Parallel Computation

Karousatou Christina

Computational Complexity

January 18, 2010

Layout

- A parallel computer has a large number of independent processors.
- Each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory.

Layout

- Given two $n \times n$ matrices A and B, we wish to compute their product $C = A \cdot B$. In fact, we wish to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n$$

- Given two $n \times n$ matrices A and B, we wish to compute their product $C = A \cdot B$. In fact, we wish to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n$$

- Sequentially this problem can be solved in $\mathcal{O}(n^3)$ arithmetic operations, in the obvious way.

- Given two $n \times n$ matrices A and B, we wish to compute their product $C = A \cdot B$. In fact, we wish to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n$$

- Sequentially this problem can be solved in $\mathcal{O}(n^3)$ arithmetic operations, in the obvious way.
- One satisfactory parallel algorithm for this problem is computing all the n^3 products $A_{ik} \cdot B_{kj}$ by different processors independently, assuming that we have n^3 processors.

- Given two $n \times n$ matrices A and B, we wish to compute their product $C = A \cdot B$. In fact, we wish to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n$$

- Sequentially this problem can be solved in $\mathcal{O}(n^3)$ arithmetic operations, in the obvious way.
- One satisfactory parallel algorithm for this problem is computing all the n^3 products $A_{ik} \cdot B_{kj}$ by different processors independently, assuming that we have n^3 processors.
- The total time required is n arithmetic operations on n^3 processors. This algorithm has brought down the complexity from n^3 to n, which is a significant reduction, but it is not the improvement that would make multiprocessors worth building. What we want to see is some exponential drop in the time required, or at least polylogarithmic parallel time like $\log^3 n$.

- One way to achieve that decrement is to organize the processors to perform a binary tree of additions. This way, we can add the results in only *logn* parallel steps, and have the exponential drop that we hoped to have.

- One way to achieve that decrement is to organize the processors to perform a binary tree of additions. This way, we can add the results in only *logn* parallel steps, and have the exponential drop that we hoped to have.
- Another important factor, we haven't counted, is that we must have a polynomial number of processors, because an exponential number of processors is even less feasible than exponential sequential time.

- One way to achieve that decrement is to organize the processors to perform a binary tree of additions. This way, we can add the results in only *logn* parallel steps, and have the exponential drop that we hoped to have.
- Another important factor, we haven't counted, is that we must have a polynomial number of processors, because an exponential number of processors is even less feasible than exponential sequential time.
- In order to find out the optimum number of processors we can use to achieve the optimum parallel time, there is a general principle that we must keep in mind:
  The amount of work done by a parallel algorithm can be no smaller than the time complexity of the best sequential algorithm (by "amount of work" we mean the steps executed by each processor, summed over all processors).

- Back to our example. Any algorithm that does work at least n^3 and achieves the optimum parallel time $logn$ requires at least $\frac{n^3}{logn}$ processors.

- Back to our example. Any algorithm that does work at least n^3 and achieves the optimum parallel time *logn* requires at least $\frac{n^3}{logn}$ processors.
- Question: How can we decrease our processor requirement from n^3 to the optimum $\frac{n^3}{logn}$ without increasing the parallel time too much?

- Back to our example. Any algorithm that does work at least n^3 and achieves the optimum parallel time *logn* requires at least $\frac{n^3}{logn}$ processors.
- Question: How can we decrease our processor requirement from n^3 to the optimum $\frac{n^3}{logn}$ without increasing the parallel time too much?
- Answer: We compute the n^3 products not in a single step, but rather in *logn* "shifts" using $\lceil \frac{n^3}{logn} \rceil$ processors at each shift. We use shifts of the same $\lceil \frac{n^3}{logn} \rceil$ processors to compute the first *loglogn* parallel addition steps. The total number of parallel steps is now no more than $2logn$, with $\frac{n^3}{logn}$ processors, thus rendering our parallelization of the $\mathcal{O}(n^3)$ sequential algorithm optimal in all respects.

- The technique, we just used, of bringing down the processor requirement to the optimum value (for given work and parallel time) by using "shifts of processors" is quite general and valuable and is known as Brent's Principle.

- The technique, we just used, of bringing down the processor requirement to the optimum value (for given work and parallel time) by using "shifts of processors" is quite general and valuable and is known as Brent's Principle.
- Expressing processor requirements as a function of n may seem bizarre, because any parallel machine has a given and fixed number of processors. But once we have an algorithm that achieves optimal parallel time using as many processors as it takes, we can now scale back our algorithm to the available hardware. In our previous example if we had P available processors, we could organize them so that they execute each parallel step of our algorithm in $\lceil \frac{n^3/logn}{P} \rceil$ shifts, where each shift employs P processors. The total time is $\frac{2n^3}{P}$, which is obviously the fastest that this algorithm can be parallelized on P processors.

