Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

L02: Introduction, 11^{th} October 2024

Cluster Computing Introductory Example

- Run your new program `hn` using `srun` on two nodes with two processes each

Cluster Computing Introductory Example

- Run your new program `hn` using `srun` on two nodes with two processes each

```
[ikoutsou@front02 l01]$ srun -N 2 -n 4 -p p100 --reservation=sds406 -A sds406f24 ./hn
 Hostname is: cyc02
 Hostname is: cyc02
 Hostname is: cyc03
 Hostname is: cyc03
```

Cluster Computing Introductory Example

- Run your new program `hn` using `srun` on two nodes with two processes each

```
[ikoutsou@front02 l01]$ srun -N 2 -n 4 -p p100 --reservation=sds406 -A sds406f24 ./hn
 Hostname is: cyc02
 Hostname is: cyc02
 Hostname is: cyc03
 Hostname is: cyc03
```

It would be good if we could distinguish between processes of the same node.

Cluster Computing Introductory Example

- Run your new program `hn` using `srun` on two nodes with two processes each

```
[ikoutsou@front02 l01]$ srun -N 2 -n 4 -p p100 --reservation=sds406 -A sds406f24 ./hn
 Hostname is: cyc02
 Hostname is: cyc02
 Hostname is: cyc03
 Hostname is: cyc03
```

It would be good if we could distinguish between processes of the same node.

- Linux assigns each process with a unique `pid`. We should retrieve it in our program and print it alongside the hostname

Cluster Computing Introductory Example

- Run your new program `hn` using `srun` on two nodes with two processes each

```
[ikoutsou@front02 l01]$ srun -N 2 -n 4 -p p100 --reservation=sds406 -A sds406f24 ./hn
 Hostname is: cyc02
 Hostname is: cyc02
 Hostname is: cyc03
 Hostname is: cyc03
```

It would be good if we could distinguish between processes of the same node.

- Linux assigns each process with a unique `pid`. We should retrieve it in our program and print it alongside the hostname
- Use `emacs my_hn.c` to open the file again, then modify the C source code as follows:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
  char hname[256];
  pid_t p;
  gethostname(hname, 256);
  p = getpid();
  printf(" Hostname is: %s, process id is: %lu\n", hname, p);
  return 0;
}
```

Cluster Computing Introductory Example

- Compile again:

```
[ikoutsou@front02 l01]$ gcc my_hn.c -o hn
```

Cluster Computing Introductory Example

- Compile again:

```
[ikoutsou@front02 l01]$ gcc my_hn.c -o hn
```

- Run using `srun` on two nodes with two processes each (four processes in total)

```
[ikoutsou@front02 l01]$ srun -N 2 -n 4 -p p100 --reservation=sds406 -A sds406f24 ./hn
Hostname is: cyc02, process id is: 70224
Hostname is: cyc02, process id is: 70225
Hostname is: cyc01, process id is: 3844
Hostname is: cyc01, process id is: 3845
```

You will likely see different `pid`s in your case

Cluster Computing Introductory Example

- Compile again:

```
[ikoutsou@front02 l01]$ gcc my_hn.c -o hn
```

- Run using `srun` on two nodes with two processes each (four processes in total)

```
[ikoutsou@front02 l01]$ srun -N 2 -n 4 -p p100 --reservation=sds406 -A sds406f24 ./hn
Hostname is: cyc02, process id is: 70224
Hostname is: cyc02, process id is: 70225
Hostname is: cyc01, process id is: 3844
Hostname is: cyc01, process id is: 3845
```

  You will likely see different `pid`s in your case

- Go nuts 😆 :

```
[ikoutsou@front02 l01]$ srun -N 2 -n 16 -p p100 --reservation=sds406 -A sds406f24 ./hn
Hostname is: cyc01, process id is: 3891
Hostname is: cyc02, process id is: 70291
Hostname is: cyc01, process id is: 3889
 ...
Hostname is: cyc02, process id is: 70284
Hostname is: cyc02, process id is: 70268
Hostname is: cyc02, process id is: 70280
```

  (You will see 16 lines; I suppressed some above)

C programming

- A short reminder on C programming

```
#include <unistd.h>           // ⟵ provides definitions for gethostname() and getpid()
#include <stdio.h>            // ⟵ provides definitions for printf()
#include <sys/types.h>        // ⟵ defines the pid_t type

// main() is a function that returns an integer
// main() takes two arguments;
//    → argv[] is an array of strings which holds all command line arguments
//    → argc holds the number of elements of argv

int
main(int argc, char *argv[])
{
  char hname[256];          // ⟵ declare hname[] as an array of 256 characters (a string of length 256)
  pid_t p;                  // ⟵ declare p as a pid_t type, in this case, an unsigned long integer
  gethostname(hname, 256); // ⟵ call gethostname(), return hostname in hname which is 256 characters long
  p = getpid();             // ⟵ call getpid(), return value in p
  printf(" Hostname is: %s, process id is: %lu\n", hname, p); // print statement, see below
  return 0; // ⟵ return a value of 0 to the operating system. By convention 0 means success.
             //     Non zero values indicate errors.
}
```

C programming

- The print statement

