Parallel Algorithm and Complexity - Samir Datta

Scribed: Soham Chatterjee

sohamchatterjee999@gmail.com
Website: sohamch08.github.io

2023

Contents

1 Addition of Two Numbers in Binary

Problem: ADD_{2n}
Input: Two n bit numbers $a = a_{n-1} \cdots a_1 a_0$ and $b = b_{n-1} \cdots b_1 b_0$
Output: $s = s_n \cdots s_1 s_0$ where $s \stackrel{\text{def}}{=} a + b$

1.1 Sequential (Ripple Carry)

For sum of any position i the two bits a_i, b_i and the carry generated by the previous position c_{i-1} is added. For the initial position we can set $c_0 = 0$. If we add two bits at most 2 bits is created. The right bit is called the sum bit and the left bit is the carry bit. $a_i + b_i + c_{i-1} = c_i s_i$. Then

$$s_i = a_i \oplus b_i \oplus c_{i-1} \text{ and } c_i = (a_i \wedge b_i) \vee (b_i \wedge c_{i-1}) \vee (c_{i-1} \wedge a_i)$$

Time Complexity: This algorithm takes $O(n)$ time complexity

1.2 Parallel (Carry Look Ahead Adder)

There is a carry that ripples into position i if and only if there is some position $j < i$ to the right where this carry is generated, and all positions in between propagate this carry. A carry is generated at position i if and only if both input bits a_i and b_i are on, and a carry is eliminated at position i, if and only if both input bits a_i and b_i are off. This leads to the following definitions:

For $0 \leqslant i < n$, let

$g_i = a_i \wedge b_i$	position i generates a carry
$p_i = a_i \vee b_i$	position i propagates a carry that ripples int o it

So we can set for $1 \leqslant i \leqslant n$

$$c_i = \bigvee_{j=0}^{i-1} \left(g_j \wedge \bigwedge_{k=j+1}^{i-1} \right)$$

Now the sumbits are calculated as before $s_i = a_i \oplus b_i \oplus c_{i-1}$ for $0 \leqslant i \leqslant n-1$ and $s_n = c_n$
Time Complexity: This algorithm takes $O(1)$ time complexity

Definition 1.1 (AC^0). *The class of circuits consists of the gates* $(\vee_n, \wedge_n, \neg_1)$ *(The subscript* n *or* 1 *denotes the fanin) of polynomial size and depth* $O(1) = O(\log^0 n)$

Theorem 1.1. $\mathrm{ADD}_{2n} = \mathrm{IterADD}_{2,n} \in \mathrm{AC}^0$

2 Iterated Addition

Problem: $\mathrm{IterADD}_{k,m}$
Input: k many m-bit numbers $a_1, \ldots, a_k$
Output: The sum of the input numbers

Definition 2.1 (Length Respecting). *Let* $f : \{0,1\}^* \to \{0,1\}^*$. f *is length respecting if for all* $x, y \in \{0,1\}^*$ $|f(x)| = |f(y)|$

Definition 2.2 (Constant Depth Reduction). *let* $f, g : \{0,1\}^* \to \{0,1\}^*$ *be length respecting. Then* f *is constant depth reducible to* g *or* $f \leqslant_{cd} g$ *if there is an unbounded fanin constant depth circuit computing* f *from the bits of* g.

2.1 Iterated Addition of Logarithmically many n-bit numbers

Theorem 2.1. $\text{IterADD}_{\log n,n} \leqslant_{cd} \text{IterADD}_{\log\log n,O(n)}$

Proof: We will denote $\log n = l$. We are given l many n-bit numbers $a_1, \ldots, a_l$, where $\text{bin}(a_i) = a_{i,n-1} \ldots a_{i,1} a_{i,0}$. We add all the l many bits at ith position of all numbers. we know if we add m bits then we have at most $\log m$ many bits. So adding the l many bits will take $\log l = \log\log n$ many bits. $s_k = \sum_{i=1}^{l} a_{i,k}$. Hence $\text{bin}(s_k) = s_{k,\log l-1} \ldots s_{k,1} s_{k,0}$. Hence $\sum_{i=1}^{l} a_{i,k} = \sum_{j=0}^{\log l-1} s_{k,j} 2^j$

$$\sum_{i=1}^{l} a_i = \sum_{i=1}^{l} \sum_{k=0}^{n-1} a_{i,k} 2^k = \sum_{k=0}^{n-1} \sum_{i=1}^{l} a_{k,k} 2^k = \sum_{k=0}^{n-1} \sum_{j=0}^{\log\log n-1} s_{k,j} 2^j \cdot 2^k = \sum_{j=0}^{\log\log n-1} \sum_{k=0}^{n-1} s_{k,j} 2^{j+k}$$

So this is converted to addition of $\log\log n$ many numbers of at most $n + \log\log n = O(n)$ many bits. ■ Recursing like this we have $\text{IterADD}_{\log\log n,n} \leqslant_{cd} \text{IterAdd}_{2,O(n)}$. Hence

Theorem 2.2. $\text{IterADD}_{\log n,n} \leqslant_{cd} \text{IterAdd}_{2,O(n)}$ *and therefore* $\text{IterADD}_{\log n,n} \in AC^0$

Remark: Apart from this $O(\log^* n)$ method to prove $\text{IterADD}_{\log n,n} \in AC^0$ there is also another method in Vinay Kumar's Lecture Notes

2.2 Iterated Addition of n many n-bit numbers

We know $\text{IterAdd}_{n,n} \leqslant_{cd} \text{IterAdd}_{n,1}$ but we dont know anything about $\text{IterAdd}_{n,n} \leqslant_{cd} \text{IterAdd}_{\log n,n}$. If that happens it will put $\text{IterAdd}_{n,n}$ to AC^0.
Remark: $\text{IterAdd}_{n,1}$ is also known as BCOUNT_n.

