

Comparison of Strings Implementations in C++ language

Petr Ovtchenkov

January 15, 2006

Abstract

This article present comparison of strings implementation in STLport and GNU `libstdc++`.

This tests also can help to answer on question: is it strings copy algorithm with constant complexity better than algorithm with linear complexity? This comparison is done on the base of STLport strings, ropes and GNU `libstdc++` strings.

Small mesure for iostreams implementations (and adequate C code) and publication of test times for two hardware (but both has x86 architecture) platforms was recently added.

I expect that this results will help to make decision between STL implementations as well proper choice of strings implementation.

Contents

1	Computers	2
2	Compilers	2
3	Time Measure	2
4	Statistic	2
5	The Tests Descriptions	3
5.1	Add Characters to String (test #1)	3
5.2	Search of Substring (test #2)	3
5.3	Mixed Operations (test #3)	4
5.4	String copy (test #4)	4
5.5	String copy again (test #5)	6
5.6	Short string copy (tests #6 and #7)	7
6	Role of Allocators	7
7	Strings in Multithreaded Environment	8
7.1	Comparison of STLport and GNU <code>libstdc++</code>	8
7.2	Time Profile in GNU <code>libstdc++</code>	10
7.3	Time Profile in STLport	10
8	Strings vs. Ropes	10

9	The Stream Tests Descriptions	16
9.1	Format output to file (test #1)	16
9.2	Raw output to file (test #2)	17
9.3	Raw output to string (test #3)	18
10	Conclusions	19
11	References	19

1 Computers

In the tests was used following computers and operational environments:

1. Hands-made machine (assembled by Tersys company) with two 1.33-GHz AMD® Athlon® XP 1500+ processors under Linux (kernel 2.6.12.5, glibc 2.2.5);
2. ASUS assembled computer with Intel® Pentium® 4 2.4-GHz processor under Linux (kernel 2.6.12.5, glibc 2.2.5).

2 Compilers

For Unix-like systems was used GNU gcc 3.4.4 with appropriate libstdc++ libraries (version 3).

3 Time Measure

Due to “time” function has different options and output format on Linux and other UNIXes, I use program `time` from `complement`¹ project bundle². By the way this function provide high-precision time measure.

4 Statistic

The measure accuracy depends upon program load time, the constant measure drift, common computer load (by other processes) and time measure inaccuracy, the random measure drifts. To reduce influence as the constant measure drift as the random measure drift, the time of test should not to be too short.

Every experiment repeat 10 times, to get more-or-less acceptable statistic. For every series of results I do ordinal statistical manipulation. Mean time is

$$\bar{t} = \sum_{i=1}^n t_i, \quad (1)$$

where t_i is time measure for test i . Mean square deviation

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (t_i - \bar{t})^2 \quad (2)$$

¹<http://complement.sourceforge.net>

²Portability note: system should has `wait3` function.

or

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n t_i^2 - \left(\frac{1}{n} \sum_{i=1}^n t_i \right)^2 \quad (3)$$

or

$$\sigma^2 = \frac{1}{n} \left(\sum_{i=1}^n t_i^2 - \frac{1}{n} \left(\sum_{i=1}^n t_i \right)^2 \right) \quad (4)$$

The equation 4 give algorithm of one-pass incremental calculation both mean value \bar{t} and it mean square deviation σ^2 . Small mean square deviation let us to be ensure that results are gain our trust.

Parameters in the tests are chosen so the time of test itself was greater than program load time/unload time, but still to be acceptable for experimentator's patience.

5 The Tests Descriptions

5.1 Add Characters to String (test #1)

We have empty string. In the loop we add to this string single character. Really this is a test for algorithm of memory allocation (and moving memory contents to reallocated string).

```
1 // -*- C++ -*- Time-stamp: <03/04/04 23:07:39 ptr>
2
3 #include <string>
4
5 using namespace std;
6
7 int main( int, char * const * )
8 {
9     string s;
10
11     for ( int i = 0; i < 100000000; ++i ) {
12         s += "_";
13     }
14
15     return 0;
16 }
```

5.2 Search of Substring (test #2)

We have string. In the loop we search three substrings. The matched substrings are positioned in the beginning, middle and end of the string. All searches are successful.

```
1 // -*- C++ -*- Time-stamp: <03/04/05 22:03:42 ptr>
2
3 #include <string>
4
5 using namespace std;
6
7 int main( int, char * const * )
8 {
```

```

9   string s( "qyweyuewunfkHBUKGYUGL,wehbyYGUW^\
10  (@T@H!BALWD:h^&@#*@(:JKHWJ:CND" );
11
12  for ( int i = 0; i < 10000000; ++i ) {
13      s.find( "unfkHBUKGY" );
14      s.find( "W^(@T@H!B" );
15      s.find( "J:CND" );
16  }
17
18  return 0;
19 }

```

5.3 Mixed Operations (test #3)

This is a mix of common operations under strings: assignment, search of substring, replace of substring by another substring, concatenation of strings.

```

1  // -*- C++ -*- Time-stamp: <03/04/05 22:13:14 ptr>
2
3  #include <string>
4
5  using namespace std;
6
7  int main( int, char * const * )
8  {
9      string s( "qyweyuewunfkHBUKGYUGL,wehbyYGUW^\
10  (@T@H!BALWD:h^&@#*@(:JKHWJ:CND" );
11      string::size_type p;
12      string ss1( "unfkHBUKGY" );
13      string ss2( "123456" );
14      string sx;
15
16      for ( int i = 0; i < 10000000; ++i ) {
17          sx = s;
18          p = sx.find( ss1 );
19          sx.replace( p, ss1.size(), ss2 );
20          sx += s;
21      }
22
23      return 0;
24 }

