New Jersey Institute of Technology

A. V. Gerbessiotis
March 31, 2006
PS 7 (60 points)

CIS 750
Spring 2006
Due:April 17

Problem 1. (60 points)

You are asked to approximate π in parallel using a Monte Carlo method. There are many ways to calculate π to variable digits of accuracy. We will not calculate π; we will only approximate it here. The π we are referring to is the well-known $\pi = 3.1415926535\ldots$. Here we will get an approximation of π with a not very fast convergence method. It is a Monte Carlo method a method otherwise used heavily in simulations, computational finance etc. One can solve our problem sequentially but the number of runs n can be quite high to get even a 4-5 digit approximation of π. For $n = 10,000,000$ the value of π one gets is 3.14089 which is accurate only to the second decimal digit. For $n = 100,000,000$ the running time on a Solaris workstation (roughly the equivalent of a Pentium 300 Mhz box) was a bit less than 2 minutes and yielded an approximation of 3.1418, whereas the same program on a 2.2Ghz Intel-based Linux machine took almost half a minute and gave a 3.14174 approximation. On `pcc16` such a run took somewhat less time than the Solaris box, it was only 89 seconds. You can only imagine what happens running-time wise if the number of runs is say 4 billion as it is going to be for your case.

However if p processors are available then one can spawn p processes, assign them to individual processors, run the sequential algorithm on each processor n/p times, and combine the results on a preagreed processor,say processor 0 to gain better accuracy. So on the 4 processors of the cluster, an 4 billion run, will require 1 billion runs per CPU.

`Pseudo-Random Number Generator.` You are going to use for this problem a random number generator. In fact you are going to use a pseudo random number generator available in C/C++ in the Standard C library called `random`. Do a `man random` for more details. `random` might give a random number in the range $[0, x]$. In order to use it in the problem you might have to scale up or down its output, appropriately and correctly. You might also have to cast it. Think how it can be done. You are not allowed to substitute other generators for `random()`; you are allowed to use other random number generators in ADDITION to `random`. That is, you first present results with `random` and then explore results one can get by using other pseudo-random number generators.

Suppose a program calls `random` 100 times. If you run the program 10 times, each time you are going to get the same sequence of "random" numbers. If you want to spice up and give different sequences in the 10 runs, you can use `srandom` to initialize the seed of `random`. What is a source of "randomness"? Perhaps the process id of a Unix process... How do you approximate π? Think of the following single experiment

```
0. Numbers below are real numbers.
   PiOne
1. You draw a random real number in the range [0,1]. Call it x;
2. You draw a random real number in the range [0,1]. Call it y;
3. Form pair (x,y). Call it point p.
4. Point p naturally belongs to the square with end points
   (0,0), (0,1), (1,1), (1,0).
5. Question: What is the probability that a uniformly at random
   drawn point such as p belongs to the circle inscribed in this square?
6. If p is inside that circle answer YES otherwise answer NO.
```

If you answer the Question of line 5, it will give you the answer to this problem i.e. how to approximate π. If you have run program `ccrand.java` while you were navigating around `pcc16` in the beginning of the semester, this is how that program worked.

```
    PiTwo (n)
0.  count = 0;
1.  Repeat n times
2.  if PiOne == YES
3.     count ++;
4.  Use count and n to estimate p.
```

Now say you run the experiment n times, i.e. call PiOne n times through PiTwo. The Law of Large Numbers tells us that if $n \to \infty$ the fraction of the hits (the YESes) observed approaches this probability (answer of the Question). The sequential version of the code `piseq` is shown below, not very elegant and not very optimized and fine-tuned.

```
#include <stdlib.h>
#include <math.h>
#define RAND_MAX ((unsigned int) ~0)
int main(int argc, char **argv)
{
   double value,x,y;
   int  i, number,N;
   value = 0.0;
   number= 0;
   N=atoi(argv[1]);
   srandom((unsigned int) 123445);
   for(i=0;i<N   ;i++) {
       x = (double) random()/RAND_MAX;
       y = (double) random()/RAND_MAX;
       value = sqrt((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5));
       if (value < 0.5 ) number++;
   }
   printf("Pi is approximately %f in %d iterations\n",(4.0*number)/N, N) ;
   return(0);
}
```

It takes one argument in the command line, the number of iterations/runs of the experiment above and should print this number along with the π approximation, i.e.

```
% ./piseq 100000000
Pi is approximately 3.14174 in 100000000 runs
```

Since the sequential program is `piseq` let us call the corresponding parallel program `pipar`.

If you run the `piseq` 100,000,000 you get an approximation π_1 of π. If you run the `piseq` 400,000,000 you get an approximation π_2 of π. π_2 should be better (and is so) than π_1.

If you run `pipar` on 4 processors with 100,000,000 runs per processor the results you get will depend on how careful you are when you use `random()`. If you are careless, you may get worse accuracy comparable to that of π_1 on 100,000,000 runs of `piseq` rather than that of π_2 with 400,000,000 runs! Can you guess why?

Write an "embarassingly parallel" program that has the same format as `piseq`. Call it `pipar`. This `pipar` also takes from the command line the number of runs n PER PROCESSOR (i.e. NOT the total number of runs!). It then allocates the processors requested in the command line of the `bsprun` (for BSPlib programs) or `mpirun` (for LAM MPI programs) and assigns that many runs to each processor. When each processor is about done, the answers (or number of hits within the circle) of each processor is averaged out appropriately in processor 0, and processor 0 prints the obtained approximation (which should be if not identical very close to the sequential case).

```
% bsprun -noload -local -npes 4 ./pipar 25000000
Pi is approximately 3.14174 using 100000000 runs (25000000 per processor).
% mpirun -np 4 ./pipar 25000000
Pi is approximately 3.14174 using 100000000 runs (25000000 per processor).
```

Run your program `piseq` for $n = 1,000,000$, $n = 10,000,000$, $n = 100,000,000$ and $p = 2, 4$.

Prepare and fill the following table with timing results (Table 1) and the approximation of π output (Table 2).

Time	PiSeq		PiPar
	P=1	P=2	P=4
n=1M	?	?	?
n=10M	?	?	?
n=100M	?	?	?

Pi's Approximation

	P=1	P=2	P=4
n=1M	?	?	?
n=10M	?	?	?
n=100M	?	?	?

0.1 Timing

Time the sequential and the parallel programs. What is the observed speedup? (Also check that no other fellow student is running his/her experiments at the same time, or you might be surprised).

A naive UNIX timing scheme can be invoked by

```
% time ./piseq 100000000
Pi is approximately 3.14174 using 100000000 runs
89.850u 0.190s 1:30.18 99.8%    0+0k 0+0io 218pf+0w
```

Use, however `bsp_time()` that provides wall-clock time not just CPU time. Or for MPI, `MPI_Wtime(void)`. As we mentioned in the title, this is an embarassingly parallel computation. You should get optimal speedups of p.

As a final note, you can use the source fragment listed here as your `piseq` implementation or you can further optimize it; there is too much room for optimization!

Deliverables. MPI and BSPlib implementations of `pipar` plus two tables of results.