Fast c++ String implementation

Prepared by: Waseem Abu Moch
Ran Zeller

Supervisor: Dmitri Perelman
Fast c++ String implementation

Contents

1. INTRODUCTION ....................................................................................................... 2
2. PROJECT GOALS .................................................................................................... 2
3. REPRESENTATIONS AND ALGORITHMS ............................................................... 3
  3.1 EQUALS OPERATION ........................................................................................... 4
  3.2 SUBSTRING OPERATION (SUBSTR) ................................................................. 5
    3.2.1 Boyer Moore Algorithm ............................................................................. 6
    3.2.2 Suffix Tree .................................................................................................. 10
  3.3 STRING REPOSITORY .......................................................................................... 12
4. STRING PROFILING FRAMEWORK ........................................................................ 14
  4.1. STRING PROFILER DESIGN ............................................................................. 14
  4.2. STRINGS TYPES .............................................................................................. 15
  4.3. THREADING ..................................................................................................... 18
  4.4. THREAD_PROFILER .......................................................................................... 19
    4.4.1. STRING_PROFILER_CLASS_DIAGRAM ................................................ 22
5. RESULTS ................................................................................................................ 23
  5.1. SINGLE THREAD RESULTS .............................................................................. 23
    5.1.1. EQUALS PERFORMANCE .......................................................................... 23
    5.1.2. SUBSTR PERFORMANCE .......................................................................... 24
    5.1.3. CONSTRUCTION PERFORMANCE ............................................................ 26
  5.2. MULTI THREAD RESULTS ............................................................................... 27
    5.2.1. CONSTRUCTION CONCURRENCY LEVEL .................................................. 27
    5.2.2. CONSTRUCTOR THREAD PROFILING ...................................................... 29
6. CONCLUSIONS ....................................................................................................... 33
7. SOURCE CODE PACKAGE ..................................................................................... 35
8. REFERENCES ........................................................................................................ 36
1. Introduction

A string datatype is a datatype modeled on the idea of a formal string. Strings are such an important and useful datatype that they are implemented in nearly every programming language. In some languages they are available as primitive types and in others as composite types.

Strings have a big impact on applications performance especially the text applications such: Editors, Web Browsers, and Document Viewers....

A lot of operation can be done on strings such: concatenation, lexical comparing, splitting and searching for substring ..., Most of those operations can be performed efficiently with simple algorithms and a trivial data structure representation, the most common used operations are “equals” and “subStr”.

In this project we will produce different string representations for the C++ programming language and different algorithms in order to optimize the most common operations running time and reduce the amount of memory consumed by the strings, we will compare between them and examine when we should use each one of these representations and algorithms.

2. Project Goals

- Implementing a C++ fast strings library (Linux).
- Design and implementing a string profiling frame work.
- Using the string profiler to compare between the different strings representations and searching algorithms.
- Estimating the parallelism level of each string representation.
3. representations and algorithms

Almost in all programming languages the string data type is represented as an array of chars.

In this representation the amount of memory consumed by a given string is minimal (without compression methods) and a trivial string operation can be executed quickly on small strings, so why to use other representations and what other representations can we use?

In a lot of test applications e.g. text editors, test viewers and web browsers the amount of string object is relatively large, and the size of the used strings is also big, thus the amount of memory consumed by each string is very large and the trivial operations on those strings works slowly.

In this project our focus was on the memory consumption issue and the “equals” and “subStr” execution time.

Assumption:

Given string A which its length is n.

Given string B which its length is m.

m > n.
3.1 equals operation

The time complexity of the simple “equals” operation between A and B is $O(\min(n,m))$ since we must compare each char of these two strings until we meet a different chars or one of the strings is shorter than the other or if the two strings are equal and all chars are the same.

So to optimize the time complexity of this operation we used the commonly used method in OOP languages “Object Repository”, all the strings in the system will be saved in a single thread safe repository and will be entered to the repository by the string constructor and removed from it by the destructor, the repository must follow the rule: two strings shares the same entry in the repository if and only if the strings are equal.

In this method we can compare the addressed of the string object and they are equal if and only if the addresses are equal, thus we optimized the comparison time from $O(n)$ to $O(1)$.

