Parallel computing focussing on multiprocessing with shared memory; problems when developing parallel programs with respect to different memory consistency models

CHRISTIAN DUBS
HSR Hochschule für Technik, Rapperswil
cdubs@hsr.ch

Abstract

There are many different papers written about the topic of parallel computing and memory consistency models, but most of them are focussing on a specific problem field.

None of these papers delivers an overview of parallel computing, as well as an overview of different memory consistency models in the field of parallel programming.

There is a strict memory consistency model called sequential consistency and there are relaxed ones like total store ordering. Depending on the used memory consistency model different problems or outcomes may arise, when running a parallel program.

Unfortunately, there is no paper so far that describes the discrepancy and the similarities - especially of the different relaxed memory consistency models.

This paper tries to fill this gap by giving an overview in a sense of what sort of different kinds of hardware configurations exist that allow to run a program concurrently. Furthermore, the paper shortly mentions some important problems that may arise when developing parallel programs. Finally, different types of memory consistency models are discussed with respect to their similarities and diversities.

One of the main aspects to be recognized is the fact that a developer automatically depends on certain memory consistency models - transparent and inherent in the system - when programming on a high level language.

Parallel computing “is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently, which is, that they are solved in parallel.”

For many years parallelism has been employed mainly in high-performance computing, but lately things have changed and true parallelism is not restricted anymore to high-performance computing. Parallelism has come to the client-side due to the physical constraints that have been reached preventing frequency scaling as major driver for performance increase of CPU’s in the future, new ways had to be found to gain further speed.

The easiest way to describe frequency scaling is to have a look at the following equation:
\[
\text{Runtime} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]  

where the first factor - instructions per program - is the total instructions being executed in a certain program. The second factor is an average value depending on the concrete program and on the architecture. Finally, the third factor is the reciprocal of the frequency, realizing that a cycle is (somehow), equal to the period of a signal.[8]

Power consumption of a chip can be calculated by the following formula:

\[
P = C \times V^2 \times F
\]

where \( P \) denotes power, \( C \) is the capacitance being switched per clock cycle, \( V \) denotes voltage and \( F \) is the processor frequency.[8]

As one can see, increase in frequency goes along with an increase in the amount of power used in a processor.[8] Increasing processor power consumption forced Intel in May 2004 to cancel the development of its Tejas and Jayhawk processors. This incident can be seen as the end of frequency scaling as the dominant computer architecture paradigm. After cancelling the development of these two single-core processors Intel reconsidered their roadmap and began to work on the development of dual core processors.[7] Clock frequency is now obviously hitting the commonly referred power wall (= brick wall 1). So it is a fact that performance increasing measures must predominantly come from increasing the number of processors (or cores) on a die, rather than trying to boost a single core’s clock rate. This “historic” shift to multicore processors at the client space causes a major change concerning the programming interface. Programmers nowadays are exposed to parallelism, after decades of sequential computing.[1 page 6]

What are the different states of the art of hardware configurations?

I. HARDWARE FOUNDATIONS

In order to a well-founded understanding of what makes the development of parallel programs difficult and on what systems today parallel programs may run, lets recall some fundamentals about current hardware set-ups in a computer nowadays.

**CPU** A Central Processing Unit (CPU) or processor is built up with many different parts:

One or more memory caches for instructions and data, instruction decoders and finally different types of execution units for performing arithmetic or logical operations.[10]

**Multiprocessor system** A multiprocessor system now contains multiple CPU’s as defined above. This enables such a system to work in parallel. That is the so-called term of Simultaneous Multiprocessing (SMP).[10]

**Multicore CPU** A multicore CPU in contrast has more than one different execution unit on one CPU. Depending on the concrete architecture a “[...] certain subset of the CPU’s components is duplicated, so that multiple cores can work in parallel on separate operations.” That is the so-called term of Chip-level Multiprocessing (CMP).[10]

**Simultaneous Multithreading** A further differentiation is the one of Simultaneous Multithreading. Even a smaller subset either of a processor’s or core’s components is duplicated. Such a Simultaneous Multithreading core might have duplicated resources for scheduling different threads. The operating system finally sees the core as having two separate processors, even though it has only one set of execution unit. That is the so-called term of Simultaneous Multithreading (SMT).[10]

