Tutorial: Algorithm Engineering for Big Data

Peter Sanders, Karlsruhe Institute of Technology

Efficient algorithms are at the heart of any nontrivial computer application. But how can we obtain innovative algorithmic solutions for demanding application problems with exploding input sizes using complex modern hardware and advanced algorithmic techniques?

This tutorial proposes algorithm engineering as a methodology for taking all these issues into account. Algorithm engineering tightly integrates modeling, algorithm design, analysis, implementation and experimental evaluation into a cycle resembling the scientific method used in the natural sciences. Reusable, robust, flexible, and efficient implementations are put into algorithm libraries. Benchmark instances provide further coupling to applications.

We begin with examples representing fundamental algorithms and data structures with a particular emphasis on large data sets. We first look at sorting in detail. Then we will have shorter examples for full text indices, priority queue data structures, route planning, graph partitioning, and minimum spanning trees. We will also give examples of future challenges centered on particular big data applications like genome sequencing and phylogenetic tree reconstruction, particle tracking at the CERN LHC, and the SAP-HANA database.

Further Information

Duration: half-day

Intended Audience: Practitioners with some basic background in algorithms (2nd semester computer science in most German universities)

Slides are attached. Some images with unclear copyright are removed.
Algorithm Engineering for Big Data

Peter Sanders
Overview

- A detailed explanation of algorithm engineering with sorting for (more or less) big inputs as a throughgoing example

- More Big Data examples from my group

Algorithmics

= the systematic design of efficient software and hardware

computer science

algorithmics ➔ efficient
logics ➔ correct

soft- & hardware

theoretical

practical
(Caricatured) Traditional View: Algorithm Theory

Theory

Design

Analysis

Perf. guarantees

Practice

Implementation

Applications

Deduction
### Gaps Between Theory & Practice

<table>
<thead>
<tr>
<th>Theory</th>
<th>→</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>simple appl. model</td>
<td>complex</td>
</tr>
<tr>
<td>simple</td>
<td>simple machine model</td>
<td>real</td>
</tr>
<tr>
<td>complex</td>
<td>algorithms FOR simple</td>
<td></td>
</tr>
<tr>
<td>advanced</td>
<td>data structures</td>
<td>arrays, ...</td>
</tr>
<tr>
<td>worst case, max</td>
<td>complexity measure</td>
<td>inputs</td>
</tr>
<tr>
<td>asympt., ( O(\cdot) )</td>
<td>efficiency</td>
<td>42% constant factors</td>
</tr>
</tbody>
</table>
Algorithmics as Algorithm Engineering

- Algorithm engineering
- Models
- Design
- Analysis
  - Deduction
  - Perf.– guarantees
- Experiments
Algorithmics as Algorithm Engineering

- Algorithmic engineering
- Models
- Design
- Falsifiable hypotheses
- Induction
- Analysis
- Deduction
- Performance guarantees
- Experiments
- Implementation

[1]
Algorithmics as Algorithm Engineering

algorithm engineering

realistic models

real. design

falsifiable hypotheses
induction

experiments

implementation

real. analysis
deduction

perf.– guarantees

[1]
Algorithmics as Algorithm Engineering

**algorithm engineering**

- realistic models
- design
- analysis
- falsifiable hypotheses
- induction
- implementation
- experiment
- real Inputs
- perf.- guarantees
- algorithm libraries

[1]
Algorithmics as Algorithm Engineering

Algorithm engineering

realistic models

real Inputs

design

definition hypotheses

falsifiable

induction

experiments

algorithm engineering

analytical analysis

operation perf. guarantees

deduction

algorithm libraries

appl. engin.
Goals

- bridge gaps between theory and practice
- accelerate transfer of algorithmic results into applications
- keep the advantages of theoretical treatment: generality of solutions and reliability, predictability from performance guarantees
Bits of History

1843– Algorithms in theory and practice

1950s, 1960s Still infancy

1970s, 1980s Paper and pencil algorithm theory.

Exceptions exist, e.g., [D. Johnson]

1986 Term used by [T. Beth],

lecture “Algorithmentechnik” in Karlsruhe.

