Comparison of Strings Implementations in C++ language

Petr Ovtchenkov[*]

October 18, 2006

Abstract

This article present comparison of STL strings with copy-on-write and non-copy-on-write argorithms, based on STLport strings, ropes and `GNU libstdc++` implementations.

Some related issues, like fstream and stringstream performance are also under consideration.

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

Contents

1 Computers

In the tests was used following computers and operational envirinments:

1. Tyan Tiger motherboard, two 1.33-GHz AMD® Athlon™ XP 1500+ processors under Linux (kernel 2.6.16.26, `glibc` 2.3.6);
2. Two 2.6-GHz AMD® Opteron™ 252 processors under Linux (kernel 2.6.15, x86_64, `glibc` 2.3.6)

2 Compilers

In tests was used GNU gcc 3.4.4 (on AMD64) or 4.1.1 (on AMD) with appropriate `libstdc++` libraries (version 3).

3 Time Measure

Due to “time” function has different options and output format on Linux and other UINIXes, I use program `time` from `complement`[1] project bundle[2]. 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 repeate 10 times, to get more-or-less acceptable statistic. For every series of results I do ordinal statistical manipulation. Mean time is

$$\bar{t} = \sum_{i=1}^{n} t_i, \tag{1}$$

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

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

[1]`http://complement.sourceforge.net`, see appropriate SVN repository on SourceForge.
[2]Portability note: system should has `wait3` function.

or

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

or

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

The equation 4 give algorithm of one-pass incremental calculation both mean value $\overline{t}$ 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

All tests are "synthetic", to show the diference in programming technique.

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).

```
// -*- C++ -*- Time-stamp: <03/04/04 23:07:39 ptr>

#include <string>

using namespace std;

int main( int, char * const * )
{
  string s;

  for ( int i = 0; i < 100000000; ++i ) {
    s += " ";
  }

  return 0;
}
```

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.

```
// -*- C++ -*- Time-stamp: <03/04/05 22:03:42 ptr>

#include <string>

using namespace std;

```

```
int main( int, char * const * )
{
  string s( "qyweyuewunfkHBUKGYUGL,wehbYGUW^\
(@T@H!BALWD:h^&@#*@(#:JKHWJ:CND" );

  for ( int i = 0; i < 10000000; ++i ) {
    s.find( "unfkHBUKGY" );
    s.find( "W^(@T@H!B" );
    s.find( "J:CND" );
  }

  return 0;
}
```

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.

```
// -*- C++ -*- Time-stamp: <03/04/05 22:13:14 ptr>

#include <string>

using namespace std;

int main( int, char * const * )
{
  string s( "qyweyuewunfkHBUKGYUGL,wehbYGUW^\
(@T@H!BALWD:h^&@#*@(#:JKHWJ:CND" );
  string::size_type p;
  string ss1( "unfkHBUKGY" );
  string ss2( "123456" );
  string sx;

  for ( int i = 0; i < 10000000; ++i ) {
    sx = s;
    p = sx.find( ss1 );
    sx.replace( p, ss1.size(), ss2 );
    sx += s;
  }

  return 0;
}
```

5.4 String copy (test #4)

This test intended to hilight the cost of strings copy (jusr copy, without modifications). Implementations of strings that use copy-on-write algorithms expected to show better results here.

```
// -*- C++ -*- Time-stamp: <04/07/14 23:39:44 ptr>

#include <string>
```

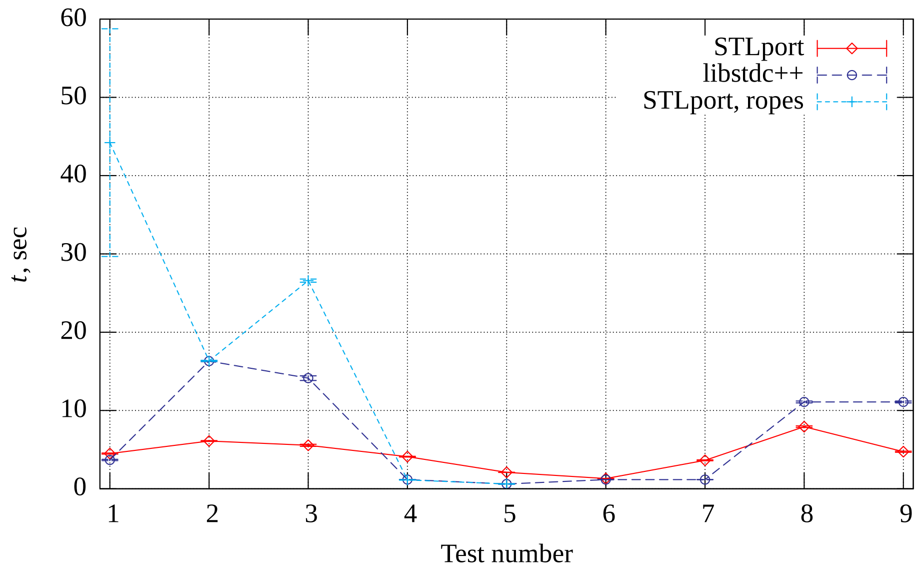

Figure 1: STL strings implementations comparison. Single thread. *t* is a wall time for tests. On X-axis you see test's number (see text). The implementation of strings in STLport (non-COW) drammaticaly faster than COW implementations (`GNU libstdc++` and ropes in STLport) in tests that modify strings. But if the test only copy string, the COW implementation show better performance.