Pros and Cons of using string repository:

1. Time complexity is minimal $O(1)$. ☺
2. Memory consumption is lower than individual string object since equal strings share the same memory region. ☺
3. Sharing the same memory reduces the parallelism since operations on the same repository entry must be synchronized. 😞
4. Complicated implementation. 😞
3.2 substring operation (subStr)

Substring operation is the operation of searching a given string A inside another given string B, the operation must find the index of the first char of substring in B that matches A.

The native algorithm to find substring index on array based strings (search for A inside B) performs $n \times m$ char comparisons to find the first substring in the worst case, it searches for A inside B[0…] and if it fails it searches for A inside B[1…] …

C example of such algorithm:

```c
int subStr(char* A, char* B)
{
    int i;

    for (i=0; i < strlen(B); i++)
    {
        if (!strcmp(A, B +i))
            return i;
    }

    return -1;
}
```

In the worst case A is not a substring of B, we perform strlen(B)=m iteration, in each iteration we call strcmp which runs in $O(\min(n,m)) = O(n)$, thus total run time is $O(n \times m)$.

We can see that if $n$ or $m$ is large the algorithm runs slowly.

To optimize the substring operation we used two methods:

1. Using “boyer moore” substring search algorithm with char array based strings.
2. Represent string as a “suffix tree”, and substring search becomes a tree traversal.
3.2.1 Boyer Moore Algorithm

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the good-suffix shift and the bad-character shift.

bad-character shift

The bad-character shift table is easy to calculate: Start at the last character of the sought string and move towards the first character. Each time you move left, if the character you are on is not in the table already, add it; its Shift value is its distance from the rightmost character. All other characters receive a count equal to the length of the search string.

Example: For the string ANPANMAN, the bad-character shift table would be as shown (for clarity, entries are shown in the order they would be added to the table): (The N which is supposed to be zero is based on the 2nd N from the right because we only calculate from letters m-1)

<table>
<thead>
<tr>
<th>Character</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>P</td>
<td>5</td>
</tr>
<tr>
<td>all other characters</td>
<td>8</td>
</tr>
</tbody>
</table>


**good-suffix shift**

The **good-suffix shift** table: for each value of \( i \) less than the length of the search string, we must first calculate the pattern consisting of the last \( i \) characters of the search string, preceded by a mis-match for the character before it; then we initially line it up with the search pattern and determine the least number of characters the partial pattern must be shifted left before the two patterns match. For instance, for the search string ANPANMAN, the table would be as follows: (N signifies any character that is not N)

<table>
<thead>
<tr>
<th>( i )</th>
<th>Pattern</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>AN</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>MAN</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>NMAN</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>ANMAN</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>PANMAN</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>NPANMAN</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>ANPANMAN</td>
<td>6</td>
</tr>
</tbody>
</table>

Assume that a mismatch occurs between the character \( x[i]=a \) of the pattern and the character \( y[i+j]=b \) of the text during an attempt at position \( j \). Then, \( x[i+1 .. m-1]=y[i+j+1 .. j+m-1]=u \) and \( x[i] \neq y[i+j] \). The good-suffix shift consists in aligning the segment \( y[i+j+1 .. j+m-1]=x[i+1 .. m-1] \) with its rightmost occurrence in \( x \) that is preceded by a character different from \( x[i] \) (see figure 1).

![Figure 1. The good-suffix shift, u re-occurs preceded by a character c different from a.](image-url)
If there exists no such segment, the shift consists in aligning the longest suffix $v$ of $y[i+j+1 \ldots j+m-1]$ with a matching prefix of $x$ (see figure 2).

![Figure 2. The good-suffix shift, only a suffix of $u$ re-occurs in $x$.](image)

The bad-character shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0 \ldots m-2]$. (see figure 3)

![Figure 3. The bad-character shift, $a$ occurs in $x$.](image)

If $y[i+j]$ does not occur in the pattern $x$, no occurrence of $x$ in $y$ can include $y[i+j]$, and the left end of the window is aligned with the character immediately after $y[i+j]$, namely $y[i+j+1]$ (see figure 4).

![Figure 4. The bad-character shift, $b$ does not occur in $x$.](image)
Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.

The good-suffix shift function is stored in a table $bmGs$ of size $m+1$.