Layout

- The search algorithm we have seen for REACHABILITY cannot be parallelized in any obvious way. Thus, we have to try a different approach to the solution.

- The search algorithm we have seen for REACHABILITY cannot be parallelized in any obvious way. Thus, we have to try a different approach to the solution.
- One solution is using Matrix multiplication.

- The search algorithm we have seen for REACHABILITY cannot be parallelized in any obvious way. Thus, we have to try a different approach to the solution.
- One solution is using Matrix multiplication.
- Suppose that A is the adjacency matrix of the graph, where all self-loops are added : $A_{ii} = 1$ for all i. We compute now the Boolean product of A with it self $A^2 = A \cdot A$, where

$$A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$$

- The search algorithm we have seen for REACHABILITY cannot be parallelized in any obvious way. Thus, we have to try a different approach to the solution.
- One solution is using Matrix multiplication.
- Suppose that A is the adjacency matrix of the graph, where all self-loops are added : $A_{ii} = 1$ for all i. We compute now the Boolean product of A with it self $A^2 = A \cdot A$, where

$$A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$$

- A hint to the solution is that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j.

- This way, computing A^4, we get all paths of length 4 or less, and so on. After $\lceil logn \rceil$ Boolean matrix multiplications, we get $A^{2^{\lceil logn \rceil}}$, which is the adjacency matrix of the transitive closure of A. That is, the concentrated answers to all possible REACHABILITY instances on the given graph.

- This way, computing A^4, we get all paths of length 4 or less, and so on. After $\lceil logn \rceil$ Boolean matrix multiplications, we get $A^{2^{\lceil logn \rceil}}$, which is the adjacency matrix of the transitive closure of A. That is, the concentrated answers to all possible REACHABILITY instances on the given graph.
- We achieved computing the transitive closure of a graph in $\mathcal{O}(log^2 n)$ parallel steps with $\mathcal{O}(n^3 logn)$ total work.
  Exactly what we wanted, polylogarithmic parallel time and polynomial amount of total work.

Layout

- One problem of Arithmetic Operations is the prefix sums problem. Given n integers $x_1, \ldots, x_n$, we wish to compute all sums of the form $\sum_{i=1}^{j} x_i, \quad j = 1, \ldots, n$.

- One problem of Arithmetic Operations is the prefix sums problem. Given n integers $x_1, \ldots, x_n$, we wish to compute all sums of the form $\sum_{i=1}^{j} x_i, \quad j = 1, \ldots, n$.
- Sequentially is solved with $n-1$ additions.

- One problem of Arithmetic Operations is the prefix sums problem. Given n integers $x_1, \dots, x_n$, we wish to compute all sums of the form $\sum_{i=1}^{j} x_i, \quad j = 1, \dots, n$.
- Sequentially is solved with $n - 1$ additions.
- In parallel we use, once more, a different approach to the solution. Assuming that n is a power of 2 , we first compute the sums $(x_1 + x_2), (x_3 + x_4), \dots, (x_{n-1} + x_n)$ (one parallel step), and then we recursively compute the prefix sums of this sequence. It follows that the total number of parallel steps is $2logn$ and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \dots \leq 2n$. And by Brent's principle, the number of processors needed is only $\frac{n}{logn}$.

In other arithmetic operations we conclude that

- We can compute the sum of two n-bit binary integers in $\mathcal{O}(logn)$ parallel time and $\mathcal{O}(n)$ work.
- We can multiply two n-bit binary integers in $\mathcal{O}(logn)$ parallel time and $\mathcal{O}(n^2 logn)$ work.

Layout

- The sequential algorithm we have seen for the MAX FLOW problem works in stages. At each stage, we start with a flow f, and try to improve it. To this end, we construct a new network $N(f)$, reflecting the improvement potential of the arcs of N with respect to f, and try to find a path from the source s to the sink t in $N(f)$.If we succeed, we improve the flow. If we fail, the current flow is maximum.

- The sequential algorithm we have seen for the MAX FLOW problem works in stages. At each stage, we start with a flow f, and try to improve it. To this end, we construct a new network $N(f)$, reflecting the improvement potential of the arcs of N with respect to f, and try to find a path from the source s to the sink t in $N(f)$.If we succeed, we improve the flow. If we fail, the current flow is maximum.
- In parallel, with enough hardware, we can construct $N(f)$ in a single parallel step. Previously, we learned how to find paths in parallel quickly. Thus, each stage can be done in $\mathcal{O}(log^2 n)$ parallel time and $\mathcal{O}(n^2)$ total work, where n is the number of nodes in the network.

