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

December 13th, 2006

Lecture Contents

Outline

What is this Lecture About?

Theory $\rightsquigarrow$ Practice

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

What is this Lecture About?

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

What is this Lecture About?

Theory $\rightsquigarrow$ Practice

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

Communication Network $\rightsquigarrow$ Shared Memory

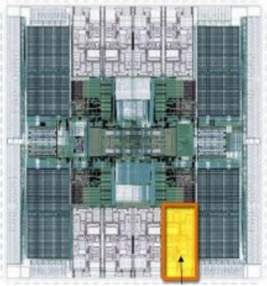
- implicit communication
  - cache hierarchy, NUMA, bandwidth sharing

Synchronous PRAM $\rightsquigarrow$ Asynchronous PEs

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

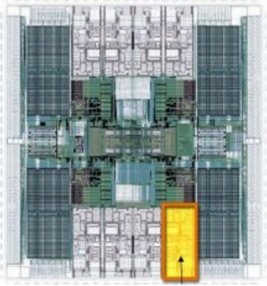
Why Multi-Cores?

- easy use of high transistor budget
- energy efficient
  (at reduced clock speeds)
- increase in clock speed
  largely exhausted
- instruction level parallelism
  exhausted
- SIMD/Vector
  only for special applications



Why Multi-Cores?

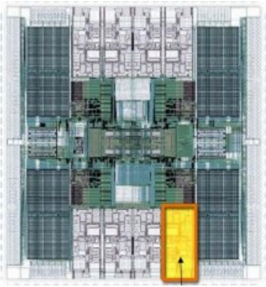
- easy use of high transistor budget
- energy efficient
  (at reduced clock speeds)
- increase in clock speed
  largely exhausted
- instruction level parallelism
  exhausted
- SIMD/Vector
  only for special applications



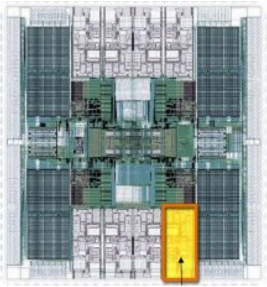
⇝ Multi-cores will be everywhere:
mobile devices . . . super computers

Hardware Nowadays

- dual-cores omnipresent
- mainstream quad-core available
- Sun T1: 8 cores, 32 threads
- high-end shared-memory servers with many more cores (on multiple chips)

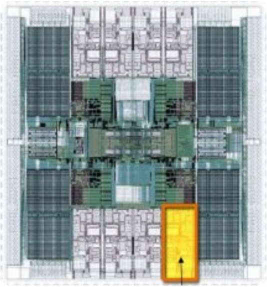


Programming Multicores



- 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
- simple 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
- dynamically choose
  algorithms and tuning parameters
- coexist with other forms of parallelization
  $\rightsquigarrow$ load balancing even for regular computations

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

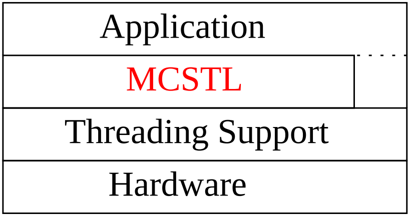
Generality

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

Compatibility to

- existing libraries
- platforms

Layers



Threading Support

- OpenMP: currently used (basic primitives).
  - example

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

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

Threading Support

- OpenMP: currently used (basic primitives).
  - example

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

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

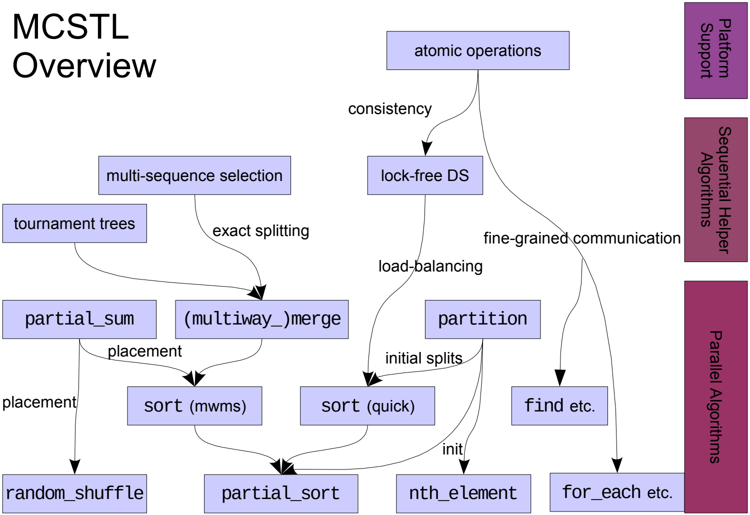
Implemented Algorithms

- `find`, `find_if`, `mismatch`, ...
- `partial_sum` (prefix sum)
- `partition`
- `nth_element`/`partial_sort`
- `merge`
- `sort`, `stable_sort`
- `random_shuffle`
- embarrassingly parallel (`for_each`, `transform`,...)
  $\geq$ 50 % of STL

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!

