Experimenting Parallel Computing in Go, C++, Cuda, Python, AutoIt

Summary

This paper tests and compares various solutions to one problem, by using true parallelism and thus accessing to the full computing power of PC desktop configurations.

The study shows how very basic but truly parallel solutions can be developed in several programming languages on various OS and PC hardware platforms. Indications of the relative levels of performance in execution times are provided. Easiness of programming and testing are also compared and discussed.

Warnings :
1/ Elementary level ! No aim to beat records, no aim to implement the latest parallel algorithm...
2/ Attached software programs were kept as clean and simple as possible, to be readable and modifiable by anyone with knowledge in computer programming.
3/ Many unanswered questions....

All program sources are attached in a zip file

Proposed keywords : PC, multiprocessing, load balancing, Amdahl's law, obsolescence

True parallelism ?

Parallelism was originally natively implemented in personal computers architectures, to begin with screen display and other Input/Output functions, thru dedicated onboard chips or « peripheral » add-on cards working side by side with the main cpu, cooperating via hardware interrupts and software buffering. Even the most basic Operating System (OS) was able to manage this real level of parallelism. From the user point of view, the computer could run only one application program, for example either word processing or « Wizardry »…. But the running task was able to use the whole set of peripherals for its own sake. And it had to save its data on a permanent storage so that these data could be shared with other applications or with itself in a later session.

Then came the days of more ambitious Oses, including a scheduling machinery that would organize the sharing of cpu power and memory between « processes », i.e. software programs, either internal services of the OS itself (for example to display a clock, compact busy memory segments, etc) or applications programs such as word processor, media player, Net browser, etc. Even on configurations with one single-core cpu and a short-sized memory, the OS machinery provided a new kind of parallelism, so that the user got the *illusion of unlimited parallelism.* The user was able to run « in parallel » a media player, a Net browser, a spreadsheet… and even have live applications share user selected data.

Internally, the new generation of Oses was based upon low level software abstractions encapsulating the basic « hardware interrupt – software buffering » mechanics into low level software « services » calling generic cooperating and synchronizing procedures.

In other words, the OS evolved from true limited parallelism to unlimited multitasking, thru generic software packed simulated parallelism.

Cpus with multiple physical cores, typically including 2 « p̂hysical cpu threads » per cpu core, considerably extended the available computing power… but nothing changed fundamentally in the internal machinery of the OS, as everything was ready for the transition to multi-core cpus, by design.

The OS (Operating System) regularly redistributes the currently alive system internal routines (services) and applications processes among the cpu threads. The aim of these OS-controlled scheduling operations is smoothing the load of each cpu over time, by regularly moving processes from one cpu thread to another and temporarily putting some of them to sleep in the background. Time critical processes will get higher

priority levels, so that the OS allocates them permanently or reallocates them more often than the less critical ones....

But, if you write you own program, the newly created process will never run on more than *one single cpu thread*, possibly up to a 100 % load level of this *single thread. And* it possibly will be moved to another *single thread* by the OS scheduling machinery several times, till the process comes to its end or is aborted.

If you are a proficient programmer, you can write special instructions in your code to somewhat control the OS scheduling machinery, for example to put your program to sleep for a given time period. In your code, if you are a very proficient programmer, you also may use specific libraries to define checkpoints and inter communications with other running processes, even define OS-controlled parallel segments in your program...

These aims and programming ways are ABSOLUTELY NOT what we mean by « true parallelism » in this paper.

True parallelism means : application-controlled permanent use of a requested level of the available computing power.

How can it be done in a modern OS ?

Typically, one single program splits itself into child processes - not more than one process per cpu thread. Then, each child process runs on its cpu thread independently from each other without interruption. Each child process feeds its own set of results, to be collected at end of program. Let us call this « parallel segmented » multiprocessing.

In other words, when I have launched my « parallel segmented» program, the load level of each cpu thread that I have selected to support a child process is 100 % till the end of my program.

For example, for 100 % global load (Mitec Task Manager Deluxe, M1 machine**) :**

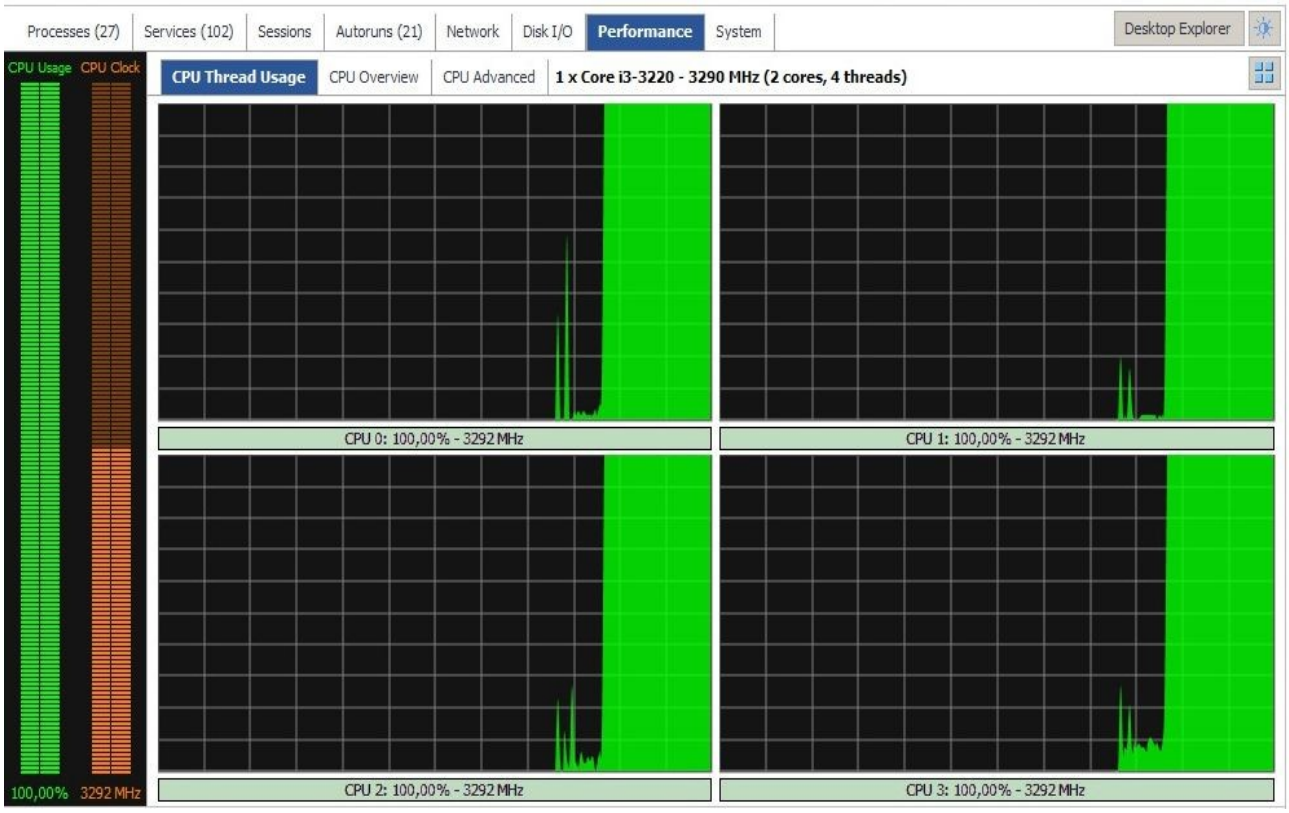

Figure 1.1

For example, for a 50 % global load :

Figure 1.2

This is no fancy : for example, the programming languages Python and Go provide the necessary instructions to do so. But many other programming languages, especially the older ones, do not. Nevertheless, even without any available specific instructions, there are ways to do parallelism « by hand », ie by directly calling the appropriate OS routine in charge of process management or by using a special compiler on the part of code to be executed in parallel.

We will come across examples of each case by testing the same algorithm, programmed in various languages on various hardware configurations.

The test tool

Let us address this simple problem :
- find the divisors of an integer number N of any size.
- find all the divisors, in natural order, with no double appearance.
- in minimum time.

Plato the philosopher defined the ideal number of citizens in a city = 5040.
This number has 60 divisors, starting with 1 2 3 4 5 6 7 8 9 10 12 14 15.
It is a very special number, allowing many combinations, probably why Plato selected it.
Just to say that our problem stands on historical foundations... still to be experimented.

How shall we test that T is a divisor of N ?
N modulo T = 0

Mathematicians recently invented new algorithms for quickly finding the prime numbers among the divisors of N. One of these new algorithms might provide a basis to solve our problem, since all the divisors of N can be computed from this set of prime numbers. But this project would probably require a lot of work with a high risk of producing a rather cryptic piece of software.

On the contrary, the « regular stupid » search method, by testing each candidate number as a divisor of N (ie N modulo candidate = 0 if and only if the candidate is a divisor), can easily be « parallel segmented ». The most immediate idea to do so is by defining segments of candidate divisors of N, then by allocating each segment to one processor (cpu thread) to find the divisors of N in its own allocated segment.

Let us note that :
1/ when T is found to be a divisor of N, so is also its « ghost mirror» N / T
2/ consequently, if we test the candidate numbers stepwise in ascending natural order, we may stop at square root(N)

Simple tests can exclude many candidates without having to execute a modulo operation :
- if N is odd (not divisible by 2), even candidates (divisible by 2) can be skipped
- if N is not divisible by 3 , no divisor can be divisible by 3
- if N is not divisible by 5, no divisor can be divisible by 5
- etc.

The first test is easily implemented, when N is odd, by an incremental step of 2 instead of 1 from odd candidate to next. As regards all other tests for the (non) divisibility of candidates, the more they get complex, the less they can compete versus a direct « N modulo candidate » test. The balance mainly depending on the storage structure of numbers (direct access to decimal digits or not ?) and on the environment of execution (ie interpreted or compiled langage) - more on this later...

Obviouly, when testing, we will allow no other running task but the OS standard services with a minimum set of drivers for human interfaces and display (plus GPU parallelism when necessary).

Expected results of tests according to common sense

Some established truths or obvious facts will be challenged by experimenting :

- Programming for parallelism is a difficult discipline

- Execution time of a parallel program sharply decreases with the number of cpu threads

- Compiled software is always faster than its equivalent interpreted sofware

- Computations in a virtual machine are slower than the real machine

- Parallelism in GPU always beats parallelism in CPU

- With your stupid algorithm, odd numbers will require 50 % less execution time than even numbers

- Modern OSes automatically organize parallelism better than you ever will by programming

Part 1. Parallel multiprocessing on cpu threads

Python

Program file : Loopw.py, ca 160 instructions.

The logical structure is obvious in such a short program.

```
import sys
import os
import math
import datetime
from multiprocessing import Pool

def getnin() -> (int,int,bool) :
    while True :
        Sin = input("\nEnter N  (separators allowed, 0 = exit)\n==> ")
        lsin = len(Sin)
        if lsin > 0 :
            if (ord(Sin[0]) - 48) == 0 :
                sys.exit(0)
        i = 0
        snum = ""
        while i < lsin :
            ch = Sin[i]
            nb = ord(ch) - 48
            if nb in range(10)  :
                snum += ch
            i += 1
        Sin = snum
        if len(Sin) == 0 :
          print("\nBad entry, retry...")
        else :
            Nin = int(Sin)
            break

    Isodd = False
    nb = len(Sin) - 1
    ch = Sin[nb]
    nb = ord(ch) - 48
    if nb % 2 != 0 :
        Isodd = True
    print(Sin + "   (" + str(len(Sin)) + " digits)")
    maxcpus = os.cpu_count()
    snum = ""
    while True :
        snum = input("\nNr of allocatable cpus " + "[1.." + str(maxcpus) + " (default)] : ")
        if len(snum) == 0 :
            ncore = maxcpus
            break
        try :
            ncore = int(snum)
        except :
            print("Bad entry, retry...")
        else :
```

```
            if (ncore <= 0) or (ncore > maxcpus) :
                print("Bad entry, retry...")
            else :
                break
    i = 0
    while i < maxcpus :  # clear the logs
        try :
            os.remove("Logpym"+ str(i) + ".txt")
        except :
            pass
        finally :
            i += 1
    return (Nin, ncore, Isodd)

def initz() :
    if sys.version.split(' ')[0] < "3" :
        print("Python v3 required !")
        sys.exit(1)
    (N, ncore, Kodd) = getnin()
    if Kodd :
        increment = 2
    else :
        increment = 1
    candfloor = 1
    candceiling = math.floor(math.sqrt(N) + 0.5)
    finalcandceiling = candceiling
    segment = math.floor(candceiling / ncore)
    while True :
        if (segment < 100) and (ncore > 1) :                # decrement ncore till segment>=1000
            ncore = ncore - 1
            segment = math.floor(candceiling / ncore)
        else :
            break
    if (segment < 100) or (ncore == 1) :
        ncore = 1
        candceiling = finalcandceiling
    else :
        if Kodd and ((segment % 2) > 0) :
            segment = segment + 1
        candceiling = candfloor + segment
    icore = 0
    params = []
    while icore < ncore :
        params.append((icore, N, candfloor, candceiling, increment))
        icore += 1
        candfloor = candceiling
        if icore < (ncore - 1) :
            candceiling += segment
        else :
            candceiling = finalcandceiling
    return (ncore,params)


def collect(nbcore : int) -> str :
    bigchup = ""
    bigchdn = ""
    icore = 0
    while (icore < nbcore) :
        fname = "Logpym" + str(icore) + ".txt"
        icore += 1
```

```
        try :
            fraw = open(fname,"r")
        except : pass
        else :
            chraw = fraw.read()
            fraw.close()
            if chraw.find("_") > 0 :
                cht = chraw.split("_")
                bigchup += cht[0]
                bigchdn = cht[1] + bigchdn
    return bigchup + bigchdn

def TheLoop(myparams) :
    (thecpu, theN, thefloor, theceiling, theinc) = myparams
    test = thefloor
    chup = ""
    chdn = ""
    ghost = 0
    if test == 1 :
        chup = " 1"
        if theN > 1 :
            chdn = " " + str(theN)
    while True :
        test += theinc
        if test > theceiling :
            break
        if (theN  % test) == 0 :                #  if modulo=0  test is a divisor of N
            ghost = theN // test                #  and so is its "ghost"
            chup += " " + str(test)
            if ghost > test :
                chdn = " " + str(ghost) + chdn
    chup += "_" + chdn
    fname = "Logpym" + str(thecpu) + ".txt"
    fwin = open(fname,"w")
    fwin.write(chup)
    fwin.close()

if __name__ == "__main__" :
    while True :
        (ncore,lparam) = initz()
        print("\nExec on "+ str(ncore) + " cpus...\n")
        twtart = datetime.datetime.now()            # start
        with Pool(ncore) as P :
            P.map(TheLoop,lparam)
        dur = datetime.datetime.now() - twtart      # stop
        itdur = math.floor((dur.seconds * 1000) + (dur.microseconds / 1000))  # ms (milliseconds)
        bigch = collect(ncore)
        sout = "["+ str(itdur) + " ms // "+ str(ncore) + " cpus  ==> " + str(bigch.count(" ")) + " divisors]\n" + bigch +
"\n"
        print(sout)
        f = open("Divpys.txt","a")
        f.write(sout)
        f.close()
        print("Results saved in Divpys.txt")
        sout = input("\nAny key to exit ")
```

