# Data Structures Libraries

Leonor Frias Moya

*Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya*

Advisors: Jordi Petit Silvestre and Salvador Roura Ferret

Supported by: ALINEX project, ALBCOM research group, and *Agència de Gestió d'Ajuts Universitaris i de Recerca*

8th June 2010

# Outline

This thesis contributes **specialized algorithms and data structures** for:

- The Standard Template Library
- Current computer architectures
- Strings

Type of contributions:

- **Theoretical** (analysis of algorithms)
- **Engineering** (implementations)
- **Experimental** (evaluation of implementations)

# The Standard Template Library (STL)

```
#include <string>
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main() {
    vector<string> v;

    string s;
    while (cin >> s) v.push_back(s);

    sort(v.begin(), v.end());

    vector<string>::iterator it;
    for (it = v.begin(); it != v.end(); ++it) {
        cout << *it << endl;
    }
}
```

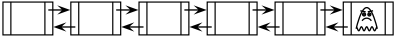
STL elements:

- Containers
- Algorithms
- Iterators

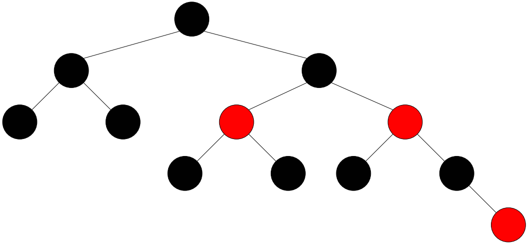
# Typical implementations

`vector`:

`list`:



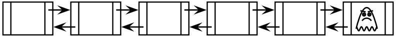
`(multi)map`, `(multi)set`:



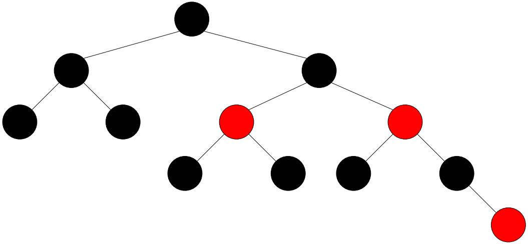
# Typical implementations

`vector`:

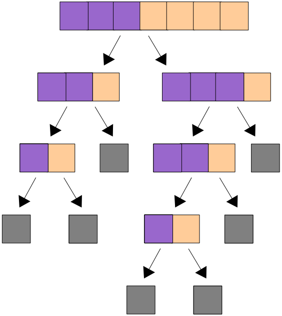
`list`:



`(multi)map`, `(multi)set`:



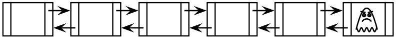
`sort`: $\Theta(n \log n)$



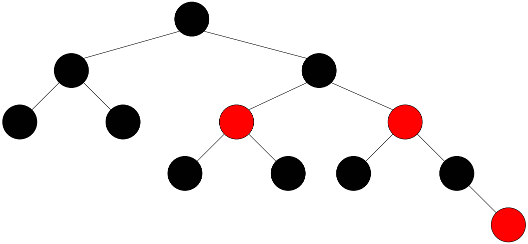
# Typical implementations

vector:

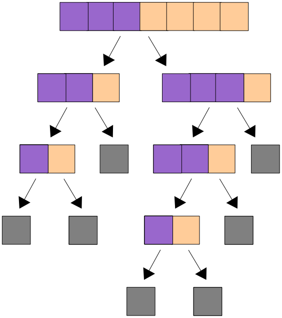
list:



(multi)map, (multi)set:



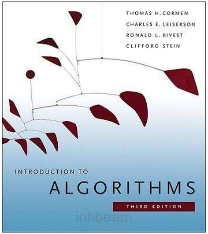
sort: $\Theta(n \log n)$

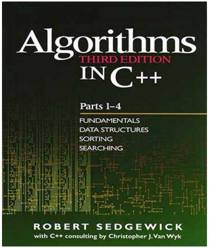


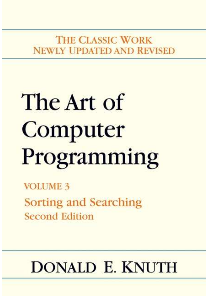
iterators:

# STL specification

Foundations of former implementations can be found in:



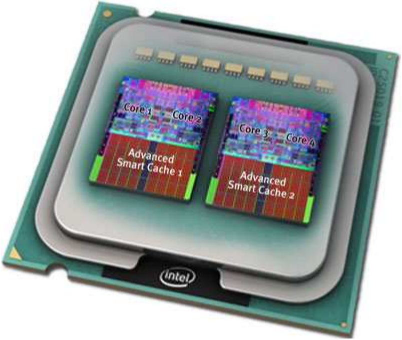



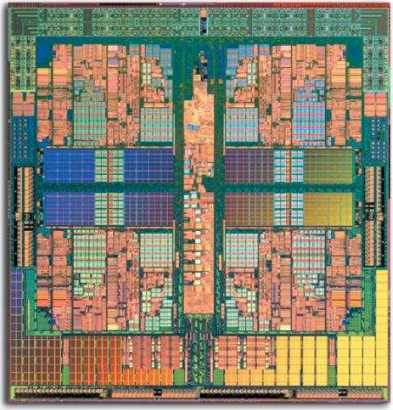




Standard cost requirements are based on those algorithms and data structures.

