Parallelization of Bulk Operations for STL Dictionaries

Leonor Frias [1] Johannes Singler [2]

[1]Dep. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya

[2]Institut für Theoretische Informatik, Universität Karlsruhe

HPPC Workshop
28/08/2007

Outline

Motivation

Multi-core processors everywhere but parallel programming is hard.

Motivation

Multi-core processors everywhere but parallel programming is hard.

Our approach: Provide parallel libraries of algorithms and data structures.

- Multi-Core Standard Template Library(MCSTL) [SSP07]: Parallel implementation of the C++ STL

(MC)STL

Standard Template Library (STL):
Algorithmic core of the C++ Standard Library.

Components:

- Containers: `list`, `vector`, `map`...
- Iterators: high-level pointers
- Algorithms: `sort`, `merge`, `find`...

(MC)STL

Standard Template Library (STL):
Algorithmic core of the C++ Standard Library.

Components:

- Containers: `list`, `vector`, `map`...
- Iterators: high-level pointers
- Algorithms: `sort`, `merge`, `find`...

Parallel implementation?

(MC)STL

Standard Template Library (STL):
Algorithmic core of the C++ Standard Library.

Components:

- Containers: `list`, `vector`, `map`...
- Iterators: high-level pointers
- Algorithms: `sort`, `merge`, `find`... $\rightarrow$ Friday [SSP07]

(MC)STL

Standard Template Library (STL):
Algorithmic core of the C++ Standard Library.

Components:

- Containers: `list`, `vector`, `map`... → This talk
- Iterators: high-level pointers
- Algorithms: `sort`, `merge`, `find`... → Friday [SSP07]

STL Dictionaries

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

STL Dictionaries

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

Operations: Given $|dictionary| = n$, $|input| = k$

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

int main(){
    vector<int> V;
    read_and_sort(V);

    set<int> S(V.begin() + V.size()/2, V.end());

    int key, x;
    cin >> key;
    while (cin >> x) S.insert(x);

    for(typename set<int>::iterator it = S.find(key); it != S.end(); ++it)
         cout << *it << endl;
}
```

STL Dictionaries

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

Operations: Given $|dictionary| = n$, $|input| = k$

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

int main(){
    vector<int> V;
    read_and_sort(V);

    set<int> S(V.begin() + V.size()/2, V.end());

    int key, x;
    cin >> key;
    while (cin >> x) S.insert(x);

    for(typename set<int>::iterator it = S.find(key); it != S.end(); ++it)
          cout << *it << endl;
}
```

Bulk construction:
Θ(k)
--if sorted input--

Bulk insertion:
min(O(k+n), O(klog n))
--if sorted input--

STL Dictionaries

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

Operations: Given $|dictionary| = n$, $|input| = k$

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

int main(){
    vector<int> V;
    read_and_sort(V);

    set<int> S(V.begin() + V.size()/2, V.end());

    int key, x;
    cin >> key;
    while (cin >> x) S.insert(x);

    for(typename set<int>::iterator it = S.find(key); it != S.end(); ++it)
          cout << *it << endl;
}
```

Bulk construction:
Θ(k)
--if sorted input--

Bulk insertion:
min(O(k+n), O(klog n))
--if sorted input--

Search-based operations:
O(log n)

STL Dictionaries

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

Operations: Given $|dictionary| = n$, $|input| = k$

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

int main(){
    vector<int> V;
    read_and_sort(V);

    set<int> S(V.begin() + V.size()/2, V.end());

    int key, x;
    cin >> key;
    while (cin >> x) S.insert(x);

    for(typename set<int>::iterator it = S.find(key); it != S.end(); ++it)
          cout << *it << endl;
}
```

Bulk construction:
Θ(k)
--if sorted input--

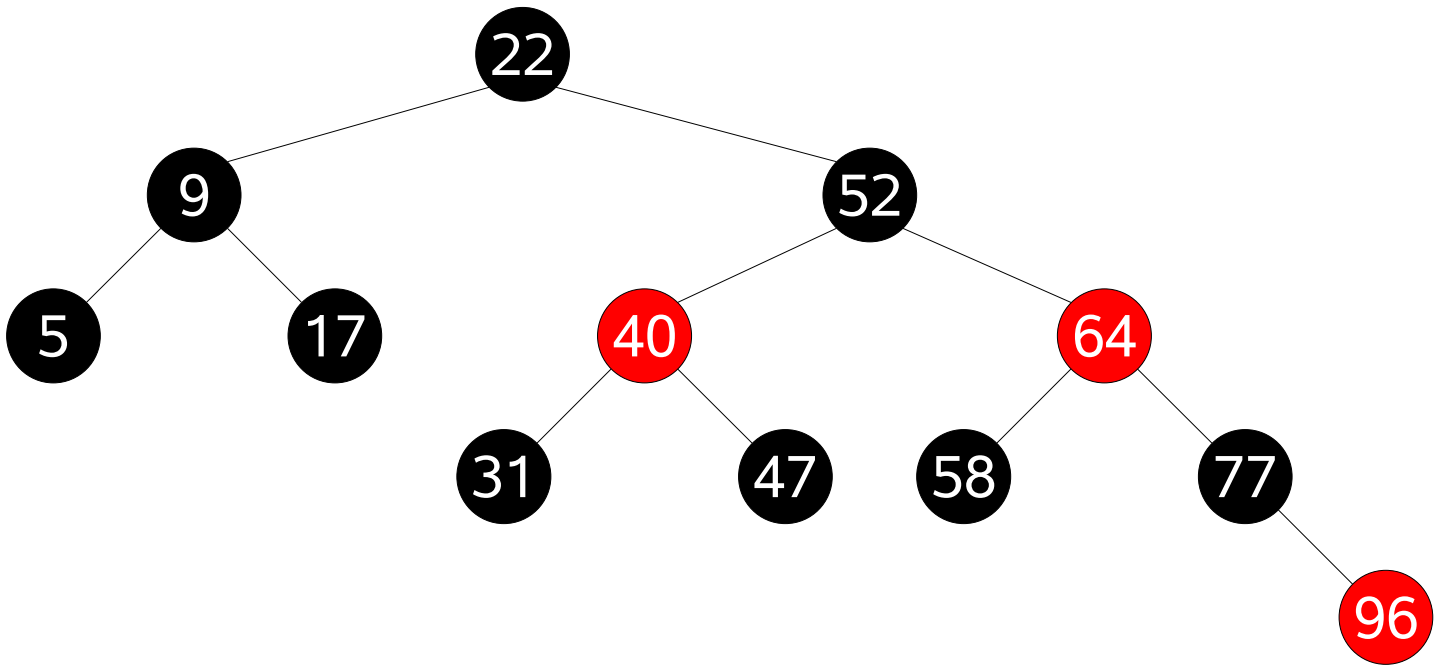
Bulk insertion:
min(O(k+n), O(klog n))
--if sorted input--

Search-based operations:
O(log n)

Scan in sorted order:
amortized O(1)

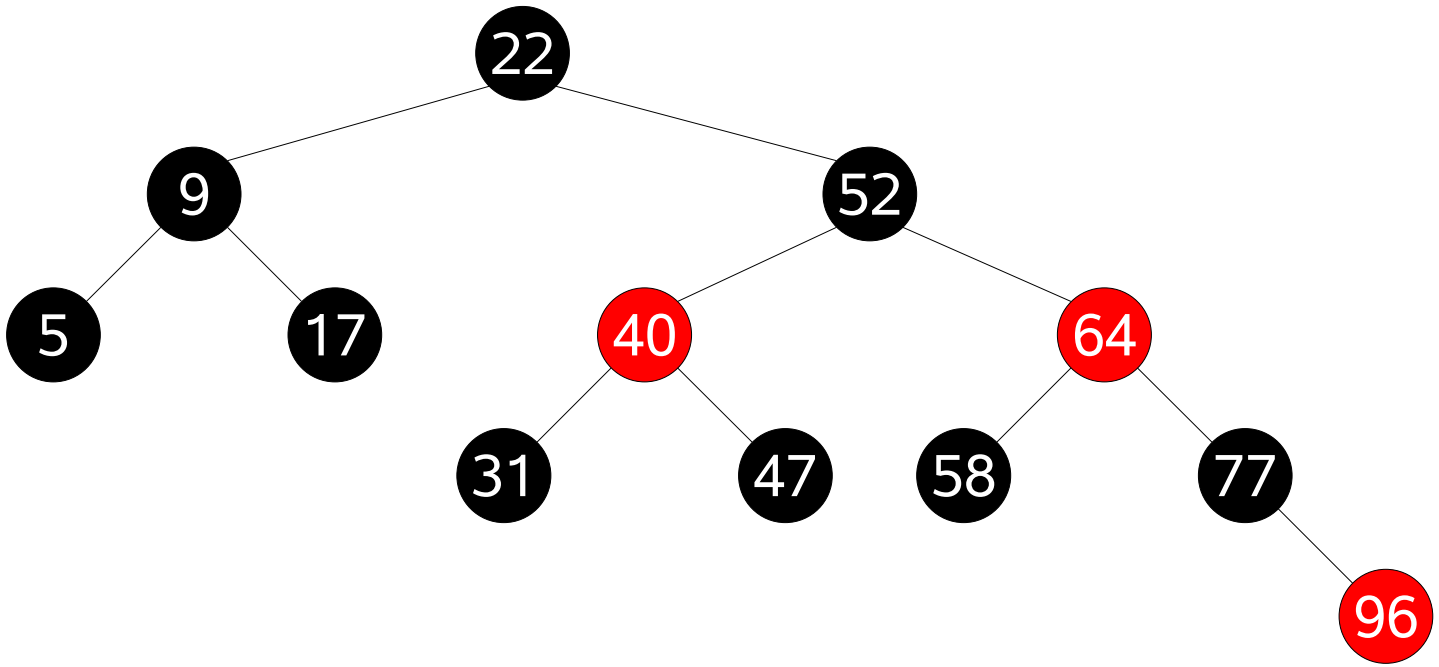
Implementation of STL Dictionaries

Balanced Binary Search Trees (red-black trees)



