The GNU libstdc++ parallel mode: Benefit from Multi-Core using the STL

Johannes Singler, Peter Sanders

{singler,sanders}@ira.uka.de

Institute for Theoretical Computer Science
University of Karlsruhe

Talk Outline

Introduction

Library Overview

Algorithms

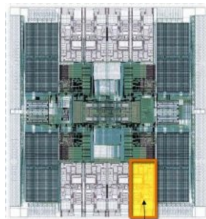
Software Engineering Aspects

Applications

Conclusion

Motivation

How to Benefit from Multi-Core Systems?



- **automatic parallelization** not sufficient
- manual/explicit parallelization needed, but expensive, complicated, error-prone, **not easy to try-out**

Our Approach

- provide a **data-parallelized library** of **basic algorithms** for shared-memory systems
  - **libraries** are an important aspect of **Algorithm Engineering**
- make the usage of (data-)parallel algorithms **very easy**
  - actual parallelism not visible to the user, but **encapsulated**

Basic Approach

Starting Point

- provide the functionality of the C++ **S**tandard **T**emplate **L**ibrary
- run the algorithms **in parallel**
- included with GCC as of version 4.3: **libstdc++ parallel mode**
  - formerly known as the Multi-Core Standard Template Library

Why STL?

- many **useful** algorithms and data structures included
- simple interface, very **well-known** among developers
- recompilation of **existing programs** may suffice
- C++ **accepted** and efficient language, standardized

Goals

Ease of Use

- **no new** language, **no language extension**
- just **few compiler options** to activate

Good Performance

- **some speedup** already for small inputs $\Rightarrow$**scale down**
- **full speedup** for larger inputs $\Rightarrow$**scale up**
- co-exist with other forms of parallelization
  - respect machine load $\Rightarrow$**dynamic load-balancing**

Competitors

STAPL

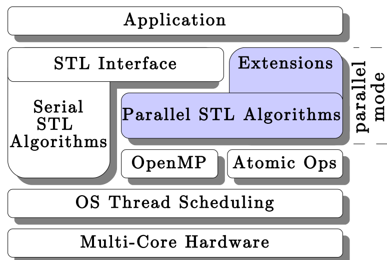
- must incorporate **distributed-memory issues**
- no code **publicly available**
- interface **only similar** to STL

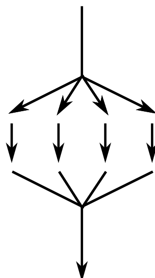
Intel Threading Building Blocks

- mostly on a **more abstract level**,
  parallel programming framework
- only combinatorial generic algorithm is **parallel sorter**

Platform Support

- based on **OpenMP** (fork-join parallelism)





- **low-level issues** are left to the OpenMP runtime, e. g. thread pooling, environment, synchronization primitives
  - **GCC**'s upcoming implementation **improved** by us
- **platform-independent**, supported by all major C++ compilers
- **task** construct upcoming
- **+** atomic operations (platform-dependent)

Overview of Important Algorithms

Strictly STL (mostly `<algorithm>`)

- `for_each` and friends (embarrassingly parallel)
- `find`
- `partial_sum` (prefix sum)
- `partition`, `partial_sort`
- `merge`
- `sort`
- `random_shuffle`
- bulk construction and bulk `insert` for `set` and `map` [1]

Extension to STL

- `multiway_merge`
- `multiseq_partition` (helper)

[1] not part of parallel mode yet

Parallel Mode Development Status

Algorithm Class	Function Call(s)	Status	w/LB	w/oLB
Embarrassingly Parallel	`for_each`, `generate(_n)`, `fill(_n)`, `count(_if)`, `transform`, `replace(_if)`, `min_element`, `max_element`, `adjacent_difference`, `unique_copy`	impl	yes	yes
Find	`find(_if)`, `find_first_of`, `adjacent_find`, `mismatch`, `equal`, `lexicographical_compare`	impl	yes	not worth-while
Search	`search(_n)`	impl	yes	not ww.
Numerical Algorithms	`accumulate`, `partial_sum`, `inner_product`	impl	planned	yes
Partition	`partition`, *`stable_partition`*	impl	yes	not ww.
Merge	`merge`, `multiway_merge`, *`inplace_merge`*	impl	tbi	yes
Partial Sort	`nth_element`, `partial_sort`	impl	yes	planned
Sort	`sort`, `stable_sort`	impl	yes	yes
Random Permutation	`random_shuffle`	impl	yes	not worthw.
Dictionaries	`(multi_)map/set`	bulk op	tbi	tbi
Complex Set Operations	`set_union`, `set_intersection`, `set_(symmetric_)difference`, ...	impl	no	yes
Vector Arithmetic	*`valarray` operations*	ongoing	yes	yes
Heap Construction	*`make_heap`*, *`sort_heap`*	tbi		
Priority Queues	amortized update operations	ongoing		
Filtering	`remove(_copy)(_if)`	ongoing		