- The sequential algorithm we have seen for the MAX FLOW problem works in stages. At each stage, we start with a flow f, and try to improve it. To this end, we construct a new network $N(f)$, reflecting the improvement potential of the arcs of N with respect to f, and try to find a path from the source s to the sink t in $N(f)$.If we succeed, we improve the flow. If we fail, the current flow is maximum.
- In parallel, with enough hardware, we can construct $N(f)$ in a single parallel step. Previously, we learned how to find paths in parallel quickly. Thus, each stage can be done in $\mathcal{O}(log^2 n)$ parallel time and $\mathcal{O}(n^2)$ total work, where n is the number of nodes in the network.
- The problem that comes up, is that stages need to be carried out one after the other, and the number of stages may be very large (certainly more than polylogarithmic n).

- We may try to develop alternative approaches to the problem, as we have done before, and reduce the number of stages to n, or even $\sqrt{n}$, but still not to a polylogarithmic number.

- We may try to develop alternative approaches to the problem, as we have done before, and reduce the number of stages to n, or even $\sqrt{n}$, but still not to a polylogarithmic number.
- With this problem we saw that parallelism has limitations and that MAX FLOW is a polynomial-time solvable problem that seems to be inherently sequential.

Layout

According to our last problem we can make a conclusion that parallel computation is not the answer to **NP**-completeness. And furthermore, is not the technological breakthrough that could make exponential algorithms feasible. The obstacle is the equation

$$\text{work} = \text{parallel time} \times \text{number of processors}$$

If the fastest sequential algorithm that we know for a problem requires exponential time, then in any parallel algorithm either the parallel time must be exponential, or the number of processors must be exponential (or both).
The final conclusion is not that parallel computers are useless in solving hard problems. Parallel computation, along with clever exponential algorithms, fast processors, and smart techniques do help solve exactly larger and larger instances of **NP**-complete problems. The point is that as we said before parallelism is not the absolute solution of **NP**-completeness and exponentiality.

Layout

Although the problem of computing the determinant of an integer matrix may seem inherently sequential, as the problem of MAX FLOW, there is a rather sophisticated approach which succeeds in providing a fast parallel algorithm.
This alternative approach solves this problem by merging it with another difficult problem, matrix inversion, and then solving both.
The result of this algorithm is that we can compute the determinant of an $n \times n$ matrix with b-bit integer entries in parallel time $\mathcal{O}(log^3 n(logn + logb))$, and $\mathcal{O}(n^8 b^2)$ total work. Although unrealistically large, these bounds still conform to our theoretical requirements of polylogarithmic parallel time and polynomial work in the size of the input.

Layout

Boolean circuits will be our basic model of parallel algorithms. We consider families of Boolean circuits with one different circuit for each input size.

Boolean circuits will be our basic model of parallel algorithms. We consider families of Boolean circuits with one different circuit for each input size.

- A circuit family is a sequence $\mathcal{C} = (C_0, C_1, \ldots)$ of Boolean circuits, such that C_i has i inputs.

Boolean circuits will be our basic model of parallel algorithms. We consider families of Boolean circuits with one different circuit for each input size.

- A circuit family is a sequence $\mathcal{C} = (C_0, C_1, \ldots)$ of Boolean circuits, such that C_i has i inputs.
- Additionally, we only consider families of circuits that are uniform. That is, there is a logarithmic space-bounded Turing machine which on input 1^n outputs C_n (intuitively, this implies that all the circuits in the family represent the same algorithm).

Definition 1

Let C be a Boolean circuit, that is, a directed acyclic graph where each node is a gate, of one of the possible sorts and matching indegree.(C could have more than one output, in which case it computes a function from $\{0,1\}^n$ to $\{0,1\}^m$, not a predicate). The size of C is, as always, the total number of gates in it. The depth of C is the number of nodes in the longest path in C.
Let now $\mathcal{C} = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to integers. We say that the parallel time of $\mathcal{C}$ is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. We say that the total work of $\mathcal{C}$ is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$.

Definition 1 (continued)

Finally define $\mathbf{PT}/\mathbf{WK}(f(n), g(n))$ to be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a uniform family of circuits $\mathcal{C}$ deciding L with $\mathcal{O}(f(n))$ parallel time and $\mathcal{O}(g(n))$ work. $\square$

Example 1

In the previous section we saw an algorithm for computing the transitive closure.This algorithm can be easily rendered as a uniform family of circuits as follows: First, consider, for each n, a cirquit Q with n^2 inputs and n^2 outputs, and such that the output Boolean matrix is the square of the input one. Now the circuit of the transitive closure is simply the composition of $\lceil logn \rceil$ copies of Q, connected in tandem so that the outputs of one coincide with the inputs of the next. We call the resulting family $\mathcal{C}_2$.

