MCSTL: Multi-Core Standard Template Library

Practical Implementation of Parallel Algorithms for Shared-Memory Systems

Peter Sanders, *Johannes Singler*

Institute for Theoretical Computer Science
University of Karlsruhe

April 29th, 2008

Lecture Contents

Outline

Model

Theory $\rightsquigarrow$ Practice

- machine model $\rightsquigarrow$ concrete machine(s)
- pseudo-code $\rightsquigarrow$ existing C++ library

Model

Theory $\rightsquigarrow$ Practice

- machine model $\rightsquigarrow$ concrete machine(s)
- pseudo-code $\rightsquigarrow$ existing C++ library

Communication Network $\rightsquigarrow$ Shared Memory

- implicit communication
  - cache hierarchy, NUMA, bandwidth sharing

Model

Theory $\rightsquigarrow$ Practice

- machine model $\rightsquigarrow$ concrete machine(s)
- pseudo-code $\rightsquigarrow$ existing C++ library

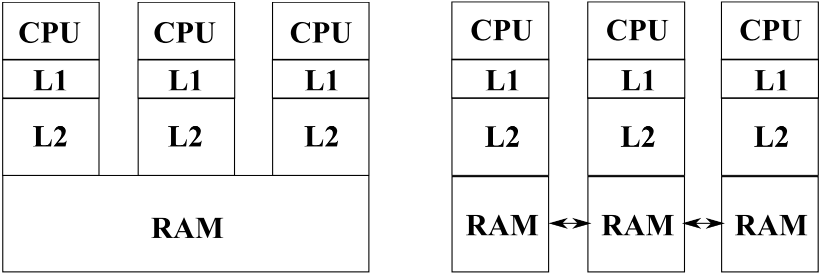
Communication Network $\rightsquigarrow$ Shared Memory

- implicit communication
  - cache hierarchy, NUMA, bandwidth sharing

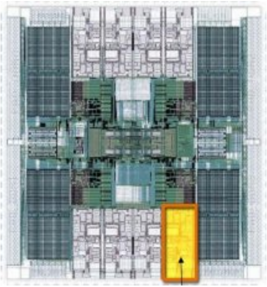
Synchronous PRAM $\rightsquigarrow$ Asynchronous PEs

- synchronization a problem itself
- $n = p \rightsquigarrow n \gg p$
- core allocation not static, other processes interfere

Memory Models Refined

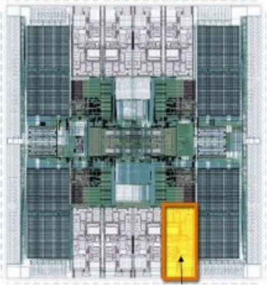


Programming Multicores



- automatic parallelization?
  only for simple loops
- explicitly parallel?
  too complicated for everyday use
- libraries of parallelized algorithms!

Programming Multicores



- automatic parallelization?
  only for simple loops
- explicitly parallel?
  too complicated for everyday use
- libraries of parallelized algorithms!

natural starting point:
standard libraries of programming languages

Basic Approach

Make Using Parallel Algorithms
“as easy as winking”.
Functionality of the C++ Standard Template Library

Basic Approach

Make Using Parallel Algorithms
“as easy as winking”.
Functionality of the C++ Standard Template Library

Why STL?

- many efficient and useful algorithms included
- interface very well-known among developers
- template mechanism is known to allow low overhead algorithm libraries
- recompilation of existing programs may suffice
- C++ accepted and efficient language

Goals

- parallelize all time consuming STL algorithms
- speedup already for small inputs $\rightsquigarrow$ scale down
- high speedup for medium/large inputs

Special Requirements for a Library

Generality

- genericity (templates)
- only few assumptions about input data types
- good scalability in terms of use cases