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!
- "cache level 0" = registers exempted, variable values not updated in memory (from other core's point of view)
  - declare variable type `volatile` (once per variable)
  - `#pragma omp flush` variable when update suspected (once per update)

Atomic Operations

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

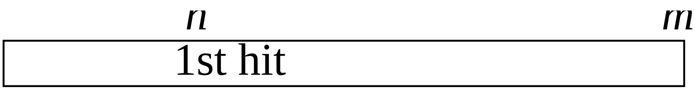
Outline

`find`, `find_if`, `mismatch`, ...

find the first position in a sequence satisfying a predicate

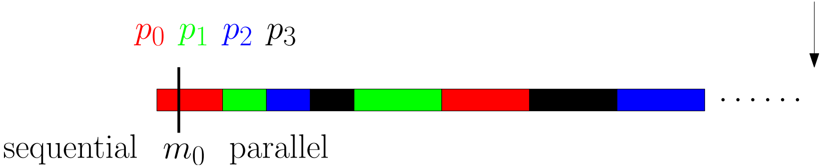
Analysis

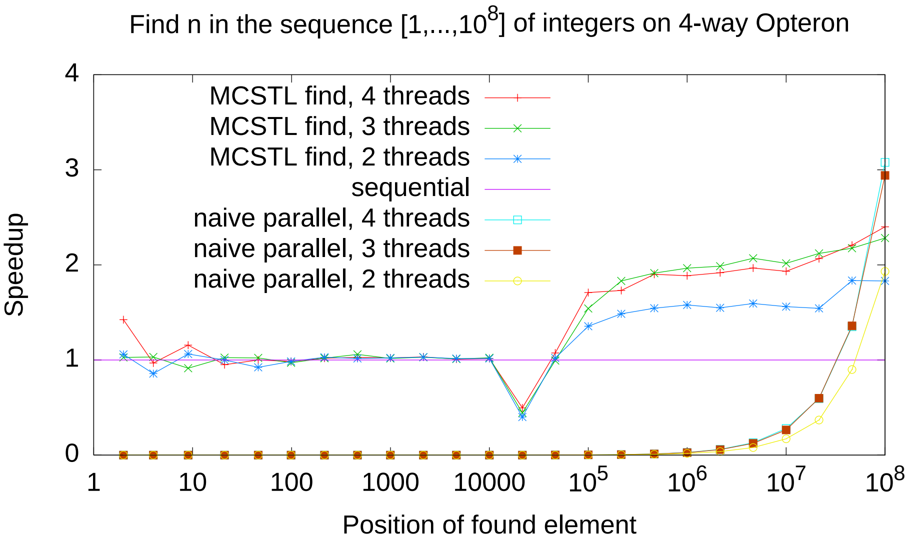
- $O(n)$ sequential time if first hit is at position n (unknown)
- naïve parallel algorithm needs $\Omega(m/p)$.
- parallelization not worthwhile for small n



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
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
Speedup
0
1
2
3
4
1
10
100
1000
10000
10^5
10^6
10^7
10^8
Position of found element

partial_sum

Discrimination to Algorithms Seen so Far

- $n \gg p$: multiple elements per PE,
  sum must be calculated in preprocessing step,
  prefix sum in postprocessing step
- $\rightsquigarrow 2n + O(1)$ additions in total, not optimal,
  speedup only $\frac{p}{2}$, particularly bad for small p
- $O(\log p)$ communication steps
- shared-memory advantage: can split data arbitrarily

partial_sum

Discrimination to Algorithms Seen so Far

- $n \gg p$: multiple elements per PE,
  sum must be calculated in preprocessing step,
  prefix sum in postprocessing step
- $\rightsquigarrow 2n + O(1)$ additions in total, not optimal,
  speedup only $\frac{p}{2}$, particularly bad for small p
- $O(\log p)$ communication steps
- shared-memory advantage: can split data arbitrarily

Practical Algorithm for Shared Memory

- divide input into $p + 1$ pieces
- 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