Theorem 2.3. $\text{IterAdd}_{n,n} \leqslant_{cd} \text{IterAdd}_{n,1}$

Proof: Let we given n many n-bit numbers $a_1, \ldots, a_n$, where $\text{bin}(a_i) = a_{i,n-1} \ldots a_{i,1} a_{i,0}$. First we compute $s_k = \sum_{i=1}^{n} a_{i,k}$ using BCOUNT_n for all $0 \leqslant k \leqslant n$. Now it becomes addition of $\log n$ many $O(n)$ bit numbers which we already know is in AC^0 by Theorem 2.2. Hence $\text{IterAdd}_{n,n} \leqslant_{cd} \text{BCOUNT}_n$ ■

3 $\text{IterAdd}_{n,n} \equiv \text{BCOUNT}_n \equiv \text{Threshold}_{n,m} \equiv \text{Majority}_n \equiv \text{MULT}_n \equiv \text{SORT}_{n,n}$

Problem: MULT
Input: 2 n-bit numbers $a = a_0, \ldots, a_{n-1}$, $b = b_0, \ldots, b_{n-1}$
Output: $c = a \cdot b$

Theorem 3.1. $\text{MULT}_{n,n} \leqslant \text{IterAdd}_{n,n}$

Proof: Given a, b where $\text{bin}(a) = a_{n-1} \cdots a_1 a_0$ and $\text{bin}(b) = b_{n-1} \cdots b_1 b_0$ then obviously

$$a \cdot b = \sum_{i=0}^{n-1} a \cdot b_i \cdot 2^i$$

Define for all $0 \leqslant i \leqslant n-1$

$$c_i = \begin{cases} 0^{n-i-1} a_{n-1} \cdots a_1 a_0 0^u & \text{when } b_i = 1 \\ 0^{2n-1} & \text{otherwise} \end{cases}$$

i.e. $c_i = a \cdot 2^i$ if $b_i = 1$. Each c_i is of $2n-1 = O(n)$ many bits long. Hence we have $a \cdot b = \sum_{i=0}^{n-1} c_i$. Hence now we can use the $\mathtt{IterAdd}_{n,n}$ gate to add the n many $O(n)$ many bits to find the multiplication of a and b. Therefore $\mathtt{MULT}_n \leqslant \mathtt{IterAdd}_{n,n}$. ■

Problem: $\mathtt{Majority}_n$
Input: n bits $a_{n-1}, \ldots, a_0$
Output: Find if at least half of the bits are 1

Theorem 3.2. $\mathtt{Majority} \leqslant \mathtt{MULT}$

Proof: Given $a_0, \ldots, a_{n-1}$. Take the number a such that $\mathtt{bin}(a) = a_{n-1} \cdots a_1 a_0$. Denote $l \coloneqq \log n$. Define

$$A = \sum_{i=0}^{n-1} a_i \cdot 2^{li} \text{ and } B = \sum_{i=0}^{n-1} 2^{li}$$

where both A and B consists of n blocks of length l. We took l length block because summation of n bits takes at most l bits. Let $C = A \cdot B$ We represent C in binary as l length blocks where $C = \sum_{i=0}^{2n-1} c_i \cdot 2^{li}$. Each c_i is a l length block. Then the middle block c_{n-1} have exactly the computation of the sum of the a_i. Therefore $c_{n-1} = \sum_{i=0}^{n-1} a_i$.

A and B are constructed in constant depth and fed into $\mathtt{MULT}$ gates yielding C. Now we have to compare c_{n-1} with $\frac{n}{2}$ which can be done in constant depth. ■

Problem: $\mathtt{ExactThreshold}_{n,m}$
Input: n bits $a_{n-1}, \ldots, a_0$
Output: Find if $\sum_{i=0}^{n-1} = m$

We have another similar problem but we have greater than instead of equality.

Problem: $\mathtt{Threshold}_{n,m}$
Input: n bits $a_{n-1}, \ldots, a_0$
Output: Find if $\sum_{i=0}^{n-1} \geqslant m$

Theorem 3.3. $\mathtt{BCOUNT} \leqslant \mathtt{ExactThreshold} \leqslant \mathtt{Threshold} \leqslant \mathtt{Majority}$

Proof: $\underline{\mathtt{BCOUNT} \leqslant \mathtt{ExactThreshold}}$: Let $\sum_{i=0}^{n-1} a_i = \sum_{i=0}^{l=\log n} s_i \cdot 2^i$. Let for all $0 \leqslant j \leqslant l$, R_j denote the set of all numbers $r \in \{0, \ldots, n\}$ whose j-th bit is 1 in its binary representation . Then we can say

$$s_j = \bigvee_{r \in R_j} \left[\!\!\left[\sum_{i=0}^{n-1} a_i = r \right]\!\!\right]$$

Now R_j don't depend on the input but only on the input n so it can be hardwired this into the circuit. Thus we have a circuit for $\mathtt{BCOUNT}$ which uses $\mathtt{ExactThreshold}$.

$\underline{\mathtt{ExactThreshold} \leqslant \mathtt{Threshold}}$: We know for any r and a variable x certainly

$$[\![x = r]\!] = [\![x \geqslant r]\!] \wedge [\![x < r+1]\!]$$

With this we have a constant depth circuit for $\mathtt{ExactThreshold}$ using the $\mathtt{Threshold}$ gates.

$\underline{\mathtt{Threshold} \leqslant \mathtt{Majority}}$: We are given $a_0, \ldots, a_{n-1}$. Let we want to find $\sum_{i=0}^{n-1} \geqslant m$ then we have this following relations

$$\sum_{i=0}^{n-1} a_i \geqslant m \iff \begin{cases} \mathtt{Maj}_{2n-2m}\left(a_0, \ldots, a_n, \underbrace{1, \ldots, 1}_{n-2m}\right) & \text{wher } m < \frac{n}{2} \\ \mathtt{Maj}_{2m}\left(a_0, \ldots, a_n, \underbrace{0, \ldots, 0}_{n-2m}\right) & \text{wher } m \geqslant \frac{n}{2} \end{cases}$$

This Maj_{2n-2m} and Maj_{2m} can be constructed in constant depth. ■

Remark: Hence using the theorems above we have the final relation

$$Majority \leqslant MULT \leqslant IterAdd_{n,n} \leqslant BCOUNT \leqslant ExactThreshold \leqslant Threshold \leqslant Majority$$

which gives the following corollary

Corollary 3.4. $IterAdd_{n,n} \equiv BCOUNT \equiv Threshold \equiv Majority \equiv MULT$

Definition 3.1 (TC^0). *Constant depth polynomial size unbounded fanin circuit family using the gates* $\wedge, \vee, \neg, Maj$. *Alternating Definition: Constant depth polynomial size unbounded fanin circuit family using the gates* $\neg, Maj$.