`for_each` Implementation

Definition

- **execute** a certain function **on a range** of elements
- many similar functions like `transform`, `generate`
- parallelization is **easy only for**
  uniform execution time per element, exclusive machine

Static Load-Balancing

- divide work into **parts of almost equal size**
- used for **`accumulate`**,
  since ends of chunks can easily be spliced (not commutative)

Dynamic Load-Balancing

- **initially** divide work into **parts of almost equal size**
- allow “unemployed” threads to take work from others
  ⇒**work-stealing**

`merge`, `multiway_merge`

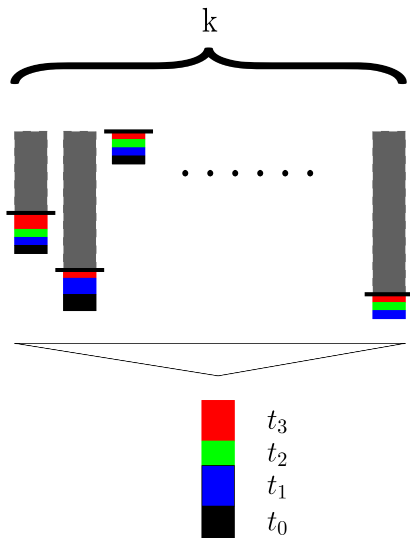
Problem Definitions

- `merge`: combine **two sorted** sequences into one sorted sequence
- `multiway_merge`:
  combine $k > 2$ **sorted sequences** into one sorted sequence
  - important for (external memory) sorting

How to divide the input?

- find slabs, i. e. **consistent sets** of ranges from the sequences
- two possibilities:
  - (randomized) splitting by **sampling**
  - exact partitioning into slabs of equal size
    (using **multi-sequence selection**) [6]

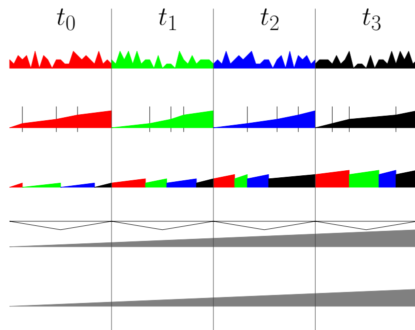
`merge`, `multiway_merge`: Diagram



Parallel Multiway Mergesort

Procedure

1. **divide sequence** into p parts of equal size
2. in parallel, **sort the parts locally**
3. use parallel ***p*-way merging** to compute the final sequence
4. **copy** result **back** to original position



sort, stable_sort

Parallel Multiway Mergesort

- \+ **less communication** necessary
- \+ **stable** variant easy to derive
- – needs **twice** the space

Parallel Load-Balanced Quicksort

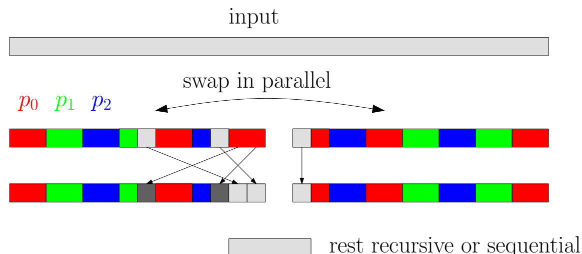
- \+ **in-place**
- ± **dynamic load-balancing** to compensate for unequal splitting
- – **concurrent access** to memory
- – **not** stable

Both variants implemented in the parallel mode, user's choice.