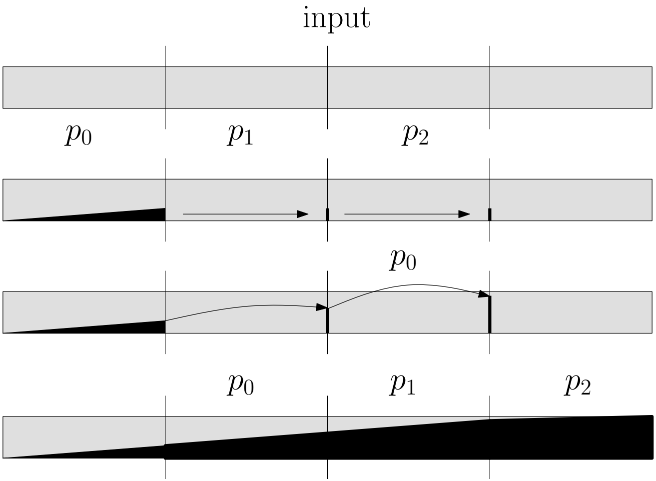
Processor $i \in 0 \ldots p-1$

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 $S[i]$

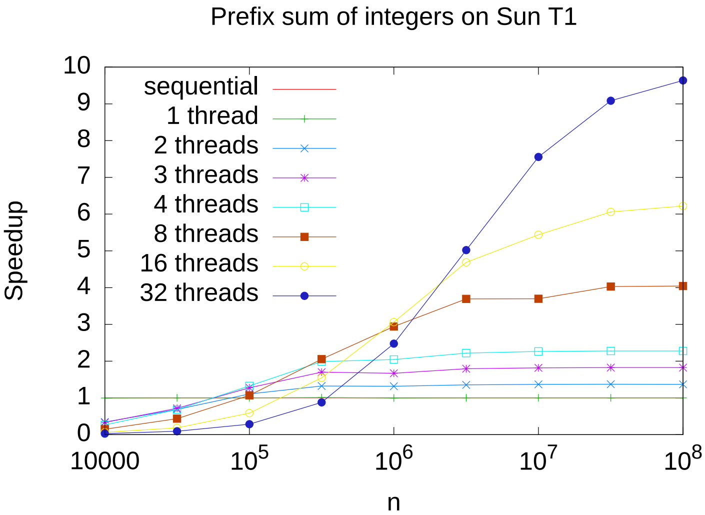
Analysis

- only 3 synchronizations (constant)
- time complexity $O(n/p + p)$, no hidden factor 2 $\rightsquigarrow$ speedup $\frac{p+1}{2}$ for $n \gg p$

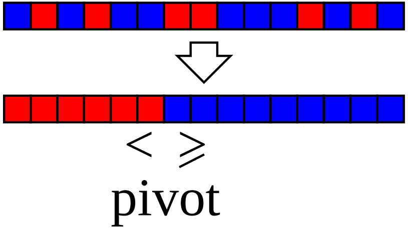
partial_sum: Scheme



partial_sum: Results



partition



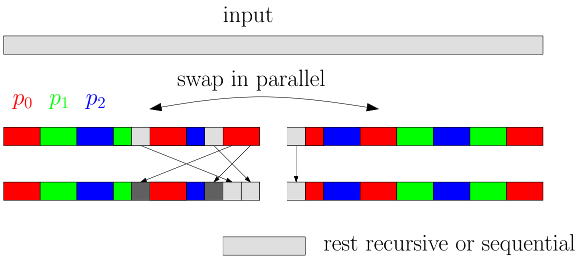
Sequential Algorithm

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

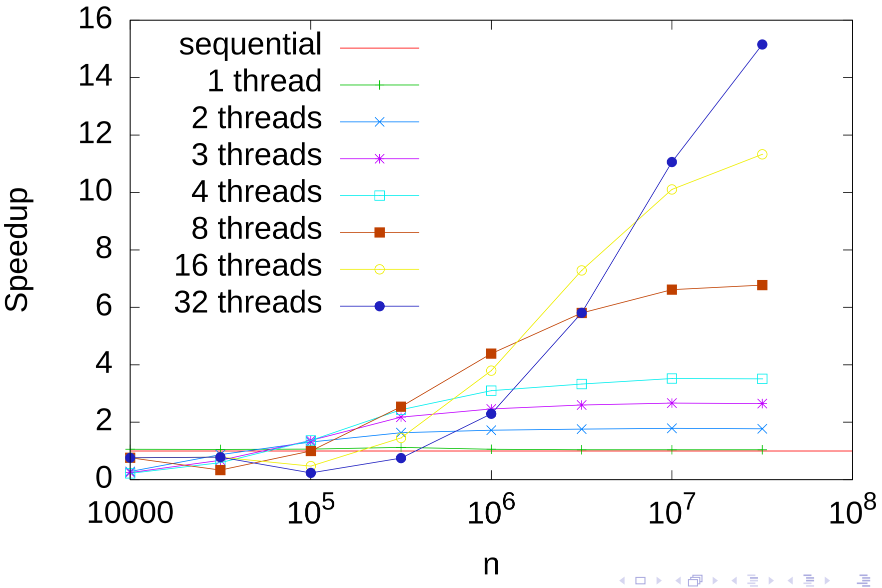
`partition`: Example

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

p_0 p_1 p_2

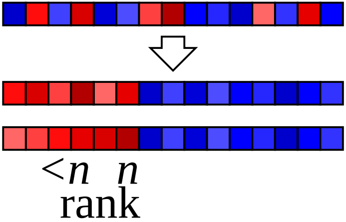
61	3	91	9	42	81	17	43	93	1	52	51	85	31	8	44	77	5	21	60	67	34	53	88	73	40
40	3	91	9	42	34	17	43	21	1	52	51	85	31	8	44	77	5	93	60	67	81	53	88	73	61
40	3	44	9	42	34	17	43	21	1	5	51	85	31	8	91	77	52	93	60	67	81	53	88	73	61
40	3	44	9	42	34	17	43	21	1	5	8	85	31	51	91	77	52	93	60	67	81	53	88	73	61
40	3	44	9	42	34	17	43	21	1	5	8	31	85	51	91	77	52	93	60	67	81	53	88	73	61