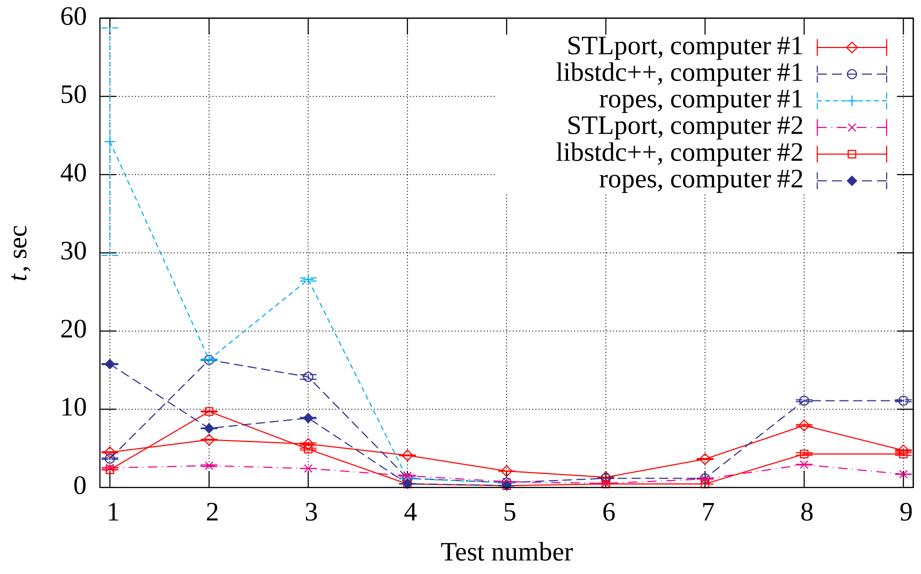

Figure 2: Progress of systems over 3 years: two processors systems. See platforms description on page 2. This graphics show that progress was not only in frequency (computer #2 has twice frequency over #1, but most times are more then twice better), but in core architecture too (or this is compiler?)

```
 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( "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 COW algorithm: 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).

```
// -*- C++ -*- Time-stamp: <04/07/14 23:40:29 ptr>

#include <string>
#include <pthread.h>

using namespace std;

string func( const string& par )
{
  string tmp( par );

  return tmp;
}

int main( int, char * const * )
{
  string s( "qyweyuewunfkHBUKGYUGL,wehbYGUW^\
(@T@H!BALWD:h^&@#*@(#:JKHWJ:CND" );

  for ( int i = 0; i < 10000000; ++i ) {
    string sx = func( s );
  }

  return 0;
}
```

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.

```
// -*- C++ -*- Time-stamp: <04/07/15 23:56:40 ptr>

#include <string>

using namespace std;

string func( string par )
{
  string tmp( par );

  return tmp;
}

int main( int, char * const * )
{
  string s( "1234567890" );

  for ( int i = 0; i < 10000000; ++i ) {
    string sx = func( s );
  }

  return 0;
}
```

Test #6 is default STLport, with "short string optimization", while test #7 present results without "short string optimization"[3]. 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 64% better!

[3]define _STLP_DONT_USE_SHORT_STRING_OPTIM

5.7 String proxy objects (tests #8 and #9)

The agregation of strings using the `+` operator is an expensive operation as it requires construction of temporary objects that need memory allocation and deallocation. The problem can be even more important if you are adding several strings together in a single expression. To avoid this problem STLport implement expression template. With this technique addition of 2 strings is not a string anymore but a temporary object having a reference to each of the original strings involved in the expression. This object carry information directly to the destination string to set its size correctly and only make a single call to the allocator. This technique also works for the addition of N elements where elements are `basic_string`, `C` string or a single character.

The drawback can be longer compilation time and bigger executable size.

Another problem is that some compilers (gcc) fail to use string proxy object if do with class derived from string.

Let's try to estimate the benefits from string proxy technique with following test.

```
// -*- C++ -*- Time-stamp: <03/04/04 23:07:39 ptr>

#include <string>

using namespace std;

int main( int, char * const * )
{
  string s;
  string s1 = "1234567";
  string s2 = "12345678901234567890";
  string s3 = ".ext";
  string s4 = "␣/*␣my␣comment␣about␣this␣*/";

  for ( int i = 0; i < 5000000; ++i ) {
    s = s1 + "/" + s2 + s3 + "␣=>␣" + s4;
  }

