

GEM 1501 Problem Solving With Computers

Lecture 10:

Parallelism and Concurrency

Frank Stephan

# Summary of Previous Lecture

- Turing Machines
- Multiple Stack Machines
- Church's Thesis
- Numerical Computation:  
Counter Programs
- Restricted Models:  
One Stack Machines and Finite Automata

# Church's Thesis

- Full computation power can be obtained by easy basic mechanisms.
- All intuitively computable functions can be computed by a machine using such basic mechanisms.
- Examples: Turing machines and multiple stack machines; computing is manipulating symbols by simple rules. Turing machines move around on a tape, multiple stack machines use several stacks as a memory.
- Turing machines can be used to formalize complexity classes adequately.
- The classes P and NP are the same when defined by Turing machines, Multiple stack machines or Java Script programs.

# Numerical Model: Counter Programs

```
function add(x,y) // Original Style
```

```
  1: z = 0;
  2: if x == 0 goto 6;
  3: x = x-1;
  4: z = z+1;
  5: goto 2;
  6: if y == 0 goto 10;
  7: y = y-1;
  8: z = z+1;
  9: goto 6;
10: return(z);
```

```
function add(x,y) // Java Script Style
```

```
{ var z = 0;
  while (x>0) { x = x-1; z = z+1; }
  while (y>0) { y = y-1; z = z+1; }
  return(z); }
```

# Power of counter programs

- Counter cover all computable functions;
- Counter programs are less efficient and use exponential time for adding and multiplying;
- Variant which can add, subtract and compare natural numbers in one step: this variant gives same notion of tractable problems as Turing machines and Multiple Stack Machines.

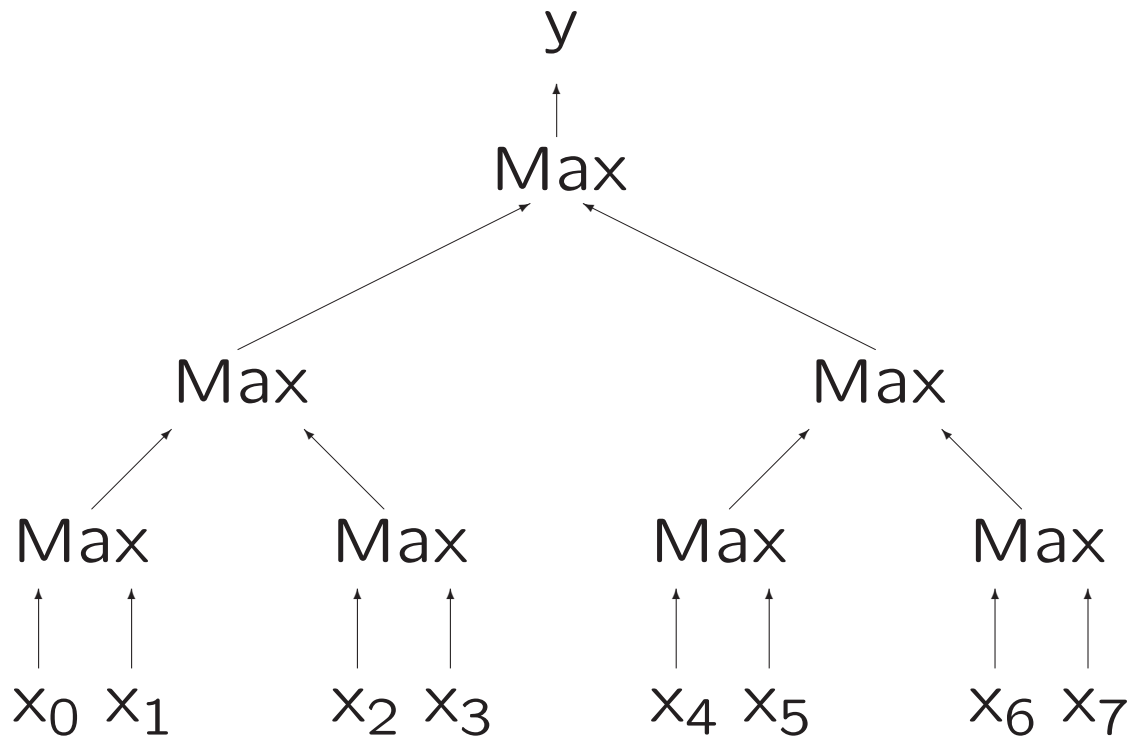
# Restricted Models

- Some restricted models do not have full computation power.
- One-Stack Automata accept all words having same quantity of zeros and ones but not those words having same quantity of zeros, ones and twos.
- Finite Automata cannot count at all.

