Why is reading lines from stdin much slower in C++ than Python?

I wanted to compare reading lines of string input from stdin using Python and C++ and was shocked to see my C++ code run an order of magnitude slower than the equivalent Python code. Since my C++ is rusty and I'm not yet an expert Pythonista, please tell me if I'm doing something wrong or if I'm misunderstanding something.

(tl;dr answer: include the statement: cin.sync_with_stdio(false) or just use fgets instead.

tl;dr results: scroll all the way down to the bottom of my question and look at the table.)

C++ code:

```
#include <iostream>
#include <time.h>

using namespace std;

int main() {
    string input_line;
    long line_count = 0;
    time_t start = time(NULL);
    int sec;
    int lps;

    while (cin) {
        getline(cin, input_line);
        if (!cin.eof())
            line_count++;
    };

    sec = (int) time(NULL) - start;
    cerr << "Read " << line_count << " lines in " << sec << " seconds." ;
    if (sec > 0) {
        lps = line_count / sec;
        cerr << " LPS: " << lps << endl;
    } else
        cerr << endl;
    return 0;
}

//Compiled with:
//g++ -O3 -o readline_test_cpp foo.cpp
```

Python Equivalent:

```
#!/usr/bin/env python
import time
import sys

count = 0
start = time.time()

for line in  sys.stdin:
    count += 1

delta_sec = int(time.time() - start_time)
if delta_sec >= 0:
    lines_per_sec = int(round(count/delta_sec))
    print("Read {0} lines in {1} seconds. LPS: {2}".format(count, delta_sec,
       lines_per_sec))
```

Here are my results:

```
$ cat test_lines | ./readline_test_cpp
Read 5570000 lines in 9 seconds. LPS: 618889

$cat test_lines | ./readline_test.py
```

```
Read 5570000 lines in 1 seconds. LPS: 5570000
```

Edit: *I should note that I tried this both under OS-X (10.6.8) and Linux 2.6.32 (RHEL 6.2). The former is a macbook pro, the latter is a very beefy server, not that this is too pertinent.*

Edit 2: *(Removed this edit, as no longer applicable)*

```
$ for i in {1..5}; do echo "Test run $i at `date`"; echo -n "CPP:"; cat test_lines |
./readline_test_cpp ; echo -n "Python:"; cat test_lines | ./readline_test.py ; done
Test run 1 at Mon Feb 20 21:29:28 EST 2012
CPP:   Read 5570001 lines in 9 seconds. LPS: 618889
Python:Read 5570000 lines in 1 seconds. LPS: 5570000
Test run 2 at Mon Feb 20 21:29:39 EST 2012
CPP:   Read 5570001 lines in 9 seconds. LPS: 618889
Python:Read 5570000 lines in 1 seconds. LPS: 5570000
Test run 3 at Mon Feb 20 21:29:50 EST 2012
CPP:   Read 5570001 lines in 9 seconds. LPS: 618889
Python:Read 5570000 lines in 1 seconds. LPS: 5570000
Test run 4 at Mon Feb 20 21:30:01 EST 2012
CPP:   Read 5570001 lines in 9 seconds. LPS: 618889
Python:Read 5570000 lines in 1 seconds. LPS: 5570000
Test run 5 at Mon Feb 20 21:30:11 EST 2012
CPP:   Read 5570001 lines in 10 seconds. LPS: 557000
Python:Read 5570000 lines in  1 seconds. LPS: 5570000
```

Edit 3:

Okay, I tried J.N.'s suggestion of trying having python store the line read: but it made no difference to python's speed.

I also tried J.N.'s suggestion of using scanf into a char array instead of getline into a std::string. Bingo! This resulted in equivalent performance for both python and c++. (3,333,333 LPS with my input data, which by the way are just short lines of three fields each, usually about 20 chars wide, though sometimes more).

Code:

```
char input_a[512];
char input_b[32];
char input_c[512];
while(scanf("%s %s %s\n", input_a, input_b, input_c) != EOF) {
    line_count++;
};
```

Speed:

```
$ cat test_lines | ./readline_test_cpp2
Read 10000000 lines in 3 seconds. LPS: 3333333
$ cat test_lines | ./readline_test2.py
Read 10000000 lines in 3 seconds. LPS: 3333333
```

(Yes, I ran it several times.) So, I guess I will now use scanf instead of getline. But, I'm still curious if people think this performance hit from std::string/getline is typical and reasonable.