The uniform family of circuits $\mathcal{C}_1$ for REACHABILITY shows that REACHABILITY $\in \mathbf{PT}/\mathbf{WK}(n, n^3)$. On the other hand the family $\mathcal{C}_2$ establishes that REACHABILITY $\in \mathbf{PT}/\mathbf{WK}(log^2 n, n^3 logn)$. $\square$

Layout

RAM program (Random Access Machine):

- Finite sequence $\Pi = (\pi_1, \ldots, \pi_m)$ of instructions (READ, ADD, LOAD, JUMP, etc.).
- Input registers $I = (i_1, \ldots, i_m)$.
- Register 0 the accumulator of the RAM.
- Program counter κ, pointing the instruction to be executed.

PRAM program (Parallel Random Access Machine):
Sequence of RAM programs $P = (\Pi_1, \Pi_2, \ldots, \Pi_q)$.

- Each machine executes its own program.
- Has its own program counter.
- Has its own accumulator (accumulator of RAM i is Register i).
- Each RAM can read and write the accumulators of the other RAMs.

PRAM program (Parallel Random Access Machine):
Sequence of RAM programs $P = (\Pi_1, \Pi_2, \ldots, \Pi_q)$.

- Each machine executes its own program.
- Has its own program counter.
- Has its own accumulator (accumulator of RAM i is Register i).
- Each RAM can read and write the accumulators of the other RAMs.

The number q of RAMs is actually a function $q(m, n)$, where m is the number of integers in the input I, and $n = \ell(I)$ is the total length of these integers.

Thus, for each value of m and n we have a different PRAM program $P_{m,n}$, each with a diferrent number of RAMs $q(m, n)$, consisting a two-dimensional family $\mathcal{P} = (P_{m,n} : m, n \geq 0)$. Where we consider that these families are uniform.

Thus, for each value of m and n we have a different PRAM program $P_{m,n}$, each with a diferrent number of RAMs $q(m,n)$, consisting a two-dimensional family $\mathcal{P} = (P_{m,n} : m, n \geq 0)$. Where we consider that these families are uniform.

A configuration of the PRAM $P_{m,n}$ is a tuple $(\kappa_1, \kappa_2, \ldots, \kappa_{q(m,n)}, R)$, which now contains all the program counters, together with R, a description of the current contents of the registers.

Definition 2

Suppose that F is a function mapping finite sequences of integers to finite sequences of integers; let $\mathcal{P} = (P_{m,n} : m, n \geq 0)$, be a uniform family of PRAM programs; and let f and g be functions from positive integers to positive integers. We say that $\mathcal{P}$ computes F in parallel time f with g processors if for each $m, n \geq 0$ $P_{m,n}$ has the following property:

Definition 2

Suppose that F is a function mapping finite sequences of integers to finite sequences of integers; let $\mathcal{P} = (P_{m,n} : m, n \geq 0)$, be a uniform family of PRAM programs; and let f and g be functions from positive integers to positive integers. We say that $\mathcal{P}$ computes F in parallel time f with g processors if for each $m, n \geq 0$ $P_{m,n}$ has the following property:
First, it has $q(m, n) \leq g(n)$ processors. Second, if the PRAM program is executed on input $I = (i_1, \ldots, i_m)$ of m integers with total number of bits $\ell(I) = n$, then all $q(m, n)$ RAMs have reached a HALT instruction after at most $f(n)$ steps, at which point the $k \leq q(m, n)$ first registers contain the output $F(i_1, i_2, \ldots, i_m) = (o_1, \ldots, o_k)$. $\square$

Theorem 1

If $L \subseteq \{0,1\}^*$ is in $\mathbf{PT}/\mathbf{WK}(f(n), g(n))$, then there is a uniform PRAM that computes the corresponding function F_L mapping $\{0,1\}^*$ to $\{0,1\}$ in parallel time $\mathcal{O}(f(n))$ using $\mathcal{O}(\frac{g(n)}{f(n)})$ processors.

Theorem 1

If $L \subseteq \{0,1\}^*$ is in $\mathbf{PT}/\mathbf{WK}(f(n), g(n))$, then there is a uniform PRAM that computes the corresponding function F_L mapping $\{0,1\}^*$ to $\{0,1\}$ in parallel time $\mathcal{O}(f(n))$ using $\mathcal{O}(\frac{g(n)}{f(n)})$ processors.