```
printf(format, ... );
```

- `format`: a string which can include any number of conversion specifications (starting with `%`)
- `...`: arguments, to be converted to strings, one for each conversion specification

C programming

- The print statement

```
printf(format, ... );
```

- `format`: a string which can include any number of conversion specifications (starting with `%`)
- `...`: arguments, to be converted to strings, one for each conversion specification

```
printf(" Hostname is: %s, process id is: %lu\n", hname, p);
```

- `%s`: take the first argument (`hname`) and convert it as a string
- `%lu`: take the second argument (`p`) and convert it as an unsigned long integer

C programming

- Let's use a simple program to compute π in parallel

C programming

- Let's use a simple program to compute π in parallel
- Copy a program I have prepared for you from our shared storage to your home directory:

```
[ikoutsou@front02 l01]$ cd ..
[ikoutsou@front02 SDS406]$ mkdir l02
[ikoutsou@front02 SDS406]$ cd l02
[ikoutsou@front02 l02]$ cp /onyx/data/sds406f24/l02/pi.c .
```

- Inspect `pi.c`, e.g.:

```
[ikoutsou@front02 l02]$ emacs -nw pi.c
```

C programming

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;
  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

C programming

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;
  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

- Compile and run on frontend:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ ./pi
N =             10000   pi = 3.136400
```

C programming

- Now run, e.g. on 4 processes:

C programming

- Now run, e.g. on 4 processes:

```
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
```

C programming

- Now run, e.g. on 4 processes:

```
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =             10000   pi = 3.148800
N =             10000   pi = 3.148800
N =             10000   pi = 3.148800
N =             10000   pi = 3.148800
```

We get exactly the same result four times 🤔

C programming

- Now run, e.g. on 4 processes:

```
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
N =            10000   pi = 3.148800
```

We get exactly the same result four times 🤔

We need to seed the random number generator differently for each process

C programming

- Now run, e.g. on 4 processes:

```
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =             10000   pi = 3.148800
N =             10000   pi = 3.148800
N =             10000   pi = 3.148800
N =             10000   pi = 3.148800
```

We get exactly the same result four times 🤔

We need to seed the random number generator differently for each process

- Use the process id (`pid`) from the previous example, to seed the random number generator

C programming

- Use the process id (`pid`) from the previous example, to seed the random number generator

C programming

- Use the process id (`pid`) from the previous example, to seed the random number generator

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;

  pid_t p = getpid(); // <-- Add this
  srand48(p);         // <-- Add this

  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

C programming

- Use the process id (`pid`) from the previous example, to seed the random number generator

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;

  pid_t p = getpid(); // ⟵ Add this
  srand48(p);         // ⟵ Add this

  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

- `srand48()` sets the random number generator seed

C programming

- Use the process id (`pid`) from the previous example, to seed the random number generator

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int
main(int argc, char *argv[])
{
  unsigned long int N = 10000;
  unsigned long int nhit = 0;

  pid_t p = getpid(); // ⟵ Add this
  srand48(p);         // ⟵ Add this

  for(int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }
  double pi = 4.0 * (double)nhit/(double)N;
  printf(" N = %16d   pi = %lf\n", N, pi);
  return 0;
}
```

- `srand48()` sets the random number generator seed
- Need a unique seed for each instance of the program $\Rightarrow$ use process id.

C programming

- Compile again and run:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

C programming

- Compile again and run:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

- Now we would like to average over these four values to obtain a better estimate of π

C programming

- Compile again and run:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

- Now we would like to average over these four values to obtain a better estimate of π
- First redirect the output to a file, e.g.:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi > pi-out.txt
```

C programming

- Compile again and run:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi
N =            10000   pi = 3.150800
N =            10000   pi = 3.143200
N =            10000   pi = 3.151200
N =            10000   pi = 3.152800
```

- Now we would like to average over these four values to obtain a better estimate of π
- First redirect the output to a file, e.g.:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ srun -N 1 -n 4 -p p100 -A sds406f24 --reservation=sds406 ./pi > pi-out.txt
```

- Now `pi-out.txt` contains the four lines of output

```
[ikoutsou@front02 l02]$ ls
pi  pi.c  pi-out.txt
[ikoutsou@front02 l02]$ more pi-out.txt
N =            10000   pi = 3.113600
N =            10000   pi = 3.128400
N =            10000   pi = 3.156800
N =            10000   pi = 3.148400
[ikoutsou@front02 l02]$
```

C programming

- The program `awk` allows us to add over columns of a file.