Implementation of STL Dictionaries

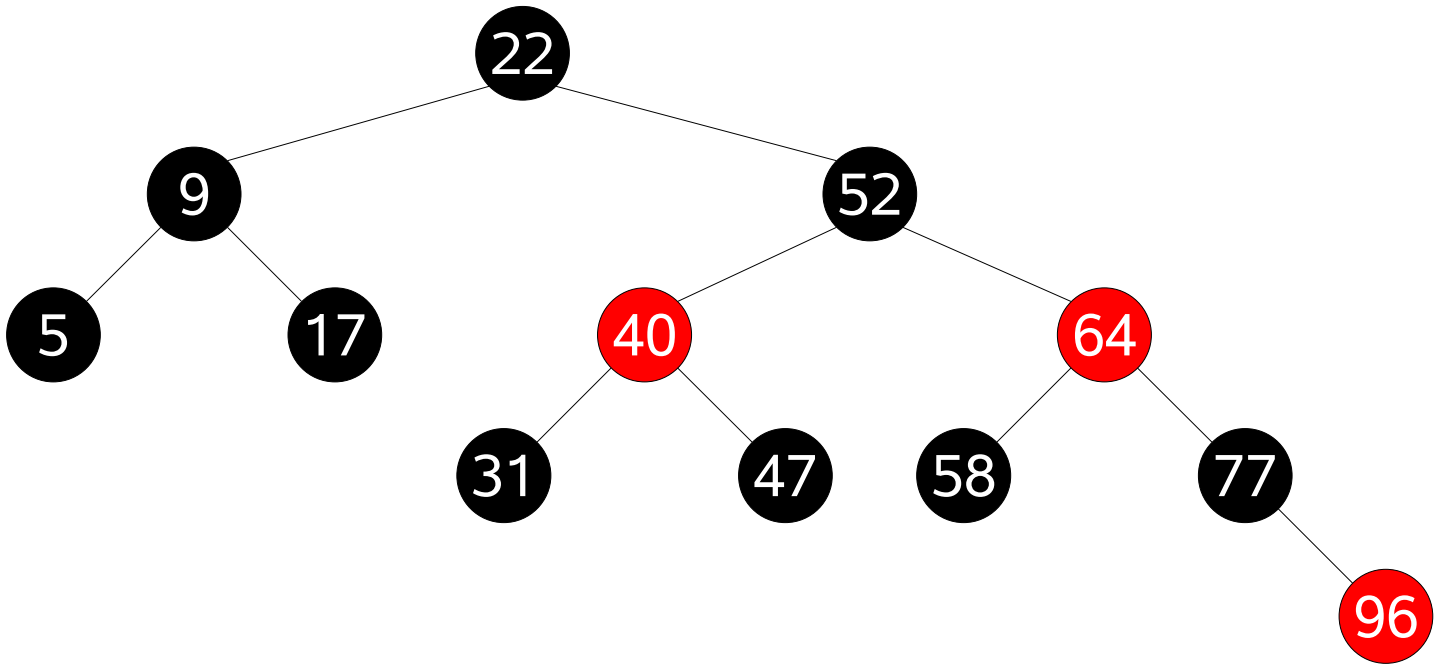
Balanced Binary Search Trees (red-black trees)



Parallelization?

Implementation of STL Dictionaries

Balanced Binary Search Trees (red-black trees)



Parallelization? Bulk operations: construction and insertion

Previous Work

PRAM Parallel Red-Black Tree Algorithms [PP01].

STAPL library [AJR+01]:

- No implementation available
- Aiming for distributed-memory systems
- Rigid partitioning of the tree
  (in worst case, work by one thread)

Previous Work

PRAM Parallel Red-Black Tree Algorithms [PP01].

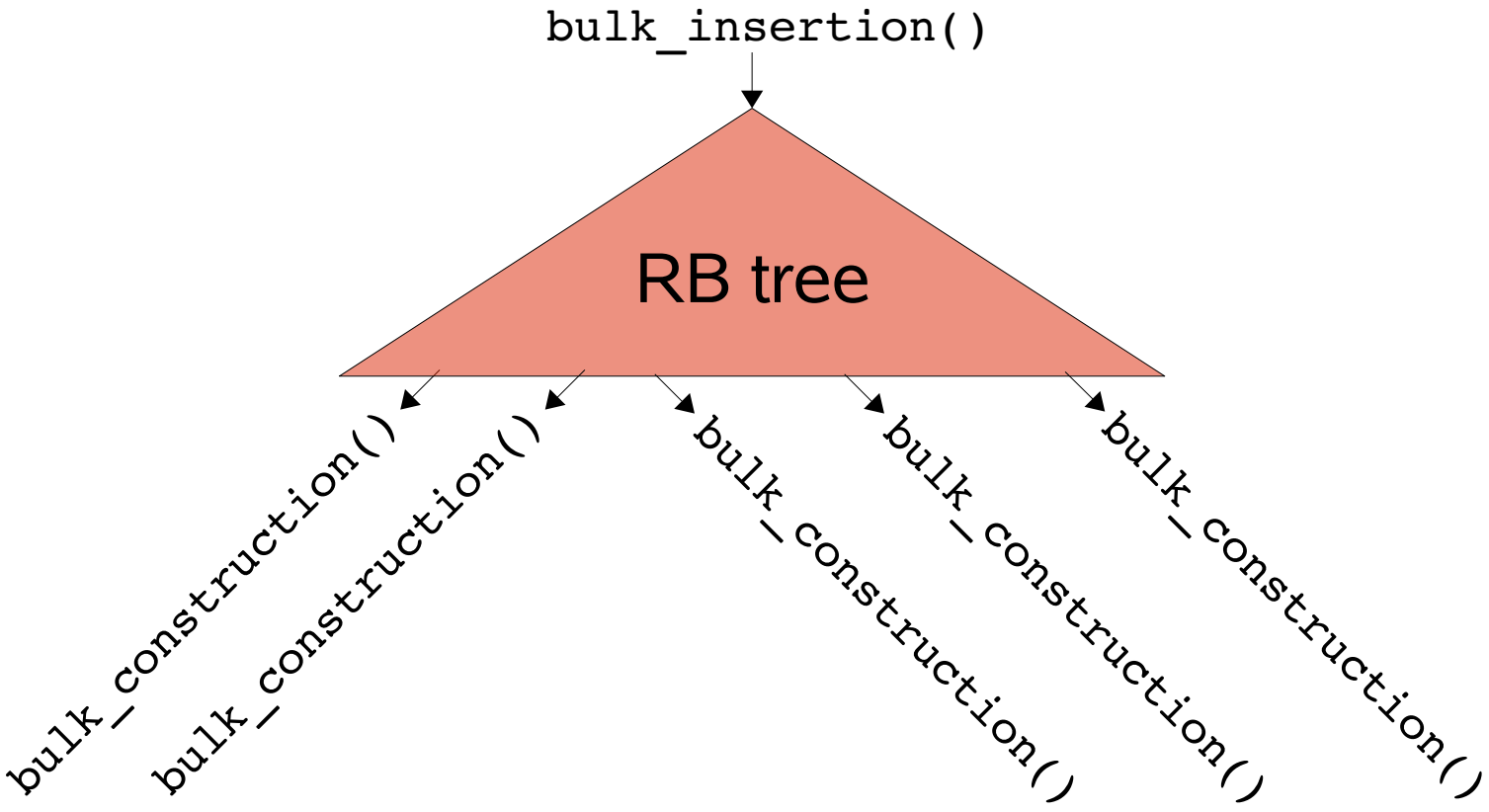
STAPL library [AJR+01]:

- No implementation available
- Aiming for distributed-memory systems
- Rigid partitioning of the tree
  (in worst case, work by one thread)

Our implementation: On the top of libstdc++ (GCC 4.2.0).

- Sequential data structure unaffected
- No rigid partitioning, but per operation