Theorem 3.5. *Both the definitions of* TC^0 *are equivalent.*

Theorem 3.6. $IterAdd_{n,n}, BCOUNT, MULT \in TC^0$

Proof: By Corollary 3.4 we have the result. ■

Problem: SORT
Input: n numbers with n bits each
Output: The sequence of the input numbers in non-decreasing order.

Definition 3.2. *We define* $un_n(k) \triangleq 1^k 0^{n-k}$ *where* $k \in \{0, \ldots, n\}$

Problem: UCOUNT
Input: $a_0, \ldots, a_{n-1} \in \{0, 1\}$
Output: $un_n\left(\sum_{i=0}^{n-1} a_i\right)$

Lemma 3.7. $BCOUNT \leqslant UCOUNT$

Proof: Given $a_0, \ldots, a_{n-1}$ suppose $b_1 \ldots b_n = un_n\left(\sum_{i=0}^{n-1} a_i\right)$. Also let $b_0 = 1$ and $b_{n+1} = 1$. Then in the number $b_0 b_1 \ldots b_n b_{n+1}$ there is at least one 1 from left and at least one 0 from right. Now take

$$d_j = b_j \wedge \neg(b_{j+1})$$

for all $0 \leqslant j \leqslant n$. Then if $d_j = 1$ that means $b_j = 1$ and $b_{j+1} = 0$ hence b_j is the last bit which is 1 afterwards every bit is 0. Hence there are in total j many 1's except b_0. Therefore $\sum_{i=0}^{n-1} a_i = j$. So we take all $r \in 0, \ldots, n$ such that the jth bit of r is on then define

$$c_j = \bigwedge_{i \in R_j} d_i$$

Hence if $c_j = 1$ then we can say $\sum_{i=0}^{n-1} a_i$ is such a number whose jth bit is 1. Thus we take $BCOUNT(a_0, \ldots, a_{n-1}) = c_{\log n - 1} \cdots c_1 c_0$ ■

Theorem 3.8. $UCOUNT \equiv BCOUNT$

Theorem 3.9. $UCOUNT \leqslant SORT \leqslant UCOUNT$

Proof: $\mathsf{UCOUNT} \leqslant \mathsf{SORT}$: Given $a_0, \ldots, a_{n-1}$ define $A_i = a_i \underbrace{0 \cdots, 0}_{n-1}$ for all $0 \leqslant i \leqslant n-1$. Sorting these numbers the sequence of most significant bits in the ordered sequence is the $un_n \left(\sum_{i=0}^{n-1} a_i \right)$ reversed.

$\mathsf{SORT} \leqslant \mathsf{UCOUNT}$: Given $a_i = a_{i,n-1} \cdots, a_{i,1} a_{0,0}$ for all $1 \leqslant i \leqslant n$. Define

$$c_{i,j} \triangleq [\![(a_i < a_j) \vee ((a_i = a_j) \wedge i \leqslant j)]\!]$$

$\forall\, 0 \leqslant i, j \leqslant n-1$. Now we define for all $1 \leqslant j \leqslant n$

$$c_j \triangleq \mathsf{UCOUNT}(c_{0,j} \ldots c_{n-1,j})$$

Hence first of all the number of 1's in $c_{0,j} \ldots c_{n-1,j}$ is the number of a_i's with value strictly less than a_j or if equal then index is less than or equal to j. Hence it represents the position of a_j when the numbers are sorted. Therefore c_j is the n-bit unary representation of the position of a_j in the ordered output sequence. If the ordered sequence is $a'_1, \ldots, a'_n$ then $a'_{c_j} = a_j$. Let $a'_i = a'_{i,n-1} \cdots a'_{i,0} a'_{i,1} a'_{i,0}$ for $1 \leqslant i \leqslant n$. Then

$$a'_{i,k} = 1 \iff [\![a'_i = a_j]\!] \wedge [\![a_{j,k} = 1]\!]$$

Hence $a'_{i,k} = \bigwedge_{1 \leqslant j \leqslant n} \left([\![c_j = 1^i 0^{n-i}]\!] \wedge a_{j,k} \right)$. ■

Corollary 3.10. $\mathsf{SORT} \in \mathsf{TC}^0$

4 Parallel Random-Access Machine (PRAM)

In [KR90] many Parallel Machine Models are very well written and explained

Definition 4.1 (PRAM). *A PRAM consists of an infinite sequence of processors $P_1, P_2, \ldots$. Each processor P_i has its local memory realized as an infinite sequence $R_{i,0}, R_{i,1}, R_{i,2}, \ldots$ of registers. Additionally there is a common (or global) memory given by the infinite sequence $C_0, C_1, C_2, \ldots$ of registers. Each register can hold as value a natural number, stored in binary as a bit string.*

A particular PRAM M is specified by a program and a processor bound. The program is a sequence of instructions $S_1, S_2, \ldots, S_s$ and the processor bound is a function $p : \mathbb{N} \to \mathbb{N}$

Workflow: *Initially the input is distributed over the lowest numbered global memory cells. Then all the processors $P_1, \ldots, P_{p(n)}$ start the execution of the program with the first instruction S_1. Each instruction S_m is one of the following 9 types.*

Instructions: *Suppose a fixed processor P_r, $1 \leqslant r \leqslant p(n)$. Let $c, i, j, k \in \mathbb{N}$ and $1 \leqslant l \leqslant s$. We describe the instructions and their effect in turn. If S_m is one of the first 7 types then after the execution of S_m processor P_r continues with instruction S_{m+1}*

1. $R_i \leftarrow c$*:* $R_{r,i}$ *gets as value the constant* c.
2. $R_i \leftarrow \#$*:* $R_{r,i}$ *gets as value the number of the processor i.e.* r
3. *(Numerical Operations)* $R_i \leftarrow R_j + R_k$, $R_i \leftarrow R_j - R_k$*: The result of adding the contents of* $R_{r,j}$ *and* $R_{r,k}$ *(Subtracting) is stored in* $R_{r,i}$*. (If subtraction results in a negative number the* $R_{r,i}$ *gets value 0)*
4. *(Bitwise Operation)* $R_i \leftarrow R_j \vee R_k$, $R_i \leftarrow R_j \wedge R_k$, $R_i \leftarrow R_j \oplus R_k$*: The result of performing the bitwise* OR *(*AND*,* PARITY*) of the contens of* $R_{r,j}$ *and* $R_{r,k}$ *is stored in* $R_{r,i}$.
5. *(Shift Operation)* $R_i \leftarrow$ *shl* R_j, $R_i \leftarrow$ *shr* R_j*: If the contents of* $R_{r,j}$ *is* a *where* $\mathtt{bin}(a) = a_l \cdots a_0$ *then register* $R_{r,i}$ *will get the value* b *where* $\mathtt{bin}(b) = a_l \cdots a_0 0$ *(for shl) and* $\mathtt{bin}(b) = a_l \cdots a_1$ *(for shr)*

6. *(Indirect Addressing 1):* $R_i \leftarrow (R_j)$*: The contents of the global memory register whose number is given by the contents of* $R_{r,i}$.

