PAP laboratory assignments
Lab 1: OpenMP parallelisation of Eratosthenes Sieve

E. Ayguadé

Spring 2019-20

Contents

1

Experimental setup

The objective of this first laboratory assignment is to remember the environment that we used in PAR and that will be used again during this semester to do the PAP laboratory assignments. In order to work on the multiprocessor server you will have to establish a connection using secure shell: `"ssh -X papXX@boada.ac.upc.edu"`, being `XX` the user number assigned to you. Remember that option `-X`[1] is necessary in order to forward the X11 and be able to open remote windows in your local terminal. You can change the password for your account using `"ssh -t papXX@boada.ac.upc.edu passwd"`.

Once you are logged in you will find yourself in `boada-1`, the login node for the whole machine from where you can submit execution jobs to the rest of the nodes in the machine. In fact `boada` is composed of 8 nodes (named `boada-1` to `boada-8`), equipped with three different processor generations, as shown in the following table:

Node name	**Sockets (number and type)**	**Cores** per socket	**Interactive**	**Queue**
boada-1	2 x Intel Xeon E5645	6	Yes	batch
boada-2 to 4	2 x Intel Xeon E5645	6	No	execution
boada-5	2 x Intel Xeon E5-2620 v2 + 4 x Nvidia K40c	6	No	cuda
boada-6 to 8	2 x Intel Xeon E5-2609 v4	8	No	execution2

Each node in `boada` has its own local disk; you can access it through `/scratch/1/papXX`. In addition, all nodes have access to a shared NAS (*Network-attached Storage*) disk; you can access it through `/scratch/nas/1/papXX` and this is your *home directory* (check by typing `pwd` in the command line). We will post all necessary files to do each laboratory assignment in `/scratch/nas/1/pap0/sessions`. For the session today, copy `lab1.tar.gz` **from** that location **to** your home directory in `boada` and uncompress it at the root of your home directory with this command line: `"tar -zxvf lab1.tar.gz"`.

In order to set up all environment variables you have to process the `environment.bash` file now available in your home directory with `"source environment.bash"`. **Note:** since you have to do this every time you login in the account or open a new console window, we recommend that you add this command line in the `.profile` file in your home directory, a file that is executed every time a new session is initiated.

There are two ways to execute your programs in `boada`: 1) via a queueing system or 2) interactively. We strongly suggest to use option 1) when you want to ensure that the execution is done in isolation inside a single node of the machine. Using option 2) your execution will share resources with other programs and interactive jobs, not ensuring representative timing results. To queue a job for execution you need to type `"qsub -l queue_name submit-xxxx.sh"`, being `submit-xxxx.sh` the name of the script to submit and `queue_name` one of the queues listed above. Use `"qstat"` to ask the system about the status of your job submission. You can use `"qdel"` to remove a job from the queueing system. Additional parameters may be specified after the script name.

[1] Option -Y if you are connecting from Mac OS X using XQuartz.

1.1 The program

In this session you will parallelise a program that finds (and counts) all prime numbers up to a certain given number. The sequential program implements the sieve of Eratosthenes (there are a number of prime number sieves: Euler's sieve, sieve of Sundaram, sieve of Atkin, ...), a simple and ancient algorithm attributed to the Greek mathematician Eratosthenes of Cyrene.

Look at the `"int eratosthenes(int lastNumber)"` function inside `sieve1.c` to understand how the sieve process works:

1. Create a list `isPrime` of consecutive integers from 2 through `lastNumber`, all of them initially considered as potential prime numbers.
2. Initially, let `i` equal 2, the first prime number.
3. Starting from `i` $\times$ `i`, enumerate its multiples by counting to `lastNumber` in increments of `i`, and mark them in `isPrime`.
4. Find the first number greater than `i` in `isPrime` that is not marked. If there was no such number, then go to step 5. Otherwise, let `i` now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, all the numbers in `isPrime` that are not marked are prime. Function `eratosthenes` counts the number of primes found and returns this number as its result.

In the following steps you will compile using the `Makefile` and interactively execute the sequential version of the Sieve of Eratosthenes.

1. Open the `Makefile` and try to understand how the target to do the sequential compilation is expressed. Then type `"make sieve1-seq"` in the command line to compile the sequential version.
2. Interactively execute and time the `sieve1-seq` binary to compute the number of prime numbers up to a certain number (specified as argument). We will use 100.000.000 as the reference number for all the executions in this laboratory assignment. So just type `"/usr/bin/time ./sieve1-seq 100000000"` to execute it. The execution returns the user and system CPU time, the elapsed time, and the % of CPU used (the output of the GNU `/usr/bin/time` command[2]) and the number of primes found.

[2]The command returns additional information which may be useful to explain certain performance improvements/degradations in your program transformations.

2

OpenMP parallelisation

The algorithm is well suited for an iterative task decomposition, although a data decomposition could also be explored in order to improve memory locality. Analyse the dependences in the algorithm and insert the appropriate `OpenMP` pragmas in `sieve1.c` (which is already prepared for `OpenMP` using conditional compilation `#ifdef _OPENMP`). Always make sure that your parallel code leads to a correct code.