1988– Library of Efficient Data Types and Algorithms (LEDA) [2]

1997– Workshop on Algorithm Engineering

⇒ ESA applied track [G. Italiano]

1997 Term used in US policy paper [Aho, Johnson, Karp, et. al]

1998 Alex workshop in Italy ⇒ ALENEX
## Realistic Models

<table>
<thead>
<tr>
<th>Theory</th>
<th>←→</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>←→</td>
<td>complex</td>
</tr>
<tr>
<td>simple</td>
<td>←→</td>
<td>real</td>
</tr>
</tbody>
</table>

- **Careful** refinements
- Try to preserve (partial) analyzability / simple results
Sorting – Model

Comparison based

true/false

arbitrary
e.g. integer

full information
Advanced Machine Models[3]

RAM / von Neumann

- Registers
- ALU
- Freely programmable
- Large memory
- Count instructions

External

- Registers
- ALU
- Fast memory
- Capacity M
- Freely programmable
- Count (block) I/Os

[3]
Distributed Memory

(also) determine communication volume
Parallel Disks

[5]
Set Associative Caches

- Cache sets
- Cache lines of the memory
- Main memory
Branch Prediction

[7]
Hierarchical Parallel External Memory
Graphics Processing Units
Combining Models?

- design / analyze one aspect at a time
- hierarchical combination
- autotuning?
Design

of algorithms that work well in practice

- simplicity
- reuse
- constant factors
- exploit easy instances
Design – Sorting

- Simplicity
- Reuse: disk scheduling, prefetching, load balancing, sequence partitioning [10, 5, 11, 8]
- Constant factors: detailed machine model (caches, TLBs, registers, branch prediction, ILP) [3, 7]
- Instances: randomization for difficult instances [5, 8]
Example: External Sorting

$n$: input size

$M$: internal memory size

$B$: block size
Procedure externalMerge \((a, b, c : \text{File of Element})\)

\[
x := a.\text{readElement} \quad // \text{Assume emptyFile.readElement} = \infty
y := b.\text{readElement}
\]

for \(j := 1 \) to \(|a| + |b|\) do

\begin{align*}
\text{if } x \leq y & \quad \text{then} \quad c.\text{writeElement}(x); \quad x := a.\text{readElement} \\
\text{else} & \quad c.\text{writeElement}(y); \quad y := b.\text{readElement}
\end{align*}
External Binary Merging

read file $a$: $\approx |a|/B$.
read file $b$: $\approx |b|/B$.
write file $c$: $\approx (|a| + |b|)/B$.
overall:
$\approx 2 \frac{|a| + |b|}{B}$
Run Formation

Sort input pieces of size $M$

I/Os: $\approx 2\frac{n}{B}$
Sorting by External Binary Merging

```
made_things___as_simple_as________________________possible_bu__t_no_simpler
formRuns  formRuns  formRuns  formRuns
___aeghikmnst  ___aailmpsss  ___aailmpsss  ___eilmnoprst
merge
___aaaeeghiiklmmnpsssst  ___bbbeiillmnoopprsssttu
merge
________aaabbeeeeghiiiiiklllmmmnnnoopprrsssssssttu

Procedure externalBinaryMergeSort
```

```
run formation
while more than one run left do
merge pairs of runs
output remaining run
```

// I/Os: ≈
// 2n/B
// \( \lceil \log \frac{n}{M} \rceil \times \)
// 2n/B

\[ \sum : 2 \frac{n}{B} \left( 1 + \lceil \log \frac{n}{M} \rceil \right) \]
Example Numbers: PC 2013

\[ n = 2^{40} \text{ Byte (1 TB)} \]
\[ M = 2^{33} \text{ Byte (8 GB)} \]
\[ B = 2^{22} \text{ Byte (4 MB)} \]

one I/O needs \(2^{-5}\) s (31.25 ms)

\[
\text{time} = 2 \frac{n}{B} \left(1 + \left\lceil \log \frac{n}{M} \right\rceil \right) \cdot 2^{-5}s \\
= 2 \cdot 2^{18} \cdot (1 + 7) \cdot 2^{-5}s = 2^{17}s \approx 36h
\]

Idea: 8 passes \(\leadsto\) 2 passes
Multway Merging

Procedure multiwayMerge\( (a_1, \ldots, a_k, c : \text{File of Element}) \)

\begin{align*}
&\text{for } i := 1 \text{ to } k \text{ do } x_i := a_i . \text{readElement} \\
&\text{for } j := 1 \text{ to } \sum_{i=1}^{k} |a_i| \text{ do} \\
&\quad \text{find } i \in 1..k \text{ that minimizes } x_i \quad // \text{ no I/Os!, } O(\log k) \text{ time} \\
&\quad c . \text{writeElement}(x_i) \\
&\quad x_i := a_i . \text{readElement}
\end{align*}