7. *(Indirect addressing 2):* $(R_i) \leftarrow R_j$*: The contents of* $R_{r,j}$ *is copied to that global register whose number is given by the contents of* $R_{r,i}$

8. *(Conditional Jump)* IF $R_i < R_j$ GOTO l*: If the contents of* $R_{r,i}$ *is smaller in value than the contents of* $R_{r,j}$ *then processor* P_r *continues with the execution of instruction* S_l*; otherwise it continues with the next instruction* S_{m+1}.

9. HALT*: The computation of processor* P_r *stops.*

The computation of M *stops when all the processors* $R_1, \ldots, R_{p(n)}$ *have halted*

Let $f : \mathbb{N}^* \to \mathbb{N}^*$. We say that a PRAM M computes f if the following holds: Given an input $(x_1, \ldots, x_n) \in \mathbb{N}^n$ (input length: n) the x_i are first distributed in global memory cells $C_1, \ldots, C_n$ i.e. C_i is initialized to x_i for $1 \leqslant i \leqslant n$. Additionally C_0 is initialized to n. Then the computation of M is started. Let m be the contents of register C_0 after the computation stops. Then $f(x_1, \ldots, x_n)$ is the vector from $\mathbb{N}^m$ whose components are the numbers in the global register $C_1, \ldots, C_m$

Definition 4.2 (EREW-PRAM)**.** *E:= Exclusive,* R := *Read, W := Write*

Question 1. *Can* n*,* n*-bit numbers be sorted in* $O(\log n)$ *time on an EREW-PRAM? Bir arithmetic is allowed and each with polynomially many processors. Bit operation takes* $O(1)$ *time. Hence is* $\mathsf{SORT} \in \mathsf{EREW}[\log n, \mathtt{poly}(n)]$*?*

Question 2. *What about if bit arithmetic are not allowed?*

Question 3. *If comparisons are allowed can we say anything?*

Answer: AKS (Ajtai–Komlós–Szemerédi) in their paper [Pad11] showed in $O(\log n)$ time

Question 4. *If number of processors is* $O(n)$ *what can be said?*

There are also other parallel machine models: CRCW (C:= Concurrent), CREW, CROW (O:= Owner), OROW. Concurrent means everyone can access the memory concurrently. Owner means only the processor who is the owner of a memory can access it.

In CRCW allows simultaneous read and writes. Hence we have to resolve write conflicts. Some commonly used methods of resolving write conflicts are COMMON, ARBITRARY, PRIORITY models.

Definition 4.3 (NC^1)**.** *Class of languages accepted by circuits of depth* $O(\log n)$ *and size* $n^{O(1)}$ *and fanin 2*

Definition 4.4. *Formulas are trees i.e. circuits with fanout 1*

Theorem 4.1. *In* NC^1 *formulas and circuits are equivalent.*

Theorem 4.2. $NC^1 \subseteq \mathsf{OROW}[\log n, \mathtt{poly}(n)]$

Proof: For any circuit $C \in NC^1$ we create the OROW PRAM where each gate of C is a processor in the PRAM and each processor has the writing access of their own memory and for any edge $u \to v$ in C processor v has the reading access of the memory of u. With this OROW PRAM each processor can read memory from its children then writes the computed value in his memory location thus it computes C. Since C has size $n^{O(1)}$ the PRAM has $n^{O(1)}$ many processors and since the circuit has depth $O(\log n)$ the OROW PRAM takes $O(\log n)$ time to compute. ■

Open Question 4.3. $NC^1 \overset{?}{\supseteq} \mathsf{OROW}[\log n, \mathtt{poly}(n)]$

Definition 4.5 (AC^k, NC^k)**.** *The class of circuits consists of the gates* $(\vee, \wedge, \neg)$ *(The subscript* n *or* 1 *denotes the fanin) of polynomial size and depth* $O(1) = O(\log^k n)$

Similarly for NC^k *but with fanin 2*

Theorem 4.4. $AC^1 = CRCW[\log n, poly(n)]$

Proof: $AC^1 \subseteq CRCW[\log n, poly(n)]$: (COMMON) For any gate all its parents have similarly reading access of the memory of the gate and all its childs has writing access. But we use COMMON model to resolve write conflicts. For an OR gate if anyone evaluates to be 0 then it ignores and if its 1 then it writes 1. Thus everybody writes 1. Every OR gate by has 0 previously in the memory. Similarly for AND gate its the opposite.

$AC^1 \supseteq CRCW[\log n, poly(n)]$: Theorem 4.5, [SV84] ■

Theorem 4.5. *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length-respecting function computed by the CRCW PRAM M in time $t(n)$. Let the processor bound of M be $p(n)$. The there is a family $\mathcal{C}$ of circuits that computes f where the depth of $\mathcal{C}$ is $O(t(n))$ and size is polynomial in $n + t(n) + p(n)$.*

Proof: See [Vol99, Theorem 2.56, Page 70] ■

5 Some Circuit Complexity Class Relations

Theorem 5.1. $AC^0 \subseteq NC^1$

Proof: For all unbounded gate in AC^1 circuit C with fanin s we can replace each gate with s many same gates fo fanin 2 and depth log s. Thus we have $AC^0 \subseteq NC^1$ ■

Theorem 5.2. $TC^0 \subseteq NC^1$

Proof: We will first show that using Redundant Algebra $ADD_{2n} \in NC^0$. In Redundant Algebra with base 4 while adding two digits the result can be at most in normal integers $-6, \ldots, 6$. We only have to check for $\pm 6, \pm 5, \pm 3$. Other case we dont have to watch out.

