

External Sorting

Jingke Li

Portland State University

External Sorting

- Initial data and final results are both stored on disks. → Expensive requires
Read and write? we different approach.
- ▶ For large data set, two passes over data are the minimum.
 - ▶ Algorithms typically consist of two phases: have very large memory (case)
 - The first phase produces a set of files containing half-processed data.
 - The second phase processes these files to produce a totally ordered permutation of the input data.
 - ▶ Disk I/O bandwidth is often the performance bottleneck.
 - Most algorithms use only two passes.
 - Most algorithms try to balance the time between the two phases. (2x performance)
 - Most algorithms try to maximize overlap between I/O and internal sorting operations.

Best possible scenario - time you spend reading writing data twice (sorting time is insignificant comparing to IO time.).

One pass and two pass algorithms considered to be the best
Assignment 3 uses two pass algorithm.

How to partition work between two phases.

Performance tuning → improve or get rid of IO cost (overhead)

Two Algorithm Categories

hard

- ▶ *Distribution-Based Sorting:* (Bucket sort based idea)

Easy

- ▶ The first phase partitions the input data into k disjoint, ordered buckets — each bucket contains numbers smaller than those in the next bucket.

VS

Easy

- ▶ In the second phase, each bucket is sorted independently.

- ▶ *Merge-Based Sorting:*

- ▶ The first phase partitions the input data into (unordered) chunks of approximately equal size; sorts these chunks in main memory and writes the “runs” to disk.

hard

- ▶ The second phase merges the runs in main memory and writes the sorted output to the disk.

For many realistic cases use of buckets is ok.

ATM transaction protocol (famous computer science)

Jim Gray's External Sort Benchmarks

An annual competition to promote *practical* parallel sorting algorithms.

Started in 1985 by Jim Gray. (sortbenchmark.org) *Conducted every year.*

- ▶ **Datamation** (1985-2001) — Sort one million 100-byte records. (The original benchmark. Deprecated since the task became too easy.)

Very popular

- ▶ **MinuteSort** (since 1994) — Sort as many records as possible in one minute.

- ▶ **PennySort** (1994-2011) — Sort as much as possible for a penny's worth of system time. → No need to have a super computer

- ▶ **TeraByteSort** (1998-2008) — Sort ten billion 100-byte records. (Deprecated since it became too similar to MinuteSort.)

- ▶ **JouleSort** (since 2007) — Minimize the amount of energy required to sort $10^8 - 10^{12}$ records (10GB – 100TB). More important nowadays.

- ▶ **GraySort** (since 2009) — Maximize the sort rate (TBs/minute) achieved while sorting a very large amount of data (currently 100TB minimum).

Many sorting algorithms came out from this competition.

1985 records are done now in 1 second or less.
Any general hardware or software can be used (almost no constraints)

Many ideas were developed to solve datamation.

In real world some records are sorted by the key.

10 byte key doesn't fit into register.
No specially designed hardware is allowed.

SMP - small scale supercomputer

Datamation Benchmark

"Sort one million records (total 95MB data) from disk to disk."

- *Data format* — 10-byte key inside 100-byte record
- *System configuration* — unrestricted except no special design allowed.
- *Fact:* Can be achieved by a *single-pass* sort.

year	name	system	#nodes	#disks	time(s)	1h 20min
1986	-	Tandem (SMP)	2	2	3,600.0	
1987	-	Tandem (SMP)	3	6	980.0	
1988	-	Cray 1 (Vector)	1+	1	28.0	
1993	AlphaSort	DEC Alpha SMP	1	16	9.1	
1994	AlphaSort	DEC Alpha SMP	3	28	7.0	
1996	Nsort +	SGI SMP	12	96	4.2	
1997	NowSort	SUN Sparc Cluster	32	64	2.4	
1999	Millennium Sort	DELL NT Cluster	16x2	?	1.2	
2000	Diaprism Sorter	HP Xeon Cluster	4	32	1.0	
2001	NowSort wow	Intel P3 Cluster	32x2	32x5	0.44	

→ Took several years to beat.

→ Entirely message passing.

In contrast, early parallel systems couldn't provide competitive results.

year	system	#nodes	#disks	time(s)
1990	Sequent (SMP)	8	4	83.0
1992	Intel iPSC/2 (DMS)	32	32	58.0

Jingke Li (Portland State University)

CS 415/515 External Sorting

5 / 20

Parallel computers were no match for sequential machines.

MinuteSort Benchmark

"Sort as much as possible in one minute."

year	name	system	#nodes	#disks	GB
1995	AlphaSort	DEC Alpha SMP	3	36	1.1
1997	Nsort	SGI SMP	32	?	3.5
1998	Nsort	SGI SMP	32	?	5.8
2000	Nsort	SGI SMP	32	?	12
2004	Nsort	Itanium2 Cluster	32	2,350	32
2006	NeoSort	Itanium2 Cluster	32	128	40
2007	TokuSampleSort	Disk Cluster	400x2	400x6	214
2009	Hadoop	Cluster	195x8	195x4	500
2012	Flat Datacenter Storage	Disk Cluster	256	1,033	1,401
2014	DeepSort	Xeon Cluster	384x2	384x8	3,700

wow!

