External Memory Suffix Array Construction

Roman Dementiev

Juha Kärkkäinen

Jens Mehnert

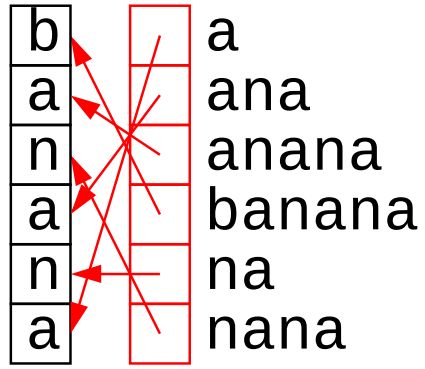
Peter Sanders

MPI Informatik, U. Karlsruhe, U. Helsinki

Suffix Arrays

sort suffixes $T[i..n]$ of string $T[0..n]$

over alphabet $\{1..n\}$.

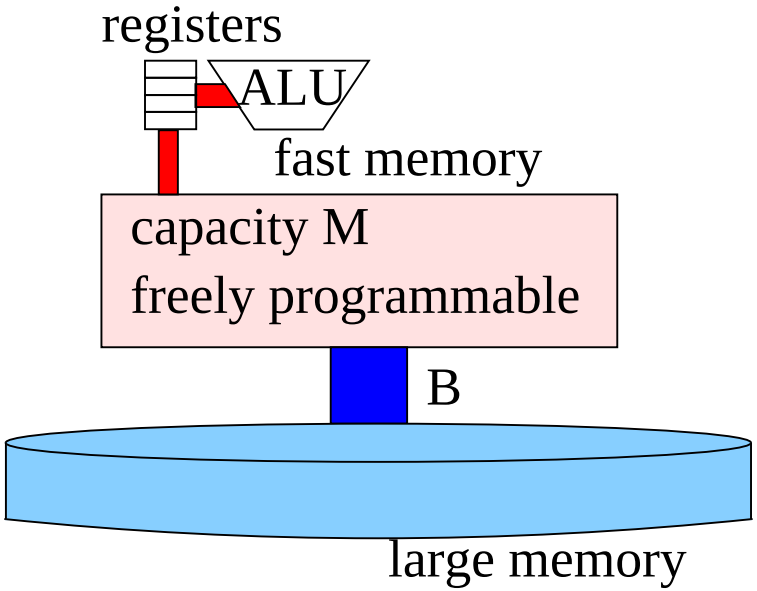


Applications

- Full text search
- Burrows-Wheeler text compression
- Bioinformatics,. . .

Big interest in BIG inputs $\rightsquigarrow$

External memory



$$\mathrm{scan}(n) = \frac{n}{B}, \quad \mathrm{sort}(n) = \frac{2n}{B}\left\lceil \log_{M/B} \frac{n}{M} \right\rceil \text{ machine words}$$

Related Work

Incremental: $\mathcal{O}\left(\frac{n}{M}\mathrm{scan}(n)\right)$ I/Os [Gonnet/Baeza-Yates/Snider 92]

[CF 97] not very scalable, a lot of internal work

Doubling: Sort by first 2^i characters in iteration i [Manber/Myers 93]

$\rightsquigarrow \mathcal{O}(\mathrm{sort}(n)\log \mathrm{maxlcp})$ I/Os [AFGV 97]

Doubling+Discarding: Avoid sorting suffixes known to be unique

[Crauser/Ferragina 97]

Best scalable algorithm in study. $> 6h$ for 26 MByte.

$\rightsquigarrow$ External construction not practical?

via Suffix-Tree: $\mathcal{O}(\mathrm{sort}(n))$ I/Os [Farach/Ferragina/Muthukrishnan 00]

very complicated

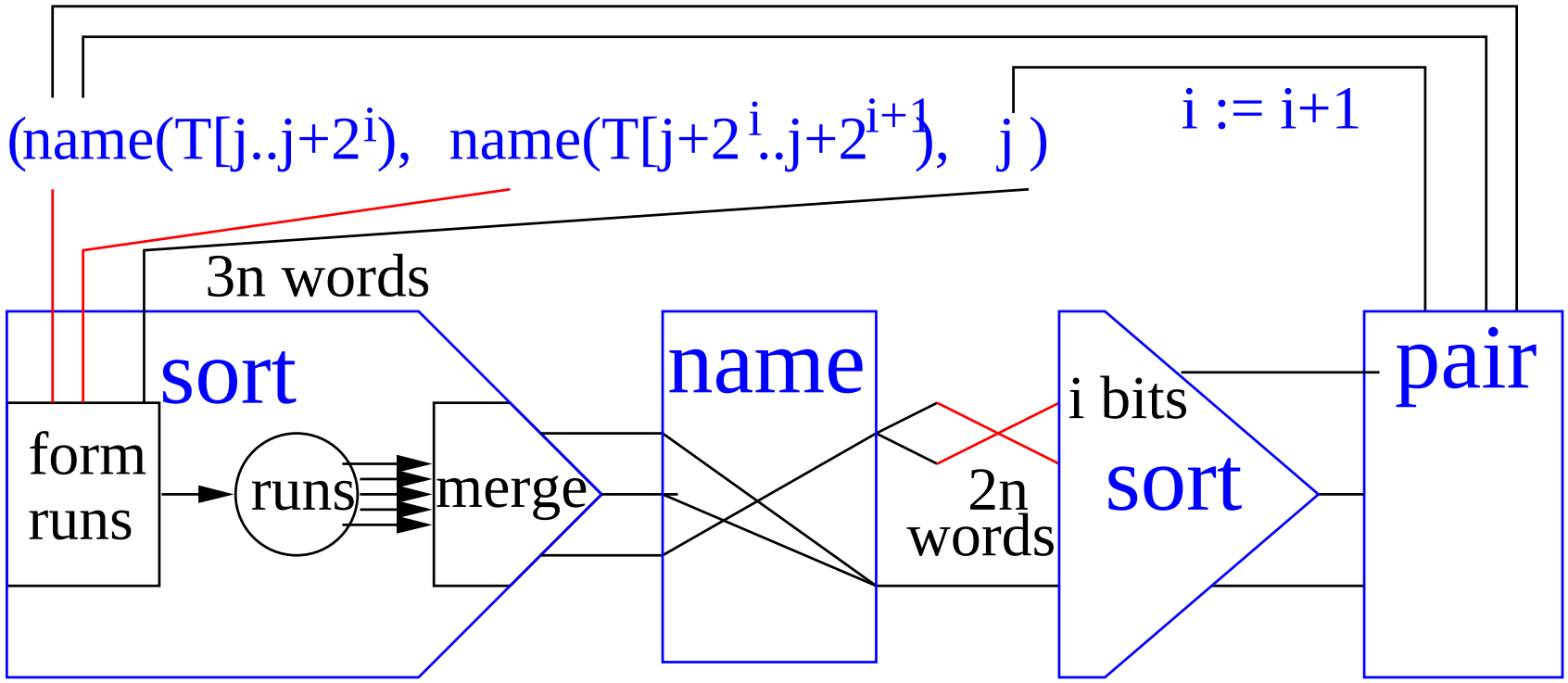
DC3: Simple, linear time, $\mathcal{O}(\mathrm{sort}(n))$ I/Os [KS 03].

