LabX-9, BY 15 December 2021

How to do the exercises in this set

We are now increasing the usage of the bash scripts and pick up additional utility commands. Please consult the material in the Files section of canvas for this course, it may help with the syntax in the examples of the exercises. Use the C book of Kernighan & Ritchie (see Pages/Textbooks on canvas[1]). In canvas Files you have a summary of C language in slides file 3.1_Linux-C[2] . Also use the summary slides in file 3.1_Linux_Bash[3], the summary or the Makefile syntax is in file 3.6_Linux-Make[4].

In this assignment we consider the organization of a programming project with make and batch submission. **Most of the assignments are for trying and observing. The assignment 2(d) is to make a table and a scalability plot.**

On the delivery of your results: When you are ready with your program assignment, please create a directory with your name and copy the files there:

```
mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/   #it should exist already from exe1
mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx7
cp yourprogram /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx7/yourprogram
```

After you have finished copying submit also to canvas, so we can grade the arrivals. You do not have to send us e-mails when you complete this exercise.

Of course, if there are any problems or questions, please email me and/or Andrey:

"Igor Zacharov" I.Zacharov@skoltech.ru

"Andrey Kardashin" Andrey.Kardashin@skoltech.ru

[1] http://cslabcms.nju.edu.cn/problem_solving/images/c/cc/The_C_Programming_Language_(2nd_Edition_Ritchie_Kernighan).pdf

[2] https://skoltech.instructure.com/files/257760/download?download_frd=1

[3] https://skoltech.instructure.com/files/262799/download?download_frd=1

[4] https://skoltech.instructure.com/files/267631/download?download_frd=1

Further commands

Login to your sandbox account: ssh –p 2200 yourname@sandbox.zhores.net

touch
tee
make
awk

Some additional commands

a) The command touch creates a file if it doesn't exist (zero-length). If it exists, it updates the access times. Try:

```
touch junk
ls –l junk
```

b) The tee program is reading stdin and writing to stdout and to a file names as its argument. Try

cat blackhol.txt | tee b-2.txt

will dump the input file to the screen and write it to the file b-2.txt at the same time. If you repeat the command it will overwrite b-2.txt. With tee –a b-2.txt it will add to the output file (the append mode).

c) We use the "make" files as common way to organize compilation. The make system is looking at the dependency of files that belong to the project and runs compilation of only what's out of date.

The make command reads a standard file Makefile or makefile. Decide which you want to use. The special syntax in the file defines the dependencies and compilation flags.

We will use the following makefile in the project (simplified here to fit better on the page):
The rules to make the object files from source files are defined implicitly; we do not spell them out here. The action is preceded by a TAB (will not work without it!)

```
#!make
CC = mpicc
CFLAGS = -O3 -fopenmp -g
LFLAGS = -lm -lrt

MAIN1 = piprog_H.c
MAIN2 = piprog_OMP.c
SRC = picalc.c piutil.c
INC = piprog.h
SCRIPT = runmpi.sh runomp.sh runarr.sh

OBJ = $(SRC:.c=.o)
EXEC = pimpi  piomp

all : $(EXEC)

pimpi : $(MAIN1) $(OBJ)
	$(CC) -o $(@) $(CFLAGS) $(^) $(LFLAGS)

piomp : $(MAIN2) $(OBJ)
	$(CC) -o $(@) $(CFLAGS) $(^) $(LFLAGS)

$(MAIN1) $(MAIN2) $(OBJ): $(INC)

test :  pimpi
	export OMP_NUM_THREADS=2; ./piomp -a
clean :
	rm $(EXEC) $(OBJ)
tar :
	tar -cvzf lab9mpi.tar.gz $(MAIN1) $(MAIN2) \
		$(SRC) $(INC) $(SCRIPT) makefile
.PHONY: all clean tar
```

Get the makefile from the distribution in

~/../LINUX_SUPERCOMPUTERS/programs/lab9mpi.tar.gz

Unpack in an empty directory with tar –xf lab9mpi.tar.gz then before anything else:

```
module load compilers/gcc-7.3.1
module load mpi/openmpi-3.1.2
make –n
make
```

The make –n will show what commands will be executed, while make will actually execute them. The command make test will compile if necessary and run the test with openmp and mpi program. Change the tar file name such as to keep the original. You will use the tar file to copy the hierarchy to the Zhores machine.