Proof:
Using the logarithmic-space machine that generates the nth circuit C_n, we will generate equivalent RAM programs. For each gate g_i of C_n we have diferrent RAM Π_i.

The program of Π_i:
First, waits for $3d$ steps, where d is the length of the longest path from any input gate to g_i.

The program of Π_i:
First, waits for $3d$ steps, where d is the length of the longest path from any input gate to g_i.
Next, Π_i in three steps computes the value of g_i and stores it in its accumulator, Register i. If g_i is an AND gate with inputs g_j and g_k, then Π_i executes the following RAM program :

The program of Π_i:
First, waits for $3d$ steps, where d is the length of the longest path from any input gate to g_i.
Next, Π_i in three steps computes the value of g_i and stores it in its accumulator, Register i. If g_i is an AND gate with inputs g_j and g_k, then Π_i executes the following RAM program :

```
$3d+1$. LOAD $j$
$3d+2$. JZERO $3d+5$
$3d+3$. LOAD $k$
$3d+4$. JUMP $3d+6$
$3d+5$. LOAD $=0$
$3d+6$. HALT
```

The program of Π_i:
First, waits for $3d$ steps, where d is the length of the longest path from any input gate to g_i.
Next, Π_i in three steps computes the value of g_i and stores it in its accumulator, Register i. If g_i is an AND gate with inputs g_j and g_k, then Π_i executes the following RAM program :

```
3d + 1. LOAD j
3d + 2. JZERO 3d + 5
3d + 3. LOAD k
3d + 4. JUMP 3d + 6
3d + 5. LOAD = 0
3d + 6. HALT
```

Similarly for OR and NOT gates.

By induction on d, is proved that, after executing these instructions, Register i will contain the correct value of gate i. The output gate must be g_1 so that the final answer is left on Register 1.

By induction on d, is proved that, after executing these instructions, Register i will contain the correct value of gate i. The output gate must be g_1 so that the final answer is left on Register 1.
For a better number of processors, we employ Brent's principle. We first compute $q(n) = \lceil \frac{g(n)}{f(n)} \rceil$. For each value of d we make a list of the gates for which d is the length of the longest path from any input gate. We then assign these gates to the $q(n)$ processors as equitably as possible. $\square$

Theorem 2

Suppose that a function F can be computed by a uniform PRAM in parallel time $f(n)$ with $g(n)$ processors, where $f(n)$ and $g(n)$ can be computed from 1^n in logarithmic space. Then there is a uniform family of circuits of depth $\mathcal{O}(\log f(n) + \log n)$ and size $\mathcal{O}(g(n)(n^k f(n) + g(n))(\log f(n) + \log n))$ which computes the binary representation of F, where n^k is the time bound of the logarithmic-space Turing machine which on input 1^n outputs the nth PRAM in the family.

Proof: For fixed size n of the binary representation there are at most $g(n)$ processors in the corresponding PRAM.

Proof: For fixed size n of the binary representation there are at most $g(n)$ processors in the corresponding PRAM.
PRAM's registers contain integers of length bounded by $\ell(n) = n + f(n) + b$, where b is the length of the longest integer referred to in an instruction of the program, at most n^k.

Proof: For fixed size n of the binary representation there are at most $g(n)$ processors in the corresponding PRAM.
PRAM's registers contain integers of length bounded by $\ell(n) = n + f(n) + b$, where b is the length of the longest integer referred to in an instruction of the program, at most n^k.
The number of instructions in each RAM program is bounded by a polynomial n.

Proof: For fixed size n of the binary representation there are at most $g(n)$ processors in the corresponding PRAM.
PRAM's registers contain integers of length bounded by
$\ell(n) = n + f(n) + b$, where b is the length of the longest integer referred to in an instruction of the program, at most n^k.
The number of instructions in each RAM program is bounded by a polynomial n.
At most $f(n)g(n)$ registers will be affected during the computation.

Hence, the configuration $C = (\kappa_1, \kappa_2, \ldots, \kappa_{q(m,n)}, R)$ can be encoded in $\mathcal{O}(g(n)f(n)logn)$ bits. R encodes the contents of the memory, given as pairs of the form (location, contents).

Hence, the configuration $C = (\kappa_1, \kappa_2, \ldots, \kappa_{q(m,n)}, R)$ can be encoded in $\mathcal{O}(g(n)f(n)logn)$ bits. R encodes the contents of the memory, given as pairs of the form (location, contents).
All integers are in binary.