![Diagram of multiway merging]

---

*internal buffers*
Mulitway Merging – Analysis

I/Os: read file $a_i : \approx \frac{|a_i|}{B}$.

write file $c : \approx \sum_{i=1}^{k} \frac{|a_i|}{B}$

overall:
$$\leq \frac{\approx 2 \sum_{i=1}^{k} |a_i|}{B}$$

constraint: We need $k + 1$ buffer blocks, i.e., $k + 1 < \frac{M}{B}$. 
**Sorting by Multiway-Merging**

- sort $\lceil \frac{n}{M} \rceil$ runs with $M$ elements each
  
  \[ \frac{2n}{B} \text{ I/Os} \]

- merge $\frac{M}{B}$ runs at a time
  
  \[ \frac{2n}{B} \text{ I/Os} \]

- unit a single run remains
  
  \[ \times \left\lfloor \log_{M/B} \frac{n}{M} \right\rfloor \text{ merging phases} \]

---

overall

\[ \text{sort}(n) := \frac{2n}{B} \left( 1 + \left\lfloor \log_{M/B} \frac{n}{M} \right\rfloor \right) \text{ I/Os} \]
External Sorting by Multiway-Merging

More than one merging phase?:

Not for the hierarchy main memory, hard disk.

$\frac{M}{B} > \frac{\approx 200}{\text{RAM Euro/\text{bit}}}$

reason:

$\frac{\text{Platte Euro/\text{bit}}}{\text{≈200}}$
More on Multiway Mergesort – Parallel Disks

- Randomized Striping  [5]
- Optimal Prefetching  [5]
- Overlapping of I/O and Computation  [10]
Shared Memory Multiway Mergesort

$t_0$  $t_1$  $t_2$  $t_3$
Combinations

parallel disk + shared memory: [13]

+ distributed memory: [8] stay tuned

load balancing, randomization, collective communication

+ energy: [14] stay tuned
Analysis

- Constant factors matter

- Beyond worst case analysis

- Practical algorithms might be difficult to analyze
  (randomization, meta heuristics, ... )
Analysis – Sorting

- Constant factors matter
  \((1 + o(1)) \times \text{lower bound}\)
  \([5, 8]\)
  \(\text{I/Os for parallel (disk) external sorting}\)

- Beyond worst case analysis

- Practical algorithms might be difficult to analyze
  \(\text{Open Problem:}\)
  \([5]\)
  Greedy algorithm for parallel disk prefetching \([\text{Knuth@48}]\)
Implementation

sanity check for algorithms!

Challenges

Semantic gaps:

Abstract algorithm
⇔
C++...
⇔
hardware

Small constant factors:

compare highly tuned competitors
**Example: Inner Loops Sample Sort**

```cpp
template <class T>
void findOraclesAndCount(const T* const a,
    const int n, const int k, const T* const s,
    Oracle* const oracle, int* const bucket) {
    for (int i = 0; i < n; i++)
        int j = 1;
    while (j < k) {
        j = j*2 + (a[i] > s[j]);
    }
    int b = j-k;
    bucket[b]++;
    oracle[i] = b;
}
```
Example: Inner Loops Sample Sort

```cpp
template <class T>
void findOraclesAndCountUnrolled([...]){
    for (int i = 0; i < n; i++)
        int j = 1;
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        j = j*2 + (a[i] > s[j]);
        int b = j-k;
        bucket[b]++;oracle[i] = b;
}
```
Example: Inner Loops Sample Sort

```cpp
template <class T>
void findOraclesAndCountUnrolled2([...]){  
    for (int i = n & 1; i < n; i+=2) {
        int j0 = 1;  int j1 = 1;
        T ai0 = a[i];  T ai1 = a[i+1];
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);  
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);  
        j0=j0*2+(ai0>s[j0]);  j1=j1*2+(ai1>s[j1]);
        int b0 = j0-k;
        bucket[b0]++;
        oracle[i] = b0;
        int b1 = j1-k;
        bucket[b1]++;
        oracle[i+1] = b1;
    }
}
```
Experiments

☐ sometimes a good surrogate for analysis

☐ too much rather than too little output data

☐ reproducibility (10 years!)

