CIS 786: Programming Component : Homework 3/Problem 1 (60 points)

1 Cluster Version of BSPlib

The BSPlib installation steps are similar to the uniprocessor installation. Some additional steps are required for the cluster version of the library to run parallel programs.

We assume that your account is `userX` in the remainder of this discussion.

0. Remove the uniprocessor version of BSPlib by doing a

```
% rm -rf /home/userX/BSP  # This is a comment line  # is equivalent to // in C++
```

1. Edit the `.cshrc` file to include at least the first two lines below

```
set path=($path /home/userX/BSP/bin)
setenv BSP_DEVICE MPASS_UDPIP
```

The last line is important if you plan to run the cluster version. The former line is probably already there from the uniprocessor installation. Interprocessor communication under BSPlib will utilize UDP/IP. You may also wish to add the following lines to `.cshrc`.

```
alias 13 'rlogin pcc13 -l userX'
alias 14 'rlogin pcc14 -l userX'
alias 15 'rlogin pcc15 -l userX'
alias 16 'rlogin pcc16 -l userX'
```

If you don't want to do all these separately and by hand, just grab the template file `cshrc` from `userX`.

```
% cp /home/user0/cshrc /home/userX/.cshrc    # You can grab the cshrc template from user0
```

2. Grab the tar file `v1.4a_bsplib_toolset.tar` and `tar xvf` it as before. Note the `a` after `1.4`!

```
% cp /home/user0/v1.4a_bsplib_toolset.tar /home/userX/
% cd /home/userX
% tar xvf v1.4a_bsplib_toolset.tar
% cd BSP
% ./configure    # and read step 3 below on how to answer the questions
% make           # be patient; it takes time
% make install   # do not forget this step to complete installation
```

The library takes time to compile, because it compiles itself about 9 times,

3. Follow the steps of the uniprocessor installation when you execute the `./configure` above except

```
a. if you are asked about architecture   use LINUX
b. communication medium                  use MPASS_UDPIP
c  number of processors                  use 8 or larger
d. switch                                use    Ethernet 100Mbit
e. full duplex or half duplex            use full duplex switch
f. Roundtrip time                        use 200 microseconds
g. Send latency                          use 100 microseconds
```

The last two figures are not very important; you can always change the values during linking time within your own program. So don't worry if you type wrong.

4. After the installation completes, test it with a `bspcc` or `which bspcc`.

```
% which bspcc
```

5. Copy the `ccp` script to `BSP/bin`. Also the `udpip` script.

```
% cp /home/user0/ccp   /home/userX/BSP/bin
% cp /home/user0/udpip /home/userX/BSP/bin
% chmod 755  /home/userX/BSP/bin/ccp
% chmod 755  /home/userX/BSP/bin/udpip
% rehash
% hashstat
```

Copy also the `bsptcphosts` file.

```
% cd /home/userX/
% cp /home/user0/bsptcphosts   .bsptcphosts
```

6. From now on you can use the `rcp, rsh` commands to send information from one machine to the other. Information is available by typing for example `man rcp`. Create a tar file of the BSP installation on `pcc16` by doing at `/home/userX` a `tar cvf BSP8.tar BSP` Do the following on `pcc16` to effect this.

```
% cd /home/userX
% mkdir run                        # this is going to be needed later
% tar cvf BSP8.tar BSP
% udpip
```

The `ccp` commands copy the `.cshrc` and `BSP8.tar` to all the machines of the cluster. The `udpip` script does these and in addition it creates a directory `run`, just as you did above on `pcc16`, untars `BSP8.tar` and installs BSPlib from that tar-file on the remaining machines.

The command `ccp`, cluster copy, takes as a first argument a single file and as a second argument a directory. It copies the file into the directory on all 4 machines of the cluster.

7. In order to run an executable file say `a.out`, it must first be copied to the `run` directory of all 4 machines. You can do that as follows.

```
% ccp a.out /home/userX/run
```

8. You are ready to start running programs. In order to allow interprocessor communication between any two of the four machines, run

```
% bsplibd -all
% bspload -all -start
```

Some communication programs are started for the cluster defined in `/home/userX/.bsptcphosts`
After you are done with your coding, testing, and program execution, do a

```
% bspshutd
% bspload -all -end
```

to shutdown the programs you had started before.

9. A straightforward way to compile a BSPlib program is to type

```
% bspcc -O3 -flibrary-level 2 file1.c -o file1   # -O3 is optimization level 3
% ccp file1 /home/userX/run
% cd /home/userX/run
% bsprun -noload -local -npes 4 ./file1
```

BSPlib will complain about : `the use of 'tempnam' is dangerous, better use 'mkstemp'`. Disregard this message. It is just a warning. It does not affect compilation. You can compile things under BSPlib under library-level 0, 1 or 2. The most efficient is level 2; the most debug-friendly but also slowest is level 0. For a uniprocessor version, it is better if you run it at level 1 than 2 for example. Do a `man bspcc` for more details on the differences.

10. Run one of the first programs of the new version of `phello` that has a modified `nprocs.c` file.

```
% cd /home/userX
% mkdir phello2
% cp /home/user0/phello2004v2.tar phello2/
% cd phello2
% tar xvf phello2004v2.tar
% make all
% ccp nprocs /home/userX/run
% cd /home/userX/run
% bsprun -noload -local -npes 4 ./nprocs
```

It should print something like

```
Hello World from process pcc16.njit.edu with id=0 of total 4
Hello World from process pcc13.njit.edu with id=1 of total 4
Hello World from process pcc14.njit.edu with id=2 of total 4
Hello World from process pcc15.njit.edu with id=3 of total 4
Number of processes allocated: 4
```

How does BSPlib correspond processor id's to IP addresses? It uses `.bsptcphosts`. The last address in that file MUST BE the local machine. It is assigned a processor id of zero. The other address assignments are top to bottom starting from one.

