Performance Analysis

Computadors – Grau en Ciència i Enginyeria de Dades – 2019-2020 Q2

Facultat d'Informàtica de Barcelona – Departament d'Arquitectura de Computadors

At the same time that it is very important to understand your application, the algorithm being used, how the compiler and toolchain work, and the libraries and Operating System services used, it is also critical to understand the reasons for their behavior and performance. Are they used properly? Can we improve on the way we used them, and get more performance out of the architecture?

In this session, we will study some of the techniques used to evaluate performance, how to obtain this information, and how the system is behaving when running your application.

Analyzing the execution time and other events from programs

The operating system provides information about time, and various events that occur during the execution of the programs.
On the matrix initialization program explained in the theoretical sessions, perform the following exercises, and write your answers in the "answers.txt" file for this session:

Exercise 1

Compile the matrix initialization example:

- make initmat

Execute it with the commands:

- time ./initmat
- /usr/bin/time initmat

Do they report similar execution times?
Do the times reported by the application itself agree with the times indicated by the "time" commands?

As an example:

```
ini: 146197.000000 us
ini: 51369.000000 us
ini: 51369.000000 us
ini: 51361.000000 us
ini: 51382.000000 us

real 0m0.358s
user 0m0.275s
sys  0m0.081s
```

Try to verify the timings also after **exchanging the matrix access to j,i**:

As an example:

```
ini: 463808.000000 us
ini: 359586.000000 us
ini: 364003.000000 us
ini: 364131.000000 us
ini: 364129.000000 us

real 0m1.926s
user 0m1.815s
sys  0m0.104s
```

At the end of the exercise, select back the matrix access (i, j), in order to do the following exercises.

Exercise 2

Change the initmat.cpp program to call the “get_resource_usage” function (in file rusage.cpp). See also the “getrusage” system call information in the man page. Call this function at the beginning and before the end of the initmat program. In order to do this, you will need to use the rusage.h file as a header in initmat.cpp, and compile and link both initmat.cpp and rusage.cpp.

Be sure to be using the version of the program that uses the (i, j) access to the matrix, as initially provided.

Indicate in “answers.txt”, which specific changes you need to do in:

- initmat.cpp
- Makefile

Exercise 3

Now that you have already collected the information about the resource usage of the initmat.cpp program, let’s see some of the relevant information, when measuring what happens during the 5 outer loop iterations:

- User mode time
  - The ru_utime field is a timeval structure (seconds and microseconds), use the support function “timeval_to_ms” in rusage.cpp to transform the times to miliseconds
- Maximum resident set size
  - The ru_maxrss field is a long integer
- The number of voluntary context switches
  - The ru_nvcsw field is a long integer
- The number of involuntary context switches
  - The ru_nivcsw field is a long integer

Write your results in the “answers.txt” file.

Exercise 4

With the initmat.cpp sample using the original matrix access, (i,j), add parallelism in the outer loop that initializes the matrix, and get the same measurements, with 1, 2, 4 and 8 hardware threads:

- User mode time
- Maximum RSS
- Number of voluntary context switches
- Number of involuntary context switches

You can use the “taskset” command to be sure to use 4 threads from 4 different cores:

```
$ OMP_NUM_THREADS=1   taskset  -c  0     ./initmat
$ OMP_NUM_THREADS=2   taskset  -c  0-1   ./initmat
$ OMP_NUM_THREADS=4   taskset  -c  0-3   ./initmat
$ OMP_NUM_THREADS=8   taskset  -c  0-7   ./initmat
```

Write the measurements in your “answers.txt” file.

Exercise 5

Now execute the initmat program with less hardware threads than software threads, for example, using these combinations:

```
$ OMP_NUM_THREADS=2  taskset -c 0    ./initmat
$ OMP_NUM_THREADS=4  taskset -c 0    ./initmat
$ OMP_NUM_THREADS=4  taskset -c 0-1  ./initmat
$ OMP_NUM_THREADS=4  taskset -c 0-2  ./initmat
```

Describe in "answers.txt" the changes in the number of voluntary and involuntary context switches that the program gets in these conditions.

Exercise 6

Overall, you should be able to fill in the following table for each metric of interest, for example: **CPU user time**, and **voluntary context switches**:

Metric of interest	Indexing (i,j)		Indexing (j, i)	
Num CPUs	Outer parallel	Inner parallel	Outer parallel	Inner parallel
1				
2				
3				
4				

Analyzing the execution time and other events from programs

Exercise 7

Study and run the "tim" program and describe in the answers.txt file what it does.

Common information for exercises 8-11

Now, let's use the detailed timing provided by the clock_gettime system call, to measure several OS services:

- usleep(1), program time_usleep1.cpp, provided
- getpid(), you will need to develop this program, based on the time_usleep1
- fd = open("myfile.txt", O_CREAT|O_TRUNC|O_RDWR, 0644) + close(fd), developed by you as well
- fork() + exit(0) + waitpid(...), developed by you as well

Observe that the "Makefile" provided is already able to compile correctly the 3 new programs, provided you name them as:

- time_getpid.cpp
- time_openclose.cpp
- time_forkwait.cpp

As you will be measuring average execution times, you can tune the value of the HOWMANY macro from 300000 iterations, to a lower or bigger value, if necessary. Lowering may be necessary in the last implementation (fork+exit+wait), as they probably take longer.