Edit 4 (was: Final Edit / Solution):

Adding: cin.sync_with_stdio(false);

Immediately above my original while loop above results in code that runs faster than Python.

New performance comparison (this is on my 2011 Macbook Pro), using the original code, the original with the sync disabled, and the original python, respectively, on a file with 20M lines of text. Yes, I ran it several times to eliminate disk caching confound.

```
$ /usr/bin/time cat test_lines_double | ./readline_test_cpp
       33.30 real         0.04 user         0.74 sys
Read 20000001 lines in 33 seconds. LPS: 606060
$ /usr/bin/time cat test_lines_double | ./readline_test_cpp1b
        3.79 real         0.01 user         0.50 sys
Read 20000000 lines in 4 seconds. LPS: 5000000
$ /usr/bin/time cat test_lines_double | ./readline_test.py
        6.88 real         0.01 user         0.38 sys
Read 20000000 lines in 6 seconds. LPS: 3333333
```

Thanks to @Vaughn Cato for his answer! ***Any elaboration people can make or good references people can point to as to why this sync happens, what it means, when it's useful, and when it's okay to disable would be greatly appreciated by posterity.*** :-)

Edit 5 / Better Solution:

As suggested by Gandalf The Gray below, gets is even faster than scanf or the unsynchronized cin approach. I also learned that scanf and gets are both UNSAFE and should NOT BE USED due to potential of buffer overflow. So, I wrote this iteration using fgets, the safer alternative to gets. Here are the pertinent lines for my fellow noobs:

```
char input_line[MAX_LINE];
char *result;
```

```
//<snip>

while((result = fgets(input_line, MAX_LINE, stdin )) != NULL)
    line_count++;
if (ferror(stdin))
    perror("Error reading stdin.");
```

Now, here are the results using an even larger file (100M lines; ~3.4GB) on a fast server with very fast disk, comparing the python, the unsynced cin, and the fgets approaches, as well as comparing with the wc utility. [The scanf version segfaulted and I don't feel like troubleshooting it.]:

```
$ /usr/bin/time cat temp_big_file | readline_test.py
0.03user 2.04system 0:28.06elapsed 7%CPU (0avgtext+0avgdata 2464maxresident)k
0inputs+0outputs (0major+182minor)pagefaults 0swaps
Read 100000000 lines in 28 seconds. LPS: 3571428

$ /usr/bin/time cat temp_big_file | readline_test_unsync_cin
0.03user 1.64system 0:08.10elapsed 20%CPU (0avgtext+0avgdata 2464maxresident)k
0inputs+0outputs (0major+182minor)pagefaults 0swaps
Read 100000000 lines in 8 seconds. LPS: 12500000

$ /usr/bin/time cat temp_big_file | readline_test_fgets
0.00user 0.93system 0:07.01elapsed 13%CPU (0avgtext+0avgdata 2448maxresident)k
0inputs+0outputs (0major+181minor)pagefaults 0swaps
Read 100000000 lines in 7 seconds. LPS: 14285714

$ /usr/bin/time cat temp_big_file | wc -l
0.01user 1.34system 0:01.83elapsed 74%CPU (0avgtext+0avgdata 2464maxresident)k
0inputs+0outputs (0major+182minor)pagefaults 0swaps
100000000


Recap (lines per second):
python:         3,571,428
cin (no sync): 12,500,000
fgets:         14,285,714
wc:            54,644,808
```

As you can see, fgets is better but still pretty far from wc performance; I'm pretty sure this is due to the fact that wc examines each character without any memory copying. I suspect that, at this point, other parts of the code will become the bottleneck, so I don't think optimizing to that level would even be worthwhile, even if possible (since, after all, I actually need to store the read lines in memory).

Also note that a small tradeoff with using a char * buffer and fgets vs unsynced cin to string is that the latter can read lines of any length, while the former requires limiting input to some finite number. In practice, this is probably a non-issue for reading most line-based input files, as the buffer can be set to a very large value that would not be exceeded by valid input.

This has been educational. Thanks to all for your comments and suggestions.