Practical? Better than improved doubling?

Pipelined Doubling with Bit Shuffling

$\text{name}(T[i..i+k]) \in \{1..n\}$ preserves order of k-substrings $\forall i$

(T[j], T[j+1], j)



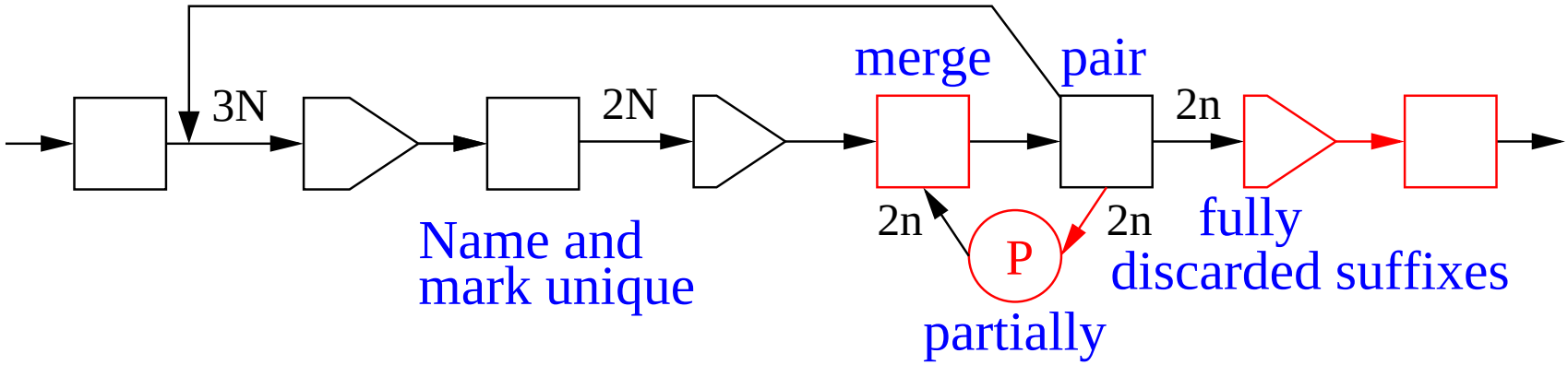
total I/O complexity: $\text{sort}(5n) \log \text{maxlcp} + \mathcal{O}(\text{sort}(n))$

Improved Discarding

- Scan all unique suffixes [CF 97]$\rightsquigarrow$

  Scan new unique suffixes [Kärkkäinen 03]

- Triples $\rightsquigarrow$ pairs



$\mathrm{sort}(5N)+\mathcal{O}(\mathrm{sort}(n))$ I/Os where $N = \sum_i \log \mathsf{distPrefixSize}(T[i..n])$