Let us define two conditions:

- $Cs(i, s)$: for each $k$ such that $i < k < m$, $s$ $k$ or $x[k·s] = x[k]$ and
- $Co(i, s)$: if $s < i$ then $x[i·s] 1 x[i]$

Then, for $0 \leq i < m$: $bmGs[i+1] = \min\{s > 0 : Cs(i, s)$ and $Co(i, s)$ hold$\}$

and we define $bmGs[0]$ as the length of the period of $x$. The computation of the table $bmGs$ use a table $suff$ defined as follows: for $1 \leq i < m$, $suff[i] = \max\{k : x[i·k+1 .. i] = x[m·k .. m-1]\}$

The bad-character shift function is stored in a table $bmBc$ of size $\sigma$. For $c$ in $\Sigma$: $bmBc[c] = \min\{i : 1 \leq i < m-1$ and $x[m-1·i] = c\}$ if $c$ occurs in $x$, $m$ otherwise.

Tables $bmBc$ and $bmGs$ can be precomputed in time $O(m + \sigma)$ before the searching phase and require an extra-space in $O(m + \sigma)$. The searching phase time complexity is quadratic but at most $3n$ text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for a $m_{-1b}$ in $b^n$ the algorithm makes only $O(n / m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.
3.2.2 Suffix Tree

Suffix tree (also called suffix trie, PAT tree or, in an earlier form, osition tree) is a data structure that presents the suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations.

The suffix tree for a string S is a tree whose edges are labeled with strings, and such that each suffix of S corresponds to exactly one path from the tree's root to a leaf. It is thus a radix tree (more specifically, a Patricia trie) for the suffixes of S.

Constructing such a tree for the string S takes time and space linear in the length of S. Once constructed, several operations can be performed quickly, for instance locating a substring in S, locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern etc. Suffix trees also provided one of the first linear-time solutions for the longest common substring problem. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

(Quoted from: http://en.wikipedia.org/wiki/Suffix_tree)

*In our project we used Suffix tree which it’s constructor run in $O(n^2)$ but there exists algorithms to construct the suffix tree in linear time.*

The suffix tree constructs a tree which each node contains one char thus each path from the root node to a leaf node represents one suffix from the original string.

In this representation the tree contains $O(n^2)$ nodes and because this is a tree the number of edges is also $O(n^2)$, each edge contains one char (1) space, thus the required space for the tree is $O(n^2)$.

For example: string = banana:

suffixes:

1-a
2-na
3-ana
4-nana
5-anana
6-banana
To minimize the required space for the suffix tree we used the following compression method:

1. remove nodes with one child.
2. replace chain of edges with one edge which contains the substring.
3. instead of saving the substrings in the edges hold two pointers (start_idx, last_idx) to point to the start and the end of the string.

After compression we get a reduced trie which contains $O(n)$ edges and nodes thus the required space for such trie is $O(n)$.

In the project we used the compressed suffix tree to minimize the required space.

Substring operation on suffix tree is very fast since the substring search becomes tree travel from the root node to one of the leaves (the one which the path to it represents the substring).
Complexity Summary:
Construction time: O(n)
Substring time: O(m), m is the length of the substring (the one we search for).
Space: O(n).

3.3 String Repository

As we mentioned before we use repository to optimize the equals operations and to save memory.

The problem of using repository is the low parallelism level, since different strings in the system holds a pointer to the same objects in the repository updating the repository must synchronized.

The repository is implemented as closed address hash table, each entry in the hash table holds a string object and a reference counter, when entering a new string to the repository it checks if an equal string already exists in the hash table if yes it increases the reference counter of the string entry by 1 and returns a pointer the string object, if not exists it initialize new string object and new hash table entry with reference counter = 1, similar behavior when deleting string object.

Since string objects may be inserted and deleted from the repository it must be a thread safe, different operations on the repository must be synchronized, the synchronization is applied using mutex objects, using one mutex for the whole repository decreases the parallelism level, so we used more than one mutex but again we can use as many mutexes as we want since the OS limit us.
In our implementation we used the following synchronization method:

Mutex is responsible for more than one hash table entry but still we can’t allocate a different mutex for each entry so we defined a mapping function between mutex and its entries.