Partitioning of 32-bit integers on Sun T1



`nth_element`, `partial_sort`, quicksort

`nth_element`
`partial_sort`



Algorithms

- `nth_element`: quickselect—
  linear recursion using `partition`
- `partial_sort`: `nth_element` then `sort`
- quicksort: recursion using `partition`,
  load balancing using work stealing

Parallel implementations profit from each other

Multi-Sequence Selection

Problem Definition

find element with global rank r in k sorted sequences S_i

Multi-Sequence Selection

Problem Definition

find element with global rank r in k sorted sequences S_i

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
$\rightsquigarrow$ 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

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

Solution

[Varman et al. 1991] see next slide

Multi-Sequence Selection: Algorithm

Idea

- partition into two sets with desired ratio (corresponds to rank)
- start with middle element
- refine partition by recursively adding the elements in the middle of both sides, taking $O(k)$ time for each step only
- running time $O(k \log |S_i|)$
  $O(k \log k \log |S_i|)$ practical variant

Multi-Sequence Selection: Example

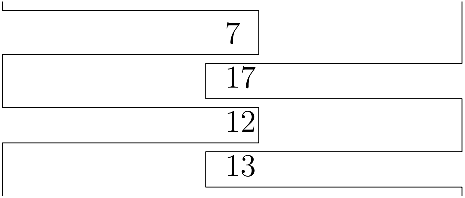
$k = 4$, $N = k \cdot n = 4 \cdot 7 = 28$; select global rank 14

1	2	6	7	9	11	15
2	8	9	17	23	24	25
6	7	9	12	23	24	25
3	8	10	13	14	17	19

Multi-Sequence Selection: Example

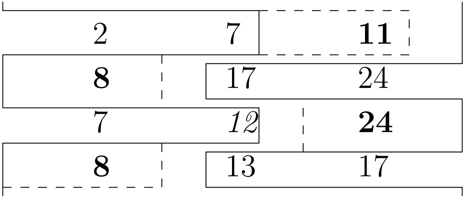
$k = 4$, $N = k \cdot n = 4 \cdot 7 = 28$; select global rank 14

1	2	6	7	9	11	15
2	8	9	17	23	24	25
6	7	9	12	23	24	25
3	8	10	13	14	17	19



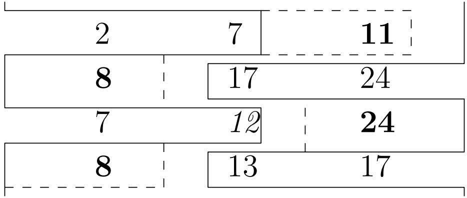
Multi-Sequence Selection: Example

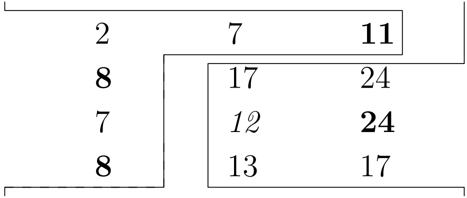
$k = 4$, $N = k \cdot n = 4 \cdot 7 = 28$; select global rank 14



Multi-Sequence Selection: Example

$k = 4$, $N = k \cdot n = 4 \cdot 7 = 28$; select global rank 14





Multi-Sequence Selection: Remarks

Implementation Problems

- non-uniform length, length not equal to $2^i - 1$: "conceptual padding" $\leadsto$ running time $\sim \log \max_i |S_i|$
- finding ranks $\neq \frac{1}{2} \sum_i |S_i|$, short sequences: complicated special cases at ends of sequences
- equal elements: find partition directly, not element with specified global rank

Sequential `multiway_merge`

Problem Definition

merge k sorted sequences into one sorted sequence

Sequential `multiway_merge`

Problem Definition

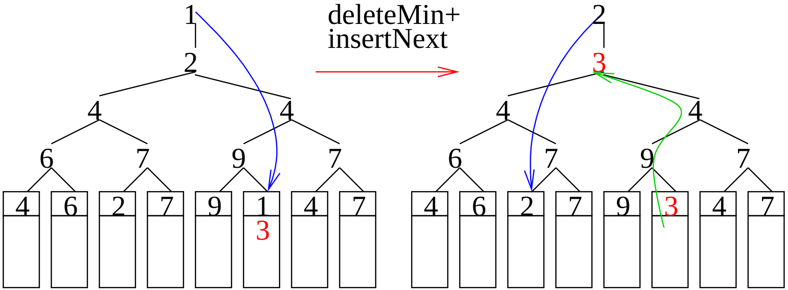
merge k sorted sequences into one sorted sequence

Solution

use a tournament tree, usually implemented as loser tree

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

Loser Tree

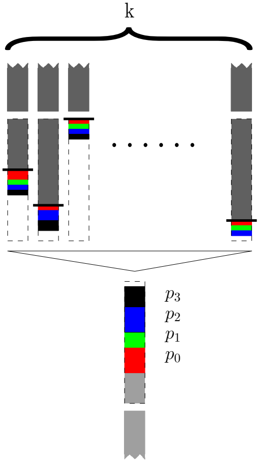


1
2
4 4
6 7 9 7
4 6 2 7 9 1 4 7
3
deleteMin+
insertNext
2
3
4 4
6 7 9 7
4 6 2 7 9 3 4 7

Parallel (multiway_)merge

How to divide the problem?

