HPC Spring 2015 - Homework 1

Fang Fang

March 7, 2015

1 MPI ring communication

1.1 Check if processors have added their rank

```
mpirun -np 10 ./int_ring 2
```

```
The 0th communication, rank 1 received from 0 the message 0
The 0th communication, rank 2 received from 1 the message 1
The 0th communication, rank 3 received from 2 the message 3
The 0th communication, rank 4 received from 3 the message 6
The 0th communication, rank 0 received from 4 the message 10
The 1th communication, rank 1 received from 0 the message 10
The 1th communication, rank 2 received from 1 the message 11
The 1th communication, rank 3 received from 2 the message 13
The 1th communication, rank 0 received from 4 the message 20
The 1th communication, rank 4 received from 3 the message 16
```

1.2 Estimate latency

I run the program on my cims desktop, which has 8 processors. If I run on less than 8 "processors" the latency is about $2.4s/1e6/8 = 3 \times 10^{-4}s$. But if I fun on more than 8 "processors", for example 16, I get latency only about $5.28s/100/16 = 3.3 \times 10^{-3}s$, which is 10 times less.

- Command: `mpirun -np 8 ./int_ring 1000000`
  Result:

```
The 1000000th communication, rank 0 received from 7 the message 28000000
The 1000000th communication, rank 1 received from 0 the message 27999972
The 1000000th communication, rank 2 received from 1 the message 27999973
The 1000000th communication, rank 3 received from 2 the message 27999975
The 1000000th communication, rank 4 received from 3 the message 27999978
The 1000000th communication, rank 5 received from 4 the message 27999982
The 1000000th communication, rank 6 received from 5 the message 27999987
The 1000000th communication, rank 7 received from 6 the message 27999993
Rank 5 time elapsed after 1000000 communications is 2.421535 seconds.
Rank 7 time elapsed after 1000000 communications is 2.421631 seconds.
Rank 0 time elapsed after 1000000 communications is 2.422608 seconds.
Rank 1 time elapsed after 1000000 communications is 2.422202 seconds.
Rank 4 time elapsed after 1000000 communications is 2.421596 seconds.
Rank 2 time elapsed after 1000000 communications is 2.422626 seconds.
Rank 6 time elapsed after 1000000 communications is 2.421561 seconds.
Rank 3 time elapsed after 1000000 communications is 2.422720 seconds.
```

- Command: `mpirun -np 16 ./int_ring 100`
  Result:

```
Rank 5 time elapsed after 100 communications is 5.284727 seconds.
Rank 12 time elapsed after 100 communications is 5.283909 seconds.
Rank 8 time elapsed after 100 communications is 5.284612 seconds.
Rank 7 time elapsed after 100 communications is 5.283743 seconds.
Rank 4 time elapsed after 100 communications is 5.283496 seconds.
Rank 11 time elapsed after 100 communications is 5.283482 seconds.
Rank 6 time elapsed after 100 communications is 5.284468 seconds.
Rank 3 time elapsed after 100 communications is 5.285097 seconds.
Rank 9 time elapsed after 100 communications is 5.283373 seconds.
Rank 2 time elapsed after 100 communications is 5.285173 seconds.
Rank 0 time elapsed after 100 communications is 5.285373 seconds.
Rank 10 time elapsed after 100 communications is 5.284706 seconds.
Rank 13 time elapsed after 100 communications is 5.283416 seconds.
Rank 15 time elapsed after 100 communications is 5.283654 seconds.
Rank 1 time elapsed after 100 communications is 5.285642 seconds.
Rank 14 time elapsed after 100 communications is 5.284665 seconds.
```

- Run on 4 different machines crunchy1, crunchy3, crunchy4 and my own desktop box608 (one processor on each machine), with command `mpirun -np 4 -f hostfile ./int_ring 1000`
  Result:

```
Rank 2 hosted on box608.cims.nyu.edu runs time 1.466217 seconds.
Rank 0 hosted on crunchy3.cims.nyu.edu runs time 0.662318 seconds.
Rank 3 hosted on crunchy1.cims.nyu.edu runs time 0.822307 seconds.
Rank 1 hosted on crunchy4.cims.nyu.edu runs time 0.724422 seconds.
```

  The box608 is my desktop on 11th floor. I think the communication of box608 with other crunchy machines is slower.