```

5.4 String copy (test #4)

This test intended to highlight the cost of strings copy (just copy, without modifications). Implementations of strings that use algorithms with constant copy time ($O(1)$) expected to show better results here.

```

1  // -*- C++ -*- Time-stamp: <04/07/14 23:39:44 ptr>
2
3  #include <string>
4
5  using namespace std;

```

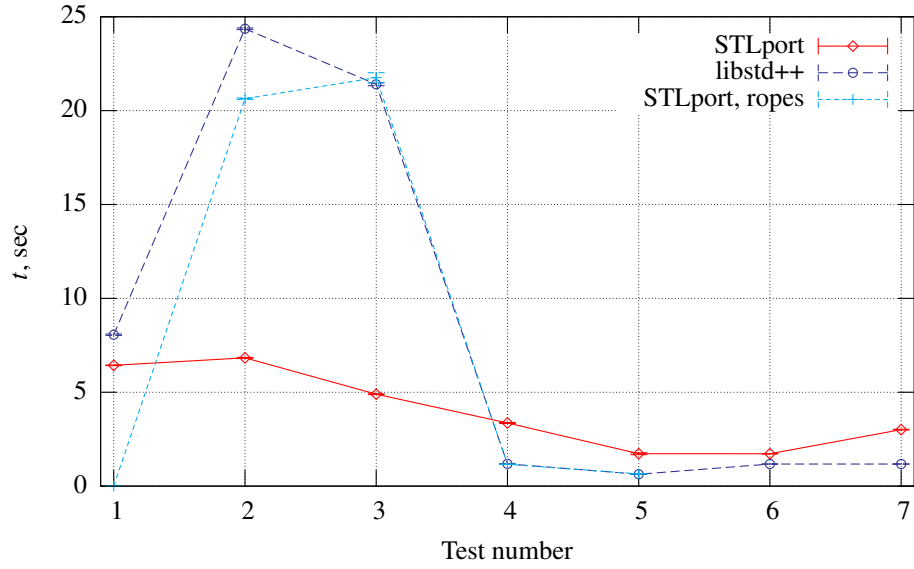


Figure 1: Comparison of GNU `libstdc++` and STLport strings implementation. Single thread. t is a wall time for tests. The implementation of strings in STLport (copy algorithm complexity $O(n)$) dramatically faster than ones in GNU `libstdc++` or ropes in STLport (copy algorithm complexity $O(1)$) for tests with strings modification. If the test only copy string, the implementation with $O(1)$ complexity is better.

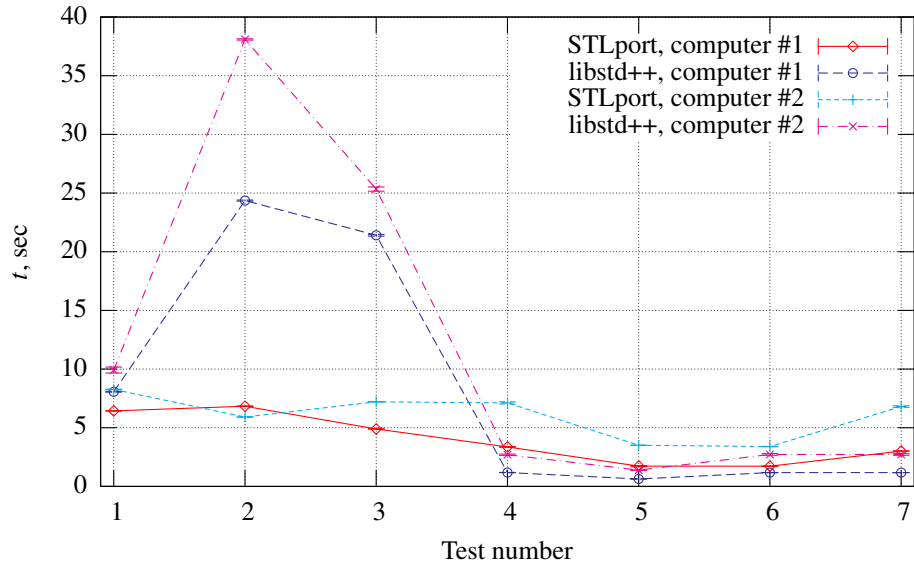


Figure 2: Comparison of two x86 platforms. See platforms description on page 2. This graphics show that higher frequency isn't synonym of better performance

```

6
7 string func( string par )
8 {
9     string tmp( par );
10
11     return tmp;
12 }
13
14 int main( int, char * const * )
15 {
16     string s( "qyweyuewunfkHBUKGYUGL,wehbyGUW^\
17 (@T@H!BALWD:h^&@#*@( #:JKHWJ:CND" );
18
19     for ( int i = 0; i < 10000000; ++i ) {
20         string sx = func( s );
21     }
22
23     return 0;
24 }

```

You can see (fig. 1) that GNU libstdc++ show better results than STLport in this test. The reason is that GNU libstdc++ use copy algorithm with $O(1)$ complexity: during copy operation there are no memory allocation occur—two strings refer to the same memory chunk (like `tmp` and `par` on line 9). This test has three copy operations (lines 7, 9 and 20).

