Design and Analysis of Parallel Programs

TDDD93 Lecture 3-4

Christoph Kessler
PELAB / IDA
Linköping university
Sweden

2017

Outline

Design and analysis of parallel algorithms
- Foster’s PCAM method for the design of parallel programs
- Parallel cost models
- Parallel work, time, cost
- Parallel speed-up; speed-up anomalies
- Amdahl’s Law
- Fundamental parallel algorithms: Parallel prefix, List ranking

- TDDD56: Parallel Sorting Algorithms
- TDDC78: Parallel Linear Algebra and Linear System Solving

Foster’s Method for Design of Parallel Programs ("PCAM")

Parallel Cost Models
A Quantitative Basis for the Design of Parallel Algorithms

Parallel Computation Model = Programming Model + Cost Model

+ abstract from hardware and technology
+ specify basic operations, when applicable
+ specify how data can be stored

→ analyze algorithms before implementation independent of a particular parallel computer
→ focus on most characteristic (w.r.t. influence on exec. time) features of a broader class of parallel machines

Programming model
- shared memory / message passing, degree of synchronous execution

Cost model
- key parameters
- cost functions for basic operations
- constraints

Parallel Computation Models

Shared-Memory Models
- PRAM (Parallel Random Access Machine) [Fortune, Wyllie ‘78]
  including variants such as Asynchronous PRAM, QRQW PRAM
- Data-parallel computing
- Task Graphs (Circuit model; Delay model)
- Functional parallel programming
  ...

Message-Passing Models
- BSP (Bulk-Synchronous Parallel) Computing [Valiant’90]
  including variants such as Multi-BSP [Valiant’08]
- MPI (programming model)
  + Delay-model or LogP-model (cost model)
- Synchronous reactive (event-based) programming e.g. Erlang
- Dataflow programming
  ...
Flashback to DALG, Lecture 1: The RAM (von Neumann) model for sequential computing

RAM (Random Access Machine)

Programming and cost model for the analysis of sequential algorithms

- Arithmetic (add, mul, ...) on registers
- Load
- Store
- Branch

Simplifying assumptions for time analysis:
- All of these take 1 time unit
- Serial composition adds time costs
  \[ T(op1, op2) = T(op1) + T(op2) \]

Cost Model

Cost model: should
- explain available observations
- predict future behaviour
- abstract from unimportant details → generalization

Simplifications to reduce model complexity:
- use idealized multitask model
  ignore hardware details: memory hierarchies, network topology, ...
- use scale analysis
  drop insignificant effects
- use empirical studies
  calibrate simple models with empirical data
  rather than developing more complex models

Analysis of sequential algorithms: RAM model (Random Access Machine)

Algorithm analysis: Counting instructions

Example: Computing the global sum of \( N \) elements

\[
T = t_{load} + t_{mem} + \sum_{i=1}^{N} (t_{load} + t_{add} + t_{mem}) = 5N + 3 \Theta(N)
\]

Data flow graph, showing dependencies (precedence constraints) between operations

→ arithmetic circuit model, directed acyclic graph (DAG) model

Remark

PRAM model is very idealized, extremely simplifying / abstracting from real parallel architectures:

- unbounded number of processors:
  abstracts from scheduling overhead
- local operations cost 1 unit of time
- every processor has unit time memory access
  to any shared memory location:
  abstracts from communication time, bandwidth limitation, memory latency, memory hierarchy, and locality
- focus on pure, fine-grained parallelism

→ Good for rapid prototyping of parallel algorithm designs:
  A parallel algorithm that does not scale under the PRAM model does not scale well anywhere else!

The PRAM Model – a Parallel RAM

Parallel Random Access Machine [Fortune/Wylie ’78]

- \( p \) processors
- MIMD
- common clock signal
- arith./jump: 1 clock cycle

Shared memory
- uniform memory access time
- latency: 1 clock cycle (!)
- concurrent memory accesses
- sequential consistency

Private memory (optional)
- processor-local access only

PRAM Variants

- Exclusive Read, Exclusive Write (EREW) PRAM:
  concurrent access only to different locations in the same cycle
- Concurrent Read, Exclusive Write (CREW) PRAM
  simultaneous reading from or writing to same location is possible:
- Concurrent Read, Concurrent Write (CRCW) PRAM
  simultaneous reading from or writing to same location is possible:
  - Weak CRCW
  - Common CRCW
  - Arbitrary CRCW
  - Priority CRCW
  - Combining CRCW
  (global sum, max, etc.)