Special Requirements for a Library

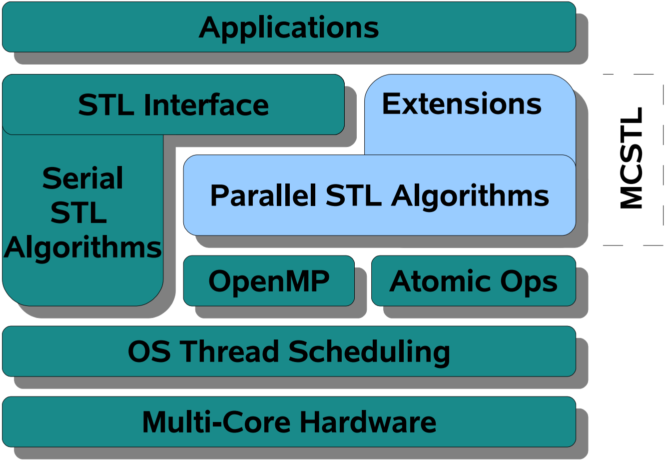
Generality

- genericity (templates)
- only few assumptions about input data types
- good scalability in terms of use cases

Compatibility to

- existing libraries
- platforms

Layers


Applications
STL Interface
Extensions
Serial STL Algorithms
Parallel STL Algorithms
MCSTL
OpenMP
Atomic Ops
OS Thread Scheduling
Multi-Core Hardware

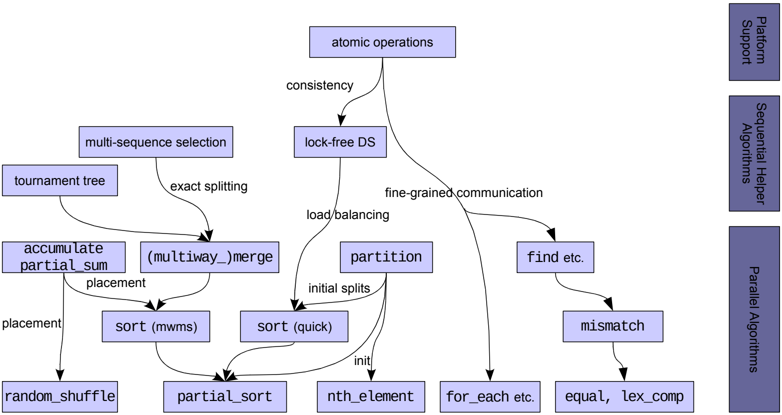
Implemented Algorithms

- embarrassingly parallel (`for_each`, `transform`,. . .)
- `find`, `find_if`, `mismatch`, . . .
- `partial_sum` (prefix sum)
- `partition`
- `nth_element/partial_sort`
- `merge`
- `sort`, `stable_sort`
- `random_shuffle`
- `(multi_)set/map::insert`

Extension to STL

- `multiway_merge`

Dependency Graph of Lecture Contents



Outline

Introduction

Platform Support

Algorithms

Conclusion

Shared-Memory Hardware

- cache coherency protocol makes memory view consistent, introduces implicit communication
  - cores invalidate entries in cache when other core writes (snooping)
  - overhead only for actual transfer of data
  - granularity is one cache-line: avoid false sharing!

Threading Support

- OpenMP: basic primitives.
  - example

```
#pragma omp parallel num_threads(p)
{ num_threads = omp_get_num_threads();
 iam = omp_get_thread_num(); ...
 #pragma omp barrier/single/master
...}
```

  - quite elegant
  - no permanent separation possible (asynchrony)
  - still works when compiler ignores pragmas
  - good compiler support (GCC, Sun, Intel, MS)

Threading Support

- OpenMP: basic primitives.
  - example

```
#pragma omp parallel num_threads(p)
{ num_threads = omp_get_num_threads();
 iam = omp_get_thread_num(); ...
 #pragma omp barrier/single/master
...}
```

  - quite elegant
  - no permanent separation possible (asynchrony)
  - still works when compiler ignores pragmas
  - good compiler support (GCC, Sun, Intel, MS)
- atomic operations
  - fetch-and-add, compare-and-swap