# Parallelism and Concurrency

- **Parallel Computing**
- Parallel Sorting
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- Nick's Class
- Coordinating Parallelism
- New Models of Computing

# Parallel Divide and Conquer



Divide and Conquer implemented as a Circuit.

Each Maximum-Gate combines either two inputs or two subproblems below.

# Parallelism and Concurrency

- Idea: Distribute the work over many workers, who work simultaneously
- Issues:
  - Independence
  - Resources
  - Overhead

# Independence

- Some tasks are independent of each other and can easily be parallelized.

Example:  $X = 3; Y = 4;$

- Other tasks depend on each other.

Example:  $X = 3; Y = X + 1;$

# Resources

- Parallelism requires more hardware resources. There must be several processors available to do work in parallel.
- Resources incur cost.
- Only for large problems, this investment pays off.

# Overhead

- Parallelism often incurs an overhead for combining the work.
- This overhead may be small or large depending on the problem.
- Sometimes the overhead is so large that parallel solutions end up slower than sequential ones.

# Example: Summing Salaries

- Recall Maximum-Example from beginning
- Same works with all commutative operations: maximum, minimum, addition, multiplication.
- Summing of  $n$  salaries in company can be done with  $n$  processors in time  $\log(n)$ .
- In each step, two previous results are combined to a new one in parallel.

# Trade-off of Cost and Time

- The more processors the more work can be done in parallel.
- 200 processors permit to add  $n$  salaries in time  $n/200$  plus the time to combine the 200 intermediate results.
- $n$  processors permit to add  $n$  salaries in time  $\log(n)$ .
- Real power of parallel algorithms is based on assumption of *expanding parallelism*: the larger the task, the more processors are allocated to solve it.

# Parallelism and Concurrency

- Parallel Computing
- **Parallel Sorting**
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- Nick's Class
- Coordinating Parallelism
- New Models of Computing

# Mergesort

Subroutine *sort-L*

- if  $L$  consists of one element then it is sorted;
- otherwise do the following:
  - split  $L$  into two halves,  $L_1$  and  $L_2$
  - call *sort-L*<sub>1</sub>;
  - call *sort-L*<sub>2</sub>;
  - merge resulting lists into a single sorted list

# Example: Sorting in Parallel

Subroutine *parallel-sort-L*

- if  $L$  consists of one element then it is sorted;
- otherwise do the following:
  - split  $L$  into two halves,  $L_1$  and  $L_2$
  - call *parallel-sort- $L_1$*  and *parallel-sort- $L_2$*  simultaneously;
  - merge resulting lists into a single sorted list

# Complexity of Parallel Sorting

- Parallel merge sort has only the subsorts parallelized but not the merges.
- Merges use  $O(n)$  time.
- Since subproblems have problem size  $n/2$ , one has that there is a constant  $c$  such that the whole algorithm needs at most  $c \cdot (n + n/2 + n/4 + n/8 + \dots) = 2 \cdot c \cdot n$  parallel time.
- Before improving the algorithm, exact model is given.

# Formalizing Parallelism

- Every processor connected to constantly many other ones;
- In every step, any information can only go from a processor to one connected with it;
- Communication in network gives as lower bound the logarithm of the number of processors;
- Question: is there also a logarithmic upper bound?

# Sorting Networks

## Odd-Even Sorting Networks

- Merging process of parallel merge sort can be parallelized.
- Basic unit are comparators which sort two inputs.
- Odd-even sorting network consists of  $O(n \cdot \log^2(n))$  comparators which sort the network in parallel time  $O(\log^2(n))$ .

## Optimal Sorting Networks

- Complicated Design.
- Need  $O(n)$  processors and parallel time  $O(\log(n))$ .