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

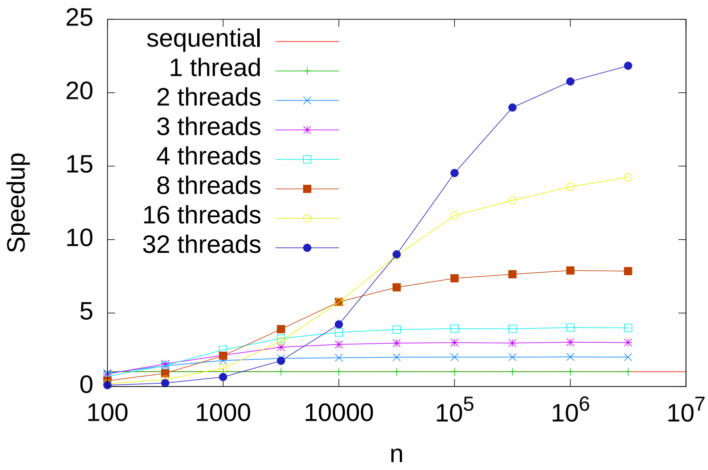


Parallel `(multiway_)merge`: Analysis

- time complexity $O(\frac{1}{p}(n \log k \quad + k \log k \cdot \log \max_j |S_j|))$
- no full linear speedup
- good in practice
- special case $k = p$: $O(\frac{n}{p} \log k + \log p \cdot \log \max_j |S_j|))$

Parallel (multiway_)merge: Results

Multiway merging of pairs of 64-bit integers on Sun T1



sort, stable_sort

Parallel Multiway Mergesort

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

sort, stable_sort

Parallel Multiway Mergesort

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

Quicksort

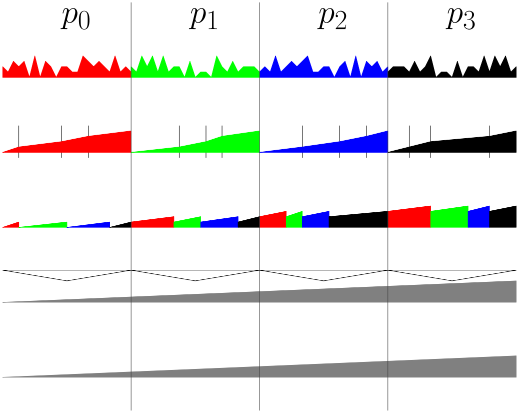
- \+ in-place
- ± dynamic load-balancing due to unequal splitting
- – more global memory access
- – not stable

both variants implemented in the MCSTL

Parallel Multiway Mergesort

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



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: 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

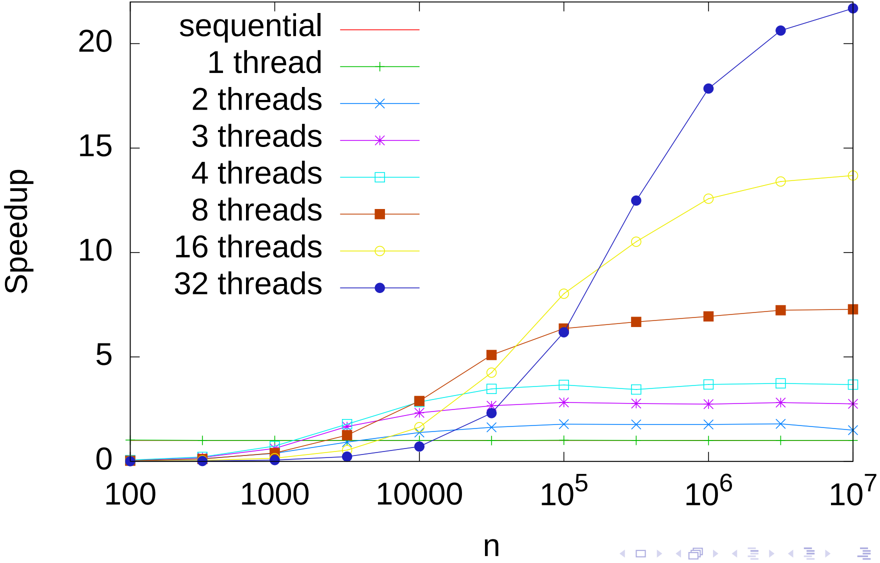
Comparison to (Deterministic) Sample Sort

