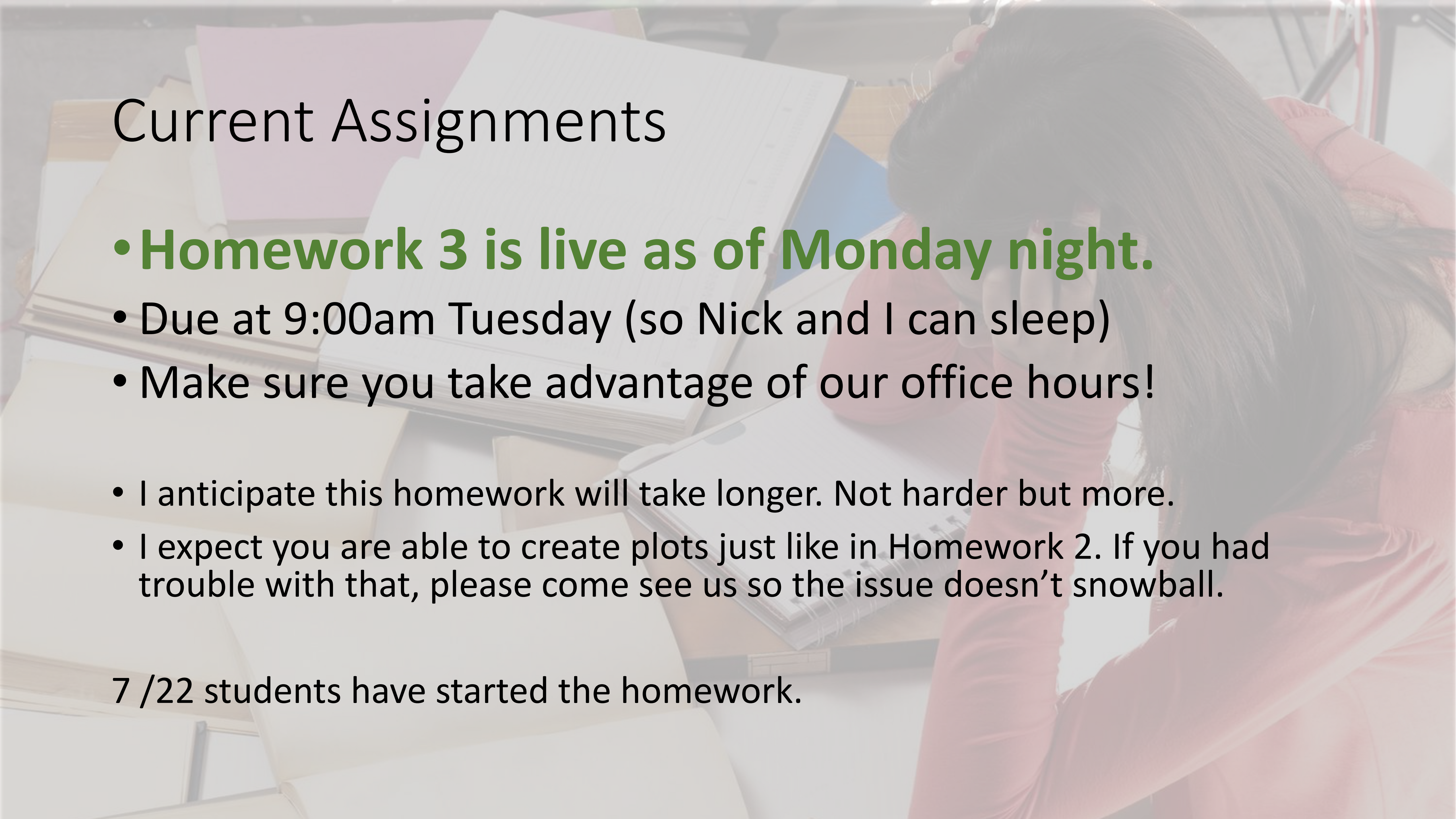
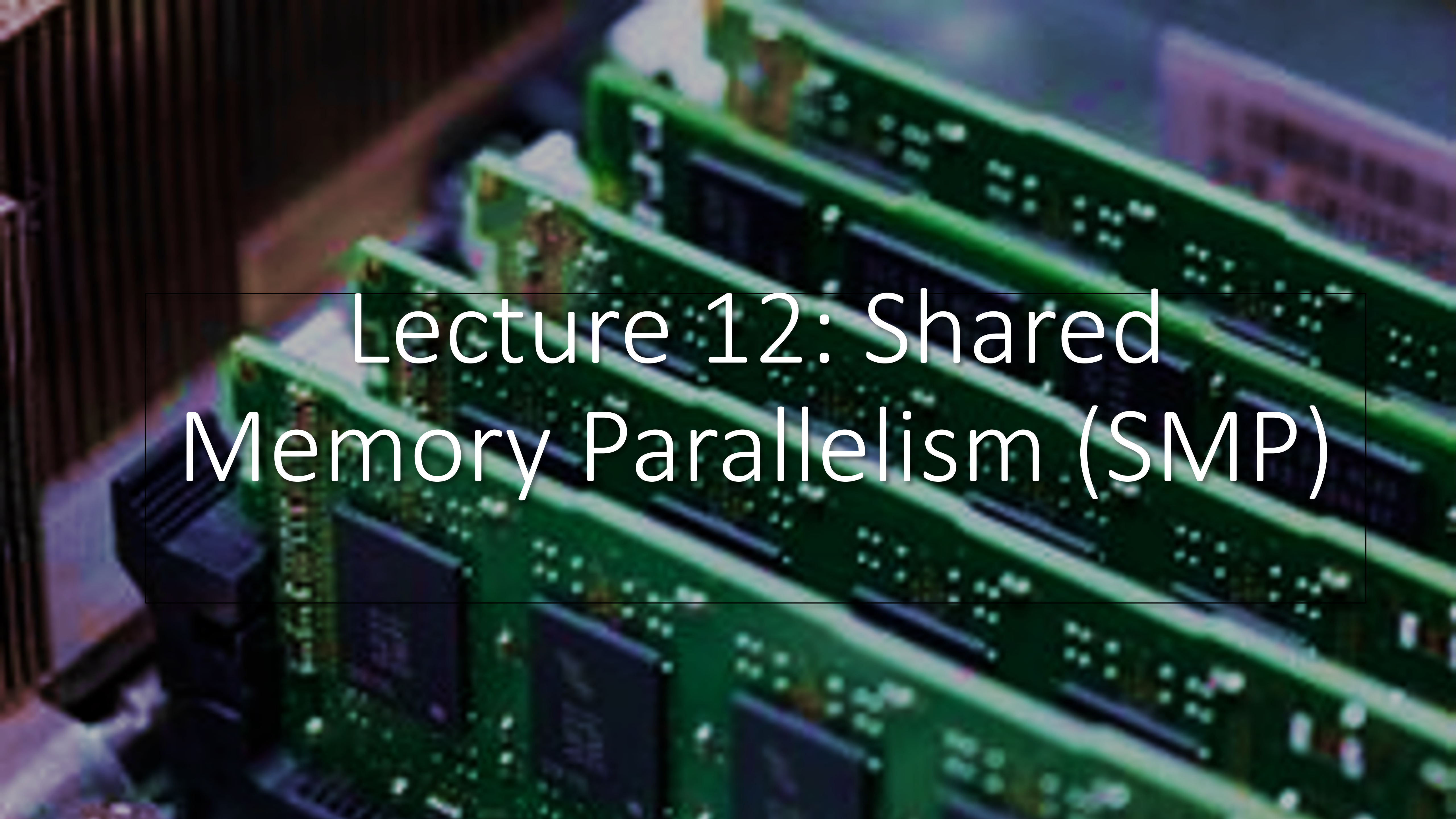


Current Assignments

- **Homework 3 is live as of Monday night.**
- Due at 9:00am Tuesday (so Nick and I can sleep)
- Make sure you take advantage of our office hours!

- I anticipate this homework will take longer. Not harder but more.
- I expect you are able to create plots just like in Homework 2. If you had trouble with that, please come see us so the issue doesn't snowball.

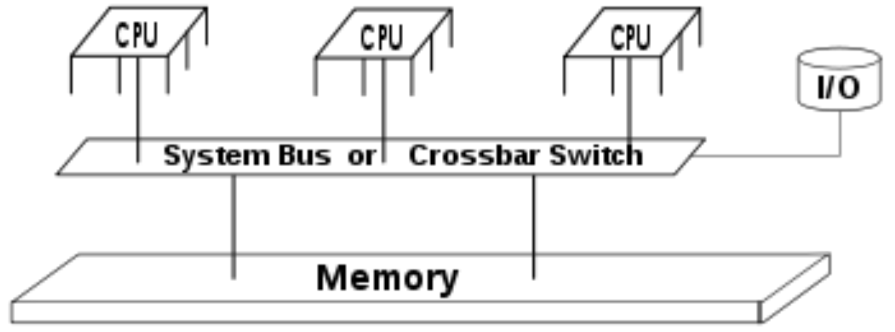
7 /22 students have started the homework.



Lecture 12: Shared Memory Parallelism (SMP)

Two Levels of Parallelism

- Shared-memory architecture : these parallel machines consist of processors which have access to a common memory. (Multiple execution threads in the same memory space)
- Distributed-memory architecture : in these parallel machines each processor has its own private memory and information is interchanged between the processors through messages.
- These approaches are often used together.