  return 0;
}
```

Test #8 is default STLport, in test #9 temporary string objects in use[4]. For GNU `libstdc++` both #8 and #9 are the same test.

The figure 1 show that STLport implementation in comparison with GNU `libstdc++` give better results even without expression templates. But with expression templates the results are better more then twice.

6 Role of Allocators

Note: in the `glibc` 2.3.6 enhancement of memory allocation (over `glibc` 2.2.5) leads to advantage of STLport's `node_alloc` became insignificant.

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?

In the STLport implemented three base memory allocators:

[4]define `_STLP_USE_TEMPLATE_EXPRESSION` turn on

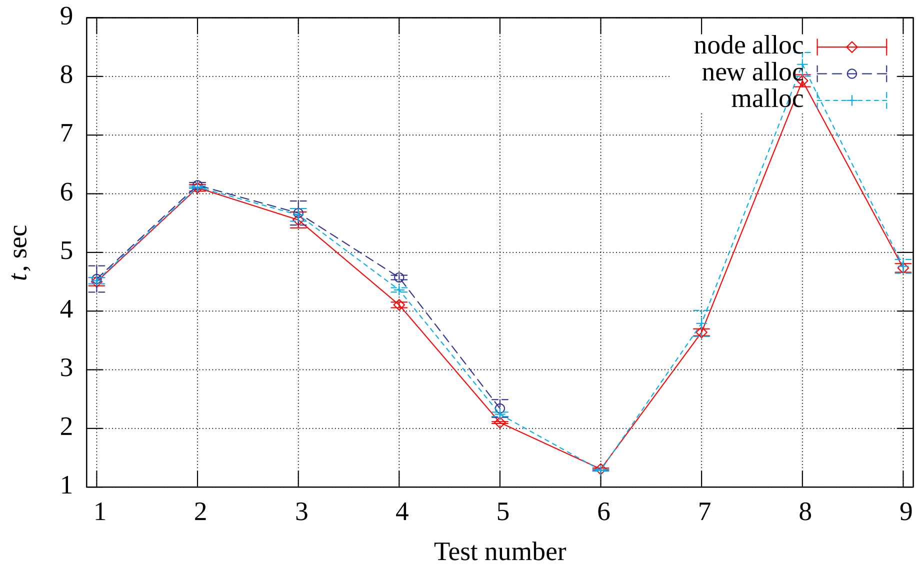

Figure 3: Role of memory allocator in strings implementation (STLport). Wall time for tests.

- “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). In the past (`glibc` 2.2.5, compiler GNU gcc 3.4.4) the cost of `malloc` was significant, but now it near zero (compare graphics 3 and 4).

The hardware are the same on figures 3 and 4, different Linux kernel (2.6.16.26 vs 2.6.12.5), different glibc (2.3.6 vs 2.2.5) and different versions of compiler (gcc 4.1.1 vs 3.4.4).

Tests for parameters pass (by referenece, #5) give the same results, in tests #7 (short strings, no “short string optimization”) older implementation slightly win. But in other tests fresh implementation is much better!

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.

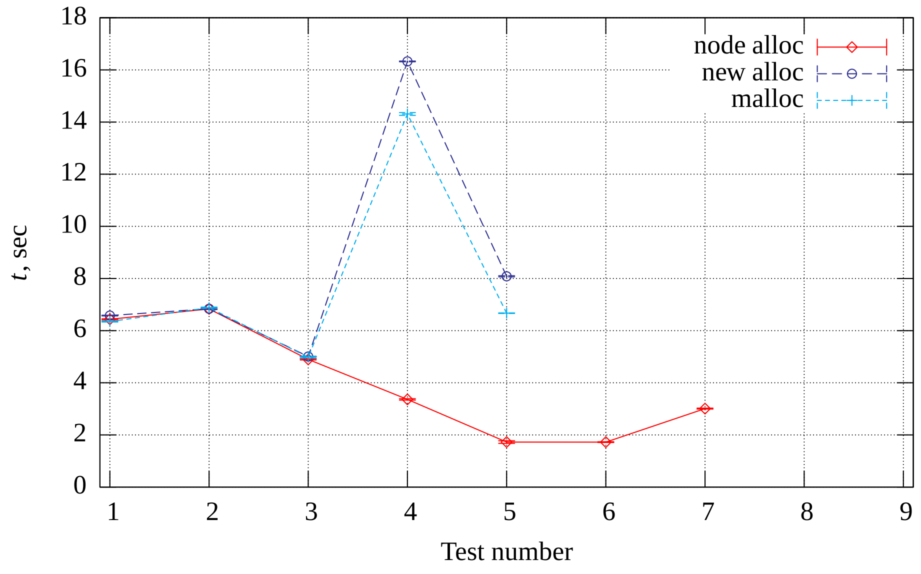

Figure 4: Role of memory allocator in strings implementation (`STLport`), `glibc` 2.2.5, compiler GNU gcc 3.4.4. Wall time for tests. The hardware are the same as for results on figure 3.