- very similar, only splitting differs
- exact splitting $\Longleftrightarrow$ approximation guaranteed
- DSS' time complexity: $O(\frac{n \log n}{p} + p \log p)$
- tradeoff possible using oversampling
- global communication volume: $2n$ (copy back)
- local memory movement: $\frac{n}{p} log_2 \frac{n}{p}$

Parallel Multiway Mergesort: Practical Issues

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

Multiway Mergesort of 64-bit integers on Sun T1



Parallel Quicksort

Basic Algorithm

1. `partition` the sequence in parallel

Parallel Quicksort

Basic Algorithm

1. `partition` the sequence in parallel
2. if group consists of more than one processor:
    2.1 divide group according to data balance
    2.2 continue with 1. recursively

Parallel Quicksort

Basic Algorithm

1. `partition` the sequence in parallel
2. if group consists of more than one processor:
   2.1 divide group according to data balance
   2.2 continue with 1. recursively
3. otherwise: sort the piece sequentially

Problem

load balancing may be very poor, in particular with small *p*, bad splitters

Solution

keep basic algorithm,
dynamically balance work in last step

Parallel Load-Balanced Quicksort

1. `partition` the sequence in parallel

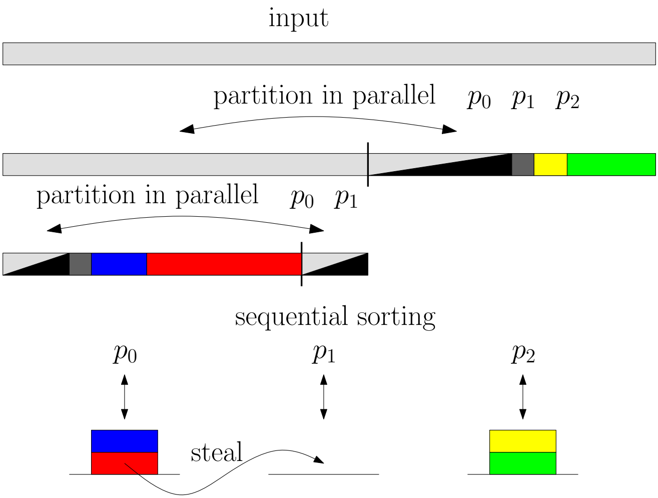
Parallel Load-Balanced Quicksort

1. `partition` the sequence in parallel
2. if group consists of more than one processor:
   2.1 divide group according to data balance
   2.2 continue with 1. recursively

Parallel Load-Balanced Quicksort

1. `partition` the sequence in parallel
2. if group consists of more than one processor:
   2.1 divide group according to data balance
   2.2 continue with 1. recursively
3. otherwise: quicksort the piece sequentially
   push the piece onto a local stack
   while unsorted elements exist
   3.1 if non-empty: pop a piece from local stack
   3.2 otherwise: take (large) piece from *bottom* of other PE's stack (work-stealing)
   3.3 partition piece
   3.4 push right part onto stack, sort left part recursively

Parallel Load-Balanced Quicksort: Scheme



Parallel Load-Balanced Quicksort: Practice

- omit stack operations for small parts
- use lock-free stack data structure
  - every thread makes progress in every step
  - no mutexes or semaphores are used
  - many lock-free data-structures known, many use linked lists
  - simple one used here
- how to detect termination?
- erratic performance if more threads than processors: why?

Lock Free (Restricted) Double-Ended Queue

Requirements

- `push_front`, `pop_front` not concurrently, issued only by one specific thread
- `pop_back` concurrently from all other threads
- number of elements is limited (logarithmic)
- no `is_empty`, no `top`, because semantics unclear
- `pop_*` may fail

Lock Free (Restricted) Double-Ended Queue

Requirements

- `push_front`, `pop_front` not concurrently, issued only by one specific thread
- `pop_back` concurrently from all other threads
- number of elements is limited (logarithmic)
- no `is_empty`, no `top`, because semantics unclear
- `pop_*` may fail

Solution

- circular buffer with front and back pointer
- encode front and back pointer into one word to allow synchronous atomic update using compare-and-swap

Lock Free (Restricted) Double-Ended Queue

Code for pop_back

```
before := pointers
while(before.front > before.back)
{
  after := (before.front    , before.back + 1)



  if(cas(pointers, before, after))
  {
    item := *(before.back)
    return true
  }
}
return false
```

Lock Free (Restricted) Double-Ended Queue

Code for pop_front

```
before := pointers
while(before.front > before.back)
{


  after := (before.front - 1, before.back     )

  if(cas(pointers, before, after))
  {
    item := *(before.back)
    return true
  }
}
return false
```

Lock Free (Restricted) Double-Ended Queue

Code for pop_front

```
before := pointers
while(before.front > before.back)
{


  after := (before.front - 1, before.back     )

  if(cas(pointers, before, after))
  {
    item := *(before.back)
    return true
  }
}
return false
```

Code for push_front

```
*(pointers.front) := item
fetch_and_add(pointers.front, 1)
```