☐ software engineering
Example, Parallel External Sorting

sort 100GiB per node

- worst case input
- worst case input, randomized
- random input
- random input, randomized

[10]
Algorithm Libraries — Challenges

- software engineering
- standardization
- performance ↔ generality ↔ simplicity
- applications are a priori unknown
- result checking, verification

STXXL

 STL–user layer
Containers: vector, stack, set, priority_queue, map
Algorithms: sort, for_each, merge

Streaming layer
Pipelined sorting, zero-I/O scanning

Block management layer
typed block, block manager, buffered streams, block prefetcher, buffered block writer

Asynchronous I/O primitives layer
files, I/O requests, disk queues, completion handlers

Operating System
Example: External Sorting

STXXL

Applications

STL-user layer
- Containers: vector, stack, set, priority_queue, map
- Algorithms: sort, for_each, merge

Streaming layer
- Pipelined sorting, zero-I/O scanning

Block management layer
- typed block, block manager, buffered streams, block prefetcher, buffered block writer

Asynchronous I/O primitives layer
- files, I/O requests, disk queues, completion handlers

Operating System

Linux
Windows
Mac, ...
Example: Shared Memory Sorting

Applications

STL Interface

Serial STL Algorithms

Parallel STL Algorithms

OpenMP

Atomic Ops

MCSTL

STL-alike \ll STL-integrated

[11, 16]
Problem Instances

Benchmark instances for **NP-hard** problems

- TSP
- Steiner-Tree
- SAT
- set covering
- graph partitioning
- ...

have proved essential for development of practical algorithms

**Strange:** much less real world instances for **polynomial problems**
(MST, shortest path, max flow, matching...)
Example: Sorting Benchmark (Indy)

100 byte records, 10 byte random keys, with file I/O

<table>
<thead>
<tr>
<th>Category</th>
<th>data volume</th>
<th>performance</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraySort</td>
<td>100 TB</td>
<td>564 GB / min</td>
<td>17×</td>
</tr>
<tr>
<td>MinuteSort</td>
<td>955 GB</td>
<td>955 GB / min</td>
<td>&gt; 10×</td>
</tr>
<tr>
<td>JouleSort</td>
<td>1 000 GB</td>
<td>13 400 Recs/Joule</td>
<td>4×</td>
</tr>
<tr>
<td>JouleSort</td>
<td>100 GB</td>
<td>35 500 Recs/Joule</td>
<td>3×</td>
</tr>
<tr>
<td>JouleSort</td>
<td>10 GB</td>
<td>34 300 Recs/Joule</td>
<td>3×</td>
</tr>
</tbody>
</table>

Also: PennySort
GraySort: inplace multiway mergesort, exact splitting

Xeon Xeon

16 GB RAM

240 GB

Infiniband switch
400 MB / s node all–all
JouleSort

- Intel Atom N330
- 4 GB RAM
- $4 \times 256$ GB SSD (SuperTalent)

Algorithm similar to GraySort
Applications that “Change the World”

Algorithmics has the potential to SHAPE applications
(not just the other way round) [G. Myers]

Bioinformatics: sequencing, proteomics, phylogenetic trees, . . .

Information Retrieval: Searching, ranking,

Traffic Planning: navigation, flow optimization,
    adaptive toll, disruption management

Geographic Information Systems: agriculture, environmental protection,
    disaster management, tourism, . . .

Communication Networks: mobile, P2P, grid, selfish users, . . .
AE for Big Data

Applications
- sensor data
- genomes
- data bases
- www
- GIS
- mobile
- ...

Techniques
- data structures
- graphs
- geometry
- strings
- coding theory
- ...

Technology
- parallelism
- memory hierarchies
- communication
- fault tolerance
- energy
- experience
- PS
Larger Sorting Problems

- millions of processors
  $\Rightarrow$ multipass algorithms
- fault tolerance
- still energy $\sim$ time?

Highly related to MapReduce, index construction, ...
More Big Data Examples From my Group

- Suffix Sorting and its applications
- Main Memory Data Bases
- Graph Partitioning
- Track Reconstruction at CERN
- Route Planning
- Genome Sequencing
- Image Processing
- Priority Queues
Suffix Sorting

sort suffixes $s_i \cdots s_n$ of string $S = s_1 \cdots s_n, s_i \in \{1..n\}$.