Atomic Operations

a few operations are executed without any chance of interference $\rightsquigarrow$ atomically

- `fetch_and_add(x, i)`
  - `t := x; r := x; r := r + i; x := r; return t;`
  - allows concurrent iteration over sequence

Atomic Operations

a few operations are executed without any chance of interference $\rightsquigarrow$ atomically

- `fetch_and_add(x, i)`
  - `t := x; r := x; r := r + i; x := r; return t;`
  - allows concurrent iteration over sequence
- `compare_and_swap(x, c, r)`
  - `if(x = c) { x := r; return c [true]; } else { return r [false]; }`
  - secure state transition, can emulate `fetch_and_add` and others by using in a loop
- slower than usual operation, in particular when concurrent

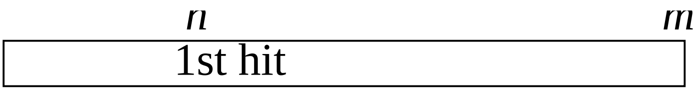
Outline

find, find_if, mismatch,...

find the first position in a sequence satisfying a predicate

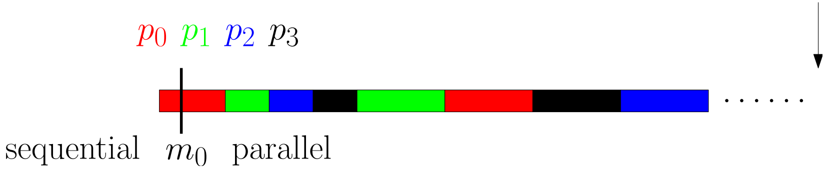
Analysis

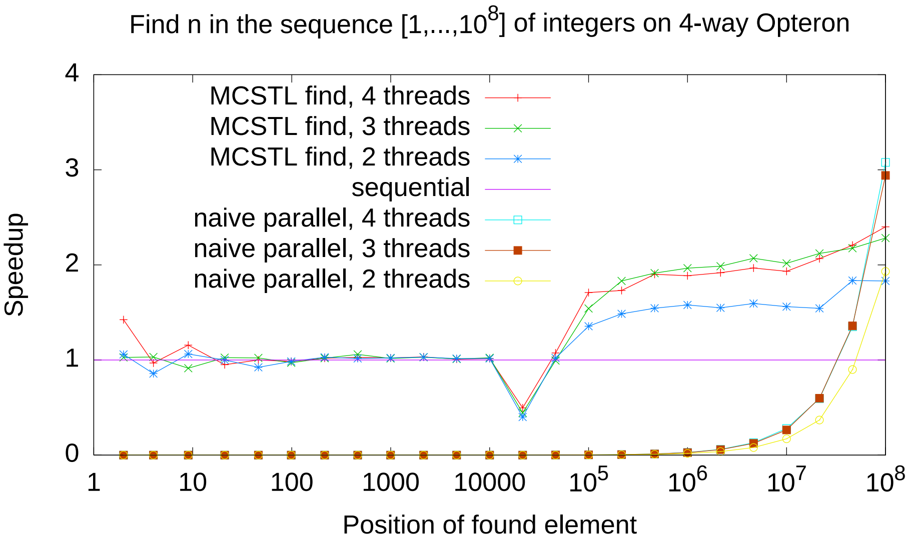
- $O(n)$ sequential time if first hit is at position n (unknown)
- naïve parallel algorithm needs $\Omega(m/p)$.
- parallelization not worthwhile for small n



`find`: Algorithm

- start sequentially up to position m_0
- dynamic load balancing using fetch-and-add
- scale up and down
  using geometrically growing block sizes
- first successful thread grabs remaining work