The following figure demonstrates our repository design, the hash table length is 16 and the number of mutexes is 8, the mutex mapping function is: $f(i\text{-hash table entry}) = i \text{ mod } 8$ (mutex index):

![Diagram showing hash table and mutex mapping]

To ensure hash table search, inserting and deleting high performance each hash table entry length must be short, so the repository performs resize if the average length of the table entries exceeds the predefined Load Factor.
4. String profiling framework

In order to perform performance and parallelism level analysis on the different string representations and algorithms we implemented string profiler.

String profiler can run test cases which can be described using XML file, in the test cases we describe which string implementation to use and how many threads and which methods to run and other properties, its output is the threads running time.

4.1. String profiler design

The string profiler has the following major components:

- **MyString** is the base class of all string types in the system.
- **MyThread** is a class which can run tasks in independent thread of execution.
- **DbXmlParser** is a class which capable of parsing XML files that contains the tests cases.
- **StringProfiler** is the main class the run the tests cases and prints the results output.
Assumption:

Given string A which its length is n.

Given string B which its length is m.

\( m > n. \)

4.2. Strings Types

In our projects we implemented three types of strings and analyzed its performance and parallelism level.

All strings are subclasses of the base string class MyString which provides the following standard API:

- `MyString* clone(const MyString& other)`
- `bool equals(const MyString& other)`
- `bool isEmpty(const MyString& other)`
- `int getLen()`
- `char charAt(int index)`
- `int strCmp(const MyString& other)`
- `int subStr(const MyString& other)`
**MyStringSTL**: this class implements the naive strings representation using array of chars (The C++ STL strings).

- The equals method is implemented using character comparison, so it’s running time is $O(\min(n,m))$.
- The substring method is also implemented using the naïve substring algorithm described before so it’s running time is $O(n*m)$.
- Construction time is $O(n)$ since we need to initialize n characters.
- The required space for the STL string object is $O(n)$.

**MyStringBM**: this class also implements the naive string representation using array of chars but it also holds two pre computed help tables to the Boyer Moore substring algorithm.

One more enhancement in this type of strings is using the string repository to save memory and enhance the equals operation.

- The equals method running time is $O(1)$ since we compare only the addresses (pointers) because of the repository usage.
- The substring method implements the Boyer Moore algorithms so it’s running time is $O(n)$.
- Construction time is $O(n) + \text{time to insert string to repository}^*$, it initializes n characters and $O(n)$ Boyer Moore tables.
- The required space for BM string object is $O(n)$.

**MyStringST**: this class represents the string as a suffix tree to accelerate the substring operation, and it also uses the repository to save memory and accelerate the equals operation.

- The equals method running time is $O(1)$.
- The substring operation running time is $O(m)$ (m is the length of substring we search for).
- Construction time is $O(n^2) + \text{time to insert string to repository}^* – \text{need to build suffix tree}$.
- The required space for the suffix tree is $O(n)$.
* - time to insert string to repository: Calculating the exact number of operation to insert string to repository is difficult, but we can calculate is on average:

Assumptions:

L - The repository pre defined Load Factor.
S - The length of the inserted string.
M – Average length of all strings in the repository.

Since the load factor is L the average length of the linked list of each entry in the hash table is L.

We have to compare the new string with L strings to check if it’s already exists. Each string comparison is O(min(S,M))

Total operations on average: O(L * min(S, M)).

MyString Class Diagram:
4.3. Threading

In order to compare the parallelism level of each string type we must perform different operations on different and same string object simultaneously, but unfortunately standard C++ does not support threading (at least OOP threading).

We implemented our threading library called MyThread,
MyThread is a wrapper class to the known linux threading library POSIX THREADS.

**MyThread class diagram:**

MyThread class constructor accepts Runnable class which is an abstract class that represents a user task.

The user must derive from the Runnable class and implement the run method, the run method is the method which will be ran by the MyThread object in a separated thread of execution.

MyThread class provides also thread running time.
4.4. thread_profiler

The thread profiler is responsible of running different tests cases “Configurations” and calculates the running time of each thread described in the configuration using XML file.

