Simulations

1. Ideal Parallel Computations
   - no communication between separate processes
2. Monte Carlo Simulations
   - counting successes of simulated experiments
   - SPRNG: scalable pseudorandom number generator
   - estimating π with SPRNG and MPI
   - the Mean Time Between Failures (MTBF) problem

MCS 572 Lecture 6
Introduction to Supercomputing
Jan Verschelde, 9 September 2024

Simulations

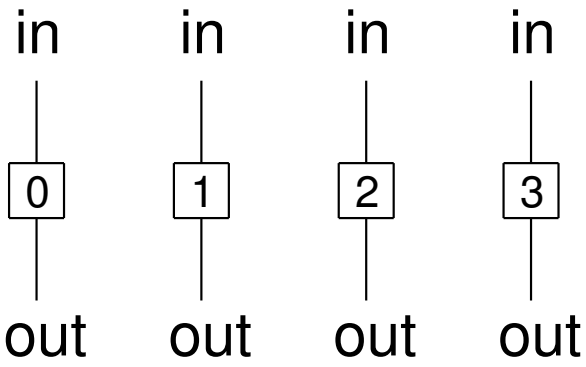
1. Ideal Parallel Computations
   - no communication between separate processes

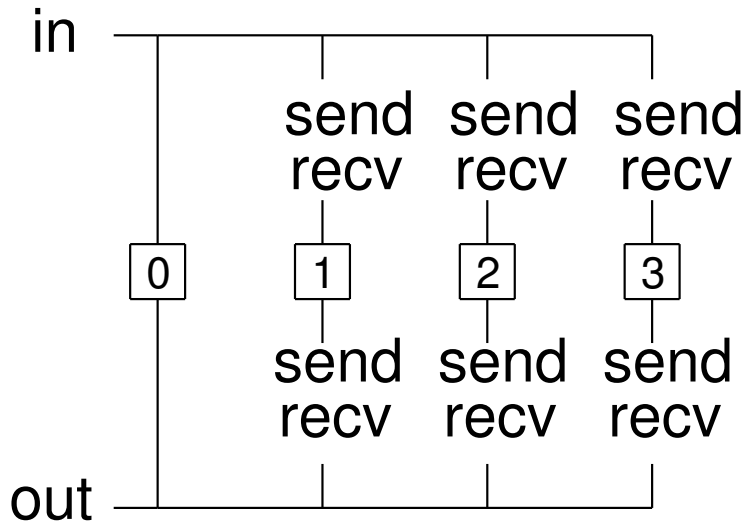
2. Monte Carlo Simulations
   - counting successes of simulated experiments
   - SPRNG: scalable pseudorandom number generator
   - estimating π with SPRNG and MPI
   - the Mean Time Between Failures (MTBF) problem

ideal parallel computations

or pleasingly parallel computations

Suppose we have a disconnected computation graph for 4 processes.





One manager node distributes input data to the compute nodes and collects results from the compute nodes.

Even if the work load is well balanced and all nodes terminate at the same time, we still need to collect the results from each node.

Without communication overhead, we hope for an optimal speedup.

examples

1. geometric transformations of images (§3.2.1 in textbook)

   Given an n-by-n matrix of pixels with RGB color encodings,
   the communication overhead is $O(n^2)$.
   The cost of transformation is at most $O(n^2)$.
   $\rightarrow$ not for message passing on distributed memory!

2. the computation of the Mandelbrot set

   Every pixel in the set may require up to 255 iterations.
   Pixels are computed *independently* from each other.

3. Monte Carlo simulations

   Every processor generates a *different* sequence of random samples and process samples *independently*.

Simulations

1. Ideal Parallel Computations
   - no communication between separate processes

2. Monte Carlo Simulations
   - counting successes of simulated experiments
   - SPRNG: scalable pseudorandom number generator
   - estimating π with SPRNG and MPI
   - the Mean Time Between Failures (MTBF) problem

Monte Carlo simulations

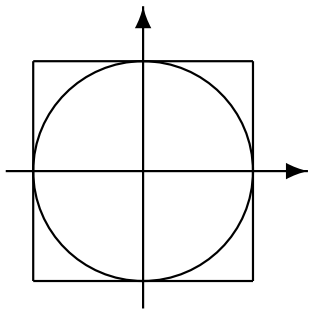
We simulate by

- repeatedly drawing samples along a distribution;
- counting the number of successful samples.