Relation between Bulk Construction and Insertion



Algorithms: Common Steps

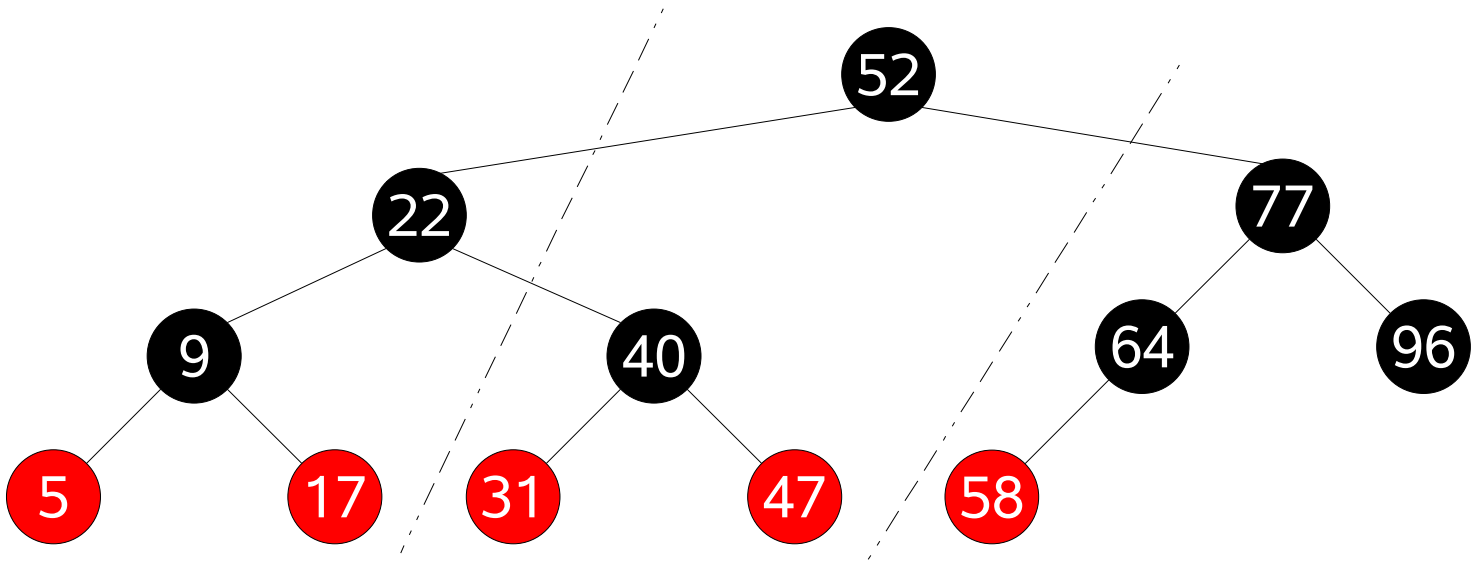
Setup

Output: sorted sequence divided into p pieces

Allocation + initialization of tree nodes in parallel

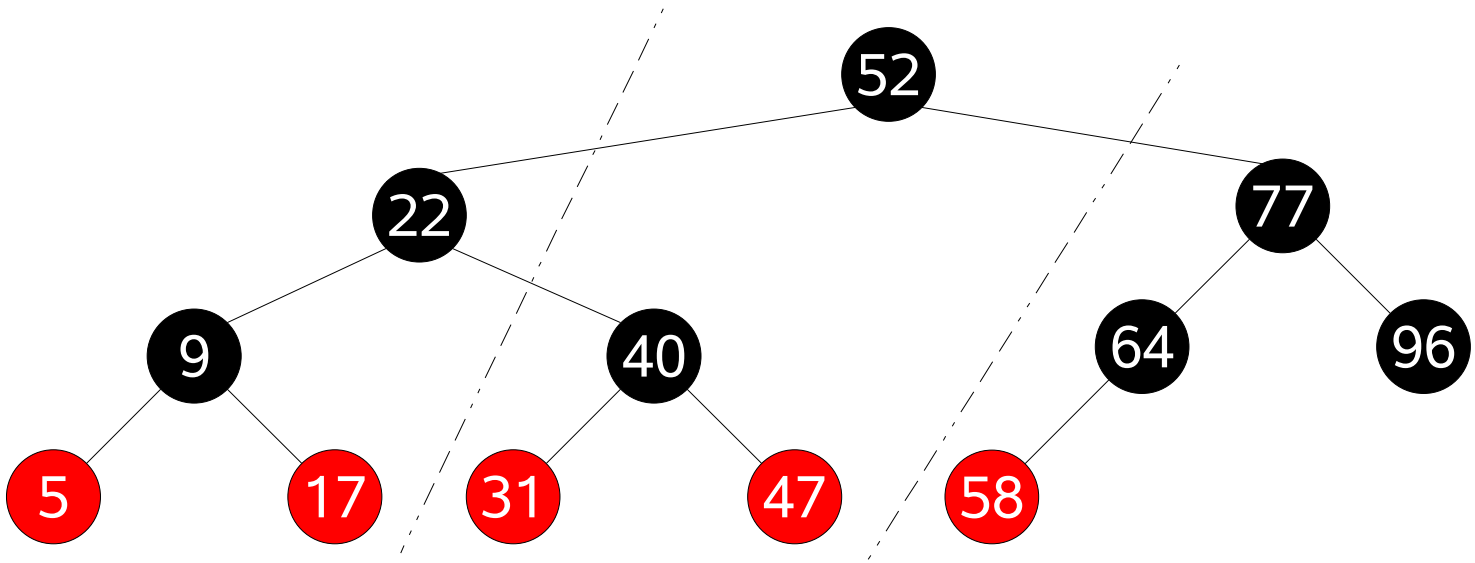
Output: Array of (unlinked) nodes

Construction



Threads are fully independent.
Similarities with theory [PP01].

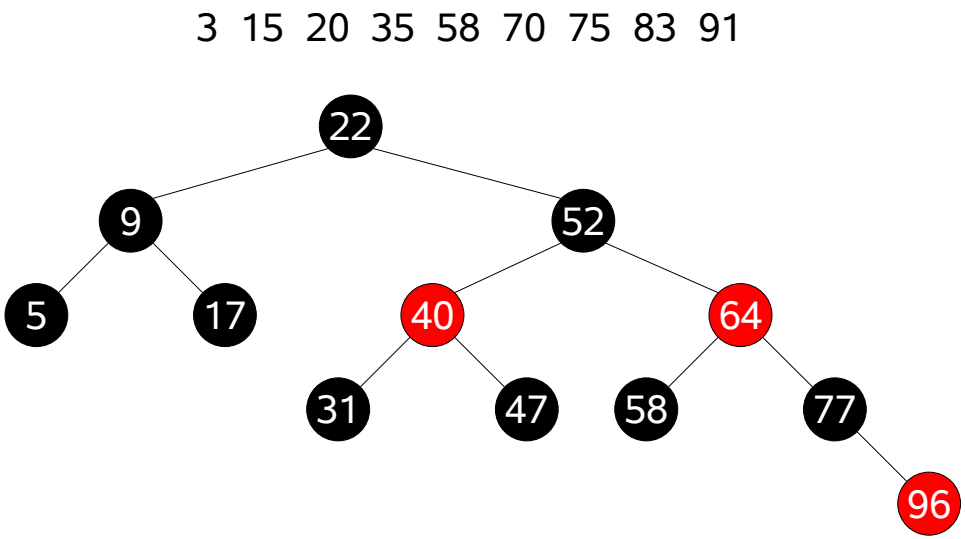
Construction



Threads are fully independent.
Similarities with theory [PP01].
Parallel time: $\Theta(k/p)$.

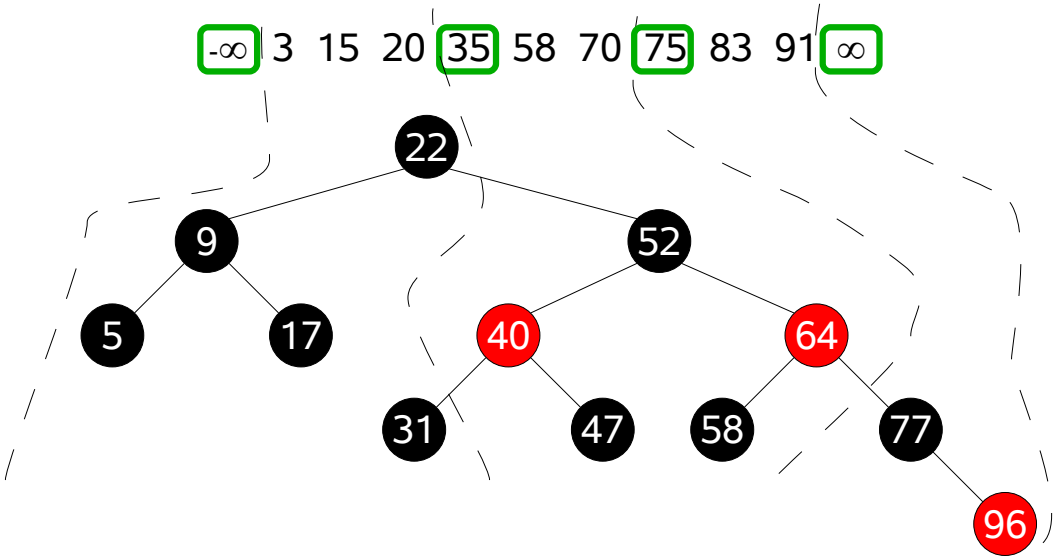
Insertion: Work Pieces

1. Pivots $\in$ input sequence $\longrightarrow$ split the tree
2. Pivots $\in$ tree $\longrightarrow$ divide the sequence



Insertion: Work Pieces

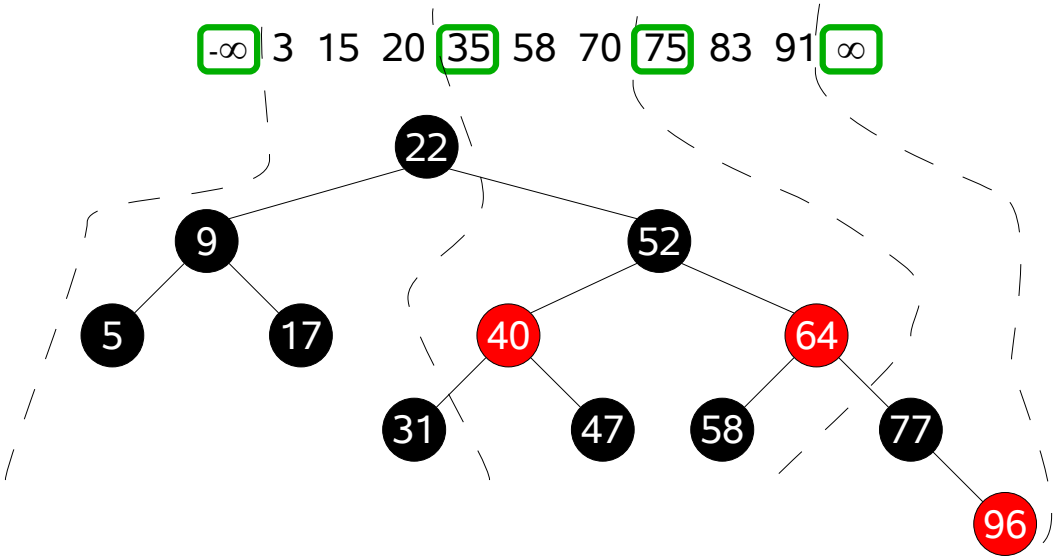
1. Pivots $\in$ input sequence $\longrightarrow$ split the tree
2. Pivots $\in$ tree $\longrightarrow$ divide the sequence



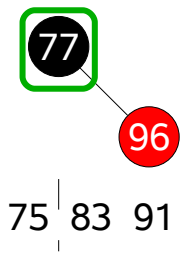
From a perfect division of the sequence

Insertion: Work Pieces

1. Pivots $\in$ input sequence $\longrightarrow$ split the tree
2. Pivots $\in$ tree $\longrightarrow$ divide the sequence