a-Tupling

Sort by first a^i characters in iteration i

Constant Factor in I/Os

a	2	3	4	5	6	7
$(a+3)/\log a$	5.00	3.78	3.50	**3.45**	3.48	3.56

Difference Cover 3 (DC3) Algorithm

1. sort $T[i..n]$ for $i \bmod 3 \in \{1, 2\}$

   sort and name triples

   recurse

2. sort $T[i..n]$ for $i \bmod 3 \in \{0\}$

   sort pairs $(T[3i], \text{name}(T[3i+1..n]))$

3. merge using difference cover property of $\{1, 2\}$

   $T[3i..n] \leq T[3j+1..n]$ iff

   $(T[3i\quad\ ], \text{name}(T[3i+1..n])) \leq$

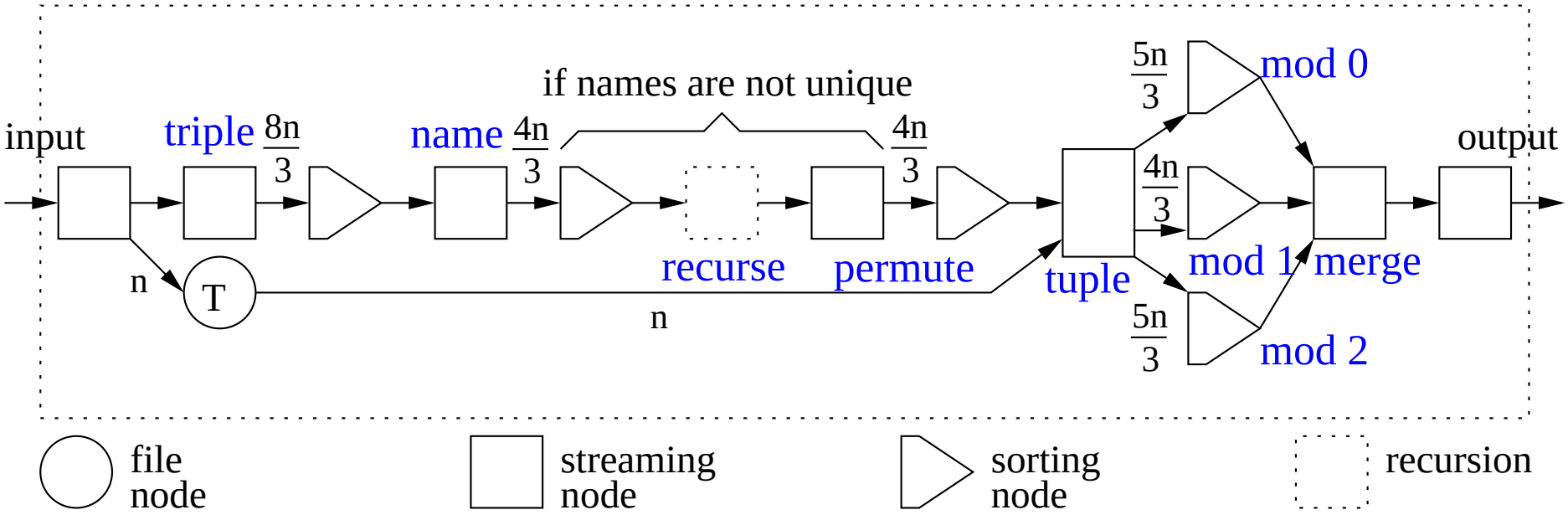
   $(T[3j+1], \text{name}(T[3j+2..n]))$

   $T[3i..n] \leq T[3j+2..n]$ iff

   $(T[3i\quad\ ], T[3i+1], \text{name}(T[3i+2..n])) \leq$

   $(T[3j+2], T[3j+3], \text{name}(T[3j+4..n]))$

Pipelined DC3



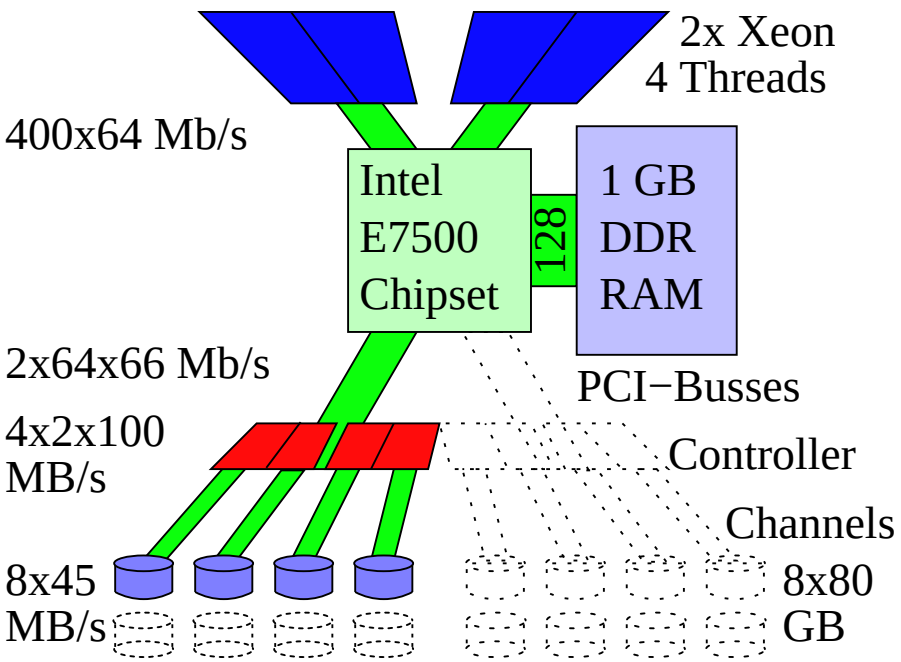
$\mathrm{sort}(30n) + \mathrm{scan}(6n)$ I/Os

Experimental Setup

`g++3.2.3 -O2`

STXXL library [Dementiev 03] with

new iterator-like pipelining feature



Genome: Human Genome

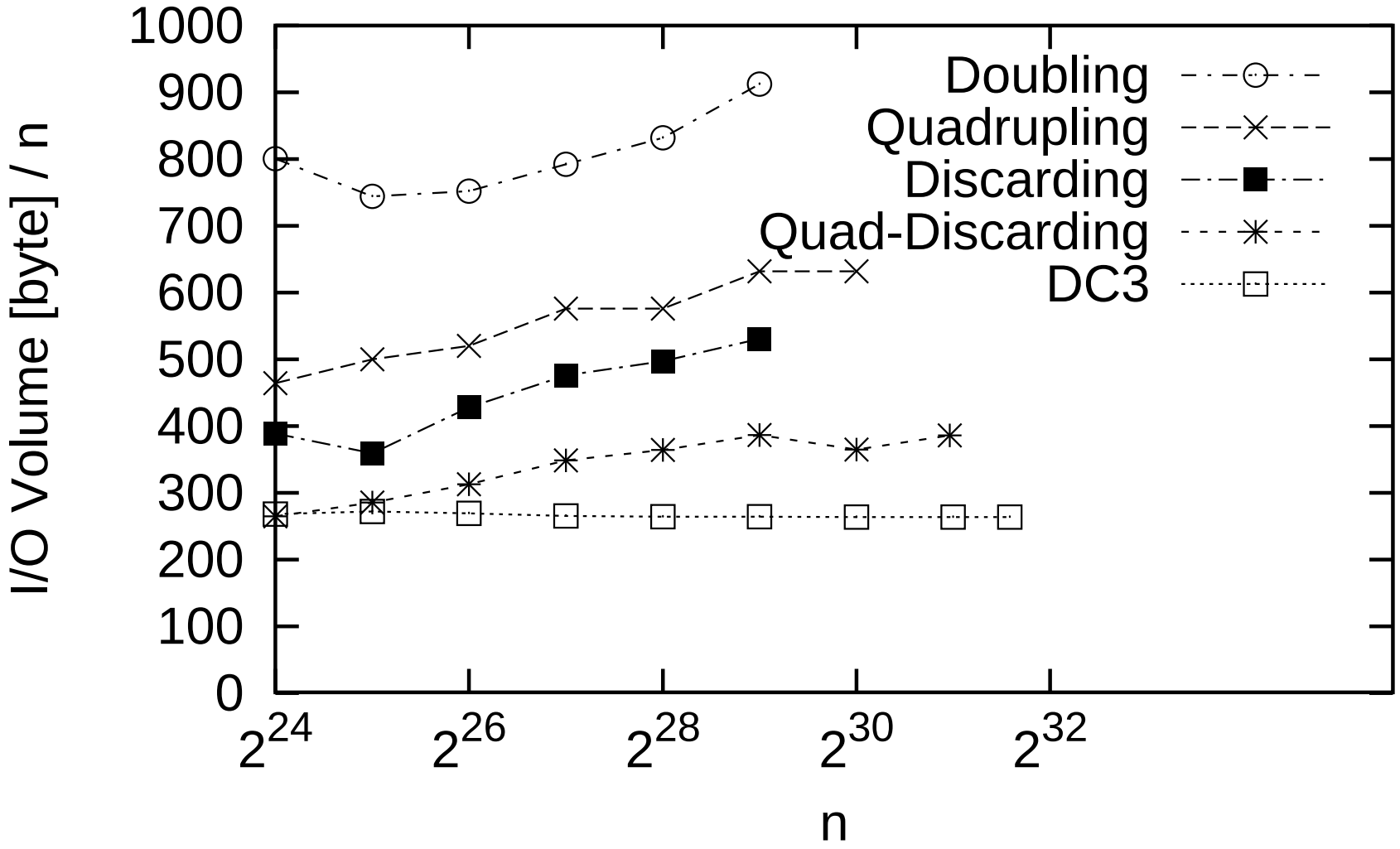
Gutenberg: ≈ 3GByte English text from Gutenberg project