No need for ERCW...
Divide & Conquer Parallel Sum Algorithm in the PRAM / Circuit (DAG) cost model

Given \( n \) numbers, \( x_0, x_1, \ldots, x_n \) stored in an array.

The global sum \( \sum_{i=0}^{n-1} x_i \) can be computed in \( \log_2 n \) time steps on an EREW PRAM with \( n \) processors.

Parallel algorithmic paradigm used: Parallel Divide-and-Conquer

Divide phase: trivial, time \( O(1) \)

Recursive calls: parallel time \( T(n/2) \)

Combine phase: addition, time \( O(1) \)

Use induction or the master theorem [Cormen+’90 Ch.4] \( T(n) = T(n/2) + O(1) \) \( T(1) = O(1) \)

Recurrence equation for parallel execution time:

\[ T(n) = T(n/2) + O(1) \]

Recursive formulation of DC parallel sum algorithm in some programming model

```
cilk int parsum ( int *d, int from, int to )
{
    int mid, sumleft, sumright;
    if (from == to)
        return d[from]; // base case
    else {
        mid = (from + to) / 2;
        sumleft = spawn parsum ( d, from, mid );
        sumright = parsum ( d, mid+1, to );
        sync; // The main program:
        return sumleft + sumright;
    }
}
```

Implementation e.g. in Cilk: (shared memory)

```
// The main program:
main()
{
    ... parsum ( data, 0, n-1 );
    ...
}
```

Fork-Join execution style:

- Single task starts, tasks spawn child tasks for independent subtasks, and synchronize with them

For a fixed number of processors ...

- Usually, \( p << n \)
- Requires scheduling the work to \( p \) processors

(A) manually, at algorithm design time:
- Requires algorithm engineering
- E.g. stop the parallel divide-and-conquer e.g. at subproblem size \( n/p \)

For parallel sum:
- Step 0. Partition the array of \( n \) elements in \( p \) slices of \( n/p \) elements each (= domain decomposition)
- Step 1. Each processor calculates a local sum for one slice, using the sequential sum algorithm, resulting in \( p \) partial sums (intermediate values)
- Step 2. The \( p \) processors run the parallel algorithm to sum up the intermediate values to the global sum.

Iterative formulation of DC parallel sum in EREW-PRAM model

```
int sum(int a[], int n)
{
    int d, dd;
    int ID = rerank();
    d = 1;
    while (d<n) {
        dd = d;
        d = d*2;
        if (dd<=n) { a[dd] = a[dd] + a[ID+dd]; }
    }
}
```

```
Circuit / DAG model

- Independent of how the parallel computation is expressed, the resulting (unrolled) task graph looks the same.

- Task graph is a directed acyclic graph (DAG) \( G=(V,E) \)
  - Set \( V \) of vertices: elementary tasks (taking time 1 resp. \( O(1) \) each)
  - Set \( E \) of directed edges: dependences (partial order on tasks)
  - \((v_i,v_j) \in E \Rightarrow v_j \) must be finished before \( v_i \) can start

- Critical path = longest path from an entry to an exit node

- Length of critical path is a lower bound for parallel time complexity

- Parallel time can be longer if number of processors is limited
  - schedule tasks to processors such that dependences are preserved (by programmer (SPMD execution) or run-time system (fork-join exec.))
```

Recursive formulation of DC parallel sum algorithm in some programming model

```
cilk int parsum ( int *d, int from, int to )
{
    int mid, sumleft, sumright;
    if (from == to) return d[from]; // base case
    else {
        mid = (from + to) / 2;
        sumleft = spawn parsum ( d, from, mid );
        sumright = parsum ( d, mid+1, to );
        sync; return sumleft + sumright;
    }
}
```

Implementation e.g. in Cilk: (shared memory)

```
// The main program:
main()
{
    ... parsum ( data, 0, n-1 );
    ...
}
```

Fork-Join execution style:

- Single task starts, tasks spawn child tasks for independent subtasks, and synchronize with them

For a fixed number of processors ...