Hence, the configuration $C = (\kappa_1, \kappa_2, \dots, \kappa_{q(m,n)}, R)$ can be encoded in $\mathcal{O}(g(n)f(n)logn)$ bits. R encodes the contents of the memory, given as pairs of the form (location, contents).
All integers are in binary.
Thus, C is a sequence of bits, where it is *a priori* known which bit correspond to the ith program counter, and which bits encode the rth location-contents pair in R.

We can now compute the encoding of the next configuration from that of the current configuration. We will show in example.

We can now compute the encoding of the next configuration from that of the current configuration. We will show in example.
Suppose we know that the current instruction of RAM i is "t: ADD j". As said before, we know the precise bits in the encoding of the configuration where the contents of Registers i and j are encoded.

We can now compute the encoding of the next configuration from that of the current configuration. We will show in example.
Suppose we know that the current instruction of RAM i is "t: ADD j". As said before, we know the precise bits in the encoding of the configuration where the contents of Registers i and j are encoded.
We get the following algorithm, for each $r \leq f(n)g(n)$:

"If program counter κ_i is t, and if the rth pair in the encoding R of the register contents is of the form (j, x) then Register i is incremented by x."

This algorithm can be implemented easily by circuits of depth $\log \ell$ and size $\mathcal{O}(\ell)$.

This algorithm can be implemented easily by circuits of depth $\log \ell$ and size $\mathcal{O}(\ell)$.
We must have such a circuit for each instruction in each RAM program, end for each (location, contents) pair, a total of $\mathcal{O}(n^k f(n) g(n))$ circuits.

This algorithm can be implemented easily by circuits of depth $\log \ell$ and size $\mathcal{O}(\ell)$.
We must have such a circuit for each instruction in each RAM program, end for each (location, contents) pair, a total of $\mathcal{O}(n^k f(n) g(n))$ circuits.
There are also, $g(n)^2$ integer comparisons, each of $\log \ell(n)$ depth, we have to do, because of the problem that comes up when more than one RAMs try to write in the same Register i.

We conclude that there is a circuit of depth $\mathcal{O}(\log \ell) = \mathcal{O}(\log f(n) + \log n)$ and of size $\mathcal{O}(g(n)(n^k f(n) + g(n))(\log f(n) + \log n))$, which given the encoding of a PRAM configuration, computes the encoding of the next one. $\square$

Layout

We define

$$\mathbf{NC} = \mathbf{PT/WK}(\log^k n, n^k)$$

We define

$$\mathbf{NC} = \mathbf{PT/WK}(\log^k n, n^k)$$

- **NC** is the class of languages decided by PRAMs in polylogarithmic parallel time with polynomially many processors.

We define

$$\mathbf{NC} = \mathbf{PT/WK}(\log^k n, n^k)$$

- **NC** is the class of languages decided by PRAMs in polylogarithmic parallel time with polynomially many processors.
- **NC** is closed under reductions.

Is **NC** the parallel version of **P** (problems satisfactorily solved by parallel computers)?

Is **NC** the parallel version of **P** (problems satisfactorily solved by parallel computers)?

- In sequential computation the diferrence between polynomial and exponential algorithms is real and obvious (2^n is much larger than n^3 for accesible values, say $n = 20$).

Is **NC** the parallel version of **P** (problems satisfactorily solved by parallel computers)?

- In sequential computation the diferrence between polynomial and exponential algorithms is real and obvious (2^n is much larger than n^3 for accesible values, say $n = 20$).
- In parallel, although $\log^3 n$ is in theory asymptotically much smaller than $\sqrt{n}$, the difference starts to become felt only when $n = 10^{12}$.

Is **NC** the parallel version of **P** (problems satisfactorily solved by parallel computers)?

- In sequential computation the diferrence between polynomial and exponential algorithms is real and obvious (2^n is much larger than n^3 for accesible values, say $n = 20$).
- In parallel, although $\log^3 n$ is in theory asymptotically much smaller than $\sqrt{n}$, the difference starts to become felt only when $n = 10^{12}$.
- Furthermore, we defined **NC** to be a class of languages. Which is a problem, since in parallel computation the more interesting problems require substantial output.

We define now, the family of important subclasses of **NC**:

$$\mathbf{NC}_j = \mathbf{PT/WK}(\log^j n, n^k)$$

That is, $\mathbf{NC}_j$ is the subset of **NC** in which the parallel time is $\mathcal{O}(\log^j n)$. The free parameter k means that we allow any degree in the polynomial accounting fot the total work.

It is clear that $\mathbf{NC} \subseteq \mathbf{P}$. But is $\mathbf{NC} = \mathbf{P}$?