Edit 6:

As suggested by J.F. Sebastian in the comments below, the GNU wc utility uses plain C read() (within the safe-read.c wrapper) to read chunks (of 16k bytes) at a time and count new lines. Here's a python equivalent based on J.F.'s code (just showing the relevant snippet that replaces the python for loop:

```
BUFFER_SIZE = 16384
count = sum(chunk.count('\n') for chunk in iter(partial(sys.stdin.read, BUFFER_SIZE), ''))
```

The performance of this version is quite fast (though still a bit slower than the raw c wc utility, of course:

```
$ /usr/bin/time cat temp_big_file | readline_test3.py
0.01user 1.16system 0:04.74elapsed 24%CPU (0avgtext+0avgdata 2448maxresident)k
0inputs+0outputs (0major+181minor)pagefaults 0swaps
Read 100000000 lines in 4.7275 seconds. LPS: 21152829
```

Again, it's a bit silly for me to compare C++ fgets/cin and the first python code on the one hand to wc -l and this last python snippet on the other, as the latter two don't actually store the read lines but merely count newlines. Still, it's interesting to explore all the different implementations and think about the performance implications. Thanks again!

Edit 7: Tiny benchmark addendum and recap

For completeness, I thought I'd update the read speed for the same file on the same box with the original (synced) C++ code. Again, this is for a 100M line file on a fast disk. Here's the complete table now:

```
Implementation      Lines per second
python (default)           3,571,428
cin (default/naive)          819,672
cin (no sync)             12,500,000
fgets                     14,285,714
wc (not fair comparison)  54,644,808
```

c++ python benchmarking readline getline

edited Jan 27 at 16:36

community wiki
16 revs, 3 users 98%
JJC

4 Did you run your tests multiple times? Perhaps there is a disk cache issue. – Vaughn Cato Feb 21 '12 at 2:20

2 @JJC : I see two possibilities (assuming you have remove the caching problem suggested by David): 1) `<iostream>` performance sucks. Not the first time it happens. 2) Python is clever enough not to copy the data in the for loop because you don't use it. You could retest trying to use `scanf` and a `char[]`. Alternatively you could try rewriting the loop so that something is done with the string (eg keep the 5th letter and concatenate it in a result). – J.N. Feb 21 '12 at 2:35

2 One more thing, this is the kind of simple test that do not actually reflect the performance of language N vs language M. Python may have a very smart optimization for that precise case you would not get when running a full featured application. – J.N. Feb 21 '12 at 2:39

5 The problem is synchronization with stdio -- see my answer. – Vaughn Cato Feb 21 '12 at 3:30

6 `wc -l` is fast because it reads the stream more than one line at a time (it might be `fread(stdin)/memchr('\n')` combination). Python results are in the same order of magnitude e.g., `wc-l.py` – J.F. Sebastian Feb 27 '12 at 0:21

12 Answers

By default, `cin` is synchronized with stdio, which causes it to avoid any input buffering. If you add this to the top of your main, you should see much better performance:

```
std::ios_base::sync_with_stdio(false);
```

Normally, when an input stream is buffered, instead of reading one character at a time, the stream will be read in larger chunks. This reduces the number of system calls, which are typically relatively expensive. However, since the `FILE*` based `stdio` and `iostreams` often have separate implementations and therefore separate buffers, this could lead to a problem if both were used together. For example:

```
int myvalue1;
cin >> myvalue1;
int myvalue2;
scanf("%d",&myvalue2);
```

If more input was read by `cin` than it actually needed, then the second integer value wouldn't be available for the `scanf` function, which has its own independent buffer. This would lead to unexpected results.

To avoid this, by default, streams are synchronized with `stdio`. One common way to achieve this is to have `cin` read each character one at a time as needed using `stdio` functions. Unfortunately, this introduces a lot of overhead. For small amounts of input, this isn't a big problem, but when you are reading millions of lines, the performance penalty is significant.

