Parallelism and cie.

Adrien GUINET <adrien@guinet.me>,
Serge GUELTON <serge.guelton@enst-bretagne.fr>,
inspired from Mehdi AMINI <mehdi.amini@mines-paristech.fr>,
and Serge GUELTON...

03-05 december 2014

1 Personal data leaks

In an IT system, storing user passwords securely is a task where lots of companies have already failed. Indeed, bad practices have been shown due to more or less recent data leaks [1] [2] [3]. As a recent example, Domino's Pizza user database was publicly uploaded on the Internet a few weeks ago. It contained personal information (with hashed passwords) of more than 500k of their customers. [4]

In this section, we're going to see how we can get back some of these hashed passwords and how this process could be optimized.
[5]

Exercises

1. Download the database here : http://files.geekou.info/tp/domino.bz2.
2. Which information are available ? What can you say about the origin of the people in this list ?
3. What can you say about the hashed passwords ? More precisely, why is it a bad idea not to have salted hashed passwords ?
4. Try and find the passwords that are used by more than one people. Keep the 100 most used ones. Why would a bad person focus on these passwords ?

1. https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic
2. http://en.wikipedia.org/wiki/2012_LinkedIn_hack
3. http://www.securityweek.com/youporn-users-warned-change-passwords-after-data-leak
4. https://nakedsecurity.sophos.com/2014/06/16/dominos-pizza-hacked-customer-database-held-to-ransom/
5. **Disclamer :** this (real) story is mainly an excuse to introduce various ways of exploiting modern parallel machines. One "real true hacker with dark glasses and a hoody in a cave" might go through easiest ways to accomplish some of these goals.

1.1 Get the words right

So we got ourselves a list of hashes. The first thing we could attempt is to build a list of potential used passwords, and check our hashes against them!

1.1.1 Dictionaries

Your very computer already got a fairly complete dictionary of all the words of a given language. **aspell** is one of the system providing that. Try to figure out how to dump all the words of the French language!

1.1.2 RTFM

The French manpages can also be a good source of potential words of a given language. Most of them are stored in /**usr**/**share**/**man**/**fr** in a compressed format, which decompress in an ancient format named `groff`. What we could do is extract the list of unique words from all these man pages.

In order to do that, we'll first practice on a single man page.

Exercises

1. Choose a random man page, decompress it using `zcat` to see what `groff` it looks like. Use `groffer -k -Kutf8 --text` to translate this language to raw ASCII (easier to process) and then use `grep -o -E '\w+'` to dump all the words of every line. Finally, pipe this to `sort -u` to get the list of unique words.
2. After having exercised by hand on a file, write a small bash script that takes a man page as an argument, and print the list of unique words.
3. Compare the performance of your script with the one of your neighbor(s). Why some are (maybe) faster than other?
4. If not already done, re-write your script without using any temporary file. Compare.

1.1.3 Data Parallelism

Now that we optimised the process on one single man page, let's try to process hundreds (or thousands) of manpages at once. First we need to get as many manpages as possible. We can use the `find` command for this :

```
%> find /usr/share/man/fr -type f
```

This will print lots of lines on your terminal! You can use `head` to just get a few lines.

Exercises

1. How many man pages did you find ?
2. Write a program that calls the script from section 1.1.2 for every man page found, and get the final list of unique words.
3. Measure the execution time of the above script.
4. Which part of the script is the most time consuming ?

We will now try to optimize this script by using the most of the multiple cores of our computer.

Exercises

Using the language of your choice, rewrite the above script such as the script of section 1.1.2 is called in a parallel way.

One possible way (but not the only one !) is to use **GNU Parallel**. This software allows to express many form of data parallelism (the man page is full of example), and can be very useful in our case !

Tu install **GNU Parallel** on your machine, do the following :

```
%> wget http://ftp.gnu.org/gnu/parallel/parallel-
    latest.tar.bz2
%> tar xf parallel-latest.tar.bz2
%> cd parallel-*
%> mkdir build && cd build
%> ../configure --prefix=~/bin && make && make install
[...]
```

It will end up in your `$HOME/bin` directory.

1.2 Crack it !

Now that we've built a fairly descent dictionary of French words, we will make a program that will try and figure out the original passwords from the hashed ones. The idea is to write a software that takes as input a list of hashes and a list of words, and try to find words whose hash values matches one of the inputs.