7.1 Comparison of `STLport` and `GNU libstdc++` in multi-threaded applications

The results (fig. 5, 6) 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.

7.2 Time Profile in `GNU libstdc++`

On figure 7 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 8 present user and system time per thread and wall time for tests with STLport with `node_alloc`-based, figure 9 show profile for tests with `malloc`-based allocators, and figure 10 for ropes.

The surprise for me was that `malloc`-based variants 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 dif-

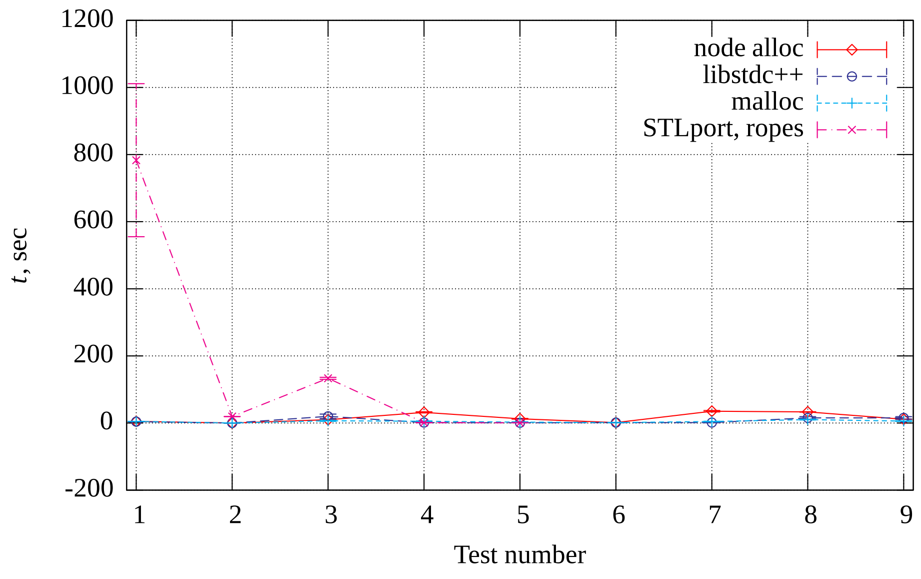

Figure 5: Tests wall time (GNU `libstdc++` and STLport strings) in multi-threaded environment (two threads).

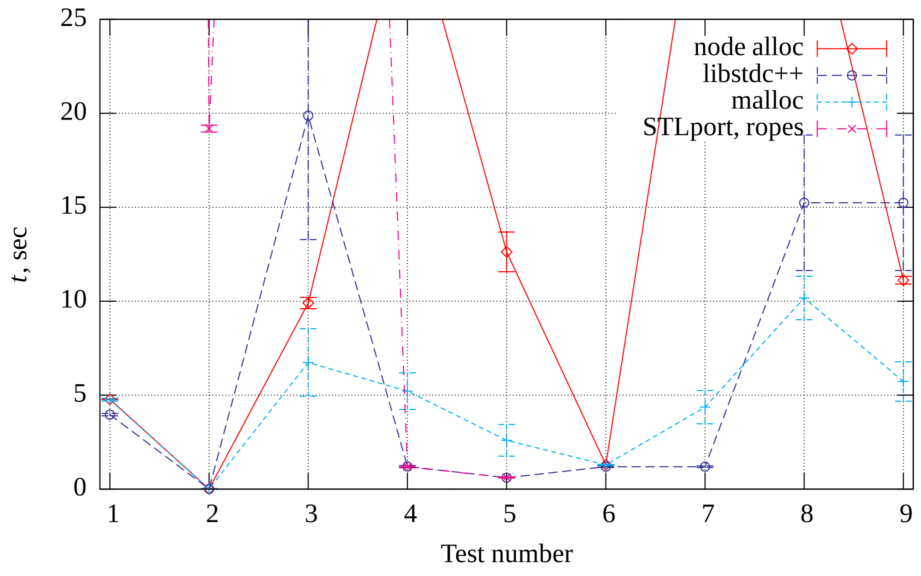

Figure 6: Tests wall time (GNU `libstdc++` and STLport strings) in multi-threaded environment (two threads) (same as fig. 5, another scale).

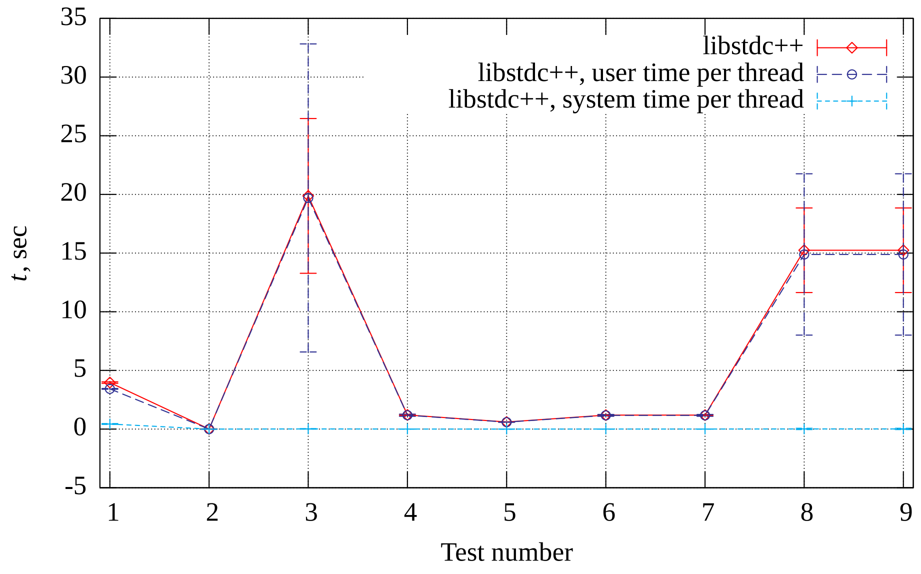

Figure 7: Wall time, user time and system time per thread for `GNU libstdc++` in multi-threaded environment.

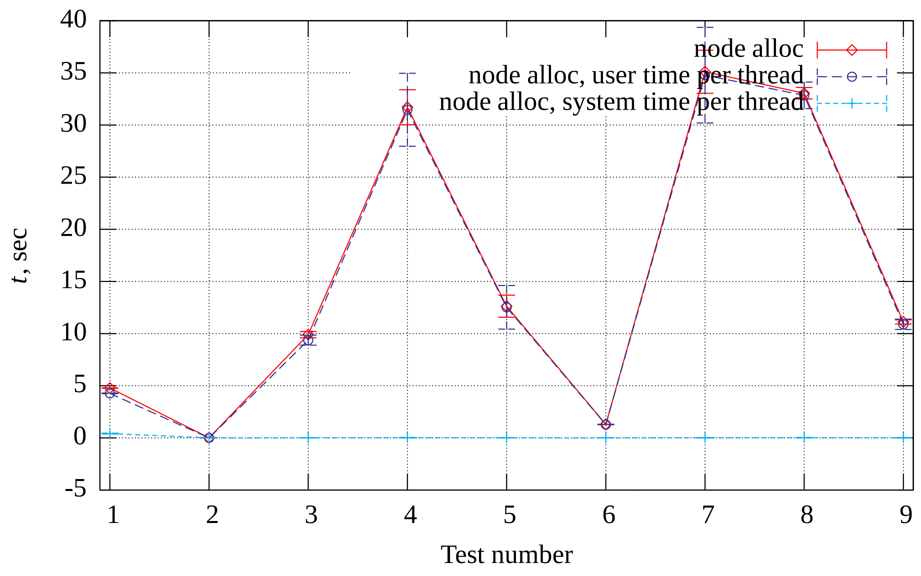

Figure 8: Wall time, user time and system time per thread for STLport (node allocator) in multi-threaded environment.

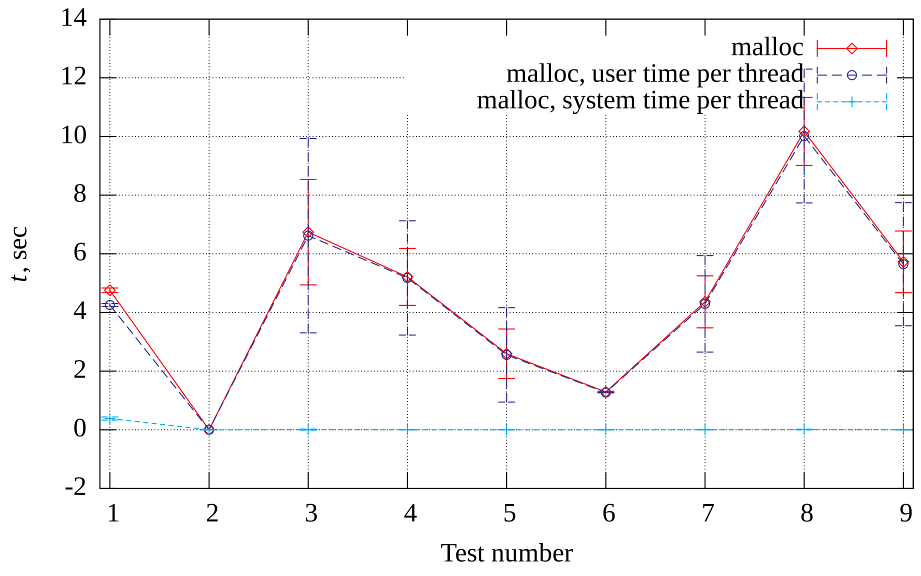

Figure 9: Wall time, user time and system time per thread for STLport (malloc allocator) in multi-threaded environment.

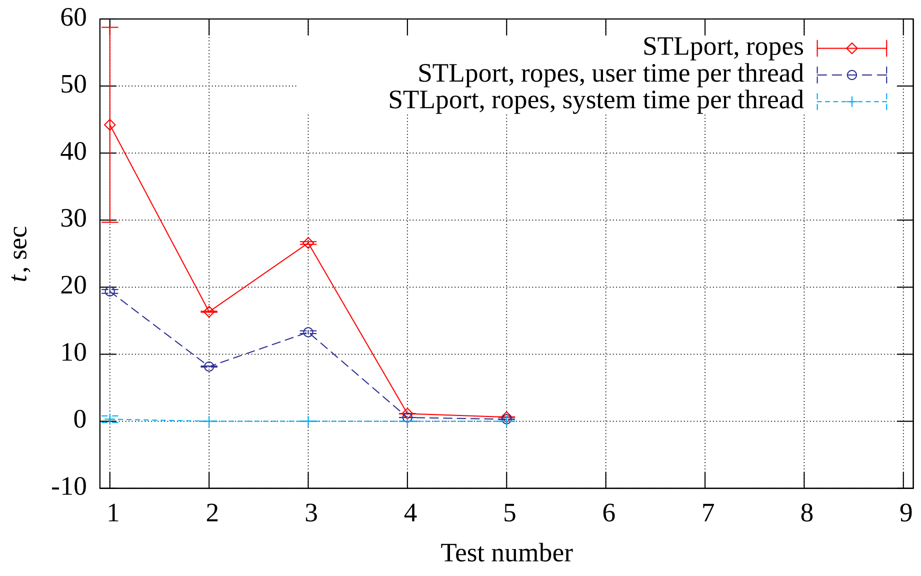

Figure 10: Wall time, user time and system time per thread for STLport ropes in multi-threaded environment.