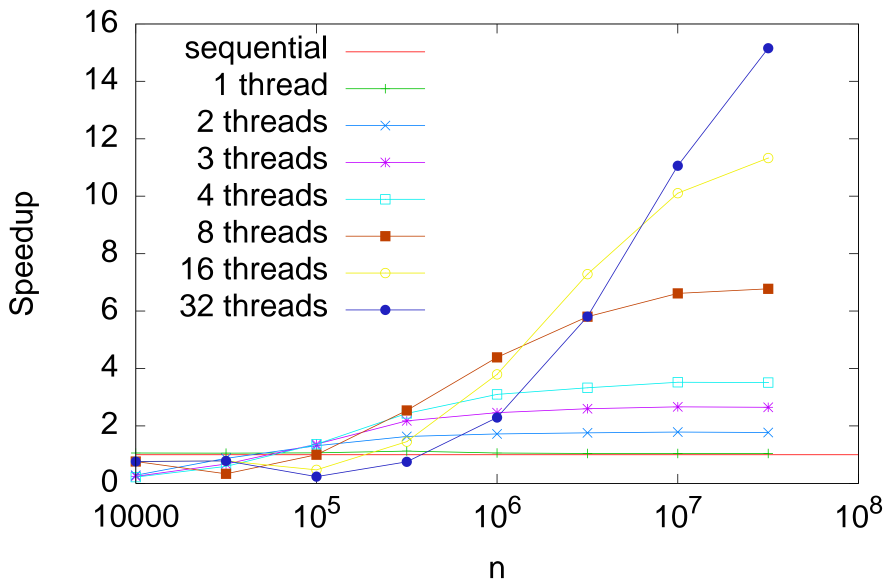
Parallel Partitioning

[Tsigas Zhang 2003]

1. scan blocks of size B from both ends
   1.1 claim new blocks when running out of data
2. swap the unfinished blocks to the “middle”
3. recurse on the middle



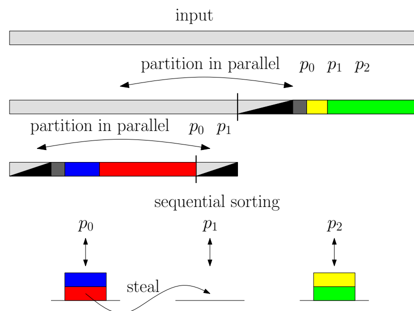
- **time complexity** $O(n/p + B \log p)$


Partitioning of 32-bit integers on Sun T1
Speedup
n
sequential
1 thread
2 threads
3 threads
4 threads
8 threads
16 threads
32 threads
0
2
4
6
8
10
12
14
16
10000
10^5
10^6
10^7
10^8

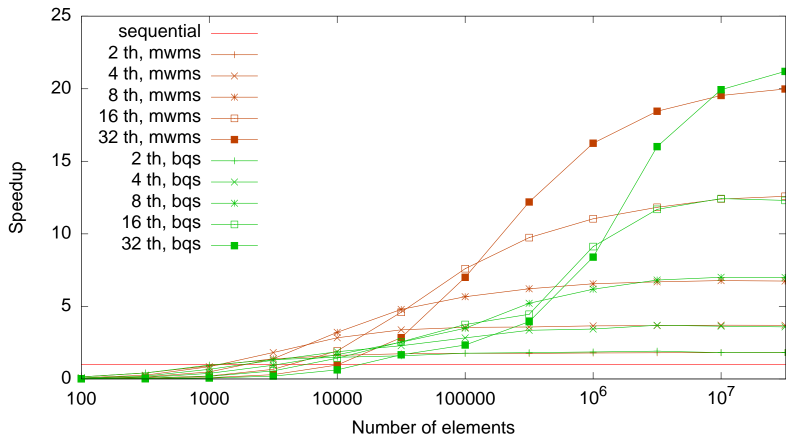
Parallel Balanced Quicksort

Procedure [5]