HTML: ≈ 3GByte text from a crawl of `.gov`

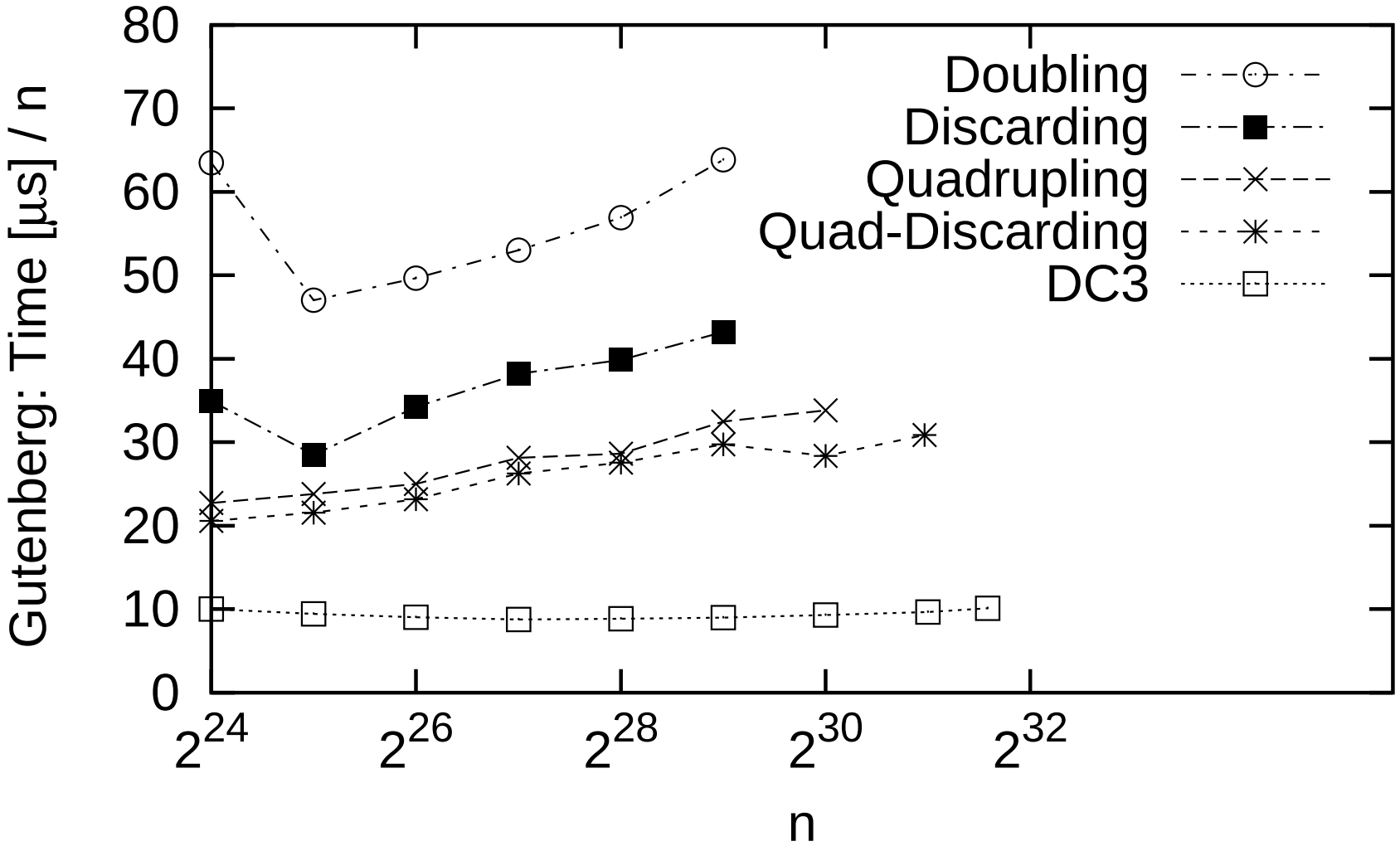
Source: ≈ 0.5GByte Linux sources

Random2: $T \circ T$ with $T := \text{randChar}^{n/2}$

Gutenberg I/Os



Gutenberg Time



I/O bound even for a single disk

Comparison with Previous Implementations

- $5\times$ less I/O volume than [CF 97]
- 7–8$\times$ less clock cycles than [CF 97] (including BGS algorithm)
- $2.4\times$ faster than internal compressed Genome [LSSSY 02]
- $1.2\times$ slower than internal Genome on 64 GByte super computer [Sadakane Shibuya 01]
- Faster than linear time internal LCP computation on MPII's SUN Starfire 15000