Configuration File (XML) format:

```
<TEST_PACKAGE>

<test name="name" string_type=MY_STRING_BM iter=100 barrier=1>
    <string>Hello World</string>
    <repeat_string repeat=50>
        <string>Hello Universe</string>
        <rand_string len=100></rand_string>
        <long_rand_string len=100></long_rand_string>
        <rand_container_string len=100>I’m contained</rand_container_string>
    </repeat_string>
    <thread name=thread1 string_idx=0>
        <method method_type=EQUALS iter=10000>
        </method>
        <method method_type=SUB_STR iter=10000 param_2_str_idx=1>
        </method>
    </thread>

    ....
</test>

</TEST_PACKAGE>
```
string_profiler xml TAGS and attributes:

1. **TEST_PACKAGE**: the main TAG, no attributes.
2. **test**: test to run, string profiler runs test sequentially,
   attributes:
   a. string_type: the string type use in this test [MY_STRING_STL, MY_STRING_BM, MY_STRING_ST, MY_STRING_ALL],
      MY_STRING_ALL run 3 tests.
   b. iter: the iteration number.
   c. barrier: use barrier to start threads together.
3. **string**: allocates a string in the test’s string pool.
4. **rand_string**: allocates a random string in the test’s string pool.
   attributes:
   a. len: the maximum length of the random string.
5. **long_rand_string**: the same as rand_string but the attribute len is the exact length of the rand string.
6. **rand_container_string**: allocates a random string in the test’s string pool, the random string will contain the given (as a child xml node) substring.
   attributes:
   a. len: the length of the random string.
7. **repeat_string**: repeat allocating the strings inside it.
   attributes:
   a. repeat – number of repetitions.
8. **thread**: thread descriptor.
   attributes:
   a. name – name of the thread.
   b. string_idx – string pool index (which string to work on)
9. **method**: method to run:
   attributes:
   a. method_type: the method to run, possible values:
   [ CTOR, COPYCTOR, EQUALS, IS_EMPTY, GET_LEN, CHAR_AT, STR_CMP, SUB_STR,
C_STYLE_STRING,
OPERATOR_EQUALS,
OPERATOR_LESS,
OPERATOR_GREATER,
OPERATOR_LESS_EQUALS,
OPERATOR_GREATER_EQUALS

b. iter – num of iteration to run this method.

c. param_2_str_idx: the index of the second string (some method works on two strings e.g SUB_STR).

The string_profilers loads the required configuration from the XML file, the DbXmlParser parse the xml file and fills a struct which contains the user configuration.

**Configuration Descriptor structs:**

```c
struct method_desc_t
+method: enum method_enum_t
+param2_pool_index: unsigned int
+other_param: int
+num_of_iterations: unsigned long
```

```c
struct thread_desc_t
+thread_name: string
+string_from_pool_index: unsigned int
+method_desc_array[MAX_METHODS]: struct method_desc_t
```

```c
struct test_desc_t
+test_name: string
+impl_type: enum string_impl_enum_t
+string_pool_len: unsigned int
+string_pool[MAX_STRING_POOL]: const char*
+num_of_threads: unsigned int
+thread_desc_array[MAX_THREADS]: struct thread_desc_t
+num_of_iterations: unsigned int
+use_barrier: bool
```

```c
struct test_package_t
+num_of_tests: unsigned int
+tests[MAX_TESTS]: struct test_desc_t
```
MAX_TESTS: 20
MAX_THREADS: 10
MAX_METHODS: 20
MAX_STRING_POOL: 100

4.4.1. StringProfiler class Diagram

* Run string_profiler –h for more information about the string_profiler capabilities.
5. Results.

We prepared a set of XML configuration files to compare between the three different string types.

we have separated the results reports to single thread and multi thread reports.

Note – We did all the measurements on Intel’s machine x86 with 4 cores.

5.1. Single thread results

5.1.1. Equals Performance

The test cases we prepared to compare equals method performance of each implementation in single thread mode contains one thread which runs the “EQUALS” method on one string, the length of the string grows from 10 – 10000.

Note – y axis scale is logarithmic.