Exercise 8

Run the "time_usleep1" program and describe in the answers.txt file what it does. Also, complete it, so that it also computes the average execution time of each usleep(1) call.

Try also to change the program to test with usleep(5) and usleep(10). You should determine that the precision of the usleep blocking mechanism is not so fine grain, and the program does not take 5 or 10 times more.

Exercise 9

Now program the new version time_getpid, and compare the timing results. Is getpid a faster system call than usleep(1)? Write your conclusions into the "answers.txt" file.

Exercise 10

The next step is to program the open+close version. Please, make the program to check the error of the "open" system call. If you do not check for errors, it may happen that an error in the system call provides no useful timing information. Usually, because of the error found, the call will behave much faster than what it should be.

Exercise 11

Finally, complete the fork + exit + wait program, and check its execution time. In this implementation, it is good that you check for errors from both the fork and the waitpid system calls.

Analyzing the bandwidth of data transfers to the file system

The "dd" (disk-dump) command allows you to easily check the bandwidth obtained from reading and writing data into the filesystem.

In order to write a large file to disk, and get the transfer statistics, we can use (please be sure that you have more than 1 GByte of disk space available in the /tmp directory):

```
$ dd if=/dev/zero of=/tmp/myfile.dat count=1024 bs=$((1024*1024))
```

And in order to read from a file, we can use:

```
$ dd if=/tmp/myfile.dat of=/dev/null # this way, dd selects the proper bs
```

Where the command line options provided to dd mean...

- if, input-file
- of, output file
- count, number of blocks to read/write
- bs, block size

You can also read from the source with a different block size with respect the destination file, use:

- ibs, input block size
- obs, output block size

And the special devices can be:

- /dev/null, provides end-of-file when reading from it, and all information written to it is lost
- /dev/zero, provides zeros (0x00) when reading from it, and all information written to it is lost
- /dev/full, provides zeros (0x00) when reading from it, and it reports the error "no space left on device" when writing to it
- /dev/random, provides random numbers when reading from it. It is usually not able to provide a

large amount of numbers, as it uses external events to increase the entropy of the numbers generated. Writing to /dev/random updates the entropy pool, but it does not accelerate the production of random numbers.

- /dev/urandom, provides pseudo-random numbers, without the limitation of "/dev/random". Also, writing to /dev/urandom updates the entropy pool of /dev/random.

Exercise 12

Determine the write bandwidth of the filesystem in the /tmp directory, using this command shown before:

```
$ dd if=/dev/zero of=/tmp/myfile.dat count=1024 bs=$((1024*1024))
```

Check increasing file sizes to determine if the measurement is stable or not.

Report the average result obtained from 5 trials in the "answers.txt" file.

Exercise 13

Determine the read bandwidth of the filesystem in /tmp directory, by reading the file previously created with the command seen before:

```
$ dd if=/tmp/myfile.dat of=/dev/null
```

Report the average result obtained from 5 trials in the "answers.txt" file.

Exercise 14

Compile the next 3 versions for the matrix.cpp program (use make, the Makefile is already performing these compilations):

- make matrix.avx2 matrix.sse2

matrix.avx2 uses the AVX2 vector extensions (more advanced, with up to 8 float values per register and operation), and matrix.sse2 uses the SSE2 extensions (less advanced, with up to 4 float values per register and operation).

Execute the programs, and compare their results. Without any environment variable, the executions will use all available threads:

```
$ OMP_NUM_THREADS=1 ./matrix.sse2
$ OMP_NUM_THREADS=1 ./matrix.avx2
$ ./matrix.sse2
$ ./matrix.avx2
```

Write your findings in your "answers.txt" file.

Upload the Deliberable

To collect all contributions to this deliverable, you can use the tar command as follows:

```
# tar czvf session11.tar.gz answers.txt *.cpp *.h Makefile
```

Now go to RACO and upload this recently created file to the corresponding session slot.

Annex A

Installing PAPI (Performance Application Programming Interface)

PAPI is a library used to access the Performance Monitoring Counters present in most of the processors available today.

Website: http://icl.utk.edu/papi -- The University of Tennessee, Knoxville

Downloading:

```
$ git clone https://bitbucket.org/icl/papi.git
```

Installation:

```
$ cd papi/src
$ ./configure –prefix=$(HOME)/papi-install
$ make
$ make install
```

Setting the environment variables:

- Bash ($HOME/.bashrc):

```
$ export PATH=$HOME/papi-install/bin:$PATH
$ export LD_LIBRARY_PATH=$HOME/papi-install/lib:$LD_LIBRARY_PATH
```

- Tsch ($HOME/.tcshrc):

```
$ setenv PATH $HOME/papi-install/bin:$PATH
$ setenv LD_LIBRARY_PATH $HOME/papi-install/lib:$LD_LIBRARY_PATH
```

Testing the installation, run:

```
$ papi_avail
...
$ papi_native_avail
...
$ papi_mem_info
…
```