# Example: 8-Odd-Even

$a, b \rightarrow c, d$  means that  $c = \min(a, b)$  and  $d = \max(a, b)$ ;  $e \rightarrow f$  means  $f = e$ ;

Parallel sort of two 4-sorting subproblems

$A_1, A_2 \rightarrow B_1, B_3$ ;  $A_3, A_4 \rightarrow B_2, B_4$ ;  $A_5, A_6 \rightarrow B_5, B_7$ ;  $A_7, A_8 \rightarrow B_6, B_8$ ;  
 $B_1, B_2 \rightarrow C_1, C_2$ ;  $B_3, B_4 \rightarrow C_3, C_4$ ;  $B_5, B_6 \rightarrow C_5, C_6$ ;  $B_7, B_8 \rightarrow C_7, C_8$ ;  
 $C_1 \rightarrow D_1$ ;  $C_2, C_3 \rightarrow D_2, D_3$ ;  $C_4 \rightarrow D_4$ ;  $C_5 \rightarrow D_5$ ;  $C_6, C_7 \rightarrow D_6, D_7$ ;  $C_8 \rightarrow D_8$ ;

Merging of the results

$D_1, D_5 \rightarrow E_1, E_2$ ;  $D_2, D_6 \rightarrow E_4, E_6$ ;  $D_3, D_7 \rightarrow E_3, E_5$ ;  $D_4, D_8 \rightarrow E_7, E_8$ ;  
 $E_1 \rightarrow F_1$ ;  $E_2, E_3 \rightarrow F_2, F_4$ ;  $E_4 \rightarrow F_3$ ;  $E_5 \rightarrow F_6$ ;  $E_6, E_7 \rightarrow F_5, F_7$ ;  $E_8 \rightarrow F_8$ ;  
 $F_1 \rightarrow G_1$ ;  $F_2, F_3 \rightarrow G_2, G_3$ ;  $F_4, F_5 \rightarrow G_4, G_5$ ;  $F_6, F_7 \rightarrow G_6, G_7$ ;  $F_8 \rightarrow G_8$ ;

Unsorted:  $A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8$ ;

Sorted:  $G_1, G_2, G_3, G_4, G_5, G_6, G_7, G_8$ .

Harel's book has a better graphical representation on page 265.

Sorting 10 inputs this way  $\rightarrow$  Assignment 19

# Product Complexity

- Parallel time complexity: Ignores number of processors.
- Product complexity: Time  $\cdot$  Size.
- Lower bound of product complexity is sequential running time, thus product complexity of sorting is at least  $O(n \cdot \log(n))$ .
- Parallel merge sort has product complexity of  $O(n \cdot n)$ .
- Sequential merge sort has product complexity of  $O(n \cdot \log(n))$ .
- Odd-even sort has product complexity of  $O(n \cdot \log^4(n))$ .
- Optimal sorting algorithms use  $O(n)$  processors and  $O(\log n)$  time, product complexity is  $O(n \cdot \log(n))$ .

# Parallelism and Concurrency

- Parallel Computing
- Parallel Sorting
- **Weighted Averages**
- Sequential PSPACE and parallel polynomial time
- Nick's Class
- Coordinating Parallelism
- New Models of Computing

# Weighted Average

- Students  $1, 2, \dots, n$ .
- Exams  $1, 2, \dots, m$  with weights  $w_1, \dots, w_m$ .
- Marks  $M_{i,j}$  for student  $i$  in Exam  $j$ .
- Average Mark  $A_i = \sum_j w_j \cdot M_{i,j}$  for student  $i$ .
- Try to compute  $A_1, \dots, A_n$  as fast as possible.

# Matrix Multiplication

- Multiplying matrices with other matrices or vectors is very common.
- Special computers for this task.
- Parallel mesh-organized or beehive-organized architectures.
- Vector computers: several highly specialized and fast processors in a pipeline; fast if data is processed in special form.
- Special variants of Fortran and other languages to program vector computers efficiently.
- User should follow certain programming conventions for effective code. Conventions are similar for vector computers and other parallel computers.

# From a Sun User Manual

What you can do:

```
do i = 1, 1000
  a(i) = b(i) * c(i)      ! Parallelized
end do
```

```
do k = 3, 1000
  x(k) = x(k-1) * x(k-2) ! Not parallelized - dependency.
end do
```

What the Compiler Does:

- \* Dependency analysis to detect loops that are parallelizable
- \* Parallelization of those loops

First loop can be distributed on four processors.