Find n in the sequence [1,...,10^8] of integers on 4-way Opteron
Speedup
Position of found element
MCSTL find, 4 threads
MCSTL find, 3 threads
MCSTL find, 2 threads
sequential
naive parallel, 4 threads
naive parallel, 3 threads
naive parallel, 2 threads
0
1
2
3
4
1
10
100
1000
10000
10^5
10^6
10^7
10^8

partial_sum

Problem

For a sequence S, compute the prefix sums $A_i = \sum_{j=1}^{i} S_j$

Discrimination on Theoretical PRAM Algorithms

- each PE has its data
- shared-memory advantage: can split data arbitrarily
- assume that p is relatively small

partial_sum

Problem

For a sequence S, compute the prefix sums $A_i = \sum_{j=1}^{i} S_j$

Discrimination on Theoretical PRAM Algorithms

- each PE has its data
- shared-memory advantage: can split data arbitrarily
- assume that p is relatively small

Practical Algorithm for Shared Memory

- divide input into $p + 1$ pieces
- reason: double calculation for first part can be avoided

partial_sum: Algorithm

Processor $i \in 0 \ldots p-1$

1. $i = 0$: compute partial sums of part 0, $S[0] :=$ last one
   $i > 0$: compute $S[i] :=$ sum of part i
2. $i = 0$: compute partial sums of $S[i]$ sequentially
3. $i \geq 0$: compute partial sums of part $i + 1$ using $S[i]$

`partial_sum`: Algorithm

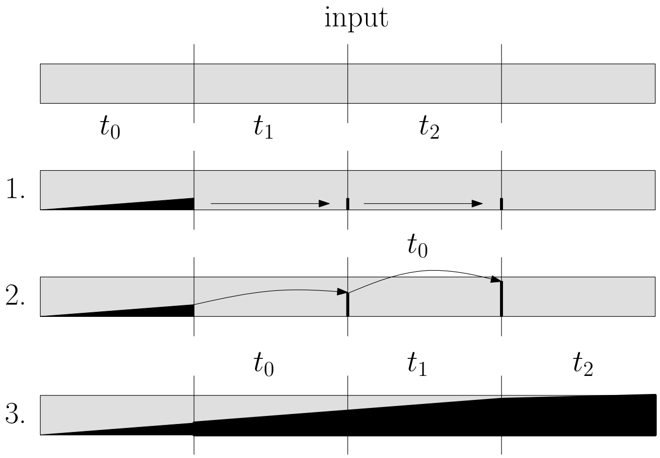
Processor $i \in 0 \ldots p-1$

1. $i = 0$: compute partial sums of part 0, $S[0] :=$ last one
   $i > 0$: compute $S[i] :=$ sum of part i
2. $i = 0$: compute partial sums of $S[i]$ sequentially
3. $i \geq 0$: compute partial sums of part $i+1$ using $S[i]$

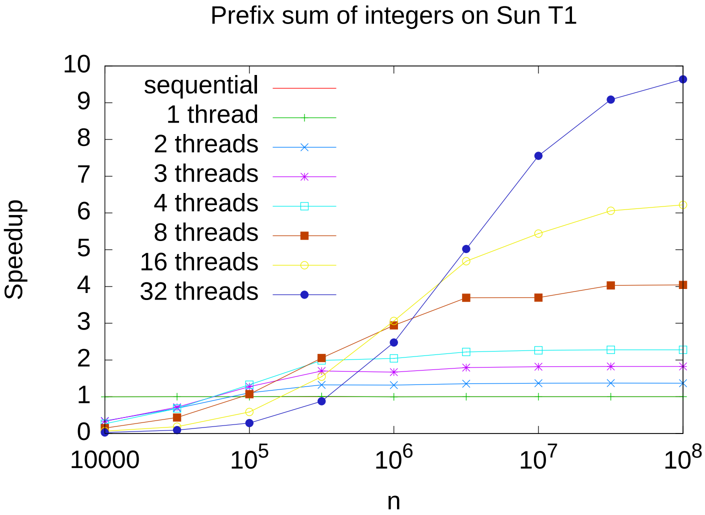
Analysis

- only 3 synchronizations (constant)
- time complexity $O(n/p + p)$
  speedup $\frac{p+1}{2}$ for $n \gg p$