ferent threads and such access should use thread synchronization primitives. Compare figures 3 and 5. The tests 1–2 has the same time in multi-threaded and single-threaded 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 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  #ifdef 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, "ghfjhfjf" );
31     v.replace( b / 2, 20U, "abcdefg" );
32   }
33 }
```

```

int bs;
int n;
bool use_str = true;

int main( int argc, char * const *argv )
{
  try {
    Argv arg;
    arg.copyright( "Copyright␣(C)␣Petr␣Ovtchenkov,␣2003,␣2005" );
    arg.brief( "Comparison␣of␣ropes␣and␣strings" );
    arg.option( "-h", false, "print␣this␣help␣message" );
    arg.option( "-r", true, "use␣ropes" );
    arg.option( "-s", true,  "use␣strings" );
    arg.option( "-b", 1, "block␣size" );
    arg.option( "-i", 1, "number␣of␣iterations" );
    try {
      arg.parse( argc, argv );
    }
    catch ( std::invalid_argument& err ) {
      cerr << err.what() << endl;
      arg.print_help( cerr );
      throw 1;
    }
    bool turn;
    if ( arg.assign( "-h", turn ) ) {
      arg.print_help( cerr );
      throw 0;
    }

    if ( arg.is( "-r" ) && arg.is( "-s" ) ) {
      cerr << "Either␣-r␣or␣-s␣allowed" << endl;
    }

    arg.assign( "-s", use_str );

    if ( arg.is( "-r" ) ) {
      use_str = false;
    }
    arg.assign( "-b", bs );
    arg.assign( "-i", n );
  }
  catch ( std::runtime_error& err ) {
    cerr << err.what() << endl;
    return -1;
  }
  catch ( std::exception& err ) {
    cerr << err.what() << endl;
    return -1;
  }
  catch ( int r ) {
    return r;
  }