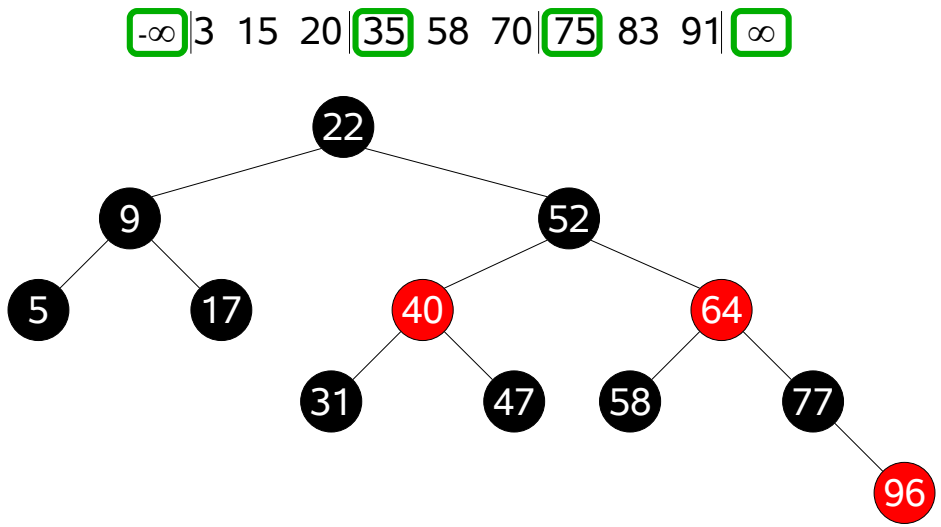
From a perfect division of the sequence



Taking subtree root key as pivot
No guarantee on the length

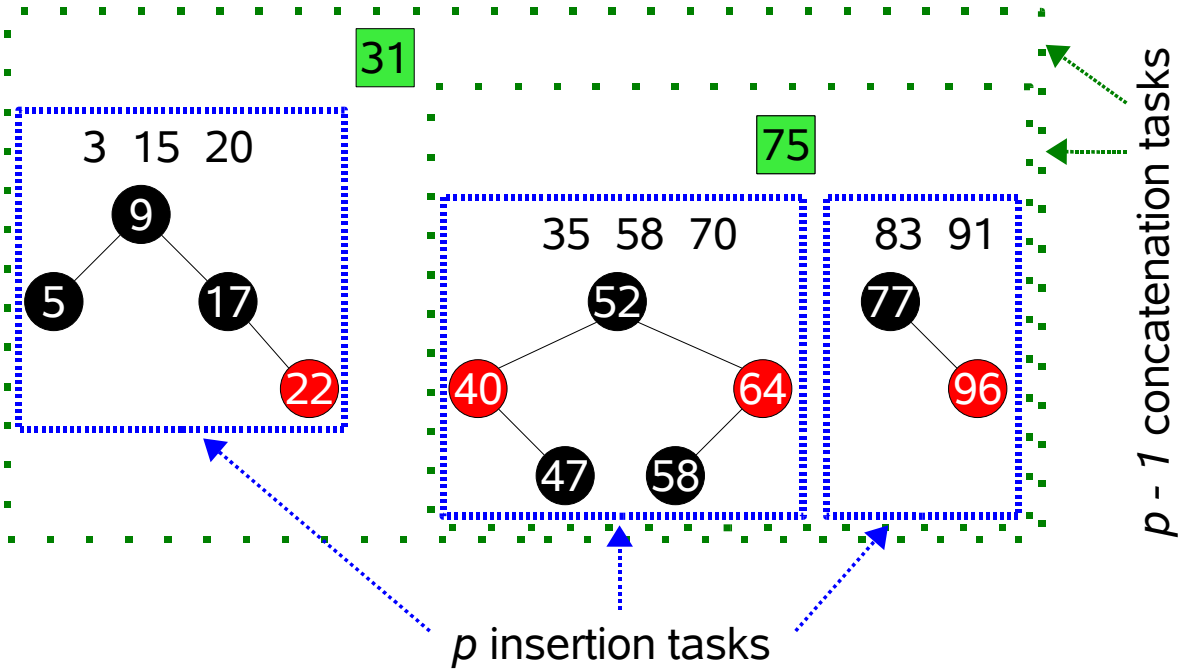
Insertion: Step 1

Split into p subtrees (p number of threads).



Insertion: Step 1

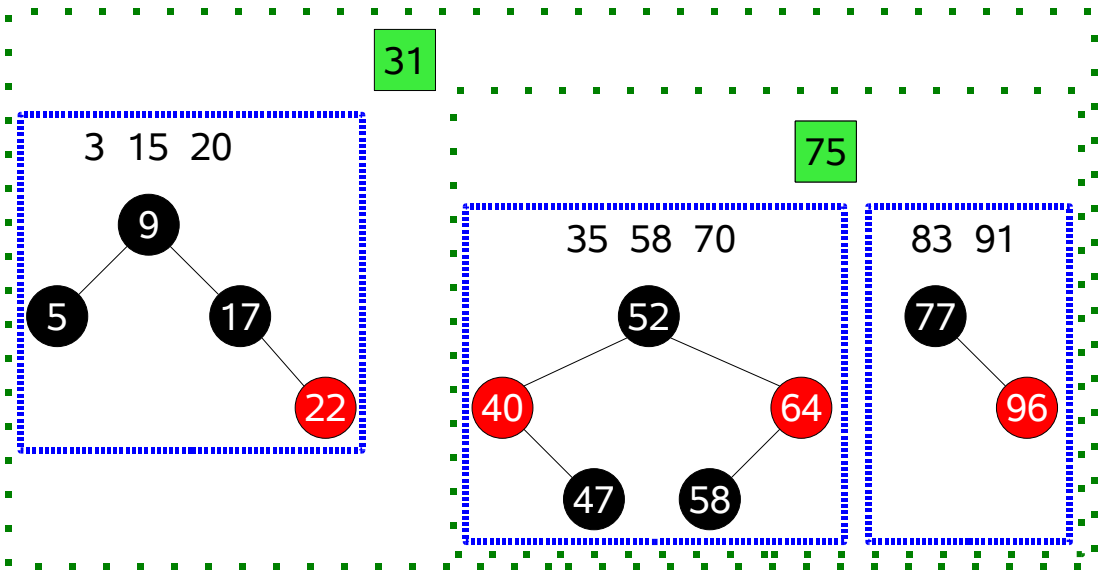
Split into p subtrees (p number of threads).



Sequential time: $O(p \log n)$

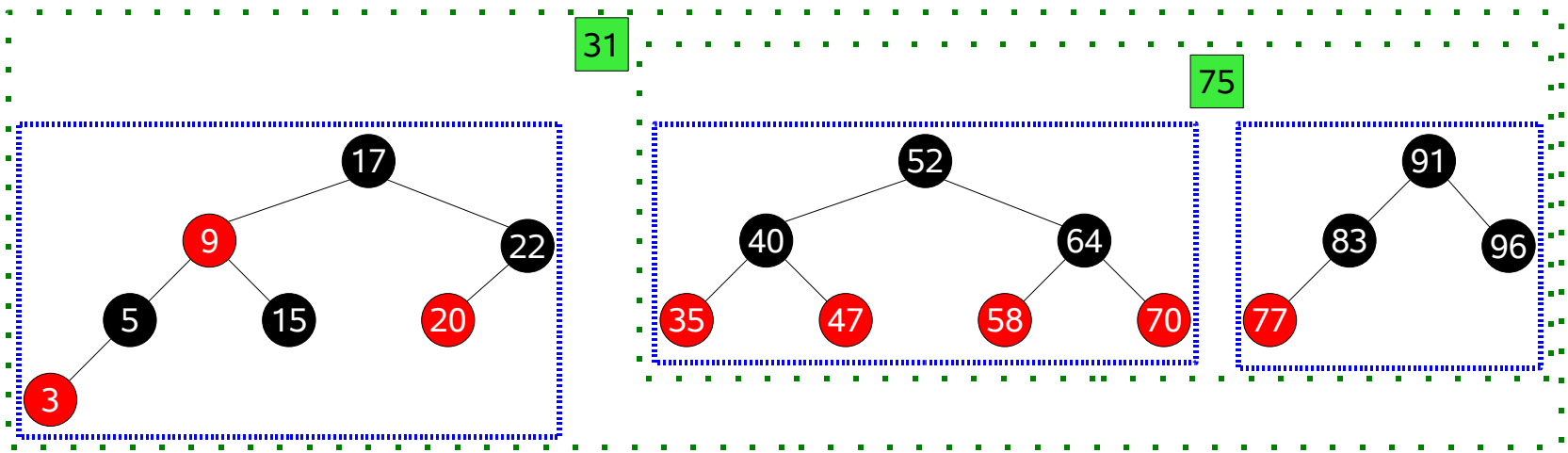
Insertion: Step 2

Process insertion tasks in parallel.