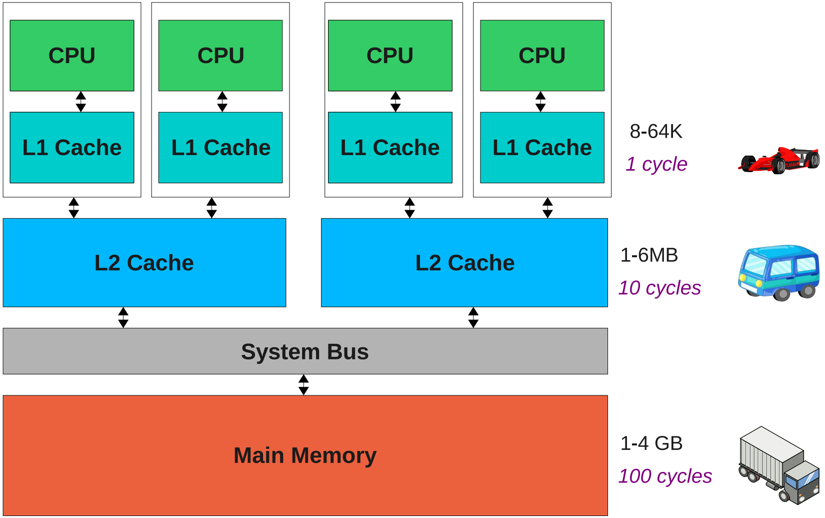
- **Random Access Machine model:** 1 CPU, 1 memory level
- **Generic** atomic keys

# Current computers: multiprocessors

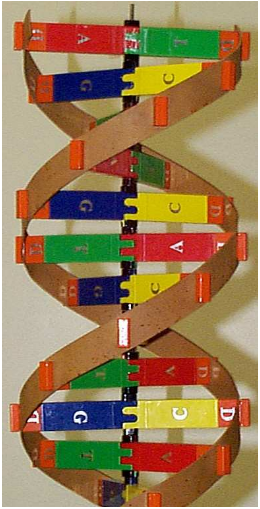


# Modern computer architectures



# An ubiquitous data type: strings



π Я 音 æ∞

# Algorithm engineering

Bring together **theory** and **practice** in algorithmics.

- Focus: implementations and experiments

Several **conferences, journals and books** devoted.

- E.g., ALENEX, SEA (WEA), ESA, JEA

**STL projects**:

- STL-XXL, Uni Karlsruhe
- MCSTL, Uni Karlsruhe
- STAPL, Texas A&M University
- CPH-STL, Performance Engineering Laboratory

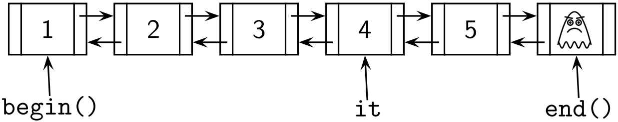
# Contributions

- Cache-conscious STL lists 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- Single-pass list partitioning 
- Parallel partition: 
    - Generic
    - String keys $\Sigma*$ 

# Contributions: Chapter 3

- **Cache-conscious STL lists** 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- Single-pass list partitioning 
- Parallel partition: 
  - Generic
  - String keys $\Sigma*$ 

# STL lists

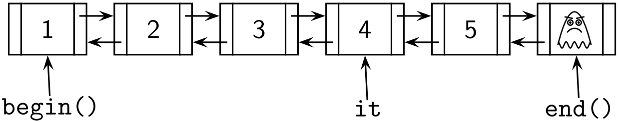



Properties:

- Perfect costs: $\Theta(1)$ insertion/deletion
- Resistant **iterators**.

**What can we improve?**

# STL lists



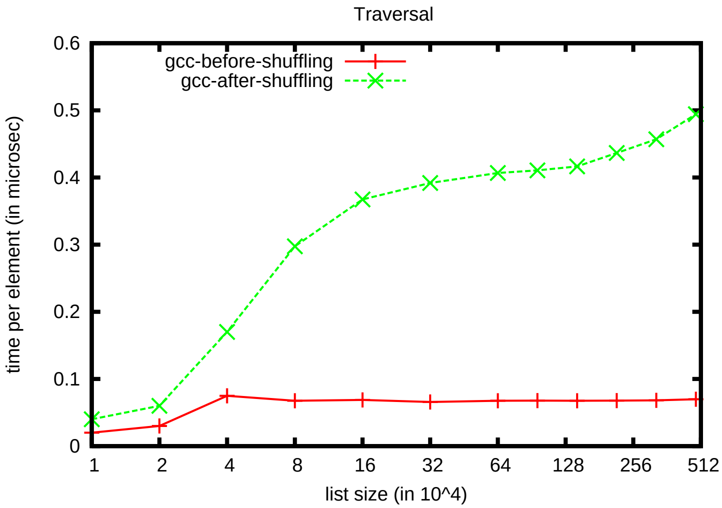

Properties:

- Perfect costs: $\Theta(1)$ insertion/deletion
- Resistant **iterators**.

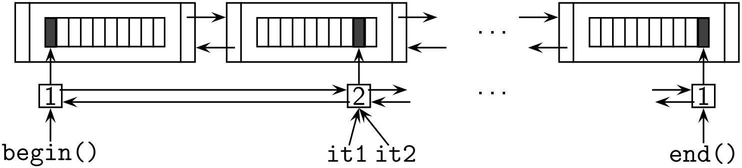
**What can we improve? Cost constant factors**

- **Our approach:** cache-conscious design
  (Lamarca 1996; Frigo et al. 1999; Demaine, 2002)