Shared Memory Parallelism

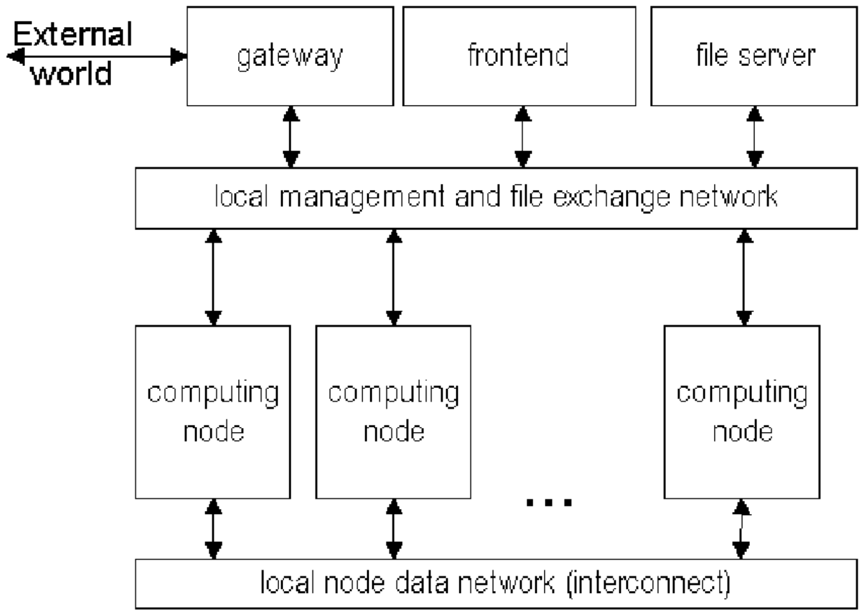
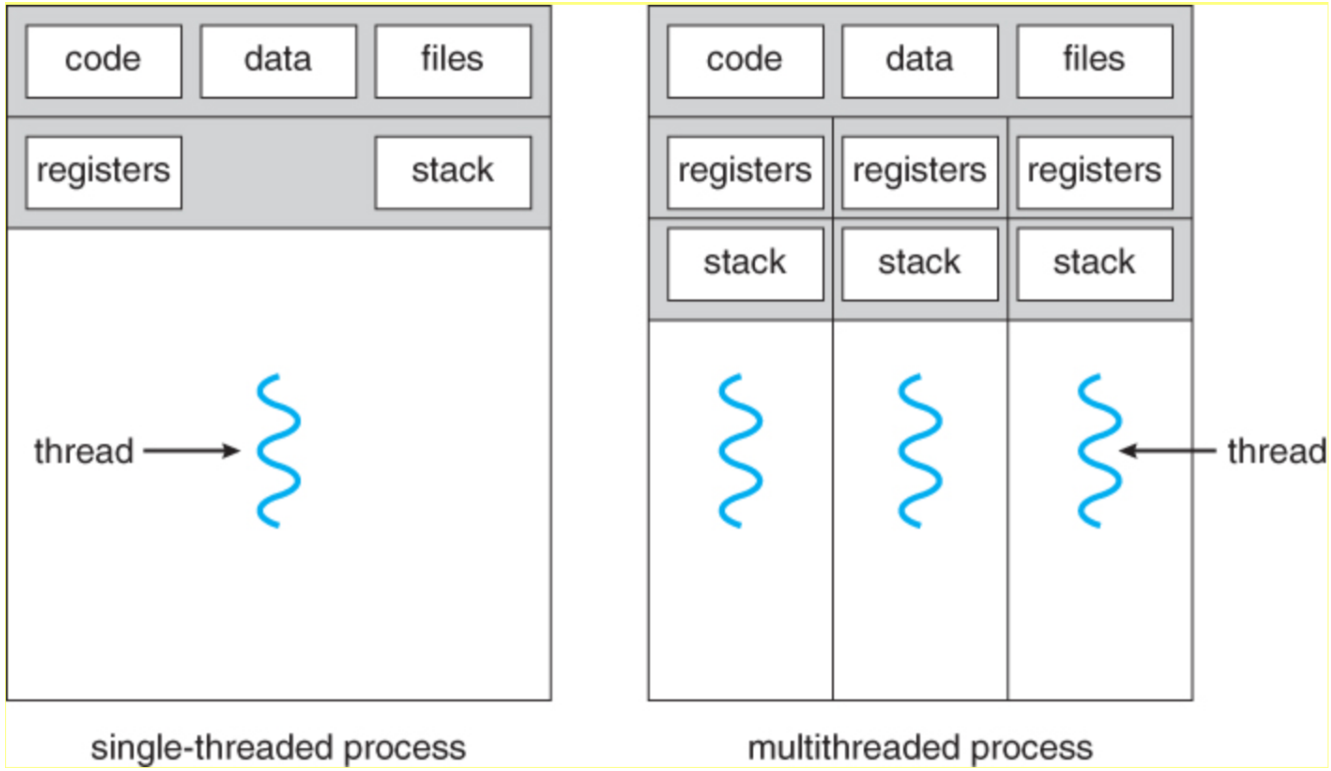


Fig. 1. Simplified cluster structure

Distributed Parallelism

Threads

- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, and a thread ID.
- **Process: processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.**
- In Linux a multi-threaded program has several program counters, stacks, and registers that all have access to the same address space.
- **Remember that the Kernel can allow different processes to access the same address space.**



When you have multiple compute cores you can distribute the threads among them.

Open Multi-Processing Initiative

- In 1996 the Accelerated Strategic Computing Initiative (ASCI)* identified that parallel programming needed a simple programming interface that worked for all computers.
- That's the same year ASCI Red was deployed at Sandia National Labs.
- Mary Zosel at Lawrence Livermore Labs told DEC, IBM, Intel, and the other big companies that the government would only buy systems that supported a common, simple, "lean and mean", SMP API for C and FORTRAN.
- The result was OpenMP.

Mary Zosel, LLNL/ASCI

ASCI Red 1996 ($140 million)

*ASCI is part of the Department of Energy (DOE) – think big national labs (https://www.osti.gov/servlets/purl/1465188)

OpenMP – Example Code

Mary Zosel, LLNL/ASCI

- Guiding Philosophy: Portable and Simple

- Portability was achieved by forcing the big hardware and software vendors to support it.
- Simplicity comes from making OpenMP code as close as possible to the serial version.
- This means the syntax is to wrap existing code with OpenMP pragmas

Helloworld example in C

This is a serial hello world program

Since this is a serial program there is only one thread of execution (np) with ID 0 (id).

```
#include <stdio.h>


int main(int argc, char *argv[])
{



    {
        int id = 0;
        int np = 1;
        printf( "Hello world %d of %d\n", id, np );
    }
return 0;
}
```

From Gropp's CS598 Exascale Course

From Gropp's CS598 Exascale Course

Helloworld example in C

To create multiple threads of execution with OpenMP we just annotate the code.

Here we create 4 threads.

After the **omp parallel** pragma the code is executed by each of the 4 threads in the same memory space.

Variables declared outside the omp parallel pragma are shared by all threads.

Those declared inside are private.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    omp_set_num_threads(4);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf( "Hello world %d of %d\n", id, np );
    }
return 0;
}
```

From Gropp's CS598 Exascale Course

FORTRAN SMP Example with OpenMP

Clone this repository:

https://github.com/gmfricke/FORTRAN_SMP

serial_vecadd.f90

```fortran
! Program to add up elements of two arrays
program serial_array_sum
  USE OMP_LIB
  implicit none

  INTEGER :: N,i
  CHARACTER(LEN=100) :: arg
  REAL, ALLOCATABLE :: a(:), b(:), c(:)

  call GET_COMMAND_ARGUMENT(1, arg)
  read(arg,*) N
  ALLOCATE(a(N))
  ALLOCATE(b(N))
  ALLOCATE(c(N))

  call RANDOM_NUMBER(b)
  call RANDOM_NUMBER(c)

  do i=1,N
     a(i) = b(i) + c(i)
  end do

end program serial_array_sum
```

```fortran
! Program to add up elements of two arrays
program serial_array_sum
  USE OMP_LIB
  implicit none

  INTEGER :: N,i
  CHARACTER(LEN=100) :: arg
  REAL, ALLOCATABLE :: a(:), b(:), c(:)

  call GET_COMMAND_ARGUMENT(1, arg)
  read(arg,*) N
  ALLOCATE(a(N))
  ALLOCATE(b(N))
  ALLOCATE(c(N))

  call RANDOM_NUMBER(b)
  call RANDOM_NUMBER(c)

  do i=1,N
    a(i) = b(i) + c(i)
  end do

end program serial_array_sum
```

```fortran
do i=1,N
  a(i) = b(i) + c(i)
end do
```

smp_vecadd.f90

```fortran
program serial_array_sum
  USE OMP_LIB
  implicit none

! Declare variables, N is the size of the array
  INTEGER :: N,i
  CHARACTER(LEN=100) :: arg
  REAL, ALLOCATABLE :: a(:), b(:), c(:)

  call GET_COMMAND_ARGUMENT(1, arg)
  read(arg,*) N
  ALLOCATE(a(N))
  ALLOCATE(b(N))
  ALLOCATE(c(N))

  call RANDOM_NUMBER(b)
  call RANDOM_NUMBER(c)

  !$OMP PARALLEL PRIVATE(i)
  !$OMP DO
  do i=1,N
    a(i) = b(i) + c(i)
  end do
  !$OMP END DO
  !$OMP END PARALLEL

end program serial_array_sum
```

```fortran
program serial_array_sum
  USE OMP_LIB
  implicit none

! Declare variables, N is the size of the array
  INTEGER :: N,i
  CHARACTER(LEN=100) :: arg
  REAL, ALLOCATABLE :: a(:), b(:), c(:)

  call GET_COMMAND_ARGUMENT(1, arg)
  read(arg,*) N
  ALLOCATE(a(N))
  ALLOCATE(b(N))
  ALLOCATE(c(N))

  call RANDOM_NUMBER(b)
  call RANDOM_NUMBER(c)