The getnin function reads and filters the inputs : the number N, the number of cpu threads for parallel execution, computes a yes/no for N parity.

The initz function prepares the set of cpu threads runs, defining the allocated segment to each cpu thread into a params structure. To make sure that the segment length includes at least 1000 candidates, the number of contributing cpu threads is decreased accordingly : there would be no point running too shortly completed parallel processes.

The collect function gathers the results that have been produced by each parallel process. Each file contains divisors as character strings. In each file, a « _ » character separates the ascending from the descending divisors (the « ghost mirrors »). If no divisor was found in a segment, its file contains only the separator character. The strings bigchup and bigchdn collect the ascending and the descending divisors respectively.

The TheLoop function is the « parallel » code in cpu threads. Divisors in the allocated segment are found in ascending and descending order (« ghosts »), collected in strings and eventually written into a specific file (the file name contains the current cpu thread).

The main is an iteration : input, initialization, parallel loops, collection of results, output.

In this Python version, there are only 3 specific instructions for parallelism.
- in the importation prelude : from multiprocessing import Pool
- in the main : if __name__ == "__main__"
- in the main : with Pool(ncore) as P and the next one P.map(TheLoop,lparam)

The last one launches TheLoop function as a parallel process.
The program calls the special Python module for « multiprocessing ». There are other modules for multithreading, concurrent programming, etc.

Do not ask how parallelism works in the Python machinery… The implementation of multiprocessing in Python is not the same in Windows as in Linux. For example, in Linux, you may move the cryptic « if __name » instruction just before the « with Pool » call for parallel executions, whereas this move will create a mess in Windows….

In this Python program, there is only one filtering test of candidate divisors : the odd test, that allows skipping even candidates when N is odd. It was found that adding more filtering tests would add execution time. Python is an interpreted langage : the shorter, the faster.

Go

Program file : Looping.go, ca 230 lines.

This version cannot accept N numbers longer than 18-19 digits, compatible with the type uint64. For bigger numbers « of any length », we developed another version with the BigInt package.

The structure of the Go program is similar to the Python program with almost identical names of functions.

But many details differ - not for more simplicity.

Note the length of the « import » leading sections. But only one package, the sync package, will be specifically required for our parallel programming purposes, apart from the runtime package for one instruction (in the getNin procedure) to get the number of available cpu threads.

The Go language has one instruction for multiprocessing : the « go » instruction.

This « go » instruction launches Theloop parallel processes in the Initz procedure.

Moreover, in our Go program, the parallel processes must be explicitly submitted to a terminal synchronization (whereas this synchro was implicit in the Python program).

This is done thru the wloops.xxx instructions (sync package) :
- wloops.Add(1) in Initz just before launching one more Theloop processes, increments the number of current parallel processes
- wloops.Done() in each Theloop process, to decrement the number of current parallel processes whenever one finishes
- wloops.Wait() is the first instruction in the Collect function as a precondition for starting it, i.e. when all Theloop processes are terminated

No more filtering test was implemented than in the Python version. It was found that more tests had no measurable effect on execution times. Besides, this preseves the functional identity of Go and Python versions, so that we may compare them..

Go BigInt

Program file : Loopbig.go, ca 230 lines

This version is for N of any length, by using the BigInt Go package (math/big package), with its (heavily) specific instructions.

The program structure and logic are the same ones as the Go version, although local details look very differently.

Part 2. Breaking the barrier to parallel multiprocessing

How can we write parallel programs when the programming langage has no instruction to do so ?

Let us prove how it can be done with one of the most improbable candidates : AutoIt, an interpreted scripting language for Windows.

Differently from Windows integrated script languages, Autoit offers a purely procedural Basic-like syntax. AutoIt is fully documented with examples of use of each instruction from within the help document.

AutoIt has special programming features for interactions with the elementary components of windows, such as menus, lists, etc . Actually, AutoIt's potential domain of use is larger than the one of an universal programming langage. But AutoIt's domain of excellence is clearly in applications prototyping and scripting for automating interactions, not as a language to develop big projects nor time critical routines.

AutoIt comes with a very light environment for software development based upon its interpreter… and there is also an integrated compiler !

AutoIt's programs, scripts or compiled .exe files, will run as processes on one single cpu thread – just like scripts and .exe files from any other programming environment.

Nevertheless, parallel multiprocessing in AutoIt is possible, without any trick, by sticking to the regular set of instructions in the language.

In the previous parts, the same design of parallel programs repeatedly showed up, especially the need to separately design the parallel code so that it can be executed independently in parallel on several cpus.

AutoIt has no instruction, no internal machinery in its interpreter to do so.

But it can be done :
- we have to split our AutoIt script into two programs : the main program and the « parallel » program
- the parallel program will run independently from the AutoIt interpreter, will be duplicated in parallel processes : the « parallel segment » process will be a compiled one
- we have to find how to directly allocate a process to a given cpu thread, and how this can be done several times with specific parameters to each instance of the parallel process

The technical core answer to the second item, is the Run instruction of the AutoIt langage, when written with appropriate parameters :

```
Run(@ComSpec & " /C start /affinity "&$Hmask&" loopkim.exe")
```

This instruction calls the Windows start command with an affinity parameter (preliminary set into the $Hmask variable) to launch the loopkim.exe (parallel) program. The affinity parameter defines the allocated cpu thread.

Thus, the parallel instances of loopkim.exe are launched sequentially via a Run instruction within a loop.

The parameters of each loopkim.exe instance (floor and ceiling of candidates in tjhe allocated segment, increment of search) cannot be directly added to the Run instruction. It was found that global « system » variables could be used to solve this problem. So, the parameters of each parallel instance are set via envset instructions, betting on a fast read of the same global variables via envget instructions by the launched instance before these variables are reset for the next instance. There are safer ways, none faster.

Note. In Windows XP, the underlying DOS start command does not accept the affinity parameter. So, when XP is detected as the OS by the AutoIt script, a special Run instruction is used, with an instrumental intermediate utility to launch the loopkim.exe on the selected cpu thread (see Bibliography).

There is not much more to say, as we stick to the same structure of split programs into a parallel and a main part, and we kept the usual names of functions as much as possible. Be warned that our AutoIt programs are written with the « main » on top.

The 15-digit size limit on N numbers is an experimental one. Numbers with more than 15 digits, thru some intermediate calculations, are internally automatically converted in float format… and the list of found divisors can be a wrong one - not only by displaying float numbers instead of integers.

Divisoptkim.au3 is the main AutoIt script, ca 230 instructions
Loopkim.au3 is the parallel loop, ca 30 instructions

Loopkim.au3 must be compiled as an .exe file by the AutoIt compiler, and installed in the same directory as the main script. (Windows XP : also with the StartAffinity utility).

Means and ways to accept bigger N numbers by adding programmed functions for arithmetics were tested with no success, i.e. terrible execution times.
Adding smart tests on candidate divisors in the parallel instances were not successul either, with clearly negative consequences on execution times : the AutoIt compiler has probably a very limited capacity for code optimization.

Believe it or not :
- this parallelizing project in AutoIt was our first one, before the Go, Python, C++, CUDA projects
- the main difficulty in projects of this kind is « documentation », to say nothing of fake solutions spread as confirmed truths on the Net : you definitely have to experiment !

Part 3. Parallel multiprocessing on GPU threads

In this Part 3, we stand in the Linux world, some details would certainly be different in Windows.

Introduction to the CUDA world

An Nvidia graphic card for screen display is assumed, with the installation of a proprietary version of the Nvidia driver for this card *and the installation of the CUDA toolkit*. For the CUDA toolkit, preferably install the (available?) version in your Linux distribution packages even if not the latest, as this version will be automatically configured.

CUDA is a word of the Nvidia vocabulary : Compute Unified Device Architecture, a proprietary technology for parallel multiprocessing, allowing GPU threads to be programmed as concurrent processors... many ones, much more than the number of cpu threads in a gamer's computer !

To do so, one has to think in vectors : « Device memory can be allocated either as linear memory or as CUDA arrays » (6,2,2 Device Memory in CUDA C++ Programming Guide, Release 12,2).

Well, this is how we decided to solve our problem by CUDA programming : let the GPU parallel threads tell us where are the divisors in a long segment of candidate numbers (ie a vector array, one index = one GPU parallel thread = one candidate divisor), by putting a « Y/N » answer on the vector array, and let our main program collect the results and recall the GPU parallel processing as many times as necessary for segments of candidate divisors.

This strategy is clearly not an optimal one as some calculations will be duplicated in the GPU and in our host CPU. The GPU threads will have to compute many modulo instructions, just to be able to answer « Y » when a modulo 0 is found and the main program will later have to compute the « ghost mirror » divisor at each « Y » index. However, considering the increasing scarcity of the divisors of N in segments of numbers when the candidate divisors get higher, this brute force strategy seemed worth testing, even if CUDA programming was not designed for it.

Once more, we certainly won't get the same level of performance with numbers that can be directly operated by regular instructions in hardware versus « bigger numbers » that will require specific sets of programmed routines for arithmetic operations. Moreover, for « bigger numbers », as the GPU code has to be in C and abide by very specific rules, one cannot directly use available libraries for big numbers arithmetics. Consequently and for the sake of readability, we developed our own (simplistic) set of arithmetic routines for big integers, by freely reusing a published library of routines for BigInt arithmetic operations (see Bibliiography).

Finally, we get 3 versions for testing :
- a regular Go version for uint64 numbers (ie not more than 19 digits) and regular C (long long) arithmetics in the GPU
- a Go BigInt version with our own C programmed arithmetic routines in the GPU
- a C++ version, both sides with our own C programmed arithmetic routines

The first Go version, the uint64 one, includes no filtering test on candidate divisors in the GPU code, apart from a locally odd/even test both on N and the candidate divisor. The other versions include the filtering tests for divisibilty by 3 and 5 in the GPU code, but have the main program send the results of divisibility tests on N to the GPU so that each GPU thread may only care for its candidate divisor. An extended testing of candidate divisors in the GPU certainly does not entail a noticeably higher load since the tests for 3 and 5 divisibility are on the digits of each tested candidate divisor, and the internal storage of big numbers (for use of our own C developed arithmetics) allows direct access to their decimal digits.

In the last two versions, the ones with our own C developed arithmetic routines, the maximum allowed length of big numbers is defined as a parameter : BIGSIZE. It is either declared as a regular constant or as a

parameter for the compiler. So, we cannot accept « numbers of any length » actually, but we can easily lift the barrier.

All versions are split in two parts. The .go or .cpp part is the main program. The .cu part contains both the GPU code and the function that activates the GPU then sends the results back to the main program. Our .cu parts are written in C.

How the two parts could be built into one executable program unexpectedly emerged as a challenging problem, with solutions depending on the langage in the main part, ultimately found in rare empathetic contributions on the Net.