5.5 String copy again (test #5)

This is a variant of test above (5.4), but use more realistic parameter pass (by reference).

```

1 // -*- C++ -*- Time-stamp: <04/07/14 23:40:29 ptr>
2
3 #include <string>
4 #include <pthread.h>
5
6 using namespace std;
7
8 string func( const string& par )
9 {
10     string tmp( par );
11
12     return tmp;
13 }
14
15 int main( int, char * const * )
16 {
17     string s( "qyweyuewunfkHBUKGYUGL,wehbyGUW^\
18 (@T@H!BALWD:h^&@#*@( #:JKHWJ:CND" );
19
20     for ( int i = 0; i < 10000000; ++i ) {
21         string sx = func( s );
22     }
23
24     return 0;

```

25 }

As in test above (5.4) the GNU `libstdc++` show better time by the same reason. But let's draw attention to the tendency: I remove 1/3 of string copy operations (two string copy operations remains in the test, no copy of strings while pass parameter into function), and wall time of this test decline 1/3 too.

If I will pass only const reference to string, then I will avoid any copy operations and a difference in time between two approaches will disappear.

5.6 Short string copy (tests #6 and #7)

This is a variant of test above (5.4), but use “short” string parameter pass. This test show effect of “short string optimization” technique in STLport. This technique use short buffer in the “string” instance, this allow to skip memory allocation for short strings.

```
1 // -*- C++ -*- Time-stamp: <04/07/15 23:56:40 ptr>
2
3 #include <string>
4
5 using namespace std;
6
7 string func( string par )
8 {
9     string tmp( par );
10
11     return tmp;
12 }
13
14 int main( int, char * const * )
15 {
16     string s( "1234567890" );
17
18     for ( int i = 0; i < 10000000; ++i ) {
19         string sx = func( s );
20     }
21
22     return 0;
23 }
```

Test #6 is default STLport, with “short string optimization”, while test #7 present results without “short string optimization”. In case of GNU `libstdc++` tests #6 and #7 are the same.

STLport was build with “short string” size 16. “Short strings optimization” give test time 3% longer then without ones (for strings with size 20, this test case not shown on figures). But you see, for “short” strings test with “short” strings optimization show time that 43% better!

6 Role of Allocators

The STLport provide default “optimized” memory allocator (`node_alloc`). This allocator was used when I run strings tests for STLport (see section 5). May be the win of STLport is due to advanced memory allocation technique?

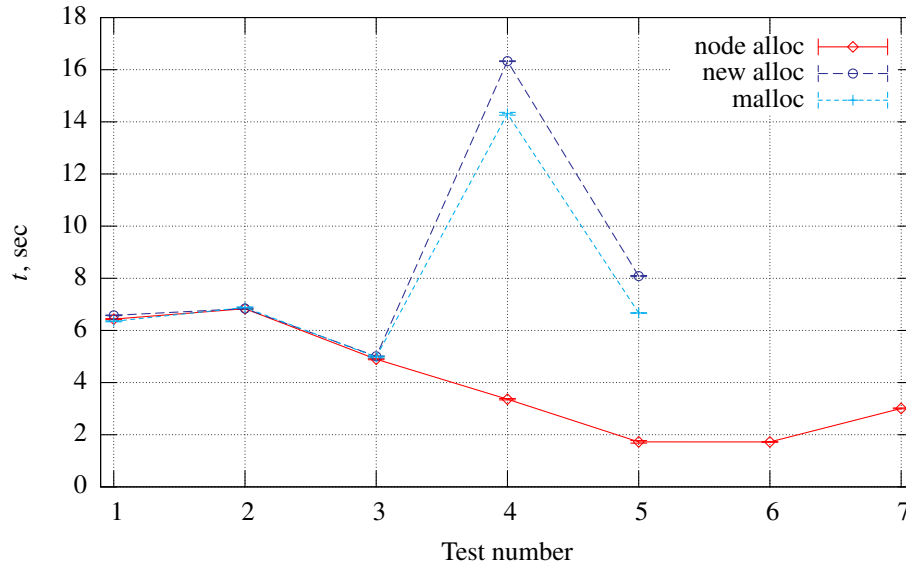


Figure 3: Role of memory allocator in strings implementation (STLport). Wall time for tests.

In the STLport implemented three base memory allocators:

- “optimized” `node_alloc`;
- adapter around `new` operator;
- adapter around `malloc` call.

Let’s repeat tests from section 5 for STLport with different allocators. We see (fig. 3) that all allocators are good enough For search operations (test 2) the results as expected are the same (within measure of inaccuracy). The tests 4 and 5 show that cost of `malloc` system call—`node_alloc` reuse once allocated memory. But copy of string’s content still present (compare with time of GNU `libstdc++` on fig. 1).

7 Strings in Multithreaded Environment

Memory allocation performance in single and multithreaded environments is an important aspect of any application. The work with C++ strings in multithreaded environments is highly depends upon underlying allocator.

The tests are the same as described in section 5, except that every test run simultaneously in two threads.

7.1 Comparison of STLport and GNU `libstdc++`

The results (fig. 4, 5) are near the same as shown on figure 1 except that for mix operations test (test 3, described in section 5.3) the STLport rope’s performance degradate too much. In previous releases of GNU `libstdc++` it performance was worse (significant) than rope’s but in present implementation it good enough.