1.3 Estimate bandwidth

Here we use an array of length $1e7/8$ which is of size $10MB$. The bandwidth is calculated as $10MB/latency$.
Command: `mpirun -np 4 ./int_ring 1`

```
Data size is 10.000000MB
Rank 1 hosted on box608.cims.nyu.edu runs time 0.027681 seconds.
Band width is 1445.013724 MB/s
Rank 2 hosted on box608.cims.nyu.edu runs time 0.028121 seconds.
Band width is 1422.436207 MB/s
Rank 3 hosted on box608.cims.nyu.edu runs time 0.027788 seconds.
Band width is 1439.472969 MB/s
Rank 0 hosted on box608.cims.nyu.edu runs time 0.028211 seconds.
Band width is 1417.905086 MB/s
```

Result on 4 machines crunchy1, crunchy3, crunchy4 and box608, one processor on each machine. Command: `mpirun -np 4 -f hostfile ./int_ring 1`

```
Data size is 10.000000MB
Rank 2 hosted on box608.cims.nyu.edu runs time 2.491606 seconds.
Band width is 16.053904 MB/s
Rank 3 hosted on crunchy1.cims.nyu.edu runs time 1.949215 seconds.
Band width is 20.521084 MB/s
Rank 0 hosted on crunchy3.cims.nyu.edu runs time 2.065284 seconds.
Band width is 19.367798 MB/s
```

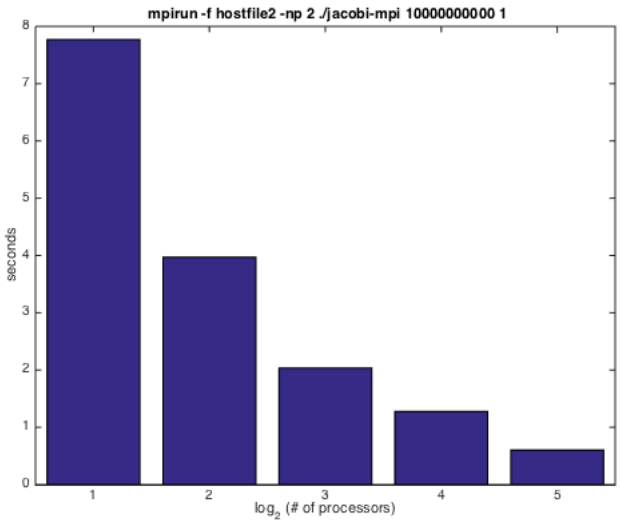

Figure 2.1: Strong scaling. One iteration of Jacobi smoother runs on machines crunchy3 and crunchy4, using 16 cores on each machine. Performance varies which may due to other users.

```
Rank 1 hosted on crunchy4.cims.nyu.edu runs time 1.968410 seconds.
Band width is 20.320972 MB/s
```

2 Distributed memory parallel Jacobi smoother

- Verify that the result is independent of p
  We verify this by direct output of the vector u.

```
mpirun -np 4 ./jacobi-mpi 12 5
u[0]=0.009549  u[1]=0.014323  u[3]=0.017144  u[4]=0.017361  u[5]=0.017361
u[6]=0.017361  u[7]=0.017361  u[8]=0.017144  u[9]=0.016493  u[10]=0.014323
u[11]=0.009549  u[2]=0.016493

mpirun -np 3 ./jacobi-mpi 12 5
u[0]=0.009549  u[1]=0.014323  u[2]=0.016493  u[4]=0.017361  u[5]=0.017361
u[6]=0.017361  u[7]=0.017361  u[8]=0.017144  u[9]=0.016493  u[10]=0.014323
u[11]=0.009549  u[3]=0.017144

mpirun -np 2 ./jacobi-mpi 12 5
u[0]=0.009549  u[1]=0.014323  u[6]=0.017361  u[7]=0.017361  u[8]=0.017144
u[9]=0.016493  u[10]=0.014323  u[11]=0.009549  u[2]=0.016493  u[3]=0.017144
u[4]=0.017361  u[5]=0.017361
```

- Study the strong scaling of your program
  We run the program on crunchy3:16 and crunchy4:16. The performance actually varies as other users are running jobs on servers. Results shown in Figure 2.1

- Why is a parallel version of the Gauss-Seidel smoother be significantly more difficult?
  For Gauss-Seidel smoother the newly computed value u_j^{k+1} is needed for computing its neighbors u_{j+1}^{k+1} and the followings. So if we distribute chunks of u^{k+1} onto different processors, the rank $p+1$ processor needs to wait until its predessesor rank p sent the updated value of the boundary points. Therefore the parallel version is basically still serial computing.