Conclusion

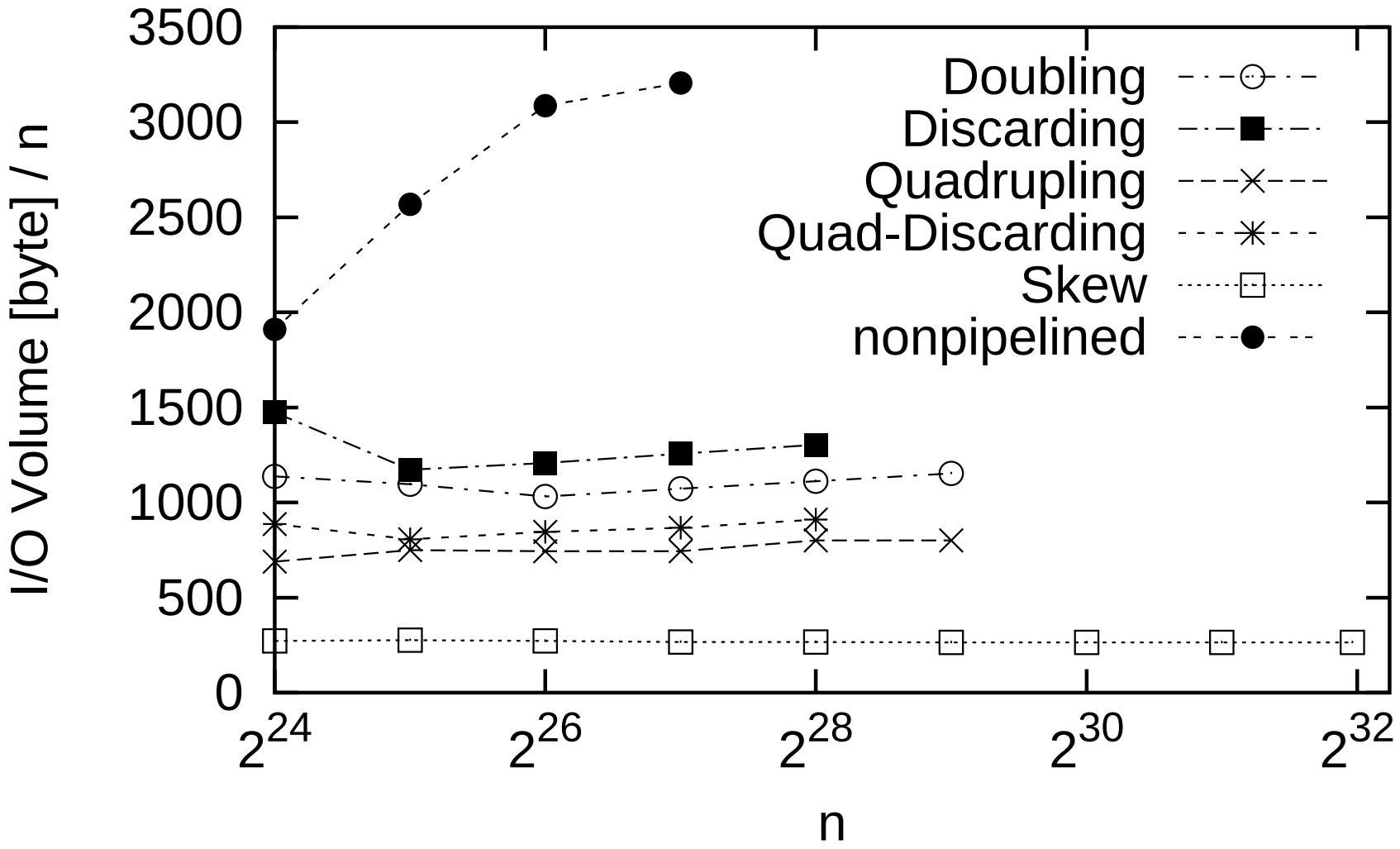
- External DC3 is practical
- Better than pipelined, shuffled 4-tupling with improved discarding
- STXXL makes pipelining easy. Saves factor 2–3 in I/O volume.

Future Work

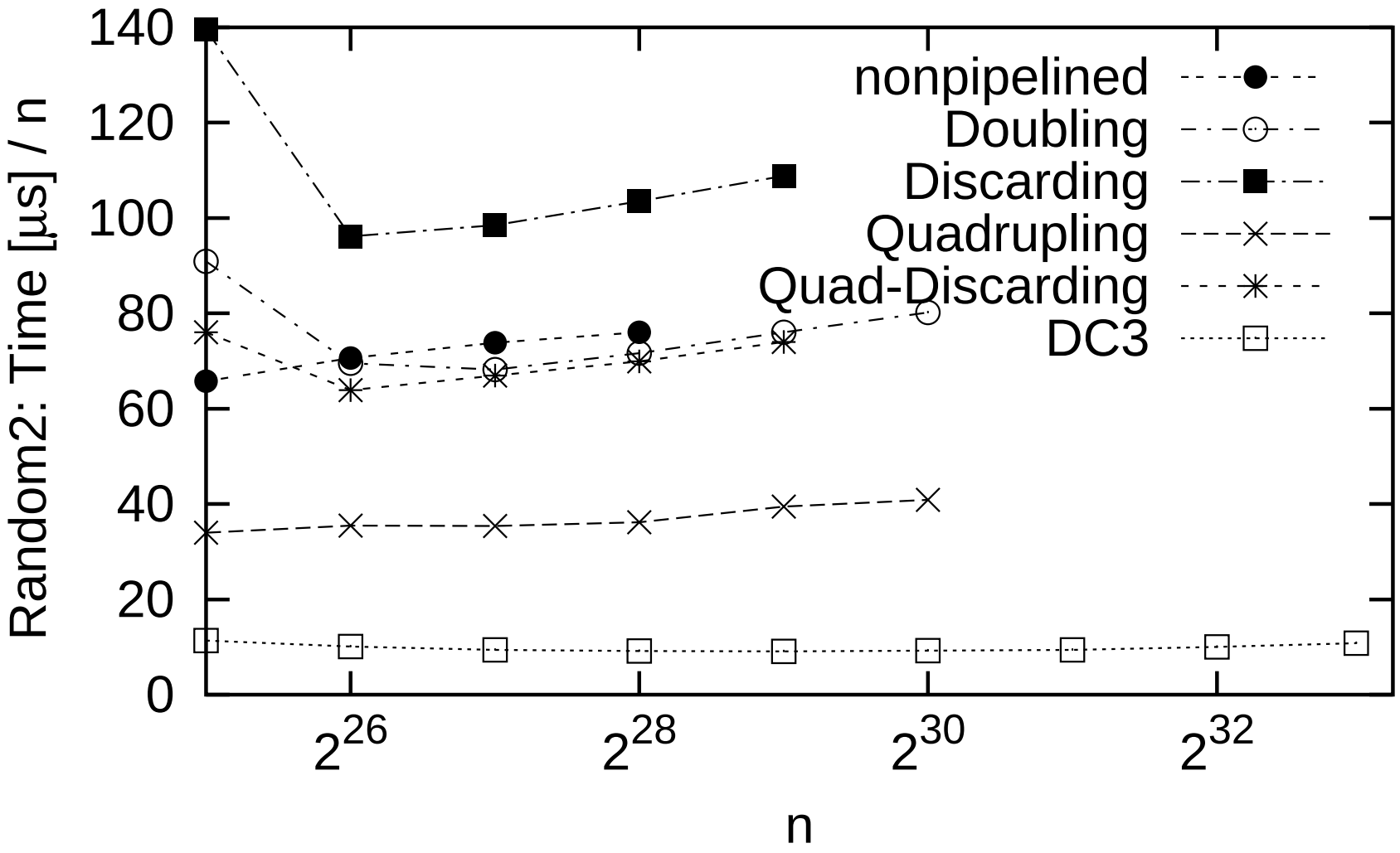
- Tune pipelined sorters
- Go parallel
- Larger difference covers for first iteration?
- Will discarding help for DC algorithms?

