Sequential Algorithms

Classical Algorithm Design:
• One machine/CPU/process/... doing a computation

RAM (Random Access Machine):
• Basic standard model
• Unit cost basic operations
• Unit cost access to all memory cells

Sequential Algorithm / Program:
• Sequence of operations
  (executed one after the other)
Parallel and Distributed Algorithms

Today’s computers/systems are not sequential:
• Even cell phones have several cores
• Future systems will be highly parallel on many levels
• This also requires appropriate algorithmic techniques

Goals, Scenarios, Challenges:
• Exploit parallelism to speed up computations
• Shared resources such as memory, bandwidth, …
• Increase reliability by adding redundancy
• Solve tasks in inherently decentralized environments
• …
Parallel and Distributed Systems

• Many different forms

• Processors/computers/machines/... communicate and share data through
  – Shared memory or message passing

• Computation and communication can be
  – Synchronous or asynchronous

• Many possible topologies for message passing

• Depending on system, various types of faults
Challenges

Algorithmic and theoretical challenges:

• How to parallelize computations
• Scheduling (which machine does what)
• Load balancing
• Fault tolerance
• Coordination / consistency
• Decentralized state
• Asynchrony
• Bounded bandwidth / properties of comm. channels
• ...

Algorithm Theory, WS 2012/13

Fabian Kuhn
Models

• A large variety of models, e.g.:

• **PRAM** (Parallel Random Access Machine)
  – Classical model for parallel computations

• **Shared Memory**
  – Classical model to study coordination / agreement problems, distributed data structures, ...

• **Message Passing** (fully connected topology)
  – Closely related to shared memory models

• Message Passing in **Networks**
  – Decentralized computations, large parallel machines, comes in various flavors...
PRAM

- Parallel version of RAM model
- \( p \) processors, shared random access memory

- Basic operations / access to shared memory cost 1
- Processor operations are synchronized
- Focus on parallelizing computation rather than cost of communication, locality, faults, asynchrony, ...
Other Parallel Models

- **Message passing**: Fully connected network, local memory and information exchange using messages

- **Dynamic Multithreaded Algorithms**: Simple parallel programming paradigm
  - E.g., used in Cormen, Leiserson, Rivest, Stein (CLRS)

```plaintext
FIB(n)
1  if  n < 2
2   then return  n
3  x ← spawn FIB(n − 1)
4  y ← spawn FIB(n − 2)
5  sync
6  return  (x + y)
```
Parallel Computations

Sequential Computation:
• Sequence of operations

Parallel Computation:
• Directed Acyclic Graph (DAG)
Parallel Computations

$T_p$: time to perform comp. with $p$ procs

- $T_1$: work (total # operations)
  - Time when doing the computation sequentially

- $T_\infty$: critical path / span
  - Time when parallelizing as much as possible

- Lower Bounds:
  \[ T_p \geq \frac{T_1}{p}, \quad T_p \geq T_\infty \]

$T_1 = 11$

$T_\infty = 5$
Parallel Computations

$T_p$: time to perform comp. with $p$ procs

- Lower Bounds:
  \[ T_p \geq \frac{T_1}{p}, \quad T_p \geq T_{\infty} \]

- Parallelism: $\frac{T_1}{T_{\infty}}$
  - maximum possible speed-up

- Linear Speed-up:
  \[ \frac{T_p}{T_1} = \Theta(p) \]
Scheduling

• How to assign operations to processors?

• Generally an online problem
  – When scheduling some jobs/operations, we do not know how the computation evolves over time

**Greedy (offline) scheduling:**

• Order jobs/operations as they would be scheduled optimally with $\infty$ processors (topological sort of DAG)
  – Easy to determine: With $\infty$ processors, one always schedules all jobs/ops that can be scheduled

• Always schedule as many jobs/ops as possible

• Schedule jobs/ops in the same order as with $\infty$ processors
  – i.e., jobs that become available earlier have priority
Brent’s Theorem: On $p$ processors, a parallel computation can be performed in time

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty.$$  

**Proof:**

- Greedy scheduling achieves this...
- \#operations scheduled with $\infty$ processors in round $i$: $x_i$

\[
\begin{align*}
T_p & \leq \sum_{i=1}^{\infty} (1 + \frac{x_i - 1}{p}) = T_\infty + \sum_{i=1}^{\infty} \frac{x_i}{p} - \frac{1}{p} \sum_{i=1}^{\infty} x_i = T_\infty + \frac{T_1 - T_\infty}{p} \\
\end{align*}
\]
Brent’s Theorem: On $p$ processors, a parallel computation can be performed in time

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty.$$ 

Proof:

• Greedy scheduling achieves this...

