Helsinki, 8 December 2003

Title:
The current truth about heaps

Speaker:
Jyrki Katajainen

Co-workers:
Claus Jensen and Fabio Vitale

This talk is about the heaps we all love. I will explain how the heap functions are implemented in the CPH STL program library. The main contribution of the work done by my co-workers and myself is an experimental evaluation of various heap variants proposed in the computing literature. We have also done micro-benchmarking which gives some directions for future research.

These slides are available at
`http://www.cphstl.dk/`.

9th Scandinavian Workshop on Algorithm Theory

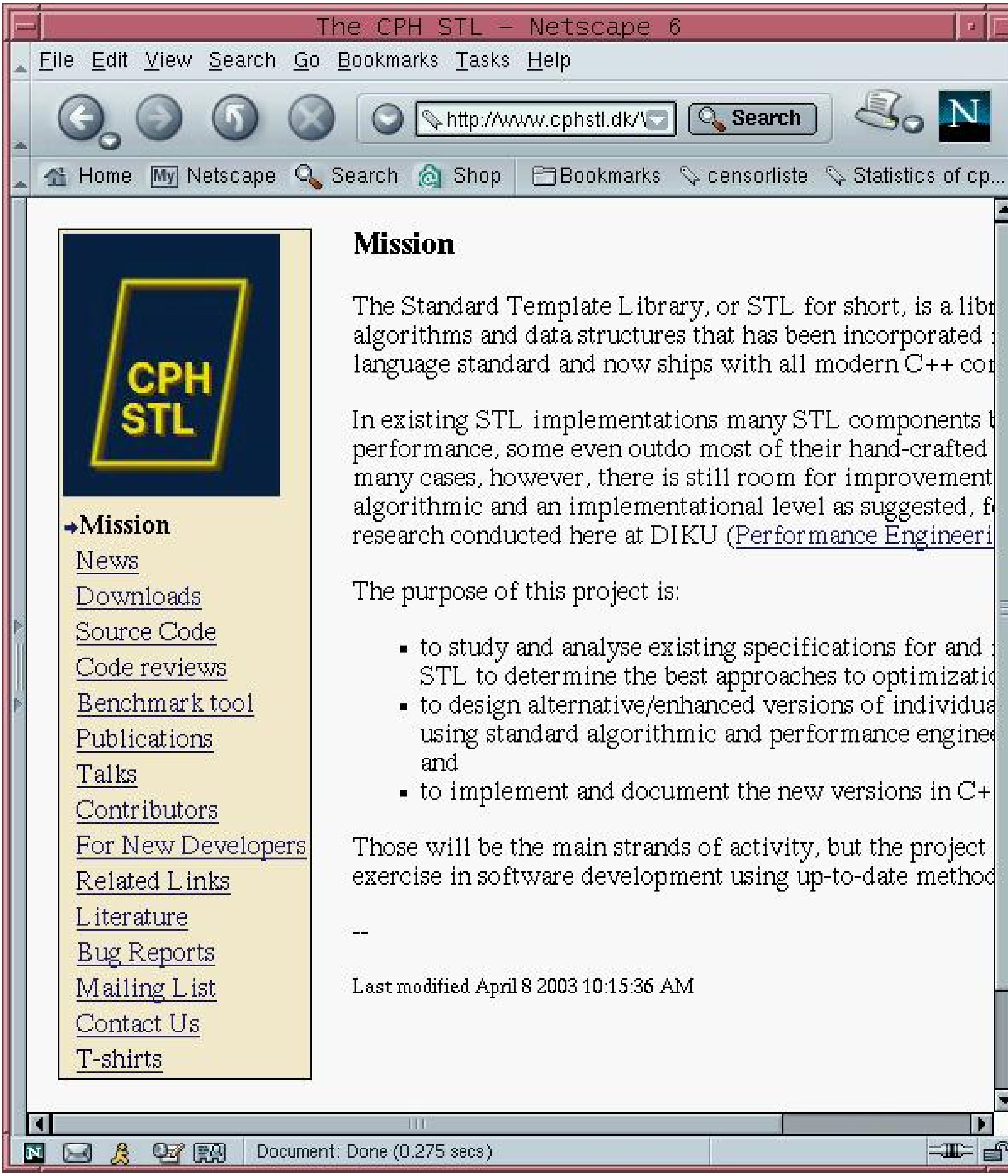
July 8–10, 2004
Louisiana Museum of Modern Art
Humlebæk, Denmark
http://swat.diku.dk/

Deadline for submission:
February 10, 2004 at noon (GMT)

Notification of authors:
March 23, 2004

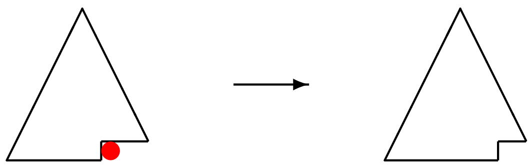
Final version due:
April 20, 2004

End of early registration:
May 4, 2004

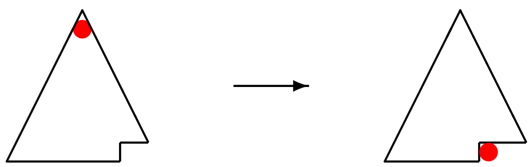

The CPH STL - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www.cphstl.dk/\
Search
Home
Netscape
Search
Shop
Bookmarks
censorliste
Statistics of cp...
CPH STL
Mission
News
Downloads
Source Code
Code reviews
Benchmark tool
Publications
Talks
Contributors
For New Developers
Related Links
Literature
Bug Reports
Mailing List
Contact Us
T-shirts
Mission
The Standard Template Library, or STL for short, is a libr
algorithms and data structures that has been incorporated i
language standard and now ships with all modern C++ co
In existing STL implementations many STL components t
performance, some even outdo most of their hand-crafted
many cases, however, there is still room for improvement
algorithmic and an implementational level as suggested, f
research conducted here at DIKU (Performance Engineeri
The purpose of this project is:
to study and analyse existing specifications for and i
STL to determine the best approaches to optimizatio
to design alternative/enhanced versions of individua
using standard algorithmic and performance enginee
and
to implement and document the new versions in C+
Those will be the main strands of activity, but the project
exercise in software development using up-to-date method
--
Last modified April 8 2003 10:15:36 AM
Document: Done (0.275 secs)

Heap functions in the STL

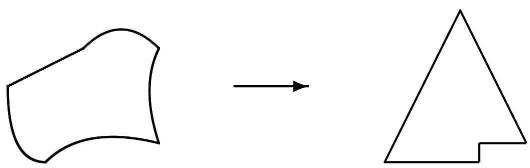
void
push_heap(position A, position Z, ordering f);

Effect:  at most $\log_2 n$ comparisons

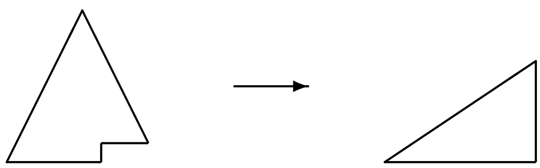
void
pop_heap(position A, position Z, ordering f);