# Effect of the memory hierarchy



# Cache-conscious STL lists



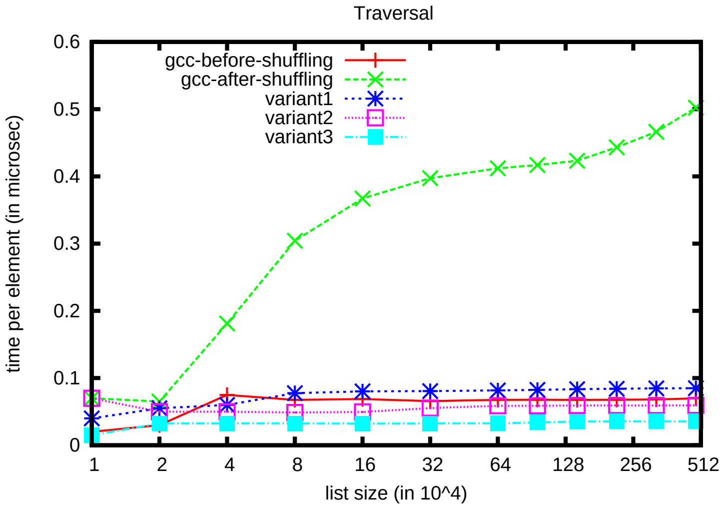

**Main point:** resistant iterators

**Several variants**

**Some assumptions for best performance:**

- "Small" number of iterators
- Usage: Mainly traversals + modifications at arbitrary points
- Plain data types

# Traversal after shuffling



# Theoretical guarantees of the reorganization algorithm

1. A **minimal** average **bucket occupancy** of 2/3.
2. **Efficient bucket management.**

**Theorem**

Let a list conform to the representation invariants. Consider an arbitrary long alternating sequence of insertions and deletions at the same point. Then, at most 2 buckets are allocated and deallocated.

**Theorem**

Let $L$ be an empty list, let $K$ be the bucket capacity. Consider a sequence of $r$ insertions and/or deletions at arbitrary positions applied to $L$. Then, the number of allocated and deallocated buckets is $O(r/K)$.

# Conclusions

Several variants of **cache-conscious** and **compliant** lists.

**Amortized analysis** of the reorganization algorithm.

Thorough **experimental** analysis:

- Traversal: x5-10 faster
- Sort: x3-5 faster
- Competitive even for **big iterator loads**.
- **bucket capacity** $K \in [10, 100]$: not critical

# Publications

L. Frias, J. Petit, and S. Roura. Lists Revisited: Cache Conscious STL Lists. In **WEA 2006**, volume 4007 of *LNCS*. Springer.

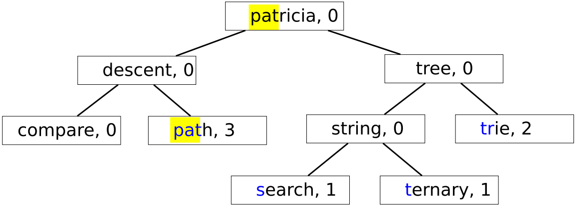
L. Frias, J. Petit, and S. Roura. Lists Revisited: Cache Conscious STL Lists. **JEA**, 14:3, 2009.

**Code at SourceForge.net**:
`http://sourceforge.net/projects/cachelists`

# Contributions: Chapter 4

- Cache-conscious STL lists 
- **Analysis of string lookups in aBSTs** $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- Single-pass list partitioning 
- Parallel partition: 
    - Generic
    - String keys $\Sigma*$ 

# Enhanced BSTs for strings: ABSTs




**Key idea:** keep combinatorial properties +
avoid character comparisons based on comparisons order.

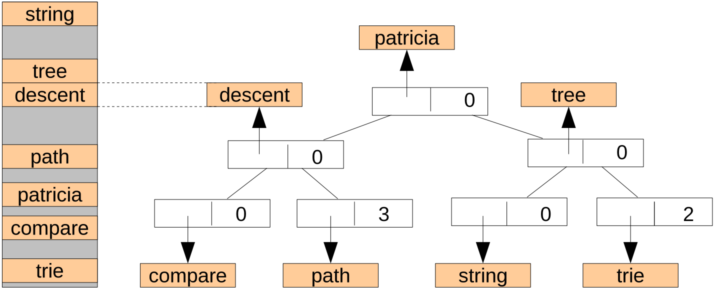
**Generalizable** techniques (Grossi and Italiano, 1999; Roura, 2001):
e.g., quicksort (AQSort) and quickselect (AQSel).

Amenable for **specializing STL components** for strings.

# String lookups in ABSTs

**Observation:** some comparisons do not need accessing the strings.

This is **relevant for cache performance:** strings are accessed through pointers (very likely cache misses).

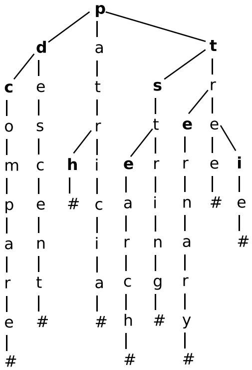


# Analysis of string lookups in ABSTs

**Key:** relationship with TST properties.

**Theorem**

Let $t$ be a TST and let $b$ be an equivalent ABST. Let $w$ be any string. Then, the number of string lookups in $b$ when searching for a string $w$ coincides with the number of search descent paths in $t$ when searching for $w$.



# Some concrete results

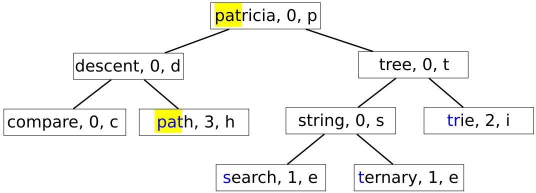
**Exact analysis on TSTs:** derived from previous results on TSTs (Clément, Flajolet, et al., 2001).