$$6 = 12 \qquad 5 = 11 \qquad 3 = 1\overline{1}$$
$$\overline{6} = \overline{12} \qquad \overline{5} = \overline{11} \qquad \overline{3} = \overline{1}1$$

For ± 4 it is the normal representation in this system ie 10 and $\overline{1}0$ respectively. Now whenever we add two digits in this system in the sum result can be at most 2 digits. Among them we call the right most digit the sum digit or s and the left digit the carry digit c becasue it becomes the carry for the addition. Now see in all of the numbers the sum digit $|s| \leqslant 2$ and carry digit $|c| \leqslant 1$ So whenever we add a carry generated before to the current sum no new carry is generated becasue of the carry. So We dont have to look for carry generation and propagation. The carry generated at the previous position will add to the sum digit in the current position after getting added to that it will not further propagate after getting added the final digit at the current place will be between 3 and $\overline{3}$ So we proved that addition of two n-bit numbers using Redundant algebra is in NC^0

Now we will show converting a number n from base 2 to base 4 is in NC^0. let

$$n = \sum_{k=0}^{m} a_k 2^k = \sum_{k=0}^{\lfloor \frac{m}{2} \rfloor} (a_{2k+1} \times 2 + a_{2k})\, 4^k \qquad \text{if } m \text{ is odd then } a_{m+1} = 0$$

So we just take two bits in binary multiply the left one with 2 and add to the right one and we have the digit at base 4 in that position. So we can change base in NC^0. From now on we will by default assume all the addition is done using Redundant Algebra.

Then we are done in showing $TC^0 \subseteq NC^1$. We will first show that adding n bits is in NC^1 ie $BCOUNT \in NC^1$. So we have n bits $a_n, a_{n-1}, \ldots, a_1$. We will group the bits into groups of two bits and add the two bits in each group. We can do the addition for all groups in parallel. And since we have already shown addition of two

k-bit numbers is in NC^0 this takes constant depth and size since we are adding two bits. Now the number of bits are halved. We do tha same process again and again. Each time it takes $poly(n)$ size and constant depth and at each iteration the number of numbers becomes half. So this whole process takes $O(\log n)$ many iterations. So adding n bits takes $poly(n)$ size and depth $O(\log n)$. So $BCOUNT \in NC^1$

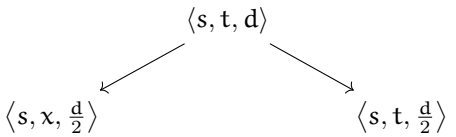
Now We will show that $Majority \in NC^1$. Let the addition of n bits we got is a and the Redundant Algebra representation of $\lceil \frac{n}{2} \rceil$ is b. Now we can calculate $-b$ with reversing the sign of every digit in b ie if $b = \sum_{i=0}^{m} a_k \times 4^k$ then $-b = \sum_{i=0}^{m} \overline{a_k} \times 4^k$. Now we add a and $-b$ which can be done in NC^0. And now we will look for the left most negative digit. If there is nonw we output 1 i.e. majority of the bits are 1 and if there exists a negative bit then output is 0. Hence we have $Majority \in TC^0$. So $TC^0 \subseteq NC^1$. ■

Definition 5.1 (SAC^k). *The class of circuits consists of the gates* $(\vee, \wedge, \neg)$ *of polynomial size and depth* $O(1) = O(\log^k n)$ *but semi unbounded fanin i.e either* $\wedge$ *gates have unbounded fanin and* $\vee$ *gate have bounded fanin or* $\wedge$ *gates have bounded fanin and* $\vee$ *gates have unbounded fanin.*

Theorem 5.3. $AC^0 \subseteq TC^0 \subseteq NC^1 \subseteq L \subseteq NL \subseteq SAC^1 \subseteq AC^1 \subseteq NC^2 \subseteq AC^2 \subseteq \cdots \subseteq NC^i \subseteq AC^i \subseteq \cdots \subseteq NC = AC \subseteq P$

Proof: $NC^1 \subseteq L$: The idea is to simply evaluate the circuit using a depth-first search, which can be defined inductively as follows: visit the output gate of the circuit, visit the gates of the left subcircuit, visit the gates of the right subcircuit. At any moment, we store the number of the current gate, the path that led us there (as a sequence of left's and right's) and any partial values that were already computed. For example, we could have L, R (0), R (1), L, L, 347. This would mean that we are visiting gate 347 and we got there by going left, right, right, left and left. The 0 after the first R indicates that the left input of the second gate on the path evaluated to 0. When we are done visiting a gate (and computed its value), we return to the previous gate on the path. Note that we can recompute the number of that gate by following the path from the beginning. The space requirements of this algorithm are therefore determined by the maximum length of the path, which is equal to the depth of the circuit. The uniformity of the circuit is used when traveling through the circuit and evaluating its gates.

$NL \subseteq SAC^1$: We know $PATH$ is $NL - complete$. Let there is a path $s \rightsquigarrow t$ of length d. Then there exists a vertex x in between s and t such that there exists a path of length $\frac{d}{2}$ from $s \rightsquigarrow x$ and $x \rightsquigarrow t$.



Since there exists one such vertex x and in circuit we will add all this in a big $\vee$ gate for all vertices in the graph. We can represent this as a circuit

$$\langle s, t, d \rangle = \bigvee_{x \in V} \left(\left\langle s, x, \frac{d}{2} \right\rangle \wedge \left\langle s, t, \frac{d}{2} \right\rangle \right)$$

Like this we extend it to 1 length paths. In this circuit we are using $\vee$ gates with unbounded fanin and $\wedge$ gates with 2 fanin. Hence we have $NL \subseteq SAC^1$ ■

6 Iterated Multiplication

Problem: $ItMult_{m,n}$
Input: m many n-bit numbers are given
Output: The product of the given numbers

Theorem 6.1 (Chinese Remainder Theorem). *Let* $k \in \mathbb{N}$, $P = p_1 p_2 \cdots p_k$. *Let* $a_1, \ldots, a_k$ *be given. Then there is a unique number* $a \in \{0, \ldots, P-1\}$ *such that* $a \equiv a_i \pmod{p_i}$ *for* $1 \leqslant i \leqslant k$. *More specifically, we have*

$$a = \left[\sum_{i=1}^{k} (a_i \pmod{p_i})\right] \pmod{P}$$

where $r_i = \frac{P}{p_i}$ *and* $s_i = \left(\frac{P}{p_i}\right)^{-1} \pmod{p_i}$ *(i.e.* s_i *is the multiplicative inverse of* $r_i \pmod{p_i}$*).*

