Lecture 6: Parallel OLAP

Owen Kaser

References: Liu’s thesis, Goil’s dissertation (see “Links” page)
Overview

1. Why parallel?

2. Types of parallel machines considered

3. Data Distribution

4. A naïve of Datacube parallelization

5. Other Parallel Projects
short lecture?
“Size” of computer: CPU muscle (MIPS), RAM (GB), online disks (TB).

Amdahl’s Observation, mid 1960s: A general-purpose machine should have a size balance between CPU, RAM, disk.

Rule “1MB RAM per 1 MIPS” is quoted.

Article by Hill et al (Computer, 1995): machine with tons of RAM is probably a multiprocessor, if balanced.

OLAP apps would like LOTS of RAM and disk.

So, a general-purpose machine for OLAP is probably multiprocessor.

Might as well use these extra processors to gain speedup.
Recall MIPS = “millions of instructions per second”. Very suspect measurement of processor speed.

Only way to get a high MIPS rating is to have multiple processors, after reaching uniprocessor limit.

Of course, a dedicated specialized server won’t be a balanced system. But if you have spent millions on disks, why skimp on RAM and processors?
Speedup is a slippery concept, related to $\frac{\text{sequential time}}{\text{parallel time}}$.

“sequential time” to solve problem on a uniprocessor.

“parallel time” to solve problem on a parallel machine.

Q: which uniprocessor, which parallel machine?

Many parallel machines can run sequential programs (they just use one CPU). So “uniprocessor” might be misleading.

Sometimes, “sequential time” is measured by running the parallel program with only one CPU enabled!

Better to run a known efficient sequential program on the parallel machine. (Definitely free of “parallelization overhead” costs.)
Speedup is a function of the number of processors.
Factors Reducing Speedup

Doubling the number of humans working on a project usually won’t halve the completion time.

Some reasons, for humans or CPUs:

- project has some “inherently sequential” parts
- tasks too big (and few): some workers idle
- strongly coupled tasks require coördination, communication
- to reduce coördination, add redundant work. Total effort ↑.
Data cube calculation has lots of divisible work. Communication overheads are a peril, though. Vast volumes of data to interchange.
Types of Parallel Systems Considered

Several CPUs:

- **SMP** (symmetric multiprocessing), shared memory, shared disks, share one copy of OS.

- networked computers. private memory, local disks, one copy of OS per node.

- hybrid systems. Network of SMP nodes.
These are all systems with multiple CPUs that can each do its “own thing”. In “Flynn’s taxonomy”, they are MIMD.

Dr Bhavsar has a couple of grad courses in parallel computation that you can take.

There is a nice theoretical basis for parallel algorithmics, quite distant from systems reality.
Commonly implemented via shared bus.

Bus saturation risk. Shared bus limited to handful of processors.

Other implementations possible, but often expensive or high-latency.
Low latency, fairly high bandwidth to sharing data

Hard to program (threads stomp on each others data).

OMP standard extensions to C, Fortran.
Distributed-Memory Systems

Each CPU has its own memory addresses, no direct access to others’ addresses.

Usually implemented by a message-passing network, typically lower-latency/higher bandwidth than “ordinary networking”.

Variety of interconnect topologies have been used.

“Cluster computing”: strongly networked nodes running an almost standard OS.
Not every parallel system is either an SMP or one of the above!

MPI (Message Passing Interface): a middleware specification to make dissimilar machines provide the same programming interface (SEND, RECEIVE, etc).

Goil built his system on an SP-2 with many nodes.
Symphony.Unb.CA is a hybrid

It’s a cluster of 4 nodes, each a 4-processor SMP.

Each node is its own computer (own IP address, own disks, ethernet card). But it can communicate well with its peers via “SP-switch”.

![Diagram of Symphony.Unb.CA system architecture]

- Node #0: 4-way SMP+disk
- CPUs: #1, #2, #3, #4
- Local disk, I/O, Mem

- Node #1: 4-way SMP with disk

- Node #2: 4-way SMP with disk

- Node #3: 4-way SMP with disk

- SP switch, shared disk tower

Ethernet connections between nodes.
This model is hard to program efficiently. MPI can send and receive messages quickly between processors on a node, but not as quickly as using OMP! But OMP only works within a node.

Richard Liu built his parallel datacube system on Symphony, an IBM SP-3.
Work composed of subtasks, maybe some “precedence constraints” between them.

In parallel programming, “who solves which subtask?” is important.

If the set of tasks and their approximate durations known \textit{a priori}, can do \textit{static load balancing}: A schedule computed in advance.

If tasks are dynamically created or task durations unknown, \textit{dynamic load balancing} means decisions made at runtime. Maybe a “manager” process to do this.
precedences: task “fold laundry” cannot start before task “dry laundry”. Better start folding after drying’s done.

“Pipelining” refers to drying a few items, then folding those items while the next few items are being dried, etc.

There are many papers on load balancing. Dynamic ones based on “work stealing” have been popular recently. (See recent ACM-SPAA conferences.)
Data Distribution for OLAP

For non-SMP systems, it is important to decide which data goes on which processor.