For instance, if the list of words is :

```
poney
cat
toto
password
1234
```

and the hashes are :

```
9684dd2a6489bf2be2fbdd799a8028e3
81dc9bdb52d04dc20036dbd8313ed055
75e9c42d8a7fde6ecd29f27b89a2d2a2
```

the output would be :

```
9684dd2a6489bf2be2fbdd799a8028e3 poney
81dc9bdb52d04dc20036dbd8313ed055 1234
```

Preliminary exercises

- ☐ Try and figure out how many MD5 hashes per second can compute one core of your computer.
- ☐ Same using all the available core/threads.
- ☐ Compared to the list you've created, what can you conclude ?

Exercises

- ☐ Write a simple program that does what we want and measure its performances.
- ☐ Parallelize the most computation intensive part of your software. Measure its performances.
- ☐ If you can't find any password, try to create a better word list ! :) (hints : try to find on the Internet some public common passwords lists) [6]

Some conclusions

1. Why is it a bad idea to use the same passwords on different sites ?
2. What would you recommend to Domino's Pizza network administrators ?
3. Will you ever buy a pizza from them again ? [7]

2 Let C : warming up

In this section, we'll go back to more low-level languages to try and understand how some basics algorithms can be sped-up.

2.1 Total Recall

You remember that a reduction is a loop that looks like this :

```
int r = 0;
int vec[n];
for (size_t i = 0; i < n; i++) {
        r += f(vec[i]);
}
```

If `f` has the good properties (which ones ?) there is a generic way to parallelize it using partial reductions (divide-and-conquer again !).

6. Un mot de passe trouvé ⇒ une bière au foyer !
7. N'oubliez pas vos quatre fruits et légumes par jour. Le houblon peut être comptabilisé comme tel en bretagne.

2.2 Warming up : Min/Max of an array

Computing the minimum and maximum values of an array is a typical case of reduction. Let us consider an array of unsigned 32-bits integers.

It's always important to have a simple sequential reference algorithm. And then to optimize it.

So we'll implement many versions of the algorithm, starting from the basics. Take a deep breath and have a look to the code in `minmax/1-serial`.

Exercises

1. The sample code iterates over the input array twice. Make sure this does not happen and measure the performance impact. Explain.
2. Try to parallelize the resulting loop using OpenMP and a `for` clause combined with a `reduction` clause. Any idea concerning how the parallel reduction is implemented ?
3. **[Only the brave]** try to use the Intel's framework TBB http://threadingbuildingblocks.org/. [8] The key idea is to use the `tbb::parallel_reduce` skeleton algorithm.

3 Edge detection

See http://en.wikipedia.org/wiki/Edge_detection

Edge detection is a fundamental tool in image processing and computer vision, particularly in the areas of feature detection and feature extraction, which aim at identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities.

The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world. It can be shown that under rather general assumptions for an image formation model, discontinuities in image brightness are likely to correspond to :

- discontinuities in depth,
- discontinuities in surface orientation,
- changes in material properties and variations in scene illumination.

In the ideal case, the result of applying an edge detector to an image may lead to a set of connected curves that indicate the boundaries of objects, the boundaries of surface markings as well as curves that correspond to discontinuities in surface orientation. Thus, applying an edge detection algorithm to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image. If the edge detection step is successful, the subsequent task of interpreting the information contents in the original image may therefore be substantially simplified.

8. You may need to install it by hand.

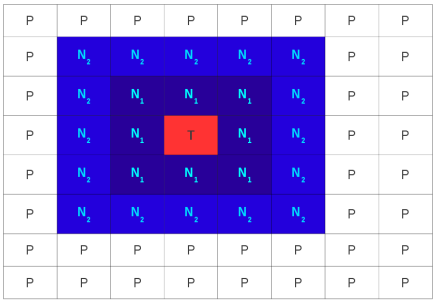

FIGURE 1 – An image (array of pixels) with a stencil in blue used to compute a single pixel T. The stencil size can be adapted, for instance N_2 are the pixels for a size of 2.

FIGURE 2 – The result of a 20 points stencil (width 5).

3.1 Sequential Algorithm

This is far from state-of-the art algorithm, but it's simple enough to be easily understandable. Pixels are processed one after one, and for each we examine the neighbors seeking a difference that would indicate that we're sitting on an edge. This classical scheme is called a stencil in image processing. Many algorithms are working this way, the only change is the shape and the width of the stencil, and the operation done on the neighbors.

Here the stencil is the set of pixels sittings in a square centered around the target pixel. The stencil width has to be adapted to the image definition. Figure 1 illustrates this shape.

The process here involve three operators : an erosion, a dilatation, and a difference. The erosion consists in assigning to the destination the minimum value in the neighborhood ; the dilatation is taking the maximum value in the neighborhood ; and the final result is obtained by the difference between dilatation and erosion. The figure 2 illustrates the result of this process.

