

Prof. Hasso Plattner



A Course in In-Memory Data Management

The Inner Mechanics
of In-Memory Databases

September 4, 2015

This learning material is part of the reading material for Prof. Plattner's online lecture "In-Memory Data Management" taking place at www.openHPI.de. If you have any questions or remarks regarding the online lecture or the reading material, please give us a note at openhpi-imdb@hpi.uni-potsdam.de. We are glad to further improve the material.

Research Group "Enterprise Platform and Integration Concepts",
<http://epic.hpi.uni-potsdam.de>

Parallel Data Processing

In the following, we discuss how to achieve parallelism in in-memory and traditional database management systems. Pipelined parallelism and data parallelism are two approaches to speed up query processing.

In pipelined parallelism, the next operator already starts while the current operator is not finished but has already produced partial results. Thus, the execution time of operators partly overlaps. For example, consider a JOIN and SORT operator involving the evaluation of a predicate. Each operator performs its tasks and if first results become available, they are used by the next operator as depicted in Figure 17.1 on the left. This cascade of operations is denoted as a pipelining and can be extended without any limits.

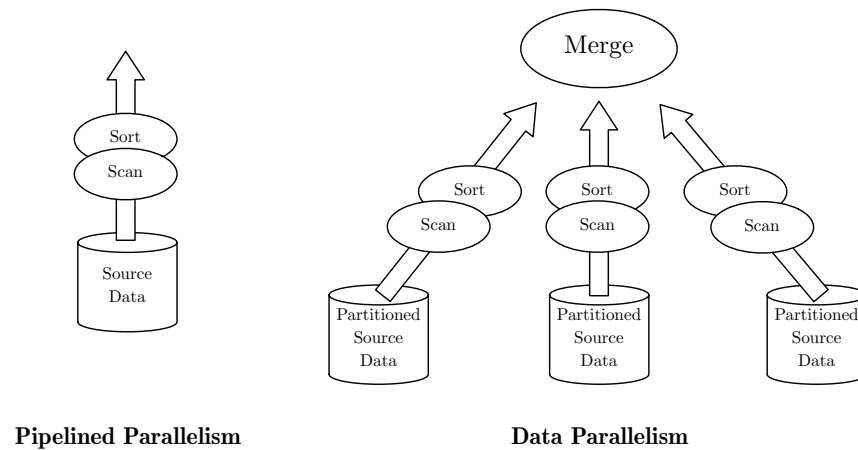


Fig. 17.1: Pipelined and Data Parallelism

In data parallelism, the data set is partitioned so that the operators of a query work on individual parts of the data set in parallel. Afterwards, the results of the parallel operations are merged to the complete result set (see Figure 17.1 on the right). The query plan becomes more complex since all operators are executed on each data partition individually and a merge operation is added.

In database management systems, further aspects can be considered for parallelization. We distinguish between intra-query and inter-query parallelism. Intra-query parallelism addresses parallelization of operators within a query, i.e., the query looks like a single operation, but it is parallelized internally, e.g., by spawning multiple threads and using data parallelism. Inter-query parallelism addresses the aspect to schedule multiple queries to execute them in parallel. This results in efficient utilization of CPU caches and a reduction of data transfers.

17.1 Hardware Layer

Parallel processing of data is an essential aspect of achieving high performance for in-memory database systems. But what are the reasons for using parallelization instead of a single CPU core running at a tremendous high frequency, let us assume 1 PHz?

17.1.1 Multi-Core CPUs

Ideally, a modern computer would consist of a single CPU core running at 1 PHz and huge persistent integrated main memory as shown in Figure 17.2. Reality looks different, though. Nowadays, we typically have multiple CPU cores on one CPU die. Furthermore, modern server systems consist of multiple CPUs. This multiplies the number of cores.

The reasons for that multi-core development are buried in the hardware developments of the last decade. The assumption that the number of transistors doubles every 18 month, known as Moore's law, is still valid [Moo65]. However, the operating frequency of the transistors cannot be increased infinitely. For example, with increasing frequency the ratio of heat loss increases. As a result, the energy efficiency degrades while additional power is required to cool transistors. Hardware vendors proved that using multiple CPU cores operated at a lower frequency, e.g., 2.4-2.7 GHz, increases efficiency while keeping cooling requirements at an adequate level. For example, Figure 17.3 depicts the conceptual architecture of single CPU consisting of four cores. Combining multiple CPUs within a single server is shown in

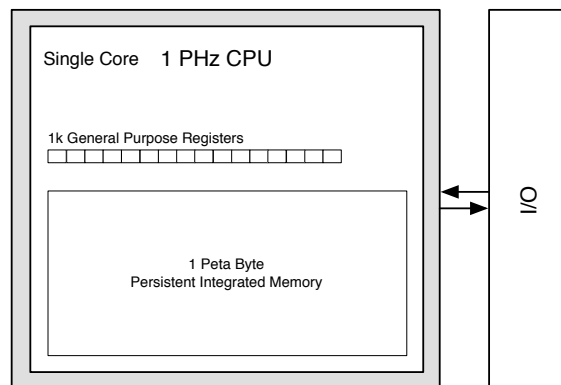


Fig. 17.2: The Ideal Hardware?