First processors does  $i=1, \dots, 250$ , second  $i=251, \dots, 500$ ,  
third  $i=501, \dots, 750$ , fourth  $i=751, \dots, 1000$ .

# Parallelism and Concurrency

- Parallel Computing
- Parallel Sorting
- Weighted Averages
- **Sequential PSPACE and parallel polynomial time**
- Nick's Class
- Coordinating Parallelism
- New Models of Computing

# Sequential PSPACE

- A Turing Machine computes a function  $f$  in PSPACE if there is a polynomial  $p$  such that the distance to its original position on the tape is always at most  $p(n)$  where  $n$  is the length of input on the tape.
- The output of functions in PSPACE has a length-bound polynomial in the length of the input.
- “Sequential Polynomial Space” is just called PSPACE.
- Terminology mostly used for decision problems.
- There are PSPACE-complete problems. For example alternating formula: Given a Boolean formula  $\psi$  with variables  $x_1, \dots, x_n, y_1, \dots, y_n$ , evaluate the expression

$$\exists x_1 \forall y_1 \exists x_2 \forall y_2 \dots \exists x_n \forall y_n \psi(x_1, \dots, x_n, y_1, \dots, y_n).$$

- Winning-strategies of several  $n \times n$  games are PSPACE-complete.

# Polynomially many processors

- Assume the algorithm needs  $p(n)$  processors and  $q(n)$  parallel time.
- For  $t = 1, \dots, q(n)$ , for  $m = 1, \dots, p(n)$ , simulate  $t$ th step on  $m$ th processor.
- Whole simulation procedure is in polynomial time.
- Parallel polynomial time with polynomially many processors does not go beyond polynomial time.

# Exponentially many processors

- Every function from  $\{0, 1\}^n$  to  $\{0, 1\}$  can be expressed by  $2^n$ -sized disjunction over terms like  $x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4 \wedge \dots \wedge \neg x_n$ .
- For example,  $(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3)$  computes whether the number of ones in the input is odd.
- Each of the  $2^n$  processors does either  $n$  conjunctions in time  $n$  or takes the constant 0.
- The resulting  $2^n$  terms are then combined by disjunctions in time  $O(n)$  which is logarithmic in  $2^n$ . A disjunction is the maximum on binary input, so see the maximum-example for such a circuit.
- Limitations:
  1.  $2^n$  is extremely large already for small  $n$ .
  2. Although there is a circuit of size  $2^n$ , how does one know which circuit to take for difficult functions.
  3. Parallelism cannot solve unsolvable problems.

# PSPACE and Parallelism

- PSPACE can be solved in parallel polynomial time with exponentially many processors; the general algorithm says also how to distribute the work on the processors and how to connect them.
- Can it be done better? Yes, if  $P = PSPACE$  with 1 processor. But it is unknown whether  $P = PSPACE$ .
- Is there a class  $F$  of functions such that a problem is in Sequential PSPACE iff it is in parallel time with  $f(n)$  processors for some  $f \in F$ ?

# Parallelism and Concurrency

- Parallel Computing
- Parallel Sorting
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- **Nick's Class**
- Coordinating Parallelism
- New Models of Computing

# Nick's Class

- NC is named after Nicholas Pippenger
- Nick's class contains problems that can be solved very fast (some power of the logarithm of  $n$ ) with only polynomially many processors.
- More precise definition: Circuits of polynomially many gates with two input lines arranged in polylogarithmic many layers.
- Problems in NC: Salary summation, finding maximum, sorting.

# Three Tape Turing Machines

- Three Tape Turing Machines are used to define computations with a space bound  $f$  where  $f$  might be sublinear.
- Three tapes with one head per tape: Input tape, working tape and output tape.
- Input tape: head is bidirectional and can read the input but not write.
- Working tape: head can read and write but the head is never more than  $f(n)$  fields away from the origin where  $n$  is the length of the input.
- Output tape: head can only move right and only write but not read.

# Sublinear Space Classes

- LOGSPACE is the set of all functions which can be computed with space bound  $f(n) = c \cdot \log(n)$  for some constant  $c$ .
- POLYLOGSPACE is the set of all functions which can be computed with space bound  $f(n) = \log^c(n)$  for some constant  $c$ .
- LOGSPACE  $\subset$  POLYLOGSPACE  $\subset$  PSPACE.