Effect:  at most $2\log_2 n$ comparisons

void
make_heap(position A, position Z, ordering f);

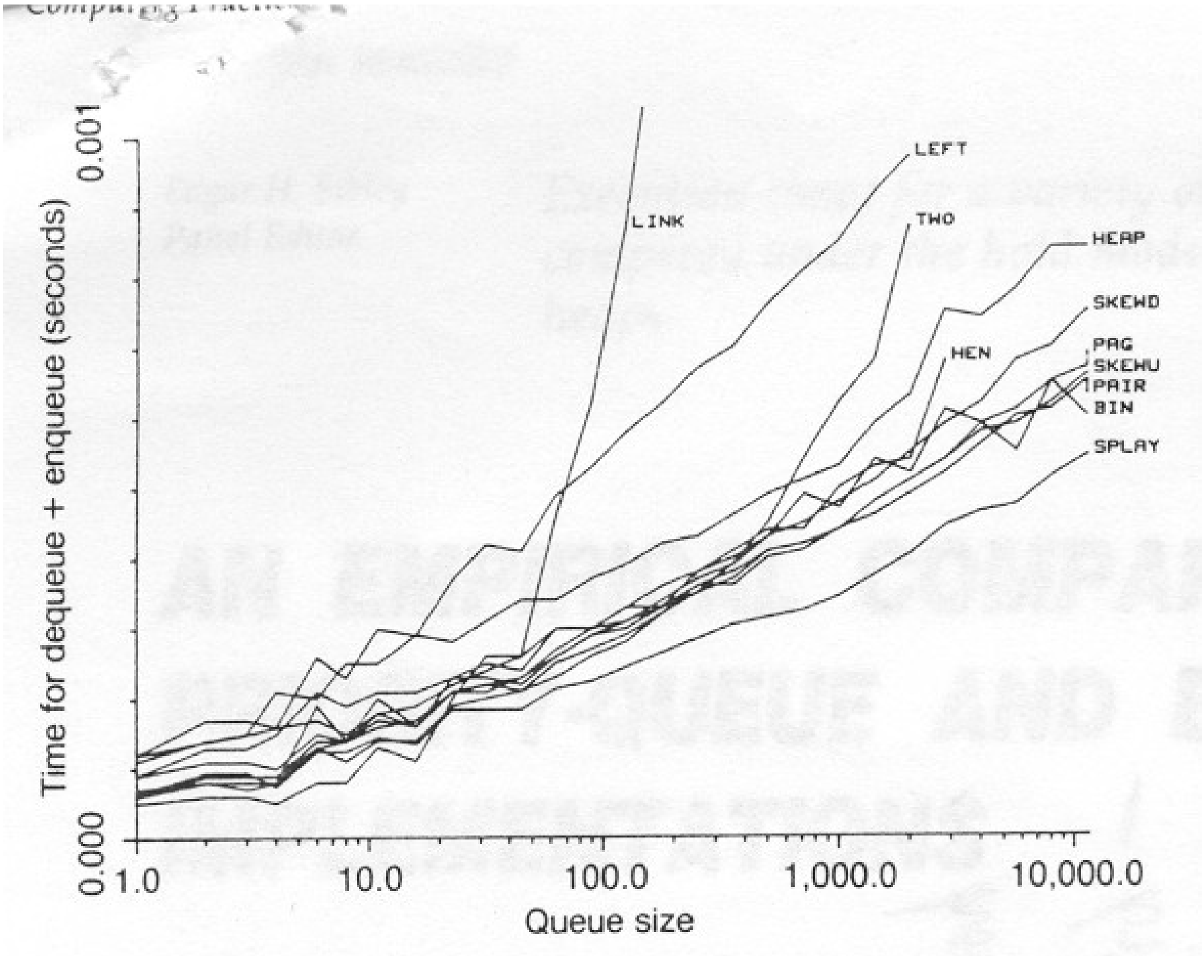
Effect:  at most $3n$ comparisons

void
sort_heap(position A, position Z, ordering f);

Effect:  at most $n\log_2 n$ comparisons

How would you do it?

Jones 1986



Operation sequence (hold model):

$push()^N[pop()push()]^K$
$e \leftarrow pop()$
increase the priority of e by $-\ln(\texttt{drand}())$
$push(e)$

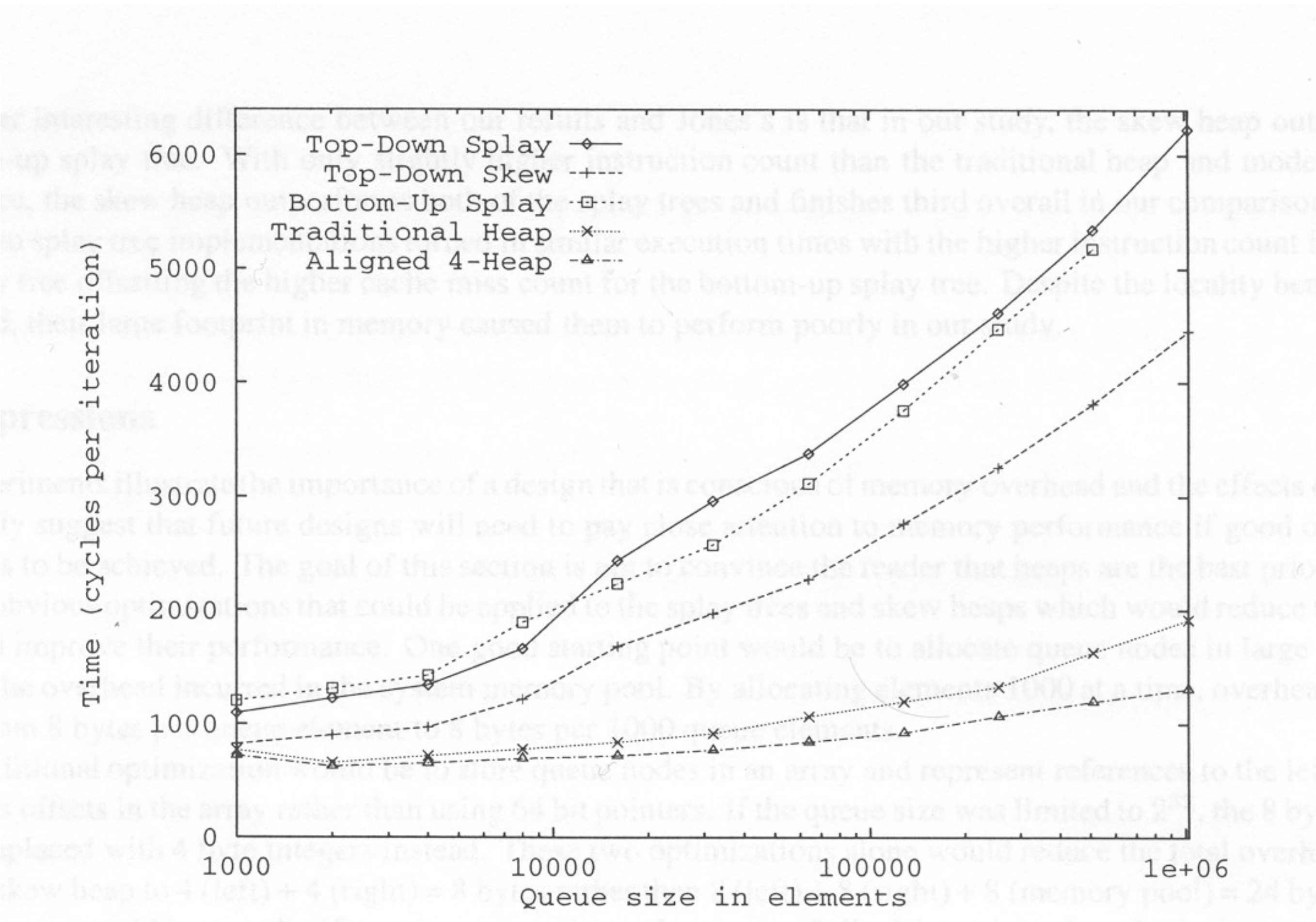
Input data:

element size: 4 B; #elements: 1–$2^{13.5}$

Environment:

computer: VAX 11/780 running UNIX (BSD 4.2); cache: 8 kB: TLB: 64 entries; compiler: Berkeley Pascal with optimization enabled

LaMarca & Ladner 1996



Operation sequence:

Hold model?

`#define NOTSORANDNUM(x) (x + RANDNUM())`

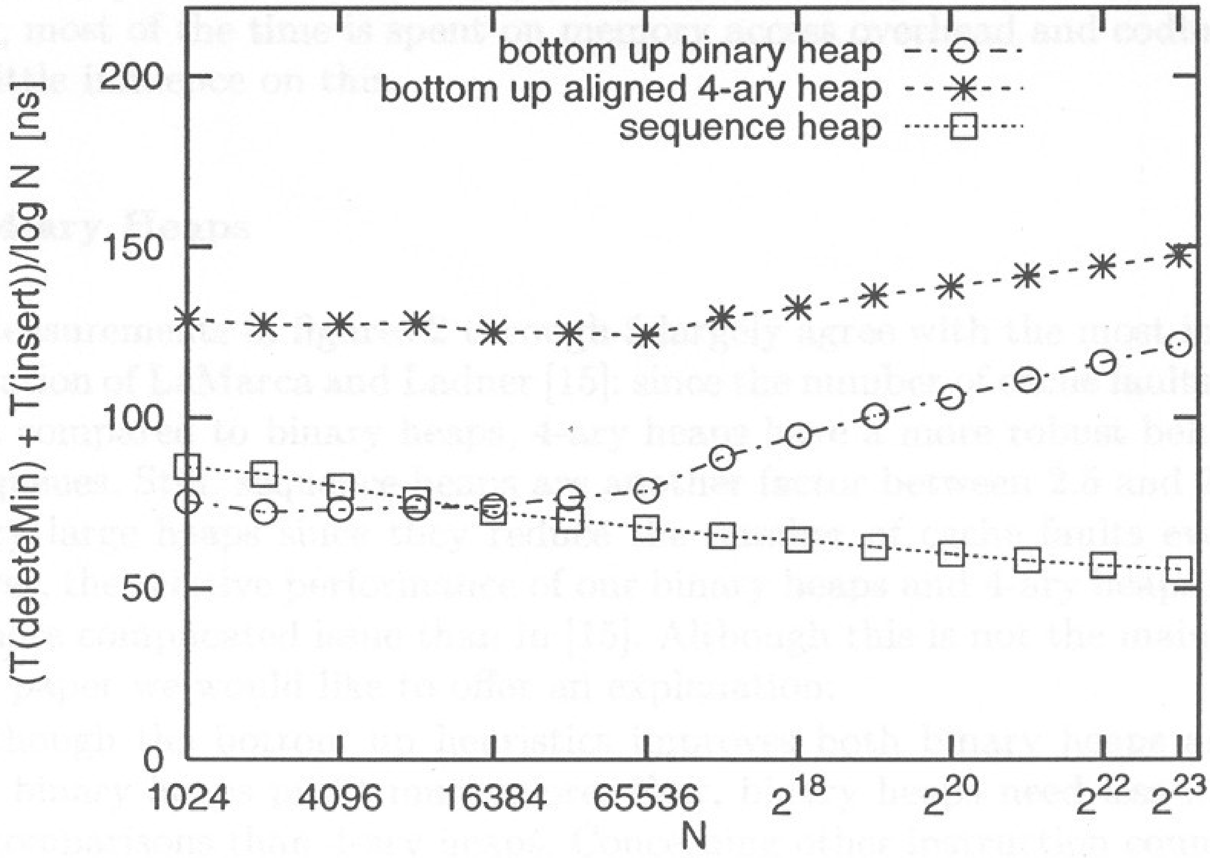
Input data:

element size: 8 B; #elements: 2^{10}–2^{23}

Environment:

computer: DEC Alphastation 250; processor: Alpha 21064A 266 MHz; L1 cache: 8 kB; L2 cache: direct-mapped, 2 MB, 32 B per line; compiler?: cc

Sanders 1999



Operation sequence:

$[push()pop()push()]^N [pop()push()pop()]^N$

Input data:

element size: 4 B, drawn randomly; satellite data: 4 B; #elements: 2^8–2^{23}

Environment:

computer: Pentium II 300 MHz; compiler `g++ -O6`

Brengel et al. 1999

Insert/Delete_min time performance of the external queues (in secs)

N [*10^6]	radix heap	array heap	buffer tree	B-tree
1	6/24	18/11	56/34	11287/259
5	17/97	74/63	148/309	66210/1389
10	35/178	353/89	201/882	-
25	85/372	724/295	311/2833	-
50	164/853	1437/645	445/6085	-
75	246/1416	2157/1005	569/9880	-
100	325/1957	2888/1408	*	-
150	478/3084	4277/2297	*	-
200	628/4036	5653/3234	*	-

Insert/Delete_min time performance of the internal queues (in secs)

N [*10^6]	Fibonacci heap	k-ary heap	pairing heap	radix heap
1	3/32	4/33	3/19	3/11
2	6/73	8/75	6/45	5/27
5	17/208	21/210	14/126	11/71
7.5	172800*/-	32/344	22/207	18/124
10	-/-	43/482	30/291	23/162
20	-/-	172800*/-	172800*/-	172800*/-

Random/Total I/Os for external queues

N [*10^6]	radix heap	array heap	buffer tree	
1	44/420	24/720	228/668	
5	422/3550	120/4560	16722/21970	
10	1124/8620	168/9440	35993/47297	
25	2780/21820	570/29520	93789/123285	
50	7798/56830	1288/66160	190147/249955	
75	12466/89370	2016/102480	286513/376625	
100	17736/124740	2776/139760	*	
150	27604/192500	4216/210080	*	
200	38284/211570	5712/284320	*	