Generally, one can have a multiprocessor, multicore and multithreaded system. So, one would come up with a system like two quad-core,
hyper-threaded (Intel’s proprietary simultaneous multithreading implementation) processors. From the operating system point of view, this would result in having 16 logical processors.\[10\]

As a result, client space programs, where parallelism has been newly introduced, need to be converted or rewritten to be able to use the advantage of parallelism. Unfortunately, the present programming models “[... are prone to subtle, hard to reproduce bugs, and parallel programs are notoriously hard to test due to data races, non-deterministic [thread] interleavings, and complex memory models.”\[1\] page 6\]

Therefore parallel programming at the client-side is a difficult task. “[...] There is a risk that while hardware races ahead to ever-larger numbers of cores, software will lag behind and few applications will leverage the potential hardware performance.”\[1\] page 6

But why exactly is parallel programming a hard task?

II. Problems

When discussing about problems of building up parallel programs it is best to just focus on multi-threading.

In the context of this paper and especially when discussing problems with parallel programs the focus is therefore on multi-threading. Apparently most parallel programs on today’s systems are written with the help of threads and shared variables. That is, multiple threads are capable to access the same memory region.\[2\] page 90\]

In general, there are a number of good reasons to rely on threads even on multiprocessor/multicore systems. Threads deliver a direct hardware support for shared-memory that might lead to a performance advantage being able to overcome the space overhead by implicitly sharing read-mostly data without having to execute a complete replication of the data, because all threads are accessing the same memory region.\[2\] page 90

But to start with, fortunately, one has not even necessarily to think about multi-threading in the first place. Starting to talk about general problems of parallel programming and different memory models is easier if it is first considered how memory is defined in a uniprocessor. “A uniprocessor executes instructions and memory operations in a dynamic execution order called program order.” If a program is executed in program order each read has to return the value of the last write to the same address.\[4\] page 28

Now, let us go a step further and let us consider a uniprocessor that is capable of multitasking. There is the possibility that the program executes as a single thread and therefore does not have to share memory. From the programmers point of view the memory interface is the same as without multitasking.\[4\] page 28

The situation is in fact more complex, if a program works with multiple threads sharing memory.\[4\] page 28 f.

Working with threads on shared memory can be difficult and directly leads to the first problem when implementing parallel software.

Race Condition vs. Data Race These problems can appear in a multithreaded surrounding, where there is a shared memory access and they can lead to non-deterministic thread interleaving. Note that it does not matter whether this multithreaded surrounding is on a multiprocessor or just on a uniprocessor.

Basically a race condition is a bug that can be observed when the timing or the ordering of the execution of different statements directly affects a program’s correctness.\[5\]

Additionally, a data race occurs when there are at least two memory accesses in a program where both:

- “target the same location”
- “are performed concurrently by two threads”
- “are not reads”
As a consequence of race conditions as well as data races leading to non-deterministic thread interleaving, a program’s outcome might be different from what is expected by the programmer. A simple example will follow. Having more than one thread, the last write to a memory address could be made by the same thread or by a different one “[..] that was context-switched onto the processor since this thread’s last write to the same memory address.”[4, page 29]

Now, how can a programmer model a program running on a multitasking uniprocessor?

One approach is to model the program as a “merging of the program orders of each executing thread into a single, totally ordered processor execution.” This is what most programmers would expect. Considering Listing 1 “the expectation would be that the code fragment sets data_copy to the value of new. Here, accesses to flag are an example of synchronization, because their purpose is to coordinate accesses to data.”[4, page 29]

There are apparently differences between running a program on a single-tasked uniprocessor and running it on a multitasking uniprocessor environment. Furthermore, these problems can also be seen when working on multiprocessor environments and therefore working with true parallelism. The problems that can be seen rely strongly on the underlying memory consistency models. Later on, in the section 3 other memory models in contrast to sequential consistency are shown.

But first of all, let us show another typical example of a difference on a relatively simple system’s architecture without cache to see how the new problems of race conditions can arise when working with multiprocessors.

**Write buffers with bypassing capability** This optimization shows the importance of maintaining program order between a write and a following read operation. We consider a simple processor which issues memory operations one-at-a-time in program order. “On a write, a processor simply puts the write operation into the write buffer and proceeds without waiting for the write to complete.” Subsequent reads can bypass any previous writes in the write buffer for faster completion. This bypassing is only permitted as long as the read address does not match the address of any of the buffered writes. The above description demonstrates a common hardware optimization used in uniprocessors to hide the latency of write operations.[3, page 5]

Let us give an example to point out exactly how the use of write buffers on a multiprocessor system is able to violate sequential consistency and therefore is a problem not existing on a uniprocessor. Have a look at Figure 2.