By the law of large numbers,
the average of the observed successes converges to the expected value or mean, as the number of experiments increases.

Estimating π, the area of the unit disk:

$$\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$$



Generate random uniformly distributed points with coordinates $(x, y) \in [0, +1] \times [0, +1]$.
We count a success when $x^2 + y^2 \leq 1$.

Simulations

1. Ideal Parallel Computations
   - no communication between separate processes

2. Monte Carlo Simulations
   - counting successes of simulated experiments
   - SPRNG: scalable pseudorandom number generator
   - estimating π with SPRNG and MPI
   - the Mean Time Between Failures (MTBF) problem

pseudorandom numbers

True random numbers do not exist on a computer...

A multiplicative congruential generator is determined by multiplier a, additive constant c, and a modulus m:

$$x_{n+1} = (ax_n + c) \bmod m, \quad n = 0, 1, \ldots.$$

Assumed: the pooled results of p processors running p copies of a Monte Carlo calculation achieves variance p times smaller.

However, this assumption is true only if the results on each processor are statistically independent. Some problems:

- choice of the seed determines period,
- repeating sequences, lattice effects.

And of course, the numbers should be generated fast.

the SPRNG software package

SPRNG: Scalable PseudoRandom Number Generators library designed to support parallel Monte Carlo applications.

- Web site: `www.sprng.org`.
- Version 5 was last updated 06/10/2021.
- Default compulation, just following the Quick Start.
- To install, type `make install`.

a simple use

```
#include <stdio.h>

#define SIMPLE_SPRNG /* simple interface */
#include <sprng.h>

int main ( void )
{
   printf("hello SPRNG...\n");
   double r = sprng();
   printf("a random double: %.15lf\n",r);
   return 0;
}
```

compiling, linking, and running

An example makefile:

```
INC=/usr/local/include  # location of sprng.h
LIB=/usr/local/lib      # location of libsprng.a
sprng_hello:
        g++ -I$(INC) sprng_hello.c \
              $(LIB)/libsprng.a -lgmp -o sprng_hello
```

then we can type `make sprng_hello` at the command prompt.
We run the program as

```
$ ./sprng_hello
hello SPRNG...
a random double: 0.014266541752498
```

a new seed with each run

To see a different random double with each run of the program, we generate a new seed, as follows:

```
#include <stdio.h>
#define SIMPLE_SPRNG
#include <sprng.h>

int main ( void )
{
   printf("SPRNG generates new seed...\n");
   /* make new seed each time program is run   */
   int seed = make_sprng_seed();
   printf("the seed : %d\n", seed);
   /* initialize the stream */
   init_sprng(seed, 0, SPRNG_DEFAULT);
   double r = sprng();
   printf("a random double: %.15f\n",r);
   return 0;
}
```

Simulations

1. Ideal Parallel Computations
   - no communication between separate processes

2. Monte Carlo Simulations
   - counting successes of simulated experiments
   - SPRNG: scalable pseudorandom number generator
   - estimating π with SPRNG and MPI
   - the Mean Time Between Failures (MTBF) problem

start of sprng_estpi.c

```
#include <stdio.h>
#include <math.h>
#define SIMPLE_SPRNG
#include <sprng.h>
#define PI 3.14159265358979

int main(void)
{
   printf("basic estimation of Pi with SPRNG...\n");
   int seed = make_sprng_seed();
   init_sprng(seed, 0, SPRNG_DEFAULT);

   printf("Give the number of samples : ");
   int n; scanf("%d", &n);
```

estimating π

```
    int i,cnt=0;
    for(i=0; i<n; i++)
    {
        double x = sprng();
        double y = sprng();
        double z = x*x + y*y;
        if(z <= 1.0) cnt++;
    }
    double estimate = (4.0*cnt)/n;
    printf("estimate for Pi : %.15f",estimate);
    printf("  error : %.3e\n",fabs(estimate-PI));

    return 0;
}
```

some runs

```
$ ./sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 100
estimate for Pi : 3.200000000000000  error : 5.841e-02
$ ./sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 10000
estimate for Pi : 3.131200000000000  error : 1.039e-02
$ ./sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 10000
estimate for Pi : 3.143600000000000  error : 2.007e-03
$ ./sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 1000000
estimate for Pi : 3.140704000000000  error : 8.887e-04
```