C programming

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

C programming

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front02 l02]$ srun -N 2 -n 80 -p p100 -A sds406f24 --reservation=sds406 ./pi > pi-out.txt
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

C programming

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front02 l02]$ srun -N 2 -n 80 -p p100 -A sds406f24 --reservation=sds406 ./pi > pi-out.txt
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

  - `$6` is the sixth column in the file, the value for π on the given line
  - `pi_sum` is our summation variabl
  - `NR` is an AWK internal variable, the number of rows

C programming

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front02 l02]$ srun -N 2 -n 80 -p p100 -A sds406f24 --reservation=sds406 ./pi > pi-out.txt
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

  - `$6` is the sixth column in the file, the value for π on the given line
  - `pi_sum` is our summation variabl
  - `NR` is an AWK internal variable, the number of rows
- Let's wrap this up in a script "for posterity"

C programming

- The program `awk` allows us to add over columns of a file.
- E.g.:

```
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.136800
```

- Use more processes

```
[ikoutsou@front02 l02]$ srun -N 2 -n 80 -p p100 -A sds406f24 --reservation=sds406 ./pi > pi-out.txt
[ikoutsou@front02 l02]$ cat pi-out.txt | awk '{pi_sum+=$6}; END {printf "%8.6f\n", pi_sum/NR}'
3.142435
```

  - `$6` is the sixth column in the file, the value for π on the given line
  - `pi_sum` is our summation variabl
  - `NR` is an AWK internal variable, the number of rows

- Let's wrap this up in a script "for posterity"
- In fact, we'll write a *Slurm batch script*

C programming

- Copy from `/onyx/data/sds406f24/l02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p p100
#SBATCH -A sds406f24
#SBATCH --reservation=sds406
#SBATCH -t 00:02:00
#SBATCH -n 64
#SBATCH -N 2
#SBATCH --ntasks-per-node=32

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

C programming

- Copy from `/onyx/data/sds406f24/l02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p p100
#SBATCH -A sds406f24
#SBATCH --reservation=sds406
#SBATCH -t 00:02:00
#SBATCH -n 64
#SBATCH -N 2
#SBATCH --ntasks-per-node=32

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

- All Slurm options — that you so far used after `srun` — are now included in the lines starting with `#SBATCH`

C programming

- Copy from `/onyx/data/sds406f24/l02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p p100
#SBATCH -A sds406f24
#SBATCH --reservation=sds406
#SBATCH -t 00:02:00
#SBATCH -n 64
#SBATCH -N 2
#SBATCH --ntasks-per-node=32

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

- All Slurm options — that you so far used after `srun` — are now included in the lines starting with `#SBATCH`
- Thus `srun` is now run without options

C programming

- Copy from `/onyx/data/sds406f24/l02/pi.sh`

```
#!/bin/bash
#SBATCH -J pi
#SBATCH -o pi.txt
#SBATCH -e pi.err
#SBATCH -p p100
#SBATCH -A sds406f24
#SBATCH --reservation=sds406
#SBATCH -t 00:02:00
#SBATCH -n 64
#SBATCH -N 2
#SBATCH --ntasks-per-node=32

### Add these two lines
srun ./pi > pi-out.txt
cat pi-out.txt | awk '{sum+=$6}; END {printf "%8.6f\n", sum/NR}'
```

- All Slurm options — that you so far used after `srun` — are now included in the lines starting with `#SBATCH`
- Thus `srun` is now run without options
- Additional options include:
  - `-J`: sets the job name
  - `-o` and `-e`: set the files where the output and error should be redirected
  - `-t`: sets a time limit. The job will be killed if it exceeds this time (here 2 minutes)
  - `--ntasks-per-node=32`: is self explanatory

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front02 l02]$ sbatch pi.sh
Submitted batch job 198021
```

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front02 l02]$ sbatch pi.sh
Submitted batch job 198021
```

- Query its status. Filter only your jobs:

```
[ikoutsou@front02 l02]$ squeue -u $(whoami)
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        198021      p100       pi ikoutsou PD       0:00      2 (Reservation)
```

- Status: `PD`, `R`, `CG`, `CF`: "Pending", "Running", "Completing", "Configuring"

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front02 l02]$ sbatch pi.sh
Submitted batch job 198021
```

- Query its status. Filter only your jobs:

```
[ikoutsou@front02 l02]$ squeue -u $(whoami)
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        198021      p100       pi ikoutsou PD       0:00      2 (Reservation)
```

- Status: `PD`, `R`, `CG`, `CF`: "Pending", "Running", "Completing", "Configuring"
- After the program completes:
  - File `pi.err` contains any errors (hopefully empty)
  - File `pi.txt` contains the output of `awk` — what would be printed to the screen had you used `srun` like before
  - File `pi-out.txt` should also contain new values from the `srun` that was run during the script

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front02 l02]$ sbatch pi.sh
Submitted batch job 198021
```

- Query its status. Filter only your jobs:

```
[ikoutsou@front02 l02]$ squeue -u $(whoami)
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        198021      p100       pi ikoutsou PD       0:00      2 (Reservation)
```

