3 Introduction

Parallel Computing
A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems
A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n **and** on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Speedup

Suppose a problem P has sequential complexity $T^*(n)$, i.e., there is no algorithm that solves P in time $o(T^*(n))$.

Definition 1
The speedup $S_p(n)$ of a parallel algorithm A that requires time $T_p(n)$ for solving P with p processors is defined as

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \ .$$

Clearly, $S_p(n) \leq p$. **Goal:** obtain $S_p(n) \approx p$.

It is common to replace $T^*(n)$ by the time bound of the best **known** sequential algorithm for P!

Efficiency

Definition 2
The efficiency of a parallel algorithm A that requires time $T_p(n)$ when using p processors on a problem of size n is

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \ .$$

$E_p(n) \approx 1$ indicates that the algorithm is running roughly p times faster with p processors than with one processor.

Note that $E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$. Hence, the efficiency goes down rapidly if $p \geq T_1(n)/T_\infty(n)$.

Disadvantage: cost-measure does not relate to the optimum sequential algorithm.

Parallel Models — Requirements

Simplicity
A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability
Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

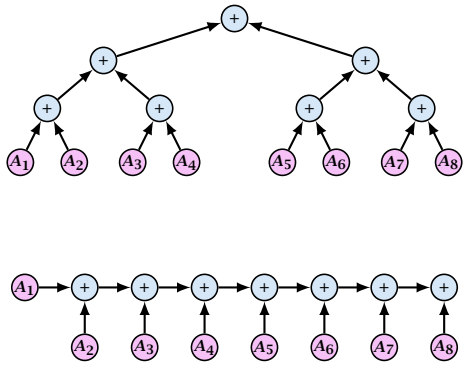
A real satisfactory model does not exist!

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

Example: Addition



Here, vertices without incoming edges correspond to input data. The graph can be viewed as a data flow graph.

DAG model — computation graph

The DAG itself is not a complete algorithm. A scheduling implements the algorithm on a parallel machine, by assigning a time-step t_v and a processor p_v to every node.

Definition 3
A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$, s.t.,

- $p_v \in \{1, \ldots, p\}$; $t_v \in \{1, \ldots, T\}$
- $t_u = t_v \Rightarrow p_u \neq p_v$
- $(u, v) \in E \Rightarrow t_v \geq t_u + 1$

where a non-internal node x (an input node) has $t_x = 0$.
T is the length of the schedule.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG
$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$
$$T_p(n) \geq T_1(n)/p$$

Lemma 4
A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5
Finding an optimal schedule is in general NP-complete.

Note that the DAG model as defined is a non-uniform model of computation.

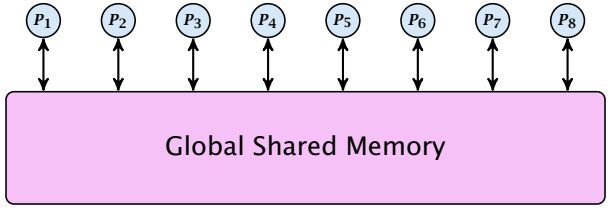
In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

PRAM Model



All processors are synchronized.

In every round a processor can:

- read a register from global memory into local memory
- do a local computation à la RAM
- write a local register into global memory

PRAM Model

Every processor executes the same program.

However, the program has access to two special variables:

- p: total number of processors
- $id \in \{1,\ldots,p\}$: the id of the current processor

The following (stupid) program copies the content of the global register $x[1]$ to registers $x[2]\ldots x[p]$.

Algorithm 1 copy

```
1: if id = 1 then round ← 1
2: while round ≤ p and id = round do
3:         x[id + 1] ← x[id]
4:         round ← round + 1
```


PRAM Model

- processors can effectively execute different code because of branching according to id
- however, not arbitrarily; still uniform model of computation

Often it is easier to explicitly define which parts of a program are executed in parallel:

Algorithm 2 sum

```
1: // computes sum of x[1]...x[p]
2: // red part is executed only by processor 1
3: r ← 1
4: while 2^r ≤ p do
5:         for id mod 2^r = 1 pardo
6:         // only executed by processors whose id matches
7:                 x[id] = x[id] + x[id + 2^(r−1)]
8:         r ← r + 1
9: return x[1]
```

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed; write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Algorithm 3 sum

```
1: // computes sum of x[1]...x[p]
2: r ← 1
3: while 2^r ≤ p do
4:         for id mod 2^r = 1 pardo
5:                 x[id] = x[id] + x[id + 2^(r−1)]
6:         r ← r + 1
7: return x[1]
```

The above is an EREW PRAM algorithm.

On a CREW PRAM we could replace Line 4 by
for $1 \le id \le p$ **pardo**

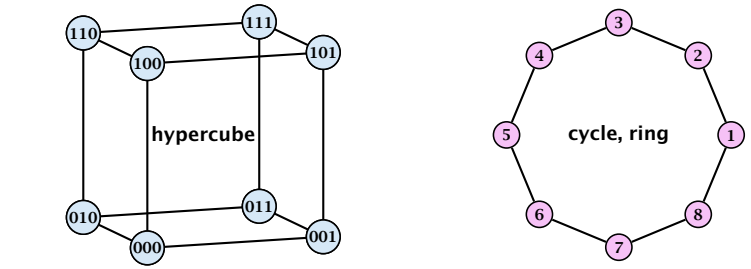
PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

Network of Workstations — NOWs

- interconnection network represented by a graph $G = (V, E)$
- each $v \in V$ represents a processor
- an edge $\{u, v\} \in E$ represents a two-way communication link between processors u and v
- network is asynchronous
- all coordination/communiation has to be done by explicit message passing

Typical Topologies



Network of Workstations — NOWs

Computing the sum on a d-dimensional hypercube. Note that $x[0] \ldots x[2^d - 1]$ are stored at the individual nodes.