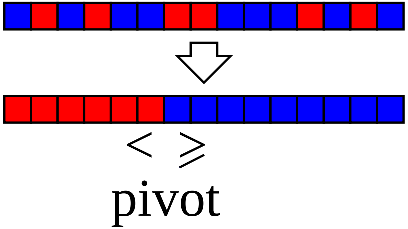
partial_sum: Scheme



partial_sum: Results



partition



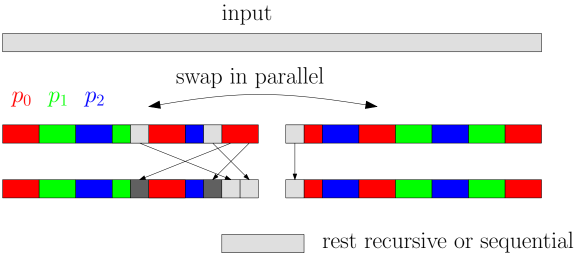
Sequential Algorithm

- scan from both ends
- swap to desired order when contrary

Parallel Partitioning

[Tsigas Zhang 2003]

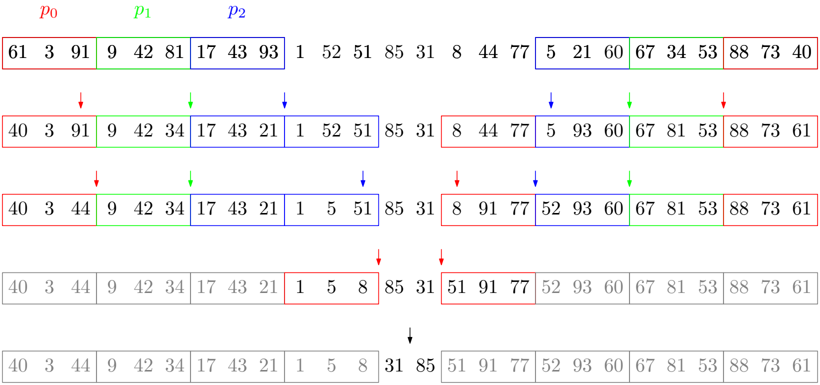
1. scan blocks of size B from both ends
   1.1 claim new blocks when running out of data
2. swap the unfinished blocks to the "middle"
3. recurse on the middle