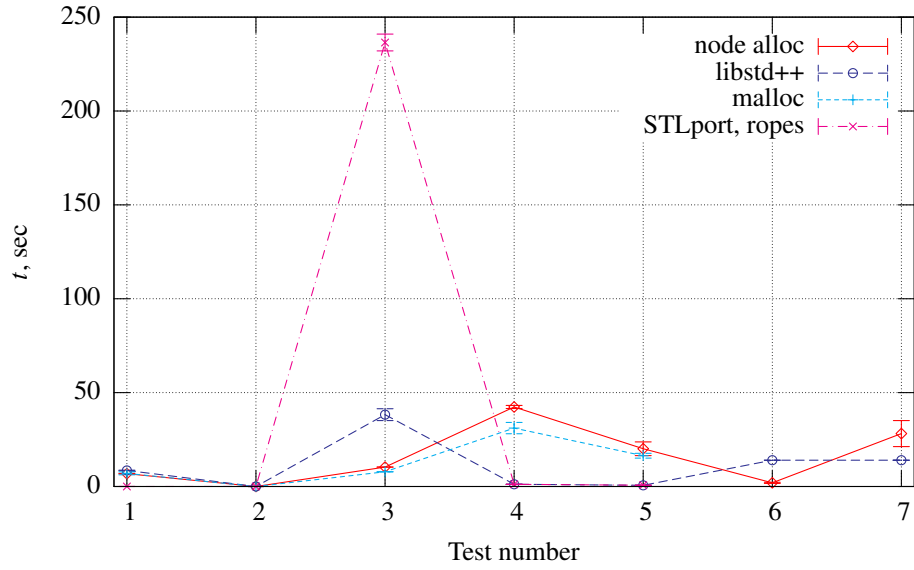


Figure 4: Tests wall time (GNU libstdc++ and STLport strings) in MT environment (two threads).

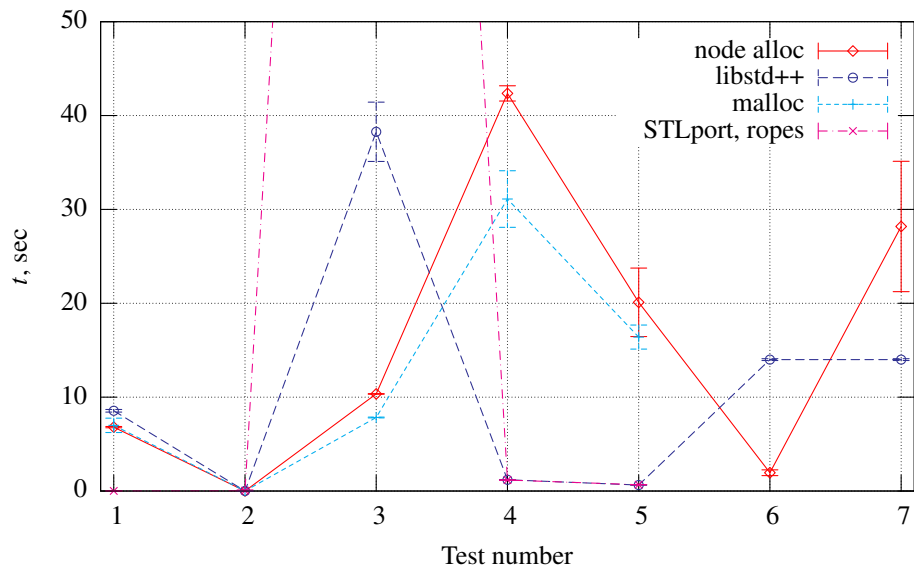


Figure 5: Tests wall time (GNU libstdc++ and STLport strings) in MT environment (two threads) (same as fig. 4, another scale).

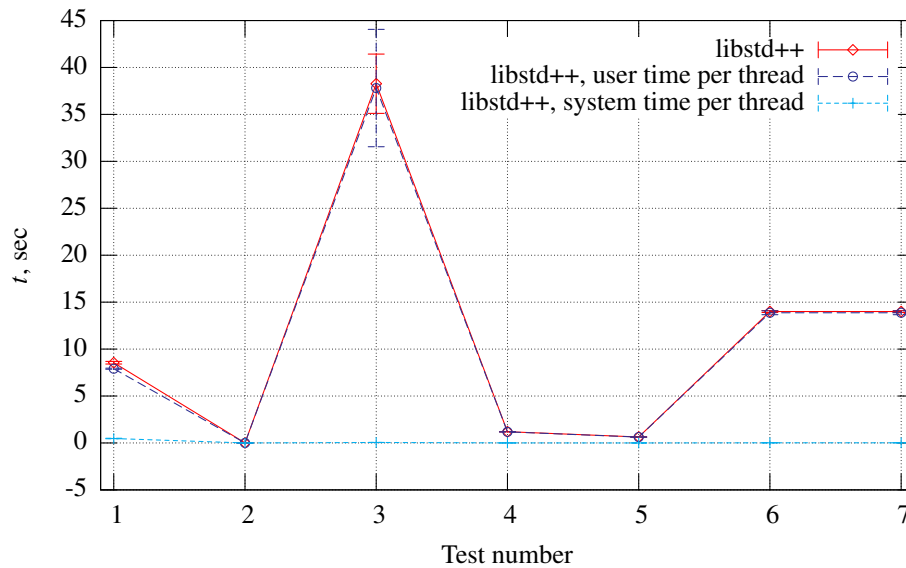


Figure 6: Wall time, user time and system time per thread for GNU libstdc++ in MT environment.

7.2 Time Profile in GNU libstdc++

On figure 6 you can see user, system and wall time for tests with GNU libstdc++ strings implementation. We see that both threads remain in user space almost all time. This may be due to a lot of waiting state.