- Usually, \( p << n \)
- Requires scheduling the work to \( p \) processors

(A) manually, at algorithm design time:
- Requires algorithm engineering
- E.g. stop the parallel divide-and-conquer e.g. at subproblem size \( n/p \)

For parallel sum:
- Step 0. Partition the array of \( n \) elements in \( p \) slices of \( n/p \) elements each (= domain decomposition)
- Step 1. Each processor calculates a local sum for one slice, using the sequential sum algorithm, resulting in \( p \) partial sums (intermediate values)
- Step 2. The \( p \) processors run the parallel algorithm to sum up the intermediate values to the global sum.
For a fixed number of processors ... ?

- Usually, \( p << n \)
- Requires scheduling the work to \( p \) processors

(B) automatically, at run time:
- Requires a task-based runtime system with dynamic scheduler
  - Each newly created task is dispatched at runtime to an available worker processor.
  - Load balancing (overhead)
    - Central task queue where idle workers fetch next task to execute
    - Local task queues + Work stealing – idle workers steal a task from some other processor

BSP-Model

Bulk-synchronous parallel programming

BSP computer = abstract message passing architecture \( (p, L, g, k) \)

- MIMD
- SPMIM
- \( h \)-relation models communication pattern / volume
  - \( h = \max_{i \in [0..p]} h_i \)

\( t_{\text{step}} = w + h g + L \)

BSP program = sequence of supersteps, separated by (logical) barriers

Delay Model

Idealized multicompurer: point-to-point communication costs overhead \( t_{\text{com}} \).

Cost of communicating a larger block of \( n \) bytes:

\[
t_{\text{com}}(n) = \text{sender overhead} + \text{latency} + \text{receiver overhead} + n \times \text{bandwidth} = t_{\text{start}} + n \times t_{\text{bandwidth}}
\]

Assumption: network not overloaded; no conflicts occur at routing

- \( t_{\text{start}} \) = startup time (time to send a 0-byte message)
- accounts for hardware and software overhead.

- \( t_{\text{bandwidth}} = \) transfer rate; sent time per word sent.
- depends on the network bandwidth.

BSP Example: Global Maximum (NB: non-optimal algorithm)

Compute maximum of \( n \) numbers \( A[0,...,n-1] \) on BSP\((p, L, g, k)\):

- \( A[0,...,n-1] \) distributed block-wise across \( p \) processors

\[
\text{step} \\
\begin{align*}
&\text{local computation phase:} \\
&\quad m := \ldots m_i \\
&\quad \text{for all } A[i] \text{ in my local partition of } A \\
&\quad m := \max \{ m, A[i] \} \\
\end{align*}
\]

\text{communication phase:}

\[
\begin{align*}
&\text{if } \text{myPID} = 0 \\
&\quad \text{send } (m, 0) \\
\end{align*}
\]

\[
\begin{align*}
&\text{else if on } P_0 \\
&\quad \text{for each } i \in \{1,...,p-1\} \\
&\quad \text{send } (m, i) \\
\end{align*}
\]

\[
\begin{align*}
&\text{step} \\
&\quad \text{if } \text{myPID} = 0 \\
&\quad \quad \text{for each } i \in \{1,...,p-1\} \\
&\quad \quad m := \max \{ m, m_i \} \\
\end{align*}
\]

Local work:

\( O(n/p) \)

Communication:

\( h = p - 1 \) (\( P_i \) is bottleneck)

\[
\begin{align*}
&\text{step} \\
&\quad t_{\text{step}} = w + h g + L \\
&\quad = \Theta \left( \frac{n}{p} + pg + L \right)
\end{align*}
\]

LogP Model \( \rightarrow \) TDDC78

LogP model

[Culler et al. 1993]

For the cost of communicating small messages (a few bytes)

4 parameters:

- latency \( L \)
- overhead \( o \)
- gap \( g \) (models bandwidth)
- processor number \( P \)

abstracts from network topology

- gap \( g \) = inverse network bandwidth per processor:
- Network capacity is \( L/g \) messages to or from each processor.
- \( L, o, g \) typically measured as multiples of the CPU cycle time.
- transmission time for a small message:

\[ 2 \cdot o + L \text{ if the network capacity is not exceeded} \]

LogP Model: Example \( \rightarrow \) TDDC78

Example: Broadcast on a 2-dimensional hypercube

With example parameters \( P = 4, \ a = 2 \mu s, \ g = 3 \mu s, \ L = 5 \mu s \)

- \( \text{P0} \)
- \( \text{P1} \)
- \( \text{P2} \)
- \( \text{P3} \)

It takes at least \( 18 \mu \text{sec} \) to broadcast 1 byte from \( \text{P0} \) to \( \text{P1}, \text{P2}, \text{P3} \)

Remark: for determining time-optimal broadcast trees in LogP, see [Papadimitriou/Yannakakis’89], [Karp et al.’93].
Analysis of Parallel Algorithms

Performance metrics of parallel programs

- **Parallel execution time**
  - Counted from the start time of the earliest task to the finishing time of the latest task
- **Work** – the total number of performed elementary operations
- **Cost** – the product of parallel execution time and #processors
- **Speed-up**
  - the factor by how much faster we can solve a problem with \( p \) processors than with 1 processor, usually in range \((0...p)\)
- **Parallel efficiency** = Speed-up / #processors, usually in \((0...1)\)
- **Throughput** = #operations finished per second
- **Scalability**
  - does speedup keep growing well also when #processors grows large?

Asymptotic Analysis

- Estimation based on a cost model and algorithm idea (pseudocode operations)
- Discuss behavior for large problem sizes, large #processors

Empirical Analysis

- Implement in a concrete parallel programming language
- Measure time on a concrete parallel computer
  - Vary number of processors used, as far as possible
  - More precise
  - More work, and fixing bad designs at this stage is expensive

Parallel work, time, cost

- Parallel work \( w_d(n) \) of algorithm \( A \) on an input of size \( n \)
  - \( w_d(n) = \max \) number of instructions performed by all procs during execution of \( A \), where in each (parallel) time step as many processors are available as needed to execute the step in constant time.

- Parallel time \( t_d(n) \) of algorithm \( A \) on input of size \( n \)
  - \( t_d(n) = \max \) number of parallel time steps required under the same circumstances

- Parallel cost \( c_d(n) \) of algorithm \( A \) on input of size \( n \)
  - \( c_d(n) = t_d(n) \cdot p_d(n) \)
  - \( c_d(n) \geq w_d(n) \)

- \( p_d(n) \) = max \#processors used in a step of \( A \)

- Work, time, and cost are thus worst-case measures.

- \( t_d(n) \) is sometimes called the depth of \( A \)
  - (cf. circuit model of (parallel) computation)

- \( p(n) \) = number of processors needed in time step \( i \), \( 0 \leq i < t_d(n) \), to execute the step in constant time. Then, \( w_d(n) = \sum p(n) \)

Analysis of Parallel Algorithms

Asymptotic Analysis

- Estimation based on a cost model and algorithm idea (pseudocode operations)
- Discuss behavior for large problem sizes, large #processors

Empirical Analysis

- Implement in a concrete parallel programming language
- Measure time on a concrete parallel computer

Parallel Time, Work, Cost

- Parallel sum algorithm

Work-optimal and cost-optimal

A parallel algorithm \( A \) is asymptotically work-optimal if \( w_d(p, n) = O(t_d(n)) \)

A parallel algorithm \( A \) is asymptotically cost-optimal if \( c_d(p, n) = O(t_d(n)) \)

Making the parallel sum algorithm cost-optimal:

- Instead of \( n \) processors, use only \( n/\log n \) processors.
- First, each processor computes sequentially the global sum of
  - the local elements. This takes time \( O(\log n) \).
- Then, they compute the global sum of \( n/\log n \) partial sums using the previous parallel sum algorithm.