Figure 17.4 and the combination of multiple servers to form a more powerful data processing system is depicted in Figure 17.5.

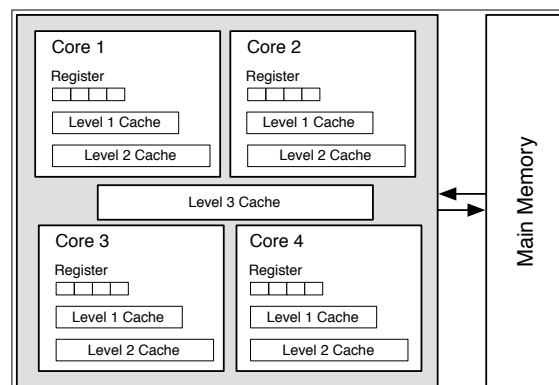


Fig. 17.3: A Multi-Core Processor Consisting of 4 Cores

17.1.2 Single Instruction Multiple Data

The foundation of parallelization can directly be found within the CPU, i.e., data processing can be parallelized using the Single Instruction Multiple Data (SIMD) paradigm. SIMD allows to perform one data operation on mul-

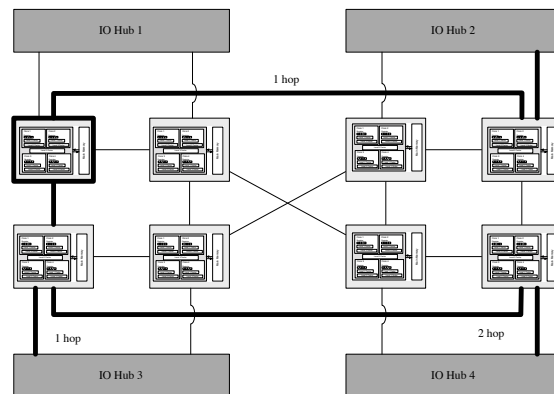


Fig. 17.4: A Server Consisting of Multiple Processors

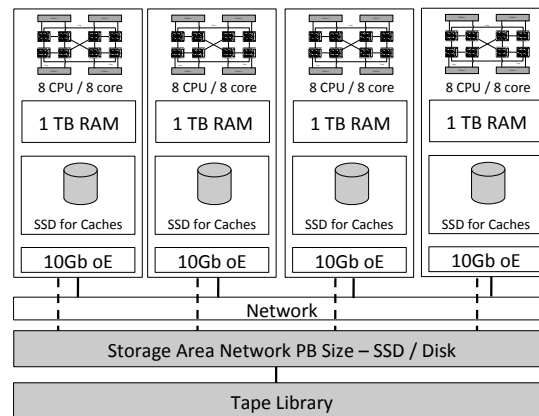


Fig. 17.5: A System Consisting of Multiple Servers

multiple values in one CPU cycle. In contrast to traditional Reduced Instruction Set Computing (RISC) CPUs, SIMD parallelization builds on the use of so-called vectorized operators. These operators are directly implemented in the CPU to perform operations on multiple data words in specialized CPU registers in parallel. Computer graphics makes use of Streaming SIMD Extensions (SSE) instructions that operate on either 128 or 256 bit wide registers. For example, in one 128 bit register you can store four 32 bit values to perform a Parallel Add (PADD) as depicted in Figure 17.6. Thus, four calculations can be processed within one instruction step instead of Scalar Add (SADD) where one calculation is performed at a time.

For instance, let us consider the aggregation of outstanding items. Using PADD reduces the time to sum up the individual items dramatically by summing up multiple items in a single instruction.

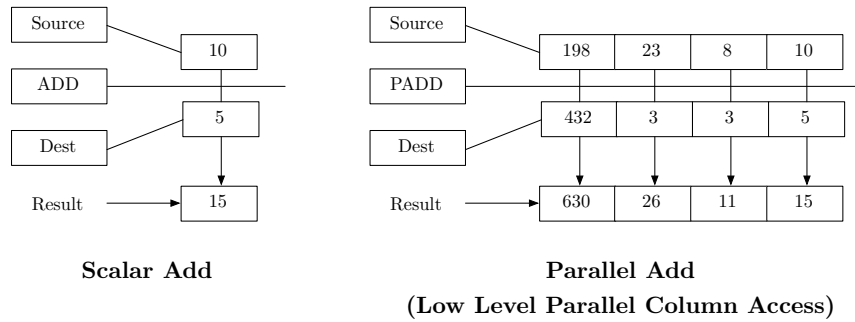


Fig. 17.6: Single Instruction Multiple Data Parallelism

Let us assume the following example: the gender attribute values can be stored in a single bit. Using SIMD you can process the gender of 128 persons in a single CPU instruction step. For example, if you encode male as 1 and female as 0, calculating the ratio of male and female persons of this group of 128 persons is performed within a single instruction by performing a PADD. For comparison, modern processor families are able to perform 100,000 Million Instructions Per Second (MIPS) and more [HP11]. SIMD is the lowest level of parallelization on a computer system.