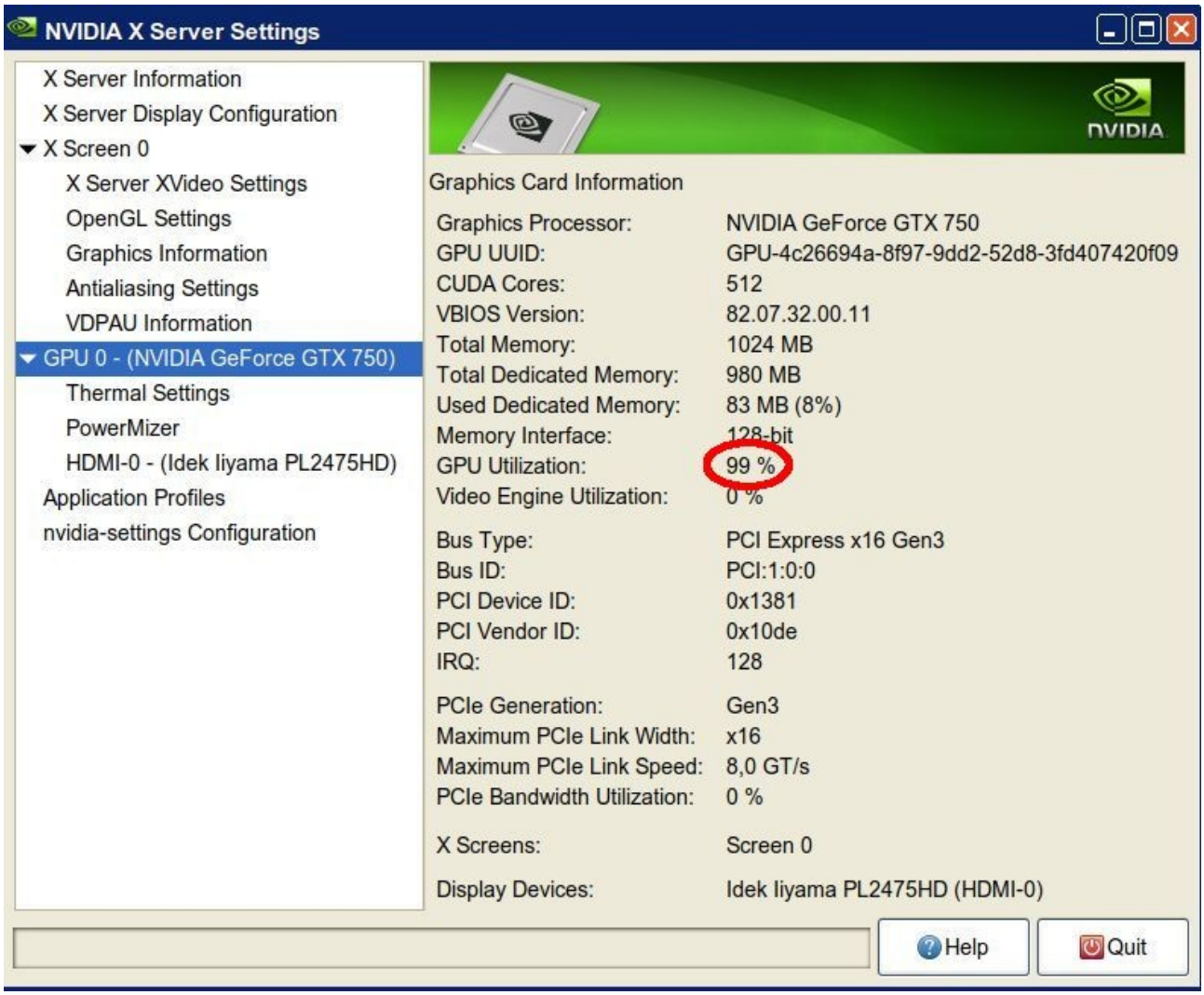

Figure 3.1

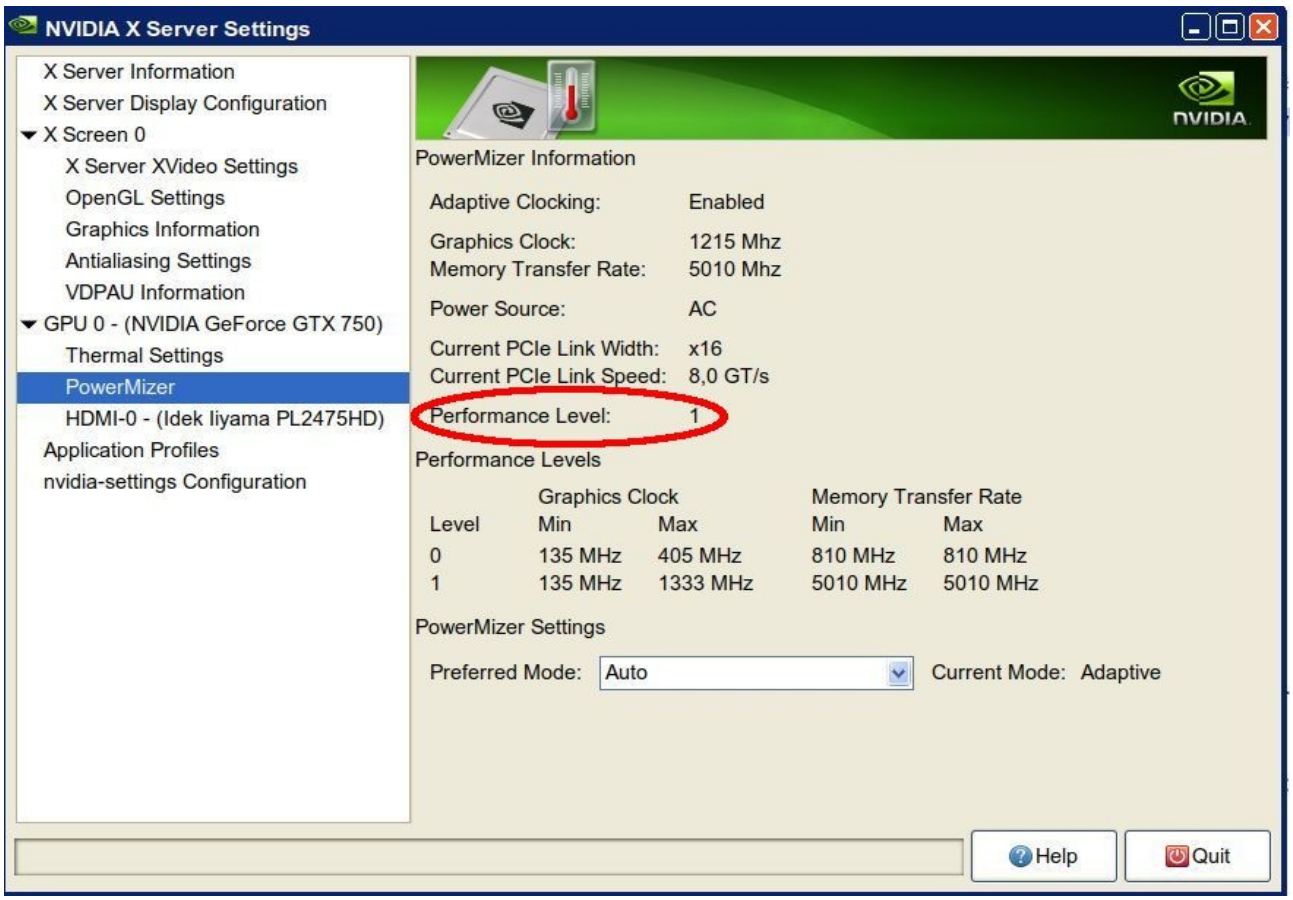

Figure 3.2

Important remark. CUDA programming has its learning curve... and it is easy to cause a bad crash in the execution of the code in the GPU – typically, with a badly assigned pointer address, you will get a « core dump » message in the Terminal… which means that the screen display function of the graphic card is still operating whereas the GPU function has died ! You will have to reboot the computer to get the GPU function back alive, but there is good news : no black screen.

Let us now have a more precise look at each one of the three versions.

Go & cuda

Loopingul.go is the main, ca 140 instructions
Loopgpul.cu is the GPU calling and running part, ca 40 instructions

This version is different from the others, as it has the GPU directly find the divisors and their ghost mirrors in segments of candidates then send them back to the main program in two separate vectors, instead of merely checking « Y » or « N » on one vector segment. *This is possible since this version uses standard data types of fixed sizes both in the GPU and the main.*

The .go program assumes the same structure and logic as the Go versions for parallel cpu threads, but the heavier tasks are on the GPU side.

In the .cu program, the __global__ void DIVISORS() function is the parallel function that will be executed by each thread.

The .cu program is called by the Initz function in the Go program, on the C.theLoopgpu() line. This instruction has to strictly match the declaration between comments /* */ on top of the main program just before the

import « C » line… and also match the function declaration in the GPU part within extern « C » brackets, that activates the GPU by calling the DIVISORS function.

As the GPU part in the .cu program is in C and will be activated by a Go program, one has to use C data types in the Go program for all travelling data. This is allowed under the import « C » on top of the Go program.

Note the two vector data V1 and V2 in the Go program that are filled by each GPU activation on a segment of candidate numbers.

Now, to have the two part compiled, built and run : two command lines in the Terminal in the directory where the two parts are installed.

nvcc --gpu-code=sm_50 --gpu-architecture=compute_50 --ptxas-options=-v --compiler-options -fPIC -o libloopgpul.so --shared Loopgpul.cu

LD_LIBRARY_PATH=${PWD}${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}} go run Loopingul.go

The first command line calls the nvcc compiler of the Nvidia toolkit to produce a .so file. *The « sm_50 » and « compute_50 » parameter values are for a Maxwell GTX 750 GPU*. Different values must be written here for other GPUs, otherwise there is a risk of malfunction, at least a risk of degraded performance (we experimented this latter one). You will have to search into the Nvidia cuda documentation and check the compatibility with your own GPU, with the version of your installed driver and with the version of your cuda toolkit… and yes, it finally works.

The second command line will call the Go compiler to link the main Go program with the .so file and run.

Alternatively, after the nvcc compilation, one can build a Go exec file, available for repeated runs.

Command line to build a Go exec file
LD_LIBRARY_PATH=${PWD}${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}} go build Loopingul.go
Command line to run the Go exec file
LD_LIBRARY_PATH=${PWD}${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}} ./Loopingul

Go BigInt & cuda

Loopingulb.go is the main, ca 210 Instructions
Loopgulb.cu is the GPU calling and runnig part, ca 210 instructions

The ways to communicate between the .cu program and the .go program and have them cooperate are the same ones as in the Go baseline version. But, in this version, there is only one vector to be filled by the GPU activities : the Yes/No vector as described in the Introduction, so that the Collect() function in the Go program has more work to do than in the Go baseline version.

The parameters of the __global__ DIVISORS kernel function are more precisely defined in a view to let the compiler optimize the produced code. So, some parameters are declared as constants and the array of digits for N is declared __restrict__…. All of this, according to our experience, for no measurable benefit.

Note that the BIGSIZE constant is declared in both parts, and it obviously must be the same value on both sides.

BIGSIZE defines the accepted range of big numbers i.e. the maximum number of decimal digits.
In the C developed library for big numbers arithmetics, the big numbers are stored as arrays of characters, each character a decimal digit (NOT the character for display of a numeric number : the numerical digit value itself). Digits are stored in reverse order, i.e. unit in the 0 index.

The .cu pprogram is relatively a big one because it homes our C developed arithmetic library for big numbers. Each arithmetic operation is implemented as a __device__ function so that it can be called by each GPU thread.

The .go program uses the BigInt type of Go (math/big package), but has to convert BigInts to arrays of chars according to our C developed library for big numbers before calling the .cu program. And this conversion is an opportunity for some easy divisibility testing of N, the results of which will be sent to the .cu program.

The « magic » two command lines are similar to the ones for the Go baseline version, with the same comments, again to be entered in the Terminal and from the directory where the files are installed :

nvcc --gpu-code=sm_50 --gpu-architecture=compute_50 --ptxas-options=-v --compiler-options -fPIC -o libloopgpulb.so --shared Loopgpulb.cu

LD_LIBRARY_PATH=${PWD}${LD_ LIBRARY_PATH:+:${LD_LIBRARY_PATH}} go run Loopingulb.go

Please note that *the sm_50 and compute_50 parameter values are for a Maxwell GTX 750 GPU*. Different values must be written for other GPUs.

C++ & cuda

Loopcpp.cpp is the main, ca 500 instructions
Loopcppgpu.cu is the GPU calling and running part, ca 220 instructions
Loopcppdims.h is a text file of common declarations

In spite of apparences, this version is basically a mere « translation » of the Go BigInt & cuda version.

There are some differences :
- our C developed library for big numbers arithmetics is now on both sides
- in the .cpp program, a local (unparallel) first round of search for divisors is done in the 2..10 segment of candidates by calling two specific functions, quick10() and Littlemod()
- the « Yes/No » vector is now local to the .cu program, where it is declared as __device__ __managed data
- the root function Gpucc in the .cu program, when the GPU has finished its parallel tasks, reads the local « Y/N » vector to fill a much more compact vector of the « Y » indexes, that it eventually returns to the .cpp program

This last feature has no important positive consequence on execution time, no negative one either, it brings some « neatness » in the code.

Just for fun, in the .cpp program, the multiplication operations in our C library for big numbers have been replaced by direct accesses to a 10*10 multiplication table. As we operate on decimal digits, this allows reducing to zero the theoretical « complexity level » of our library… much better than any mathematician designed algorithm !

Only one line in the Terminal will produce an exec file :
nvcc --gpu-code=sm_50 --gpu-architecture=compute_50 Loopcppgpu.cu Loopcpp.cpp -o loop

Then :
./loop

Please note that *the sm_50 and compute_50 parameter values are for a Maxwell GTX 750 GPU*. Different values must be written for other GPUs.

Part 4. Measurements, pro and against expectations

The head titles in graphical reports start with the hardware configuration name MX, see Part 6 for details.

Hyperbolic !

Let us start with a few reports (cpu parallelism) on the M3 machine with our Python program.