  !$OMP PARALLEL PRIVATE(i)
  !$OMP DO
  do i=1,N
    a(i) = b(i) + c(i)
  end do
  !$OMP END DO
  !$OMP END PARALLEL

end program serial_array_sum
```

```fortran
!$OMP PARALLEL PRIVATE(i)
!$OMP DO
do i=1,N
  a(i) = b(i) + c(i)
end do
!$OMP END DO
!$OMP END PARALLEL
```

```fortran
program serial_array_sum
  USE OMP_LIB
  implicit none

! Declare variables, N is the size of the array
  INTEGER :: N,i
  CHARACTER(LEN=100) :: arg
  REAL, ALLOCATABLE :: a(:), b(:), c(:)

  call GET_COMMAND_ARGUMENT(1, arg)
  read(arg,*) N
  ALLOCATE(a(N))
  ALLOCATE(b(N))
  ALLOCATE(c(N))

  call RANDOM_NUMBER(b)
  call RANDOM_NUMBER(c)

  !$OMP PARALLEL PRIVATE(i)
  !$OMP DO
  do i=1,N
    a(i) = b(i) + c(i)
  end do
  !$OMP END DO
  !$OMP END PARALLEL

end program serial_array_sum
```

All variables declared before the OMP Parallel pragma are shared by all threads.

We tell OpenMP that the loop iterator should not be shared.

We need each thread to track which part of the loop it is responsible for – so each thread needs its own private copy of i

```
[matthew@moonshine FORTRAN_SMP]$ gfortran serial_vecadd.f90 -o serial_vecadd

[matthew@moonshine FORTRAN_SMP]$ gfortran -fopenmp smp_vecadd.f90 -o smp_vecadd
```

OpenMP Parameters

- OpenMP parameters are often provided by setting shell environment variables.
- Environment variables are used for all sorts of things in Linux.

- Run the “env” command to see a list of environment variables you have set right now.

```
[matthew@moonshine FORTRAN_SMP]$ env | head -n 10
SHELL=/bin/bash
HISTCONTROL=ignoredups
HISTSIZE=1000
HOSTNAME=moonshine
PWD=/home/matthew/FORTRAN_SMP
LOGNAME=matthew
XDG_SESSION_TYPE=tty
MOTD_SHOWN=pam
HOME=/home/matthew
LANG=en_US.UTF-8
```

```
[matthew@moonshine FORTRAN_SMP]$ env | grep PATH
PATH=/home/matthew/.local/bin:/home/matthew/bin:/usr/local/bin
:/usr/bin:/usr/local/sbin:/usr/sbin
```

Time command

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel.*

```
[matthew@moonshine Fortran_SMP]$ time ./serial_vecadd 1000000000


real 0m10.259s
user 0m7.993s
sys 0m2.238s
[matthew@moonshine Fortran_SMP]$ time OMP_NUM_THREADS=8 ./smp_vecadd 1000000000


real 0m7.460s
user 0m10.906s
sys 0m3.038s
```

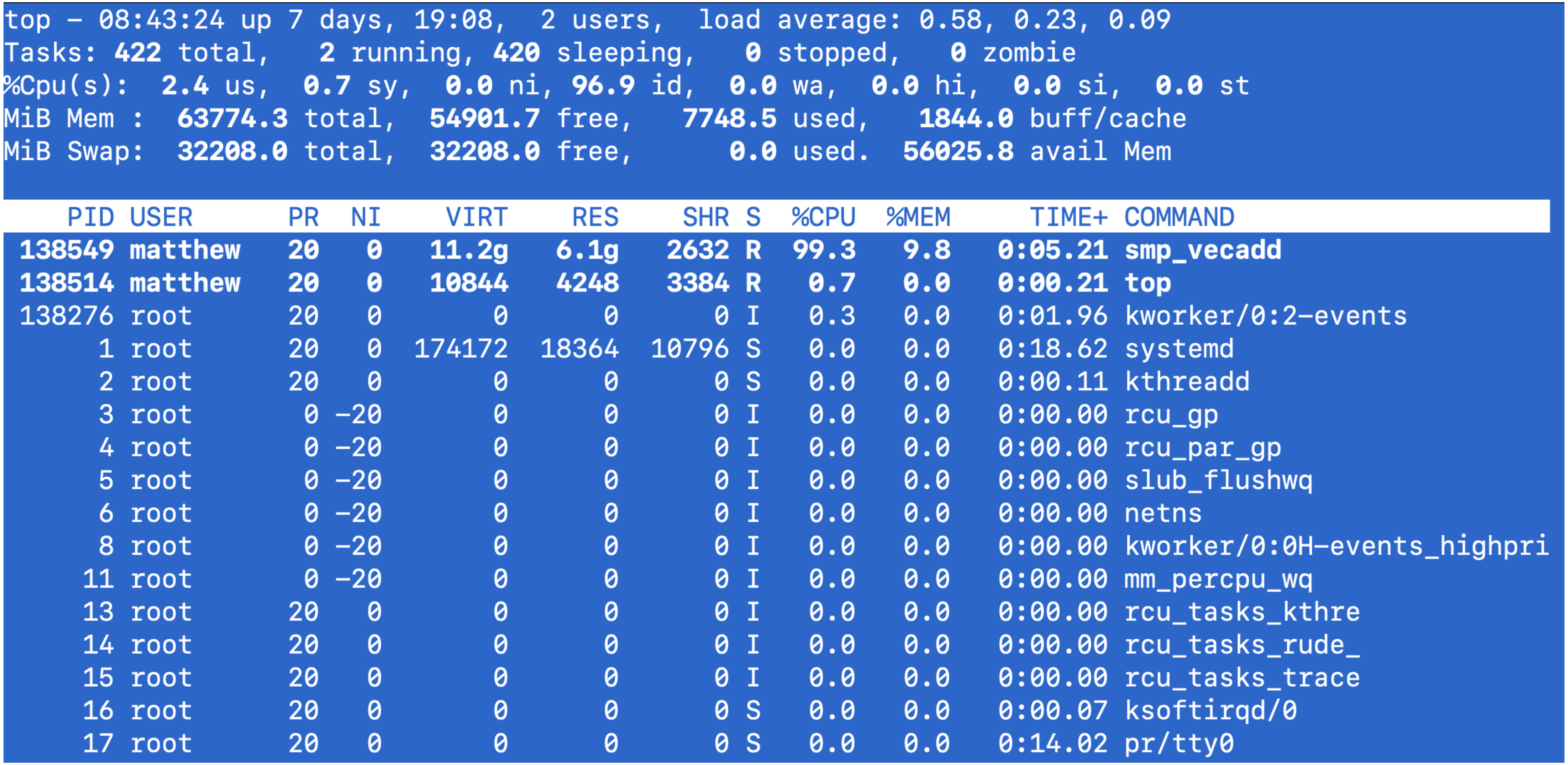
To set an environment variable for the whole shell we can write:

```
export OMP_NUM_THREADS=4
```

To set it for just the process we will execute we can write

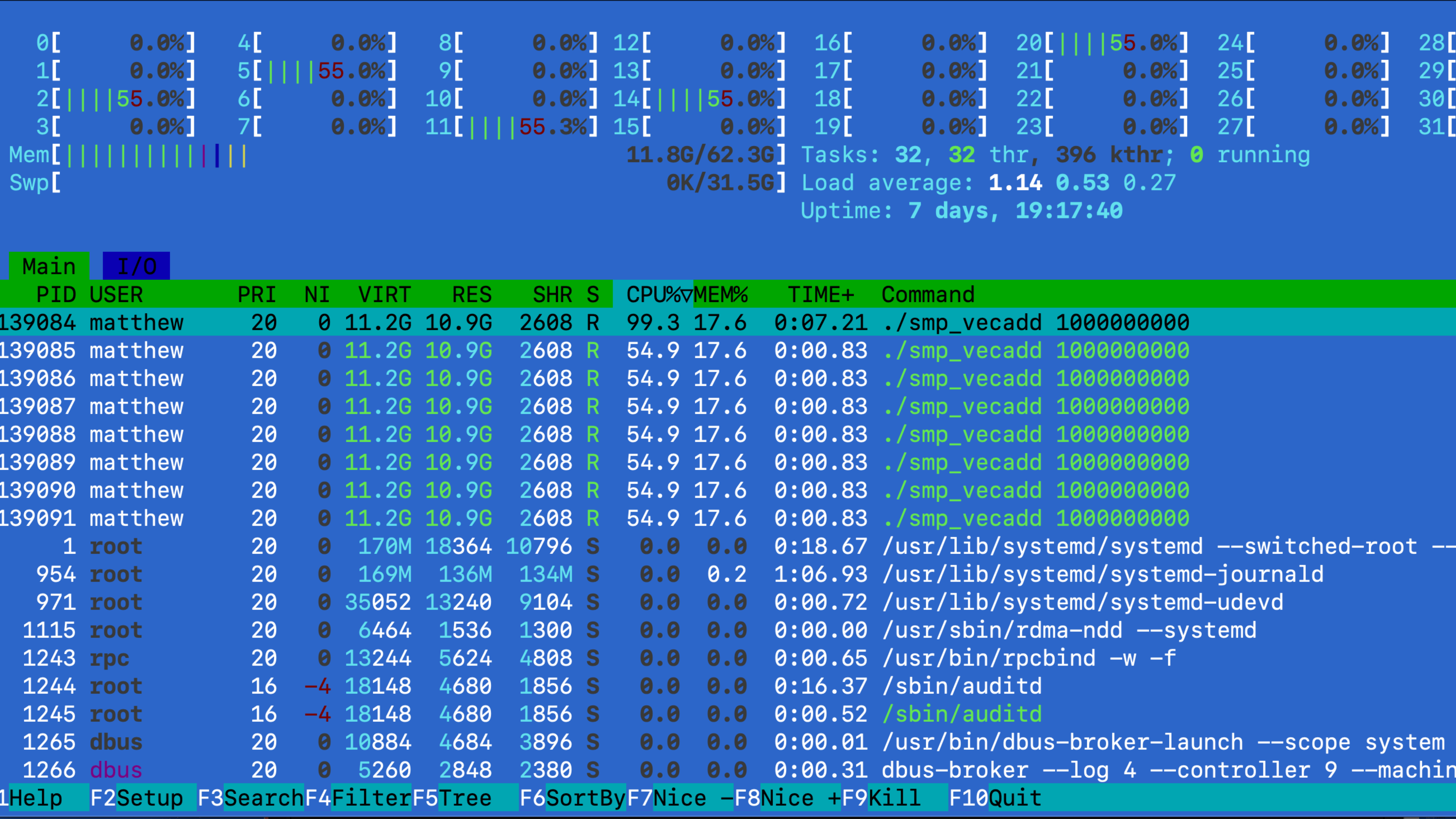
```
OMP_NUM_THREADS=4 ./someprogram
```

Top command

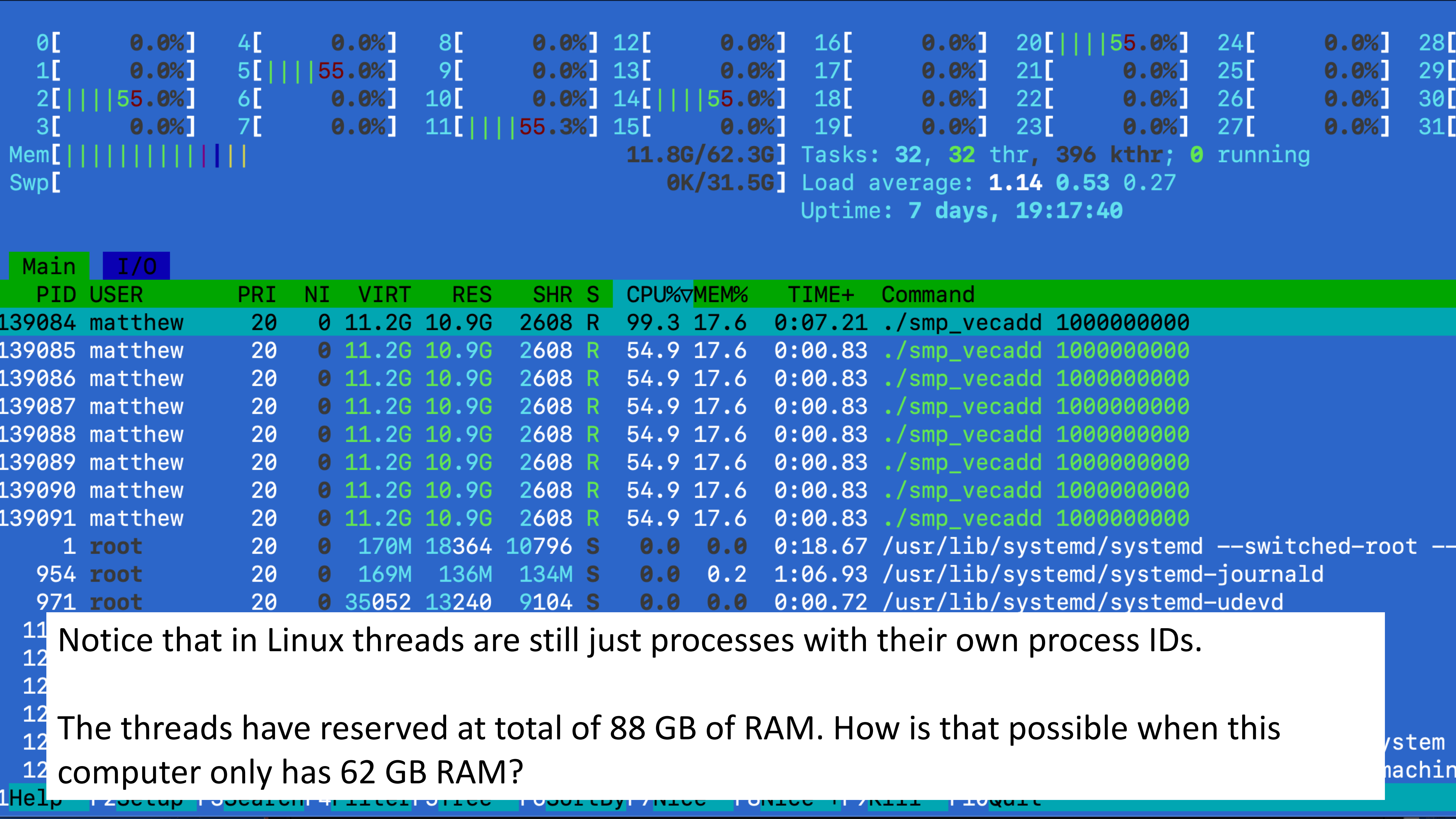


```
top - 08:43:24 up 7 days, 19:08,  2 users,  load average: 0.58, 0.23, 0.09
Tasks: 422 total,   2 running, 420 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.4 us,  0.7 sy,  0.0 ni, 96.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  63774.3 total,  54901.7 free,   7748.5 used,   1844.0 buff/cache
MiB Swap:  32208.0 total,  32208.0 free,      0.0 used.  56025.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 138549 matthew   20   0   11.2g   6.1g   2632 R  99.3   9.8   0:05.21 smp_vecadd
 138514 matthew   20   0   10844   4248   3384 R   0.7   0.0   0:00.21 top
 138276 root      20   0       0      0      0 I   0.3   0.0   0:01.96 kworker/0:2-events
      1 root      20   0  174172  18364  10796 S   0.0   0.0   0:18.62 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.11 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highpri
     11 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_kthre
     14 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_rude_
     15 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_trace
     16 root      20   0       0      0      0 S   0.0   0.0   0:00.07 ksoftirqd/0
     17 root      20   0       0      0      0 S   0.0   0.0   0:14.02 pr/tty0
```

```
[matthew@moonshine]$ sudo yum install epel-release
[matthew@moonshine]$ sudo yum install htop
[sudo] password for matthew:
Last metadata expiration check: 2:01:29 ago on Wed 28 Feb
2024 06:48:20 AM CST.
Dependencies resolved.


Total download size: 2.3 M
Installed size: 3.5 M
Is this ok [y/N]:
```



```
  0[      0.0%]   4[      0.0%]   8[      0.0%] 12[      0.0%]  16[      0.0%]  20[||||55.0%]  24[      0.0%]  28[
  1[      0.0%]   5[||||55.0%]   9[      0.0%] 13[      0.0%]  17[      0.0%]  21[      0.0%]  25[      0.0%]  29[
  2[||||55.0%]   6[      0.0%]  10[      0.0%] 14[||||55.0%]  18[      0.0%]  22[      0.0%]  26[      0.0%]  30[
  3[      0.0%]   7[      0.0%]  11[||||55.3%] 15[      0.0%]  19[      0.0%]  23[      0.0%]  27[      0.0%]  31[
Mem[||||||||||||||                          11.8G/62.3G] Tasks: 32, 32 thr, 396 kthr; 0 running
Swp[                                           0K/31.5G] Load average: 1.14 0.53 0.27
                                                         Uptime: 7 days, 19:17:40

 Main   I/O
  PID USER       PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
139084 matthew     20   0 11.2G 10.9G  2608 R  99.3 17.6  0:07.21 ./smp_vecadd 1000000000
139085 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139086 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139087 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139088 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139089 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139090 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139091 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
     1 root        20   0  170M 18364 10796 S   0.0  0.0  0:18.67 /usr/lib/systemd/systemd --switched-root --
   954 root        20   0  169M  136M  134M S   0.0  0.2  1:06.93 /usr/lib/systemd/systemd-journald
   971 root        20   0 35052 13240  9104 S   0.0  0.0  0:00.72 /usr/lib/systemd/systemd-udevd
  1115 root        20   0  6464  1536  1300 S   0.0  0.0  0:00.00 /usr/sbin/rdma-ndd --systemd
  1243 rpc         20   0 13244  5624  4808 S   0.0  0.0  0:00.65 /usr/bin/rpcbind -w -f
  1244 root        16  -4 18148  4680  1856 S   0.0  0.0  0:16.37 /sbin/auditd
  1245 root        16  -4 18148  4680  1856 S   0.0  0.0  0:00.52 /sbin/auditd
  1265 dbus        20   0 10884  4684  3896 S   0.0  0.0  0:00.01 /usr/bin/dbus-broker-launch --scope system
  1266 dbus        20   0  5260  2848  2380 S   0.0  0.0  0:00.31 dbus-broker --log 4 --controller 9 --machin
1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```



```
  0[      0.0%]   4[      0.0%]   8[      0.0%] 12[      0.0%]  16[      0.0%]  20[||||55.0%]  24[      0.0%]  28[
  1[      0.0%]   5[||||55.0%]   9[      0.0%] 13[      0.0%]  17[      0.0%]  21[      0.0%]  25[      0.0%]  29[
  2[||||55.0%]   6[      0.0%]  10[      0.0%] 14[||||55.0%]  18[      0.0%]  22[      0.0%]  26[      0.0%]  30[
  3[      0.0%]   7[      0.0%]  11[||||55.3%] 15[      0.0%]  19[      0.0%]  23[      0.0%]  27[      0.0%]  31[
 Mem[|||||||||||||                    11.8G/62.3G] Tasks: 32, 32 thr, 396 kthr; 0 running
 Swp[                                    0K/31.5G] Load average: 1.14 0.53 0.27
                                                   Uptime: 7 days, 19:17:40

 Main  I/O
   PID USER       PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
139084 matthew     20   0 11.2G 10.9G  2608 R  99.3 17.6  0:07.21 ./smp_vecadd 1000000000
139085 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139086 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139087 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139088 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139089 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139090 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
139091 matthew     20   0 11.2G 10.9G  2608 R  54.9 17.6  0:00.83 ./smp_vecadd 1000000000
     1 root        20   0  170M 18364 10796 S   0.0  0.0  0:18.67 /usr/lib/systemd/systemd --switched-root --
   954 root        20   0  169M  136M  134M S   0.0  0.2  1:06.93 /usr/lib/systemd/systemd-journald
   971 root        20   0 35052 13240  9104 S   0.0  0.0  0:00.72 /usr/lib/systemd/systemd-udevd
  11
  12
  12
  12
  12                                                                                                    ystem
  12                                                                                                    achin
1Help
```

Notice that in Linux threads are still just processes with their own process IDs.