---

*Figure 1: Is data_copy always set to new? [4, p. 29]*

*Figure 2: Write buffer [3, p. 6]*
The program in Figure 2 depicts Dekker’s algorithm. A sequentially consistent system has to prohibit an outcome where both reads of the flags return the value 0. However, this outcome may arise in the example system described in the Figure above. Each processor is allowed to buffer its write and is allowed to let subsequent reads bypass the write in its write buffer. Therefore, both reads may be serviced by the memory system before either write is serviced. This makes it possible that both reads may return the value 0.[3, page 5]

The above mentioned optimization is unproblematic if executed on a conventional uniprocessor since bypassing (between operations to different locations) does not lead to a violation of uniprocessor data dependence. However, in a multiprocessor environment things look different and sequential consistency can be violated easily. More about sequential consistency later on, compare section 3.[3, page 5]

Finally, in order to understand the basic problem of parallel programming at the client-side. Let us again discuss some of the fundamentals.

Middleware High-level languages like C, C++ and Java create the need for two memory interface levels. Speaking about the higher level, each programming language has to define memory for its programmers. On the lower level thereafter, hardware has to define memory for low-level software, commonly called middleware. Compilers, libraries, device drivers, operating systems are part of middleware. “Software written at a high level requires that compilers translate high-level memory operations into low-level ones in a manner that preserves memory semantics.” Recalling Listing 1 a compiler must not reorder P1’s stores to data and flag.[4] page 29]

The need for this two memory models brings us to another problem, the one of complex memory models!

Complex memory consistency models Essentially complex memory models make the implementation of parallel programs difficult to write.[2] page 91] In order to understand the complexity and probable problems that lie in different memory consistency models, one has to have a clear understanding of what a memory consistency model is and what different flavors of memory consistency models exist. Let us start with a simple definition of a memory consistency model.

Memory consistency model A memory consistency model is the interface for memory in a shared memory multiprocessor.

On a multiprocessor as well as on a uniprocessor, having high-level programming languages makes it necessary to work with two levels of memory consistency models as depicted in Figure 3 and mentioned before in the paragraph “Middleware”. There are high-level models for each high-level language and one low-level model for hardware.[4] page 29]

In the past there was the problem that there had been different high-level memory consistency models as well as different hardware-dependent low-level memory consistency models that differed very much and were not really reliable, because they were not specified clearly or concluded errors.

The lack of specification can be partly explained by the fact that “formally specifying a model that balances all desirable properties of programmability, performance, and portability has proven surprisingly complex.”[2] page 92]

Hence, the interplay between high-level and low-level models did not work out well. “After much prior confusion, major programming languages are converging on a model that guarantees simple interleaving-based semantics for data-race-free programs and most hardware vendors have committed to support this model.”[2] page 92]

In the late 1980s and 1990s the hardware community took great effort in formally describing low-level memory consistency models, but with little consensus. Commercial hardware memory models still differed greatly in precision. Some vendors denied making commitments with unclear future implications.[2] 92]
“Part of the challenge for hardware architects was the lack of clear memory models at the programming language level. It was unclear what programmers expected hardware to do. Although hardware researchers proposed approaches to bridge this gap, widespread adoption required consensus from the software community. Before 2000, there were a few programming environments that addressed the issue with relative clarity, but the most widely used environments had unclear and questionable specifications. Even when specifications were relatively clear, they were often violated to obtain sufficient performance.”[2, page 92]

Because there were no high-level memory consistency models, low-level memory consistency models issues - or hardware issues - came up “[...] created by hardware that had evolved without the benefit of a clear programming model. This often made it difficult to reconcile the need for a simple and usable programming model with that for adequate performance on existing hardware.”[2 page 92]

Luckily today things get better! Language developers and most hardware vendors have already published or are about to publish compatible memory consistency model specifications.[2, 92]

This article is mostly about hardware consistency models.

Sequential Consistency (SC) vs. relaxed models According to Lamport’s definition a multiprocessor is sequential consistent if

- “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and”[4 page 29]
- “the operations of each individual processor appear in this sequence in the order specified by its program.”[4 page 29]

Note that the sequentiality of each individual processor gives no guarantee that the entire multiprocessor computer is sequentially consistent.