- Status: `PD`, `R`, `CG`, `CF`: "Pending", "Running", "Completing", "Configuring"
- After the program completes:
  - File `pi.err` contains any errors (hopefully empty)
  - File `pi.txt` contains the output of `awk` — what would be printed to the screen had you used `srun` like before
  - File `pi-out.txt` should also contain new values from the `srun` that was run during the script

Now your task

Cluster Computing Introductory Example

- Submit the job

```
[ikoutsou@front02 l02]$ sbatch pi.sh
Submitted batch job 198021
```

- Query its status. Filter only your jobs:

```
[ikoutsou@front02 l02]$ squeue -u $(whoami)
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        198021      p100       pi ikoutsou PD       0:00      2 (Reservation)
```

- Status: `PD`, `R`, `CG`, `CF`: "Pending", "Running", "Completing", "Configuring"
- After the program completes:
  - File `pi.err` contains any errors (hopefully empty)
  - File `pi.txt` contains the output of `awk` — what would be printed to the screen had you used `srun` like before
  - File `pi-out.txt` should also contain new values from the `srun` that was run during the script

Now your task

Strong scaling of the calculation of π using this combination of program and script

Strong scaling example

Modify the script to obtain a *strong scaling* of the calculation of π using this combination of program and script

Strong scaling example

Modify the script to obtain a *strong scaling* of the calculation of π using this combination of program and script

- First, modify the C file. Change:

  ```
  unsigned long int N = 10000;
  ```

  to:

  ```
  unsigned long int N = atol(argv[1]);
  ```

Strong scaling example

Modify the script to obtain a *strong scaling* of the calculation of π using this combination of program and script

- First, modify the C file. Change:

```c
unsigned long int N = 10000;
```

to:

```c
unsigned long int N = atol(argv[1]);
```

- This allows passing the number of iterations from the *command line*. Compile and run this new version and run as follows:

```
[ikoutsou@front02 l02]$ gcc pi.c -o pi
[ikoutsou@front02 l02]$ ./pi 100
N =               100   pi = 3.120000
[ikoutsou@front02 l02]$ ./pi 1024
N =              1024   pi = 3.117188
[ikoutsou@front02 l02]$
```

Strong scaling example

Now modify the batch script looping over different numbers of tasks to provide a strong scaling

Strong scaling example

Now modify the batch script looping over different numbers of tasks to provide a strong scaling

- Note that:
  - In the same script you can call `srun` as many times as you like, or in a loop

Strong scaling example

Now modify the batch script looping over different numbers of tasks to provide a strong scaling

- Note that:
  - In the same script you can call `srun` as many times as you like, or in a loop
  - In the script you can use `--ntasks-per-node=` in the `srun` line (rather than after `#SBATCH`) which allows a different number for each invocation of `srun`

Strong scaling example

Now modify the batch script looping over different numbers of tasks to provide a strong scaling

- Note that:
  - In the same script you can call `srun` as many times as you like, or in a loop
  - In the script you can use `--ntasks-per-node=` in the `srun` line (rather than after `#SBATCH`) which allows a different number for each invocation of `srun`
  - You can use the shell command `date` to get the current time in nanoseconds (see slides that follow)

Strong scaling example

Now modify the batch script looping over different numbers of tasks to provide a strong scaling

- Note that:
    - In the same script you can call `srun` as many times as you like, or in a loop
    - In the script you can use `--ntasks-per-node=` in the `srun` line (rather than after `#SBATCH`) which allows a different number for each invocation of `srun`
    - You can use the shell command `date` to get the current time in nanoseconds (see slides that follow)
    - There are multiple ways to loop in a shell script. E.g.:

    ```
    for((n=2; n<=64; n+=2)); do
     ...
    done
    ```

    and the value of `n` is referenced as `$n` in the body of the iteration (denoted above by `...`)

Strong scaling example

Now modify the batch script looping over different numbers of tasks to provide a strong scaling

- Note that:
  - In the same script you can call `srun` as many times as you like, or in a loop
  - In the script you can use `--ntasks-per-node=` in the `srun` line (rather than after `#SBATCH`) which allows a different number for each invocation of `srun`
  - You can use the shell command `date` to get the current time in nanoseconds (see slides that follow)
  - There are multiple ways to loop in a shell script. E.g.:

    ```
    for((n=2; n<=64; n+=2)); do
     ...
    done
    ```

    and the value of `n` is referenced as `$n` in the body of the iteration (denoted above by `...`)
  - Integer math can be evaluated by enclosing expressions within `$((` and `))`.
    - E.g. `$(($n / 2))` will divide the value of `n` by two and return the result
    - As a shorthand, you can omit the leading `$` sign inside `$(( ... ))`. I.e. `$((n / 2))` is equivalent to the above

Bash scripting

Some bash basics

- Define variables. Whitespace matters

```
foo=1
```

Bash scripting

Some bash basics

- Define variables. Whitespace matters

```
foo=1
```

- Reference variables, e.g. to assign to another variable or to print

```
bar=$foo
echo $bar
```

Bash scripting

Some bash basics

- Define variables. Whitespace matters

```
foo=1
```

- Reference variables, e.g. to assign to another variable or to print

```
bar=$foo
echo $bar
```