Applications: full text search, Burrows-Wheeler text compression, bioinformatics,
E.g. phrase search in time logarithmic or even independent of input size.

~ particularly interesting for large data
Linear Work Suffix Sorting

**simple:** Radix-Sort + linear recursion + merging.

⇒ trivial external [19], parallel [20] adaptation

```
012345678
I  anananas.
```

```
anananas.0
3 2 5
anananas.00
2 4 1
```

```
.00ananananas.s.0
```

```
1 2 2 3 4 5
lexicographic triple names
```

```
I' 3251241
```
Current Work

- distributed memory (external) query
- parallel distributed construction of query data structure (longest common prefixes,...)
Data Bases – Our Approach

[with SAP HANA team, PhD students Dees, Müller]

- main memory based
- column based
- many-core machines
- NUMA-aware
- no precomputed aggregates
- aggressive indexing
- generate C++ code close to tuned manual implementation
TPC-H Decision Support Benchmark

- 22 realistic queries of varying complexity
- Pseudorealistic random data
- F GByte space

**TPC–H Scheme**

- LINEITEM: 1.5M*F
- PARTSUPP: 800K*F
- SUPPLIER: 10K*F
- PART: 200K*F
- ORDERS: 1.5M*F
- CUSTOMER: 150K*F
- NATION: 25
- REGION: 5

F = scale factor
0 = attribute
Typical TPC-H Queries

Q1: Revenue etc. of all shipped LINEITEMs (aggregated into 6 categories)
→ plain flat scan of all LINEITEMs

Q9: Sum profit for all LINEITEMs with a given color for each nation and order year.
→ scan PARTs,
use inverted index to access matching LINEITEMs
Go down from there using forward indices (≈pointers).
First Results

- \( \approx 30 \times \) faster than current record in 300GB category (manual implementation)

- Compiler: seems to be largely orthogonal to algorithmic and parallelization issues

TPC–H Scheme

- \( 6M \times F \)
- \( 800K \times F \)
- \( 10K \times F \)
- \( 200K \times F \)
- \( F = \text{scale factor} \)
- \( 0 = \text{attribute} \)

- PART
- PARTSUPP
- LINEITEM
- SUPPLIER
- ORDERS
- CUSTOMER
- NATION
- REGION

[21]
Larger Inputs

- Already needed by some large customers of SAP
- Move to clusters
  Master thesis Martin Weidner seems to give positive results (5 TPC-H queries) [22]
- fault tolerance beyond recovery?
- energy efficiency using many small nodes (ARM)?

Algorithmic Meat: Randomization, collective communication, communication complexity, sorting, data structures, multi-level memory hierarchies, coding theory
Graph Partitionierung

**Input:** Graph \((V, E)\) (possibly with node and edge weights), \(\epsilon, k\)

**Output:** \(V_1 \cup \cdots \cup V_k\) mit \(|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil\)

**Objective Function:** minimize cut

**Applications:** finite element simulations, VLSI-design, route planning, . . .

**Variants:** hypergraphs, clustering, different objective functions, . . .
Multilevel Graph Partitioning

input graph

... local improvement

contract

output partition

... initial

partitioning

uncontract
Reengineering Multilevel Graph Partitioning

- Distributed evolving algorithm [Alenex12]
- Input graph
- Output partition
- Cycles a la multigrid
- V− F− W−

- Initial partitioning
- Local improvement
- Parallel [IPDPS10] edge ratings match + contract [SEA12]
- n-level [ESA10] flows etc. [ESA11] augment. paths [SEA13]
- TODO
Our Contribution

- scalable parallelization KaPPa
  (matching, edge coloring, evolutionary)
- thorough reengineering of multilevel approach
  (use flows, SCCs, BFS, matching, edge coloring, negative cycle detection, ...)

⇝ high quality (e.g. 90–99% entries in Walshaw’s benchmark)
Large Data Graph Partitioning

- difficult inputs: social networks, WWW, 3D/4D models, VLSI, knowledge graph?
- more difficult parallelization
Future Work

- parallel external
- other variants
- fault tolerant
- component of a graph processing framework
Track reconstruction

Input: clouds of $\approx 10^4$ 3D points
Output: $< 10^3$ spiral tracks of high energy particles
Also cluster tracks by emergence point

Large Data???

- up to $10^5$ instances / s
- cost of processors / energy
- memory constrained
- exploit SIMD/GPU parallelism?