```

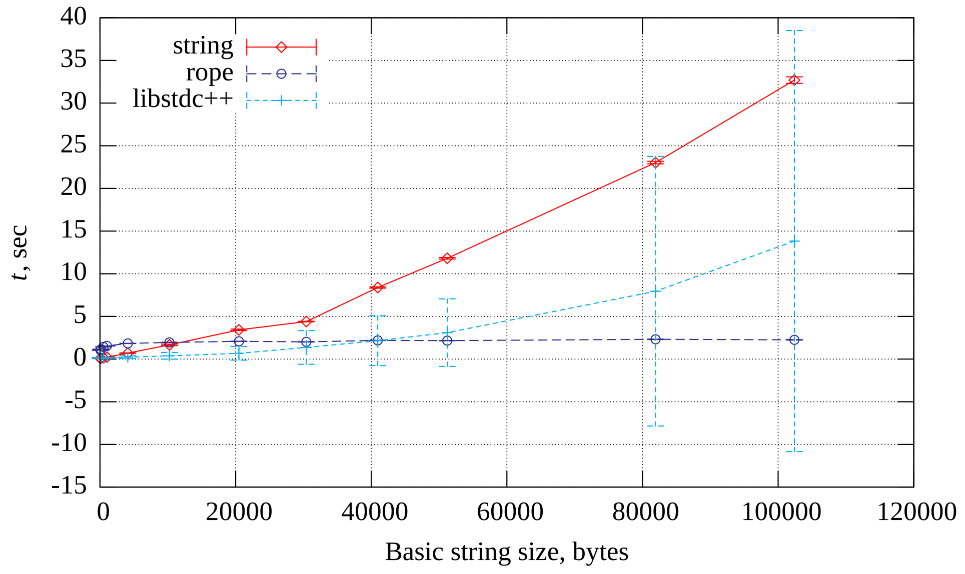

Figure 11: Strings vs. Ropes.

```
  if ( !use_str ) {
#ifdef STLPORT
    test<rope<char> > t( bs, n );
#endif
  } else {
    test<string> t( bs, n );
  }

  return 0;
}
```

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 then 12K, really depends upon compiler's optimization quality), as you can see on figures 11 and 12. The strings implementation in `GNU libstdc++` show better results beginning from size 1K (figures 11–13).

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 (interger, char and double). Each record terminated by 0 (end-of-string).

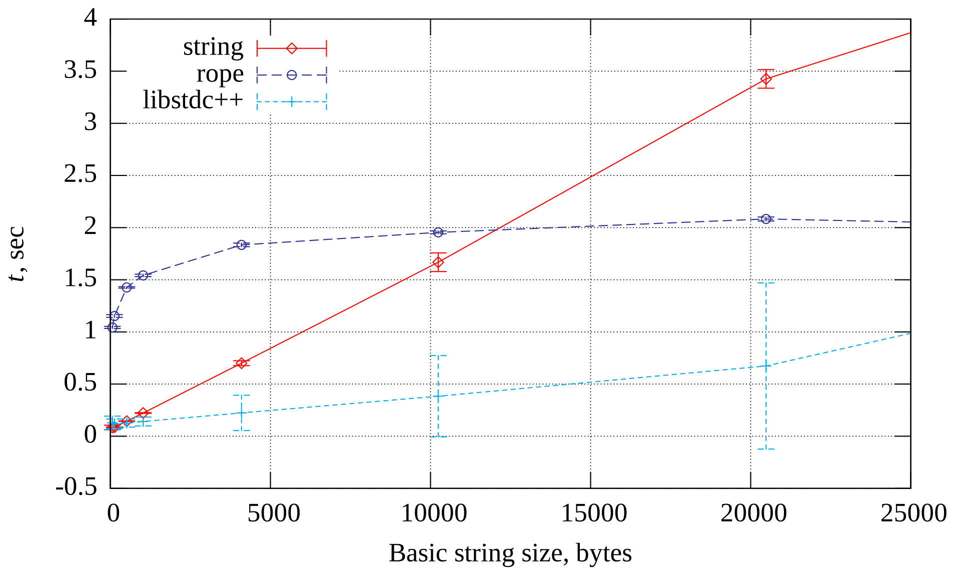

Figure 12: Strings vs. Ropes.