SMP - shared memory systems.

Jingke Li (Portland State University)

CS 415/515 External Sorting

6 / 20

Case Study: AlphaSort ('93-'95) (winners '93-'95)

- ▶ **Hardware:** A SMP with commodity components
 - 1-3 DEC Alpha AXP 3000/7000 processors
 - 256MB memory (memory bus 640MB/s) *large enough to fit entire data set -*
 - 10-28 disks (I/O bus 4 x 100MB/s)
- ▶ **Software:** DEC OpenVMS, threads over shared memory
 - Threads allow overlapping between I/O and computing
 - Locks are used to ensure proper access to shared data
- ▶ **Algorithm:** Quicksort (*Based on Quicksort*)
 - Focusing on overlapping I/O and computing
 - Small degree of parallelism — only 3 concurrent processes

VMS — standard under digital machines
(1988)

Jingke Li (Portland State University)

CS 415/515 External Sorting

7 / 20

First version Single-Node AlphaSort

1. Divide one million records into 30-50 groups; read them one by one from disk.
2. As each group becomes available, use a separate thread to quicksort it into a run. *(runs kept inside memory)*
3. Merge the runs using a replacement-selection tree. *(standard way)*
4. Gather records into contiguous buffer and write to disk. *(standard way)*

Key Optimizations:

- ▶ *Multiple threads* — overlapping I/O and internal sorting.
- ▶ *Disk striping* — to allow parallel reads and writes. (*Hardware: 3 controllers and 28 disks; Software: a file striping layer*)
- ▶ *Lots of memory* — 256-384MB for sorting 95MB data; this affords quicksort for internal sorting

Small debacles allowed them to win -

Jingke Li (Portland State University)

CS 415/515 External Sorting

8 / 20

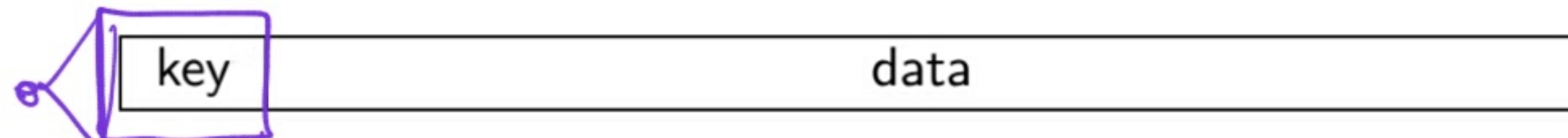
Needed lots of memory to allow single pass -

Need 3x more memory to avoid reusing memory

Reallocate memory? Or do something about keys?

Internal Quicksort Options

Each record is 100 bytes, with a 10-byte key:



Option 1. Sort records directly. *Move! (moving is overhead)*

- ▶ An array of 100-byte records is quicksorted in place.
- ▶ Simple and straightforward; no additional memory is needed.

Option 2. Sort pointers. *4 bytes vs 100 bytes,*

- ▶ An array of 4-byte record pointers is generated and quicksorted.
- ▶ The records must be referenced during the sort to resolve each key comparison, but only the pointers are moved during the sort.

Cost: Extra memory reference

Internal Quicksort Options (cont.)

value + pointer to data

Option 3. Sort (key, pointer) pairs. *Pull out keys*

- ▶ An array of 16-byte (key, pointer) pairs is generated and quicksorted.
- ▶ The pointers are not dereferenced during the quicksort. This is known as a detached key sort. *//Should be better than option 2.*

Option 4. Sort (key-prefix, pointer) pairs. *as a bucket.*

- ▶ An array of 8-byte (key-prefix, pointer) pairs is generated and quicksorted.
- ▶ If two key-prefixes are equal, their pointers are dereferenced and the full keys are compared.
- ▶ Motivation for this approach is to fit more (key, pointer) pairs in cache and to align by cache-line.

Performance Comparison

Choice	Array-Gen	Quicksort	Output
<i>records</i>	—	20.47s	2.49s
<i>pointers</i>	0.08s	12.74s	3.52s
(key, pointer)'s	1.07s	4.02s	3.41s
(key-prefix, pointer)'s	0.84s	3.32s	3.41s

Parallel AlphaSort

1. Each thread requests affinity to a processor.
2. Master thread responsible for all file operations; worker threads do sorting and memory-intensive operations.
3. Master reads records; workers quicksort the data groups.
4. Master merges the runs; workers gather records into contiguous buffers.
5. Master writes sorted buffers to disk.