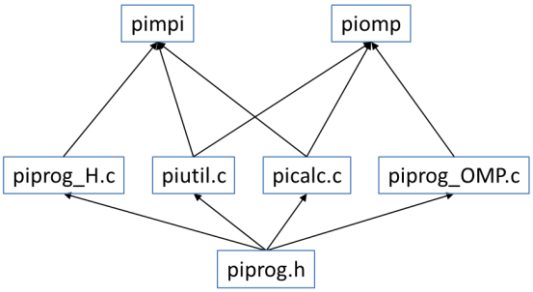
```
#!/usr/bin/bash

awk '/MASTER/{ nc=$4; cc=$6;
               if(cc > 0) {line++; n+=nc; circ+=cc;}
             }
 END{ pi = 4.0*circ/n; dpi = pi*sqrt(2.0/n);
      printf "\rline: %10d pi: %f +- %g\n",line,pi,dpi;
    } ' < log
```

d) The awk (gawk) is a utility to analyze text files. This reads the log file (part of the distribution) and in lines containing MASTER reads position 4 and 6, then computes PI. The file is called analyze.sh in the distribution. Run it. **What could be a way to check if all seeds are unique?**

Writing bash scripts for SLURM

1) The project structure is described by a make file and it is shown on the right.



All executables are obtained with `make` (or `make all`). Individual executables can be obtained with `make pimpi` and `make piomp` commands. This is visible from the structure of the makefile.

The meaning of different files is as follows:

The picalc.c contains the calculation part of the program and is parallelized with OpenMP.

The piutil.c is a collection of the utilities, in particular to analyze program arguments.

The piprog.h in include file with templates of functions and common definitions.

The piprog_H.c is a main program that contains MPI calls to facilitate message passing between processes and calls OpenMP parallel function for the calculation (the "Amdahl" approach). The piprog_OMP.c is the main program without MPI, calling same function with OpenMP.

a) use sdiff command to compare piprog_H.c and piprog_OMP.c

observe the piutil.c - it builds the input structure

the picalc.c is the familiar OpenMP calculation of the PI with random numbers.

b) Allocate resources and obtain interactive session on the nodes in sandbox:

The session is limited to 5 minutes (-t 5) , one node (-N 1), two tasks (-n 2) needed for MPI setup and run, two CPUs for each task (-c 2) and 1G of memory (--mem=1G).

After issuing CNTR-D the session is finished.

```
srun -p cpu -t 5 -N 1 -n 2 -c 2 --mem=1G
module load compilers/gcc-7.3.1
module load mpi/openmpi-3.1.2
make test
^D
```

c) the project can be unpacked with tar –xf lab9mpi.tar.gz (or using your own file name –change in the makefile). The hierarchy can be packed into a single (compressed) file with make tar command. Single file is easier to ship around with **`scp -P 2200 yourname@sandbox.zhores.net:lab9mpi.tar.gz .`** (**dot** at the end is for current directory where you issue the command i.e your laptop).

Then you can ship it to Zhores with the command **`scp lab9mpi.tar.gz yourname@cdise.login:`** (note the colon at the end – without it the copy will create file in your current directory).

Login to Zhores and unpack the project in an empty directory on Zhores proper.

2) Login on Zhores proper, allocate resources with salloc:

```
salloc -p htc -t 5 -N 1 -n 1 -c 4 --mem=1G
./piloop.sh
scancel $SLURM_JOBID
```

The piloop.sh starts 8 jobs, but srun serializes them because only 1 task is allowed to run at a time. Look at runPIomp.sh called from piloop.sh – this script sets number of OpenMP threads based on SLURM environment variable SLURM_CPUS_PER_TASK, which is set with the –c flag on the salloc or sbatch commands.

a) Running OpenMP parallel on Zhores. Allocate the resources:

```
salloc -p htc -t 5 -N 1 -c 8 --mem=1G
srun runPIomp.sh
scancel $SLURM_JOBID
```

Notice, if you just run ./runPIomp.sh it will try to run on the local node (an01). With the srun, it will run on the allocated node. (The node name is printed in the output).

b) Instead of the interactive session, same can be accomplished with the submission script. Look at the script runomp.sh:

The default parameters of the batch system are captured in the #SBATCH lines.