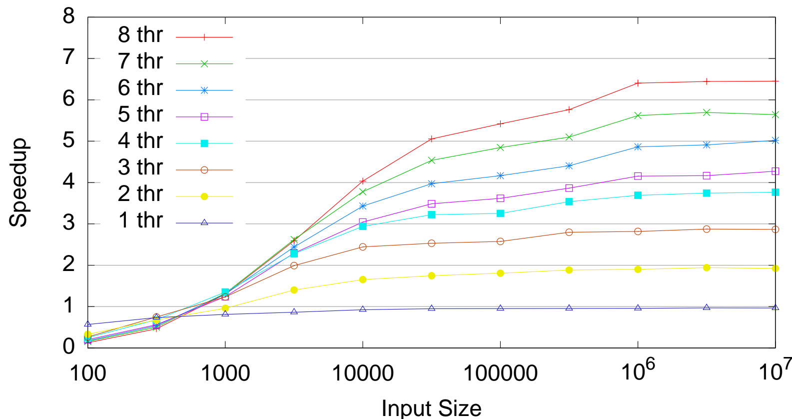
1. **split sequence** using parallel `partition`, **descend recursively** with the appropriate number of threads
2. as soon as there is only one processor left per partition: start **local sorting**
3. each processor sorts locally, **pushes** parts to process later into **lock-free dequeue**
4. other processors can **steal** parts when out of work



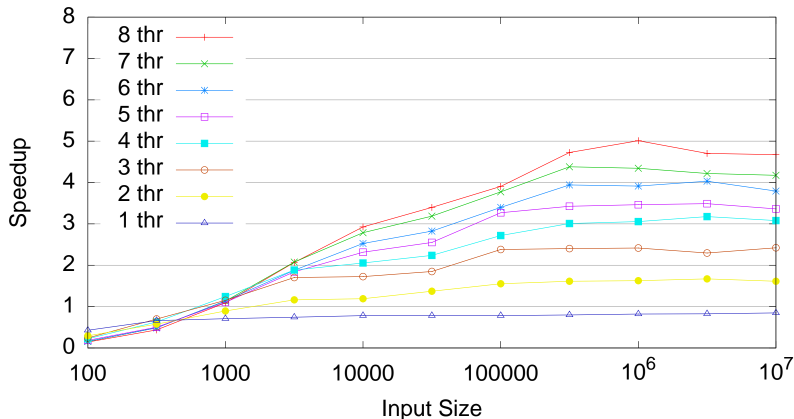
Sorting Performance Results



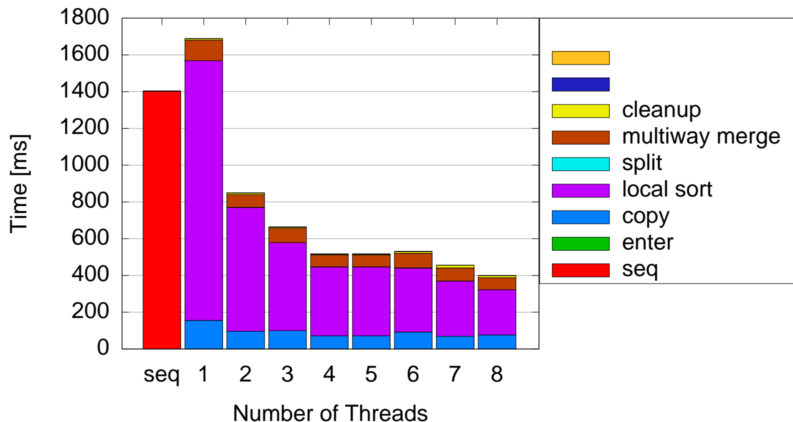
Sorting Performance Results



Sorting Performance Results



Detail Timings



Dictionary Bulk Operations

Algorithmic Problem

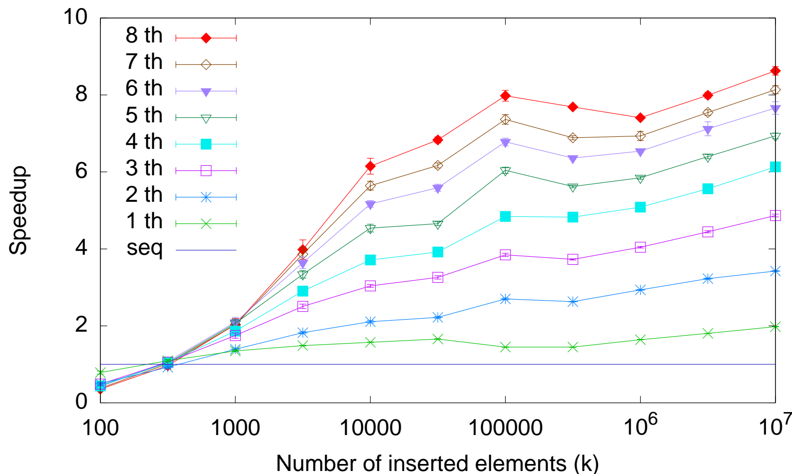
- construct/insert into **red-black tree**
- complicated **splitting and balancing** of work
- **bulk algorithm** already brings sequential speedup
- not yet in parallel mode, but only in MCSTL