It is clear that $\mathbf{NC} \subseteq \mathbf{P}$. But is $\mathbf{NC} = \mathbf{P}$?
This question is equivalent to the $\mathbf{P} \overset{?}{=} \mathbf{NP}$, in sequential computation.

It is clear that **NC** $\subseteq$ **P**. But is **NC** $=$ **P**?
This question is equivalent to the **P** $\overset{?}{=}$ **NP**, in sequential computation.

Intuition and experience suggest a negative answer. In addition, persistent failures to develop **NC** algorithms for some fairly simple problems in **P** seem to imply that there are problems that are *inherently sequential* so, **NC** $\neq$ **P**.

Layout

Theorem 3

If L reduces to $L' \in \mathbf{NC}$, then $L \in \mathbf{NC}$.

Theorem 3

If L reduces to $L' \in \mathbf{NC}$, then $L \in \mathbf{NC}$.

Proof:
Let R be the logarithmic reduction from L to L'. There is a logarithmic space-bounded Turing machine R' which accepts input (x, i) if and only if the ith bit of $R(x)$ is one.

Theorem 3

If L reduces to $L' \in \mathbf{NC}$, then $L \in \mathbf{NC}$.

Proof:
Let R be the logarithmic reduction from L to L'. There is a logarithmic space-bounded Turing machine R' which accepts input (x, i) if and only if the ith bit of $R(x)$ is one.
We solve the reachability problem for the configuration graph of R' on input (x, i), in parallel by $\mathbf{NC}_2$ circuits, to compute all bits of $R(x)$.

Theorem 3

If L reduces to $L' \in \mathbf{NC}$, then $L \in \mathbf{NC}$.

Proof:
Let R be the logarithmic reduction from L to L'. There is a logarithmic space-bounded Turing machine R' which accepts input (x, i) if and only if the ith bit of $R(x)$ is one.
We solve the reachability problem for the configuration graph of R' on input (x, i), in parallel by $\mathbf{NC}_2$ circuits, to compute all bits of $R(x)$.
Once we have $R(x)$ we can use the $\mathbf{NC}$ circuit for L' to tell wether $x \in L$, all in $\mathbf{NC}$. $\square$

Corollary

If L reduces to $L' \in \mathbf{NC}_j$, where $j \geq 2$, then $L \in \mathbf{NC}_j$.

Corollary

If L reduces to $L' \in \mathbf{NC}_j$, where $j \geq 2$, then $L \in \mathbf{NC}_j$.

We have seen that computing the maximum flow in a network, is a task that seems to be inherently sequential.

Corollary

If L reduces to $L' \in \mathbf{NC}_j$, where $j \geq 2$, then $L \in \mathbf{NC}_j$.

We have seen that computing the maximum flow in a network, is a task that seems to be inherently sequential.

Theorem 4

ODD MAX FLOW is **P**-complete.

Layout

RNC is the randomized version of **NC**

RNC is the randomized version of **NC**

Definition 3

A language L is in **RNC** if there is a uniform family of **NC** circuits, with the following additional properties: First, the circuit C_n specializing in strings of length n has now $n + m(n)$ input gates, where $m(n)$ is a polynomial. If a string x of length n is in L, then at least half of the $2^{m(n)}$ bit strings y of length $m(n)$ the circuit C_n outputs **true** on input $x; y$. If $x \notin L$, C_n outputs **false** on $x; y$ for all y. $\square$

So far, we have seen for decision problems, given an efficient algorithm that decides whether a solution exists, there is a general "dynamic programming technique" that actually computes the desired solution, if of course it exists.

So far, we have seen for decision problems, given an efficient algorithm that decides whether a solution exists, there is a general “dynamic programming technique” that actually computes the desired solution, if of course it exists.
The catch is that this method is *inherently sequential*.

So far, we have seen for decision problems, given an efficient algorithm that decides whether a solution exists, there is a general "dynamic programming technique" that actually computes the desired solution, if of course it exists.
The catch is that this method is *inherently sequential*.
Fortunately, there is a clever trick that works for the matching problem.
Which is best explained in terms of a more general problem, the minimum-weight perfect matching problem.

Suppose that each edge $(u_i, v_j) \in E$ has a weight w_{ij} associated with it.
We are seeking the perfect matching π that minimizes
$w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$.
There is an **NC** algorithm for this problem, which works under two conditions:

Suppose that each edge $(u_i, v_j) \in E$ has a weight w_{ij} associated with it.
We are seeking the perfect matching π that minimizes
$w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$.
There is an **NC** algorithm for this problem, which works under two conditions:

- The weights must be small, polynomial in n.