In our Python program, the stop watch starts just before launching the parallel loops and stops exactly when all parallel loops are ended, just before the collection of results by the main program.

NB. In our programs in other langages, the time for terminal collection will also be included. We will see that this difference has no visible impact on individual measures, nor on the shape of the curves..

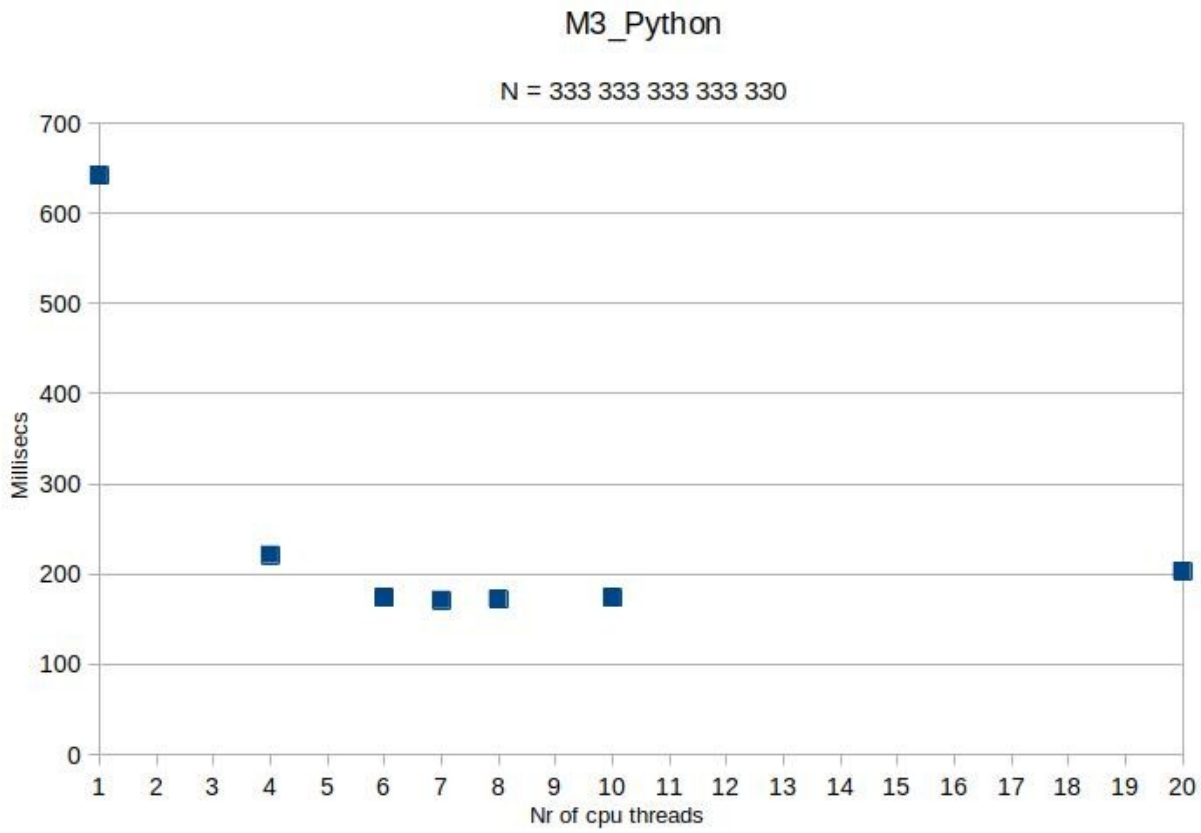

Figure 4.1

The N in Figure 4.1 has 128 divisors.

Do you notice the fast decreasing execution time, roughly till half of available cpu power is busy, then the almost flat curve, with a not so clearly decreasing end ?

Let us try to explain, on another figure...

The N in Figure 4.2 has 192 divisors.

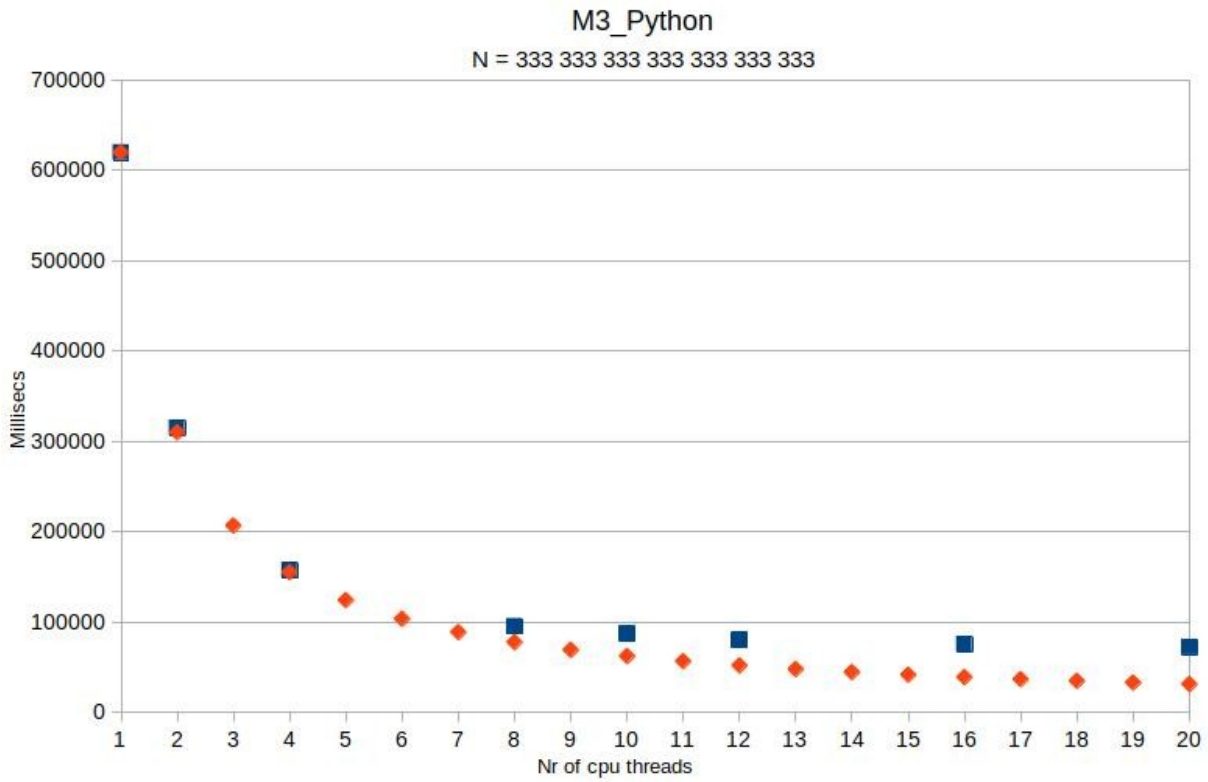

Figure 4.2

A rough math model of execution time T vs nthreads and ndivisors (nr of found divisors) or our Python program would be : T = a + T0 / nthreads + b x nthreads + (c x ndivisors) / nthreads

Trying to estimate the a,b,c constants (?) would not be easy. There is a leading fact against such an effort : the total execution time T is always the execution time of the slowest thread and we cannot assume the same number of found divisors in each thread….

So, let us keep it simple stupid and stick to the raw hyperbolic model.

The orange dots in Figure 4,2 are computed ones, according to a raw hyperbolic model :
T = T1 / nthreads, with T1=execution time on one single thread.
The orange hyperbol does look as « truth », at least till ca 50 % of the number of threads.

As regards the « not so clearly decreasing » end of the real curves in Figures 4.1 and 4.2, can this « anomaly » be explained by the (b x nthreads) term in our math model ? As our program is written, this term only accounts for the time in the internal Python machinery to execute parallel processes - which stands outside of what a user or a programmer can control. Nevertheless, a look at the « real end of curves» of measured execution times excludes any explaination by a single factor. This will be confirmed by more measurements on other machines, using other languages.

Anyway, the hyperbolic model as best optimistic model is no surprise. And there is a consequence : the marginal benefit of more threads quickly comes down. According to the hyperbolic model, for a given « parallel segmented » program of total execution T on one single thread, the marginal benefit of spreading on one more thread is :
T x [1/n – 1/(n+1)]. This means 8 % from 3 to 4 threads, 5 % from 4 tp 5 threads… 1.4 % from 8 to 9 threads, 1.1 % from 9 to 10 threads...

Interpreted vs compiled

Figure 4.3 shows two curves in Go, a compiled langage on the M2 machine (8 cpu threads, Linux).

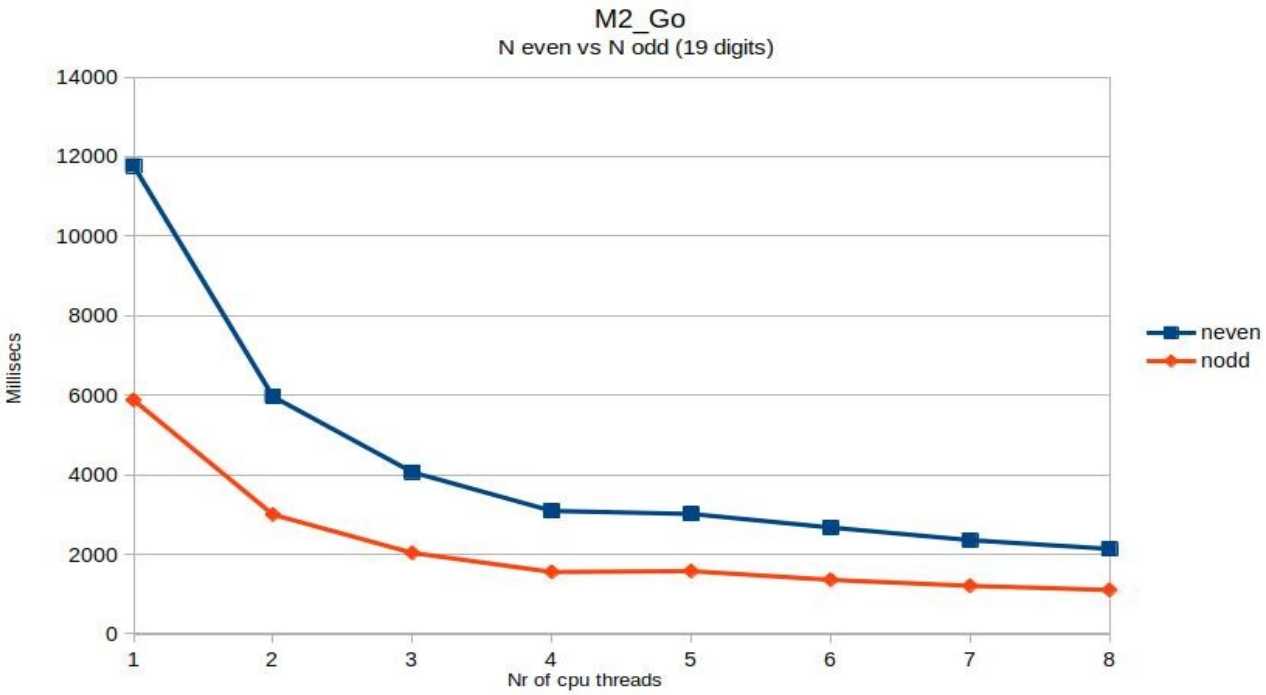

Figure 4.3

Neven = 333 333 333 333 333 333 0 (2,048 divisors) Nodd = 333 333 333 333 333 333 1 (8 divisors)

Notice the same shape of the curves in (compiled) Go as the Python curves. As expected, the odd number requires ca 50 % less time than an even number of same size.

Figures 4.4 and 4.5 show the results of some tests in Go, Gobig and Python.