Memory Management

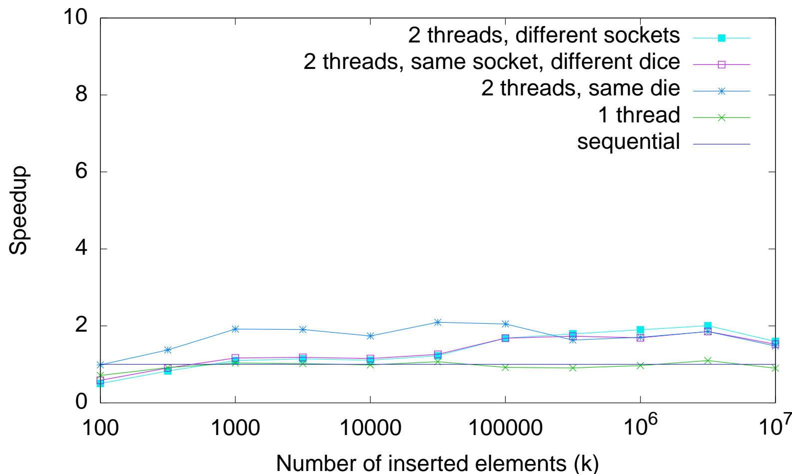
- **memory allocation** takes a considerable share of the time
- C++ does **not** allow **asymmetric** allocation/deallocation,
  i. e. allocate several nodes at once, later deallocate one by one

Dictionary Bulk Operations Performance

Insertion, n=0.1k, 2-way Quadcore Xeon



Effect of Core Mapping (Dictionary Bulk Construction)



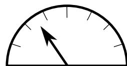
General Speedup Insight

- memory **bandwidth** is usually **shared**
  - the more **computation per memory accesses**, the better
  - highly **depending on user-defined functors**
  - large **shared cache** improves situation
  - cannot compete with computation in L1 cache
- **thread starting overhead is constant (1 microsecond) after first time**
  - the more **input per algorithm call**, the better
- simple heuristic: **stay sequential for too small inputs** to not worsen performance in bad cases

Software-Engineering-Related Goals

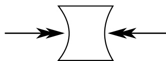
- **transparent** integration of parallel algorithms
  - compile in **sequential or** in **parallel** mode
  - pragmatic **balance** between **standard adherence** and **benefits**
    - sequential semantics, exceptions, space requirements

parallel
sequential



- **selection and tuning** of algorithms

- **maintainability**:
  little code **duplication**, much code **reuse**
  build on **existing infrastructure**



- **limited increase** in compilation time and executable size for the user application.

Usage

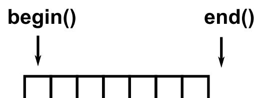
Example Code

```
#include <algorithm>
vector<double> v(1000000);
std::random_shuffle(v.begin(), v.end());

g++-4.3.1 -D_GLIBCXX_PARALLEL -fopenmp random_shuffle.cpp
```

Sequence Access through Iterators

Most STL algorithms take **one or more sequences** as main argument(s).

- 

- iterators might have **restrictions**, e. g. no random access
- **information** gets **lost**,
  e. g. length in linked list, inefficient to recompute
- *data-parallelism* $\rightsquigarrow$ **efficient splitting absolutely necessary**
  - "best-effort solution": single-pass splitting without copying [2]
- decision on whether **random access** at **compile-time**
  - `iterator_traits` must be available, also for custom iterators

Binary Size and Compilation Time

Provide compile-time switches to user, in order to limit increase in executable size and compilation time

```
#include <vector>
#include <algorithm>
int main() {
  std::vector<int> vi(100000000);
  std::sort(vi.begin(), vi.end(), TAG);
}
```

Binary Size and Compilation Time

Provide compile-time switches to user, in order to limit increase in executable size and compilation time

```
#include <vector>
#include <algorithm>
int main() {
  std::vector<int> vi(100000000);
  std::sort(vi.begin(), vi.end(), TAG);
}
```