Time: \( O(\log n) \) for local summation, \( O(\log n) \) for global summation

Cost: \( n/\log n \cdot O(\log n) = O(n) \) linear

This is an example of a more general technique based on Brent’s theorem.
**Amdahl's Law: Upper bound on Speedup**

Consider execution (trace) of parallel algorithm \( A \):

- sequential part \( A' \) where only 1 processor is active
- parallel part \( A^p \) that can be sped up perfectly by \( p \) processors

\[
\text{total work } w_T(n) = w_A(n) + w_{A'}(n), \text{ time } T = T_A + T_{A'}.
\]

**Amdahl’s Law**

If the sequential part of \( A \) is a fixed fraction of the total work irrespectively of the problem size \( n \), that is, if there is a constant \( \beta \) with \( \beta = \frac{w_A(n)}{w_T(n)} \leq 1 \)

the relative speedup of \( A \) with \( p \) processors is limited by

\[
\frac{p}{\beta p + (1-\beta)} < \frac{1}{\beta}
\]

**Proof of Amdahl’s Law**

\[
S_{rel} = \frac{T(1)}{T(p)} = \frac{T_A + T_{A'}}{T_A(1) + (1-\beta)T_A(1)/p}
\]

Assume perfect parallelizability of the parallel part \( A^p \), that is, \( T_{A'}(p) = (1-\beta)T_A(1)/p \):

\[
S_{rel} = \frac{T_A}{\beta T_A + (1-\beta)T_A(1)/p} \leq \frac{1}{\beta}
\]

Remark:

For most parallel algorithms the sequential part is not a fixed fraction.

**Remarks on Amdahl’s Law**

Not limited to speedup by parallelization only!
Can also be applied with other optimizations
- e.g. SIMDization, instruction scheduling, data locality improvements ...

Amdahl’s Law, general formulation:

If you speed up a fraction \((1-\beta)\) of a computation by a factor \(p\), the overall speedup is \(\frac{p}{\beta p + (1-\beta)}\), which is \(\frac{1}{\beta}\).

Implications

- Optimize for the common case.
- If \(1-\beta\) is small, optimization has little effect.
- Ignored optimization opportunities (also) limit the speedup.

As \(p \to \infty\), speedup is bound by \(\frac{1}{\beta}\).
Search Anomaly Example: Simple string search

Given: Large unknown shared string of length $n$, pattern of constant length $m << n$
Search for any occurrence of the pattern in the string.
Simple sequential algorithm: Linear search
Simple parallel algorithm: Contiguous partitions, linear search

Case 1: Pattern not found in the string
- measured parallel time $n/p$ steps
- speedup $= n / (n/p) = p$ ★

Case 2: Pattern found in the first position scanned by the last processor
- measured parallel time 1 step, sequential time $n-v/p$ steps
- observed speedup $n-v/p$, "superlinear" speedup?!?

But,...
...this is not the worst case (but the best case) for the parallel algorithm;
...and we could have achieved the same effect in the sequential algorithm,
...too, by altering the string traversal order

Simple Analysis of Cache Impact

- Call a variable (e.g. array element) live between its first and its last access in an algorithm’s execution
  - Focus on the large data structures of an algorithm (e.g. arrays)
- Working set of algorithm $A$ at time $t$
  $WS_A(t) \equiv \{ v: \text{variable } v \text{ live at } t \}$
- Worst-case working set size / working space of $A$
  $WSS_A = \max \{ WSS(t) \}$
- Average-case working set size / working space of $A$
  $\ldots = \text{avg} \{ WS_A(t) \}$

But, in the string search algorithm:
- Working set of $A$ at time $t$
  $WS_A(t) \equiv \{ v: \text{variable } v \text{ live at } t \}$
- Worst-case working set size / working space of $A$
  $WSS_A = \max \{ WSS(t) \}$
- Average-case working set size / working space of $A$
  $\ldots = \text{avg} \{ WS_A(t) \}$

Rule of thumb: Algorithm $A$ has good (last-level) cache locality if $WSS_A < 0.9 \cdot \text{SizeOfLastLevelCache}$
  - Assuming a fully associative cache with perfect LRU impl.
  - Impact of the cache line size not modeled
  - 10% reserve for some "small" data items (current instructions, loop variables, stack frame contents, ...)

