave() function.

Once the master process entered its function, it would compute the scaling vector c based on the number of dimensions of the integral (defined at the top of the program as NDIM).

```
1    r=gsl_rng_alloc(gsl_rng_sprng20);
2    for(i = 0; i < NDIM; ++i)
3      c[i] = gsl_rng_uniform(r);
```

Once the scaling vector had been generated, the master process would then proceed to send the scaling vector to each of the slave processes using the MPI_Send function.

```
1    for(i = 1; i < size; ++i)
2      MPI_Send(c, NDIM, MPI_DOUBLE, i, MSG_DATA, MPI_COMM_WORLD);
```

The initial parameter of the send function is the address of the data being sent. The second parameter indicates the number of blocks that correspond with that address. The third parameter indicates the type of object being sent (there are a variety of built-in types for MPI that roughly correspond to the primitives available inside of C, including things like MPI_INT and the like). The fourth parameter is an integer indicating the rank of the process that the data is being sent to, which will depend upon the integer value stored the counter variable. The loop itself will only run until every process has been sent the scaling vector. The variable size contains the value indicating how many

processes have been created. The MSG_DATA value is an integer value that associates a given MPI_Send function with a corresponding MPI_Recv function with the same integer value. Lastly, MPI_COMM_WORLD is a communicator that contains all of the processes associated with the parallelized program. After sending the data to each slave process, the master process waits to receive the final calculations back from the individual slave processes.

Each slave process receives the scaling vector using a receive function that is structured similarly to the send function.

```
1   MPI_Recv(c, NDIM, MPI_DOUBLE, 0, MSG_DATA, MPI_COMM_WORLD, &
        status);
```

Each slave process will receive the scaling vector from the master process and then proceed to generate some number of random variables to perform Monte Carlo integration (10,000 times the dimensionality in this case). However, because this integral is multidimensional, the variable that's being evaluated will be represented as an array of randomly-generated integers named x. For each round of Monte Carlo integration, a new set of N (where N corresponds to the dimensionality of the integral) random variables is generated and stored inside of the array x. Once the variables that make up x have been generated, the appropriate math is performed and a temporary value for a given round of Monte Carlo is calculated called iteration_total. This value is later added to a running total of each round of Monte Carlo for a given process named processtotal.

```
1   for(i = 0; i < 10000; ++i){
2     for(j = 0; j < NDIM; ++j)
3       x[j] = gsl_rng_uniform(r);
4     for(j = 0; j < NDIM; ++j)
5       summation += x[j] * c[j];
6
7     iteration_total = pi * 2 + summation;
8     iteration_total = cos(iteration_total);
9
10    processtotal += iteration_total;
11
12  }
```

Once the process total has been calculated, it is then sent back to the master process using a call to MPI_Send().

```
1 MPI_Send(&processtotal, 1, MPI_DOUBLE, 0, MSG_RESULT,
      MPI_COMM_WORLD);
```

This data is then received by the master process using a similar MPI_Recv call nested within a loop that sums the totals of each process into a single variable named total.

```
for(i = 1; i < size; total += tmptotal, ++i)
  MPI_Recv(&tmptotal, 1, MPI_DOUBLE, i, MSG_RESULT, MPI_COMM_WORLD,
       &status);
```

Once the total calculations of each process have been condensed into a single variable, the master process proceeds to print out the final result of the Monte Carlo approximation.

```
printf("The Monte Carlo approximation of the oscillatory function
    is
%f with dimension %d with %d processes\n",
(total/10000)/(size - 1), NDIM, size);
```

Note that in this version of the parallelized, integral calculator, the master process doesn't perform any Monte Carlo work and so the total is divided by the number of variables generated on a given process and then again divided by the number of processes minus one.