**Corollary**

Let $t$ be a TST and let $w$ be a string. The number of search descent paths in $t$ for $w$ is equal to $R(t, w) + 1$.

The **number of string lookups in aBSTs** is reduced by a **constant factor** for memoryless and Markovian distributions.

# Extension: CaBSTs




**Key idea:** Avoid string lookups (cache misses) using **redundancy**.

Applicable also to **quicksort** and **quickselect.**

# Analysis of string lookups in CaBSTs

Relationship with **TSTs**:

**Theorem**

Let $t$ be a TST, let $\beta$ be an equivalent CaBST, let $w$ be any string. The number of proper search descent paths in the searching path of $t$ for $w$ coincides with the number of strings looked up in $\beta$ when searching for $w$.

Relationship with **Patricia tries**:

**Corollary**

The number of strings looked up in CaBST $\beta$ when searching for any string $w$ is upper bounded by the search cost in a Patricia trie storing the same set of strings.

# Conclusions

Analysis of the number of string lookups in (C)aBSTs, (C)aQSort, (C)aQSel relating them to TSTs.

Concrete results for **aBSTs** and **aQSort** for some string distributions.

Follow-ups:

- **CaBSTs** on red-black trees for STL `map` (Master thesis of F. Martínez, 2009)
- **(C)aQSort** and **(C)aQSel**: Chapter 9

# Publications

L. Frias. On the number of string lookups in BSTs (and related algorithms) with digital access. Technical report LSI-09-14-R, 2009.

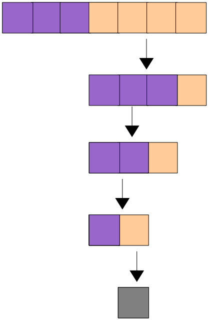
**Code at SourceForge.net**:
`http://sourceforge.net/projects/stringbsts`

# Contributions: Chapter 5

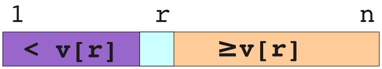
- Cache-conscious STL lists 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- **Multikey quickselect MkQSel** $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- Single-pass list partitioning 
- Parallel partition: 
    - Generic
    - String keys $\Sigma*$ 

# Selection problem

Given an unsorted array of size $n$,
find the $r$-**th element** in sorted order.



**STL** `nth_element`:
selection + partitioned output




Average **cost:** $O(n)$

**String elements**?

# Specialized selection algorithms for strings

**Existing:** aQSel, radixselect

- Linear additional space
- More than one traversal per iteration

# Specialized selection algorithms for strings

**Existing:** AQSel, radixselect

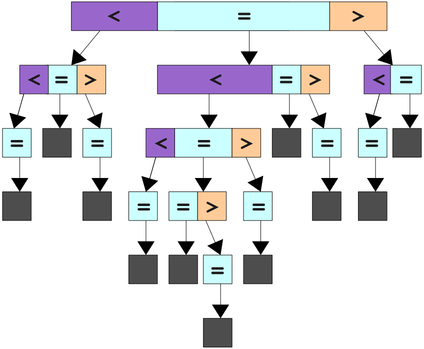
- Linear additional space
- More than one traversal per iteration

Our proposal: **Multikey quickselect** (MkQSel)

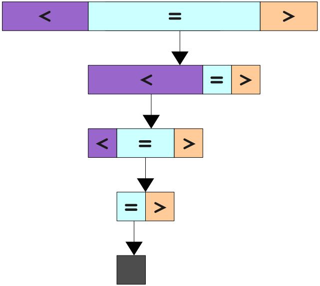
- In-place
- Easy-to-implement

# Multikey quicksort and multikey quickselect

MkQSort
(Bentley and Sedgewick, 1997)




MkQSel



# Recurrence for ternary MkQSel

Random uniform distribution, infinite keys.

$$T_k(n) \quad = \quad t_k(n) + \sum_{m=0}^{n} P\left(n, m, \frac{1}{k}\right) \frac{m T_C(m)}{n} + \sum_{i=0}^{k-1} \sum_{\ell=0}^{n} P\left(n, \ell, \frac{i}{k}\right) \frac{2\ell T_i(\ell)}{kn}$$

where

- $C$ : alphabet cardinality
- $k$ : remaining alphabet cardinality for the current character
- $t_k(n)$ : toll function
- $P(n, \ell, p) = \binom{n}{\ell} p^\ell (1-p)^{n-\ell}$ is the probability of a binomial r.v.

# Solution for ternary MkQSel

**Theorem**

The cost of ternary MkQSel is described by the following statements:

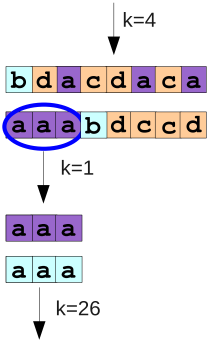
- The expected number of ternary comparisons is: $(3 + \frac{13}{(C-1)} - \frac{6(C+1)H_C}{C(C-1)})n + o(n)$
- The expected number of *second* binary comparisons is: $(2 + \frac{59}{9(C-1)} - \frac{24(C+1)H_C+1}{9C(C-1)})n + o(n)$
- The expected number of swaps for the *partitioned output* variant is: $(\frac{1}{2} - \frac{14}{9(C-1)} + \frac{30(C+1)H_C-7}{18C(C-1)})n + o(n)$
- The expected number of swaps for the *only selection* variant is: $(\frac{1}{2} + \frac{7}{18(C-1)} + \frac{4(C+1)H_{\lfloor C/2 \rfloor}+3}{12C(C-1)} - \frac{(2C-1)[C \text{ is even}]}{12C(C-1)^2} + \frac{(2C+1)[C \text{ is odd}]}{12C^2(C-1)})n + o(n)$

# Using $k$ in the algorithm

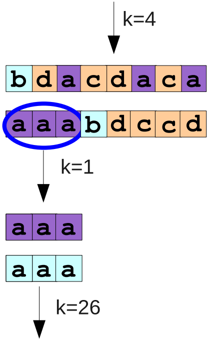
**Observation:** MkQSel could also proceed to the next character position when $k = 1$.