Lock Free (Restricted) Double-Ended Queue

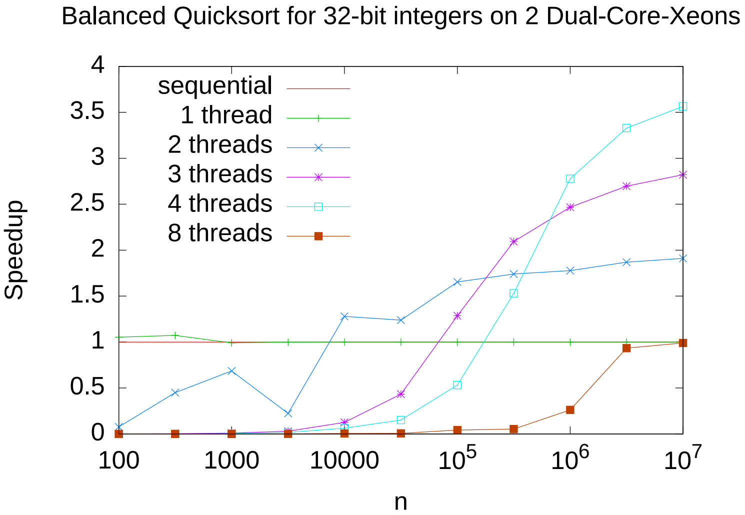
Properties

- lock-free, but not wait-free
- pointer back increases monotonically
  $\rightsquigarrow$ no concurrency problems at queue back
- pointer front does not increase monotonically
  $\rightsquigarrow$ no problem, since no concurrent `push` *and* `pop` allowed at queue front
- in case of failure: retry or done

Balanced Quicksort: Analysis

- time complexity $O(\frac{n \log n}{p} + B \log p)$
- communication volume + local memory movement: $n \log_2 n$
- good speedups require fast random-access across PE boundaries

Balanced Quicksort: Results



Balanced Quicksort: Problem Analysis

Problem

- not so nice performance
- particularly bad with too little processors
- where is the problem?
- processor fully loaded while stealing when there is no piece available

Balanced Quicksort: Problem Analysis

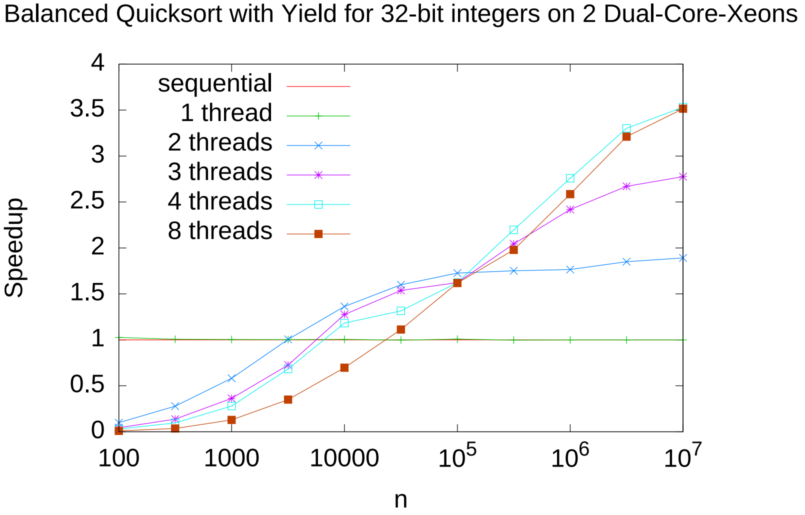
Problem

- not so nice performance
- particularly bad with too little processors
- where is the problem?
- processor fully loaded while stealing when there is no piece available

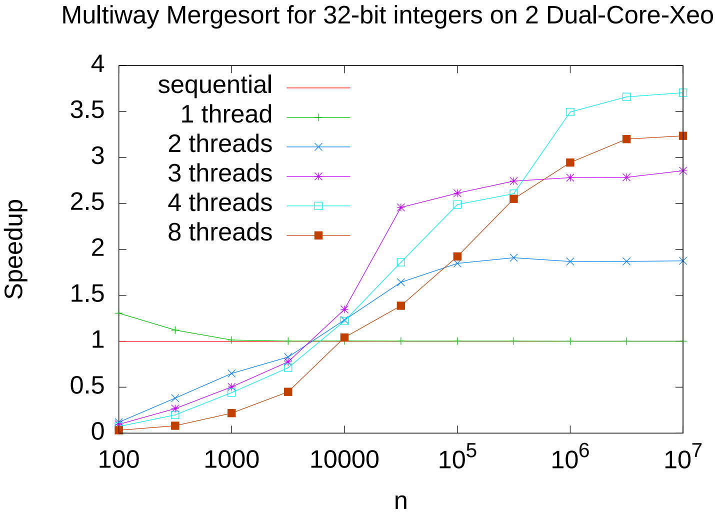
Solution

- switch to other processor if no work found $\Rightarrow$ `yield`

Balanced Quicksort: Results with `yield`



Balanced Quicksort: Comparison to PMWMS



Random Permutation (`random_shuffle`)

Standard Sequential Algorithm (e. g. STL)