- Statements enclosed in `$( … )` are executed, e.g.

```
[ikoutsou@front02 l02] all_files=$(ls)
[ikoutsou@front02 l02] echo $all_files
hn my_hn.c pi.sh pi pi.c
```

Bash scripting

Some bash basics

- Define variables. Whitespace matters

```
foo=1
```

- Reference variables, e.g. to assign to another variable or to print

```
bar=$foo
echo $bar
```

- Statements enclosed in `$( … )` are executed, e.g.

```
[ikoutsou@front02 l02] all_files=$(ls)
[ikoutsou@front02 l02] echo $all_files
hn my_hn.c pi.sh pi pi.c
```

- `$( … )`: other useful examples for capturing output:

```
# Get current time in nanoseconds
t0=$(date +%s%N)
```

Bash scripting

Some bash basics

- Define variables. Whitespace matters

```
foo=1
```

- Reference variables, e.g. to assign to another variable or to print

```
bar=$foo
echo $bar
```

- Statements enclosed in `$( … )` are executed, e.g.

```
[ikoutsou@front02 l02] all_files=$(ls)
[ikoutsou@front02 l02] echo $all_files
hn my_hn.c pi.sh pi pi.c
```

- `$( … )`: other useful examples for capturing output:

```
# Get current time in nanoseconds
t0=$(date +%s%N)
```

- Integer math can be enclosed in `$(( … ))` e.g.

```
[ikoutsou@front02 l02]$ a=16; b=8
[ikoutsou@front02 l02]$ echo $((a/b))
2
```

Strong scaling & plotting

Plotting the scaling

- Opportunity for a short python tutorial

Strong scaling & plotting

Plotting the scaling

- Opportunity for a short python tutorial
- Download or copy an example notebook:

  https://colab.research.google.com/drive/1E5wt7ww4FD51_g5MhECIdxqsWSj4mcIC?usp=sharing

Strong scaling & plotting

Plotting the scaling

- Opportunity for a short python tutorial
- Download or copy an example notebook:

  https://colab.research.google.com/drive/1E5wt7ww4FD51_g5MhECIdxqsWSj4mcIC?usp=sharing

```
from matplotlib import pyplot as plt
import matplotlib as mpl
import numpy as np
%matplotlib inline

data = np.loadtxt("pi.txt", usecols=(0,2))
times = data[:,0]/1e6
nproc = data[:,1]

fig = plt.figure(1)
fig.clf()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.plot(nproc, times, ls="-", color="r", marker="o")
ax.set_ylabel(r"T [ms]")
ax.set_xlabel(r"$n_{proc}$")
fig.canvas.draw()
fig.show()
```

Strong scaling & plotting

Plotting the scaling

- Opportunity for a short python tutorial
- Download or copy an example notebook:

  https://colab.research.google.com/drive/1E5wt7ww4FD51_g5MhECIdxqsWSj4mcIC?usp=sharing

```
from matplotlib import pyplot as plt
import matplotlib as mpl
import numpy as np
%matplotlib inline

data = np.loadtxt("pi.txt", usecols=(0,2))
times = data[:,0]/1e6
nproc = data[:,1]

fig = plt.figure(1)
fig.clf()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.plot(nproc, times, ls="-", color="r", marker="o")
ax.set_ylabel(r"T [ms]")
ax.set_xlabel(r"$n_{proc}$")
fig.canvas.draw()
fig.show()
```

- If using a local (i.e. on your laptop) installation of Python, you will need to copy `pi.txt` from the cluster

Strong scaling & plotting

Plotting the scaling

- Opportunity for a short python tutorial
- Download or copy an example notebook:

  https://colab.research.google.com/drive/1E5wt7ww4FD51_g5MhECIdxqsWSj4mcIC?usp=sharing

```
from matplotlib import pyplot as plt
import matplotlib as mpl
import numpy as np
%matplotlib inline

data = np.loadtxt("pi.txt", usecols=(0,2))
times = data[:,0]/1e6
nproc = data[:,1]

fig = plt.figure(1)
fig.clf()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.plot(nproc, times, ls="-", color="r", marker="o")
ax.set_ylabel(r"T [ms]")
ax.set_xlabel(r"$n_{proc}$")
fig.canvas.draw()
fig.show()
```

- If using a local (i.e. on your laptop) installation of Python, you will need to copy `pi.txt` from the cluster
- If using Colab you will then also need to upload `pi.txt`

Strong scaling & plotting

Plotting the scaling

- Opportunity for a short python tutorial
- Download or copy an example notebook:

  https://colab.research.google.com/drive/1E5wt7ww4FD51_g5MhECIdxqsWSj4mcIC?usp=sharing

```
from matplotlib import pyplot as plt
import matplotlib as mpl
import numpy as np
%matplotlib inline

data = np.loadtxt("pi.txt", usecols=(0,2))
times = data[:,0]/1e6
nproc = data[:,1]

fig = plt.figure(1)
fig.clf()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.plot(nproc, times, ls="-", color="r", marker="o")
ax.set_ylabel(r"T [ms]")
ax.set_xlabel(r"$n_{proc}$")
fig.canvas.draw()
fig.show()
```