“Interpreting the SC definition literally could lead to the opinion that implementing it requires one memory module and precludes per-processor caches.” But against this feeling, both SC and the relaxed models (described below) allow many optimizations important for high-performance implementations.[4 page 30]
SC as well as relaxed models allow coherent caching, nonbinding prefetching, and multithreading.\[4\] page 30

**Coherent caching** All the models allow coherent caching. Note that caches may be private to processors or shared among some processors. Furthermore the caches may be in one level or may be provided in multilevel hierarchies. “A standard implementation of coherence processes operations of each processor in program order and does not perform a write until after it invalidates all other cached copies of the block.”[4 page 30]

**Nonbinding prefetching** All kind of models can use nonbinding prefetching. Nonbinding prefetching describes the process of moving a block into a cache just in anticipation of use. The block is still under jurisdiction of the coherence protocol. Nonbinding prefetching is a technique that affects memory performance and not memory semantics. If another processor likes to write to the block a prefetched block can be invalidated. “Nonbinding prefetches can be initiated to overlap cache miss latency by either hardware or software, through special prefetch instructions.”[4 page 30]

**Multithreading** “Finally, all models can support multithreading, where a processor contains several threads (or processes) that it can run in an interleaved fashion.”[4 page 30]

Speaking about commercial hardware architectures most of them have selected alternatives to SC as mentioned before. These alternatives are called relaxed (or weak) memory consistency models. Initially these models were introduced to facilitate additional implementation optimizations. These optimizations would not be allowed by simple SC without speculative execution.[4 page 30]

Using write buffers with the memory model of SC is difficult, because write buffers present operations to the cache-coherence protocol out of program order. Straightforward processor are also not allowed to overlap multiple reads and writes in the memory system. This restriction decreases performance in systems without caches, where all operations are issued directly to the memory. If a system has cache coherency this restriction has an impact whenever operations miss or bypass the cache.[4 page 30]

Relaxed memory models try to improve performance by relaxing the order of certain statements issued to memory. There are different kinds of relaxed models. These models will be discussed in the next paragraphs.

**Relaxing write-to-read order** “Defining relaxed models is complex.” This class of memory models mainly discussed by Adve and Gharachorloo - they call the class relaxing write-to-read program order - is sometimes also called processor-consistent models. This sort of class “[...] more or less exposes first-in first-out (FIFO) write buffers to low-level software.” A processor’s write may not immediately affect other processors, but if they do, the writes are seen in program order.[4 page 30] In other words, these models “[...] allow a read to be reordered with respect to previous writes from the same processor.”[3 page 14]

Let’s have a look at a simple example: “If a processor does write \(x\), write flag, and read \(y\), it can be sure that \(x\) is updated before flag, but it cannot know if either is done when it reads \(y\).” This behavior cannot be observed on a uniprocessor, because all common processors nowadays make sure that they see their own writes immediately. To see the own writes immediately the mechanism of write buffer bypassing, for example, can be used.[4 page 30]

Furthermore, the differences from these models compared to SC are inexistent when speaking about the most shared memory programs, because these programs create shared data by first writing the data and then writing a flag or counter as mentioned in the example above and sketched in listing[3 page 30]

Anyway, there are of course situations, where programs encounter differences from SC. Figure 4 for example, shows that these models allow both \(x\)\_copy and \(y\)\_copy to obtain old values, while SC requires that at least one gets the value of new[4 page 30]
Models that count to this sort of memory consistency models are Wisconsin/Stanford processor consistency (PC), IBM 370, Intel Pentium Pro, and Sun’s Total Store Ordering (TSO). Not all of them work exactly the same way. They differ in subtle ways. A processor reading its own write may ensure that other processors are also immediately capable to see it. “All of the commercial models also guarantee causality.”[4, page 30]

Causality This term describes exactly the problem sketched beforehand.

The process to give a guarantee that all other processors see the effect of a processor’s operation when any other processor sees it is called causality.[4, page 30]

A good example concerning causality delivers Figure 5 causality guarantees that data_copy gets the value new. Apparently - according to[4, page 30] - processor consistency can fail without causality to look like SC in many cases involving three or more processors!

Write atomicity For the purpose of well understanding the new term write atomicity should also be explained briefly. “Write atomicity says that all writes to a location should appear to all processors to have occurred in the same order.”[6] Please, compare Figure 6.