1. Write at least two versions of the parallel `sieve1`: one using the `for` work–sharing construct; and another using the `taskloop`[1] construct. Make sure to use the appropriate synchronisation constructs and/or clauses to avoid data races.

2. In the `Makefile` you have the target to compile your `OpenMP` parallel version. Type `"make sieve1-omp"` in the command line to compile it and generate an executable file. We provide you with scripts to interactively run (`run-omp.sh`), queue (`submit-omp.sh`) and do the strong scalability analysis (`submit-strong-omp.sh`). Each script has a number of arguments that you can see from the script itself.

3. Find the most appropriate granularity for the work assigned to threads (or tasks) in order to find the best parallelisation option (e.g. play with different `schedule` types and `chunk` sizes, number of tasks and their granularity for `taskloop`).

4. We also provide you scripts to interactively run (`run-extrae.sh`) or queue (`submit-extrae.sh`) the instrumented execution with *Extrae* that will generate a trace that you can visualise with *Paraver*. Remember from PAR the existence of *Paraver* configuration files (`.cfg` files available in the `cfgs` directory linked in your home directory) to visualise/analyse different aspects of your parallel program. We strongly suggest to do this analysis with *Paraver* in order to understand the parallelisation behaviour. Table 2.1 summarises the most useful ones.

Note: You can submit your jobs to the `execution` and `execution2` queues and observe and analyse differences in execution time and scalability.

[1] Remember that in the current specification of OpenMP there is no support for reductions in `task` and `taskloop` constructs, so you will have to manually implement them if needed.

***Paraver* cfg file**	**Timeline showing ...**
OMP_parallel_constructs	when a `parallel` construct is executed
OMP_parallel_functions	the function each thread executes in a parallel region
OMP_parallel_functions_duration	the duration for the function executed in a parallel region
OMP_worksharing_constructs	when threads are in a worksharing construct (`for` or `single`)
OMP_worksharings_duration	the duration of worksharing regions
OMP_in_barrier	when threads are in a barrier synchronisation
OMP_in_schedforkjoin	when threads are scheduling work, forking or joining
OMP_in_critical	when threads are in/out/entering/exiting critical sections
OMP_tasks	when tasks in `task` and `taskloop` are created and executed
OMP_tasks_functions	task function created and executed (file and line inside file)
OMP_taskloop	when the `taskloop` construct is executed
OMP_taskwait	when a `taskwait` construct is executed
OMP_taskgroup	when a `taskgroup` construct is executed
OMP_in_taskgroup	when a task is starting or waiting in a `taskgroup`

***Paraver* cfg file**	**Profile showing ...**
OMP_state_profile	the time spent in different OpenMP states (useful, scheduling/fork/join, synchronisation, ...)
OMP_task_profile	the total time, percentage of time, number of instances or average duration spent in task creation (2 and 3)/execution (1)

Table 2.1: Some configuration files to support analysis in *Paraver* – upper part: timeline views; lower part: statistical summaries.

3

Improving memory behaviour

Next you will parallelise a second version of the Eratosthenes sieve that we provide in the sequential `sieve2.c`. This new sequential version includes two changes: 1) all even numbers, except number 2, are initially dropped from the list of prime numbers; and 2) the computation of the prime numbers is done in a range between `from` and `to`, so that the computation of all numbers between 1 and `lastNumber` requires an outer loop that sieves blocks of a certain block size. Look at the new version of the code and make sure you understand the differences with the original sieve in `sieve1.c`.

1. Before trying to parallelise it, compile it for sequential execution and execute exploring different values for the block size:

Version/Block size	**Execution time**	**Speed-up wrt. sieve1-seq**
sieve1-seq 100000000		1
sieve2-seq 100000000 100000000		
sieve2-seq 100000000 10000000		
sieve2-seq 100000000 1000000		
sieve2-seq 100000000 100000		
sieve2-seq 100000000 10000		
sieve2-seq 100000000 1000		

   Try to explain the reasons behind the behaviour that is observed in the execution time.

2. Again, this algorithm is well suited for an iterative task decomposition. As in the previous chapter, write at least two parallel versions, one making use of `for` and another making use of `task` (**not** `taskloop`) to distribute the work among processors.

3. Execute and trace the execution of your parallel versions[1]. You should explore different task granularities for your decomposition in order to find the one that minimises or better trade-offs load balancing issues, overheads, memory issues, ... Always make sure that your parallel code leads to the correct solution.

4. For the scalability analysis we recommend that you use 10^9 instead of 10^8 as we used before. Appropriately change the submission scripts to execute `sieve2` and with this input size.

[1] Change all submission and execution scripts to handle the execution of `sieve2`, which includes an additional argument.

4

What to deliver?

Deliver a single compressed file (`GZ` or `ZIP`) with all versions (C source codes) developed for `sieve1` (`for` and `taskloop`) and `sieve2` (`for` and `task`). Also include a file summarising and comparing the execution times obtained for the 4 different versions when executed with different numbers of processors (e.g. 1, 2, 4, 8 and 12) for the same input size (e.g. $100,000,000$); for `sieve2` please indicate the block size that has been used.