Operation sequence:

$push()^N / pop()^N$

Input data:

element size: 4 B, drawn randomly from $[0 \,.\,.\, 10^7]$;
#elements: $1 \cdot 10^6$–$200 \cdot 10^6$

Environment:

computer: Sparc Ultra 1/143; main memory: 256 MB, 8 kB per page; local disk: 9 GB fastwide SCSI; logical block size: 64 kB; buffer size: 16 MB

Edelkamp & Stiegeler 2002

	f^0	f^1	f^2	f^3	f^4	f^5	f^6	f^7
QUICKSORT	3.86	14.59	26.73	39.04	51.47	63.44	75.68	87.89
CLEVER-QUICKSORT	3.56	12.84	23.67	33.16	43.80	54.37	64.62	75.08
BOTTOM-UP-HEAPSORT	5.73	13.49	22.60	32.05	41.59	51.14	60.62	70.11
MDR-HEAPSORT	7.14	15.39	24.37	33.82	43.04	52.63	61.87	71.02
WEAK-HEAPSORT	7.15	14.89	23.66	32.82	41.97	51.08	60.13	69.27
RELAXED-WEAK-HEAPSORT	8.29	15.96	24.63	33.51	42.32	51.33	60.06	68.83
GREEDY-WEAK-HEAPSORT	9.09	16.60	25.24	34.12	43.03	51.77	60.72	69.78
QUICK-HEAPSORT	6.35	15.89	26.20	36.98	47.59	58.16	69.37	79.59
QUICK-WEAK-HEAPSORT	6.06	14.49	23.86	33.49	43.30	52.99	62.85	72.54
CLEVER-HEAPSORT	5.30	14.01	23.66	33.65	43.95	53.79	63.60	73.94
CLEVER-WEAK-HEAPSORT	5.97	13.82	22.83	31.95	41.31	50.40	59.58	69.61

Operation sequence:

$make(N)[pop()]^N$

Input data:

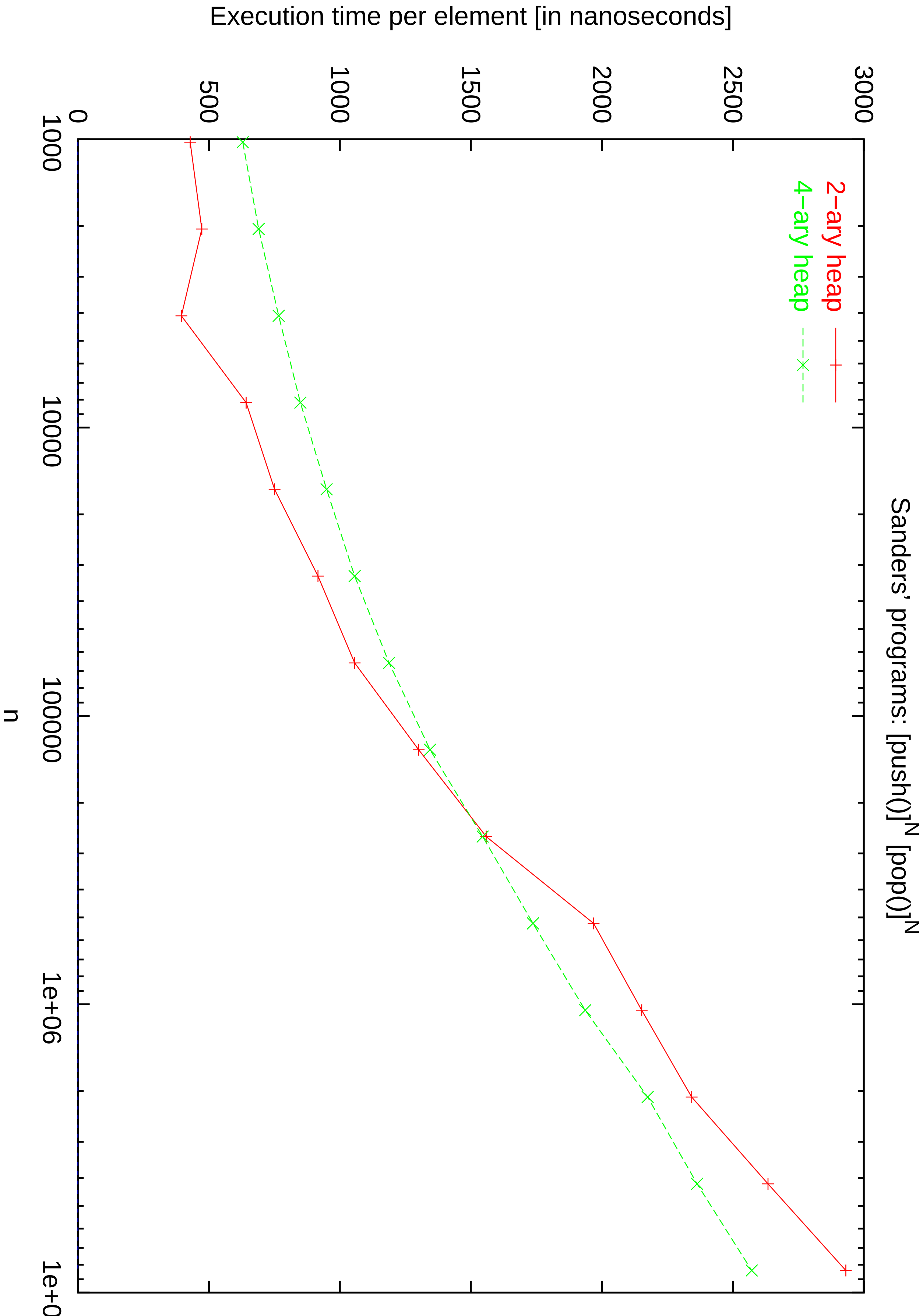
element size: 4 B, floating point numbers drawn randomly; #elements: 10^6; ordering: $f^0(x) = x$ and $f^i(x) = \ln(f^{i-1}(x+1))$ for $i > 0$

Environment:

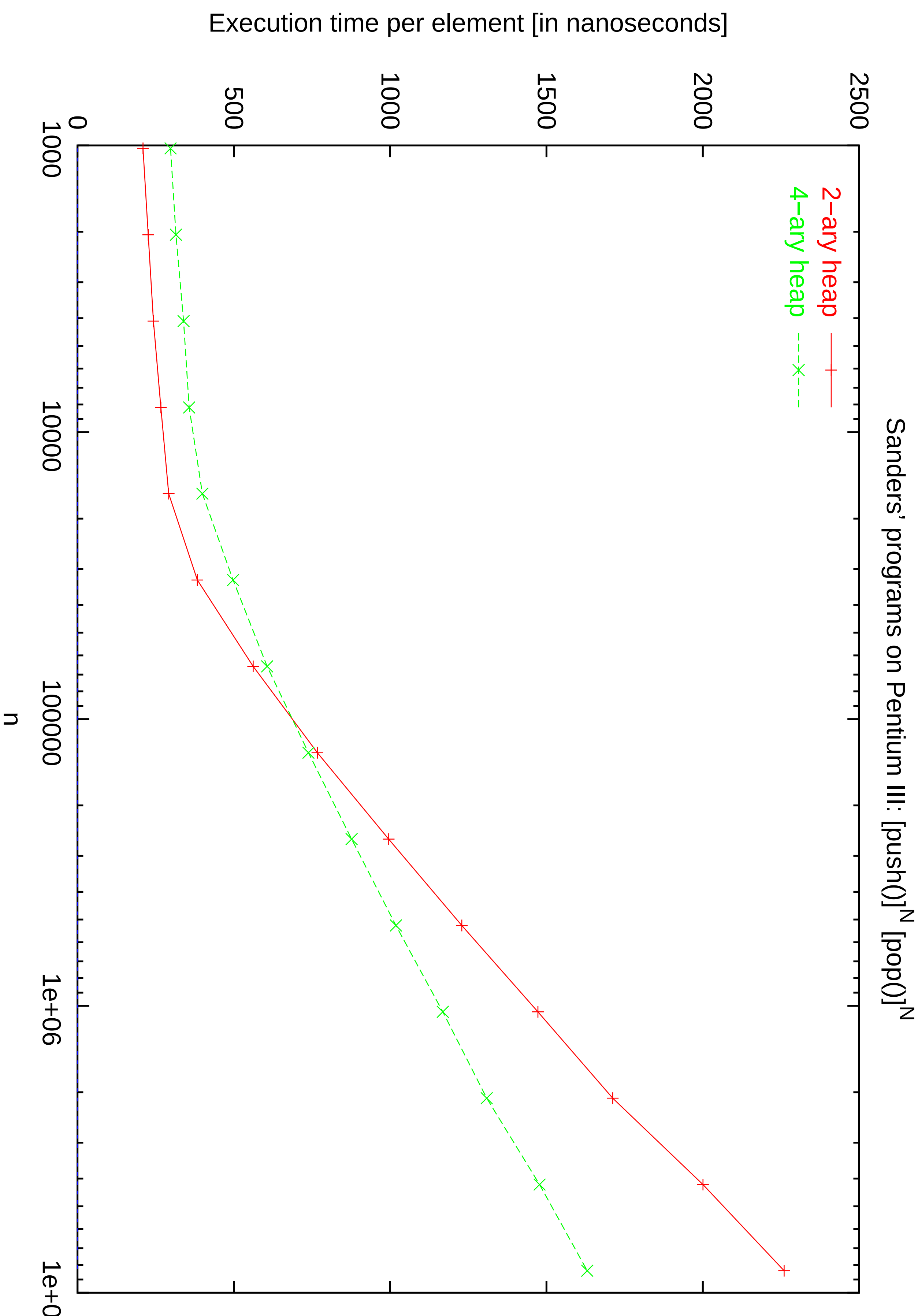
computer: Pentium III 450 MHz; compiler `g++ -O2`

How would you do it now?