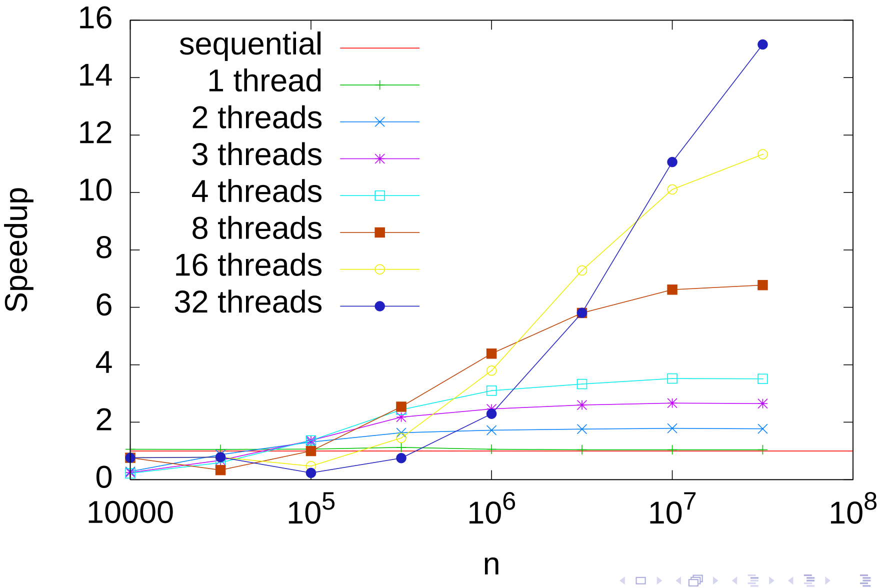
- time complexity $O(n/p + B \log p)$

partition: Example

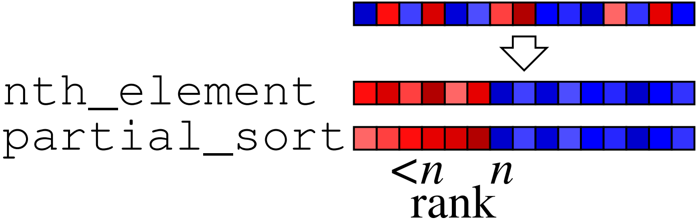
3 processors, B=3, pivot 50, no special cases



Partitioning of 32-bit integers on Sun T1



nth_element, partial_sort, quicksort



Algorithms

- nth_element: quickselect—
  linear recursion using partition
- partial_sort: nth_element then sort
- quicksort: recursion using partition

parallel implementations profit from each other

Multi-Sequence Selection

Problem Definition

find element with global rank r in k sorted sequences S_i and corresponding splitters.

Multi-Sequence Selection

Problem Definition

find element with global rank r in k sorted sequences S_i and corresponding splitters.

Usage

split at elements with global rank

$n/p \quad 2n/p \quad 3n/p \quad \ldots \quad (p-1)n/p$

and redistribute elements

$\rightsquigarrow$ sequences of the same length (± 1) on each PE

- guaranteed even for many equal elements

Multi-Sequence Selection

Problem Definition

find element with global rank r in k sorted sequences S_i and corresponding splitters.

Usage

split at elements with global rank

$n/p \quad 2n/p \quad 3n/p \quad \dots \quad (p-1)n/p$

and redistribute elements

$\leadsto$ sequences of the same length (± 1) on each PE

- guaranteed even for many equal elements

Solution

[Varman et al. 1991], used as black box here

Sequential `multiway_merge`

Problem Definition

merge k sorted sequences into one sorted sequence

Sequential `multiway_merge`

Problem Definition

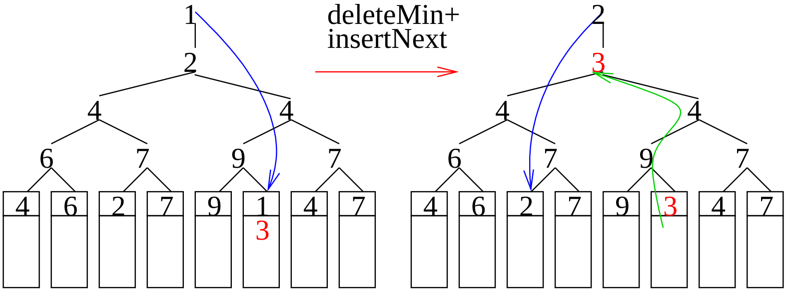
merge k sorted sequences into one sorted sequence

Solution

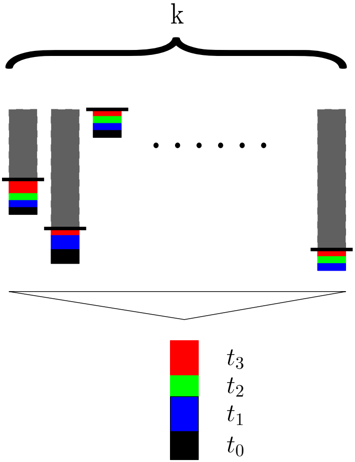
use a tournament tree, usually implemented as loser tree

- binary tree in array
- efficient computation of indices
- optimal $O(\log k)$ running time per merge step
- downside: tricky without sentinels and/or k not being a power of 2

Loser Tree


1
2
4 4
6 7 9 7
4 6 2 7 9 1 4 7
3
deleteMin+
insertNext
2
3
4 4
6 7 9 7
4 6 2 7 9 3 4 7