The terms causality and write atomicity have to be understood well when discussing the TSO model a bit more in detail.

The TSO model allows “a read to return the value of its own processor’s write even before the write is serialized with respect to other writes to the same location. However, as with sequential consistency, a read is not allowed to return the value of another processor’s write until it is made visible to all other processors.”[3, page 13]

“For TSO, a safety net for write atomicity is required only for a write that is followed by a read to the same location in the same processor; the atomicity can be achieved by ensuring program order from the write to the read using read-modify-writes.”[3, page 14]

Safety nets Relaxed models typically provide programmers with an instrument for overriding certain relaxations a relaxed model normally does. These instructions are called safety nets in[3]. It is an instrument that allows programmer to ensure the exact program order between any two important operations. As an example: The synchronization flag must not be written before the data has been written, (compare Figure 7) and note that changing write order is possible with relaxing write-to-write order models that are shortly discussed later on in this paper.[3, page 11]

These hardware memory consistency models have been invented in order to make it easier for hardware implementors to use certain hardware optimizations originally found in uniprocessors. In particular, one approach that is

Figure 4: Processor Consistency, IBM 370, Pentium Pro, and Total Store Order (TSO) allow both x_copy and y_copy to get old values, which violates sequential consistency.[4, p. 30]

Figure 5: Causality makes sure that data_copy is set to new.[4, p. 31]

Figure 6: All writes to a location should appear to all processors to have occurred in the same order.[6]
allowed by these models is the one of buffering writes. The write operations can be buffered in a FIFO write buffer in front of cache and coherence protocol. Furthermore, values of these buffered writes can be bypassed to subsequent reads. This bypassing can be done by the same processor to the same address. The bypassing can occur even before coherence permission has been obtained. Especially for architectures with few general-purpose registers such as Intel’s IA 32 the optimization of write buffer bypassing is vital.[4, page 31]

To conclude, it is important to note that these kind of models have a minor impact on middleware authors. Assuming that these authors think that SC is the underlying model, they will rarely be surprised by these kind of relaxed models. “These models look exactly like SC for the common idioms of data sharing (as in Figure 1 and Figure 5, for example, but not in Figure 4).”[4, page 31]

Nevertheless, under certain circumstances it is not enough, just to relax the write-to-read order. As explained by the following quote:

“Relaxing the program order from a write followed by a read can improve performance substantially at the hardware level by effectively hiding the latency of write operations. For compiler optimizations, however, this relaxation alone is not beneficial in practice. The reason is that reads and writes are usually finely interleaved in a program; therefore, most reordering optimizations effectively result in reordering with respect to both reads and writes. Thus, most compiler optimizations require the full flexibility of reordering any two operations in program order; the ability to only reorder a write with respect to a following read is not sufficiently flexible.”[3, page 14]

Relaxing write-to-read and the write-to-write order This class of memory consistency models describes a set of memory consistency models, which further relaxes the program order requirement by eliminating the ordering constraints between writes to different locations. An example for this set of models is SPARC V8 partial store ordering (PSO). PSO adds the possibility of new hardware optimization compared to the preceding set of models. Writes to different memory locations from the same processor can be pipelined or overlapped and are permitted to reach memory or other cached copies out of program order. PSO is similar to TSO with respect to atomicity requirements. It allows a processor to read the value of its own write early, and prohibits a processor from reading the value of another processor’s write before the write is visible to all other processors.[3, page 14]

For PSO there are again instructions necessary that build up a so-called safety net. The safety net delivered by PSO for assuring the program order from a write to a read, and for enforcing write atomicity, is exactly the same as for TSO. In addition, PSO provides an explicit STBAR instruction in order to be able to steer program order between two writes.[3, page 15]

One approach to support a STBAR in an implementation with FIFO write buffers is to insert the STBAR to the write buffer. Afterwards, the retiring of writes that are buffered after a STBAR have to be delayed until all writes that were buffered before the STBAR have retired and completed. It is the easiest to use a counter to keep track whether all the writes before a STBAR statement have been processed. A write sent to the memory increments the counter, a write acknowledgement decrements the counter, and a counter having the value 0 indicates that all previous writes are completed. Referring to the program in Figure 7 a STBAR has to be inserted between the two writes on processor 1 to ensure sequentially consistent results with PSO.[3, page 15]
Again, as with previous set of models, the optimizations permitted by PSO are still not sufficiently flexible to be useful to a compiler.\cite{3, page 15}