# Nick's Class and other Complexity Classes

- $\text{LOGSPACE} \subseteq \text{NC} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$ .
- $\text{NC} \subseteq \text{POLYLOGSPACE} \subset \text{PSPACE}$ . Thus  $\text{NC} \subset \text{PSPACE}$ .
- The remainder-variant of Euclid's Algorithm shows that GCD is in P. GCD is conjectured to be outside NC.
- The conjecture  $\text{NC} \subset \text{P}$  is unproven.

# Parallelism and Concurrency

- Parallel Computing
- Parallel Sorting
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- Nick's Class
- **Coordinating Parallelism**
- New Models of Computing

# Hotel Shower Problem

- Hotel has  $n$  rooms, each with a shower,  $n > 1$ .
- Due to water-problems, only one shower can be operated at the same time.
- Coordination necessary. No guest shall be permitted to start a showering process and then run out of water.

# Coordination for hotel shower

- Guest have to arrange protocol by communication;
- Coordination carried out by using blackboard with exclusive and shared writing areas.
- Goal 1: No conflict – it should not happen that several processes use a non-sharable resource at the same time;
- Goal 2: No deadlock – it should not happen that all processes stop in waiting of each other;
- Goal 3: No starvation – it should not happen that some few processes never make progress because of all others having priority.

# Example of Deadlock

Guest I wants to shower.

- Guest I writes message on boards in protected area:  $X[I]=1$ ;
- Guest I waits until all others are ready, that is, until  $X[J]==0$  for all  $J \neq I$ ;
- Then Guest I showers;
- Then Guest I erases his message:  $X[I]=0$ .

Deadlock can occur.

- Guest 1 and 2 simultaneously start the process and then wait for each other to set the variable to 0.
- More involved protocol necessary.

# A Protocol That Works

- Shared and Private variables.
- Private variable  $X[I]$  to mark position in request-queue;  
Shared variable  $Z[..]$  to permit others to advance in queue.
- Algorithm for Guest  $I$  to Shower

```
For (J=1;J<N;J=J+1)
  { X[I]=J; Z[J]=I;
    Wait until either X[K]<J for all K != I or Z[J] != I; }
Use Shower;
X[I] = 0;
```
- Parameter  $J$  represents position in Queue;  
Event  $Z[J] \neq I$  stands for receiving permission to advance;  
Each advancement is (indirectly) caused by a new member joining the queue or by all others being behind ( $X[K] < J$  for all  $K$ ).

# Semaphores

- Programming languages for concurrent programming have constructs for synchronization
- Example: Semaphore makes sure that not more than  $n$  processes use some resource simultaneously.
  - $S$  is shared integer variable initialized as  $n$ .
  - $request(S)$  decrements  $S$  if it is positive and waits otherwise.
  - $release(S)$  increments  $S$ .
- Both actions are atomic (cannot be interrupted).
- Process *requests*  $S$  and waits until access is granted. Then process carries out activity on the resource. Then process *releases*  $S$ .

# Queues as Semaphores

- Last protocol was imperfect in the sense that some process might repeatedly request a resource and starve out another one.
- Implementing Queues with Semaphores, shared variable implemented as a Java Script array; starvation does not happen.
- Protocol for one resource with a (\*) being an atomic instruction.

```
    var shared = new Array()  
    (*)    shared.push(guestname);    // Placing a request  
    while (shared[0] != guestname)  
        { do something else or wait; }  
    use the hotel shower;    // Request is now in first place  
    (*)    shared.shift();    // Removing the used request
```

- If queues cannot be used as a semaphore, one can protect their usage with a yes-no semaphore.

# Parallelism and Concurrency

- Parallel Computing
- Parallel Sorting
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- Nick's Class
- Coordinating Parallelism
- **New Models of Computing**

# DNA-Computing

- Goal: Solve NP-decision problems like satisfiability by exhaustive search. Formula given as set of clauses.
- Idea: Do search on molecular basis.
- Representation: DNA-molecules can represent long arrays over a finite alphabet. Each entry represents a Boolean variable.
- Search Algorithm for  $n$  variables:  
Generate molecules representing all combinations of truth-values for  $n$  variables.  
For each clause, select all molecules which satisfy one literal in the clause by some method from molecular biology.  
If at the end some molecules have passed all selection steps then the NP problem is solvable.  
A solution can be reconstructed from any remaining molecule.