Parallel (multiway_)merge



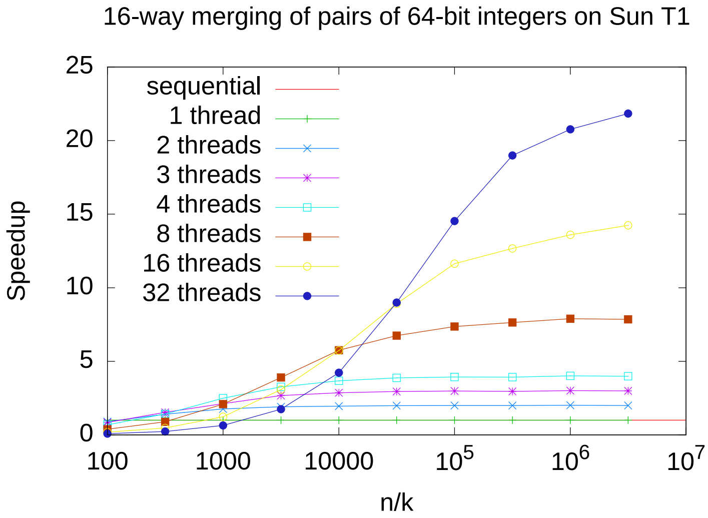
How to divide the problem?

- find slabs, i. e. consistent sets of sections from the sequences
- exact splitting into parts of equal size (using *multi-sequence selection*)

Parallel (multiway_)merge: Analysis

- time complexity $O(\frac{n}{p} \log k \quad + k \log k \cdot \log \max_j |S_j|)$
- one multi-sequence partition per PE
- no full linear speedup
- good in practice

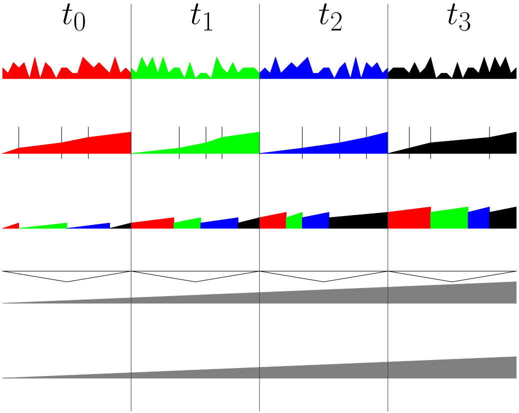
Parallel (multiway_)merge: Results



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

- \+ few, cache-efficient local memory accesses
- \+ stable variant easy
- – needs twice the space temporarily

sort, stable_sort

Parallel Multiway Mergesort

+ few, cache-efficient local memory accesses
+ stable variant easy
– needs twice the space temporarily

Quicksort

+ in-place
± dynamic load-balancing due to unequal splitting
– more global memory access
– not stable

both variants implemented in the MCSTL

Parallel Multiway Mergesort: Analysis

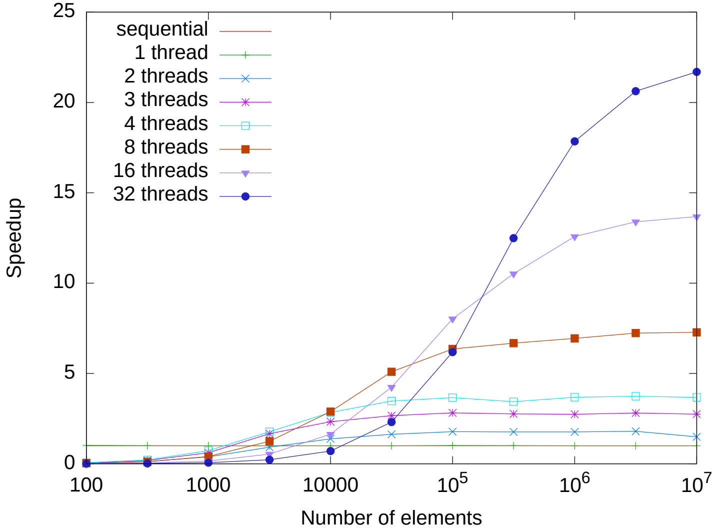
Running Time

- time complexity $O(\frac{n \log n}{p} + p \log p \cdot \log \frac{n}{p})$
- one multi-sequence partition per PE