★ Allows realistic performance prediction for simple regular algorithms
★ Hard to analyze WSS for complex, irregular algorithms

Parallel Simple string search

Given: Large unknown shared string of length $n$, pattern of constant length $m << n$
Search for any occurrence of the pattern in the string.
Simple parallel algorithm: Contiguous partitions, linear search
Simple sequential algorithm: Linear search

Parallel Simple string search

Given: Large unknown shared string of length $n$, pattern of constant length $m << n$
Search for any occurrence of the pattern in the string.
Simple parallel algorithm: Contiguous partitions, linear search
Simple sequential algorithm: Linear search

Simple Analysis of Cache Impact

- Call a variable (e.g. array element) live between its first and its last access in an algorithm’s execution
  - Focus on the large data structures of an algorithm (e.g. arrays)
- Working set of algorithm $A$ at time $t$
  $WS_A(t) \equiv \{ v: \text{variable } v \text{ live at } t \}$
- Worst-case working set size / working space of $A$
  $WSS_A = \max \{ WSS(t) \}$
- Average-case working set size / working space of $A$
  $\ldots = \text{avg} \{ WS_A(t) \}$

Rule of thumb: Algorithm $A$ has good (last-level) cache locality if $WSS_A < 0.9 \cdot \text{SizeOfLastLevelCache}$
  - Assuming a fully associative cache with perfect LRU impl.
  - Impact of the cache line size not modeled
  - 10% reserve for some "small" data items (current instructions, loop variables, stack frame contents, ...)

★ Allows realistic performance prediction for simple regular algorithms
★ Hard to analyze WSS for complex, irregular algorithms

Parallel Simple string search

Given: Large unknown shared string of length $n$, pattern of constant length $m << n$
Search for any occurrence of the pattern in the string.
Simple parallel algorithm: Contiguous partitions, linear search
Simple sequential algorithm: Linear search

Parallel Simple string search

Given: Large unknown shared string of length $n$, pattern of constant length $m << n$
Search for any occurrence of the pattern in the string.
Simple parallel algorithm: Contiguous partitions, linear search
Simple sequential algorithm: Linear search

Simple Analysis of Cache Impact

- Call a variable (e.g. array element) live between its first and its last access in an algorithm’s execution
  - Focus on the large data structures of an algorithm (e.g. arrays)
- Working set of algorithm $A$ at time $t$
  $WS_A(t) \equiv \{ v: \text{variable } v \text{ live at } t \}$
- Worst-case working set size / working space of $A$
  $WSS_A = \max \{ WSS(t) \}$
- Average-case working set size / working space of $A$
  $\ldots = \text{avg} \{ WS_A(t) \}$

Rule of thumb: Algorithm $A$ has good (last-level) cache locality if $WSS_A < 0.9 \cdot \text{SizeOfLastLevelCache}$
  - Assuming a fully associative cache with perfect LRU impl.
  - Impact of the cache line size not modeled
  - 10% reserve for some "small" data items (current instructions, loop variables, stack frame contents, ...)

★ Allows realistic performance prediction for simple regular algorithms
★ Hard to analyze WSS for complex, irregular algorithms
Further fundamental parallel algorithms

Parallel prefix sums
Parallel list ranking

... as time permits ...

The Prefix-Sums Problem

Given: a set $S$ (e.g., the integers)
- a binary associative operator $\oplus$ on $S$
- a sequence of $n$ items $x_0, x_1, ..., x_{n-1} \in S$

compute the sequence $y$ of prefix sums defined by

$$y_i = \bigoplus_{j=0}^{i} x_j \text{ for } 0 \leq i < n$$

An important building block of many parallel algorithms [Blelloch'89]

Typical operations $\oplus$:
- integer addition, maximum, bitwise AND, bitwise OR

Example:
- bank account: initially $0$, daily changes $x_0, x_1, ...$
- compute daily balances: $(0, x_0, x_0 + x_1, x_0 + x_1 + x_2, ...)$

Parallel prefix sums algorithm 1

A first attempt...