Executable size and compilation time for different variants

Algorithm Variant(s)	Size (B)	Time (s)
Sequential	15 479	0.74
Quicksort	22 387	1.49
Balanced Quicksort	26 989	1.84
Multiway-Mergesort Sampling	36 002	3.49
Default (Multiway-Mergesort Exact)	41 229	4.68
Multiway-Mergesort Exact	41 237	4.78
Multiway-Mergesort (splitting choice at run-time)	46 003	5.48
All Parallel Variants (run-time choice)	61 543	6.50

Combination with Other Libraries

STXXL: external memory STL

- parallelization of **internal computation**, e. g. sorting, multi-way merging
- **+ task-paralleli**zation framework

CGAL: computational geometry

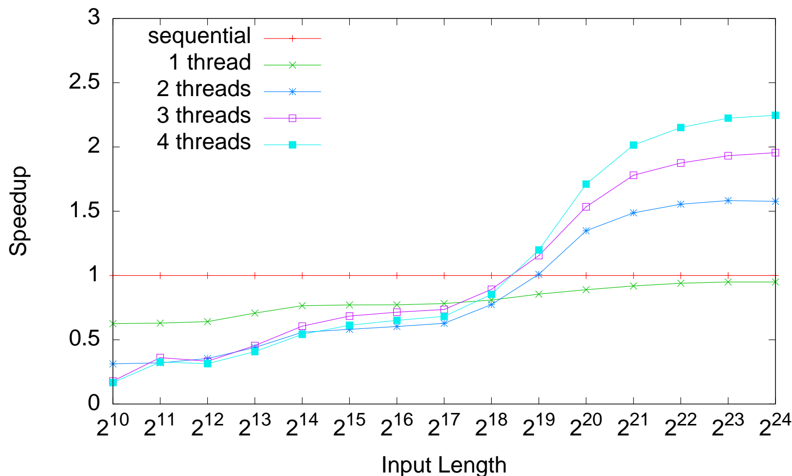
- mostly **preprocessing**: sorting, random shuffling
- **+ manually** parallelized geometric algorithms

Upcoming: distributed external memory sorting

- add **shared**-memory parallelism to **distributed**-memory algorithm easily
- library?

Use Case 1: Suffix Array Construction

parallel mode + manually parallelized integer sorter



Use Case 2: Minimum Spanning Tree Construction

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
  if $m \leq$ kruskalThreshold($n, |E|, |T|$)
    then kruskal(E, T, P)
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq p \rangle$; $E_{>} := \langle e \in E : e > p \rangle$
    filterKruskal($E_{\leq}, T, P$)
    $E_{>} :=$ filter($E_{>}, P$)
    filterKruskal($E_{>}, T, P$)

Function filter(E)
  return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

Use Case 2: Minimum Spanning Tree Construction

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
    if $m \leq$ kruskalThreshold($n, |E|, |T|$)
        then kruskal(E, T, P)
    else
        pick a pivot $p \in E$
        $E_{\leq} := \langle e \in E : e \leq p \rangle$; $E_{>} := \langle e \in E : e > p \rangle$
        filterKruskal($E_{\leq}, T, P$)
        $E_{>} :=$ filter($E_{>}, P$)
        filterKruskal($E_{>}, T, P$)

Function filter(E)
    return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

Use Case 2: Minimum Spanning Tree Construction

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
  if $m \leq$ kruskalThreshold($n, |E|, |T|$)
    then kruskal(E, T, P)
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq p \rangle$; $E_{>} := \langle e \in E : e > p \rangle$
    filterKruskal($E_{\leq}, T, P$)
    $E_{>} :=$ filter($E_{>}, P$)
    filterKruskal($E_{>}, T, P$)

Function filter(E)
  return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

Use Case 2: Minimum Spanning Tree Construction

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
  if $m \leq$ kruskalThreshold($n, |E|, |T|$)
    then kruskal(E, T, P)
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq p \rangle$; $E_{>} := \langle e \in E : e > p \rangle$
    filterKruskal($E_{\leq}, T, P$)
    $E_{>} :=$ filter($E_{>}, P$)
    filterKruskal($E_{>}, T, P$)