Insertion: Step 2

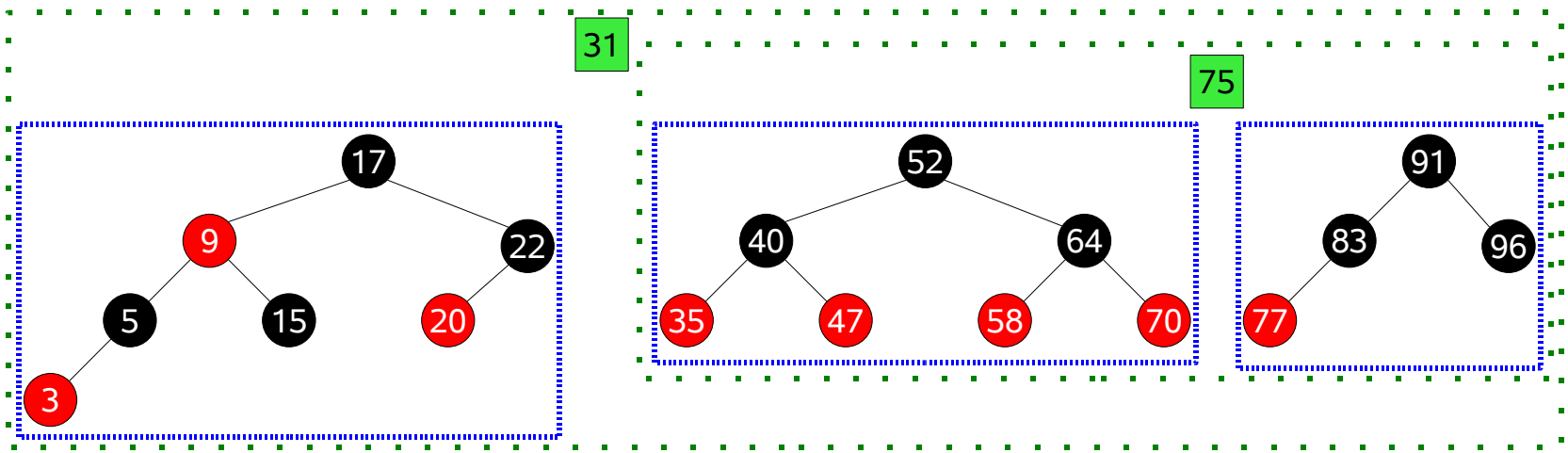
Process insertion tasks in parallel.



Parallel time: $O(k/p \log n)$

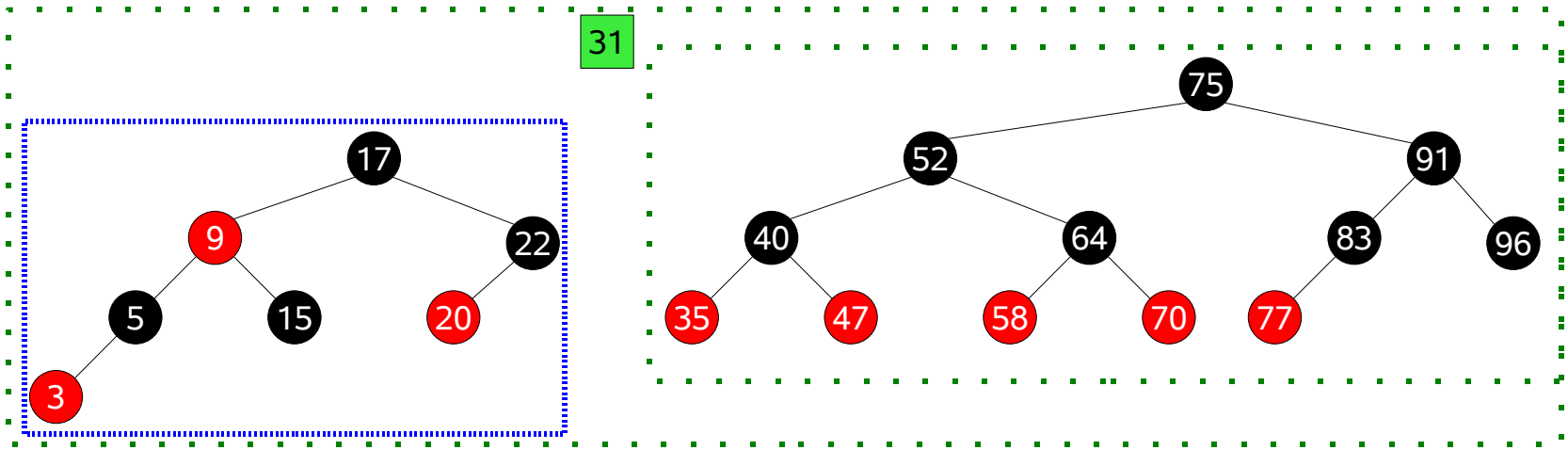
Insertion: Step 3

Process concatenation tasks in parallel.



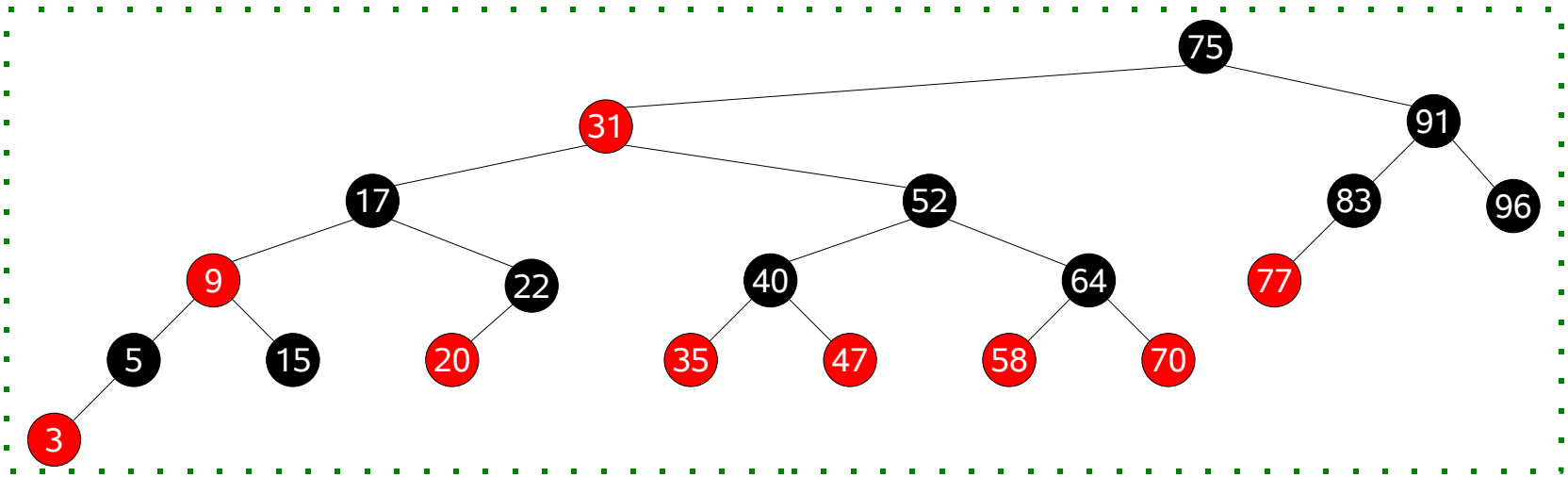
Insertion: Step 3

Process concatenation tasks in parallel.



Insertion: Step 3

Process concatenation tasks in parallel.

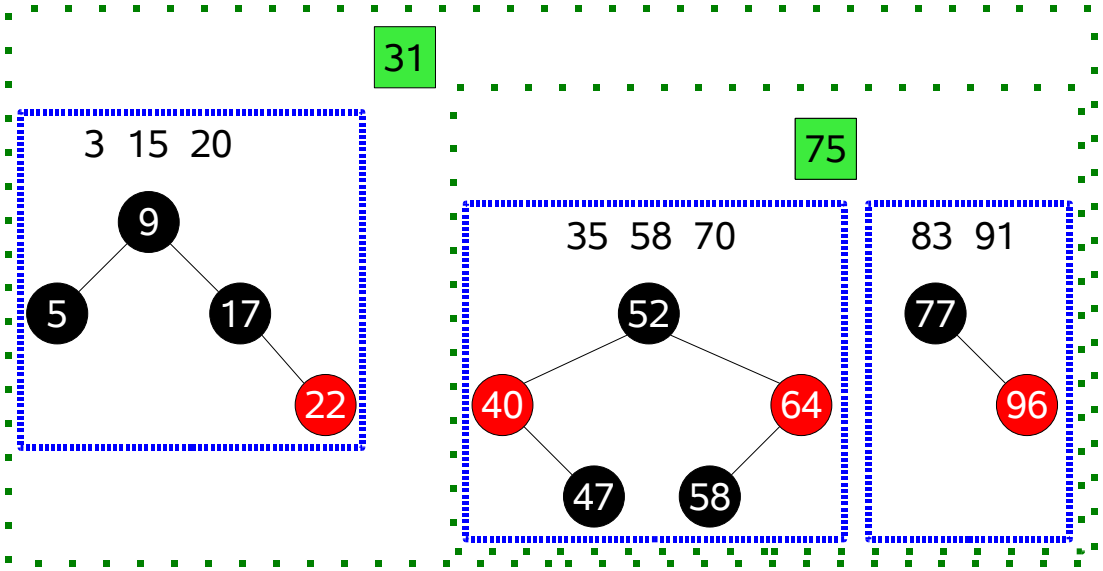


Sequential time of one concatenation: $O(\log n_1/n_2)$

Parallel time of doing all concatenations: $O(\log p \log n)$

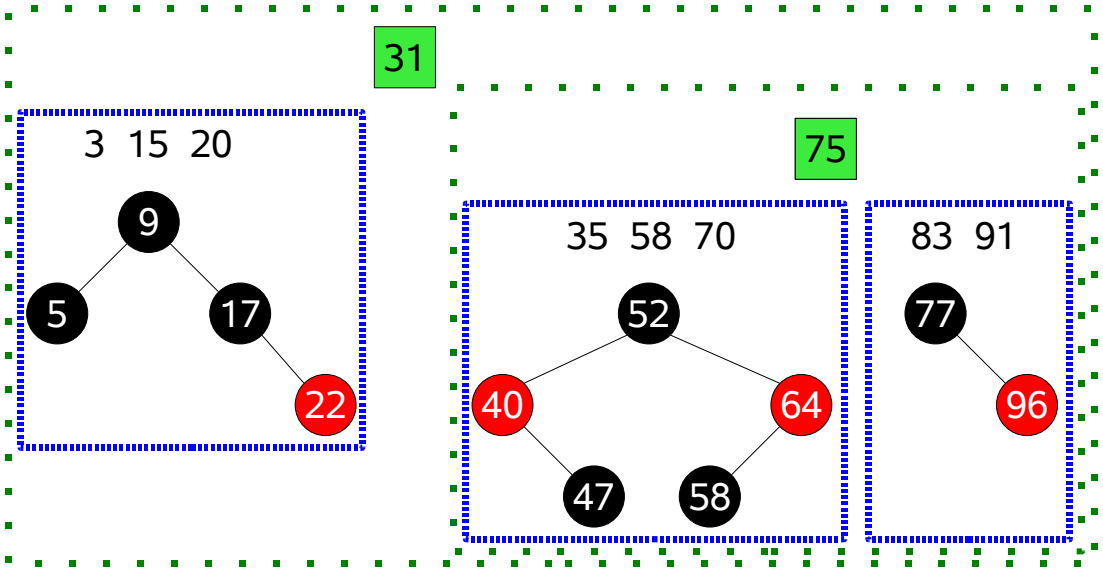
Dynamic Load-Balancing: Motivation

The tree size of insertion problems may be very different.