Theorem 6.2 (Prime Number Theorem). *Let* $\Pi(n)$ *denote the number of prime numbers not greater than* n. *Then*

$$\Pi(n) = \Theta\left(\frac{n}{\log n}\right)$$

Corollary 6.3. *Let* p_k *denote the* k*th prime.* $p_n = O(n \log n)$

Theorem 6.4. *Let* p_i *denote the* i*th prime. Then*

$$\prod_{i=1}^{n} p_i \leqslant 4^{n \log n}$$

Theorem 6.5. $\mathrm{ItMult}_{n,n} \in \mathsf{TC}^0$

Proof: Let $a_i = a_{i,n-1} \cdots a_{i,1} a_{i,0}$ for $1 \leqslant i \leqslant n$ be the given numbers which we want to multiply up. Hence

$$a = a_1 \cdot a_2 \cdots a_n < (2^n)^n = 2^{n^2} < p_1 \cdot p_2 \cdots p_{n^2}$$

Then take $p = p_1 \cdot p_2 \cdots p_{n^2}$. Then our goal is to compute $a \pmod{p}$. Our first goal is to compute $a \pmod{p_i}$ for all $1 \leqslant i \leqslant n^2$.

Now take $r_i = \frac{p}{p_i}$ and $s_i = \left(\frac{p}{p_i}\right)^{-1} \pmod{p_i}$. Then by Chinese Remainder Theorem we have

$$a = \sum_{i=1}^{n^2} (a \pmod{p_j}) \cdot r_i \cdot s_i$$

Now we have

$$a \pmod{p_j} = \prod_{i=1}^{n} a_i \pmod{p_j} = \prod_{i=1}^{n} (a_i \pmod{p_j})$$

Take

$$a'_{i,j} \coloneqq a_i \pmod{p_j} = \left(\sum_{k=0}^{n-1} a_{i,k} \cdot 2^k\right) \pmod{p_j} = \sum_{k=0}^{n-1} a_{i,k} \cdot (2^k \pmod{p_j}) \quad \text{for } 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant n^2$$

This gives n^3 many numbers all of which can be computed in constant depth. The values $2^k \pmod{p_j}$ can be hardwired into the circuit since they do not depend on the input but only the input length. So we can compute $a'_{i,j}$ using ItAdd and MULT gates (since we know they are already in TC^0) in constant depth. Using the Corollary 6.3 we have $p_j = O(n^2 \log n)$ for all $1 \leqslant j \leqslant n^2$. Therefore all the $a'_{i,j}$ consists of $O(\log n)$ bits only.

For $1 \leqslant j \leqslant n^2$ let g_j be the generator of the multiplicative group $\mathbb{Z}^*_{p_j}$ group of the finite field $\mathbb{Z}_{p_j}$. Let for $k \in \{1, \ldots, p_j - 1\}$ take $g^{m_j(k)} = k$ where $m_j(k) \in \{0, \ldots, p_j - 2\}$. And we set $m_j(0) \coloneqq p_j - 1$. So now we will compute $m_j\left(a'_{i,j}\right)$. Since the numbers $a'_{i,j}$ contains $O(\log n)$ many bits, these computations can be done in constant depth. Therefore

$$m_j\left(\prod_{i=1}^{n} a_i \pmod{p_j}\right) = m_j\left(\prod_{i=1}^{n} (a_i \pmod{p_j})\right) = \left(\sum_{i=1}^{n} m_j\left(a'_{i,j}\right)\right) \pmod{p_j - 1}$$

for $1 \leqslant j \leqslant n^2$. This can be computed in constant depth using ItAdd gates. The case when $a'_{i,j} = 0$ can be taken care of. These n^2 numbers have logarithmically many bits. Thus we can compute the value $\prod_{i=1}^{n} a_i \pmod{p_j}$ for $1 \leqslant j \leqslant n^2$ in constant depth. Finally we compute

$$a' \coloneqq \sum_{i=1}^{n^2} \left(\prod_{i=1}^{n} a_i \pmod{p_j} \right) \cdot r_j \cdot s_j$$

in constant depth with ItAdd and MULT gates. The values r_j and s_j can be hardwired into the circuit since they dont depend on the input instead the length of the input.

By Chinese Remainder Theorem we have $a = a' \pmod{a'}$ that is $q \coloneqq \left\lfloor \frac{a'}{p} \right\rfloor$ we have $a = a' - q \cdot p$. Since $r_j = \frac{p}{p_j}$ and $s_j \leqslant p_j$ we have $r_j \cdot s_j \leqslant p$. Therefore

$$a' = \sum_{i=1}^{n^2} \left(\prod_{i=1}^{n} a_i \pmod{p_j} \right) \cdot r_j \cdot s_j \leqslant \sum_{i=1}^{n^2} \left(\prod_{i=1}^{n} a_i \pmod{p_j} \right) p \leqslant \sum_{i=1}^{n^2} p_{n^2} p = n^2 \cdot p_{n^2} \cdot p \implies q \leqslant n^2 \cdot p_{n^2}$$

This the value of q is polynomial in n and can be determined by parallely testing for all $i \leqslant n^2 \cdot p_{n^2}$ if $i \cdot p \leqslant a' \leqslant (i+1)p$ which can be done in constant depth and q is the one value of i for which the answer of this is yes. Then the result of the iterated multiplication is $a = a' - q \cdot p$. ■

7 Maximal Independent Set (MIS)

Theorem 7.1. $\mathrm{MIS} \in \mathrm{P}$

7.1 Matching and Independent Set of Line Graph

Definition 7.1 (Line Graph)**.** *The line graph of the graph* G*, written* $L(G)$ *is the graph whose vertices are the edges of* G*, with* $(e_1, e_2) \in E(L(G))$ *when* $e_1 \cap e_2 \neq \phi$

Theorem 7.2. *Given a graph* G *a set of edges* $S \subseteq E$ *is a matching if and only if it is a independent set in the line graph* $L(G)$

Proof: ($\Rightarrow$): Let S be a matching of G. Therefore for all $e_1, e_2 \in S$ we have $e_1 \cap e_2 - \phi$. Hence e_1, e_2 are not adjacent in $L(G)$. Hence S is an independent set of $L(G)$.