using MPI, the start of `sprng_estpi_mpi.c`

```
#include <stdio.h>
#include <math.h>
#define SIMPLE_SPRNG
#include "sprng.h"
#define PI 3.14159265358979
#include <mpi.h>

double estimate_pi ( int i, int n );
/* Estimation of pi by process i,
 * using n samples. */

int main ( int argc, char *argv[] )
{
   int id,np;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&np);
   MPI_Comm_rank(MPI_COMM_WORLD,&id);
```

the second half of the program

```
   int n;
   if(id == 0){
      printf("Reading the number of samples...\n");
      scanf("%d",&n);
   }
   MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
   double est4pi = estimate_pi(id,n);
   double sum = 0.0;
   MPI_Reduce(&est4pi,&sum,1,MPI_DOUBLE,MPI_SUM,0,
              MPI_COMM_WORLD);
   if(id == 0){
      est4pi = sum/np;
      printf("Estimate for Pi : %.15lf",est4pi);
      printf("  error : %.3e\n",fabs(est4pi-PI));
   }
   MPI_Finalize();
   return 0;
}
```

MPI_Reduce

`MPI_Reduce` is a collective communication function to reduce data gathered using some operations, e.g.: addition.

```
MPI_REDUCE(sendbuf,recvbuf,count,datatype,
           op,root,comm)
```

where the arguments are

sendbuf	:	address of send buffer
recvbuf	:	address of receive buffer
count	:	number of elements in send buffer
datatype	:	data type of elements in send buffer
op	:	reduce operation
root	:	rank of root process
comm	:	communicator

The predefined reduction operation `op` we use is `MPI_SUM`.

the estimate function

```
double estimate_pi ( int i, int n )
{
   int seed = make_sprng_seed();
   init_sprng(seed,0,SPRNG_DEFAULT);

   int j,cnt=0;
   for(j=0; j<n; j++)
   {
      double x = sprng();
      double y = sprng();
      double z = x*x + y*y;
      if(z <= 1.0) cnt++;
   }
   double estimate = (4.0*cnt)/n;
   printf("Node %d estimate for Pi : %.15f",i,estimate);
   printf("  error : %.3e\n",fabs(estimate-PI));

   return estimate;
}
```

the makefile

Because `g++` (the gcc C++ compiler) was used to build SPRNG, we must compile the code with `mpic++`.

The makefile contains the following lines:

```
sprng_estpi_mpi:
        mpic++ -I/usr/local/sprng \
               sprng_estpi_mpi.c -lsprng \
               -o sprng_estpi_mpi
```

Simulations

1. Ideal Parallel Computations
   - no communication between separate processes

2. Monte Carlo Simulations
   - counting successes of simulated experiments
   - SPRNG: scalable pseudorandom number generator
   - estimating π with SPRNG and MPI
   - the Mean Time Between Failures (MTBF) problem

mean time between failures

The Mean Time Between Failures (MTBF) problem asks for the expected life span of a product made of components.

Every component is critical. The multi-component product fails as soon as one of its components fails.

For every component we assume that the life span follows a known normal distribution, given by μ and σ. For example, consider 3 components with respective means 11, 12, 13 and corresponding standard deviations 1, 2, 3. Running 100,000 simulations, we compute the average life span of the composite product.

If $f_i(t)$ is the cumulative distribution function of ith component, then we estimate the triple integral:

$$\mu = \sum_{i=1}^{3} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} t \prod_{j \neq i} (1 - f_j(t)) df_i(t).$$

implementing the normal distribution

```
int normal ( double *x, double *y );
/*
 * DESCRIPTION :
 *   Generates two independent normally distributed
 *   variables x and y, along Algorithm P (Knuth Vol 2),
 *   the polar method is due to Box and Muller.
 *
 * ON ENTRY :
 *   x,y      two independent random variables,
 *            uniformly distributed in [0,1].
 *
 * ON RETURN : fail = normal(&x,&y)
 *   fail     if 1, then x and y are outside the unit disk,
 *            if 0, then x and y are inside the unit disk;
 *   x,y      independent normally distributed variables. */
```