Suppose that each edge $(u_i, v_j) \in E$ has a weight w_{ij} associated with it. We are seeking the perfect matching π that minimizes $w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$.
There is an **NC** algorithm for this problem, which works under two conditions:

- The weights must be small, polynomial in n.
- The minimum-weight matching must be unique.

We define a matrix $A^{G,w}$ whose i,jth element is $2^{w_{ij}}$ if (u_i, u_j) is an edge and 0 otherwise. We know that the determinant of $A^{G,w}$ is computed by

$$detA^{G,w} = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} A^{G,w}_{i,\pi(i)}$$

The summation ranges over all perfect matchings. Also, $detA^{G,w}$ is a sum of powers of two, where the exponents are the weights of the perfect matchings.

We have assumed that the minimum-weight perfect matching is unique, suppose its weight is w^*.
We get that $detA^{G,w} = 2^{w^*}(1+2k)$ for some integer k.
Thus 2^{w^*} is the highest power of two that divides $detA^{G,w}$.
Based on this fact, we can compute w^* by calculating $detA^{G,w}$, and then counting the trailing zeros in the binary representation of $detA^{G,w}$, which is w^*.

Once we have w^*, we test whether an edge (u_i, v_j) is in the minimum-weight perfect matching by deleting this edge and its nodes from G and computing the minimum-weight in the resulting graph.

Edge (u_i, v_j) is in the minimum-weight matching of G if and only if the new minimum-weight is precisely $w^* - w_{ij}$. We test all edges in parallel.

Conclusion, if the minimum-weight perfect match exists and is unique, then it can be computed efficiently in parallel.

Conclusion, if the minimum-weight perfect match exists and is unique, then it can be computed efficiently in parallel.

Is the minimum-weight matching unique?

Conclusion, if the minimum-weight perfect match exists and is unique, then it can be computed efficiently in parallel.

Is the minimum-weight matching unique?

Lemma 1 (The Isolating Lemma)

Suppose that the edges in E are assigned independently and randomly weights between 1 and $2|E|$. If a perfect matching exists, then with probability at least $\frac{1}{2}$ the minimum-weight perfect matching is unique.

Proof:
Call an edge bad if it is on one minimum-weight matching but not in another.

Proof:
Call an edge bad if it is on one minimum-weight matching but not in another.
Consider an edge $e = (u_i, v_j)$, and suppose that all weights have been assigned except for e's.
$w^*[\bar{e}]$ is the smallest weight among all perfect matchings that do not contain e.
$w^*[e]$ is the smallest weight among all perfect matchings that contain e, but not counting the weight of e.
Define $\Delta = w^*[\bar{e}] - w^*[e]$.

Proof:
Call an edge bad if it is on one minimum-weight matching but not in another.
Consider an edge $e = (u_i, v_j)$, and suppose that all weights have been assigned except for e's.
$w^*[\bar{e}]$ is the smallest weight among all perfect matchings that do not contain e.
$w^*[e]$ is the smallest weight among all perfect matchings that contain e, but not counting the weight of e.
Define $\Delta = w^*[\bar{e}] - w^*[e]$.
We next draw the weight w_{ij} of e.

- If $w_{ij} < \Delta$ then e is in every minimum-weight perfect matching.

- If $w_{ij} < \Delta$ then e is in every minimum-weight perfect matching.
- If $w_{ij} > \Delta$ then e is included in no minimum-weight matching.

- If $w_{ij} < \Delta$ then e is in every minimum-weight perfect matching.
- If $w_{ij} > \Delta$ then e is included in no minimum-weight matching.
- If $w_{ij} = \Delta$ then e is bad.

- If $w_{ij} < \Delta$ then e is in every minimum-weight perfect matching.
- If $w_{ij} > \Delta$ then e is included in no minimum-weight matching.
- If $w_{ij} = \Delta$ then e is bad.

It follows that $\textbf{prob}[e \text{ is bad}] \leq \frac{1}{2|E|}$.
Therefore the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, and thus no more than half. $\square$

- If $w_{ij} < \Delta$ then e is in every minimum-weight perfect matching.
- If $w_{ij} > \Delta$ then e is included in no minimum-weight matching.
- If $w_{ij} = \Delta$ then e is bad.

It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$.
Therefore the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, and thus no more than half. $\square$

Back to our algorithm, if a perfect matching exists, with probability at least $\frac{1}{2}$ this algorithm will return one.

Layout

Finally, because matching problem is a special case of MAX FLOW with unit capacities, we conclude that there is an **RNC** algorithm for MAX FLOW when the capacities are expressed in unary. In fact, it leads to an **RNC**-approximation scheme.