Fortunately, the library designers decided that you should also be able to disable this feature to get improved performance if you knew what you were doing, so they provided the `sync_with_stdio` method.

edited Sep 30 at 16:26


Lightness Races in Orbit
180k 30 260 496

answered Feb 21 '12 at 3:24


Vaughn Cato
35.6k 4 26 52

58 This should be at the top. It is almost certainly correct. The answer cannot lie in replacing the read with an `fscanf` call, because that quite simply doesn't do as much work as Python does. Python must allocate memory for the string, possibly multiple times as the existing allocation is deemed inadequate - exactly like the C++ approach with `std::string`. This task is almost certainly I/O bound and there is way too much FUD going around about the cost of creating `std::string` objects in C++ or using `<iostream>` in and of itself. – Karl Knechtel Feb 21 '12 at 3:34

20 Yes, adding this line immediately above my original while loop sped the code up to surpass even python. I'm about to post the results as the final edit. Thanks again! – JJC Feb 21 '12 at 3:45

4 I couldn't find a good reference, but I added some explanation based on my own understanding. – Vaughn Cato Feb 21 '12 at 4:34

3 Yes, this actually applies to cout, cerr, and clog as well. – Vaughn Cato Mar 11 '12 at 13:56

4 Note that `sync_with_stdio()` is a static member function, and a call to this function on any stream object (e.g. `cin`) toggles on or off synchronization for *all* standard iostream objects. – John Zwinck Jan 21 at 1:16

I reproduced the original result on my computer using g++ on a Mac.

Adding the following statements to the C++ version just before the `while` loop brings it inline with the Python version:

```
std::ios_base::sync_with_stdio(false);
char buffer[1048576];
std::cin.rdbuf()->pubsetbuf(buffer, sizeof(buffer));
```

sync_with_stdio improved speed to 2 seconds, and setting a larger buffer brought it down to 1 second.

edited Apr 2 '14 at 18:26 — Peter Mortensen — 9,039 — 10 61 96

answered Feb 21 '12 at 3:33 — karunski — 2,307 — 9 9

1 You might want to try different buffer sizes to get more useful information. I suspect you will see rapidly diminishing returns. – Karl Knechtel Feb 21 '12 at 3:37

4 I was too hasty in my reply; setting the buffer size to something other than the default did not produce an appreciable difference. – karunski Feb 21 '12 at 3:51

41 I would also avoid setting up a 1MB buffer on the stack. It can lead to stackoverflow (though I guess it's a good place to debate about it!) – Matthieu M. Feb 21 '12 at 7:30

4 Matthieu, Mac uses a 8MB process stack by default. Linux uses 4MB per thread default, IIRC. 1MB isn't that much of an issue for a program that transforms input with relatively shallow stack depth. More importantly, though, std::cin will trash the stack if the buffer goes out of scope. – SEK Jan 14 '14 at 9:28

7 @SEK Windows default Stack size is 1MB. – Étienne Mar 15 '14 at 2:11

Just out of curiosity I've taken a look at what happens under the hood, and I've used dtruss/strace on each test.

C++

```
./a.out < in
Saw 6512403 lines in 8 seconds.  Crunch speed: 814050
```

syscalls `sudo dtruss -c ./a.out < in`

```
CALL                                        COUNT
__mac_syscall                                   1
<snip>
open                                            6
pread                                           8
mprotect                                       17
mmap                                           22
stat64                                         30
read_nocancel                               25958
```

Python

```
./a.py < in
Read 6512402 lines in 1 seconds. LPS: 6512402
```

syscalls `sudo dtruss -c ./a.py < in`

```
CALL                                        COUNT
__mac_syscall                                   1
<snip>
open                                            5
pread                                           8
mprotect                                       17
mmap                                           21
stat64                                         29
```

edited Apr 2 '14 at 18:27 — answered Mar 11 '12 at 18:10


Peter Mortensen
9,039 10 61 96


2mia
499 5 10

Nice, thats a cool tool! – Gerard Jun 30 '14 at 9:13

I can reproduce your results on my system. I fed a 351 megabyte binary file to both programs and the Python version divides by zero because it executes so quickly and the C++ version takes 12 seconds to execute.

I took out the average speed arithmetic and ran the tests a few times:

`cat` takes an average of `0.055` seconds (over eight runs) to dump the file to `/dev/null`.