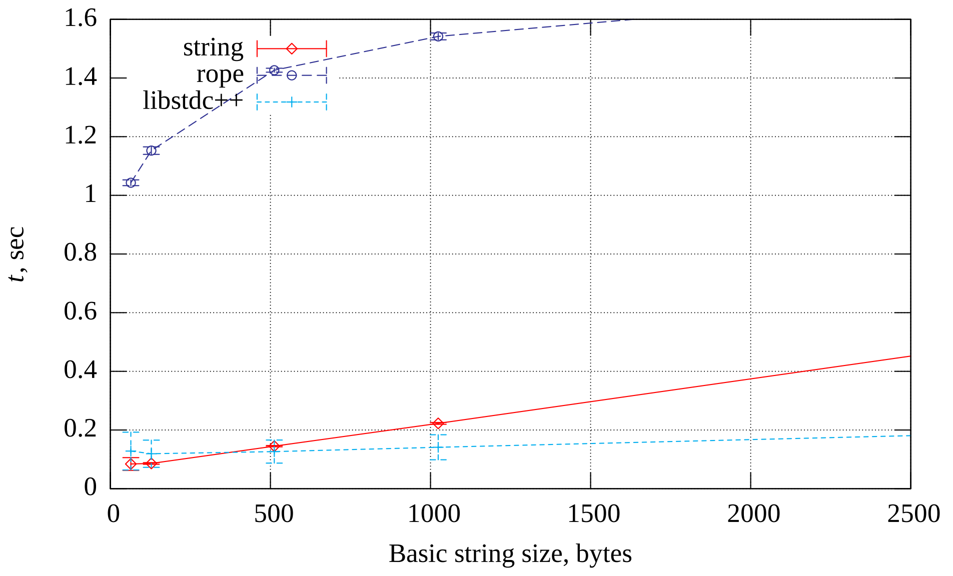

Figure 13: Strings vs. Ropes.

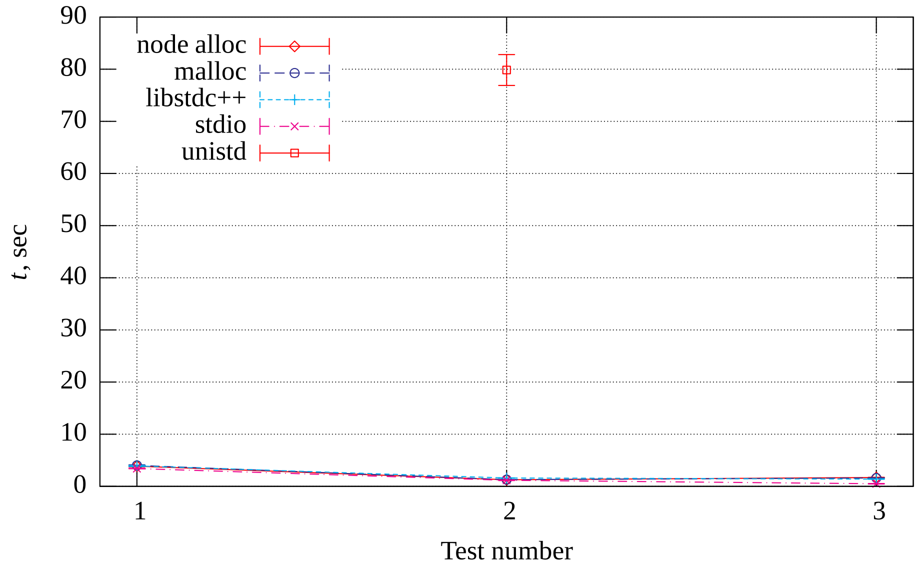

Figure 14: Streams.