Terabytes over night?

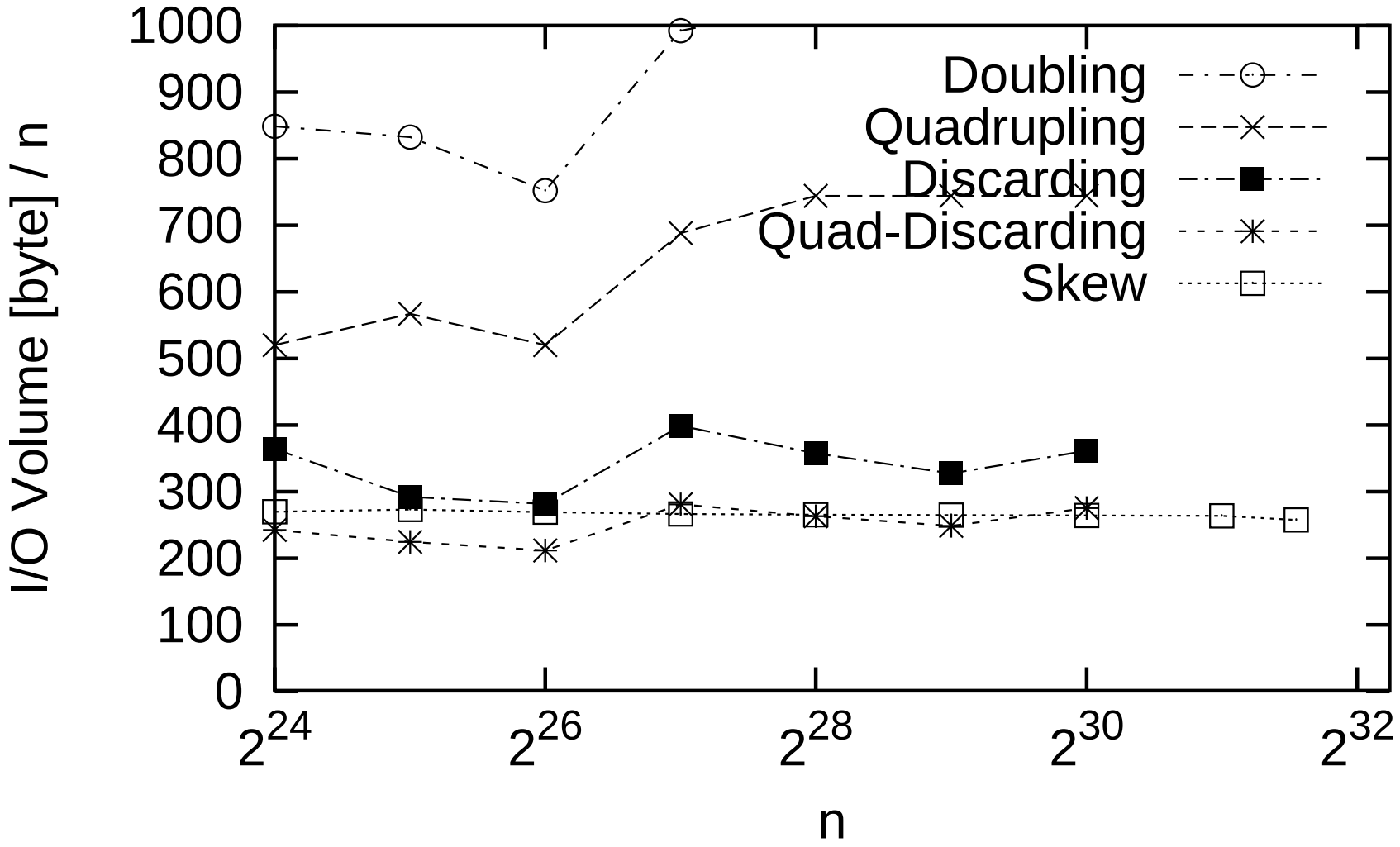
Random2 I/Os


I/O Volume [byte] / n
3500
3000
2500
2000
1500
1000
500
0
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
n
Doubling
Discarding
Quadrupling
Quad-Discarding
Skew
nonpipelined