1.1 Checklist

- Have you updated `.cshrc` ?
- Have you copied `cshrc` from `user 0` into `.cshrc` in `userX`?
- Did you update/reinstall BSPlib on `pcc16`? Did you use the `1.4a` copy?
- Did you tar cvf the installation, copy it to the remaining cluster machines, and install it there using `udpip`? If in doubt, `rsh pcc13 ls -l BSP`.
- Did you create `run`?
- Before you run a BSP program did you `bsplibd -all , bspload` ?
- Did you by any chance reuse an old executable (compiled under the uniprocessor version) to run it under the cluster? It won't work!
- When you run a sequential program say, `iseq` located in the current working directory are you doing a `./iseq` or are you trying in vain an `irun`? Linux, for security reasons, does not have `.` (i.e. the local directory defined in the `$path` variable. Local files cannot be thus executed, unless one requests the execution explicitly, i.e. do an `./irun` on the command line.

- When you run a BSPlib program say, `ipar` located in the current working directory are you doing a `bsprun -noload -local -npes n ./ipar` or are you trying in vain a `bsprun -noload -local -npes irun` or even `bsprun -noload -local -npes n irun`? n is a number that indicates the number of processors requested.
- Are yoy trying `bsprun -noload -local -npes 1 ./irun`? $p = 1$ programs won't run under the cluster version!

2 Preliminaries

You might want to read the Postscript file `v1.4_bsplib.README.ps` that is available at any `BSP/` directory after you decompress it with `gzip -d v1.4_bsplib.README.ps.gz`.

In homework 2, you installed your own BSPlib uniprocessor version at `/home/userX`, where `X` is your account number. For this problem, you can still use for debugging that version; however, it will be quite slow (unless you compile/link things at library-level 1). Otherwise, follow the steps of the previous section and install your own copy of the cluster version of BSPlib.

Just to make sure things are ok,

```
% setenv
.... other lines printed
BSP_DEVICE=MPASS_UDPIP
```

run `setenv` and make sure you see a line like the line above.

Compile your executable file in some arbitrary directory and then run `ccp file /home/userX/run`, where `file` is the name of the BSPlib created executable. What `ccp` does is something as simple as

```
rcp ./$1 pcc13.njit.edu:$2/$1
rcp ./$1 pcc14.njit.edu:$2/$1
rcp ./$1 pcc15.njit.edu:$2/$1
rcp ./$1 pcc16.njit.edu:$2/$1
```

If you are not familiar with `rcp`, then do a `man rcp` to see how it works.

You are ready to run your `executable`. Go to the `run` directory and type in

```
bsprun -noload -local -npes 4 file command-line-arguments-if-any
```

Note that BSPlib manual pages are available on line

```
% man bspcc
% man bsp_put
% man bsp_time
```

3 An embarassingly parallel program

This is the first task of Problem 1 of HW3.

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 and it is 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 8 billion as it is going to be for your case.

3.1 Random Numbers and Pseudo Random Number Generators

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 8 processors of the cluster, an 8 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 `sandom` to initialize its seed. What is a source of "randomness"? Perhaps the process id of a Unix process...

3.2 Sequential Monte Carlo

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.

```
    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).

Run first a sequential version to this problem, i.e. implement sequentially PiOne, PiTwo.

Call your executable `piseq`. It should take 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 using 100000000 runs
```

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

If you run the `piseq` 8,000,000 times you call `random` twice as many in lines 1 and 2 of PiOne. Doing so you are going to get an approximation π_1 of π.

If you run `pipar` on 8 processors with 1,000,000 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 `piseq` on 1,000,000 runs not 8,000,000 runs. So be careful and think how to resolve these issues!

3.3 Parallel Monte Carlo

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 later in class), 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 8 ./piseq 12500000
Pi is approximately 3.14174 using 100000000 runs (12500000 per processor).
```

3.4 Testing sizes

Run your program `piseq` for $n = 10,000,000$, $n = 100,000,000$, $n = 1,000,000,000$, and $n = 2,000,000,000$. Run your program `pipar` for $p = 2, 4, 8$ with n/p runs per processor.

Prepare and fill the following table with timing results (Table 1) and the approximation of π output (Table 2). This will gain you a total of 16 points.

Time	PiSeq	PiPar			
	P=1	P=2	P=4	P=8	Table 1
n=10M	?	?	?	?	
n=100M	?	?	?	?	
n=1000M	?	?	?	?	
n=2000M	?	?	?	?	

Pi's Approximation

	P=1	P=2	P=4	P=8	Table 2
n=10M	?	?	?	?	
n=100M	?	?	?	?	
n=1000M	?	?	?	?	
n=2000M	?	?	?	?	

3.5 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 non bsplib-based timing is

```
% 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.

3.6 BSPlib functions

You will only use very simple comunication patterns. `bsp_time ()` will also be used. As we mentioned in the title, this is an embarassingly parallel computation. You should get optimal speedups of p.

3.7 Deliverables

1. A complete sequential program with the Makefile, and main function that works as claimed for `piseq`. (7 points)
2. A complete parallel program with the Makefile, and main function that works as claimed for `pipar`. (30 points)
3. A standalone function `pipar(int runsperproc)` that implements PiOne and PiTwo. You may decide to implement both PiOne and PiTwo within the body of this function. (7 points)

Note the overlap between 2 and 3. In fact it is even more so between 1 and 2 or 3. Freeze the code as per instructions of previous programming homeworks.