Check the `qt_edge/0-naive-linear` directory for a naive (and working) implementation of the above algorithm.

Exercises

1. Check the effect of changing the optimization flag from `-O0` to `-O2` then `-O3` and `-O3 -march=native -mtune=native -ftree-vectorizer-verbose=1`. Comment.
2. Use **valgrind**(1) and the `-tool=callgrind` switch to identify hotspots. `-g` must be used to get good information.

3.2 Task based parallelism

You should have noticed that the algorithm consists of three loop nests. The two first loop nest produced erosion and dilatation, and the third use both. It means that we should be able to produce erosion and dilatation at the same on different processors.

Exercises

1. Use the `#pragma omp sections` and `#pragma omp section` clauses to benefit from task parallelism. What speedup did you obtained ? What is the limitation of this approach ?
2. The `sections` clause is quite old-fashioned. Try to use the `task` clause instead. Any change ? Unlike `sections`, tasks can be nested (but you actually do not care here).

3.3 Data parallelism

Inside each task, you have a big loop. Let us parallelize it using OpenMP.

Exercises

1. Use `#pragma omp parallel for` to take advantage of data parallelism. What speedup did you obtain ? What can you notice with respect to the previous version ?
2. Play with the `schedule` clause to try different scheduling. Any performance impact ?

3.4 Improve locality

Locality is critical for performance **and** for parallelism. Let us illustrate this.

Exercises

1. Perform inlining then loop fusion (cf. http://en.wikipedia.org/wiki/Loop_fusion) as you already did to improve the sequential algorithm. Measure the performance gain.
2. Try to explain the performance boost. Is it only linked to the loop overhead ? Count the number of thread synchronizations before and after the fusion.

3.5 Vectorizing

We will now try to make use of vector instruction to speedup the process.

Exercises

1. Which vector instruction sets are available on your computer ?
2. How many pixels can you handle in one vector register ?
3. Implement a version and measure the speedup. You need to use the `__m128i` type and the `_mm_setzero_si128`, `_mm_set_epi32`, `_mm_loadu_si128`, `_mm_min_epu8`, `_mm_max_epu8`, `_mm_subs_epu8` and `_mm_storeu_si128` from the header `xmmintrin.h` and `immintrin.h`.

   The basic idea is to unroll the inner loop by the right factor and then process multiple data at once...

4 Histograms

Histograms is a widely used method to display statistics on a wide range of data.

The general problem can be expressed as the following : given a sequence of floating point numbers between $[0, 1[$, we are going to compute their distribution across fixed size intervals.

For instance, if the size of the intervals is 0.5, we will compute the number of points between $[0, 0.5[$ and $[0.5, 1[$. For 0.25, we will compute the numbers between $[0, 25[$, $[0.25, 0.5[$, $[0.5, 0.75[$, $[0.75, 1[$, and so on...

Figures 3 and 4 show some examples of such histograms of values between 0 and 1 with an interval size of respectively 0.2 and 0.33.

One can notice that the size of the intervals will determine their number.

In the code repository, there is an application that computes such histograms and displays them. The usage is the following :

```
Usage: ./hist [--file file] [--gen-uniform N] [--gen-
    normal N]
```

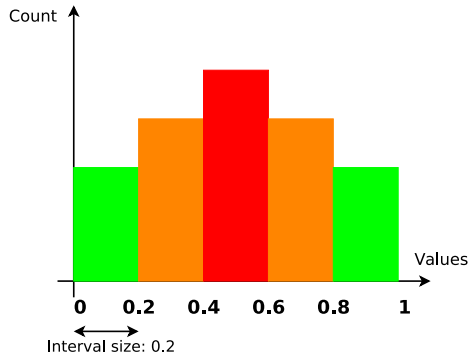

FIGURE 3 – Histogram of values between 0 and 1 with an interval size of 0.2.

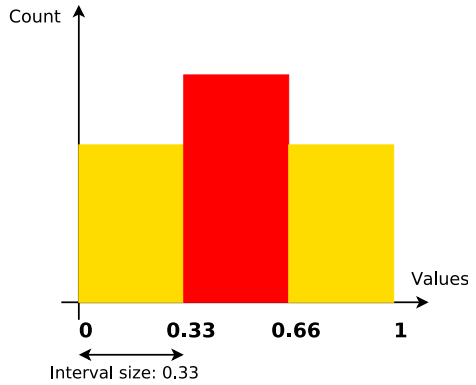

FIGURE 4 – Histogram of values between 0 and 1 with an interval size of 0.33.

You can generate an histogram from a list of points provided in a text file (as the one previously generated) or generate randomly or normally distributed numbers.