17.2 Software Layer

In addition to hardware parallelism, we consider software-level parallelism in the following section.

17.2.1 Amdahl's Law

Gene Amdahl conducted fundamental considerations about software-level parallelism. He defined that the maximum speedup of executing a piece of code in parallel is limited by the time needed to process the longest sequential fraction of the code. This principle is known as Amdahl's law [Amd67].

$$\text{max. speedup}(N) = \frac{1}{(1 - P) + \frac{P}{N}} \quad (17.1)$$

Equation (17.1) defines Amdahl's law with P giving the fraction of the code that can be processed in parallel and N giving the level of parallelism, e.g. the number of CPU cores.

Let us assume the following example: the ratio of parallel and the sequential part are 3:1. If the execution time of the parallel part is decreased, e.g. by increasing the number of cores, the maximum speedup cannot exceed four as can be seen in Equation (17.2).

$$\lim_{N \rightarrow \infty} \text{speedup}(N) = \frac{1}{\left(1 - \frac{3}{4}\right) + \frac{3}{4N}} = \frac{1}{\frac{1}{4}} = 4 \quad (17.2)$$

Amdahl's law assumes that there is a fixed size of the solution space, i.e. a task generates repeatable a finite number of results. In contrast, Gustafson assumes that there is a maximum acceptable response time while the solution space is not known beforehand [Gus88]. Equation (17.3) defines Gustafson's law with C defining the number of cores and α defining the non-parallelizable fraction of the program code.

$$\text{max. speedup}(C) = C - \alpha(C - 1) \quad (17.3)$$

17.2.2 Shared Memory

In a shared memory system [Li86], data that is stored in the shared memory segment is accessible by all processors in a uniform way. Special programming concepts, such as mutexes and semaphores, are used to avoid conflicting data access in the shared memory segment, e.g., simultaneous write access. Although shared memory is an easy way to share data across processes or CPU cores, it comes with the problem of scaling.

Shared memory systems suffer from scalability issues since the maximum size of the shared memory segment is limited by available memory size. The total memory size of a single system is small compared to the total main memory size formed by multiple servers.

17.2.3 Message Passing

Message passing is a very powerful paradigm to improve the processing of algorithmic problems [GLS94]. Instead of sharing memory between all threads, only messages are passed between individual processing threads. This paradigm is widely used for number crunching tasks, such as prediction of meteorology, earthquakes, and other kinds of simulations.

All CPU cores perform tasks independently while cores depending on results of each other exchange messages for coordination. In comparison to the shared memory approach, message passing can easily scale out, because processors are independent of shared memory. Thus, they can perform their tasks individually while exchanging messages, e.g., via network links. However, if the sum of exchanged messages exceeds the network capacity, the network becomes a bottleneck for this parallelism paradigm.

17.2.4 MapReduce

This data parallel paradigm aims at identifying a portion of data so that each portion can be processed in parallel. Thus, each processing job is performing the same task on an individual partition of the complete data. Examples of data parallel paradigm are the MapReduce framework and the OpenMP library [DG08, DM98].

MapReduce consists of two specific functions: the map and the reduce function. The former operates on individual data partitions in parallel and produces partial results $r_1..r_n$ for its assigned partition $p_1..p_n$. The reduce function forms one overall result r_{all} by merging all partial result $r_1..r_n$. Map and reduce steps can be chained to produce arbitrary results for complex tasks.

The canonical example for MapReduce is counting the number of occurrences of a specific word in a defined set of text documents. Each map function processes an individual text document or a part of it. It counts the number of occurrences for a specific word within this document. Since map functions are executed in parallel, multiple text documents are scanned for the desired word simultaneously. The following reduce function calculates the total number of occurrences for the specific word by summing up the individual results.

MapReduce requires the developer to define the “how” and the “what”, i.e., if your algorithm does not scale efficiently, the overall response time will not be reduced. This direct control may also be a disadvantage for some tasks since you only want to define the “what”. For example, in a database management system you expect an optimizer to generate the proper code – the “how” – to retrieve the desired data – the “what”.

Thus, MapReduce does not address all problems efficiently. It is designed for parallel processing of a batch job, e.g., word counting. However, interactive analytical queries require flexible access to data. For example, exploring overdue payers requires subsequent analyses of data subsets, which makes it hard to have all possible map functions available.

17.3 References

- [Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.
- [GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994.
- [Gus88] John L. Gustafson. Reevaluating amdahl’s law. *Commun. ACM*, 31(5):532–533, May 1988.
- [HP11] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach; 5th ed.* Elsevier Science, Burlington, 2011.
- [Li86] Kai Li. *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, New Haven, CT, USA, 1986. AAI8728365.
- [Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114 ff., 1965.