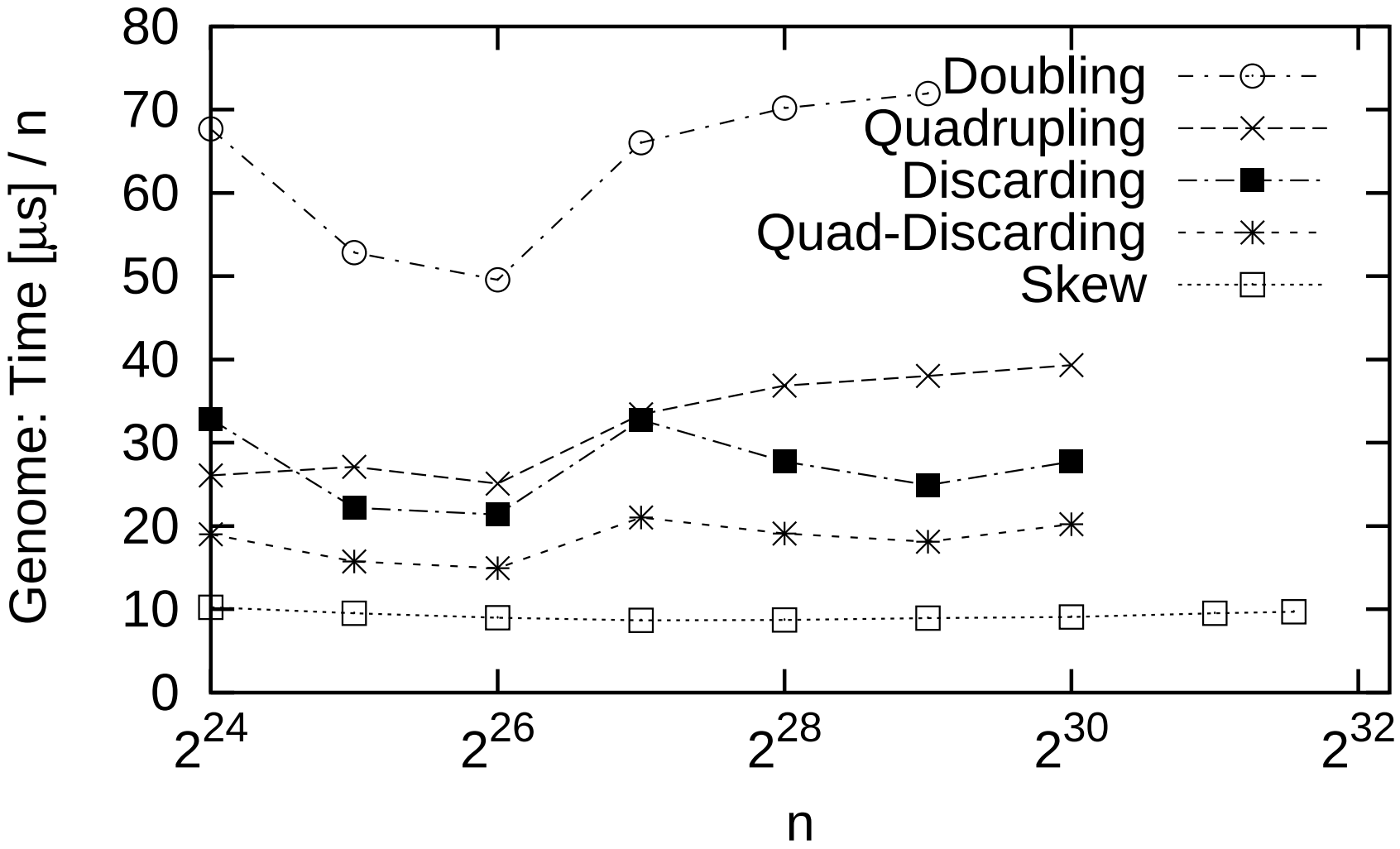
Random2 Time

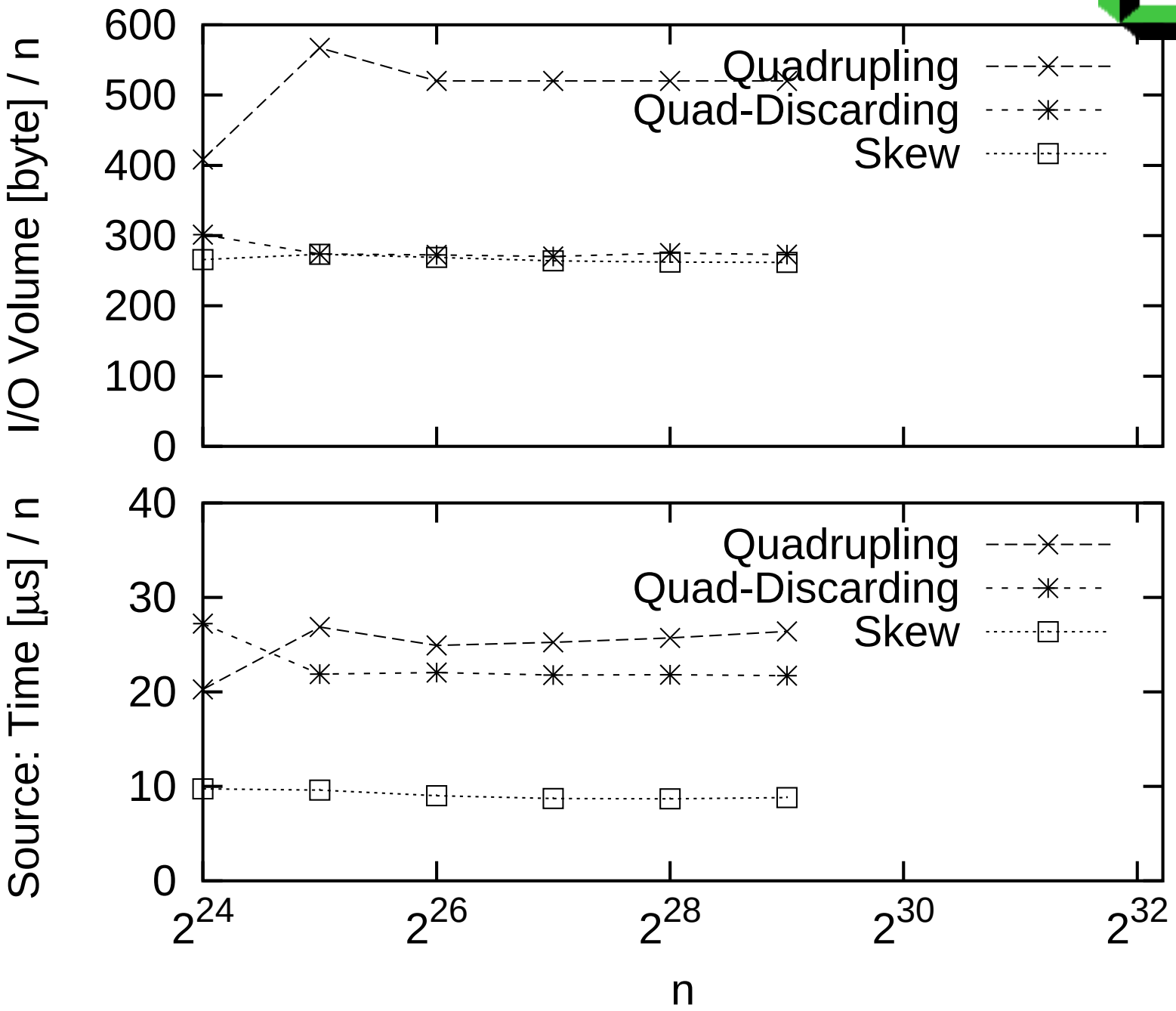


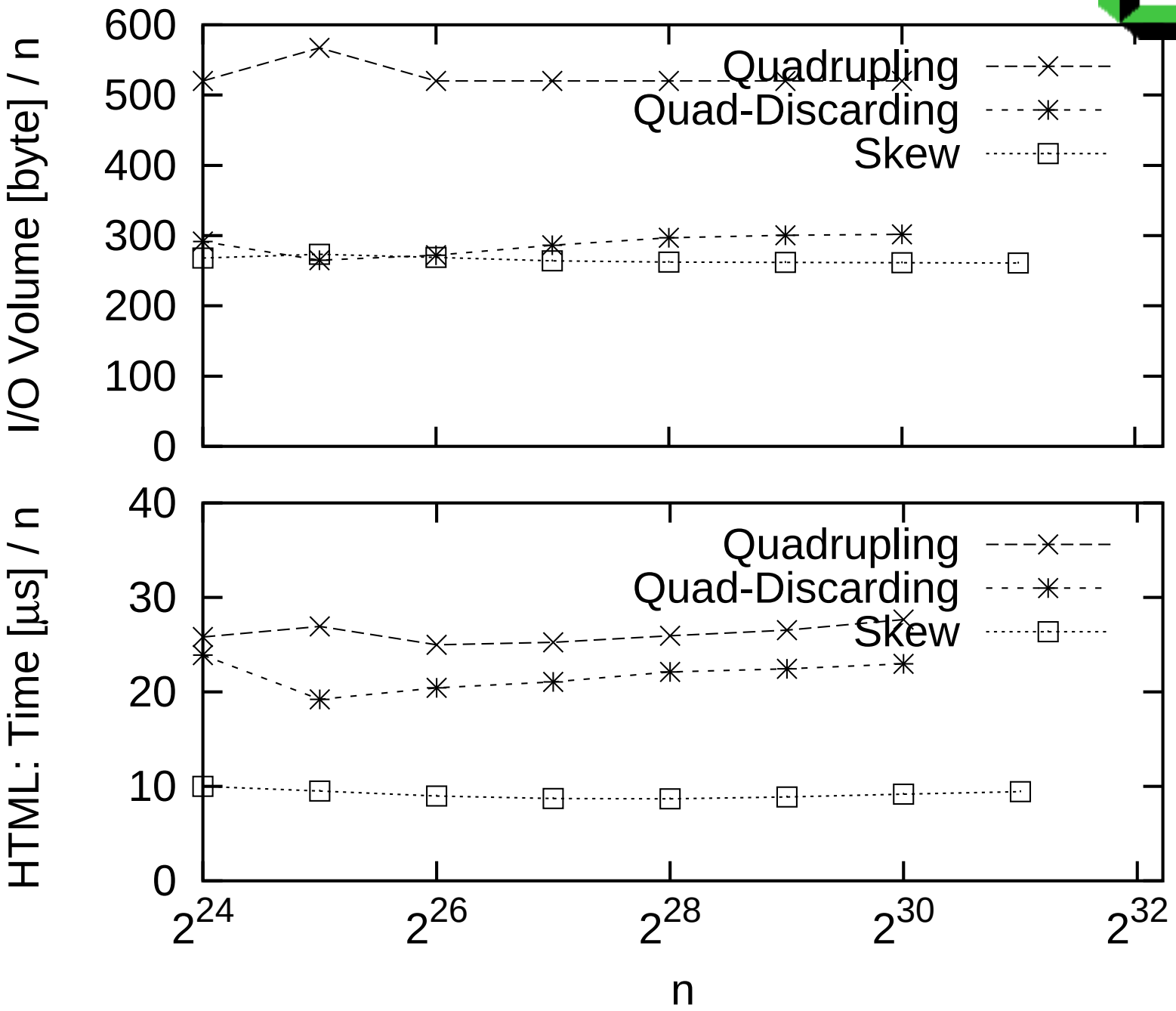
Genome I/Os


I/O Volume [byte] / n
1000
900
800
700
600
500
400
300
200
100
0
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
n
Doubling
Quadrupling
Discarding
Quad-Discarding
Skew

Genome Time


Genome: Time [µs] / n
80
70
60
50
40
30
20
10
0
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
n
Doubling
Quadrupling
Discarding
Quad-Discarding
Skew


I/O Volume [byte] / n
600
500
400
300
200
100
0
Quadrupling
Quad-Discarding
Skew
Source: Time [µs] / n
40
30
20
10
0
Quadrupling
Quad-Discarding
Skew
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
n


I/O Volume [byte] / n
600
500
400
300
200
100
0
Quadrupling
Quad-Discarding
Skew
HTML: Time [µs] / n
40
30
20
10
0
Quadrupling
Quad-Discarding
Skew
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
n