Parallel Multiway Mergesort: Practical Issues

- copy to temporary memory first? or merge to temporary memory and copy back later?
- compute starting positions sequentially

Parallel Multiway Mergesort: Results on T1



Random Permutation (`random_shuffle`)

Standard Sequential Algorithm (e. g. STL)

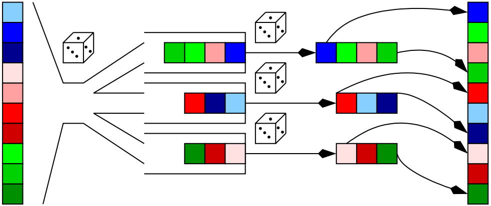
for $0 \leq i < n$ swap $(a[i], a[\text{rand}(i+1, n-1)])$

Cache-efficient (parallel) algorithm

1. distribute randomly to (local) buckets

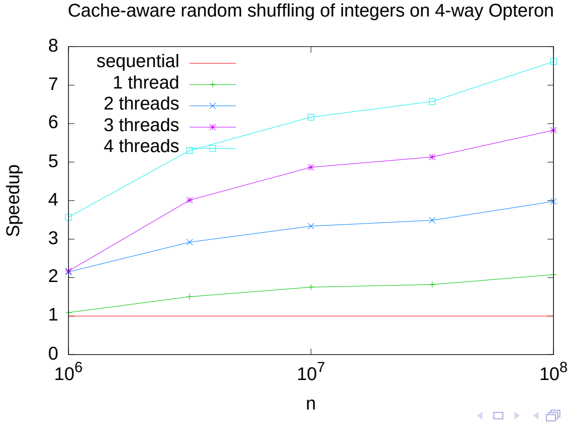
1b. (copy local buckets to global buckets)

2. permute buckets



Random Permutation (random_shuffle)

- time complexity $O(\frac{n}{p} + p)$, global communication volume n
- cache efficiency very important (factor 2)



Embarrassingly Parallel Computation

- semantics
  - process a set of elements completely independently
  - atomic units called jobs, running time unknown
- parallelization
  - easy in principle (uniform workload)
    $\leadsto$ static load-balancing
  - interesting for non-uniform workload
    $\leadsto$ dynamic load-balancing
- possible solutions
  - equal splitting: perfect for uniform workload
  - master-worker: possibly considerable overhead (communication in each step)
  - dynamic load-balancing:
    more general problem, see upcoming lectures

Outline

Conclusion

- MCSTL provides a very easy way to incorporate parallelism into programs on an algorithmic level
- performance is excellent for large inputs
- basic algorithms known but detailed design and performance engineering nontrivial
- successfully integrated into STXXL (external memory)
- integrated into GCC 4.3 (parallel mode)

Code Sample

```
#include <algorithm>
int a[10000000];
int main() {
 std::sort(a, a+10000000);
}
```

```
g++-4.3.0 -D_GLIBCXX_PARALLEL -fopenmp
sort.cpp
```

Future Work

- complete STL functionality
- better automatic algorithm and parameter selection
- machine model adequate for design and analysis of multithreaded algorithms
- beyond STL

Algorithms & DS to be Implemented

- priority queues
- some embarrassingly parallel functions (e. g. `valarray`)
- memory transfer operations (`reverse`, `copy`)?

More About All That

- MCSTL website:

  http://algo2.iti.uni-karlsruhe.de/singler/mcstl/
- libstdc++ parallel mode:

  http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html