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL</td>
<td>0.83</td>
<td>1.043</td>
<td>2.85</td>
<td>21.769</td>
</tr>
<tr>
<td>BM</td>
<td>0.39</td>
<td>0.483</td>
<td>0.389</td>
<td>0.459</td>
</tr>
<tr>
<td>ST</td>
<td>0.395</td>
<td>0.397</td>
<td>0.467</td>
<td>0.398</td>
</tr>
</tbody>
</table>
We can see that as we expected the running time of equals method of MY_STRING_BM and MY_STRING_ST is a constant and not affected by the length of the strings and that because of the repository usage (pointers comparison), and it’s running time is faster than the MY_STRING_ST which grows linearly with the length of the strings.

Conclusion:

→ BM and ST running time is the same O(1) and much better than the STL O(n).

5.1.2. SubStr Performance

The test cases we prepared to compare subStr method performance of each implementation in single thread mode contains one thread which runs the “SUB_STR” method on one string, the length of the string grows from 10 – 10000.

And we searched for a constant length substring (14 characters).

We can see that the MY_STRING_ST running time is the lowest starting from string length larger than 1000 and it’s constant and that’s because the running time of substring operation on suffix tree is O(m) while m is the length of the substring, in our case the substring is constant and it’s length is 14.
The MY_STRING_STL is the worst for strings larger than 100 characters and that because of the naïve algorithm of the substring it uses $O(n^2)$, but we can see that for small string ~10 characters it’s faster than the other string types.

The MY_STRING_BM running time is much better than STL almost for every string length and better than ST for small strings shorter than ~100 characters, but for large strings the ST is best.

**Conclusion:**

- for large strings $> 100$ ST is better than BM better than STL.
- for ~100 char strings: BM is better than ST better than STL.
- for small strings $< 100$: STL is better than BM better than STL.

We saw that BM and ST is much better than ST in both equals and subStr methods, so why to use STL anyway?

**Answer:** Construction Time.
5.1.3. Construction Performance

The test cases we prepared to compare constructor method performance of each implementation in single thread mode contains one thread which runs the “CTOR” method on one string, the length of the string grows from 10 – 10000.

As we expected construction of STL string is much faster than BM and ST, since it initialize n characters O(n).

BM is worse than STL because it initializes n characters and two Boyer Moore algorithm help tables, but the big in running time difference is because of the repository insertion time. O(L*min*n,m))

ST is the worst and that is reasonable, our constructor implementation to build suffix tree takes O(n^2).
5.2. Multi thread results

On our single thread analysis we saw that except of construction time the BM and ST string types are much better than STL especially for large strings.

But to provide a thread safe api we used mutexes, but as we all know: mutex = low parallelism level.

We should mention that the only problematic method in parallelism aspect is the constructor since the string object is immutable so it’s not risky that more than one thread performs equals or substring or any other method on the same string object.

To examine the parallelism level of each string type we used Inter Thread Profiler tool which helped us to collect thread running information and wait frequency between different thread.

5.2.1. Construction Concurrency Level

To compare concurrency level and wait frequency of the different strings types we have prepared XML configuration files that run 4 threads that create strings, after collecting information using Intel Thread Profiler we got the following concurrency results:

- STL is almost have full concurrency level against number of lock (STL is not actually contains locks)
- BM have very low concurrency level when using 1 lock, as the lock num grows the concurrency level becomes better since each lock becomes responsible to small amount of entries in the hash table.
We expected that ST and BM behave the same but we noticed that most of the time the threads are working on the suffix tree construction and the repository insertion is not the bottleneck.

**Conclusion:**

STL concurrency level is the highest and BM is better than the ST despite of the high level of concurrency of the ST and that because ST constructor is very slow and the lock waits are not the bottleneck.

We can see that the wait frequency of ST and BM are similar despite of the difference in the concurrency level, that’s because both are using the same repository and wait frequency does not depend on the string construction time:
5.2.2. Constructor Thread profiling

To analyze the concurrency level of the constructor of each string type, we prepared an XML configuration files which runs 4 threads that creates strings, we performed this test 4 times each time with a different number of repository locks.

STL:

We can see that the STL strings have full concurrency (4 threads except of the string_profiler initializing time).
We can see that for one lock the concurrency is almost 1 all the time and we see a lot of “red” (under utilized) and the threads waits for each other most of the time.