4.3 Product Peak Integral

Lastly, I designed a program for evaluating a product-peak integral in N dimensions.

$$\int_0^1 \frac{1.0}{\prod_{i=1}^{N}(c_i)^2 + (x_i - x0_i)^2} \tag{5}$$

The final program I created was very similar to the integral calculator for the oscillatory integral. This calculator was designed to estimate the integral for a product peak integral using the same sort of topology and design as the oscillatory integral calculator. The only significant difference between the two programs was the portion of the slave function comprising the mathematical function to be evaluated.

```
for(i = 0; i < 90000000; ++i){
  for(j = 0; j < NDIM; ++j)
    x[j] = gsl_rng_uniform(r);
  for(j = 0; j < NDIM; ++j)
    iteration_total *= 1.0 / (pow(c[j], 2) + pow(x[j] - 1, 2));

  processtotal += 1.0 / iteration_total;
  iteration_total = 1.0;
}
```

The loop is similar to the loop in the previous program responsible for depicting the integrand. In this particular instance, I'm performing 90,000,000 rounds of Monte Carlo per process (save for the master process).

5 Results

In addition to implementing Monte Carlo integration, I also wanted to observe some of the behaviors of my programs. I wanted to see how the programs responded to changes in the number of rounds of Monte Carlo each calculator performed and also changes in the number of processes that I generated in a given execution of the program. In particular, I wanted to see how these changes related to the execution time of a given run of my program.

First, I began to increase the number of rounds of Monte Carlo a given process would execute. I noticed that the time it took for a program to execute increased linearly as the number of rounds of Monte Carlo per process increased. To calculate this, I created time_t variables and used functions created in the time.h library to computer the runtime in seconds of a given execution.

Rounds	Execution Time (in seconds)	Results
50000000	1	.333329
100000000	2	.333354
150000000	3	.333339
200000000	4	.333335

I also evaluated how the execution time increased in accordance with the number of concurrent processes that I created. I noticed that once the number of processes exceeded two, every two processes invoked from three onward increased my overall execution time linearly (so three processes resulted in a linear increase in execution time, followed by five, then seven and so on). I suspect this is because each core in my machine was assigned half of the processes.

Processes	Execution Time (in seconds)	Results
2	10	.333345
3	21	.333341
5	34	.333343
10	51	.333340

It's worth mentioning that the granuality at which I was evaluating the execution time of my programs was extremely coarse, given the limitations of the time.h library.

6 Future Work

Future work I would want to conduct using the implementation I'd set up on my machine and the example programs I'd developed include expanding my study of the behaviors of parallelized, Monte Carlo algorithms to more complex integrands, finding fine-grained tools for analyzing the time a given program takes to execute and attempting to execute these programs on machines with

additional cores available to me to better evaluate how additional cores and processors contribute to parallelization.

7 Conclusion

My primary goals in working on this project were to acquaint myself with parallel programming and to learn about implementing Monte Carlo integration in a programming context. Having no knowledge of parallelization to start with having only recently encountered Monte Carlo integration, I feel as though my primary goals were accomplished. I was able to set up a working MPI implementation on my workstation as well as compile from source the SPRNG libraries and install GSL. Once all of the necessary tools were installed, I was able to develop some examples of Monte Carlo integration that took advantage of parallelization with regard to several different kinds of integrands.

Secondary to getting implementing the aforementioned libraries and tools and getting a number of examples working, I wanted to evaluate the behavior of the calculators I developed. Unfortunately, I wasn't able to evaluate the programs at the granularity that I wanted to, but given the limitations of the tools I was using to evaluate the behavior of my programs, I was still able to notice some interesting things about parallel programming and also Monte Carlo integration.

Finally, I was hoping to provide documentation such that individuals interested in implementing parallelized, Monte Carlo calculators would be able to benefit from what I had encountered during my own work on the project by way of code samples and this document.

References

[1] Dartmouth Software Development in the UNIX Environment Using MPI (Message Passing Interface) http://www.dartmouth.edu/ rc/classes/-soft_dev/mpi.html

[2] Argonne National Laboratory http://www.mcs.anl.gov/research/projects/mpi/www/www3/

[3] The Scalable Parallel Random Number Generators Library (SPRNG) http://sprng.cs.fsu.edu/

[4] GNU Scientific Library http://www.gnu.org/software/gsl/

[5] Darren Wilkinson Darren Wilkinson's Research Blog http://www.staff.ncl.ac.uk/d.j.wilkinson/software/gsl-sprng.h

[6] Wolfram Alpha Demonstrations http://demonstrations.wolfram.com/AmdahlsLaw/

[7] Drexel University Monte Carlo Techniques http://www.physics.drexel.edu/courses/PHYS405/Monte_Carlo/index.html#Parallel_MC_Codes