As we can see in test 3, the general performance problem is seems in a lot of thread synchronization operation (a lot of system time, that is greater than user time, in test 3 I can associate only with thread synchronization primitives).

7.3 Time Profile in STLport

Figure 7 present user and system time per thread and wall time for tests with STLport with `node_alloc`-based, figure 8 show profile for tests with `malloc`-based allocators, and figure 9 for ropes.

The surprise for me was that `malloc`-based variant win (in 1.5–2 times faster) over `node_alloc`-based variant in tests #3, #4 and #5. This fact can be explained by usage of memory allocated chunks vector in `node_alloc`. This vector accessed from different threads and such access should use thread synchronization primitives. Compare figures 3 and 4. The tests 1–2 has the same time in MT and non-MT context for `malloc`-based and `node_alloc`-based allocators. In test 3 `malloc`-based variant climb down, while `node_alloc`-based keep position.

8 Strings vs. Ropes

Some time ago, there are many discussions about complexity of string assignment algorithm. In this time SGI made experimental implementation of standard string interfaces

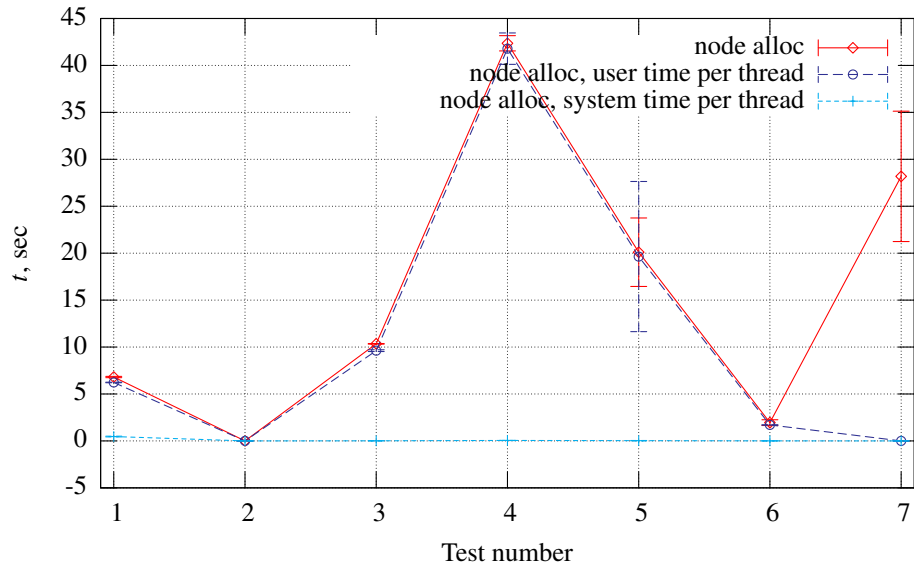


Figure 7: Wall time, user time and system time per thread for STLport (node allocator) in MT environment.

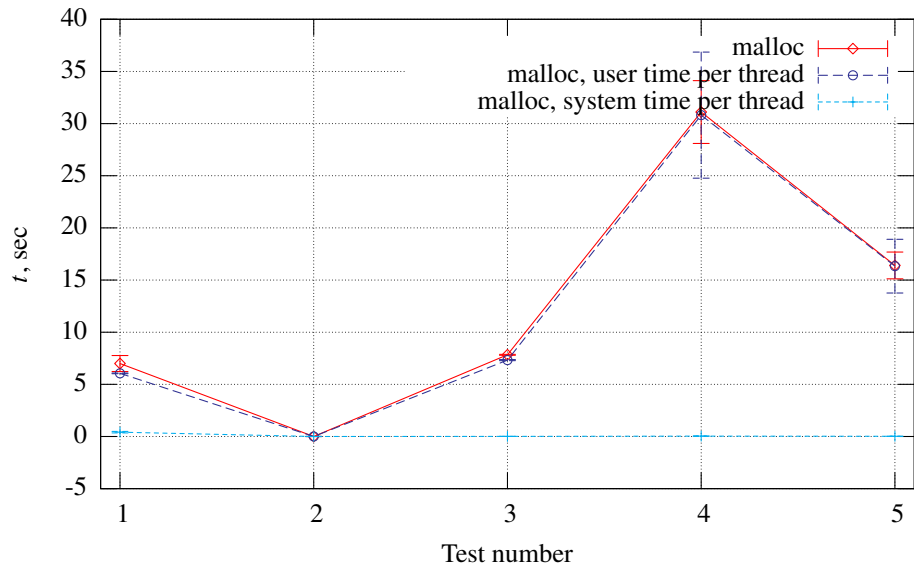


Figure 8: Wall time, user time and system time per thread for STLport (malloc allocator) in MT environment.