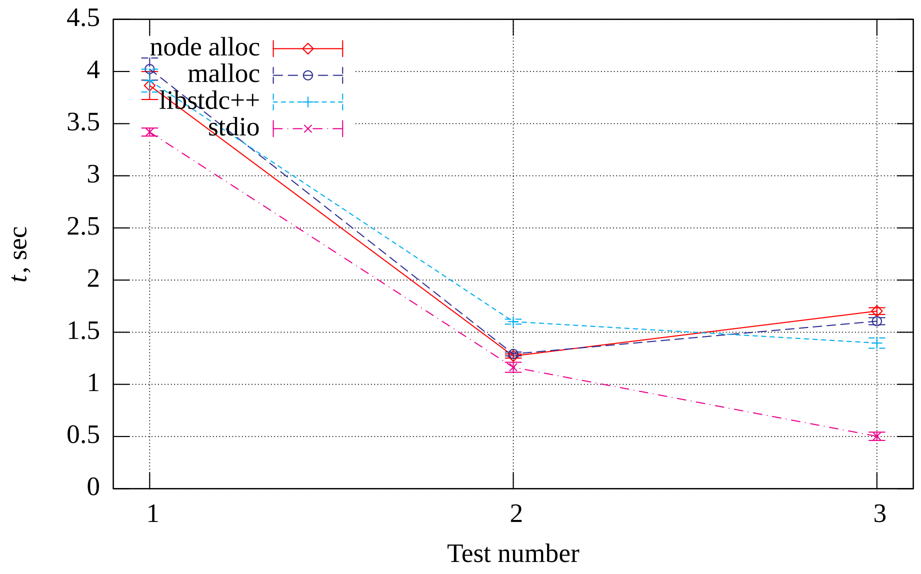

Figure 15: Streams.

```
// -*- C++ -*- Time-stamp: <05/04/27 18:47:41 ptr>

#include <fstream>

using namespace std;

int main( int, char * const * )
{
  ofstream s( "test" );

  for ( int i = 0; i < 1000000; ++i ) {
    s << i << " " << (static_cast<double>(i) + 0.1415926) << ends;
  }

  return 0;
}
```

As reference I use C program (titled as stdio at figure 14) that do near the same.

```
/* Time-stamp: <06/01/04 00:16:20 ptr> */

#include <stdio.h>

int main( int argc, char *argv[] )
{
  FILE *f;
  int i;

  f = fopen( "test", "w" );

  for ( i = 0; i < 1000000; ++i ) {
    fprintf( f, "%d %f\n", i, (double)i + 0.1415926 );
  }

  fclose( f );

  return 0;
}
```

9.2 Raw output to file (test #2)

Raw write to file.

```
// -*- C++ -*- Time-stamp: <05/04/27 18:47:41 ptr>

#include <fstream>

using namespace std;

int main( int, char * const * )
{
  ofstream s( "test" );

  for ( int i = 0; i < 10000000; ++i ) {
    s.write( (const char *)&i, sizeof(i) );
  }

  return 0;
}
```

Near the same test on pure C (stdio).

```
/* Time-stamp: <06/01/04 00:21:07 ptr> */

#include <stdio.h>

int main( int argc, char *argv[] )
{
  FILE *f;
  int i;

  f = fopen( "test", "w" );

  for ( i = 0; i < 10000000; ++i ) {
    fwrite( (const void *)&i, sizeof(i), 1, f );
  }

  fclose( f );

  return 0;
}
```

And, as reference, unbuffered output (unistd):

```
/* Time-stamp: <06/01/04 00:21:07 ptr> */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
  int f;
  int i;

  f = open( "test", O_WRONLY | O_CREAT | O_TRUNC, 0666 );

  for ( i = 0; i < 10000000; ++i ) {
    write( f, (const void *)&i, sizeof(i) );
  }

  close( f );

  return 0;
}
```

9.3 Raw output to string (test #3)

Raw write to string stream.

```
// -*- C++ -*- Time-stamp: <05/04/27 18:16:09 ptr>

#include <sstream>

using namespace std;

int main( int, char * const * )
{
  stringstream s;

  for ( int i = 0; i < 10000000; ++i ) {
    s.write( (const char *)&i, sizeof(i) );
  }

  return 0;
}
```

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

```
/* Time-stamp: <06/01/04 01:31:49 ptr> */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
  char *buf;
  size_t sz = 16;
  int i;

  buf = malloc( sz );

  for ( i = 0; i < 10000000; ++i ) {
    if ( sz < (i * sizeof(i) + sizeof(i)) ) {
      char *tmp;

      tmp = malloc( sz * 2 );
      memcpy( tmp, buf, sz );
      free( buf );
      buf = tmp;
      sz = sz * 2;
    }
    memcpy( buf + i * sizeof(i), &i, sizeof(i) );
  }

  free( buf );

  return 0;
}
```

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++` (C `fwrite` show best results here, but breakaway is minimal; all has huge advantage over unbuffered output). With raw write to string stream (or in char buffer in case of C) the pure C variant is 3–4 times faster, but this not what iostreams was intended for.

10 Conclusions

This tests show that

- for processing long strings (greater than 50K) the best choice is ropes;
- if you use strings with sizes 0.5K–50K then strings implementation from native `GNU libstdc++` is a good choice; this will be good too if you only pass strings as parameters, without modifictions (but this is bad programming technique nevertheless);
- if you general work is modification of strings with sizes less then 0.5K, the strings from STLport are for you;
- the time of advantage of `node_alloc` in STLport is in the past; progress in allocation algorithms in core system eliminate positive effect of `node_alloc` in single-threaded applications and demonstrate significant advantage of core system allocators in multi-threaded applications;
- no valuable performance advantage of C format output over C++ iosreams;
- unbuffered output has ugly performance (is it new fact for you?);
- no reasons to use `node_alloc` in STLport—`malloc_alloc` show better (or significant better) results;
- string proxy object (aka expression template) technique is very useful.

11 References

STLport `http://stlport.sourceforge.net`

GCC `http://gcc.gnu.org`

SGI `http://www.sgi.com/tech/stl/`

Complement `http://complement.sourceforge.net`