Incorporating $k$ into the algorithm:

- Negligible cost
- Saves comparisons and swaps



# Using $k$ in the algorithm




**Observation:** MkQSel could also proceed to the next character position when $k = 1$.

Incorporating $k$ into the algorithm:

- Negligible cost
- Saves comparisons and swaps

Using $k$, we define **binary MkQSel**.


- Cheaper comparisons
- Avoids useless swaps

# Analysis for binary MkQSel

Random uniform distribution, infinite keys.

**Recurrence:**

$$X_k(n) = x_k(n) + \sum_{m=0}^{n} P\left(n, m, \frac{1}{k}\right) \frac{2mX_C(m)}{(k-1)n} + \sum_{i=2}^{k-1} \sum_{\ell=0}^{n} P\left(n, \ell, \frac{i}{k}\right) \frac{2\ell X_i(\ell)}{(k-1)n}$$

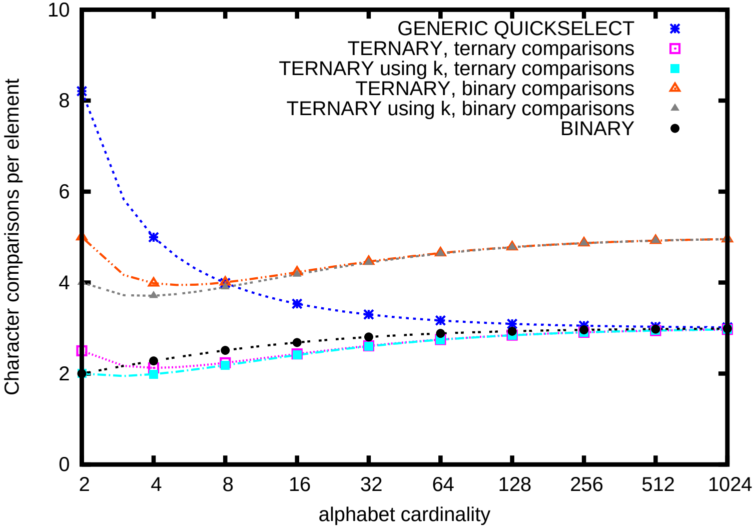
$$X_1(n) = X_C(n)$$

**Solution:**

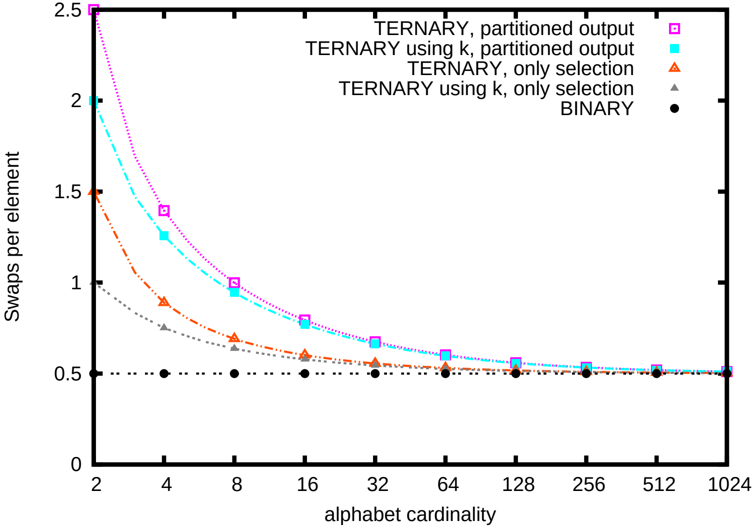
**Theorem**

On the average, binary MkQSel performs $n/2 + o(n)$ swaps and $(3 - \frac{2(H_C - 1)}{C-1})n + o(n)$ comparisons.

# Confronting algorithms: comparisons



# Confronting algorithms: swaps



# Conclusions

MkQSel: **new**, efficient, in-place **string selection** algorithm.

- Ternary partitioning
- Binary partitioning

Detailed **analysis** for a random uniform distribution.

- Binary partitioning: least number of binary **comparisons** and **swaps**.

# Publications

L. Frias and S. Roura. Multikey Quickselect. Technical report LSI-09-27-R, 2009.

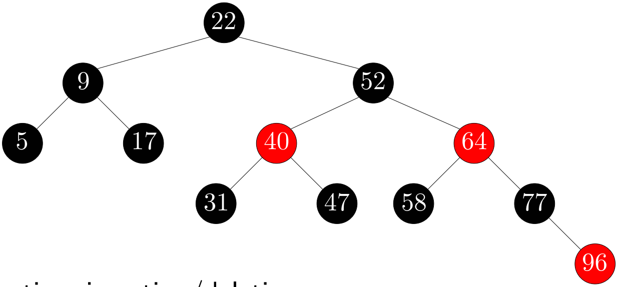
**Code at SourceForge.net**:
http://sourceforge.net/projects/mkqsel

# Contributions: Chapter 6

- Cache-conscious STL lists 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- **Parallel bulk operations for STL dictionaries** 
- Single-pass list partitioning 
- Parallel partition:
  - Generic 
  - String keys $\Sigma*$ 

# STL dictionaries

`set`, `multiset`, `map`, `multimap`.