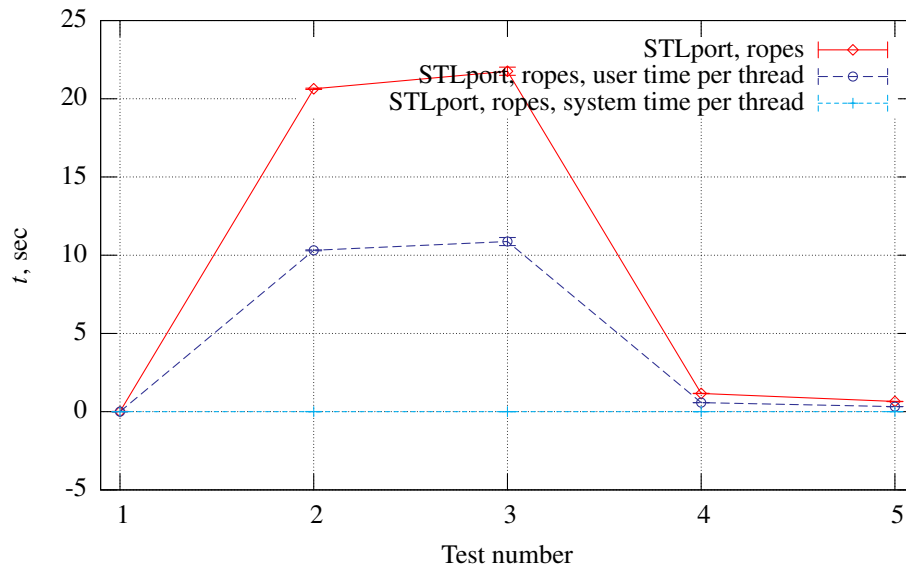


Figure 9: Wall time, user time and system time per thread for STLport ropes in MT environment.

with constant copy/insert/replace algorithm complexity—the ropes.

Really ropes and strings has different usage scope. I use STLport implementation (that come from SGI string and rope classes) to compare ones. You can see STLport story here: <http://stlport.sourceforge.net/History.shtml>. Some words about string and rope from first hands you can find here: http://www.sgi.com/tech/stl/string_discussion.html.

This test based on mix of copy, insert, append and replace operations under classic strings or ropes. The main part of test you can see on lines 23–32 (where T is either string or rope).

```

1 // -*- C++ -*- Time-stamp: <06/01/03 23:09:43 ptr>
2
3 #include <misc/args.h>
4
5 #ifndef STLPORT
6 # include <rope>
7 #endif
8 #include <string>
9 #include <iostream>
10
11 using namespace std;
12
13 template <class T>
14 class test
15 {
16 public:
17     test( int b, int i );
18 };

```

```

19
20 template <class T>
21 test<T>::test( int b, int n )
22 {
23     T s( b, 'a' );
24     T v;
25     for ( int i = 0; i < n; ++i ) {
26         v = s;
27         v.insert( 0, "qwerty");
28         v.insert( b / 2, "zxcvb" );
29         v.append( "rtyui" );
30         v.replace( 0, 20U, "ghfjhfjff" );
31         v.replace( b / 2, 20U, "abcdefg" );
32     }
33 }
34
35 int bs;
36 int n;
37 bool use_str = true;
38
39 int main( int argc, char * const *argv )
40 {
41     try {
42         Argv arg;
43         arg.copyright( "Copyright_(C)_Petr_Ovtchenkov,_2003,_2005" );
44         arg.brief( "Comparison_of_ropes_and_strings" );
45         arg.option( "-h", false, "print_this_help_message" );
46         arg.option( "-r", true, "use_ropes" );
47         arg.option( "-s", true, "use_strings" );
48         arg.option( "-b", 1, "block_size" );
49         arg.option( "-i", 1, "number_of_iterations" );
50         try {
51             arg.parse( argc, argv );
52         }
53         catch ( std::invalid_argument& err ) {
54             cerr << err.what() << endl;
55             arg.print_help( cerr );
56             throw 1;
57         }
58         bool turn;
59         if ( arg.assign( "-h", turn ) ) {
60             arg.print_help( cerr );
61             throw 0;
62         }
63
64         if ( arg.is( "-r" ) && arg.is( "-s" ) ) {
65             cerr << "Either_-r_or_-s_allowed" << endl;
66         }
67
68         arg.assign( "-s", use_str );
69
70         if ( arg.is( "-r" ) ) {
71             use_str = false;
72         }

```

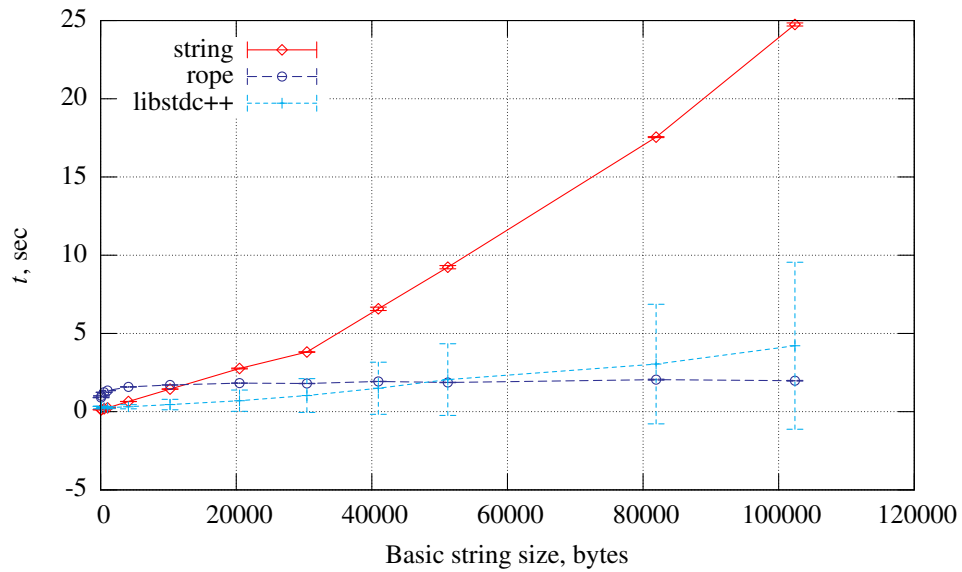


Figure 10: Strings vs. Ropes.

```

73     arg.assign( "-b", bs );
74     arg.assign( "-i", n );
75 }
76 catch ( std::runtime_error& err ) {
77     cerr << err.what() << endl;
78     return -1;
79 }
80 catch ( std::exception& err ) {
81     cerr << err.what() << endl;
82     return -1;
83 }
84 catch ( int r ) {
85     return r;
86 }
87
88 if ( !use_str ) {
89 #ifdef STLPORT
90     test<rope<char> > t( bs, n );
91 #endif
92 } else {
93     test<string> t( bs, n );
94 }
95
96 return 0;
97 }