Dynamic Load-Balancing: Motivation

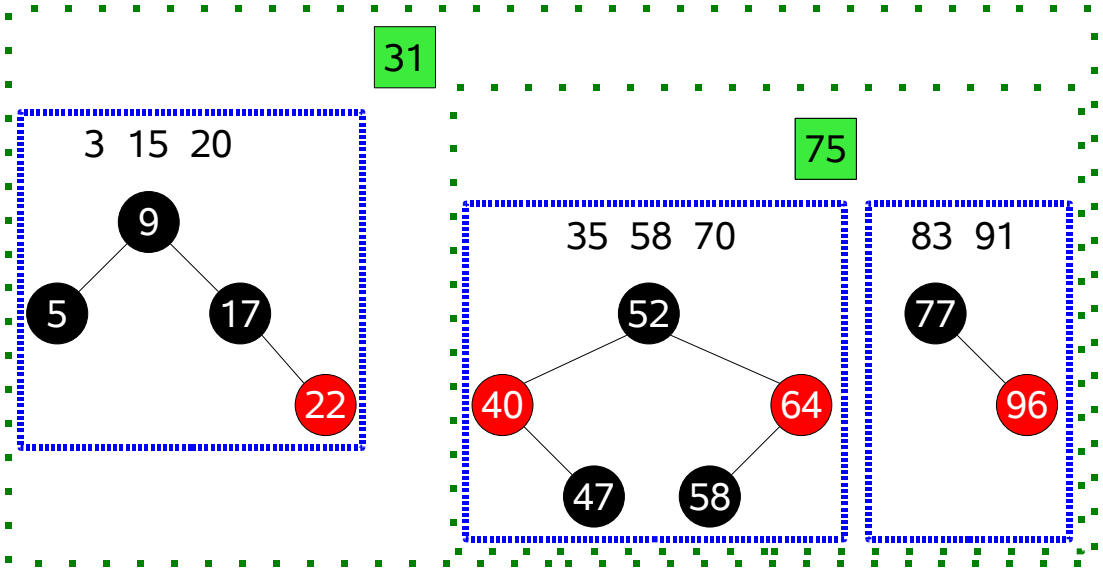
The tree size of insertion problems may be very different.



This could degrade performance!

Dynamic Load-Balancing: Motivation

The tree size of insertion problems may be very different.

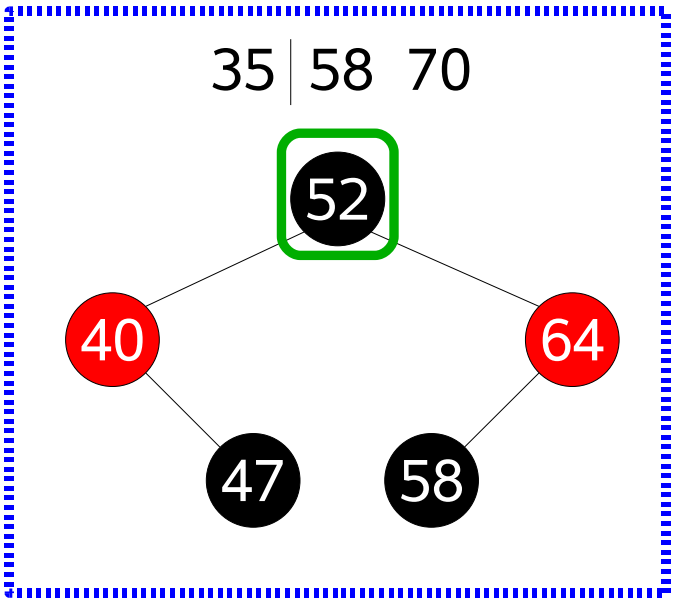


This could degrade performance!

Use dynamic load-balancing to process them instead.

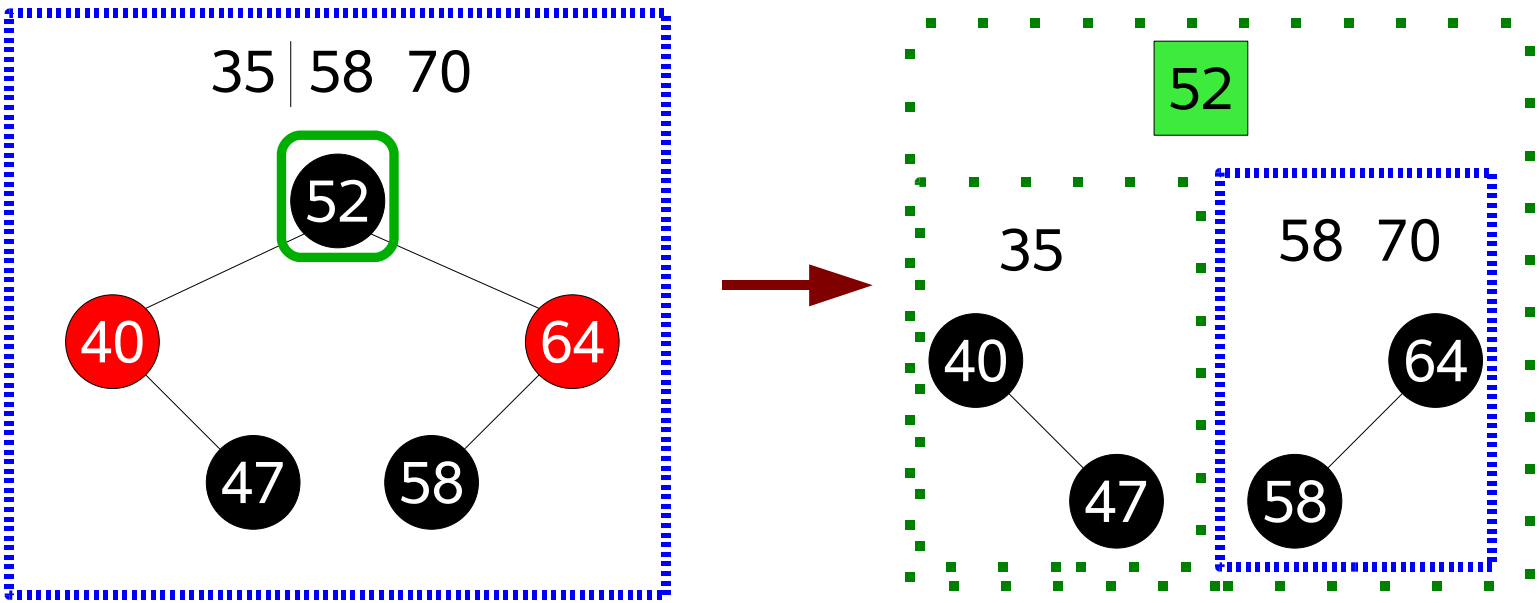
Dynamic Load-Balancing: Approach

Division of insertion tasks into smaller ones.



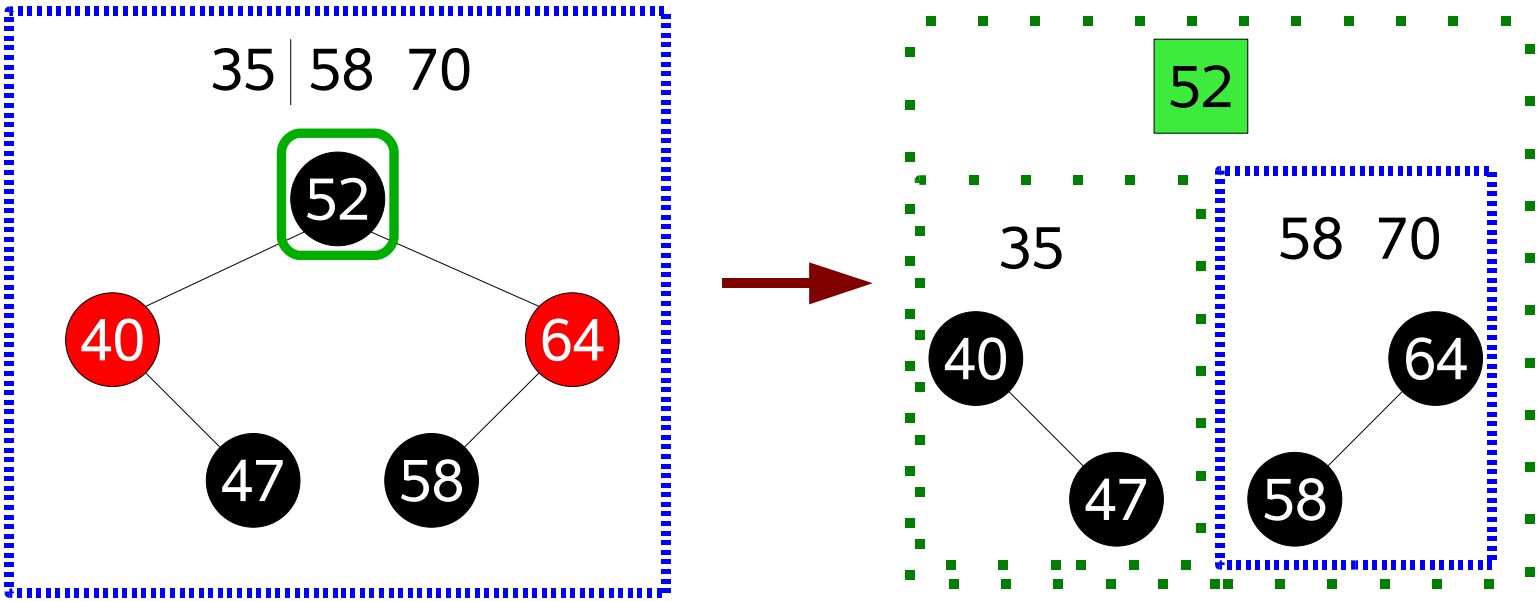
Dynamic Load-Balancing: Approach

Division of insertion tasks into smaller ones.
Creation of concatenation tasks to reestablish the tree.