Processors are numbered consecutively starting from 0

Algorithm 4 sum

```
1:  // computes sum of x[0]...x[2^d − 1]
2:  r ← 1
3:  while 2^r ≤ 2^d do // p = 2^d
4:      if id mod 2^r = 0 then
5:          temp ← receive(id + 2^(r−1))
6:          x[id] = x[id] + temp
7:      if id mod 2^r = 2^(r−1) then
8:          send(x[id], id − 2^(r−1))
9:      r ← r + 1
10: if id = 0 then return x[id]
```

Network of Workstations — NOWs

Remarks

- One has to ensure that at any point in time there is at most one active communication along a link
- There also exist synchronized versions of the model, where in every round each link can be used once for communication
- In particular the asynchronous model is quite realistic
- Difficult to develop and analyze algorithms as a lot of low level communication has to be dealt with
- Results only hold for one specific topology and cannot be generalized easily

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \leq P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \leq i \leq T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \leq \sum_i \left(\lfloor W_i(n)/p \rfloor + 1 \right) \leq \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Why nearly always?

We need to assign processors to operations.

- every processor p_i needs to know whether it should be active
- in case it is active it needs to know which operations to perform

design algorithms for an arbitrary number of processors; keep total time and work low

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

If such an algorithm has running time $T(n)$ it has speedup

$$S_p(n) = \Omega\left(\frac{T^*(n)}{T^*(n)/p + T(n)}\right) = \Omega\left(\frac{pT^*(n)}{T^*(n) + pT(n)}\right) = \Omega(p)$$

for $p = \mathcal{O}(T^*(n)/T(n))$.

This means by improving the time $T(n)$, (while using same work) we improve the range of p, for which we obtain optimal speedup.

We call an algorithm worktime (WT) optimal if $T(n)$ cannot be asymptotically improved by any work optimal algorithm.

Example

Algorithm for computing the sum has work $W(n) = \mathcal{O}(n)$. optimal

$T(n) = \mathcal{O}(\log n)$. Hence, we achieve an optimal speedup for $p = \mathcal{O}(n/\log n)$.

One can show that any CREW PRAM requires $\Omega(\log n)$ time to compute the sum.

Communication Cost

When we differentiate between local and global memory we can analyze communication cost.

We define the communication cost of a PRAM algorithm as the worst-case traffic between the local memory of a processor and the global shared memory.

Important criterion as communication is usually a major bottleneck.

Communication Cost

Algorithm 5 MatrixMult(A, B, n)

1: **Input:** $n \times n$ matrix A and B; $n = 2^k$
2: **Output:** $C = AB$
3: **for** $1 \le i, j, \ell \le n$ **pardo**
4: $\quad X[i,j,\ell] \leftarrow A[i,\ell] \cdot B[\ell,j]$
5: **for** $r \leftarrow 1$ **to** $\log n$
6: $\quad$ **for** $1 \le i, j \le n$; $\ell \bmod 2^r = 1$ **pardo**
7: $\quad\quad X[i,j,\ell] \leftarrow X[i,j,\ell] + X[i,j,\ell + 2^{r-1}]$
8: $C[i,j] \leftarrow X[i,j,\ell]$

On n^3 processors this algorithm runs in time $\mathcal{O}(\log n)$.
It uses n^3 multiplications and $\mathcal{O}(n^3)$ additions.

What happens if we have n processors?

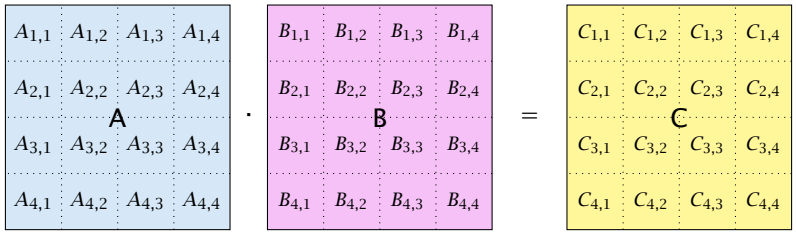
Phase 1
p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j, \ell \le n$
n^2 time; n^2 communication for every processor

Phase 2 (round r)
p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$
$n \cdot n/2^r$ time; no communication

Phase 3
p_i writes i-th row into $C[i,j]$'s.
n time; n communication

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.



Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$
work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$
time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

The communication cost is only $\mathcal{O}(n^{4/3} \log n')$ as a processor in the original problem touches at most $\log n$ entries of the matrix.

Each entry has size $\mathcal{O}(n^{4/3})$.

The algorithm exhibits less parallelism but still has optimum work/runtime for just n processors.

much, much better in practise