The threads have reserved at total of 88 GB of RAM. How is that possible when this computer only has 62 GB RAM?

```
[matthew@moonshine Fortran_SMP]$ time ./serial_vecadd 1000000000


real 0m10.259s
user 0m7.993s
sys 0m2.238s
[matthew@moonshine Fortran_SMP]$ time OMP_NUM_THREADS=8 ./smp_vecadd 1000000000


real 0m7.460s
user 0m10.906s
sys 0m3.038s
```

Notice the amount of time on the CPU went up with SMP but the overall time to complete the computation went down.

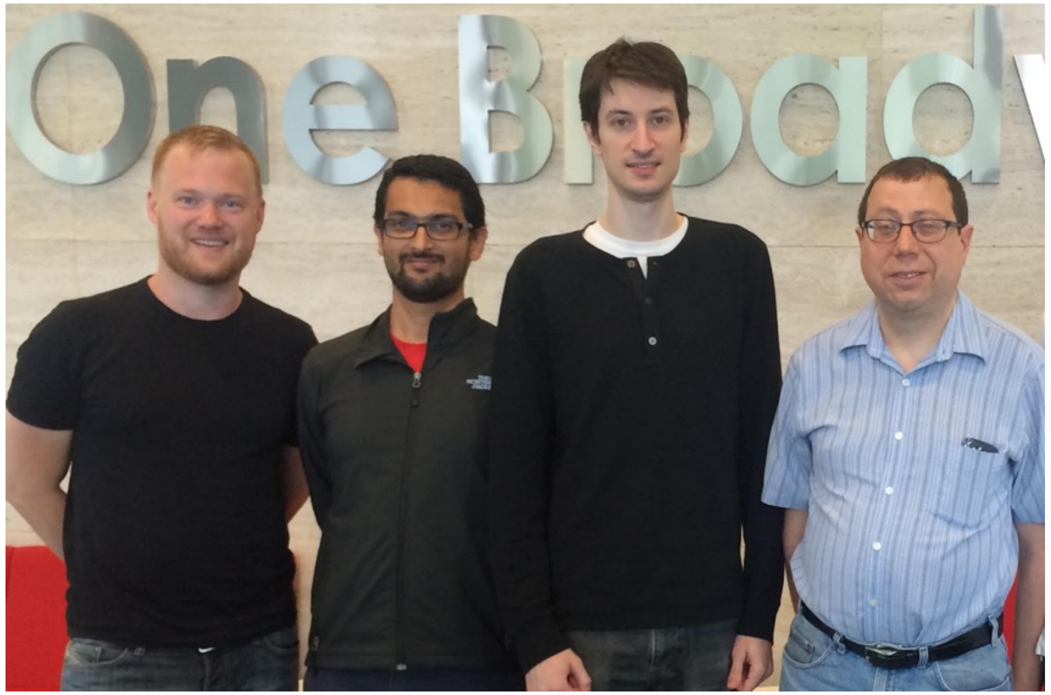
Why?

```
[matthew@moonshine Fortran_SMP]$ time ./serial_vecadd 1000000000

real 0m3.838s
user 0m3.123s
sys 0m0.705s

[matthew@moonshine Fortran_SMP]$ time OMP_NUM_THREADS=4 ./smp_vecadd 1000000000

real 0m1.616s
user 0m5.618s
sys 0m0.677s

[matthew@moonshine Fortran_SMP]$ time OMP_NUM_THREADS=8 ./smp_vecadd 1000000000

real 0m0.840s
user 0m5.805s
sys 0m0.694s
```

If we don’t include the random number generation…

julia

- Julia is designed to be an interpreted, functional, **high performance computing** language.
- Interpreted languages tend to be slow. (Yes really!)
- Julia is written to be interpreted but just as fast as compiled languages like C and FORTRAN.
- It's the **J** in **J**upyter



Julia started in 2009 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman (mostly Harvard and MIT).

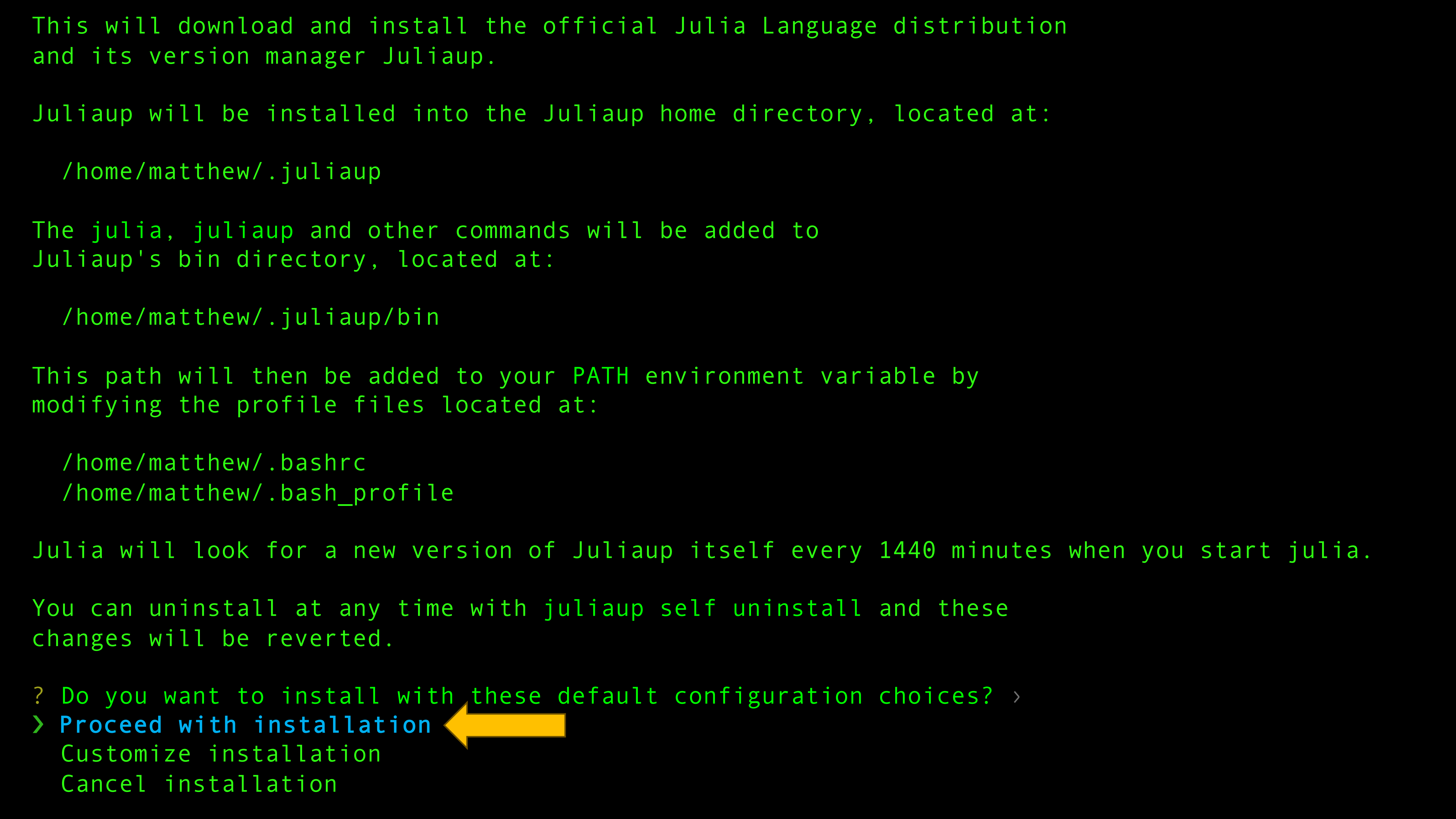
It is a young language with version 1.0 coming out in 2018. Even though it is "stable" features still appear and disappear at an alarming rate, but seemed to have settled down a bit after versions 1.5.
BUT there is a bug in version 1.10.1 we will have to work around :(

So let’s install Julia

```
[matthew@moonshine ~]$ curl -fsSL https://install.julialang.org | sh


info: downloading installer
Welcome to Julia!
```

You already installed FORTRAN and C with YUM.

We can download Julia directly from their website with curl (client URL). We pipe the downloaded data to the sh shell which executes the installation shell script.



```
This will download and install the official Julia Language distribution
and its version manager Juliaup.

Juliaup will be installed into the Juliaup home directory, located at:

  /home/matthew/.juliaup

The julia, juliaup and other commands will be added to
Juliaup's bin directory, located at:

  /home/matthew/.juliaup/bin

This path will then be added to your PATH environment variable by
modifying the profile files located at:

  /home/matthew/.bashrc
  /home/matthew/.bash_profile

Julia will look for a new version of Juliaup itself every 1440 minutes when you start julia.

You can uninstall at any time with juliaup self uninstall and these
changes will be reverted.

? Do you want to install with these default configuration choices? ›
❯ Proceed with installation
  Customize installation
  Cancel installation
```

```
✓ Do you want to install with these default configuration choices? ·
Proceed with installation

Now installing Juliaup
Installing Julia 1.10.1+0.x64.linux.gnu
Configured the default Julia version to be 'release'.
Julia was successfully installed on your system.

Depending on which shell you are using, run one of the following
commands to reload the PATH environment variable:

  . /home/matthew/.bashrc
  . /home/matthew/.bash_profile

[matthew@moonshine ~]$
```

The PATH environment variable defines where the shell looks for programs

```
[matthew@moonshine ~]$ echo $PATH
/home/matthew/.local/bin:/home/matthew/bin:/usr/local/bin:/usr/bin:/usr/loca
l/sbin:/usr/sbin
```

Colon separated list of paths to search.