Dynamic Load-Balancing: Approach

Division of insertion tasks into smaller ones.
Creation of concatenation tasks to reestablish the tree.



Use (MCSTL) work-stealing queue.

Memory Management

Memory allocation takes a considerable share of the time.

C++ does not allow asymmetric allocation/deallocation, i. e. allocate several nodes at once, and then, deallocate one by one.

Memory Management

Memory allocation takes a considerable share of the time.

C++ does not allow asymmetric allocation/deallocation, i. e. allocate several nodes at once, and then, deallocate one by one.

Parallelization?

Memory Management

Memory allocation takes a considerable share of the time.

C++ does not allow asymmetric allocation/deallocation, i. e. allocate several nodes at once, and then, deallocate one by one.

Parallelization?

Allocation + initialization scales quite well.

Hoard allocator [BMBW00] used successfully on the Sun machine, but not in the 64-bit Intel Platform.

Environment

1. Sun T1
   - 1 socket, 8 cores, 1.0 GHz, 32 threads
   - 3 MB shared L2 cache
   - GCC 4.2.0
2. Intel Xeon E5345
   - 2 sockets, 2×4 cores, 2.33 GHz
   - $2 \times 2 \times 4$ MB L2 cache, shared among two cores each
   - Intel C++ compiler 10.0.25

Compiler/linker options:
-O3, OpenMP support, libstdc++ (GCC 4.2.0).

Parameters

Sequence

- Presorted, otherwise unfair
- 32-bit signed integers elements
- Randomness: {RAND_MIN...RAND_MAX} (default), {RAND_MIN/100...RAND_MAX/100} (limited range).

Parameters

Sequence

- Presorted, otherwise unfair
- 32-bit signed integers elements
- Randomness: {RAND_MIN...RAND_MAX} (default), {RAND_MIN/100...RAND_MAX/100} (limited range).

Initial tree for insertion tests: built by sequential algorithm

Parameters

Sequence

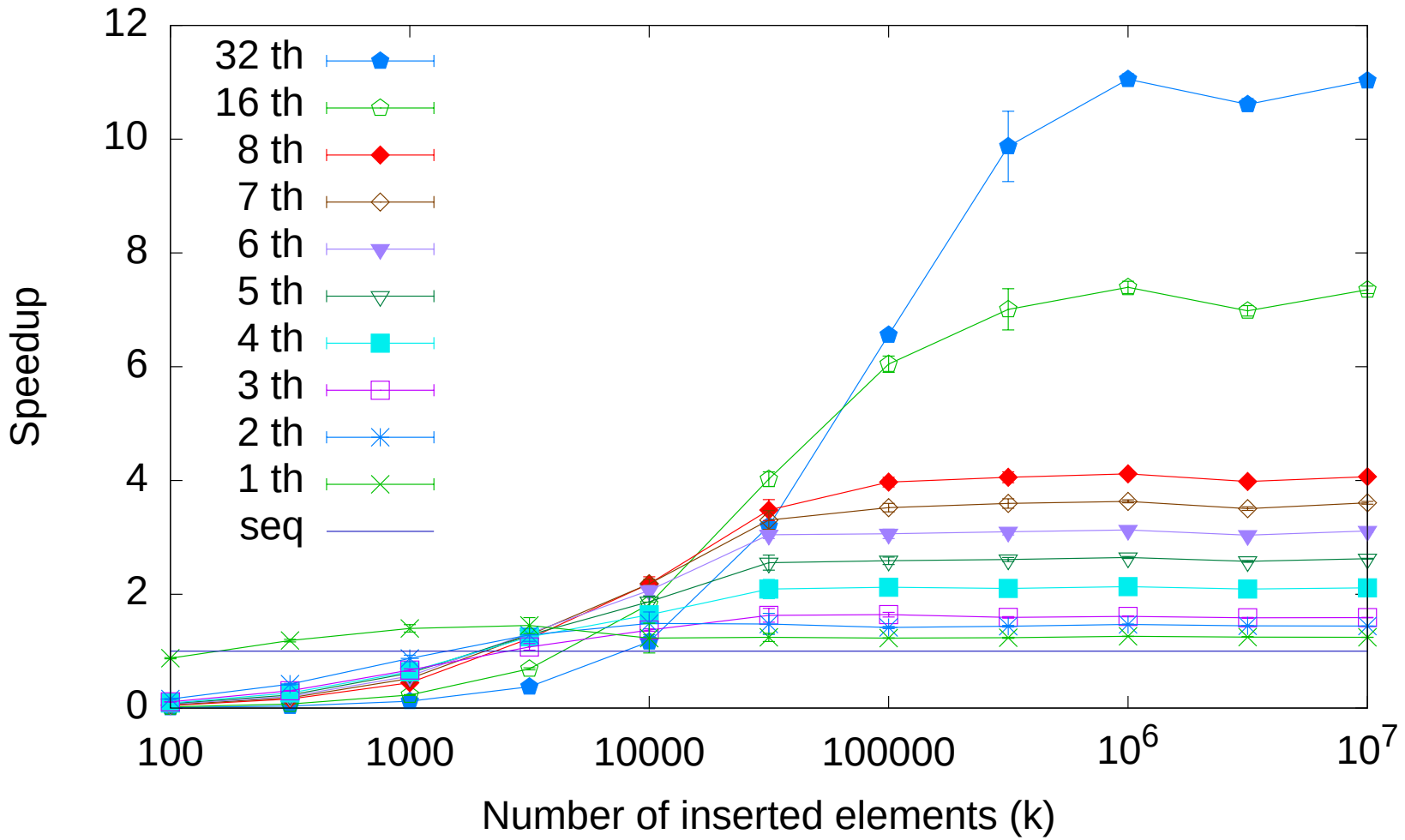
- Presorted, otherwise unfair
- 32-bit signed integers elements
- Randomness: {`RAND_MIN`...`RAND_MAX`} (default), {`RAND_MIN`/100...`RAND_MAX`/100} (limited range).

Initial tree for insertion tests: built by sequential algorithm

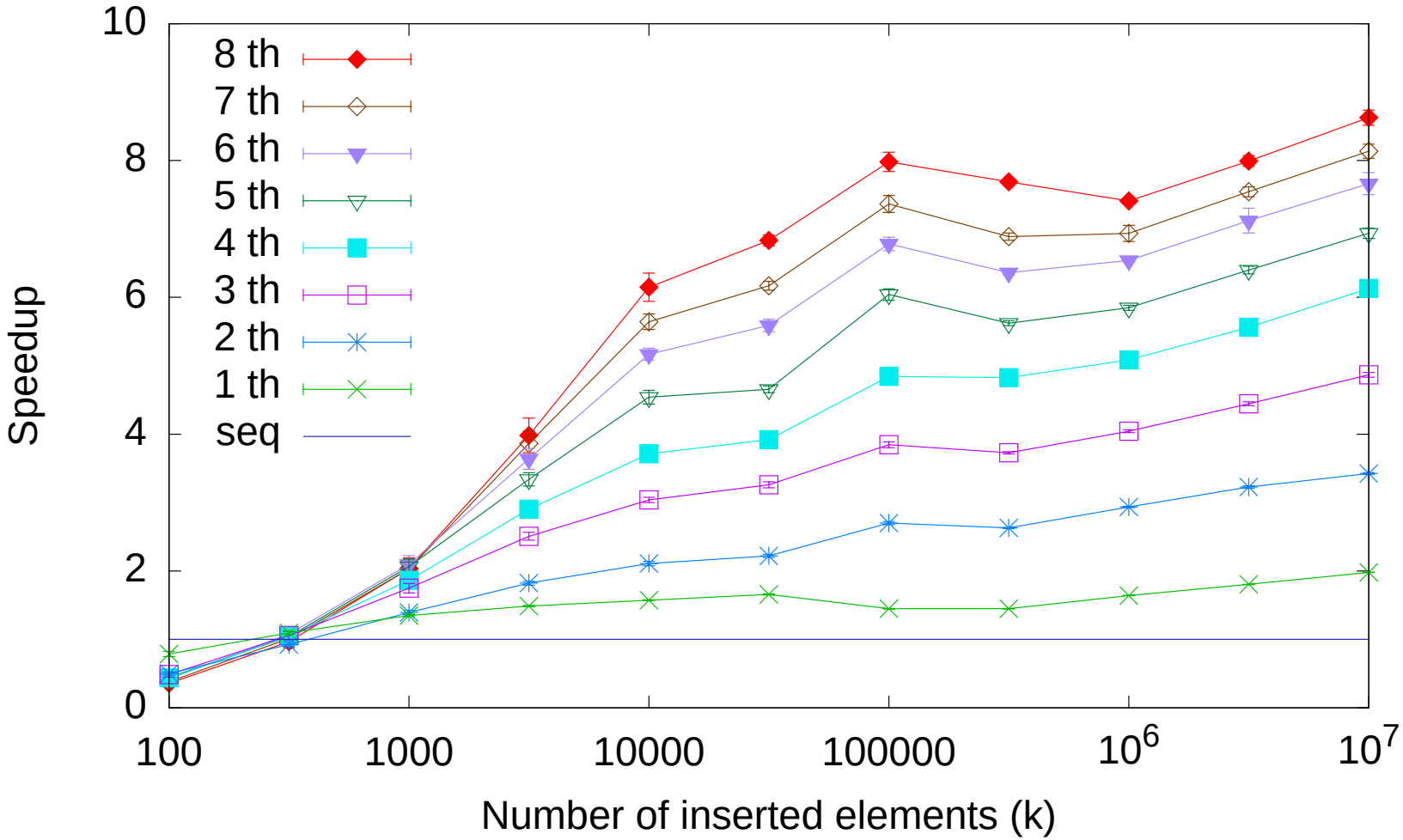
Algorithm variants:

- Dynamic load-balancing: activated (default), deactivated
- Initial tree splitting: activated (default), deactivated

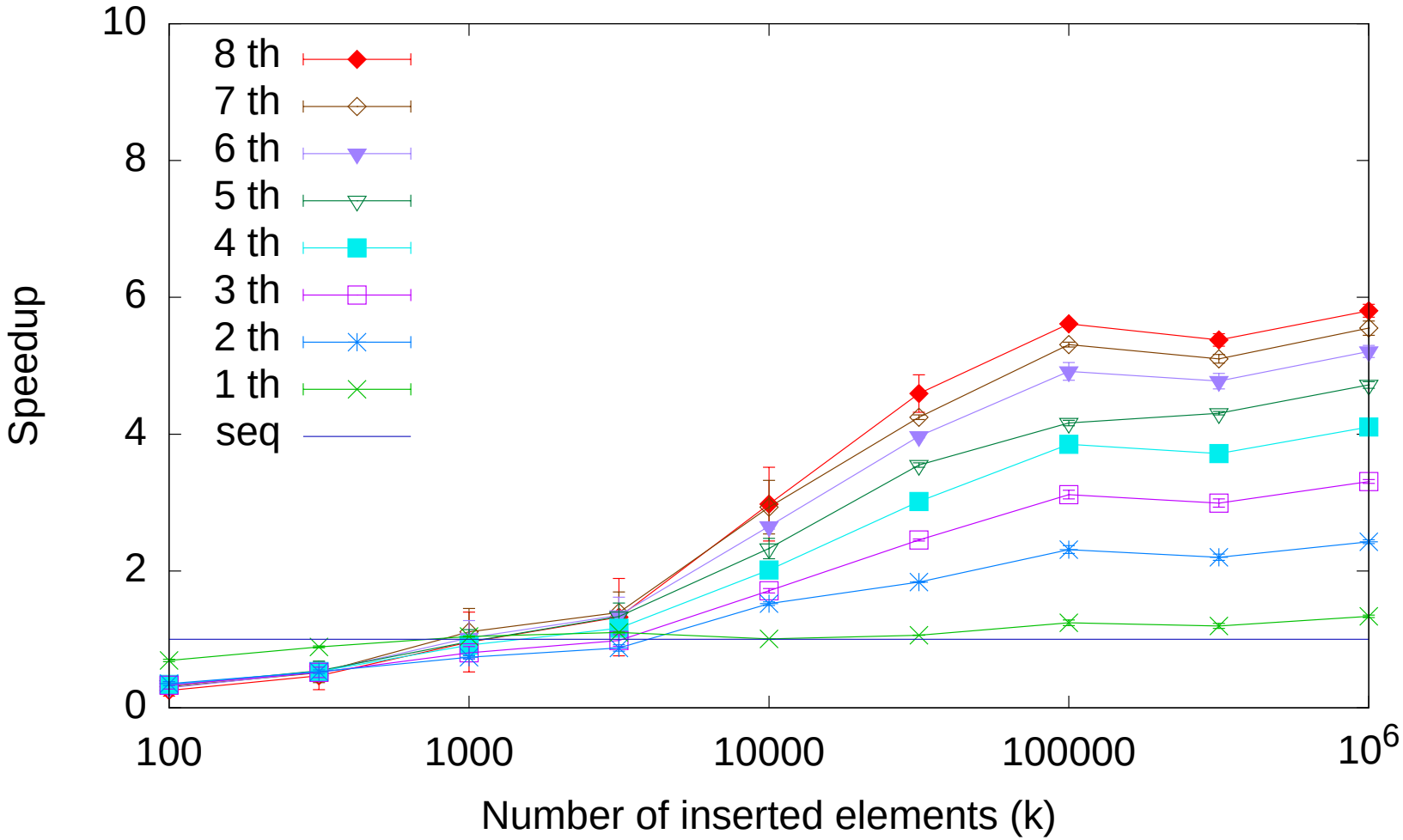
Construction on the T1



Insertion, $n = 0.1k$, Xeon



Insertion, $n = 10k$, Xeon



Effect of Algorithm Variants

Initial tree splitting:

- crucial for good performance

Effect of Algorithm Variants

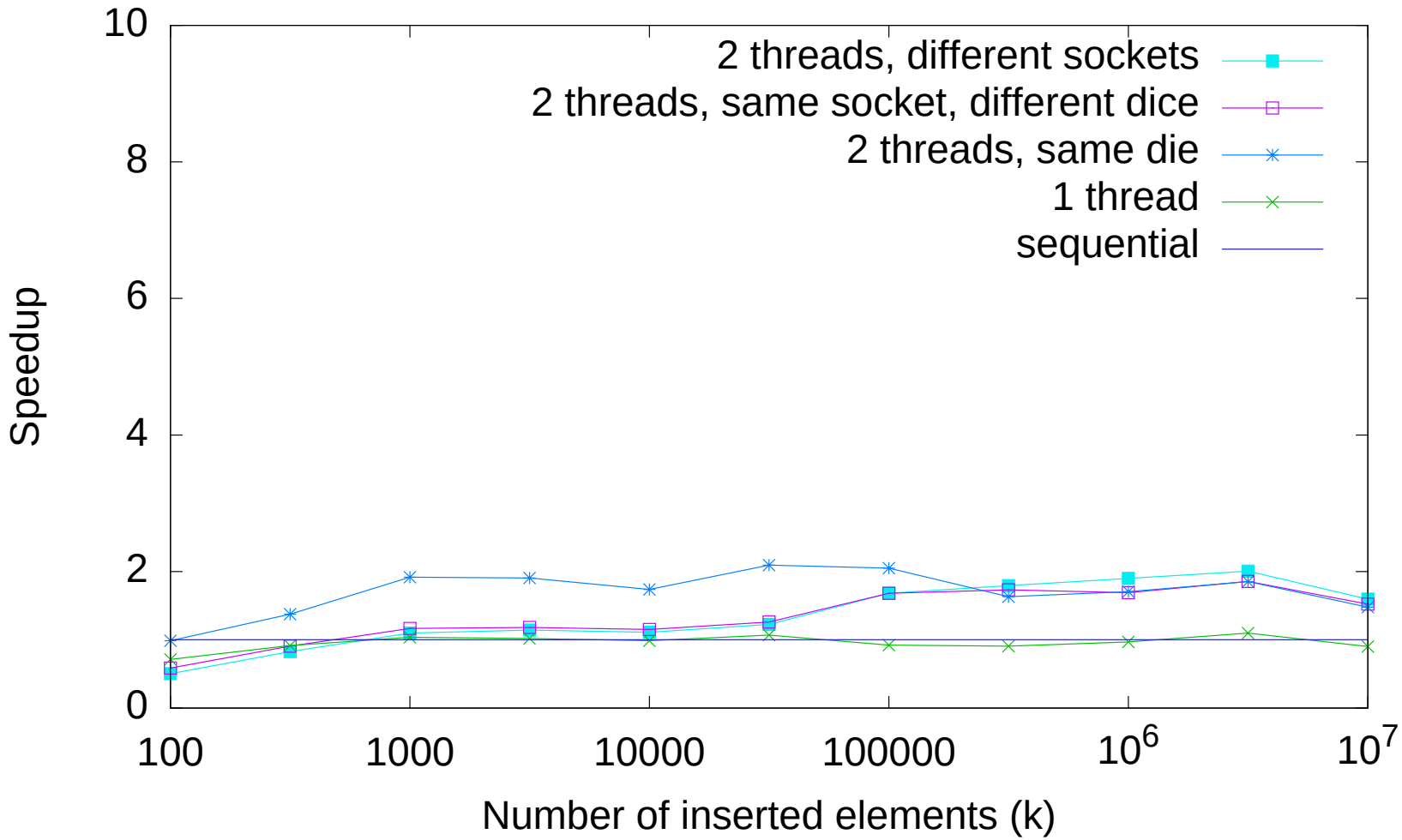
Initial tree splitting:

- crucial for good performance

Dynamic-load balancing:

- Robustness
- Improves performance for big input sizes
- No damage for small input sizes

Effect of Core Mapping (Construction)



Conclusions

Bulk construction and insertion can be effectively parallelized. On the top of the sequential libstdc++ implementation. This remains unaffected.

Code for the four STL associative containers is now released with the MCSTL [Sin06].

Future directions

Lazy update: split sequences of library calls into homogeneous subsequences of maximal length.
Could be offered transparently by the library.

Better library support for memory allocation:

- Asymmetric usage
- Perfectly scalable parallel allocation

Future Work: Algorithms and Data Structures

Other tree operations to parallelize

- Bulk deletion of elements using `remove_if`
- `set_difference` for dictionaries

Other data structures

- Search tree storing subtree sizes: allows perfect tree size partitioning in logarithmic time
  - What is the effect on performance?
- Priority queues: lazy parallel data structure update

Further Performance Evaluation

Detailed evaluation of hardware counters: cache misses, branches, limitation by (random-access) memory bandwidth

References


P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger.
STAPL: An Adaptive, Generic Parallel C++ Library.
In *LCPC*, pages 193–208, 2001.
http://parasol.tamu.edu/groups/rwergergroup/research/stapl/.


Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson.
Hoard: A scalable memory allocator for multithreaded applications.
In *ASPLOS-IX*, 2000.


Heejin Park and Kunsoo Park.
Parallel algorithms for red-black trees.
Theoretical Computer Science, 262:415–435, 2001.


Johannes Singler.
The MCSTL website, June 2006.
http://algo2.iti.uni-karlsruhe.de/singler/mcstl/.


Johannes Singler, Peter Sanders, and Felix Putze.
The Multi-Core Standard Template Library.
In *Euro-Par 2007: Parallel Processing*, 2007.