- If using a local (i.e. on your laptop) installation of Python, you will need to copy `pi.txt` from the cluster
- If using Colab you will then also need to upload `pi.txt`
- This is an example. It is **not** a unique way of plotting the

Strong scaling & plotting

Plotting the scaling

- Add a second plot that plots the speed-up

Strong scaling & plotting

Plotting the scaling

- Add a second plot that plots the speed-up
- Vary the number of hits (N); overlay, on the same plot, the scaling (Speed-up vs n_{proc}) for various at least three values of N

Strong scaling & plotting

Plotting the scaling

- Add a second plot that plots the speed-up
- Vary the number of hits (N); overlay, on the same plot, the scaling (Speed-up vs n_{proc}) for various at least three values of N

Some notes:

- We're using `numpy` which allows for array operations, e.g.:

```
nproc = data[:,1]
```

Strong scaling & plotting

Plotting the scaling

- Add a second plot that plots the speed-up
- Vary the number of hits (N); overlay, on the same plot, the scaling (Speed-up vs n_{proc}) for various at least three values of N

Some notes:

- We're using `numpy` which allows for array operations, e.g.:

```
nproc = data[:,1]
```

- We're using `matplotlib` for plotting, and specifically its `pyplot` submodule (which we alias to `plt`)
  - `%matplotlib inline` works inside notebooks; draws the plots "inline" with the notebook cells
  - You can also save the figure at the end, e.g.: `plt.figure(1).savefig("scaling.pdf")`

Parallel calculation of π

For next lesson

- Carry out a *strong scaling* analysis of our π calculation script
- Use two sizes for N and scale using 2, ..., 64 processes with all even intermediate values:
  - N_1=518,918,400
  - N_2=4,151,347,200
- You should provide me with: i) the bash and ii) python script, as well as iii) your two `pi.txt` files (name them differently), and iv) the resulting plot as a figure

Programming in C: good practices

Some good practices when programming in C for our course

Programming in C: good practices

Some good practices when programming in C for our course

- The C programming language is rather low-level

Programming in C: good practices

Some good practices when programming in C for our course

- The C programming language is rather low-level
- Many checks are omitted in favor of performance and are therefor left to the programmer (here: *programmer* `==` *you*)

Programming in C: good practices

Some good practices when programming in C for our course

- The C programming language is rather low-level
- Many checks are omitted in favor of performance and are therefor left to the programmer (here: *programmer* `==` *you*)
- Common cases

Programming in C: good practices

Some good practices when programming in C for our course

- The C programming language is rather low-level
- Many checks are omitted in favor of performance and are therefor left to the programmer
(here: *programmer* `==` *you*)
- Common cases
    - Going off the end of an array:

    ```c
    int A[10];
    A[11] = 8;
    ```

Programming in C: good practices

Some good practices when programming in C for our course

- The C programming language is rather low-level
- Many checks are omitted in favor of performance and are therefor left to the programmer (here: *programmer* `==` *you*)
- Common cases
    - Going off the end of an array:

      ```
      int A[10];
      A[11] = 8;
      ```

    - Not checking the return value of a function:

      ```
      FILE *fp = fopen("file.txt", "r");
      fscanf(fp, fmt, &var);
      ```

      when "file.txt" doesn't exits

Programming in C: good practices

Some good practices when programming in C for our course

- The C programming language is rather low-level
- Many checks are omitted in favor of performance and are therefor left to the programmer (here: *programmer* `==` *you*)
- Common cases
  - Going off the end of an array:

    ```
    int A[10];
    A[11] = 8;
    ```

  - Not checking the return value of a function:

    ```
    FILE *fp = fopen("file.txt", "r");
    fscanf(fp, fmt, &var);
    ```

    when "file.txt" doesn't exits
  - Using `malloc()` and not checking the return value

    ```
    double *r = malloc(sizeof(double)*n);
    r[0] = 1;
    ```

    when `malloc()` returned `NULL`

Programming in C: good practices

In the following we will introduce some good practices when programming in C

Programming in C: good practices

In the following we will introduce some good practices when programming in C

- These are mostly functions we will define that wrap call system functions but also check their return value

Programming in C: good practices

In the following we will introduce some good practices when programming in C

- These are mostly functions we will define that wrap call system functions but also check their return value
- We will be reusing these repeatedly throughout the course

Programming in C: good practices

File I/O

- Opening files with `fopen()` does not fail gracefully in case of errors

```c
/***
 * Open file; print error if file pointer returned is NULL
 ***/
FILE *
uopen(const char *path, const char *mode)
{
  FILE *fp = fopen(path, mode);
  if(fp == NULL) {
    fprintf(stderr, "fopen(\"%s\", \'%s\') returned null; quitting...\n", path, mode);
    exit(-3);
  }
  return fp;
}
```

