SF2568: Homework 3

Michael Hanke

January 22, 2021

1. In the lecture we considered the Rank Sort algorithm on a shared memory machine. The parallel algorithm had a running time $O(N)$ for N data on $P = N$ processors. We will modify the algorithm further. The innermost loop computes

$$r_i = \#\{a_j | a_j < a_i\} = \sum_{a_j < a_i} 1 = \sum_{j=1}^{N} (a_j < a_i)$$

where I used the matlab-definition

$$a_j < a_i = \begin{cases} 0, & \text{if false,} \\ 1 & \text{if true.} \end{cases}$$

   (a) Propose an algorithm based on Rank Sort with a running time of $O(\log N)$! (2)

   (b) How many processors are needed? What is the processor efficiency? (1)

   (c) Would you use this algorithm in practice? Why or why not? (1)

2. The following is an attempt to code the Odd-Even Transposition Sort as one SPMD program (as an MPI implementation would be).

```
evenprocess = (rem(i,2) == 0);
evenphase = 1;
for step = 0:N-1
   if (evenphase & evenprocess) | (~evenphase & ~evenprocess)
      send(a, i+1);
      receive(x, i+1);
      if x < a
         a = x;
      end
   else
      send(a, i-1);
      receive(x, i-1);
```

```
      if x > a
         a = x;
      end
   end
   evenphase = ~evenphase;
end
```

Determine wheter this code is correct, and if not, correct any mistakes. (3)

3. Write an MPI program which implements Odd-Even Transposition Sort!

   Your program should work for any number of data N and any number of processors. The parameter N should be given as a command line parameter. The data can be generated by using a (pseudo-) random number generator. I recommned to use the standard C-function `random` for that purpose. Since this function returns a `long int` between 0 and `RAND_MAX`, I recommend to use double values normalized to [0,1). The following (incomplete!) code snippet will provide a hint:

```
int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &P);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   /* Find problem size N from command line */
   if (argc < 2) error_exit(‘‘No size N given’’);
   N = atoi(argv[1]);
   /* local size. Modify if P does not divide N */
   I = N/P;
   /* random number generator initialization */
   srandom(myrank+1);
   /* data generation */
   for (i = 0; i < I; i++)
      x[i] = ((double) random())/(RAND_MAX+1);
}
```

   (a) Implement the program and show its correctness for a small N and a small number of processors! (3)

   (b) Measure the efficiency of your program by using a large number N (say, $N = 10^7$) on $P = 1,2,3,4,8$ processors. Provide a plot of the Speedup. Draw conclusions! Do not forget to use optimization switches when compiling and linking! (2)

For timing your program on `tegner` you need to use batch jobs. Instead of issuing the spattach command you need to use a procedure as follows:

1. I assume that you have given the `module` commands as in the interactive case. Moreover, the program to run is compiled and linked.

2. Convince yourself that you have forwardable tickets with a sufficiently long life time by issuing the command

   ```
   klist -Tf
   ```

   Your tickets should carry the flag "F" (forwadable). The expiration date should be several hours (if not even days) in the future. If this is not the case, execute the command `kinit --forwardable`.

3. Now you can submit your program to the batch queue. This is done by executing

   ```
   sbatch ./job.sh
   ```

   in the directory where your program resides. The shell script `job.sh` contains all information for the program run. Edit it according to your needs! Save the script `job.sh` in your program folder and make it executable (`chmod +x job.sh`). For those who are curious, here is its contents:

   ```
   #!/bin/bash -l
   # job name
   #SBATCH -J myjob
   # account
   #SBATCH -A edu21.sf2568
   # email notification
   #SBATCH --mail-type=BEGIN,END
   # <minutes> wall-clock time will be given to this job
   #SBATCH -t 00:<minutes>:00
   # Number of nodes
   #SBATCH --nodes=2
   # set tasks per node to max 24 in order to disable hyperthreading
   #SBATCH --ntasks-per-node=4
   module add i-compilers intelmpi
   mpirun -np 8 ./<your program>
   ```

   `<minutes>` indicates a limit on the expected run time. Don't use larger values than 5 minutes! Note that each nodes has 24 cores such that it is sufficient to use only one node. In order to get a feeling about the quality of the inter-node communication run your program with 8 processes both on one node and on two nodes. Compare the exacution times!

   The output of your job will be found after execution in the file `slurm-%jobid.out`. Please observe that you cannot use `stdin` for interactively providing input to your program!