Function filter(E)
  return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

Use Case 2: Minimum Spanning Tree Construction

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
  if $m \leq$ kruskalThreshold($n, |E|, |T|$)
    then kruskal(E, T, P) **parallel**
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq p \rangle$; $E_{>} := \langle e \in E : e > p \rangle$ **parallel**
    filterKruskal($E_{\leq}, T, P$)
    $E_{>} :=$ filter($E_{>}, P$) **parallel**
    filterKruskal($E_{>}, T, P$)

Function filter(E)
  return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

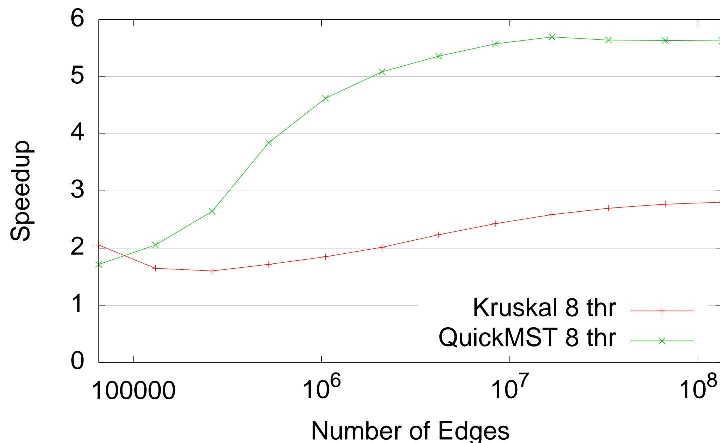
Use Case 2: Minimum Spanning Tree Construction

Use STL algorithms from *libstdc++* ***parallel*** *mode*

Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
  if $m \leq$ kruskalThreshold($n, |E|, |T|$)
    then kruskal(E, T, P)   `sort`
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq p \rangle$; $E_{>} := \langle e \in E : e > p \rangle$ `partition`
    filterKruskal($E_{\leq}, T, P$)
    $E_{>} :=$ filter($E_{>}, P$)   `remove_if`
    filterKruskal($E_{>}, T, P$)

Function filter(E)
  return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

Use Case: Minimum Spanning Tree Construction



QuickMST is **still** faster in absolute time [3] (ALENEX 2009).

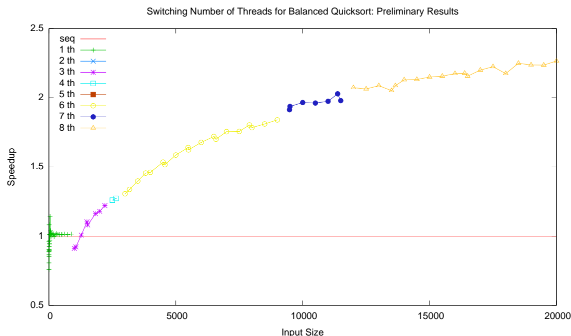
Conclusions

Benefits

- parallel mode provides a **easy way** to incorporate **data parallelism** into programs on an algorithmic level
- fully **generic**
- performance is **good for large inputs**
- speedup at hand for **small inputs** as well, depending on circumstances
- could transparently **support new paradigms**, e. g. transactional memory
- **repository** for parallel algorithm implementations

Future Work

- **integration** of missing algorithms
- performance estimation ⇒automatic **switching point** detection



- working **affinity**

Thank you for your attention. Questions?

References

L. Frias and J. Singler.
Parallelization of bulk operations for STL dictionaries.
In *Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2007.

L. Frias, J. Singler, and P. Sanders.
Single-pass list partitioning.
Scalable Computing: Practice and Experience, 9(3):179–184, 2008.

V. Osipov, P. Sanders, and J. Singler.
The filter-kruskal minimum spanning tree algorithm.
In *11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2009.

J. Singler, P. Sanders, and F. Putze.
The Multi-Core Standard Template Library.
In *Euro-Par 2007: Parallel Processing*, volume 4641 of *LNCS*, pages 682–694. Springer-Verlag.

P. Tsigas and Y. Zhang.
A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000.
In *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, page 372, 2003.

P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard.
Merging Multiple Lists on Hierarchical-Memory Multiprocessors.
Journal of Parallel and Distributed Computing, 12(2):171–177, 1991.