**Relaxing all orders** There are memory consistency models that are relaxing all order. This class of models is introduced to allow all the hardware implementation possibilities of a uniprocessor. Models being part of this class may completely reorder reads and writes. USC/Rice weak ordering (WO), Stanford release consistency (RC), DEC Alpha, IBM PowerPC, and Sun’s Relaxed Memory Order (RMO) are part of this model class. These models assure that a processor sees its own reads and writes in program order, regardless of the current reordering undertaken.\cite{4, page 31}

The models again differ in subtle ways and especially in how programmers restore order between memory operations to make examples like Figure 1 behave as expected. It is important to notice that further actions have to be undertaken in order to make the program appear as sequential consistent or at least to behave as expected. WO and RC leave it to the programmer to distinguish or declare certain reads and writes as synchronization. The hardware has to handle these sort of reads and writes more carefully.\cite{4, page 31}

All the commercial models rely on the so-called safety nets, these certain special operations to tell the system when order is required. Unfortunately, these special operations differ from system architecture to system architecture and are variously called fences, barriers, membars, and syncs.\cite{4, page 31}

Let us go for an example. Figure 8 sketches how the example in Figure 1 could be changed to work with Sun RMO as underlying memory consistency model.

The membar \#StoreStore is used to ensure that data is written before flag, whereas membar \#LoadLoad ensures flag is read before data. Now, it should be obvious that models from this class can make full use of many optimizations, because they need only to implement order between operations when defined by the software. The rest of the time operations can be aggressively out of order. In addition, processors are able to complete reads and writes to cache, even while previous reads and writes in program order have not obtained coherence permission.\cite{4, page 31}

A hardware model that relaxes all orders should not be too great a challenge for compiler writers. Sequential high-level languages with threads like C or Java allow programmers to use synchronization libraries or declare critical variables volatile. Having all the necessary information about synchronization the compiler or the library writer can add appropriate fences.\cite{4, page 31}
IV. Conclusion

In the end, one can conclude that nowadays multiprocessing has become a standard at client-side computing. Therefore it is inevitable for programmers to think about concurrency. The most understandable model for programmers is sequential consistency. If working with sequential consistency on a multitasking system - let us say a multiprocessor system - the first impression is that basically, relaxed models provide more hardware implementation possibilities than Sequential Consistency. So the first impression is that hardware should use relaxed models instead of Sequential Consistency.

Interestingly, there are people as can be seen in [4, page 31 ff.], who argue differently. According to their argumentation Sequential Consistency is the thing to be implemented on multiprocessor systems. In a nutshell, according to their argumentation, “the performance gained by using relaxed models does not justify their complexity” [4 page 33]. Apparently, as already mentioned in the paper, it is difficult to formally prove the correctness of certain low-level memory consistency models. Additionally, the performance boost is not as big as expected [4 page 31] and using the class of memory consistency models that are relaxing all orders implies an additional burden on middleware authors [4 page 33]. To conclude, one can say that it is best to stick to the traditional sequential consistency model, probably with some modifications, as can be read at [4 page 33], just to avoid unnecessary complexity. Moreover, with Sequential Consistency it is possible to hide the out-of-order complexity from software. On the contrary, using relaxed models, complexity is visible to the software interface. Sequential Consistency provides the solution to keep complexity off the interface and in the implementation of the memory consistency model [4 page 34]. Hiding complexity is one principle that should be followed and strongly supports the idea of using Sequential Consistency in future systems.
IV. References

[As from: 29.10.2012]

[As from: 29.10.2012]

[As from: 29.10.2012]

Accessible at: http://pages.cs.wisc.edu/~markhill/includes/publications.html#year1998
[As from: 29.10.2012]

Accessible at: http://blog.regehr.org/archives/490
[As from: 29.10.2012]

[As from: 29.10.2012]

Accessible at: http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007
[As from: 29.10.2012]

Accessible at: http://en.wikipedia.org/wiki/Frequency_scaling
[As from: 29.10.2012]
Accessible at:
http://en.wikipedia.org/wiki/Parallel_computing
[As from: 29.10.2012]

[10] Forum superuser (2010): *What is the difference between MultiCore and MultiProcessor*
Accessible at:
http://superuser.com/questions/214331/what-is-the-difference-between-multicore-and-multiprocessor
[As from: 29.10.2012]