To support OLAP, suppose we compute *entire* datacube (all $2^d$ views).

Insufficient space to replicate the datacube on every processor.

Option 1: partition entire cubes/views between nodes. (Chosen host gets ALL of certain cubes.)

Option 2: every processor gets certain slices of every cube/view. ("Range partition source cube on attribute $i$")
But there *may* be enough storage to keep a full copy of the source cube for each processor.
Why Option 2?

If all nodes store part of each view, to parallelize an OLAP query using view V, each node can do part.

With Option 1, the owning node would have to send parts to the other nodes, before they can help.
A Simple Parallelization of DataCube Operator

Step 1: Distributing source cube $S$.

Say we’re using 4 processors. Choose the longest dimension (eg dimension $k$, with $n_k$ values).

Imagine $S$ broken into $n_k$ slices along dim $k$.

The first $n_k/4$ slices sent to Proc 0, next $n_k/4$ to Proc 1, ...
Problems looming: given that data density is not uniform, we may end up giving one processor mostly good data, while another processor gets mostly empty cells.

Datacube in parallel is interesting as a *nontraditional* application. Normally parallelism is used for compute-bound apps. Here, there is little computation but vast I/O to be subdivided.

Upshot: compute-server admin policies may be hurtful. (Eg, originally the SP-3 node local disks were only for swap. Fortunately, ACRL was responsive.)
Step 2: Local computation:

Every processor pretends its slices forms a complete source cube, and computes the datacube using some efficient sequential algorithm.

Consider any view that keeps dimension $k$. This view is correctly computed AND already distributed correctly.
but half the views get rid of dimension $k$...
Step 3: Finish Aggregation along Dimension $k$.

Problem with every view $V$ in which dimension $k$ has been “extreme rollup’ed”, ie aggregated.

Every processor has a full-size but incorrect version of $V$, that aggregates only the slice of $k$ given to it.

Procs jointly choose some other dimension for partitioning $V$. Everyone sends first 1/4 slices of their version of $V$ to Proc0. Proc0 can then finish the aggregation.

Likewise, for procs 1, 2, and 3.
Data partitioned along dimension A. Wish to calculate AB from ABC.

Illustration of Step 3

Processor 0’s partial BC

Processor 1’s partial BC

Need to Combine
Data distribution shown by colour.
Need to get cube BC and partition by B.

Processor 0’s partial BC

Processor 1’s partial BC

Final BC
So What’s Wrong?

This algorithm requires a lot of data movement. (That’s bad!)

Half the views have to be redistributed for step 3. With $p$ processors, every processor sends $\frac{p-1}{p}$ of such views. So if a view has size $|V|$, it costs $\frac{|V|p(p-1)}{p} = |V|(p - 1)$ communication.
Also Poor Load Balancing

This algorithm may not divide work evenly.

Suppose processor 3 gets mostly empty slices, but processor 0 gets mostly non-empty slice.

Then processor 3 will finish step 2 quickly, and then have to wait for processor 0.

This is bad.
To reduce communication, interleave step 2 (cube calculation) with step 3 (fixup). Once a “parent” view fixed, many of the children computed from it will not need fixup.

To improve load balancing, normalize the data randomly, so that the slices assigned to each processor should have a similar number of empty cells.

Or use some known view-size estimation techniques to help you adjust number of slices per processor.
Random normalization is a bad idea in general, as it makes the cube more uniformly sparse. But it worked well for Richard.
Various parallel datacube systems

- Goil’s. MOLAP, partitioning done according to two dimensions. Do-it-yourself buffer mgmt. Aggregation algorithm supports single-path pipelining, I think. Platform is SP-2, and good speedups are claimed despite static load balancing.

- Richard Liu’s prototype implementation. Cubes are memory mapped (so OS does buffer mgmt), partitioning is 1d, otherwise like simplified Goil. SP-3 platform, speedups are poor.

- Dehne, Eavis, Rau-Chaplin (“Course Grained. . . ”): ROLAP, using PipeSort Each processor computes its own set of group-bys, independently. Appears source cube replicated on each machine, and pipes are confined to individual processors. Platform: low-end PC cluster. Good speedups reported.
Richard’s implementation does not carefully estimate the work assigned to each processor. Goil’s does. Goil’s implementation useds high performance “asynchronous” communication, whereas Richard uses easier/safer synchronous mechanisms. Uneven work distribution and synchronous communication are a bad combination.

Dehne et al have several parallel datacube papers, using different approaches.
Ng, Wagner, Yin (2001) “Iceberg-cube Computation with PC Clusters”. Several simple ROLAP algorithms. Ones using dynamic load balancing work best.

Yang, Jin, Agrawal (FGCS, 2003) “Implementing Data Cube Construction Using a Cluster Middleware...”. MOLAP, built around a multidimensional parallel middleware called ADR.
In Yang’s setup, partitioning data in many dimensions is worthwhile.