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

Introduction

Platform Support

Algorithms

Conclusion
Outline

Introduction

Platform Support

Algorithms

Conclusion
Model

Theory $\leadsto$ Practice

- machine model $\leadsto$ concrete machine(s)
- pseudo-code $\leadsto$ 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 $\leadsto$ Practice

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

Communication Network $\leadsto$ Shared Memory

- implicit communication
  - cache hierarchy, NUMA, bandwidth sharing

Synchronous PRAM $\leadsto$ Asynchronous PEs

- synchronization a problem itself
- $n = p \leadsto n \gg p$
- 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
- reccompilation of existing programs may suffice
- C++ accepted and efficient language
Goals

- parallelize all time consuming STL algorithms
- speedup already for small inputs $\leadsto$ 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

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

MCSTL - Practical Parallelism
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`
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

```c
#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 \(\leadsto\) atomically

- \texttt{fetch\_and\_add(x, i)}
  - \(t := x;\ r := x;\ r := r + i;\ x := r;\)
  - return \(t;\)
  - allows concurrent \textit{iteration} over sequence
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

- `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

Introduction

Platform Support

Algorithms

Conclusion
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**

\[ p_0 \quad p_1 \quad p_2 \quad p_3 \]

sequential \quad $m_0$ \quad parallel
Find $n$ in the sequence $[1, \ldots, 10^8]$ of integers on 4-way Opteron

![Graph showing speedup and position of found element for different parallel algorithms on 4-way Opteron.](image-url)
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
**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

1. 

2. 

3. 

input
partial_sum: Results

Prefix sum of integers on Sun T1

Speedup of partial sum algorithm for different thread counts.
**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

![Diagram showing partitioning process]

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

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

<table>
<thead>
<tr>
<th></th>
<th>$p_0$</th>
<th>$p_1$</th>
<th>$p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>61 3 91</td>
<td>9 42 81</td>
<td>17 43 93</td>
</tr>
<tr>
<td></td>
<td>40 3 91</td>
<td>9 42 34</td>
<td>17 43 21</td>
</tr>
<tr>
<td></td>
<td>40 3 44</td>
<td>9 42 34</td>
<td>17 43 21</td>
</tr>
<tr>
<td></td>
<td>40 3 44</td>
<td>9 42 34</td>
<td>17 43 21</td>
</tr>
<tr>
<td></td>
<td>40 3 44</td>
<td>9 42 34</td>
<td>17 43 21</td>
</tr>
</tbody>
</table>

$\downarrow$ $\downarrow$ $\downarrow$

|   | 8 52 51 85 31 | 44 77 5 21 60 67 34 53 88 73 40 |
|   | 8 91 77 52 93 60 67 81 53 88 73 61 |
|   | 51 91 77 52 93 60 67 81 53 88 73 61 |
|   | 51 91 77 52 93 60 67 81 53 88 73 61 |
|   | 51 91 77 52 93 60 67 81 53 88 73 61 |

$\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$
Partitioning of 32-bit integers on Sun T1

![Graph showing speedup of partitioning with increasing number of threads.]

Peter Sanders, Johannes Singler

MCSTL - Practical Parallelism
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 $n/p$ $2n/p$ $3n/p$ $\ldots$ $(p - 1)n/p$ and redistribute elements
$\leadsto$ 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 $n/p$ \(2n/p\) \(3n/p\) \(\ldots\) \((p-1)n/p\) and redistribute elements
\(\mapsto\) sequences of the same length \((\pm1)\) on each PE

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

Insert Next

deleteMin+

Peter Sanders, Johannes Singler  MCSTL - Practical Parallelism
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 $O\left(\frac{n}{p} \log k + k \log k \cdot \log \max_j |S_j| \right)$
- one multi-sequence partition per PE
- no full linear speedup
- good in practice
Parallel (multiway) merge: Results

16-way merging of pairs of 64-bit integers on Sun T1

Speedup vs. \( n/k \)

- Sequential
- 1 thread
- 2 threads
- 3 threads
- 4 threads
- 8 threads
- 16 threads
- 32 threads

Peter Sanders, Johannes Singler
MCSTL - Practical Parallelism
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 \(O\left(\frac{n \log n}{p} + p \log p \cdot \log \frac{n}{p}\right)\)
- 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

![Graph showing speedup vs. number of elements for parallel multiway mergesort with different thread counts.](image-url)
Random Permutation (**random_shuffle**)

Standard Sequential Algorithm (e.g. STL)

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

Cache-efficient (parallel) algorithm

1. distribute randomly to (local) buckets
2. permute buckets
Random Permutation (random\_shuffle)

- time complexity $O(n/p + p)$,
  global communication volume $n$

- cache efficiency very important (factor 2)

Cache-aware random shuffling of integers on 4-way Opteron sequential

<table>
<thead>
<tr>
<th>Threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Peter Sanders, Johannes Singler
MCSTL - Practical Parallelism
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

Introduction

Platform Support

Algorithms

Conclusion
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).
#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:**
  