Naive parallel implementation:
- apply the definition:

$$y_i = \bigoplus_{j=0}^{i} x_j \text{ for } 0 \leq i < n$$

and assign one processor for computing each $y_i$

$\rightarrow$ parallel time $\Theta(n)$, work and cost $\Theta(n^2)$

But we observe:
- a lot of redundant computation (common subexpressions)

Sequential prefix sums algorithm

```c
void seq_prefix(int x[], int n, int y[]) {
    int i;
    int ps; // i'th prefix sum
    if (n > 0) ps = y[0] = x[0];
    for (i=1; i<n; i++) {
        ps += x[i];
        y[i] = ps;
    }
}
```

if run in parallel on $n$ virtual processors:
- time $\Theta(n)$, work $\Theta(n)$, cost $\Theta(n^2)$

Task dependence graph: linear chain of dependences

$\rightarrow$ seems to be inherently sequential --- how to parallelize?

Parallel prefix sums algorithm 2: Upper-Lower Parallel Prefix

Algorithmic technique: parallel divide & conquer

We consider the simplest variant, called Upper/Lower parallel prefix:

Recursive formulation:

- $N$-prefix is computed as

$$\begin{array}{c}
\text{Upper/Lower parallel prefix, unfolded for } N = 8 \\
\text{Upper prefix} \quad \text{Lower prefix}
\end{array}$$

Parallel time: $\log n$ steps, work: $n/2 \log n$ additions, cost: $\Theta(n \log n)$

Not work-optimal! And needs concurrent read...
Parallel Prefix Sums Algorithm 3:
Recursive Doubling (for EREW PRAM)

EREW (exclusive read, exclusive write) prefix sums algorithm:

Iterative formulation in data-parallel pseudocode:

```plaintext
real a: array[0...N - 1];
int stride;
stride := 1;
while stride < N do
  forall i : [0..N - 1] in parallel do
    a[i] := a[i] - stride + a[i + stride];
  stride := stride * 2;
end
```

Work: \( \Theta(n \log n) \)

("finally, sum in \( a[N - 1] \)"")

Parallel Prefix Sums Algorithm 4:
Odd-Even Parallel Prefix

Recursive definition: \( P^n(n) \):

```plaintext
EREW, 2 \log n - 2 \) time steps, work \( 2n - \log n - 2 \), cost \( \Theta(n \log n) \)
```

Not cost-optimal! But may use Brent's theorem...

Parallel Prefix Sums Algorithm 5
Ladner-Fischer Parallel Prefix Sums (1980)

Odd-Even Parallel Prefix Sums algorithm after work-time rescheduling:

cost-optimal (cost \( \Theta(n) \)) if using \( \Theta(n/\log n) \) virtual processors only

Parallel List Ranking (1)

Parallel list: (unordered) array of list items (one per proc.), singly linked

Problem: for each element, find the end of its linked list.

Algorithmic technique:
recursive doubling, here: "pointer jumping" [Wylie79]

The algorithm in pseudocode:

```plaintext
forall i in [1...N] in parallel do
  chum[i] := next[i];
  while chum[i] ≠ null and chum(chum[i]) ≠ null do
    chum[i] := chum(chum[i]);
  od
```

(lengths of chum lists halved in each step → \( \log N \) pointer jumping steps)

Parallel List Ranking (2)

Extended problem: compute the \( \text{rank} = \text{distance to end of list} \)

By pointer jumping:
in each step:

by my own distance value, I add the distance of my \( \text{chum} \)
that I splice out of the list

every step:
- doubles #lists
- halves lengths
→ \( \log_2 n \) steps
Not work-efficient!

Parallel List Ranking (3)

NULL-checks can be avoided by marking list end by a self-loop.