The Python version takes an average of `.484` seconds and `0.03` ssd (over eight runs) to count the lines. Here's one representative output from `/usr/bin/time`, which is kind enough to show memory used (`20800` max resident kilobytes) and disk IO (`0major` == everything was read from cache).

```
0.48user 0.08system 0:00.56elapsed 98%CPU (0avgtext+0avgdata 20800maxresident)k
0inputs+0outputs (0major+1604minor)pagefaults 0swaps
```

The C++ version takes an average of `12.32` seconds and `0.23` ssd (over eight runs) to count the lines. One representative output from `/usr/bin/time` shows only `4672` max resident kilobytes and again, `0major` shows everything was read from cache:

```
12.34user 0.09system 0:12.45elapsed 99%CPU (0avgtext+0avgdata 4672maxresident)k
0inputs+8outputs (0major+349minor)pagefaults 0swaps
```

I've got more free memory than I know what to do with:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          5979       4413       1566          0        226       2594
-/+ buffers/cache:       1591       4387
Swap:         6347          1       6346
```

As a quick summary, the `4387` in the `free` column of the `-/+ buffers/cache` line indicates that I've got roughly four gigabytes of memory "free" to the kernel any time it pleases. Memory pressure is not an issue.

The Python version created `54898` lines in `strace -o /tmp/python /tmp/readlines.py < /input/file`.

The C++ version created `89802` lines in `strace -o /tmp/cpp /tmp/readlines < /input/file`.

answered Feb 21 '12 at 3:17


samold
70.9k 11 99 147

1 This is not an answer. – immibis Oct 5 at 9:43

Getline, stream operatoes, scanf, can be convenient if you don't care about file loading time or if you are loading small text files...but if performance is something you care about, you should really just buffer the entire file into memory (assuming it will fit). Here's an example:

```
//open file in binary mode
std::fstream file( filename, std::ios::in|::std::ios::binary );
if( !file ) return NULL;

//read the size...
file.seekg(0, std::ios::end);
size_t length = (size_t)file.tellg();
file.seekg(0, std::ios::beg);

//read into memory buffer, then close it.
char *filebuf = new char[length+1];
file.read(filebuf, length);
filebuf[length] = '\0'; //make it null-terminated
file.close();
```

If you want, you can wrap a stream around that buffer for more convenient access like this:

```
std::istrstream header(&buffer[0], length);
```

Also, if you are in control of the file, consider using a flat binary data format instead of text. It's more reliable to read and write because you don't have to deal with all the ambiguities of whitespace. It's also smaller and much faster to parse.

answered Mar 13 '12 at 23:04
Stu
451 5 10

Nice post. But I would just like to mention that the buffer overflow problem with scanf can be handled by specifying the number of characters to be read (for any datatype).

See the width parameter mentioned in the link.

As an example:

```
char s[10];
scanf("%9s",s);    //This will read at most 9 characters from the input.

int x;
scanf("%2d",&x);   //This will read a 2 digit number from the input. (just mentioning)
```

This can take care of buffer overflow. Also dynamic width can not be specified, but to overcome that one could simply generate the format string at run-time (though this will prevent the scanf to do the sanity check at compilation).

answered Mar 11 '12 at 18:05
pareshverma91
315 2 12

By the way, the reason the line count for the C++ version is one greater than the count for the Python version is that the eof flag only gets set when an attempt is made to read beyond eof. So the correct loop would be:

```
while (cin) {
    getline(cin, input_line);