($\Leftarrow$): Let S be a independent set in $L(G)$. Then for all $e_1, e_2 \in S$, e_1 and e_2 are not adjacent. Therefore $e_1 \cap e_2 = \phi$. Hence the set S is a set of edges of G where none of them shares any endpoint. Hence S is a matching in G. ■

Fact 1. *Maximal (Maximum) Matching in* G *is an Maximal (Maximum) Independent Set in the line graph* $L(G)$.

7.2 Luby's Algorithm (Randomized Algorithms)

Definition 7.2 (RNC^k)**.** *The class* RNC^k *is the class of problems that can be solved by a randomized algorithm that runs in* $O(\log^k n)$ *time with a polynomial number of processors.*

Remark: Therefore RNC is the randomized counterpart of NC. Luby's Algorithm puts MIS in RNC^2. The main steps or the ideas of the algorithm are:

- The algorithm tries to find I in each stage.

- Each stage finds an independent set I in parallel, using calls on a
- Create a set S of candidates for I as follows: For each vertex v in parallel, include $v \in S$ with probability $\frac{1}{2d(v)}$
- For each edge in E if both its endpoints are in S, discard the one of lower degree, ties are resolved arbitrarily
- The resulting set is I

Here we denote for any $v \in V$ $d(v) \coloneqq \deg(v)$.

Algorithm 1: Luby's Randomized Algorithm on MIS

```
1  begin
2      I ⟵ ∅
3      while G ≠ ∅ do
4          S ⟵ ∅
5          for v ∈ V(G) in parallel do
6              add v to S with probability 1/(2d(v))
7          for {u, v} ∈ E(G) in parallel do
8              if u ∈ S and v ∈ S then
9                  if d(u) < d(v) then
10                     delete u from S
11                 else if d(v) < d(u) then
12                     delete v from S
13                 else if u < v then
14                     delete u from S
15                 else
16                     delete v from S
17         I ⟵ I ∪ S
18         G ⟵ G \ (I ∪ N(I))
19     A ⟵ I
20     return A
```

7.3 Analysis of Luby's Algorithm

Now we will define certain things which will help us to analyze the algorithm:

- A vertex $v \in V$ is *good* if

$$\sum_{u \in N(v)} \frac{1}{2d(u)} \geqslant \frac{1}{6}$$

  A pair of vertices $u, v \in V$ is said to be *good* if

$$\mathrm{good}(u,v) \iff [\![(u,v) \in E]\!] \wedge [\![\mathrm{good}(u) \vee \mathrm{good}(v)]\!]$$

  Intuitively a vertex is good if it has lots of neighbors of low degree. This will give it a decent chance of making into $N(I)$. Therefore in case of bad vertex $\mathrm{bad}(v) = \neg\mathrm{good}(v)$ and for a pair of vertices $u, v \in V$ we have $\mathrm{bad}(u,v) = \neg\mathrm{good}(u,v)$

- We also define

$$N^-(v) = \{u \in N(v) \mid d(u) \leqslant d(v)\} \qquad d^-(v) = |N^-(v)|$$
$$N^+(v) = \{u \in N(v) \mid d(u) > d(v)\} \qquad d^+(v) = |N^+(v)|$$

Therefore we have $d^+(v) + d^-(v) = d(v)$.

Lemma 7.3. *For any* $v \in V$

$$bad(v) \implies d^+(v) \geqslant 2d^-(v) \iff d^-(v) \leqslant \frac{d(v)}{3} \iff d^+(v) \geqslant \frac{2d(v)}{3}$$

Proof: We make the graph directed where if $(u, v) \in E$ previously then the direction of this edge will be

$$u \to v \iff [\![d(u) < d(v)]\!] \vee \Big([\![d(u) = d(v)]\!] \wedge [\![u < v]\!]\Big)$$

Then $d^-(v)$ in the original graph indicates the in-degree of v and $d^+(v)$ indicates the out-degree of v. Therefore $d^+(v) \geqslant \frac{2d(v)}{3}$ means out=degree of v is twice more than the in-degree of v. Now assume the statement is not true. Let v is *bad*. Then

$$\frac{1}{6} > \sum_{u \in N(v)} \frac{1}{2d(u)} \geqslant \sum_{u \in N^-(v)} \frac{1}{2d(u)} \geqslant \sum_{u \in N^-(v)} \frac{1}{2d(v)} \geqslant \frac{d^-(v)}{2d(v)} > \frac{d(v)}{3} \times \frac{1}{2d(v)} \geqslant \frac{1}{6}$$

Hence contradiction. ■

Lemma 7.4. *At least half of the edges in the graph are good*

Proof: We again construct the same directed graph from G as in the proof of Lemma 7.3. If $(u, v) \in E$ then the direction of this edge will be

$$u \to v \iff [\![d(u) < d(v)]\!] \vee \Big([\![d(u) = d(v)]\!] \wedge [\![u < v]\!]\Big)$$

Now for any edge $e = (u, v) \in E$

$$bad(e) \iff bad(u) \wedge bad(v)$$

Let the direction of e is $u \to v$. Since e is bad the vertex v is bad. Therefore using Lemma 7.3 out-degree of v is twice more than the in-degree of v. Hence there is at least two edges out-going from v. Let those edges are $e_1 = v \to w_1$ and $e_2 = v \to w_2$. Hence for every bad vertex there are two edges in E. Hence

$$2\,|\{e \in E \mid bad(e)\}| \leqslant |E|$$

Therefore we have $2\#bad(e) \leqslant \#good(e) + \#bad(v) \implies \#good(e) \geqslant \#bad(e) \implies 2\#good(v) \geqslant |E|$. ■

Lemma 7.5. *For any* $v \in V$

$$\Pr[v \notin I \mid v \in S] \leqslant \frac{1}{2}$$

Proof: If $v \in S$ and $v \notin I$ only if there exists some element of $N^+(v)$ which is also in S. Then

$$\begin{aligned}
\Pr[v \notin I \mid v \in S] &= \Pr[\exists\, u \in N^+(v) \cap S \mid v \in S] \\
&\leqslant \sum_{u \in N^+(v)} \Pr[u \in S \mid v \in S] \\
&= \sum_{u \in N^+(v)} \Pr[u \in S] && \text{[Pairwise independence]} \\
&\leqslant \sum_{u \in N^+(v)} \frac{1}{2d(u)} \leqslant \sum_{u \in N^+(v)} \frac{1}{2d(v)} \\
&\leqslant \frac{d(v)}{2d(v)} && \text{[Lemma 7.3]} \\
&\leqslant \frac{1}{2}
\end{aligned}$$