The submission is:

```
sbatch runomp.sh
```

The parameters on the command line have precedence over the default parameters in the file. Same can be accomplished with:

```
#!/usr/bin/bash
#SBATCH --job-name=piomp
#SBATCH --output=piomp_%j.out
#SBATCH --partition=cpu         # -p cpu
#SBATCH --nodes=1               # -N 1 number of nodes
#SBATCH --cpus-per-task=8       # -c X cpus assigned to each task
#SBATCH --time=2:00
#SBATCH --mem=1G                # memory per node

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK:-1}
echo "N-list: $SLURM_JOB_NODELIST #nodes: $SLURM_JOB_NUM_NODES"
echo "#tasks: $SLURM_NTASKS cpu: $SLURM_CPUS_PER_TASK"

module load compilers/gcc-7.3.1
make piomp &>/dev/null

mtrials=${1:-1}
numtrials=$mtrials
seed=$RANDOM

#echo " /usr/bin/time -p ./piomp -t $numtrials"
/usr/bin/time -p ./piomp -t $numtrials -a -t $numtrials -s $seed
```

```
sbatch –job-name=piomp –output=piomp_%j.out –p cpu –N 1 –c 8 –t 2 –mem=1G runomp.sh
```

c) The MPI parallel programs is best started with the script runmpi.sh:

The submission is:

```
sbatch –N 4 runmpi.sh
```

When nodes (–N) is 4, it matches the number of tasks, which is the number of MPI processes. In that case each MPI process will be placed on a different node.

If –N is 2, then we expect two MPI processes per node.

Note the setting –mincpus that is one higher than cpus-per-task. This

```
#!/usr/bin/bash
#SBATCH --job-name=pimpi
#SBATCH --output=pimpi_%j.out
#SBATCH --partition=cpu         # -p cpu
##SBATCH --nodes=4              # -N 4 number of nodes
#SBATCH --ntasks=4              # -n 4 total number of mpi processes
#SBATCH --cpus-per-task=4       # -c X cpus assigned to each task
#SBATCH --mincpus=5             # to ensure sufficient resources for MPI master thread
#SBATCH --time=2:00
#SBATCH --mem=1G                # memory per node

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK:-1}
export MPI_NUM_PROCESS=${SLURM_JOB_NUM_NODES:-4}
echo "N-list: $SLURM_JOB_NODELIST #nodes: $SLURM_JOB_NUM_NODES"
echo "#tasks: $SLURM_NTASKS cpu: $SLURM_CPUS_PER_TASK"

export OMPI_MCA_btl_base_warn_component_unused=0
export OMPI_MCA_btl="self,tcp"
export OMPI_MCA_orte_base_help_aggregate=0

module load mpi/openmpi-3.1.2
module load compilers/gcc-7.3.1
make pimpi &>/dev/null

mtrials=${1:-1}
numtrials=$mtrials
seed=$RANDOM

/usr/bin/time -p mpirun ./pimpi -a -t $numtrials -s $seed
```

is to make sure CPUS are available for the MPI starter thread.

d) Script to start multiple MPI parallel jobs with different number of CPUs: runSERIES.sh

```
#!/usr/bin/bash

time="2:00"
mem=1G
queue=cpu

for tasks in 2 4 8
do
    nodes=$tasks
    for ((cpus=1; cpus < 9; cpus*=2))
    do
        mincpus=$((cpus+1))
        echo "submit $queue output=PImpi_${tasks}_${cpus}.out -N $nodes -n $tasks -c $cpus --mincpus=$mincpus --mem=$mem --time=$time"

        sbatch -J PI-${tasks}-${cpus} -p $queue --output=PImpi_${tasks}_${cpus}.out -N $nodes -n $tasks -c $cpus \
                                                --mincpus=$mincpus --mem=$mem --time=$time runPImpi.sh 10
        sleep 5
    done
done
echo END
```

Use it to record run-times with different combinations of MPI and OMP tasks and make a scalability graph.