Performance Tuning: Balancing workload

- Time to read vs. Time to sort *sorting faster than reading doesn't improve*
- Time to merge vs. Time to gather records *make them equal as*

To ↵ the bottleneck Many doses and thus

NowSort - Network of workstations Sort.

Each one of the stations can internalise the entire set.

Very simple cluster (no advanced functionality)

Case Study: NowSort ('97-'01)

► **Hardware:** A cluster of commodity workstations

- Up to 64 SUN ultraSPARC I workstations, each has 64-128MB memory and 2-4 SCSI disks
- Connected with an Marinet switch (160MB/s) / faster than ethernet

► **Software:** GLUnix + N copies of Solaris

- Supports a simple parallel environment — process start-up, job control, etc., but no dynamic scheduling
- Shared-nothing with explicit communication — One process per node
- Active messages → 10 μ s latency, 35 MB/s bandwidth

► **Algorithm:** Quicksort + Bucket Sort

- Input data is assumed to be drawn from an uniform distribution, and is evenly partitioned across processors' local disks.

Bucket sort plays a key role in improving performance

Single-Node NowSort

Very simple

1. Read all records into memory (no group partitioning).

2. Internally sort the records (several options available). (using whatever sort)

3. Gather records into contiguous buffer and write to disks.

! Note: No threading is used — overlapping is achieved by other means, i.e. reading records directly into buckets.

They used bucket sort so multithreading was not applicable

Parallel NowSort

1. *Read* — Each processor reads records from its local disk into memory.
2. *Communicate* — Key values are examined and the records are sent to their destination buckets. (One of the buckets is local.)
3. *Sort* — Each processor sorts its local records. *data is partitioned at this point.*
4. *Write* — Each processor gathers and writes its records to local disk.

Main Issue: Hiding the overhead of communication.

Exactly as we do in HW assignment № 3?

*How to implement Bucket sort in C?
Internal Sort Options*

Option 1. Quicksort

- Quicksort the (key-prefix, pointer) pairs. *as in Radix Sort.*

Option 2. Bucket Sort + Quicksort

- While reading records into memory, simultaneously examine the high-order b bits of the keys and place keys into the appropriate buckets.
- Each bucket is sorted individually with quicksort.
- The number of buckets (hence the value of b) is determined such that the average number of keys per bucket fits into the L2 cache.
- Each bucket entry contains the most significant 32-bits of a key after the top b -bits, and a pointer to the full record.

Put record into buckets while reading

Can we do it?

$b = 11$ in their testing :)

Is there a way to use that idea?

Internal Sort Options (cont.)

Option 3. Bucket Sort + Partial Radix-Sort *(Sort kinda like bucket sort)*

- ▶ Distribute keys into buckets as in the previous algorithm.
- ▶ But use a partial radix-sort instead of quicksort to sort each bucket.
- ▶ Two passes are performed over the keys, each with a radix size of 11 bits. *2·n - complexity of the radix sort!*
- ▶ Since the two passes together examine only 22-bits of (80-b)-bit keys, a final clean-up phase is performed, where keys with ties in the top 22-bits are bubble-sorted.

Experimental data show that this last option has the best performance.

How to create buckets in C?

NowSort Optimizations

- ▶ User-level software striping.
- ▶ Use `mmap` instead of `read`, to avoid the system double-buffering problem.
- ▶ Use (key-prefix, pointer) pairs in internal sorting.
- ▶ Overlap I/O with CPU computation:
 - With the use of buckets, the need for this overlap is low.
- ▶ Overlapping I/O with communication:
 - *Interleaved* — A single thread alternates reading and communicating. Input data are saved in (small) send buffers, one for each destination. As soon as a buffer is full, its data is sent out. (Active message is a key for this to work.)
 - *Threaded* — One I/O thread and one communication thread.

Compare int

↓ Don't wait until bucket is sorted, wait until bucket size is big enough to send over.

Two-Pass NowSort

- ▶ *Create Runs* — The one-pass sort (*read*, *sort*, and *write*) is repeated to create multiple sorted runs on disk.

Two threads are used:

- A *reader* for reading records from disk, moving keys and pointers into buckets, and signaling when the buffer is full;
- A *writer* for sorting each bucket, writing records to disk, and signaling when the buffer is empty.

- ▶ *Merge Runs* — The sorted runs are merged into a single sorted file.

Three threads are used:

- A *reader* for prefetching chunks from sorted runs;
- A *merger* for performing merging and copying records into a write buffer;
- And a *writer* for writing the records to disk.

Other External Sorting Highlights



- ▶ *GpuTeraSort* ('06 Penny Benchmark) — Sorting with GPU co-processor with a bitomic sorting algorithm.
- ▶ *Hadoop Sort* ('09 Minute Benchmark) — Based on the “MapReduce” model. Essentially a bucket sort.