For 4 lock we start to get fully utilized periods, but still there are a lot of not under utilized periods.

Starting from 10 locks we get fully utilized all the time, it means that having 10 or more locks the repository starts to have concurrency level 4 and BM starts to acts exactly as STL.
Here we can see that even we use the same repository as BM one lock give as almost full concurrency and thread almost don’t wait for each other, again as we explained before that’s because the repository is not the bottleneck in ST implementation since the thread spend most of the time in suffix tree construction and repository insertion.

Also here we can see that starting of 10 locks we have full concurrency.

To compare the three strings implementation we need to run a read applications on our strings, but since the string_profiler is not capable of that, we tried to emulate a real text application using our XML configuration system.

We prepared an XML configuration file which runs 4 threads, each thread works on one long string (1000 characters) and performs a lot of “equals” and “subStr” operations.
After running our `string_profiler` using Intel Thread Profiler we got the following results:

![Chart showing performance comparison]

We can see that ST is much faster than BM, and BM is much faster than STL.
6. Conclusions

After having all the results, performance analysis and concurrency graphs we can conclude that each string type has advantages and disadvantages, and we should use the best implementation for our application.

The most important thing is to know our application behavior:

1. single or multi threaded.
2. uses a lot of short strings or a little of long ones.
3. performs a lot of equals or subStr operations.
4. memory is critical or we have a large RAM.

Single threaded applications:

We saw that BM and ST have a limited concurrency level when using reduced number of locks, but in single thread application this is not issue since we have a single thread.

The constructor of STL is faster and if we create and destroy a lot of string objects in our application we may consider using STL strings since its constructor is much faster than BM and ST.

If our application creates a small number of strings but performs a lot of “equals” and “subStr” operations we should use BM or ST, if the strings are long the ST gives better subStr performance than the BM.

If our application is designed for embedded system which have a limited memory amount the best choice may be the BM since it much better than STL (equals and subStr is much faster than STL) and it uses repository so it may save memory (two or more equal string share the same memory), and BM construction time is much better than ST.

If our application is text viewer like (pdf viewer, web browser…) the best choice is ST since we create the string only one and then we perform a lot of text search inside the document and STL gives the best performance of subStr (searching).
Multi Threaded application:

If our application is multithreaded we should consider using a large number of locks (~ 2*number of threads).

If the application creates a reduced number of strings we should use BM or STL, since the parallelism issue is only on the construction, so if we creates and destroy a small number of strings the concurrency is not a big issue.

If the application creates and destroy a lot of strings we may prefer to use STL since it’s constructor is the fastest and it’s concurrency is full.

If the application performs a lot of equals and subStr operations we should use ST since it has the fastest subStr operation and it’s worthy to use a lot of locks.
7. Source Code Package

The project code can be downloaded from EE networked software lab site: http://softlab.technion.ac.il
or from our svn repository: http://svn.assembla.com/svn/free_svn/projectB

7.1. Source Code tarball contents

_root folder:_

db: the base directory of the XML configuration files.

_Documentation:_ contains project documentation and presentations.

_Misc:_ scripts to run intel thread profiler.

_Uml:_ contains class diagrams of the string profiler.

_Src: my_string:_ the source code of the 3 strings implementation, can be compiled using make and builds a strings library.

_my_thread:_ the source code of the my_thread packaged, can be compiled using make and builds an OOP threads library

_string_framework:_ the string_profiler source code, can be compiled using make.

_utils:_ contains general usefully functionality implemented in header file.

_makefile:_ makefile for the whole project, compiles the string and threads libraries then compiles and links the string_profiler, the executable “string_profiler” will be located at _root_folder/bin/string_profiler._
8. References

- XML tutorial –
  - http://www.w3schools.com/XML/xml_whatis.asp
- Xml Parsing Lib– tinyxml
  - www.sourceforge.net/projects/tinyxml
- Command Line parsing Lib- anyoption
  - http://www.hackorama.com/anyoption/
- Boyer Moore Algorithm
- Suffix Tree-
- STL strings-
- Intel Thread Profiler-
- The Art Of Multiprocessor Programming book by Herlihy & Shavit- chapter 13.1 Concurrent Hashing and Natural

“We hope you enjoyed reading our project.”