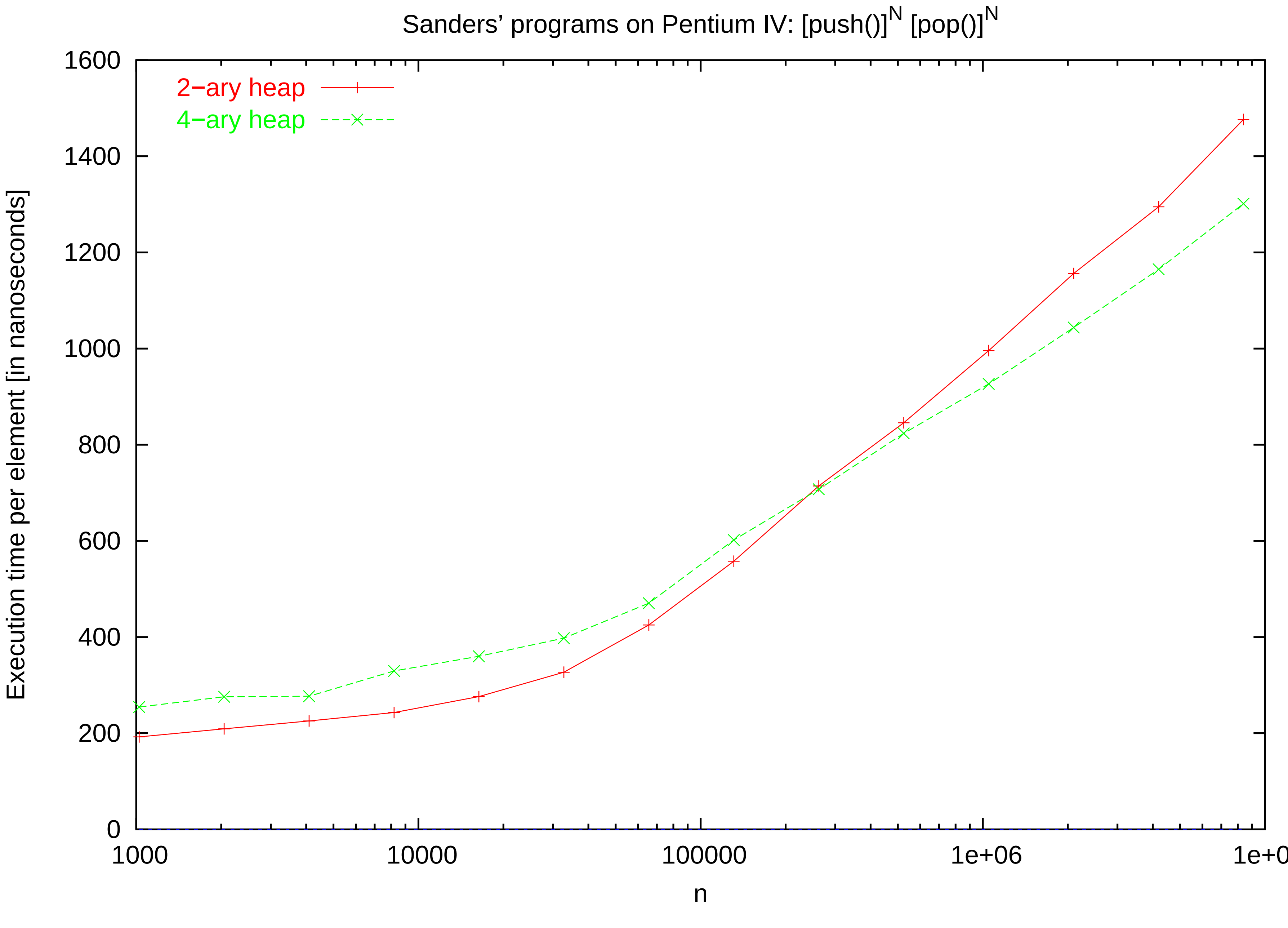
Sanders' programs on Pentium II



Sanders' programs on Pentium III



Sanders' programs on Pentium IV



Cost of unsigned int operations

initializations	instruction	unsigned int
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} .. 2^{24}$ 4.1–4.7 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} .. 2^{14}$ 7.3–8.9 ns $n = 2^{15}$ 12 ns $n = 2^{16}$ 29 ns $n = 2^{16} .. 2^{22}$ 62–63 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} .. 2^{24}$ 3.3–3.8 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} .. 2^{15}$ 3.3–4.1 ns $n = 2^{16}$ 23 ns $n = 2^{17} .. 2^{22}$ 45–55 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$r \leftarrow (a[i] < x)$	$n = 2^{10} .. 2^{24}$ 5.3–5.8 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$r \leftarrow (\ln(a[i]) < \ln(x))$	$n = 2^{10} .. 2^{24}$ 580–610ns

Cost of bigint operations

initializations	instruction	bigint
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} \,..\, 2^{21}$ 60–66 ns $n = 2^{22}$ 290 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$a[i] \leftarrow x$	$n = 2^{10} \,..\, 2^{12}$ 75–78 ns $n = 2^{13}$ 117 ns $n = 2^{14}$ 229 ns $n = 2^{15} \,..\, 2^{20}$ 297–318 ns $n = 2^{21} \,..\, 2^{22}$ 748–752 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} \,..\, 2^{22}$ 18–21 ns
$p \leftarrow 617$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$x \leftarrow a[i]$	$n = 2^{10} \,..\, 2^{12}$ 24 ns $n = 2^{13}$ 83 ns $n = 2^{14}$ 180 ns $n = 2^{15} \,..\, 2^{22}$ 230–260 ns
$p \leftarrow 1$ $a[i] \leftarrow 0$ $x \leftarrow 2^{20}$	$r \leftarrow (a[i] < x)$	$n = 2^{10} \,..\, 2^{22}$ 13–16 ns

Other current research

Pointer-based methods:

hopelessly slow

$\longrightarrow$ theoretical computer science

Methods with good amortized bounds:

terrible worst case

$\longrightarrow$ not relevant for us

Methods with few element moves:

bad cache behaviour

$\longrightarrow$ not good for us

External-memory methods:

high constants

$\longrightarrow$ relevant only for very large data sets

Cache-oblivious methods:

huge constants

$\longrightarrow$ theoretical computer science

Our policy-based framework

```
template <arity d, typename position, typename ordering>
class heap_policy {
public:
  typedef typename
    std::iterator_traits<position>::difference_type index;
  typedef typename
    std::iterator_traits<position>::difference_type level;
  typedef typename
    std::iterator_traits<position>::value_type element;

  template <typename integer>
  heap_policy(integer n = 0);

  bool is_root(index) const;
  bool is_first_child(index) const;
  index size() const;
  level depth(index) const;
  index root() const;
  index leftmost_leaf() const;
  index last_leaf() const;
  index first_child(index) const;
  index parent(index) const;
  index ancestor(index, level) const;
  index top_some_absent(position, index,
    const ordering&) const;
  index top_all_present(position, index,
    const ordering&) const;
  void update(position, index, const element&);
  void erase_last_leaf(position, const ordering&);
  void insert_new_leaf(position, const ordering&);
private:
  index n;
};
```

Input data

	cheap move	expensive move
cheap comparison	unsigned int	bigint
expensive comparison	unsigned int In comparison	(int, bigint) In comparison

One new old idea: local heaps

Our solution for sort_heap()

In-place mergesort by Katajainen, Pasanen, and Teuhola [1996]

Fine-tuning not yet implemented

Almost as fast as quicksort, see CPH STL Report 2003-2

Our solution for make_heap()

Depth-first heap construction by Bojesen, Katajainen, and Spork [2000]

Almost optimal in all respects

Other work:

less element comparisons

$\longrightarrow$ theoretical computer science

Various approaches for pop_heap()

– top-down → many element comparisons

– bottom-up → typical case good

– move-saving bottom-up → theoretical computer science

– binary-search top-down

– two-levels-at-a-time top-down

Various approaches for push_heap()