The PATH environment variable defines where the shell looks for programs.
Let’s print the current value of PATH.

```
Now installing Juliaup
Installing Julia 1.10.1+0.x64.linux.gnu
Configured the default Julia version to be 'release'.
Julia was successfully installed on your system.

Depending on which shell you are using, run one of the following
commands to reload the PATH environment variable:

  . /home/matthew/.bashrc
  . /home/matthew/.bash_profile

[matthew@moonshine ~]$
```

The .bashrc and .profile bash scripts are executed when you login (usually).

```
[matthew@moonshine ~]$ cat .bashrc
# >>> juliaup initialize >>>

# !! Contents within this block are managed by juliaup !!

case ":$PATH:" in
    *:/home/matthew/.juliaup/bin:*)
        ;;
    *)
        export PATH=/home/matthew/.juliaup/bin${PATH:+:${PATH}}
        ;;
esac

# <<< juliaup initialize <<<
```

Concatenated the Julia path to front of the existing PATH.

Since the Julia software isn't installed in /bin or any of the other usual places it modified your .bashrc so that the path that includes Julia is added to your PATH variable every time you login.

```
[matthew@moonshine ~]$ source .bashrc
```

```
"Depending on which shell you are using*, run one of the following
commands to reload the PATH environment variable:

. /home/matthew/.bashrc
. /home/matthew/.bash_profile"
```

You can execute the .bashrc in several ways to make sure you can find Julia.

1) Log out and back in
2) Run the .bashrc script manually with source .bashrc ("." is shorthand for source)

*This is badly worded – these scripts are both specific to the BASH shell.

```
[matthew@moonshine ~]$ source .bashrc
[matthew@moonshine ~]$ echo $PATH
/home/matthew/.juliaup/bin:/home/matthew/.local/bin:/home/matthe
w/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
[matthew@moonshine ~]$
```

Running the script adds the Julia location to the path.

Now we can run Julia ☺

Julia Benchmark Tools

- Since Julia is aimed at HPC is comes with some really nice benchmarking tools.
- We will use them to measure how much multithreading speeds up our computations.
- But before we can install Julia packages we have to deal with issue 533339: https://github.com/JuliaLang/julia/issues/53339

Certificate Bug Workaround
(Github Julia Issue 53339)

I’m keeping this in the slides because it is the kind of **DevOps** HPC engineers have to worry about all the time.

Julia version 1.10.1 looks for an encryption certificate using the wrong path.
We have to give it the right path by creating a startup.jl file and setting a variable.

```
$ mkdir -p .julia/config/
$ echo 'ENV["JULIA_SSL_CA_ROOTS_PATH"]="/etc/ssl/certs/ca-bundle.crt"' >\
> .julia/config/startup.jl
```

The “\” means continue the command on the next line. “>” Sends the output into a file. Notice the nested quotes ' ” ” '!
Careful if you copy and past the code above – the quote symbols often get garbled.

Certificate Bug Workaround
(Github Julia Issue 53339)

I’m keeping this in class because it is the kind of **DevOps** HPC engineers have to worry about all the time.

Julia version 1.10.1 looks for an encryption certificate using the wrong path.
We have to give it the right path by creating a startup.jl file and setting a variable.

```
$ mkdir -p .julia/config/
$ echo 'ENV["JULIA_SSL_CA_ROOTS_PATH"]="/etc/ssl/certs/ca-bundle.crt"' >\
> .julia/config/startup.jl

$ cat .julia/config/startup.jl
ENV["JULIA_SSL_CA_ROOTS_PATH"]=“/etc/ssl/certs/ca-bundle.crt"
```

Check the file was created and has the right contents with “cat”

```
[matthew@moonshine ~]$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.1 (2024-02-13)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> 1 + 1
2
```

This sis the Julia REPL* (Read, Evaluate, Print, and Loop). Interpreted languages have REPLs where you enter code – or they can read from file.

Now we can run Julia ☺

… and exit with exit()

*I kind of hate the term REPL – so I'll just call it the interpreter.

```
julia> import Pkg;

julia> Pkg.add("BenchmarkTools")
    Updating registry at `~/.julia/registries/General.toml`
   Resolving package versions...
   Installed BenchmarkTools ── v1.5.0
<snip>
Precompiling project...
  10 dependencies successfully precompiled in 21 seconds.

julia> using BenchmarkTools
```

We import the Julia package manager, add Benchmark tools, and use the package.

```
julia> @benchmark sort(data) setup=(data=rand(10)) samples=1000
```

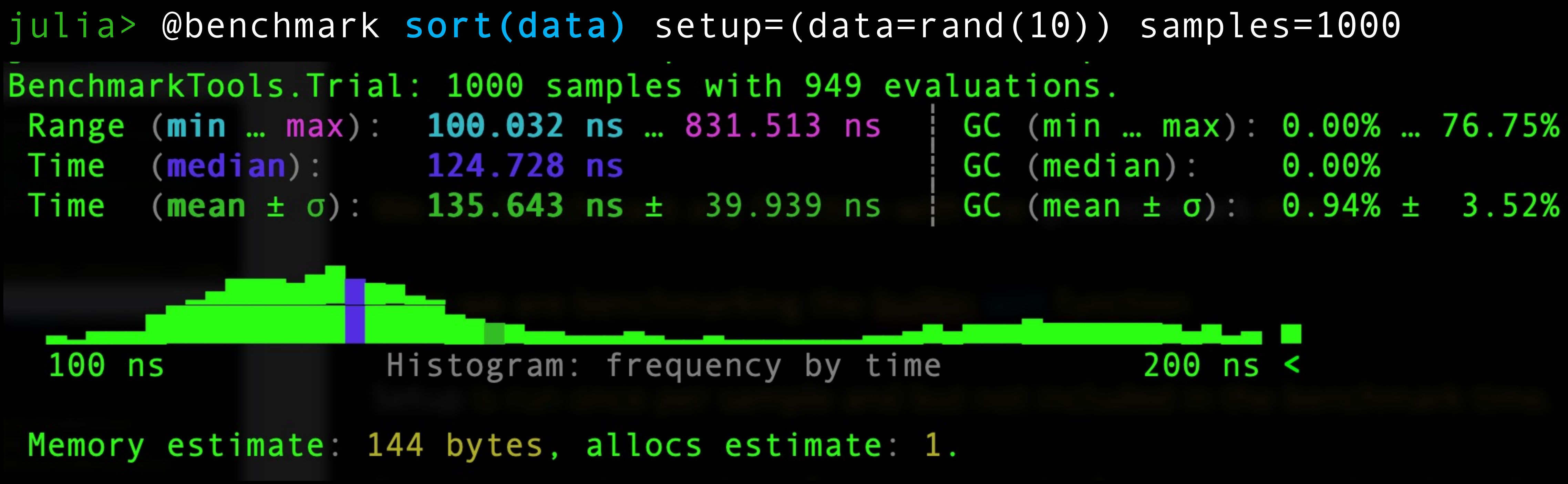
```
julia> data=rand(10)
10-element Vector{Float64}:
 0.34728822898083356
 0.9389609594353292
 0.6127950927838776
 0.18733946403131663
 0.9262208042266743
 0.7009554495876372
 0.32478731787184756
 0.615681279938658
 0.4231459280329968
 0.27727842077336873
```

We can benchmark any function with the @benchmark macro

Here we are benchmarking the builtin sort function

Setup is run once per sample and but not included in the benchmark time.

Samples sets the number of times to run the function we are benchmarking.

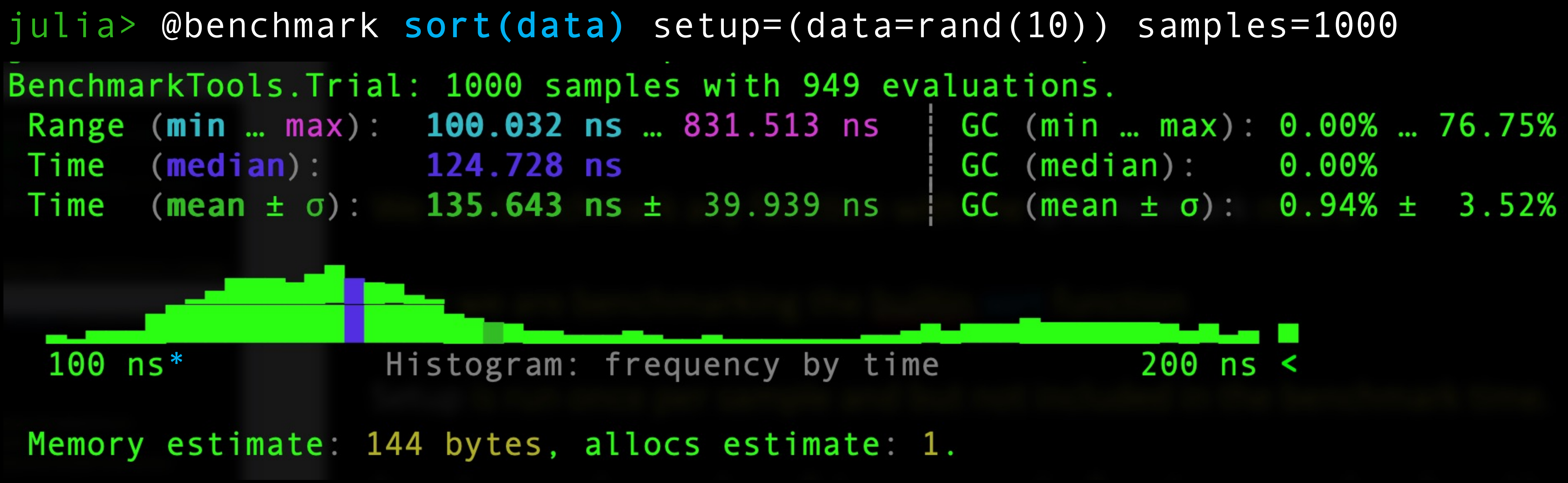


```
julia> @benchmark sort(data) setup=(data=rand(10)) samples=1000
BenchmarkTools.Trial: 1000 samples with 949 evaluations.
 Range (min … max):  100.032 ns … 831.513 ns  ┊ GC (min … max): 0.00% … 76.75%
 Time  (median):     124.728 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   135.643 ns ±  39.939 ns  ┊ GC (mean ± σ):  0.94% ±  3.52%

  100 ns          Histogram: frequency by time           200 ns <

 Memory estimate: 144 bytes, allocs estimate: 1.
```

Now we have a distribution of times over 1000 runs!