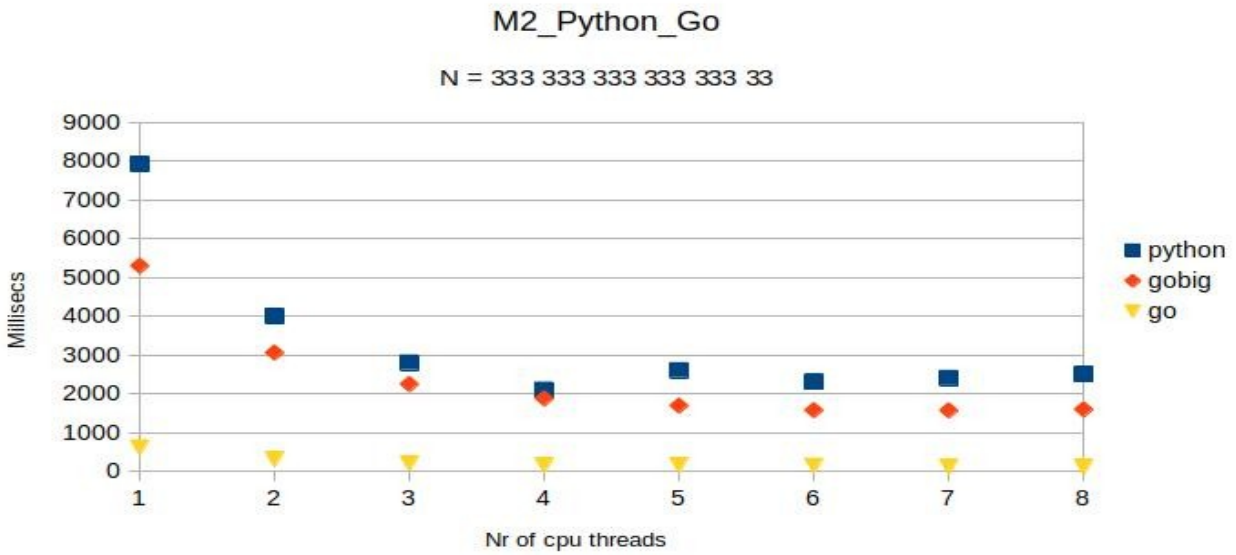

Figure 4.4

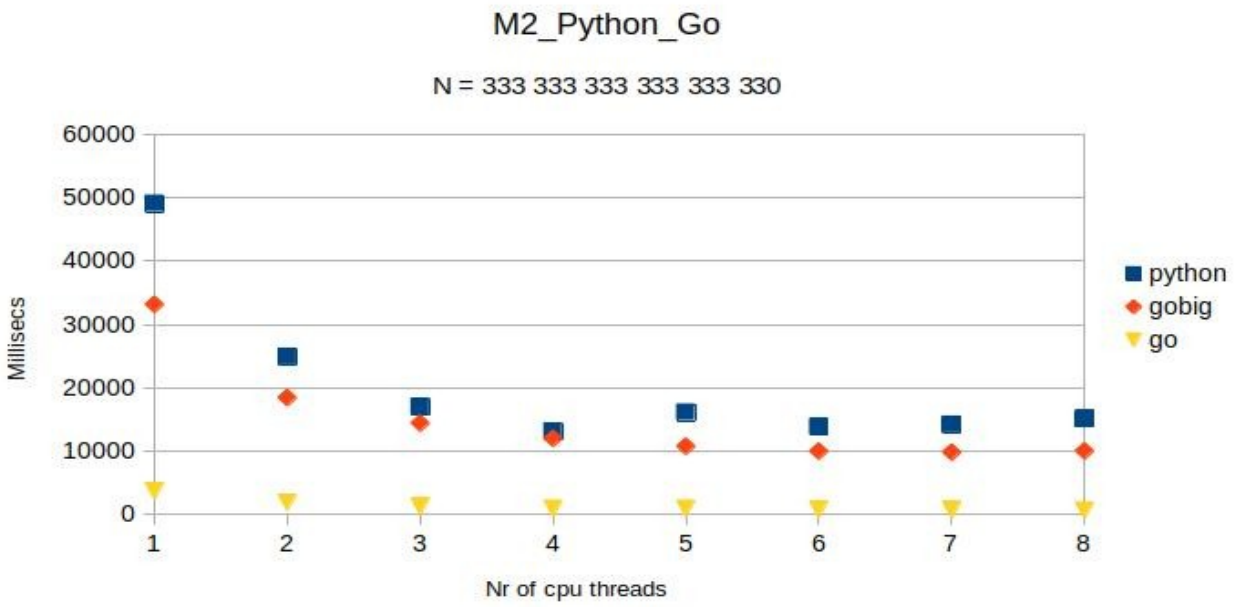

Figure 4.5

The N in Figure 4.4 has 8 divisors.
The N in Figure 4.5 has 32 divisors.

Figures 4.4 and 4.5 show the gap in execution times between Python (an interpreted language) and Go (a compiled program).

Unexpectedly, they show that a Gobig program (a compiled Go program using the math/big package for numbers of any length) performs no better (« not much better », in a more optimistic view) than its equivalent Python program.

Please notice once more the not so flat, not so evenly decreasing curves...

Raw tests

Could we explain some anomalies in the shape of the curves by the imbalance between the tasks of cpu threads, considering that the bigger candidate numbers are allocated to the higher numbered cpu threads ?

Figures 4.6 and 4.7 show real runs *with modified codings of the parallel loops, so that the exact same instructions are executed within each parallel loop independently from the candidate divisors to be tested.*

Figure 4.6 for two Go compiled runs on two different N.

Figure 4.7 for two Python runs on two different N.

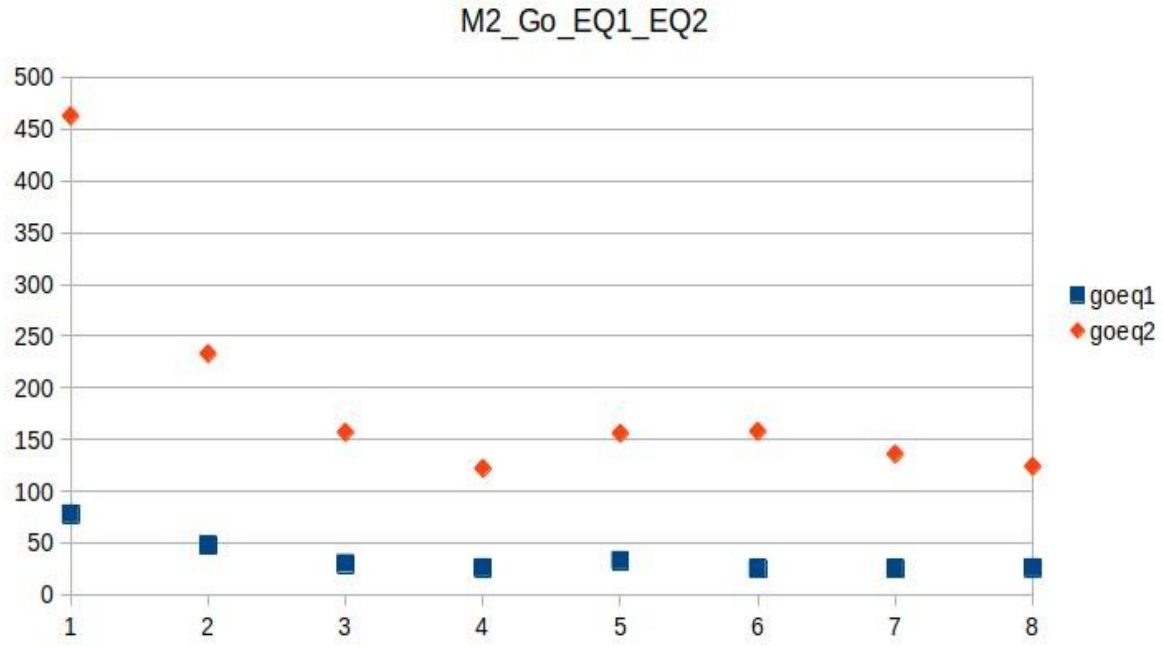

Figure 4.6

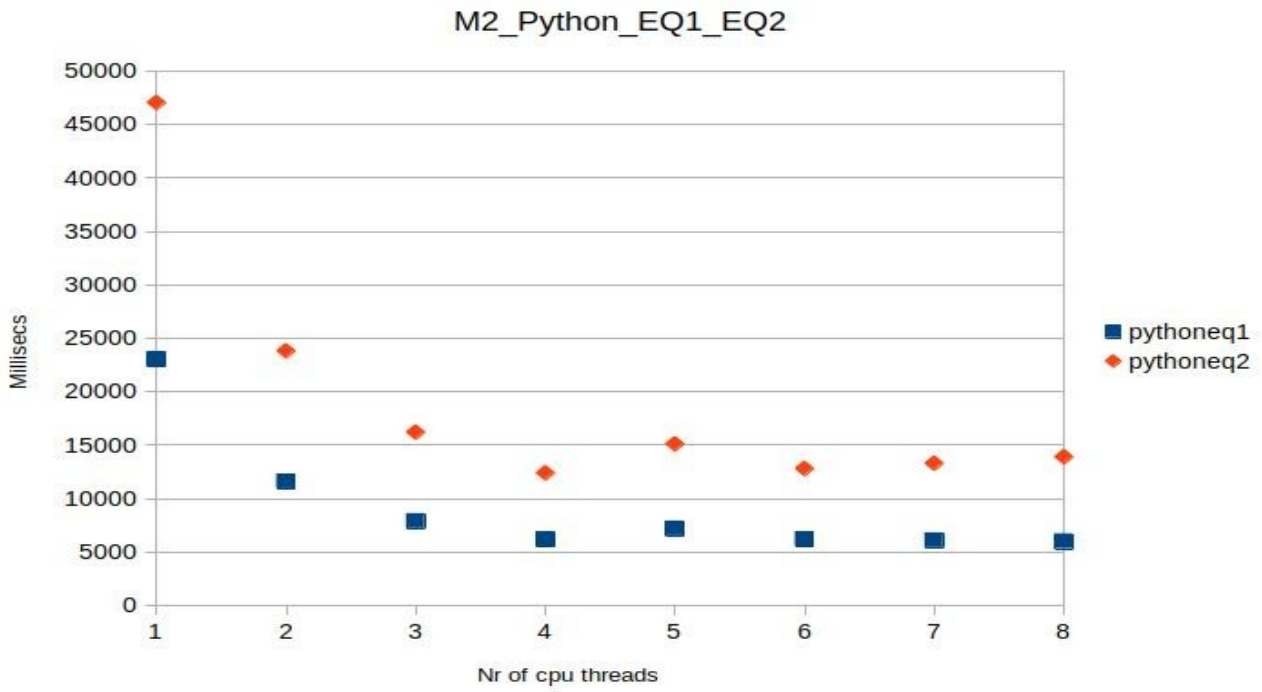

Figure 4.7

We get the same anomalies as the real runs, in Go and Python !!
(and even so if we skip all file read/write instructions that call OS routines in the parallel loop and in time critical functions - with no difference in execution times by the way)

The conclusion is clear : the source of the anomalies is neither the algorithm nor the program logic, it is an external one, where the programmer has no control.

By the way, the two runs in Python in Figure 4.7 show one more interesting fact. The modulo testing instruction in the modified loop of the pythoneq1 was programmed on litteral numbers vs (repeatadly initialized) variables in pythoneq2 : the gap between the curves measures the effect of a local optimization by the Python interpreter on one single instruction in the loop. This confirms the sensitivity of the number of executed instructions in interpreted languages.

Escape to the virtual worlds

In a virtual machine, following the recommendations in the VirtualBox documentation, we will not use more than typically half of cpu threads.

Even so, how much cpu power will be lost vs a real machine ?

Figures 4.8 and 4.9 show a comparison of runs in virtual and real environments.
Both N have 4 divisors.

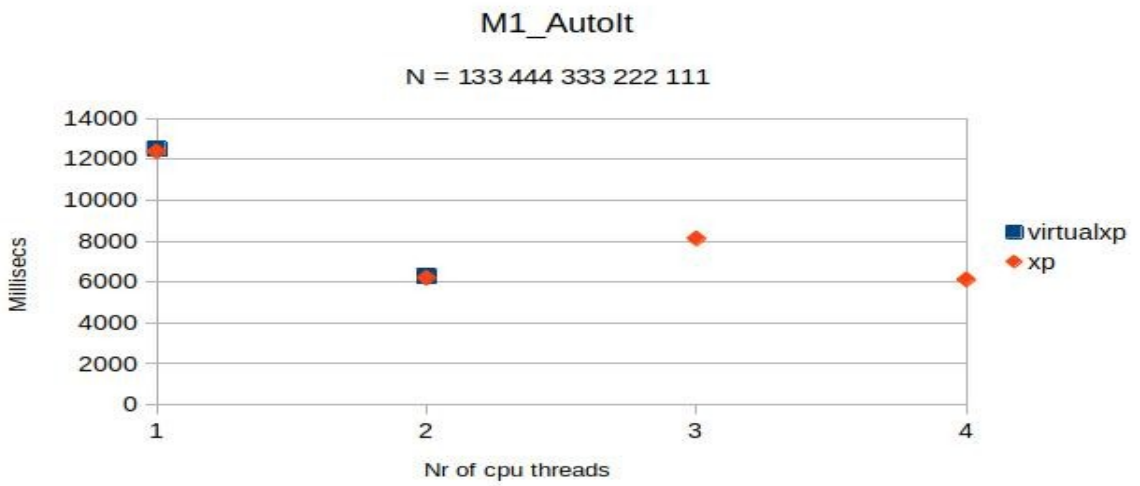

Figure 4.8

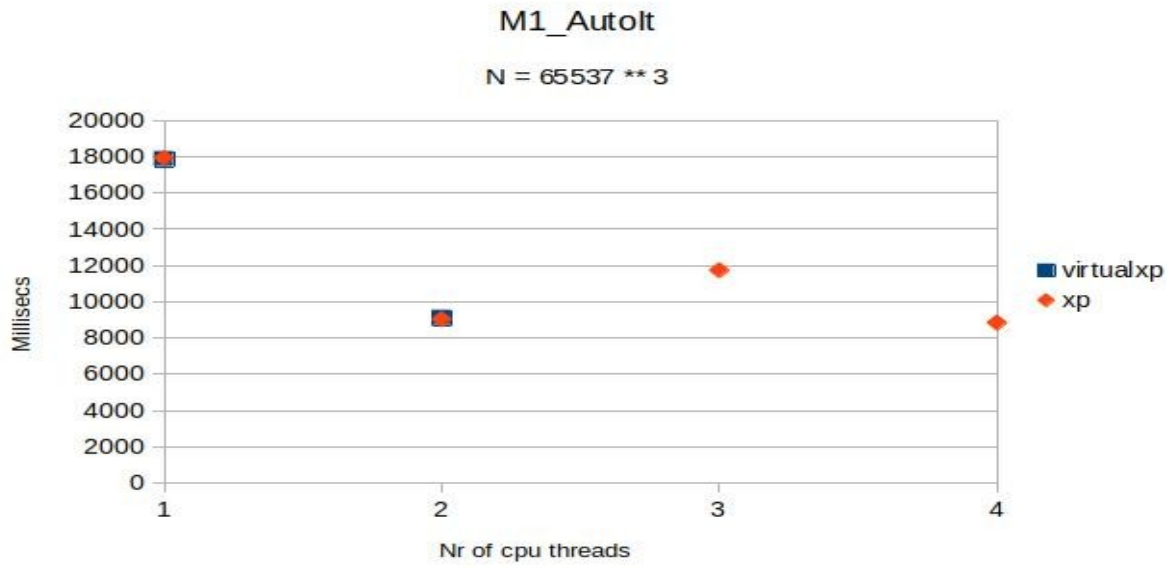

Figure 4.9

Figures 4,8 and 4,9 show an exact equivalence of the real and the virtual machine for the kind of computation in our program.
The M1_AutoIt virtual machine has only 2 cpu threads (2 blue rectangles on Figures 4.8 and 4.9) whereas the real cpu has 4 threads (2 threads per physical cpu core).
NB. The M1 machine has a double boot configuration, the virtual XP machine is installed in the Ubuntu partition under VirtualBox.