code for the function `normal`

```
int normal ( double *x, double *y )
{
   double s;

   *x = 2.0*(*x) - 1.0;
   *y = 2.0*(*y) - 1.0;
   s = (*x)*(*x) + (*y)*(*y);

   if(s >= 1.0)
      return 1;
   else
   {
      double ln_s = log(s);
      double rt_s = sqrt(-2.0*ln_s/s);
      *x = (*x)*rt_s;
      *y = (*y)*rt_s;
      return 0;
   }
}
```

testing the function `normal`

For the generated numbers,
we compute the average μ and standard deviation σ.
We count how many samples are in $[\mu - \sigma, \mu + \sigma]$.

```
$ ./sprng_normal
normal variables with SPRNG ...
a normal random variable : 0.645521197140996
a normal random variable : 0.351776102906080
give number of samples : 1000000
mu = 0.000586448667516, sigma = 1.001564397361179
ratio of #samples in [-1.00,1.00] : 0.6822
generated 1572576 normal random numbers
```

To compile, the makefile contains the following

```
sprng_normal:
        g++ -I/usr/local/sprng sprng_normal.c \
            -lsprng -lgmp -o sprng_normal
```

mapping to any normal distribution

The header (specification) of the function is

```
double map_to_normal ( double mu, double sigma, double x );
/*
 * DESCRIPTION :
 *   Given a normally distributed number x with mean 0 and
 *   standard deviation 1, returns a normally distributed
 *   number y with mean mu and standard deviation sigma. */
```

The C code (implementation) of the function is

```
double map_to_normal ( double mu, double sigma, double x )
{
   return mu + sigma*x;
}
```

running sprng_mtbf

```
$ ./sprng_mtbf
MTBF problem with SPRNG ...
Give number of components : 3
average life span for part 0 ? 11
standard deviation for part 0 ? 1
average life span for part 1 ? 12
standard deviation for part 1 ? 2
average life span for part 2 ? 13
standard deviation for part 2 ? 3
mu[0] = 11.000  sigma[0] = 1.000
mu[1] = 12.000  sigma[1] = 2.000
mu[2] = 13.000  sigma[2] = 3.000
Give number of simulations : 100000
expected life span : 10.115057028769346
```

header of the `mtbf` function

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define SIMPLE_SPRNG
#include "sprng.h"

double mtbf ( int n, int m, double *mu, double *sigma );
/*
 * DESCRIPTION :
 *   Returns the expected life span of a composite product
 *   of m parts, where part i has expected life span mu[i],
 *   with standard deviation sigma[i],
 *   using n simulations. */
```

implementation of the `mtbf` function

```
double mtbf ( int n, int m, double *mu, double *sigma )
{
   int i,cnt=0;
   double s[n+1];
   do {
      normald(mu[0],sigma[0],&s[cnt],&s[cnt+1]);
      for(i=1; i<m; i++) {
         double x,y;
         normald(mu[i],sigma[i],&x,&y);
         s[cnt] = min(s[cnt],x);
         s[cnt+1] = min(s[cnt+1],y);
      }
      cnt = cnt + 2;
   } while (cnt < n);
   double sum = 0.0;
   for(i=0; i<cnt; i++) sum = sum + s[i];
   return sum/cnt;
}
```

further reading

- Stuart L. Anderson: **Random number generators on vector supercomputers and other advanced architectures**. *SIAM Review* 32(2): 221–251, 1990.
- Donald E. Knuth: **The Art of Computer Programming. Volume 2. Seminumerical Algorithms**. Third Edition. Addison-Wesley, 1997. Chapter Three.
- Michael Mascagni and Ashok Srinivasan: **Algorithm 806: SPRNG: a scalable library for pseudorandom number generation**. *ACM Transactions on Mathematical Software* 26(3): 436–461, 2000.

Summary + Exercises

Monte Carlo simulations are pleasingly parallel computations.

Exercises:

1. Consider the code for the estimation of π. For a fixed choice of the seed, examine the relationship between the error ϵ and the number of samples n. Make a plot relating n to $-\log_{10}(\epsilon)$, for sufficiently many experiments for different values of n so the trend becomes clear.
2. Consider the MPI code for the estimation of π. Fix the seeds so you can experimentally demonstrate the speedup. Execute the code for $p = 2, 4$, and 8 compute nodes.
3. Write a parallel version with MPI of `sprng_mtbf.c`.
   Verify the correctness by comparison with a sequential run.