High Performance Benchmarking is an EXPERIMENTAL science (inductive reasoning) as opposed to formal algorithmic analysis which is deductive reasoning (proofs).

HPC benchmarking’s language is statistics.



```
julia> @benchmark sort(data) setup=(data=rand(10)) samples=1000
BenchmarkTools.Trial: 1000 samples with 949 evaluations.
 Range (min … max):  100.032 ns … 831.513 ns  ┊ GC (min … max): 0.00% … 76.75%
 Time  (median):     124.728 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   135.643 ns ±  39.939 ns  ┊ GC (mean ± σ):  0.94% ±  3.52%

  100 ns*           Histogram: frequency by time          200 ns <

 Memory estimate: 144 bytes, allocs estimate: 1.
```

HPC Benchmarking is an experimental science.

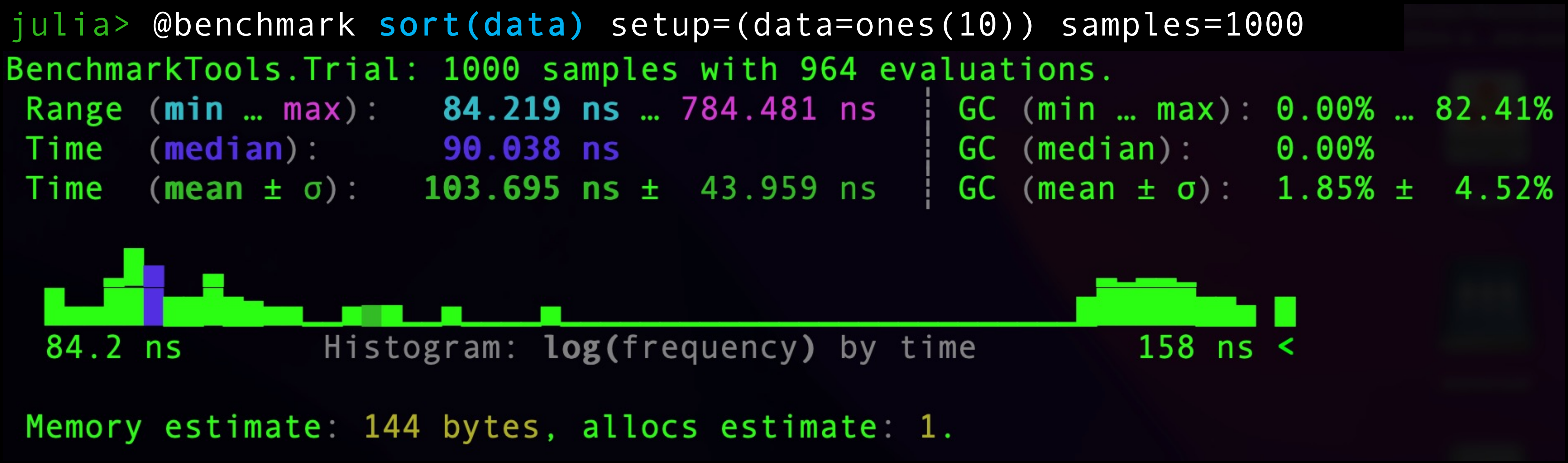
If computers are deterministic, why would we get a range of times for our sorting function?

*A nanosecond (ns) is one-billionth (10^{-9}) of a second. A 1Ghz CPU executes one cycle per nanosecond.

```
julia> @benchmark sort(data) setup=(data=ones(10)) samples=1000
```

```
julia> ones(10)
10-element Vector{Float64}:
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
```

Let's remove the randomness from the input.



```
julia> @benchmark sort(data) setup=(data=ones(10)) samples=1000
BenchmarkTools.Trial: 1000 samples with 964 evaluations.
 Range (min … max):    84.219 ns … 784.481 ns  ┊ GC (min … max): 0.00% … 82.41%
 Time  (median):       90.038 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   103.695 ns ±  43.959 ns  ┊ GC (mean ± σ):  1.85% ±  4.52%

  84.2 ns       Histogram: log(frequency) by time       158 ns <

 Memory estimate: 144 bytes, allocs estimate: 1.
```

We still get variation. Can you think of some reasons why from the previous lectures?

Git clone Julia looping code

- Clone https://github.com/gmfricke/Julia_SMP.git

Multithreading

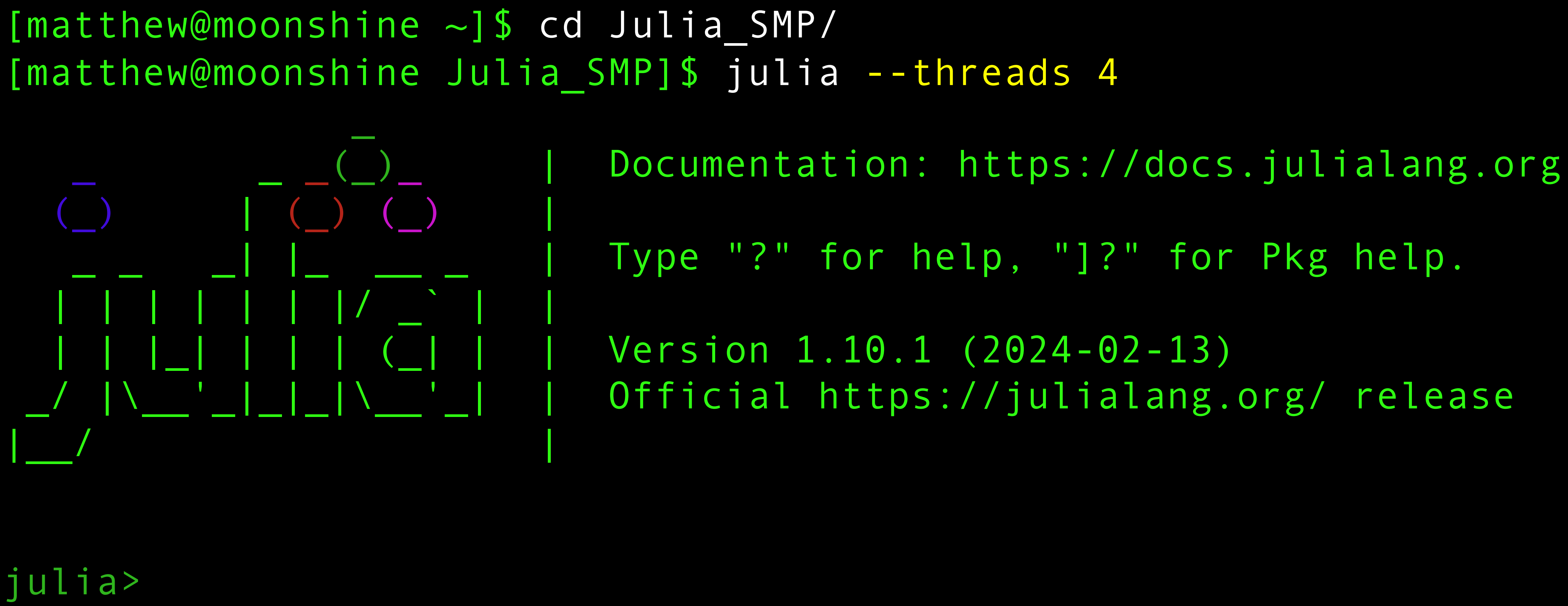
- We just saw how Shared Memory Task Parallelism works with OpenMP.
- Julia supports the same thing with its own mechanism (Julia abandoned OpenMP)
- The paradigm is the same as the Fork and Join principle in OpenMP.