• #operations scheduled with $\infty$ processors in round $i$: $x_i$
Brent’s Theorem

Brent’s Theorem: On \( p \) processors, a parallel computation can be performed in time

\[
T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty.
\]

Corollary: Greedy is a 2-approximation algorithm for scheduling.

\[
\begin{align*}
T_p &\geq \frac{T_1}{p} \\
T_p &\geq T_\infty \\
T_p &< \frac{T_1}{p} + T_\infty \leq 2 \cdot \max \left\{ \frac{T_1}{p}, T_\infty \right\} \\
T_p &\geq \infty
\end{align*}
\]

Corollary: As long as the number of processors \( p = O(T_1/T_\infty) \), it is possible to achieve a linear speed-up.

\[
\frac{T_1}{p} = \Omega(T_\infty)
\]
Back to the PRAM:

- Shared random access memory, synchronous computation steps
- The PRAM model comes in variants...

**EREW (exclusive read, exclusive write):**
- Concurrent memory access by multiple processors is not allowed
- If two or more processors try to read from or write to the same memory cell concurrently, the behavior is not specified

**CREW (concurrent read, exclusive write):**
- Reading the same memory cell concurrently is OK
- Two concurrent writes to the same cell lead to unspecified behavior
- This is the first variant that was considered (already in the 70s)
PRAM

The PRAM model comes in variants...

**CRCW (concurrent read, concurrent write):**

- Concurrent reads and writes are both OK
- Behavior of concurrent writes has to be specified
  - Weak CRCW: concurrent write only OK if all processors write 0
  - Common-mode CRCW: all processors need to write the same value
  - Arbitrary-winner CRCW: adversary picks one of the values
  - Priority CRCW: value of processor with highest ID is written
  - Strong CRCW: largest (or smallest) value is written

- The given models are ordered in strength:

  weak \leq \text{common-mode} \leq \text{arbitrary-winner} \leq \text{priority} \leq \text{strong}
Some Relations Between PRAM Models

**Theorem:** A parallel computation that can be performed in time $t$, using $p$ processors on a strong CRCW machine, can also be performed in time $O(t \log p)$ using $p$ processors on an EREW machine.

- Each (parallel) step on the CRCW machine can be simulated by $O(\log p)$ steps on an EREW machine.

**Theorem:** A parallel computation that can be performed in time $t$, using $p$ probabilistic processors on a strong CRCW machine, can also be performed in expected time $O(t \log p)$ using $O(p/\log p)$ processors on an arbitrary-winner CRCW machine.

- The same simulation turns out more efficient in this case.
Some Relations Between PRAM Models

**Theorem:** A computation that can be performed in time $t$, using $p$ processors on a strong CRCW machine, can also be performed in time $O(t)$ using $O(p^2)$ processors on a weak CRCW machine.

**Proof:**
- **Strong**: largest value wins, **weak**: only concurrently writing 0 is OK

\[ t = O(1) \quad p = \sqrt{n} \]

\[ O(t) = O(1) \quad O(p^2) = O(n) \]
Some Relations Between PRAM Models

**Theorem:** A computation that can be performed in time \( t \), using \( p \) processors on a strong CRCW machine, can also be performed in time \( O(t) \) using \( O(p^2) \) processors on a weak CRCW machine.

**Proof:**

- **Strong:** largest value wins, **weak:** only concurrently writing 0 is OK.
Computing the Maximum

**Observation:** On a strong CRCW machine, the maximum of a $n$ values can be computed in $O(1)$ time using $n$ processors

- Each value is concurrently written to the same memory cell

**Lemma:** On a weak CRCW machine, the maximum of $n$ integers between 1 and $\sqrt{n}$ can be computed in time $O(1)$ using $O(n)$ proc.

**Proof:**

\[
\max i \text{ such that } f_i = 0
\]

- We have $\sqrt{n}$ memory cells $f_1, \ldots, f_{\sqrt{n}}$ for the possible values
- Initialize all $f_i := 1$
- For the $n$ values $x_1, \ldots, x_n$, processor $j$ sets $f_{x_j} := 0$
  - Since only zeroes are written, concurrent writes are OK
- Now, $f_i = 0$ iff value $i$ occurs at least once
- Strong CRCW machine: max. value in time $O(1)$ w. $O(\sqrt{n})$ proc.
- Weak CRCW machine: time $O(1)$ using $O(n)$ proc. (prev. lemma)
Computing the Maximum

**Theorem:** If each value can be represented using $O(\log n)$ bits, the maximum of $n$ (integer) values can be computed in time $O(1)$ using $O(n)$ processors on a weak CRCW machine.

**Proof:**

- First look at $\frac{\log_2 n}{2}$ highest order bits
- The maximum value also has the maximum among those bits
- There are only $\sqrt{n}$ possibilities for these bits
- max. of $\frac{\log_2 n}{2}$ highest order bits can be computed in $O(1)$ time
- For those with largest $\frac{\log_2 n}{2}$ highest order bits, continue with next block of $\frac{\log_2 n}{2}$ bits, ...