Таб. 1 Elapsed time of the run with multiple MPI Processes (2,4,8) and various number of OpenMP threads (1,2,4,8)

MPI/OMP	1	2	4	8
2				
4				
8				

e) Embarrassingly parallel

This is a way to start as many independent processes as possible with different parameters to achieve maximum throughput from the system. We demonstrate this capability with the PI calculation program and we use piomp when OMP_NUM_THREADS=1.

The script for array submission:
The distinctive feature is the line

```
SBATCH --array=1-1000%100
```

It asks the system to create 1000 jobs and execute them 100 jobs at the same time.

Same can be given on the command line, like:

```
sbatch -a 1-1000%100 runarr.sh
```

```
#!/usr/bin/bash
#SBATCH --job-name=piarr
#SBATCH --output=piarrJN.out
#SBATCH --partition=htc        # -p cpu
#SBATCH --cpus-per-task=1      # -c X cpus assigned to each task
#SBATCH --time=3:00
#SBATCH --mem=1G               # memory per node
#SBATCH --array=1-1000%100

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK:-1}
echo "N-list: $SLURM_JOB_NODELIST #nodes: $SLURM_JOB_NUM_NODES"
echo "#tasks: $SLURM_NTASKS cpu: $SLURM_CPUS_PER_TASK array: $SLURM_ARRAY_TASK_ID"

#env | grep SLURM_ARRAY
#echo

module load compilers/gcc-7.3.1
make piomp &>/dev/null

mtrials=${1:-1}
numtrials=$mtrials
seed=$(($RANDOM*$SLURM_ARRAY_TASK_ID))

#echo " /usr/bin/time -p ./piomp -t $numtrials"
./piomp -t $numtrials -a -t $numtrials -s $seed | tee -a log
```

We store all output inside a single file: --output=piarrJN.out, but the output of each job can be split with %j and %A modifiers in the name.

The coordination of the jobs is accomplished by collecting the output in to common file called log (it is with the filter: … | `tee -a log`). Each job **adds** a line with results (the tee –a flag) to a common file. Please, rename the original log file in the project to keep it for the comparison or rename your own output.

f) Analyze the log file with the script analyze.sh:

```
#!/usr/bin/bash


awk '/MASTER/{ nc=$4; cc=$6;
               if(cc > 0) {line++; n+=nc; circ+=cc;}
               pi = 4.0*circ/n; dpi = pi*sqrt(2.0/n);
               printf "\rline: %10d pi: %f +- %g  ",line,pi,dpi;
               system("sleep 0.01");
             }
    END{ pi = 4.0*circ/n; dpi = pi*sqrt(2.0/n);
               printf "\rline: %10d pi: %f +- %g\n",line,pi,dpi;
       } ' < log
```

It reads the file structured as follows:

```
……..
MASTER gn20.zhores Trials: 8000000000   Ncirc: 6283160786   Seed: 32209272     Threads(MPI,OMP): 1 1 Elapsed: 107.04  PI: 3.14158039 dpi: 5e-05
MASTER gn20.zhores Trials: 8000000000   Ncirc: 6283132115   Seed: 16778827     Threads(MPI,OMP): 1 1 Elapsed: 107.07  PI: 3.14156606 dpi: 5e-05
MASTER ct02.zhores Trials: 8000000000   Ncirc: 6283230912   Seed: 7579         Threads(MPI,OMP): 1 4 Elapsed: 33.35  PI: 3.14161546 dpi: 5e-05
MASTER ct02.zhores Trials: 8000000000   Ncirc: 6283228889   Seed: 1361         Threads(MPI,OMP): 1 4 Elapsed: 33.32  PI: 3.14161444 dpi: 5e-05
……..
```

and picks up in each line with MASTER the fields 4 (Trials) and 6 (Ncirc), then calculates the PI=4*Ncirc/Trials

After running your own experiments with array jobs, analyze your own log.

If you are more familiar with Python, *optionally* write a python script to do the same type of analysis.