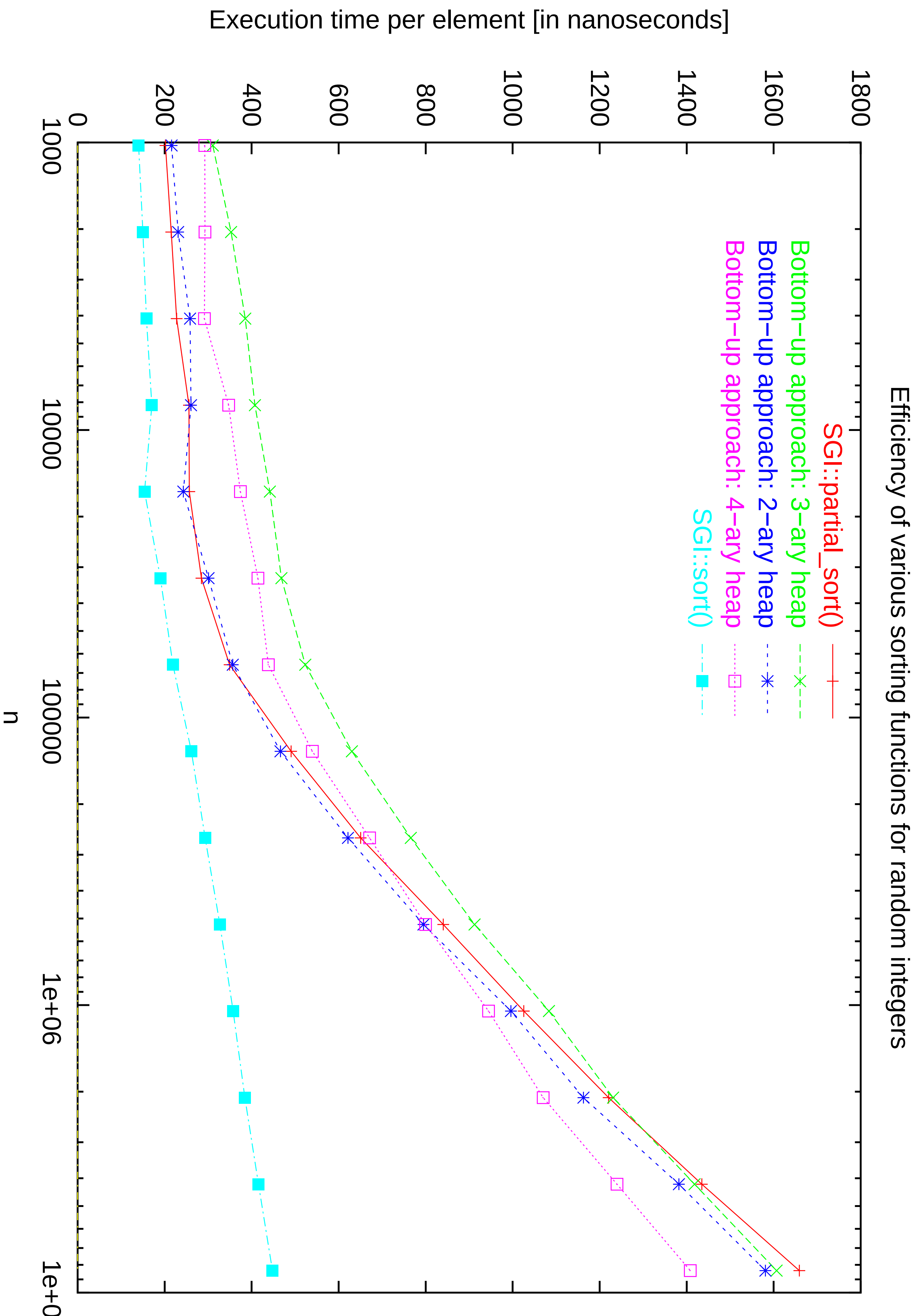
– move-saving top-down → slow

– bottom-up → typical case good

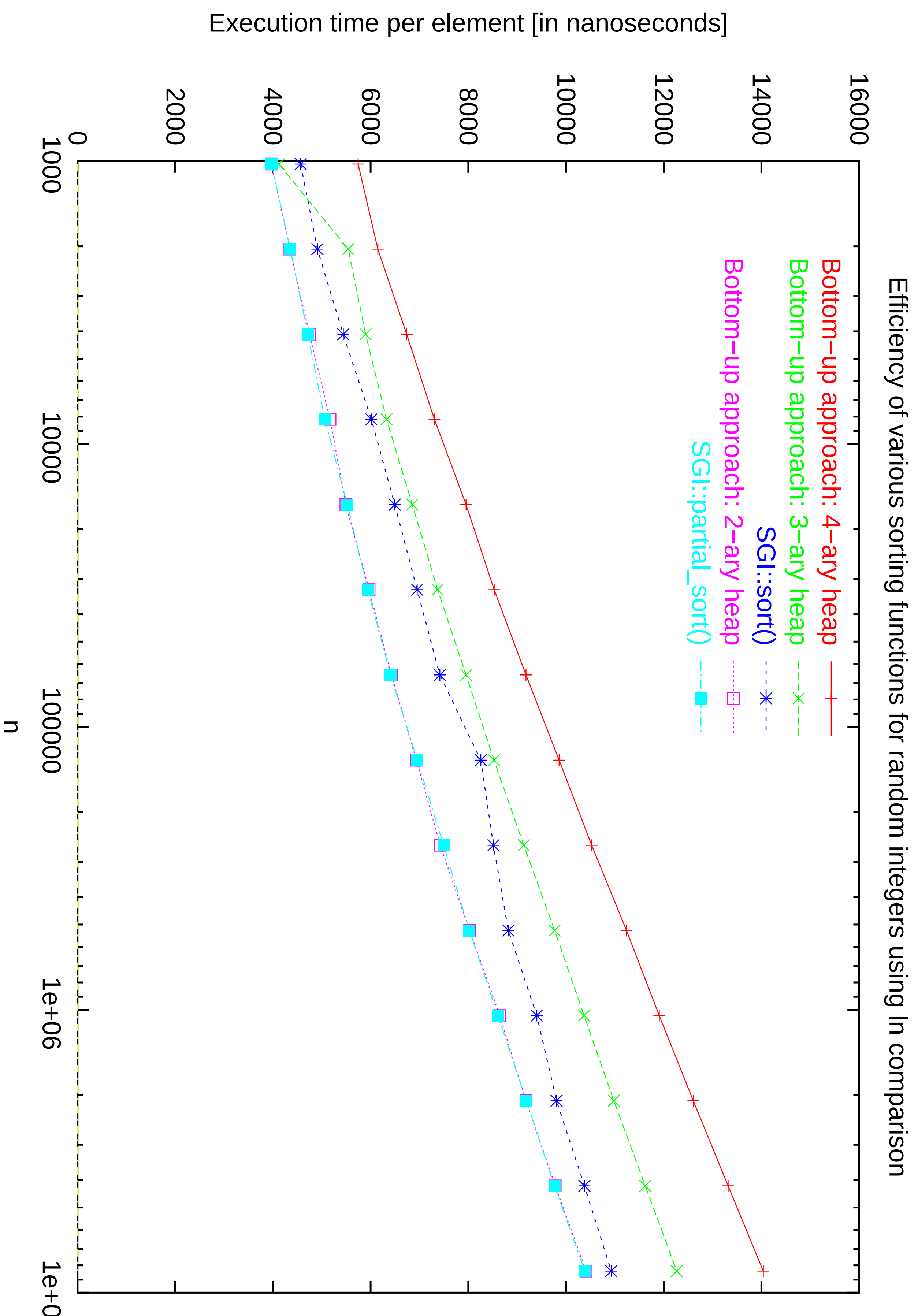
– bottom-up with buffering → complicated