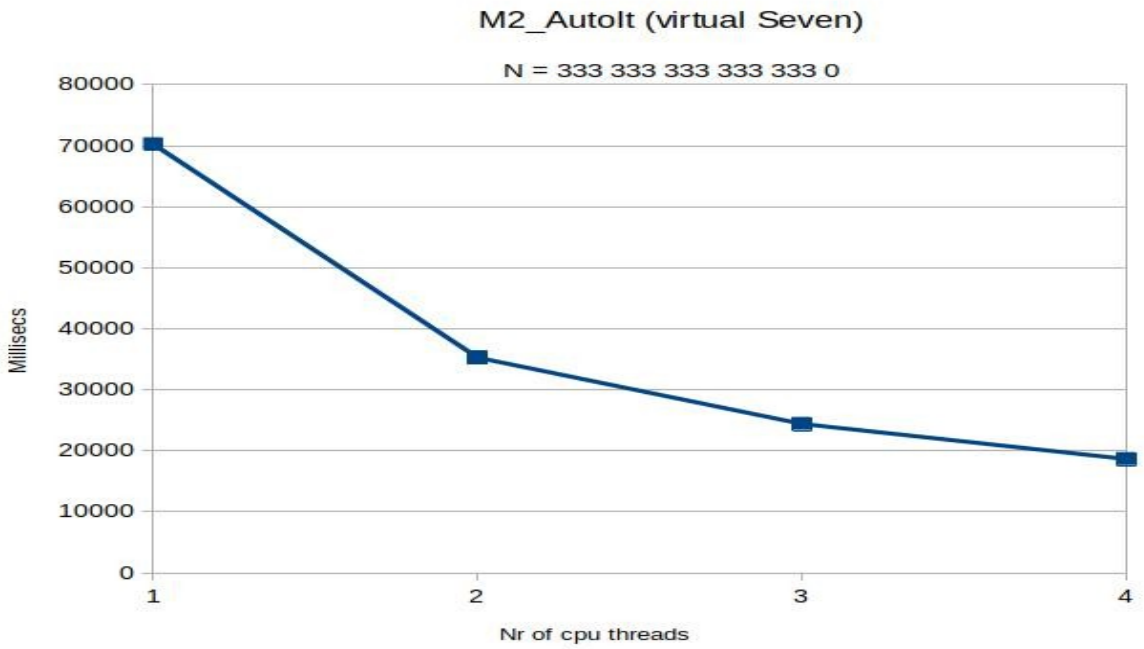

Figure 4.10

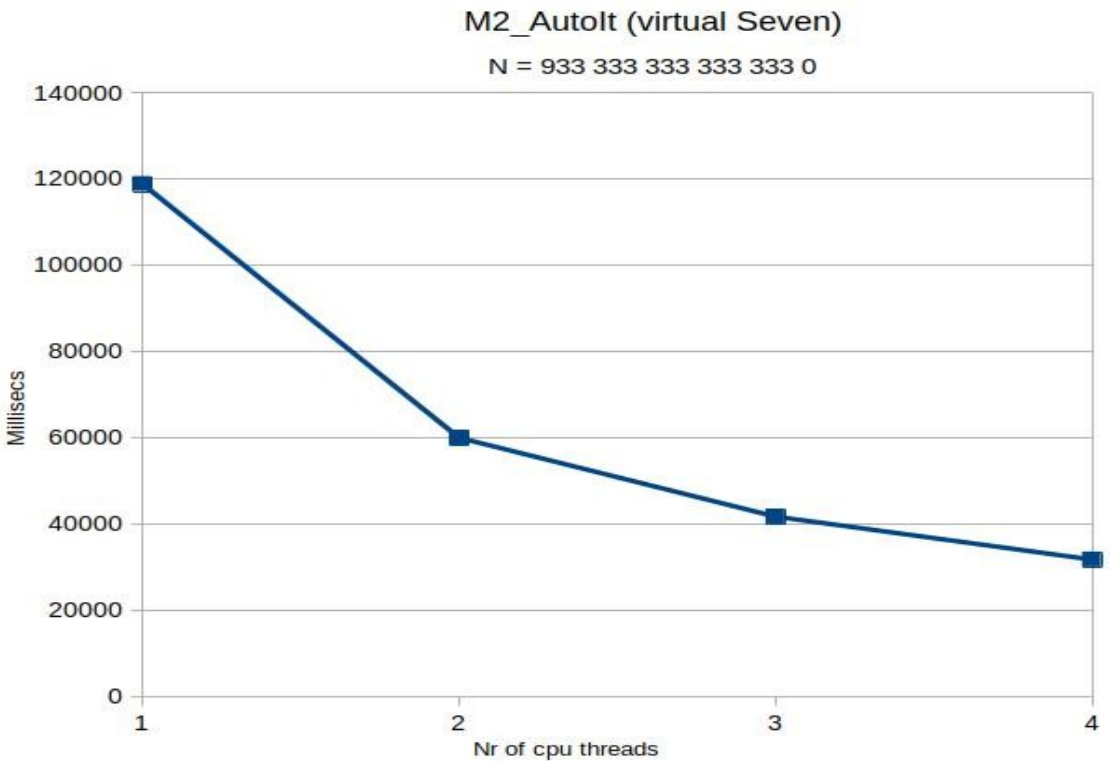

Figure 4.11

Figures 4.10 and 4.11 show examples of runs in a virtual machine on M2. N in figure 4.10 has 384 divisors, N in figure 4.11 has 64 divisors.

As expected, the curves are regularly decreasing, as half of available cpu threads are allocated to the virtual machine. *If we allocate more cpus to the virtual machine, we get exactly the same curves as the real machine, including their anomalies.*

Hidden competition in the real world ?

Let us now have a look at what happens when we run one of our « parallel » programs, in Linux environment and with no other alive application, via the System Monitor utility.

At 100 % cpu power request by our Go program

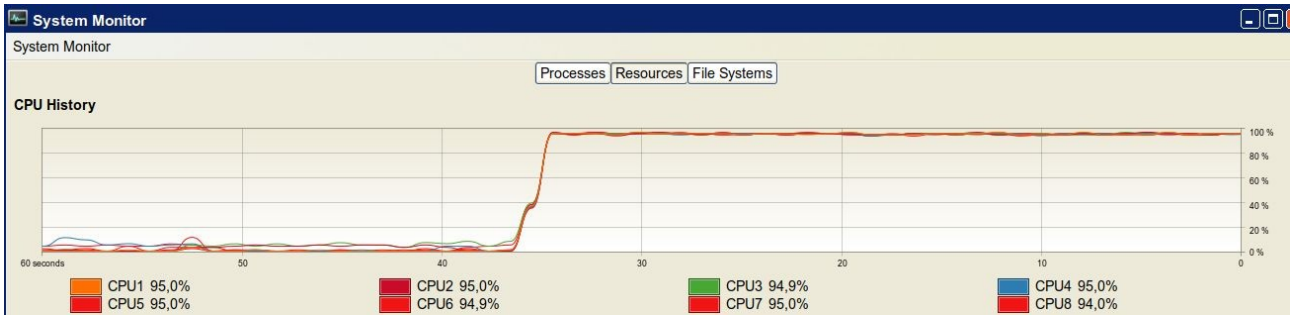

Figure 4.12

The total cpu power request by Go is not 100 %.

At 50 % (ie 100 % request by our program in Go on 4 cpu threads)

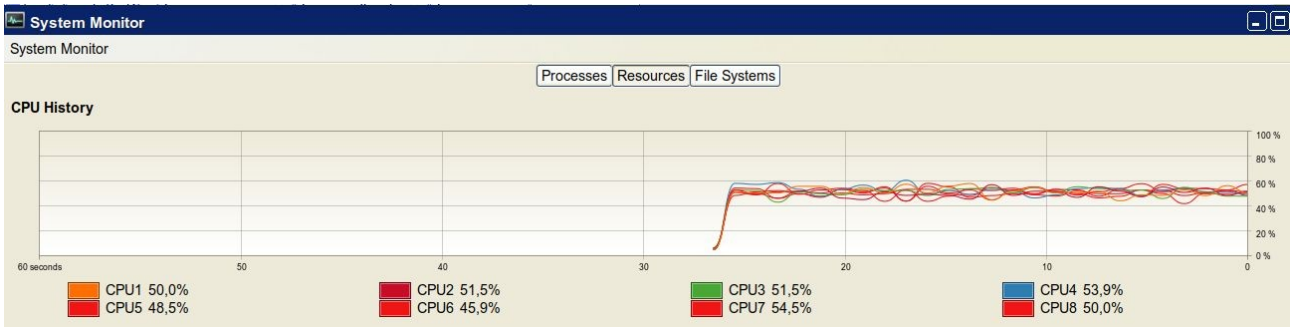

Figure 4.13

The global load is 50 %, but the load is equally spread on the full set of cpu threads

Quite differently : a one cpu thread 100 % request by our AutoIt program in a VirtualBox / Seven

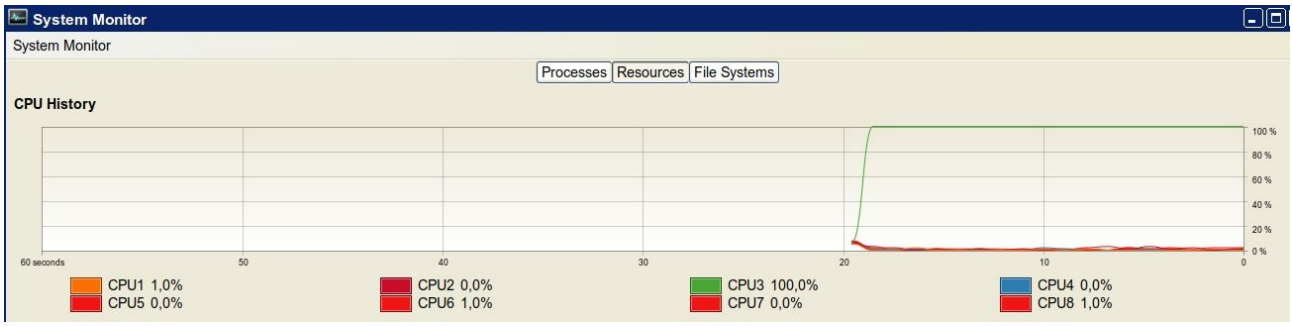

Figure 4.14

100 % request by our Python program

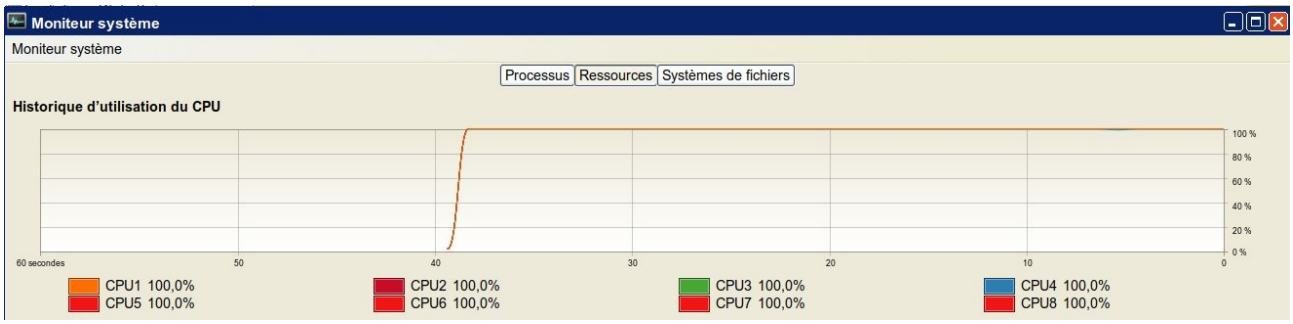

Figure 4.15

50 % request in our Python program

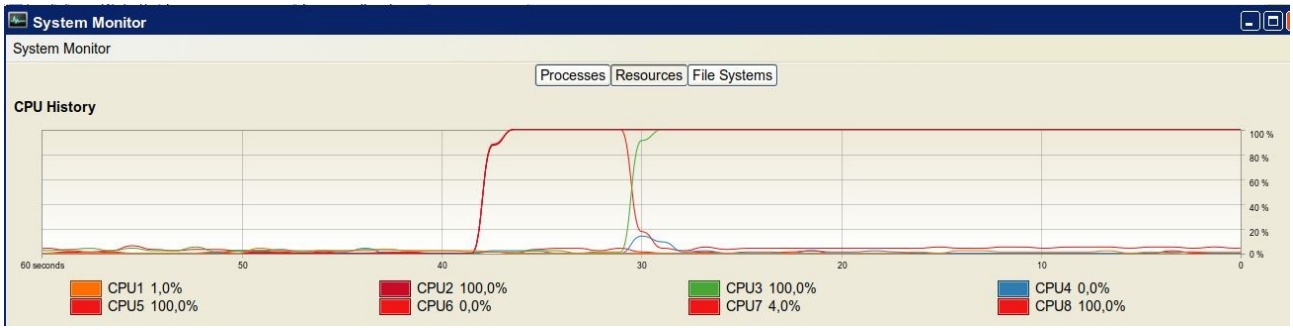

Figure 4.16

Later on...