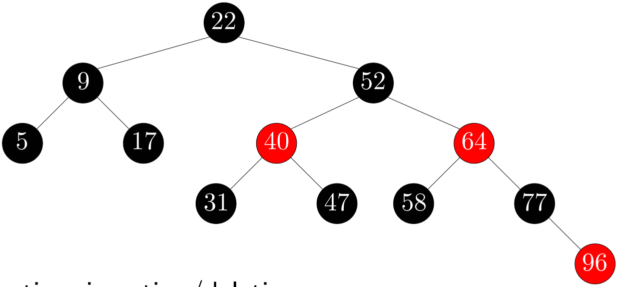

**Properties:**

- **Logarithmic** time insertion/deletion
- Linear time traversal in sorted order

**Parallelization?**

# STL dictionaries

`set`, `multiset`, `map`, `multimap`.




**Properties:**

- **Logarithmic** time insertion/deletion
- Linear time traversal in sorted order

**Parallelization?** Bulk operations

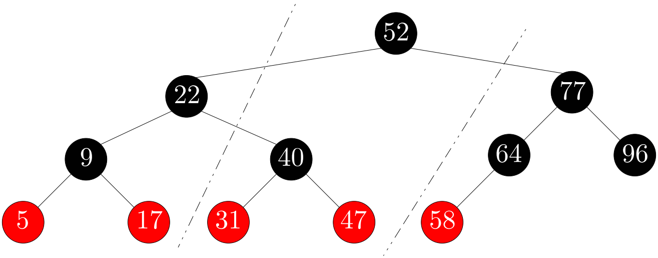
# Parallelization of bulk insertion and construction

Consider $p$ processors.

1. **Preprocessing** $\rightarrow$ sorted sequence divided into $p$ parts.
2. **Allocation** and **initialization** of an array of nodes.
3. Bulk operations
   - Construction
   - Insertion

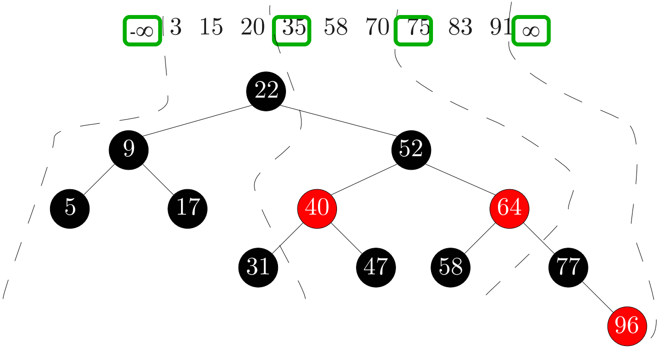
**Tools:** OpenMP + MCSTL

# Construction



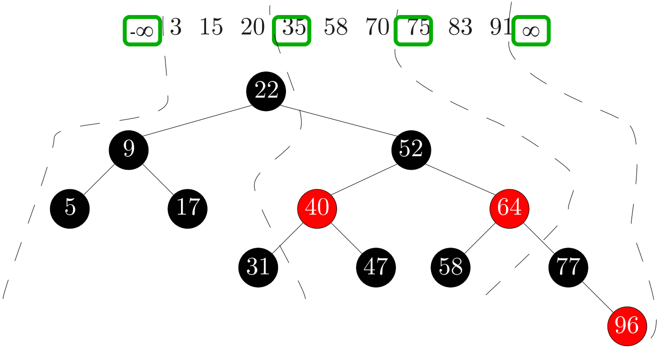

**Key property:** independent calculation of each element.
(Park and Park, 2001)

# Insertion




**Key property:** negligible work of tree split/concatenate with respect to actual insertion.

# Insertion

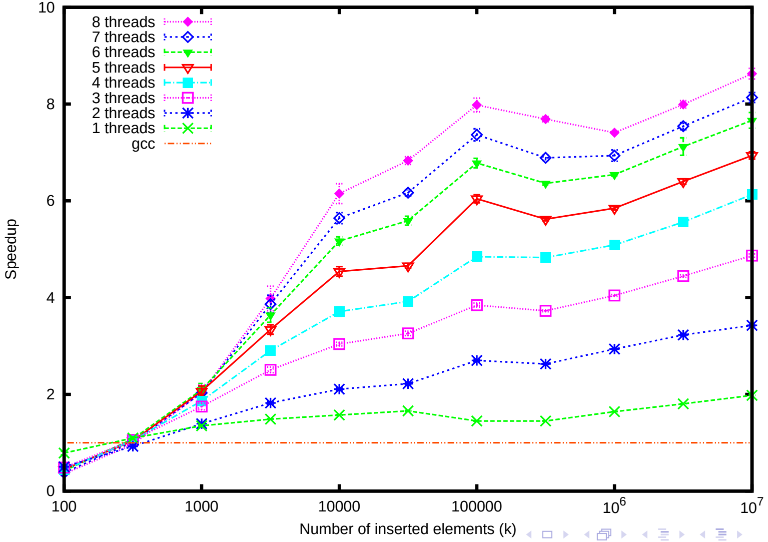



**Key property:** negligible work of tree split/concatenate with respect to actual insertion.

+ **Dynamic load-balancing** for enhanced robustness.

# Experimental results

**Insertion in an 8-core Xeon**, tree 10x smaller than the input.



# Conclusions

New parallel algorithms for **bulk insertion** and **construction**.