Inside the `histogram` directory, there is a piece of code that computes this histogram and perform various benchmarks. Here is how it is organised :

- `core` : static library with the Histogram class that contains the computation algorithm
- `benchs` : benchmark the different algorithms with a given interval size
- `benchs_plot` : benchmark the different algorithms for a range of interval sizes and plot the results
- `qt` : a small Qt application that displays the final histogram according to a configurable interval size

The input value scan be of different kind :

- a uniform set of random values between 0 and 1
- random values according to a gaussin curve between 0 and 1
- a file of floating point values

All the algorithm are in the Histogram class in the file `core/histogram.cpp`. The `compute_serial` function is already implemented and can compute a histogram of values between 0 and 1. The goal of the following exercises is to implement optimized versions (`compute_sse` and `compute_omp`) and compare their performance.

Exercises

1. Compile the code inside the `histogram` directory :

```
%> cd /path/to/histogram
%> qmake -project hist.pro
%> make
```

2. Play with the Qt application a bit, using different kind of inputs (check the output of the command `qt -help` for more information). Don't be scared and try to compute an histogram of about millions of values. Do you notice any slowdown at some point ?

From now on, we will only play with uniformly random values (using the `-gen-uniform` option).

1. The benchmark application in the `bench` directory will show the time taken by the different histogram algorithms for a given histogram size. To begin with, all the algorithms are the same, so you should get more or less the same values for each test.

   Try to run it for various input and histogram sizes. Note the parameters used for significant results above half a second.

   Bonus : take a look at the `histogram_bench.cpp` file and explain how it works.

2. Run the `benchs_plots` application. Identify various steps where the performance decreases significantly. Can you correlate these values with some specifications of your processor? (**hints :** take a look at `hwloc-ls` and think about the flow of data inside your algorithm).
3. Let's try to optimize this by implementing a parallel version of the `compute_serial` function using atomic operations in `compute_omp`. Is there any significant gain against the previous runs you've done? Why?
4. Implement a parallel version using a reduction and thread-local intermediate buffers (by replacing `compute_omp` or adding a new function). Is there any improvements? Why?
5. Let's try to adopt a different strategy where we first optimize the "serial" version. Try to vectorize the computation part of this algorithm (by implementing the `compute_sse` function).

   You can get help on the vector operations that your processor provides thanks to this Intel website : https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

   The instructions that you will need are :
   - ☐ loading a 128-bit vector of two double precision floating point numbers : `_mm_load_pd`
   - ☐ setting a 128-bit vector with the same two double precision floating point value : `_mm_set1_pd`
   - ☐ mathematical operations : `_mm_sub_pd` and `_mm_div_pd`
   - ☐ number conversion from floating point values to integer values : `_mm_cvttpd_epi32`

   Benchmark your new implementation and conclude.
6. Implement a parallelized version of the previous implementation. Do you notice any improvements? Comment.
7. Re-run the `benchs_plots` application. How do you explain the different curves? Why do they all seem to converge to the same point?

5 Tree Graph

A simple way to write a memory-safe Tree structure in modern C++ is to use `std::unique_ptr`. The sources of this TP contains a very simple implementation of a generic tree data type, `<tree.hpp>`. A simple client code is reproduced below :

```
#include <tree.hpp>

int main() {
  auto Root = makeTree(1);
  Root->add_child(makeTree(2));
  Root->add_child(makeTree(3));
  std::cout << Root->size() << std::endl;
```

```
  return 0;
}
```

Note how the code avoids using any call to `new` :-)

Exercises

1. Write a sequential (generic) algorithm to apply a function to all the values contained in the tree. choose an arbitrary order between breadth first and deep first.
2. Make it parallel. Try to avoid over-subscription :-) Is your shceduling balanced ?
3. Write a sequential (generic) algorithm to find a value in the tree.
4. Make it parallel. What's the problem with find, with respect to performance, scheduling and ordering ?

6 Parallel Prefix

Computing partial sum is trivial in sequential :

```
b = [0] * len(a)
b[0] = a[0]
for i in xrange(1,len(a)):
    b[i] = b[i-1] * a[i]
```

but it is not parallel at all.

There exist a recursive version that does not have this issue, as described in http://en.wikipedia.org/wiki/Prefix_sum#Parallel_algorithm. Read the algorithm carefully

Exercises

1. Implement the parallel algorithm without parallel constructs in Python. Is it faster ? Find two reasons, one related to the algorithm, the other related to the language.
2. Implement it in C/C++, then use the `task` clause to parallelize it. Is it faster ? Does using the `if` clause changes something ?
3. Using a threshold, switch from parallel version to sequential version when relevant. Is it faster ?