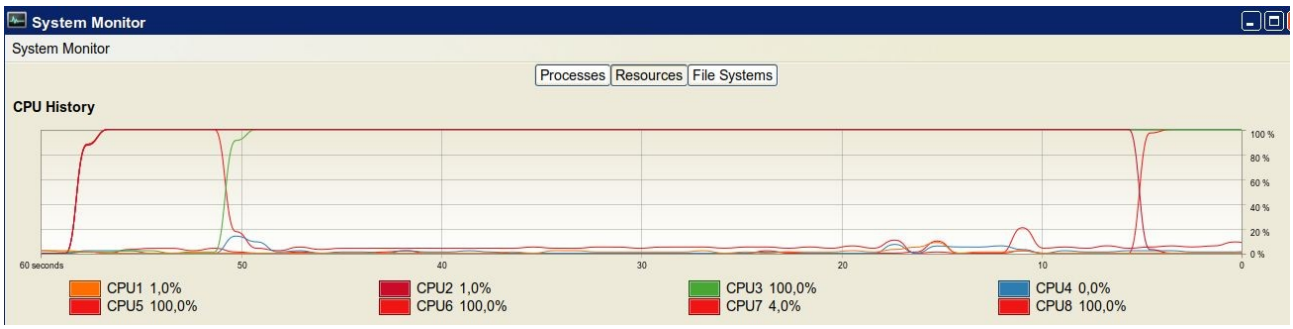

Figure 4.17

During the Python run, the regular load reallocating activity by the OS scheduling routine is obvious. It was also active with our Go program, in less visible ways.

Can this OS activity compete with our program, especially when our program requests more than 50 % of available cpu threads, to such a level that this competition could account for the « not so flat » curves of execution times vs nr of threads ?

There is no obvious answer, but although the Python and Go own internal « parallelizing » machineries are clearly different, our Python and Go curves show the same irregular shape when the number of requested cpu threads is higher than 50 % of the available total.

There is no obvious answer since the display (by the OS utility, Task Manager, System Monitor...) of the individual loads of active processes shows a flat zero load of each systems process... with tiny bursts.

Evening

As there are typically two « physical » threads per cpu core, and we found that there is no great benefit above a global 50 % load, what if we allocated only one thread per physical core ?

Figure 1.2 shows a regular allocation method. Figure 4.18 shows an « even » allocation method.

In a virtual machine, the « evening » experiment showed no difference between the regular request of 2 threads and the modified request of 2 even numbered threads. No surprise.

But on M1 in XP, with our AutoIt program, *sometimes*, the « even » requesting method would produce much worse results than the regular requesting method…. Can this be explained by the use of « old unmaintained software » ? We do not think so. This is one more hint that « something in the background » has to be rearranged by the OS to accomodate the processes created by our program. And we can imagine whyt an « even » requesting method may *sometimes* disturb the OS more than the regular method.

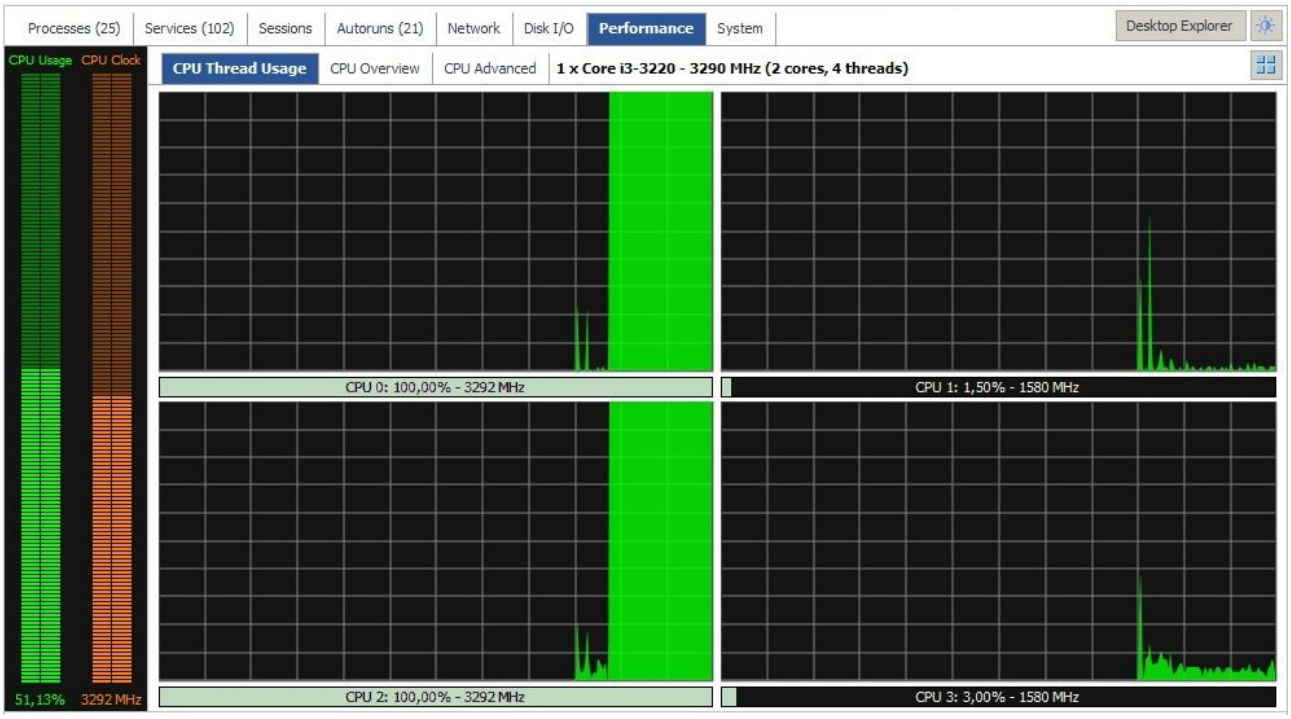

Figure 4.18

Escape to GPU parallelism

How does Go + Cuda perform vs (fully cpu parallel) regular Go ?

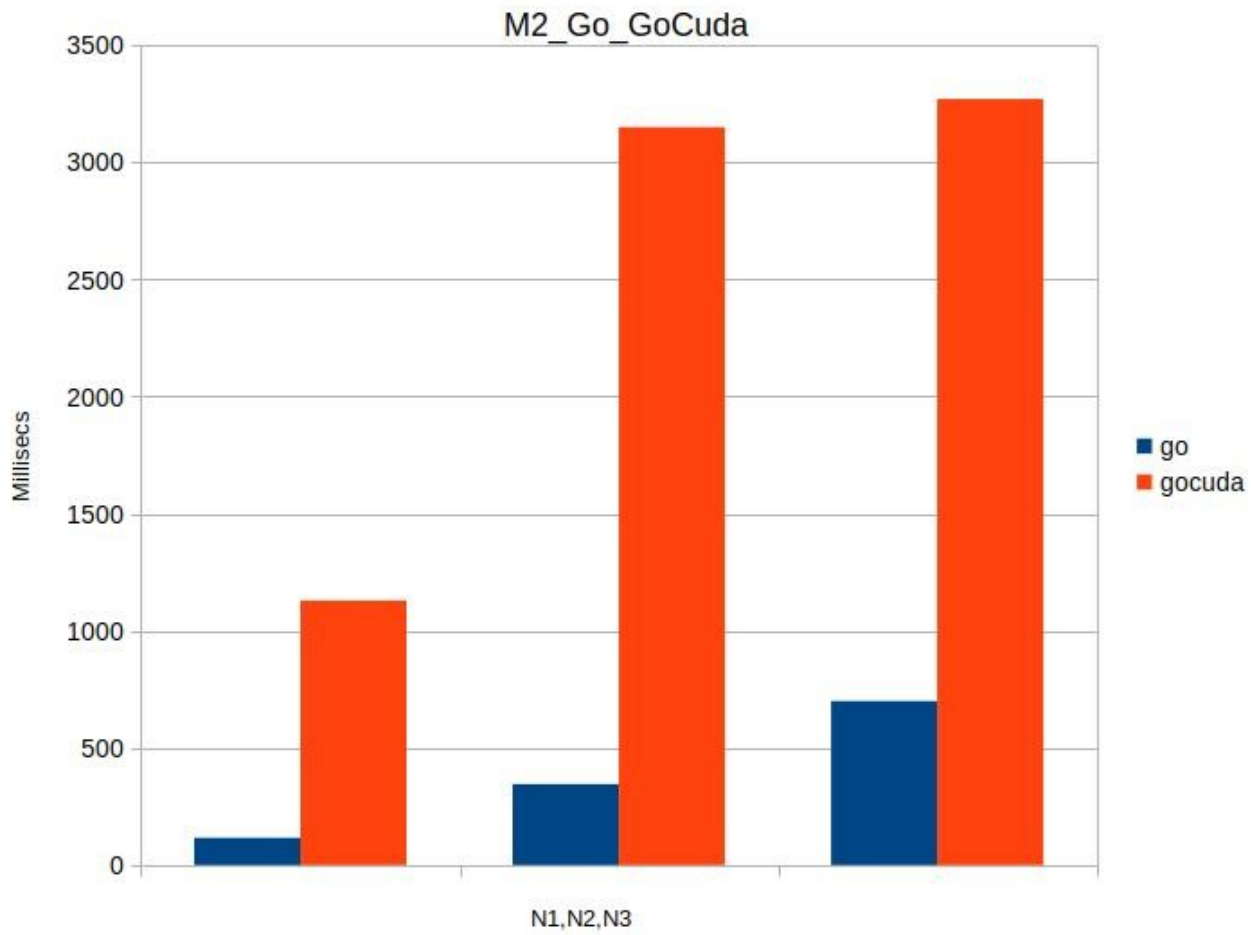

Figure 4.19

For the record,
N1 = 333 333 333 333 333 33 (17 digits, odd, divisible by 3) (2786 calls to GPU)
N2 = 333 333 333 333 333 331 (18 digits, prime) (8810 calls to GPU)
N3 = 333 333 333 333 333 330 (18 digits, even, divisible by 3 and 5) (8810 calls to GPU)

There is a clear conclusion : our problem and its programmed solution are no good candidates for cuda parallelism.

Now, what about Gobig and C++ ?

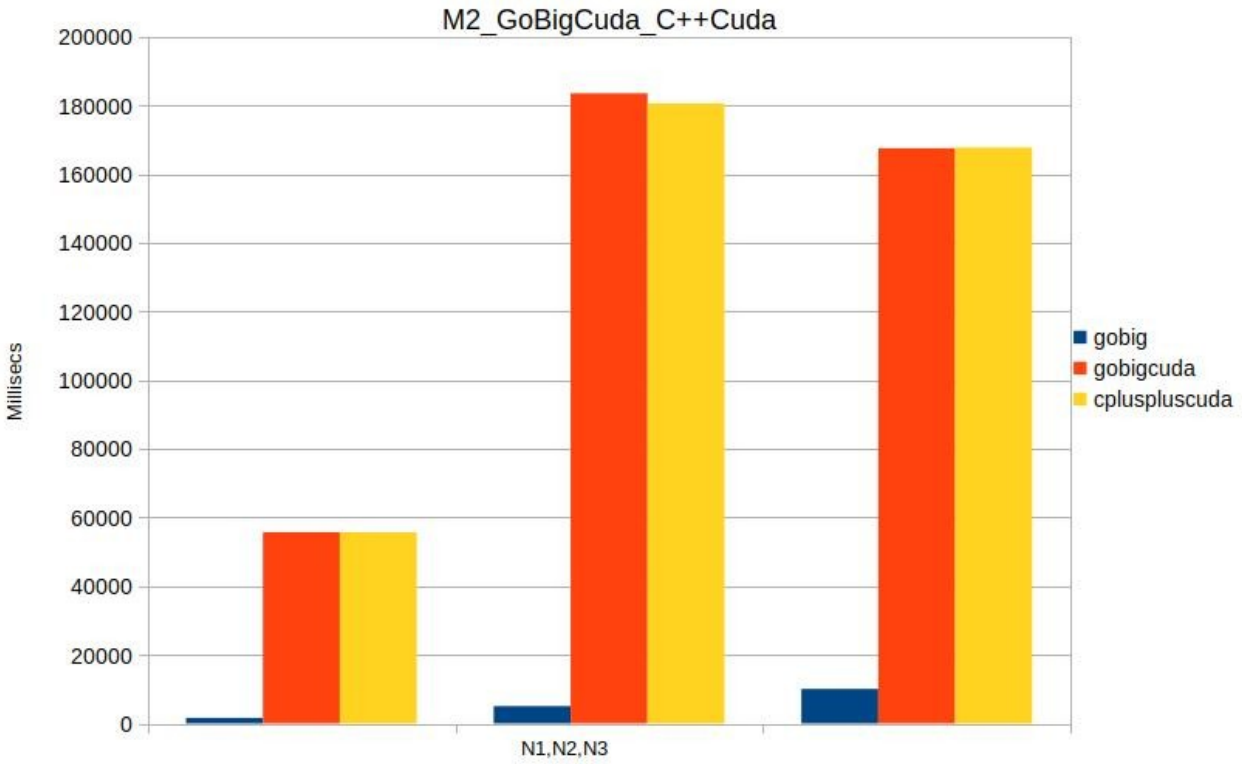

Figure 4.20

On the positive side, our C++ cuda version performs as well as the GoBig cuda version, which somewhat « validates » our own programmed arithmetic library (although this same library is also in the cuda part of the GoBig version) vs the math/big Go package.