# Example

Consider equations  $x_1 \vee x_2$ ,  $x_1 \vee x_3 \vee \neg x_4$ ,  $x_2 \vee x_3 \vee x_4$ .

Select eight molecular patterns:  $A_k$  for  $x_k$ ,  $B_k$  for  $\neg x_k$ .

Generate long molecules covering all possible combinations of patterns.

Remove all molecules having patterns  $A_1$  and  $B_1$  for  $\neg(x_1 \wedge \neg x_1)$ .

Remove all molecules having patterns  $A_2$  and  $B_2$  for  $\neg(x_2 \wedge \neg x_2)$ .

Remove all molecules having patterns  $A_3$  and  $B_3$  for  $\neg(x_3 \wedge \neg x_3)$ .

Remove all molecules having patterns  $A_4$  and  $B_4$  for  $\neg(x_4 \wedge \neg x_4)$ .

Select all molecules having patterns  $A_1$  or  $A_2$  for  $x_1 \vee x_2$ .

Select all molecules having patterns  $A_1$  or  $A_3$  or  $B_4$  for  $x_1 \vee x_3 \vee \neg x_4$ .

Select all molecules having patterns  $A_2$  or  $A_3$  or  $A_4$  for  $x_2 \vee x_3 \vee x_4$ .

Test whether molecule in remaining solution.

A molecule with patterns  $B_1$  and  $A_2$  and  $A_3$  and  $B_4$  could be found.

Problem is solvable:  $\neg x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4$ .

# Evaluation

- There are less than  $10^{50}$  atoms on earth.
- Even if one uses all atoms on earth, exhaustive search could only solve satisfiability with less than 167 variables ( $2^{167} > 10^{50}$ ).
- More realistic bounds use facts that only a tiny fraction of the atoms on the earth can be used, that DNA-molecules need hundreds of atoms and so on.
- Thus, satisfiability with up to 70 variables might be solvable by exhaustive search with a DNA-computer.
- The same can be obtained by a highly optimized algorithm on sequential computers.
- So DNA-computing has some applications and might become a good storage medium, but it does not make NP feasible.

# The Practical Side

- Workgroup on “Computation with Molecules” in Dresden has a patent for solving NP-problems with DNA computing.
- Better DNA-algorithms might cut down on exhaustive search, so one might eventually break the theoretical 70 variable boundary, but this is still far away from what is actually done.
- Leonard Adleman proposed the model of DNA-computing and did also some experiments in the laboratory. The first ones were done around 1995.
- In the year 2002, Adleman, Braich, Chelyapov, Johnson and Rothemund conducted an experiment which solved instances of the Satisfiability problem with 20 variables by working with DNA-molecules.
- The same problem can be solved fast on a normal computer, one just has to test 1048576 possible solutions.
- Many genetic technologies are today available, but DNA-computing is still not beyond the level of experiments for publishing papers.

# Quantum-Computing

- Superposition of Quantum-States permits to run certain searches on  $n$  qubits in a search-space of  $2^n$  possibilities.
- Shor found a fast prime-factoring algorithm for Quantum-Computers: Given a number  $x$ , list its prime-factors.
- There are no fast classical algorithm, only fast primality test: Given a number  $x$ , decide whether it is prime or not.
- Somehow, by 2003 only quantum-computers with 7 qubits were built; it might be that it is technically impossible to build them with an interesting number of qubits.
- Nevertheless, due to the bit different model for tractable computing, Quantum-Computing and Quantum-Algorithms are theoretically interesting.
- It seems to be unlikely that Quantum-Computers, if ever built, would be able to solve all NP decision problems fast.

# Summary

- Parallel Computing
- Parallel Sorting
- Weighted Averages
- Sequential Polynomial Space, Parallel Polynomial Time
- Nick's Class
- Coordinating Parallelism
- New Models of Computing

# Next Week: Probabilistic Algorithms

- Comparing Polynomials
- The Class RP
- Fast Probabilistic Pattern Matching
- Introduction to Cryptography
- Public Key Cryptography
- Interactive Proof Systems