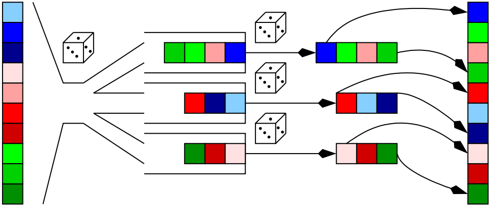
for $0 \leq i < n$ swap $(a[i], a[\text{rand}(i + 1, n - 1)])$

Cache efficient (parallel) algorithm

1. distribute randomly to (local) buckets

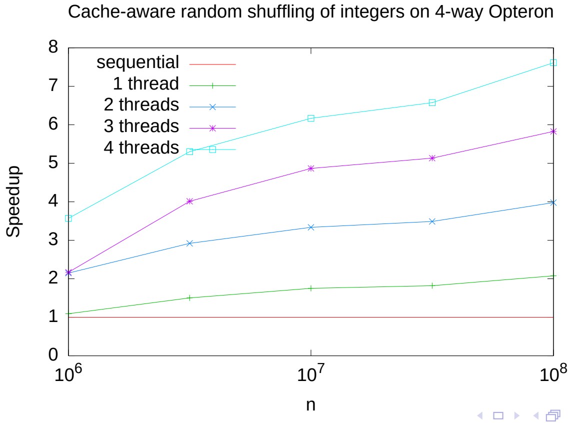
1b. (copy local buckets to global buckets)

2. permute buckets



Random Permutation (`random_shuffle`)

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

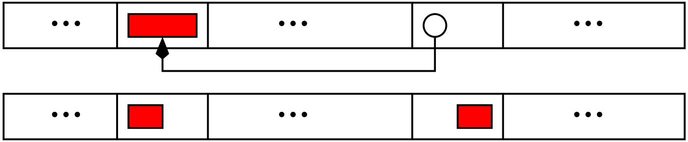


Embarrassingly Parallel Computation

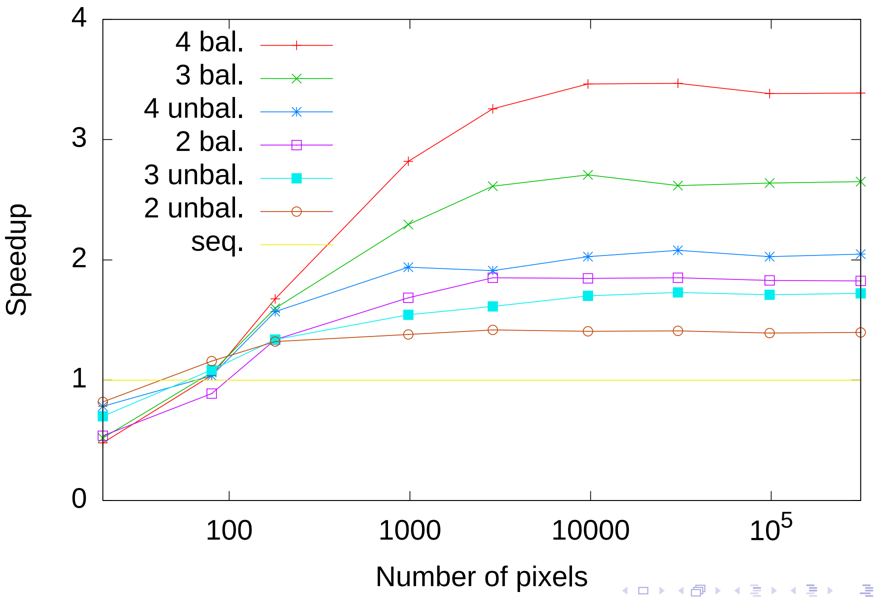
- semantics
  - process a set of elements completely independently
  - atomic units called jobs, running time unknown
- parallelization
  - easy in principle (uniform workload)
    $\rightsquigarrow$ static load-balancing
  - interesting for non-uniform workload
    $\rightsquigarrow$ dynamic load-balancing
- possible solutions
  - equal splitting: perfect for uniform workload
  - master-worker: possibly considerable overhead (communication in each step)
  - work-stealing: communication only when necessary

Dynamic Load Balancing for `for_each` etc.

- using work-stealing
- divide iteration range into equal intervals initially
- idle threads steal half the interval from random victim
  - no explicit synchronization with victims needed (using `fetch_and_add`)
  - adaptive granularity control (cache!)
  - logarithmic number of steals suffice with high probability



Mandelbrot on 4-way Opteron, at most 1000 iterations per pixel



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
- successful integration into STXXL (external memory)

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

- containers: initialization, bulk operations
- priority queues
- some embarrassingly parallel functions (e. g. `valarray`)
- memory transfer operations (`reverse`, `copy`)?
- set operations (`set_union`,...)

More About All That

- MCSTL website:

  http://algo2.iti.uni-karlsruhe.de/singler/mcstl/
- Praktikum next semester:
  extension/usage of MCSTL
- Studien-/Diplomarbeiten