Thorough **experimental** analysis:

- Scalable insertion and construction
- Fast sequential insertion algorithm
- Dynamic load-balancing shows useful

# Publications

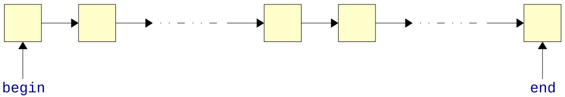
L. Frias and J. Singler. Parallelization of Bulk Operations for STL Dictionaries. In **HPPC** *2007*, volume 4854 of *LNCS*. Springer.

**Code in the MCSTL** 0.8.0-beta.

# Contributions: Chapter 7

- Cache-conscious STL lists 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- **Single-pass list partitioning** 
- Parallel partition: 
    - Generic
    - String keys $\Sigma*$ 

# List partitioning problem

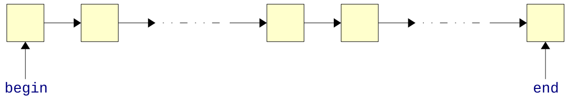



**Problem:** Divide a **sequence** into $p$ **parts** of "equal" length.

- Unknown size
- Only sequential access

**Application example:** prerequisite for parallelization
$\longrightarrow$ limits speedup (Amdahl law)

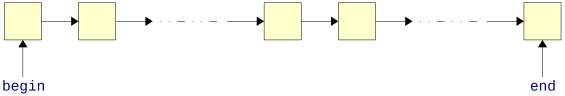
# SinglePass: basic algorithm




**Naïve solutions:**

- Traversing twice the sequence
- Using linear additional space

# SinglePass: basic algorithm



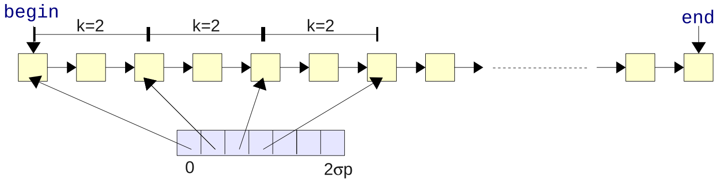

**Naïve solutions:**

- Traversing twice the sequence
- Using linear additional space

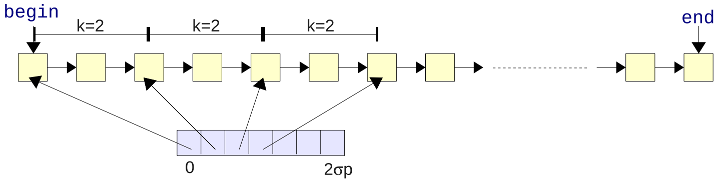
**Our solution:**

- One traversal (online)
- Sublinear additional space

# Basic SinglePass algorithm



# Basic SinglePass algorithm




**Theorem**

The basic SinglePass algorithm has the following properties:

- Time complexity: $\Theta(n + \sigma p \log n)$
- Quality guarantee: $g = \frac{\sigma+1}{\sigma}$

where $g = \frac{|\text{longest part}|}{|\text{shortest part}|}$ (optimal $g = 1$)

# Generalized SinglePass algorithm

Change in **making room** for new subsequences:

- Every *m*-**th iteration**: basic algorithm
- Otherwise: **double the size** of the array

→ **Trade-off quality/space**

# Generalized SinglePass algorithm

Change in **making room** for new subsequences:

- Every ***m*-th iteration**: basic algorithm
- Otherwise: **double the size** of the array

$\rightarrow$ **Trade-off quality/space**

**Theorem**

The generalized SinglePass algorithm for $m = 2$ has the following properties:

- Time complexity: $\Theta(n + p\sqrt{n}\log n)$
- Quality guarantee: $g = 1 + \frac{\sqrt{n}}{\sigma n} \xrightarrow{n\to\infty} 1$

# Conclusions

SinglePass: new algorithm for the **list partitioning problem**.

- One traversal
- Sublinear additional space

Theoretical analysis on the **quality of solutions**.

- Quality improves with the input size

Thorough **experimental** analysis:

- Very fast list partitioning
- Practical for parallelization

# Publications

L. Frias, J. Singler, and P. Sanders. Single-Pass List Partitioning. In **MuCoCoS 2008**. IEEE Computer Society Press.

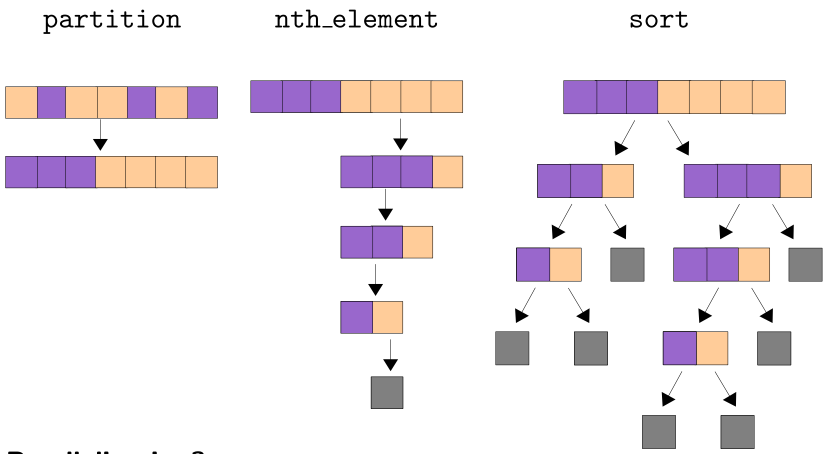
L. Frias, J. Singler, and P. Sanders. Single-pass list partitioning. **SCPE**, 9(3), 2008.