    if (!cin.eof())
        line_count++;
};
```

edited Mar 12 '12 at 4:04

answered Mar 11 '12 at 16:37
Gregg
79 4

25 The really correct loop would be: `while (getline(cin, input_line)) line_count++;` – Jonathan Wakely May 5 '12 at 14:42

@Endrju Why is getline() insecure? – Relish Jun 17 at 0:07

It isn't. I misread it for gets(). – Endrju Jun 17 at 7:17

Well, I see that in your second solution you switched from `cin` to `scanf`, which was the first suggestion I was going to make you (cin is slooooooooooow). Now, if you switch from `scanf` to `gets`, you would see another boost in performance: `gets` is the fastest C++ function for string input.

BTW, didn't know about that sync thing, nice. But you should still try `gets`.

answered Feb 22 '12 at 2:33
José Ernesto Lara Rodrígu
661 5 13

1 Excellent! Although gets() is bad (we should all be using fgets instead, to thwart haxors), I implemented using fgets and am seeing much better performance. Please see my latest edit (5) above. Thanks for pointing this out! – JJC Feb 22 '12 at 11:22

36 You should **********NEVER********** use `gets`. Unrestricted buffer input is a serious problem and `gets` is/was a major contributor. – dreamlax Mar 11 '12 at 7:44

3 Never use gets unless you want a buffer overflow vulnerability in your program. GCC warns you about gets at compile time even if you have no warning flags set up. – Manish Burman Mar 11 '12 at 7:55

In your second example (with scanf()) reason why this is still slower might be because scanf("%s") parses string and looks for any space char (space, tab, newline).

Also, yes, CPython does some caching to avoid harddisk reads.

answered Feb 21 '12 at 3:32

davinchi
58 1 5

When the C++ program had to read the lines, it had to read the file off the disk. When you ran the Python program, the file was already cached in memory. That's probably why the Python program appeared to run quicker.

Also, your C++ program will always count an extra line because you don't check if `getline` succeeded before incrementing the count. Your check of `eof` is both unnecessary and wrong (because it's after the failure you mishandle).

edited Apr 2 '14 at 18:24

Peter Mortensen
9,039 10 61 96

answered Feb 21 '12 at 2:21

David Schwartz
90.3k 6 78 134

@JJC : if you test several times in a row, do you get a different result ? – J.N. Feb 21 '12 at 2:24

1 -1, I think not justified, OP tested several times with the same result. – J.N. Feb 21 '12 at 2:26

1 @J.N. Testing several times with the same method won't help. The problem is his testing method, not the number of times he tested. (For example, if the python program blew out the disk cache when it was done, repeating the test alternating versions won't help.) – David Schwartz Feb 21 '12 at 2:27

Testing several times will bring the file in the cache the second time the C++ code is ran. Of course, not if you wait a long time. – J.N. Feb 21 '12 at 2:28

1 It appears that cin isn't being buffered. If I use an ifstream to open the file directly, I get performance faster than the python version. – Vaughn Cato Feb 21 '12 at 3:18

A first element of an answer: `<iostream>` is slow. Damn slow. I get a huge performance boost with `scanf` as in the below, but it is still two times slower than Python.

```
#include <iostream>
#include <time.h>
#include <cstdio>

using namespace std;

int main() {
    char buffer[10000];
    long line_count = 0;
    time_t start = time(NULL);
    int sec;
    int lps;

    int read = 1;
    while(read > 0) {
        read = scanf("%s", buffer);
        line_count++;
    };
    sec = (int) time(NULL) - start;
    line_count--;
    cerr << "Saw " << line_count << " lines in " << sec << " seconds." ;
    if (sec > 0) {
        lps = line_count / sec;
        cerr << "  Crunch speed: " << lps << endl;
    }
    else
        cerr << endl;
    return 0;
}
```

edited Apr 2 '14 at 18:25

Peter Mortensen
9,039 10 61 96

answered Feb 21 '12 at 3:17

J.N.
5,289 14 27

Didn't see this post until I made my third edit, but thanks again for your suggestion. Strangely, there is no

2x hit for me vs. python now with the scanf line in edit3 above. I'm using 2.7, by the way. – JJC Feb 21 '12 at 3:32

2 After fixing the c++ version, this stdio version is substantially slower than the c++ iostreams version on my computer. (3 seconds vs 1 second) – karunski Feb 21 '12 at 3:39

1 Same here. The sync to stdio was the trick. – J.N. Feb 21 '12 at 4:08

fgets is even faster; please see edit 5 above. Thanks. – JJC Feb 22 '12 at 11:49

The following code was faster for me than the other code posted here so far: (Visual Studio 2013, 64-bit, 500 MB file with line length uniformly in [0, 1000)).

```
const int buffer_size = 500 * 1024;  // Too large/small buffer is not good.
std::vector<char> buffer(buffer_size);
int size;
while ((size = fread(buffer.data(), sizeof(char), buffer_size, stdin)) > 0) {
    line_count += count_if(buffer.begin(), buffer.begin() + size, [](char ch) { return ch
== '\n'; });
}
```

It beats all my Python attempts by more than a factor 2.

answered Apr 23 '14 at 14:56 community wiki Petter

protected by Michael Mrozek Mar 11 '12 at 19:14

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?