Pointer jumping algorithm for list ranking, implementation in Fork:

```plaintext
wyllie( sh LIST list[], sh int length )
|\nLIST *e; // private pointer
int nn;
e = list[ss]; // ss is my processor index
if (e->next != e) e->rank = 1; else e->rank = 0;
nn = length;
while (nn>1) {
e->rank = e->rank + e->next->rank;
e->next = e->next->next;
nn = nn>>1; // division by 2
}
```
Parallel Mergesort

… if time permits …

More on parallel sorting in TDDD56

Sequential Mergesort

Divide&Conquer
(here, divide is trivial, merge does all the work)
- `seqMrg(1,n2)` in time `O(n1+n2)
- `mergesort(n)` in time `O(n log n)

Split array (trivial, calculate n/2)

SeqMergesort

Sequential Mergesort

Time: `O(n log n)

void SeqMergesort ( int *array, int n ) // in place
{
  if (n==1) return;
  // divide and conquer:
  SeqMergesort ( array, n/2);
  SeqMergesort ( array + n/2, n-n/2 );
  // now the subarrays are sorted
  SeqMerge ( array, n/2, n-n/2 );
}

void SeqMerge ( int array, int n1, int n2 ) // sequential merge in place
{
  ... ordinary 2-to-1 merge in `O(n1+n2)` steps ... 
}

Towards a simple parallel Mergesort...

Divide&Conquer - independent subproblems!
- could run independent subproblems (calls) in parallel on different resources (cores)
  (but not much parallelism near the root)
Recursive parallel decomposition up to a maximum depth, to control #tasks

Simple Parallel Mergesort

void SParMergesort ( int *array, int n ) // in place
{
  if (n==1) return; // nothing to sort
  if (depth_limit_for_recursive_parallel_decomposition_reached())
    SParMergesort( array, n ); // switch to sequential
  // parallel divide and conquer:
  in parallel do {
    SParMergesort ( array, n/2);
    SParMergesort ( array + n/2, n-n/2 );
  }
  // now the two subarrays are sorted
  seq SeqMerge ( array, n/2, n-n/2 );
}

void SeqMerge ( int *array, int n1, int n2 ) // sequential merge in place
{
  ... merge in `O(n1+n2)` steps ... 
}
Simple Parallel Mergesort, Analysis

Parallel Time:
\[ T(n) = T(n/2) + T_{\text{split}}(n) + T_{\text{SeqMerge}}(n) + O(1) \]
\[ = T(n/2) + O(n) \]
\[ = O(n) + O(n/2) + O(n/4) + \ldots + O(1) \]
\[ = O(n) \]

Parallel Work:
\[ O(n \log n) \]

NB: SeqMerging (linear in input size) does all the heavy work in Mergesort.

How to merge in parallel?
- For each element of the two arrays to be merged, calculate its final position in the merged array by cross-ranking
  - rank( x, (a[0], …, a[n-1])) = #elements a[i] < x
  - Compute rank by sequential binary search, time \( O(\log n) \)

ParMerge ( int a[n1], int b[n2] )

```
for all i in 0…n1-1 in parallel
  rank_a_in_b[i] = compute_rank( a[i], b, n2 );
for all i in 0…n2-1 in parallel
  rank_b_in_a[i] = compute_rank( b[i], a, n1 );
for all i in 0…n-1 in parallel
  c[i + rank_a_in_b[i]] = a[i];
  c[i + rank_b_in_a[i]] = b[i];
```

Time for one binary search: \( O(\log n) \)
Par Time for ParMerge: \( O(\log n) \)
Par Work for ParMerge: \( O(n \log n) \)

Example: ParMerge

```
a = (2, 3, 7, 9 ), b = (1, 4, 5, 8 ), indices start at 0
rank_a_in_b = ( 1, 1, 3, 4 )
rack_b_in_a = ( 0, 2, 2, 3 )
a[0]  to  pos. c[ 0+1] = 1
a[1]  to  pos. c[ 1+1] = 2
b[0]  to  pos. c[ 0+0] = 0
b[1]  to  pos. c[ 1+2] = 3
b[2]  to  pos. c[ 2+2] = 4
```

After copying,
\[ c = ( 1, 2, 3, 4, 5, 7, 8, 9 ) \]

Summary

Parallel computation model = programming model + performance model
\( \to \) quantitative basis for design and analysis of parallel algorithms

Use simple performance models (PRAM, Delay, BSP)
early in the design process.

Refine performance model at later stages (BSP, LogP, LogGP)
and conduct simple experiments to derive model parameters
During implementation, compare performance to predictions by the model
\( \to \) may identify implementation errors and improve quality.

Survey article (see compendium appendix)

Further Reading