Algorithmic Meat:
Geometric data structures, parallelization, clustering

[25]
Route Planning

Large Data 2004: Western European network (18M nodes).

Dijkstra’s algorithm needs 6s.

- too much time for servers
- too much memory for mobile devices

⇒ inaccurate heuristics with tedious “manual preprocessing”

Our contribution: Automatic preprocessing techniques

- $10^4$–$10^6$ times faster exact query on servers
- still “instantaneous” on mobile devices (external implementation)
Large Data 2013

- 1.6G nodes OpenStreetMap routing graph (edge based)
- billions of GPS traces
  (+ road based sensors + elevation data)
- public transportation

Potential use:

- time-dependent edge weights  \[\text{[27]}\]
- detailed traffic jam detection  Google, TomTom,…
- multi-modal route planning  \[\text{[28]}\]
- probabilistic route planning  attempts
- really useful detours around traffic jams  ???
  use real time traffic simulation??
Genome Sequencing

[29]: 20 000 CPU hours for shotgun sequencing of the human genome
(3 \cdot 10^9 \text{ base pairs, 5–10 times oversampling.}
Prototypical large data problem?

Today: a few minutes on a work station [ZieglerDFMS work in progr.]
(use template, modern hardware, AE + cheap sequencing)
\rightarrow \text{routine use for personal medicine}

New Challenge:

processing many sequences
Phylogenetic Tree Reconstruction
Image Processing

Gigapixel aerial images.
Filters, Segmentation, Change detection

Algorithmic meat: Graph algorithms, parallelization, memory hierarchies, range minimum data structures, …
External Priority Queues

Problem: Binary heaps need
\( \Theta \left( \log \frac{n}{M} \right) \) I/Os per deleteMin

We would rather have:
\( \Theta \left( \frac{1}{B} \log_{M/B} \frac{n}{M} \right) \) I/Os (amortized)
Medium Size PQs – $km \ll M^2 / B$ Insertions

Insert: Initially into insertion buffer.

Overflow $\rightarrow$

sort; flush; smallest key is now in merge PQ

Delete-Min: deleteMin from the PQ with smaller min
Large Queues

$$\approx \frac{2n}{B} \left( 1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$$

I/Os for $n$ insertions

$\mathcal{O}(n \log n)$ Arbeit.

[31].

deleteMin:

“amortisiert umsonst”.

Sequence HeapPriority Queues

- external
- swapped
- cached

$\text{group−merge}$

$\text{group−buffer}$

$\text{group−buffer}$

$\text{group−buffer}$

$\text{R−merge}$

$\text{deletion buffer}$

$\text{insertion buffer}$
Experiments

Keys: random 32 bit integers

Associated information: 32 dummy bits

Deletion buffer size: 32  
Near optimal

Group buffer size: 256

Merging degree $k$: 128  
: performance on all machines tried!

Compiler flags: Highly optimizing, nothing advanced

Operation Sequence:

$(\text{Insert-DeleteMin-Insert})^N (\text{DeleteMin-Insert-DeleteMin})^N$

Near optimal performance on all machines tried!
Alpha-21164, 533 MHz

The diagram shows the performance of different heap structures on a computer with an Alpha-21164, 533 MHz processor. The x-axis represents the number of elements N, and the y-axis represents the time (T(deleteMin) + T(insert))/\log N in nanoseconds (ns).

- **Bottom up binary heap**
- **Bottom up aligned 4-ary heap**
- **Sequence heap**

The graph illustrates the growth of the time complexity with the increase in the number of elements N.
Core2 Duo Notebook, 1.??? GHz

\[ \frac{T(\text{deleteMin}) + T(\text{insert})}{\log N} \text{ [ns]} \]

- bottom up binary heap
- sequence heap

- Graph showing the performance of bottom up binary heap and sequence heap with varying N values.
Future Work

- see above
- find more algorithmic application problems
- algorithmic cores of application independent libraries and tools
data structures, MapReduce, graphs, data bases, ...
- distributed memory external algorithms
- back to massive parallelism including exascale
- fault tolerance
Commercial Break

I am hiring

PhD students, Postdocs in algorithm engineering.
Desirable Skills:

- Desire to bridge gaps between theory and practice
- Algorithmics
- Performance oriented C++ programming
- Parallelization, e.g., MPI, OpenMP,…
Literatur