**Code in the MCSTL** 0.8.0-beta + *libstdc++ parallel mode*

# Contributions: Chapters 8-9

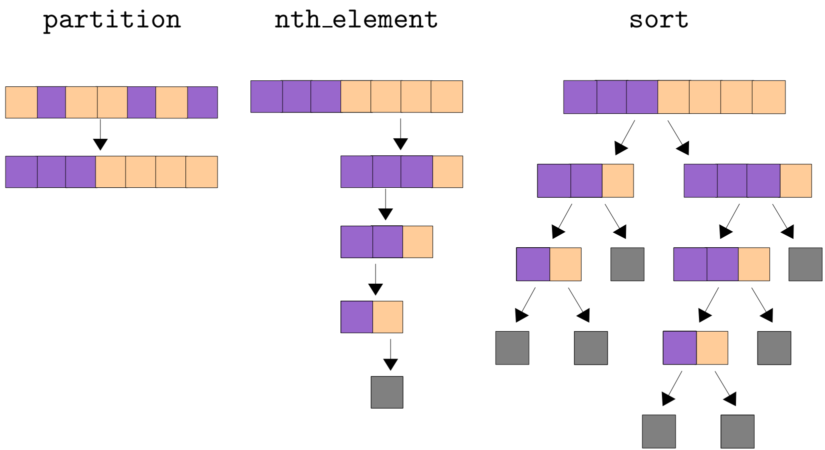
- Cache-conscious STL lists 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- Single-pass list partitioning 
- **Parallel partition:** 
  - **Generic**
  - **String keys** $\Sigma *$ 

# STL partition, sort, nth_element




**Parallelization?**

# STL partition, sort, nth_element




**Parallelization?**

Parallelizing **partition** is fundamental for scalability.

# Practical parallel partitioning algorithms

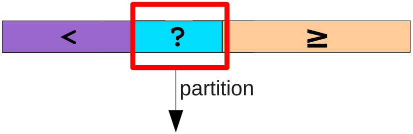
Several **practical algorithms** for multi-core computers:

- Blocked: Blocked variant of (Francis and Panan, 1992)
- F&A: (Tsigas and Zhang 2003; Singler et al. 2007)

**Properties:**

- Use blocks
- 3 steps:
  1. Sequential **setup** of each processor
  2. Parallel **main phase**: most of the partitioning is done
  3. **Cleanup** phase

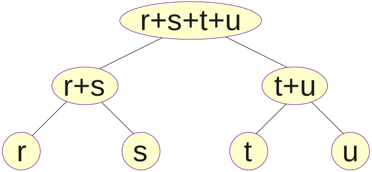
# Number of element comparisons




**Observation:** Cleanup partitions an already processed range
$\longrightarrow$ more than $n$ comparisons in total.

# New cleanup algorithm

**Key idea:** manage information about misplaced elements using a small order-statistics tree.




**Properties:**

- Additional space: $\Theta(p)$.
- A processed element is not compared again.
- Perfect parallelizable swaps.

# New parallel partitioning algorithms

Let $p$ be the number of processors, let $b$ be the size of blocks.

**Theorem**

Blocked and F&A perform exactly $n$ comparisons when using our cleanup algorithm.

**Theorem**

Blocked takes $\Theta(n/p + \log p)$ parallel time using our cleanup algorithm.

**Theorem**

Consider $p \leq b$. F&A takes $\Theta(n/p + \log^2 p + b)$ parallel time using our cleanup algorithm.

# Conclusions for the generic case

New **cleanup algorithm** for parallel partitioning.

Resulting parallel partitioning algorithms:

- Optimal in the number of comparisons
- STL compliant

Thorough **experimental** comparison:

- Scalable
- Optimality does not bring performance improvements

# Taking advantage for strings

Using comparison optimal parallel partitioning algorithms, **parallel** aQSort and aQSel can be defined.

# Taking advantage for strings

Using comparison optimal parallel partitioning algorithms, **parallel** aQSort and aQSel can be defined.

**Key points**:

- Parallelism and string techniques are orthogonal.
- Keeping up with the relative order of comparisons needs comparison optimal partitioning.

**Properties**: as in the sequential case.

# Conclusions for strings

Novel combination of techniques:

- Specialized comparison-based algorithms for **strings + parallelism**

Thorough **experimental** analysis:

- Sequential: AQSort/AQSel pay off
- + Parallel: reasonable speedups

# Publications

L. Frias and J. Petit. Parallel partition revisited. In **WEA 2008**, volume 5038 of *LNCS*. Springer.

L. Frias and J. Petit. Combining digital access and parallel partition for quicksort and quickselect. In **IWMSE '09**. IEEE Computer Society.

**Code at SourceForge.net**:
`http://sourceforge.net/projects/{parpartition,stringbsts}`

# Contributions

- Cache-conscious STL lists 
- Analysis of string lookups in ABSTs $\Sigma^*$ 
- Multikey quickselect MkQSel $\Sigma^*$
- Parallel bulk operations for STL dictionaries 
- Single-pass list partitioning 
- Parallel partition: 
    - Generic
    - String keys $\Sigma*$ 

# Dissemination of results

**Publications** at: JEA, SCPE, WEA, HPPC, MuCoCoS, IWMSE.

**Implementations** at:

- Sourceforge.net
- MCSTL
- `/usr/include/c++/4.3/parallel/list_partition.h`

# Further work

**Some difficulties:**

- Tight requirements
- Parallelism and exceptions
- Parallel allocators

**A wish:** statistical data, benchmarks on STL usage.

**An open issue**: parallel&string algorithms and data structures.

- **Theory:** algorithms + analysis
- **Practice:**
  - Further experiments
  - STL: specializations, new Standard

# Thanks!