■

Lemma 7.6. *For any $v \in V$*

$$\Pr[v \in I] \geqslant \frac{1}{4d(u)}$$

Proof: We have

$$\begin{aligned}
\Pr[v \in I] &= \Pr[v \in I \mid v \in S] \Pr[v \in S] \\
&\geqslant (1 - \Pr[v \notin I \mid v \in S]) \frac{1}{2d(v)} \\
&= \frac{1}{2} \times \frac{1}{2d(v)} = \frac{1}{4d(v)} && \text{[Lemma 7.5]}
\end{aligned}$$

■

Lemma 7.7. *If $v \in V$ is good then*

$$\Pr[v \in N(I)] \geqslant \frac{1}{36}$$

Proof: We will consider two cases.
Case 1: v has a neighbor u of degree 2 or less. Then using Lemma 7.6

$$\Pr[v \in N(I)] \geqslant \Pr[u \in I] \geqslant \frac{1}{4d(u)} \geqslant \frac{1}{8}$$

Case 2: $d(u) \geqslant 3$ for all $u \in N(v)$. Then for all $u \in N(v)$ we have $\frac{1}{2d(u)} \leqslant \frac{1}{6}$. Now since v is good we have $\sum_{u \in N(v)} \frac{1}{2d(u)} \geqslant \frac{1}{6}$. Therefore there must exist a subset $M(v) \subseteq N(v)$ such that

$$\frac{1}{6} \leqslant \sum_{u \in M(v)} \frac{1}{2d(u)} \leqslant \frac{1}{3} \tag{1}$$

Therefore

$$\begin{aligned}
\Pr[v \in N(v)] &\geqslant \Pr[\exists\, u \in M(v) \cap I] \\
&\sum_{u \in M(v)} \mathrm{pr}[u \in I] - \sum_{\substack{u,w \in M(v) \\ u \neq w}} \Pr[u \in I \wedge w \in I] && \text{[Inclusion-Exclusion]} \\
&\sum_{u \in M(v)} \frac{1}{4d(u)} - \sum_{\substack{u,w \in M(v) \\ u \neq w}} \mathrm{pr}[u \in S \wedge w \in S] && \text{[Lemma 7.6]} \\
&\sum_{u \in M(v)} \frac{1}{4d(u)} - \sum_{\substack{u,w \in M(v) \\ u \neq w}} \Pr[u \in S]\Pr[v \in S] && \text{[Pairwise independence]} \\
&\sum_{u \in M(v)} \frac{1}{4d(u)} - \sum_{u \in M(v)} \sum_{w \in M(v)} \frac{1}{2d(u)} \frac{1}{2d(w)} \\
&\sum_{u \in M(v)} \frac{1}{2d(u)} \left[\frac{1}{2} - \sum_{w \in M(v)} \frac{1}{2d(w)} \right] \\
&\geqslant \frac{1}{6} \left(\frac{1}{2} - \frac{1}{3} \right) = \frac{1}{36} && \text{[By (1)]}
\end{aligned}$$

■

Lemma 7.8. *If $e \in E$ is good then*

$$\Pr[e \textit{ is deleted}] \geqslant \frac{1}{36}$$

Proof: If the edge $e = (u, v) \in E$ is deleted then either u or v is good vertex. Let u is good. Since e is deleted then $\Pr[u \in N(I)] \geqslant \frac{1}{36}$ by Lemma 7.7. Hence

$$\Pr[e \text{ is deleted}] \geqslant \Pr[\mathrm{good}(u) \wedge u \in N(I)] + \Pr[\mathrm{good}(v) \wedge v \in N(I)] \geqslant \frac{1}{36}$$

■

Lemma 7.9. *Let X ve the random variable representing the number of deleted edges. Then*

$$\mathbb{E}[X] \geqslant \frac{|E|}{72}$$

Proof: Take the indicator random variable for each edge $e \in E$

$$X_e = \begin{cases} 1 & \text{if } e \text{ is deleted} \\ 0 & \text{otherwise} \end{cases}$$

Therefore $X = \sum_{e \in E} X_e$. Then we have

$$\begin{aligned}
\mathbb{E}[X] = \mathbb{E}\left[\sum_{e \in E} X_e\right] &= \sum_{e \in E} \mathbb{E}[X_e] \\
&\geqslant \sum_{good(e)} \mathbb{E}[X_e] \geqslant \sum_{good(e)} \Pr[e \text{ is deleted}] \\
&\geqslant \sum_{good(e)} \frac{1}{36} && \text{[Lemma 7.8]} \\
&\geqslant \frac{|E|}{2} \times \frac{1}{36} = \frac{|E|}{72} && \text{[Lemma 7.4]}
\end{aligned}$$

■

Theorem 7.10. *Luby's Algorithm puts* $\mathsf{MIS} \in \mathsf{RNC}^2$

References

[SV84] Larry Stockmeyer and Uzi Vishkin. "Simulation of Parallel Random Access Machines by Circuits". In: *SIAM Journal on Computing* 13.2 (1984), pp. 409–422. DOI: 10.1137/0213027. eprint: https://doi.org/10.1137/0213027. URL: https://doi.org/10.1137/0213027.

[KR90] Richard M. KARP and Vijaya RAMACHANDRAN. "CHAPTER 17 - Parallel Algorithms for Shared-Memory Machines". In: *Algorithms and Complexity*. Ed. by JAN VAN LEEUWEN. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, 1990, pp. 869–941. ISBN: 978-0-444-88071-0. DOI: https://doi.org/10.1016/B978-0-444-88071-0.50022-9. URL: https://www.sciencedirect.com/science/article/pii/B9780444880710500229.

[Vol99] Heribert Vollmer. "Relations to Other Computation Models". In: *Introduction to Circuit Complexity: A Uniform Approach*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 35–78. ISBN: 978-3-662-03927-4. DOI: 10.1007/978-3-662-03927-4_3. URL: https://doi.org/10.1007/978-3-662-03927-4_3.

[Pad11] "Ajtai–Komlós–Szemerédi Sorting Network". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 16–16. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2379. URL: https://doi.org/10.1007/978-0-387-09766-4_2379.