On the negative side, the nature of our problem and/or our programmed solutions are clearly no good examples for fast cuda parallelism. In other words, the relatively little computing power (compared to the cpu) in each GPU thread cannot, in our problem as it is programmed, be compensated by a high number of parallel threads.

Different values of the vector segment (nr of GPU threads) and the GPU block size were tested.
Increasing the vector size up to 64k had no effect although allowing a less big number of calls to GPU (2203 instead of 8810). Decreasing the block size from 256 to 128 had a positive but very small effect.

We noticed 100 % on one cpu thread during GPU runs, even while the main program was waiting for the end of GPU runs. This is no surprise, but suggests one question : is there an easy (documented) way to put the main program to sleep when the GPU runs, then have it waken up after each GPU run – or nobody cares ?

For our problem, parallel Python or GoBig will be much faster and probably more energy efficient than GPU parallelism... considering that our parallel programs can be used by loading only 50 % of cpu power without major increase in execution time and without visible impact on the regular uses of the machine. Nevertheless, for very very big numbers, GPU parallelism, even with our « bad » programs, would bring one benefit : execution time is high but may be less dependent on numbers sizes than cpu parallelism - this would have to be confirmed by more experiments.

Part 5. Conclusions and unanswered questions

Specific conclusions within the frame of our problem, the algorithms and the software programs, have already been drawn in Part 4.

General conclusions can only be extrapolations of these specific conclusions based upon a limited number of « facts ». But, put aside with unanswered questions, they may be considered as openings to new horizons.

Let us sum them up versus the « Expected results according to common sense » in the introductory part.

5.1 Programming for parallelism is a difficult discipline

Programming for true (cpu) parallelism is not by itself a difficult discipline, once you designed the structure of your program with its clearly defined « parallel » part.

Starting the software development with more and more detailed prototypes is advisable.

The most challenging difficulty is elsewhere : documentations of programming languages, packages… are generally terrible, sometimes utterly confusing.

5.2 Execution time of a parallel program decreases with the number of cpu threads

Not so much ! For a given « parallel segmented » program, when increasing the number of allocated theads (with at least 4 threads), do not expect substantial benefit (in total execution time) beyond 50 % global cpu load.

You cannot get free from a quasi hyperbolic curve model : for a given « parallel segmented » program, the marginal benefit of the addition of one more thread is under 2 % after 7 threads, under 1 % after 10 threads... even if you could get free from the OS and directly program your cpu on a naked machine.

5.3 Compiled software is always faster than its equivalent interpreted sofware

True with a good compiler and as long as you do not have to require special packages for data that cannot be directly operated by the regular set of instructions in the language.

False otherwise.

Moreover, for true parallelism, compiled and interpreted langages have to use some internal machinery to spread the processes among the cpu threads and control them. This machinery may be a minimal one for a compiled language but it does exist with the same purposes as the one of an interpreted language.

5.4 Computations in a virtual machine are slower than the real machine

False for purely computational purposes and interactive regular office tasks.
Certainly true for tasks requiring specific features, for example « parallel » features for video display, very fast data exchanges with peripherals.

5.5 Parallelism in GPU beats parallelism in CPU

Our experiments showed that you cannot expect « GPU parallelism beats them all » to be automatically true, in spite of a high number of parallel threads.

Benefits highly depend on the kind of computation.

5.6 With your stupid search algorithm, odd numbers will require 50 % less execution time than even numbers

True. Moreover, adding more intelligent filters in our programs proved of no value or of very littlle value. In the field of numbers that can be directly operated by compiled langages, the benefit was negligible. With interpreted languages, the number of instructions in loops is so critical, that adding « smart » tests had negative effects.

5.7 Modern OSes automatically organize parallelism better than you ever will by programming

This precisely identifies the real problem : the OS, by design, cannot « understand », does not support, even excludes true parallelism by an application program !

By design, the aim of the OS is load balancing in CPU threads and memory sharing for « multitasking ». To the user, this provides the illusion of unlimited parallelism within human sense of simultaneity.

But this has nothing to do with true parallelism. This is simulated parallelism thru generic multitasking, and it badly competes with attempts to realize true parallelism.

Let us stop being hypnotized by the neat OS utilities that display cpu loads and other measurements.
They all display average values in periods of time. Besides, they can be misleading by nature, as they were not meant to represent activities contrary to the aims of the OS.

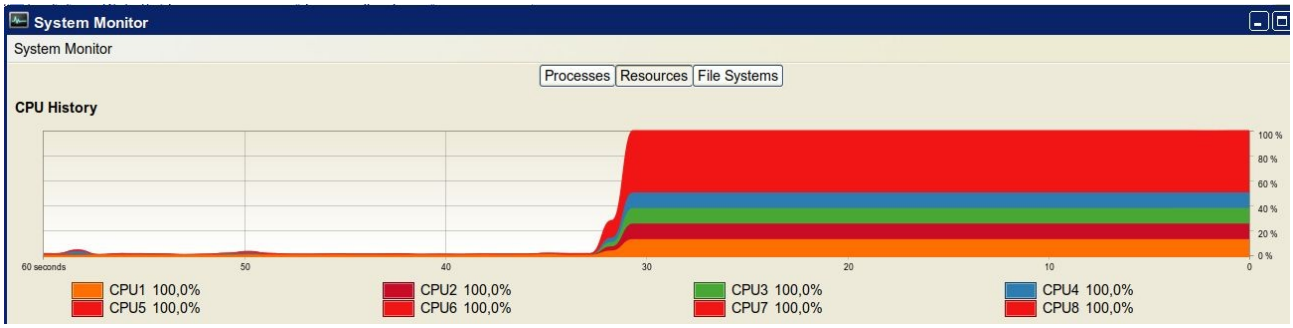

Figure 5.1

For example, Figure 5.1 lets us proudly confirm that we succeeded, with our Python program, in loading our cpu at a global 100 % level... and 100 % level on each cpu thread ! The segments of colors are of same sizes on the vertical axis on the right, but is each one really entirely permanently dedicated to one of our Python parallel processes? This immediate screen snapshot is by itself a proof that the 100 % load is a generous fake, a smoothed representation where micro events cannot be seen by the human eye.

Q/A

Why not use a better programming language ?
Define « better »…
For basic experiments, common programming languages of various kinds seemed appropriate.

Why not use existing libraries or packages with included parallelism features ?
As regards special libraries such as pthread and Mpopen, they are primarily for multithreading, outside of the framework and purposes of our experiments, as stated in the intro paragraph.
Moreover, under the « stick to basics » idea, the study had impicit leading rules : 1/ Preserve visibility of execution flows in the code 2/ Do not delegate the direct control of the level of parallelism (how many cpus are allocated) 3/ Keep the software developments at (one) human size.
Yes, the 3rd rule may have impacted the resulting code efficiency, certainly not the first two rules, that were found good for experimenting and testing.

Why not use GMP (GNU multiple precision arithmetic library) for tests in C++ ?
With no aim to beat any record, why not use a comparatively much simpler library instead ?
Moreover, as the « device » CUDA code (executed by the GPU) has to be in C, any C++ library had to be modified for C compatible data structures. So the simpler lib, the better.
Note that the Go math/big package for multiple precision arithmetic is heavily used in two test versions.

Why not use Go's packages for cuda in your cuda tests ?
Could not find (10/2023) any working example to illustrate the « documentation » of these packages.

My number is much bigger than yours !
Yes, you can display a monstruous integer number in the Python console in less than 1 s by entering a single instruction such as (33333**128) * (555555**231) and you get 1,906 digits to contemplate.
Thanks to a well programmed Karatsuba algorithm for arithmetic mulitplication in the Python machine.
So, what ?
Why are the computing software libraries, especially in the engineering fields, still using the float format of numbers instead of a big integer format that would allow faster arithmetics ? No, the weight of the past is not the only reason, please think why were « floating » formats created...

Crypto mining is true parallel processing in the GPU !
When will the crypto mining « feat » eventually be seen as one of the most stupid, badly designed, energy hungry fashion - higher slavery for « freedom » ?
Parallel processing in GPUs has been in use for much more years than crypto mining. It started with codec routines in media players and media processing applications. For example, when running VLC with appropriate parameters to display a video, the cpu load goes down to almost nothing while the GPU works.
And many video games and 3D applications...

Amateur, rediscovering the Amdahl's law !
We have chosen a humble programming tutorial path that apparently had not been walked upon since a long time, leaving many unanswered questions.
Our findings confirm the quasi-hyperbolic decrease of benefit in total execution time when increasing the number of cpus. The so-called « Amdahi's law » should not be quoted as a kind of conjecture that could be discussed or contested : the hyperbolic model, as best optimistic model, is a natural fact.
Why is the gap so large, in computer techniques, between knowledgeable people and experts ? Could there be a side effect of « social » networking, that would account for the experts regularly despising attitudes, just to make feel their superior levels of « expertise» in front of supposed pretenders ?

Nuts, I have all my computations done in the cloud
Some dinosaurs get bigger and bigger these days. Looks as if their small-sized brothers lived in our hands to distract our minds so that they (our minds) get eaten by the bigger dinosaurs

Outdated, outperformed : no neural computing, no quantum computing !
Why should « innovation » mean more energy wastes, more simulations in the cloud, more talking in the media, more competing for funds ? Besides, in the framework of our study, a simple deterministic algorithm was found to be the right way to compare parallel strategies by using « elementary » languages.

Who cares anyway ?
This question kills any project that does not fit in current themes, trends and « shared beliefs » in the media.
When the computer software industry is mindlessly rushing on the same one-way track for more virtuality & power, without care for energy wastes and human addictions, maybe more of us should care.
Too many contributions in the computing fields are based upon common opinions rather than experience. This is no indication of good health in « computer science ».

Two really daring questions :

- Are the yearly increasing sizes of modern OSes really for more comfort, more power to the user, or are these OSes no longer « in progress », victims of some kind of bureaucratic syndrom (ah, yes, « security » and AI fashion...) , getting more and more unmaintainable, more and more power hungry, undefinitely spiralling towards wholesale perfection ?

- What could be the architecture and the user manual of a simple, efficient Operating System, designed for multicore cpus, current memory sizes and fast storage means, based upon a common understanding of « user services » (meaning real user tasks such as word processing, media playing, web surfing), instead of heavily packaged layers of software mechanics based upon obsolete abstractions ?

Part 6. Documentation

All programs sources are attached in a zip file

Our PC desktop configurations

Hardware
[M1]. Intel Core i3 3220 (2 cores, 2 threads/core) 3.3 Ghz, 55 W + 8 GB DDR3
[M2]. Intel Core i3 10100 (4 cores, 2 threads/core) 4.3 Ghz, 65 W + 8 GB DDR4
[M3]. Intel Core i5 13600K (6 cores, 2 threads/core, plus 8 cores) 4.4 Ghz, 145 W + 32 GB DDR5
GPU on [M1] and [M2]: Nvidia GTX 750

Oses
- Linux 64 bit (Ubuntu Bionic)
- MS Windows : from XP to 11

Bibliography and utilities for programming

The Art Of Computer Programming, Vol 2, by Donald Knuth (Addison Wesley)
4.3 Multiple Precision Arithmetic (for editions before the 3rd : search the errata in D.Knuth's web site to find how the algorithm for division was revised in 1997)

CUDA no-stress programming
https://hackernoon.com/no-stress-cuda-programming-using-go-and-c-fy1y3agf
https://github.com/cleuton/golang-network/tree/master/english/cuda/nostress?ref=hackernoon.com
(example of Go – C - Cuda bridge, full code plus compile/link commands, 2019)

BigInt library in C++ (full code)
https://www.geeksforgeeks.org/bigint-big-integers-in-c-with-example/

Stackoverflow forums
https://stackoverflow.com/
(search engines directly give links to interesting Q/A forum threads)

NVIDIA's CUDA programming guide
https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

AUTOIT scripting langage (MS Windows)
https://www.autoitscript.com/site/autoit/

StartAffinity utility (MS Windows)
http://www.adsciengineering.com/StartAffinity/
(for tests with AutoIt, only useful for XP)

Launch 1.6 utility (MS Windows)
https://www.heise.de/download/product/launch-45468
(same function as StartAffinity with different parameters)

Useful freeware utilities

Mitec Task Manager Deluxe (MS Windows)
http://www.mitec.cz/tmx.html
(The Performance tab shows much more than the regular Task Manager)

ColorConsole (MS Windows)
http://www.softwareok.com/Download/ColorConsole_Portable.zip
(cut/paste feature, colors management... as Linux Terminals utilities)

Bill2's Process Manager (MS Windows)
https://www.bill2-software.com/processmanager/exe/BPM-Portable.zip
(compact, powerful, but help doc only in French language)