```

As expected, efforts to establish constant assign/insert/replace algorithm complexity lead to overhead—so ropes are preferable if you want to process long strings (longer than 12K, really depends upon compiler's optimization quality), as you can see on figures 10 and 11. The strings implementation in GNU libstdc++ show better results

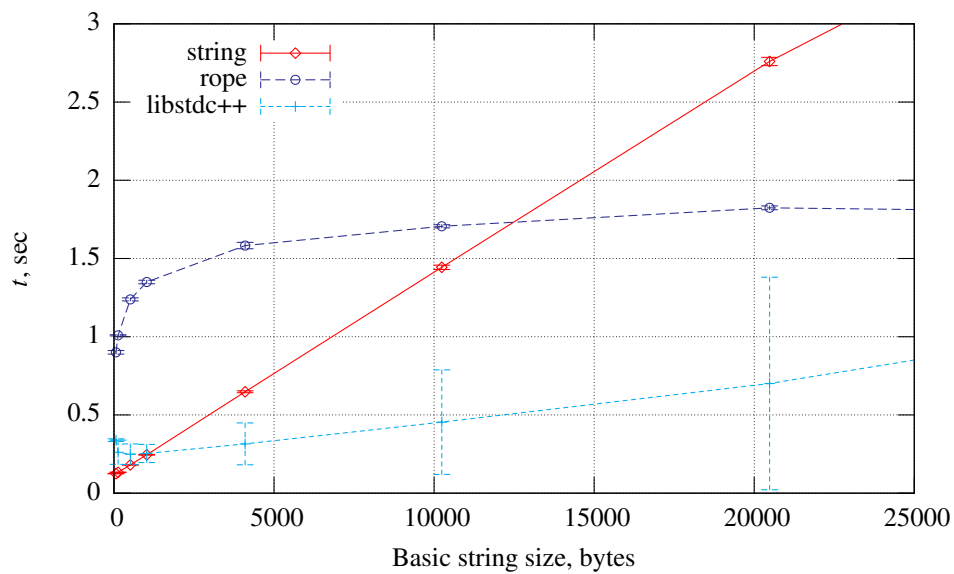


Figure 11: Strings vs. Ropes.

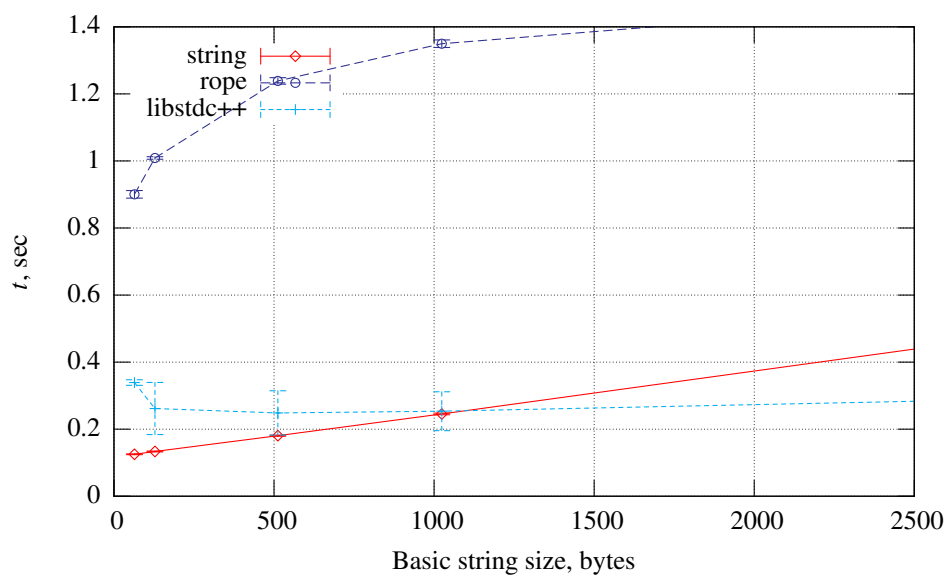


Figure 12: Strings vs. Ropes.

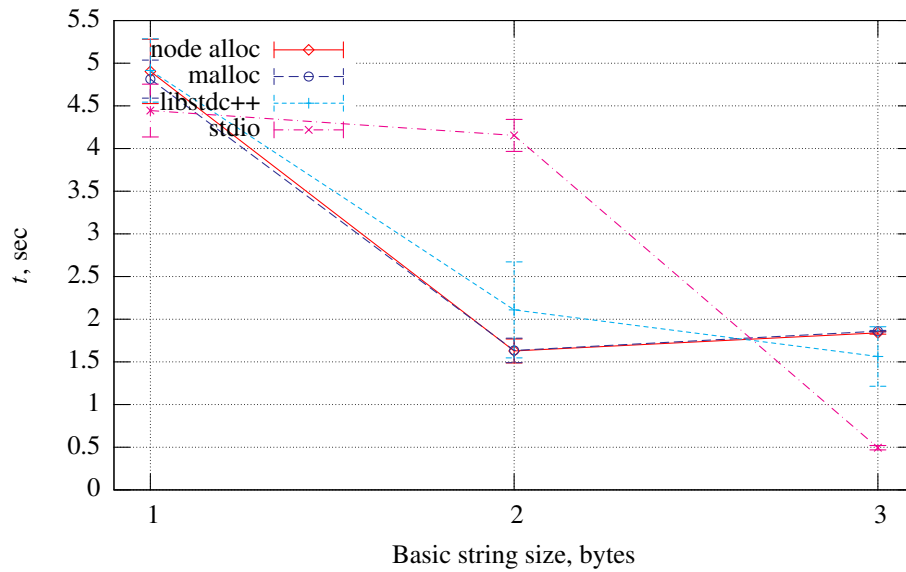


Figure 13: Streams.