– binary-search bottom-up

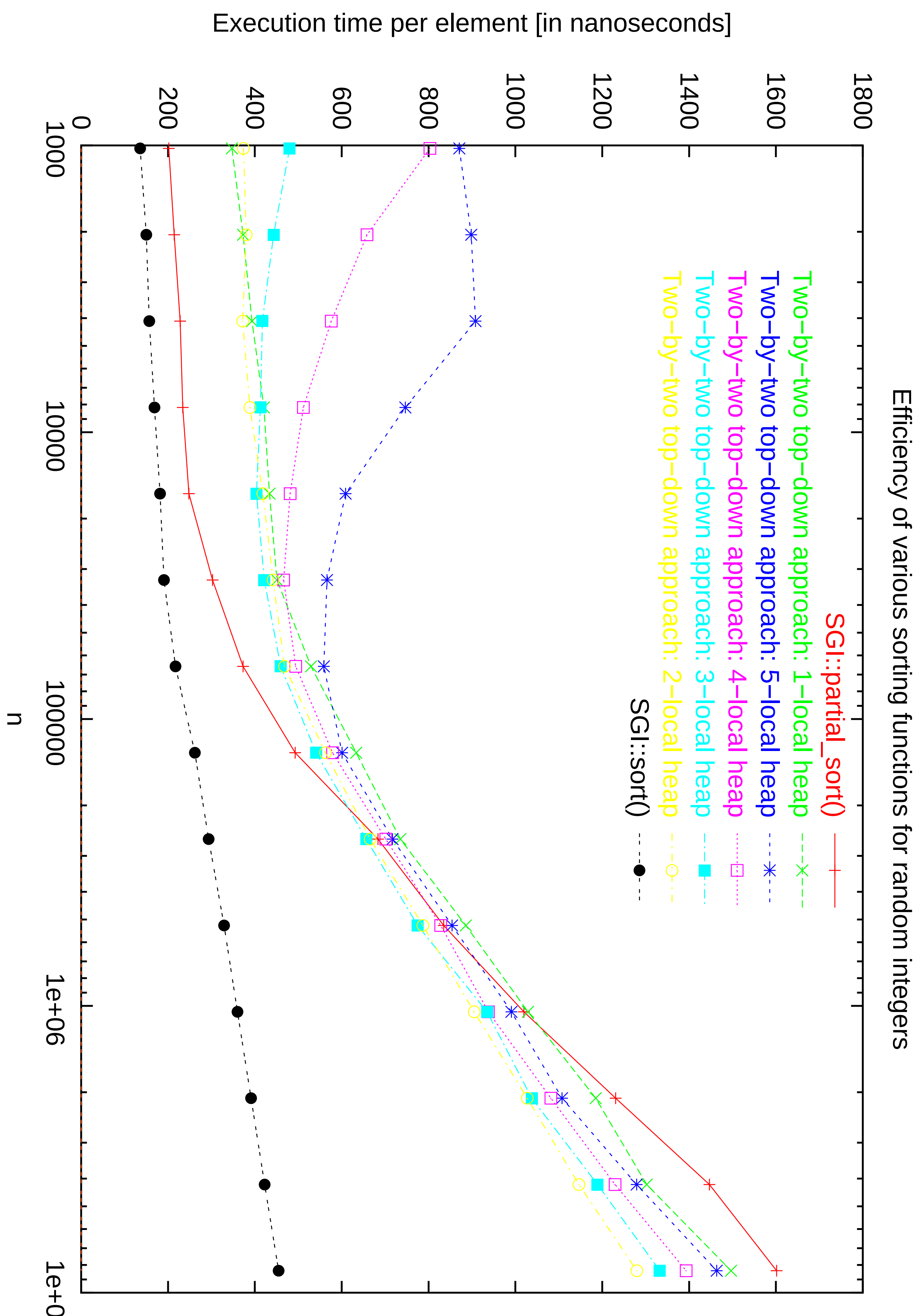
Efficiency of 2-, 3-, 4-ary heaps



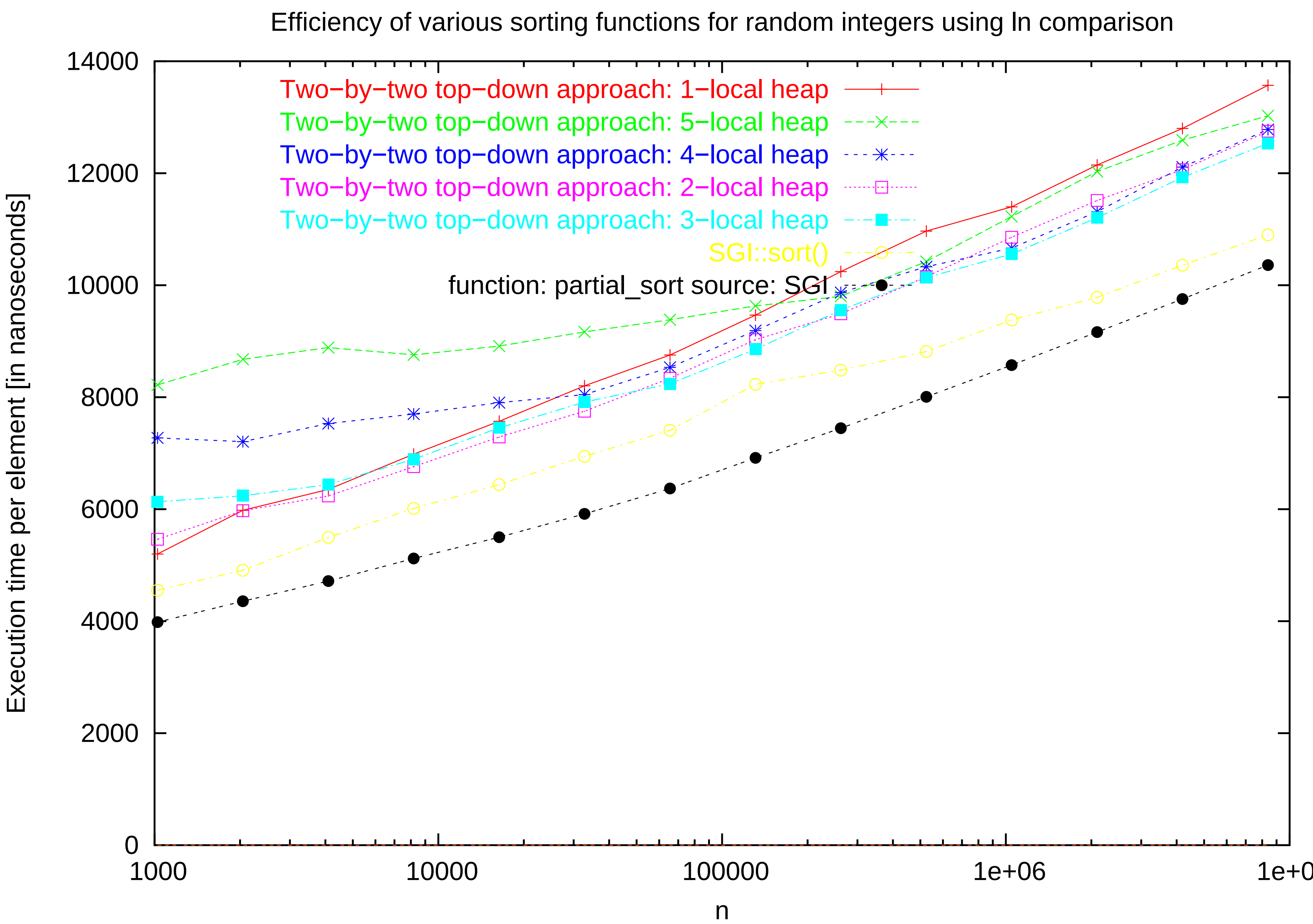
Efficiency of 2-, 3-, 4-ary heaps



Efficiency of local heaps



Efficiency of local heaps



Conclusions

– In 40 years — not much progress
– At the moment it is not clear how big the overhead of local heaps is for small problem sizes.
– Some combinations of various approaches have still to be tested.
– Code-tuning of the best approaches is still to be done.
– It takes time to develop fast library routines.
– How does technology influence on the efficiency of the library routines?

Exercise of the week

How many element comparisons incur the operation sequence

$$[push() \mid pop()]^N$$

in the worst case? Or what is the amortized complexity of each of these operations?

$1.5N \log_2 N$ is an obvious upper bound and $N \log_2 N$ an obvious lower bound.

Recall that the operation sequence

$$make(N)[pop()]^N$$

requires about $1.5N \log_2 N$ element comparisons.