Programming in C: good practices

File I/O

- Opening files with `fopen()` does not fail gracefully in case of errors

```
/***
 * Open file; print error if file pointer returned is NULL
 ***/
FILE *
uopen(const char *path, const char *mode)
{
  FILE *fp = fopen(path, mode);
  if(fp == NULL) {
    fprintf(stderr, "fopen(\"%s\", \'%s\') returned null; quitting...\n", path, mode);
    exit(-3);
  }
  return fp;
}
```

- We will regularly be using code in which the `fopen()` function is wrapped in a function that checks its return value

Programming in C: good practices

Command-line arguments

- The length of `argv[]` is determined at run-time, meaning undefined behavior can occur if an insufficient number of arguments is specified on the command-line.

Programming in C: good practices

Command-line arguments

- The length of `argv[]` is determined at run-time, meaning undefined behavior can occur if an insufficient number of arguments is specified on the command-line.

```
/***
 * Print usage
 ***/
void
usage(char *argv[])
{
  fprintf(stderr, "usage: %s N\n", argv[0]);
  return;
}
```

- You will regularly see examples using a function to print the program usage (i.e. the command-line arguments expected by the program)

Programming in C: good practices

Parsing input

- Many times we need to check errors specific to the case, such as for `fscanf()` in the case of `average.c`

Programming in C: good practices

Parsing input

- Many times we need to check errors specific to the case, such as for `fscanf()` in the case of `average.c`

```c
int n_scanned = fscanf(fp, " Hostname is: %*s process id is: %*s   N = %*d   pi = %lf\n", &pi);
if(n_scanned != 1) {
    fprintf(stderr, " Wrong number of variables scanned\n");
    exit(2);
}
```

- When in doubt, use `man <function_name>` to lookup details on the invocation of functions

Programming in C: good practices

Parsing input

- Many times we need to check errors specific to the case, such as for `fscanf()` in the case of `average.c`

```
int n_scanned = fscanf(fp, " Hostname is: %*s process id is: %*s   N = %*d   pi = %lf\n", &pi);
if(n_scanned != 1) {
    fprintf(stderr, " Wrong number of variables scanned\n");
    exit(2);
}
```

- When in doubt, use `man <function_name>` to lookup details on the invocation of functions
- E.g. in this case:

  ```
  [ikoutsou@front02 ~]$ man fscanf
  ```

  will give you the *manual page* for the function `fscanf()`, which explains that the return value is the number of converted arguments

Programming in C: good practices

- Here are these good practices applied to `pi.c`

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

/***
 * Print usage
 ***/
void
usage(char *argv[])
{
  fprintf(stderr, " Usage: %s N\n", argv[0]);
  return;
}

int
main(int argc, char *argv[])
{
  if(argc != 2) {
    usage(argv);
    exit(1);
  }
  unsigned long int N = atol(argv[1]);
  char hname[256];
  pid_t p;
  gethostname(hname, 256);
  p = getpid();
  srand48(p);
  unsigned long int nhit = 0;
  for(unsigned long int i=0; i<N; i++) {
    double x = drand48();
    double y = drand48();
    if((x*x + y*y) < 1)
      nhit++;
  }

  double pi = 4.0 * (double)nhit/(double)N;
  printf(" Hostname is: %s, process id is: %lu   N = %16d   pi = %lf\n", hname, p, N, pi);
  return 0;
}
```

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;      /* ⟵ A pointer to a memory address that will hold an integer variable */
double *ptr1;   /* ⟵ A pointer to a memory address that will hold a double precision variable */
```

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;     /* ⟵ A pointer to a memory address that will hold an integer variable */
double *ptr1;  /* ⟵ A pointer to a memory address that will hold a double precision variable */
```

- Pointers are important in C because arrays are defined using offsets from a starting memory address

  ```
  int *ptr0 = malloc(sizeof(int)*7);
  ```

  Allocates memory that can hold seven integers and returns the starting memory address into `ptr0`

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;     /* ⟵ A pointer to a memory address that will hold an integer variable */
double *ptr1;  /* ⟵ A pointer to a memory address that will hold a double precision variable */
```

- Pointers are important in C because arrays are defined using offsets from a starting memory address

  ```
  int *ptr0 = malloc(sizeof(int)*7);
  ```

  Allocates memory that can hold seven integers and returns the starting memory address into `ptr0`

- Accessing elements is the same as calculating offsets from the starting memory address

  ```
  int element_3 = ptr0[3];
  ```

  is equivalent to:

  ```
  int element_3 = *(ptr0+3);
  ```

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;     /* ⟵ A pointer to a memory address that will hold an integer variable */
double *ptr1;  /* ⟵ A pointer to a memory address that will hold a double precision variable */
```

- This way of accessing elements also works when assigning:

  ```
  ptr0[3] = 42;
  ```

  is equivalent to:

  ```
  *(ptr0+3) = 42;
  ```

- Note that pointer offsets take into account the type of the pointer, e.g.:

  ```
  int element_2 = *(ptr0+2);
  ```

  adds `2*sizeof(int)` to the address pointed to by `ptr0` because `ptr0` is a pointer to an `int`, whereas:

  ```
  double element_4 = *(ptr1+4);
  ```

  adds `4*sizeof(double)` to the address pointed to by `ptr1` because `ptr1` is a pointer to a `double`.

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;      /* ⟵ A pointer to a memory address that will hold an integer variable */
double *ptr1;   /* ⟵ A pointer to a memory address that will hold a double precision variable */
```

- Note also the ampersand operator (`&`) which is in some way the inverse to `*`:

  ```
  double *ptr_to_element_4 = ptr1+4;
  ```

  is equivalent to:

  ```
  double *ptr_to_element_4 = &(ptr1[4]);
  ```

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;     /* ⟵ A pointer to a memory address that will hold an integer variable */
double *ptr1;  /* ⟵ A pointer to a memory address that will hold a double precision variable */
```

- The C compiler and runtime do not check whether a pointer being dereferenced is pointing to allocated memory. E.g.:

  ```
  int *ptr0;
  int element_3 = ptr0[3];
  ```

  The above code will compile and may even run. The value stored in `element_3` is undefined and using it may cause errors difficult to debug:

  - It may cause a generic error, such as `segmentation fault`
  - It may cause an error at a later stage in the program

Programming in C: pointers

Pointers in C

A pointer is a variable that holds a memory address

```
int *ptr0;     /* <-- A pointer to a memory address that will hold an integer variable */
double *ptr1;  /* <-- A pointer to a memory address that will hold a double precision variable */
```

- The C compiler and runtime do not check whether a pointer being dereferenced is pointing to allocated memory. E.g.:

  ```
  int *ptr0;
  int element_3 = ptr0[3];
  ```

  The above code will compile and may even run. The value stored in `element_3` is undefined and using it may cause errors difficult to debug:

  - It may cause a generic error, such as `segmentation fault`
  - It may cause an error at a later stage in the program
- This feature of C contributes to its efficiency while requiring more care taken from the programmer
  - No runtime required to keep count of allocated space, array bounds, etc. every time you access or modify an array
  - Easier to introduce bugs that are especially hard to track

Programming in C: arrays

Arrays in C: Static vs Dynamic allocation

- Statically allocated arrays

```c
char line[256];
double x[10];
```

  - Allocated on the stack
  - Length known at compile time
  - Limited in size

Programming in C: arrays

Arrays in C: Static vs Dynamic allocation

- Statically allocated arrays

```
char line[256];
double x[10];
```

  - Allocated on the stack
  - Length known at compile time
  - Limited in size
- Dynamically allocated arrays

```
char *line;
double *x;
line = malloc(256*sizeof(char));
x = malloc(10*sizeof(double));
...
free(x);
free(line);
```

  - Allocated on the heap
  - Length known at run time (unknown at compile time)
  - Size restricted only by RAM constraints (e.g. maximum available contiguous RAM)

Programming in C: arrays

Arrays in C: Static vs Dynamic allocation

- Statically allocated arrays

```
char line[256];
double x[10];
```

  - Allocated on the stack
  - Length known at compile time
  - Limited in size

- Dynamically allocated arrays

```
char *line;
double *x;
line = malloc(256*sizeof(char));
x = malloc(10*sizeof(double));
...
free(x);
free(line);
```

  - Allocated on the heap
  - Length known at run time (unknown at compile time)
  - Size restricted only by RAM constraints (e.g. maximum available contiguous RAM)

If `malloc()` fails, it will return `NULL`

Programming in C: arrays

Arrays in C: Static vs Dynamic allocation

- Statically allocated arrays

```
char line[256];
double x[10];
```

  - Allocated on the stack
  - Length known at compile time
  - Limited in size

- Dynamically allocated arrays

```
char *line;
double *x;
line = malloc(256*sizeof(char));
x = malloc(10*sizeof(double));
...
free(x);
free(line);
```

  - Allocated on the heap
  - Length known at run time (unknown at compile time)
  - Size restricted only by RAM constraints (e.g. maximum available contiguous RAM)

  If `malloc()` fails, it will return `NULL`

- Another good practice: always wrap `malloc()` in own function that checks return value is not `NULL`

Programming in C: good practices

Arrays in C: Static vs Dynamic allocation

```
/***
 * Allocate memory; print error if NULL is returned
 ***/
void *
ualloc(size_t size)
{
  void *ptr = malloc(size);
  if(ptr == NULL) {
    fprintf(stderr, "malloc() returned null; quitting...\n");
    exit(-2);
  }
  return ptr;
}

 ...

int
main(int argc, char *argv[])
{
    ...
    char *line;
    double *x;
    x = ualloc(10*sizeof(double));
    line = ualloc(256*sizeof(char));
    ...
    free(x);
    free(line);
    return 0;
}
```