In looping.jl:

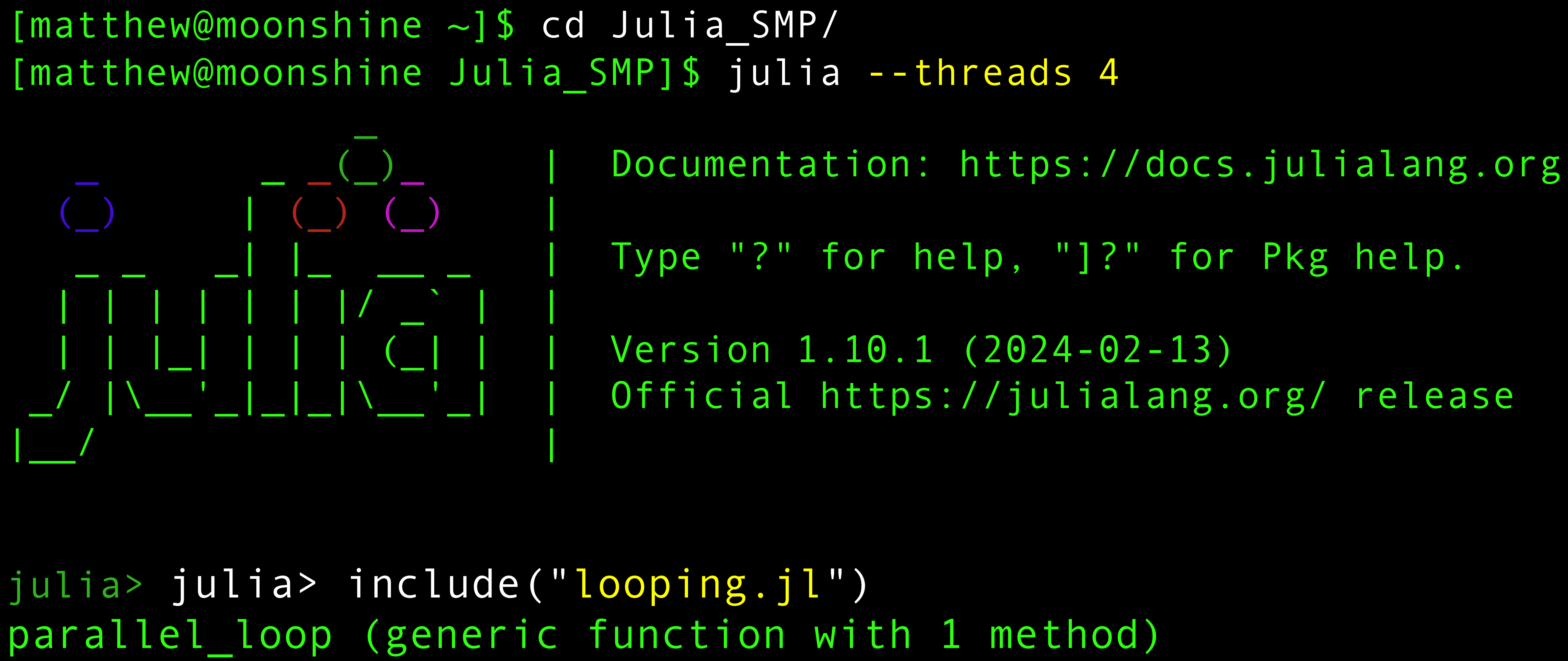
```
# Sum the elements of a
function serial_loop( a )
  total = 0

  for x in a
    total += x  # Compute running sum
  end

  return total
end
```

In looping.jl:

```julia
# Sum the elements of an array in parallel
using Base.Threads

function parallel_loop( a )
  p = zeros(nthreads()) # Somewhere to store the partial sums

  # The @threads macro does the work for us. Dividing the loop evenly
  # between the threads
  @threads for x in a
    p[threadid()] += x  # Each thread computes a partial sum
  end

  total = sum(p) # Add up the partial sums from each thread.
  return total
end
```

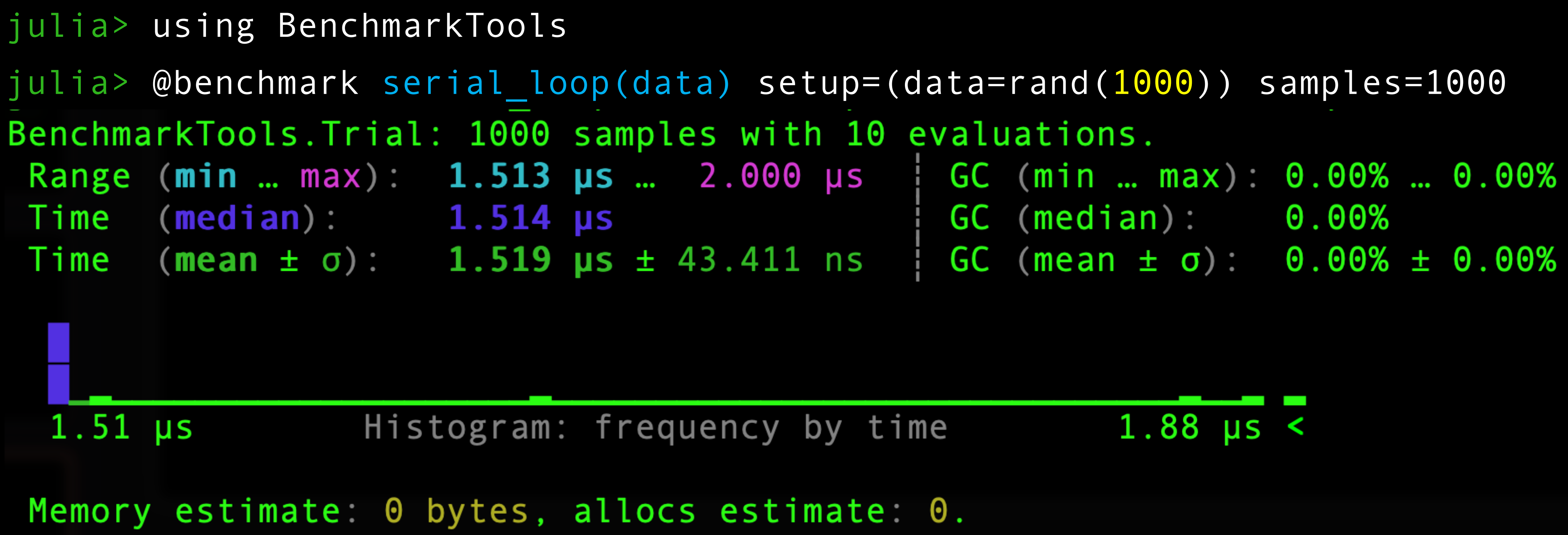


```
[matthew@moonshine ~]$ cd Julia_SMP/
[matthew@moonshine Julia_SMP]$ julia --threads 4
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.1 (2024-02-13)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia>
```

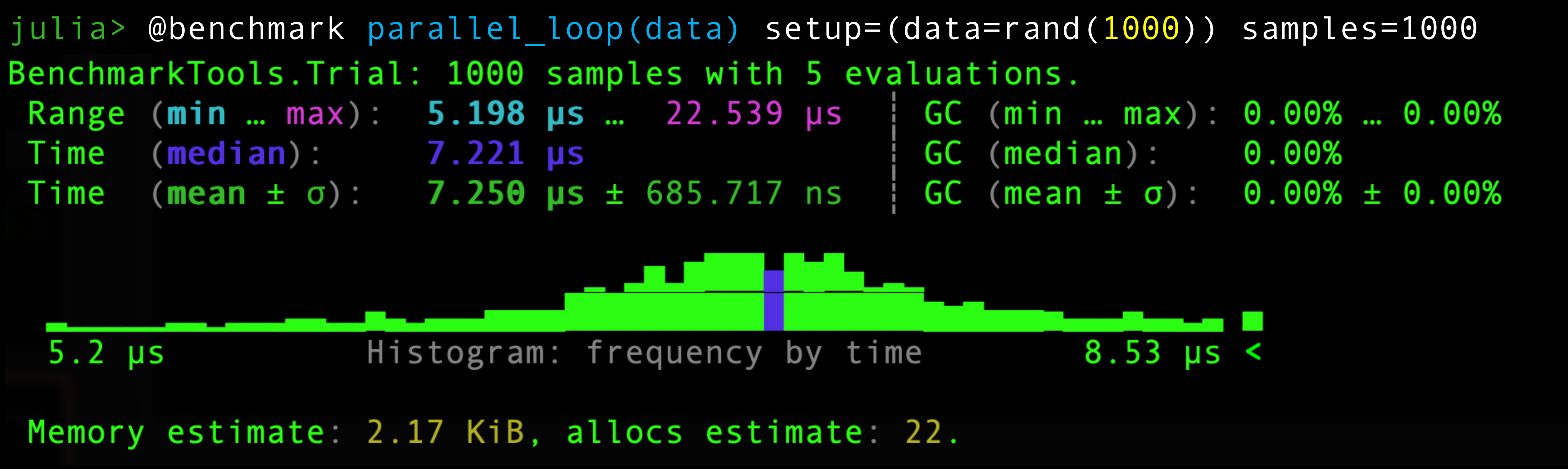


```
[matthew@moonshine ~]$ cd Julia_SMP/
[matthew@moonshine Julia_SMP]$ julia --threads 4
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.1 (2024-02-13)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia> julia> include("looping.jl")
parallel_loop (generic function with 1 method)
```

```
julia> julia> include("looping.jl")
parallel_loop (generic function with 1 method)

julia> serial_loop(rand(100))
51.361311058281835


julia> parallel_loop(rand(100))
48.56971707948273


julia>
```



```
julia> using BenchmarkTools

julia> @benchmark serial_loop(data) setup=(data=rand(1000)) samples=1000
BenchmarkTools.Trial: 1000 samples with 10 evaluations.
 Range (min … max):  1.513 μs …  2.000 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.514 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.519 μs ± 43.411 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  █
  █▁▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▂▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▂▁▁▂ ▂
  1.51 μs          Histogram: frequency by time        1.88 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```



```
julia> @benchmark parallel_loop(data) setup=(data=rand(1000)) samples=1000
BenchmarkTools.Trial: 1000 samples with 5 evaluations.
 Range (min … max):   5.198 μs …  22.539 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):      7.221 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):    7.250 μs ± 685.717 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  5.2 μs           Histogram: frequency by time          8.53 μs <

 Memory estimate: 2.17 KiB, allocs estimate: 22.
```

SMP is slower! Find the size of array that makes SMP worthwhile…

10^{-9} seconds	1 nanosecond	ns	1 CPU cycle on 1 GHz processor
10^{-6}	1 microsecond	μs	1000 CPU cycles
10^{-3}	1 millisecond	ms	1,000,000 cycles