beginning from size 1K (figures 10–12).

Evident, that complexity of `string` operations is near the linear (as in STLport as in GNU `libstdc++` implementations), while `rope` has constant complexity. But the overhead of `rope` is significant, so cost of constant algorithm complexity is high enough.

9 The Stream Tests Descriptions

9.1 Format output to file (test #1)

First test is a format output to a file. File stream opened for writing and printing a set of records (integer, char and double). Each record terminated by 0 (end-of-string).

```

1 // -*- C++ -*- Time-stamp: <05/04/27 18:47:41 ptr>
2
3 #include <fstream>
4
5 using namespace std;
6
7 int main( int, char * const * )
8 {
9     ofstream s( "test" );
10
11     for ( int i = 0; i < 1000000; ++i ) {
12         s << i << "_" << (static_cast<double>(i) + 0.1415926) << ends;
13     }
14
15     return 0;

```



```
16 }
```

As reference I use C program (titled as stdio at figure 13) that do near the same.

```
1  /* Time-stamp: <06/01/04 00:16:20 ptr> */
2
3  #include <stdio.h>
4
5  int main( int argc , char *argv[] )
6  {
7      FILE *f;
8      int i;
9
10     f = fopen( "test", "w" );
11
12     for ( i = 0; i < 1000000; ++i ) {
13         fprintf( f, "%d_%.f\n", i, (double)i + 0.1415926 );
14     }
15
16     fclose( f );
17
18     return 0;
19 }
```

9.2 Raw output to file (test #2)

Raw write to file.

```
1  // -*- C++ -*- Time-stamp: <05/04/27 18:47:41 ptr>
2
3  #include <fstream>
4
5  using namespace std;
6
7  int main( int, char * const * )
8  {
9      ofstream s( "test" );
10
11     for ( int i = 0; i < 10000000; ++i ) {
12         s.write( (const char *)&i, sizeof(i) );
13     }
14
15     return 0;
16 }
```

Near the same test on pure C (stdio).

```
1  /* Time-stamp: <06/01/04 00:21:07 ptr> */
2
3  #include <stdio.h>
4
5  int main( int argc , char *argv[] )
6  {
7      FILE *f;
8      int i;
9  }
```

```

10  f = fopen( "test", "w" );
11
12  for ( i = 0; i < 100000000; ++i ) {
13      fwrite( (const void *)&i, sizeof(i), 1, f );
14  }
15
16  fclose( f );
17
18  return 0;
19 }

```

9.3 Raw output to string (test #3)

Raw write to string stream.

```

1  // -*- C++ -*- Time-stamp: <05/04/27 18:16:09 ptr>
2
3  #include <sstream>
4
5  using namespace std;
6
7  int main( int, char * const * )
8  {
9      stringstream s;
10
11     for ( int i = 0; i < 100000000; ++i ) {
12         s.write( (const char *)&i, sizeof(i) );
13     }
14
15     return 0;
16 }

```

Analog on C. To be near the real life I simulate that I don't know the maximum size of buffer.

```

1  /* Time-stamp: <06/01/04 01:31:49 ptr> */
2
3  #include <stdio.h>
4
5  int main( int argc, char *argv[] )
6  {
7      char *buf;
8      size_t sz = 16;
9      int i;
10
11     buf = malloc( sz );
12
13     for ( i = 0; i < 100000000; ++i ) {
14         if ( sz < (i * sizeof(i) + sizeof(i)) ) {
15             char *tmp;
16
17             tmp = malloc( sz * 2 );
18             memcpy( tmp, buf, sz );
19             free( buf );
20             buf = tmp;

```

```

21         sz = sz * 2;
22     }
23     memcpy( buf + i * sizeof(i), &i, sizeof(i) );
24 }
25
26 free( buf );
27
28 return 0;
29 }

```

10 Conclusions

This tests show that

- for processing long strings (greater than 50K) the best choice is ropes;
- if you use strings of size 1K–50K then the best choice is strings from GNU libstdc++; GNU libstdc++ will be good too if you only pass strings as parameters, without modifications;
- if you modify strings with sizes less then 1K, the strings from STLport are for you.

As for streams, we see that format output to file show near the same time for STLport, GNU libstdc++ and (even!) C `fprintf`. For raw write the STLport's implementation is a bit better than GNU libstdc++ (and both are significant better than C `fwrite`). With raw write to string stream (or in char buffer in case of C) the pure C variant is 3–4 times faster.

As about Intel® vs. AMD®, no comments—see figure 2 on page 5.

11 References

STLport <http://stlport.sourceforge.net>

GCC <http://gcc.gnu.org>

SGI <http://www.sgi.com/tech/stl/>

Complement <http://complement.sourceforge.net>